

Public Health Information Network Messaging System Server Installation Guide



Prepared by
U.S. Department of Health & Human Services June 2003

Table of Contents

Table of Contents	1
Introduction	5
Installing the Message Receiver	6
To Install Java Runtime Environment	6
Installing Tomcat 4.04	7
To Install Tomcat 4.04.....	8
To Start Tomcat.....	8
To Stop Tomcat.....	9
To Test Your Tomcat Installation.....	9
To Configure SSL for Tomcat	9
Installing the Message Receiver Servlet.....	11
To Install and Configure the Message Receiver	12
To Set Up the Message Receiver Directory.....	13
Installing the File-Based Message Handler	13
To Install the File-Based Message Handler	14
To Create a Service Map for the Message Handler	15
Configuration Overview	16
Message Receiver and File-Based Message Handler Configuration Files	17
Configuring the web.xml File.....	18
Configuring the receiver.xml File.....	19
Variables in the receiver.xml File	21
Database Pooling.	24
Database Tag Values.....	24
Configuring the servicemap.xml File	26
Configuring the Collaboration Protocol Agreement.....	26
PartyInfo Segments	27
Message Sender PartyInfo	27
Message Receiver PartyInfo	27
Transport Sub-segment	27
Authentication Types	28
Configuring Message Handlers	29
Asynchronous and Synchronous Message Handling.....	29
Configuring the Service Map.....	30

Configuring Queues	32
Configuring the Service Map.....	33
Configuring the Queue Map	34
Worker Queue Table Schema.....	35
To Configure the Queues	37
Installing the Route-Not-Read Message Handler	38
Route-not-Read Servlet.....	38
Tables.....	38
Configuration Files	38
Messagebins Table.....	38
Broadcast Table.....	39
Users Table.....	39
Partyid_user Table	39
Security.....	41
Java Keystores	41
To Manage the PKCS12 KeyStore using Internet Explorer	42
Managing the Java Trust Store with Keytool	43
Managing Passwords	46
IReceiver Password File.....	46
Password Utilities	47
Password Based Encryption (pbe)	47
Substitution Cipher Key Generator.....	49
Authentication.....	49
Encryption.....	51
Enabling SSL Authentication	51
Encryption.....	52
Enabling Encryption	53
Digital Signatures.....	54
Generating Digital Signatures.....	54
Verifying Digital Signatures	54
To Enable Digital Signatures	56
Enabling Route-not-Read Encryption.....	57
Security Best Practices.....	58
System Maintenance	60
Setting Log Levels	60
Log File Maintenance	60

Integration Issues	62
Message Receiver Interface	62
Example Message Handler Servlet	63
Appendix	68
Appendix A.....	68
Worker Queue Generation Scripts	68
Appendix B.	70
Transport Queue Generation Scripts.....	70
Appendix C.	73
Worker Queue Generation Scripts	73
Appendix D.....	75
Route-not-Read SQL Generation Scripts.....	75
Appendix E.	79
File-Based Message Queue.....	79
Appendix F.....	81
Transport Level Status and Error Codes	81
Appendix G.	82
Example receiver.xml File	82
Appendix H.....	83
Example Collaborative Protocol Agreement (CPA).....	83
Appendix I.	85
Example Receiver Password File.....	85

Introduction

The Public Health Information Network Messaging System Server Installation Guide provides step-by-step procedures for the system administrator to install and configure the Message Receiver server software for the Public Health Information Network Messaging System. The procedures include installing and configuring the:

- Java Virtual Machine
- Java Application Server
- ebXML server software

The configuration details of ebXML features such as data encryption, data signing and authentication schemes, as well as the system administration tasks, such as log file maintenance, are also included in this guide. In addition, the Message Handler Development section provides instructions for building custom message handlers.

To use this guide to install and maintain The Public Health Information Network Messaging System's Message Receiver software, you need to be familiar with:

- Server system administration for Windows or Unix
- Java
- Web application servers
- Database Connectivity
- Web security

Installing the Message Receiver

Before you install the Message Server software, install the Message Sender software. See *The Public Health Information Network Messaging System Client Installation Guide* for details.

To install the Message Receiver do the following procedures. Step-by-step instructions are included.

1. Install Java Runtime Environment.
2. Install Tomcat 4.04.
3. Start and stop Tomcat.
4. Test Tomcat Installation.
5. Configure SSL for Tomcat.
6. Install and configure the Message Receiver.
7. Set up the Message Receiver Directory.
8. (Optional) Install the File-Based Message Handler.
9. Create a Service Map for the Message Handler.

To Install Java Runtime Environment

For Windows:

1. Copy the **j2re-1_4_0_03-windows-i586.exe** file from <XXXXXX> on the Public Health Messaging Software's CD into a temporary location on disk. If the directory doesn't exist create it.
2. Double-click the installer's icon and follow the instructions.
3. Type <installedPath>\java\j2re1.4.0_03 for the destination folder and then click **Next**.
4. Select **Microsoft Internet Explorer Java Plug-in (optional)** and then click **Next**.

Note You must have administrative permissions to install the Java 2 JRE on Microsoft Windows 2000 and XP

5. When you are finished with the installation you can delete the download file to recover disk space.

For Linux:

1. Copy the **j2re-1_4_0_03-linux-i586.bin** file from **<XXXXXX>** on the Public Health Messaging Software's CD to **<installedPath>/java** on disk. If the directory doesn't exist create it.
2. Type the following command to start the installer's script:
`./<installedPath>/java/j2re-1_4_0_03-linux-i586.bin`
3. Follow the instructions.
4. A message appears: **"Do you agree to the above license terms?"** Type **yes** and then press **Enter**.
5. When you are finished with the installation you can delete the download file to recover disk space.

Note All Java configurations are handles via environment and startup scripts, which eliminates potential conflicts with multiple versions of Java installed on the server.

Installing Tomcat 4.04

Tomcat 4.04 is a J2EE compliant server that implements the Servlet 2.3 and JSP 1.2 specifications. The Public Health Information Network Message System, PHINMS, comes with Tomcat version 4.04. PHINMS requires a J2EE compliant application server. To obtain a list of approved application servers contact your PHINMS representative. For any additional questions about Tomcat configuration see the on-line documentation at:

<http://jakarta.apache.org/tomcat/tomcat-4.0-doc/index.html>

To Install Tomcat 4.04

The following procedure shows you how to install Tomcat 4.04.

1. Copy the **jakarta-tomcat-4.0.4.zip** file from **<XXXXXX>** on the PHINMS software CD to **<installedPath>** location on disk. The directory structure **jakarta-tomcat-4.0.4** is be created when you expand the file.
2. Using **WinZip** or **PKZIP** extract **jakarta-tomcat-4.0.4.zip** into you **<installedPath>** directory.
3. For Windows, edit the **catalina.bat** file and for Linux edit the **catalina.sh** file in **<installedPath>\jakarta-tomcat-4.0.4\bin**. Modify the following line to reference your installation of the Java Runtime Environment. If the entry doesn't exist, add it.

For Windows: set JAVA_HOME=<installedPath>\java\j2re1.4.0_03

For Linux: JAVA_HOME=<installedPath>/java/j2re1.4.0_03

4. The basic Tomcat 4.04 installation is now complete. Follow the steps in **To Start Tomcat.**

To Start Tomcat

Execute the following script to start Tomcat 4.04.

For Windows:

```
<installedPath>\jakarta-tomcat-4.0.4\bin\catalina.bat start
```

For Linux:

```
<installedPath>/jakarta-tomcat-4.0.4/bin/catalina.sh start
```

Follow the steps in **To Stop Tomcat.**

To Stop Tomcat

Execute the following script to stop Tomcat 4.0.

For Windows:

```
<installedPath>\jakarta-tomcat-4.0.5\bin\catalina.bat stop
```

For Linux:

```
<installedPath>/jakarta-tomcat-4.0.4/bin/catalina.sh stop
```

Follow the steps in **To Test Your Installation**.

To Test Your Tomcat Installation

Go to the following URL. If the page displays, your installation of Tomcat 4.04 was successful:

```
http://localhost:8080/
```

For more information about configuring and running Tomcat 4.04 go to the following web site:

<http://jakarta.apache.org/tomcat/>

Follow the steps in **To Configure SSL for Tomcat**.

To Configure SSL for Tomcat

To configure SSL support on Tomcat 4.04, do the following:

1. Execute the following command to create a certificate keystore:

For Windows:

```
<installedPath>\j2re1.4.0_03\bin\keytool -genkey -alias tomcat -  
keyalg RSA \  
-keystore <installedPath>\keystores\tomcat
```

For Linux:

```
<installedPath>\j2re1.4.0_03/bin/keytool -genkey -alias tomcat -  
keyalg RSA \  
-keystore <installedPath>\keystore\tomcat
```

2. To specify a different location or filename, add the **-keystore** parameter, followed by the complete pathname to your **keystore** file, to the **keytool** command in step 1.
3. To simplify keystore management, keep all keystores in the directory **<installedPath>/keystores**. Put this new location in the **server.xml** configuration file as described below. If the **-keystore** parameter is omitted, the keystore will be created under the JRE security directory.

Note : Use the **keytool** utility to create a certificate only during testing. You need to get an official certificate from a certified certificate authority (CA) when you move your application to a production environment.

4. After executing the command in step 1, you will be prompted for the keystore password. The default password is **changeit**. All letters are lower case. You can create a custom password if you want. If you create a custom password, you need to specify the it in the **server.xml** configuration file as described later.
5. You will be prompted for general information about the certificate, such as company, contact name, and so on. This information is displayed to users who access a secure page in your application, so make sure the information is complete, clear and specific. Do not use abbreviations. Completely spell the names of cities, States and so on.
6. You will be prompted for the **key password**, the password specifically for this Certificate. (Other certificates may be stored in the same keystore file.) The **key password** must be the same as the **keystore** password.

You now have a **keystore** file with a Certificate that can be used by your server.

Note The certificate will be extracted and loaded into the sender's **TtrustedStore keystore**. See the Client Installation and Configuration guide for details.

7. Uncomment the **SSL HTTP/1.1 Connector** entry in **<installedPath>/conf/server.xml**.

```
<!--  
  <Connector  
  className="org.apache.catalina.connector.http.HttpConnector"  
    port="8443" minProcessors="5" maxProcessors="75"  
    enableLookups="true"  
    acceptCount="10" debug="0" scheme="https" secure="true">  
    <Factory  
  className="org.apache.catalina.net.SSLServerSocketFactory"  
    clientAuth="false" protocol="TLS"  
    keystoreFile="<installedPath>/keystores/tomcat"/>  
</Connector>  
-->
```

8. Restart the Tomcat 4.04 server using the stop and start procedures previously listed.
9. Test the certificate installation by browsing the following URL:
https://localhost:8443/

Because you generated a test certificate without using a Certificate Authority known by the browser, a pop-up window warning appears which reads **Security Alert** and asks if you want to proceed. Accept this certificate to display the secure page.

After you have received a valid certificate from a known Certificate Authority this warning will not be displayed. When the Tomcat 4.04 home page is displayed you have successfully installed SSL within Tomcat and the Tomcat 4.0 installation is complete.

Installing the Message Receiver Servlet

The message receiver is a servlet that runs on a J2EE compliant application server. The message receiver receives the ebXML message and after processing the message envelope, performs message decryption and signature verification. After verification is complete, the Message Receiver will either write the message to a queue or forward the message payload onto an appropriate message handler.

To Install and Configure the Message Receiver

To install and configure the Message Receiver do the following:

1. Copy the **ebxml.war** file from <XXXXXX> on the Public Health Messaging Software's CD to the <installedPath>\jakarta-tomcat-4.0.4\webapp directory. Tomcat 4.04 will automatically expand the war file in a newly created **ebxml** directory under **webapps**.
2. Edit the **web.xml** file in <installedPath>\jakarta-tomcat-4.0.4\webapps\ebxml\WEB-INF. Modify the <param-value> to point to the message receiver's configuration file. This file is usually found in the <installedPath>\config\ directory.

```
<servlet>
  <servlet-name>
    receiver
  </servlet-name>
  <servlet-class>
    gov.cdc.nedss.services.filerecv.ReceiveFileServlet
  </servlet-class>
  <init-param>
    <param-name>receiverConfig</param-name>
    <param-value>{installedPath}\config\receiver.xml</param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>

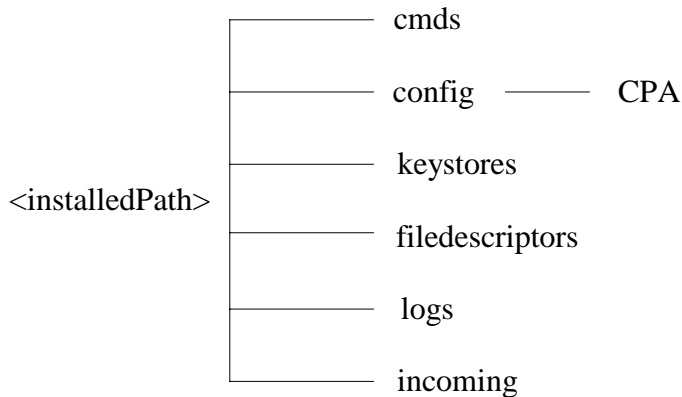
<servlet-mapping>
  <servlet-name>
    receiver
  </servlet-name>
  <url-pattern>
    /receiver
  </url-pattern>
</servlet-mapping>
```

3. The **web.xml** file contains a servlet-mapping entry for the message receiver. The URL default value is **/receiver**. Update this value if you require a different URL mapping.
4. Follow the instructions in **To Set Up the Message Receiver Directory**.

Note: The **receiver.xml** file is described in detail later in this document.

To Set Up the Message Receiver Directory

Use the following structure for the message receiver directory:



Directory	Description
cmds	Includes utilities required to support the operations of the system such as the password cipher and pbe encryption programs.
config	Includes all the configuration scripts. Also includes the CPA directory, which contains all the Collaborative Protocol Agreements.
keystores	Contains all the keystore files that support the encryption.
filedescriptors	Contains the file descriptors files used in file-based transfers.
logs	Contains the log files.
incoming	Contains the files that have been received in a file-based transfer.

Installing the File-Based Message Handler

You do not have to install the file-based Message Handler. It is optional. It is delivered with the Public Health Information Network Messaging System as an example Message Handle and you can use it to test the installation of the Message Receiver.

The file-based Message Handler receives the message from the Message Receiver and writes the payload to a file. The Message Handler should be installed on the same computer as the Message Receiver to eliminate an intruder's ability to eavesdrop on communication between them. Enforce firewall rules to protect the Message Handler from direct access from any internal or external computer.

To Install the File-Based Message Handler

To install and configure the Message Handler do the following:

1. Copy the **messagehandler.war** file from <XXXXXX> on the PHINMS software CD to the <installedPath>\jakarta-tomcat-4.0.4\webapp directory. Tomcat automatically expands the war file in a new **messagehandler** directory under **webapps**.
2. Edit the **web.xml** file in <installedPath>jakarta-tomcat-4.0.4\webapps\messagehandler\WEB-INF. Modify the <param-value> to point to the Message Receiver's configuration file, which is usually in the "<installedPath>\config\" directory.

```
<servlet>
  <servlet-name>
    messagehandler
  </servlet-name>
  <servlet-class>
    gov.cdc.nedss.services.messagehandler.MessageHandler
  </servlet-class>
  <init-param>
    <param-name>receiverConfig</param-name>
    <param-value>{installPath}\config\receiver.xml</param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>
```

The **web.xml** file contains a servlet-mapping entry for the Message Handler. The default value of the URL is **/messagehandler**. This URL will be used in the route map to route an incoming service/action to this Message Handler.

```
<servlet-mapping>
  <servlet-name>
    messagehandler
  </servlet-name>
  <url-pattern>
    /messagehandler
  </url-pattern>
</servlet-mapping>
```

3. After installing and configuring the Message Handler, add an entry to the Message Receiver's **servicemap.xml** file to route messages to the Message Handler. Use the **To Create a Service Map for the Message Handler** procedures in the following section.

To Create a Service Map for the Message Handler

The **servicemap.xml** file maps an incoming service/action to specific Message Handlers. You need to add a service entry for the file-based Message Receiver. Do the following procedure to create a service map for the Message Handler servlet.

1. Add a unique name for your service using the **<Name>** tag. This name is used to log information in the log files.
2. Add a type for your service using the **<Type>** tag.

The **Type** attribute tells the Message Receiver the type of Message Handler. The Message Receiver supports web-based Message Handlers such as servlets, JSPs, ASPs and CGI. The Message Receiver establishes a HTTP connection with the web Message Handler. For more details see the **Configuring Queues** section in this document.

3. Add an action name using the **<Name>** tag within the **<Action>** tag.

The Message Receiver uses the **<Name>** tag in the **servicemap.xml** file to an action name to an associated action URL.

4. Add an URL using the **<Url>** tag within the **<Action>** tag.

This tag defines the URL of the Message Handler that will be initiated when this Action is requested. In the following example a request with an action of **receiveFile** will be mapped to the Servlet's URL, <http://server:port/messagehandler>.

```
<Service>
  <Name>FileReceive</Name>
  <Type>Servlet</Type>
  <Action>
    <Name>receiveFile</Name>
    <Url>http://server:port/messagehandler</Url>
    <Argument> </Argument>
  </Action>
</Service>
```

5. Optionally you can add an argument to your **Action** using the **<Argument>** tag. It is used to pass name-value-based parameters to the Message Handler.

Configuration Overview

This section provides an overview of the configuration of the Public Health Information Network Messaging System, describes the software components and explains how to configure the components to accept and handle messages.

The main component of the system is the Message Receiver. The receiver is a servlet that accepts incoming messages. After the servlet has accepted a message it will write the message directly into a worker queue or it will forward the message to a Message Handler.

The Message Handler receives the message from the Message Receiver and performs a task on the payload. The task performed by the Message Handler may be simple such as writing the payload to disk or complex such as calling a data transformation engine to insert data into a public health information system.

A file-based Message Handler is delivered with the Public Health Information Network Messaging System. The file-based Message Handler is a servlet that writes the payload directly to disk. This simple Message Handler is used throughout the configurations in this section

The file-based Message Handler provides a straightforward method to test the installation and configuration of the system. You can test features in the system incrementally by using the file-based Message Handler. In a production environment, you do not have to use the file-based Message Handler. It is optional.

Message Receiver and File-Based Message Handler Configuration Files

Configurations for the Message Receiver and file-based Message Handler are maintained in five XML configuration files.

File Name	Description
web.xml	A standard Java configuration file for web modules. The Message Receiver servlet and servlet mapping entries may need modification to reflect specific environment settings such as install directory. Also, security constraint tags may need to be created if the web container security is implemented.
receiver.xml	The main configuration file for the Message Receiver. It contains references to application directories, settings, and certificates.
servicemap.xml	Maps incoming service requests with specified Message Handlers.
queuemap.xml	Maps a queue to a database and its associated queue table. Configuration of the queuemap.xml file is optional. It must be configured only when the Message Receiver is configured to write messages into queues.
passwds.xml	Contains system passwords including passwords to access certificates in keystores. The passwds.xml file is encrypted before deployment on a production server. See the Password Utilities section in this document for more information on password encryption.

After configuring the software to accept and handle messages, configure the Collaboration Protocol Agreements (CPAs) for each partner that will send messages to the system.

Configuring the web.xml File

The **web.xml** file describes the Message Receiver's servlet attributes, mappings and, optionally, security constraints. The **web.xml** file is packaged within the **ebxml.war** file. In a Tomcat deployment, the file can be found under Tomcat's **webapps** directory in the expanded **/ebxml/WEB-INF** directory.

The deployed **web.xml** file contains a servlet entry for the Message Receiver. This entry maps a servlet name, **receivefile**, to the servlet class, **gov.cdc.nedss.services.filerecv.ReceiveFileServlet**. This is the actual Java class that will be instantiated by the web container. The Message Receiver servlet entry contains a parameter called **receiverConfig**. This parameter points to the Message Receivers' configuration file, **receiver.xml**.

```
<servlet>
  <servlet-name>
    receivefile
  </servlet-name>
  <servlet-class>
    gov.cdc.nedss.services.filerecv.ReceiveFileServlet
  </servlet-class>
  <init-param>
    <param-name>receiverConfig</param-name>
    <param-value>C:/tomcat4/ebxml/Config/receiver.xml</param-value>
  </init-param>
  <load-on-startup>2</load-on-startup>
</servlet>
```

Servlet Mapping Entry

The **web.xml** file contains a servlet-mapping entry that maps a URL to a Message Receiver servlet. The server's end point for a message exchange will consist of protocol, server name, port and the URL entry in the **web.xml** servlet mapping. The default value is **receivefile**. Set the load on the startup parameter to a number greater than the Message Handler load on startup parameters. This allows servlet-based Message Handlers to initialize before the Message Receiver initializes.

```
<servlet-mapping>
  <servlet-name>
    receivefile
  </servlet-name>
  <url-pattern>
    /receivefile
  </url-pattern>
</servlet-mapping>
```

Security Constraints

Optionally, security constraints can be added to the **web.xml** file to support basic, form and certificate authentication. Ideally, use the web proxy layer to manage authorization. This level of abstraction ensures system integrity by isolating the Message Receiver (ebXML Receiver) layer. If you are unable to place the ebXML receiver software behind a web proxy and require configuration of authorization parameters at the application server layer, see Tomcat documentation to learn more about Tomcat authentication configuration.

Configuring the receiver.xml File

The **receiver.xml** file is the Message Receiver's main configuration file. This section covers the configuration of the following system settings:

- Deployment Directories
- System Logging Level
- Password File Cipher Key
- Location of CPAs
- Location of passwords file **
- Logging directory
- Queue Mappings
- Party Id of the receiver
- Location of servicemap.xml file.
- Keystore Information*
- Trusted Keystore Information*
- Domain Settings
- Timeout Settings
- Database Pools

*** For more information on the uses of the keystore and trusted**

keystore see the Security section.

**** For more information on password file configuration see the Security section.
The ebXML distribution contains utilities to encrypt and to decrypt the
password file and to create cipher keys.**

Variables in the receiver.xml File

The following table describes the variables in the **receiver.xml** file:

Variable	Description
logDir	The directory in which the Message Receiver servlet writes log files. The directory must be an existing directory on the server. The default value is <installPath>\logs
logLevel	The logging level of messages within the log file. Supported logging levels include: none , error , info , detail , and messages . The amount of log information written to the file increases as the level moves from none to messages . At the messages level all possible logging messages are written including the contents of the messages. The default value is info .
maxLogSize	The maximum size of a log file. After the size limit is reached the process will stop writing to the log. The default value is 100 megs .
logArchive	Values are true or false . If true, the process will archive log files when they reach their size limit and then start a new log file. The default value is true .
incomingDir	The system looks in this directory for incoming messages when the system is configured for file-based polling. If file-based polling is not being used the parameter can be left blank. The default value is <installPath>\incoming .
myPartyId	The Message Receiver's party Id. The value of the party Id must be the same as the Message Receiver's party Id in the CPAs. For example, the party Id for the Center for Disease Control is CDC. Make sure the party Id for the State or agency is unique.
cpaLocation	The directory to the CPAs. All CPAs must be retained in this directory. The directory can be any directory accessible by the servlet. The default directory is <installPath>\config\CPA .
serviceMap	The full path name to the service map. The service map contains mappings between actions and Message Handlers. The default directory and file name is <installedPath>\config\servicemap.xml .
passwordFile	Full path to the encrypted passwords file. The clear text

Variable	Description
	password file should not be deployed in a production environment. The default directory and file name is <installedPath>\config\passwd
keyStore	Full path of the Message Receiver's Java keystore. The key store maintains the Message Receiver's private key and associated certificate in PKCS12 format. This information is used to enable encryption between the Message Sender and the Message Receiver. See the Security section for more information on managing certificates and keystores.
keyStorePasswd	A pointer to the Message Receiver's Java Keystore password within the Message Receiver's encrypted password store. The value of this entry is not the keystore password itself. It is the name of the tag within the password file. The value of this tag within the passwords file contains the actual password. See Appendix B for configuration of the passwords file.
trustStore	Full path of the Message Receiver's trusted store. The trusted store maintains trusted public keys in JKS format for the sending parties that are using encryption and digital signatures. See the Security section for more information on the managing certificates and keystores.
trustStorePasswd	A pointer to the Message Receiver's trusted store password within the Message receiver's encrypted password store. The value of this entry is not the password to the trusted store itself; it is the name of the tag within the password file. The value of this tag within the passwords file contains the actual password. See Appendix B for configuration of the passwords file.
signatureRequired	Values are true or false . When set to true, the message must have a signature. If it doesn't have a signature the message will fail. When the variable is set to false, which is the default, signatures are verified when present and the messages without signatures are accepted.
signingCertsLocation	<p>The directory in which signed certificates are stored. The directory can be any directory accessible by the servlet. The default directory is <installPath>\config\CPA.</p> <p>This directory is used when the server is communicating with a Message Sender that does not send signed certificates within digitally signed messages.</p> <p>In this case, the Message Receiver looks in the signingCertsLocation directory for the Message Sender's signed certificate. If a certificate is found, the Message Receiver uses the public key to decrypt the</p>

Variable	Description
	<p>signed hash. If the hash matches the hash computed on the Message Receiver side, the identity of the Message Sender is verified.</p> <p>You do not need to use the signingCertsLocation when communicating with the CDC's ebXML client software. The CDC's ebXML client software sends the client's certificate information in the keyInfo of the message. The Message Receiver uses this key to interrogate the computed hash.</p>
queueMap	<p>Full path to the queue map XML file. The queue map file maps a message to a queue by matching the message's argument to the queue name. The default directory and file name is <installedPath>\config\queuemap.xml</p>
payloadToDb	<p>Possible values are true or false. When true, the system places the payload in a BLOB field in a table. When false, the system writes the payload out to a file. The file will reside in the specified outgoing directory. The default value is false.</p>
key	<p>Substitution cipher key. The key and seed are used to mask the password store's password. Because the Message Receiver is a service, it is inefficient to require the system administrator to enter a password at every restart.</p> <p>This feature allows the system to access the encrypted passwords file without keeping any passwords in clear text on the production environment. See Appendix D for more information about the password cipher functionality.</p>
seed	<p>Cipher text obtained by encrypting the password to the encrypted password store using the substitution cipher key. See Appendix D for more information about the password cipher functionality.</p>
usePersistentCache	<p>When true, the file uses a persistent cache to detect duplicate messages.</p>
applelevelCaching	<p>When true, recordId field is used to detect duplicates. When false, the conversationId is used to detect duplicates.</p>
persistentCache.dbType	<p>Type of database used by the persistent cache such as sqlserver or oracle.</p>
persistentCache.jdbcDriver	<p>JDBC driver for the persistent cache.</p>

Variable	Description
persistentCache.databaseUrl	Databae URL for the persistent cache.
persistentCache.databaseUser	Database user name for the persisten cache.
persistentCache.tableName	Table name for the persistent cache.
persistentCache.cacheEntryAgeHours	Maxium age for a persistent cache entry.

Database Pooling

To enhance system performance, the Message Receiver supports database pooling. The system can support multiple database connection pools. The **receiver.xml** file stores connection pool information in its **<databasePool>** tag, which can contain one or more **<database>** tags. Each **<database>** tag represents a database pool. The following values must be set for each database tag.

Database Tag Values

Tag Value	Description
databaseId	The unique name for the database connection pool. The database Id is referenced in the queue map. The service map uses the databaseId to map the queue to a specific database.
dbType	Designates the type of database. These databases are supported: oracle, sqlserver, mysql, access.
poolSize	The number of database connections to open. When setting the pool size make sure the system can handle the maximum client load while keeping enough memory available.
jdbcDriver	The type of JDBC driver. The JDBC driver should be appropriate for the type of database such as com.microsoft.jdbc.sqlserver.SQLServerDriver for Microsoft SQL Server and oracle.jdbc.OracleDriver for Oracle.
databaseUrl	The URL to the database. The format of the URL depends on the type of database and driver used such as jdbc:microsoft:sqlserver://host:portnumber;DatabaseName=database for Microsoft SQL Server and jdbc:oracle://host:port:sid for Oracle.
databaseUser	A pointer to the database user entry in the Message Receiver's encrypted password store. The value is not the database user. It is the name of the tag within the password file. The value of the tag within the passwords file contains the actual database user name. See Appendix B for the password

Tag Value	Description
	file configuration for both the databaseUser and databasePasswd entries.
databasePasswd	A pointer to the database password entry in the Message Receiver's encrypted password store. The value is not the database password. It is the name of the tag within the password file. The value of the tag within the passwords file contains the actual database password. See Appendix B for the password file configuration for both the databaseUser and databasePasswd entries.

Configuring the servicemap.xml File

The **ServiceMap** file maps a Service and Action attribute pair to the URL of a Message Handler. The **ServiceMap** file contains the name of the service, the service type, the action name, the URL of the Message Handler and the arguments that must be sent to this URL if the Service and Action attribute pair are invoked.

```
<ServiceMap>
  <Service>
    <Name>Router</Name>
    <Type>Servlet</Type>
    <Action>
      <Name>send</Name>
      <Url>http://158.111.1.164:8080/router/router</Url>
      <Argument>action=send</Argument>
    </Action>
  </Service>
</ServiceMap>
```

The Message Sender specifies the Service and Action in the message. The Message Receiver retrieves the Service and Action from the request then uses the service map to find the URL of the Message Handler and any arguments that need to be sent to this URL.

Configuring the Collaboration Protocol Agreement

The Collaboration Protocol Agreement, CPA, specifies the conditions under which the parties will conduct transactions. They are standard ebXML 2.0 documents that describe unique party identifiers, transport protocol, security constraints and end points URLs and so on.

These CPAs are ebXML compliant files that describe the action between the Message Sender and the Message Receiver. Each party, the Message Sender and the Message Receiver, must have a copy of the CPA. They use the CPA to find endpoints and transport related information such as protocol and security settings.

The Message Sender references **the routemap.xml** file to lookup a corresponding CPA for a party and then the Message Sender retrieves the Message Receiver's end point and transports information from the CPA.

Similarly, the Message Receiver uses the party ID of the Message Sender to look up the associated CPA and then the Message Receiver uses the CPA to validate the identity of the Message Sender.

PartyInfo Segments

The CPA contains two **PartyInfo** segments, which describe the Message Sender and the Message Receiver.

- Message Sender PartyInfo
- Message Receiver PartyInfo

Message Sender PartyInfo

The **Message Sender PartyInfo** segment contains a **PartyId**, a unique number such as a DUNS number, which identifies the Message Sender. The Message Sender and the Message Receiver have agreed on which numbers they use as PartyIds.

Message Receiver PartyInfo

The **Message Receiver PartyInfo** segment is similar to the Message Sender PartyInfo. The segment contains a PartyID, a unique number, such as a DUNS number, which identifies the Message Receiver. The Message Sender and the Message Receiver have agreed on which numbers they use as PartyIDs.

Transport Sub-segment

The Message Sender and Message Receiver **PartyInfo** segments also contain a sub-segment, called **Transport**. The Transport sub-segment contains these attributes:

Transport	Description
sendingProtocol	The protocol used to send messages. Values are HTTP and HTTPS.
receivingProtocol	The protocol used to accept messages. Values are HTTP and HTTPS.
Endpoint URI	The URI or endpoint of the party.
transportSecurity	The TransportSecurity sub-tree contains the following custom attributes: authentication types Values are none , basic , custom , sdn , clientCert .

Authentication Types

For each of the four types of authentication, there is a descriptor block that specifies the configuration of that authentication:

Authentication Type	Description
sdnAuth	This block is read if authenticationType is set to sdn . The attributes of this block include the sdnConfig file (full path name of the sdn properties file and the sdnPassword . The sdnPassword value in the CPA is actually the name of a password variable within the encrypted passwords file.
clientCertAuth	This block is read if authenticationType is set to clientCert . The attributes of this block include the config file, the full path name of the properties file.
customAuth	This block is read if authenticationType is set to custom . The attributes of this block include the customLoginPage , which is the URL of the login page, relative to the end-point. It also contains publicParams and secretParams attributes. Both these parameters are name-value pairs. The public params are read directly from the CPA, whereas the values within secretParams are the names of entries within the encrypted password file.
basicAuth	This block is read if authenticationType is set to basic . The attributes in this block include indexPath (relative URL of the first page to be loaded). It also includes basicAuthUser and basicAuthPasswd , which are both references to entries within the encrypted password file.

For an example of the CPA see the appendix in this document.

Configuring Message Handlers

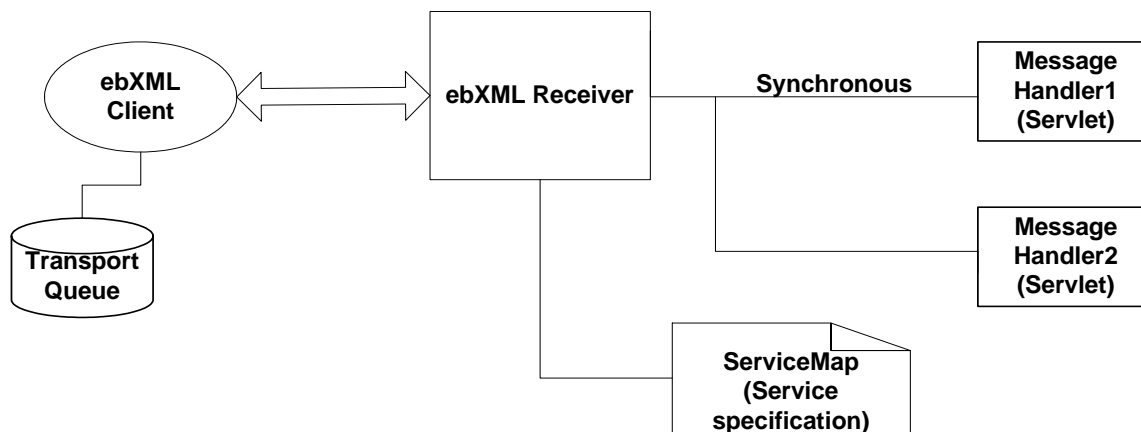
Asynchronous and Synchronous Message Handling

The Message Handler can handle messages synchronously or asynchronously. In a **synchronous** exchange, the Message Handler executes logic against the received message and returns application level status to the ebXML client (Message Sender).

In an **asynchronous** exchange, the Message Handler writes the message to permanent storage. In an asynchronous exchange, it is the responsibility of a back-end health system to perform work against the message. The application has the responsibility of sending an application level acknowledgement back to the Message Sender.

Message Handlers can be any web-based component such as a servlet, JSP, ASP, CGI and so on that can be called by the Message Receiver. The Message Handler receives the message from the Message Receiver and then performs work on the message.

The following diagram illustrates the message flow in **synchronous** mode:



1. The ebXML Receiver (Message Receiver) receives an incoming message from the ebXML Client (Message Sender).
2. The ebXML Receiver parses the envelope, decrypts the payload and then verifies the signature.
3. The ebXML Receiver looks in the **servicemap.xml** file for a service entry matching the Service/Action specified in the incoming message.

If the entry type is **Servlet**, the ebXML Receiver sends the payload to a Message Handler.

4. The ebXML Receiver waits for a response from the Message Handler and returns the Message Handler's response to the ebXML client.

Configuring the Service Map

A Message Handler's service entry in the service map must specify **servlet** as the service type. The service type for a Message Handler is the program that receives and handles the message at that URL but it does not have to be a servlet. It can also be an ASP or CGI script.

A **servlet** service has an **Action** tag. The Action tag has an URL tag which maps to the Message Handler's end point. The Message Receiver calls this URL to invoke the Message Handler. In the following example, a request with an action of **processLRNMessage** is mapped to the <http://localhost:8080/lrn/processMessage> URL.

```
<ServiceMap>
  <Service>
    <Name>BT</Name>
    <Type>Servlet</Type>
    <Action>
      <Name>processLRNMessage</Name>
      <Url>http://localhost:8080/lrn/processMessage</Url>
      <Argument>arg1=data1&arg2=data2</Argument>
    </Action>
  </Service>
</ServiceMap>
```

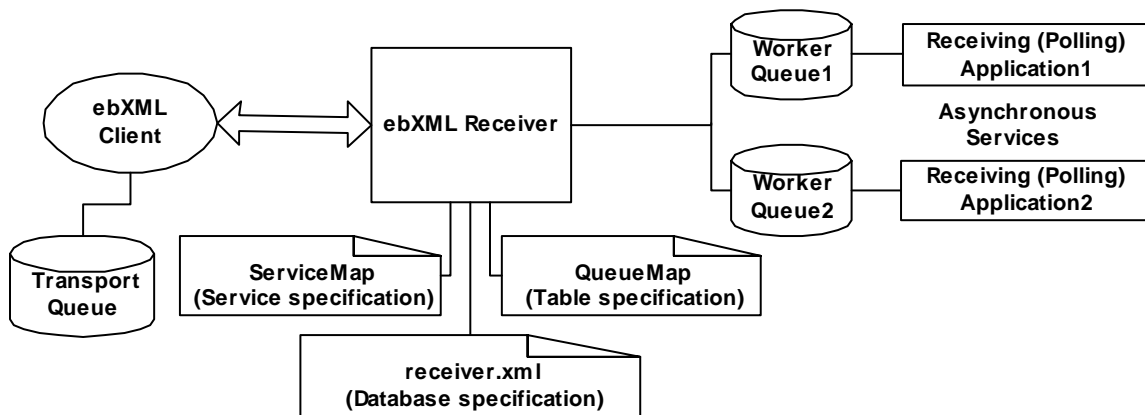
Note: Install the Message Handler and the Message Receiver on the same computer to eliminate an intruder's ability to eavesdrop on communication between them. Enforce firewall rules to prevent internal or external computers from directly accessing the Message Handler.

The **Argument** tag allows the programmer to send additional information to the Message Handler. Use the HTTP query string format for arguments such as **arg1=data1&arg2=data2**. Use URL encoding for special characters such as underscores and dashes.

Configuring Queues

The Public Health Information Network Messaging System uses queues to handle messages. The ebXML Receiver (Message Receiver) handles the requests. No external Message Handlers are called in this configuration. A system using queues is asynchronous in nature. The application level status is not sent in the initial request/response exchange between the ebXML Client and ebXML Receiver.

The following diagram illustrates the ebXML Receiver's operations in **asynchronous** mode:



1. The ebXML Receiver (Message Receiver) receives the incoming message from the ebXML Client (Message Sender).
2. The ebXML Receiver parses the message envelope, decrypts the payload, and verifies the signature.
3. The ebXML Receiver looks up the ServiceMap in the **servicemap.xml** file for an entry that matches the Service/Action in the ebXML envelope of the incoming message.
4. If the entry type is **WorkerQueue**, the ebXML Receiver writes the payload to a set of worker queues, which are defined in the servicemap for that Service/Action.
5. The ebXML Receiver sends a synchronous response to the ebXML client, which consists of the transport status only.
6. Receiving (Polling) applications poll their individual worker queues for incoming messages and then the Receiving (Polling) application sends an application level

response to the ebXML Client (Message Sender), which originally sent the message.

Configuring the Service Map

Specify **WorkerQueue** as the service type for the service entry in the service map.

```
<ServiceMap>
  <Service>
    <Name>Servicename</Name>
    <Type>WorkerQueue</Type>
    <payloadToDisk>true/false</payloadToDisk>
    <textPayload>true/false</textPayload>
    <Action>
      <Name>Receive</Name>
      <QueueId>QID123</QueueId>
      <QueueId>QID456</QueueId>
      ...
    </Action>
  </Service>
</ServiceMap>
```

The entry of type **WorkerQueue** maps to one or more queue IDs. The **QueueIDs** in the **WorkerQueue** are defined within a **QueueMap** shown below. There are two additional fields in the service element for the **WorkerQueue** service type, **payloadToDisk** and **textPayload**.

When the **payloadToDisk** flag is set to **true**, the incoming payload is written to disk instead of to the database field. In this case the name of the local file on disk is stored in the worker queue table.

When the **textPayload** flag is set to **true**, the payload is written to the **payloadTextContent** field. When the **textPayload** flag is set to **false**, the payload is written to the **payloadBinaryContent** field in the worker queue.

Configuring the Queue Map

The queue map is a definition file that resides on the Message Receiver's configuration folder and is used to map worker queues to database/table combinations.

The following example queuemap contains the definition of three queue-Ids: QID123, QID456 and QID789. Each **QueueId** maps to a database and a **tablename**. The tablename corresponds to a table that conforms to the worker queue schema. The databaseIDs are defined in the **receiver.xml** file.

```
<QueueMap>
  <workerQueue>
    <queueId>QID123</queueId>
    <databaseId>sqlserver2</databaseId>
    <tableName>workerqueue</tableName>
  </workerQueue>
  <workerQueue>
    <queueId>QID456</queueId>
    <databaseId>sqlserver1</databaseId>
    <tableName>workerqueue</tableName>
  </workerQueue>
  <workerQueue>
    <queueId>QID789</queueId>
    <databaseId>oracleserver1</databaseId>
    <tableName>workerqueue</tableName>
  </workerQueue>
</QueueMap>
```

See the Receiver Configuration section for more information about configuring database pools.

Worker Queue Table Schema

The following example illustrates the schema for a worker queue. For examples of the SQL Server and Oracle DDL scripts see Appendix C:

Field	Description	Data Type
recordId	Unique ID (integer), auto-incremented, of the record in the table. This is also the primary key of the table.	Required SQL Server: Integer, Identity=Yes, Identity Increment=1 Oracle: Integer (20), not null
messageId	Application level message identifier.	Optional SQL Server: varchar (255), null Oracle: varchar2 (255), null
payloadName	File name of payload as specified by the Message Sender.	Optional SQL Server: varchar (255,) null Oracle: varchar2 (255), null
payloadBinaryContent	Image/BLOB field, which is written by the Message Receiver servlet.	Optional SQL Server: Image data type, null Oracle: BLOB data type, null
payloadTextContent	Text/CLOB field that is used when the textPayload value in the servicemap entry is true .	Optional SQL Server: Text data type, null Oracle: CLOB data type, null
localFileName	This field is used when the file is on disk instead of written to a database field. When the payloadToDisk value in the servicemap entry is true .	Optional SQL Server: varchar (255,) null Oracle: varchar2 (255), null
service	The ebXML service.	Required SQL Server: varchar (255,) null Oracle: varchar2 (255), null
action	The ebXML action.	Required SQL Server: varchar (255,) null Oracle: varchar2 (255), null
arguments	Arguments specified by the Message Sender.	Optional SQL Server: varchar (255,) null Oracle: varchar2 (255), null
fromPartyId	Party ID of the sending party.	Required SQL Server: varchar (255,) null Oracle: varchar2 (255), null

Field	Description	Data Type
messageRecipient	Identifies the Message Recipient, which is specified by the Message Sender in the transportQ_Out queue.	Optional SQL Server: varchar (255), null Oracle: varchar2 (255), null
errorCode	Error code	Optional SQL Server: varchar (255), null Oracle: varchar2 (255), null
errorMessage	Error message	Optional SQL Server: varchar (255), null Oracle: varchar2 (255), null
processingStatus	Status of this record. The initial value of this field when a record is created is queued	Optional SQL Server: varchar (255), null Oracle: varchar2 (255), null
applicationStatus	The applications processing status.	Optional SQL Server: varchar (255), null Oracle: varchar2 (255), null
encryption	The value is yes if the payload is encrypted and no if it isn't.	Optional SQL Server: varchar (10), null Oracle: varchar2 (10), null
receivedTime	Time when payload was received in UTC format such as 2001-10-01T16:01:01 .	Optional SQL Server: varchar (255), null Oracle: varchar2 (255), null
lastUpdateTime	Time when record was last updated in UTC format such as 2001-10-01T16:01:01	Optional SQL Server: varchar (255), null Oracle: varchar2 (255), null
processId	Identifies the record's latest process.	Optional SQL Server: varchar (255), null Oracle: varchar2 (255), null

To Configure the Queues

To configure the queues for the PHINMS do the following:

- 1 Create the worker queues.
See the DDL and SQL Server scripts in Appendix C.
- 2 Add the **DatabasePool** entries to the **servicemap**.
See the example in the blank section.
- 3 Add the **QueueMap**.
See the example in the blank section.
- 4 Add the ServiceMap entries for the queues.
See the example in the blank section.
- 5 Add the database user and password entries to the encrypted passwords file and reference them from the **DatabasePool** entries.
- 6 Add the database drivers, JDBC, to the ebXML Receiver **WEB-INF\lib** folder for SQL Server or Oracle.

Installing the Route-Not-Read Message Handler

Route-not-Read Servlet

The route-not-read servlet contains the following tables and configuration files:

Tables

- Messagebins – for reading and writing messages.
- Broadcast – defines broadcast lists.
- Users – defines authorized route-not-read user IDs.
- PartyID_User – maps valid partyIDs and route-not-read UserIDs.

Configuration Files

- **RouterConfig.xml** – contains the router servlet's configuration parameters
- **Web.xml** – contains the deployment descriptor for the router servlet, which contains the **RouterConfig.xml** file.

Messagebins Table

The router servlet writes and reads to the **messagebins** table described below:

Field	Description
recordId	Integer.
fromPartyId	Party ID of the sending party.
messageRecipient	Message Recipient's identifier.
messageRecordId	RecordID of the message.
messageApplicationId	Identifier that is specified by the application that uses the PHINMS system.
arguments	Arguments sent by the Message Sender, which are conveyed to the Message Handler and to the route-not-read Message Recipient.
payloadFile	File name of the payload, as specified by the sender
localFile	File name of the payload, as stored on disk by the route-not-read handler
processingStatus	Flag which indicates the processing status: written – Message was written by the router handler and is ready to be read. read – Message has been read by a poll request.
receivedTime	Time when message was received.
pickedupTime	Time when message was picked up.

Broadcast Table

The broadcast table is used to set up broadcast lists. Their function is similar to an e-mail distribution list, a message is sent to two or more addresses.

For example:

If a broadcast list is `name=sepcialinterestgroup` and `addresses="john,jill"`, then when a message is sent with `messageRecipient=specialinterestgroup`, both john and jill will receive a copy of the message.

Field	Description
name	Broadcast list name
addresses	Comma separated list of user names

Users Table

Each user in the route-not-read system is identified by a unique user name. To use the route-not-read system, the user's identity must be present in the users table.

Field	Description
name	User name
description	Optional description

Partyid_user Table

The `partyid_user` table maps a route-not-read user to an ebXML partyID, which can be the same. However, several route-not-read users can have the same ebXML partyID. The following table is used to validate send and poll requests, which means, it determines whether the `messageRecipient` of the poll request corresponds to the correct ebXML party ID. To use the route-not-read system, each route-not-read user must be mapped to their `partyID` as shown below.

Field	Description
PartyId	EbXML Party ID
user	User name – the name as it appears in the users table.
sdnuser	User key of the SDN user.

routerconfig.xml

The **routerconfig.xml** file is an XML format configuration file, which specifies configuration parameters of the router. This file contains the following parameters.

Field	Description
dbType	Database type: sqlserver or oracle
jdbcDriver	JDBC Driver name
databaseUser	Database user key –an index into the encrypted passwords file.
databasePasswd	Database password key – an index into the encrypted passwords file.
passwordFile	Password file name.
logLevel	Logging level: none , error , info , detail , messages .
logDir	Logging directory.
key	Key used for substitution cipher. Together, the key and the seed are used to determine the password to the passwords file.
seed	Seed used for substitution cipher
SdnAuth	Indicates whether SDN authorization is used. Values are True , False .

See Appendix J for an example of the **routerconfig.xml** file.

Security

Security is a very important aspect of the Public Health Information Network Messaging System. Managing security is vital to ensure the messages are confidential and their integrity is preserved.

Java Keystores

A **keystore** is a repository of keys and certificates. The **KeyStore** class is an engine class that supplies well-defined interfaces to access and modify the information in a keystore. You can have several implementations at the same time, with a different implementation for each type of keystore.

Sun Microsystems provides a default implementation. It implements the keystore as a file, which utilizes a proprietary keystore format called **JKS**. It protects each private key with an individual password and protects the integrity of the entire keystore with another password, which may be different than the individual password.

The **JKS** keystore type is used to store all trusted certificated within the **TrustedStore**. Use the **keytool** utility, described later in this section, to manage information within the **JKS** keystore type.

RSA Laboratories, along with other vendors such as Apple, DEC, Microsoft, and Sun create the Public Key Cryptography Standards, **PKCS**, specifications. The functions of the **PKCS** specifications vary and they address issues related to security and cryptography. The **PKCS12 Keystore** type is used to store your site-specific keys and certificates. You can use either Internet Explorer or Netscape browsers to manage the information in this keystore type.

To Manage the PKCS12 KeyStore using Internet Explorer

1. If you haven't already, load your private key and certificate into the Internet Explorer or Netscape browser.
2. Select **Tools->Internet Options->Content->Certificates**. Select the certificate you want to export and then click **Export**.
3. A wizard appears. Click **Next**.
4. Select the **Yes, export the private key** check box and then click **Next**.
5. Select the **Personal Information Exchange – PKCS12** option, and then select the check boxes which read:
 - Include all certificates in the certification path if possible
 - Enable strong protection
6. **Do Not** select the **Delete private key after export** check box. It will delete your private key from the store after you export it.
7. Click **Next**.
8. Specify the password for encrypting the file and then click **Next**.
This password is the new **keystore** password. This password will be encrypted in the **passwd.xml** file using the **pbe** utility described in the **Security** section.
9. Specify the file name such as **<installedPath>\keystores\ServerCert** and then click **Next**.
You will receive a message indicating the export was successful.

To Manage the PKCS12 Keystore using Netscape Communicator

1. From the browser's main window click **Security** icon.
2. Click the **Certificates->Yours** link.
A window with a list of certificate serial numbers appears. If there is more than one, view each certificate and then select the correct one.
3. Click **Export**.
A window appears.
4. Type the Netscape Communicator password for keystores. This is the same password you used when you initially installed this key/certificate on the Netscape browser.
A window appears.
5. Enter the **keystore** password to protect the data you are exporting and when the next window appears, enter the password again to confirm it.
6. A browser window appears. Select the file name of the exported file and then click **Save**.

Managing the Java Trust Store with Keytool

Keytool is a key and certificate management utility. You can use it to administer your public/private key pairs and associated certificates, which are used in self-authentication. Self-authentication is a process in which a person uses digital certificates to authenticate his or her identity to other users, services, or data integrity and authentication services. You can also use keytool to cache the public keys, in the form of certificates, of your peers.

Viewing the Trusted CA Certificates in the Java Trust Store

By default, this command prints a certificate's MD5 fingerprint. When the **-v** option is specified, the certificate is printed in a format that can be read by humans and it contains information such as the owner, issuer, and serial number.

Command Syntax:

```
<javabin>keytool -list -v -keystore <truststorefile> -storepass  
<storepass>
```

Where:

- **javabin:** Path indicating the location where the Java binaries are stored on your machine such as **d:\jdk1.4\bin**)
- **truststorefile:** File containing **truststore** such as **cacerts**.
- **Storepass:** Password to trust store

Example:

```
<installedPath>\java\j2re1.4.0_03\bin\keytool -list -keystore \  
    <installedPath>\keystores\tomcat -storepass changeit
```

Changing the Password of the Java Trust Store

Change the password to the Java Trust Store to protect the integrity of the keystore contents. The new password is **newpass**, which must be at least 6 characters long.

Command Syntax:

```
<javabin>keytool -storepasswd -new <newpass> -keystore <truststorefile>  
-storepass <oldpass>
```

Where:

- **javabin:** Path indicating the location where the Java binaries are stored on your machine such as **d:\jdk1.4\bin**)
- **truststorefile:** File containing **truststore** such as **cacerts**.
- **oldpass:** Old password to trust store.
- **newpass:** New password to trust store.

Example:

```
<installedPath>\java\j2re1.4.0_03\bin\keytool -storepasswd -new  
changeit2 \  
-keystore <installedPath>\keystores\tomcat -storepass changeit
```

Importing a Certificate Authority Certificate to the Java Trust Store

Use the following command to read the certificate or certificate chain from the file **cert_file** and store it in the keystore entry identified by **alias**. The certificate chain is supplied in a reply in **PKCS#7** format.

Command Syntax:

```
<javabin>keytool -import <alias> -trustcacerts -file <cert_file> -  
storepass <storepass> -keystore <truststore>
```

Where:

- **javabin:** Path indicating location where Java binaries are stored on your machine such as **d:\jdk1.4\bin**
- **truststorefile:** File containing **truststore** such as **cacerts**.
- **storepass:** Password to trust store.

Example:

```
<installedPath>\java\j2re1.4.0_03\bin\keytool -import cacert1 -  
trustcacerts -file \  
    <installedPath>\keystores\changeit3.cer -storepass changeit \  
    -keystore <installedPath>\keystores\cacert1
```

Exporting a Certificate from a Java Key Store

The following command reads the certificate associated with **alias** from the keystore and stores it in the **cert_file** file.

Command Syntax:

```
<javabin>keytool -export <alias> -file <cert_file> -  
storepass <storepass> -keystore <keystore>
```

Example:

```
<installedPath>\java\j2re1.4.0_03\bin\keytool -export \  
-file <installedPath>\keystores\changeit3.cer -storepass  
changeit \  
-keystore <installedPath>\keystores\cacert1
```

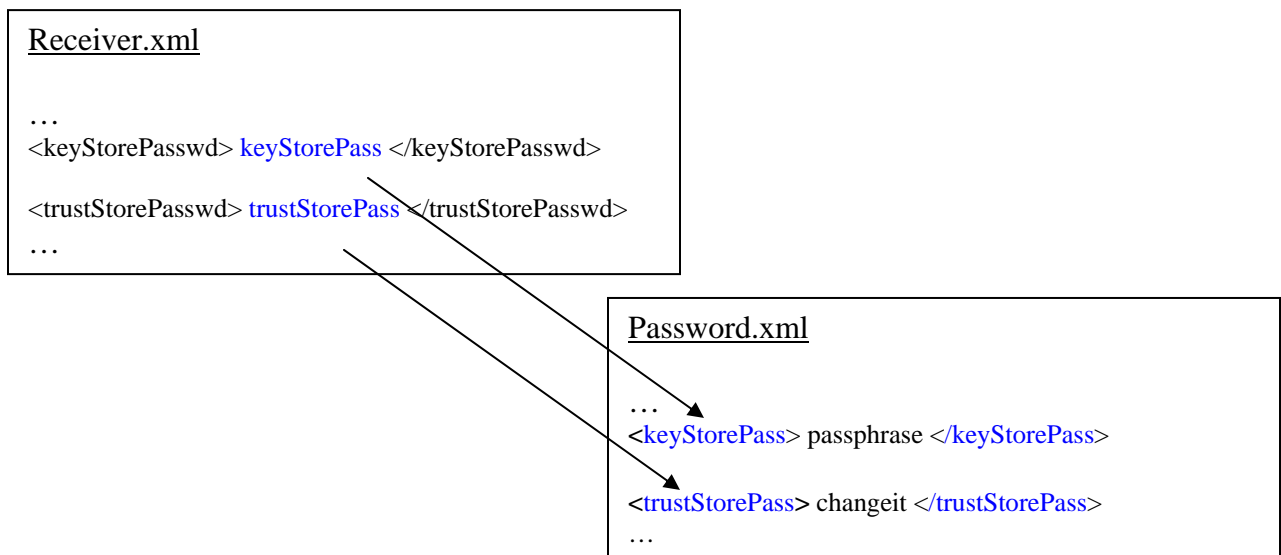
Managing Passwords

Receiver Password File

All passwords are stored in an encrypted password file. You define the name of the password file. It is referenced in the **receiver.xml** file with the **<passwordFile>** tag in XML format.

Passwords referenced in the **receiver.xml** file are pointers to the XML tag in the password file. The password itself is contained within the associated XML tags.

Example:



Because the Receiver is a servlet-based application and has no human intervention during initialization, the password for the password file must be protected. A cipher key generator is used to protect the password file's password. Use the **<key>** and **<seed>** tags are used to define the cipher text.

During the creation of the cipher text, enter a key value that will be used to generate the cipher text. This key value can be any number. The larger the number the larger the seed value generated. The **<seed>** tag references the actual cipher text for the password.

The Public Health Information Network Messaging System offers two utilities to manage passwords; **pbe.bat**, which encrypts and decrypts the password file, and **substitutue.bat**, which creates the cipher text used to protect the password file. These commands are defined in more detail in the Password Utilities section.

Password Utilities

Password Based Encryption (pbe)

The Password-Based Encryption command line utility, **pbe.bat**, encrypts and decrypts resource files, such as the password files. These files hold references to all password information throughout the system including passwords for Trusted KeyStore, KeyStore, database, user name and password.

To Encrypt a File

1. Start the **pbe** utility by executing the following command:
`<installedPath>\cmds\pbe.bat`
2. Type the letter **e** to encrypt the file and then press **Enter**.
3. Type the name of the file you want to encrypt and then press **Enter**. This file must be at least 6 characters long and contain at least one numeric digit.
4. Type the location you want to put the file and then press **Enter**.
5. Type the password and then press **Enter**.

To Decrypt a File

1. Start the **pbe** utility by executing the following command:
`<installedPath>\cmds\pbe.bat`
2. Type the letter **d** and then press **Enter**.
3. Type the name of the file you want to decrypt and then press **Enter**. This file must be at least 6 characters long and contain at least one number.

4. Type the location you want to put the decrypted file and then press **Enter**.
5. Type the password and then press **Enter**.

Substitution Cipher Key Generator

Use the cipher key generator utility to generate cipher text for a password. The cipher text-based password is used to protect the Message Receiver's password file. The Cipher Key utility requires a unique numeric key value that is used to generate the unique cipher seed value. The key and seed values are placed in the **receivers.xml** configuration file.

To generate a cipher seed value do the following:

1. Start the cipher utility by executing the following command:
<installedPath>\cmds\substitute.bat.
2. Type the substitution cipher key and then press **Enter**. The key value must be a numeric number at least 3 digit.
3. Type the password text you want to encrypt and then press **Enter**.

The encrypted value is displayed on the command prompt. Cut and paste this value into the **receiver.xml** **<seed>** tag. The associated key value you enter during the generation of the key will also need to be included in the **receiver.xml** file using the **<key>** tag.

Authentication

The Collaboration Protocol Agreement specifies the authentication mode each route uses. There are four mode types and all use the SSL transportation mechanism.

Mode Type	Description
Basic	Using basic authentication, the client sends the user name and password, encoded in Base64 format, as part of the HTTP header. The web server uses a user password database or it maps user names and passwords to the underlying operating system. The cookies that are returned from the login process are sent with subsequent requests throughout the session.
Custom	Using customized authentication, the client, the Message Sender, logs in using a customized method, which may include login and password fields. After logging in, the Message Receiver usually installs cookies on the client. The client sends these cookies to the Message Receiver in subsequent requests in the same session.
SDN	The Centers for Disease Control and Prevention use the Secure Data Networking security and public-key infrastructure. To use SDN, obtain a User-ID and a client certificate, then install these on the client. To access a resource on the SDN system, the client identifies itself using its certificates and then performs authentication using the

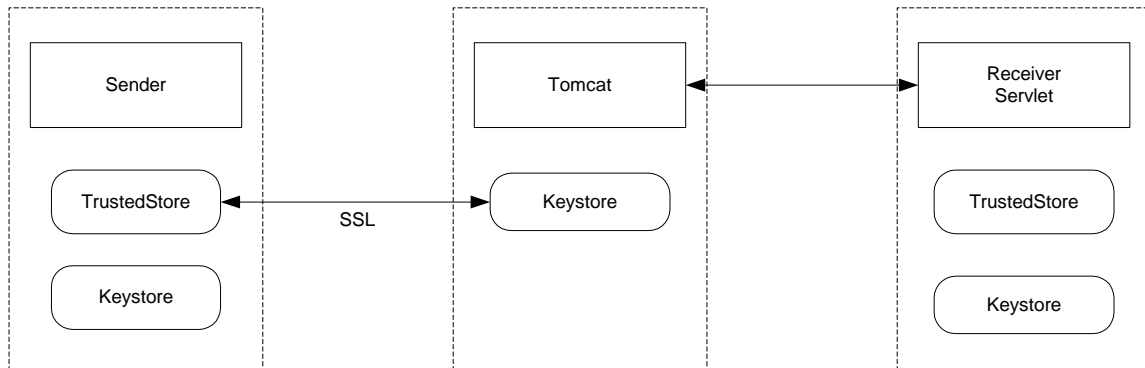
Mode Type	Description
	client certificates and passphrases. During authentication, the Message Receiver installs cookies on the client. The client sends these cookies to the Message Receiver in subsequent requests in the same session. The user credentials control accesses to web resources on the Message Receiver server.
Client Certificate-Based Authentication	During the SSL exchange, the web server uses the client's certificate to verify that the client possesses the private key for the public key certificate. If available, the web server contains a list of approved client certificates. Only those clients on the list will be able to access the web resources. Digital signatures prevent the message from being rejected because of its origin.

Encryption

Enabling SSL Authentication

SSL provides for encryption of a session, authentication of a server, and, optionally, a client and message authentication.

SSL Authentication Flow



To Enable SSL

To enable SSL authentication do the following:

1. Execute the following command to export the Tomcat certificate, which was generated during the **Configuring SSL for Tomcat** section.

```
<installedPath>\java\j2re1.4.0_03\bin\keytool -export -file  
tomcat.cer \  
-keystore <installedPath>\keystores\tomcat -alias  
tomcat
```

2. Execute the following command to import the Tomcat certificate into your **sender's** Trusted Store.

```
<installedPath>\java\j2re1.4.0_03\bin\ keytool -import -  
file tomcat.cer  
-keystore cacerts -alias tomcat
```

3. Restart the sender application.

Encryption

Asymmetric Encryption

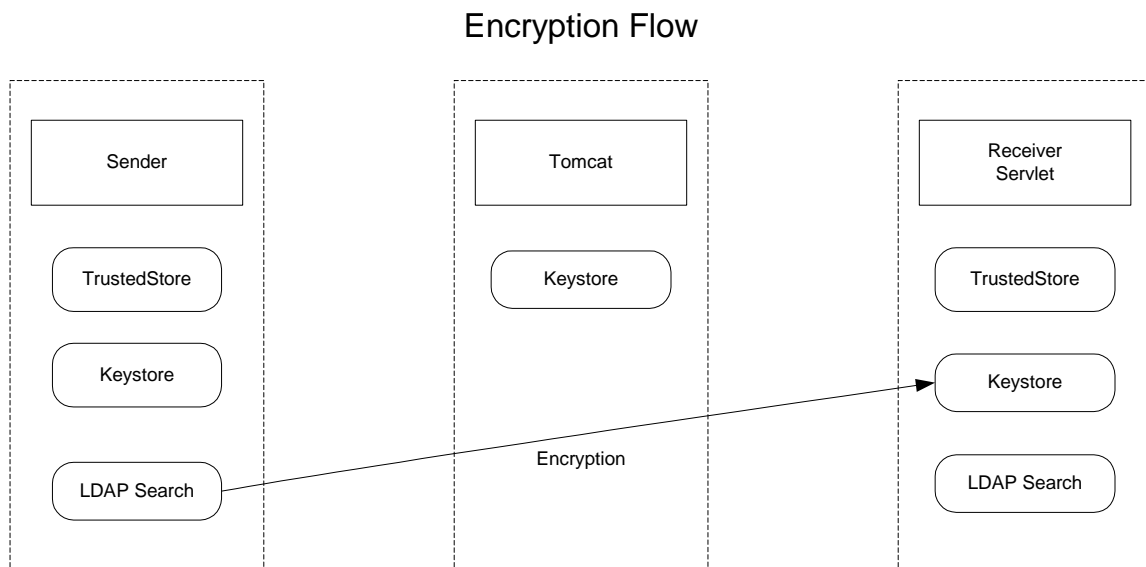
Based on the configuration of the PHINMS Message Sender configuration, the Message Sender obtains the information that needs encrypting, along with the LDAP attributes of the message recipient, which include the LDAP server name, the base address, and the recipient's common name.

When the Message Sender receives the message from the database or file system it performs a cache lookup for the recipient's certificate. If a recent certificate for the Message Receiver exists in the cache, it retrieves it from the cache and uses it. If the key is not found in the cache, it queries the Verisign LDAP directory and obtains the Message Receiver's public key certificate.

After the Sender obtains the public key certificate, it retrieves the recipient's public key from the certificate. The Message Sender uses the XML Encryption library to perform asymmetric encryption on the plaintext supplied.

Asymmetric Decryption

During asymmetric decryption, the Message Receiver receives the XML encrypted message. The Message Receiver retrieves the private key of its asymmetric key pair from the local Java keystore, and uses this key to decrypt the ciphertext using XML encryption library.



Enabling Encryption

To enable encryption do the following:

1. Obtain a valid certificate either through the CDC's Secure Data Network (SND) request or directly with Verisign at <http://www.verisign.com>.
2. Follow the steps in **Managing PKCS12 Keystore** to export your new key into a PKCS12 keystore.
3. Copy the new keystore created in step 2 to **<installedPath>\keystores**.
4. Modify the **<keyStore>** tag in the **receiver.xml** file to the appropriate path and file name created in step 2.
5. Modify the **<keyStorePasswd>** tag in the **receiver.xml** file to the appropriate tag in the **password.xml** file that points to your keystore password. See the Password Management section of this document for more information.
6. Restart the Message Receiver servlet. There are several steps that must be completed on the Message Sender side before encryption can be completely enabled.

Digital Signatures

Generating Digital Signatures

Digital signatures are generated using the Message Sender's private key. The Message Sender retrieves the user's private key of its asymmetric key pair from the local Java key store and uses this key to sign the text. The signature string is returned to the user.

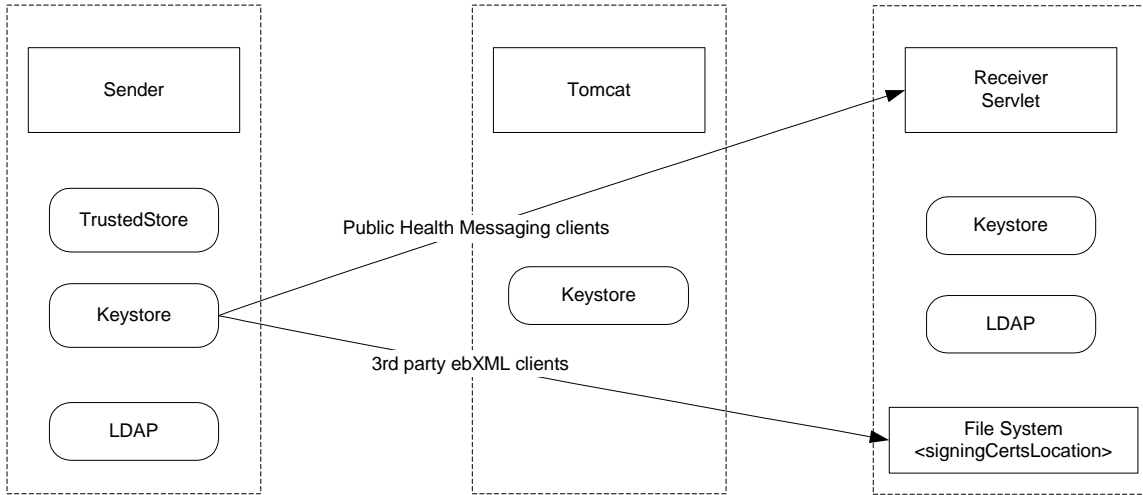
Verifying Digital Signatures

The digital signature, the signer's identity (distinguished name) and the text that is signed are passed to the Message Receiver for verification. There are two methods for obtaining the public key used to verify the signed message:

- If the sender is using the PHINMS software, the Message Receiver obtains the public key using the **keyInfo** attribute in the ebXML envelope.
- If the sender is a third-party ebXML client and the keyInfo attribute is empty, the Message Receiver will expect to retrieve the public key from the **<signingCertsLocation>** tag defined in the **receiver.xml** configuration file. In this case, the CPA for this sender will need to have a **NonRepudiation Certificate** entry that points to the signing certificate.

After the Message Receiver retrieves the public key it uses it to decrypt the signed hash. After the signed hash is decrypted, the Message Receiver compares the decrypted hash to the hash, which was computed on the digitally signed message. The identity of the Message Sender is verified only when the hashes are identical.

Digital Signature Flow



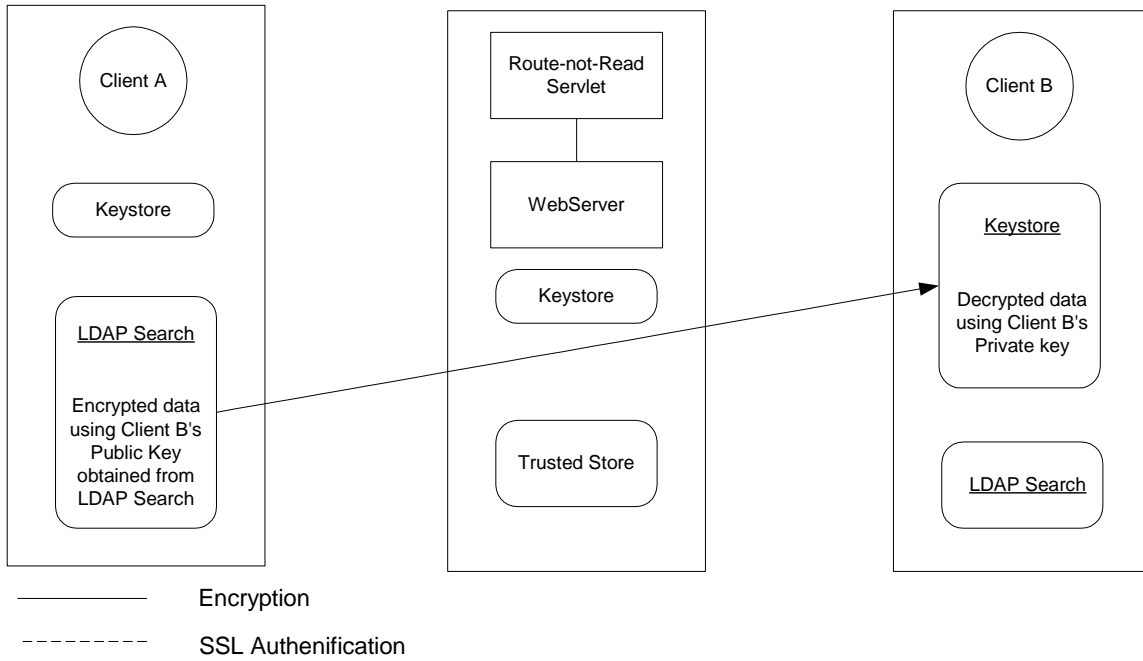
To Enable Digital Signatures

To enable digital signatures do the following:

1. Edit the **receiver.xml** file in **<installedPath>\config** and change the **<signatureRequired>** tags value to **true**.
2. If your Message Receiver will be supporting any third- party clients, modify the **<signingCertsLocation>** tag in the **receiver.xml** file to point to a location on disk that will store all third- party public key certificates.
3. Restart the Message Receiver servlet.

Enabling Route-not-Read Encryption

Encrypted Message from A to B



Security Best Practices

Use the following guidelines to preserve your message's integrity and secrecy:

- Run the Message Receiver on an application server that accepts HTTPS requests. Connect the web proxy to the server using HTTPS to reduce wiretapping attacks on the DMZ.
- Permit access to the Message Receiver from the web proxy only. Block all other direct accesses to the Message Receiver.
- Permit access to the Message Handler from the Message Receiver only. Block all other direct accesses to the Message Handler.
- Protect the service maps on the Message Receiver by allowing only authorized users to modify them..
- Do not store passwords as plain text files on the Message Sender or Message Receiver. Promptly delete the plain text files that you use to generate encrypted password files.
- Use digital signatures for non-repudiation of message origin. You need a client certificate and a public key infrastructure, such as LDAP, to manage the client certificates.
- Use a combination of at least eight numbers and letters for passwords to the key stores and trusted CA certificates. Each user should have a unique password.
- Use only Message Handler services that are within the Message Receiver's Intranet. The service map on the Message Receiver should not contain any URLs for message handler services that point outside the Intranet.
- Using file system permissions, allow administrators only to modify the service map.
- If wire-tapping is a risk within the Message Receiver's Intranet, put the Message Handler and the Message Receiver on the same host because communication between the Message Receiver and the Message Handler is not encrypted.

- Using file system permissions, allow only authorized users to modify the configuration files on the client and on the server.
- Using file system permissions, allow only authorized users to read and write incoming and outgoing payload (file) directories.

System Maintenance

Setting Log Levels

The system's log level is set using the **logLevel** entry in the **receiver.xml** file. If the setting is set to **info**, **detail**, or **messages**, make sure the log information does not exceed acceptable disk limits. The **greater than** and **info** settings should be used only during testing and problem-solving sessions. The software should not run in **detail** or **messages** settings for general use because they can write a large amount of information to disk, which reduces usable disk space and consumes processing power. For example, when running Tomcat on Windows default logging is displayed on the console. It can take several seconds to spool a several megabyte Base64 payload on file in format

Log Level	Description
none	No log information will be written to the file.
error	System error messages will be written to the file.
info	Basic message information will be written to the file.
detail	Detailed message information and specific logical tasks executed by the program will be written to the file.
messages	All possible logging messages are written including the contents of the messages.

Log File Maintenance

As the system administrator, you must manage log files to permit easy accessibility to vital log information while maintaining system performance. Maintenance of system log files is controlled using the following parameters in the **receiver.xml** file.

Parameters	Description
maxLogSize	The maximum size of a log file. Once the size limit is reached the process will stop writing to the log. The default value is 100 megs .
logArchive	Possible values are true or false . If true, the process will archive log files once they reach their size limit. The process will then start a new log file. The default value is true .
logDir	The directory in which the receiver servlet will write log files. The directory must be an existing directory on the server. The default value is <code><installPath>\logs</code>

After the size limit is reached, the system will stop writing to the log file. When **logArchive** is set to **true**, the system will write a new log file in the **logDir** directory.

The maximum size of the log file can be set to a size that will allow log-file-creation to correlate with a specific timeframe. For example, the maximum size can be set so that logs are written daily or once a week. This configuration allows backup process to write logs to permanent storage without concern for the log file contention.

Integration Issues

In situations where the standard worker queues or file-based Message Handler do not meet an agency's needs, the agency may need to build a custom Message Handler. For example, an agency may need to respond immediately to a party's request. In this type of synchronous exchange, the system must receive and process the message then deliver a response directly back to the calling party. This message exchange scenario is best handled using a custom Message Handler. A custom Message Handler can accept and immediately act upon the request. The Message Handler can be a Java servlet, Java Server Page (JSP), Active Server Page (ASP) or any other type of web-based executable.

Consider the following factors when designing a custom Message Handler.

- Length of the transactions.
- Number of transactions at normal and peak times.
- Availability characteristics.

Make sure the system can respond in a timely manner, especially during peak times. If the system is unable to respond in a timely manner, an asynchronous process utilizing worker queues may be a more appropriate solution.

Message Receiver Interface

The interface between the Message Receiver and the custom Message Handler is a standard HTTP request/response exchange. The Message Receiver un-packages the ebXML request and sends the payload and associated parameters to the Message Handler as a multi-part request. The custom Message Handler retrieves parameters from the text/**plain** part of the multi-part request. By default, the Message Receiver sends the following parameters.

Parameters	Description
from	The party ID of the ebXML client, the Message Sender.
manifest	The ebXML manifest sent by the ebXML client, the Message Sender.
"Arguments"	Arguments specified in the Message Handler's service map. Can be one or more arguments as a name/value query string. Example: arg1name=arg1value&arg2name=arg2value.

The Message Handler retrieves the Base64 encoded payload from the **Application/Octet-Stream** part of the multi-part request.

The Message Receiver passes cookies from its session with the ebXML client, the Message Sender, in the request to the Message Handler. In some environments, the Message Sender's cookies may contain user-specific information needed by the back end system such as a user name.

Example Message Handler Servlet

This is a code listing of an example Message Handler servlet which copies the payload file to disk:

```
/**
 * Title:          <p> MessageHandler
 * Description:    <p> Handles message passed in by ReceiveFileServlet
 * Copyright:     Copyright (c) CDC
 * Company:      CDC
 * @author       Raja Kailar, PhD.
 * @version     1.0
 */
package gov.cdc.nedss.services.messagehandler;

import java.util.Properties;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;
// Import the nedss.common package for?
import gov.cdc.nedss.common.*;
// Incorporate logging mechanisms used in the message receiver
// by importing the nedss logging package.
import gov.cdc.nedss.services.logging.*;
// Import nedss xml utilities to read receiver.xml file.
import gov.cdc.nedss.utilities.xml.*;
import gov.cdc.nedss.services.transport.message.*;
import gov.cdc.nedss.services.security.encryption.*;

public class MessageHandler extends HttpServlet {

    // static property to store incoming directory setting
    // system will write payload out to incoming directory
    private static String incomingDir = null;

    public void setIncomingDir(String s) { incomingDir = s; }
    public String getIncomingDir() { return incomingDir; }

    /**
     * Message Handler initialization routine
     * @param Servlet Configuration
     */
}
```

```

    public void init(ServletConfig servletConfig) throws
ServletException {

        super.init(servletConfig);
        System.out.println("MessageHandler started");
        System.out.println(Defines.VERSION);

        String configFile =
servletConfig.getInitParameter("receiverConfig");
        XMLReader xmlrdr = new XMLReader();
        xmlrdr.readProperties(configFile);
        Properties props = xmlrdr.getProperties();

        // Set the log level, default to debug, development?
        // using CDC nedss log features
        Log.setDebug(true);
        Log.setLogDir(props.getProperty(Defines.RECEIVERLOG));
        String logLevel = props.getProperty(Defines.RECEIVERLOGLEVEL);

        if (logLevel.equalsIgnoreCase("none")) {
            Log.setLogLevel(Log.NONE);
        } else if (logLevel.equalsIgnoreCase("error")) {
            Log.setLogLevel(Log.ERROR);
        } else if (logLevel.equalsIgnoreCase("info")) {
            Log.setLogLevel(Log.INFO);
        } else if (logLevel.equalsIgnoreCase("detail")) {
            Log.setLogLevel(Log.DETAILED);
        }

        Log.logIt("Started MessageHandler servlet", Log.INFO);
        setIncomingDir(props.getProperty(Defines.RECEIVERINCOMINGDIR));
        Log.logIt("messageHandler, incomingDir=" + getIncomingDir(),
Log.INFO);
    }

    /**
     * Servlet post
     * The Servlet accepts the payload and parameters as an HTTP post.
     * @param Http Servlet Request object
     * @param Http Servlet Response object
     */
    protected void doPost(HttpServletRequest request,
HttpServletRequest response)
        throws ServletException, IOException {

        StringBuffer resp = null;
        try {
            // call processRequest to write file to disk
            if ((resp = processRequest(request)) == null) {
                System.out.println("In message handler, error processing
request");
                return;
            }
        }
    }

```



```

        PrintWriter out = response.getWriter();
        out.println(resp.toString());
        out.close();
    } catch (Exception e) {
        System.out.println("Error in doPost");
        e.printStackTrace();
    }
}

/**
 * Parses the MIME multipart request message and returns a response
 * @param Http Servlet Request object
 * @return Response string buffer
 */
private StringBuffer processRequest(HttpServletRequest request) {

    BufferedReader in = null;
    String token = null;
    String fromPartyId = null;
    String manifest = null;

    // use the ? multi part parser
    HttpMultiPartParser hmp = new HttpMultiPartParser();

    try {
        // Splits mime message fields into text and payload
        if (!hmp.processHttpRequest(request)) {
            System.out.println("Error parsing multipart fields in
request");
            return null;
        }
    } catch (Exception e) {
        Log.logIt(e, "Error parsing message in messagehandler",
Log.ERROR);
    }

    // parse the text parameters
    // the message receiver can send multiple text parameters
    StringTokenizer st = new StringTokenizer(hmp.getTextPart(), "&");

    while (st.hasMoreTokens()) {
        token = st.nextToken();
        if (token.startsWith("from")) {
            fromPartyId =
token.substring(token.indexOf("=")+1).trim();
        } else if (token.startsWith("manifest")) {
            manifest = token.substring(token.indexOf("=")+1).trim();
        }
    }

    // Do necessary processing of data here
    // e.g., Copy file to disk as below

```

```

        try {
            if (hmp.getPayloadPart() != null) {
                FileOutputStream fos = new
FileOutputStream(getIncomingDir() +
hmp.getFilename());

                if (Base64Converter.isBase64(hmp.getPayloadPart())) {

fos.write(Base64Converter.base64StringToByteArray(hmp.getPayloadPart()
);
                } else {
                    fos.write(hmp.getPayloadPart().getBytes());
                }
                fos.flush();
                fos.close();
            }
        } catch (Exception e) {
            e.printStackTrace();
        }

        // create response
        // Note - these routines will be implemented by the message
handler programmer
        // depending on the specific data processing requirements and
logic.
        String status = getResponseStatus(); // application
status
        String error = getResponseError(); // application
error
        String appdata = getResponseAppData(); // get
application data
        String payload = getResponsePayload(); // get response
payload
        String filename = getResponseFilename(); // get response
filename

        // compose mime multi-part response
        StringBuffer response = MimeComposer.composeMessage(status,
error,
appdata,
payload,
filename);

        return response;
    }

    // The following routines need to be implemented by the message
handler
    // programmer
    private String getResponseStatus() {
        return ("success");
    }
}

```

```

private String getResponseError() {
    return ("noError");
}

private String getResponseAppData() {
    return ("applicationData");
}

private String getResponsePayload() {
    return null;    // UCC
}

private String getResponseFilename() {
    return null;    // UCC
}

/**
 * Dummy get method
 */
public void doGet(HttpServletRequest req, HttpServletResponse resp)
    throws ServletException, IOException {

    PrintWriter out = resp.getWriter();
    resp.setContentType("text/html");
    out.println("<html>");
    out.println("ebXML Default MessageHandler 2.0<br>");
    out.println("Centers for Disease Control and
Prevention<br>");
    out.println(Defines.VERSION+"<br>");
    out.println("</html>");
}
}

```

Appendix

Appendix A.

Worker Queue Generation Scripts

Worker Queue Generation Script for Oracle

```
DROP TABLE <Tablename>;
CREATE TABLE <Tablename> (
    recordId NUMBER (20) NOT NULL,
    messageId VARCHAR2 (255) NULL,
    payloadName VARCHAR2 (255) NULL,
    payloadBinaryContent BLOB,
    payloadTextContent CLOB,
    localFilename VARCHAR2 (255) NULL,
    service VARCHAR2 (255) NOT NULL,
    action VARCHAR2 (255) NOT NULL,
    arguments VARCHAR2 (255) NULL,
    fromPartyId VARCHAR2 (255) NULL,
    messageRecipient VARCHAR2 (255) NULL,
    errorCode VARCHAR2 (255) NULL,
    errorMessage VARCHAR2 (255) NULL,
    processingStatus VARCHAR2 (255) NULL,
    applicationStatus VARCHAR2 (255) NULL,
    encryption VARCHAR2 (10) NOT NULL,
    receivedTime VARCHAR2 (255) NULL,
    lastUpdateTime VARCHAR2 (255) NULL,
    processId VARCHAR2 (255) NULL,
);
ALTER TABLE <Tablename>
ADD PRIMARY KEY (recordId);
CREATE SEQUENCE <Tablename>_record_count
INCREMENT BY 1
START WITH 10
MINVALUE 1
MAXVALUE 9999999999999999999999999999999
NOCYCLE
NOORDER
CACHE 20;
```

Note: The sequence name must be **<Tablename>_record_count** for the Message Receiver to properly increment the worker queue records.

Script for SQL Server

```
CREATE TABLE [dbo].[<Tablename>] (
    [recordId] [bigint] IDENTITY (1, 1) NOT NULL ,
    [messageId] [varchar] (255) NULL,
    [payloadName] [varchar] (255) COLLATE SQL_Latin1_General_CP1_CI_AS NULL ,
    [payloadBinaryContent] [IMAGE] NULL ,
    [payloadTextContent] [TEXT] NULL,
    [localFileName] [varchar] (255) COLLATE SQL_Latin1_General_CP1_CI_AS NULL ,
    [service] [varchar] (255) COLLATE SQL_Latin1_General_CP1_CI_AS NOT NULL ,
    [action] [varchar] (255) COLLATE SQL_Latin1_General_CP1_CI_AS NOT NULL ,
    [arguments] [varchar] (255) COLLATE SQL_Latin1_General_CP1_CI_AS NULL ,
    [fromPartyId] [varchar] (255) COLLATE SQL_Latin1_General_CP1_CI_AS NULL ,
    [messageRecipient] [varchar] (255) COLLATE SQL_Latin1_General_CP1_CI_AS NULL ,
    [errorCode] [varchar] (255) COLLATE SQL_Latin1_General_CP1_CI_AS NULL ,
    [errorMessage] [varchar] (255) COLLATE SQL_Latin1_General_CP1_CI_AS NULL ,
    [processingStatus] [varchar] (255) COLLATE SQL_Latin1_General_CP1_CI_AS NULL ,
    [applicationStatus] [varchar] (255) COLLATE SQL_Latin1_General_CP1_CI_AS NULL ,
    [encryption] [varchar] (10) COLLATE SQL_Latin1_General_CP1_CI_AS NOT NULL ,
    [receivedTime] [varchar] (255) COLLATE SQL_Latin1_General_CP1_CI_AS NULL ,
    [lastUpdateTime] [varchar] (255) COLLATE SQL_Latin1_General_CP1_CI_AS NULL ,
    [processId] [varchar] (255) COLLATE SQL_Latin1_General_CP1_CI_AS NULL ,
) ON [PRIMARY]
GO
```

Server-Side Persistent Cache Schema:

Messaging Cache for SQL Server (server-side persistent cache):

```
CREATE TABLE [dbo].[messagingcache] (
    [sequence] [int] IDENTITY (1, 1) NOT NULL ,
    [partyId] [char] (50) COLLATE SQL_Latin1_General_CP1_CI_AS NULL ,
    [convId] [char] (50) COLLATE SQL_Latin1_General_CP1_CI_AS NULL ,
    [recordId] [char] (50) COLLATE SQL_Latin1_General_CP1_CI_AS NULL ,
    [response] [text] COLLATE SQL_Latin1_General_CP1_CI_AS NULL ,
    [timestamp] [char] (30) COLLATE SQL_Latin1_General_CP1_CI_AS NULL ,
    [status] [char] (10) COLLATE SQL_Latin1_General_CP1_CI_AS NULL
) ON [PRIMARY] TEXTIMAGE_ON [PRIMARY]
```

Messaging Cache for Oracle:

```
CREATE TABLE messagingcache (
    sequence NUMBER (20) NOT NULL,
    partyId VARCHAR2 (255) NULL,
    convId VARCHAR2 (255) NULL,
    recordId VARCHAR2 (255) NULL,
    response CLOB,
    timestamp VARCHAR2 (50) NULL,
    status VARCHAR2 (20) NULL);
CREATE SEQUENCE messagingcache_record_count
INCREMENT BY 1
START WITH 10
MINVALUE 1
MAXVALUE 9999999999999999999999
NOCYCLE
NOORDER
CACHE 20;
```

Appendix B.

Transport Queue Generation Scripts

Transport Queue Generation Script for Oracle

```
DROP TABLE TransportQ_out;
CREATE TABLE TransportQ_out (
    recordId NUMBER (20) NOT NULL,
    messageId VARCHAR2 (255) NULL,
    payloadFile VARCHAR2 (255) NULL,
    payloadContent BLOB,
    destinationFilename VARCHAR2 (255) NULL,
    routeInfo VARCHAR2 (255) NOT NULL,
    service VARCHAR2 (255) NOT NULL,
    action VARCHAR2 (255) NOT NULL,
    arguments VARCHAR2 (255) NULL,
    messageRecipient VARCHAR2 (255) NULL,
    messageCreationTime VARCHAR2 (255) NULL,
    encryption VARCHAR2 (10) NOT NULL,
    signature VARCHAR2 (10) NOT NULL,
    publicKeyLdapAddress VARCHAR2 (255) NULL,
    publicKeyLdapBaseDN VARCHAR2 (255) NULL,
    publicKeyLdapDN VARCHAR2 (255) NULL,
    certificateURL VARCHAR2 (255) NULL,
    processingStatus VARCHAR2 (255) NULL,
    transportStatus VARCHAR2 (255) NULL,
    transportErrorCode VARCHAR2 (255) NULL,
    applicationStatus VARCHAR2 (255) NULL,
    applicationErrorCode VARCHAR2 (255) NULL,
    applicationResponse VARCHAR2 (255) NULL,
    messageSentTime VARCHAR2 (255) NULL,
    messageReceivedTime VARCHAR2 (255) NULL,
    responseMessageId VARCHAR2 (255) NULL,
    responseArguments VARCHAR2 (255) NULL,
    responseLocalFile VARCHAR2 (255) NULL,
```



```

        [messageCreationTime] [char] (255) COLLATE SQL_Latin1_General_CP1_CI_AS
NULL ,
        [encryption] [char] (10) COLLATE SQL_Latin1_General_CP1_CI_AS NOT NULL ,
        [signature] [char] (10) COLLATE SQL_Latin1_General_CP1_CI_AS NOT NULL ,
        [publicKeyLdapAddress] [char] (255) COLLATE SQL_Latin1_General_CP1_CI_AS
NULL ,
        [publicKeyLdapBaseDN] [char] (255) COLLATE SQL_Latin1_General_CP1_CI_AS
NULL ,
        [publicKeyLdapDN] [char] (255) COLLATE SQL_Latin1_General_CP1_CI_AS NULL ,
        [certificateURL] [char] (255) COLLATE SQL_Latin1_General_CP1_CI_AS NULL,
        [processingStatus] [char] (255) COLLATE SQL_Latin1_General_CP1_CI_AS NULL ,
        [transportStatus] [char] (255) COLLATE SQL_Latin1_General_CP1_CI_AS NULL ,
        [transportErrorCode] [char] (255) COLLATE SQL_Latin1_General_CP1_CI_AS NULL ,
        [applicationStatus] [char] (255) COLLATE SQL_Latin1_General_CP1_CI_AS NULL ,
        [applicationErrorCode] [char] (255) COLLATE SQL_Latin1_General_CP1_CI_AS NULL
,
        [applicationResponse] [char] (255) COLLATE SQL_Latin1_General_CP1_CI_AS NULL
,
        [messageSentTime] [char] (255) COLLATE SQL_Latin1_General_CP1_CI_AS NULL ,
        [messageReceivedTime] [char] (255) COLLATE SQL_Latin1_General_CP1_CI_AS
NULL ,
        [responseMessageId] [char] (255) COLLATE SQL_Latin1_General_CP1_CI_AS NULL ,
        [responseArguments] [char] (255) COLLATE SQL_Latin1_General_CP1_CI_AS NULL,
        [responseLocalFile] [char] (255) COLLATE SQL_Latin1_General_CP1_CI_AS NULL ,
        [responseFilename] [char] (255) COLLATE SQL_Latin1_General_CP1_CI_AS NULL ,
        [responseContent] [IMAGE] NULL ,
        [responseMessageOrigin] [char] (255) COLLATE SQL_Latin1_General_CP1_CI_AS
NULL ,
        [responseMessageSignature] [char] (255) COLLATE SQL_Latin1_General_CP1_CI_AS
NULL ,
        [priority] [int] NULL
) ON [PRIMARY]
GO

```


Appendix C.

Worker Queue Generation Scripts

Note: The sequence name below must be <Tablename>_record_count for the Message Receiver to properly increment the worker queue records.

Worker Queue Generation Scripts for Oracle

```
DROP TABLE <Tablename>;
CREATE TABLE <Tablename> (
    recordId NUMBER (20) NOT NULL,
    messageId VARCHAR2 (255) NULL,
    payloadName VARCHAR2 (255) NULL,
    payloadBinaryContent BLOB,
    payloadTextContent CLOB,
    localFilename VARCHAR2 (255) NULL,
    service VARCHAR2 (255) NOT NULL,
    action VARCHAR2 (255) NOT NULL,
    arguments VARCHAR2 (255) NULL,
    fromPartyId VARCHAR2 (255) NULL,
    messageRecipient VARCHAR2 (255) NULL,
    errorCode VARCHAR2 (255) NULL,
    errorMessage VARCHAR2 (255) NULL,
    processingStatus VARCHAR2 (255) NULL,
    applicationStatus VARCHAR2 (255) NULL,
    encryption VARCHAR2 (10) NOT NULL,
    receivedTime VARCHAR2 (255) NULL,
    lastUpdateTime VARCHAR2 (255) NULL,
    processId VARCHAR2 (255) NULL);
ALTER TABLE <Tablename>
ADD PRIMARY KEY (recordId);
CREATE SEQUENCE <Tablename>_record_count
INCREMENT BY 1
START WITH 10
MINVALUE 1
```


Appendix D.

Route-not-Read SQL Generation Scripts

Broadcast Generation Script for SQL Server

```
CREATE TABLE [dbo].[broadcast] (  
    [name] [char] (100) COLLATE SQL_Latin1_General_CP1_CI_AS NOT NULL ,  
    [addresses] [char] (1000) COLLATE SQL_Latin1_General_CP1_CI_AS NOT NULL  
    ) ON [PRIMARY]  
GO
```

Broadcast Generation Script for Oracle

```
CREATE TABLE broadcast (  
    name VARCHAR2 (255) NOT NULL,  
    addresses VARCHAR2 (255) NOT NULL);
```

Messagebin Generation Script for Oracle

```
DROP TABLE <Tablename>;  
CREATE TABLE <Tablename> (  
    recordId NUMBER (20) NOT NULL,  
    messageId VARCHAR2 (255) NULL,  
    payloadName VARCHAR2 (255) NULL,  
    payloadBinaryContent BLOB,  
    payloadTextContent CLOB,  
    localFilename VARCHAR2 (255) NULL,  
    service VARCHAR2 (255) NOT NULL,  
    action VARCHAR2 (255) NOT NULL,  
    arguments VARCHAR2 (255) NULL,  
    fromPartyId VARCHAR2 (255) NULL,  
    messageRecipient VARCHAR2 (255) NULL,  
    errorCode VARCHAR2 (255) NULL,  
    errorMessage VARCHAR2 (255) NULL,  
    processingStatus VARCHAR2 (255) NULL,
```

```
applicationStatus VARCHAR2 (255) NULL,  
encryption VARCHAR2 (10) NOT NULL,  
receivedTime VARCHAR2 (255) NULL,  
lastUpdateTime VARCHAR2 (255) NULL,  
processId VARCHAR2 (255) NULL);
```

```
ALTER TABLE <Tablename>  
ADD PRIMARY KEY (recordId);  
CREATE SEQUENCE <Tablename>_record_count  
INCREMENT BY 1  
START WITH 10  
MINVALUE 1  
MAXVALUE 99999999999999999999999999999999  
NOCYCLE  
NOORDER  
CACHE 20;
```

Messagebin Generation Script for SQL Server

```
CREATE TABLE [dbo].[<Tablename>] (  
    [recordId] [bigint] IDENTITY (1, 1) NOT NULL ,  
    [messageId] [varchar] (255) NULL,  
    [payloadName] [varchar] (255) COLLATE SQL_Latin1_General_CP1_CI_AS NULL ,  
    [payloadBinaryContent] [IMAGE] NULL ,  
    [payloadTextContent] [TEXT] NULL,  
    [localFileName] [varchar] (255) COLLATE SQL_Latin1_General_CP1_CI_AS NOT  
NULL ,  
    [service] [varchar] (255) COLLATE SQL_Latin1_General_CP1_CI_AS NOT NULL ,  
    [action] [varchar] (255) COLLATE SQL_Latin1_General_CP1_CI_AS NOT NULL ,  
    [arguments] [varchar] (255) COLLATE SQL_Latin1_General_CP1_CI_AS NULL ,  
    [fromPartyId] [varchar] (255) COLLATE SQL_Latin1_General_CP1_CI_AS NULL ,  
    [messageRecipient] [varchar] (255) COLLATE SQL_Latin1_General_CP1_CI_AS NULL  
,  
    [errorCode] [varchar] (255) COLLATE SQL_Latin1_General_CP1_CI_AS NULL ,  
    [errorMessage] [varchar] (255) COLLATE SQL_Latin1_General_CP1_CI_AS NULL ,  
    [processingStatus] [varchar] (255) COLLATE SQL_Latin1_General_CP1_CI_AS NULL ,  
    [applicationStatus] [varchar] (255) COLLATE SQL_Latin1_General_CP1_CI_AS NULL ,
```

```
[encryption] [varchar] (10) COLLATE SQL_Latin1_General_CP1_CI_AS NOT NULL ,
[receivedTime] [varchar] (255) COLLATE SQL_Latin1_General_CP1_CI_AS NULL ,
[lastUpdateTime] [varchar] (255) COLLATE SQL_Latin1_General_CP1_CI_AS NULL ,
[processId] [varchar] (255) COLLATE SQL_Latin1_General_CP1_CI_AS NULL ,
) ON [PRIMARY]
GO
```

Users Generation Script for SQL Server

```
CREATE TABLE [dbo].[users] (
    [name] [char] (100) COLLATE SQL_Latin1_General_CP1_CI_AS NOT NULL ,
    [description] [char] (255) COLLATE SQL_Latin1_General_CP1_CI_AS NULL
) ON [PRIMARY]
GO
```

Users Generation Script for Oracle

```
CREATE TABLE users (
    name VARCHAR2 (255) NOT NULL,
    description VARCHAR2 (255) NULL);
```

PartyID_User Generation Script for SQL Server

```
CREATE TABLE [dbo].[partyid_user] (
    [partyId] [char] (255) COLLATE SQL_Latin1_General_CP1_CI_AS NOT NULL,
    [user] [char] (255) COLLATE SQL_Latin1_General_CP1_CI_AS NOT NULL
    [sdnuser] [char] (255) COLLATE SQL_Latin1_General_CP1_CI_AS NOT NULL
) ON [PRIMARY]
GO
```

PartyID_User generation Script for Oracle

```
CREATE TABLE partyid_user (  
    partyId VARCHAR2 (255) NOT NULL,  
    user VARCHAR2 (255) NOT NULL),  
    sdnuser VARCHAR2 (255) NOT NULL);
```

Appendix E.

File-Based Message Queue

The message queue can also be implemented using operating system files, without using a relational database table. The file descriptors can be name-value pairs or XML files.

Name-Value Based File Descriptor

The following is a name-value based file descriptor :

```
recordId=22
payloadFile=d:\\projects\\clebint\\ebxmlvob\\outgoing\\test.txt
destinationFilename=test.txt
routeInfo=OKLAHOMA
service=Router
action=send
arguments=xyz
messageRecipient=list56
messageCreationTime=time
encryption=no
signature=yes
publicKeyLdapAddress=directory.verisign.com:389
publicKeyLdapBaseDN=o=Centers for Disease Control and Prevention
publicKeyLdapDN=cn=Rajashekar Kailar
acknowledgementFile=d:\\projects\\clebint\\ebxmlvob\\file_send_acks
\\ack_send.props
```

The response from a file-send operation is written to the **acknowledgementFile** specified in the outgoing file descriptor, shown previously, and looks like the following:

```
transportStatus=success
transportError=none
applicationStatus=retrieveSucceeded
applicationError=none
applicationData=TargetTable=payroll
responseLocalFile=1018379449158
responseFileName=test.txt
responseSignature=unsigned
responseMessageOrigin=LABCORP
```

For a detailed description of these fields, refer to the Message Queue table schema, which contains fields with the same name and semantics.

XML Based File Descriptor

The following is an example of an XML file descriptor:

```
<fileDescriptor>
  <recordId>22</recordId>
  <payloadFile>d:\\projects\\clebint\\ebxmlvob\\outgoing\\test.txt<
/payloadFile>
  <payloadContent></payloadContent>
  <destinationFilename>test.txt</destinationFilename>
  <routeInfo>OKLAHOMA</routeInfo>
```

```

<service>Router</service>
<action>send</action>
<arguments>xyz</arguments>
<messageRecipient>list56</messageRecipient>
<messageCreationTime>time</messageCreationTime>
<encryption>yes</encryption>
<signature>yes</signature>
<messageRecipient>list56</messageRecipient>
<publicKeyLdapAddress>directory.verisign.com:389</publicKeyLdapAddress>
<publicKeyLdapBaseDN>o=CDC</publicKeyLdapBaseDN>
<publicKeyLdapDN>cn=Rajashekar Kailar</publicKeyLdapDN>
<acknowledgementFile>
d:\projects\clebint\ebxmlvob\filesend_acks\ack_send.xml
</acknowledgementFile>
</fileDescriptor>

```

The response from a file-send operation is written to the **acknowledgementFile** specified in the outgoing file descriptor, shown previously, and looks like the following:

```

<acknowledgement>
  <transportStatus>success</transportStatus>
  <transportError>none</transportError>
  <applicationStatus>retrieveSucceeded</applicationStatus>
  <applicationError>none</applicationError>
  <applicationData>targetTable=payroll</applicationData>
  <responseLocalFile>1018387200432</responseLocalFile>
  <responseFileName>test.txt</responseFileName>
  <responseSignature>unsigned</responseSignature>
  <responseMessageOrigin>LABCORPDUNSNUMBER</responseMessageOrigin>
</acknowledgement>

```

For a detailed description of these fields, refer to the Message Queue Table Schema, which contains fields with the same name, and semantics.

Appendix F.

Transport Level Status and Error Codes

The following status and error codes may be written to the message queues based on the outcome of the message delivery or processing. Applications that use the PHINMS system can read these codes and act on them.

Status Codes

Status	Description
success	Message Send/Receive operation successful
failure	Message Send/Receive operation failure

Error Codes

ErrorCode	Description
SecurityFailure	Error logging into Message Receiver
DeliveryFailure	Failed to deliver message
NotSupported	Format of ebXML message or CPA unsupported
Unknown	Not a standard ebxml error
NoSuchService (*)	Service/Action did not map to a service on the Message receiver
ChecksumFailure (*)	File checksum verification failure at the Message Receiver

(*) Custom error codes - not in ebXML specification.

Appendix G.

Example receiver.xml File

```
<Receiver>
  <logDir>c:\phmsgServer_2_0\logs\</logDir>
  <logLevel>messages</logLevel>
  <incomingDir>c:\phmsgServer_2_0\incoming\</incomingDir>
  <myPartyId>cdc</myPartyId>
  <cpaLocation>c:\phmsgServer_2_0\config\CPA\</cpaLocation>
  <serviceMap>c:\phmsgServer_2_0\config\servicemap.xml</serviceMap>
  <passwordFile>c:\phmsgServer_2_0\config\receiverpasswds</passwordFile>
  <keyStore>c:\phmsgServer_2_0\config\phmsg2.pfx</keyStore>
  <keyStorePasswd>keyStorePasswd1</keyStorePasswd>
  <trustStore>c:\phmsgServer_2_0\keystores\cacerts</trustStore>
  <trustStorePasswd>cacertsPasswd1</trustStorePasswd>
  <signatureRequired>>false</signatureRequired>
  <signingCertsLocation>c:\phmsgServer_2_0\config\signingcerts</signingCertsLocation>
  <key>q490uradf</key>
  <seed>214150145125193</seed>
  <uccTest>>false</uccTest>
</Receiver>
```

Appendix H.

Example Collaborative Protocol Agreement (CPA)

```
<?xml version="1.0" ?>

<tp:CollaborationProtocolAgreement
xmlns:tp="http://www.ebxml.org/namespaces/tradePartner"
xmlns:xsi="http://www.w3.org/2000/10/XMLSchema-instance"
xsi:schemaLocation="http://www.ebxml.org/namespaces/tradePartner
http://ebxml.org/project_teams/trade_partner/cpp-cpa-v1_0.xsd"
xmlns:xlink="http://www.w3.org/1999/xlink"
xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
tp:cpaid="uri:yoursandmycpa" tp:version="1.2">

<tp:Status tp:value="proposed" />
<tp:Start>2001-05-20T07:21:00Z</tp:Start>
<tp:End>2002-05-20T07:21:00Z</tp:End>
<tp:ConversationConstraints tp:invocationLimit="100"
tp:concurrentConversations="100" />

<tp:PartyInfo>
  <tp:PartyId tp:type="DUNS">LABCORPDUNSNUMBER</tp:PartyId>
  <tp:PartyRef xlink:href="http://www.lab.com/about.html" />

  <tp:Transport tp:transportId="N05">
    <tp:SendingProtocol tp:version="1.1">HTTP</tp:SendingProtocol>
    <tp:ReceivingProtocol
tp:version="1.1">HTTP</tp:ReceivingProtocol>
    <tp:Endpoint tp:uri="www.lab.com/soapreceiver/receiver"
tp:type="allPurpose" >
    <tp:TransportSecurity>
      <tp:Protocol tp:version="3.0">SSL</tp:Protocol>
      <tp:CertificateRef tp:certId="N03" />
    </tp:TransportSecurity>
  </tp:Transport>
</tp:PartyInfo>

<tp:PartyInfo>

  <tp:PartyId tp:type="DUNS">CDCDUNSNUMBER</tp:PartyId>
  <tp:PartyRef xlink:type="simple"
xlink:href="http://www.cdc.gov/about.html" />

  <tp:Transport tp:transportId="N35">
    <tp:SendingProtocol tp:version="1.1">HTTPS</tp:SendingProtocol>
    <tp:ReceivingProtocol
tp:version="1.1">HTTPS</tp:ReceivingProtocol>
    <tp:Endpoint tp:uri="phmsg.cdc.gov/ebxml/receivefile"
tp:type="allPurpose" />
    <tp:TransportSecurity>
      <tp:Protocol tp:version="3.0">SSL</tp:Protocol>
      <tp:CertificateRef> </tp:CertificateRef>
      <tp:authenticationType>basic</tp:authenticationType>
    </tp:TransportSecurity>
  </tp:Transport>
</tp:PartyInfo>

</tp:CollaborationProtocolAgreement>
```

```
<!-- basic, custom, sdn, clientcert -->
<tp:basicAuth>
  <tp:indexPage>/test.html</tp:indexPage>
  <tp:basicAuthUser>phmsgUser1</tp:basicAuthUser>
  <tp:basicAuthPasswd>phmsgPasswd1</tp:basicAuthPasswd>
</tp:basicAuth>
</tp:TransportSecurity>
</tp:Transport>

</tp:PartyInfo>

<tp:Comment xml:lang="en-us">send/receive agreement between cdc and
Labcorp</tp:Comment>

</tp:CollaborationProtocolAgreement>
```

Appendix I.

Example Receiver Password File

```
<?xml version="1.0"?>
<passwordFile>
  <trustStorePass>changeit</trustStorePass>
  <keyStorePass>passphrase</keyStorePass>
  <dbUser>phmsg</dbUser>
  <dbPasswd>phmsg123</dbPasswd>
  <mysqlDbUser>root</mysqlDbUser>
  <mysqlDbPasswd>sql</mysqlDbPasswd>
</passwordFile>
```

Appendix J.

Example Routerconfig.xml File

```
_ <Router>
  _ <!--
    SETTINGS FOR MYSQL DB ON LINUX
    <jdbcDriver>org.gjt.mm.mysql.Driver</jdbcDriver>
    <databaseUrl>jdbc:mysql://158.111.1.164:3306/test</databaseUrl>
    <databaseUser>mysqlDbUser</databaseUser>
    <databasePasswd>mysqlDbPasswd</databasePasswd>
  -->
  _ <!--
    SETTINGS FOR SQL SERVER DB ON NEDSS-SQL3
    values can be sqlServer, oracle
  -->
  <dbType>sqlServer</dbType>

  <jdbcDriver>com.microsoft.jdbc.sqlserver.SQLServerDriver</jdbcDriver>
  <databaseUrl>jdbc:microsoft:sqlserver://nedss-
    sql3.cdc.gov:1433;DatabaseName=Phmsg</databaseUrl>
  <databaseUser>sqlServerDbUserRemote</databaseUser>
  <databasePasswd>sqlServerDbPasswdRemote</databasePasswd>
  - <!--
  END SETTINGS FOR SQL SERVER
  -->
  _ <!--
    SETTINGS FOR ORACLE ON NEDSS-REPORT
```

```
<dbType>oracle</dbType>
<jdbcDriver>oracle.jdbc.driver.OracleDriver</jdbcDriver>
<databaseUrl>jdbc:oracle:thin:@nedss-report:1521:ebxml</databaseUrl>
<databaseUser>oracleServerDbUserRemote</databaseUser>
<databasePasswd>oracleServerDbPasswdRemote</databasePasswd>

-->
<sdnAuth>>false</sdnAuth>

  <passwordFile>d:\tomcat4\ebxml\config\receiverpasswords</passwordFile>
<payloadDir>d:\tomcat4\ebxml\payloaddir\</payloadDir>
<logLevel>info</logLevel>
<logDir>d:\tomcat4\ebxml\logs\</logDir>
<maxLogSize>10000000</maxLogSize>
<logArchive>>true</logArchive>
<maxMessageBufferSize>1000000</maxMessageBufferSize>
<deleteOnPickup>true</deleteOnPickup>
<key>2o3i23</key>
<seed>151209139182126100100162</seed>
</Router>
```