Building real-time web applications with Flex®

Adobe LiveCycle® Data Services ES supports real-time rich Internet applications developed on the Adobe Flash® Platform

Table of contents

It's easier than ever to build and deploy dynamic, scalable web applications that can transact with live data in real time. Using powerful client-side technologies in the Adobe Flash Platform—Flex®, Adobe AIR®, and Flash—developers can create rich Internet applications (RIAs) that subscribe to live feeds via integration with back-end services. Adobe LiveCycle Data Services ES runs on the server to provide real-time transactional services, enabling RIAs to pull and aggregate information from core enterprise applications and feeds outside the firewall. In the past, deploying real-time applications has been a challenge, often relying on client/server architectures that are expensive to maintain. Web-based Ajax applications solve part of the problem, but they might have difficulty handling large data sets and supporting multiple browsers and versions. The Flash Platform, combined with LiveCycle Data Services ES, delivers an application platform that solves many of your real-time data requirements.

The success of your real-time application depends on selecting the appropriate platform as well as implementing the right information management strategy. This white paper explores the different requirements of a real-time application, along with high-performance data-handling strategies to help you select the appropriate methodology.

The challenge of real time

Real-time and near-real-time applications must be able to quickly detect and respond to information—sometimes within milliseconds. Developing these types of applications poses unique challenges and often involves making informed trade-offs to deal with existing constraints.

There are many potential sources of latency, the greatest of which is beyond control: the public network. Businesses might not have an IT infrastructure that's tuned and optimized to deliver data quickly. Any resource constraints in web or application servers, the network, or a client can impact the performance of a real-time RIA.

Here are some things to consider when approximating future performance of your RIA.

Adobe

Performance Considerations	Impact	Requirements
Data sources	• When merging external and internal data sources, sources are often out of sync, which can cause confusion • Checking for data at too-small or too-large of intervals creates delays in your application • Transmitting a variety of types of data sources, such as video and text, creates deployment and development challenges • Synchronizing concurrent user and system events across the client base requires sophisticated logic, thereby impacting performance	• Synchronize data sources that are time sensitive • Automatically monitor and detect changes to the data • Support a wide range of data source formats with a single client • Have a solution that can handle different polling schemes • Use a solution that works across both client and server
Scalability	• A large base of concurrent subscribers can tax server hardware resources and clog networks • The greater the frequency of change in your data, the more persistent the client-server communication, which can tax performance	• Use the client to perform sophisticated operations, reducing the burden on the server • Scale the deployment both vertically and horizontally, and provide real-time failover • Only transmit the data that has changed
Deployment	• Working on a variety of different platforms and browsers creates deployment and development challenges • Connection speeds differ across clients, which creates a potential for data loss • External clients often need access to data behind the firewall, which creates configuration and deployment complications • Internal client collaborating with external clients creates deployment and development challenges	• Standardize clients that work across several platforms and browsers • Use bandwidth throttling to not overload the client machine and lose information • Securely retrieve and transmit data through firewalls • Guarantee information delivery to the client

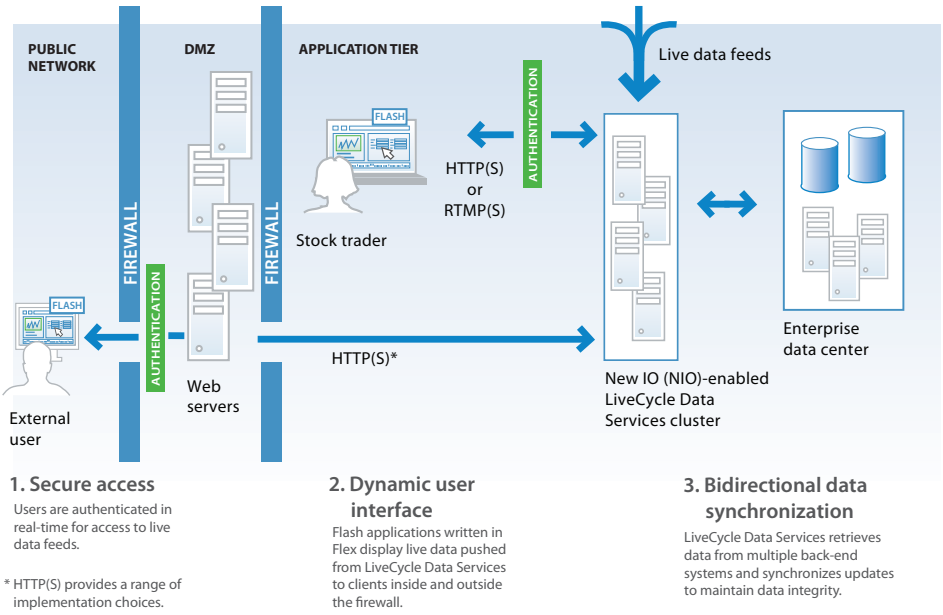
Real-time application requirements

A stock trading application provides an excellent example of a real-time application. The solution is based on several components. Multiple external data feeds from a variety of worldwide exchanges, news, and research services, enter a server or server cluster. The system then distributes the pertinent information to clients in real time according to subscription policies.

The stock trader views all the information on screen using a consolidated dashboard interface that displays live graphs, charts, portfolio information, video feeds, and so on. The stock trader can drill down on certain information, such as a commodity, and instantly view the relevant charts and analysis.

When stock traders perform a trade on behalf of a client, they can directly access the client's information, including a current portfolio, desired risk levels, past trades, and available credit.

Example: Real-time stock trading application



A real-time application deployment using HTTP or RTMP for client connections behind the IT demilitarized zone (DMZ), and HTTP for communications across public networks.

LiveCycle Data Services ES provides the communication hub connecting both internal and external clients to the required data sources. Communication with internal users can also be over HTTP, HTTPS, Real Time Messaging Protocol (RTMP), and RTMPS. Communication with external users can be over HTTP and HTTPS.These protocols deliver data in real time to dynamic client RIAs developed using Flex and Flash technologies.

LiveCycle Data Services ES overview

LiveCycle Data Services ES is a high-performance, scalable, and flexible framework that streamlines the development of RIAs using Adobe software and the Adobe AIR runtime. LiveCycle Data Services ES abstracts the complexity required to create server push–based applications and supports a rich set of features to create real-time and near-real-time solutions. Backed by a powerful data services API, the software simplifies data management problems such as tracking changes, synchronization, paging, and conflict resolution.

Adobe solutions for the enterprise can provide these real-time features and capabilities:

- The Adobe Flash Platform provides the interactivity required for applications to present real-time data to clients.
- The Flash Player and Adobe AIR technologies enable the client to maintain a persistent connection to the server—a requirement of real-time communication.
- The Flash Player uses Action Message Format (AMF), which enables faster data transmission than either the JavaScript Object Notation (JSON) or XML formats typically used by web applications.
- Applications using Flash technology can embed logic on the client, helping to reduce the load on the server.
- LiveCycle Data Services ES provides an enterprise-class, real-time platform that can push data from multiple sources, including changes from other users, to the Flash and Adobe AIR runtimes.

LiveCycle Data Services ES is a high-performance J2EE server-based framework that enables a variety of real-time methodologies and protocols to suit your specific application requirements. Developers can also configure multiple channels per connection, work around environmental constraints on the network or client, and safeguard against exhausting browser HTTP connections.

Server-side architecture

LiveCycle Data Services ES contains a set of libraries that you can add to your J2EE web application. It also contains a Flex library that lets you link into your Flex application to talk to the server. When a Flex or Adobe AIR client requests a service, the request is routed to the LiveCycle Data Services ES server, and from there to an adapter object. The adapter fulfills the request either locally or by contacting a back-end system or remote service such as Java Message Service (JMS).

Message services

A typical messaging system has a consumer that consumes messages produced by the producer. Messaging services facilitate publishers to publish (or send) messages, and consumers to subscribe (or receive) messages. When a message is published, a message event is triggered for subscribed consumers. The messaging service provides messaging adapters that manage client subscriptions and publish messages to clients as they become available. Developers can write their own message adapters to perform custom message processing or subscription management.

Clients can employ a range of channel types (a means for client and server to communicate), including HTTP simple polling, HTTP long polling, HTTP streaming, RTMP real time, or RTMPT (tunneling RTMP over HTTP). You can specify a channelset (a collection of channels) per client, each tried in sequence until one succeeds.

A message services application consists of a destination (provide a way to bind a client-side service component to server-side functionality via a simple “name), a producer (Java™ or Adobe ActionScript®) to publish messages to the destination, and a client that receives messages from the destination. The producer represents the live data source and can be triggered by another user a (Flex client) or server process (an external feed or application). The following example shows the destination definition in message-services-config.xml.

```
<?xml version="1.0" encoding="UTF-8"?>
<service id="message-service"
 class="flex.messaging.services.MessageService">
 <!-- Represents the name of the feed that producers and consumer
 can use-->
 <destination id=”feed”/>
</service>
```

Data Management Services

Within LiveCycle Data Services ES, Data Management Services provides advanced functionality such as data synchronization between clients and servers and among clients. It also contains a conflict-resolution API to help ensure data integrity, such as when multiple clients attempt to edit the same data simultaneously, by making it easy to detect and resolve conflicts. Data Management Services can broker access to back-end data and provides flexibility to handle conflicts according to the business needs of the application.

Data Management Services also offers paging facilities for large sets of data. Paging can significantly improve application performance by decreasing query times and reducing the amount of memory used. Paging and lazy fetching features make it easier to build applications that deal with large, complex data models. Data Management Services supports automatic and manual synchronization of a common data set across multiple clients and server-side data resources, as well as client-side data persistence for occasionally-connected clients. It uses a robust, high-performance data synchronization engine that removes the complexity and error potential of rich client data synchronization. Change-tracking, revert, offline, refresh, and conflict-detection features make it easy to build web applications that provide a robust user experience. Existing back-end integrations, such as Hibernate and the SQL assembler, also speed application development.

In the following message destination configuration sample from message-services-config.xml, the application is both a message consumer and a message source, capable of sending to and receiving from other clients.

```
<destination id=”chat”/>
```

This destination can be used with simple MXML code to create a chat-style application. The following excerpt from the producer component code illustrates sending messages via the send() method and receiving via messageHandler. Additional code would handle display to the user, for example, using a TextArea control.

```
<mx:Script>
 <![CDATA[

 import mx.messaging.messages.AsyncMessage;
 import mx.messaging.messages.IMessage;

 private function send():void
 {
 var message:IMessage = new AsyncMessage();
 message.body.chatMessage = msg.text;
 producer.send(message);
 msg.text = "";
 }

 private function messageHandler(message:IMessage):void
 {
 log.text += message.body.chatMessage + "\n";
 }

 ]]>
</mx:Script>

<mx:Producer id="producer" destination="chat"/>
<mx:Consumer id="consumer" destination="chat"
message="messageHandler(event.message)"/>
```

The following configuration from data-management-config.xml contains one destination that uses the Java adapter to interact with a data resource. The auto-sync-enabled element controls the default value of the client-side DataService.autoSyncEnabled property for clients that are using the destination. A Data Management Services destination contains one or more identity elements that you can use to designate data properties to be used to guarantee unique identity among items in a collection of objects.

```
<destination id="inventory">
 <adapter ref="java-dao" />
 <properties>
 <source>flex.samples.product.ProductAssembler</source>
 <scope>application</scope>
 <auto-sync-enabled>true</auto-sync-enabled>
 <metadata>
 <identity property="productId"/>
 </metadata>
 <network>
 <paging enabled="false"/>
 </network>
 </properties>
</destination>
```

An MXML client for this destination could be as simple as the following application, which gets data from the destination and displays it in a DataGrid control. The DataGrid is editable. Any changes the user makes to values in the grid are committed to the destination and processed by the destination's assigned assembler Java class, committed to the database, and synchronized with the versions of the same data that other instances of the client have open.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" xmlns="*"
backgroundColor="#FFFFFF">

 <mx:ArrayCollection id="products"/>
 <mx:DataService id="ds" destination="inventory"/>
 <Product/>

 <mx:DataGrid dataProvider="{products}" editable="true" width="100%"
height="100%">
 <mx:columns>
 <mx:DataGridColumn dataField="name" headerText="Name"/>
 <mx:DataGridColumn dataField="category" headerText="Category"/>
 <mx:DataGridColumn dataField="price" headerText="Price"/>
 <mx:DataGridColumn dataField="image" headerText="Image"/>
 <mx:DataGridColumn dataField="description" headerText="Description"/>
 </mx:columns>
 </mx:DataGrid>
 <mx:Button label="Get Data" click="ds.fill(products)"/>
</mx:Application>
```

Endpoints

LiveCycle Data Services ES provides servlet-based and NIO-based endpoints. Servlet-based endpoints run from within the J2EE servlet container, with the servlet handling I/O and HTTP sessions for the endpoints. An HTTP session listener is registered with the J2EE server, providing LiveCycle Data Services ES with HTTP session attribute and binding listener support. NIO-based endpoints run in a NIO-based socket server. These endpoints can offer significant scalability gains because they are not limited to one thread per connection, and a single thread can manage multiple I/Os. Flex and Adobe AIR client applications communicate with LiveCycle Data Services ES endpoints over channels, which are mapped to endpoints on the server using the same message format, such as AMF or streaming.

LiveCycle Data Services ES provides secure channels and endpoints to implement a secure transport mechanism. LiveCycle Data Services ES provides a secure version of the AMF, HTTP, and RTMP channels and endpoints that transport data over an RTMPS or HTTPS connection.

Determining real-time application needs

When choosing a communication methodology for your real-time RIA, consider its performance requirements as well as a range of other factors, including the need to authenticate users, the number of anticipated concurrent connections, the network topology, and any data management requirements.

This paper examines four methodologies of communication, outlining the strengths and weaknesses of each approach as it relates to the IT infrastructure and the business requirements for your RIA.

Piggybacking and polling

A simple application such as a real-time sports scoreboard might tolerate relatively high latencies, such as five or ten seconds. These applications typically need to traverse firewalls and proxies on the public Internet between source and destination. In this case, HTTP polling is an acceptable option.

After handshaking with the server to establish a connection, the client subscribes to the available push messages such as Football, Baseball, and Basketball or, more specifically, to individual teams. Subscribed clients poll the server on some interval that defines a maximum latency, and pushed messages are communicated back to the client in the response. Any subscribed messages arriving at the server between polls are queued briefly. Such an approach has the advantage that server performance can be tuned by simply adjusting the polling interval. Although this is not a complex application, it does have scaling limitations.

An even simpler approach to HTTP-based, real-time development is piggybacking. After the clients are subscribed to messages, they do not poll the server, but instead rely on other HTTP requests to the server and pass real-time messaging with that traffic. Because these requests are unpredictable, latency is potentially unbounded. Although piggybacking alone might not be suitable for a scoreboard application, combining polling and piggybacking can guarantee delivery within a finite period while providing greater responsiveness by using piggybacking opportunistically.

The following example illustrates HTTP piggybacking with polling enabled on an eight-second interval, as defined in the properties section.

```
<!-- AMF with piggybacking and polling every 8 seconds -->
<channel-definition id="samples-polling-amf"
 type="mx.messaging.channels.AMFChannel">
 <endpoint url="http://{server.name}:8700/dev/messagebroker/amfpolling"
 type="flex.messaging.endpoints.AMFEndpoint"/>
 <properties>
 <polling-enabled>true</polling-enabled>
 <polling-interval-seconds>8</polling-interval-seconds>
 <piggybacking-enabled>true</piggybacking-enabled>
 </properties>
</channel-definition>
```

Long polling

Piggybacking, polling, and combinations of the two have limitations, particularly when latency is a key consideration. An application like a call monitoring system for call centers, which provides agents with information such as the current number of queued calls, average wait times, and call completion percentages, has a lower tolerance for message latency. In this case, you could utilize HTTP long polling as the real-time messaging mechanism. In long polling, subscribed clients poll a LiveCycle Data Services ES server at some interval such as 10 seconds, as in simple polling. However, if no messages are available, the request is parked for some preconfigured period. If no messages are received within that period, the request expires and is deleted. If, on the other hand, a message is received while the request is parked, it is unparked, and the message is passed to the client immediately. This approach improves latency. Like polling and piggybacking, long polling utilizes network-transparent HTTP traffic, an important consideration if multiple call centers and home-based agents must be reached via the Internet.

Long polling ties up a browser connection to the server. Because connections are a scarce resource, and the number available varies among browsers, this could become problematic if agents must simultaneously connect to multiple resources, such as a knowledgebase and a troubleticketing system. Planning around connection limits is further complicated if browsers are not standardized across clients, as in the case of home-based agents. Using the default AMF can greatly alleviate this problem because requests are batched within a single frame by Flex or Adobe AIR, and only one outbound HTTP request is made, making significantly more efficient use of limited connection resources. Another concern in using long polling is the potential for latency degradation if the messages to be processed arrive at the server at a high rate. Delivering each discrete set of queued messages involves multiple network traversals, from the server to the client, back the server, and again to the client. In the case of a call center application, this can become a problem during spikes in call volume.

The following example illustrates long polling for a channel that uses a nonstreaming AMF or HTTP endpoint by setting the polling-enabled, polling-interval-millis, wait-interval-millis, and client-wait-interval-millis properties in a channel definition.

```
<!-- Long polling AMF -->
<channel-definition id="my-amf-longpoll" class="mx.messaging.channels.
AMFChannel">
 <endpoint
 url="http://servername:8700/contextroot/messagebroker/myamflongpoll"
 class="flex.messaging.endpoints.AMFEndpoint"/>
 <properties>
 <polling-enabled>true</polling-enabled>
 <polling-interval-seconds>0</polling-interval-seconds>
 <wait-interval-millis>60000</wait-interval-millis>
 <client-wait-interval-millis>3000</client-wait-interval-millis>
 <max-waiting-poll-requests>100</max-waiting-poll-requests>
 </properties>
</channel-definition>
```

Streaming

Some applications require true real-time responsiveness. Piggybacking, polling, and long polling offer varying degrees of real-time emulation, but none is a suitable choice for developing applications that must guarantee delivery of messages with no latency. Within a stock brokerage, for example, where pricing information is highly volatile and momentary fluctuations are significant, true real-time performance is required. For such applications, HTTP streaming might be a good choice, particularly where traffic must pass through DMZ-based web servers and the Internet. Streaming utilizes an HTTP connection for request and response messaging, similar to polling. This connection is opened and closed on an as-required basis. A second connection is kept open throughout the session, allowing messages to be pushed to clients as they become available. In effect, this stream allows individual data chunks to be sent at will within the body of an infinite HTTP response.

HTTP streaming works extremely well in controlled network environments such as an intranet. However, some web servers and proxies might buffer messages, rendering the stream non real time. Web servers and proxies might also disconnect the persistent connection.

Streaming also consumes a minimum of one browser connection for the duration of the session and a second connection intermittently. If only two connections are available, there could be contention for client-side access to the network. To help overcome these limitations, LiveCycle Data Services ES includes safeguards that track and limit the number of streaming connections per client. The client libraries also allow developers to configure a connection timeout. If a stream cannot be set up within the specified interval due to buffering, the application tries alternate connection models until one is successful.

The following example shows the configuration of streaming AMF over HTML.

```
<!-- AMF with streaming -->
<channel-definition id="my-amf-stream"
 class="mx.messaging.channels.StreamingAMFChannel">
 <endpoint url="http://servername:2080/myapp/messagebroker/streamingamf"
 class="flex.messaging.endpoints.StreamingAMFEndpoint"/>
</channel-definition>
```

All HTTP-based methods inherit underlying characteristics that you must be aware of when planning their applications. HTTP is not a fully duplexed protocol, requiring that simultaneous bidirectional communications be simulated. Doing so involves making additional connections between clients and the server. Because browser connections are a scarce resource, there is an

inherent risk that HTTP-based, real-time applications can exhaust connections, particularly if the user makes other use of the browser at the same time. Mitigating this risk involves additional server-side logic to track the connections open to each client. HTTP is also a stateless protocol, which imposes challenges for applications needing advanced features such as graceful handling of dropped connections. HTTP cannot guarantee consistent ordering of all client operations whenever more than one socket connection is used.

Many applications, such as the brokerage example, require authentication to ensure that only authorized users can access real-time data, and that access is limited to those resources for which they are credentialed. If you are using HTTP and conventional servlet-based I/O, you can use basic authentication in LiveCycle Data Services ES. Basic authentication relies on standard J2EE basic authentication from the application server. If you use basic authentication to secure access to destinations, you typically secure the endpoints of the channels that the destinations use via the web.xml configuration file and then configure the destination to access the secured resource to be challenged for a username (principal) and password (credentials). LiveCycle Data Services ES checks that a currently authenticated principal exists before routing any messages to the destination. If no authenticated principal exists, the server returns an HTTP 401 error message to indicate that authentication is required, and the browser prompts the user to enter a username and password. This challenge is performed by the web browser independently of the Flex client application. After users successfully log in, they remain logged in until the browser is closed.

In the case of the brokerage application in which the entire network is private and the network topology is known, you can overcome the limitations of HTTP-based messaging by adopting the RTMP. RTMP is a full-duplex TCP-based protocol and does not require multiple connections or the associated logic to safeguard against exhausting limited resources on the client. A full-duplex RTMP socket connection guarantees consistent ordering. While RTMP has many technical advantages over HTTP for meeting real-time requirements, RTMP can encounter connectivity issues in some environments because of the behavior of firewalls and proxies. RTMP is also stateful and can immediately detect dropped connections due to crashes, forced browser shutdowns, network interruptions, or other causes. You can then take appropriate actions such as attempting to reconnect using fallbacks or cleanly canceling transactions in progress such as a stock purchase.

When evaluating Java-based server options, it is important to understand the differences between the I/O options available. While RTMP offers true real-time performance, stateful operations, and simpler implementation on the client, there are still face scaling implications when conventional servlet-based I/O is employed. Servlet-based I/O is a blocking implementation and requires a thread-per-client connection, which can have implications for both memory and CPU usage as the number of concurrent connections grows, each of which consumes memory and must be scheduled. As a general rule, servlet based I/O scales to the hundreds of concurrent users, but not the thousands. New IO (NIO), released as part of J2SE 1.4, provides independent management for connections and I/O operations. As a result, NIO uses far fewer threads for a given number of connections and can scale to the thousands or tens of thousands of connections.

As shown in this example, configuring channels to use NIO endpoints is very similar to using servlet-based endpoints, but uses a different endpoint URL and class definition.

```
<channel-definition id="my-nio-amf" class="mx.messaging.channels.
AMFChannel">
 <endpoint url="http://localhost:2080/mynioamf"
 class="flex.messaging.endpoints.NIOAMFEndpoint"/>
 <server ref="my-nio-server"/>
</channel-definition>
```

If you employ RTMP and NIO in an application, you can utilize custom authentication in LiveCycle Data Services ES. In this model, the client application passes credentials to the server without relying on the browser. Although you apply security constraints to a destination, you actually log in and out of the channels associated with the destination. Therefore, to send authentication credentials to a destination that uses custom authentication, you specify a username and password as arguments to the ChannelSet.login() method and remove credentials by calling the ChannelSet.logout() method. Because multiple destinations can use the same channels and corresponding ChannelSet object, logging in to one destination also logs the user in to any other destination that uses the same channel or channels.

Recommendations for real-time architecture

You can implement a real-time or near-real-time application many different ways. Each method has its own strengths and weaknesses; therefore, it is important that you identify all your application and deployment requirements. The following recommendations will help you design your real-time architecture:

- **HTTP piggybacking and HTTP polling—**Good approaches for simple applications with modest latency and scaling requirements, and they can be used together.
- **HTTP long polling—**Good choice for applications that tolerate lower levels of latency, and if the infrastructure and client load allow dedicating a browser connection to the server.
- **HTTP streaming—**Enables true real-time responsiveness for applications in controlled environments. Drawbacks include tying up multiple browser connections to the server intermittently and sensitivity to buffering.
- **RTMP with NIO—**A full-duplex, TCP-based protocol best for highly scalable, mission-critical applications running on a private network with a known topology.

Conclusion

Web-deployed applications built with Flex and Adobe AIR technologies have the potential to deliver rich, responsive user experiences while lowering development and deployment costs. When real-time performance is needed, developers must carefully consider the application's latency tolerance, client limitations, network topology, and the server's scaling requirements to make the right design choices. LiveCycle Data Services ES supports a range of communications options, providing near- and true real-time performance over HTTP, and stateful true real-time performance over RTMP. Channels and endpoints in LiveCycle Data Services ES are fully configurable, allowing deployments to be adjusted to the limitations of local installations and supporting channel fallbacks to help ensure connectivity under adverse conditions. NIO-based connectivity on the server can scale to support thousands or tens of thousands nonblocking sessions.

LiveCycle Data Services ES messaging services greatly simplify the development of collaborative real-time applications requiring conflict resolution, client-to-client updates, and management facilities such as supervisory dashboards and alerting. By leveraging the LiveCycle ES foundation and other solution components, real-time Flex and Adobe AIR applications can be integrated into end-to-end business solutions.

Adobe Systems Incorporated
345 Park Avenue
San Jose, CA 95110-2704
USA
www.adobe.com

91009625 7/09