MAX 2006 Beyond Boundaries

Flex Data Services 101

Kai Koenig

Software Solutions Architect

kai@kaikoenig.de

14/11/2006

Adobe

Flex Data Services 101 - Agenda

- The Flex 2 product line
- Flex Data Services 2 – concepts and ideas
- FDS: RPC services
  - XML over HTTP
  - XML web services
  - Remote objects
- FDS: Messaging
  - Server-side message service
  - Client API
- FDS: Data management
  - Server-side data management service
  - Client API
- Q&A

About me

- Kai Koenig, Software Solutions Architect
- Based in Wellington, New Zealand – but I enjoy travelling ☺
- Work focus: Development, Training, Consulting, Mentoring in…
  - Flex
  - Java
  - ColdFusion
  - Software Engineering best practices
- Adobe Certified Master Instructor
- Adobe Certified Professional

- My blog:
  - Blog in Black (http://www.bloginblack.de)

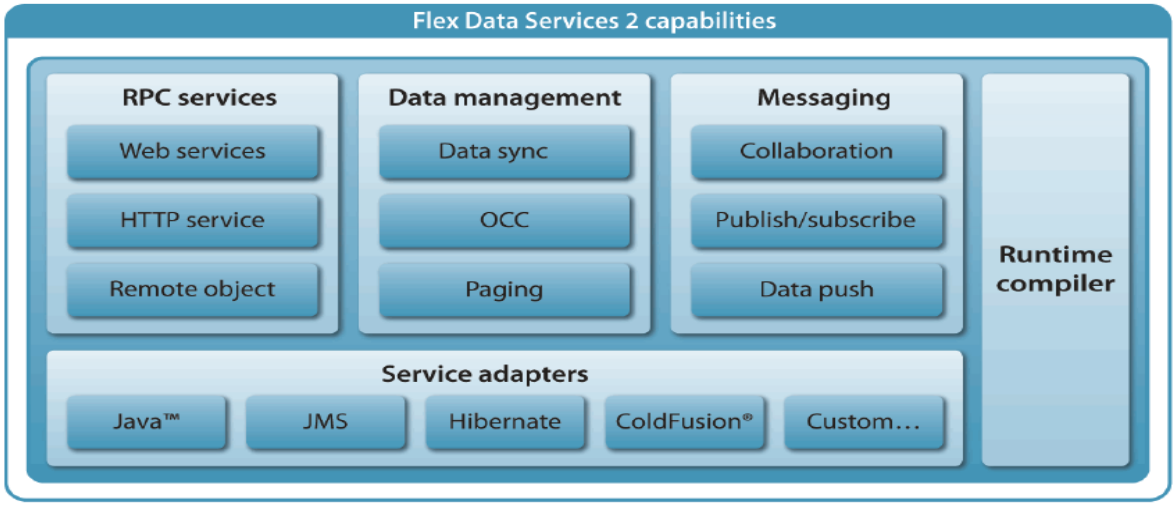
The Flex 2 product line

- You might know this already – or another session has covered it in depth…
- Flex 2 is an umbrella term for various products, the product line comprises:
  - Flex 2 SDK
    - Free SDK, consisting of the Flex 2 class library and command-line tools to work with. The free SDK allows to start developing Flex 2-based Rich Internet Applications without any costs or license fees.
  - Flex Builder 2
    - Eclipse-based IDE for Flex 2 development – provides features such as syntax highlighting, code completion, built-in compiler, code and design mode, built-in debugger etc.
    - Has to be licensed, retails at US$ 499 per developer seat (or US$ 749 including the charting components).
  - Flex 2 Charting Components
    - Additional class library containing UI components for charting purpose and building data dashboard applications
    - Have to be licensed, retail at US$ 249 (to be used with the free SDK) or to be licensed in combination with Flex Builder 2

The Flex 2 product line

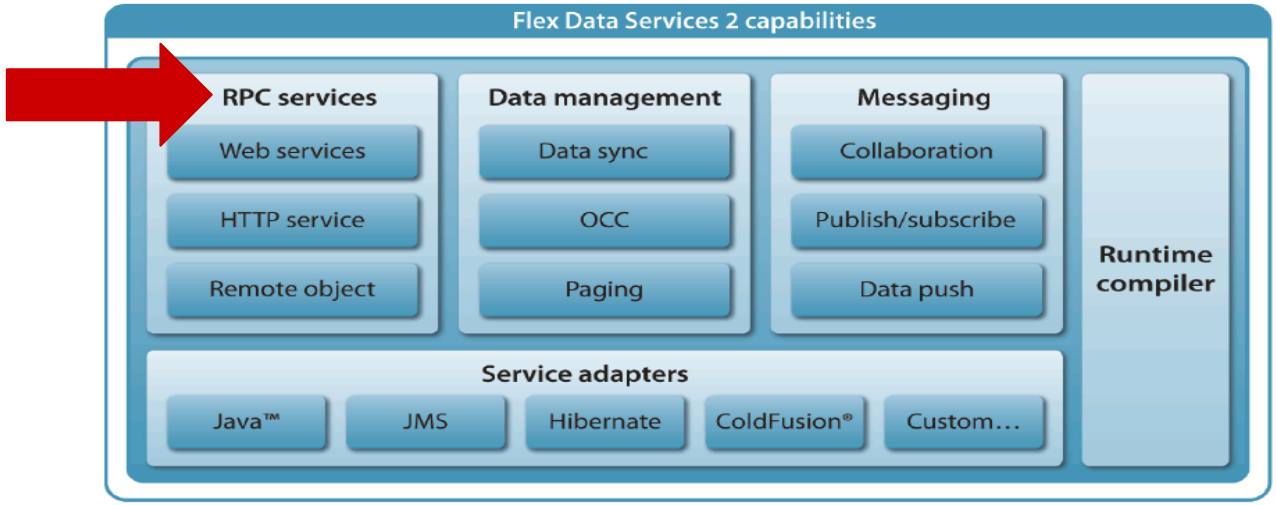
- And finally:
  - Flex Data Services 2
    - FDS add enterprise messaging support and an enhanced data services architecture to the Flex 2 SDK.
    - License fees depend on your infrastructure, licensed per machine/CPUs/clients, but there is the free Flex Data Services 2 Express (restrictions: 1 physical server, 1 CPU, no clustering), even for commercial installations.
    - Flex Data Services 2 provide
      - Access to remote objects (POJOs, JavaBeans, EJBs and ColdFusion Components)
      - Data sharing among multiple clients
      - Support for client-to-client data communications
      - Automated server data push
      - Authentication of client access to server resources
      - Logging
    - Execute on Java application server or Java servlet container
      - JRun 4 SP 6, Apache Tomcat 5.5, BEA WebLogic 8.1 SP2 / 9, IBM WebSphere 5/6, JBoss 4.0.3 and 4.0.4 etc.

Concept and ideas



- FDS2 provide three core building blocks: RPC services, Messaging and Data management – each of them comprising sub services
- Additionally:
  - Possibility to plug in different service adapters – to enable Flex to talk to existing enterprise infrastructures.
  - Runtime compiler – Applications running on FDS could be compiled, cached and executed at runtime, whereas regular Flex 2 SDK application HAVE to be compiled using either the command-line compiler or Flex Builder 2.

RPC services



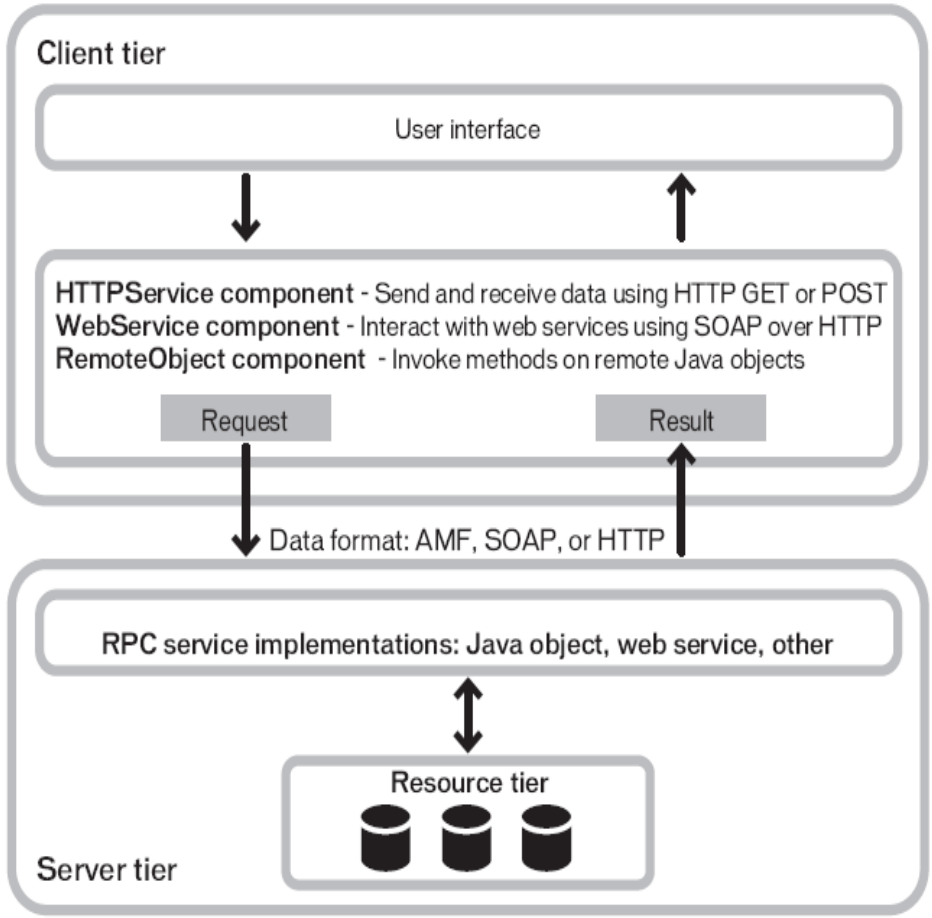
- Remote Procedure Call – allows Flex applications to call methods/operations on a local or remote server-side service, examples could be:
  - A web service providing financial information (stock data) – WS
  - A .jsp template that dynamically delivers XML data whenever pinged – HTTP service
  - A Java class that provides an API to talk to your company's CRM backend system – RO
  - A ColdFusion component that provides a service interface to the web CMS system – RO/WS

RPC services

- In general, the approach of using RPC services works as below:
  - Service developer provides an API for any of the existing connectivity options
  - The Flex 2 client application connects to the service and invokes a method or an operation
  - The service either returns the expected information (result) or an error (fault)
- This scenario moves the responsibility for dealing with the result or fault to the Rich Internet Application
- The communication between RIAs and a Service Oriented Architecture infrastructure is of an asynchronous character – and by that event-based
  - The Flex 2 client application listens to receive either a result or a fault event from its RPC invocation
  - Result: data could be processed or displayed
  - Fault: exception handling should be in place to act appropriately
  - Both parts of the application (Flex 2 client and service layer) are responsible for tracking and communicating data changes one either sides – and for resolving conflicts

RPC services

- Some RPC services can be implemented without Flex Data Services 2, particularly:
  - HTTP services
  - Web services
- There are good reasons though, why to use Flex Data Services 2, even if not necessary:
  - Ease of data management
  - Security and sandboxing



HTTPService

- Implemented by the HTTPService component (<mx:HTTPService>)
- HTTPService is used to send the common HTTP GET, POST, HEAD, TRACE etc. requests to a specified URI – data returned can be used in a Flex 2 application

```
<mx:HTTPService
   id="myHS"
   url="http://abc.com/my.jsp"/>

<mx:Button label="Retrieve Data"
   click="myHS.send()"/>
```

- Instance of HTTPService class is created, no call is being made yet!
- Invoking `.send()` on the instance actually sends the request.

HTTPService – results and faults

- There are several ways of retrieving the result or fault data from HTTPService objects – a good way is using the result and the fault event of the HTTPService and provide event listeners for both.
- The result type to be returned can be specified by using the `resultFormat` property of the HTTPService object – defaults to `object`, alternatives: `flashvars`, `e4x`, `xml` etc.

```
<mx:HTTPService id="myHS"
  url="http://mysite.com/flex
  HTTP.cfm"
  result="httpResultHandler
  (event)"
  fault="httpFaultHandler
  (event)"/>
```

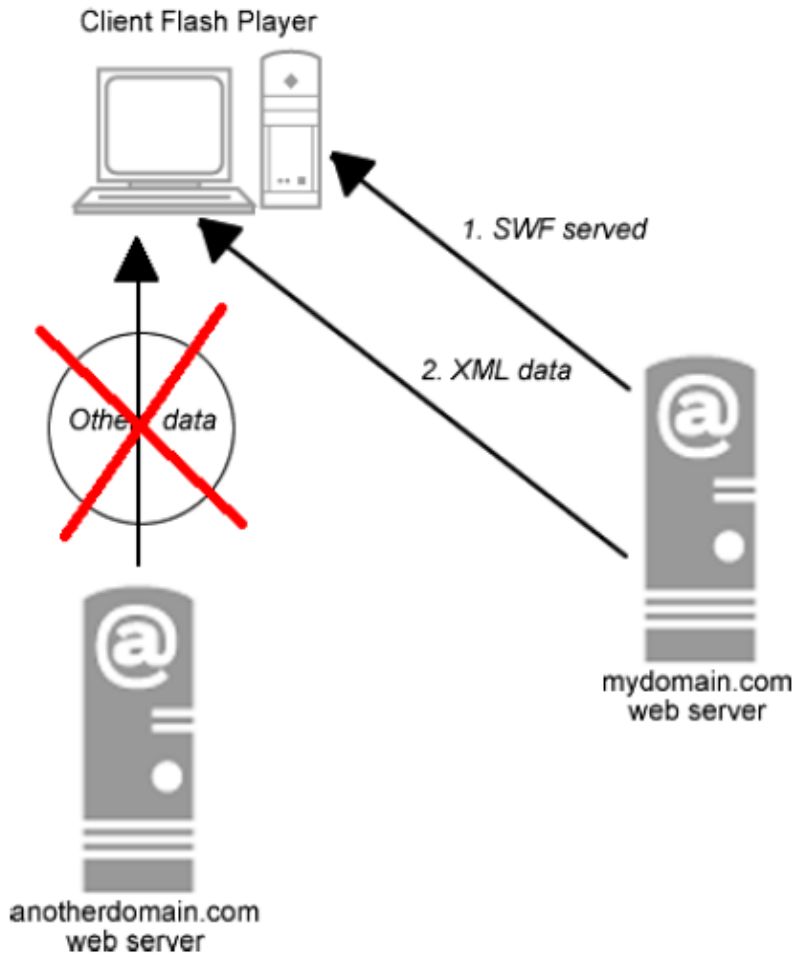
- The result event is of type mx.rpc.events.ResultEvent and contains the resultset in its result property
- The fault event has a fault property that contains fields such as `faultDetail`, `faultCode` and `faultString`

Note: mx.collections.ArrayCollection

- Flex by default converts XML data into an ArrayCollection
- ArrayCollection is a complex data structure that is the recommended data structure to be used as a data provider of complex UI components such DataGrid, List etc.
- Advantages:
  - The elements of an ArrayCollection could be used in a data binding and still be monitored
  - That doesn't work with an Array – as soon as you use an element in a data binding it would not be monitored anymore
  - ArrayCollection implements two important interfaces: ICollectionView and IList, both provide a rich toolset for data manipulation

Security sandbox

- By default the Flash Player doesn't allow an application to receive data from a domain other than the one the application was loaded from.

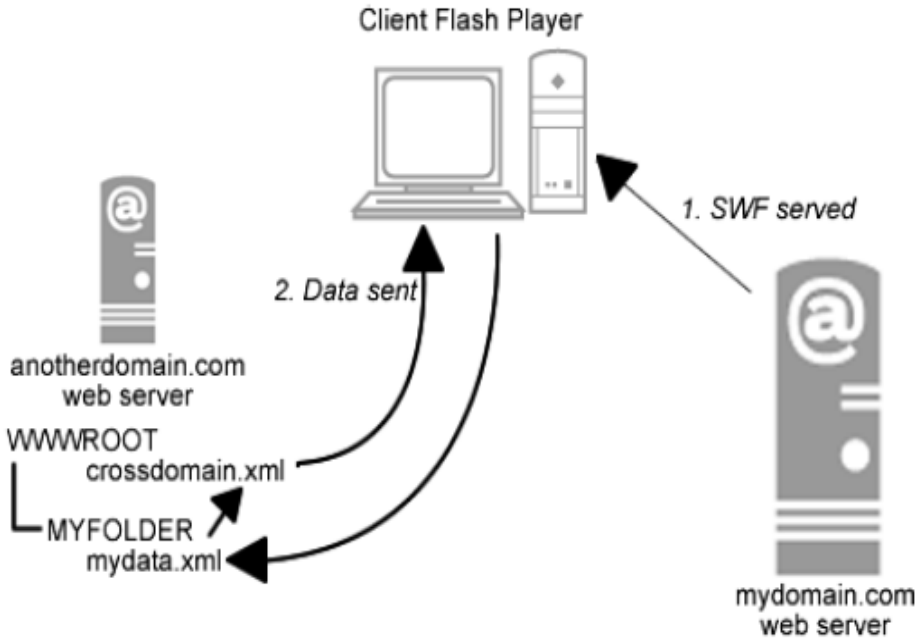


Security sandbox – crossdomain.xml

- One solution is to place a crossdomain.xml file in the webroot of the server the data should be loaded from.
- The code blow allows access from any domain:

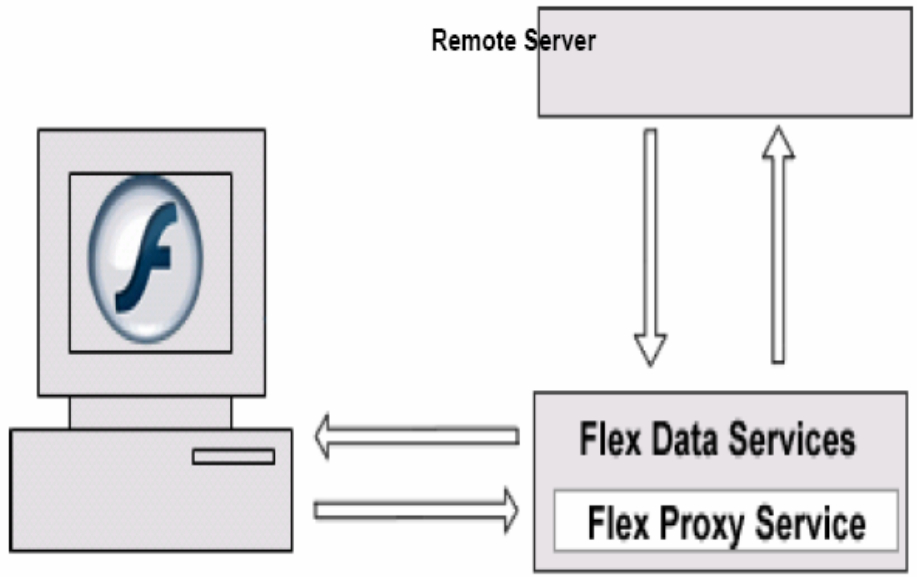
```
<cross-domain-policy>

<allow-access-from
   domain="*"/>

</cross-domain-policy>
```

- But in reality it might be difficult to get this file placed on other people's servers



Security sandbox – using FDS proxy service

- The Proxy service relays HTTP and web service requests from the Flex application in the browser to the server that's hosting the remote service
- Since the communication with the remote server now comes from Data Services, rather than from the Flash Player, you no longer need to create a cross-domain policy file
- To enable the Proxy service, provide `useProxy="true"` in the HTTPService tag



Web services

- Web services in Flex are rather straightforward and pretty close to HTTP services, the class to use is WebService (<mx:WebService>)
- Requirements:
  - The web service offers a WSDL description
  - The web service is accessible via SOAP over HTTP
  - Be careful when dealing with document-based web services, they are difficult to handle
- Process of calling a web service:
  - Ensure access to the web service from the application
  - Create WebService object
    - No request is made at object instantiation
    - WSDL file is parsed and necessary properties to access the web service object assigned to the WebService object
  - Call a method of the web service
  - Use the results

Web services - example

- Use result and fault handler for retrieving the result of an operation call, data will by default be in an ArrayCollection – also web services have to deal with the sandbox security!
- The code below creates a WebService object and invokes `arrayOutMethod` immediately after it has been created (in the `load` event handler).
- The `wsHandler` function deals with the result which is bound to a DataGrid

```
private function wsHandler(event:ResultEvent):void
{
  dataBack=event.result as ArrayCollection;
}
```

```
<mx:WebService id="simpletest"
wsdl="http://localhost:8700/Simple?wsdl" load="
simpletest.arrayOutMethod()" result="wsHandler(event)"/>
<mx:DataGrid dataProvider="{dataBack}"/>
```

Web services – more info

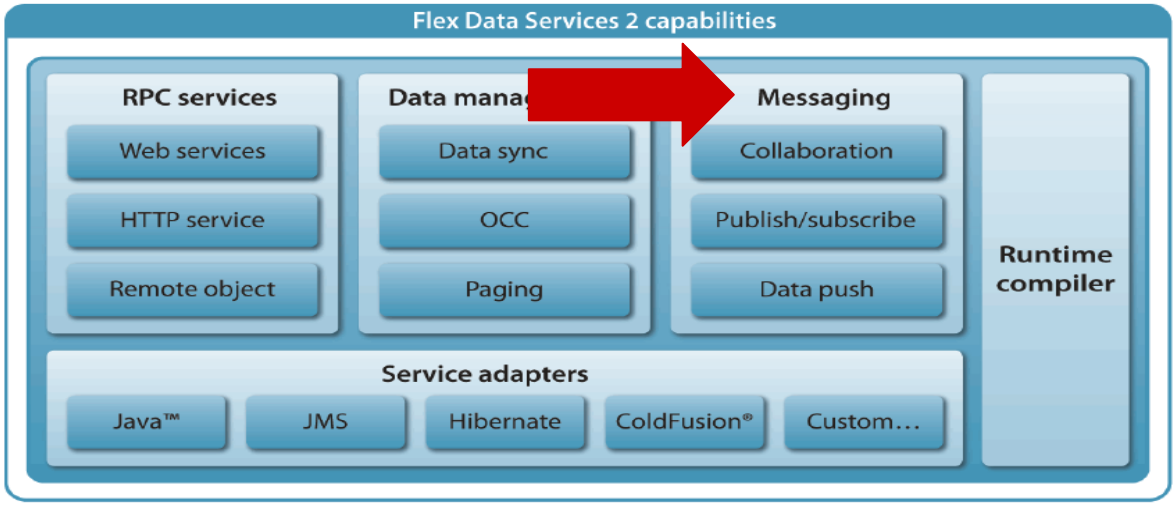
- A web service might contain more than one operation
  - Create the WebService once and specify multiple <mx:operation> tags
  - Each of those can have its own `result` and `fault` handler
- Passing parameters to a web service
  - explicitly: `<mx:WebService … load="wsData.arrayOutMethod(arg1, arg2)" />`
  - via parameter binding:

```
<mx:operation name="methodOne" result="handlerOne(event.result)"/>
  <mx:request>
    <fname>{myModel.fname}</fname>
    <lname>{myModel.lname}</lname>
  </mx:request>
</mx:operation>
```

Remote objects

- Remote object services are very similar to web services, but the communication between the application and the server is done using the AMF (Action MessageFormat) protocol, a **compressed binary protocol that is faster** than a content-equivalent SOAP message.
- AMF automatically translates the ActionScript 3 data types into appropriate data types of the used server technology (Java or ColdFusion) – detailled tables on how that works are contained in the Flex documentation.
- The class to use for implementing Remote objects is RemoteObject (<mx:RemoteObject>) – using Flex Data Services allows you to connect to server-side Java classes (such as PoJos, EJBs, Java Beans etc.)
- For ColdFusion developers: If you run ColdFusion MX 7.02, you can use the Remote object technology to connect to ColdFusion components WITHOUT running the Flex Data Services.
- For RemoteObject, <mx:operation> becomes <mx:method>…

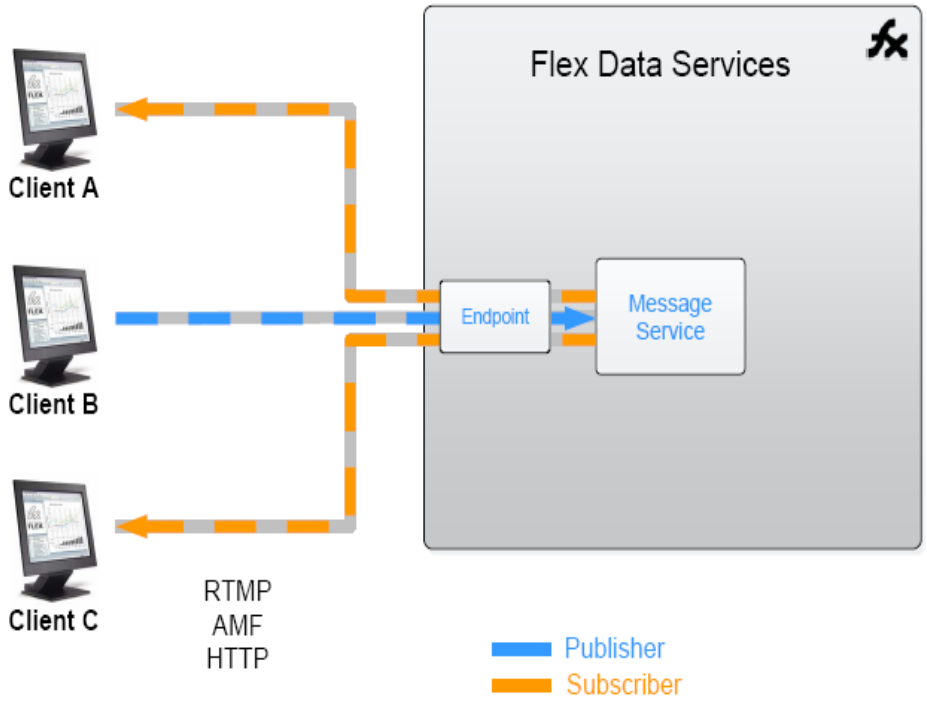
Messaging



- Messaging consists of two core components:
  - Client API
  - Messaging server component on FDS
- Flex client applications use the Client API to send messages and receive messages from/to server-side destinations
- Examples: real-time chat, real-time collaboration, server-pushed communication

Communication flow

- Messaging relies on a publisher/subscriber model
- Flex clients subscribe or publish to a particular server-side destination, in the example provided:
  - Client B publishes information
  - Client A and C are subscribed to the destination that client B has published to and consume the data via FDS
- To allow clients to publish and subscribe, you have to configure the destinations in FDS
- Possible considerations:
  - Message adapter (AS or JMS)
  - Message channel (push/pull) etc.



Configuring destinations

- `services-config.xml` and `messaging-config.xml`, example:

```
<destination id="chatDestination">
  <properties>
    <network>
      <session-timeout>0</session-timeout>
    </network>
    <server>
      <max-cache-size>1000</max-cache-size>
      <message-time-to-live>0</message-time-to-live>
      <durable>false</durable>
    </server>
  </properties>
  <channels>
    <channel ref="my-rtmp"/>
  </channels>
</destination>
```

Development: Producer

- Declaring a producer

```
<mx:Producer id="producer" destination="chat" />
```

- Publishing a message

```
import mx.messaging.messages.AsyncMessage;
var message:AsyncMessage = new AsyncMessage();
message.headers.user = user;
message.body = msg.text;
producer.send(message);
```

- `body` can be any ActionScript object
- `headers` is a generic ActionScript object that holds key/value pairs

Development: Consumer

- Declaring a consumer

```
<mx:Consumer id=“consumer" destination="chat"
message="handleMessage(event)" />
```

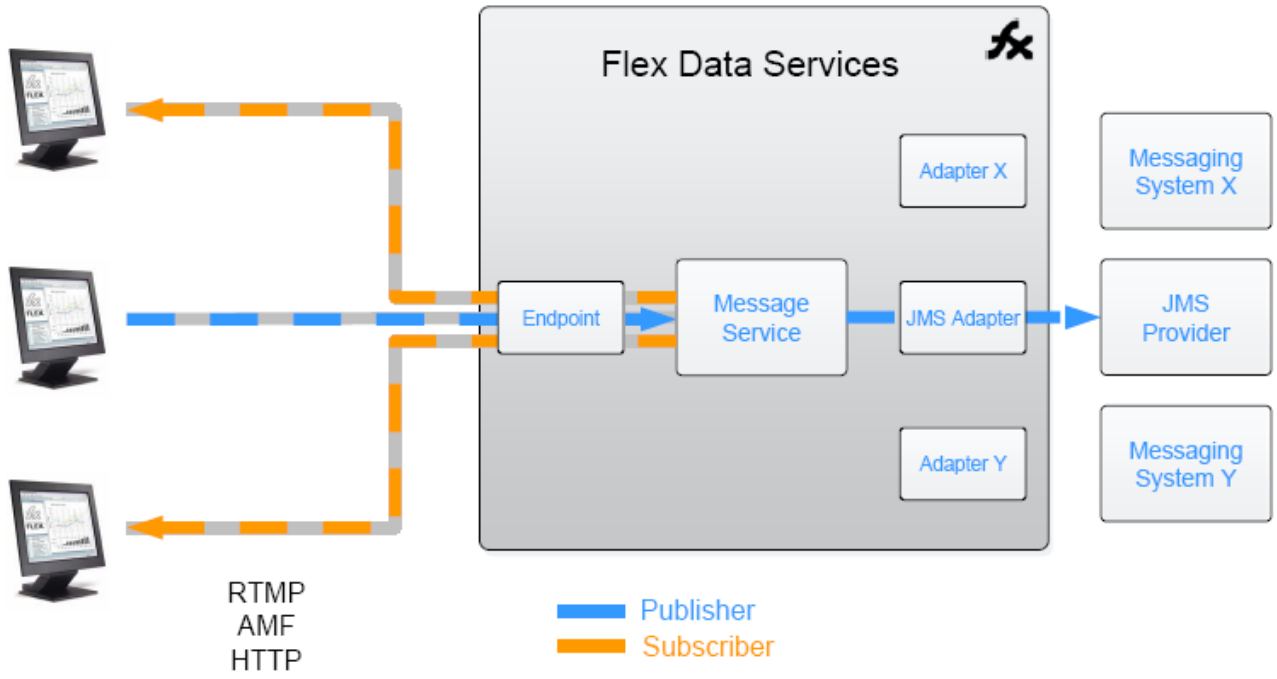
- Subscribing to a destination

```
consumer.subscribe();
```

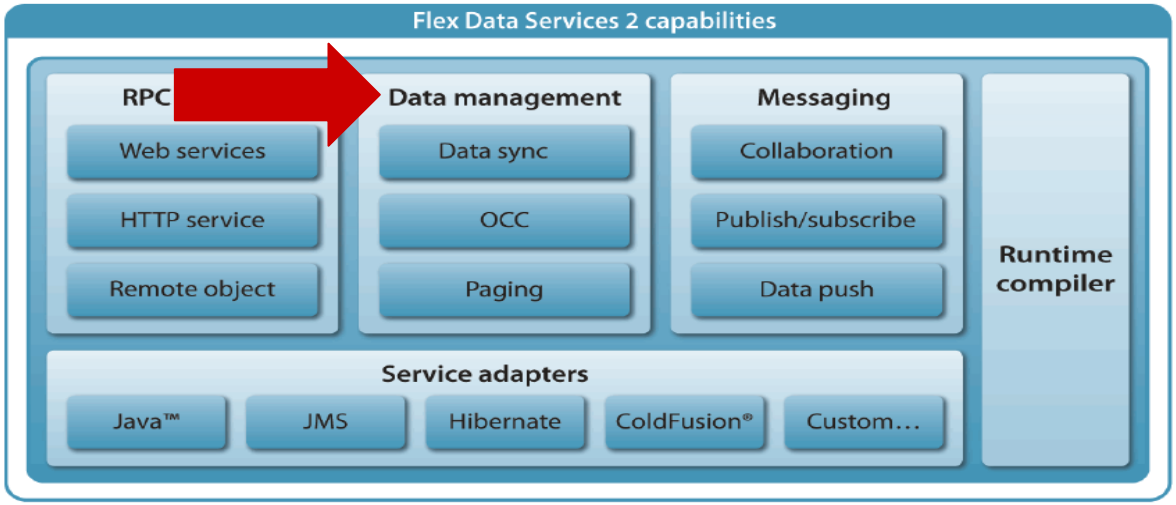
- Once a Consumer has subscribed, any message sent to this destination will be forwarded to the Consumer component by the Message service.
- When a message is received, the Consumer broadcasts a message event
  - The event object will be an instance of mx.messaging.events.MessageEvent
  - The event object has a `message` property
- The details of the message will be available in the message object’s `body` and `headers` properties (where the details were placed by the Producer when the message was published).
- The specified event handler can then retrieve `body` and `headers` to process the message data.
- Additional feature: Consumer objects can filter the incoming messages by using the selector property of <mx:Consumer>: … `selector="user='someUser'"` …

External 3rd party message services

- By providing data push and real-time communication, the Message service allows new categories of applications to be delivered to the browser.
- The open and scalable architecture of FDS allows integration into existing messaging solutions.



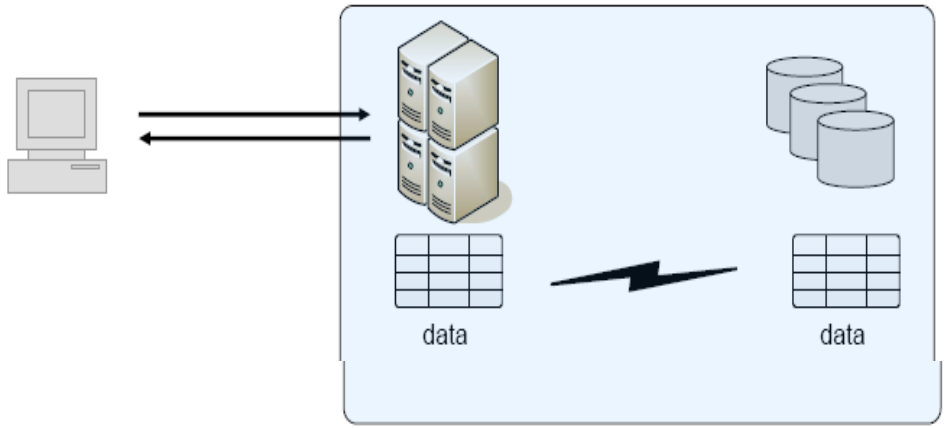
Data management



- Data management consists of two core components and relies on Messaging
  - Client API
  - Data management server component on FDS

Working with data in HTML

- In the traditional HTML-based and page-centric web application model, the client has the role of a data display and capture device – the client NEVER owns the data.
- Data needs to be synchronized between middle-tier and data storage



Working with data in RIAs

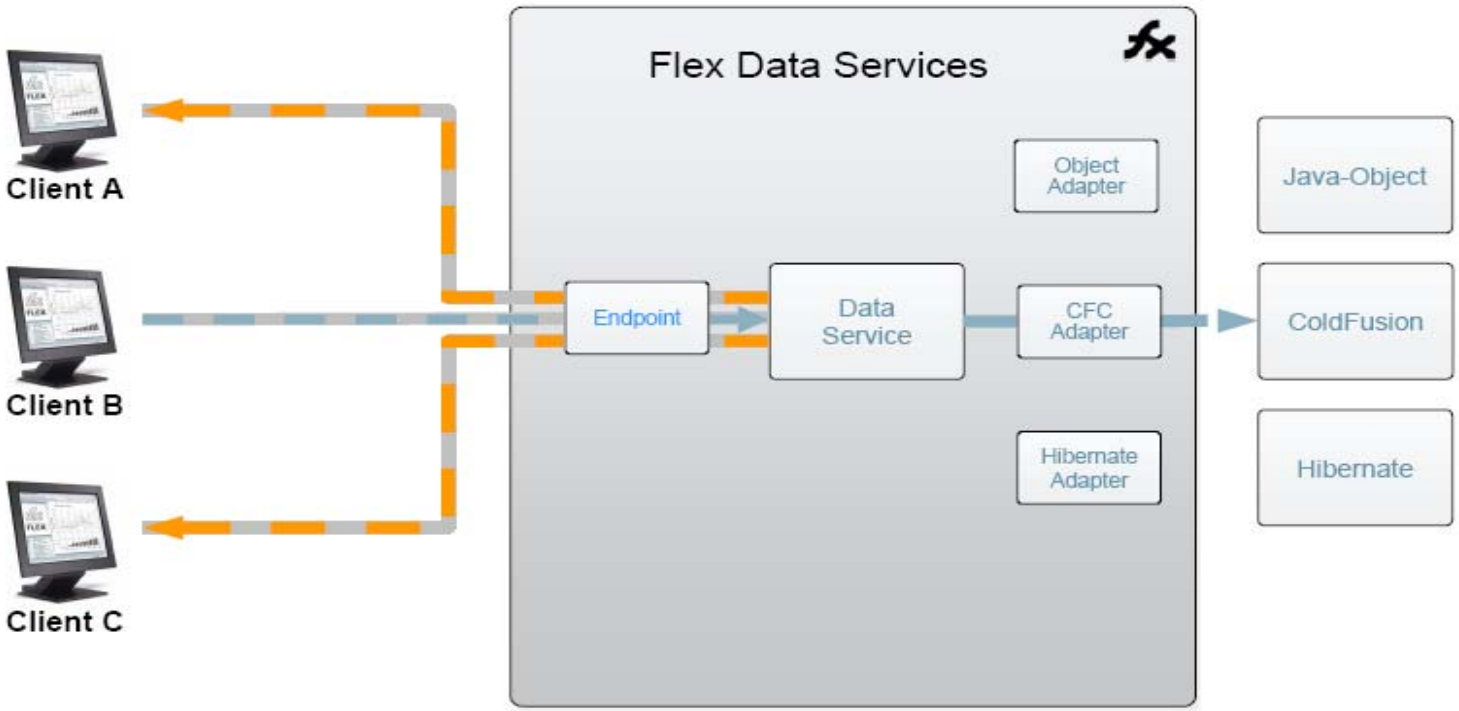
- In Rich Internet Applications, the clients owns a version of the data.
- That implies that data needs to be synchronized between the rich client and the middle-tier as well.

- By using the RPC services, this synchronization is the responsibility of the developer – flagging data that needs to be updated, issue and conflict resolution etc.
- Flex Data Management provides an automated solution.

Client API

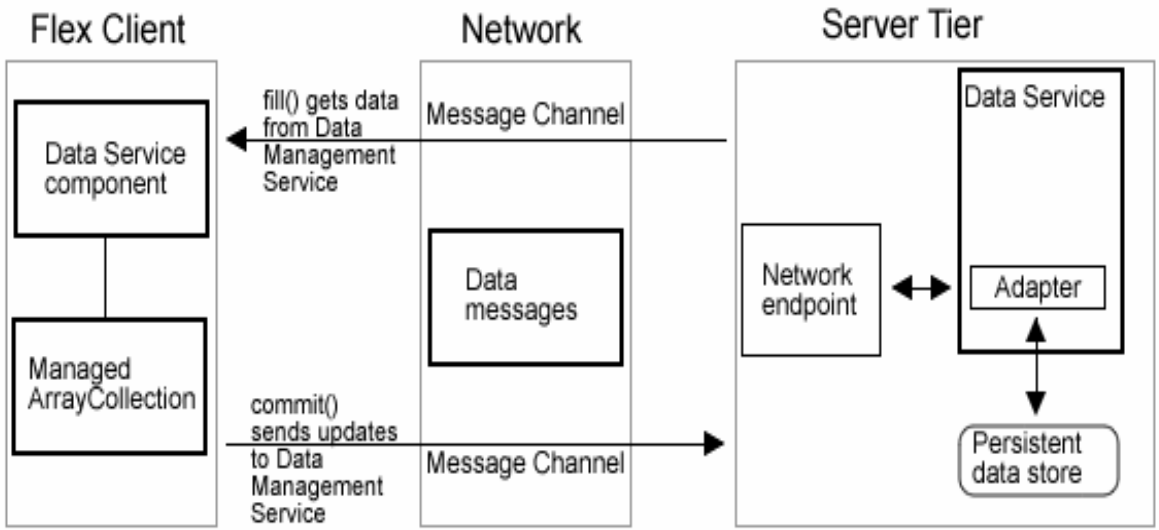
- Flex Data Management Client API features:
  - Flags changes (update, delete, insert)
  - Maintains original version of the data for appropriate locking
  - Notifies the Flex Data Services instances of changes via messages
  - Is able to deal with conflict resolution or concurrency (what happens i.e. if two clients change the same data concurrently)
- Server-side features:
  - **Data Service:**
    - De-serializes list of changes coming from client
    - Serializes data before sending to client
    - Passes list of changes to configured adapter
    - Notifies clients who subscribed to destination
  - **Adapter** (that could be an adapter to JDO, Hibernate, EJB, ColdFusion or any custom solution)**:**
    - Processes changes
    - Identifies conflicts
    - Passes results to Data Service

Data Management architecture



Implementing Data Management

- From 30,000 feet – a three step process:
  - Create Assembler classes or components in ColdFusion or Java
  - Create a data management destination in FDS
  - Use <mx:DataService> and implement your application



Data Transfer Object

- Complex data should be transferred between the Flex client and FDS by using the Data Transfer Object pattern (DTO – also known as value objects).
- DTOs in Flex are ActionScript objects and Java classes or CFCs on the server.
  - http://java.sun.com/blueprints/corej2eepatterns/Patterns/TransferObject.html
- The AS DTO class in Flex needs to be `managed` and needs to specify a mapping to a DTO class on the server – data type conversion is done automatically – details provided in the documentation.

```
package dto {
[Managed]
[RemoteClass(alias="dms.Sandwich")]
  public class Sandwich {
    public var sandwichId:int;
    public var sandwichName:String="";
  }
}
```

Assembler

- Transfer Object Assembler is another design pattern.
  - http://java.sun.com/blueprints/corej2eepatterns/Patterns/TransferObjectAssembler.html
- The assembler class is the first point of contact after the Data Management Service received a message from a flex client and has to be specified with the destination used in FDS.
- The destination can define the following methods:

Method	Purpose	Return datatype
`<fill-method>`	Retrieves data from server	List
`<count-method>`	Retrieves number of items from server without retrieving the entire dataset	`int`
`<get-method>`	Retrieves a single item from the server	`typed DTO object`
`<sync-method>`	Updates persistent data store with pending changes from client	`void`

Assembler

- A common approach is that the methods of the Assembler class call methods of the Data Access Objects (DAO), which is another design pattern.
  - http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html
- The DAOs are an abstraction layer of the data storage.
- Enforces a separation of concerns – there is no need that the Assembler class knows how the data is stored and accessed.

Data Management destination

- The Data Management `destination` is defined in `services-config.xml` and `data-management-config.xml`, example:

```
<destination id="artists">
  <adapter ref="coldfusion-dao"/>
  <channels>
    <channel ref="cf-dataservice-rtmp"/>
    <channel ref="cf-polling-amf"/>
  </channels>
  <properties>
    <component>CFDataServices.MyAssembler</component>
    <scope>request</scope>
    <metadata>
      <identity property="PersonID"/>
    </metadata>
    <server>
      <fill-method>
        <use-fill-contains>false</use-fill-contains>
        <auto-refresh>true</auto-refresh>
        <ordered>true</ordered>
      </fill-method>
    </server>
  </properties>
</destination>
```

<mx:DataService>

- Defining a Data Management service instance in your Flex application is pretty straight forward:

```
<mx:DataService id="ds" destination="artists"
autoCommit="true" />
```

- The `.fill(target)` method retrieves the initial data from the Data Management service and places it in the object instance `target` – which pretty often would be of type ArrayCollection.
- `autoCommit=true` implies that each and any change to the data in the Flex client (each keystoke) is sent to the FDS server immediately. This message would trigger a server action (such as a SQL update) in the data storage – that's not necessary what you want.
- Disabling `autoCommit` gives you the control when the data is sent back to the server by using the `.commit()` resp. `revertChanges()` methods.
- You can also use `.deleteItem()` or `.createItem()` to change the data in the Flex client.

Conflicts

- When two or more clients are editing the same data simultaneously in an application that uses synchronized data, it's possible for conflicts to occur between versions submitted by different clients
- The Data Management Service provides both client-and server-side APIs to detect and manage those scenarios.

```
<mx:DataService id="ds" destination="artists" autoCommit="false"
conflict="conflictHandler(event)" />

…

public function conflictHandler(event:DataConflictEvent):void {
  var conflicts:Conflicts = ds.conflicts;
  var c:Conflict;
  for (var i:int=0; i<conflicts.length; i++) {
    c = Conflict(conflicts.getItemAt(i));
    Alert.show("Reverting to server value", "Conflict");
    c.acceptServer();
  }
}
```

Paging and OCC

- OCC = Occasionally connected clients -> Visit the Apollo session tomorrow!
- Paging:
  - When a dataset has a large number of records, you may want to cause only a small number of records to be sent from server to client at a time
  - The server-side destination has a paging property that allows you to define how many records should be sent from client to server at a time
    - Maximum message size can be specified on destination
    - If size exceeds maximum value, multiple message batches are used
    - Client reassembles separate messages
    - Enables asynchronous data paging across the network
  - User interface controls can display portions of the collection without waiting for the entire collection to load

Contact

CERTIFIED PROFESSIONAL
Macromedia® Flash® MX
Developer

CERTIFIED PROFESSIONAL
Macromedia® ColdFusion® MX 7
Advanced Developer

CERTIFIED INSTRUCTOR
Adobe® Flex™
Macromedia® Breeze®
Macromedia® ColdFusion®

Kai Koenig

Software Solutions Architect

development - training - consulting - mentoring

154 Parkvale Road
Wellington
New Zealand

kai@kaikoenig.de

skype: +49 201 9586 902
mobile: +64 21 0234 7466

Better by Adobe™