Mirth Connect and NHIN Direct REST Integration Notes

This document details the steps required to make use of NHIN Direct REST as a secure transport for Mirth Connect channels, progressing from the sending of data from a Mirth Connect channel running on a Mirth Connect server on one machine to the receiving of the data in a second Mirth Connect channel running on a second Mirth Connect server on a second machine.

Throughout the remainder of this document, the single term Mirth is a reference to the Mirth Connect product, and the single term Direct is a reference to the NHIN Direct REST Java implementation. The Java implementation of NHIN Direct REST is specifically addressed; no other implementations of Direct are herein addressed, and no implications regarding the suitability of any other Direct implementations are intended.

Hardware / Network Requirements

As a prerequisite for integrating Direct for use as a secure transport with Mirth, a minimum of two machines must be identified between which data is to be securely transported, on each of which Mirth has been installed. For purposes of this discussion, one of these machines will be identified as Mirth-Send and the other as Mirth-Receive. It is possible that the roles of the machines may occasionally be reversed, but the requirements for Mirth-Send are different than for Mirth-Receive, so if data flow will only be required in one direction, it is not necessary to perform all steps on both machines.

A machine must be identified for installation of the Direct instance used for sending data on behalf of Mirth-Send; this machine will be referred to as Direct-Send. Likewise, a machine must be identified for installation of the Direct instance used for receiving data on behalf of Mirth-Receive; this machine will be referred to as Direct-Receive. The machines used for the Direct installations must fit the following criteria:

1. Direct-Send must have network connections to Mirth-Send and Direct-Receive, with ports open on each to support the Direct connections.
2. Direct-Receive must write its data files to a file location accessible by Mirth-Receive.
3. Direct-Receive and Mirth-Receive must reside behind the same firewall (or provide their own local means of secure transport, which is outside the scope of this document).

Note that the transport of data between Mirth-Send and Direct-Send will be encrypted by the process described herein; it makes no use of Direct per se in so doing, but only of SSL. It is possible for Direct to be installed on Mirth-Send, so that Mirth-Send and Direct-Send are one and the same machine, but this is not required.

The communication between Direct-Receive and Mirth-Receive is unencrypted. This is because the Direct process results in the writing of files to a file directory on Direct-Receive, and Mirth-Receive must read the files from that directory. Optionally, some other process may be put in place for reading the data from the files and delivering the data to the Mirth channel running on Mirth-Receive, but such is not described herein. It is assumed in this document that Direct-Receive will write files to a shared location residing behind the same firewall as Direct-Receive and Mirth-Receive. It is possible for Direct-Receive and Mirth-Receive to be one and the same machine, and this satisfies the requirement that they have access to a shared data directory.

From the above, it can be seen that two machines are required at a minimum, one for sending and one for receiving, but the set up is not restricted to two machines. Indeed, if machine loads require it, multiple machines for any of the above-mentioned roles could be set up, but configurations required for such a scaling issue are not specifically addressed herein.

For the remainder of this document, it will be assumed that Direct-Send and Mirth-Send are one and the same machine; this machine will be referred to as Sender. Likewise, it will be assumed that Direct-Receive and Mirth-Receive are one and the same machine; this machine will be referred to as Receiver. But continue to bear in mind that Direct and Mirth can be installed on different machines, providing the criteria specified above are met.

The Mirth-Direct integration was tested on a Windows server with the following specifications:

- Windows Server 2008 Service Pack 2
- Intel Pentium 4 CPU 3.80GHz 3.79 GHz
- Memory (RAM): 3.00 GB
- System type: 32-bit Operating System

The Mirth-Direct integration was also tested on a Windows machine with the following specifications:

- Windows XP 2002 Service Pack 3
- Intel Core Duo CPU
- 2.93 GHz, 3.48 GB of RAM
- System type: 32-bit Operating System

Software Requirements

The following software is required for Direct REST. The versions listed were used in actual tests. Other versions of these software packages might also work.

- Java 1.6.0_18 (Java 1.6.0_24 was also tested)
- Java Cryptography Extension Unlimited Strength Jurisdiction Policy Files 6
- Maven 2.2.1
- NHIN Direct REST 0.0.1

The above tools must be installed according to instructions provided with them or delivered as part of the Public Health Information eXchange (PHIX) installation notes. Ensure that the JAVA_HOME environment variable is set for Java. At least one user must be defined on each Direct REST machine. The user defined on the Sender machine will be used to send Direct messages, and the user defined on the Receiver machine will be used to receive those messages. Each user must have a Direct address that contains the user name before the "@" symbol in the address. (Direct is more flexible than this, but the intent was to keep it simple for the HUB.)

Installing NHIN Direct REST 0.0.1 requires copying the software to the location from which it will be executed. The directory to which this software is copied will be referred to as $DIRECT_HOME.

Installing NHIN Direct REST 0.0.1 also requires the use of Maven to pull other software packages to your local Maven repository, but Maven will do this automatically when the Direct software is installed, so those packages are not herein listed, except to note that JavaMail 1.4.1 will be a part of this install.

On the machines running Mirth Connect, the following software is required. The versions listed were used in actual tests, but other versions might work.

- Java 1.6.0_18 (Java 1.6.0_24 was also tested)
- Mirth Connect 2.0.1 (Mirth Connect 2.1.0 was also tested)

Again, these tools must be installed according to instructions provided with the tools, and the JAVA_HOME environment variable must be set. The top level directory into which you install Mirth Connect will be referred to herein as $MIRTH_HOME.

JavaMail 1.4.3 is automatically installed under Mirth when Mirth is installed, in $MIRTH_HOME\lib.

Custom PHIX Software Installation

Some files to be installed reside in the PHIX project directory. The top level PHIX project directory will be referred to as $HUB_HOME.

A Java jar file resides at $HUB_HOME\DirectRESTIface\lib\hubdirect.jar. This file must be copied into the directory $MIRTH_HOME\custom-lib. Mirth must be restarted after this so as to pick up the classes in the jar file.

The directory $HUB_HOME\DirectRESTIface\HubDirectTransport is the Eclipse project directory for this jar file, and may be copied to an Eclipse workspace and installed as an Eclipse project. The .classpath file for the Eclipse project specifies the use of the JavaMail 1.4.1 library. This is because the .classpath file uses the JavaMail jar file in the Maven repository, which happens to be the version of JavaMail used by NHIN Direct REST 0.0.1. If your instance of Eclipse already has the M2_REPO variable set to point to your local Maven repository, then you don't need to change the .classpath file; the JavaMail version 1.4.1 jar file suffices for building the HubDirectTransport project. If you use the jar file in the local Maven repository, you will need to ensure that you have defined the M2_REPO variable in Eclipse to point to your local Maven repository. You may wish, however, to modify the class path for the Eclipse project to

use the JavaMail 1.4.3 jar file in Mirth instead, so as to build with the same version of JavaMail as Mirth uses.

If changes are made to the custom PHIX Java classes in the Eclipse project, the jar file can be exported from Eclipse and placed at $MIRTH_HOME\custom-lib. Each time this is done, Mirth must be restarted to pick up the change.

Creating Mirth Channels to Invoke the Custom PHIX Software

The custom hubdirect.jar file contains two classes of primary importance: HubDirectSender and HubDirectReceiver. As the names suggest, HubDirectSender is for use on the Sender machine, and HubDirectReceiver is for use on the Receiver machine. These classes support the secure transport of plain text messages over Direct. Only plain text messages are supported. The plain text messages are automatically packaged as RFC822-formatted (email) messages by the software to ensure compliancy with Direct standards.

On the Sender machine, Mirth is responsible for obtaining data in some fashion. This could be through the use of a FileReader, for example, to monitor a known directory and to read files as they are written to the directory. For PHIX purposes, it is assumed that the data obtained on the Sender machiner are plain text HL7 messages to be securely transported to the Receiver machine.

Once Mirth on the Sender machine acquires data, Mirth constructs a HubDirectSender object and calls its *sendDirectMessage* method to package the plain text message as an email message, connect via SSL to Direct on the Sender machine, and instruct Direct on the Sender machine to encrypt and send the email message to the Receiver machine.

On the Receiver machine, when a Direct message is received, Direct automatically decrypts the message and writes the entire unencrypted email message to disk. Mirth can monitor this directory with a FileReader. Mirth then can construct a HubDirectReceiver object, call its *parse* method to parse the message, and call the *getMessage* method to fetch the original plain text message, which for PHIX purposes will be the original HL7 message.

Sample JavaScript for Mirth on Sender

Below is presented sample JavaScript for use in Mirth to wrap a plain text message (presumably an HL7 message) and send it to a known Direct address. The following variables are assumed to be set: *host, port, fromAddress, toAddress, subject, truststore,* and *truststorePassword*.

host: (String) the domain name of the Direct Sender machine
port: (int) the port on *host* on which Direct is listening
fromAddress: (String) the Direct address from which the message is to be sent
fromPassword: (String) the password for the user specified in *fromAddress*
toAddress: (String) the Direct address to which to send the message
subject: (String) the subject for the Direct email message
truststore: (String) the full path on the Sender machine to the truststore file for Direct
truststorePassword: (String) the password for the truststore file

How these variables come to be set is implementation specific. Note that some of these variables can be null, in which case Java properties will be used, as discussed below.

Also note that despite Direct's flexibility to do otherwise, HubDirectSender makes the assumption that *fromAddress* contains the Direct user name. That is, if *fromAddress* is specified as jdoe@sample.com, then *jdoe* is taken as the Direct user name on the Sender machine, and *fromPassword* would be taken as the password for *jdoe*, who must have been configured in Direct to be authorized for the Direct address jdoe@sample.com.

Sample JavaScript code follows. Note that some long lines wrap in this document, but are required to be on one line in Mirth. Continuation lines below are indented with a tab (five spaces). Lines that are enclosed inside if/else constructs are indented two spaces, and are not continuation lines.

This code assumes that the above-mentioned variables, and an *hl7msg* variable, have been pre-defined. The *hl7msg* variable is assumed to be a String containing the HL7 message to be sent via Direct; this should be only the HL7 message, with no email headers.

```
var sndr = new
     Packages.gov.cdc.phlissa.hub.direct.HubDirectSender(host, port,
     fromAddress, fromPassword, trustStore, trustStorePassword);
sndr.sendDirectMessage(toAddress, subject, hl7msg, false);
if (sndr.hasError()) {
  logger.info("Error Message: " + sndr.getErrorMessage());
} else {
  logger.info("Message Location: " + sndr.getLocation());
}
```

Any of the variables passed as arguments to the HubDirectSender constructor may be null. Those that are null default to the value of a Java property, shown as follows:

```
host: gov.cdc.phlissa.hub.direct.host
port: gov.cdc.phlissa.hub.direct.port
fromAddress: gov.cdc.phlissa.hub.direct.fromAddress
fromPassword: gov.cdc.phlissa.hub.direct.fromPassword
truststore: javax.net.ssl.trustStore
truststorePassword: javax.net.ssl.trustStorePassword
```

None of the arguments to the *sendDirectMessage* method may be null.

Note that there is a no-argument constructor for HubDirectSender. If used, all its arguments are derived from Java properties.

The sample shows that after *sendDirectMessage* is called, it can be checked for errors, and the error message may be retrieved if an error occurred. If no error occurred, a location for the message may be retrieved. The location is the URL for the message, which may be of little importance in Mirth. Another method of HubDirectSender, *getResponse*, returns the response from the server request, but experience has shown that this method typically returns an empty string.

The important information is the return value of *hasError*, with a desired return value of *false*, indicating that no error condition occurred.

The last (boolean) argument to *sendDirectMessage* indicates whether exceptions are to be thrown. If *false* is passed, it is assumed that *hasError* will be called to determine if an error occurred. Passing *true* will cause *sendDirectMessage* to throw an exception if an error occurs. There are other versions of the *sendDirectMessage* method, which may be of use depending on the circumstances. The Javadocs for the Java classes contain detailed info.

Setting Java Properties for HubDirectSender in Mirth

If it is desired to set defaults for HubDirectSender arguments using Java properties, edit either mcserver.vmoptions or mcservice.vmoptions in the $MIRTH_HOME directory. Which file to edit depends on how Mirth is executed, whether as a server or a service; it works to edit both files and enter the same values in each.

Below are sample lines that might be added to the mentioned .vmoptions files.

```
-Djavax.net.ssl.trustStore=C:\nhin-d-rest\spring-maven-poc\etc\truststore
-Djavax.net.ssl.trustStorePassword=password
-Dgov.cdc.phlissa.hub.direct.fromAddress=hub@sample.cdc.gov
-Dgov.cdc.phlissa.hub.direct.fromPassword=hubpassword
-Dgov.cdc.phlissa.hub.direct.host=localhost
-Dgov.cdc.phlissa.hub.direct.port=8555
```

Remember that any value entered in the .vmoptions file as a Java property can be overridden in the HubDirectSender constructor by passing a non-null value for the associated argument.

Mirth must be restarted to pick up any changes to a .vmoptions file.

Sample JavaScript for Mirth on Receiver

On the Receiver machine, the HubDirectReceiver class may be used to extract the original plain text HL7 message from the Direct RFC822-formatted message. As with the sample code for the Sender, continuation lines below are indicated by a tab (five spaces).

The entire Direct RFC822-formatted messages are read by the Mirth Channel from the location on disk where Direct on the Receiver machine writes received messages. Each message is written to a file in the directory $DIRECT_HOME\spring-maven-poc\data\<toAddress>, where <toAddress> is the *toAddress* in the call to *sendDirectMessage* on the Sender machine—provided the address is recognized by the Receiver machine as a valid Direct address. The actual file name will consist of a unique id generated by Direct on the Receiver machine, appended with the suffix *.NEW.txt*. Thus, a FileReader Mirth Channel Source aimed at the specified directory and reading files with names of the pattern **.NEW.txt* will pick up all messages received by Direct on the Receiver machine. These RFC822-formatted messages must then be parsed to extract the plain text HL7 message from the body of the email for further processing. This is done through use of the HubDirectReceiver class and its *parse* method.

This code assumes that an *emailmsg* variable has been pre-defined. The *emailmsg* variable is assumed to be a String containing the RFC822-formatted email message received over Direct; this should be the entire email message, with headers. It is also assumed that the email message has a top-level content type of text/plain; if such is not the case, an error occurs in the *parse* method.

```
var prsr = new
     Packages.gov.cdc.phlissa.hub.direct.HubDirectReceiver();
prsr.parse(emailmsg, true);
var parsedMessage = prsr.getMessage();
channelMap.put("parsedMessage", parsedMessage);
```

HubDirectReceiver only has a no-argument constructor. The *parse* method is passed the RFC822-formatted message and a boolean which indicates whether an exception is to be thrown if an error occurs. In this example we pass *true* for the boolean, to indicate that we want an exception thrown. We then proceed as though no error would occur, knowing that if one does, the process here will be interrupted, and Mirth will log the exception. This is in contrast to using the *hasError* method, which is also supported by HubDirectReceiver. See the Javadocs for more details.

Once the *parse* method returns, call *getMessage* on the HubDirectReceiver object to extract the original HL7 plain text message. The parsed message may then be placed in a channel variable for use elsewhere in the channel.

Synchronization Issues

If any HubDirectSender or HubDirectReceiver object is reused, you must synchronize calls to their methods so that you do not corrupt results before you have a chance to fetch them. To help with this, synchronized methods have been included in each class, and these synchronized methods return objects that will not be corrupted by subsequent calls to the synchronized methods. The Javadocs for the classes explain this.

Using HubDirectSender to Test a Direct Install

The HubDirectSender class contains a main method that may be used to connect to a Direct host and send messages. The JavaMail library is required in the class path. Executing HubSenderDirect with no arguments presents the following help message:

```
Options:
  --host <host>
  --from <fromuser>@<fromdomain>
  --to <touser>@<todomain>
  --pass <password>
  --port <port>
  --subject <subject>
  --file <filename>
  --trust <trustStore>
  --trustpass <trustStorePassword>
```

```
Send Direct message from fromuser at fromdomain to touser at todomain.
The password is the fromuser's password. The port is the port on host
to which to connect to Direct. The host defaults to the fromdomain if
not specified. The subject is the subject line for the message. The
filename is the name of the file containing the message to be sent.
The trustStore is the trustStore with the certficate for SSL. The
trustStorePassword is the password for the trustStore. If the
trustStore or trustStorePassword are omitted, they will be read from
system properties javax.net.ssl.trustStore and
javax.net.ssl.trustStorePassword, respectively.
```

The trustStore specified for the --trust option should be the truststore containing the CA certificate for Direct. It is possible to invoke HubDirectSender in this fashion from a machine other than the one where Direct resides, but in this case it is necessary to copy the trustStore file used by Direct to the local machine from which HubDirectSender is being invoked. The --trust option will then point to the copy of the trustStore file on the local machine.

While the help message does not explicitly state it, the --from, --pass, --host, and --port options may be omitted, in which case they default to the values specified by Java properties, as described above.

If an error occurs with the attempt to send a message across Direct from the specified *from* and *to* addresses, an error message is printed. Otherwise, the location header received from the Direct host is printed, along with any non-empty response from the host. You can check in $DIRECT_HOME\spring-maven-poc\data\<toUser>@<toDomain> on the receiving machine for the received message.