BEA WebLogic Collaborate

A Component of BEA WebLogic Integration

Programming BEA WebLogic Collaborate Messaging Applications

BEA WebLogic Collaborate Release 2.0
Document Edition 2.0
July 2001

Copyright

Restricted Rights Legend

Trademarks or Service Marks

Programming BEA WebLogic Collaborate Messaging Applications

Document Edition	Date	Software Version
2.0	July 2001	2.0

Contents

About This Document

1. Developing XOCP C-Enabler Applications to Exchange Business Messages

2. Programming Steps for C-Enabler Applications

3. Sending XOCP Business Messages

4. Receiving XOCP Business Messages

Index

About This Document

This document describes how to use the BEA WebLogic Collaborate Messaging API to develop XOCP protocol messaging applications.

This document includes the following topics:

- Chapter 1, "Developing XOCP C-Enabler Applications to Exchange Business Messages," discusses steps required to develop applications that exchange business messages using the BEA WebLogic Collaborate eXtensible Open Collaboration Protocol (XOCP).
- Chapter 2, "Programming Steps for C-Enabler Applications," discusses the steps required to program applications that exchange business messages using the XOCP protocol.
- Chapter 3, "Sending XOCP Business Messages," discusses the requirements for sending XOCP business messages.
- Chapter 4, "Receiving XOCP Business Messages," discusses requirements for receiving XOCP business messages.

What You Need to Know

This document is intended for independent software vendors (ISVs) who want to extend BEA WebLogic Collaborate. It assumes a familiarity with the BEA WebLogic Collaborate platform and Java programming.

e-docs Web Site

BEA product documentation is available on the BEA corporate Web site. From the BEA Home page, click on Product Documentation or go directly to the "e-docs" Product Documentation page at `http://e-docs.bea.com`.

How to Print the Document

You can print a copy of this document from a Web browser, one file at a time, by using the File→Print option on your Web browser.

A PDF version of this document is available on the WebLogic Collaborate documentation Home page on the e-docs Web site (and also on the documentation CD). You can open the PDF in Adobe Acrobat Reader and print the entire document (or a portion of it) in book format. To access the PDFs, open the WebLogic Collaborate documentation Home page, click the PDF files button and select the document you want to print.

If you do not have the Adobe Acrobat Reader, you can get it for free from the Adobe Web site at `http://www.adobe.com`.

Related Information

The following BEA WebLogic Collaborate documents contain information that will help you understand how to extend WebLogic Server:

- BEA WebLogic Collaborate documentation (available online):
  - *Administering BEA WebLogic Collaborate*
  - *Programming BEA WebLogic Collaborate Management Applications*
  - *Programming BEA WebLogic Collaborate Logic Plug-Ins*

- The Sun Microsystems, Inc. Java site at `http://java.sun.com/`

For more information about BEA WebLogic Server and Java, refer to the WebLogic Server documentation available at `http://edocs.bea.com/`.

Contact Us!

Your feedback on the BEA WebLogic Collaborate documentation is important to us. Send us e-mail at **docsupport@bea.com** if you have questions or comments. Your comments will be reviewed directly by the BEA professionals who create and update the WebLogic Collaborate documentation.

In your e-mail message, please indicate that you are using the documentation for the BEA WebLogic Collaborate 6.0 release.

If you have any questions about this version of BEA WebLogic Collaborate, or if you have problems installing and running BEA WebLogic Collaborate, contact BEA Customer Support through BEA WebSupport at **www.bea.com**. You can also contact Customer Support by using the contact information provided on the Customer Support Card, which is included in the product package.

When contacting Customer Support, be prepared to provide the following information:

- Your name, e-mail address, phone number, and fax number
- Your company name and company address
- Your machine type and authorization codes
- The name and version of the product you are using
- A description of the problem and the content of pertinent error messages

Documentation Conventions

The following documentation conventions are used throughout this document.

Convention	Item
boldface text	Indicates terms defined in the glossary.
Ctrl+Tab	Indicates that you must press two or more keys simultaneously.
italics	Indicates emphasis or book titles.

`monospace text`	Indicates code samples, commands and their options, data structures and their members, data types, directories, and file names and their extensions. Monospace text also indicates text that you must enter from the keyboard.
	Examples:
	`#include <iostream.h> void main ( ) the pointer psz`
	`chmod u+w *`
	`\tux\data\ap`
	`.doc`
	`tux.doc`
	`BITMAP`
	`float`

`monospace boldface text`	Identifies significant words in code. *Example*: `void` **`commit`** `( )`
`monospace italic text`	Identifies variables in code. *Example*: `String` *`expr`*
UPPERCASE TEXT	Indicates device names, environment variables, and logical operators. *Examples*: LPT1 SIGNON OR

Convention	Item
`{ }`	Indicates a set of choices in a syntax line. The braces themselves should never be typed.

`[ ]`	Indicates optional items in a syntax line. The brackets themselves should never be typed.
	Example:
	`buildobjclient [-v] [-o name ] [-f file-list]...`
	`[-l file-list]...`

`|`	Separates mutually exclusive choices in a syntax line. The symbol itself should never be typed.

`...`	Indicates one of the following in a command line:
	■ That an argument can be repeated several times in a command line
	■ That the statement omits additional optional arguments
	■ That you can enter additional parameters, values, or other information
	The ellipsis itself should never be typed.
	Example:
	`buildobjclient [-v] [-o name ] [-f file-list]...`
	`[-l file-list]...`

. . .	Indicates the omission of items from a code example or from a syntax line. The vertical ellipsis itself should never be typed.

CHAPTER

1 Developing XOCP C-Enabler Applications to Exchange Business Messages

XOCP is the default protocol used by WLC for exchanging business messages. This section includes the following topics:

Introduction

The WebLogic Collaborate C-Enabler API is no longer recommended for developing XOCP applications to exchange business messages. This documentation is provided to support preexisting WebLogic Collaborate installations in which the XOCP protocol is used to manage business message processes.

Instead of using the API the recommended method is to use the WebLogic Process Integrator Studio with the WebLogic Collaborate plug-in. To create applications that exchange business messages, see the WebLogic Process Integrator Studio.

Terminology in this documentation differs in several significant ways from current WebLogic Collaborate terminology. For information about migrating from previous releases of WebLogic Collaborate to WebLogic Collaborate Release 2.0, including a table that maps the terminology differences, see "Migrating Applications that Exchange Business Messages Using the XOCP Protocol" in *Migrating BEA WebLogic Collaborate to Release 2.0*.

Note: If you migrated a Java messaging application that was written using the WebLogic Collaborate C-Enabler API to WebLogic Collaborate Release 2.0, the migrated application must be run in a separate Java Virtual Machine (JVM) in nonpersistent mode.

Many of the code examples in this documentation derive from the installation verification example. For more information, see *Installing BEA WebLogic Collaborate* and the "Hello Partner Sample" in *Using BEA WebLogic Collaborate Samples*.

Developers can also design and implement workflows by using the WebLogic Process Integrator Studio. For more information, see *Using the BEA WebLogic Process Integrator Studio*, Chapter 2, "Using Workflows to Exchange Business Messages."

The following sections describe XOCP c-enabler applications and related concepts:

- Key Concepts
- Key Tasks for C-Enabler Applications
- Run-Time Information Flow

Key Concepts

This section describes the following key concepts associated with c-enabler applications:

- XOCP C-Enabler Applications
- C-Enabler Class Library

- Conversations and Conversation Definitions
- XOCP Business Messages and Message Envelopes
- Conversation Initiators and Participants
- Conversation Coordinators
- Trading Partner States
- Secure Messaging

XOCP C-Enabler Applications

XOCP c-enabler applications are Java applications that run on c-enabler nodes and use the C-Enabler Class Library to execute the following tasks: join and leave c-spaces; initiate or participate in conversations; terminate or leave conversations; and exchange XOCP business messages with other trading partners in the c-space. A c-enabler node can host many XOCP c-enabler applications.

C-Enabler Class Library

The C-Enabler Class Library provides APIs for exchanging XOCP business messages and consists of the packages listed in the following table.

Table 1-1 C-Enabler Class Library Packages

Package Name	Description
`com.bea.b2b.enabler`	Used for working with c-enabler nodes and c-enabler sessions.
`com.bea.b2b.enabler.xocp`	Used for working with c-enabler sessions for the eXtensible Open Collaboration Protocol (XOCP).
`com.bea.b2b.protocol.xocp.conversation.local`	Used for working with conversations based on the eXtensible Open Collaboration Protocol (XOCP).
`com.bea.b2b.protocol.messaging`	Used for working with messages in a conversation.

Table 1-1 C-Enabler Class Library Packages (Continued)

Package Name	Description
`com.bea.b2b.protocol.xocp.messaging`	Used for working with messages in conversations based on the eXtensible Open Collaboration Protocol (XOCP).

For detailed information about these packages, see the Javadoc on the WebLogic Collaborate documentation CD or in the `classdocs` subdirectory of your WebLogic Collaborate installation.

Conversations and Conversation Definitions

In WebLogic Collaborate, a *conversation* is a series of message exchanges between trading partners that takes place in a collaboration space and that is predefined according to a conversation definition. Each message in the conversation may cause any number of back-end transactions.

A *conversation definition* consists of a unique conversation name, conversation version, message definitions, trading partner IDs, and trading partner roles for one conversation. At design time, you use the WebLogic Process Integrator Studio to link a workflow template definition to a particular role (such as *buyer* or *seller*) in a WebLogic Collaborate conversation definition.

XOCP Business Messages and Message Envelopes

An *XOCP business message* is the basic unit of communication exchanged between trading partners in an XOCP conversation. An XOCP business message is represented in the c-enabler class library by the `com.bea.b2b.protocol.xocp.messaging.XOCPMessage` class.

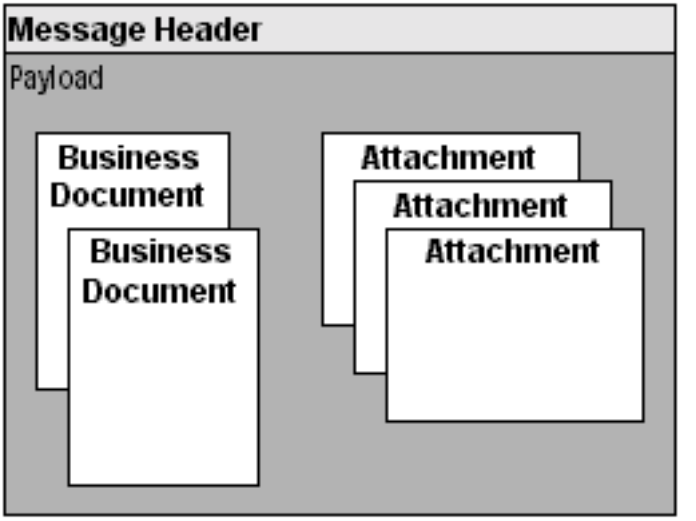
A *message envelope* is a container for a business message. A message envelope contains information about the sender (such as the sender URL) and recipient (such as the destination URL). A message envelope is represented in the C-Enabler Class Library by the `com.bea.b2b.protocol.messaging.MessageEnvelope` class. However, only logic plug-ins (not c-enabler applications) have programmatic access

to message envelopes. For more information, see "Information Flow for Message Envelopes" on page 1-7 and "Routing and Filtering Business Messages" in *Programming BEA WebLogic Collaborate Logic Plug-Ins*.

Diagram of an XOCP Business Message

The following figure shows a message envelope and the components of an XOCP business message.

Figure 1-1 Components of an XOCP Business Message



Components of an XOCP Business Message

An XOCP business message is a multipart MIME (Multipurpose Internet Mail Extensions) message. It consists of the following components.

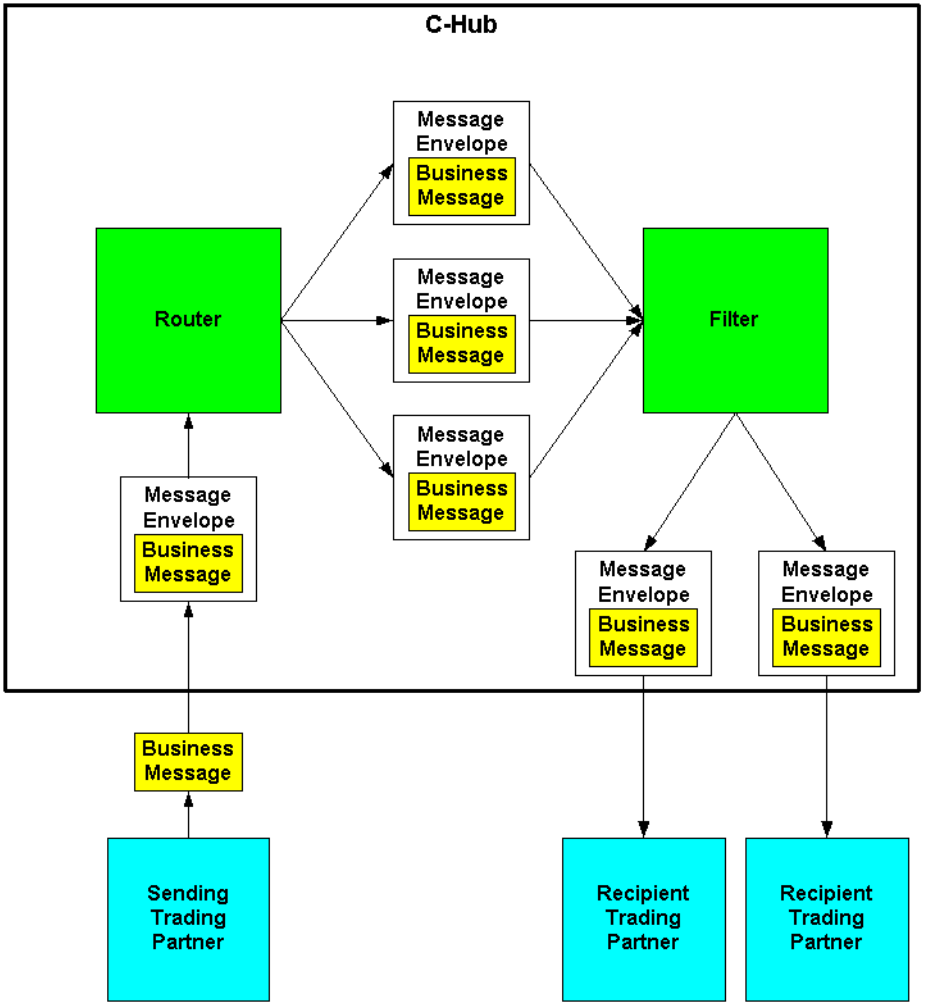
Table 1-2 Components of an XOCP Business Message

Component	Description
Message header	Message attributes, including the sender and recipient information, conversation information, Qualities of Service information, and so on.
Payload	Container for business document(s) and attachment(s) in this business message. The payload container has one or more business documents, one or more attachments, or a combination of both. A payload part is represented in the C-Enabler Class Library by the `com.bea.b2b.protocol.messaging.PayloadPart` interface.
Business document(s)	XML-based payload part of a business message. Represented in the C-Enabler Class Library by the `com.bea.b2b.protocol.messaging.BusinessDocument` class.
Attachment(s)	NonXML-based payload part of a business message. Binary content. Represented in the C-Enabler Class Library by the `com.bea.b2b.protocol.messaging.Attachment` class.

Information Flow for Message Envelopes

The following figure shows an example of how message envelopes are processed in the c-hub.

Figure 1-2 Message Envelope Processing in the C-Hub



Message envelope processing occurs in the following sequence:

1. The sending c-enabler application creates and sends the business message to the c-hub.
2. The c-hub receives the business message and wraps it with a message envelope, extracting certain sender and recipient information from the business message.
3. The router processes the business message, and then validates and finalizes the list of recipients.
4. The router creates a separate message envelope for each recipient in the recipients list, inserts a logical copy of the business message in the message envelope, and then forwards all message envelopes to the filter.

   As shown in Figure 1-2, the router creates message envelopes for three recipients.
5. Within the filter, the applicable protocol-specific filter for each recipient trading partner evaluates each business message to determine whether it will be sent to the recipient. The filter forwards accepted messages to the next processing step in the c-hub.

   In Figure 1-2, the three business messages are evaluated in the filter. Two are accepted and one is rejected.
6. The c-hub validates the recipient, and then sends the business message (in its message envelope) to the recipient trading partner.
7. The recipient trading partner receives the business message.

Conversation Initiators and Participants

In any XOCP conversation, there are two types of trading partner roles:

- *Conversation initiator* is the trading partner who creates the conversation and sends the first business message (such as a request) to one or more recipient trading partners. The conversation initiator usually awaits a reply from each trading partner and might exchange subsequent business messages. When finished, the conversation initiator terminates the conversation (unless the conversation has timed out).

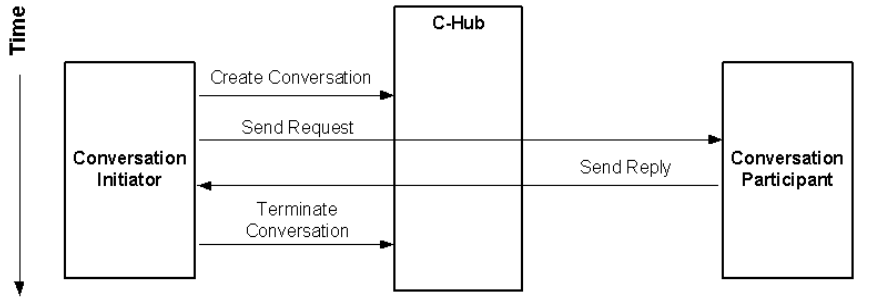
- *Conversation participant* is a trading partner who is enlisted in the conversation when it receives the first business message from the conversation initiator. The conversation participant usually sends a reply to the conversation initiator and, optionally, might exchange subsequent business messages. When finished, the conversation participant either leaves the conversation or waits until the conversation terminates.

Each conversation definition in the repository includes at least both of these types of roles. A trading partner must be subscribed to the appropriate role in the conversation in order to initiate or participate in conversations associated with that conversation definition.

The initiator of a conversation is usually determined by the role in which a trading partner is registered. For example, in a `GetQuote` conversation, the trading partner who is in the role of the buyer normally initiates a `GetQuote` conversation. Any trading partner who is in the role of the seller would normally act as a conversation participant in the `GetQuote` conversation.

The following figure shows some of the tasks that conversation initiators and conversation participants perform.

Figure 1-3 Conversation Initiators and Participants

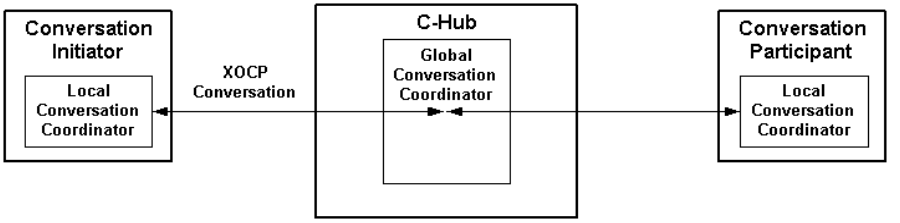


Conversation Coordinators

WebLogic Collaborate supports two types of conversation coordinators that manage conversations at run time: a *global conversation coordinator* manages active conversations on the c-hub, and *local conversation coordinators* in c-enablers help the global coordinator manage active conversations locally.

The following figure shows global and local conversation coordinators in the WebLogic Collaborate architecture.

Figure 1-4 Global and Local Conversation Coordinators



Global Conversation Coordinator

A global conversation coordinator is a c-hub-based service that coordinates conversation life cycles according to the rules of XOCP and supports long-living, durable conversations that span multiple organizational boundaries. The global conversation coordinator maintains a list of active conversations in the c-hub.

The global conversation coordinator performs the following services:

- Enlists and delists trading partners in a conversation
- Enforces the XOCP conversation termination protocol
- Maintains status information about conversations
- Provides the conversational context for the execution of the business protocol

Local Conversation Coordinators

A local conversation coordinator is a c-enabler-based service that coordinates conversations in which the c-enabler node is participating. The local conversation coordinator maintains a list of active conversations in which the c-enabler node is participating. Each c-enabler session has a separate local conversation coordinator.

The local conversation coordinator performs the following tasks:

- Locally enlists in a conversation when the initial business message in a conversation is received from the c-hub
- Locally delists from a conversation when the system message that terminates the conversation is received from the c-hub

Trading Partner States

The following table describes the states assigned to trading partners as they perform tasks related to c-space and conversation participation.

Table 1-3 Trading Partner States

State	Description
`CONNECTED`	Trading partner has joined a c-space.
`REGISTERED`	Connected trading partner has registered for roles in conversations and is ready to initiate or participate in conversations.
`ACTIVE`	Registered trading partner has participated in (that is, has sent or received a business message) at least one conversation.
`DROPPEDOUT`	Trading partner has left a conversation.
`DISCONNECTED`	Trading partner has left a c-space.

Some of these trading partner states are visible in the WebLogic Collaborate Administration Console. For more information, see "Getting Started Using WebLogic Collaborate" in *Introducing BEA WebLogic Collaborate*.

Secure Messaging

Communication between the c-hub and c-enablers is secured via the Secure Sockets Layer (SSL). Before allowing the trading partner to exchange business messages, the c-hub must authenticate the identity of the trading partner using the trading partner's

certificate. Once authenticated, business messages are exchanged securely among trading partners via the c-hub. For more information about WebLogic Collaborate security, see *Using BEA WebLogic Collaborate Security*.

Key Tasks for C-Enabler Applications

This section introduces the key tasks that c-enabler applications perform:

- Joining a C-Space
- Registering for a Role in a Conversation
- Engaging in Conversations with Trading Partners
- Shutting Down a C-Enabler Session to Leave a C-Space

Joining a C-Space

Before exchanging business messages, a c-enabler application must join a c-space. To join a c-space, the c-enabler application must create a *c-enabler session*, which is a logical session between a c-enabler node and one c-hub for one particular c-space.

Before a trading partner (c-enabler application) can create a c-enabler session to join a c-space:

- The c-space and trading partner configuration information must be defined in the WebLogic Collaborate repository on the spoke and hub that hosts the c-space.
- The session's configuration information (as well as the c-space, c-hub and c-enabler URL, trading partner name, and c-enabler session name) must be defined in the c-enabler XML configuration file. For more information, see "Configuration Requirements" in *Administering BEA WebLogic Collaborate*.
- The trading partner must be authorized to join the c-space.

When a c-enabler session is created, the c-enabler sends a system message to the c-hub with a request to join the c-space using the configuration settings specified in the c-enabler XML configuration file. This message acts as an authentication request to

join the WebLogic Collaborate system. The c-hub validates the registration of the trading partner in the requested c-space and, if the registration is valid, allows that trading partner to join that particular c-space. At this point, the trading partner is in a `CONNECTED` state but it cannot yet participate in conversations.

Note: If the c-enabler node crashes after joining a c-space, the c-enabler application can rejoin the c-space upon normal startup. The previous c-enabler session is discarded and new resources are assigned to the new c-enabler session. However, the c-hub cannot deliver business messages while the c-enabler node is down. Undelivered business messages are discarded if the number of retry attempts is exceeded or if the business message or conversation times out.

When a trading partner wants to leave a c-space, the c-enabler application shuts down the associated c-enabler session, as described in "Shutting Down a C-Enabler Session to Leave a C-Space" on page 1-16.

Registering for a Role in a Conversation

Once connected, a trading partner needs to register a conversation handler for a particular role in a specific conversation definition in a given c-space. The conversation handler must be registered for the conversation type that defines how the trading partner participates in the conversation.

Role registration requires the following information in the c-hub repository:

- The *conversation type* is a subset of a conversation definition that defines a conversation for one trading partner based on the role in the conversation definition to which the trading partner subscribed.
- A *message definition* consists of ordered message parts. A message part contains a content type (XML or binary) and can contain a document definition. If the content type for a part is XML, then a document definition is required for that part. For type binary, no other information is required.

For an introduction to these concepts, see "Getting Started Using WebLogic Collaborate" in *Introducing BEA WebLogic Collaborate*.

Before registering for a conversation type, the trading partner must first be authorized to register. Authorization is configured by the c-hub administrator and is based on the trading partner's subscription to a role in a conversation definition.

When a c-enabler session attempts to register a conversation handler for a specific conversation type, the c-enabler sends an XOCP system message, register for conversation, to the c-hub. The c-hub validates the role of the trading partner for the requested conversation type in the associated c-space. If the registration is valid, the trading partner is then allowed to initiate and participate in conversations associated with the registered conversation type. At this point, the trading partner is in a `REGISTERED` state and is ready to initiate or participate in conversations.

Engaging in Conversations with Trading Partners

Once registered for a role in a conversation, a trading partner can engage in conversations in accordance with that role. Conversation initiation and participation occurs on the c-hub itself. However, the c-enabler session maintains some state information about the conversations in which it is involved.

The overall tasks for conversation initiator c-enabler applications and conversation participant c-enabler applications are very similar. However, conversation initiator c-enabler applications can terminate conversations while conversation participant c-enabler applications cannot. Conversation participant c-enabler applications can only leave a conversation.

Initiating a Conversation and Sending a Business Message

To initiate a conversation, a conversation initiator c-enabler application creates the conversation. Optionally, the conversation initiator c-enabler application can specify a timeout value, after which the conversation automatically terminates; this value overrides the timeout value that is specified in the associated conversation definition in the repository.

The local conversation coordinator on the c-enabler node sends an XOCP system message, create conversation, to the c-hub. The global conversation coordinator in the c-hub creates a conversation in the appropriate c-space and enlists the trading partner as the conversation initiator. After the conversation is created, the conversation initiator c-enabler application creates and sends a business message, as described in "Sending XOCP Business Messages" on page 3-1.

Participating in a Conversation

The global conversation coordinator in the c-hub handles all business messages that the c-hub receives for a given conversation. After the c-hub delivers the initial business message to recipient trading partners, the global conversation coordinator enlists those trading partners in that conversation. Once a trading partner is enlisted in a conversation, the trading partner is in an `ACTIVE` state and can send and receive business messages in that conversation.

When the c-enabler session on a target c-enabler node receives the initial business message in a conversation, it performs the necessary housekeeping (such as registering the conversation in the local list) before invoking the `onMessage` callback on the conversation handler. For more information, see "Receiving XOCP Business Messages" on page 4-1.

Once a registered trading partner is enlisted in a conversation, the trading partner is in an `ACTIVE` state and can send and receive business messages in that conversation.

Leaving a Conversation

When it has finished participating in a conversation, a conversation participant trading partner can leave the conversation. When a trading partner leaves a conversation, it is removed, by the conversation coordinator, from the list of participating trading partners. Subsequent business messages in that conversation are *not* sent to that trading partner. After a trading partner leaves, it is kept in a `DROPPEDOUT` state for the remainder of that conversation.

Terminating Conversations

A conversation terminates when the initiating trading partner explicitly terminates the conversation, or when the conversation times out; whichever occurs first. A trading partner who has initiated a conversation must terminate that conversation at the appropriate time in a business process.

Note: Only the conversation initiator can terminate a conversation.

When a conversation is terminated, the conversation coordinator sends all of the participating trading partners an XOCP system message, terminate message, which is propagated as the callback `onTerminate` on registered conversation handlers in c-enabler sessions at respective c-enabler nodes.

Shutting Down a C-Enabler Session to Leave a C-Space

When a trading partner has finished its activities in a c-space, the c-enabler application should leave the c-space by shutting down the c-enabler session. When a c-enabler application shuts down a c-enabler session, the c-enabler sends an XOCP system message, leave c-space, to the c-hub. When the c-hub receives this system message, the conversation coordinator automatically terminates all of the conversations that the trading partner has initiated in the c-space and delists the trading partner from all other conversations in which it was participating in the c-space.

When a trading partner leaves a c-space, the consequences are as follows:

- The c-hub is stopped from sending any further messages to the trading partner associated with the shutdown c-enabler session.
- All conversations that were initiated by the trading partner are terminated.
- The trading partner leaves any conversations in which it was participating.
- The trading partner reclaims resources allocated in the c-hub for that c-enabler session.

At this point, the trading partner is in a `DISCONNECTED` state in that c-space.

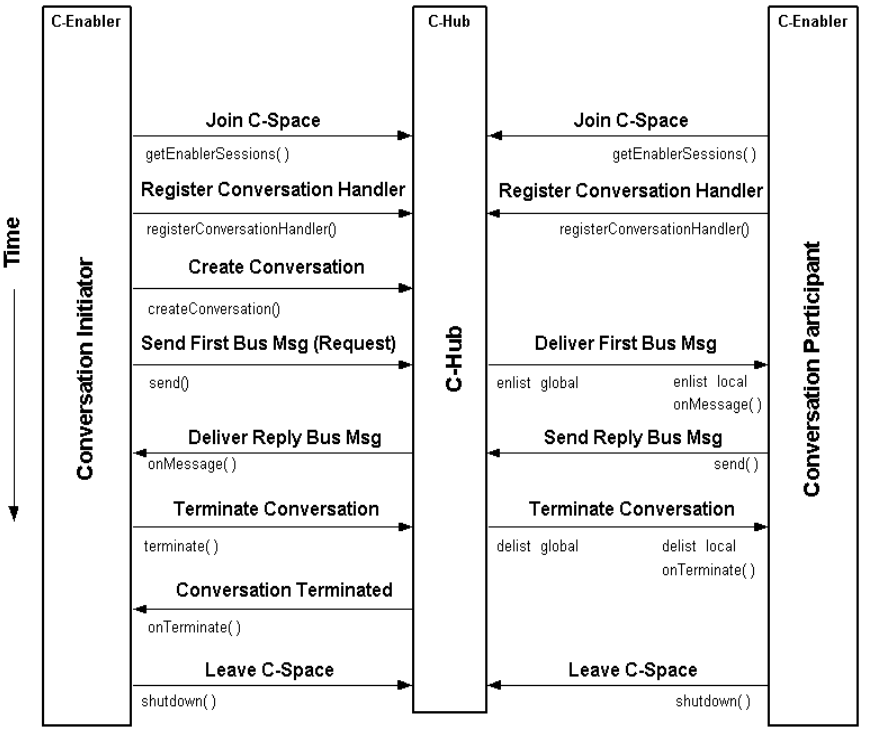
Run-Time Information Flow

At run time, all c-enabler applications perform certain tasks identically: they join a c-space, register conversation handlers, and leave the c-space in the same way. During individual conversations, however, conversation initiators and conversation participants perform a series of distinct, interweaving tasks.

Information Flow Diagram

The following figure shows the run-time information flow between a conversation initiator and a participant.

Figure 1-5 Information Flow Between Conversation Initiator and Participant



This is a simplified example that uses a single conversation and a minimal exchange of business messages (request and reply). In practice, a trading partner may participate in multiple conversations after registering a conversation handler and before leaving the c-space. In addition, within a single conversation, trading partners might exchange many business messages, not just a single request and a single reply.

Steps in the Information Flow

At run time, the flow of information between trading partners (via c-enabler applications communicating through the c-hub) proceeds in the following sequence:

1. Trading partner c-enabler applications join the c-space.
2. Each trading partner c-enabler application registers a conversation handler with the c-enabler session which, in turn (with the help of the local conversation coordinator), registers that trading partner for a given role in a given conversation at the c-hub.
3. The conversation starts when the conversation initiator c-enabler application creates a conversation.
4. The global conversation coordinator adds the conversation instance to its global conversation list and marks the trading partner as the initiator.
5. The local conversation coordinator in the conversation initiator c-enabler node adds the conversation instance to its local conversation list.
6. The conversation initiator's c-enabler application creates and sends a business message (such as a request).
7. The conversation initiator's c-enabler session delivers the business message to the c-hub.
8. The c-hub delivers the business message to the conversation participant's c-enabler node.
9. The global conversation coordinator in the c-hub enlists the participating trading partner in the conversation, adding the participating trading partner to the conversation instance entry in the global conversation list.
10. The local conversation coordinator receives the business message and enlists the trading partner in the conversation locally, adding the conversation instance to the local conversation list.
11. The `onMessage` implementation in the conversation participant c-enabler application is invoked, and the `onMessage` implementation processes the business message.
12. The conversation participant c-enabler application creates and sends a business message (such as a reply) back to the conversation initiator.

13. The c-enabler session on the conversation participant c-enabler node delivers the business message to the c-hub.
14. The c-hub receives the business message and delivers it to the conversation initiator c-enabler node.
15. The conversation initiator c-enabler node receives the business message.
16. The `onMessage` implementation in the conversation initiator c-enabler application is invoked, and the `onMessage` implementation processes the business message.
17. To end the conversation, the conversation initiator c-enabler application terminates the conversation.

    Note: A conversation might terminate automatically if the conversation timeout is exceeded.

18. The local conversation coordinator in the conversation initiator c-enabler node delivers notification of termination to the global conversation coordinator in the c-hub.
19. The global conversation coordinator in the c-hub delists the conversation participant in the global conversation list and delivers notification of termination to the local conversation coordinator on the conversation participant c-enabler node.
20. The local conversation coordinator on the conversation participant c-enabler node receives the termination notification and delists the conversation in the local conversation list.
21. The `onTerminate` implementation in the conversation participation c-enabler application is invoked.
22. The global conversation coordinator in the c-hub marks the conversation terminated and informs the conversation initiator by sending a conversation termination confirmation.
23. The conversation initiator c-enabler node receives the conversation termination confirmation.
24. The local conversation coordinator on the conversation initiator c-enabler node receives the termination notification and delists the conversation in the local conversation list.

25. The `onTerminate` implementation in the conversation initiator c-enabler application is invoked.
26. Trading partner c-enabler applications leave the c-space.

For more information about these steps, see "Key Tasks for C-Enabler Applications" on page 1-12.

2 Programming Steps for C-Enabler Applications

The following sections describe each step in the procedure that a developer usually provides in a c-enabler application:

- Step 1: Import Packages
- Step 2: Implement the ConversationHandler Interface
- Step 3: Create a C-Enabler Session
- Step 4: Register a Conversation Handler
- Step 5: Initiate or Participate in a Conversation
- Step 6: Exchange Business Messages
- Step 7: End the Conversation
- Step 8: Shut Down the C-Enabler Session

Each section includes example code.

Note: You must provide a c-enabler XML configuration file that contains the information required by the c-enabler application at run time. Only one c-enabler XML configuration file exists per c-enabler node. However, the c-enabler XML configuration file can specify configuration information for multiple c-enabler sessions; specifically, it can provide configuration information for each c-space that the associated trading partner joins.

For more information, see "Configuration Requirements" in *Administering BEA WebLogic Collaborate*. In addition, for help in defining the c-enabler XML configuration file, see the comments in the `EnablerConfig.dtd` file in the `dtd` subdirectory of your WebLogic Collaborate installation.

Step 1: Import Packages

C-enabler applications import the required packages from the C-Enabler Class Library. For a description of these packages, see "C-Enabler Class Library" on page 1-3.

The following example listing shows the type of packages that must be imported.

Listing 2-1 Importing Packages

```
import org.w3c.dom.*;
import org.apache.html.dom.*;
import org.apache.xml.serialize.*;
import org.apache.xerces.dom.*;

import com.bea.b2b.protocol.conversation.ConversationType;
import com.bea.b2b.enabler.*;
import com.bea.b2b.enabler.xocp.*;
import com.bea.b2b.protocol.messaging.*;
import com.bea.b2b.protocol.xocp.conversation.local.*;
import com.bea.b2b.protocol.xocp.messaging.*;

import com.bea.eci.logging.*;
```

Step 2: Implement the ConversationHandler Interface

To receive messages, a c-enabler application must implement the following interface:

`com.bea.b2b.protocol.xocp.conversation.local.ConversationHandler`

This interface provides the `onMessage` and `onTerminate` methods that are used to handle incoming business messages and conversation termination notifications, respectively. The `onMessage` method is invoked when the c-enabler receives a business message. The `onTerminate` method is invoked when the c-enabler receives a conversation termination.

The conversation handler is required in order for the trading partner to receive business messages in a conversation. A conversation handler must support at least one conversation type (`com.bea.b2b.protocol.conversation.ConversationType`), which represents a role in a conversation. A c-enabler session supports one conversation handler per conversation type.

Listing 2-2 Implementation of the ConversationHandler Interface

```
public class MyConversationHandler
    implements ConversationHandler{

   private String collaboratorId;

    MyConversationHandler(String collaboratorId){
      this.collaboratorId = collaboratorId;
    }

    public void onMessage(XOCPMessage msg){
System.out.println("onMessage: received for collaborator:" +
collaboratorId );
       Conversation conv = msg.getConversation();
     QualityOfService qos = msg.getQoS();
…
    }

    public void onTerminate(Conversation conv, int result) {
      System.out.println("onTerminate: received for collaborator:"
+ collaboratorId);
    }
}
```

For detailed information about the `ConversationHandler` interface, see the Javadoc on the WebLogic Collaborate documentation CD or in the `classdocs` subdirectory of your WebLogic Collaborate installation.

Step 3: Create a C-Enabler Session

To initiate or participate in conversations, a trading partner creates a c-enabler session on a c-enabler node. Each c-enabler session enables the trading partner to exchange messages with other trading partners in one c-space.

To create a new c-enabler session or to get an existing one, use the `com.bea.b2b.enabler.Enabler` class. The following listing is an example of getting the `session1` c-enabler session, based on the information defined in the c-enabler XML configuration file. Alternatively, an application can get all the c-enabler session definitions from the c-enabler XML configuration file and then create c-enabler sessions as needed.

Listing 2-3 Obtaining a C-Enabler Session

```
Enabler enabler = Enabler.getEnabler("enabler.xml");
EnablerSession es = enabler.getEnablerSession("session1");
// Create all enabler session(s) defined in "enabler.xml"
// EnablerSession[] ess = enabler.getEnablerSessions();
// Optionally, get names of Enabler Sessions
// and use name to create enabler session individually
// String[] sessionNames = enabler.getSessionNames();
// EnablerSession es = null;
```

Step 4: Register a Conversation Handler

To participate in a conversation, a c-enabler application must register a conversation handler. A conversation handler can be associated with multiple conversation types (each type has a conversation name, version and role). A conversation handler can also be shared among multiple conversations. A conversation handler is implemented by the application; the developer is responsible for using it as needed.

To register a conversation handler, a c-enabler application calls the `registerConversationHandler` method on the `XOCPEnablerSession` instance, passing the conversation type and the conversation handler object as parameters.

The following example listing shows how to register a conversation handler for a buyer role (generally a conversation initiator) in the `BuyProcessor` conversation. Note that the specified conversation definition and role must be defined in the spoke and hub repository.

Listing 2-4 Registering a Conversation Handler

```
XOCPEnablerSession session = null;
if(es instanceof XOCPEnablerSession)
   session = (XOCPEnablerSession)es;
MyConversationHandler ch = new
MyConversationHandler(session.getTradingPartner());

ConversationType ctype = new ConversationType("BuyProcessor",
"1.0", "buyer");
ConversationType[] types = { ctype };
session.registerConversationHandler(types, ch);
```

Step 5: Initiate or Participate in a Conversation

A conversation initiator application explicitly starts a conversation. To initiate a conversation, the initiating trading partner calls the `createConversation` method on the `com.bea.b2b.enabler.xocp.XOCPEnablerSession` instance, passing the conversation type and, optionally, the conversation timeout value, in seconds. (The default value is zero, or no timeout, if the configured timeout is also zero in the conversation definition in the spoke and hub repository.) The trading partner must be registered in the initiator role in the conversation definition.

The following example listing shows how a conversation is initiated:

Listing 2-5 Initiating a Conversation

```
ConversationType ctype = new ConversationType("BuyProcessor",
"1.0", "buyer");
Conversation conv = session.createConversation(ctype, 0);
```

Step 6: Exchange Business Messages

After the conversation initiator application has created the conversation, it can begin exchanging business messages with other trading partners in the c-space.

Initially, the conversation initiator application creates and sends a business message (such as a request) to one or more trading partners in the c-space. When a trading partner receives the business message, its conversation participant application processes the business message and (usually) creates and sends a reply business message. The trading partners may send and receive several business messages in the conversation. For more information about exchanging business messages, see "Sending XOCP Business Messages" on page 3-1 and "Receiving XOCP Business Messages" on page 4-1.

Step 7: End the Conversation

A conversation can end after trading partners have finished exchanging business messages in that conversation. The way in whiuch a trading partner ends its involvement in a conversation depends on its role in the conversation.

Participant Leaves a Conversation

Participant trading partners can *leave* a conversation. To leave a conversation, a participant c-enabler application calls the `leave` method on the `Conversation` instance, passing `false`. No messages are retained on the c-hub while the participant is not participating.

Note: In this release, only the `false` argument is supported.

The following example listing shows how a participant leaves a conversation.

Listing 2-6 Leaving a Conversation

```
c.leave(false);
```

Initiator Terminates a Conversation

Conversation initiators can explicitly *terminate* a conversation or wait until the conversation times out. (The conversation initiator can specify a timeout value when it creates the conversation, or it can specify zero to use the timeout value defined for the conversation in the spoke and hub repository.) When a conversation terminates, the conversation initiator and all participating trading partners are delisted from the conversation, any undelivered business messages are discarded, and associated system resources are released.

To terminate a conversation explicitly, the initiating c-enabler application calls the `terminate` method in its implementation of the `Conversation` interface, as shown in the following listing.

Listing 2-7 Terminating a Conversation

```
c.terminate(Conversation.SUCCESS);
```

Step 8: Shut Down the C-Enabler Session

To shut down a c-enabler session and leave the c-space, an application uses the `shutDown` method in its implementation of the `EnablerSession` interface, always passing `false`. The following example listing shows how a c-enabler session is shut down.

Listing 2-8 Shutting Down a C-Enabler Session

```
es.shutDown(false);
```

If a c-enabler application shuts down a c-enabler session, the trading partner leaves the c-space automatically and permanently.

3 Sending XOCP Business Messages

The following sections describe how a c-enabler application sends XOCP business messages to one or more trading partners in a c-space:

- Step 1: Create the Business Message
- Step 2: Specify the Recipients of the Business Message
- Step 3: Specify the Quality of Service for Message Delivery
- Step 4: Send the XOCP Business Message
- Step 5: Check the Delivery Status of the Business Message

To send an XOCP business message, a c-enabler application constructs a business document, creates the business message, specifies the message routing criteria and Quality of Service delivery options, and sends the business message to the c-hub for processing. The c-enabler application can also check the delivery status of the business message, including whether it was successfully delivered. For an introduction to XOCP business messages, see “XOCP Business Messages and Message Envelopes” on page 1-4.

Step 1: Create the Business Message

To create a business message, a c-enabler application first creates the message payload, which consists of any business documents and attachments that the business message will contain. For an introduction to the components of a business message, see "XOCP Business Messages and Message Envelopes" on page 1-4.

Importing the Required Packages

To create a business message, a c-enabler application imports the necessary packages, as shown in the following listing.

Listing 3-1 Importing Packages for Business Message Creation

```
import org.w3c.dom.*;
import org.apache.html.dom.*;
import org.apache.xml.serialize.*;
import org.apache.xerces.dom.*;
import com.bea.b2b.protocol.conversation.ConversationType;
```

Creating Payload Parts

A c-enabler application next creates the message payload, which can include business documents and attachments.

Creating XML Documents

A business message can contain one or more business documents. A business document is the XML-based payload part of a business message. A business document is an instance of the `com.bea.b2b.protocol.messaging.BusinessDocument` class.

A BusinessDocument object contains an XML document, which is an instance of the org.w3c.dom.Document class in the org.w3c.dom package published by the World Wide Web Consortium (www.w3.org). A c-enabler application can also use a third-party implementation of that package, such as the org.apache.xerces.dom package provided by The Apache XML Project (www.apache.org), which is the package used by the Verifier application to create and process XML documents.

Note: The specified document type parameters must map to a part content type of message definition associated with the conversation definition in the repository.

The following listing from the Partner1Servlet of the Verifier application creates a request in the form of an XML document.

Listing 3-2 Creating an XML Document

```
// Create a request document
DOMImplementationImpl domi = new DOMImplementationImpl();
DocumentType dType = domi.createDocumentType("request", null,
"request.dtd");
org.w3c.dom.Document rq = new DocumentImpl(dType);
Element root = rq.createElement("request");
// the actual string data to be processed by the other partner
String sendStr = "ABCDEFGHI";
root.appendChild(rq.createTextNode(sendStr));
rq.appendChild(root);
```

After creating the XML document, a c-enabler application creates a BusinessDocument object, passing the XML document (request) as a parameter to the constructor, as shown in the following listing.

Listing 3-3 Creating a BusinessDocument

```
BusinessDocument bdoc = new BusinessDocument(rq);
```

Creating Attachments

A business message can contain one or more attachments. An attachment is a nonXML-based payload part of a business message that contains binary content. An attachment is an instance of the `com.bea.b2b.protocol.messaging.Attachment` class. For more information, see the WebLogic Collaborate Javadoc.

The following example listing shows how to create an attachment.

Listing 3-4 Creating an Attachment

```
FileInputStream fis = new FileInputStream("somefile");
Attachment att = new Attachment (fis);
```

Creating the XOCP Business Message and Adding Payload Parts

After creating the message payload, a c-enabler application creates the XOCP business message and adds the payload parts to it. The `com.bea.b2b.protocol.xocp.messaging.XOCPMessage` class represents an XOCP business message. For more information, see the WebLogic Collaborate `Javadoc`.

To construct the business message, a c-enabler application:

1. Creates an instance of the `XOCPMessage` class.
2. Adds the payload parts to the business message by calling either of the following methods on the `XOCPMessage` message object:
   - `addPayLoadPart` adds a single business document or attachment to the business message.
   - `addPayLoadParts` adds multiple business documents or attachments to the business message.

In the following listing an XOCP business message is created and payload parts are added to it.

Listing 3-5 Creating a Business Message and Adding Payload Parts

```
XOCPMessage smsg = new XOCPMessage("");
smsg.addPayloadPart(bdoc);
smsg.addPayloadPart(att);
```

Note: The c-enabler application clones `XOCPMessage` content (except its payload parts) before sending it to the c-hub. Therefore, a payload part must not be changed after the application invokes the `send` or `sendAndWait` methods on the `XOCPMessage`.

Step 2: Specify the Recipients of the Business Message

After creating a business message, a c-enabler application optionally specifies the trading partner to which the message will be sent. A c-enabler application might send the business message to a specific trading partner (a point-to-point exchange), such as when it replies to a request received from a conversation initiator. Alternatively, a c-enabler application might send a business message to a set of trading partners (via multicasting) when certain business criteria (represented by c-enabler XPath expressions) are met. For example, an application might sentd a message via multicasting when a buyer sends a bid request to multiple sellers of a particular product.

Either way, the set of eligible trading partners is constrained by those who are subscribed to the appropriate role in the conversation definition. In addition, router and filter expressions defined in the c-hub repository may also affect message delivery to particular trading partners. For more information, see *Administering BEA WebLogic Collaborate*.

Specifying a Particular Trading Partner

If an XOCP business message is being sent to a single, known trading partner, a c-enabler application can call the `setRecipient` method on the `XOCPMessage` object, passing the trading partner name as the parameter. The specified trading partner must be defined in the c-hub repository.

The following example listing shows how a trading partner named `ChipMaker` is specified as the recipient of the business message.

Listing 3-6 Specifying a Single Trading Partner

```
String tradingPartnerName = "ChipMaker";
XOCPMessage msg = new XOCPMessage();
msg.setRecipient(tradingPartnerName);
```

Using `setRecipient` for a business message expedites message delivery because the c-hub does not perform the usual router processing, such as the evaluation of trading partner or c-hub XPath expressions. However, the business message is still subject to applicable filtering in the c-hub. For more information, see *Administering BEA WebLogic Collaborate*.

Using C-Enabler XPath Expressions to Specify Message Recipient Criteria

A c-enabler application can use XPath expressions to specify the criteria for a set of trading partners that are to receive a business message. C-enabler XPath expressions are used to address parts of an XML document. For more information, see *Administering BEA WebLogic Collaborate*.

The XPath expression should be specific to the document format of the c-hub repository and should define a node-specific set of trading-partner elements. The XPath expression selects recipient trading partners based on the following attributes, which are defined in the c-hub repository:

- Standard attributes, such the trading partner name or a postal code
- Extended properties: custom attributes, elements, and text defined by the c-hub administrator

The XPath expression is passed as part of the message header in the business message from the c-enabler to the c-hub. The c-hub uses this XPath expression, along with other XPath expressions defined in the c-hub repository, to determine the set of message recipients for the business message.

If applicable trading partner and c-hub XPath expressions are defined in the c-hub repository, the c-hub evaluates these expressions after it receives the business message. Depending on how they are configured, these XPath expressions might override or append the c-enabler XPath expression that the c-enabler application specifies. For more information, see *Administering BEA WebLogic Collaborate*.

To specify a c-enabler XPath expression for an XOCP business message, the c-enabler application calls the `setExpression` method on the `XOCPMessage` object, passing the XPath expression as the parameter.

Notes: Apache Xalan v 1.0.1 supports single quotes, but not double quotes, to delimit string literals.

Before the business message is delivered, it is still subject to applicable router and filter processing in the c-hub.

Specifying Standard Trading Partner Attributes

The following listing shows a c-enabler XPath expression that selects the trading partner with the specified name.

Listing 3-7 C-Enabler XPath Expression Specifying a Trading Partner Name

```
msg.setExpression("//trading-partner[@name=\'"+
tradingPartnerName+"\']")
```

The following listing shows a c-enabler XPath expression that selects the trading partner whose address contains the string `San`.

Listing 3-8 C-Enabler XPath Expression Specifying a Trading Partner Name

```
msg.setExpression("//trading-partner[contains(address,\'San\')]")
;
```

Specifying a C-Enabler XPath Expression Using Extended Properties

Extended properties are user-defined elements, attributes, and text that can be associated with trading partners in the c-hub repository. These properties provide application extensions to the standard predefined attributes in the repository. The extended property sets are modeled in the repository so that they can be retrieved as subtrees within an XML document. Extended properties are configured on the Trading Partners tab in the WebLogic Collaborate Administration Console. For more information, see *Administering BEA WebLogic Collaborate*.

C-enabler XPath expressions can refer to these extended properties to assist with business message routing. For example, suppose a c-hub administrator adds an extended property called Maximum Order Quantity so that sellers can indicate, in the c-hub repository, the largest quantity that they can accommodate. With this property defined, a buyer with a large order can specify a c-enabler XPath expression that sends the business message only to the sellers that can process the order.

The following code listing shows an XML document generated from the repository with an extended property set for a given seller.

Listing 3-9 Extended Property Set in XML Document Generated from the Repository

```
<c-hub context="message-router">
…
<trading-partner name="ABC Seller"
email="orderprocessing@somedomain.com"
phone="999-999-9999">
<address>123 Main St., San Jose, CA 95131</address>
<extended-property-set name="Capacity">
       <max-order-quantity>1000</max-order-quantity>
</extended-property-set>
</trading-partner>
…
</c-hub>
```

The following listing shows a c-enabler XPath expression that selects trading partners that can accommodate orders larger than 500 units.

Listing 3-10 C-Enabler XPath Expression Specifying an Order Size

```
msg.setExpression("//trading-partner[extended-property-set/(@max-
order-qty > \'500\')]")
```

Because the seller can accommodate orders of up to 1000 units, the seller is selected as a recipient of this business message.

Step 3: Specify the Quality of Service for Message Delivery

The WebLogic Collaborate messaging system allows c-enabler applications to define the *Quality of Service* (QoS), or level of reliability, to use when delivering a business message to recipient trading partners. The Quality of Service settings are stored in the message header of the business message. The messaging system supports the reliable delivery of messages in the event of network-link or node failures. The messaging system provides other capabilities to support reliable messaging, such as message logging and tracking, correlation of messages, delivery retry attempts, message timeouts, and choice of message delivery methods.

Automatic Quality of Service Features

The WebLogic Collaborate messaging system provides certain automatic Quality of Service features that do not require input from c-enabler applications:

- WebLogic Collaborate prevents duplicate message delivery.

- WebLogic Collaborate affixes a timestamp to every business message when it arrives at the c-hub or a c-enabler node. Timestamps can be helpful when taking performance measurements and with debugging applications.

QualityOfService Class

The `com.bea.b2b.protocol.xocp.messaging.QualityOfService` class represents Quality of Service settings for business messages. The `QualityOfService` class defines the quality of service required from the WebLogic Collaborate messaging system to deliver a specific message. It also identifies, to the WebLogic Collaborate messaging system, the c-enabler application's expectation for delivering the business message. A c-enabler application creates an instance of this class, calls methods on this instance to specify various Quality of Service settings, and then calls the `setQoS` method on the message instance, passing the `QualityOfService` object, to associate the settings with the message. If a c-enabler application does not specify Quality of Service settings, the WebLogic Collaborate messaging system uses the default values where applicable.

Quality of Service Settings, Options, and Default Values

The following table describes the available Quality of Service settings, options, and default values.

Table 3-1 Quality of Service Settings, Options, and Default Values

QoS Setting / Description	Options	Default Value(s)
`CONFIRMED_DELIVERY_TO_APPLICATION` ■ Provides confirmation of application delivery up to the receiving application. ■ Provides complete delivery status from each destination, including receipt timestamp, router selected trading partners, final list of recipient trading partners, and so on. ■ Provides complete message tracking information (all potential locations) for the c-hub administrator and the sending c-enabler administrator.	Not Applicable	Not Applicable

Table 3-1 Quality of Service Settings, Options, and Default Values (Continued)

QoS Setting / Description	Options	Default Value(s)
`CONFIRMED_DELIVERY_TO_DESTINATION(S)` ■ Provides the complete delivery status from each destination, including receipt timestamp, router selected trading partners, final list of recipient trading partners, and so on. ■ Provides complete message tracking information (all potential locations) for the c-hub administrator and the sending c-enabler's administrator.	Not applicable	Not applicable
`CONFIRMED_ROUTING` ■ Provides information from the c-hub router about the trading partners selected to receive the business message. ■ Provides message tracking for the sending c-enabler's administrator (until the business message reaches the c-hub router).	Not applicable	Not applicable
`CONFIRMED_DELIVERY_TO_HUB` (Default) ■ Message reached the c-hub. ■ No message tracking for sending c-enabler's administrator.	Not applicable	Not applicable
`DURABILITY`	■ `PERSISTENT` ■ `NON-PERSISTENT`	`NON-PERSISTENT`
`TIMEOUT`	Timeout, in milliseconds, after send	Ignored
`RETRY_ATTEMPTS`	`0-n`	As defined in the c-hub configuration
`CORRELATION_ID`	Application-defined field	Ignored

The following table describes how the Quality of Service setting affects message tracking and delivery acknowledgments.

Table 3-2 QoS, Acknowledgment, and Message Tracking

Quality of Service Setting	Message Tracking (Y/N)?	Acknowledgment (Y/N)?
Confirmed Delivery to Application	Y	Y
Confirmed Delivery to Destination(s)	Y	Y
Confirmed Delivery To Router	Y	N
Confirmed Delivery To C-Hub	N	N

If the Confirmed Delivery to Destination(s) setting is used, then complete message tracking is available and acknowledgments are used to make sure that the message is delivered reliably to its destination(s). If the Confirmed Delivery to Hub setting is used, then no message tracking is available and no acknowledgments are sent from recipient trading partners..

Code Example

The following example listing shows how to set the Quality of Service for a business message.

Listing 3-11 Setting the Quality of Service for a Business Message

```
// Relevant imports
import com.bea.b2b.protocol.messaging.MessageToken;
import com.bea.b2b.protocol.messaging.DeliveryStatus;
import com.bea.b2b.protocol.messaging.BusinessDocument;
import com.bea.b2b.protocol.xocp.conversation.local.*;
import com.bea.b2b.protocol.xocp.messaging.*;
import com.bea.b2b.enabler.*;
import com.bea.b2b.enabler.xocp.*;

XOCPMessage msg = …
// Create QoS object
QualityOfService qos = new QualityOfService();
// Specify message to be persisted
qos.setPersistent(true);
// Specify confirmed delivery to destination(s)
```

```
qos.setConfirmedDeliveryToDestination(true);
msg.setQoS(qos);
```

Setting the Message Delivery Confirmation Level

To specify the level of message delivery confirmation, a c-enabler application calls one of the following methods on the `QualityOfService` instance, passing the Boolean `true` parameter to enable the desired option.

Table 3-3 Message Delivery Confirmation Levels

Durability	Description
`setConfirmedDeliveryToDestination`	Specifies whether to confirm message delivery up to its destination (true) or only up to the c-hub (false).
`setConfirmedDeliveryToHub`	Specifies whether to confirm message delivery up to the c-hub (true) or not (false).
`setConfirmedDeliveryToRouter`	Specifies whether to confirm message delivery up to the router in the c-hub (true) or only up to the c-hub (false).

The following example listing shows how to set the message confirmation level up to its destination.

Listing 3-12 Setting the Message Delivery Confirmation Level

```
qos.setConfirmedDeliveryToDestination(true);
```

For more information about confirming message delivery, see “Step 5: Check the Delivery Status of the Business Message” on page 3-20.

Setting Message Durability

In the WebLogic Collaborate messaging system, message durability is a Quality of Service option that determines whether a durable message store is used in order to guarantee delivery of messages in case of node failures.

Message Durability Options

A c-enabler application has two message durability options: non-persistent (the default) and persistent, as described in the following table.

Table 3-4 Message Durability Options

Durability	Description
Nonpersistent	For nonpersistent QoS, the message is not stored anywhere in a durable data store in the WebLogic Collaborate system while it is in the process of being delivered to its destination. A nonpersistent business message en route to its destination is not recoverable in case of whole or partial system failures. Using this option requires fewer system resources and improves throughput.
Persistent	For persistent QoS, message is persisted to a durable data store while it is in the process of being delivered to its destination. This quality of service increases the guarantee of delivery, as the message is stored in a reliable data store. The message delivery guarantee increases at the expense of throughput of the system. Such a message travels more slowly in the system and consumes more resources. The message is persisted to a data store chosen by the owner of the WebLogic Collaborate component or serialized to a file on disk, based on the size of the message.

Message and Conversation Durability

A c-enabler application can specify message durability on a per-message basis. In addition, message durability can be defined on a per-conversation basis in the c-hub repository.

How business messages are persisted on a per-message or a per-conversation basis depends on a combination of whether persistence is enabled or disabled in the c-hub, the conversation, and the message, as shown in the following table.

Table 3-5 Message Persistence

Persistent Object	**Persistence Enabled?**				
If persistence is enabled (Y) or disabled (N) for:					
■ C-Hub	Y	Y	Y	Y	N
■ Conversation	Y	N	Y	N	Y/N
■ Business Message	Y	N	N	Y	Y/N
Then the conversation or business message is persisted (Y) or not persisted (N):	**Persisted?**				
■ Conversation	Y	N	Y	N	N
■ Business Message	Y	N	Y	N	N

A business message is considered persistent if persistence (recovery) is enabled in the c-hub, if the conversation in which the message is propagated is persistent, and if the message QoS indicates persistence. Even if persistence is enabled for conversations or messages, if persistence is *not* enabled in the c-hub, then no conversations or messages are stored to a reliable data store.

Specifying Message Persistence

To enable message persistence, a c-enabler application calls the `setPersistent` method on the `QualityOfService` instance, passing the Boolean `true` parameter, as shown in the following listing.

Listing 3-13 Specifying Message Persistence

```
qos.setPersistent(true);
```

Setting the Message Timeout

If specified, the message timeout determines how long a sender waits for acknowledgments. If a business message expires (times out), the receiver of the business message does not process it, and all other processing of the business message, including acknowledgment processing and delivery retries, is abandoned.

Timeout Algorithm

WebLogic Collaborate does not synchronize the clocks used by its different components, which can reside in different machines at different locations. Instead, WebLogic Collaborate uses a relative time algorithm.

Based on this algorithm, the time left before the timeout of a business message (relative to the absolute time of the component processing the business message) is included in the business message when the business message is sent to the other component.

In the receiving component, the timeout calculations are based on an absolute time (at the arrival of the business message) and a relative time (embedded in the incoming message) left to process the message. This algorithm at least ensures that the actual message timeout in the system always occurs after the original timeout specified by the application.

```
Message Timeout on the C-Hub = Message timeout specified by the
c-enabler application when sending a message
```

```
Message Timeout on the Sending C-Enabler = Message Timeout on the
C-Hub + N x Delta
```

In these settings:

- *N* = A predefined number in the system, such as `10`
- *Delta* = Estimated amount of time required for a message to travel, round-trip, between the sending c-enabler and the c-hub

Setting the Number of Delivery Retry Attempts

If an attempt to deliver a business message fails due to intermittent network failures, the WebLogic Collaborate messaging system attempts to retry sending the business message repeatedly until one of the following occurs:

- The business message is delivered (that is, delivery succeeds).
- The number of retry attempts is exceeded.
- The message times out.
- The conversation in which the business message is sent either terminates or times out.

The default values for message timeouts and retry intervals are defined in the c-hub repository and are retrieved by a c-enabler when the c-enabler session is created. The WebLogic Collaborate messaging system waits for the configured interval before attempting to resend a business message.

To override the default retry attempt limit, a c-enabler application calls the `setTimeout` method on the `QualityOfService` instance, passing the timeout value (number of milliseconds) as a parameter, as shown in the following listing.

Listing 3-14 Specifying the Message Timeout

```
qos.setTimeout(10000);
```

Setting the Correlation ID for a Business Message

A c-enabler application can specify a unique correlation ID for a business message so that it can correlate received business messages (such as replies to a request) from trading partners to a previously sent message (such as a request). The correlation ID accompanies the business message to its destination. The c-enabler application of the recipient trading partner can use this value to unambiguously identify the reply message sent back to the originating trading partner.

To specify the correlation ID, a c-enabler application calls the `setCorrelationId` method on the `QualityOfService` instance, passing a string representing the correlation ID as a parameter, as shown in the following listing.

Listing 3-15 Specifying the Correlation ID for a Business Message

```
qos.setCorrelationId("ABC123");
```

Step 4: Send the XOCP Business Message

After specifying the recipients of a business message and the Quality of Service, a c-enabler application sends the business message in one of the following ways:

- Synchronous message delivery
- Deferred synchronous message delivery

Synchronous Message Delivery

With synchronous message delivery, the application waits until the sent message is delivered to the destination(s). The WebLogic Collaborate messaging system returns control to the application once the outcome of the activity of sending the message is known. The application waits until any of the following events occurs:

- Acknowledgments are received from all potential destinations.
- The message times out.
- The conversation in which the message was sent terminates.

To send a business message synchronously, a c-enabler application calls the `sendAndWait` method on the `XOCPMessage` instance, passing the amount of time to wait (in milliseconds) as a parameter. If zero (0) is specified, the c-enabler application waits until the business message reaches its destination(s), as shown in the following listing.

Listing 3-16 Sending a Message Using Synchronous Message Delivery

```
MessageToken token = msg.sendAndWait(0);
```

Deferred Synchronous Message Delivery

With deferred synchronous message delivery, the WebLogic Collaborate messaging system returns control to the c-enabler application immediately after a message is sent, and returns a message token that the c-enabler application can use to check the status of message delivery. Once a message token is accessed, the application waits for a specified time or until any of the following events occurs:

- Acknowledgments are received from all potential destinations.
- The message times out.
- The conversation in which the message was sent either terminates or times out.

To send a business message with deferred synchronous message delivery, a c-enabler application calls the `send` method on the `XOCPMessage` instance, continues executing business logic, and then checks the status by calling the `waitForACK` method on the `MessageToken` instance, as shown in the following listing.

Listing 3-17 Sending a Message Using Deferred Synchronous Message Delivery

```
token = msg.send();
...
token.waitForACK();
```

The `waitForAck` method blocks until the status of the business message is available (if no timeout is specified) or until the specified timeout (in milliseconds) is exceeded.

Step 5: Check the Delivery Status of the Business Message

Both the `send` and `sendAndWait` methods on the `XOCPMessage` instance return a message token that a c-enabler application can query to check the delivery status of the associated business message.

Message Tokens

A message token is an instance of the `com.bea.b2b.protocol.xocp.messaging.XOCPMessageToken` class. A message token has the following attributes.

Table 3-6 Message Token Information

Attribute	Description
Message ID	Unique ID of the business message.
Exception	If applicable, any exception that occurred before the business message left the sending c-enabler. An exception is usually returned when the message is sent, but for deferred synchronous message delivery, the business message might be kept in an internal send queue temporarily before being delivered to the c-hub.
Elapsed Time	Time taken to deliver the business message to all destination(s). This information is available only after acknowledgments have been received from all message destinations. Availability is subject to the specified Quality of Service delivery option.
Delivery Status	Delivery status from recipient destination(s). This information depends on the availability of such information. Availability is subject to the specified Quality of Service delivery option.

Table 3-6 Message Token Information (Continued)

Attribute	Description
Number of Recipients (Router)	Number of recipient trading partners after the business message has been processed by the XOCP router in the c-hub. Availability is subject to the specified Quality of Service delivery option.
Number of Recipients (Filter)	Number of recipient trading partners after the business message has been processed by the XOCP filter in the c-hub. Availability is subject to the specified Quality of Service delivery option.

If the business message was sent using the synchronous send delivery option, then the message token cannot be used to wait for acknowledgments. If used, the method returns immediately.

Delivery Status Tracking

In the WebLogic Collaborate messaging system, when a business message reaches its destination (the receive queue of the destination c-enabler node), a system message is returned to the sender to acknowledge the message delivery if the Quality of Service setting requires it.

A c-enabler application can use either of the following methods to obtain the delivery status:

- `getAllDeliveryStatus` if the business message was sent to multiple recipients
- `getDeliveryStatus` if the business message was sent to a single recipient

Both methods return a `DeliveryStatus` object, an instance of the `com.bea.b2b.protocol.messaging.DeliveryStatus` class that provides the following information:

- Recipient (name of the recipient trading partner or message tracking location)
- Timestamp of the receipt of the business message
- Status code, valid values for which are shown in the following table

Table 3-7 Message Delivery Status Codes

Status Code	Description
SUCCESS	Business message was successfully delivered to the destination. No errors or exceptions occurred.
FAILURE	An error occurred during delivery of the business message to this destination.
RETRIES_EXHAUSTED	All delivery retry attempts have been exhausted and the business message has been discarded.
TIMEDOUT	Timeout occurred before message delivery and the business message has been discarded.

Message Tracking Locations

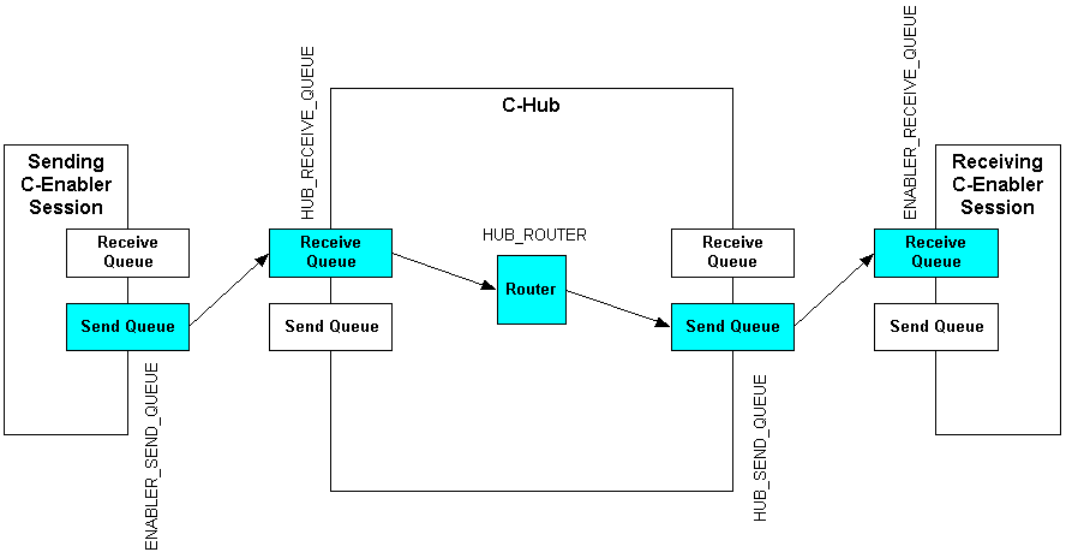
The WebLogic Collaborate messaging system provides message tracking features that allow c-hub and c-enabler administrators to check the progress of a business message as it moves through various predefined message tracking locations along the message path en route to its destination. The WebLogic Collaborate Administration Console can display status information if a business message passes through these tracking points. Administrators can use message tracking information for debugging and to identify bottlenecks in applications.

Note: The availability of message tracking locations depends on the configuration of the WebLogic Collaborate system and the specified Quality of Service for a given business message (such as CONFIRMED_DELIVERY_TO_DESTINATION(S), which is described in Table 3-1). For example, if the c-enabler and c-hub are collocated on the same node, some locations are not available. Similarly, some locations may not be available for synchronous message delivery.

Diagram of Message Tracking Locations

The following figure shows the message tracking locations in the WebLogic Collaborate messaging system.

Figure 3-1 Message Tracking Locations



Description of Message Tracking Locations

The following message tracking locations are potentially visible in the WebLogic Collaborate Administration Console.

Table 3-8 Message Tracking Locations

Message Tracking Locations	Location	Activity Performed
ENABLER_SEND_QUEUE	Send queue in the c-enabler session of the sending trading partner.	Message is enqueued for sending.

Table 3-8 Message Tracking Locations (Continued)

Message Tracking Locations	Location	Activity Performed
HUB_RECEIVE_QUEUE	Receive queue for the sending trading partner in the c-hub	Message is enqueued in the receive queue of the trading partner at the c-hub.
HUB_ROUTER	XOCP-Router in the c-hub	Message has reached the router.
HUB_SEND_QUEUE	Send queue of the receiving trading partner in the c-hub	Message has been enqueued for delivery in the target trading partner's queue at the c-hub.
ENABLER_RECEIVE_QUEUE	Receive queue in the c-enabler session of the receiving trading partner	Message has been enqueued in the queue of the listener thread of the target trading partner's c-enabler session.

4 Receiving XOCP Business Messages

The following sections describe how to receive XOCP business messages in a c-enabler application:

About Receiving XOCP Business Messages

C-enabler applications must implement the `onMessage` method in the `ConversationHandler` interface to receive and process business messages. The `onMessage` method has the following signature.

Listing 4-1 Signature for onMessage Method

```
public void onMessage(XOCPMessage msg)
```

The c-enabler session invokes the `onMessage` method whenever a c-enabler receives a business message, passing the business message as an input parameter. The c-enabler application retrieves the `XOCPMessage` object containing the business message and then calls methods on that instance to process the message.

If a c-enabler application receives multiple business documents in a conversation, the `onMessage` implementation first determines the type of document received (such as a bid request or bid reward), and then processes that document accordingly.

In addition, the `onMessage` implementation might contain code that constructs and sends a business message. For example, a conversation participant c-enabler application might implement `onMessage` to receive a request, process the request, and then create and send the reply document.

Receiving an XOCP Business Message

Listing 4-2 describes the `onMessage` implementation in the `Partner2Servlet` of the Verifier application. This `onMessage` implementation processes the initial business document (a request) sent from the `Partner1Servlet`. It then creates and sends a reply document back to the Partner1 node.

Tasks Performed

The `onMessage` code performs the following tasks:

1. Retrieves the Quality of Service for the business message by calling the `getQoS` method on the `XOCPMessage` instance.

   The application uses the same Quality of Service settings to send the reply message.

2. Retrieves the payload parts of the business message by calling the `getPayloadParts` method on the `XOCPMessage` instance.

3. Retrieves the first (and only) business document in the `PayloadPart[]` array.

4. Extracts the associated XML document by calling the `getDocument` method on the `BusinessDocument` instance.

5. Retrieves and examines parts of the XML document using methods on the `Document` instance, which is an instance of the `org.w3c.dom.Document` class provided in the `org.w3c.dom` package published by the World Wide Web Consortium (`www.w3.org`).

   A c-enabler application can also use a third party implementation of that package, such as the `org.apache.xerces.dom` package provided by The Apache XML Project (`www.apache.org`), which is what the Verifier application uses to create and process business documents.

6. Retrieves the data string (`"ABCDEFGHI"`) embedded in the business document and converts it to all lowercase letters.

7. Constructs a reply document, specifies the same Quality of Service as the request document, and sends the document to Trading Partner 1.

Code Listing

The following listing is the `onMessage` implementation in the `Partner2Servlet` of the Verifier application.

Listing 4-2 onMessage Implementation in Partner2Servlet

```
public void onMessage(XOCPMessage rmsg) {
      try{
        QualityOfService qos = rmsg.getQoS();

        PayloadPart[] payload = rmsg.getPayloadParts();
        Document rq = null;

        if (payload != null && payload.length > 0){
          BusinessDocument bd = (BusinessDocument)payload[0];
          rq = bd.getDocument();
        }
        if (rq == null){
          throw new Exception("Did not get a request document");
        }
        Conversation conv = rmsg.getConversation();

        Element root = rq.getDocumentElement();
        String name = root.getNodeName();
        if (!name.equals("request")){
```

```
        debug("Received "+name+" instead of a request");
        return;
      }
      Text revStr = (Text)root.getFirstChild();

      // Create the return document
      DOMImplementationImpl domi = new DOMImplementationImpl();
     DocumentType dType = domi.createDocumentType("reply", null, "reply.dtd");
      rq = new DocumentImpl(dType);
      root = rq.createElement("reply");
      String sendStr = new String(revStr.getData());
      root.appendChild(rq.createTextNode(sendStr.toLowerCase()));
      rq.appendChild(root);

      XOCPMessage smsg = new XOCPMessage("");
      smsg.addPayloadPart(new BusinessDocument(rq));
      smsg.setQoS(qos);
      smsg.setExpression("//trading-partner[@name=\'Partner1\']");

      smsg.setConversation(conv);
      smsg.sendAndWait(0);

    }catch(Exception e){
      e.printStackTrace();
    }
  }
```

Index

D

E

G

I

J

L

M

P

Q

R

S

T

X