

Chapter 14: Measuring Message Processing Performance

As part of preparing your application for final deployment, you can test its performance to look for ways to optimize it. One place to examine performance is in the message processing part of the application. To help you gather this performance information, enable the gathering of message timing and sizing data.

Topics

About measuring message processing performance	132
Measuring message processing performance	136

About measuring message processing performance

The mechanism for measuring message processing performance is disabled by default. When enabled, information regarding message size, server processing time, and network travel time is available to the client that pushed a message to the server, to a client that received a pushed message from the server, or to a client that received an acknowledge message from the server in response a pushed message. A subset of this information is also available for access on the server.

You can use this mechanism across all channel types, including polling and streaming channels, that communicate with the LiveCycle Data Services ES server. However, this mechanism does not work when you make a direct connection to an external server by setting the `useProxy` property to `false` for the `HTTPService` and `WebService` tags because this bypasses the LiveCycle Data Services ES Proxy Server.

The [MessagePerformanceUtils](#) class defines the available message processing metrics. When a consumer receives a message, or a producer receives an acknowledge message, the consumer or producer extracts the metrics into an instance of the `MessagePerformanceUtils` class, and then accesses the metrics as properties of that class. For a complete list of the available metrics, see [“Available message processing metrics” on page 134](#).

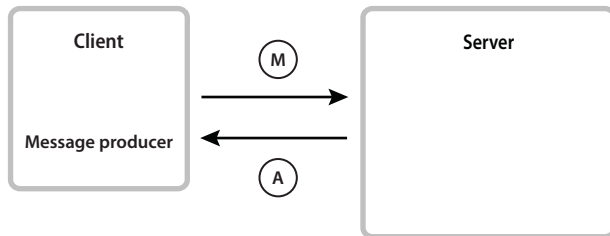
Measuring performance for different channel types

The types of metrics that are available and their calculations, depend on the channel configuration over which a message is sent from or received by the client.

Producer-acknowledge scenario

In the producer-acknowledge scenario, a producer sends a message to a server over a specific channel. The server then sends an acknowledge message back to the producer.

The following image shows the producer-acknowledge scenario:



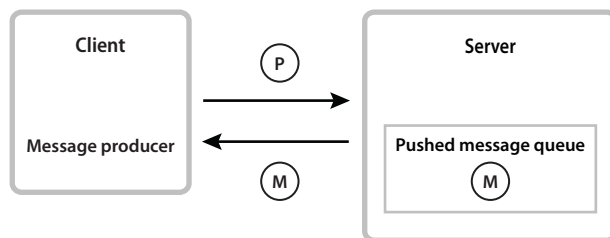
M. Message sent to server. A. Acknowledge message.

If you enable the gathering of message processing metrics, the producer adds information to the message before sending it to the server, such as the send time and message size. The server copies the information from the message to the acknowledge message. Then the server adds additional information to the acknowledge message, such as the response message size and server processing time. When the producer receives the acknowledge message, it uses all of the information in the message to calculate the metrics defined by the `MessagePerformanceUtils` class.

Message-polling scenario

In a message-polling scenario, a consumer polls a message channel to determine if a message is available on the server. On receiving the polling message, the server pushes any available message to the consumer.

The following image shows the polling scenario:



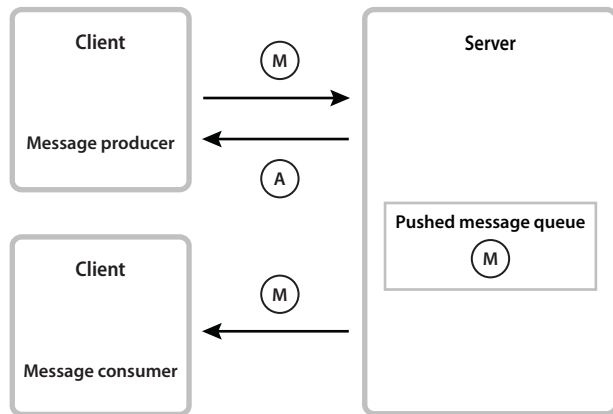
P. Polling message sent. M. Message pushed to client from server.

If you enable the gathering of message processing metrics in this scenario, the consumer obtains performance metrics about the poll-response transaction, such as the response message size and server processing time. The metrics also include information about the message returned by the server. This information lets the consumer determine how long the message was waiting before it was pushed. However, the metric information does not identify the client that originally pushed the message onto the server.

Message-streaming scenario

In the streaming scenario, the server pushes a message to a consumer when a message is available; the consumer itself does not initiate the transaction.

The following image shows this scenario:



M. Message sent to server, and then pushed to client. A. Acknowledge message.

In this scenario, the message producer pushes a message, and then receives an acknowledge message. The producer can obtain metric information as described in [“Producer-acknowledge scenario” on page 132](#).

When the server pushes the message to the consumer, the message contains information from the original message from the producer, and the metric information that the server added. The consumer can then examine the metric data, including the time from when the producer pushed the message until the consumer received it.

Measuring message processing performance for streaming and RTMP channels

For streaming and RTMP channels, a pushed message is sent to a consumer without the client first sending a polling message. In this case, the following metrics are not available to the consumer for the pushed message, but instead are set to 0:

- `networkRTT`
- `serverPollDelay`
- `totalTime`

Available message processing metrics

The following table lists the available message processing metrics defined by the [MessagePerformanceUtils](#) class:

Property	Description
<code>clientReceiveTime</code>	The number of milliseconds since the start of the UNIX epoch, January 1, 1970, 00:00:00 GMT, to when the client received response message from the server.
<code>messageSize</code>	The size of the original client message, in bytes, as measured during deserialization by the server endpoint.
<code>networkRTT</code>	<p>The duration, in milliseconds, from when a client sent a message to the server until it received a response, excluding the server processing time. This value is calculated as <code>totalTime - serverProcessingTime</code>.</p> <p>If a pushed message is using a streaming or RTMP channel, the metric is meaningless because the client does not initiate the pushed message; the server sends a message to the client whenever a message is available. Therefore, for a message pushed over a streaming or RTMP channel, this value is 0. However, for an acknowledge message sent over a streaming or RTMP channel, the metric contains a valid number.</p>

Property	Description
<code>originatingMessageSentTime</code>	The timestamp, in milliseconds since the start of the UNIX epoch on January 1, 1970, 00:00:00 GMT, to when the client that caused a push message sent its message. Only populated for a pushed message, but not for an acknowledge message.
<code>originatingMessageSize</code>	Size, in bytes, of the message that originally caused this pushed message. Only populated for a pushed message, but not for an acknowledge message.
<code>pushedMessageFlag</code>	Contains <code>true</code> if the message was pushed to the client but is not a response to a message that originated on the client. For example, when the client polls the server for a message, <code>pushedMessageFlag</code> is <code>false</code> . When you are using a streaming channel, <code>pushedMessageFlag</code> is <code>true</code> . For an acknowledge message, <code>pushedMessageFlag</code> is <code>false</code> .
<code>pushOneWayTime</code>	Time, in milliseconds, from when the server pushed the message until the client received it. Note: This value is only relevant if the server and receiving client have synchronized clocks. Only populated for a pushed message, but not for an acknowledge message.
<code>responseMessageSize</code>	The size, in bytes, of the response message sent to the client by the server as measured during serialization at the server endpoint.
<code>serverAdapterExternalTime</code>	Time, in milliseconds, spent in a module invoked from the adapter associated with the destination for this message, before either the response to the message was ready or the message had been prepared to be pushed to the receiving client. This corresponds to the time that the message was processed by LiveCycle Data Services ES.
<code>serverAdapterTime</code>	Processing time, in milliseconds, of the message by the adapter associated with the destination before the response to the message was ready or the message was prepared to be pushed to the receiving client. The processing time corresponds to the time that your code on the server processed the message, not when LiveCycle Data Services ES processed the message.
<code>serverNonAdapterTime</code>	Server processing time spent outside the adapter associated with the destination of this message. Calculated as <code>serverProcessingTime - serverAdapterTime</code> .
<code>serverPollDelay</code>	Time, in milliseconds, that this message sat on the server after it was ready to be pushed to the client but before it was picked up by a poll request. For a streaming or RTMP channel, this value is always 0.
<code>serverPrePushTime</code>	Time, in milliseconds, between the server receiving the client message and the server beginning to push the message out to other clients.
<code>serverProcessingTime</code>	Time, in milliseconds, between server receiving the client message and either the time the server responded to the received message or has the pushed message ready to be sent to a receiving client. For example, in the producer-acknowledge scenario, this is the time from when the server receives the message and sends the acknowledge message back to the producer. In a polling scenario, it is the time between the arrival of the consumer's polling message and any message returned in response to the poll. For more information on these scenarios, see .
<code>serverSendTime</code>	The number of milliseconds since the start of the UNIX epoch, January 1, 1970, 00:00:00 GMT, to when the server sent a response message back to the client.
<code>totalPushTime</code>	Time, in milliseconds, from when the originating client sent a message and the time that the receiving client received the pushed message. Note: This value is only relevant if the two clients have synchronized clocks. Only populated for a pushed message, but not for an acknowledge message.
<code>totalTime</code>	Time, in milliseconds, between this client sending a message and receiving a response from the server. This property contains 0 for a streaming or RTMP channel.

Considerations when measuring message processing performance

The mechanism that measures message processing performance attempts to minimize the overhead required to collect information so that all timing information is as accurate as possible. However, there are several considerations to take into account when you use this mechanism.

Synchronize the clocks on different computers

The metrics defined by the `MessagePerformanceUtils` class include the `totalPushTime`. The `totalPushTime` is a measure of the time from when the originating message producer sent a message until a consumer receives the message. This value is determined from the timestamp added to the message when the producer sends the message, and the timestamp added to the message when the consumer receives the message. However, to calculate a valid value for the `totalPushTime` metric, the clocks on the message-producing computer and on the message-consuming computer must be synchronized.

Another metric, `pushOneWayTime`, contains the time from when the server pushed the message until the consumer received it. This value is determined from the timestamp added to the message when the server sends the message, and the timestamp added to the message when the consumer receives the message. To calculate a valid value for the `pushOneWayTime` metric, the clocks on the message consuming computer and on the server must be synchronized.

One option is to perform your testing in a lab environment where you can ensure that the clocks on all computers are synchronized. For the `totalPushTime` metric, you can ensure that the clocks for the producer and a consumer applications are synchronized by running the applications on the same computer. Or, for the `pushOneWayTime` metric, you can run the consumer application and server on the same computer.

Perform different tests for message timing and sizing

The mechanism for measuring message processing performance lets you enable the tracking of timing information, of sizing information, or both. The gathering of timing-only metrics is minimally intrusive to your overall application performance. The gathering of sizing metrics involves more overhead time than gathering timing information.

Therefore, you can run your tests twice: once for gathering timing information, and once for gathering sizing information. In this way, the timing-only test can eliminate any delays caused by calculating message size. You can then combine the information from the two tests to determine your final results.

Measuring message processing performance

The mechanism for measuring message processing performance is disabled by default. When you enable it, you can use the `MessagePerformanceUtils` class to access the metrics from a message received by a client.

Enabling message processing metrics

You use two parameters in a channel definition to enable message processing metrics:

- `<record-message-times>`
- `<record-message-sizes>`

Set these parameters to `true` or `false`; the default value is `false`. You can set the parameters to different values to capture only one type of metric. For example, the following channel definition specifies to capture message timing information, but not message sizing information:

```
<channel-definition id="my-streaming-amf"  
  class="mx.messaging.channels.StreamingAMFChannel">
```

```

<endpoint
  url="http://{server.name}:{server.port}/{context.root}/messagebroker/streamingamf"
  class="flex.messaging.endpoints.StreamingAMFEndpoint"/>
<properties>
  <record-message-times>true</record-message-times>
  <record-message-sizes>>false</record-message-sizes>
</properties>
</channel-definition>

```

Using the MessagePerformanceUtils class

The [MessagePerformanceUtils](#) class is a client-side class that you use to access the message processing metrics. You create an instance of the MessagePerformanceUtils class from a message pushed to the client by the server or from an acknowledge message.

The following example shows a message producer that uses the acknowledge message to display in a TextArea control the metrics for a message pushed to the server:

```

<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

  <mx:Script>
    <![CDATA[

      import mx.messaging.messages.AsyncMessage;
      import mx.messaging.messages.IMessage;
      import mx.messaging.events.MessageEvent;
      import mx.messaging.messages.MessagePerformanceUtils;

      // Event handler to send the message to the server.
      private function send():void
      {
        var message:IMessage = new AsyncMessage();
        message.body.chatMessage = msg.text;
        producer.send(message);
        msg.text = "";
      }

      // Event handler to write metrics to the TextArea control.
      private function ackHandler(event:MessageEvent):void {
        var mpiutil:MessagePerformanceUtils =
          new MessagePerformanceUtils(event.message);
        myTAAck.text = "totalTime = " + String(mpiutil.totalTime);
        myTAAck.text = myTAAck.text + "\n" + "messageSize= " +
          String(mpiutil.messageSize);
      }

    ]]>
  </mx:Script>

  <mx:Producer id="producer" destination="chat" acknowledge="ackHandler(event)"/>

  <mx:Label text="Acknowledge metrics"/>
  <mx:TextArea id="myTAAck" width="100%" height="20%"/>

  <mx:Panel title="Chat" width="100%" height="100%">
    <mx:TextArea id="log" width="100%" height="100%"/>
    <mx:ControlBar>
      <mx:TextInput id="msg" width="100%" enter="send()"/>
      <mx:Button label="Send" click="send()"/>
    </mx:ControlBar>

```

```
</mx:Panel>
```

```
</mx:Application>
```

In this example, you write an event handler for the `acknowledge` event to display the metrics. The event handler extracts the metric information from the `acknowledge` message, and then displays the `MessagePerformanceUtils.totalTime` and `MessagePerformanceUtils.messageSize` metrics in a `TextArea` control.

You can also use the `MessagePerformanceUtils.prettyPrint()` method to display the metrics. The `prettyPrint()` method returns a formatted `String` that contains nonzero and non-null metrics. The following example modifies the event handler for the previous example to use the `prettyPrint()` method:

```
// Event handler to write metrics to the TextArea control.
private function ackHandler(event:MessageEvent):void {
    var mpiutil:MessagePerformanceUtils = new MessagePerformanceUtils(event.message);
    myTAAck.text = mpiutil.prettyPrint();
}
```

The following example shows the output from the `prettyPrint()` method that appears in the `TextArea` control:

```
Original message size(B): 509
Response message size(B): 562
Total time (s): 0.016
Network Roundtrip time (s): 0.016
```

A message consumer can write an event handler for the `message` event to display metrics, as the following example shows:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    creationComplete="consumer.subscribe();">

    <mx:Script>
        <![CDATA[

            import mx.messaging.messages.AsyncMessage;
            import mx.messaging.messages.IMessage;
            import mx.messaging.events.MessageEvent;
            import mx.messaging.messages.MessagePerformanceUtils;

            // Event handler to send the message to the server.
            private function send():void
            {
                var message:IMessage = new AsyncMessage();
                message.body.chatMessage = msg.text;
                producer.send(message);
                msg.text = "";
            }

            // Event handler to write metrics to the TextArea control.
            private function ackHandler(event:MessageEvent):void {
                var mpiutil:MessagePerformanceUtils =
                    new MessagePerformanceUtils(event.message);
                myTAAck.text = mpiutil.prettyPrint();
            }

            // Event handler to write metrics to the TextArea control for the message consumer.
            private function messageHandler(event:MessageEvent):void {
                var mpiutil:MessagePerformanceUtils =
                    new MessagePerformanceUtils(event.message);
                myTAMess.text = mpiutil.prettyPrint();
            }
        ]]>
```

```

    }
  ]]>
</mx:Script>

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

<mx:Label text="ack metrics"/>
<mx:TextArea id="myTAAck" width="100%" height="20%" text="ack"/>

<mx:Label text="receive metrics"/>
<mx:TextArea id="myTAMess" width="100%" height="20%" text="rec"/>

<mx:Panel title="Chat" width="100%" height="100%">
  <mx:TextArea id="log" width="100%" height="100%" />
  <mx:ControlBar>
    <mx:TextInput id="msg" width="100%" enter="send()" />
    <mx:Button label="Send" click="send()" />
  </mx:ControlBar>
</mx:Panel>

```

```
</mx:Application>
```

In this example, you use the `prettyPrint()` method to write the metrics for the received message to a `TextArea` control. The following example shows this output:

```

Response message size(B): 560
PUSHED MESSAGE INFORMATION:
Total push time (s): 0.016
Push one way time (s): 0.016
Originating Message size (B): 509

```

You can gather metrics for `HTTPService` and `WebService` tags when they use the `Proxy Service`, as defined by setting the `useProxy` property to `true` for the `HTTPService` and `WebService` tags. The following example gathers metrics for an `HTTPService` tag:

```

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

  <mx:Script>
    <![CDATA[

      import mx.messaging.events.MessageEvent;
      import mx.messaging.messages.MessagePerformanceUtils;

      // Event handler to write metrics to the TextArea control for the message consumer.
      private function messageHandler(event:MessageEvent):void {
        var mpiutil:MessagePerformanceUtils =
          new MessagePerformanceUtils(event.message);
        myTAMess.text = mpiutil.prettyPrint();
      }
    ]]>
  </mx:Script>

  <mx:Label text="Message metrics"/>
  <mx:TextArea id="myTAMess" width="100%" height="20%" />

  <mx:HTTPService id="srv" destination="catalog"
    useProxy="true"
    result="messageHandler(event);"/>

  <mx:DataGrid dataProvider="{srv.lastResult.catalog.product}"

```

```

        width="100%" height="100%"/>

        <mx:Button label="Get Data" click="srv.send()" />
    </mx:Application>

```

When using LiveCycle Data Services ES Data Management Service, you can use event handlers on the `DataService` class to handle metrics, as the following example shows:

```

<mx:Script>
    <![CDATA[

        import mx.messaging.events.MessageEvent;
        import mx.messaging.messages.MessagePerformanceUtils;

        // Event handler to write metrics to the TextArea control.
        private function messageHandler(event:MessageEvent):void {
            var mpiutil:MessagePerformanceUtils = new MessagePerformanceUtils(event.message);
            myTAMess.text = mpiutil.prettyPrint();
        }
    ]]>
</mx:Script>

<mx:DataService id="ds" destination="inventory" result="messageHandler(event);"/>

<mx:Label text="receive metrics"/>
<mx:TextArea id="myTAMess" width="100%" height="20%" text="rec"/>

<mx:Button label="Get Data" click="ds.fill(products)"/>

```

Using the server-side classes to gather metrics

For managed endpoints, you can access the total number of bytes serialized and deserialized by using the following methods of the `flex.management.runtime.messaging.endpoints.EndpointControlMBean` interface:

- `getBytesDeserialized()`
Returns the total number of bytes that were deserialized by this endpoint during its lifetime.
- `getBytesSerialized()`
Returns the total number of bytes that were serialized by this endpoint during its lifetime.

The `flex.management.runtime.messaging.endpoints.EndpointControlMBean` class implements these methods.

Writing messaging metrics to the log files

You can write messaging metrics to the client-side log file if you enable the metrics. To enable the metrics, set the `<record-message-times>` or `<record-message-sizes>` parameter to `true`, and the client-side log level to `DEBUG`. Messages are written to the log when a client receives an acknowledgment for a pushed message, or a client receives a pushed message from the server. The metric information appears immediately following the debug information for the received message.

The following example initializes logging and sets the log level to `DEBUG`:

```

<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    creationComplete="consumer.subscribe();initLogging();">

    <mx:Script>
        <![CDATA[

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

```

```
import mx.messaging.events.MessageEvent;
import mx.messaging.messages.MessagePerformanceUtils;
import mx.controls.Alert;
import mx.collections.ArrayCollection;
import mx.logging.targets.*;
import mx.logging.*;

// Event handler to send the message to the server.
private function send():void
{
    var message:IMessage = new AsyncMessage();
    message.body.chatMessage = msg.text;
    producer.send(message);
    msg.text = "";
}

// Event handler to write metrics to the TextArea control.
private function ackHandler(event:MessageEvent):void {
    var mpiutil:MessagePerformanceUtils =
        new MessagePerformanceUtils(event.message);
    myTAAck.text = mpiutil.prettyPrint();
}

// Event handler to write metrics to the TextArea control for the message consumer.
private function messageHandler(event:MessageEvent):void {
    var mpiutil:MessagePerformanceUtils =
        new MessagePerformanceUtils(event.message);
    myTAMess.text = mpiutil.prettyPrint();
}

// Initialize logging and set the log level to DEBUG.
private function initLogging():void {
    // Create a target.
    var logTarget:TraceTarget = new TraceTarget();

    // Log all log levels.
    logTarget.level = LogEventLevel.DEBUG;

    // Add date, time, category, and log level to the output.
    logTarget.includeDate = true;
    logTarget.includeTime = true;
    logTarget.includeCategory = true;
    logTarget.includeLevel = true;

    // Begin logging.
    Log.addTarget(logTarget);
}
]]>
</mx:Script>

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

<mx:Label text="ack metrics"/>
<mx:TextArea id="myTAAck" width="100%" height="20%" text="ack"/>

<mx:Label text="receive metrics"/>
<mx:TextArea id="myTAMess" width="100%" height="20%" text="rec"/>

<mx:Panel title="Chat" width="100%" height="100%">
    <mx:TextArea id="log" width="100%" height="100%"/>
</mx:Panel>
```

```

    <mx:ControlBar>
        <mx:TextInput id="msg" width="100%" enter="send()" />
        <mx:Button label="Send" click="send()" />
    </mx:ControlBar>
</mx:Panel>
</mx:Application>

```

For more information on logging, see *Building and Deploying Adobe Flex 3 Applications*.

By default, on Microsoft Windows the log file is written to the file C:\Documents and Settings\USERNAME\Application Data\Macromedia\Flash Player\Logs\flashlog.txt. The following excerpt is from the log file for the message "My test message":

```

2/14/2008 11:20:18.806 [DEBUG] mx.messaging.Channel 'my-rtmp' channel got connect attempt
status. (Object)#0
    code = "NetConnection.Connect.Success"
    description = "Connection succeeded."
    details = (null)
    DSMessagingVersion = 1
    id = "D46A822C-962B-4651-6F2A-DCB41130C4CF"
    level = "status"
    objectEncoding = 3
2/14/2008 11:20:18.837 [INFO] mx.messaging.Channel 'my-rtmp' channel is connected.
2/14/2008 11:20:18.837 [DEBUG] mx.messaging.Channel 'my-rtmp' channel sending message:
(mx.messaging.messages::CommandMessage)
    body=(Object)#0
    clientId=(null)
    correlationId=""
    destination="chat"
    headers=(Object)#0
    messageId="E2F6B35E-42CD-F088-4B7A-18BF1515F142"
    operation="subscribe"
    timeToLive=0
    timestamp=0
2/14/2008 11:20:18.868 [INFO] mx.messaging.Consumer 'consumer' consumer connected.
2/14/2008 11:20:18.868 [INFO] mx.messaging.Consumer 'consumer' consumer acknowledge for
subscribe. Client id 'D46A82C3-F419-0EBF-E2C8-330F83036D38' new timestamp 1203006018867
2/14/2008 11:20:18.884 [INFO] mx.messaging.Consumer 'consumer' consumer acknowledge of
'E2F6B35E-42CD-F088-4B7A-18BF1515F142'.
2/14/2008 11:20:18.884 [DEBUG] mx.messaging.Consumer Original message size(B): 626
Response message size(B): 562

2/14/2008 11:20:25.446 [INFO] mx.messaging.Producer 'producer' producer sending message
'CF89F532-A13D-888D-D929-18BF2EE61945'
2/14/2008 11:20:25.462 [INFO] mx.messaging.Producer 'producer' producer connected.
2/14/2008 11:20:25.477 [DEBUG] mx.messaging.Channel 'my-rtmp' channel sending message:
(mx.messaging.messages::AsyncMessage)#0
    body = (Object)#1
        chatMessage = "My test message"
    clientId = (null)
    correlationId = ""
    destination = "chat"
    headers = (Object)#2
    messageId = "CF89F532-A13D-888D-D929-18BF2EE61945"
    timestamp = 0
    timeToLive = 0
2/14/2008 11:20:25.571 [DEBUG] mx.messaging.Channel 'my-rtmp' channel got message
(mx.messaging.messages::AsyncMessageExt)#0
    body = (Object)#1
        chatMessage = "My test message"
    clientId = "D46A82C3-F419-0EBF-E2C8-330F83036D38"
    correlationId = ""

```

```
destination = "chat"
headers = (Object)#2
DSMPPIO = (mx.messaging.messages::MessagePerformanceInfo)#3
  infoType = "OUT"
  messageSize = 575
  overheadTime = 0
  pushedFlag = true
  receiveTime = 1203006025556
  recordMessageSizes = false
  recordMessageTimes = false
  sendTime = 1203006025556
  serverPostAdapterExternalTime = 0
  serverPostAdapterTime = 0
  serverPreAdapterExternalTime = 0
  serverPreAdapterTime = 0
  serverPrePushTime = 0
DSMPPIP = (mx.messaging.messages::MessagePerformanceInfo)#4
  infoType = (null)
  messageSize = 506
  overheadTime = 0
  pushedFlag = false
  receiveTime = 1203006025556
  recordMessageSizes = true
  recordMessageTimes = true
  sendTime = 1203006025556
  serverPostAdapterExternalTime = 1203006025556
  serverPostAdapterTime = 1203006025556
  serverPreAdapterExternalTime = 0
  serverPreAdapterTime = 1203006025556
  serverPrePushTime = 1203006025556
messageId = "CF89F532-A13D-888D-D929-18BF2EE61945"
timestamp = 1203006025556
timeToLive = 0
```

```
2/14/2008 11:20:25.696 [DEBUG] mx.messaging.Channel Response message size(B): 575
PUSHED MESSAGE INFORMATION:
Originating Message size (B): 506
```

```
2/14/2008 11:20:25.712 [INFO] mx.messaging.Producer 'producer' producer acknowledge of
'CF89F532-A13D-888D-D929-18BF2EE61945'.
2/14/2008 11:20:25.712 [DEBUG] mx.messaging.Producer Original message size(B): 506
Response message size(B): 562
Total time (s): 0.156
Network Roundtrip time (s): 0.156
```