Institutional Client Connectivity : Configuring LCDS Software Clustering

This page last changed on Apr 13, 2007 by pattonne.

This page describes the steps necessary to configure software clustering for an LCDS application. It also includes some general advice and known issues on the subject. A higher-level overview of LCDS clustering is provided on the Overview of LCDS Clustering page.

Introduction

The *licenced version* of LCDS includes a software clustering feature that can provide fail-over and message-state replication. This means that server nodes can fail without disrupting user experience, and messages sent by one client can be received by others, regardless of the node they are connected to.

The first time a client service component connects to a clustered destination, it receives a set of destinations for the nodes of the cluster. These are used for fail-over purposes, so if one destination becomes unavailable, the service component will automatically start to use another destination.

Flex software clustering does not provide load-balancing, but can be used in conjunction with software and hardware load-balancing solutions. An LCDS application may be deployed to a cluster of Tomcat nodes, fronted by an Apache web server, which uses the mod_jk module for load-balancing purposes.

There are known bugs in the clustering implementation of LCDS 2.5 beta 2 which restrict the channels and service that can be used, but these have been resolved in the latest release candidate build, so all channel and service types should be supported in the production release.

How to apply LCDS Software Clustering

The following steps are involved:

1. Configure the cluster management
2. Define the Flex cluster
3. Revise the channel endpoints
4. Reference the cluster from service destinations
5. Deploy the application

These are now described in more detail, using the example of a message service deployed to a Tomcat cluster. It is presumed that the Tomcat cluster nodes have already been configured for standard LCDS services by installing a transaction manager such as JOTM. It is also assumed that a valid LCDS licence is available, since the software clustering functionality is disabled in the express version.

An example web application, containing the message service used in this article and an accompanying client (`MessagingExample.mxml`), is available from Neale Patton.

1. Configure the cluster management

Flex makes use of the JGroups communication toolkit for implementing its software clustering feature. This is a reliable and configurable open-source library which is included in the LCDS distribution. The library runs in conjunction with an XML configuration file.

Copy the JGroups JAR from the LCDS distribution into your web application:

- `LCDS/resources/clustering/jgroups.jar` to `APPLICATION/WEB-INF/lib`

A JGroups cluster is configured using an XML *Protocol Stack Configuration* file. The LCDS distribution includes two sample files. Copy one of the samples into your web application:

- `LCDS/resources/clustering/jgroups-tcp.xml` to `APPLICATION/WEB-INF/flex`

The configuration file is daunting at first, but the various elements are quite well documented on the JGroups Wiki page. Basically, there is a root `<config>` element, under which sit a sequence of different protocol elements comprising the stack. A protocol element configures a specific aspect of the cluster, such as the means of communication between the nodes, the way that node discovery takes place, or the strategy for detecting node failure. An annotated configuration file is given below:

```
<!-- JGroups Protocol Stack Configuration                                    -->
<!-- This file configures the JGroup used for LCDS clustering.               -->
<!-- See http://wiki.jboss.org/wiki/Wiki.jsp?page=JGroups for configuration doc. -->
<config>

    <!-- Use a mesh of TCP connections to send messages between group members. -->
    <TCP
        start_port="8070"
        loopback="true"/>

    <!-- Discovers the initial group membership by directly contacting specific hosts. -->
    <TCPPING
        initial_hosts="host1.somewhere.com[~sugdento:8080],host2.somewhere.com[~sugdento:8080],host3.somewhe
        timeout="3000"
        port_range="3"
        num_initial_members="3"/>

    <!-- Failure detection based on heartbeat messages. -->
    <FD
        timeout="2000"
        max_tries="4"/>

    <!-- Verifies that suspected members are really dead. -->
    <VERIFY_SUSPECT
        timeout="1500"
        down_thread="false"
        up_thread="false"/>

    <!-- Perform loss-less and FIFO delivery of multicast messages using negative
acknowledgments. -->
    <!-- E.g. when receiving P:1, P:3, P:4, a receiver asks P to retransmit the missing message
2. -->
    <pbcast.NAKACK
        gc_lag="100"
        retransmit_timeout="600,1200,2400,4800"/>

    <!-- Garbage collect messages that have been seen by all members of the group. -->
    <pbcast.STABLE
        stability_delay="1000"
        desired_avg_gossip="20000"
        down_thread="false"
        max_bytes="0"
        up_thread="false"/>

    <!-- The Group Membership Service which is responsible for joining/leaving members, -->
    <!-- handling suspected members and sending views (topology information) to all     -->
```

```
    <!-- members whenever membership changes.                                    -->
    <pbcast.GMS
        print_local_addr="true"
        join_timeout="5000"
        join_retry_timeout="2000"
        shun="true"/>

</config>
```

For an initial configuration, copy the above contents into your own configuration file. Note that the `TCPPING` discovery protocol is being used, which requires a set of initial host names and port numbers for the nodes of the cluster. You will need to alter these settings to suit your own environment. Several alternative means of cluster discovery are available, as described on the JGroups Wiki.

When the servers on each node of a cluster are started, they will communicate with one another to establish the topology of the cluster using the chosen discovery protocol. When a Flex client is loaded, it will receive the URLs for the nodes of the cluster, and these will then be used to achieve fail-over, if necessary.

2. Define the Flex cluster

A Flex cluster must be defined in the `services-config.xml` file of your application using the `<clusters>` element. This must contain at least one `<cluster>` element that has an `id` and references the JGroups protocol stack configuration file for the cluster. The `<clusters>` element is positioned at the same level as the `<channels>` element, as shown in the example below:

```
...
<clusters>
    <cluster id="my-cluster" properties="jgroups-tcp.xml"/>
</clusters>

<channels>
...
```

3. Revise the channel endpoints

It may be necessary to revise the channel endpoint definitions within the `services-config.xml` of your application. The `server.name` and `server.port` tokens cannot be used in any channel endpoints that are involved in a cluster. Direct URLs must be used instead. This ensures that each server has an absolute and unique URL which can be used to address connections directly to it. Shown below are some example channels:

```
<channel-definition id="my-rtmp" class="mx.messaging.channels.RTMPChannel">
    <endpoint url="rtmp://localhost:2038" class="flex.messaging.endpoints.RTMPEndpoint"/>
    <properties>
        <idle-timeout-minutes>20</idle-timeout-minutes>
        <!-- for deployment on WebSphere, must be mapped to a WorkManager available in
             the web application's jndi context.
        <websphere-workmanager-jndi-name>java:comp/env/wm/MessagingWorkManager</websphere-workmanager-jndi-n
        -->
    </properties>
</channel-definition>
```

Note that there are known bugs in the clustering implementation of LCDS 2.5 beta 2 for AMF and HTTP channels, but RTMP channels seems to work fine. These bugs have been resolved in the current release candidate build, so all channel types should be supported in the production release.

4. Reference the cluster from service destinations

The service destinations that should be clustered must now be modified to reference the cluster we have defined. This involves the addition of a `<cluster>` element within the `<network>` property element of the destination definition. Shown below is an example messaging service destination from a `messaging-config.xml` file:

```
<destination id="my-message-destination">
    <properties>
        <network>
            <session-timeout>0</session-timeout>
            <throttle-inbound policy="ERROR" max-frequency="50"/>
            <throttle-outbound policy="REPLACE" max-frequency="500"/>
            <cluster ref="my-cluster" /> <!-- shared-backend="false" -->
        </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>
        </server>
    </properties>
    <channels>
        <channel ref="my-rtmp"/>
    </channels>
</destination>
```

The value of the `ref` attribute must match the `id` of the cluster that was defined in the `services-config.xml` file.

There is also an optional `shared-backend` attribute that is applicable only to data management service destinations. This specifies whether or not a common back-end is being used for storing state, and determines whether or not messages should be reprocessed by other nodes of a cluster. Note that when a message is sent to one node, it is processed by that node then broadcast to other nodes. If this is set to `true` the other nodes will not reprocess replicated messages, if it is set to `false` they will reprocess replicated messages.

Deploy the application

The web application is now ready to be deployed. This can be achieved by manually copying it into the `webapps` folders of each of your Tomcat nodes, though more sophisticated deployment methods may be available. After deployment, it may be necessary to adjust the channel endpoint definitions further, depending on the host names and ports of your cluster nodes.

You should now be able to load your application from any cluster node, and continue to use it as long as there is one cluster node still standing. Test the clustering fail-over by opening a number of clients,

loaded from different nodes, then killing the Tomcat servers one by one and continuing to use the clients.

Further Advice

1. Adjust the Logging Level

When working with LCDS it is not always easy to determine the cause of any problems. Altering the level of logging can help to diagnose problems. To do this search for `level=` in the `services-config.xml` file and change the attribute value to `Debug`. Also ensure that the `Service.*` and `Message.*` patterns are set within the the `<filters>` element. An example logging configuration is shown below:

```
<logging>
    <target class="flex.messaging.log.ConsoleTarget" level="Debug">
        <properties>
            <prefix>[~sugdento:Flex] </prefix>
            <includeDate>false</includeDate>
            <includeTime>false</includeTime>
            <includeLevel>false</includeLevel>
            <includeCategory>false</includeCategory>
        </properties>
        <filters>
           <pattern>Configuration</pattern>
           <pattern>DataService.*</pattern>
           <pattern>Endpoint.*</pattern>
           <pattern>Message.*</pattern>
           <pattern>Service.*</pattern>
        </filters>
    </target>
</logging>
```

This should cause LCDS to log more detailed messages to the console or the `Tomcat/logs/catalina.out` log file, depending on your Tomcat configuration. This level of logging is not recommended for production environments due to its performance implications.

2. Handle Client-side Faults

To state the obvious: be sure to handle fault events raised by the various data service client components. The fault events contain messages that may help to diagnose any problems.

Common Problems

1. Unable to broadcast a replicated service operation

If you see the message "*unable to broadcast a replicated service operation*" when using a data management service destination, this may be caused by remote Java objects that do not implement the `java.io.Serializable` interface. Under a non-clustered deployment this is not necessary, but when clustering is used these object are transmitted between nodes and the Java serialization mechanism is used. For this reason, any remote objects used by a clustered data management service destination must implement `java.io.Serializable`.

2. Channel.Connect.Failed error

This occurs when tokens have been used in the channel endpoint URL definitions of your `services-config.xml` file. In order for software clustering to work, these must be replace with direct URL values, as described in the Revise the channel endpoints section.

3. Client fault when connecting to a channel

This may be due to known bugs in the LCDS 2.5 beta 2 release to do with AMF and HTTP channels. These have been resolved in the latest release candidate build, so all channel and service types should be supported in the production release. In the meantime, stick to the RTMP channel and message or data management services.

4. Clustering for destination 'somewhere' is disabled

This error occurs if there is an attempt to use clustering with the express version of LCDS. A full licence is required to use the clustering functionality. To resolve this, obtain a licence number and specify it using the `fds` property of the `WEB-INF/flex/licence.properties` file.