## Public Health Information Network Messaging System

# Client Installation Guide




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

# Table of Contents

# Introduction

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

- JDK1.4
- Java Trust Store
- Collaboration Protocol Agreements

# Detailed Description

The Message Sender, the client, is an ebXML compliant Java application that resides on the host that performs the message send operation. The Message Sender performs the following:

- Initialization
- Polling Modes
- Operations and Transformations

## Initialization

During initialization the Message Sender does the following:

- Reads its configuration file and prompts the user for a single password.
- Uses the password to decrypt the user's passwords file, which contains all of the passwords the Message Sender needs to perform authentication. Other configuration files make references to the passwords within this encrypted passwords file.
- Stores the decrypted passwords in memory and uses them during the Message Sender's uptime.

## Polling Modes

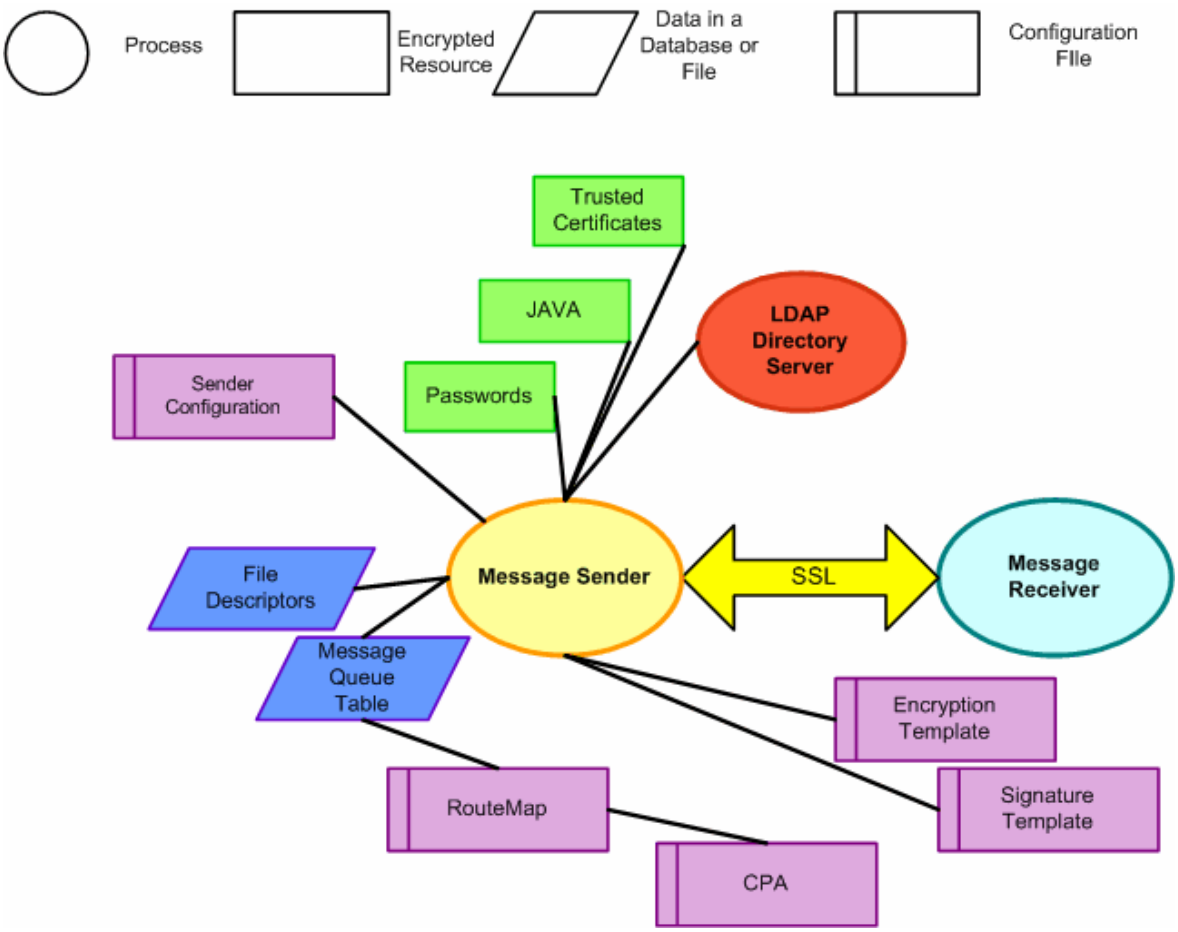
The Message Sender operates in one of two polling modes, which is specified in its configuration file:

- **Database Polling Mode** – The application that creates messages writes data to a Transport Queue table. The Message Sender polls the Transport Queue table for outgoing messages. Message responses are written back to the Transport Queue table.

- **File-Based Polling Mode** - The application that uses the PHINMS creates file descriptors in a file system directory. The Message Sender polls this directory for file descriptors and, when it finds them, sends the corresponding file as an ebXML compliant message and then creates a response descriptor.

## *Operations and Transformations*

When the Message Sender finds new outgoing data, in a Transport Queue database table or a file descriptor, it performs the following operations and transformations in the diagram below:



### *Route Mapping*

A configuration file, called **routeMap**, maps the route to its Collaboration Protocol Agreement, the CPA. The route is specified in a field in the Transport Queue database table or as a field in the file descriptor that is associated with an outgoing message.

### *Collaboration Protocol Agreement Parsing*

The CPA is read to determine the Message Receiver's end point, and security attributes, such as the authentication mode.

### *LDAP Public Key Searching*

If the Message Sender has selected the encryption option and if the LDAP attributes of the Message Receiver's public key certificate are specified, an LDAP search is performed and the Message Receiver's public key is retrieved.

### *Message Encryption and Signing*

If the Message Sender has selected the message signature option, which is a field in the Transport Queue table or a field in the file system directory, the message is signed using the XML Signature standard.

A signature template configuration file, in XML format, is used to generate the signature. Afterward, the message is encrypted using the XML Encryption standard. An encryption template configuration file, in XML format, is used to generate the cipher text.

### *SSL and Authentication*

Using SSL, the Message Sender connects to the Message Receiver's end point, a URL specified in the CPA, and then performs authentication with the Message Receiver. If the CPA specifies a basic or custom authentication mode, the user name and password parameters are read from the CPA and from an encrypted passwords file.

### *SOAP Call*

The ebXML compliant message, which includes the file payload and message envelope, is sent to the Message Receiver in a SOAP call.

### *Response Parsing and Writing to Database Table or Descriptor*

The response from the Message Receiver is parsed for transport and application status information. This information is written to the Transport Queue table or the acknowledgement descriptor.

## *Transport Queue Interface*

The interface between the message creation component and the message transport component is a relational database table, called the **transport queue**. The schema of this table is shown below:

| Field | Description | Data Type |
|---|---|---|
| recordId | Unique ID of the record in the table and the table's primary key. The ID is an integer. | Required<br>SQL Server: Integer Identity=Yes Identity Increment=1<br>Oracle: Integer (20), not null<br>For Oracle, use the sequence object **transport_record_count** to populate this field. |
| messageId | ID of the application level message. | Optional<br>SQL Server: char (255), null<br>Oracle: varchar2 (255), null |
| payloadFile | Filename of the payload file of an outgoing message, relative to a local directory, such as **myinputs.txt**. The local directory is configurable. | Optional<br>SQL Server: char (255), null<br>Oracle: varchar2 (255), null |
| payloadContent | This field is used only when the **payloadFile** field is not specified. It directly populates the contents of a file within the table. | SQL Server: Image data type, null<br>Oracle: BLOB data type, null |
| destinationFilename | The name of the payload file when it is stored on the receiver/handler. | SQL Server: char (255,) null<br>Oracle: varchar2 (255), null |
| routeInfo | Points to the **routemap** table, which points to the message route. This entry within the **routemap** maps to a CPA, a configuration file, which maps this to the URL of the Message Receiver to which the message is forwarded. | Required<br>The value is lowercase.<br>SQL Server: char (255), not null<br>Oracle: varchar2 (255), not null |
| Service | The name of the service that gets invoked by the Message Receiver when it receives the | Required<br>SQL Server: char (255), not null<br>Oracle: varchar2 (255), not null |

| Field | Description | Data Type |
|---|---|---|
| | message. The message payload is passed to the service. | |
| Action | The action or method within the service that gets invoked by the Message Receiver when it receives the message. The message payload is passed to the service. | Required<br>The value is lowercase.<br>SQL Server: char (255), not null<br>Oracle: varchar2 (255), not null |
| arguments | Arguments sent to the service, which is invoked by the Message Receiver when it receives the message. | Optional<br>SQL Server: char (255), null<br>Oracle: varchar2 (255), null |
| messageRecipient | This field is used in the **route-not-read** case to specify the target message recipient's party ID. | SQL Server: char (255), null<br>Oracle: varchar2 (255), null |
| messageCreationTime | Message creation time. Currently, the Message Sender does not use/validate the **messageCreationTime** field. This field is used by the application and its format is determined by the application. However, UTC time format is recommended. | SQL Server: char (255), null<br>Oracle: varchar2 (255), null |
| encryption | **Yes/No** If yes, XML encryption is applied to the payload. | SQL Server: char (10), not null<br>Oracle: varchar2 (10), not null |
| signature | **Yes/No** If yes, XML signature is applied to the payload. | SQL Server: char (10), not null<br>Oracle: varchar2 (10), not null |
| publicKeyLdapAddress | LDAP address of the LDAP directory server. | SQL Server: char (255), null<br>Oracle: varchar2 (255), null |
| publicKeyLdapBaseDN | LDAP Base Distinguished Name of the public key, such as o=Centers for Disease Control and Prevention. | SQL Server: char (255), null<br>Oracle: varchar2 (255), null |
| publicKeyLdapDN | LDAP Distinguished Name of the public key such as cn=Rajashekar Kailar. | SQL Server: char (255), null<br>Oracle: varchar2 (255), null |

| Field | Description | Data Type |
|---|---|---|
| certificateURL | URL of a recipient's public key certificate. This field is used if **ldapKeyRetrieval**=false in **sender.xml**. By default, **ldapKeyRetrieval**=true). Only http:// or file:/// URLs may be specified, not HTTPS. Only BASE64 encoded certificate files are supported. | SQL Server: char (255), null<br>Oracle: varchar2 (255), null |
| processingStatus | Processing status of the record in the table. Values are **queued**, **attempted**, **sent**, **received**, and **done**. | Required<br>The value is lowercase.<br>SQL Server: char (255), not null<br>Oracle: varchar2 (255), not null |
| transportStatus | Transport level status. Values are: **success** - indicates message was successfully delivered to **routeInfo**, **failure** indicates message delivery failed. | SQL Server: char (255), null<br>Oracle: varchar2 (255), null |
| transportErrorCode | Error code that describes the transport failure. | SQL Server: char (255), null<br>Oracle: varchar2 (255), null |
| applicationStatus | The application status that is returned by the service/action, which is invoked by the Message Receiver in a synchronous manner. | SQL Server: char (255), null<br>Oracle: varchar2 (255), null |
| applicationErrorCode | The error code returned by the service/action in a synchronous manner. | SQL Server: char (255), null<br>Oracle: varchar2 (255), null |
| applicationResponse | The synchronous response returned by the service/action. | SQL Server: char (255), null<br>Oracle: varchar2 (255), null |
| messageSentTime | Time when the message was sent. Format is UTC.<br>Example:<br>`2001-10-011T16:01:01` | SQL Server: char (255), null<br>Oracle: varchar2 (255), null |
| messageRecievedTime | Message received time-stamp in UTC format.<br>Example:<br>`2001-10-01T16:01:01` | SQL Server: char (255), null<br>Oracle: varchar2 (255), null |

| Field | Description | Data Type |
|---|---|---|
| responseMessageId | Message ID of the response message in the **route-not-read** scenario. | SQL Server: char (255), null<br>Oracle: varchar2 (255), null |
| responseArguments | This field is used in the **route-not-read** scenario to convey arguments that have been sent by a Message Sender to a receiving client. | SQL Server: char (255), null<br>Oracle: varchar2 (255), null |
| responseLocalFile | In the **route-not-read** scenario, the response to a poll type request may contain a payload file. This file written to a local configurable folder under a unique filename. This filename is written to **responseLocalFile** field. | SQL Server: char (255), null<br>Oracle: varchar2 (255), null |
| responseFilename | In the **route-not-read** scenario this field is the response filename. | SQL Server: char (255), null<br>Oracle: varchar2 (255), null |
| responseContent | Response contents in Image/BLOB format.<br>This field is used when the **sender.xml** configuration file in the Message Sender specifies that the response payload should be written into a database field instead of to disk (When **responseToDb**=true). | SQL Server: Image data type, null<br>Oracle: BLOB data type, null |
| responseMessageOrigin | In the **route-not-read** scenario this field is the Party ID of the party that originated this message. | SQL Server: char (255), null<br>Oracle: varchar2 (255), null |
| responseMessageSignature | In the **route-not-read** scenario this field is the Party ID of the party that signed this message. | SQL Server: char (255), null<br>Oracle: varchar2 (255), null |
| priority | An integer that indicates the request's priority. 1 is the highest priority. Higher integers have lower priority. 0 is reserved for ping messages. | Required<br>SQL Server: integer, null<br>Oracle: integer, null |

## Transport Queue Scripts

To create the transport queue you can cut and paste the following Oracle and SQL Server scripts into your program:

### Script (DDL) 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,
responseFilename VARCHAR2 (255) NULL,
responseContent BLOB,
responseMessageOrigin VARCHAR2 (255) NULL,
responseMessageSignature VARCHAR2 (255) NULL,
priority NUMBER (1) NULL
);
ALTER TABLE TransportQ_out
ADD PRIMARY KEY (recordId);
CREATE SEQUENCE transport_record_count
 INCREMENT BY 1
 START WITH 10
 MINVALUE 1
 MAXVALUE 9999999999999999999999999999
 NOCYCLE
 NOORDER
 CACHE 20;
```

### *Script for SQL Server:*

```
if exists (select * from dbo.sysobjects where id =
object_id(N'[dbo].[TransportQ_out]') and OBJECTPROPERTY(id, N'IsUserTable') =
1)
drop table [dbo].[TransportQ_out]
GO

CREATE TABLE [dbo].[TransportQ_out] (
[recordId] [bigint] IDENTITY (1, 1) NOT NULL ,
[messageId] [char] (255) NULL,
[payloadFile] [char] (255) COLLATE SQL_Latin1_General_CP1_CI_AS NULL ,
[payloadContent] [IMAGE] NULL ,
[destinationFilename] [char] (255) COLLATE SQL_Latin1_General_CP1_CI_AS NULL ,
[routeInfo] [char] (255) COLLATE SQL_Latin1_General_CP1_CI_AS NOT NULL ,
[service] [char] (255) COLLATE SQL_Latin1_General_CP1_CI_AS NOT NULL ,
[action] [char] (255) COLLATE SQL_Latin1_General_CP1_CI_AS NOT NULL ,
[arguments] [char] (255) COLLATE SQL_Latin1_General_CP1_CI_AS NULL ,
[messageRecipient] [char] (255) COLLATE SQL_Latin1_General_CP1_CI_AS 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
```

### XML File Descriptor

The following is an example of an XML File Descriptor:

```
<fileDescriptor>
      <recordId>22</recordId>
      <payloadFile>d:\projects\clebint\ebxmlvob\outgoing\test.txt</paylo
            adFile>
      <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</publicKeyLdapAdd
            ress>
      <publicKeyLdapBaseDN>o=CDC</publicKeyLdapBaseDN>
      <publicKeyLdapDN>cn=Rajashekar Kailar</publicKeyLdapDN>
      <acknowledgementFile>
      d:\projects\clebint\ebxmlvob\filesend_acks\ack_send.xml
      </acknowledgementFile>
</fileDescriptor>
```

### XML File Descriptor Response

The following is an example of an XML File Descriptor response.

```
<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>
```

## File-Based Transport Queue

When the interface, the **Transport Queue**, is implemented as a file system directory, the file descriptors may be name-value pairs or XML standard files. The fields used in the file system directory have the same name and semantics as the ones used in the relational database table.

### *Name-Value Based File Descriptor*

The following is an example of 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
```

### *Response from Sending a File*

The following is an example of a response from a file send operation. The response is written to the **acknowledgementFile** specified in the outgoing file descriptor:

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

### *Transport Level Status and Error Codes*

When a message is delivered or processed a transport status code is sent back to the Transport Queue. If there was an error in the delivery or processing of the message an error code is also sent back to the Transport Queue. Applications that use the PHINMS read these codes and act on them.

The following table describes the transport status and error codes:

| Transport Status Code | Description |
|---|---|
| success | Message send or receive operation was successful. |
| failure | Message send or receive operation failed. |
| **Transport Error Code** | **Description** |
| SecurityFailure | Error logging into Message Receiver. |
| DeliveryFailure | Failed to deliver message. |
| NotSupported | Format of the ebXML message or CPA is 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*

The asterisk (*) symbol indicates a custom error code, which means, the code is not in the ebXML specifications.

**NoSuchService*** - Service/Action did not map to a service on the Message Receiver.

**CheckSumFailure*** - File checksum verification failure at the Message Receiver.

# Installing the Message Sender

## Client System Requirements

- Windows 2000/NT or Unix/Linux operating system with 256 MB RAM.
- Internet Connectivity
- JDK1.4 or above
- 70Mb free disk space for software installation
- For database polling configuration: JDBC compliant relational database with the Message Queue table built as specified.

## Installing the Client on Windows

To install the Message Sender, the client, on Windows, do the following procedures. Step-by-step instructions are included.

1. Download JDK1.4 and install it on your desktop.
2. Set up your environment to use Java.
3. Install Java Trust Store.
4. Configure the client.
5. Set up your host computer.

## Installing the Client on Solaris

To install the Message Sender, the client, on Solaris, do the following procedures.

1. Install the client software in you local directory.
2. Set the **Display** environment variable. See your Solaris guide for commands that will check this variable.
3. Give the user write permissions on the directory to which you installed the client software.
4. Set the execute permissions on the install binary. You can use the **`CHMOD`** command. See your Solaris guide for details.
5. Run the intall binary by typing **SolarisSetup2.0** .
6. To add logging, add the following option: **`-is:log logfilename`**
   This option creates a file in the install directory.
7. Follow the installation prompts.
8. Install the Java Trust Store.

### *To Install JDK1.4*

1. Go to **http://java.sun.com/j2se/1.4/download.html**.
2. Download **J2SE v1.4** and install it on your desktop.

### *To Set up the Java Environment*

1. Add the Java binary directory **JAVA_HOME/bin** to your environment's PATH.
   For example: **d:\jre1.4\bin**

2. Add the **JAVA_HOME** variable to your environment.
   For example: **JAVA_HOME=d:\jre1.4**

3. Verify the **JAVA_HOME** variable is correctly added to your environment by typing the following:

   For Windows: **`echo %echo%PATH%`**
   For UNIX: **`echo $PATH`**
   You will see **JAVA_HOME\bin** in your path.

4. Test your Java installation by doing the following:
   From a DOS window or from a UNIX command line type: **`java`** to run the java interpreter. The following information appears, which indicates the installation is successful :

```
Usage: java [-options] class [args...]
           (to execute a class)
   or  java -jar [-options] jarfile [args...]
           (to execute a jar file)

where options include:
    -hotspot      to select the "hotspot" VM
          -server       to select the "server" VM
    If present, the option to select the VM must be first.
    The default VM is -hotspot.
 …..
```

## *Installing the Client on Solaris*

9. Install the client software in you local directory.
10. Make sure the **Display** environment variable is set. See your Solaris guide for commands that will check this variable.
11. Give the user write permissions on the directory to which you installed the client software.

12. Set the execute permissions on the install binary. You can use the **CHMOD** command. See your Solaris guide for details.
13. Run the intall binary by typing **SolarisSetup2.0** .

### *Installing Java Trust Store*

Java Trust Store is part of the Java Virtual Machine. In version JRE1.4 the file is **\jre1.4\liv\security\cacerts**. This file contains a list of certificates and their serial numbers, which belong to trusted certificate authorities. You can use the Java Trust Store as it is, without modifying it. To add a CA certificate or view a certificate, use the following procedures:

### *To View the Contents of a Trust Store*

1. Type the following command:
   **keytool –list –v keystore <trustorefile> -storepass <storepass>**

2. Substitute **javabin** for the path to the location Java binaries are stored on your machine. For example: **d:\jdk1.4\bin**

3. Substitute **trustorefile** for the file containing the Java Trust Store such as **cacerts**.

4. Substitute **storepass** for the password to the Java Trust Store.

### *To Add a Certificate to a Trust Store*

1. Type the following command:
   **keytool –import –truscacerts –file <cert_file> -storepass <storepass> -keystore <truststore>**

2. Substitute **Javabin** for the path to the location Java binaries are stored on your machine such as **d:\jdk1.4\bin**

3. Substitute **trustorefile** for the name of the file containing Java Trust Store such as **cacerts**.

4. Substitute **storepass** for the password to the Java Trust Store.

# Configuring the Message Sender

To configure the Message Sender, the client, do the following procedures:

1. Specify sensitive passwords in an XML password file in XML format. See the appendix for an example.

2. Using the **pbe.bat** utility described in this document, encrypt the password file in step 1 with a password that has at least eight characters, which are mixed case and a combination of alpha and numeric characters.

3. Set the polling configuration.

4. Create a CPA file for each of the routes to which you want to send messages.

5. Specify the end-point and security attributes within the CPA. See the appendix for an example.

6. Create a **routemap** entry for each CPA.

7. If you are using Secure Data Network, set up your client to perform SDN. Specify the SDN authentication properties in the CPA. See Appendix A for an example.

### *To Use the Database Polling Configuration*

To set up your client to use the database polling configuration, do the following:

1. In the **sender.xml** file, set **dbPoll** to **true**. Set **filePoll** to **false**.
2. Set up the Transport Queue database table.
3. In the **sender.xml** file, add the appropriate JDBC driver and the JDBC URL of the database that contains the Transport Queue database table.
4. Add the JDBC driver to the **lib** project directory.
5. In **setenv.bat** add the JDBC driver classes or JAR files to the class path.
6. Create the outgoing payload directory and then set the **dbPollDi**r to the full path name of this directory.
7. In **sender.xml** set the **databaseUser** and **databasePasswd** indices. Add these indices to your password XML file in XML format.
8. Set the other configurables in **sender.xml**.
9. Create a CPA file for every route you use to send messages.
10. Specify the end point and security attributes in the CPA.
11. Create a **routeMap** entry for each CPA.
12. If you are using Secure Data Network, set up your client to perform SDN. Specify the SDN authentication properties in the CPA. See Appendix A for an example.

### *To Use File- Based Polling*

To set up your client to use the file-based polling configuration, do the following:

1. In the **sender.xml** file, set **filePoll** to **true** and set **dbPoll** to **false**.
2. Specify the **fileDescriptorFormat** as **XML** or **nameValue** and specify the **fileDescriptorDir**.
3. Create a file poll directory and specify it in the **sender.xml** file.
4. Create a CPA file for every route you use to send messages.
5. Specify the end point and security attributes in the CPA.
6. Create a **routeMap** entry for every CPA.

## *Client Configuration Files*

The client, the Message Sender, uses the following configuration files. Each of these files is described in detail in this document.

| File Name | Description |
|---|---|
| sender.xml | Message Sender configuration file. Contains site or host specific information about the Message Sender. |
| routeMap.xml | Maps routes to their CPAs, Collaborative Protocol Agreements. |
| Collaborative Protocol Agreement | Describes mutually agreed upon communication details, such as the Message Receiver's URL and authentication method, between two parties. One CPA for every Message Receiver. The client administrator defines the file name and specifies it in **routeMap.xml**. |
| Encrypted passwords file | XML file containing the template used for XML encryption. |

### *Message Sender Configuration File*

The Message Sender reads its configuration file, **sender.xml**. The following table lists the **sender.xml** fields and their descriptions.

| Field Name | Description |
|---|---|
| filePoll | If the value is **True**, the file system is polled for file descriptors. |
| dbPoll | If the value is **True**, the Transport Queue database table is polled for outgoing messages. |
| routeNotReadPoll | If the value is **True**, a route-not-read server is polled automatically at specified intervals for incoming messages. |
| test | If the value is **True**, a ping message is sent to CDC once every minute. |
| passwordFile | Full path name of the encrypted passwords file. |
| maxAttempts | Maximum number of times a message delivery is attempted. |
| filePollMode | Values are **loop** or **once**. In loop mode, files are continuously polled. In once mode, files are polled once. |
| filePollInterval | Number of seconds between file-based polls. |
| fileDescriptorDir | When the sending application uses file-based polling, it is the directory in which the Message Sender drops the file descriptors. |
| fileDescriptorFormat | File descriptor format: **XML** or **nameValue**. |
| dbPollMode | Database polling mode: **loop** or **once**. |
| dbPollDir | When the sending application uses the database polling mode, it is the directory containing outgoing payload files. |
| incomingDir | Directory to which incoming payload files are written. |
| dbPollInterval | Number of seconds between database polls. |
| dataReadTimeout | Maximum number of seconds the client, the Message Sender, waits to receive a response from the Message Receiver before timing out. |
| myPartyId | Message Sender's party ID. |
| routeMap | Full path name of the **Routemap** file. |
| cpaLocation | Full path name of the directory where Collaboration Protocol Agreements are stored. |
| connectionTimeout | Number of milliseconds the Message Sender waits before timing out the SSL connection attempt. |
| trustedCerts | Full path name of the certificate store, which contains the trusted certificates of authority. |
| trustedCertsPasswd | Name of the password, which enables access to the password store. The password store contains the passwords that enable access to the trusted certificates file. |
| keyStore | Full path name of the Message Sender's keystore. |

| Field Name | Description |
|---|---|
| delayedRetry | When true, retries to send failed records. |
| delayedRetryInterval | When set to true, indicates the amount of time between sending failed records. |
| maxDelayedRetries | Maximum number of times to send a failed record. |
| useWebProxy | When true, the HTTP requests from the PHINMS client are sent through a proxy server. |
| proxyHost | Host name of the proxy server. |
| proxyPort | Proxy server port. |
| ProxyUser | Name of the proxy user. This field, which is used for authenticating to the proxy server, references an entry in the passwords file, which contains the proxy password. |
| proxyPasswd | This field, which is used for authenticating to the proxy server, references an entry in the passwords file, which contains the proxy password. |
| useLdapProxy | When true, LDAP requests, in the form of an ebxml message, are sent to an LDAP request handler. |
| ldapProxyRoute | Route used when proxy server sends the LDAP request. |
| ldapProxyService | ebXML service that proxies the LDAP request. |
| ldapProxyAction | EbXML action performed when proxying the LDAP request |
| keyStorePasswd | Name of the password which enables access to the password store. The password store contains the password that enables access to the keystore file. |
| logLevel | Logging level: **none**, **error**, **info**, **detail**, **messages**. |
| logDir | Directory where logging is done. |
| logArchive | If true, the log files are archived when they reach their maximum size |
| maxLogSize | Maximum log size in bytes. |
| processedDir | Full path name of the directory to which processed file descriptors are written. |
| jdbcDriver | Name of the JDBC Driver the Transport Queue database table uses. |
| databaseUrl | JDBC URL of the database that contains the Transport Queue database table. |
| messageTable | Name of the Transport Queue database table. |
| databaseUser | Name of the database user within the password store. |
| databasePasswd | Name of the database password within the password store. |
| encryptionTemplate | Full path name of the XML encryption template. |
| signatureTemplate | Full path name of the XML encryption template. |
| dbType | Database type. Values are: **oracle**, **sqlserver**, **foxitex**. Foxitex is the default. |
| responseToDb | When this field is set to **true**, the response file is written to the **responseContent** field in the |

| Field Name | Description |
|---|---|
| | Transport Queue instead of writing the content to disk. The default is set to **false**. |
| ldapKeyRetrieval | Specifies whether LDAP search should be used to retrieve the public key of the recipient for encryption. If set to true, LDAP is used. Otherwise, the **certificateURL** field in the Transport Queue is read for the URL of the recipient's certificate. The default is true. |
| ldapCache | If true, keys retrieved using an LDAP search are cached for efficiency. |
| ldapCacheTimeoutHours | Lifetime, in hours, of an LDAP cache entry. |
| ldapCachePath | Path of the LDAP cache, relative to the installation directory. |
| syncReply | When this field is set to true, a synchronous reply is requested from the Message Receiver. The default is true. |
| ackRequested | When this field is set to true, an acknowledgement is requested from the Message Receiver. The default is true. |
| signedAck | When this field is set to true a signed acknowledgement is requested from the Message Receiver. The default is false. |
| monitorTimerInterval | Amount of time between monitor refreshes. |
| service | If true, the client is configured as a service. |
| serviceKey | The key used to decrypt the seed, which obtains the password to the encrypted password file. The **serviceKey** is one of the inputs in the **substitute.bat** utility. The serviceKey is used only if service=true. |
| serviceSeed | The ciphertext obtained when the password to the password file is encrypted using the **substitute.bat** utility. |
| queueMap | Path, relative to the install directory, of the client-side queue map. This file defines the client-side worker queues. |
| serviceMap | Path, relative to the install directory, of the client-side service map. This file maps the arguments in the route-not-read poll responses to the worker queues. |
| **XML Segment or Sub-Segment** | **Description** |
| pollList | List of poll destinations. |
| pollList.destination | A **pollList** destination. |
| pollList.destination.route | Route name of the poll destination, as it appears in the **routemap**. |
| pollList.destination.service | Service used at the poll destination. |
| pollList.destination.action | Action used at the poll destination. |
| pollList.destination.recipient | Message Recipient for which the Message Sender polls the poll destination. |
| pollList.destination.pollInterval | Interval, in seconds, between polls. |

| Field Name | Description |
|---|---|
| databasePool | Pool of database connections, which are needed for client-side worker queues. |
| databasePool.database | A database pool block. |
| databasePool.database.databaseId | Database identifier - can be any text string with no spaces. |
| databasePool.database.dbType | Database type - sqlserver, oracle, or access. |
| databasePool.database.poolSize | Maximum number of simultaneous connections to the database. |
| databasePool.database.jdbcDriver | Name of the JDBC driver |
| databasePool.database.databaseUrl | URL of the database. |
| databasePool.database.databaseUser | Tag name of the database user name that is stored in the encrypted password file. |
| databasePool.database.databasePasswd | Tag name of the database password that is stored in the encrypted password file. |

## 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 and 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 URL | The URL or endpoint of the party. |
| transportSecurity | The **TransportSecurity** sub-tree contains the following custom attributes: authentication types Values are **none, basic, custom, sdn, clientCert, netegrity** |

### *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 |
|---|---|
| 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 **indexPage** (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. |
| sdnAuth | This block is read if **authenticationType** is set to **sdn**. The attributes of this block include the **sdnLoginPage** and the **sdnPassword**. The **sdnPassword** value in the CPA is actually the name of a password variable within the encrypted passwords file. |

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

### *Encrypted Passwords File*

The encrypted password file is used by the Message Sender and the Message Receiver to store sensitive passwords. The following is an example of the plain text version of the password file:

```
<?xml version="1.0"?>
<passwordFile>
      <cacertsPasswd1asfasdfkjlkjl</cacertsPasswd1>
      <sdnPassword1>mysdnpassword</sdnPassword1>
      <keyStorePasswd1>mykeystorepasswd</keyStorePasswd1>
      <dbUser1>scott</dbUser1>
      <dbPasswd1>tiger</dbPasswd1>
</passwordFile>
```

Entries in this file are referenced from other configuration files. For example, the CPA might have an entry as such:

```
<sdnPassword>sdnPassword1</sdnPassword>
```

The password XML file is encrypted using password based encryption, and the encrypted file is a resource for the Message Sender. A utility **cmds\pbe.bat** is provided with this library that can be used to encrypt and decrypt the passwords file. On startup of the Message Sender program, the user is prompted for the password that was used to encrypt this file. The decrypted file contents are stored in memory, not written to disk. The resulting in-memory password-map is used throughout the uptime of the Message Sender.

### *Example of the RouteMap File*

The RouteMap maps routes to the Collaboration Agreement Protocol files. The RouteMap uses the following format:

```
<RouteMap>
	<Route>
		<Name>OKLAHOMA</Name>
		<Cpa>CPA_cdc_oklahoma</Cpa>
	</Route>
	<Route>
		<Name>NEBRASKA</Name>
		<Cpa>CPA_cdc_nebraska</Cpa>
	</Route>
</RouteMap>
```

## To Set up the Client as a Service

By default, the client runs interactively, not as a service. To set up the client as a service, do the following:

1. In the client **sender.xml** file, set the **service** value to **true**. Specify the s**erviceKey** and **serviceSeed** values. Use the **substitute.bat** utility to derive these from your password file password.

   For example, the **sender.xml** entries may look like the following:

   ```
   '<service>true</service>
   <serviceKey>2o3i23</serviceKey>
   <serviceSeed>151209139182126100100162</serviceSeed>
   ```

2. If you are using the UNIX platform, add an **inittab** entry for **cmds\spawn.sh** or for **cmds\routenotread.sh**.

3. If you are using Windows NT/2000 platforms, download and install **JavaService**, a third party tool, from the following URL:

   **www.alexandriasc.com/software/JavaService**:

   *There are other similar tools, however, the following instructions are specific to the* **JavaService** *tool.*

4. Run the following command (with appropriate path changes) to install the spawner as a service:

   ```
   call (installdirectory)\cmds\setenv.bat

   javaservice -install "spawner" (javahome)\jre\bin\server\jvm.dll
   \
   -Djava.class.path=%CLASSPATH%
   -start gov.cdc.nedss.applications.spawner.Spawner
   -params (installdirectory)\config\sender.xml
   ```

5. Run the following command (with appropriate path changes) to install the RNR poller as a service:

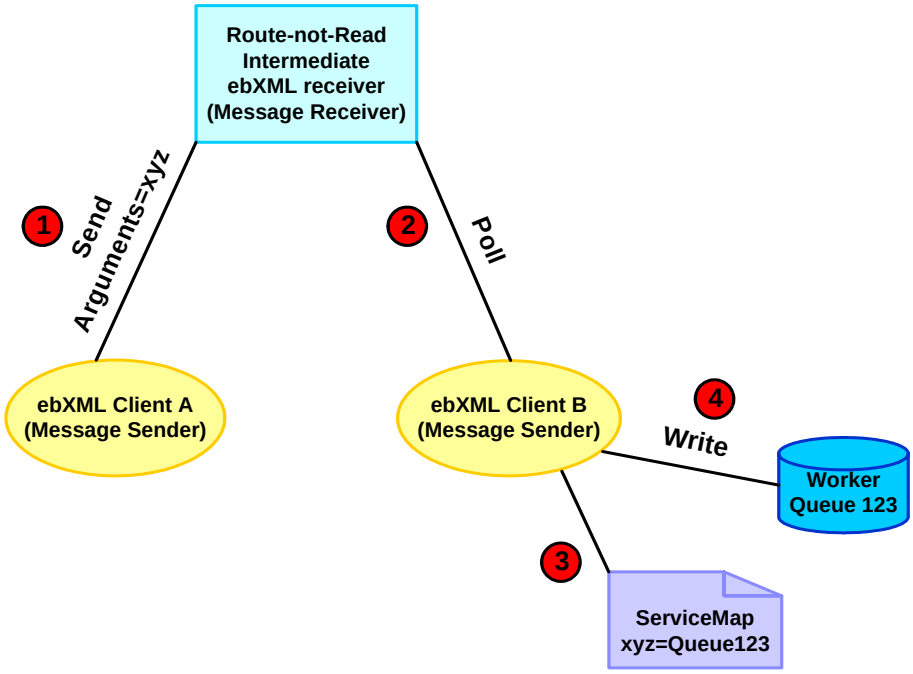
   ```
   call (installdirectory)\cmds\setenv.bat

   javaservice -install "poller" (javahome)\jre\bin\server\jvm.dll \
   -Djava.class.path=%CLASSPATH%
   -start gov.cdc.nedss.applications.rnrpoller.RNRPoller
   -params (installdirectory)\config\sender.xml
   ```

# *Configuring the PHINMS Client for Route-Not-Read Automatic Polling*

In version 2.0 of the Public Health Information Network Messaging System you can configure the ebXML Client, the Message Sender, to automatically poll route-not-read sites for incoming messages. You can specify the polling destinations and the intervals at which you want to poll these sites in the configuration files.

The following diagram illustrates an ebXML client sending a message to another ebXML client using automatic polling and a route-not-read Intermediate ebXML receiver:




The following steps correspond to the numbers in the diagram:

1. Client **A** sends a message, which includes arguments **xyz**, to the Route-not-Read Intermediate ebXML receiver.
2. Client **B** polls the Route-not-Read Intermediate ebXML receiver and then receives the response message, which includes arguments **xyz**.
3. Client **B** reads the **ServiceMap** to determine to which worker queue to send the incoming message and arguments, in this example, Worker Queue **123**.
4. Client **B** sends the message, including arguments **xyz** to Worker Queue **123**.
   If the arguments in a poll response do not map to any worker queue, which is defined in Client B, the incoming message is written to an error queue, whose schema is similar to the worker queue.

## *Worker/Error Queue Schema*

The following table describes the fields and data in the worker/error queue. See Appendix C and D for examples of the Oracle and SQL Server scripts:

| Field | Description | Data Type |
|---|---|---|
| recordId | Unique ID of the record in the table and the table's primary key. | Required<br>SQL Server: Integer Identity=Yes, Identity Increment=1<br>Oracle: Integer (20) not null |
| messageId | Application level message identifier. | Optional<br>SQL Server: varchar (255) null<br>Oracle: varchar2 (255) null |
| payloadName | File name of the payload, specified by the Message Sender. | Optional<br>SQL Server: varchar (255) null<br>Oracle: varchar2 (255) null |
| payloadBinaryContent | Image/BLOB field that is written to by the receiver servlet. | Optional<br>SQL Server: Image data type, null<br>Oracle: BLOB data type, null |
| payloadTextContent | Text/CLOB field that is populated if **textPayload=true** in the **servicemap** entry. | Optional<br>SQL Server: Text, null<br>Oracle: CLOB, null |
| localFileName | Used when the file is written to disk instead of a database field. (If **payloadToDisk=true** in the **servicemap** entry) | Optional<br>SQL Server: varchar (255,) null<br>Oracle: varchar2 (255), null |
| service | ebXML service name. | Required<br>SQL Server: varchar (255,) null<br>Oracle: varchar2 (255), null |
| action | ebXML action. | Required<br>SQL Server: varchar (255,) null<br>Oracle: varchar2 (255), null |
| arguments | Arguments specified by the Message Sender. | Optional<br>SQL Server: varchar (255,) null<br>Oracle: varchar2 (255), null |

| Field | Description | Data Type |
|---|---|---|
| fromPartyId | PartyId of the Message Sender. | Required<br>SQL Server: varchar (255,) null<br>Oracle: varchar2 (255), null |
| messageRecipient | Message recipient's identifier, specified by the Message Sender in the **transportQ_Out** queue. | Optional<br>SQL Server: varchar (255,) null<br>Oracle: varchar2 (255), null |
| errorCode | Error code. | Optional<br>SQL Server: varchar (255,) null<br>Oracle: varchar2 (255), null |
| errorMessage | Error message. | Optional<br>SQL Server: varchar (255,) null<br>Oracle: varchar2 (255), null |
| processingStatus | Status of the record. When a record is created, the initial value of the **processingStatus** field is **queued**. | Optional<br>SQL Server: varchar (255,) null<br>Oracle: varchar2 (255), null |
| applicationStatus | Status of the application. | Optional<br>SQL Server: varchar (255), null<br>Oracle: varchar2 (255), null |
| encryption | The value is **yes** if the payload is encrypted and **no** if not encrypted. | Optional<br>SQL Server: varchar (10) null<br>Oracle: varchar2 (10), null |
| receivedTime | Time when payload was received, in UTC format such as **`2001-10-01T16:01:01`**. | Optional<br>SQL Server: varchar (255), null<br>Oracle: varchar2 (255), null |
| lastUpdateTime | Time when record was last updated, in UTC format such as **`2001-10-01T16:01:01`**. | Optional<br>SQL Server: varchar (255), null<br>Oracle: varchar2 (255), null |
| processId | Identifies the process that is processing or most recently processed the record. | Optional<br>SQL Server: varchar (255), null<br>Oracle: varchar2 (255), null |

### *Additions to the sender.xml Configuration File*

The following fields in the **sender.xml** configuration file on the Message Sender, the client, support route-not-read automatic polling:

- **routeNotReadPoll** - If using automatic polling, set to true.
- **queueMap** - Path name of the client-side queue map, relative to the install directory.
- **serviceMap** - Path name of client-side service map, relative to the install directory.

### *Polling Destinations*

The following example is a segment of the **sender.xml** file, which defines the polling destinations:

```
<Sender>
….
<pollList>
 <destination>
  <route>CDC</route>
  <service>Router</service>
  <action>autopoll</action>
  <recipient>nedoh</recipient>
  <pollInterval>30</pollInterval>
 </destination>
</pollList>
…
</Sender>
```

### *Destination Nodes*

The **pollList** element in the **sender.xml** file segment contains a list of polling **destination nodes**, which can vary in length. Each destination node contains the following tags:

| Tag | Description |
|---|---|
| route | Name of the route to poll. |
| service | Name of the service to use in the poll request message. |
| action | Action to use in the poll request message. |
| recipient | Identifies the message recipient, which is specified in the poll request message. |
| pollInterval | Number of seconds between poll requests. |

## Database Connection

The following segment of the **sender.xml** file defines the database connection, which is required by the worker and error queues:

```
<Sender>
…..
<DatabasePool>
  <database>
    <databaseId>sqlserver1</databaseId>
    <dbType>sqlserver</dbType>
    <poolSize>1</poolSize>
  <jdbcDriver>com.microsoft.jdbc.sqlserver.SQLServerDriver</jdbcDriver>
<databaseUrl>jdbc:microsoft:sqlserver://localhost:1433;DatabaseName=Phmsg</datab
aseUrl>
    <databaseUser>sqlServerDbUserLocal</databaseUser>
    <databasePasswd>sqlServerDbPasswdLocal</databasePasswd>
  </database>
  <database>
    <databaseId>sqlserver2</databaseId>
    <dbType>sqlserver</dbType>
    <poolSize>1</poolSize>
    <jdbcDriver>com.microsoft.jdbc.sqlserver.SQLServerDriver</jdbcDriver>
<databaseUrl>jdbc:microsoft:sqlserver://nedsssql3.cdc.gov:1433;DatabaseName=Phms
g</databaseUrl>
    <databaseUser>sqlServerDbUserRemote</databaseUser>
    <databasePasswd>sqlServerDbPasswdRemote</databasePasswd>
  </database>
  <database>
    <databaseId>oracleserver1</databaseId>
    <dbType>oracle</dbType>
    <poolSize>1</poolSize>
    <jdbcDriver>oracle.jdbc.driver.OracleDriver</jdbcDriver>
    <databaseUrl>jdbc:oracle:thin:@nedss-report:1521:ebxml</databaseUrl>
    <databaseUser>oracleServerDbUserRemote</databaseUser>
    <databasePasswd>oracleServerDbPasswdRemote</databasePasswd>
  </database>
</DatabasePool>
…
</Sender>
```

## Database Pool Attributes

The following table describes the attributes for each of the database entries in the database pool:

| Attribute | Description |
|---|---|
| databaseId | Unique identifier for the database - defined by the user. |
| dbType | Specifies the database type. Values: **oracle** , **sqlserver**. |
| poolSize | Defines the number of connections that a specific database needs to have open at a given time. For high volume applications, a larger connection pool may be needed, such as **poolSize=10**. For low volume applications, a small connection pool may be sufficient, such as **poolSize=2**. |
| jdbcDriver | JDBC driver name for the database. |
| databaseUrl | URL of the database. |
| databaseUser | Index to the tag within the encrypted passwords file containing the user ID to the database. |
| databasePasswd | Index to the tag within the encrypted passwords file, which contains the password to the database. |

### *Client-Side QueueMap*

The **QueueMap** is a definition file that resides on the ebXML client configuration folder. It maps worker queues to database/table combinations. Its structure is identical to the queue map on the ebXML receiver.

Example:

```
<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>
```

The **Queuemap** contains the definition of three queues IDs:

- QID123
- QID456
- QID789

Each **QueueId** maps to a database and a table name. The **tablename** corresponds to a table that conforms to the worker queue schema. The databaseIDs are defined within the **sender.xml** file.

See **Appendix B** for the complete **sender.xml** file.

### *Client-Side Servicemap*

The **ClientServiceMap** is a configuration file that resides in the configuration file on the ebXML sender. It contains two types of entries, **service** and **errorqueue**. The following table lists the attributes for the service entry:

| Attribute | Description |
|---|---|
| Arguments | This field will be matched against the arguments that are returned as part of a route-not-read response. If they match, then the response is written to the queue indicated in the QueueId of this Service entry. |
| QueueId | Unique identifier for each worker queue |
| payloadToDisk | If this is true, then payload response is written to the *incomingDir* folder as defined in *sender.xml*, and the name of the file is entered into the *localPayload* field in the worker queue. If this is false, then the response payload is written to the database. |
| textPayload | If this is true, then the payload is written to the text field in the worker queue. |

**Example:**

```
<ClientServiceMap>
<Service>
        <arguments>asdf</arguments>
        <queueId>QID123</queueId>
        <payloadToDisk>false</payloadToDisk>
        <textPayload>true</textPayload>
</Service>
<ErrorQueue>
        <QueueId>QID000</QueueId>
</ErrorQueue>
</ClientServiceMap>
```

## Password Based Encryption

### Spawn Utility

You can use the command line utility **spawn.bat** to spawn the Message Sender's polling function. It implements the Message Sender as a relational database table if the **dbPoll** field in the **sender.xml** file is set to **true** or spawns the polling function as a file system directory if the **filePoll** field in the **sender.xml** file is set to **true**.

### To Spawn the Message Sender using spawn.bat

1. On the command line type **spawn.bat** and then press **Enter**.
2. Type the password and then press **Enter**.

   After displaying the **"waiting for files..."** message, the **Message Queue Monitor** opens.

### To Decrypt a File

1. On the command line type **pbe.bat** and then press **Enter**.
2. Type the letter **d** and then press **Enter**.
3. Type the name of the file you want to decrypt and then press **Enter**.
4. Type the location you want to put the decrypted file and then press **Enter**.
5. Type the password and then press **Enter**.

### To Encrypt a File

1. On the command line type **pbe.bat** and then press **Enter**.
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**.
4. Type the location you want to put the file and then press **Enter**.
5. Type the password and then press **Enter**.

# Security

It is extremely important that the confidentiality and integrity of the messages are preserved. You can help safeguard the messages by properly managing passwords, authentication, and by closely following the recommended security practices.

## Recommended Security Practices

To maintain the security and integrity of the messages and to safeguard The Message System use the following practices:

| Subject | Recommended Practice |
|---|---|
| Passwords | Do not store passwords as plain text files. When you use a plain text password file to generate an encrypted password file, promptly delete the plain text file. |
| KeyStores and Trusted Certificates | When creating passwords to the keyStores and trusted CA certificate files, choose a password with at least eight characters, including a mixture of alpha and numeric and upper and lower case. |
| Digital Signatures | To prevent the message from being rejected because of its origin, use digital signatures. Digital signatures require client certificates and a Public Key Infrastructure, such as LDAP, to manage the client certificates. |
| File System Permissions | Use file system permissions to prevent unauthorized users from accessing the configuration files and to prevent unauthorized users from accessing the incoming and outgoing payload directories. |

## *Managing Passwords*

As the administrator, you need to protect the secrecy of several configuration variables which include passwords, keystore passwords, login passwords and so on. All of the passwords are kept in an encrypted password map so that you do not have to remember or enter several passwords.

During startup, the system prompts the user for the password to the encrypted password map. During the client's runtime, the password map is stored in the client's memory. Other configuration files make references to this password map. You can use the utility **pbe.bat** to encrypt plain text password maps and obtain their cipher text.

For example, the password map in plain text looks similar to the following:

```
<passwordFile>
<loginUser>raja</loginUser>
<loginPasswd>Kailar</loginPasswd>
</passwordFile>
```

The sender configuration file, **sender.xml**, references the encrypted password map and also references **loginUser** and **LoginPasswd**. The password used to encrypt the password map is supplied at startup time.

# Appendix A

## Example of the Collaboration Protocol Agreement:

```
<?xml version="1.0" ?>
<tp:CollaborationProtocolAgreement>
<tp:PartyInfo>
  <tp:PartyId tp:type="DUNS">NEBRASKADUNSNUMBER</tp:PartyId>
  <tp:PartyRef xlink:href="http://www.nebraska.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="icdc-xdv-sdn7.cdc.gov/ebxml/receivefile"
tp:type="allPurpose" />
        <tp:TransportSecurity>
            <tp:Protocol tp:version="3.0">SSL</tp:Protocol>
            <tp:CertificateRef></tp:CertificateRef>
            <tp:authenticationType>netegrity</tp:authenticationType>
            <!-- basic, custom, sdn, clientcert, netegrity -->

            <tp:sdnAuth>
      <tp:sdnPassword>sdnPassword1</tp:sdnPassword>
              <tp:sdnLoginPage>/sdn_ext/sdn4.dll?ValidateInitialLogin</tp:sdnLog
       inPage>
            </tp:sdnAuth>

             <tp:netegrityAuth>
      <tp:sdnPassword>sdnPassword1</tp:sdnPassword>
              <tp:sdnLoginPage>/certphrase/login.fcc</tp:sdnLoginPage>
            </tp:netegrityAuth>

            <tp:clientCertAuth>
                <tp:config>
       d:\\projects\\clebint\\ebxmlvob\\config\\sdnlogin.props</tp:config>
            </tp:clientCertAuth>

           <tp:customAuth>
                <tp:customLoginPage>/login.asp</tp:customLoginPage>
                <tp:publicParams>logine=check</tp:publicParams>
         <tp:secretParams>username=user2&amp;userpwd=passwd2</tp:secretParams>
           </tp:customAuth>

          <tp:basicAuth>
                <tp:indexPage>/session.asp</tp:indexPage>
                <tp:basicAuthUser>user1</tp:basicAuthUser>
                <tp:basicAuthPasswd>passwd1</tp:basicAuthPasswd>
           </tp:basicAuth>

</tp:TransportSecurity>
</tp:Transport>
</tp:PartyInfo>
```

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

# Appendix B

## Example of the Message Sender Configuration File, Sender.xml:

```
<Sender>
<installDir>d:\\projects\\evalebxmlint\\ebxmlvob\\</installDir>
<filePoll>true</filePoll>
<dbPoll>false</dbPoll>
<routeNotReadPoll>true</routeNotReadPoll>
<test>false</test>
<passwordFile>config\\passwds</passwordFile>
<maxAttempts>2</maxAttempts>
<filePollMode>loop</filePollMode>
<!-- possible values for filePollMode are { once, loop } -->
<filePollInterval>30</filePollInterval>
<fileDescriptorDir>filedescriptors\\</fileDescriptorDir>
<!-- fileDescriptor XML, nameValue -->
<fileDescriptorFormat>nameValue</fileDescriptorFormat>
<!-- <filePollDir>outgoing\\</filePollDir> -->
<!-- sqlserver, oracle, fositex -->
<dbType>fositex</dbType>
<!-- possible values for dbPollMode are { once, loop } -->
<dbPollMode>loop</dbPollMode>
<!-- possible values for dbPollMode are { once, loop } -->
<dbPollDir>outgoing\\</dbPollDir>
<responseToDb>true</responseToDb>
<incomingDir>incoming\\</incomingDir>
<dbPollInterval>10</dbPollInterval>
<dataReadTimeOut>30</dataReadTimeOut>
<myPartyId>LABCORPDUNSNUMBER</myPartyId>
<routeMap>config\\routemap.xml</routeMap>
<cpaLocation>config\\CPA\\</cpaLocation>
<connectionTimeOut>20</connectionTimeOut>
<trustedCerts>config\\cacerts</trustedCerts>
<trustedCertsPasswd>cacertsPasswd1</trustedCertsPasswd>
<keyStore>config\\dddtest.pfx</keyStore>
<keyStorePasswd>keyStorePasswd1</keyStorePasswd>
<logLevel>messages</logLevel>
<maxLogSize>10000000</maxLogSize>
<logArchive>true</logArchive>
<logDir>logs\\</logDir>
<processedDir>processed\\</processedDir>
<ldapKeyRetrieval>false</ldapKeyRetrieval>
<ldapCache>true</ldapCache>
<ldapCacheTimeoutHours>1</ldapCacheTimeoutHours>
<ldapCachePath>config\\ldapCache</ldapCachePath>

<queueMap>config\\clientqueuemap.xml</queueMap>
<serviceMap>config\\clientservicemap.xml</serviceMap>

<!-- Monitor Parameters -->
<monitorTimerInterval>10000</monitorTimerInterval>

<!-- To setup this command as a service -->
<service>true</service>
<serviceKey>2o3i23</serviceKey>
<serviceSeed>15120913918212610010016 2</serviceSeed>

<!-- ebXML Transport parameters -->
<syncReply>true</syncReply>
<ackRequested>true</ackRequested>
<signedAck>false</signedAck>

<!-- MS Access database configuration
```

```
<jdbcDriver>sun.jdbc.odbc.JdbcOdbcDriver</jdbcDriver>
<databaseUrlPrefix>jdbc:odbc:</databaseUrlPrefix>
<databaseUrlSuffix>messagequeue</databaseUrlSuffix>
<messageTable>messagequeue</messageTable>
<databaseUser>accessdbUser</databaseUser>
<databasePasswd>accessdbPasswd</databasePasswd>
→
<!—SQL Server Database Configuration
<jdbcDriver>com.microsoft.jdbc.sqlserver.SQLServerDriver</jdbcDriver>
<databaseUrlPrefix>jdbc:microsoft:sqlserver:</databaseUrlPrefix>
<databaseUrlSuffix>//nedss-sql3:1433;DatabaseName=Phmsg</databaseUrlSuffix>
<messageTable>TransportQ_out</messageTable>
<databaseUser>sqlServerDbUser</databaseUser>
<databasePasswd>sqlServerDbPasswd</databasePasswd>
-->

<!-- CSV File - Fositex Driver -->
<jdbcDriver>com.inet.csv.CsvDriver</jdbcDriver>
<databaseUrlPrefix>jdbc:csv:</databaseUrlPrefix>
<databaseUrlSuffix>data\\phmsg</databaseUrlSuffix>
<messageTable>messagequeue</messageTable>

<!—ORACLE Database Configuration
<jdbcDriver>oracle.jdbc.driver.OracleDriver</jdbcDriver>
<databaseUrlPrefix>jdbc:oracle:thin:</databaseUrlPrefix>
<databaseUrlSuffix>@nedss-report:1521:ebxml</databaseUrlSuffix>
<messageTable>TransportQ_out</messageTable>
<databaseUser>oracleServerDbUser</databaseUser>
<databasePasswd>oracleServerDbPasswd</databasePasswd>
-->

<!-- <tempFileFolder>data\\</tempFileFolder> -->
<encryptionTemplate>config\\encryptiontemplate.xml</encryptionTemplate>
<signatureTemplate>config\\payloadSignature.xml</signatureTemplate>

<pollList>
 <destination>
  <route>CDC</route>
  <service>Router</service>
  <action>autopoll</action>
  <recipient>nedoh</recipient>
  <pollInterval>10</pollInterval>
 </destination>
</pollList>

<databasePool>
  <database>
    <databaseId>sqlserver1</databaseId>
    <dbType>sqlserver</dbType>
    <poolSize>1</poolSize>
    <jdbcDriver>com.microsoft.jdbc.sqlserver.SQLServerDriver</jdbcDriver>

<databaseUrl>jdbc:microsoft:sqlserver://localhost:1433;DatabaseName=Phmsg</datab
a
seUrl>
    <databaseUser>sqlServerDbUserLocal</databaseUser>
    <databasePasswd>sqlServerDbPasswdLocal</databasePasswd>
  </database>
  <database>
   <databaseId>access1</databaseId>
   <dbType>access</dbType>
   <poolSize>1</poolSize>
   <jdbcDriver>sun.jdbc.odbc.JdbcOdbcDriver</jdbcDriver>
   <databaseUrl>jdbc:odbc:workerqueue</databaseUrl>
   <databaseUser>accessdbUser</databaseUser>
   <databasePasswd>accessdbPasswd</databasePasswd>
  </database>
  <database>
    <databaseId>sqlserver2</databaseId>
    <dbType>sqlserver</dbType>
    <poolSize>1</poolSize>
```

```
    <jdbcDriver>com.microsoft.jdbc.sqlserver.SQLServerDriver</jdbcDriver>
    <databaseUrl>jdbc:microsoft:sqlserver://nedss-
sql3.cdc.gov:1433;DatabaseName=Phms
g</databaseUrl>
    <databaseUser>sqlServerDbUserRemote</databaseUser>
    <databasePasswd>sqlServerDbPasswdRemote</databasePasswd>
  </database>
  <database>
    <databaseId>oracleserver1</databaseId>
    <dbType>oracle</dbType>
    <poolSize>1</poolSize>
    <jdbcDriver>oracle.jdbc.driver.OracleDriver</jdbcDriver>
    <databaseUrl>jdbc:oracle:thin:@nedss-report:1521:ebxml</databaseUrl>
    <databaseUser>oracleServerDbUserRemote</databaseUser>
    <databasePasswd>oracleServerDbPasswdRemote</databasePasswd>
  </database>
</databasePool>

</Sender>
```

# Appendix C

## Example of Script (DDL) 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 9999999999999999999999999
 NOCYCLE
 NOORDER
 CACHE 20;
```

**Note** To correctly increment the worker queue records, the sequence name must be **<Tablename>_record_count**.

# Appendix D

## Example of 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
```