White Paper

Flex Data Services Performance Tests

Version 1.0 – January 19th 2007

Sven Ramuschkat

Dirk Eismann

http://www.richinternet.de

http://www.richinternet.de/blog

http://www.flexperten.de

http://www.flexforum.de

Herrlich & Ramuschkat GmbH
Vahrenwalder Strasse 156
D-30165 Hannover, Germany
Phone +49 (0)511 - 59 0 95 - 0 Fax – 590

1. Table Of Contents

2. Introduction

On November 30th 2006, Adobe released the **Flex Stress Testing Framework** on Adobe Labs [1]

The Flex Stress Testing Framework – as the name implies – allows you to test and measure the performance of a Flex Data Services (FDS) powered application. The framework provides an API and some tools to easily setup new test scenarios.

The Flex Data Services (FDS) product is available in three Editions:

- The free **Express Edition**, which is limited to 1 physical CPU (Dual Core counts as 1 CPU). No restrictions concerning maximum connections / users. Cannot be clustered.
- The **Department Edition**, needs one license per physical CPU. Can be clustered but has a maximum of 100 concurrent connections / users in total (i.e. even with 3 CPUs only 100 concurrent users can be managed)
- The **Enterprise Edition,** also needs one license per CPU, no further limitations.

In order to give a good estimate about how many licenses are needed for a given customer project it is essential to know how the Flex Data Services will perform and scale.

On the same topic, Adobe released a white paper called „Flex Data Services 2: Capacity planning“ [2]

Adobe's white paper first gives an overview on the core Flex Data Services architecture and then discusses how to tune certain aspects of a FDS powered application. Finally, the results of some load and stress tests are shown.

After studying this whitepaper you probably still do not know how a FDS powered applications that use Remote Objects, Messaging and the Data Management feature will actually perform and scale.

Maybe you already have a good idea about the kind of data your application will produce and how many concurrent connections are needed and you only need some real test results. Maybe you just want to check out the total amount of concurrent connections FDS can handle. Maybe you are just curious.

We have created several test setups for the Flex Stress Testing Framework that can be used as guide for your own tests. The tests are explained in more detail in the next sections.

Of course, not every test here may be applicable for your project and your requirements – or to quote from Adobe's white paper: "*No two applications are exactly alike, and there are many factors that may impact performance"* but they should give you a good idea about FDS's capabilities.

3. Test Environment

The following configuration was used on the server side:

1 x IBM eServer 336, 3.2 GHz XEON (1 CPU, Single Core), 2 GB RAM, 140 GB RAID 1 HDD

- Windows Server 2003, Standard Edition R2
- Apache TomCat 5.5 (using Suns JDK 1.5 JRE)
- Java Open Transaction Manager (JOTM)
- Flex Data Services 2 (Express Edition)
- HSQLDB (in memory database)
- jTDS 1.2 SQLServer JDBC Driver

---

[1] *http://labs.adobe.com/wiki/index.php/Flex_Stress_Testing_Framework*

[2] *http://www.adobe.com/products/flex/whitepapers/pdfs/flex2wp_fdscapacityplanning.pdf*

In addition, to use the Flex Stress Testing Framework you need two different client roles: one or more clients that make requests against FDS using a remote controlled Flex application (i.e. the application that implements a test) and a client that runs the main controller console (i.e. the machine that starts / stops the test run and also measures the performance)

Each client that acts as a test runner instantiates one or more browsers and in turn loads the test application into the browser – so in effect, with 4 clients configured to start 10 browsers each 40 concurrent connections to FDS can be created.

For the test runner we used

4 x Acer Notebooks, Intel Dual Core 1.6 GHz, 1.5 GB RAM, 60 GB HDD

- Windows XP Professional, SP2
- Internet Explorer 7 with Flash 9 Player
- Suns 1.5 JRE

Every test runner hosts the *BrowserServer* application that comes with the Stress Testing Framework and can be controlled by the computer hosting the TestAdmin-Console:

1 x IBM Thinkpad T40, Intel Pentium M 1.6 GHz, 2 GB RAM

- Windows XP Professional, SP2
- Internet Explorer 7 with Flash 9 Player

This computer also hosts a data producing application that is used in the Data Management test below.

4. Performance Test #1: Remote Object Service

The Remote Object test uses a test application based on the Flex Stress Testing Framework. The test application uses the RemoteObject class to connect an FDS destination. The destination uses the default AMF3-over-HTTP channel of a default installation and the JavaAdapter to invoke the business logic. The business logic is exposed as a Java class and connects to a MS SQL Server 2000 database containing 20.000 records of census data. The database is running on a remote machine in the same local network. This test was done with a result set size of 50 and 100 rows.

4.1. MXML-Test File

This test calls a method on the remote Java class. After a result or fault event has been received, the application invokes the same remote method call again. For every result and fault a counter gets increased. The throughput is calculated based on these two values after the test is finished.

```
<mx:Application xmlns:mx=http://www.adobe.com/2006/mxml creationComplete="run()">
<mx:Script>
    <![CDATA[
        private var count:int = 0;
        private var failureCount:int = 0;
        //every stress test will need to create an instance of Participant
        private var p:Participant;
        public function run():void {
          // the framework passes in the unique name or id
          // of the Participant at runtime
          var id:String = Application.application.parameters.id;
          // property specifying whether or not this test should be
          // managed by the LoadRunner application.
          var monitored:Boolean = (id != null);
          var name:String = id;
          // create a Participant instance passing in the unique name
          // and whether or not the test is managed.
          p = new Participant(name, monitored);
          // setup event listeners for starting and stopping the test
          Participant.eventdispatcher.addEventListener("startRequest",startTest);
          Participant.eventdispatcher.addEventListener("timeUp", stopTest);
        }
        // this is called by the test framework. From here you call
        // startTest on the participant
        private function startTest(e:Object):void {
          p.startTest(startRemoteObjectTest, e.duration);
        }
        private function startRemoteObjectTest():void {
          callRemoteMethod();
        }
        private function callRemoteMethod():void {
```

```
          // 50 = number of rows to fetch from the database
          service.getElements(0, 50);
        }

        //stop test is called by the framework
        private function stopTest(e:Object):void {
          if (p.testCount == 0)
          {
            // when the framework calls stopTest we set the stop time
            p.testStopAt = getTimer();
            // update the testCount property with the number
            // of successful requests.
            p.testCount = count;
            // update the failureCount property with the number
            // of failed requests.
            p.failureCount = failureCount;
          }
        }

        private function onResult():void {
          // increase success counter and call remote method again
          count++;
          callRemoteMethod();
        }

        private function onFault():void {
          // increase failure counter and call remote method again
          failureCount++;
          callRemoteMethod();
        }
    ]]>
    </mx:Script>
    <mx:RemoteObject
       id="service" destination="census"
       result="onResult()"
       fault="onFault()"
    />
</mx:Application>
```

4.2. Server Side Business Logic

The server side business logic is written in Java. We used the Java classes from Adobe's Census application that comes with the FDS installation. The server side business Logic uses the jTDS 1.1

JDBC Driver to connect to a MS SQL Server 2000 where the census data is stored. The census table contains 20.000 rows with the following data structure:

AGE	varchar	3
CLASSOFWORKER	varchar	255
EDUCATION	varchar	255
MARITALSTATUS	varchar	255
RACE	varchar	255
SEX	varchar	255
ID	int	4

4.3. XML Configuration File: remoting-config.xml

There is nothing special about this configuration. It uses the *my-amf* AMF channel.

```
<destination id="census">
        <channels>
            <channel ref="my-amf"/>
        </channels>
        <adapter ref="java-object"/>
        <properties>
            <source>samples.census.CensusService</source>
        </properties>
</destination>
```

4.4. Test Results

Several test runs were made using an increasing amount of browser instances. Run time per test was 60 seconds. We measured the total number of successful requests and used the Windows Task Manager application to get an idea of the CPU load. The best message throughput is indicated in **orange**.

RemoteObject Test, Census DB, Result set size: 100 rows					
# of Browser-Servers	# of Browser instances	Total # of Browsers	Total # of result sets	# of result sets/sec.	% CPU load
4	1	4	3900	65,00	30-40
4	5	20	5878	97,96	40-80
4	10	40	12701	211,68	80-90
4	15	60	12240	204,00	80-90
4	20	80	8541	142,35	80-90

RemoteObject Test, Census DB, Result set size: 50 rows					
# of Browser-Servers	# of Browser instances	Total # of Browsers	Total # of result sets	# of result sets/sec.	% CPU load
4	1	4	4583	76,38	30-40
4	5	20	18969	316,15	60-80
4	10	40	16468	274,46	70-95
4	15	60	14986	249,76	70-95
4	20	80	13927	232,11	70-95

5. Performance Test #2: Messaging Service

The Messaging Services test uses a test application based on the Flex Stress Testing Framework. The test application subscribes as a consumer to a FDS messaging destination and also acts as a message publisher. This setup uses the FDS ActionScriptAdapter (i.e. no additional server side infrastructure like JMS).

5.1. MXML-Test File

The test application creates uses a Producer and Consumer to send messages to FDS and to receive broadcasted messages from FDS. After a message is received from the network the application sends another message. Every Message correctly sent and received increases a counter.

```
<mx:Application xmlns:mx=http://www.adobe.com/2006/mxml creationComplete="run()">
    <mx:Script>
    <![CDATA[
        import mx.messaging.messages.*;
        import mx.messaging.events.*;
        import mx.utils.ObjectUtil;

        private var count:int = 0;
        private var failureCount:int = 0;
        private var p:Participant;

        public function run():void {
          var id:String = Application.application.parameters.id;
          var monitored:Boolean = (id != null);
          var name:String = id;
          p = new Participant(name, monitored);
          Participant.eventdispatcher.addEventListener("startRequest", startTest);
          Participant.eventdispatcher.addEventListener("timeUp", stopTest);
        }

        private function startTest(e:Object):void {
          consumer.subscribe();
          p.startTest(startMessagingTest, e.duration);
        }
```

```
        private function startMessagingTest():void {
          sendMessage();
        }

        private function stopTest(e:Object):void {
          if (p.testCount == 0) {
            p.testStopAt = getTimer();
            p.testCount = count;
            p.failureCount = failureCount;
          }
        }

        private function sendMessage():void {
          var message: AsyncMessage = new AsyncMessage();
          message.body = "Automessage";
          producer.send(message);
          count++;
        }

        private function onResult(event:MessageEvent):void {
          log.text += event.message.body + "\n";
          lograw.text += "Message: " + ObjectUtil.toString(event.message) + "\n";
          count++;
          sendMessage();
        }

        private function onFault():void {
          log.text += "Error" + "\n";
          lograw.text += "Message: " + "Fehler" + "\n";
          failureCount++;
          sendMessage();
        }
    ]]>
    </mx:Script>

    <mx:Producer id="producer" destination="myChat"/>
    <mx:Consumer id="consumer" destination="myChat"
        message="onResult(event)"
        fault="onFault()"
    />
    <mx:TextArea id="log" width="100%" height="100%" editable="false"/>
    <mx:TextArea id="lograw" width="100%" height="100%" editable="false"/>
</mx:Application>
```

5.2. XML Configuration File: messaging-config.xml

```
<destination id="myChat">
        <adapter ref="actionscript" />
        <channels>
            <channel ref="my-rtmp"/>
        </channels>
        <properties>
            <network>
                <session-timeout>20</session-timeout>
                <throttle-inbound policy="ERROR" max-frequency="0"/>
                <throttle-outbound policy="REPLACE" max-frequency="0"/>
            </network>
            <server>
                <max-cache-size>1000</max-cache-size>
                <message-time-to-live>0</message-time-to-live>
                <durable>true</durable>
                <durable-store-manager>
                    flex.messaging.durability.FileStoreManager
                </durable-store-manager>
                <max-file-size>200K</max-file-size>
                <batch-write-size>10</batch-write-size>
            </server>
        </properties>
    </destination>
```

5.3. Test Results

Several test runs were made using an increasing amount of browser instances. Run time per test was 60 seconds. We measured the total number of successful sent and received messages and used the Windows Task Manager application to get an idea of the CPU load. The best message throughput is indicated in orange.

Messaging Test, every client sends and receives a message					
# of Browser-Servers	# of Browser instances	Total # of Browsers	Total # of messages	# of messages/sec.	% CPU load
4	1	4	12556	209,26	15-20
4	5	20	35930	598,83	95-100
4	10	40	55808	930,13	95-100
4	15	60	63987	1066,45	95-100

6. Performance Test #3: Data Management Service

The Data Management Service Test uses two applications. The first application is based on the Flex Stress Testing Framework and acts as data consumer. The consuming application connects to a Data Management destination and fills a client side ArrayCollection.

The second application is a simple Flex application (not a test run by the framework) that acts as a data manipulating application. The application connects to the same destination and automatically invokes create, update and delete on the managed ArrayCollection.

6.1. MXML-Test File: Data Service Producer Application

The data manipulating application runs on a separate PC and fills an ArrayCollection by using a DataService instance. It then automatically updates, creates and deletes records in the ArrayCollection and calls commit() on the DataService. It does so in a sequential manner and restarts the update / create / delete process after the last delete operation returns successfully.

```
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    layout="vertical"
    creationComplete="init()">
    <mx:Script>
    <![CDATA[
       import mx.collections.ItemResponder;
       import mx.rpc.AsyncToken;

       [Bindable]
       private var faults:uint;
       [Bindable]
       private var results:uint;
       private var timer:Number;
       private var isRunning:Boolean;

       private function init():void {
         if (parameters.autoRun && parameters.autoRun.toLowerCase() == "true"){
           // is an URL parameter autoRun=true then start automatically
           // otherwise the start Button has to be clicked
           start();
         }
       }

       private function start():void {
         // start the data manipulation process
         relase();
         timer = getTimer();
         isRunning = true;
         faults = 0;
         results = 0;
```

```
        metrics.text = "";
        items = new ArrayCollection();
        addResponder(ds.fill(items), "fill");
      }

      private function stop():void {
        // stop the data manipulation process
        isRunning = false;
        timer = getTimer() - timer;
        metrics.text = "Total: " + uint(results + faults).toString() + "
        messages received in " + timer + " ms";
      }

      private function relase():void {
        try {
          ds.releaseCollection(items, true);
        } catch (e:Error) {}
      }

      private function addResponder(token:AsyncToken, name:String):void {
        token.addResponder(new ItemResponder(result, fault, name));
      }

      private function createItem():Object {
        // create a new empty record
        return getItem();
      }

      private function updateItem(o:Object):Object {
        // updates an item and returns the new version
        var n:Object = getItem(o.PRODUCT_ID);
        var p:String;
        for (p in o) {o[p] = n[p]; }
        return o;
      }

      private function getItem(id:int = -1):Object {
        // creates a new item
        var o:Object = {};
        o.DESCRIPTION = getText(100);
        o.NAME = getText(12);
        o.PRICE = (Math.random() * 1000);
        o.QTY_IN_STOCK = Math.floor(Math.random() * 500);
```

```
          if (id > -1) {
            o.CATEGORY = "NEW123";
            o.PRODUCT_ID = id;
          }
          return o;
        }

        private function getText(chars:uint):String {
          // returns a random string of a given length
          var str:String = "";
          var i:int;
          for (i = 0; i < chars; i++) {
            str += String.fromCharCode(65 + Math.ceil(Math.random() * 25));
          }
          return str;
      }

      private function result(data:Object, token:String):void {
        // process an incoming result. Next action depends on the token string
        var item:Object
        results++;
        if (isRunning) {
          if (token == "fill" || token == "delete") {
            if (items.length > 0) {
              item = items.getItemAt(Math.ceil(Math.random() * items.length-1));
              updateItem(item);
              addResponder(ds.commit(), "update");
             }
           } else if (token == "update") {
             items.addItem(createItem());
             addResponder(ds.commit(), "create");
           } else if (token == "create") {
              if (items.length > 0) {
               item = items.getItemAt(Math.ceil(Math.random() * items.length-1));
               ds.deleteItem(item);
               addResponder(ds.commit(), "delete");
              }
          }
        } else {
          release();
        }
      }
```

```
     private function fault(data:Object, token:String):void {
       faults++;
     }
   ]]>
 </mx:Script>

 <mx:DataService id="ds" destination="jdbc-test" autoCommit="false"/>
 <mx:ArrayCollection id="items"/>
 <mx:HBox>
   <mx:Button label="Start" click="start()"/>
    <mx:Button label="Stop" click="stop()"/>
    <mx:Label text="Results: {results}"/>
    <mx:Label text="Faults: {faults}"/>
    <mx:Label id="metrics" />
  </mx:HBox>
  <mx:DataGrid id="dg" width="100%" height="100%" dataProvider="{items}"/>
</mx:Application>
```

6.2. MXML-Test File: Data Service Consumer Application

The data consuming application is based on the Flex Stress Testing Framework. The test application fills an ArrayCollection by using a DataService instance. For maximum performance, this application does not use any user interface.

Every message received from the FDS destination increases a counter (depending on the type of message, either result or fault)

```
<mx:Application xmlns:mx=http://www.adobe.com/2006/mxml creationComplete="run()">
  <mx:Script>
    <![CDATA[
     import mx.data.messages.DataMessage;
     import mx.messaging.events.MessageEvent;
     import mx.messaging.messages.ErrorMessage;

     private var count:int = 0;
     private var failureCount:int = 0;
     private var p:Participant;

     public function run():void {
       var id:String = Application.application.parameters.id;
       var monitored:Boolean = (id != null);
       var name:String = id;
       p = new Participant(name, monitored);
```

```
        Participant.eventdispatcher.addEventListener("startRequest",startTest);
        Participant.eventdispatcher.addEventListener("timeUp", stopTest);
      }

      private function startTest(e:Object):void {
        p.startTest(startDataServiceTest, e.duration);
      }

      private function startDataServiceTest():void {
        items = new ArrayCollection();
        ds.fill(items);
      }

      private function stopTest(e:Object):void {
        if (p.testCount == 0) {
          ds.releaseCollection(items, true);
          p.testStopAt = getTimer();
          p.testCount = count;
          p.failureCount = failureCount;
        }
      }

      private function message(event:MessageEvent):void {
        if (event.message is ErrorMessage) {
          failureCount++;
        } else if (event.message is DataMessage) {
          count++;
        }
      }
    ]]>
 </mx:Script>

 <mx:DataService
     id="ds"
     destination="jdbc-test"
     autoCommit="false"
     message="message(event)"
  />
  <mx:ArrayCollection id="items"/>
</mx:Application>
```

6.3. Server Side Business Logic

For this test we used the FDS JDBCAssembler by Christophe Coenraets [3]. By using the JDBCAssembler you can easily connect FDS to JDBC data sources and do simpile CRUD operations.

The JDBCAssembler connects to an HSQLDB database on the same server. The PRODUCT table contains 100 record sets. The table structure is as follows:

Table PRODUCT:

Column name	Type
PRODUCT_ID	INTEGER
NAME	VARCHAR(40)
CATEGORY	VARCHAR(40)
IMAGE	VARCHAR(40)
PRICE	DOUBLE
QTY_IN_STOCK	INTEGER
DESCRIPTION	VARCHAR(255)

6.4. XML Configuration Files

6.4.1. services-config.xml

```
<channel-definition id="rtmp-jdbcassembler"
     class="mx.messaging.channels.RTMPChannel">
     <endpoint uri="rtmp://fds:2037"
               class="flex.messaging.endpoints.RTMPEndpoint"/>
     <properties>
       <idle-timeout-minutes>20</idle-timeout-minutes>
       <client-to-server-maxbps>100K</client-to-server-maxbps>
       <server-to-client-maxbps>100K</server-to-client-maxbps>
     </properties>
 </channel-definition>
```

[3] *http://coenraets.org/blog/2006/11/building-database-driven-flex-applications-without-writing-client-or-server-side-code/*

6.4.2. data-management-config.xml

```
<destination id="jdbc-test" channels="rtmp-jdbcassembler" adapter="java-dao">
  <properties>
    <source>flex.samples.assemblers.SimpleJDBCAssembler</source>
    <scope>application</scope>
    <metadata>
      <identity property="PRODUCT_ID"/>
    </metadata>
    <database>
      <driver>org.hsqldb.jdbcDriver</driver>
      <url>jdbc:hsqldb:/Tomcat 5.5/flex/jdbcassembler/db/flexdemo</url>
      <table>product</table>
      <autoincrement>true</autoincrement>
     </database>
   </properties>
</destination>
```

6.5. Results

Several test runs were made using an increasing amount of browser instances. Run time per test was 60 seconds. We measured the total number of successful received messages (i.e. FDS notifications about updated records) and used the Windows Task Manager application to get an idea of the CPU load. The best message throughput is indicated in **orange**.

Data Management Test with JDBC-Assembler and HSQL					
# of Browser-Servers	# of Browser instances	Total # of Browsers	Total # of messages	# of messages/sec.	% CPU Load
4	1	4	1190	19,83	01-02
4	5	20	5970	99,50	04-10
4	10	40	7223	120,38	05-15
4	15	60	13620	227,00	05-15
4	20	80	14619	243,65	05-15

7. Conclusion

By using the Flex Stress Testing Framework we were able to get a good idea about general FDS performance. Of course, not every test we did necessarily resembles a real world situation. Especially when using the Data Management Service you typically use nested relations to map database relations. You may also want to use paging and lazy loading which adds additional overhead to FDS – we haven't tested this. Depending on your requirements, your mileage will vary.

Our focus was to measure the maximum amount of messages FDS is able to handle per second. As you see from the result tables there is no rule of thumb but the overall performance seems pretty good.