CHAPTER 19

Using Event Services Software

Event Services Software (ESS) is an asynchronous messaging service that provides a messaging infrastructure based on a publish-and-subscribe paradigm. It enables distributed, loosely coupled interprocess communications.

ESS is one of two event messaging components supplied with CWCS. The other – the Event Distribution System (EDS) – is a means for sending messages from one process to another in a distributed, networked environment. *These two components are disjoint systems and do not work together.*

Note ESS is the preferred messaging service. EDS (see Chapter 20, "Using the Event Distribution System") has been deprecated, and support for it will be withdrawn in a future CWCS release. Cisco urges developers to avoid new development with EDS and begin using ESS as soon as possible.

The following topics explain ESS and how to use it:

Understanding ESS Subsystems

The two main subsystems of ESS are Tibco's Rendezvous and the Lightweight Messaging Service (LWMS). Both contain implementations of the Java Messaging Service (JMS) API.

For information on using Tibco's Rendezvous, see the "Using Tibco's Rendezvous" section on page 19-2. For information on using LWMS, see the "Using the Lightweight Messaging Service" section on page 19-8 .

How Does ESS Work?

Messages are asynchronous requests, reports, or events that contain precisely formatted data that describe specific actions, queries, or declarations. Through the exchange of these messages, applications with distributed processes in a networked environment can coordinate with and track the progress of other applications in the network. Messaging services are synonymous with event services.

Messages are published to subjects. An *event bus* is the logical channel that carries these messages. These messages are sent with a specified message format:

$Subject_{TIBCO} = Topic_{JMS} = Mailbox_{LWMS}$

Subjects are essentially logical message addresses that provide a virtual connection between distributed processes. Subscribers register listeners on subjects and receive asynchronous callbacks as messages arrive. Publish/subscribe messaging is considered loosely coupled because publishers and subscribers may be anonymous and the data to be shared between the distributed processes is formatted into messages that provide a type of buffering mechanism.

A publisher simply pushes a message to a subject name and does not generally know or care who or how many subscribers receive its message. Similarly a subscriber generally does not know what process or processes are sending messages on a given subject. This architecture allows a great deal of flexibility for future changes since it makes few assumptions on who, where, when or how another process will use an event.

Developers need to create unique subject names. In order to decide what their project's namespace should be, you need to know the current recommendation which provides a segregation of subject names so you can independently decide the subject names for your applications without colliding with different application subject names. For recommendations and more information on subject names, see Next Generation Foundation Event Services (ESS) Developer's Guide (ENG-112035).

How Is ESS Organized?

The ESS architecture is divided into two main subsystems:

- A Server Event Bus (SEB) is used between backend processes that perform the actual network management computing functions and is implemented with Tibco Software Inc.'s Rendezvous product.
- A Client Event Bus (CEB) is used for messaging with CiscoWorks web client front ends and is implemented by the internally developed LWMS.

A gateway between the SEB and CEB transparently and automatically forwards messages between the two event buses.

If you need additional information on these ESS subsystems, see Event Services Software (ESS) Primer and Overview in EDCS (ENG-110601).

Using Tibco's Rendezvous

Application developers must use Tibco's Rendezvous product to communicate between backend processes such as your application's processes and other interested processes. The TibcoRV version 7.1 is used for this release of CiscoWorks Common Services.

For more information, see:

- Event Services Software (ESS) Primer and Overview in EDCS (ENG-110601)
- Next Generation Foundation Event Services (ESS) Developer's Guide (ENG-112035), Section 7, sample code for Rendezvous and LWMS
- TB/RV Overview (ENG-110810)

About Subject Names and Event Formats

This section contains the following topics:

How Subject Names are Structured

This section explains the general structure of Subject names. For specific recommendations, see the "Choosing Subject Names and Namespaces" section on page 19-3.

Each subject name is a dotted string. For example, *cisco.mgmt.vms.vhm.alarm.voiceGateway.*

The subject namespace forms a hierarchical structure with a prefix portion (the first three dotted elements) that is centrally administered by CANA (Cisco Assigned Numbers Authority). This is mainly to avoid namespace collisions. The remaining part of the subject name is defined by application developers, as shown here:

```
cisco.mgmt.<systemId>.<subsystemId>...<eventClass>.<topic>
 cisco.mgmt.<systemId>.<subsystemId>...<eventClass>.<eventSubclass>.<topic>
 <--CANA Registered--> <------------organization defined--------------->
```

The ***subsystemId*** element identifies the subsystem within the management server.

The ***eventClass*** element can be used to define a general class of events used within an application. For consistency we will track the event class element names used within EMBU and reuse them whenever possible. For more on Event Classes, see.

The ***eventSubclass*** element, if present, can be used if there are a large number of events in the event class and further partitioning would be helpful.

The final element ***topic*** should be used to identify as clearly as possible the type of events that will be published on the complete subject name.

Choosing Subject Names and Namespaces

Follow the subject namespace guidelines shown in Table 19-1 to decide on a subject name that will not cause your application's subject name to collide with other applications.

These guidelines take into account the fact that events may be used for communication in a variety of environments, including between components or services:

a. Within one product on the same machine.

b. Within the same product on different machines.

c. Within different products on the same

d. Within different products on different machines.

For example, multiple servers on the same subnet, all with ESS-based network management applications installed on them, can pose a problem for situations like that in (a) above, if broadcast is used instead of multicasting. Unless subject names are chosen carefully, events from one product may be passed to another product on a different server unintentionally. One way to prevent this is to include the IP address of the server in the subject name. If this convention is adopted, each product can subscribe to messages from the services on the same server only.

Table 19-1 Subject Namespace Guidelines

For	Use the Subject Name Prefix
CWCS shared services	cisco.mgmt.cw.<IPAddress_with_underscores>.cmf.<service name>.<event-class>
CWCS per-product services	cisco.mgmt.cw.<IPAddress_with_underscores>.<product-name>.<service name>.<event-class> Where: **cw** stands for CiscoWorks. **<product-name>** is the name of the product which uses one instance of a service. In case, different applications in a given product use different instances of a service, then <product-name> should qualify such an application rather than the product itself. An example for <product-name> is RME. If RME applications A and B use two different instances of a service, then <product-name> should be of the form <RME-A> and <RME-B>, to ensure separation among the events between the service as used by application A and the service used by application B. <IPAddress_with_underscores> is the IP address of the host server in the string form "%d.%d.%d.%d" with the dots (".") replaced by underscores ("_"). This can be achieved using the methods in the class com.cisco.nm.cmf.ess.utils.Namespace.
Interproduct communication	cisco.mgmt.cw.<product name> Where**<product-name>** is the identifier of a product which is the source or publisher of the event.
Communication across multiple product instances (of same or different product types)	cisco.mgmt.cw. <cluster id>.<product name>

When choosing subject names, be aware of the following:

- Generate a dotted name hierarchy for all of your subject names such that related events are grouped. This allows convenient subscription using wildcards.
- Create a unique new subject name for each event group to which subscribers might want to selectively subscribe. For example, if most subscribers are likely to be interested in all events in a given group, put them all under a common subject name. If many or most of the subscribers are likely to be interested only in specific event types then give them each a unique subject name.
- Subject names are case sensitive.
- Limit subject names to less than 100 characters.

- Do not use the reserved subject names listed in the "About Reserved Subject Names" section on page 19-5.

Subscribing with Wildcards

Tibco Rendezvous supports two wildcard characters: the asterisk (*) and greater-than symbol (>). The * character matches one single whole element, but not a partial substring of characters, within an element. The > character matches one or more trailing elements to the right. The semantics of listening to wildcard subjects are shown in Table 19-2.

Table 19-2 Wildcard Subject Semantics[1]

Listening to this wildcard name	Matches messages with names like these:	But does not match messages with names like these (reason):
RUN.*	RUN.AWAY	RUN.run.run(extra element)
	RUN.away	Run.away (case differs)
		RUN (missing element)
Yankees.vs.*	Yankees.vs.Red_Sox	Giants.vs.Yankees (position)
	Yankees.vs.Orioles	Yankees.beat.Sox (beat used instead of vs[2])
		Yankees.vs (missing element)
.your.	Amaze.your.friends	your (missing elements)
	Raise.your.salary	pick.up.your.foot (position)
	Darn.your.socks	
RUN.>	RUN.DMC	SWIN.RUN (position)
	RUN.RUN.RUN	Run.away (case differs)
	RUN.SWIM.BIKE.SKATE	RUN (missing element)

1. This table is taken from the Tibco/RV Concepts Manual.
2. LWMS also supports subscriptions with wildcard characters with the single limitation that the publisher must be on the SEB Tibco/Rv side of the event bus.

About ESS Event Formats

ESS supports two types of message body content:

- Text: This can be an unstructured string, or structured ASCII text (such as XML).
- Java Object: The content must implement the java.io.Serializable interface.

About Reserved Subject Names

The following Subject Names are in use and reserved by NMTG:

- ***subsystemId*** Element Identifiers:
  - cisco.mgmt.trx.
  - inventory

- *eventClass* Element Identifiers:
  - .<eventClass>.
  - alarm
  - status
  - test
  - debug
  - command
- Legacy Subject Names:
  - **cisco.mgmt.cmf.**
  - **cisco.mgmt.cmf.events.ltGateway.**control
  - cisco.mgmt.cmf.events.ltGateway.allSubjects
  - **cisco.mgmt.vhm.**
  - **cisco.mgmt.vhm.alarm.**summary
  - cisco.mgmt.vhm.alarm.vhmserver
  - cisco.mgmt.vhm.alarm.dfmserver
  - cisco.mgmt.vhm.alarm.status
  - cisco.mgmt.vhm.alarm.gershwin
  - **cisco.mgmt.vhm.invupdate.summary.**add
  - cisco.mgmt.vhm.invupdate.summary.delete
  - cisco.mgmt.vhm.invupdate.summary.update
  - **cisco.mgmt.vhm.invupdate.status.**add
  - cisco.mgmt.vhm.invupdate.status.delete
  - cisco.vpnsc
  - cisco.cns

Support for Map Messages

From Common Services 3.0 Service Pack 1, map messages are supported. Publish and subscribe can happen via map messages. A map message is a message whose body contains a set of name-value pairs where names are Strings and values are Java primitive types. This map message feature is not supported for LWMS and JMS.

Sample code for Map message Publisher:

```
import com.cisco.nm.cmf.jms.*;
import java.io.*;
import javax.jms.*;

public class EssJtibMapPub{
    private static String subject = "cisco.mgmt.cw.cmf.jrm";
    private TopicConnectionFactory Factory;
    private TopicConnection conxn;
    private Topic topic;
    private TopicSession session;
    private TopicPublisher publisher;
```

```
    private MapMessage msg;

    public static void main(String args[]){

        new EssJtibMapPub().start();
    }
    private void start(){

        try{
            Factory = new TopicConnectionFactoryImp();
            conxn = Factory.createTopicConnection();
            session = conxn.createTopicSession(false,javax.jms.Session.AUTO_ACKNOWLEDGE);

            topic = session.createTopic(subject);
            publisher = session.createPublisher(topic);

            msg =session.createMapMessage();
        msg.setString("map","I am a Tibco MAP message !!!!!.");
        msg.setJMSReplyTo(new TopicImp("mapreply"));
        msg.setJMSDestination(topic);
            msg.setJMSPriority(7);
        publisher.publish(msg);
        }
        catch(Exception ex){ ex.printStackTrace();};
    }
}
```

Sample code for Map message Subscriber:

```
import java.io.*;
public class EssJtibMapSub implements javax.jms.MessageListener {

    private static String subject =    "cisco.mgmt.cw.cmf.jrm";
    private static boolean isFileBased = true;
    private javax.jms.TopicConnectionFactory Factory;
    private javax.jms.TopicConnection conxn;
    private javax.jms.Topic topic;
    private javax.jms.TopicSession session;
    private javax.jms.TopicSubscriber subscriber;

    public static void main(String args[]){
       new EssJtibMapSub().start();
    }

    private void start(){

        try{

            Factory = new com.cisco.nm.cmf.jms.TopicConnectionFactoryImp();
            conxn = Factory.createTopicConnection();
            session = conxn.createTopicSession(false,javax.jms.Session.AUTO_ACKNOWLEDGE);

            topic = session.createTopic(subject);
            subscriber= session.createSubscriber(topic);
            subscriber.setMessageListener(this);
            conxn.start();
        }catch(Exception ex) { ex.printStackTrace();};
    }

    public void onMessage(javax.jms.Message msg){
    javax.jms.MapMessage mapmsg;
        try{
        if (!(msg instanceof javax.jms.MapMessage) ) return;
```

```
            mapmsg = (javax.jms.MapMessage)msg;
        String map_msg = mapmsg.getString("map");
        return;
         }
         catch( Exception ex ) { ex.printStackTrace(); }
    }
}
```

Using the Lightweight Messaging Service

The Lightweight Messaging Service (LWMS) is a light weight XML-based messaging service used primarily between client desktops and the Common Services server. The client desktops are authenticated before sending messages.

The LWMS service provides:

- XML message format (without requiring an XML parser on the client)
- HTTP transport (for traversing firewalls)
- A zero-administration client—No administration configuration is required to create mailboxes or send or receive messages
- Native API and Java Messaging Service (JMS) API support
- A small client footprint:
  - Less than 50 KB (using native LWMS API)
  - Less than 90 KB (using JMS API)
- Message send queues, with high and normal priorities
- Efficient message filtering (using JMS message selectors)
- Support for publish/subscribe and point-to-point messaging models:
  - Publish/subscribe messaging services are a type of distributed interprocess communication (IPC) generally intended for loosely coupled asynchronous communications between software processes. Publish/subscribe messaging is useful for one-to-many, many-to-one, and many-to-many communication relationships.
  - Point-to-point (P2P) messaging is a special case of publish/subscribe messaging where there is a single publisher and a single subscriber.

The following topics describe the Lightweight Messaging Service and how to use it in your applications:

- Understanding LWMS
- Configuring LWMS
- Using the LWMS API
- Using the JMS API
- LWMS Command Reference

For more information, see:

- *CMF Lightweight Messaging Service (LWMS) Functional Specification*, ENG-58367
- *CMF Lightweight Messaging Service (LWMS) Overview Presentation*, ENG-72595
- JMS (Java Messaging Service) 1.0.2 API Specification at Sun's web site

Understanding LWMS

The following topics describe the components and features of the Lightweight Messaging Service (LWMS):

About the LWMS Components

LWMS consists of two main components:

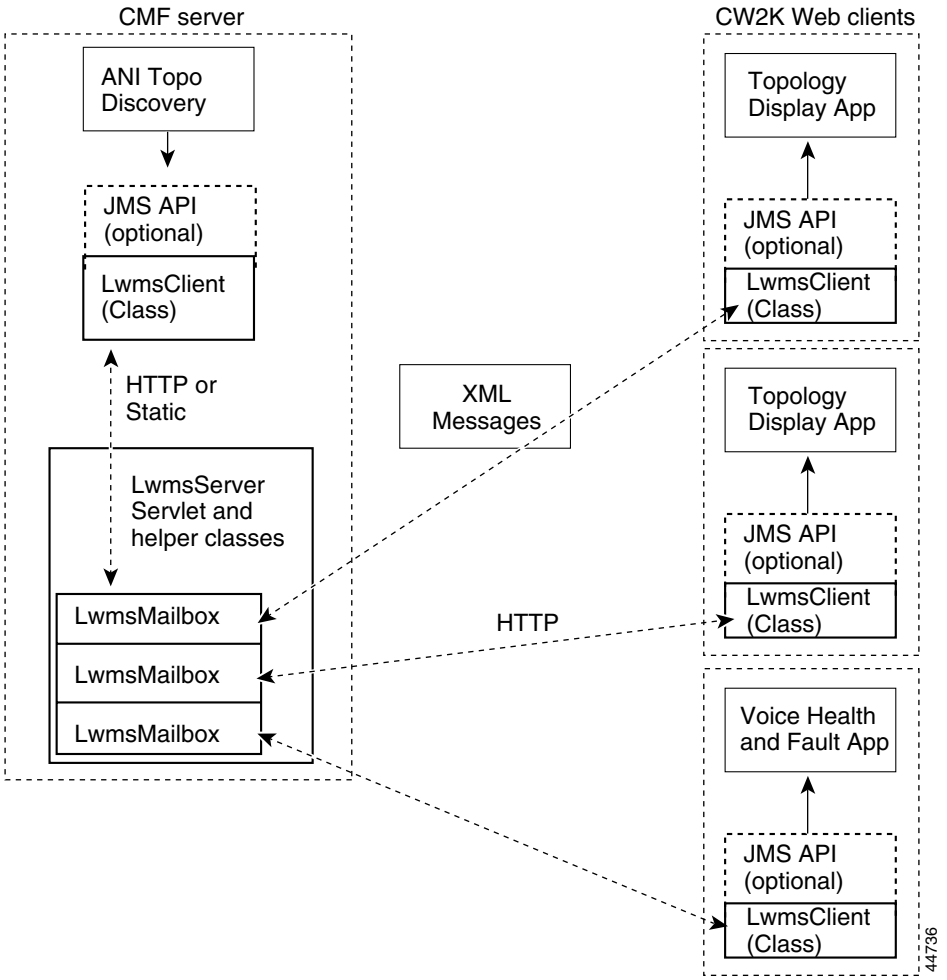
- **LwmsClient**: Provides the client interface for creating mailboxes and sending and receiving messages. The Java class that implements this interface can be used by applets in the desktop client web browser environment to send and receive messages with an LwmsServer.
- **LwmsServer**: Runs as part of the CWCS server infrastructure and handles XML message distribution.

Two more components provide additional services:

- **LwmsMailbox**: Stores and manages messages and message forwarding.
- **LwmsMessage**: Simplifies and controls creation of XML messages.

Figure 19-1 shows how all of these components work together.

Figure 19-1 LWMS Usage Example



How LWMS Works

LWMS supports three primary operations:

1. Creating mailboxes
2. Publishing messages to mailboxes
3. Listening for new messages to mailboxes.

An LWMS mailbox may have one or more senders and one or more receivers; internally, LWMS makes no distinctions between these usage types. A named mailbox simply receives messages sent to it, time stamps and ages them according to the time-to-live (TTL) field in each message, and forwards copies of new messages to each polling client.

Each LwmsClient keeps track of the last message it has seen for a given mailbox. On subsequent polls only messages later than this are forwarded to the client.

About LwmsClient

To provide the client interface for creating mailboxes and sending and receiving messages, the LwmsClient:

- Makes a single URL connection to the CWCS Server.
- Uses HTTP to send messages to the LwmsServer.
- Uses prioritized message queues (see the "About LWMS Message Queues" section on page 19-12).
- Performs the following for each mailbox with a registered listener:
  - Uses HTTP to poll the LwmsServer every N seconds to request new messages.
  - Remembers the latest message time sta.mp that it has seen and requests new messages from that point in the message stream.
  - Forwards new messages to registered listeners.
  - Supports non-blocking, asynchronous listener registration (callbacks).
- Allows the user to:
  - Set the maximum send-queue size.
  - Set the receive polling interval (the default is 1.5 seconds; the configurable range is from 0.5 to 30 seconds).
  - Set the send push interval for normal-priority messages (the default is 1 second; the configurable range is from 0.5 to 30 seconds).
  - Set the maximum number of messages returned by a server per request.

About LwmsServer

To handle message distribution, the LwmsServer:

- Maintains a set of named mailboxes that store messages until they expire.
- Timestamps new messages and adds them to the mailbox.
- Returns any new messages to clients when polled.

About LwmsMailbox

The LwmsMailbox component stores and manages messages and handles message forwarding. An LwmsMailbox:

- Receives the messages sent to it.
- Ages messages according to the value in the TTL (time-to-live) field in the message header.
- Forwards copies of new messages to each polling client

  Each LwmsClient keeps track of the latest time stamp it has seen for a given mailbox. On subsequent polls, only messages later than this time stamp are forwarded to its client.
- Supports filtering based on message header fields

An LwmsMailbox may have one or more senders and one or more receivers.

About LwmsMessage

The LwmsMessage component simplifies and controls the creation of XML messages. The body of an LwmsMessage may be either:

- A string. The string may be structured (normally, XML) or unstructured.
- An object. The object must be Base64-encoded on wire, and the receiver can cast back to the original object type.

LwmsMessage supports extensible message header elements via the msg.addHeaderElement( ) class. For example:

```
msg.addHeaderElement("deviceType", "Cat6000");
```

In this example, the client application could perform filtering operations based on the contents of the message header.

For more information about LWMS message formats, see *CMF Lightweight Messaging Service (LWMS) Functional Specification*, ENG-58367.

About LWMS Message Queues

LWMS supports two priority levels for sending messages:

- NORMAL: LWMS bundles and sends messages with normal priority periodically, at a predefined push interval (for example, once every second). Normal-priority messages are always received in the order they are sent, are the most efficient to transmit, and provide the best throughput for large bursts of messages.
- HIGH: LWMS sends high-priority messages to the LwmsServer message server immediately. This priority level provides the lowest message latency but has a higher overhead, so use it only when necessary.

  By default there is only one high priority push thread per LWMS client, which preserves the message sending order. If, however, there is more than one thread, high-priority messages may be received out of order. Multiple high priority pusher threads are recommended when higher throughput performance is required (see the "Configuring Client Properties" section on page 19-13).

Note A maximum rate limit is enforced when sending high-priority messages. If this rate is exceeded, the sending thread is blocked until the high-priority send queue and send thread-pool load drops to a lower threshold.

About JMS API Support

LWMS supports messaging via the Java Messaging Service API, Version 1.0.2. For a list of the subset of JMS services that LWMS supports, see the "JMS to LWMS Mappings" section on page 19-21.

About LWMS Server Logging

The LWMS Server outputs exceptions and important status information to the file lwms.log in the CWCS log directory.

About LWMS Usage Assumptions

LWMS was designed to meet the following usage limits:

- Typical throughput: less than 100 messages/second
- Typical fan in (number of publishers): less than 10

- Typical fan out (number of subscribers): less than 25

About Tibco-LWMS Gateway Support

LWMS implements the Client Event Bus (CEB) in the CWCS event architecture. There is also a Server Event Bus (SEB) which is currently implemented by Tibco's Rendezvous product.

When messages need to transit on both the CEB and the SEB buses, a gateway is required between LWMS and Tibco. The gateway handles message traffic in both directions:

- Gateway publishing to LWMS (Tibco to LWMS)

  LWMS provides control messages on a specified topic that notify the gateway about the topics in the LWMS bus that have active listeners. The gateway forwards messages published on these topics on the Tibco bus to LWMS.

  Control messages are sent to the gateway when:

  - A listener is added to a topic that was not in the previous control message.
  - A topic no longer has any registered listeners (as determined by specific listener deregistration or listener table aging).

- Gateway subscribing to LWMS (Tibco from LWMS)

  The gateway subscribes to a special topic that will forward all messages published in the LWMS domain. This special topic is designed to prevent messages that were published earlier by the gateway from echoing back to the gateway. This is similar to the JMS NoLocal option.

Configuring LWMS

Two configuration files are loaded at startup to provide operating properties to LWMS. The following topics describe how to modify these files to configure your LWMS implementation:

- Configuring Client Properties
- Configuring Server Properties

Configuring Client Properties

To configure the LWMS client properties, use an ASCII editor to edit the LwmsClientProperties file.

Runtime Location	$*NMSROOT*/lib/classpath/com/cisco/nm/cmf/lwms/LwmsClientProperties.xml where *NMSROOT* is the directory in which the product is installed. This directory also contains the supporting files.

The client properties file uses the tags shown in Table 19-3.

Table 19-3 LwmsClientProperties.xml File Tags

Tag	Attributes	Default	Description
serverPollInterval	VALUE	1500	Server mailbox polling interval, in milliseconds.
normalMsgPushInterval	VALUE	1000	Pushes normal messages to server, in milliseconds.
numHighPriorityPusherThreads	VALUE	1	Set to 1 to preserve message order. Higher values may increase throughput.
maxHighPriorityMsgQueueSize	VALUE	500	Blocks next sender if queue is full.
useGZIPioStreams	VALUE	False	Compresses applet-servlet streams.

Example 19-1 shows a typical LWMS client configuration file.

Example 19-1 LWMS Client Configuration File

```
<?xml version="1.0"?>
<lwmsClientProperties>
    <serverPollInterval VALUE="1500" /> # server mailbox polling interval in ms
    <normalMsgPushInterval VALUE="1000" /> # push normal messages to server, in ms
    <numHighPriorityPusherThreads VALUE="1" /> # set to 1 to preserve message order.
                      # Higher values may increase throughput
    <maxHighPriorityMsgQueueSize VALUE="500" />  # next sender is blocked if queue is full
    <useGZIPioStreams VALUE="false" /> # compress applet-servlet streams
</lwmsClientProperties>
```

Note Changes to client properties take effect on the next applet startup or reload in the client browser.

Configuring Server Properties

To configure the LWMS Server properties, use an ASCII editor to edit the LwmsServerProperties file.

Runtime Location	$*NMSROOT*/lib/classpath/com/cisco/nm/cmf/lwms/LwmsServerProperties.xml where *NMSROOT* is the directory in which the product is installed. This directory also contains the supporting files.

The server properties file uses the tags shown in Table 19-4.

Table 19-4 LwmsServerProperties.xml File Tags

Tag	Attributes	Default	Description
msgAgeingInterval	VALUE	60	Message aging interval, in seconds.
tempMailboxAgeingInterval	VALUE	30	Temporary mailbox aging interval, in seconds. Also used for the Tibco-LWMS Gateway.

Table 19-4 LwmsServerProperties.xml File Tags

Tag	Attributes	Default	Description
maxMsgsPerPoll	VALUE	500	This value divided by client poll interval determines maximum throughput.
autoCreateMailboxes	NAME	(none)	Mailboxes created at LWMS Server startup time. If TTL (time-to-live)="0" then the mailbox is persistent (useful for JMS).

Example 19-2 shows a typical LWMS Server configuration file.

Example 19-2 LWMS Server Configuration File

```
<?xml version="1.0"?>
<lwmsServerProperties>
  <msgAgeingInterval VALUE="60" />
  <tempMailboxAgeingInterval VALUE="30" />
  <maxMsgsPerPoll VALUE="500" />
  <autoCreateMailboxes>
    <create NAME="cisco.mgmt.ani.event" TTL="0" />
    <create NAME="JmsTestTopic" TTL="0" />
  </autoCreateMailboxes>
</lwmsServerProperties>
```

Note Server properties are read in on the first access to the LWMS servlet after the servlet engine starts. Therefore, after changing any server properties, you must restart the servlet engine.

Using the LWMS API

The following topics describe how to use the native LWMS API to perform typical messaging tasks:

- Creating a Mailbox with LWMS
- Posting a Message to a Mailbox with LWMS
- Polling Mailboxes for New Messages with LWMS
- Removing a Message Listener with LWMS
- Filtering Messages with LWMS

Creating a Mailbox with LWMS

Use the following code to get a reference to an LwmsClient singleton object and create a mailbox:

```
import com.cisco.nm.cmf.lwms.*;
LwmsClient lc= LwmsClient.getInstance( "http://server:1741",cookie_value,user,is_cli );
lc.createMailbox("mbox");
```

Posting a Message to a Mailbox with LWMS

To create a message in LWMS, set the message properties and send the message:

```
LwmsMessage msg = new LwmsMessage();
    msg.setToMailbox ("mbox");
    msg.setMsgBody("Hello Gang");
lc.sendMsg(msg);
```

To get a reply, add these lines:

- Create a replyMailbox:

  ```
  lc.createMailbox("Replymbox");
  ```

- Set the replyTo field in the message:

  ```
  msg.setReplyTo("Replymbox");
  ```

- Listen on that mailbox:

  ```
  lc.addMsgListener("Replymbox", this);
  ```

Polling Mailboxes for New Messages with LWMS

To register a message listener with LWMS:

```
lc.addMsgListener("mbox", this);
```

where the current object:

- Implements the LwmsMsgListener interface
- Has a public newMessage(LwmsEvent) method

Removing a Message Listener with LWMS

To unregister a message listener with LWMS:

```
void removeMsgListener(String mailbox, LwmsMsgListener msgListener);
```

Filtering Messages with LWMS

An LWMS message consumer may optionally specify a message filter for each mailbox at listener registration time. This filter has the form of a Boolean expression containing equality, relational, or logical operators. Each message that evaluates to true based on the filter criteria will be returned to the listener. Filters check only the standard and extended headers.

Table 19-5 summarizes the filter operator types LWMS supports.

Table 19-5 LWMS-Supported Filter Operators

Operator	Types
Equality	=, <> (equal to, not equal to)
Relational	>, <, >=, <= (greater than, less than, greater than or equal to, less than or equal to)
Logical	AND, OR

Base Filter Expression (bfe) Syntax

The base filter expression uses this format:

```
match_string operator literal_value
```

where:

- The white space before and after the *operator* field is required.
- The *match_string* field must begin with a '<' and end with either:
  - A '>' for a standard XML element header field
  - An '=' for an attribute in an XML empty element header field
- The *match_string* field identifies a message header field and is of one of the following two forms:
  - <tag>—identifies a standard XML element value
  - <tag VALUE=—identifies an XML element attribute
- If *match_string* ends in ">", then its value is a substring extraction up to the next "<" character.

  For example, a subscriber with

  ```
  filter = "<foo> = 'bar'"
  ```

  would receive messages with an extended header <foo>bar</foo>.
- If *match_string* ends in "=", then its value is a substring extraction from the first quote mark up to the next quote mark.

  For example, a subscriber with

  ```
  filter = "<foo VALUE= < 7"
  ```

  would receive messages with an extended header of <foo VALUE="5"/> (or any value less than 7).
- The *operator* is one of: =, <>, >, <, >=, <=.
- String and character literals are enclosed in single quotes (for example, `'stringValue'`).
- Each base filter expression evaluates to Boolean true or false.

Filter Strings Syntax

Filter strings supplied to the listener registration call use this syntax:

```
String filter= "bfe [AND bfe | OR bfe]";
```

where:

- Additional base filter expression terms enclosed in brackets are optional.
- Additional base filter expression terms must be preceded by either AND or OR.

Evaluation Rules

LWMS evaluates filter rules as follows:

- Base filter expression evaluation order is left to right. Parentheses are not supported.
- If the LwmsServer cannot parse the filter, it is ignored and all new messages on the mailbox are returned.

Sample Filters

Here are two separate examples of typical filters:

```
String filter_1= "<from> = 'AniServer'";
```

```
String filter_2= "<deviceType> = 'Cat6000' AND <deviceResets> > 5";
```

Using the JMS API

The following topics describe how to use the JMS APIs to perform typical messaging tasks:

- Creating a Mailbox with JMS APIs
- Posting a Message to a Mailbox with JMS
- Polling Mailboxes for New Messages with JMS
- Removing a Message Listener with JMS
- Using JMS Message Selectors

Creating a Mailbox with JMS APIs

To start a connection and a session and get a topic publisher using the JMS API:

```
import com.cisco.nm.cmf.lwms.jms.*;
import javax.jms.*;
TopicConnectionFactory tcf=
    new TopicConnectionFactoryImp (protocol, hostName, port);
TopicConnection tc= tcf.createTopicConnection();
TopicSession sess=
    tc.createTopicSession (false, DUPS_OK_ACKNOWLEDGE);
Topic pubTopic= new lwms.jms.TopicImp ("JmsTestTopic");
TopicPublisher publisher= pubSession.createPublisher (pubTopic);
tc.start();
```

Posting a Message to a Mailbox with JMS

To post a message to a topic using the JMS API:

```
javax.jms.TextMessage message= sess.createTextMessage();
 message.setText ("Hello from LWMS/JMS");
 message.setJMSDestination (pubTopic);
publisher.publish (message);
```

Polling Mailboxes for New Messages with JMS

To subscribe to a topic using the JMS API:

```
import com.cisco.nm.cmf.lwms.jms.*;
import javax.jms.*;

TopicConnectionFactory tcf=
     new TopicConnectionFactoryImp (protocol, hostName, port);
TopicConnection tc= tcf.createTopicConnection();
TopicSession sess= tc.createTopicSession (false, DUPS_OK_ACKNOWLEDGE);
Topic subTopic= new lwms.jms.Topic ("JmsTestTopic");
TopicSubscriber subscriber= sess.createSubscriber (subTopic);
subscriber.setMessageListener (this);
tc.start();
```

Removing a Message Listener with JMS

To unregister a JMS message listener:

```
TopicSubscriber.setMessageListener(null)
```

Using JMS Message Selectors

LWMS supports JMS message selectors. The JMS message-selector syntax is identical to the LWMS message filter syntax (see the "Filtering Messages with LWMS" section on page 19-16).

LWMS Command Reference

These topics provide reference information about LWMS and JMS API calls:

- LWMS Native API Messaging Methods
- JMS to LWMS Mappings

LWMS Native API Messaging Methods

The following tables list the public Java interfaces needed to create a mailbox, send a message, and perform other messaging tasks.

Table 19-6 LwmsClient Public Methods

Returns	Syntax and Description
Setup	
`Boolean`	**`checkMailboxExists`** `(String` *mailbox*`);` Returns true if named mailbox already exists on LwmsServer.
`Boolean`	**`createMailbox`** `(String` *mailbox*`);` Creates a mailbox on server.
`Boolean`	**`createMailbox`** `(String mailbox, int ttl);` Creates a temporary mailbox on server, will be deleted after TTL (time-to-live) seconds. For a permanent mailbox, set TTL=0. Returns success.
Sender (Message Producer)	
`void`	**`sendMsg`** `(LwmsMessage` *msg*`);` Sends a message to server. Note: Message object should not be mutated after calling.
Receiver (Message Consumer)	
`void`	**`addMsgListener`** `(String` *mailbox*`, LwmsMsgListener` *listener*`);` Registers a message listener.
`void`	**`addMsgListener`** `(String` *mailbox*`, LwmsMsgListener` *listener*`, String filter);` Registers a message listener with a filter specification.
`void`	**`removeMsgListener`** `(String` *mailbox*`, LwmsMsgListener` *msgListener*`)`

Table 19-7 LwmsMessage Message Creation Methods

Returns	Syntax and Description
void	**addHeaderElement** (String name, String value); Adds "<name>value</name>".
void	**setFrom** (String *from*);
void	**setPriority** (int *priority*); Allowed values: • LwmsMessageIF.HIGH • LwmsMessageIF.NORMAL
void	**setMsgBody** (String *msgBody*);
void	**setMsgBody** (java.io.Serializable *msgBody*); Object will be serialized and Base64 encoded.
void	**setMsgName** (String *msgName*);
void	**setMsgType** (String *msgType*);
void	**setReplyTo** (String *replyToMailbox*);
void	**setSenderTimeStamp** (long *senderTimeStamp*);
void	**setTimeToLive** (long *timeToLive*); Messages TTL in seconds.
void	**setToMailbox** (String *toMailbox*);

Table 19-8 LwmsMsgListener Interface

Returns	Syntax and Description
void	**newMessage** (LwmsMsgEvent *event*); Callback method in listener.

Table 19-9 LwmsMsgEvent Methods

Returns	Syntax and Description
String[]	**getExtendedHeaderElementNames**();
String	**getExtendedHeaderElementValue** (String name);
String	**getFrom**();
long	**getMboxTimeStamp**();
String	**getMsg**(); Returns whole XML message.
Object	**getMsgBody**(); Returns message body: • If MsgBodyType== STRING then cast as String. • If MsgBodyType== OBJECT then cast as appropriate for your application.
String	**getMsgBodyType**();
String	**getMsgID**();
String	**getMsgName**();
String	**getMsgType**();

Table 19-9 LwmsMsgEvent Methods (continued)

Returns	Syntax and Description
int	**getPriority**();
String	**getReplyToMailbox**();
long	**getSenderTimeStamp**()
long	**getTimeToLive**();
String	**getToMailbox**();

JMS to LWMS Mappings

The following tables list significant mappings between the JMS and LWMS APIs and message fields.

For a complete list of JMS to LWMS mappings, see the *CMF Lightweight Messaging Service (LWMS) Functional Specification*, ENG-58367.

Table 19-10 JMS to LWMS API Mappings

JMS	LWMS

`publisher.publish( msg );`
`publisher.publish( topic, msg);`
`publisher.publish(msg, deliveryMode,`
`priority, ttl );`
`publisher.publish( topic, msg,`
`deliveryMode, priority, ttl );`
	Converts JMS Message to a LWMS Message per the mappings in Table 19-11 and then calls LwmsClient.sendMsg().
	If publish() is called with TTL (time-to-live), the TTL should be set to at least two times the client polling interval.
	If ttl=0 (in JMS, this means the message never expires), LWMS sets the message's TTL to one hour; this is done because LWMS architecture specifies that messages must expire eventually.

Table 19-11 JMS to LWMS Message Field Mappings

JMS	LWMS
Body	body
JMSCorrelationID	If present, this field is mapped to an extended header field named JMSCorrelationID.
JMSDeliveryMode	LWMS supports the equivalent of the JMS NON_PERSISTENT delivery mode
JMSDestination:Topic (topic name string is used)	toMailbox
JMSExpiration	Mapped to an extended header field of the same name. Calculated as the sum of the TTL (time-to-live) value in the publish() method call and the client's current time.
JMS Message Properties	LWMS Message Extended Header fields
JMS Message types: TextMessage or ObjectMessage	msgBodyType (TYPE="STRING" or TYPE="OBJECT")
JMSMessageID	LWMS msgID

Table 19-11 JMS to LWMS Message Field Mappings (continued)

JMS	LWMS
JMSPriority LWMS —> JMS Normal —> 2 High —> 7	Priority JMS —> LWMS 0-4 —> Normal 5-9 —> High
JMSRedelivered	not mapped
JMSReplyTo:Topic (topic name string is used)	replyToMailbox
JMSTimeStamp	senderTimeStamp The LWMS-JMS client generates this time stamp in the client VM when the publisher.publish() method is called.
JMSType	If present, this field is mapped to an extended header field named JMSType.
Time-to-live (TTL) value from publish() method call, or default value of 60 seconds if not specified	timeToLive
no direct JMS equivalent	from
no direct JMS equivalent	msgName
no direct JMS equivalent	mboxTimeStamp