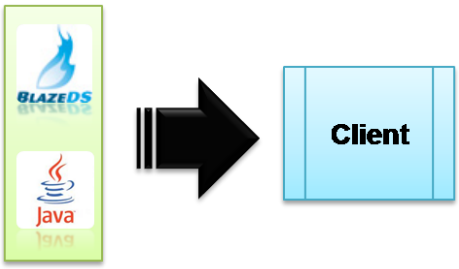
Server Push (data push) in Flex using Blaze DS and Java.



Knowledge level: Intermediate – Advanced.

Server push generally means that the server pushes the content to the client, instead of client requesting for the content. Yes there are many ways of achieving this, more about it here:

http://en.wikipedia.org/wiki/Push_technology

Blaze Data Services are open source messaging and remoting technology provided by Adobe, Blaze DS may not be as powerful as Live Cycle Data Services of Adobe, still adequate for most of the enterprise requirements.

If you were looking out for something which can provide many more powerful features under open source umbrella to replace Live Cycle data services take a look at Granite Data Services.

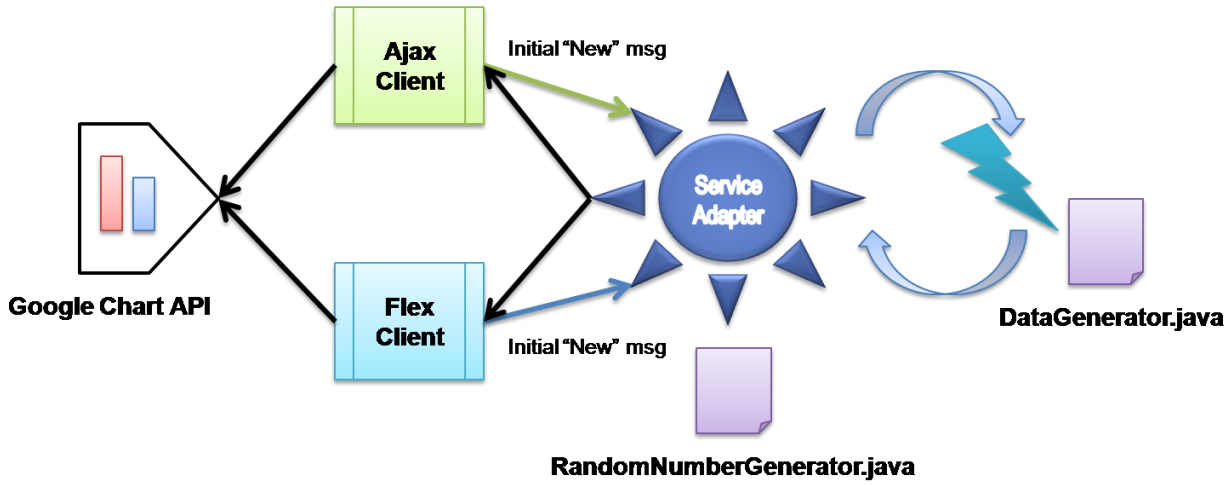
As per the Adobe's definition of Blaze DS "push data in real-time" was something which made to take a look at this interesting technology, the basic example of achieving this has been included in the samples provided by the Blaze DS Turnkey Download.

Before going any further, I assume you have basic knowledge of Blaze Data Services and familiar with invoking remote methods. (If not please start here http://www.adobe.com/devnet/livecycle/articles/blazeds_gettingstarted.html).

Let me dive in to the example demonstrating the server push via Blaze DS, I have tried my best to keep this example as simple as possible, this example uses a Random Number generator, which pushes six randomly generated numbers to the client and client dynamically constructs a Bar chart using Google Chart API and display.

In order to achieve a completely usable server push, we combine two different aspects of Blaze DS, they are Message Broker and Service Adapter. Message Broker is responsible for routing the generated messages to the service and Custom Service Adapter acts as a service, which registers all the clients subscribed and publish the changes. we can also restrict the number of client via this, hence we are gaining complete control over the application behavior.

Let's get started with implementation and configurations required for this application, the complete setup of the application looks as shown below:



The Ajax client shown in the above setup is out of scope for this article, flex clients are neatly covered in the developers guide. (Flash CS3 client may be possible too!).

The Blaze DS uses four main configuration files namely:

- messaging-config.xml
- proxy-config.xml
- remoting-config.xml
- services-config.xml

We have to do some modifications to the services-config and messaging-config xmls.

As we will be using a streaming AMF channel, we need to define this in **service-config.xml** as shown below.

```
<channel-definition id="my-streaming-amf"
class="mx.messaging.channels.StreamingAMFChannel">
            <endpoint
url="http://{server.name}:{server.port}/{context.root}/messagebroker/streamin
gamf" class="flex.messaging.endpoints.StreamingAMFEndpoint"/>
             <properties>
                <idle-timeout-minutes>0</idle-timeout-minutes>
                <max-streaming-clients>10</max-streaming-clients>
                <server-to-client-heartbeat-millis>5000</server-to-client-
heartbeat-millis>
                <user-agent-settings>
                    <user-agent match-on="MSIE" kickstart-bytes="2048" max-
streaming-connections-per-session="3"/>
                    <user-agent match-on="Firefox" kickstart-bytes="2048"
max-streaming-connections-per-session="3"/>
                </user-agent-settings>
            </properties>
</channel-definition>
```

StreaminfAMFEndpoint is provided by Blaze DS and as we can see from the above channel definition we are restricting the number of clients to 10 and Number of user agent sessions (in same browser) to 3. More details are in the developer guide.

Once we have defined the channel, we need to define a destination and an adapter in the **messaging-config.xml** as shown below.

```
<adapters>

      <adapter-definition id="RandomDataPushAdapter"
class="com.RandomNumberGenerator"/>

</adapters>
```

Please note in the above definition com.RandomNumberGenerator is a custom service adapter which we will develop next.

Now we need to add a new destination as shown,

```
<destination id="RandomDataPush">
        <channels>
            <channel ref="my-streaming-amf"/>
        </channels>
        <adapter ref="RandomDataPushAdapter"/>
</destination>
```

As we can see from the above definition we are configuring this destination to use my-streaming-amf channel, which we have defined in the service-config.xml.

That's about the configuration, now let's see the implementation, but before that let see why we need a Service Adapter in first place.

We could have achieved a data push just by using the Message Broker as described in the data push example provided by Blaze DS but the this approach will work only if the push delay is very minimal.

Consider a scenario in which we need to push the data to all the subscribed clients after 5 minutes, the above approach definitely works provided all the clients start at the same time, else the client which starts after the initial data push has to wait till the next data push happens to receive the data (say if the client started 0:01 minute late after the server pushing the data, client has to wait for 4:59 minutes to receive the data)

We can overcome this problem by introducing a custom service adapter whose "invoke" method will be invoked by the client as soon as it registers and we can use the acknowledgement to send the current state (previously pushed data) of the server. Hence this is the best approach to make sure that all the clients are in same state after start and will refresh simultaneously when the next data push occurs.

Here is implementation of the custom service adapter (Random Number Generator) and also Data Generator, which is a thread generating Random number this may be replaced by a database trigger or web service call or whatever data provider depending on the requirement.

------------------------RandomNumberGenerator.Java--------------------------

```
package com;

import java.util.Random;

import flex.messaging.MessageBroker;
import flex.messaging.io.ArrayCollection;
import flex.messaging.messages.AsyncMessage;
import flex.messaging.messages.Message;
import flex.messaging.services.MessageService;
import flex.messaging.services.ServiceAdapter;
import flex.messaging.util.UUIDUtils;

/**
 * Custom Service Adapter which does the following
 *
 * 1) Generating an initial set of random numbers.
 * 2) Process the incoming messages.
 * 3) Responsible for pushing the message to all the clients.
 * 4) Responsible for sending the current state as part of ack.
 *
 * @author Siva Prasanna Kumar .P
 */
public class RandomNumberGenerator extends ServiceAdapter {

	/**
	 * A thread which will be initialized and started on start of the
	 * service adapter.
	 */
	private static DataGenerator thread;

	/**
	 * A collection of Random Generated numbers.
	 */
	private static ArrayCollection randomNumbers = new ArrayCollection();

	public RandomNumberGenerator() {

		Random random = new Random();

		for (int i = 0; i < 6; i++) {
			randomNumbers.add(new Integer(random.nextInt(100)));
		}

	}

	public void start() {

		if (thread == null) {
			thread = new DataGenerator();
			thread.start();
		}
	}

	public void stop() {

		thread.running = false;
```

```
		thread = null;
	}

	/**
	 * This is static Random Number generator class which acts as a source
	 * or Data provider for the data push application.
	 *
	 * @author Siva Prasanna Kumar .P
	 *
	 */
	public static class DataGenerator extends Thread {

		public boolean running = true;

		public void run() {

			MessageBroker msgBroker =
				MessageBroker.getMessageBroker(null);

			String clientID = UUIDUtils.createUUID();

			Random random = new Random();

			while (running) {

				randomNumbers.clear();
				{
					for (int i = 0; i < 6; i++) {
						randomNumbers.add(new
					Integer(random.nextInt(100)));
					}
				}

				// creating a new async message and setting the
				// random generated number as content.
				AsyncMessage msg = new AsyncMessage();

				// setting the destination for this message.
				msg.setDestination("RandomDataPush");

				msg.setClientId(clientID);

				msg.setMessageId(UUIDUtils.createUUID());

				msg.setBody(randomNumbers);

				msgBroker.routeMessageToService(msg, null);

				try {
					Thread.sleep(60000); // 1 minute delay.
				} catch (InterruptedException e) {
					e.printStackTrace();
				}

			}
		}
	}
```

```
        // @Override (uncomment this if you are using java 5 and above.)
        public Object invoke(Message message) {

                if (message.getBody().equals("New")) {
                        // this send the current state to the client,
                        // (random numbers previously generated as part of the
                        // acknowledgment).
                        return randomNumbers;
                } else {
                        AsyncMessage newMessage = (AsyncMessage) message;

                        newMessage.setBody(randomNumbers);

                        MessageService msgService = (MessageService)
                        getDestination().getService();

                        // This is most important call which pushes,
                        // the recivied msg to all the clients.
                        msgService.pushMessageToClients(newMessage, false);
                }
                return null;
        }

}
```

RandomNumberGenerator generates initial six random numbers between 0-100 and the invoke method checks if the incoming message body's content is "New" this check defines if the client in a new client or a existing client (this is a simple custom contract b/w the clients and server, you will understand this better when you see the client side). So every time a New client starts, it send or produces a message whose content is "New" and server can acknowledge by sending the current state of the application.

Once the server is started, the service starts pushing the messages (irrespective of the clients, we can handle and customize this behavior also) after a specified time delay (60000 milliseconds). The clients subscribed for this message, automatically updated them self based on the client's message processing logic.

Compiling the RandomNumberGenerator.java file generates two class files namely

- RandomNumberGenerator.class
- RandomNumberGenerator$DataGenerator.class

Create a folder by name "com" under "blazeds\WEB-INF\classes\" and Copy these class files to it.

That's about the server side, now once you start the tomcat or the server hosting the blaze DS war file, the service is automatically started and running.

Now let us see how to create client which can register and receive these pushed messages.

In order to configure flex builder to use Blaze DS, please check this getting started with Blaze DS article. Once you create a new Flex Project which use **Application server type : J2EE** and with Life Cycle Data Services (even though we are using Blaze DS, flex builder shows LCDS only.)

Create a new mxml file say GoogleChart.mxml and copy the below content into it.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" layout="absolute"
creationComplete="init()">
<mx:Script>
      <![CDATA[
                  import mx.messaging.channels.AMFChannel;
                  import mx.messaging.ChannelSet;
                  import mx.messaging.events.MessageFaultEvent;
                  import mx.messaging.events.MessageAckEvent;
                  import mx.messaging.messages.AsyncMessage;
                  import mx.controls.Alert;
                  import mx.rpc.remoting.mxml.RemoteObject;
                  import mx.rpc.events.ResultEvent;

                  import mx.messaging.messages.IMessage;

                  import mx.collections.ArrayCollection;

                  [Bindable]
                  public var randomNumbers:ArrayCollection;

                  private function messageHandler(message:IMessage):void
                  {
                         randomNumbers = message.body as ArrayCollection;
                         chart.source =generateURL(randomNumbers.toArray());
                  }

                  public function generateURL(array:Array):String
                  {
                        //this is a url for generating google map from the
                        //received values.
                        var url:String =
                        "http://chart.apis.google.com/chart?cht=bvs&chd=t:"+
                        array[0]+","+array[1]+","+array[2]+","+array[3]+","+a
                        rray[4]+","+array[5]+"&chs=210x300&chtt=Generated%20R
                        andom%20Numbers&chco=ff9900|00ccff&chxt=x,y&chxl=0+ar
                        ray[0]+"|"+array[1]+"|"+array[2]+"|"+array[3]+"|"+arr
                        ay[4]+"|"+array[5]+"|1:||20||60||100";

                        return url;
                  }
                  private function init():void
                  {
                        var msg:AsyncMessage = new AsyncMessage();

                        //creating new msg with "New" to get current state.
                        msg.body = "New";
                        producer.send(msg);
                        consumer.subscribe();
                  }

                  private function handleFault(event:MessageFaultEvent):void
                  {
```

```
                        Alert.show(event.faultString);
                }

                private function ack(event:MessageAckEvent):void
                {
                        chart.setVisible(true);
                        randomNumbers =  event.message.body as
ArrayCollection;
                        chart.source =generateURL(randomNumbers.toArray());
                }
            ]]>
</mx:Script>
<mx:Producer id="producer" destination="RandomDataPush"
acknowledge="ack(event)"/>
<mx:Consumer id="consumer" destination="RandomDataPush"
message="messageHandler(event.message)" />

<mx:VBox width="100%" height="100%" horizontalAlign="center"
verticalAlign="middle">
      <mx:Image id="chart"  />
</mx:VBox>
</mx:Application>
```

As you can see from the above mxml file, we have a combination of a Producer and Consumer in the flex client. Producer is only needed in order to overcome the problem of getting the current state initially.

The application starts by invoking the init() method, where the producer creates a new message with body content as “New” and the client subscribes to the remote destination. The destination (`RandomDataPush`) destination is same for both producer and consumer, producer only sends the message to get the initial state and the server will send the current state as part of the acknowledgement. Received ack is processed by the ack() function, which reads the values send and generates a URL accordingly and sets the image dynamically.

Consumers which have subscribed will be notified by the server as and when there is an update on the subscribed topic, hence the consumer refreshes the view on receiving the new data from the server. (In order to learn more about using Google api, please look at my previous blog post).

Hence by combining two different aspects of Blaze DS, we have achieved a simple server push which works really well and efficient enough for real time streaming applications.

Author: Siva Prasanna Kumar.P
Blog: http://soa2world.blogspot.com/