macromedia® white paper

Flex Application Performance: Tips and Techniques for Improving Client Application and Server Performance

November 2004

Brandon Purcell

Deepa Subramanian

Contents

Executive Summary

Macromedia Flex is a powerful platform that offers the ability to create Rich Internet Application (RIAs). Misusing this power can result in areas performance. This article explores some of these performance issues and offers tips on how to make the most of your Flex application with tiers involved: the Flex client and the Flex presentation server.

Flex Application Performance: Tips and Techniques for Improving Client Application Performance

Flex is like any other programming model because certain coding practices can be detrimental to the overall performance of your client application. This article describes some of the MXML coding practices that affect application performance on the client side and demonstrates simple techniques that will enhance the performance of your Flex application. More specifically, this article explores how to do the following tasks:

- Decrease startup time
- Take advantage of deferred instantiation to improve performance
- Play complex effects smoothly
- Use dynamically repeating controls for better performance
- Improve performance of charting components
- Leverage Runtime Shared Libraries (RSLs) for performance gains
- Performance test your own Flex application

Note that the tips suggested in this article do not apply to all Flex applications. It is important to analyze the structure of your own application and modify the suggestions to tailor them to your needs. For ongoing coding and conceptual help, you can use the Flex support forums (www.macromedia.com/cfusion/webforums/forum/index.cfm?forumid=60) and Flex Developer Center (www.macromedia.com/devnet/flex/).

Note also that this is the first part in a two-part article. The second section of this article, "Flex Application Performance: Tips and Techniques for Improving Server-Side Performance," discusses the server-side enhancements and deployment options that you can use to improve the performance of your Flex application.

Requirements

To make the most of this article, you need the following:

- Familiarity with Macromedia Flex and J2EE application servers (JRun, IBM Websphere, or BEA Weblogic)
- Some experience building Flex applications. Ideally, you should have built at least one simple Flex application. If you haven't, see Creating Your First Flex Application (www.macromedia.com/devnet/flex/articles/first_flexapp.html).

Macromedia Flex

Learn more about Macromedia Flex (www.macromedia.com/software/flex/).

Macromedia Flash Player 7

www.macromedia.com/shockwave/download/download.cgi?P1_Prod_Version=ShockwaveFlash

Feedback and Support

We have made every effort to ensure the accuracy of this article and all code included. Feedback for this article and all Flex performance issues is always appreciated. To submit feedback, please email us.

Architecting Flex Applications That Perform Well

Ideally, performance is a topic that is in the back of your mind during every step of the development process—from application design to implementation and deployment. When creating a Flex application, think through the choice of containers and components that you use to ensure that the code is maintainable, organizationally clear, and performs well.

You can use the Flex navigator containers (Accordion, TabNavigator, and ViewStack) across all application types to organize content. The navigator containers organize content in a way that:

- Minimizes users' confusion
- Exhibits good user interface (UI) design principles for browser-based applications
- Aids performance

More specifically, Flex navigator containers help you organize content easily into different child views and control the creation of these views with deferred instantiation. Organizing content into these child views spreads out creation time for each child view because Flex creates a specific child view the first time a user requests that view. The "Navigator Containers Have Built-in Deferred Instantiation" section later in this article explains why Flex navigator containers perform better with deferred instantiation and how you can leverage deferred instantiation to make your application more robust.

Dashboard-style applications have also resulted in successful deployments. This type of application organizes content into modular, self-contained views that offer a more intuitive approach to application organization. Like the navigator containers, this approach performs well because it organizes complex views, with Flex creating them when the user drills down. Flex does not have to size, measure, and draw the views in the background, so it creates the selected view more quickly.

A Note About Flash Player

Before going into techniques for preventing common performance issues, ensure that you have installed Macromedia Flash Player 7 (7.0.14 or 7.0.19 or later). Flash Player is a multiple-platform client that lets users interact with Flash content. There are two types of players, the Flash Player release version and the Flash Debug Player version. The Flash Debug Player version is best used during the development phase, because it enables the Flex debugging and profiling features. Since most users use the release version of Flash Player, use this version to do performance tuning; running Flex applications with the Flash Debug Player version does not accurately represent the performance of your application. When running a SWF file, the Flash Debug Player version reports trace statements and warnings. This task requires ActionScript processing cycles that would otherwise resource a running application, which impacts the perceived application performance.

When you are ready to test application performance, verify you are running the application with Flash Player 7, the release version. Many Flex application developers make this simple mistake! To verify the version of Flash Player, run a Flex application in the browser you use during development, and then right-click inside the browser window. If you see a debug option in the context menu, you are running the Flash Debug Player version. If you fine-tune your application to perform well with the Flash Debug Player version, you can ensure that your application will perform the same, if not significantly better, with Flash Player 7.

For more on using the Flash Debug Player to debug client-side code, read Debugging Client-Side Code in Flex Applications (www.macromedia.com/devnet/flex/articles/client_debug.html).

Using Layouts, Hierarchy, and Containment Properly

The biggest Flex performance danger is yielding to the temptation to use containers randomly. Using too many containers dramatically reduces the performance of your application. This is the number one performance danger that Flex developers succumb to—and luckily it is 100 percent avoidable. The performance penalty occurs because Flex layout containers and their children follow sizing and measuring algorithms that determine x,y positions, preferred sizes, and styles. These calculations are resource-intensive; it is these calculations, coupled with Flash Player drawing complex objects, that cause a noticeable delay when starting a Flex application or when instantiating a new view in a navigator container. One principle dramatically speeds up application startup and interactivity time: Avoid unnecessary container nesting.

Avoid Nesting Containers Many Levels Deep

A good rule of thumb is to avoid excessive container nesting. At first, you might find it difficult to pinpoint superfluous container nesting. The following describes some of the more common cases of nesting containers and offers useful tips for choosing and using containers.

Below are a few examples of deeply nested code:

```
<mx:VBox>
  <mx:HBox>
    <mx:Form>
      <mx:FormItem>
       .....
       ...
      </mx:FormItem>
    </mx:Form>
  </mx:HBox>
</mx:VBox>
and
<mx:Grid>
  <mx:GridRow>
    <mx:GridItem>
      <mx:VBox>
       <mx:Button />
      </mx:VBox>
    </mx:GridItem>
  </mx:GridRow>
</mx:Grid>
```

When you nest containers, each container instance runs measuring and sizing algorithms on its children (some of which are containers themselves, so this measuring procedure can be recursive). When the layout algorithms have processed, and the relative layout values have been calculated, Flash Player draws the complex collection of objects comprising the view. By eliminating unnecessary work at object creation time, you give your application a boost and the performance benefits are readily apparent.

Typically, fewer containers provides good results with respect to creation time. If you find yourself nesting many levels deep, re-evaluate your choice of containers. Perhaps you can achieve the same layout with a different layout container in conjunction with style attributes, such as horizontal and vertical alignment, margins, spacers, and gaps. You can use margins and gaps to manipulate the space around controls and between the edge of controls and the edge of their parent containers. You can use spacer objects to fill unwanted space or to push controls around the screen. You can also align controls horizontally or vertically within their container. For example, take a look at the layout in Figure 1.

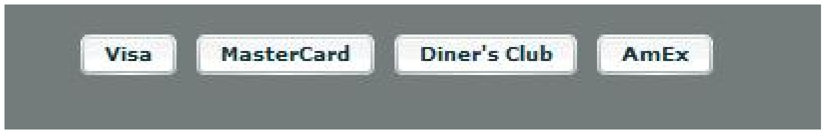

Figure 1: *You can achieve this layout without using a Grid container.*

It is tempting to use a Grid container to achieve this layout:

```
<mx:Grid>
  <mx:GridRow>
    <mx:GridItem>
      <mx:Button label="Visa"/>
    </mx:GridItem>
    <mx:GridItem>
      <mx:Button label="MasterCard"/>
    </mx:GridItem>
    <mx:GridItem>
      <mx:Button label="Diner's Club"/>
    </mx:GridItem>
    <mx:GridItem>
      <mx:Button label="AmEx"/>
    </mx:GridItem>
  </mx:GridRow>
</mx:Grid>
```

However, this code is unnecessarily bloated. In fact, you can easily revise the code so that it looks exactly like Figure 1. The following snippet uses less code and results in a faster creation time and a slightly smaller SWF output.

```
<mx:HBox>
  <mx:Button label="Visa"/>
  <mx:Button label="MasterCard"/>
  <mx:Button label="Diner's Club"/>
  <mx:Button label="AmEx"/>
</mx:HBox>
```

Steven Webster, an active Flex community member, has an excellent entry in his blog (www.richinternetapps.com/archives/000042.html) on the dangers of nested containers, as well as tips on how to avoid nesting.

Absolute Positioning and Sizing

The Flex container classes are relative layout containers that arrange contents on the screen for you. However, the calculations to decipher how big each container and its children are, as well as where to place them, can potentially be resource-intensive. Here are two tips that can help reduce these calculations:

- **Hard-code object positions**—Hard-coding object positions can save a lot of time because the Flex layout containers do not need to calculate object positions at runtime. If you want to go this route, you can only use the Canvas container. Other types of containers, like the VBox container, do not respect absolute positions. When using the Canvas container, you must explicitly declare the x and y properties of all the Canvas children. If you omit setting x and y properties, the Canvas container's children lay out on top of each other at the default x,y coordinates (0,0). Absolute positioning does not work well if you want your application to resize when the browser window resizes. Using the Canvas container to create faster layouts should be a last-resort solution.
- **Hard-code object widths and heights**—Hard-coding object widths and heights can also save time, because the Flex layout containers do not need to calculate the size of the object at runtime. By hard-coding container or control widths or heights, you lighten the relative layout container's processing load and subsequently speed up container and control creation time. This technique works with any container or control.

Use Grid Containers Wisely

Think of a Grid container as a layout choice already pushing the deep nesting rule. Grid, GridItem and GridRow are all containers in their own right, although GridItem and GridRow are only used in conjunction with the Grid container. You should only use a Grid container when your controls must line up both horizontally and vertically. Developers often gravitate to the Grid container, because they see the similarity to the HTML `<table>` tag. However, as a Flex developer, you can choose from **multiple** container choices to position objects (and some are less resource-intensive to use then others). This is not true of the HTML world, where `<table>` is really the only choice. The following code provides an example of when to use a Grid container—the controls in the different columns and rows must all line up (see Figure 2):

```
<mx:Grid>
 <mx:GridRow>
  <mx:GridItem><mx:TextInput text="TextInput"/></mx:GridItem>
  <mx:GridItem><mx:NumericStepper/></mx:GridItem>
  <mx:GridItem><mx:TextInput text="TextInput"/></mx:GridItem>
  <mx:GridItem><mx:NumericStepper/></mx:GridItem>
 </mx:GridRow>
 <mx:GridRow>
  <mx:GridItem><mx:Button label="button"/></mx:GridItem>
  <mx:GridItem><mx:DateField /></mx:GridItem>
  <mx:GridItem><mx:Button label="button"/></mx:GridItem>
  <mx:GridItem><mx:DateField /></mx:GridItem>
 </mx:GridRow>
 <mx:GridRow>
  <mx:GridItem><mx:TextInput text="TextInput"/></mx:GridItem>
  <mx:GridItem><mx:NumericStepper/></mx:GridItem>
  <mx:GridItem><mx:TextInput text="TextInput"/></mx:GridItem>
  <mx:GridItem><mx:NumericStepper/></mx:GridItem>
 </mx:GridRow>
</mx:Grid>
```

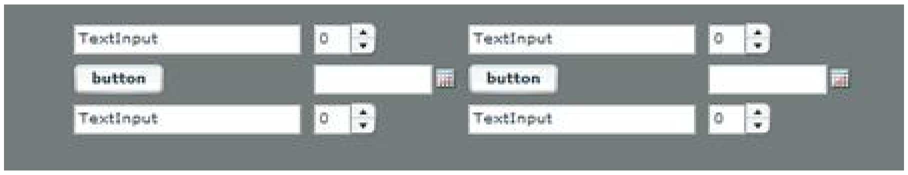

Figure 2: *This UI is the perfect candidate for a Grid container.*

Some common misuses of Grid containers include the following:

- Using the Grid container when you want to left-justify or right-justify controls in the same container (see Figure 3). Developers often try to do this using the following code:

```
<mx:Grid borderStyle="solid" width="400">
 <mx:GridRow>
  <mx:GridItem horizontalAlign="left">
   <mx:Button label="left" />
  </mx:GridItem>
  <mx:GridItem horizontalAlign="right">
   <mx:Button label="right" />
  </mx:GridItem>
 </mx:GridRow>
</mx:Grid>
```

However, using an HBox container with a Spacer object to fill unwanted space works the exact same way, as shown in the following snippet:

```
<mx:HBox borderStyle="solid" width="400">
  <mx:Button label="left" />
  <mx:Spacer width="100%" />
  <mx:Button label="right" />
</mx:HBox>
```

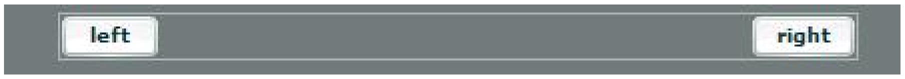

Figure 3: *You can achieve this left-justify/right-justify layout without a Grid.*

- Using a Grid container as a child of a Repeater object when alternate mechanisms would work better. Take a look at the Repeater object in Figure 4.

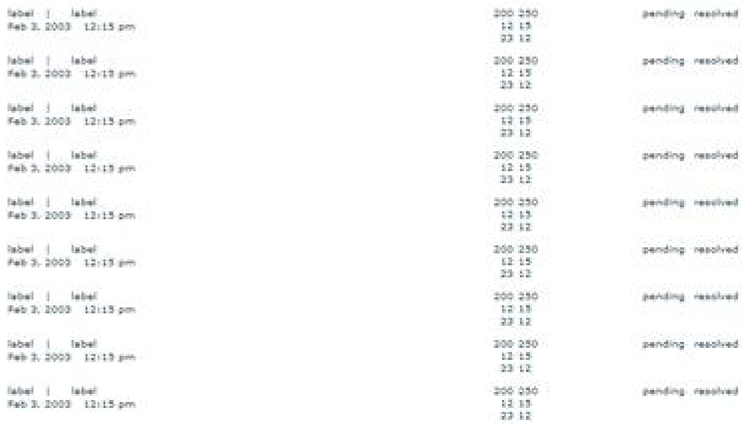

Figure 4: ***Figure 4.*** *Using a Grid container in a Repeater object to achieve this layout is resource-intensive*

This Repeater object repeats a heavily populated Grid container with labels retrieved from a web-service. Creating just one of these Grid containers would take a noticeable creation time, but repeating this Grid many times is worse.

As your Flex repertoire expands, you will see that there are alternate containers and controls that you can easily customize to meet your needs. For example, you can achieve the layout shown in Figure 4 with a List control and a custom cell renderer. You can use a cell renderer with any list-based component to create custom-formatted cells, and the cells can be different heights. The Flex documentation will have more information and examples on how to create a custom cell renderer. The performance of a List control with a custom cell renderer is spectacularly better than with a repeated Grid container. Or you can use the HorizontalList and TileList controls to create custom controls in an HBox or Tile-like layout. These controls perform very well because they only create elements visible in the initial view and then the user can scroll to see subsequent elements—instantiation time is quicker.

Examples of Common Container Redundancies to Avoid

The following list provides examples of common container redundancies to avoid:

- **The VBox container inside an `<mx:Panel>` tag**—A Panel container is a VBox container with support for a title bar, rounded borders, and other Panel styles. If you want Panel children to lay out as they would in a VBox container, populate the `<mx:Panel>` tag directly with controls; do not wrap a VBox container around the controls. The VBox container would be a redundant container wrapper, and removing this eliminates one more level of unnecessary container nesting.

  For example, instead of writing this:

```
<mx:Panel title="Grocery List" width="150" height="150">
  <mx:VBox>
   <mx:Label text="Fruits" />
   <mx:Label text="Veggies" />
   <mx:Label text="Cookies" />
   <mx:Label text="Crackers" />
  </mx:VBox>
</mx:Panel>
```

  Use this code instead:

```
<mx:Panel title="Grocery List" width="150" height="150">
   <mx:Label text="Fruits" />
   <mx:Label text="Veggies" />
   <mx:Label text="Cookies" />
   <mx:Label text="Crackers" />
</mx:Panel>
```

  Both achieve an identical layout.

- **VBox container inside an `<mx:Application>` tag**—An Application object is inherently a VBox container layout. It is unnecessary to wrap your `<mx:Application>` tag with a VBox container; removing this wrapper will eliminate one more level of container nesting.

  For example, instead of writing this:

```
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml">
 <mx:VBox horizontalAlign="center" backgroundColor="#EFEFEF">
    <mx:Label label="Shopping Cart" />
   .
   .
   .
 </mx:VBox>
</mx:Application>
```

  Use this code instead:

```
<mx:Application xmlns:mx=http://www.macromedia.com/2003/mxml
 horizontalAlign="center" backgroundColor="#EFEFEF">
  <mx:Label label="Shopping Cart" />
    .
    .
    .
</mx:Application>
```

- **Containers as top-level tags for MXML components**—The beauty of MXML components is that they provide a way to modularize repeated code. However, these components are vulnerable to the same performance pitfalls as the larger application. You do not need to use a container tag as the top-level tag of your MXML component definition. It is perfectly valid for an MXML component to be a simple control, like:

```
<mx:Image xmlns:mx=http://www.macromedia.com/2003/mxml
   source="@embed('foo.jpg')" width="200" height="200" />
```

  By stepping through MXML component definitions and purging unneeded container wrappers, you can reduce container nesting (and many unnecessary objects), and, subsequently, ease the load of your application.

- **Container wrappers for an MXML component instance**— Normally, there is no need to wrap a container around an MXML component tag. You can set different styles, labels, and ids within the instance of the MXML component—you don't need to set them within a container that wraps the MXML component. For example, instead of wrapping an unnecessary VBox container around your MXML component to set some styles like this:

```
<mx:VBox backgroundColor=" #FFCCCC" borderStyle=" solid">
  <myComponent xmlns=" *" />
</mx:VBox>
```

  You can set those styles within the MXML component tag itself, like this:

```
<myComponent xmlns=" *" backgroundColor=" #FFCCCC" borderStyle="
   solid" />
```

  Developers often couple this bad practice with using unnecessary containers as top-level tags for MXML components, which can create at least two superfluous containers for each MXML component used! This creates many unused objects that, when eliminated, yield a dramatic improvement in the responsiveness of your application.

- **Re-evaluate your container choices**—Inspect every container tag and decide if it is necessary and if it is useful for the overall structure of the application. Could the same layout be mimicked with another layout container or by using layout styles? Revising your layout ensures that your application is as lean as possible, reduces overall SWF file output size, and makes performance much more robust.

Using Deferred Instantiation to Improve Perceived Performance

If the number one cause of performance problems is unnecessary measurement and layout from superfluous container nesting, the number two cause is creating objects before they are needed. To avoid this problem, you can use deferred instantiation. Flex uses deferred instantiation to determine which components to create at application startup. When using deferred instantiation, you can decide at which stages the user incurs the costs of component creation. Containers have a `creationPolicy` property that you set to specify when Flex should create the container (at startup, incrementally, when a user navigates to that container, or based on other user action).

Navigator Containers Have Built-In Deferred Instantiation

The Flex navigator containers (ViewStack, Accordion, TabNavigator) have built-in deferred instantiation behavior. The default deferred instantiation behavior means that Flex does not create all the child views at startup, but only when a user triggers it by navigating to the container. The following code shows two navigator containers, TabNavigator and ViewStack, in use:

```
<mx:TabNavigator>
  <mx:VBox id="tabNavView1">
    <mx:LinkBar dataProvider="myViewStack" />
    <mx:ViewStack id="myViewStack">
       <mx:VBox id="view1" >
           .
           .
           .
       </mx:VBox>
       <mx:VBox id="view2" >
           .
           .
           .
       </mx:VBox>
       <mx:VBox id="view3" >
           .
           .
           .
       </mx:VBox>
    </mx:ViewStack>
  </mx:VBox>
  <mx:VBox id="tabNavView2">
    .
    .
    .
  </mx:VBox>
</mx:TabNavigator>
```

The TabNavigator container creates tabNavView1 because it is the first view displayed when Flex instantiates the TabNavigator container. Instantiating tabNavView1 will cause the LinkBar and the first view of the ViewStack, view1, to be instantiated. When the user interacts with the LinkBar to select another view in the ViewStack, Flex will create that view. Flex continues in this way, creating the navigator container descendants as it calls them.

The `creationPolicy` property on container tags control the creation of child views. The following list explains what each `creationPolicy` property does when set on Flex navigator containers:

- `creationPolicy="auto"`—When Flex creates the navigator containers, it does not immediately create all of their descendants, only those that are initially visible. The result of this deferred instantiation is that an MXML application with a navigator container loads quickly, but users experience a brief pause the first time they navigate from one view to another. Usability studies have shown this is a better user experience then having to wait a noticeable amount of time at application startup to create the navigator containers' child views. Also, there is always the possibility that the user may never visit some of the child views, so creating them at startup is potentially wasteful.

  Note that if you set `creationPolicy="auto"` on a non-navigator container, you must add extra code to specify when to create the container's children. This extra code is built-in in the navigator containers, which is why you can set `creationPolicy="auto"` on a navigator container without doing any extra work.

- `creationPolicy="all"`—When Flex creates the navigator containers, it creates all of the controls in all their child views. This setting causes a delay in application startup time, but results in a quicker response time when navigating from view to view.

- `creationPolicy="none"`—Flex does not instantiate any component within the navigator container or any of the navigator component's child views until you explicitly call the instantiation methods. You explicitly instantiate views with the `createComponents()` method. The Flex documentation has more information on setting up a custom component creation plan.

- `creationPolicy="queued"`—Flex creates all containers and then creates the children of the queued containers in the order in which they appear in the application unless you specify a `creationIndex` property. This setting causes components in your application to become visible in successive fashion, reducing the amount of time it takes for the user to start viewing the application. In the following "Progressive Layout" section I give an example and more information on this feature.

By playing with the `creationPolicy` properties, you can manually handle the creation of child views and decide where in your application architecture you want to incur the cost of creating the child views of your navigator containers. Usability studies show that a better user experience is proffered when using the `auto` setting. A common mistake that can inadvertently slow your application startup time is to mistakenly set `creationPolicy="all"` on one of your navigator containers. Use `creationPolicy="all"` only if you are completely sure that the component creation plan you have in place is efficient and timely.

Progressive Layout–Queued Creation of Components

In the process of fine-tuning performance, you may reach a point where you have whittled your application's startup time to as fast as possible. However, this does not mean there are no more performance improvements you can make; you could choose to use progressive layout. Progressive layout is a UI concept that involves wiring your application to lay out components in a piece-by-piece fashion so there is a shorter initial delay before components begin appearing on the screen. Progressive layout is similar to the way HTML applications load content in succession in a client.

Progressive layout does not quantifiably reduce application startup time, but it significantly improves the *perceived* startup time. You implement progressive layout by using the queued creationPolicy in the deferred instantiation architecture of Flex. See the example below, which loads three panels in successive fashion.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >

<mx:HBox>

<mx:Panel id="panel1" title="Panel 1" width="210" height="240"
   horizontalAlign="center" verticalAlign="middle"
   creationPolicy="queued" creationIndex="1">
  <mx:DataGrid width="190" height="95">
        <mx:dataProvider>
            <mx:Array>
                <mx:Object Artist="Death Cab for Cutie" />
                <mx:Object Artist="The Postal Service" />
            </mx:Array>
        </mx:dataProvider>
  </mx:DataGrid>
  <mx:DataGrid width="190" height="95">
        <mx:dataProvider>
            <mx:Array>
        <mx:Object Album="Such Great Heights" />
        <mx:Object Album="We Know the Facts" />
            </mx:Array>
        </mx:dataProvider>
  </mx:DataGrid>
</mx:Panel>

<mx:Panel id="panel2" title="Panel 2" width="210" height="240"
   horizontalAlign="center" verticalAlign="middle"
   creationPolicy="queued" creationIndex="2">
  <mx:List width="190" height="95">
        <mx:dataProvider>
```

```
            <mx:Array>
                <mx:String>one</mx:String>
                <mx:String>two</mx:String>
                <mx:String>three</mx:String>
                <mx:String>four</mx:String>
            </mx:Array>
        </mx:dataProvider>
    </mx:List>
    <mx:List width="190" height="95">
        <mx:dataProvider>
            <mx:Array>
                <mx:String>red</mx:String>
                <mx:String>green</mx:String>
                <mx:String>yellow</mx:String>
                <mx:String>blue</mx:String>
            </mx:Array>
        </mx:dataProvider>
    </mx:List>
</mx:Panel>
<mx:Panel id="panel3" title="Panel 3" width="210" height="240"
   horizontalAlign="center" verticalAlign="middle"
   creationPolicy="queued" creationIndex="3">
  <mx:Tree width="190" height="95">
      <mx:dataProvider>
          <mx:XML>
              <node label="File">
              <node label="Open"/>
              </node>
              <node label="Close">
              <node label="Help"/>
              </node>
          </mx:XML>
      </mx:dataProvider>
   </mx:Tree>
   <mx:TextArea width="190" height="95">
      <mx:text>The Macromedia Flex presentation server offers a
   familiar, standards-based programming framework and powerful
   set of components for creating a rich, responsive presentation
   tier for enterprise Rich Internet Applications
   (RIAs).</mx:text>
   </mx:TextArea>
</mx:Panel>
</mx:HBox>

</mx:Application>
```

Jason Szeto, a developer on the Flex team, has written an in-depth article on implementing progressive layout in Flex applications, including using progressive layout in data-driven applications: "Building Flex Applications with Progressive Layout (www.macromedia.com/devnet/flex/articles/prog_layout.html)."

Handling Large Data Sets

Sometimes your application may appear slow because it manages a large amount of data at once. Matt Chotin, a software engineer on the Flex team, has some excellent entries on data management that address this issue. He discusses how to incorporate paging, sorting, and improving perceived performance on his blog (www.markme.com/mchotin/archives/cat_data_management.cfm).

Playing Complex Effects Smoothly

You may notice that transition effects seem choppy, especially when your effect has a short duration applied to a large view. What defines choppy? For example, a Fade effect that fades in a few distinct alpha stages, instead of a smooth and seamless fade. Or a Zoom effect that zooms in a few distinct sizes, instead of a gradual and smooth Zoom. There are a few ways to tweak your transition effects to play in a smoother fashion. Try the following suggestions and see which work best to improve effects in your application:

- **Increase the duration of your effect with the duration property.** Doing this spreads the distinct, choppy stages over a longer period of time, which lets the human eye fill in the difference for a smoother effect. It really makes a difference.

- **The less there is for Flash Player to redraw during an animation, the smoother the effect plays**. To do this, make parts of the target view invisible when the effect starts, play the effect, and then make those parts visible when the effect has completed. The populating of controls happens so quickly that the human eye does not notice any sort of delay or sudden appearance of the controls. Coding this is simple: You hook into the `effectStart` and `effectEnd` events to control what is visible before and after the effect.

  If you look at the SWF file in the online version of this article, at www.macromedia.com/devnet/flex/articles/client_perf_08.html, the panel has a populated DataGrid control with a fast Fade effect applied to it, and a duration of 250 milliseconds. As you toggle the panel's visibility, see how the Fade plays in an abrupt fashion? Adding the following code to the `<mx:Panel>` tag lessens the number of objects Flash Player must redraw during an animation:

  ```
  effectStart="myDataGrid.visible=false"
     effectEnd="myDataGrid.visible=true"
  ```

  Now Flash Player does not draw the DataGrid control, but concentrates on redrawing the empty Panel container. Now look at the second SWF file at www.macromedia.com/devnet/flex/articles/client_perf_08.html, See the difference? The effect is smoother and better-looking. Because the duration is so fast, there is no noticeable disappearing and appearing of the DataGrid. Only use this technique if the effect duration is relatively short (500 ms or less), and only on a showEffect or hideEffect. In other situations, hiding pieces of the object will not appear seamless.

The Resize effect, a native effect in Flex, has some of this functionality already built in. You can use the `hideChildren` property to specify an array of panels whose children should be hidden while the effect plays. This property only works with Panels and help your Resize effects to play smoother. Before the Resize animation plays, Flex iterates through the `hideChildren` array and hides the children of each of the specified panels. Note: You cannot use the `hideChildren` property with an effect declared in MXML (such as in the `<mx:Resize>` tag). The effect must be triggered in ActionScript to use of the `hideChildren` functionality.

- **Avoid bitmap-based backgrounds.** Oftentimes designers give their views background images that are solid colors with gradients, slight patterns, and so forth. To ease what Flash Player redraws, try switching your background image to a solid background color. Or, if you want a slight gradient instead of a solid color, use a background image that is a SWF or SVG file. These are easier for Flash Player to redraw than standard JPG or PNG files.

  Sometimes animations play in a choppy fashion because background processing occurs and interferes with the animation. You may notice this when you have an effect attached to a handler that populates controls from a web-service result, or when you have an effect accompanying the creation of a large view. The set of tags are a subclass of the Effect class (Fade, Move, Resize, WipeLeft, and so on) have a public property, `suspendBackgroundProcessing`. When it is true, it blocks all background processing like measurement and layout while the effect plays. The default is `false`. Macromedia suggests that you set this property to `true` for a smooth playing of effects. However, you must realize that when you switch on `suspendBackgroundProcessing`, your effect cannot be interrupted while playing. Because of this, there are a few cases where you should avoid using `suspendBackgroundProcessing="true"`. One common use of effects is to play an effect while the application waits for a web-service result to return. After the web-service result returns, the result handler tries to interrupt and stop the effect. If `suspendBackgroundProcessing` is set to `true`, the result handler cannot interrupt the effect, and the effect plays continuously, hanging the application. Avoid using `suspendBackgroundProcessing` in these cases.

Achieving Great Performance with Runtime Styles

Runtime cascading styles are very powerful, but you should use them sparingly and in the correct context. Dynamically setting styles on an instance of an object means calling UIObject's `setStyle()` method. The `setStyle()` method is one of the most expensive calls in the Flex application model framework, because the call requires notifying all the children of the newly styled object to do another style lookup. The resulting tree of children that must be notified can be quite large.

A common mistake that impacts performance is overusing or unnecessarily using the `setStyle()` method. In general, you *only* need the `setStyle()` method when you want to change styles on existing objects. Do not use it when setting up styles for an object for the first time. Instead, set styles in an `<mx:Style>` block, as explicit style properties on the MXML tag, through an external CSS style sheet, or as global styles. It is important to initialize your objects with the correct style information, if you do not expect these styles to change while your program executes (whether it is your application, a new view in a navigator container, or a dynamically created component).

Some applications need to call the `setStyle()` method during the application or object instantiation. If this is the case, call the `setStyle()` method *early* in the instantiation phase to avoid unnecessary lookups. Early in the instantiation phase means setting styles from the component or application's `initialize` event, instead of the `creationComplete` or other event. By setting the styles as early as possible during initialization, you avoid unnecessary style notification and lookup.

Using Dynamically Repeating Controls for Better Performance

New in Flex 1.5 is the addition of the HorizontalList and TileList controls. You can use these controls for layouts that require dynamically repeating elements. They perform much better than layouts that used Repeaters. In fact, layouts created during the Flex 1.0 timeframe that used a Repeater may often be replaced by a combination of the HorizontalList or TileList controls and cell renderers for better performance.

Using the HorizontalList and TileList Controls

The HorizontalList and TileList controls are List controls that display data elements horizontally or in a tile layout, much like the HBox or Tile containers. Unlike the Repeater object, performance with these controls is determined by what is visible in the HorizontalList and TileList at that time. This behavior reduces instantiation time of the view significantly; a Repeater's instantiation time will always be equal to or worse than the HorizontalList and TileList controls.

The HorizontalList and TileList controls usually contain a horizontal or vertical scroll bar, used to access all the items in the list. When a user scrolls, it triggers the creation of subsequent elements in the List. Thus, the user avoids creating all the possible elements at startup; they are created only when requested.

The HorizontalList and TileList controls perform much better than a Repeater object. In most cases it is a better choice to use them instead of a Repeater. However the Repeater control is still available in Flex; it is better to use the Repeater control to repeat simple elements. For example, it would make more sense to repeat a collection of RadioButton controls using a Repeater then a Horizontal or TileList control.

Improving a Repeater Object's Performance

There are a few things to think about if you need to improve your Repeater object's performance. First, if you are using containers as the child of the Repeater object, check whether using a HorizontalList or TileList control would be better. If that is not the case, ensure that the containers used as the children of the Repeater do not have unnecessary container nesting and are as trim as possible. If a single instance of the repeated view takes a noticeable time to instantiate, repeating makes it worse. As mentioned previously in this article, multiple Grid containers in a Repeater object do not perform well because Grid containers themselves are resource-intensive containers to instantiate. An alternative solution is to try a List control with a custom cell renderer or the HorizontalList or TileList controls.

You should also set the `recycleChildren` property to `true` to improve a Repeater object's performance. The `recycleChildren` property is a boolean value that, when set to `true`, binds new data items into existing Repeater children, incrementally creates new children if there are more data items, and destroys extra children that are no longer required.

When you set this property to `false`, the Repeater object recreates all the repeated objects when you swap dataProvider properties, sort, and so on, which causes a performance lag. The FlexStore sample application (www.macromedia.com/flex/samples/flexstore/flexstore.mxml?versionChecked=true) has an example of `recycleChildren` at play. When you load the application, notice the Sort by option under the product thumbnail view. This enables users to sort the product thumbnails based on name or price. The ordering of the Repeater object's dataProvider property is what changes, the thumbnail views do not. By setting the `recycleChildren` property to `true`, the Repeater object does not recreate each thumbnail view; it simply reshuffles the dataProvider property based on name or price.

The default value of the `recycleChildren` property is `false` to ensure that you do not leave stale state information in a repeated instance. For example, suppose you use a Repeater object to display photo images and each Image control has an associated NumericStepper control for how many prints you want to order. Some of the state information, such as the image, comes from the dataProvider property, while other state information, such as the print count, is set by user interaction. If you set the `recycleChildren` property to `true` and page through the photos by incrementing the Repeater object's `startingIndex` value, the Image controls bind to the new images, but the `NumericStepper` control maintains the old information! You should use `recycleChildren="false"` only if it is too cumbersome to reset the state information manually, or if you are confident that modifying your dataProvider property should not trigger a recreation of the Repeater object's children.

This may go without saying, but the `recycleChildren` property has no effect on a Repeater object's speed when the Repeater object loads the first time. The `recycleChildren` property only improves performance for subsequent Repeater occurrences. If you know that your Repeater object will only create children once, there is no need to use the `recycleChildren` property or worry about the stale state situation.

Improving Performance of Charting Components

Flex provides a very robust library of charting components that give you a two-dimensional visual representation of your data. These charting components follow the same guidelines as other Flex components, including the same performance drawbacks.

The Flex charts have been designed to perform well. All charts cache intermediary values in the transformation from data to screen, so that only the minimum amount of recalculation occurs in response to any change to the data or chart. The most expensive actions to perform in Flex charts is forcing a chart to redraw an axis, or forcing a chart to recalculate its labels. In fact, it is faster to resize a chart than to change its dataProvider (a change that requires a chart to potentially redraw an axis or recalculate labels). Below are further tips that you can use to improve the performance of your Flex charting components.

- **When possible, set the `filterData` property to false**. In the transformation from data to screen coordinates, the various Series types filter the incoming data to remove any missing values or values outside the range of the chart that would render incorrectly if drawn to the screen. For example, a chart representing vacation time for each week in 2003 might not have a value for the July fourth weekend since the company was closed. If you know your data model will not have any missing values at runtime, or values that fall outside the chart's data range, you can instruct a series to explicitly skip the filtering step by setting its `filterData` property to false, earning you a small performance boost.
- **If possible, don't let a `LinearAxis` autocalculate its range.** A `LinearAxis` calculating its numeric range can be a costly calculation. If you know reasonable minimum/maximum values for the range of your `LinearAxis`, specifying them help your charts render quicker. Also, specifying an interval (the numeric distance between label values along the axis) value improves performance.
- **Have your `CategoryAxis` `dataProvider` and Series `dataProvider` refer to different objects**. Modifying a `CategoryAxis` object's `dataProvider` is more expensive than modifying a Series object's `dataProvider`. If the data bound to your chart is going to change, but the categories in your chart will stay static, have the `CategoryAxis`' `dataProvider` and Series' `dataProvider` refer to different objects. This prevents the `CategoryAxis` from reevaluating its `dataProvider`, which is a resource-intensive computation.
- **Improve render time of your `AxisRenderers` by setting particular styles**. The `AxisRenderers` perform many calculations to make sure they render correctly in all situations. The more help you can give them in restricting their options, the faster they will render. Explicitly setting the following styles on the `AxisRenderer` will improve performance: `labelRotation` and `canStagger`. You can set these styles within the tag or in CSS.

- **Specify gutter styles when possible.** The gutter area of a Cartesian chart is the area between the margins and the actual axis lines. With default values, the chart adjusts the gutter values to accommodate axis decorations. Calculating these gutter values can be resource intensive. By explicitly setting `gutterLeft`, `gutterRight`, `gutterTop`, and `gutterBottom` values, your charts draw quicker and more efficiently.
- **DropShadows are optional; if you don't need them don't use them.** The default HALO styles for `BarSeries`, `ColumnSeries`, and `LineSeries` use `ShadowRenderers`. `ShadowRenderers` draw drop shadows beneath the data elements. If `DropShadows` are not necessary, switch to the `Simple` renderers (`SimpleBoxRenderer` for `ColumnSeries` and `BarSeries`, `SimpleLineRenderer` for `LineSeries`); this improves the rendering speed of your charts.

Performance Tuning and Profiling Your Own Flex Application

Test the performance of your application early and often. It is always best to identify problem areas early and resolve them in an iterative manner, rather then trying to shove performance enhancements into existing, poorly performing code at the end of your application development cycle. The following subsections investigate using Runtime Shared Libraries (RSLs) to improve performance, as well as describe two approaches to performance testing your Flex client: using the ActionScript profiler and using a code snippet that times application initialization. We also provide a handy solution to time component and data gestures using the `ActionScript getTimer()` method. (See also the "Load Testing Your Flex Application" section in the "Flex Application Performance: Tips and Techniques to Improving Flex Server Performance" article section.)

Using Runtime Shared Libraries (RSLs)

You can shrink the size of your application's resulting SWF file by externalizing shared assets into standalone files that you can separately download and cache on the client. Multiple Flex applications can load these shared assets at runtime, but each client need only to download them once. These shared files are called Runtime Shared Libraries.

Flex projects that have multiple Flex applications downloaded to the client can leverage RSLs for better performance. More specifically, the time it takes to download a Flex application once the initial RSL has been downloaded is significantly reduced.

However, not all applications benefit from RSLs. The following are examples of Flex applications that might use RSLs for better performance:

- Large applications that load multiple smaller applications can be linked to a shared RSL
- A family of applications on a server built with a shared RSL
- A frequently changing application that has a large set of infrequently changing components

- An infrequently changing application that has frequently changing data or assets

  Depending on the type of Flex project you are developing, RSLs may or may not offer a performance benefit. Roger Gonzalez, a developer on the Flex team, has a more in-depth article focused on RSLs: "Using Runtime Shared Libraries (www.macromedia.com/devnet/flex/articles/rsl.html)."

Using the ActionScript Profiler

The ActionScript profiler records the time Flash Player takes to perform tasks in ActionScript. Most commonly, you use the profiler to determine how long an ActionScript function or method takes to execute, how often it is called, and how much time is spent executing in its descendant. This helps identify which objects might be taking too long to initialize, or whether there are bottlenecks due to heavy graphics use or poor coding. However, running the profiler adds overhead to the application you are analyzing. This is because the profiler runs with the Flash Debug Player version, which is slower than the release version. Analyze the results returned with the profiler relative to each other, but do not take them as correct absolute times. Running an application in the release version of Flash Player will yield different results when compared to running the same application with the Flash Debug Player version. The Flex documentation has more information on how to install and run the ActionScript profiler. Macromedia Flex Evangelist Christophe Coenraets also has an excellent blog entry (http://www.markme.com/cc/archives/2004_04.cfm) on optimizing application performance with the ActionScript profiler.

Calculating the Application Initialization Time

A more simple approach to performance profiling is to use code to gauge startup time. The following snippet times application initialization time (the time it takes the Application object to create, measure, lay out, and draw all of its children); it does not include the time to download the client SWF, or any of the server-side processing such as checking the Flash Player version, checking the SWF cache, and so on. The following example shows a sample Flex application that, when invoked in a browser, shows a simple form populated with controls, with the time it took to initialize and print a label.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml"
   creationComplete="doLater(this,'doneNow')">
<mx:Script><![CDATA[
var dp = [{food:"apple", type:"fruit", color:"red"},
   {food:"potato", type:"vegetable", color:"brown"}, {food:"pear",
   type:"fruit", color:"green"},
  {food:"orange", type:"fruit", color:"orange"},{food:"spinach",
   type:"vegetable", color:"green"},{food:"beet",
   type:"vegetable", color:"red"}];
function doneNow()
{
  doLater(this, "reallyDoneNow");
}
function reallyDoneNow()
{
  timerLabel.text += getTimer() + " ms"
}
]]></mx:Script>
<mx:Form>
  <mx:FormHeading label="Sample Form" />
  <mx:FormItem label="List Control">
   <mx:List dataProvider="{dp}" labelField="food"/>
  </mx:FormItem>
  <mx:FormItem label="DataGrid control">
   <mx:DataGrid width="200" dataProvider="{dp}"/>
  </mx:FormItem>
  <mx:FormItem label="Date controls">
   <mx:DateChooser />
   <mx:DateField />
  </mx:FormItem>
  <mx:FormItem label="Load Time">
   <mx:Label id="timerLabel" fontSize="12" fontWeight="bold"
   text="The application initialized in "/>
  </mx:FormItem>
</mx:Form>
</mx:Application>
```

The first thing is to get a baseline reading of your application startup time. Ensure that you exit all other running applications, verify that you are using the release version of Flash Player, and run the application three times to get three initialization times. The average of these three times is your baseline reading. As you implement performance tuning, confirm that the startup time is indeed getting faster.

This code will also help you see which portion of your application is the slowest and identify whether you can speed it up. To try this, selectively remove parts of your application to see whether the single part's startup time is relative to the entire application's startup time. This iterative process highlights the problem areas of your application. For example, this methodology might show you that View B of your application loads in six seconds out of an overall startup time of twelve seconds. View B is a problem area for which you can drill down into and investigate performance alternatives. This process is simpler than setting up and using the ActionScript profiler, although it yields much less detailed information.

Using getTimer() To Time Component and Data Gestures

The `getTimer()` function is a very useful ActionScript function that returns the number of milliseconds since a Flex client application has been running in the browser. Attaching `getTimer()` calls to events enables you to time component and data gestures. Brandon Purcell, coauthor of this article and a support engineer at Macromedia, has a blog entry (www.bpurcell.org/blog/index.cfm?mode=entry&entry=1017) explaining how to time Flex data service calls using `getTimer()`.

Feedback and Support

We have made every effort to ensure the accuracy of this article and all code included. Feedback regarding this article and all Flex performance issues is always appreciated.

Flex Application Performance: Tips and Techniques for Improving Flex Server Performance

Macromedia Flex is a powerful platform that offers the ability to create Rich Internet Applications (RIAs). Misusing this power can result in areas of poor performance. This article explores these performance issues and offers tips on how to get the most out of your Flex server. More specifically it explores:

- The flexconfig.xml file and its impact on performance
- The Flex caching model
- Runtime Shared Libraries (RSL)
- Efficient data delivery to the Flex client
- Using the JSP tag library efficiently
- Precompiling MXML pages
- JVM tuning
- Flex deployment options

Note that the tips suggested in this article do not apply to all Flex applications. It is important to analyze the structure of your own application and modify the suggestions to tailor them to your needs. For ongoing coding and conceptual help, you can use the Flex support forums (www.macromedia.com/cfusion/webforums/forum/index.cfm?forumid=60).

Note also that this is the second part of a two-part article. The first part of this article, Flex Application Performance: Tips and Techniques for Improving Client Application Performance, discusses the client-side coding techniques that you can implement to improve the performance of your Flex client application.

Exploring Caching in the Flex Presentation Server

Flex uses an in-memory cache to handle requests for MXML pages. The first time it invokes an MXML page, the compiler creates an SWF file and stores it in the Flex cache. Flex also creates and caches an HTML wrapper page, and serves subsequent page requests from this cache. Figure 1 shows the flow of traffic.

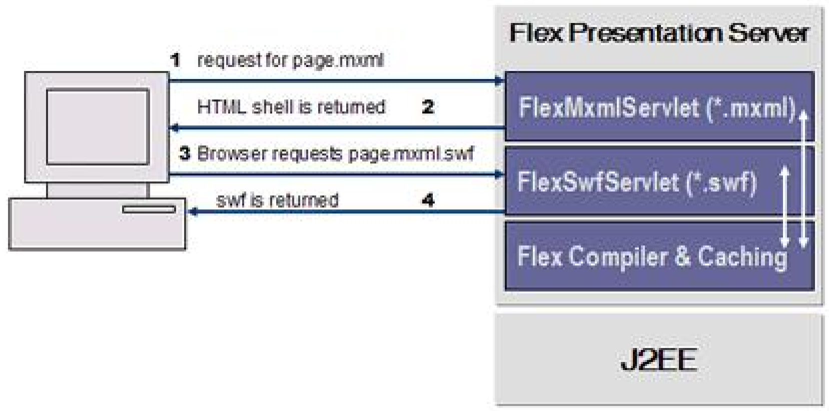

Figure 5: *Caching requests for MXML pages*

Flex also caches dependent files such as SWC, MXML, and ActionScript files. Flex keeps a reference of all dependent files, such as CSS style sheets, images used in the `<mx:Image>` tag, and ActionScript files included with the `<mx:Script>` tag, and stores them in the /WEB-INF/flex/cache.dep file. If any of these dependent files change, Flex recompiles the MXML file.

The Flex caching settings are configured in the `<cache>...</cache>` section of the flex-config.xml configuration file. Flex enables caching by default, using the entry `<cache-mxml>true</cache-mxml>`. If you set this entry to `false`, all requests to an MXML file force a recompilation of the MXML document. The `<content-size>500</content-size>` entry in the flex-config.xml file defines the default maximum size of the cache, 500 entries. The content-value size is not defined by the total number of MXML files, but by the total number of cached entries. For example, a single MXML file writes two entries in the cache: one for the HTML shell and one for the SWF file. In most cases, the default setting of 500 is more than enough but there might be cases in large enterprise applications where you have a sizeable number of MXML documents. Flex will create additional cache entries for different URL pairs (for example, `page.mxml?accessible=true`). By setting `accessible=true` in the URL, you create a new unique SWF file and HTML shell that is added to the cache. There is not a significant amount of memory overhead to increasing this value because most HTML shells are only a few kilobytes in size and most Flex SWF files are around 100 to 150K.

When you use the JSP tag library to inline MXML into a JSP or CFML page, Flex uses a separate cache for the MXML code fragments. Flex defines this cache with the `<mxml-size>` entry with a default value of 500. The following example shows a code fragment from a JSP page:

```
<mx:Application>
  <mx:Label text="Hello"/>
</mx:Application>
```

Flex caches this code fragment in the source cache defined by the `<mxml-size>` entry. The resulting SWF file and HTML shell generated from compiling the source are cached in the content cache, defined in the `<content-size>` caching setting. If at anytime the source fragment changes within the page, a new cache entry is entered into the source cache, and Flex compiles a new SWF file.

```
<mm:mxml>
 <mx:Application>
  <mx:Label text="Hello" />
 </mx:Application>
</mm:mxml>
```

When Flex reaches the size limit in a cache, it flushes the least recently used file. In Flex 1.0 there is no way to see how many entries are in the cache and there is no way to flush the cache while the server is running. As you will see in the next section, this is only a problem if the server is running with `production-mode=true`. During development, the file-watcher monitors all dependent files defined in /WEB-INF/flex/cache.dep; if a file changes, Flex recompiles the corresponding MXML file and refreshes the cache. For more details on the caching implementation in Flex, read the Configuring Caching (livedocs.macromedia.com/flex/1/flex_docs/36_admi5.htm#wp121792) section in the Developing Flex Applications documentation.

Modifying the flex-config.xml File Configuration Settings

The simplest way to improve performance in a production environment is to set the `production-mode` attribute to `true` in the flex-config.xml file. This achieves several things in Flex.

- It disables all debugging and profiling features set in the `<debugging>` block of the flex-config.xml file.
- It also forces Flex to ignore query string parameter overrides, such as `?debug=true` and `?asprofile=true`.

When you enable production mode, Flex only checks for changed files on server start-ups. Flex does not use the `<file-watcher-interval>` setting to continuously poll files. You must restart the application or the J2EE server instance to update MXML or dependent files while in production.

One other configuration setting that can improve performance is the `<optimize>` property in the `<compiler>` section of the flex-config.xml file. The `<optimize>` property defaults to `true`. By setting `<optimize>` to `true`, the ActionScript optimizer reduces the size of the resulting SWF file by 10 to 15 percent. The optimizer changes generated ActionScript code prior to compilation. This results in reduced compilation time and improved runtime performance of Macromedia Flash Player. During development, if you are using trace statements, you should set `<optimize>` to `false` to prevent the compiler from removing your trace statements.

Runtime Shared Libraries (RSLs)

With Flex 1.0 when Flex compiled an MXML file and its assets into a SWF file, the resulting SWF file is a single entity, consisting of all the base application model components (such as Button, CheckBox, and Panel components), graphical assets, embedded data, and custom components. The result can be a large file that can be slow to download. In many cases, Flex implementations have multiple applications that share some of the same assets. But clients were required to download the entire application and all of the components for each Flex application they accessed.

Flex 1.5 introduced the concept of Runtime Shared Libraries (RSLs). With RSLs, you can reduce the size of your application's SWF file by externalizing shared assets into standalone files that can be separately downloaded and cached on the client. These shared assets are loaded by any number of applications at runtime, but only need to be transferred to the client once.

If you have multiple applications but those applications share a core set of graphic files, components, and other assets, your users only have to download those assets once in an RSL. You can reduce the resulting file size for your main application dramatically. If a file in one of the RSLs changes, Flex recompiles and resends that RSL to the client. Flex does not recompile the application and other unchanged RSLs.

Accessing Flex Data Services

How you choose to access data in your Flex application impacts performance. Because the application is cached on the browser after the first request, data access is responsible for significantly affecting performance while the application runs. Flex provides several solutions for data delivery to the client. It delivers data through runtime services that invoke Java classes loaded in the Flex classpath, or sends proxy requests to web services or HTTP servers. Figure 2 shows the available Flex data services.

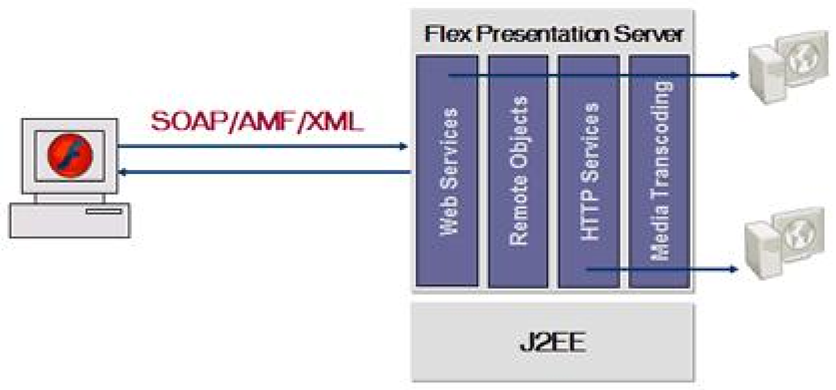

Figure 6: *Flex data services*

The following list describes the Flex data services:

- **Web services proxy**—Due to the Flash Player security sandbox, requests to web services can only be made to the same domain that loaded the SWF file. In many cases, web services will be located on another domain. In Flex, all web service requests go through a web services proxy running on the Flex presentation server. The following snippet shows a call to a web service:

  ```
  <mx:WebService id="ws" wsdl="http://acme.com/stock.wsdl">
    <mx:operation name="getQuote"/>
  </mx:WebService>
  ```

  When the SWF file loads on the client, it sends a request to the web services proxy requesting the WSDL file. Then, the web services proxy makes a request to the WSDL file and sends the response back to the client, using the proxy that the response sent back to the original client. All client server interaction using the `<mx:Webservice>` tag passes through the proxy unless you use the `useProxy="false"` property. When you use `useProxy="false"`, the Flash client makes the web service call directly to the end service. For this to work properly with the sandbox security in the Flash client, the end service must have a crossdomain.xml file in place. In some cases you have no control over the end service, so you need the proxy. In some cases you may experience issues using HTTPS and retrieving error messages when not using the proxy. Macromedia recommends that you use the proxy unless you have a good reason not to.
- **Remote object AMF gateway —The remote object AMF gateway** enables you to access server-side objects (Java Beans, EJBs, POJOs) running on the Flex presentation server. The remote object proxy uses two different encoding mechanisms, which you choose by modifying the encoding property in the `RemoteObject` tag. Flex 1.0 provided two encoding mechanisms for transferring the data AMF (Action Message Format) and SOAP. In Flex 1.5, SOAP encoding has been deprecated. AMF provides a binary protocol for data transfer between server and client. The following code shows an example of using the `RemoteObject` tag with AMF:

  ```
  <mx:RemoteObject id="ro" src="samples.StockBean" encoding="AMF">
    <mx:method name="getQuote">
    </mx:method>
  </mx:RemoteObject>
  ```

- **HTTP services**—Another data access method uses the HTTPService tag with remote URLs to load XML into Flash Player. The HTTP services proxy handles requests from the Flash client, and the proxy invokes the URL and sends the response back to the client using a proxy. The following snippet shows how to use the HTTPService tag:

  ```
  <mx:HTTPService id="myRequest" url=http://acme.com/mydata.xml>
  </mx:HTTPService>
  ```

You can use a standards-based approach to access data from your Flex applications, but in some cases you can improve performance with other options. Using `the <mx:Webservice>` tag enables you to use a standards-based approach but doesn't always yield the best performance. Also, there is an extra bit of overhead for the additional XML with the SOAP encoding in comparison to AMF. On average, AMF is three times faster than SOAP and in cases of very large payloads can be up to six times faster. This is because AMF uses a binary protocol that greatly reduces the size of the payload compared to XML-based soap packets for the same data. The performance of SOAP with web services is also dependent on your web services implementation. Different application servers use different web service back ends, so you might see performance differences depending on the implementation. The only way to understand how well your implementation will perform is to load test your services. Carrying the same data, SOAP is usually double the size of AMF. By using AMF, you can reduce the overall bandwidth of your applications.

The two best-performing methods of data delivery are XML using HTTPService or RemoteObject. As mentioned previously, web services may be slower due to the overhead of deserializing and serializing the SOAP packets on the server and client. Many times, the choice depends on your existing applications and how you choose to integrate with your back end server-side resources. The performance of web services can be highly dependent on your application server's underlying implementation of the web services engine, so you should load test to see how well it performs.

You also have the option to use remote classes for more complex objects with the <mx:RemoteObject> tag. Always consider the overhead of serializing and deserializing remote classes and use them only in cases where they are needed. When using remote classes, make sure that you use the `Object.registerClass()` method to specify the fully-qualified name of the corresponding Java class on the server. By mapping the ActionScript class directly to its Java server-side equivalent, you can prevent the AMF gateway from having to search for the equivalent. Find more information on `Object.registerClass()` in the Developing Flex Applications document. Serialization for list-based objects will occur faster in the client.

You also have the option of bypassing the Webservice and HTTPService proxies in Flex allowing you to invoke the services directly. If the service you are trying to invoke is on the same host or shares the same domain name as the location of the MXML and SWF files, you can invoke the service directly by adding the `useProxy="false"` property. When you specify `useProxy="false"`, the Flash client no longer routes calls through the Flex proxy; instead it routes it directly to the end service. If the MXML document and service are served from different domains or hosts, then you must place a crossdomain.xml file on the server that is hosting the service. Read more about this procedure in the Flash 7 security article (www.macromedia.com/devnet/mx/flash/articles/fplayer_security_03.html). Macromedia recommends that you develop using the proxy because it aids in debugging and that you only turn off the proxy if you're confident that performance requires it during production. The proxy also handles authentication and authorization. With it, you can use named services instead of embedding the URLs in the MXML files; it also resolves conflicts by handling same-named cookies from different domains.

Using the JSP Tag Library

Flex provides a JSP tag library that enables you to integrate MXML directly into a JSP or ColdFusion page. The tag library enables developers to build hybrid applications that integrate both HTML and Flex applications into a page. For example, if you had an HTML application and wanted to slowly introduce rich user interfaces into it, you could leverage the tag library to quickly and easily add it to your application. A good example is a stock ticker that you add to an HTML application. Using an HTML post, a user would have to refresh the entire page for a quote. By adding a rich user interface, the user would see the results appear without the entire page refreshing.

Used incorrectly, however, the tag library can cause significant performance degradation by forcing excessive compilation. The following snippet shows an example of using the Flex tag library. In this example, Flex compiles the MXML file and stores the SWF file in the cache. As long as the MXML code does not change, Flex continues to use the cached version of the content file.

```
<%@ taglib uri="FlexTagLib" prefix="mm" %>
<mm:mxml>
  <mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml"
   width="200" height="200">
    <mx:Label id="label0" text="Hello World "/>
  </mx:Application>
</mm:mxml>
```

If the Flex code changes dynamically, as it does in the following example, Flex compiles a new version every time the MXML content changes. In this case, for each unique user who logs in through the application, the application sets a new session.username value. This causes the MXML content to change, and causes Flex to cache an additional version of the code and the SWF content.

```
<%@ taglib uri="FlexTagLib" prefix="mm" %>
<% session.setAttribute("username", "brandon"); %>
<mm:mxml>
```

```
  <mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml"
   width="200" height="200">
    <mx:Label id="label0" text="Hi <%=
   session.getAttribute("username") %> "/>
  </mx:Application>
</mm:mxml>
```

This is not the preferred approach since it causes excessive compilation. For example, if 500 unique users accessed this application at the same time, the value of session.username would be 500 different values resulting in 499 additional MXML compilations. It will also add a significant number of entries in the cache. A better approach is to pass variables to the Flex application using the `<param>` tag from the Flex tag library. You can then use the value of that variable, as long as you declare the variable inside an MXML script block. The following code shows the best practices approach to solve the problem explained in the previous code. With this approach the page only compiles once, and each time a unique user accesses the application, the app passes the unique user name through the `mm:param` value.

```
<% session.setAttribute("username", "brandon"); %>
<%@ taglib uri="FlexTagLib" prefix="mm" %>
<mm:mxml>
<mm:param name="userName" value="<%=
   session.getAttribute("username") %>" />
  <mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml"
   width="200" height="200">
  <mx:Script>
   var userName:String;
  </mx:Script>
    <mx:Label id="uName" text="Hi {userName}"/>
  </mx:Application>
 </mm:mxml>
```

The `<param>` tag works fine for simple values, but what if you need to pass an XML tree or complex objects into your Flex application? The following example passes XML into the MXML document using the tag library. There are two ways to accomplish this task and the preferred option for good performance depends on whether the XML is static or dynamic. If the XML is static, it performs better if you embed the JSP code within the MXML document, as shown in this example:

```
<%@ taglib uri="FlexTagLib" prefix="mm" %>
<html>
<%
  String tree="<node label='Option 1'><node label='Option
   1.1'/><node label='Option 1.2'/></node><node label='Option
   2'/>"; %>
<mm:mxml>
  <mx:Application width="250" height="250"
   xmlns:mx="http://www.macromedia.com/2003/mxml">
    <mx:XML id="myTree">
      <%= tree %>
    </mx:XML>
    <mx:Tree dataProvider="{myTree}"/>
  </mx:Application>
</mm:mxml>
</html>
```

If the XML is dynamic, Flex recompiles the MXML each time the XML changes, which can lead to poor performance. The best approach for handling dynamic data is to load the XML dynamically from the server, using the `<mx:HttpService>` entry to a JSP or servlet that delivers the XML. The following code invokes a JSP that feeds the XML back to the Flex application when it initializes. After the XML returns, Flex binds the result directly to the `<mx:Tree>` control.

```
<%@ taglib uri="FlexTagLib" prefix="mm" %>
<html>
<mm:mxml>
   <mx:Application width="250" height="250"
    xmlns:mx="http://www.macromedia.com/2003/mxml"
   initialize="xmlFeed.send()">
     <mx:HTTPService id="xmlFeed"
    url="@ContextRoot()/perfpaper/xmlfeed.jsp" resultFormat="xml"/>
     <mx:Tree dataProvider="{xmlFeed.result}"/>
   </mx:Application>
</mm:mxml>
</html>
```

The Flex JSP tag library provides a powerful set of features to developers for embedding MXML into your existing applications and building hybrid Rich Internet Applications (RIAs). If used correctly, they perform very well, just remember that dynamic MXML causes recompilation and can fill up the cache quickly. For complete details on using the JSP tag library, see the Flex documentation (livedocs.macromedia.com/flex/1/flex_docs/35_jsps.htm#wp121778).

Precompiling MXML Pages

The earlier section on caching discusses how, for best performance, you must recompile the MXML files and load them into the cache after a server restart. Another option is to precompile your MXML pages with the headless compiler, mxmlc, and create your own HTML shell to load the SWF files into the browser. This option enables you to deploy bytecode, but is somewhat more involved than letting Flex build your HTML wrappers for you on the fly. Many of the things, such as player detection and history management will no longer be generated automatically. You must code them into your HTML wrapper to provide those features.

The following code demonstrates an example of precompiling an MXML file that creates an HTML wrapper. It has a TabNavigator container with four tabs. If you put this code in an MXML file and invoke it through a browser, you can click the tabs and use the browser's Back and Forward buttons to move through the Flex application.

```
<?xml version="1.0"?>
 <mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml">
 <mx:Panel title="My Application">
 <mx:TabNavigator borderStyle="solid">
 <mx:VBox label="Pane1" width="300" height="150" >
 <mx:TextArea text="Hello World" />
 </mx:VBox>
 <mx:VBox label="Pane2" width="300" height="150" >
 </mx:VBox>
 <mx:VBox label="Pane3" width="300" height="150" >
 </mx:VBox>
 <mx:VBox label="Pane4" width="300" height="150" >
 </mx:VBox>
 </mx:TabNavigator>
 </mx:Panel>
 </mx:Application>
```

To prevent this file from compiling each time the server restarts, you can use mxmlc to precompile the file as follows:

```
{flex-root}bin\mxmlc ..\jrun4\servers\default\flex\TabNav.mxml
```

For complete details on using mxmlc, see the Flex documentation livedocs.macromedia.com/flex/1/flex_docs/36_admi2.htm#wp121781).

Note: Please read the Flex end-user license agreement for details on distributing SWF files created with mxmlc. In the trial and development versions of Flex, all SWF files created with mxmlc expire one day after compilation. With the full version of Flex, they will not expire.

In the following example, mxmlc creates the SWF file in the same directory as the MXML. The next step is to build the HTML shell and integrate the SWF file into the HTML, JSP, or CFML files, or in a servlet. The easiest approach is to use a shell that an MXML file within the same Flex application generated. By using a Flex-generated shell, you retain the Flex history management features. All you need to do is replace the file.mxml.swf entry with the file.swf entry in four locations. For example:

```
<script language='javascript' charset='utf-8' src='/flex/flex-
   internal?action=js'></script>
 <noscript>
   <object classid='clsid:D27CDB6E-AE6D-11cf-96B8-444553540000'
   codebase='http://download.macromedia.com/pub/shockwave/cabs/fla
   sh/swflash.cab#version=7,0,14,0' width='100%' height='100%'>
     <param name='flashVars' value=''>
     <param name='src' value='TabNav.swf'>
     <embed
   pluginspage='http://www.macromedia.com/go/getflashplayer'
   width='100%' height='100%' flashVars='' src='TabNav.swf'/>
   </object>
 </noscript>
 <script language='javascript' charset='utf-8'>
   document.write("<object classid='clsid:D27CDB6E-AE6D-11cf-96B8-
   444553540000'
   codebase='http://download.macromedia.com/pub/shockwave/cabs/fla
   sh/swflash.cab#version=7,0,14,0' width='100%' height='100%'");
   document.write(">");
   document.write(" <param name='flashVars'
   value='historyUrl=%2Fflex%2Fflex%2Dinternal%2Fhistory%2Fhistory
   %2Ehtml&lconid=" + lc_id +"'>");
   document.write(" <param name='src' value='TabNav.swf'>");
   document.write(" <embed
   pluginspage='http://www.macromedia.com/go/getflashplayer'
   width='100%' height='100%'");
   document.write("
   flashVars='historyUrl=%2Fflex%2Fflex%2Dinternal%2Fhistory%2Fhis
   tory%2Ehtml&lconid=" + lc_id +"'");
   document.write(" src='TabNav.swf'");
   document.write(" />");TabNav.swf
   document.write("</object>");
 </script>
 <script language='javascript' charset='utf-8'>
```

```
  document.write("<br><iframe src='/flex/flex-
    internal/history/history.html' name='_history' frameborder='0'
    scrolling='no' width='22' height='0'></iframe></br>");
</script>
```

For history management to work, you must ensure that the context-root of the application is correct. If you deploy Flex with a context root of /flex, then you must prefix all URLs in the HTML wrapper with /flex. In the previous example, the context-root is /flex. Using the information provided, you can improve the startup time of the first request for your application, because Flex does not need to compile the MXML files. Precompiling the application also enables you to distribute applications without the source code. Using ant, or another build tool, you could wrap up the mxmlc functionality and build HTML shells into an automated process.

You can find more information on building your own HTML wrappers in the Flex documentation (livedocs.macromedia.com/flex/1/flex_docs/38_dep24.htm#wp147786). The documentation also describes the steps for adding player detection to your HTML wrappers.

Deployment Options

You have two options when deploying Flex. You can leverage your existing hardware and application servers, or you can deploy Flex on a dedicated system. The best investment is to leverage your existing hardware. You may be concerned that adding Flex to an existing environment would tax the existing infrastructure by adding additional load to the servers. In fact, the opposite is true. Using Flex can actually decrease server load.

Adapting an existing JSP or servlet-based application to a Rich Internet Application (RIA) decreases server load by decreasing the amount of data passed on each request. Meanwhile, HTML applications are page-based, and demand complete page refreshes when information is submitted to a server, as well as when a client navigates from one page to another. Each page load uses up network bandwidth and server resources. RIAs behave like desktop applications, instead of series of pages. Flash Player manages the client interface as a single, uninterrupted flow and does not require that a page load from the server when the client moves from one section of the application to another. With an RIA, the browser loads the SWF file only once; all subsequent requests are smaller data-driven requests using either XML or AMF.

The other, less preferred option for deploying Flex is to create a dedicated Flex environment. In this model, the existing infrastructure provides data connectivity through web services or XML feeds. This model adds a significant amount of cost and administrative overhead, because it requires that you install and configure new hardware and the J2EE application servers where you plan to deploy Flex.

A note about JVM and J2EE server tuning: Flex supports a wide range of J2EE application servers, so it is difficult to outline specific actions for server performance tuning. The best place to start is with your application server vendor's website; look for tuning resources there. The compiler and data services in Flex can use a large amount of short-term objects under heavy load. In some case that can cause delays due to heavy use of garbage collection. If your JVM vendor is independent of your J2EE application server vendor, you may want to review information on tuning the JVM and garbage collection.

Load Testing Your Flex Application

The performance of your Flex application is highly dependent on your existing applications and infrastructure. For example, the Flex application server and data services may be able to sustain several hundred requests per second. However, if your web service or remote object can only sustain 35 requests per second, your web service becomes the bottleneck and hinders scalability. This is why you must test each independent tier of your application on its own. The following sections explore how you can load test your Flex application. (See also the "Performance Tuning and Profiling Your Own Flex Application" section in the article section on, "Flex Application Performance: Tips and Techniques to Improving Client Application Performance.")

Load-Testing Tools

First, load test each of your data services independent of the Flex proxies. If you have a web service or an XML feed from a servlet, use a load-test tool to gauge the maximum number of requests per second that it can sustain. Second, run the same test through the Flex proxy, because all traffic is sent from the Flash client to the proxy, and the proxy, in turn, sends the request to your web service or HTTP feed.

You can choose among several load-testing tools, ranging from free tools to tools that cost tens of thousands of dollars. During the development and staging phases, a free tool works just fine for load testing individual services (such as web services, remote objects, or XML feeds). Some free load-testing tools are:

- Microsoft Web Application Stress Tool
  www.microsoft.com/downloads/details.aspx?familyid=e2c0585a-062a-439e-a67d-75a89aa36495&DISPLAYLANG=en
- OpenSTA
  www.opensta.org
- Apache JMeter
  jakarta.apache.org/jmeter/index.html
- Apache Bench
  perl.apache.org/docs/1.0/guide/performance.html

However, to load test a finished application and randomize request data, use one of the following commercial load-testing tools:

- SegueSilkPerformer
  www.segue.com/html/s_solutions/s_performer/s_performer.htm
- Mercury LoadRunner or LoadTest
  www.mercuryinteractive.com/products/
- Emperix e-Test Suite
  www.empirix.com/Empirix/Web+Test+Monitoring/Web+Test+Monitor+Overview.html

Load Testing Sample Scenarios

During development, test each of the services that your Flex application consumes. This can be as simple as using one of the free tools listed previously to record interaction between the Flash client and the server and playing back the recorded request/response under a multiuser scenario. This section describes several scenarios using different approaches and data services.

Scenario 1

A common approach is to use a JSP or servlet that exposes XML to the Flex application and is invoked using the `<mx:HTTPService>` tag in Flex. After building a portion of the application that consumes this XML feed, load test to ensure the back-end service can sustain the production load. Since most development systems do not provide production horsepower, use a staging environment that mirrors the production system. Using the staging system and a testing tool, you can record the traffic that occurs between the browser and server. The testing tool records the traffic and can play it back using multiple threads (or users) to simulate a production load. It is important that at this stage that you only test the data services, not the loading of the SWF file or the other services that Flex provides. Delete the other requests from the recorded script. The following snippet shows requests from a sample script recorded in Microsoft WAS:

```
 GET /sharedapps/xmltest/WeeklyReport.mxml
 GET /flex-internal?action=js
 GET /sharedapps/xmltest/WeeklyReport.mxml.swf
 GET /flex-internal/history/history.html
POST /flex/flashproxy
(post data
   transport=http%2Dget&target=http%3A%2F%2F192%2E169%2E1%2E24%3A8
   100%2Fsharedapps%2Fxmltest%2Fxml%2Ejsp)
 GET /flex-internal?action=swf
```

Delete all requests except **POST /flex/flashproxy** to focus on load testing only the XML feed. Later, before you go to production, load test all of the requests to the server. After running your first load test, you set a baseline. Afterwards, by modifying code or tuning the server, you can improve the numbers until you meet your target. Use this testing approach for all server-side resources that Flex consumes.

Scenario 2

You may run across cases where it seems that the Flex proxy services, network, or some unknown factor is a bottleneck. To troubleshoot this, first rule out the end service. To find the baseline maximum, point your HTTPService request at a static document. To do this, request the JSP or servlet feeding the XML file, save it to a static XML file, and then put the XML file in your webroot. Change the URL of your HTTPService to point to this static document and run your load test again. This provides you not only with a theoretical maximum, but also targets other bottlenecks.

The steps to load test a web service using the `<mx:WebService>` tag in Flex are almost identical. When viewing your traffic results, you may notice that when the SWF file first loads, it makes a request to the WSDL to load information about the web service. The SWF file makes the request once for each client when the Flex application loads. Requests to methods in the web service follow the WSDL request. If you load test a web service, you may want to load test the WSDL invocation separately from the method calls, because it is not requested nearly as often as the method calls to the service. Some load-test tools let you set up a ratio using different scripts to get the model as close to a real-world scenario as possible.

Scenario 3

Many of the commercial tools also provide powerful scripting languages to customize your scripts and randomize data passed into the request. For example, if you were testing the log-on section of your site, you would not want the same user to log on over and over again in your script. You would instead randomize a set of users in the script so that you mimic different users logging in throughout the load test.

There are some challenges to load testing using RemoteObject with AMF encoding. SOAP uses XML in the request to the server, so it is fairly simple to write a script so that a portion of the XML data is random. AMF is a binary protocol and the request data to the Flash gateway is binary, so not only is it difficult to read what is passed, but it is also difficult to script the requests to include random data. Some commercial load-testing vendors, such as Segue, include AMF as a native protocol to their testing suites to simplify this process.

Near the end of your development cycle, you should run several full-fledged load tests. The commercial tools mentioned previously work best in this situation. Record several common paths that users will take through the application. After recording the different paths, you might need to modify each script, to randomize the data passed to the server. You can then run the scripts against the server while monitoring several different areas on the back end. Monitor the following areas on both your application and database servers: CPU, JVM heap usage, network, and any tools that the application server provides for monitoring.

After completing the tests, analyze the results and try to identify the requests that take the longest to process. The goal is to find bottlenecks and optimize any areas in your application or server. After optimizing your applications, run several long-term MTBF (mean time between failure) tests to ensure that after several hours in production, no preexisting or new issues cause downtime. Although it is impossible to simulate exactly what occurs in a production environment, you can come close. Following the steps and tips outlined previously, and you can be confident that your application will perform well in production with very little downtime.

Feedback and Support

We have made every effort to ensure the accuracy of this article and all code included. Feedback regarding this article and all Flex performance issues is always appreciated.

About the Authors

Brandon Purcell started at Macromedia/Allaire four years ago as a support engineer working with ColdFusion and JRun. He has worked with the Professional Services Group helping customers with their architecture planning, code reviews, customized training, load testing, and performance tuning. He has also worked on special projects including the clustering, load testing and deployment of the new macromedia.com website. During his tenure with Macromedia, he has supported ColdFusion, JRun, Flash Remoting, and has written white papers and articles on clustering and high availability with ColdFusion and JRun. Currently he is working as an escalation engineer for the Flex Server Support Organization. You can visit him at his website.

Deepa Subramanian is a quality assurance engineer for the Flex team. Straight out of UC Berkeley (Go Bears!), with an undergraduate degree in computer science, she has been at Macromedia for just over one year and is very excited to be working on all things Flex related.