# The MyDelivery Client and API

Frank Walker and Girish Lingappa

2010

**The MyDelivery Client and API**

**National Library of Medicine**
**8600 Rockville Pike, Bethesda, MD - 20894**

# The MyDelivery Client and API

**Summary:**

This document presents a high level view of the user interface, architecture, internal design and operation of the MyDelivery client software and Applications Programming Interface (API). It should be sufficient to allow an organization to customize the client or use the API to replace the client for custom applications.

The client user interface is similar to that of an email user interface. It permits a user to exchange with another user information that may be as small as a simple text message, or it could be gigabytes in size, and consisting of large as thousands of files. The MyDelivery client is comprised of six modules capable of running on Windows XP and later operating systems (Server 2003, Server 2008, Vista and Windows 7). The core server module (ComMgr.dll) is responsible for all Internet communications. The user interface executable (MyDelivery.exe) provides the database support for organizing the deliveries, presentation and user interaction. The Application Programming Interface (API) services are provided by the platform service (ComMgr+.exe). The platform service is an on demand server that wraps the ComMgr.dll and provides services to both the user interface client and the command line tool (MDServer.exe). The MDServer.exe command line tool translates the command line arguments into a suitable form and passes it to the platform service for processing. The API Tester.exe is intended to provide a working example that shows how to use the API to interface to the platform service from an external system, replacing the client for send-only transmissions. The platform service is available as an on demand local server and follows the component object model (COM) for easy integration with other languages. The sixth module, usage.dll, helps to throttle the client to prevent overutilization of system resources.

# Table of Contents

*The MyDelivery Client and API* ____ *1*

*1.0 Using the MyDelivery Client* ____ *7*

**1.1 Prepare to use the Client** ____ **7**

**1.2 Logging In** ____ **8**

**1.3 Add User to Address Book** ____ **8**

**1.4 Address Book: Editing an Address.** ____ **11**

**1.5 Address Book: Block / Unblock a user** ____ **12**

**1.6 Address Book: Remove a User** ____ **14**

**1.7 Address Book: Export the Address Book** ____ **15**

**1.8 Address Book: Import an Address Book** ____ **17**

**1.9 Create a New Delivery** ____ **19**

**1.10 Monitor Delivery Progress** ____ **21**

**1.11 Folder Management: Create a Folder** ____ **25**

**1.12 Folder Management: Rename a Folder** ____ **26**

**1.13 Folder Management: Delete a Folder** ____ **27**

**1.14 Delivery Management: Add Attachments** ____ **28**

**1.15 Delivery Management: Save Attachments** ____ **32**

**1.16 Delivery Management: Print a Text Message** ____ **34**

**1.17 Delivery Management: Move a Delivery between Folders** ____ **35**

**1.18 Delivery Management: Delete a Delivery** ____ **36**

**1.19 Delivery Management: Reply to a Delivery** ____ **37**

**1.20 Delivery Management: Forward a Delivery** ____ **38**

*2.0 Client Modules* ____ *39*

**2.1 MyDelivery Database** ____ **40**

**2.2 Client Source Code Compilation** ____ **41**

**2.3 Additional Modules** ____ **41**

*3.0 Module: MyDelivery.exe* ____ *42*

**3.1 Folders** ____ **42**

3.2 Deliveries ____ 45
3.3 Delivery Move ____ 47
3.4 MyDelivery Event Sink Setup ____ 48
3.5 MyDelivery Event Sink Implementation ____ 50
3.6 Software Updates ____ 51
*4.0 Module: ComMgr.dll ____ 52*
4.1 Initialization of ComMgr.dll ____ 53
4.2 JobManager ____ 54
4.3 CopyThread ____ 54
4.4 Compression Thread ____ 57
4.5 Delivery Layout ____ 58
4.6 Send Thread ____ 59
4.7 DownloadThread ____ 61
4.8 Decompression Thread ____ 63
4.9 ClientStatusPage Thread ____ 65
4.10 Manager Thread ____ 66
4.11 Additional Files used by the Client ____ 67
*5.0 SOAP Functions Handled by ComMgr.dll ____ 73*
5.1 GetVersion ____ 73
5.2 ClientStatusPage (CSP) ____ 73
5.3 Initialize ____ 73
5.4 ChangePassword ____ 74
5.5 UploadUserInformation ____ 74
5.6 CheckRecipient ____ 75
5.7 DeliveryStatus ____ 75
5.8 IsDeliveryValid ____ 75
5.9 DownloadAttachment ____ 76
5.10 DownloadDeliveryHeader1 ____ 76
5.11 UpdateStatus ____ 77
5.12 UploadAttachment ____ 78

5.13 UploadDeliveryHeader1 78
5.14 StartDeliveryUpload 79
5.15 TerminateDelivery 79
5.16 RequestToUpload 79
*6.0 Sample Procedures Handled by ComMgr.dll 81*
6.1 Login 81
6.2 Change Password 81
6.3 Addressbook Updates 81
6.4 Changes in Free Disk Space 82
6.5 Terminate Delivery 83
6.6 Error Handling and Management 83
*7.0 Module: Usage.dll 84*
*8.0 Module: ComMgr+.exe 85*
8.1 API Class 85
8.2 Event Management 85
8.3 Intialization 89
8.4 Shutdown 90
8.5 Event Routing 91
8.6 Event Routing to Client 92
8.7 Process to Completion 93
*9.0 Module: MDServer.exe – Using the API 95*
9.1 Command: GetVersion 96
9.2 Command: GetAddressBook 97
9.3 Command: Send 98
9.4 Command: GetStatus 100
9.5 An Example of a Possible API Application 101
*10.0 Module:APITest.exe (MyDelivery API Tester) 102*
10.1 Proxy Settings 103
10.2 Get the current version of the MyDelivery system 104
10.3 Downloading your AddressBook 105

**10.4 Sending a delivery ____________________________________________________________ 107**

**10.5 Getting the status of a delivery ___________________________________________________ 110**

# 1.0 Using the MyDelivery Client

The MyDelivery client is easy to install and use. These are its basic features:

- The client sends messages in a manner similar to email. In this document, a message will be referred to as a "delivery." The client user interface is easy to learn.
- File attachments may be almost any size. The only limitation is the free disk space for processing on the sending and receiving clients.
- Thousands of files may be attached to a delivery.
- Structures of folders may be sent, and folder structure is maintained during transmission.
- The client can communicate in compliance with Health Insurance Portability and Accountability Act (HIPAA) requirements. When the MyDelivery server is equipped with a Transport Layer Security (TLS) certificate, the communication is compliant with HIPAA requirements for data encryption and verification.
- No user data is stored on the server, where it can be lost or compromised. Data passes through server memory where it resides from a few hundred milliseconds to several seconds before passing onto the receiving client.
- MyDelivery communication passes through firewalls, since it is standard HTTP or HTTPS communication.
- User roaming is permitted. This means a user can install a client on more than one computer, and the address book will be downloaded and up to date on the computer where the user is logged in.
- Spam is minimized because users communicate only with people listed in their address book.
- MyDelivery is fast. Text-only deliveries can arrive in a few seconds.
- A sending client automatically receives indication that the recipient has received a delivery.
- MyDelivery is reliable. Its robust communication algorithms detect network outages due to intermittent communication (such as sometimes experienced with wireless networks). No information is lost if the communication is interrupted, because when the client subsequently re-establishes communication, the transmission or reception picks up where it was interrupted.

## 1.1 Prepare to use the Client

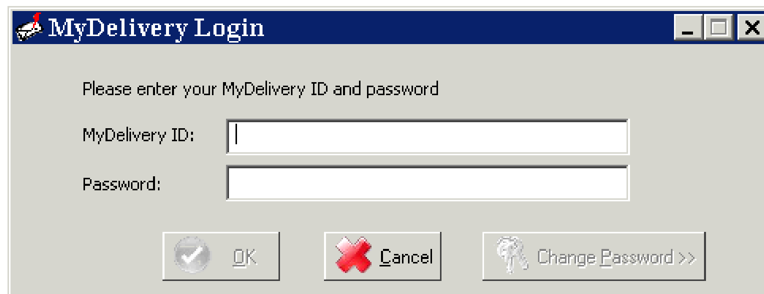
The MyDelivery client can use either unsecured HTTP communications or secured HTTPS communications. The selection depends on how the MyDelivery server is configured. The client source code must be compiled to handle a specific MyDelivery server or protocol (HTTP or HTTPS). For secure communications, the MyDelivery server can be equipped with an SSL certificate for standard HTTPS communication. For United States government systems requiring FIPS 140-compatible communications, the server can be equipped with the appropriate TLS certificate. For the latter, the client platform must be configured properly to handle TLS. These are the steps for four popular browsers:
**Internet Explorer**: Go the Tools/Internet Options menu. On the Advanced tab, check "Use TLS 1.0".
**Mozilla Firefox**: Go to the Tools/Options menu. Select the Advanced tab, then select the Encryption tab. Check "Use TLS 1.0".
**Opera**: Go to the Tools/Preferences menu. On the Advanced tab, select Security/Security Protocols. Check "Enable TLS1".

**Google Chrome**: No adjustments necessary.

Once the browser is configured, the user should register to use MyDelivery through the MyDelivery website. Then the software may be downloaded and installed. The steps for registration and installation of client software will not be given here, since they will be customized by whoever implements a MyDelivery system.

## 1.2 Logging In

To start the client, run mydelivery.exe. This will produce the login dialog box. Enter your MyDeliveryID and Password that you created during registration, then click Ok.




After you click OK, you will be logged into the system. During login, you may be asked to enter your username and password for a proxy server. This will occur only if your MyDelivery client detects a proxy server at your location requesting appropriate identification. You should enter your username and password appropriate for that proxy server. This information is sent only to the proxy server, and does not go out onto the Internet.

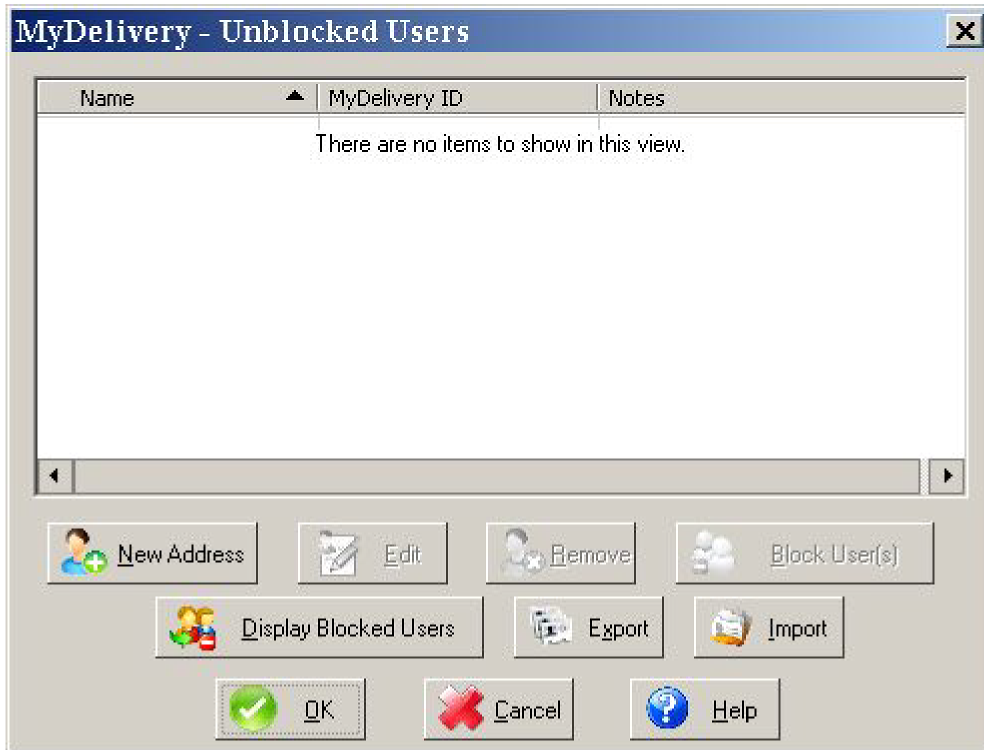
## 1.3 Add User to Address Book

In order for two users of MyDelivery to send messages to each other, each person must add the other person to their address book. So, if users John and Mary want to communicate through MyDelivery, John must add Mary to his address book, and Mary must add John to her address book. Once they have taken this important step, then the system permits them to communicate. This step is known as "User Trust." The MyDelivery system as currently designed does not authenticate the actual identity of any user. However, the system software can be modified to include user authentication. The act of each entering the other's MyDelivery ID into their address book signifies that the two users trust each other.

If you do not know anybody else who has MyDelivery, but you still want to see it in action, try sending a test message to yourself. To do this, first add your own MyDelivery ID to your address book.

What happens if John adds Mary to his address book, but Mary does not add John to her address book? Then, when John sends a message to Mary, MyDelivery will immediately notify him that he is not in Mary's address book. At the same time, Mary will receive a dummy message called "Pending Sender", indicating that John had attempted to send her a message, but was blocked. At this point, Mary has the option of adding John to her address book. She can also choose to block all incoming messages from John by adding him to her address book, but blocking him.

The picture below shows the address book containing no users.   The Address Book allows you to list the individuals with whom you wish to communicate through MyDelivery.  Your Address Book is linked to your MyDelivery ID, so if you should log into MyDelivery on more than one computer, your Address Book will automatically be sent to the most recent computer from which you log in.  MyDelivery sessions for the same MyDelivery ID on previous computers will be terminated as soon as you use the ID on a new computer.  This ensures that you can update your Address Book from only one computer at a time.  The dialog box below allows you to add new addresses (contacts), edit the address, remove an address, block an address, and display blocked users.  The first view presented by the Address Book is the list of "Unblocked Users," which is the list of addresses with which you permit communication.



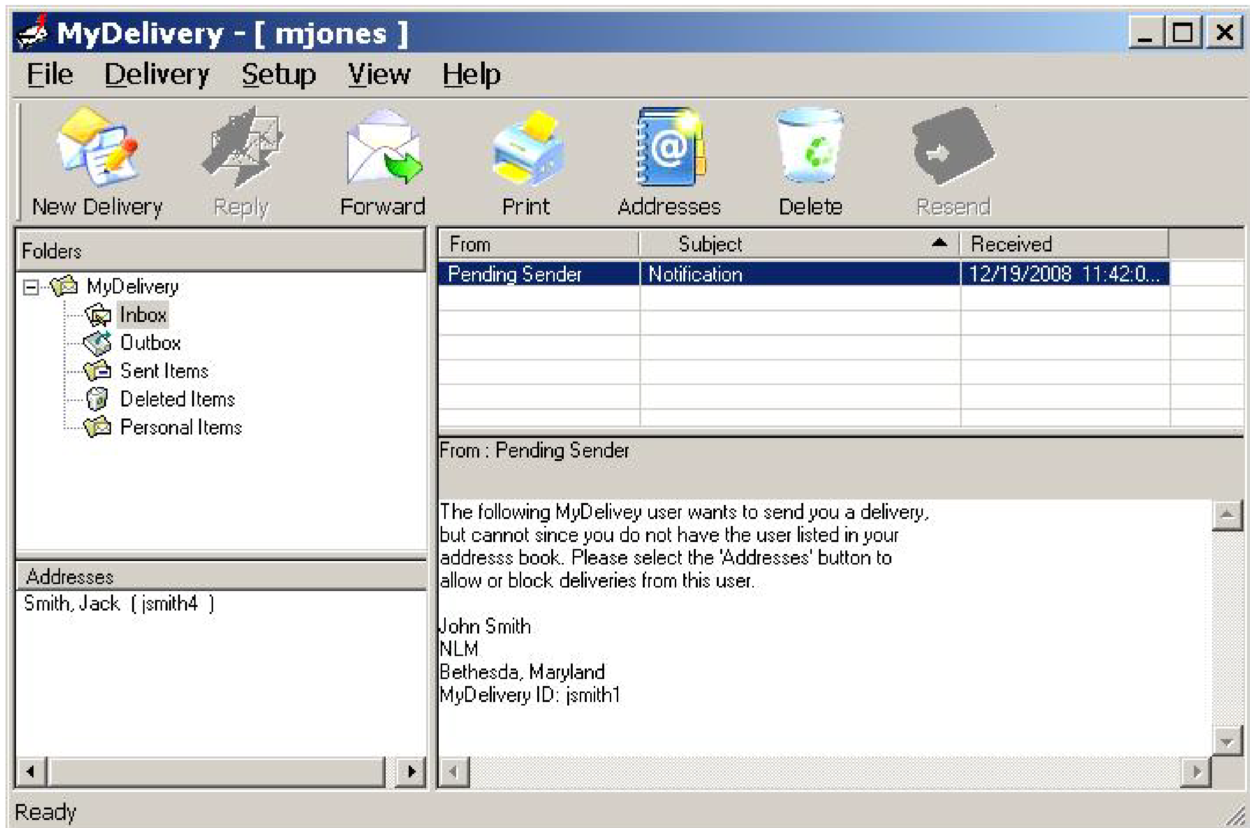
The New Address dialog box allows you to create a new contact with whom you may communicate or block through MyDelivery. You should enter the first and last names of the new contact. The MyDelivery ID must be entered correctly; it is not case-sensitive. All other fields may contain typing errors. While the First Name, Last Name and MyDelivery ID fields must all contain entries, the Notes field is optional. You may enter anything in the Notes field, such as mailing address, phone number, and email address.

## 1.4 Address Book: Editing an Address.

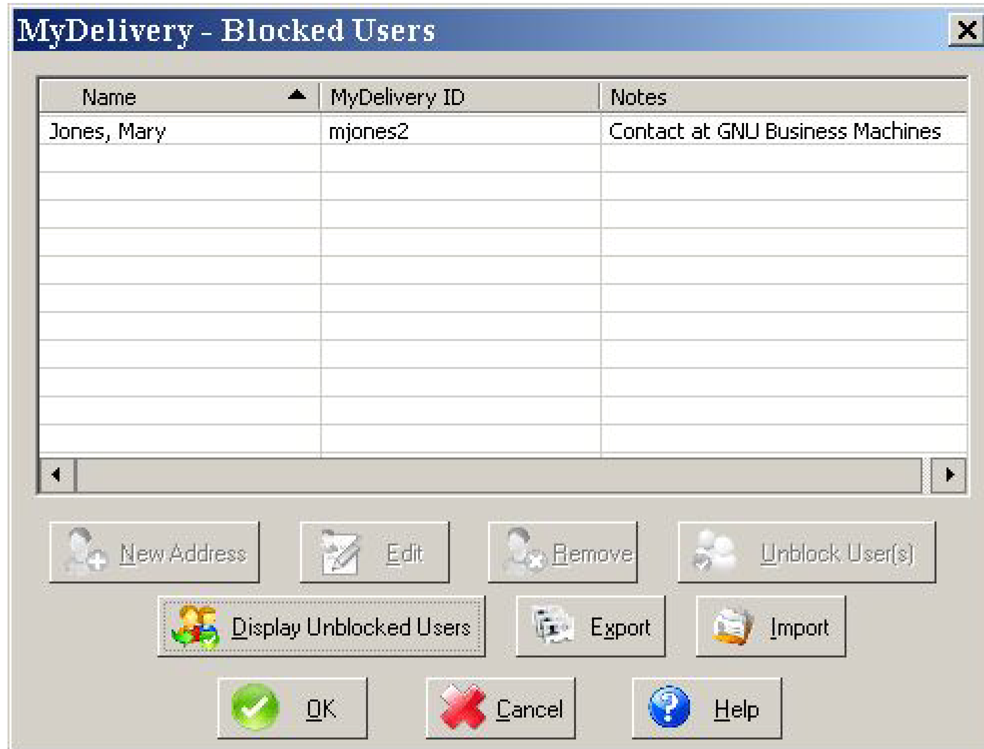
It is easy to change any address in your Address Book. You may change any field, but the contents of the MyDelivery ID field must be correct. The entry is not case-sensitive.

### 1.5 Address Book: Block / Unblock a user

*Blocking* a user prevents spam from somebody who may have obtained your MyDelivery ID without your permission. Each time someone who is *not* listed in your address book attempts to send you a delivery, you will not receive the actual delivery, but instead you will receive a message in your Inbox from "Pending Sender". The text message indicates who tried to contact you, and it lists their MyDelivery ID. You can prevent any future "Pending Sender" messages from this person if you add them to your address book, then block them.

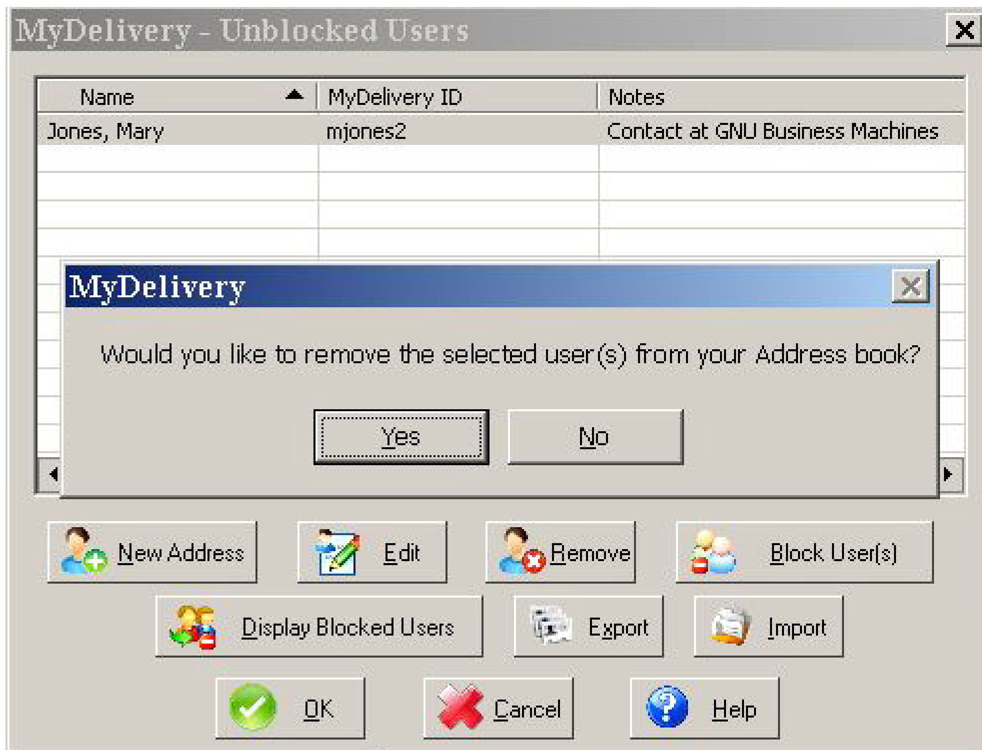


You may prevent any contact from sending you a delivery by blocking that user in your Address Book. Blocked users are automatically removed from the "Unblocked Users" list and sent to the "Blocked Users" list. It is easy to switch back and forth from the two lists by clicking the "Display Unblocked Users" and "Display Blocked Users" buttons, which appear in the respective dialog boxes.



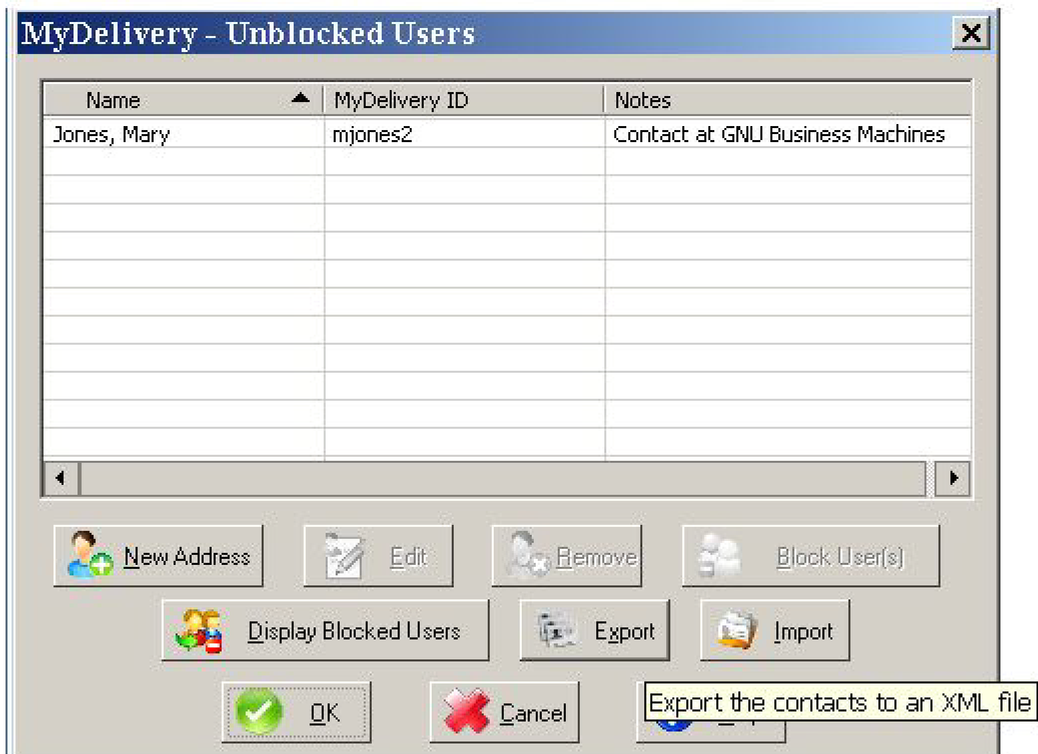
## 1.6 Address Book: Remove a User

If you click the Remove button in your Address Book you can delete the selected user. MyDelivery will prompt you with a dialog box for you to confirm the deletion.

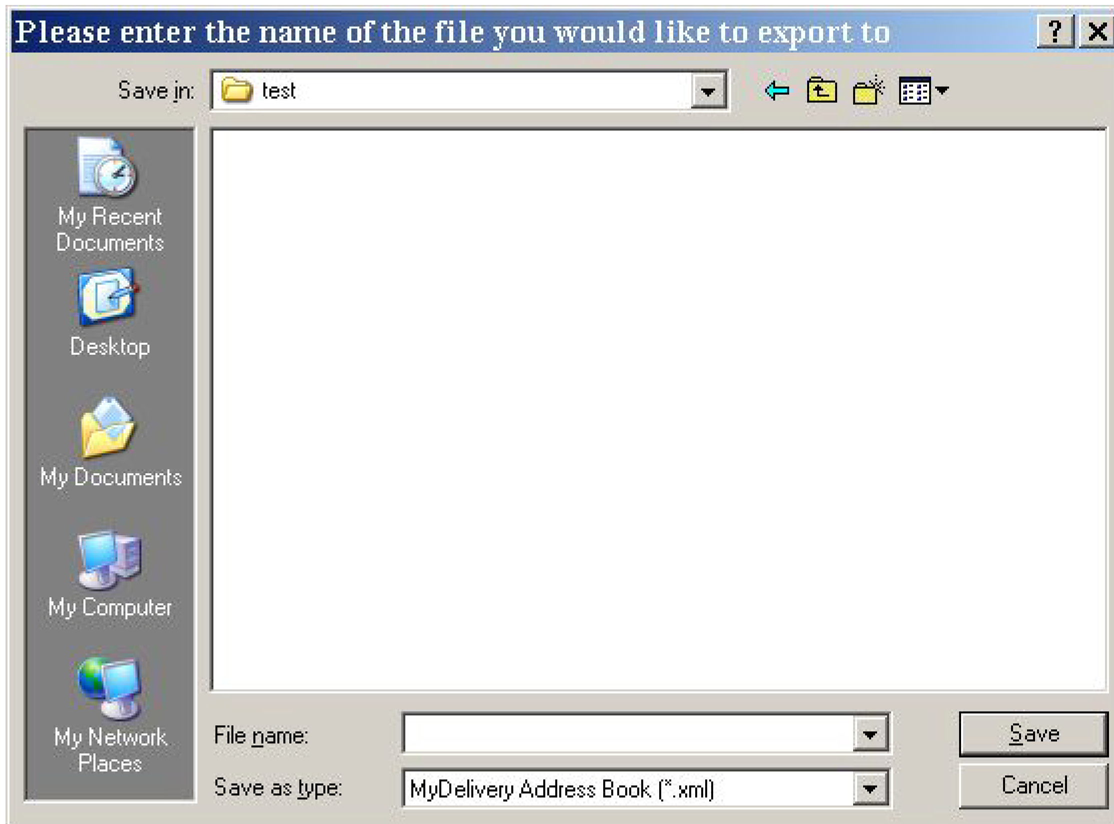


### 1.7 Address Book: Export the Address Book

You can export the list of contacts in your Address Book to an XML file for use by other MyDelivery users. This is useful in environments such as library document delivery services, where it is desirable for librarians to share a common set of patron addresses. After one librarian updates their Address Book, they can export it, so that others can import it. This way, everyone's Address Book is the same.

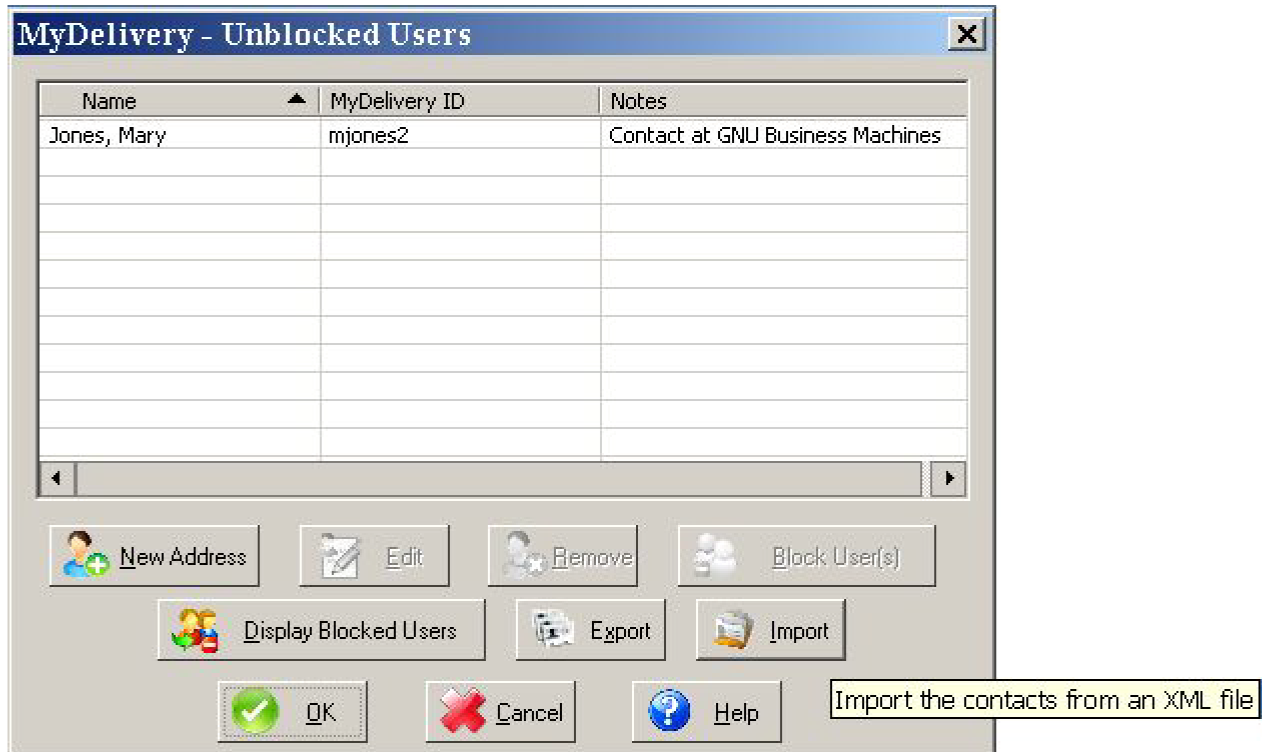



After you select the Export button, a dialog box will prompt you to give the name and location on disk of the XML file to be stored.


Please enter the name of the file you would like to export to
Save in:
test
My Recent Documents
Desktop
My Documents
My Computer
My Network Places
File name:
Save as type:
MyDelivery Address Book (*.xml)
Save
Cancel

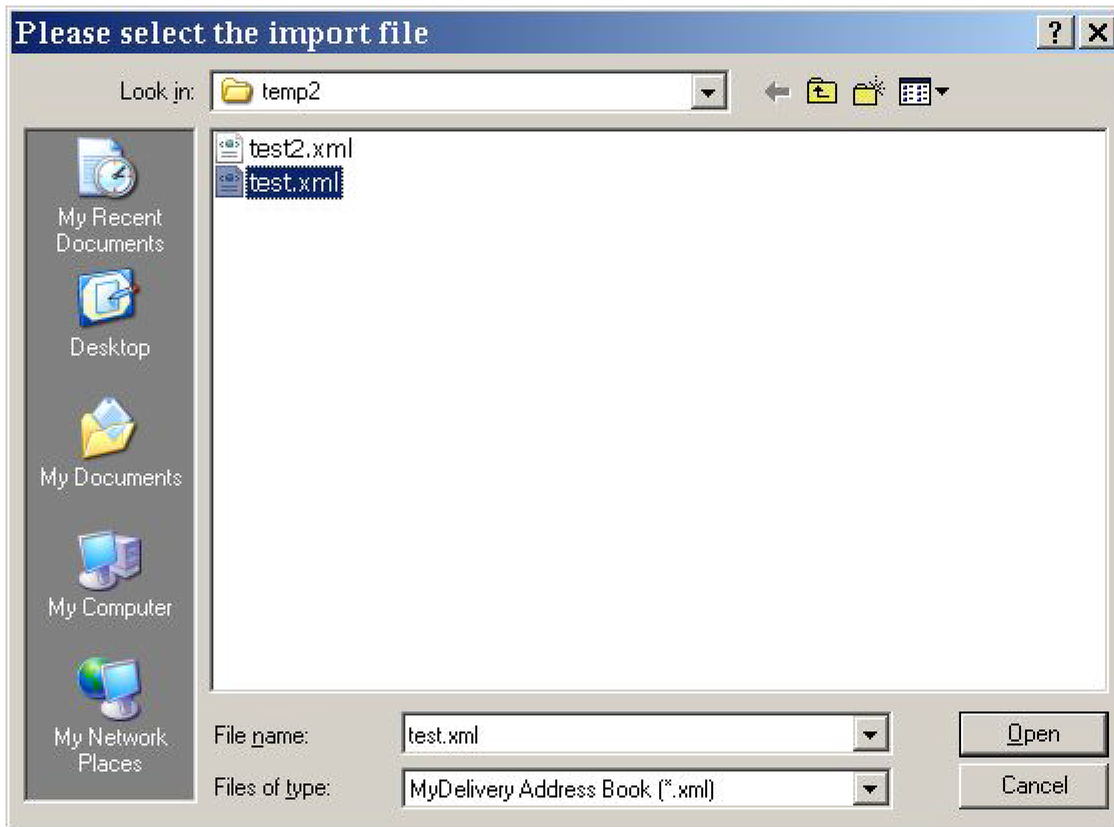
### 1.8 Address Book: Import an Address Book

You can import the list of contacts in an Address Book from another MyDelivery user. This is useful in environments such as library document delivery services, where it is desirable for librarians to share a common set of patron addresses. After one librarian updates their Address Book, they can export it, so that others can import it. This way, everyone's Address Book is the same.



After you select the Import button, a dialog box will prompt you to specify the XML file to be imported.

During importing, a contact from the XML file will be added to the current Address Book only if it is not already there.  If the MyDelivery ID already exists, the imported contact will overwrite the existing contact.  Any existing contact will be kept if the MyDelivery ID for that contact is not in the imported list.



## 1.9 Create a New Delivery

Creating a new delivery is easy. Just follow these steps.

**1.** Bring up the **New Delivery** window by clicking the "New Delivery" button in the main window. You can also do this through the Delivery / New Delivery menu item. The "To" (recipient) field in the window is the only mandatory field; all others are optional.

**2. Enter the recipient's MyDelivery ID.** To do that, click the "To" button (or File /Address in the menu), and select the recipient from your Address Book. If you need to add the recipient to your Address Book, you must add it at this time. Your delivery will be sent only if the recipient is listed in your Address Book. In addition, the recipient must have your MyDelivery ID in his Address Book. Another way to enter the recipient's MyDelivery ID is to directly type it in the field provided. A drop-down list of possible names will be displayed, and when you select the correct name, it will be automatically entered into the field.

**3. Enter the subject.**

**4. Add attachments** by clicking the Attach button (or File / Attach in the menu). Attachments are

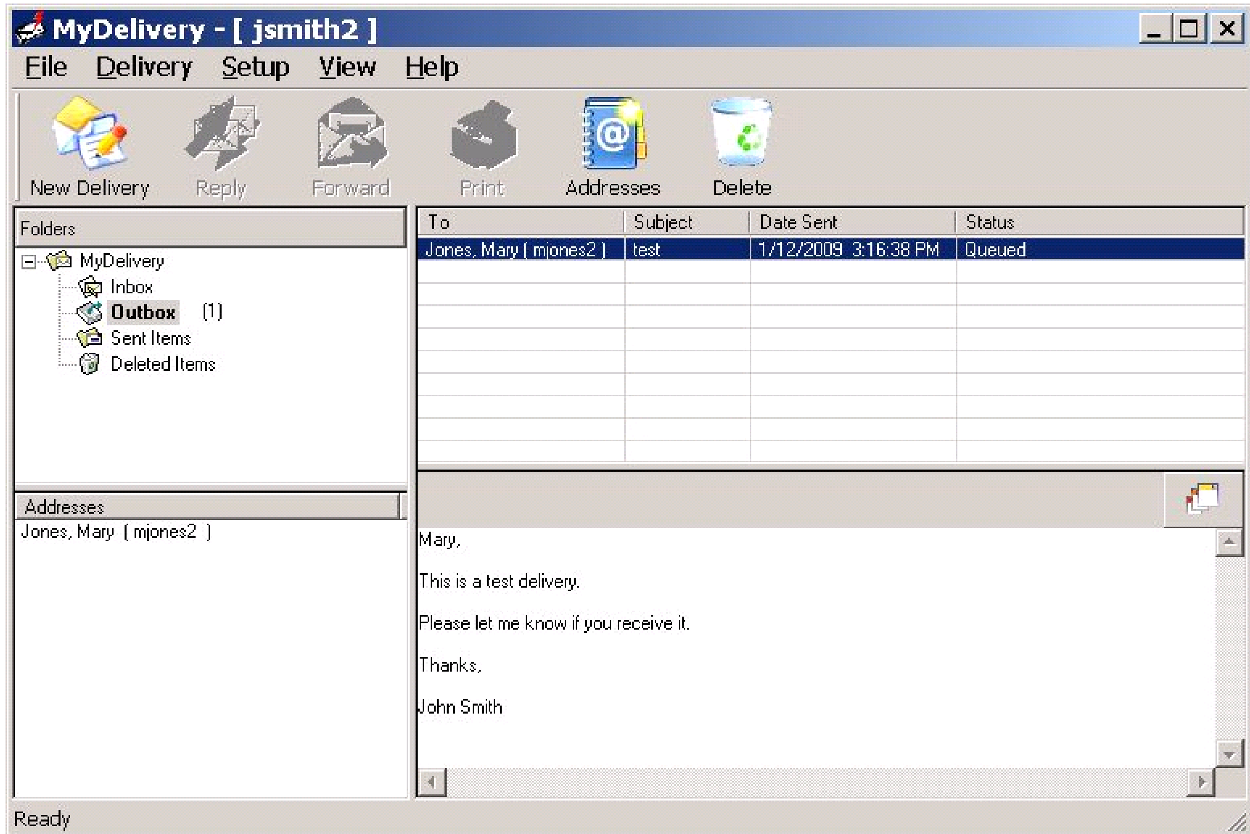
optional, and you may choose to just send a text message.

**5**. **Enter your text message.** Your signature will be automatically appended to the text message if you had selected that option in Setup Signature.

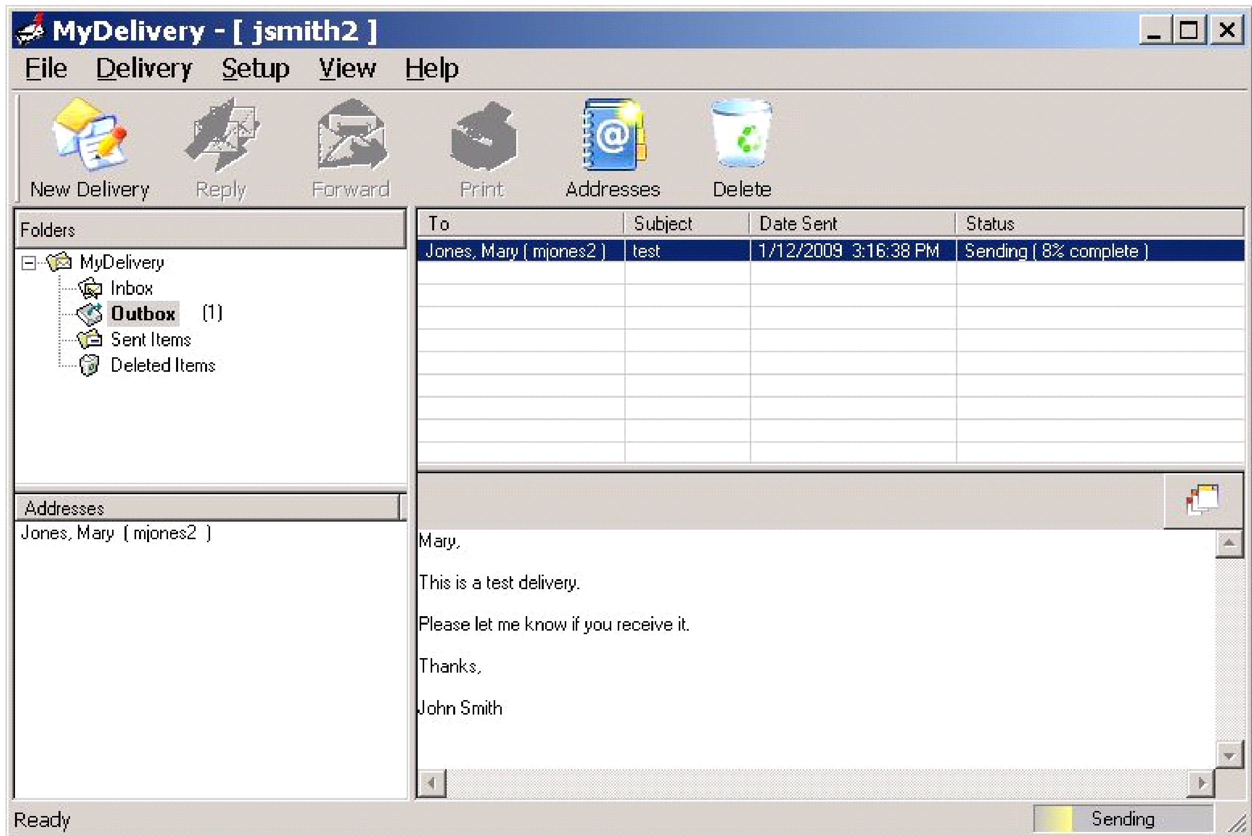
**6. Click Send** (or File / Send in the menu). The delivery will be placed in your Outbox, where you can view its progress while MyDelivery processes it.

## 1.10 Monitor Delivery Progress

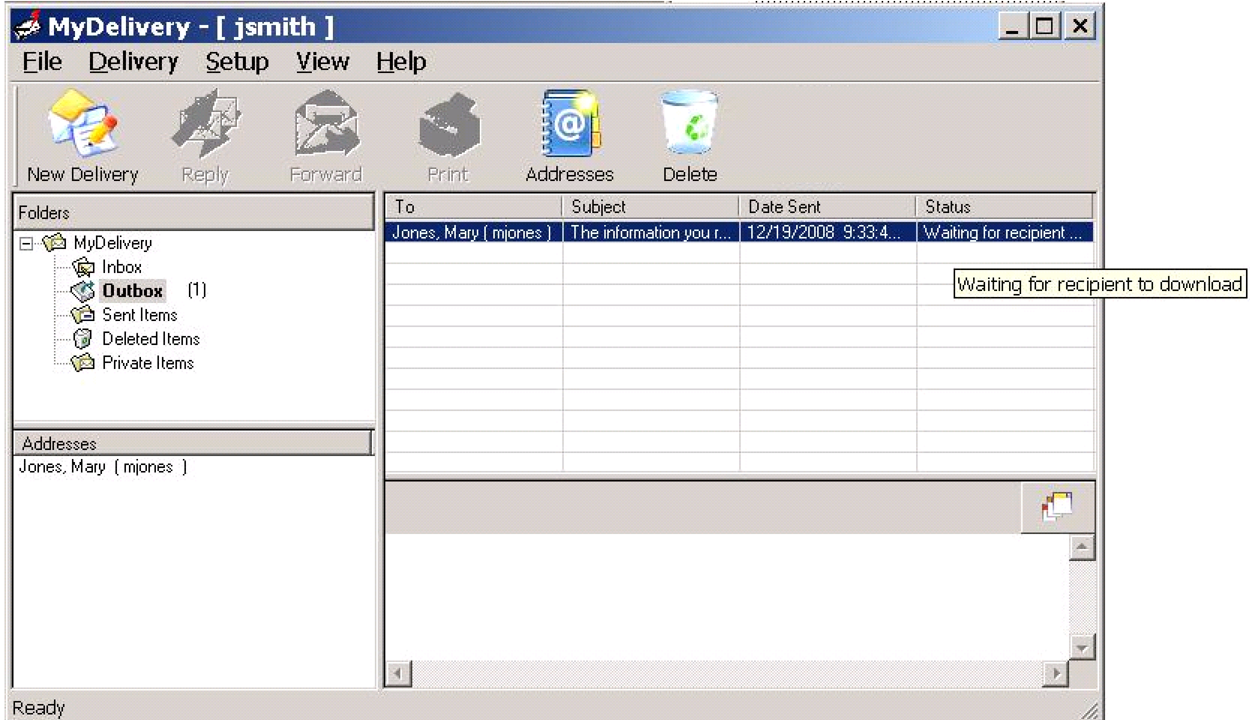
Deliveries that are being sent are placed in the **Outbox**. For each delivery, the Status indicates the progress of processing and transmission. When a delivery is first placed in the Outbox, its status is Queued. During this time MyDelivery makes a backup copy of the data, then compresses it.



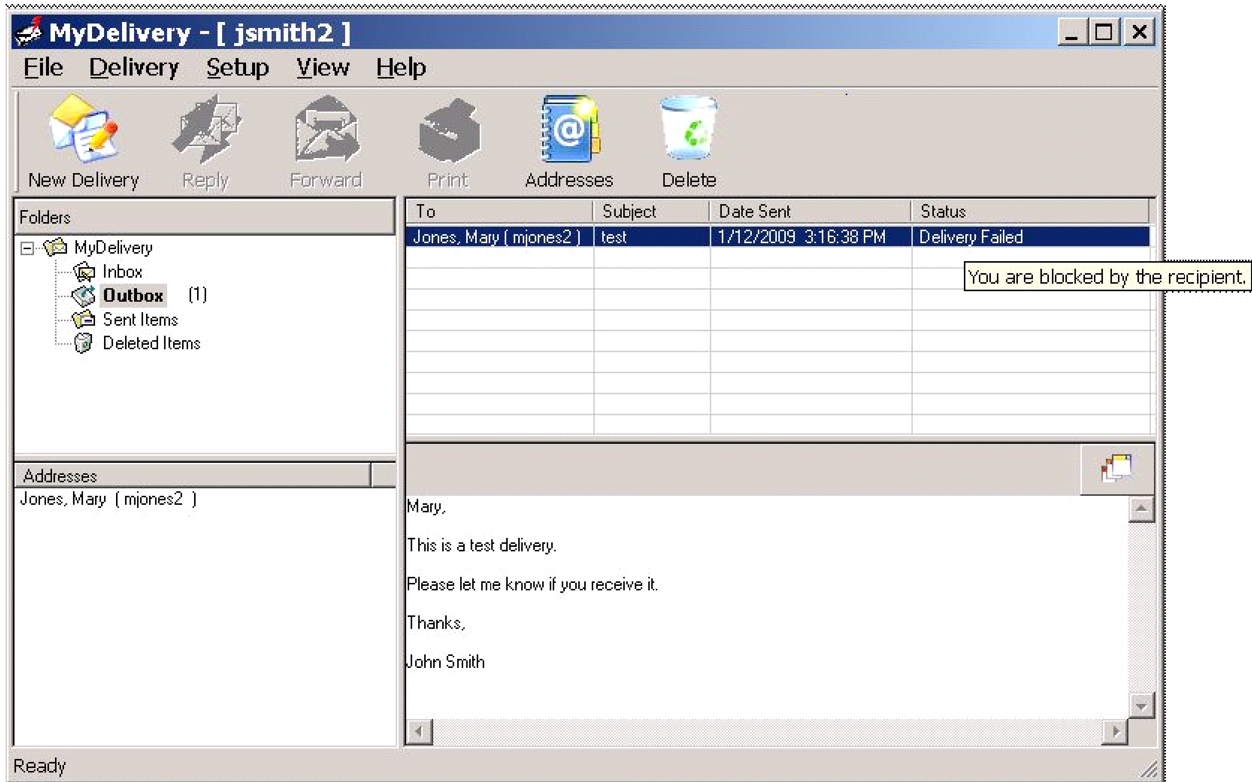

Then it begins transmitting the delivery to the MyDelivery Server only if the receiving client is logged in and online. If the recipient is not online, your client will keep the delivery in its OutBox. If the recipient is online and the delivery commences, the Status indicates the percentage of the delivery that has been transmitted. Notice the animation in the Status bar at the bottom, where it says "Sending". This is another indication that the client is working.


MyDelivery - [ jsmith2 ]
File Delivery Setup View Help
New Delivery
Reply
Forward
Print
Addresses
Delete
Folders
MyDelivery
Inbox
Outbox (1)
Sent Items
Deleted Items
To
Subject
Date Sent
Status
Jones, Mary ( mjones2 )
test
1/12/2009 3:16:38 PM
Sending ( 8% complete )
Addresses
Jones, Mary ( mjones2 )
Mary,
This is a test delivery.
Please let me know if you receive it.
Thanks,
John Smith
Ready
Sending

When the delivery has been sent, but the recipient has not finished processing it, the Outbox Status indicates "Waiting for recipient to download." Once a delivery has been received and processed by the recipient, the delivery is moved from the Outbox to the **Sent Items** folder. That is your indication that the delivery is complete, and the recipient has it. Recipients receive their deliveries in the **Inbox**.

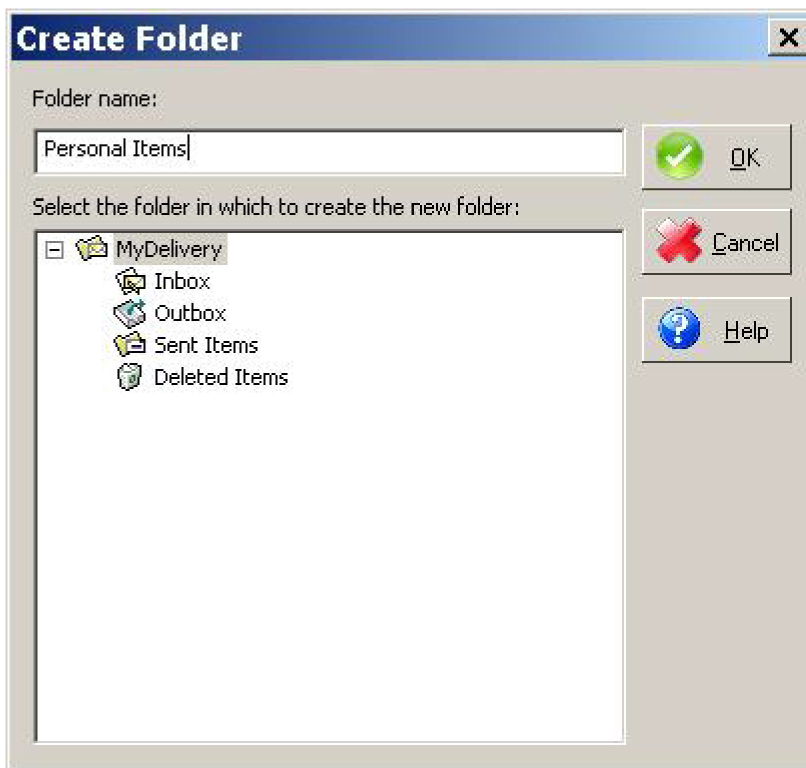


**Why deliveries may fail.** Once in a while a delivery may fail, as indicated in the picture below. If you hover your mouse cursor above the Status, you will see detailed status information, if it exists. This is useful for finding out why deliveries may not get received by the recipient, if they should fail. Deliveries may fail for several reasons. One is if you, the sender, or the recipient lack adequate disk space to receive and process the delivery. A second reason is if the recipient decides to block deliveries that you send.



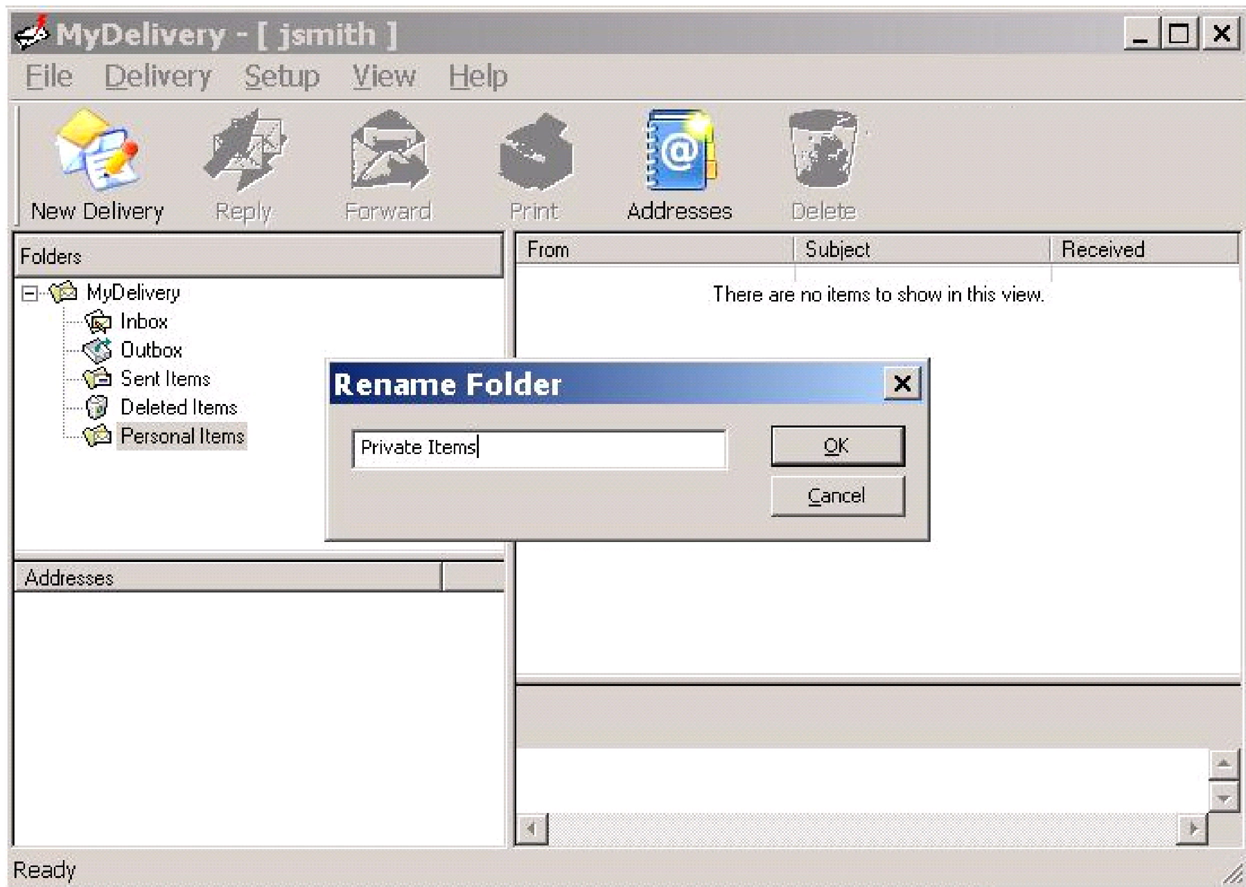
## 1.11 Folder Management: Create a Folder

It is easy to create new folders. There are three ways to bring up the Create Folder dialog box. Two menu options allow you to do this: File / New / Folder, or File / Folder /Create. You may also click your right mouse on any folder in the upper left port of the MyDelivery main window, to bring up a context sensitive menu containing an option for creating folders. Once the Create Folder dialog box appears, enter the name of the folder you wish to create. Then point your cursor to the folder under which the new folder is to be created, and click your left mouse button. Then click OK. In the example below, a folder called Personal Items will be created beneath the MyDelivery root folder. It is possible to create up to 255 levels of folders.



### 1.12 Folder Management: Rename a Folder

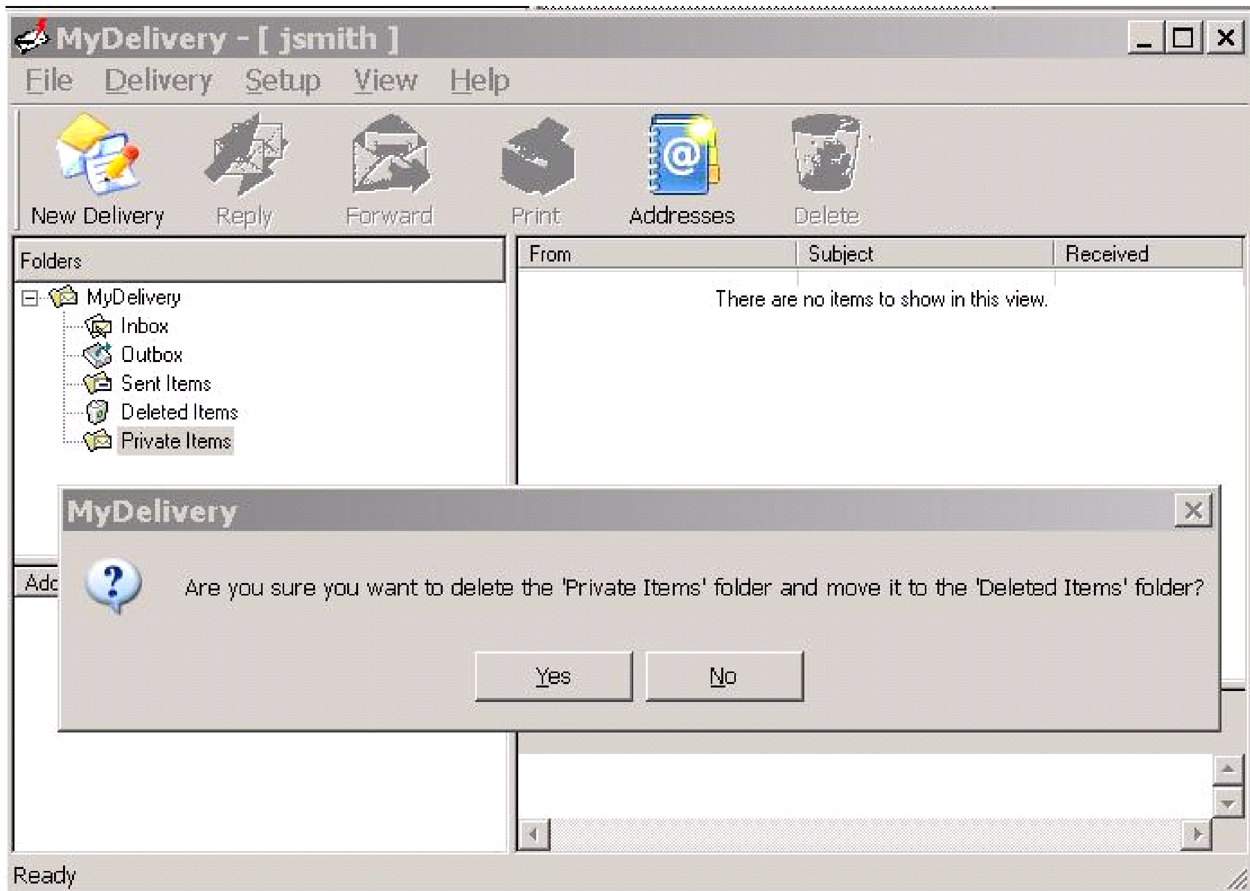
It is possible for you to rename any folder you have created. MyDelivery's four built-in folders (Inbox, Outbox, Sent Items, and Deleted Items) may not be renamed. There are two ways you can invoke the Rename a Folder dialog box. The first is through the File / Folder / Rename menus. The second is through the context sensitive menu. In the latter, point the mouse cursor to the folder to be renamed, and click the right mouse button. This brings up a menu in which Rename is a menu option. Click the left mouse button on Rename. After you enter the new folder name, click OK to save it. In the example below the "Personal Items" folder is renamed to "Private Items."



### 1.13 Folder Management: Delete a Folder

It is easy to delete any folder you have created. MyDelivery's four built-in folders (Inbox, Outbox, Sent Items, and Deleted Items) may not be deleted. You can delete a folder in one of three ways. First, point your mouse cursor to the folder to be deleted. Then press your keyboard's **Delete** key. That brings up the confirmation dialog box as shown in the example below. For this example, if you click "Yes", the folder and its contents will be sent to the Deleted Items folder. The second way to delete a folder is to use the File / Folder / Delete menu items to bring up the confirmation dialog box shown below. The third way to delete a folder is through the context-sensitive menu that pops up when you right-click your mouse button on a folder.

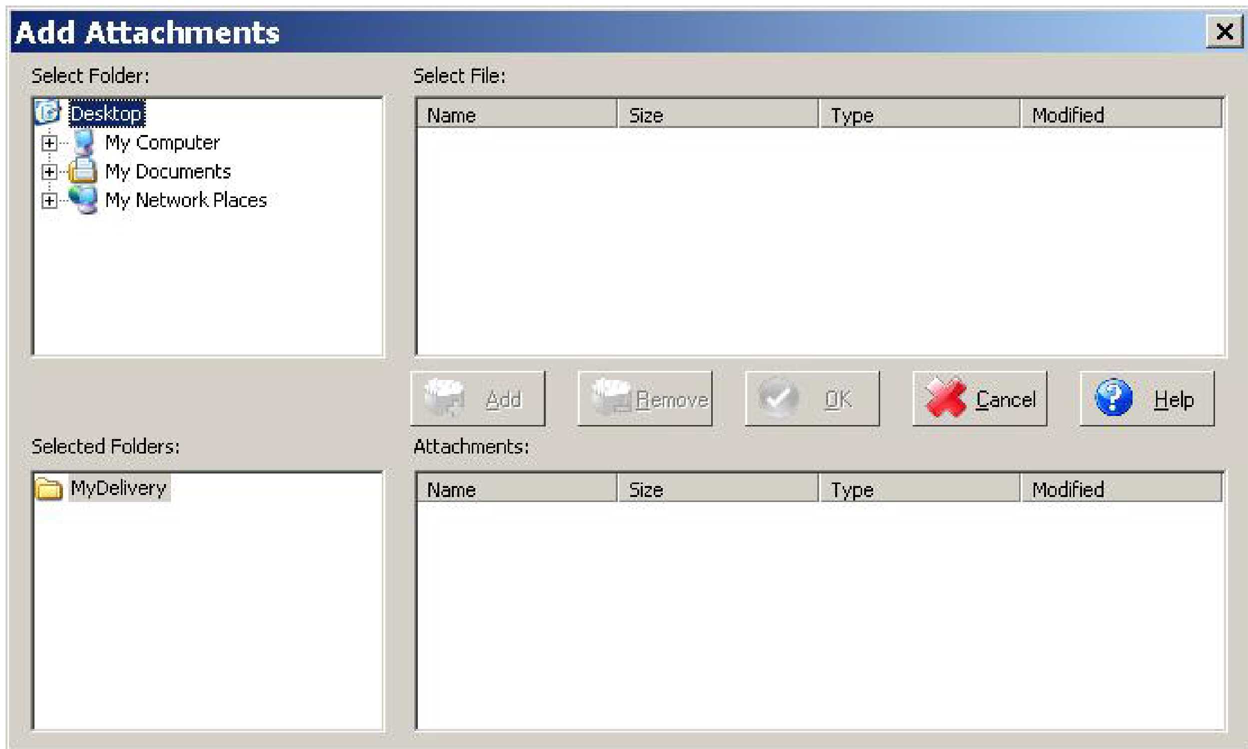
Once a folder and its contents are moved to the **Deleted Items** folder, they still consume disk space. If you want to recover that disk space, you will then have to delete the folder from the Deleted Items folder. You can delete items from the Deleted Items folder either manually or automatically (through the Setup / Options menu item). Once a folder is removed from the Deleted Items folder, it is gone and cannot be restored. The contents of any folder that are in the Deleted Items folder can be moved to any user-created folder outside of the Deleted Items folder by selecting them, dragging them with the mouse, and dropping them into the desired folder.



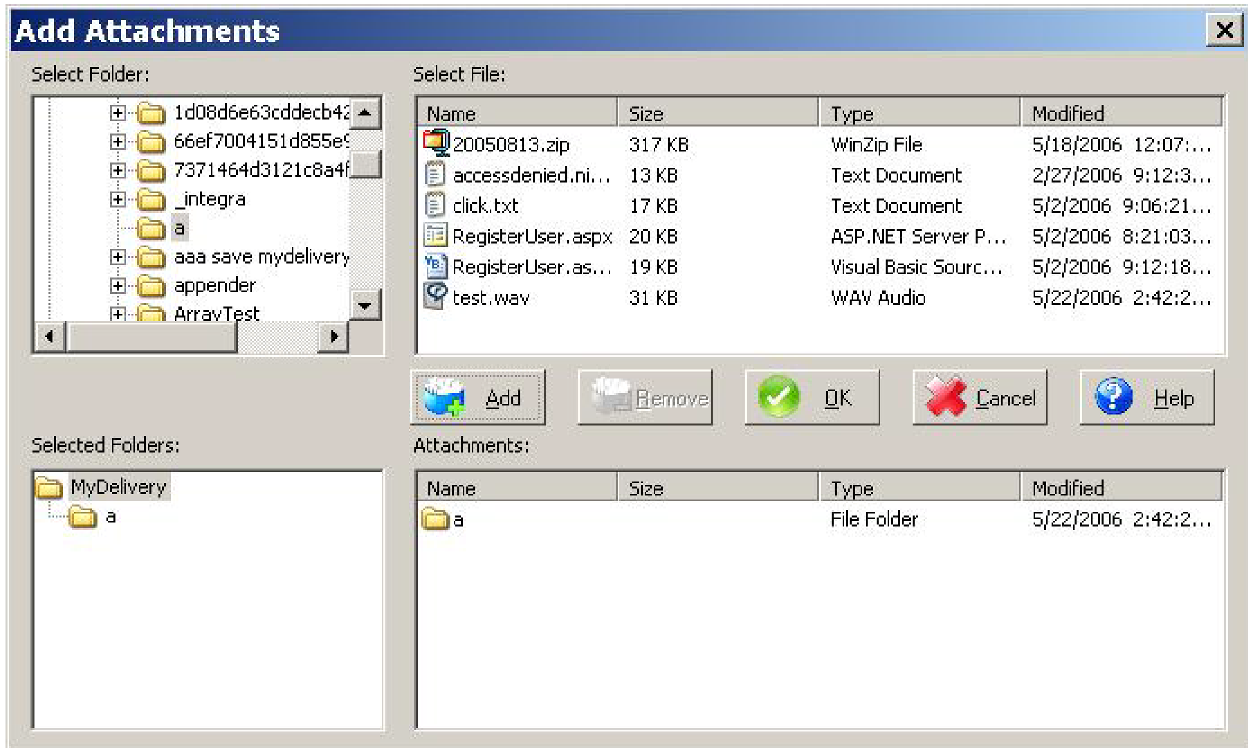
## 1.14 Delivery Management: Add Attachments

The Add attachments dialog box allows you to select folders and files to attach to your delivery. The upper two windows in this dialog box allow you to navigate your computer or network, so that you can select your attachments. The lower two windows show all folders and files that you have selected as attachments.

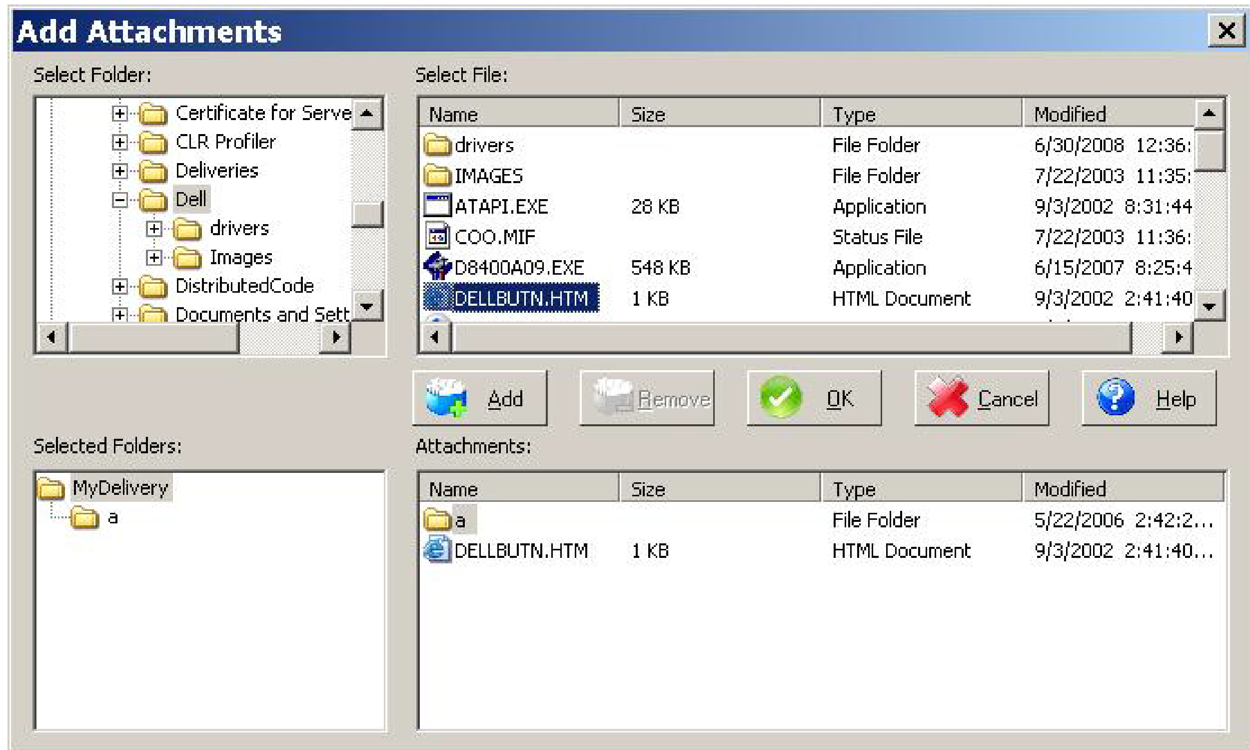
**Navigation.** The upper left window as shown below lists My Computer, My Documents and My Network Places. The names listed here will vary, depending on your operating system. Click the + sign to expand any of these. For example, you can expand My Computer so that you list your disk drives and their contents.



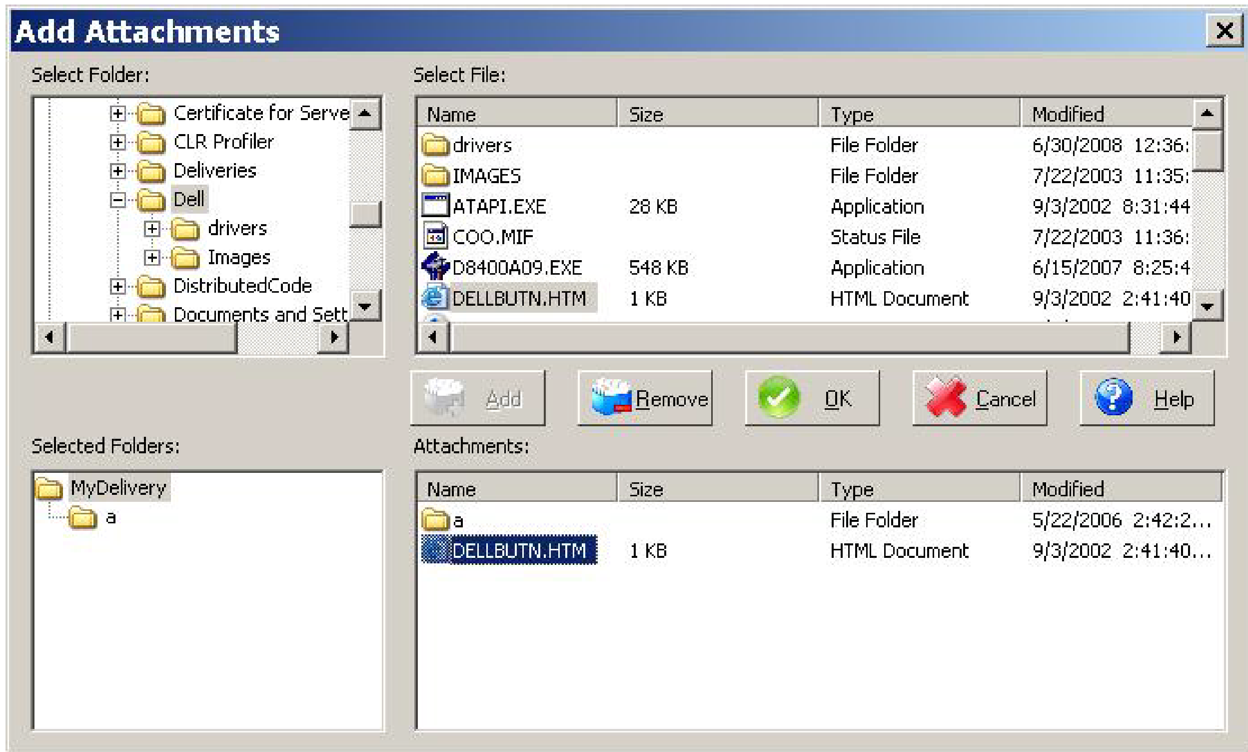
If you click a folder name in the upper left window, its contents will be listed in the upper right window. In the example below the folder named "a" has been selected, and it has six files listed in the upper right window. At this point you can add one or more of the files as attachments to your delivery, or you can add the entire "a" folder. To add an attachment, first select it in the upper left or upper right window, then click the Add button. In this example, the user selected the "a" folder, then clicked the Add button. The "a" folder and the six files will be attached to your delivery.



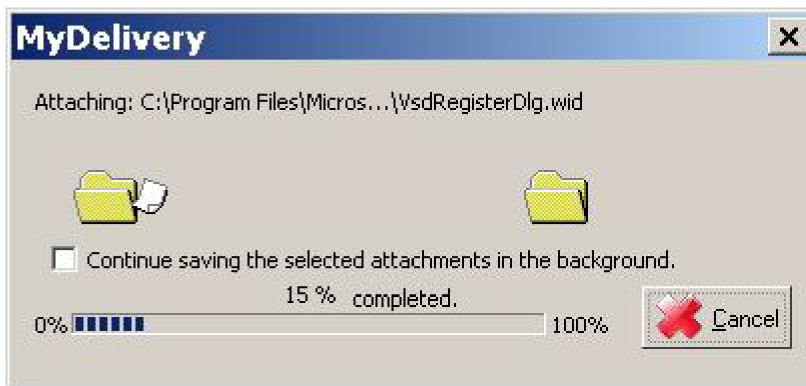
**Add more attachments.** In the example below the user expanded the "Dell" folder in the upper left window and decided to add just one file in that folder. He selected that file in the upper right window, and clicked the Add button.



**Remove an attachment.** If you decide you want to remove an attachment from the delivery, select it in the lower right window. In the example below, the user has decided to remove the file just added, so he clicks the file name, then clicks the Remove button.

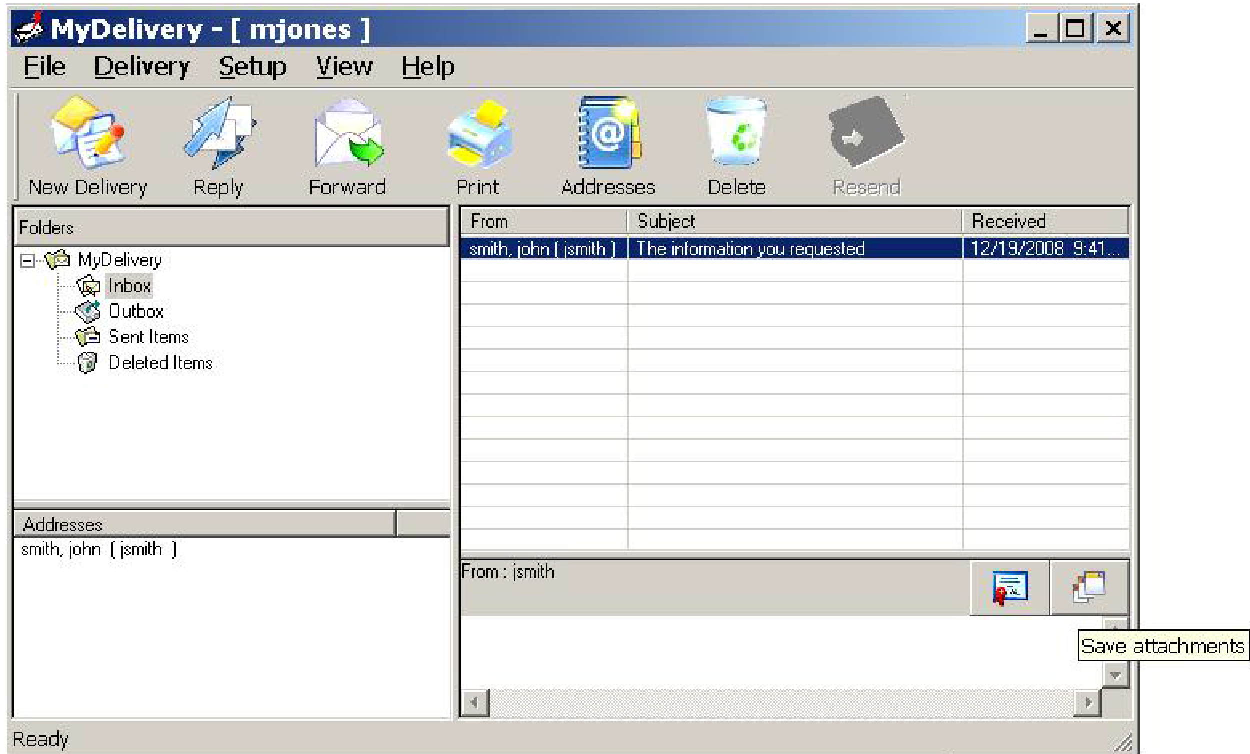



**Finish adding attachments.** Once all attachments have been added to the delivery, click OK. This closes the Add Attachments dialog box and returns you to the Delivery window. If your delivery contains large attachments or large numbers of attachments, the following dialog box appears, and gives you the opportunity to avoid waiting for the attached files to be copied for processing. If you do not want to wait, then select the checkbox for background copying. If you choose the Cancel button, then the attachments you selected will **not** be attached to your delivery.



## 1.15 Delivery Management: Save Attachments

When you receive a delivery that contains attachments, the Save Attachments button in the lower right portion of your client window will be visible and enabled. If you click on it, or select the File / Save Attachments menu item, this will bring up a dialog box that allows you to save the attachments.



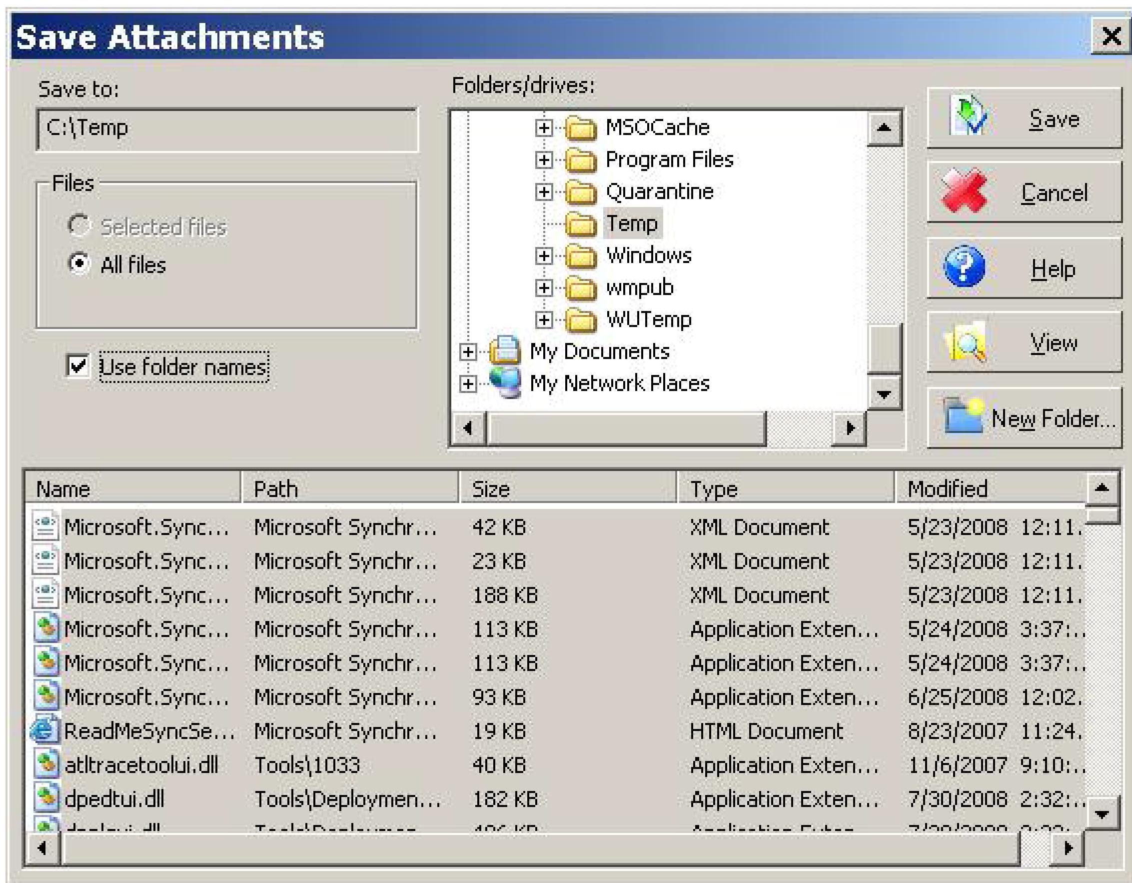

**Folders/drives.** The Save Attachments dialog box allows you to save your attachments to your computer or to a networked computer. Use the Folders/drives navigation window to select where you want to save the attachments.

**Selected Files / All files.** Normally you would want to save all the attached folders and files in the delivery. By default, all attached files in the bottom window are highlighted, and All files is selected. You may also choose to save a portion of the total files if you click the desired files in the bottom window, and choose Selected files.

**Use folder names.** If your delivery contains folders, it is usually desirable to maintain the sender's directory structure when you save the attachments to your hard drive. Select Use folder names to maintain the directory structure.

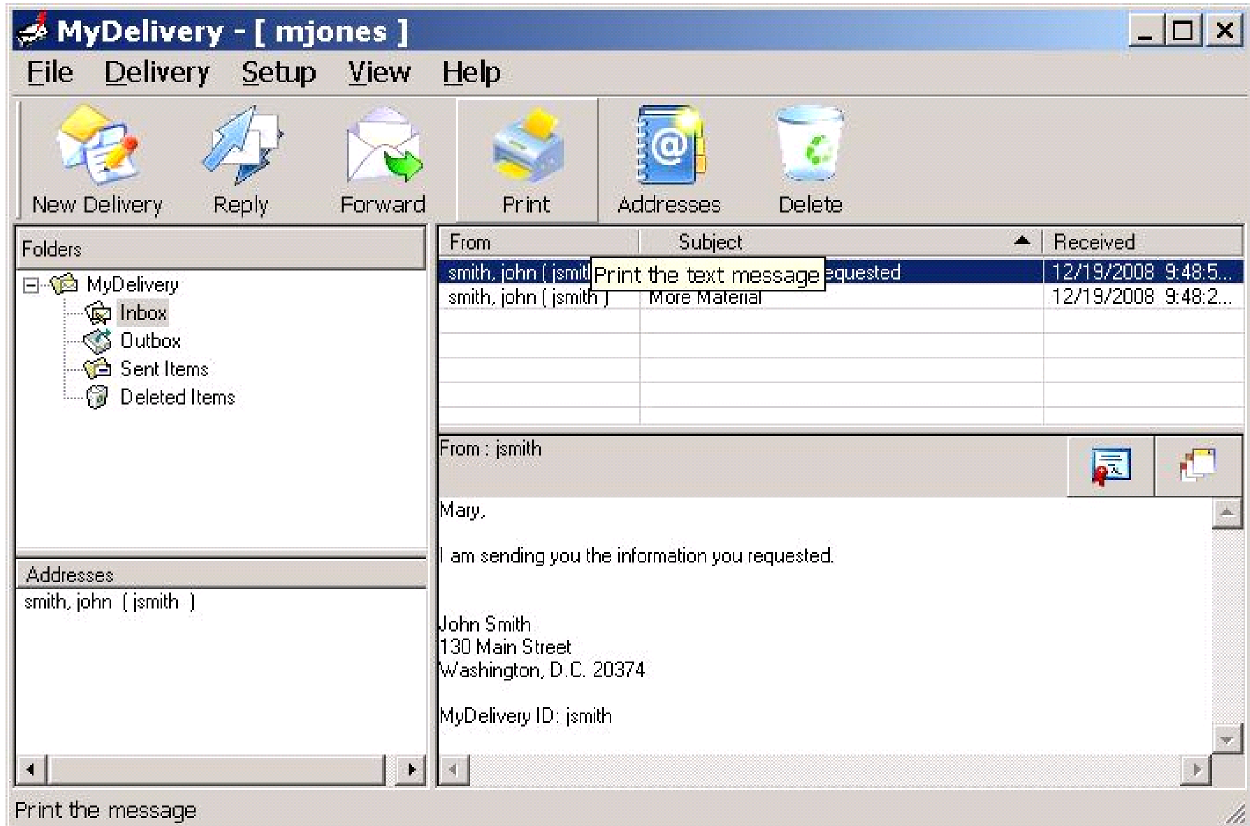
**View.** You may choose to view any attached file if your computer contains the appropriate viewing software. Do this by first selecting the file to be viewed in the bottom window, then click the View button.

**New Folder.** The New Folder button allows you to create a new folder on your hard disk for storing the attachments.


Save Attachments
Save to:
C:\Temp
Files
Selected files
All files
Use folder names
Folders/drives:
MSOCache
Program Files
Quarantine
Temp
Windows
wmpub
WUTemp
My Documents
My Network Places
Save
Cancel
Help
View
New Folder...
Name
Path
Size
Type
Modified
Microsoft.Sync... Microsoft Synchr... 42 KB XML Document 5/23/2008 12:11.
Microsoft.Sync... Microsoft Synchr... 23 KB XML Document 5/23/2008 12:11.
Microsoft.Sync... Microsoft Synchr... 188 KB XML Document 5/23/2008 12:11.
Microsoft.Sync... Microsoft Synchr... 113 KB Application Exten... 5/24/2008 3:37:..
Microsoft.Sync... Microsoft Synchr... 113 KB Application Exten... 5/24/2008 3:37:..
Microsoft.Sync... Microsoft Synchr... 93 KB Application Exten... 6/25/2008 12:02.
ReadMeSyncSe... Microsoft Synchr... 19 KB HTML Document 8/23/2007 11:24.
atltracetoolui.dll Tools\1033 40 KB Application Exten... 11/6/2007 9:10:..
dpedtui.dll Tools\Deploymen... 182 KB Application Exten... 7/30/2008 2:32:..

## 1.16 Delivery Management: Print a Text Message

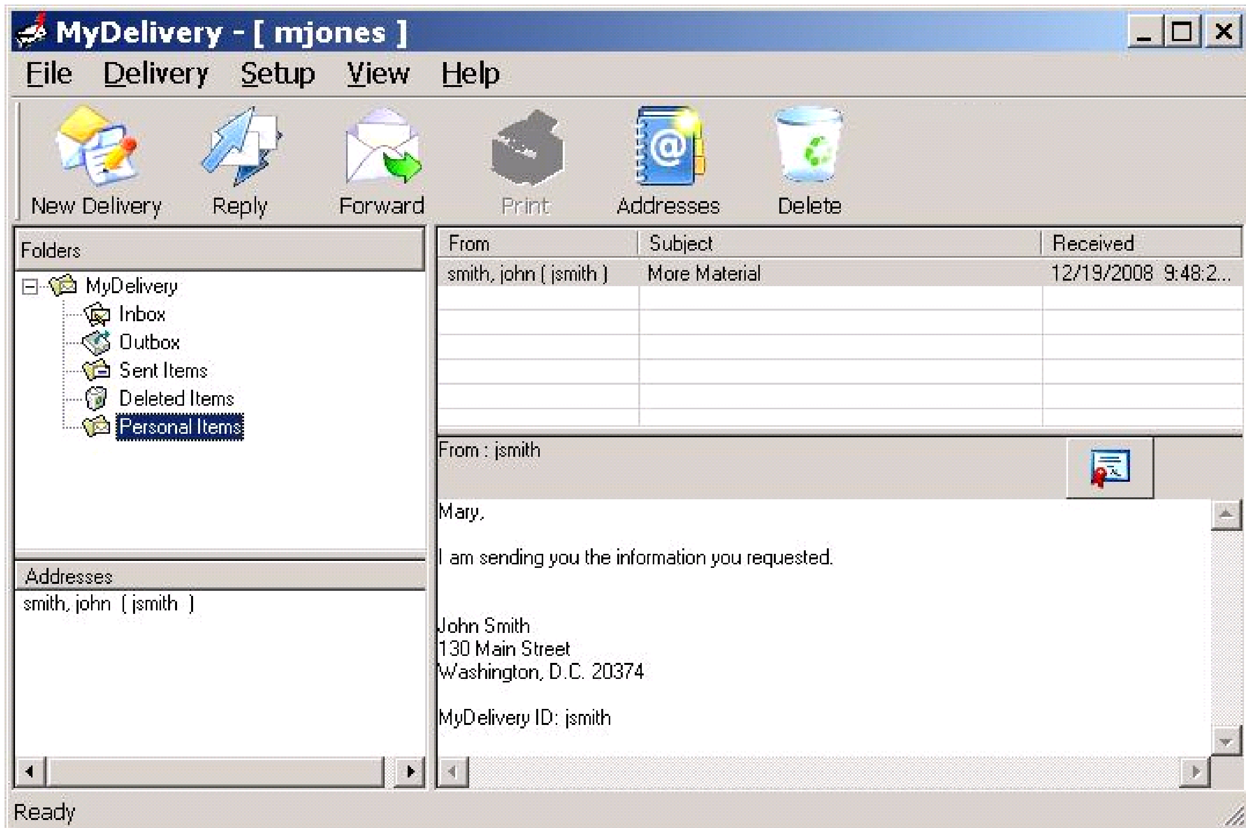
The text message portion of a delivery, displayed in the lower right window, may be printed to a local or networked printer. First, click your mouse button on the text message. This enables the Print button and the File / Print menu item. Select either one to bring up a print dialog box for printing the text message.



### 1.17 Delivery Management: Move a Delivery between Folders

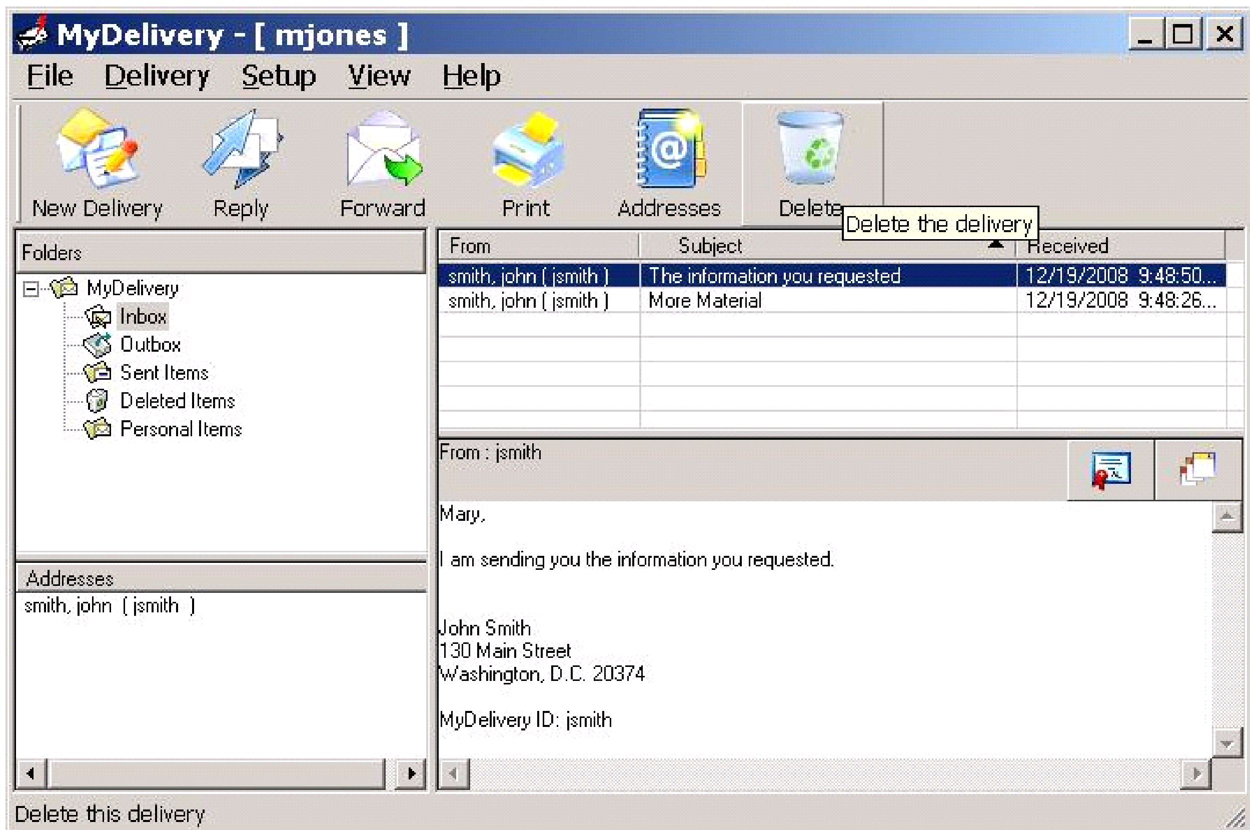
You can move a delivery from the Inbox, Sent Items, or Deleted Items folders to any folder you create, or between user-created folders. There are three ways to do this:

1. Select the delivery to be moved. Using the Delivery menu, select "Move to folder." Use your left mouse button to select the folder that will receive the delivery. Then click OK. This will move the delivery to the selected folder.
2. Context-sensitive menu. Click your right mouse button on a delivery. This brings up a context-sensitive menu of actions you can perform on a delivery. One of them will be "Move to Folder." Use your left mouse button to select the folder that will receive the delivery. Then click OK. This will move the delivery to the selected folder.
3. "Drag and drop." To do this, point your mouse cursor to the delivery to be moved, then click your left mouse button. While keeping your mouse button depressed, move your cursor to the folder that will receive the delivery. After that folder becomes highlighted, release your mouse button. You have "dragged" the delivery out of one folder and "dropped" it into another. In the example below, the delivery is moved from the Inbox to the Personal Items folder.



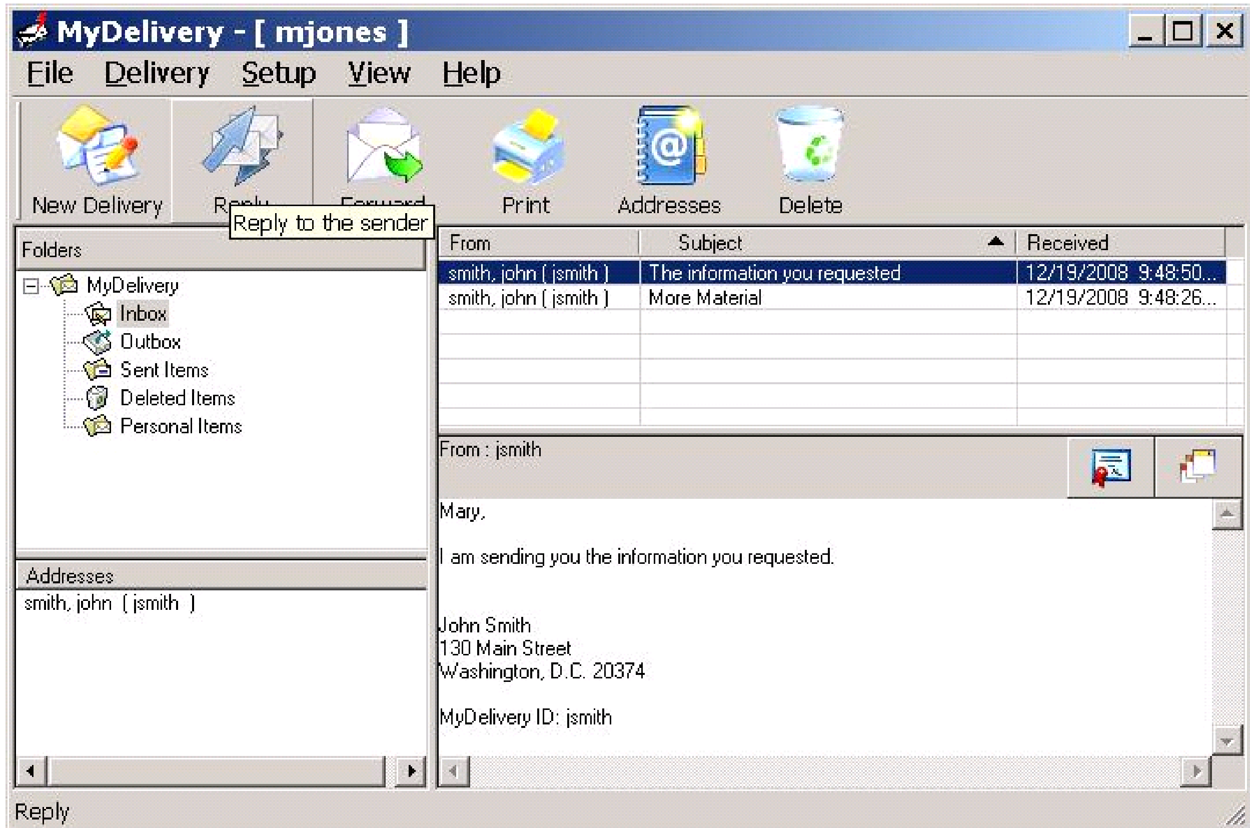
## 1.18 Delivery Management: Delete a Delivery

To delete a delivery, first select it, then click the Delete button (or use the Delivery / Delete menu item. This will move the delivery to the Deleted Items folder. In the example below the delivery in the Personal Items folder is highlighted. This enables the Delete button, which can then be clicked to move the delivery to the Deleted Items folder. Deliveries in the Deleted Items folder still consume hard disk space. You can free up that space by then going to the Deleted Items folder and deleting the folders and deliveries contained there. When you delete items from the Deleted Items folder, MyDelivery will prompt you for confirmation before it proceeds. If there are multiple deliveries in a folder that you want to delete, you can select all of them by using **Control-A** key combination on your keyboard. Then, when you click Delete, all selected deliveries will be deleted simultaneously. If you want to delete only a portion of the deliveries in a folder, press the **Control** key, keep it pressed, and click your mouse button on the deliveries you wish to select for deletion. Another way to remove the contents of the Deleted Items folder is to have MyDelivery do it automatically when it exits.



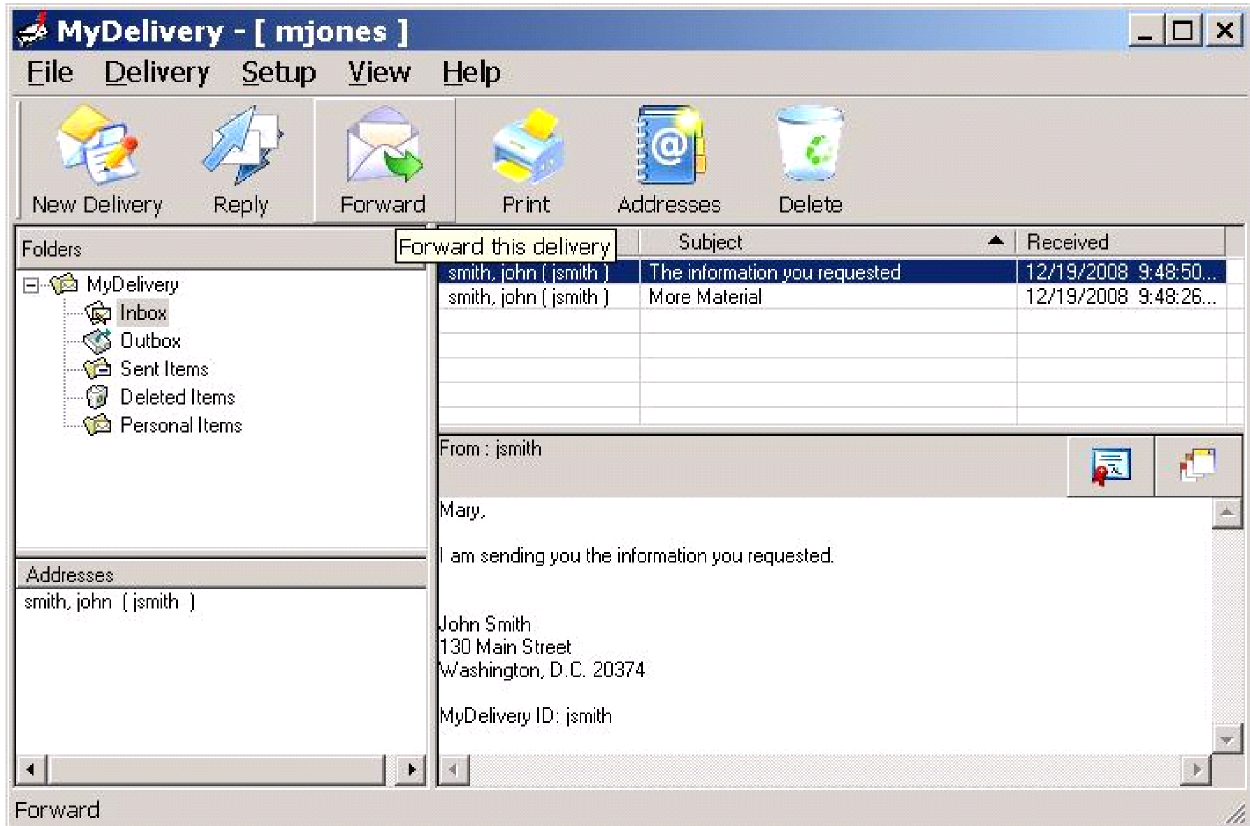
## 1.19 Delivery Management: Reply to a Delivery

To reply to the sender of a delivery that you have received, select the delivery in your Inbox, then click the Reply button (or select Delivery / Reply in the menu). This creates a new delivery with the recipient's name (To field) already filled, and the original text message. Add to the text message if needed. You may or may not wish to add attachments to the delivery. Click Send when you are ready to send the reply.



### 1.20 Delivery Management: Forward a Delivery

You can forward a delivery, including the text message and all attachments by selecting the delivery in the Inbox. Then click the Forward button (or Delivery / Forward in the menu). This creates a new delivery containing the original text message and attachments. You will need to enter the recipient's MyDelivery ID. Click Send when ready.

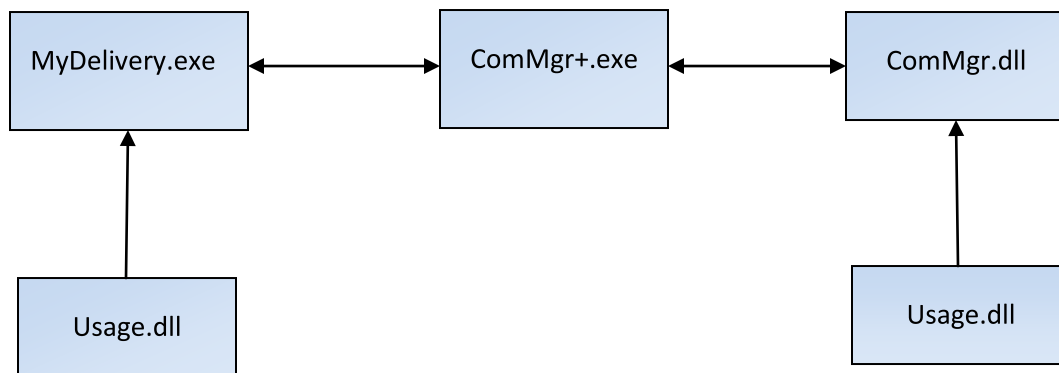


## 2.0 Client Modules

The six modules that comprise the MyDelivery client platform are:

| Module | Description |
|---|---|
| ComMgr.dll | Core DLL responsible for Internet communications |
| MyDelivery.exe | User interface client |
| ComMgr+.exe | API server |
| MDServer.exe | Command line tool for the API |
| APITester.exe | Test tool for API |
| Usage.dll | I/O speed controller |

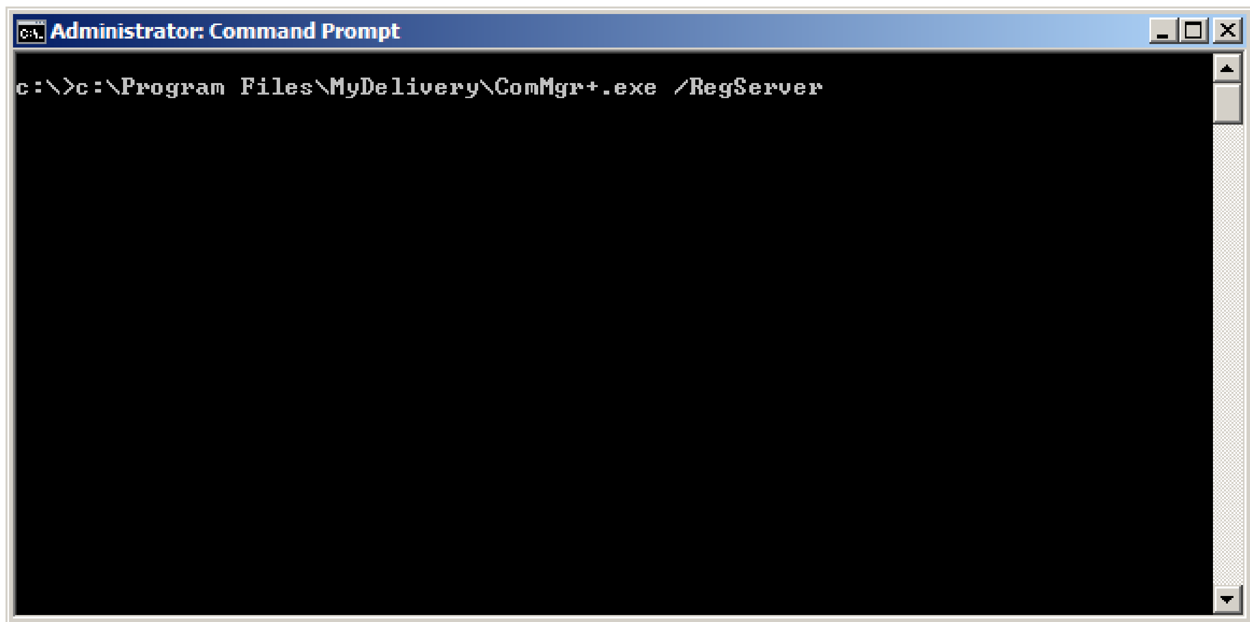
This figure shows five of the six modules required for client operation without the API.




MyDelivery Client Platform without the API

All the modules in the table above are necessary for client execution and are installed by default in the folder MyDelivery which is created below the location specified by the environment variable %PROGRAMFILES% in 32-bit systems or %PROGRAMFILES(X86)% in 64-bit windows systems. Usually the default values are C:\Program Files\MyDelivery on 32-bit systems or C:\Program Files (x86) on 64-bit systems. The value can be checked by running the command echo %PROGRAMFILES% at the command prompt.

The two modules ComMgr+.exe and MyDelivery.exe need to be registered before use. The registration is done by running each module with the argument /RegServer. An example is shown below.




The above registration should also be repeated for mydelivery.exe.

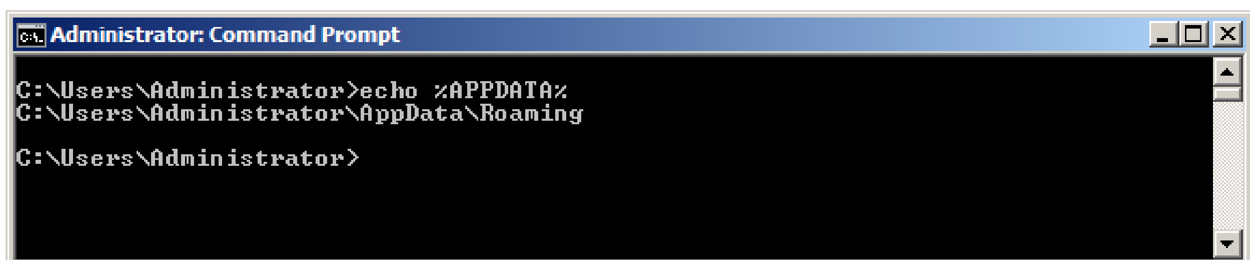
To execute the client, run mydelivery.exe. This will spawn commgr+.exe and associated modules. The MyDelivery Applications Programming Interface (API) allows the client to be replaced (for sending only) by external software that calls the command line interface, provided by MDServer.exe. Instructions for using the API are found elsewhere in this document.

## 2.1 MyDelivery Database

The MyDelivery client platform stores a local database and the deliveries and associated data for each MyDelivery user in a separate folder. The database file 'dbgen.txt' is stored in the folder %APPDATA%\MyDelivery\MyDeliveryID\Database\. The MyDeliveryID is the unique identifier created by a user upon registration with the MyDelivery website.

The `%APPDATA%` folder location varies on each Windows operating system. It is returned by the Windows Shell API function SHGetFolderPath when called with a CSIDL value CSIDL_APPDATA. A typical path is C:\Documents and Settings*username*\Application Data\.

This path can also be easily obtained by executing the command echo %APPDATA% at the command prompt. The image below shows the value of this variable on a machine running Windows 2008 Server.



The database file 'dbgen.txt' holds the list of delivery GUID's.  Each GUID in this list of indentifies a particular delivery.

Each delivery is stored in its own data file named *{GUID}.msg* which has the contents of the delivery. This file is created in the same folder that holds dbgen.txt.  The contents of the delivery are XML tags that hold the values such as subject, message text and attachments.

## 2.2 Client Source Code Compilation

The MyDelivery client source code is intended to be compiled by Visual Studio 2010.  Using this development platform, open the solution file:

..\NIH\ComMgr+\ComMgr+.sln .  There will be projects for APITest, ComMgr, ComMgr+, MDServer, MyDelivery, and Usage.

The only customization that needs to be done to the source code is to modify it to point to the URL of the MyDelivery server.  Open the wsdl.cpp source file found in the ComMgr project.  Modify the definition of WEBSERVER to point to the appropriate URL.  The default URL in the distribution source code is:

```
#define WEBSERVER      http://mydelivery8.nlm.nih.gov/
```

For instance, if secure communications is used, and the website is tester.com, use:

#define WEBSERVER https://tester.com .

To build the client software, execute: Rebuild Solution.  The six modules will be created in the ..\NIH\bin folder.

## 2.3 Additional Modules

The MyDelivery system requires additional merge modules for proper operation. These are available from the Microsoft Developer's Network, http://msdn.microsoft.com.  These modules should be included with an installation program that installs the six basic client modules.

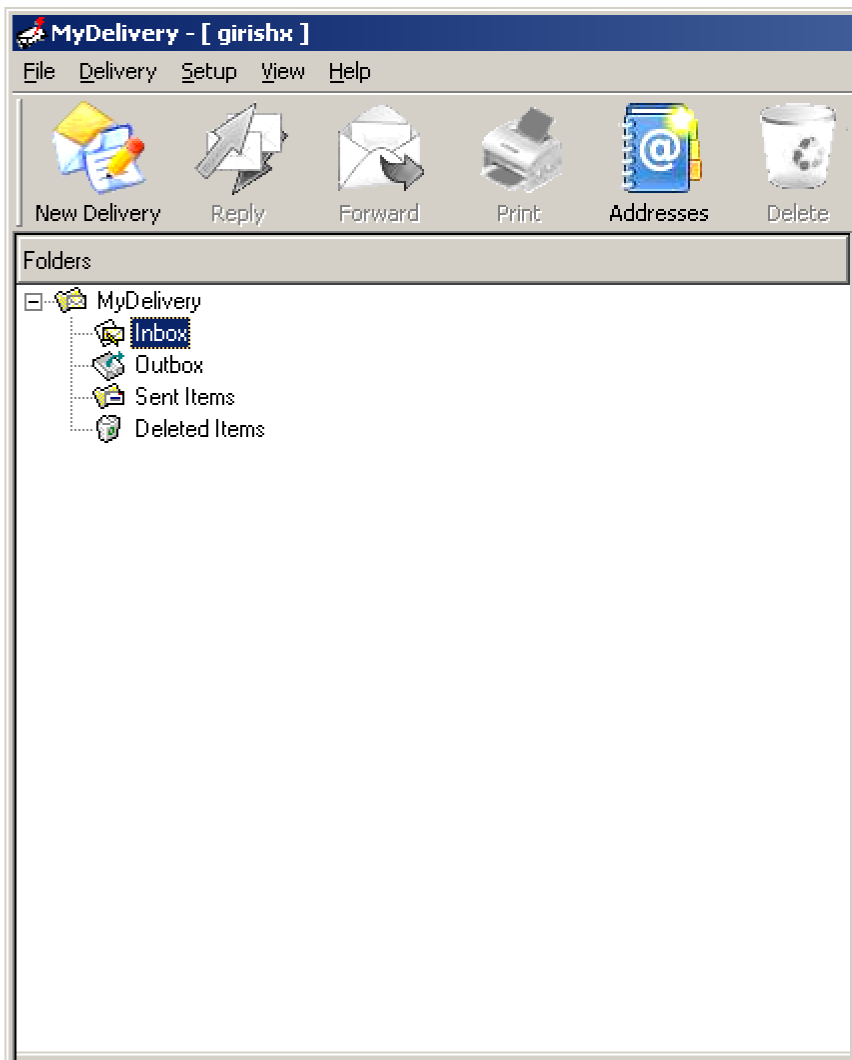
- MSXML4 merge module, version 4.20.9818.0 Module name: msxml4.msm.
- SOAP SDK files merge module, version 3.00.1325.3 Module name: soap3_core.msm.
- SOAP SDK ISAPI files version 3.00.1325.3. Module name: isapi3_files.msm.
- WinHTTP QFE installer merge module, version 3.00.1325.3. Module name WinHTTP51.msm.

## 3.0 Module: MyDelivery.exe

The MyDelivery client user interface, MyDelivery.exe, is a Microsoft Foundation Class (MFC) application that has Single Document Interface (SDI) architecture with various views split into panels that display the list of deliveries, folders, address book and contents of a single delivery.

### 3.1 Folders

The MyDelivery client has four permanent folders that hold deliveries. They are Inbox, Outbox, Sent Items and Deleted Items. The Inbox holds the deliveries which are received, the Outbox holds deliveries that are scheduled to be sent or being sent, the Sent Items holds deliveries that have been sent out and the Deleted Items folder holds deliveries that have been deleted from other folders. One may also create new user created folders by selecting the File-> New -> Folder menu item.



The Deliveries and Folders are stored inside a structured storage document file named dbgen.txt. This file is located at the %APPDATA%\MyDelivery\mydeliveryid\Database. The structured storage database dbgen.txt internally models a file system. Each folder that we create in the folder view (the upper left quadrant of the Mydelivery client user interface) is a stream internally stored in the database. It is accessed using an IStream enumerator obtained by the parent stream. This hierarchy eventually terminates at the root stream. All folders in the database are children of the root stream. This hierarchy is traversed and the folder list populated as shown below.

The class CDBMgr defined in the file dbmgr.h and dbmgr.cpp is responsible for reading and writing from these files. The database IO is exclusively performed by the module MyDelivery.exe. The MyDelivery.exe queries the file dbgen.txt when it needs to display the list of files in a folder. It reads from the file {GUID}.msg when it needs to display the contents of a particular delivery.

Module: MyDelivery\DBMgr.cpp

```
HRESULT CDBMgr::UpdateTree(   IStoragePtr stg,
                              CTreeCtrl* tree,
                              HTREEITEM parent
                            )
{

      //obtain enumerator
      IEnumSTATSTGPtr ptrEnum;
      m_hr = stg->EnumElements(0, NULL, 0, &ptrEnum);
      if(FAILED(m_hr)){
            return m_hr;
      }

      //find type data
      STATSTG statstg;
      memset(&statstg, 0, sizeof(statstg));

      HRESULT hr = ptrEnum->Next( 1, &statstg, 0 );
      m_hr = hr;
      if(FAILED(m_hr)){
            return m_hr;
      }

      std::vector<STATSTG> folderlist;

      // Loop through all the direct children of this storage.
      while( S_OK == hr )
      {
          // Check if this is a storage that isn't a property set
          // property sets start with '\005' reserved character.
          if( STGTY_STORAGE == statstg.type  && L'\005' !=
  statstg.pwcsName[0])
          {
                  folderlist.push_back(statstg);
          }
```

```
            hr = ptrEnum->Next( 1, &statstg, 0 );
        }

        :
        :

        for(std::vector<STATSTG>::iterator itor = folderlist.begin();
              itor < folderlist.end();
              itor++, image++)
        {
              if( STGTY_STORAGE == statstg.type …)
              {
                    IStoragePtr child;
                    HTREEITEM newparent = NULL;

                    m_hr = stg->OpenStorage(statstg.pwcsName, NULL,
STG_OPEN_MODE, 0, 0, &child);

                    m_hr = ReadMultipleProperties(child, 2, property, value);

                    if(m_hr == S_OK)
                    {
                          //populate tree
                          :
                          tree->SetItem(&tvi);
                          :

                    }

                    //recurse
                    UpdateTree(child, tree, newparent);
              }

              :

        }

    return m_hr;
}
```

### 3.2 Deliveries

Each child within the folder is an IStream element with a link to another structured storage document that has all the details of a delivery embedded in it. In this fashion a list of deliveries are housed inside a folder. A suitable comparison would be a directory holding a list of files. The list of deliveries is obtained as shown below.

Module: MyDelivery\DBMgr.cpp

```
HRESULT CDBMgr::EnumerateFiles(TCHAR* at, vector<string>& _file_list)
{
      if(at == NULL || m_database == NULL)
            return E_FAIL;

      IStoragePtr stg;
      basic_string<WCHAR> stgname;
      CAutoStorageList al;

      m_hr = FindFolder(m_database, at, stg, stgname, al.slist);
      if(FAILED(m_hr))
        return m_hr;

      //obtain enumerator
      IEnumSTATSTGPtr ptrEnum;
      m_hr = stg->EnumElements(0, NULL, 0, &ptrEnum);
      if(FAILED(m_hr))
      return m_hr;

      //find type data
      STATSTG statstg;
      memset(&statstg, 0, sizeof(statstg));
      HRESULT hr = ptrEnum->Next( 1, &statstg, 0 );
       m_hr = hr;

      //there are no substorages/streams
      if(hr == S_FALSE)
         return S_FALSE;

      if(FAILED(m_hr)){
         return m_hr;
      }

    // Loop through all the direct children of this storage.
    basic_string<WCHAR> fileguid;
    while( S_OK == m_hr )
    {
        if( STGTY_STREAM == statstg.type &&
            L'\005' != statstg.pwcsName[0] &&
                  wcsstr(statstg.pwcsName, L"stream_") != NULL )
        {
            // Yes, this is a normal storage, not a propset.
            IStreamPtr _element;
            m_hr = stg->OpenStream(statstg.pwcsName, .., &_element);

            BYTE* data = NULL;
            DWORD dwBytes = 0;
```

```
            if(ReadBytes(&data, dwBytes, _element))
            {
                  //release memory
                  std::auto_ptr<BYTE> ar_data(data);
                  std::auto_ptr<char> filename(_wcmb((WCHAR*)data));

                  string str = filename.get();
                  _file_list.push_back(str);
            }
            else
            {
                  if(statstg.pwcsName != NULL)
                  {
                        CoTaskMemFree(statstg.pwcsName);
                        statstg.pwcsName = NULL;
                  }

                  m_hr = E_FAIL;
                  return E_FAIL;
            }

        }

        // Move to the next element in the enumeration of this storage.
        :
        m_hr = ptrEnum->Next( 1, &statstg, 0 );

      }

      return S_OK;
}
```

## 3.3 Delivery Move

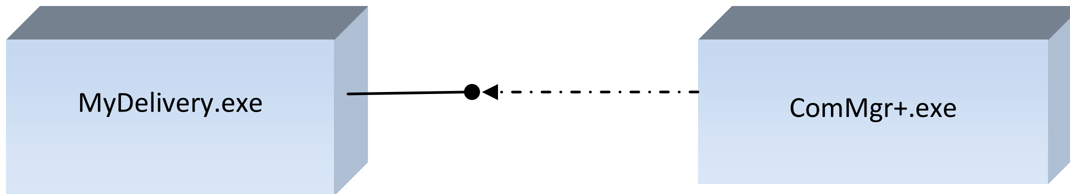
When a move operation occurs to a different folder only the links are updated by transferring ownership of a stream to a different parent. These IStream elements are destroyed and released when we clear items from the Deleted Items folder.

Module: MyDelivery\DBMgr.cpp

```
HRESULT CDBMgr::MoveFolder(TCHAR* from, TCHAR* to, char* newfoldername)
{
	:
	:

	//get the name for the source folder and its IStorage pointer
	m_hr = FindFolder(m_database, from, .., stgname, ..);

	:
	:

	//move the source folder
	if(m_hr == S_OK)
	{
		:
		:

		//get the IStorage object for the target folder
		m_hr = FindFolder(m_database, to, _target_stg, _name_target, ..);

		:

		//move source folder below target folder
		m_hr = _parent_stg->MoveElementTo(stgname.c_str(), _target_stg,
stgname.c_str(), STGMOVE_MOVE);
	}

	//done!
	return m_hr;
}
```

### 3.4 MyDelivery Event Sink Setup

The MyDelivery application, mydelivery.exe, subscribes to the API server, ComMgr+.exe, by providing an IUnkown interface pointer to its sink interface. The sink captures 'events' as they occur and converts them to suitable deliveries that are 'processed' as they arrive by the main application thread.




Event callback using passed IUnknown

The sink interface itself is described in the file MyDelivery.idl. The interface IDL described below is the sink definition which is passed to the ComMgr+.exe server as outlined in the next step.

```
                         Module: MyDelivery\MyDelivery.idl
:
:
interface IMyDeliveryClient : IDispatch{
      [id(1), helpstring("Initializes the MyDelivery client.")] HRESULT
InitializeClient(void);
      [id(2), helpstring("Notifies an incoming delivery.")] HRESULT
NotifyNewDelivery([in] BSTR from, [in] BSTR subject, [in] BSTR message, [in]
BSTR Key);
      [id(3), helpstring("Notifies the sent status of a message.")] HRESULT
NotifyDeliveriestatus([in] BSTR msgID, [in] BSTR status, [in] LONG ack);
      [id(4), helpstring("Notifies delivery failure.")] HRESULT
NotifyDeliveryFailure([in] BSTR from, [in] BSTR subject, [in] BSTR message,
[in] BSTR msgID);
      [id(5), helpstring("Shuts down the client interface.")] HRESULT
ShutdownClient([in] LONG exitcode, [in] BSTR Reason);
      [id(6), helpstring("method CommStatus")] HRESULT CommStatus(ULONG
status);
      [id(7), helpstring("method Sets the caption in the active window")]
HRESULT SetWindowMessage([in] BSTR text);
      [id(8), helpstring("method Gets the caption in the active window")]
HRESULT GetWindowMessage([in, out] BSTR* text);
      [id(9), helpstring("method Displays a message box")] HRESULT
DisplayMessage([in] BSTR text, [in] LONG style);
      [id(10), helpstring("Initializes the MyDelivery client.")] HRESULT
IncrementOutboxUnread(void);
      [id(11), helpstring("method Deletes a message from view")] HRESULT
DeleteMessageFromOutboxView([in] BSTR msgid);
      [id(12), helpstring("Request terminate clear")] HRESULT
NotifyTerminate([in] BSTR msgid, [in,out] LONG* ok);
```

```
	[id(13), helpstring("Release terminate lock")] HRESULT 
ReleaseTerminate(void);
};
```

The sink is passed to the ComMgr+.exe by obtaining a pointer to the connection point interface as shown below. It is connected by calling the method **Advise** as shown below.

```
                              Module: MyDelivery\MyDelivery.cpp

bool CMyDeliveryApp::CreateServiceInterface(LPCTSTR username)
{
	//create class object
	HRESULT hr = CoCreateInstance(__uuidof(MyDeliveryService), 0, 
CLSCTX_LOCAL_SERVER, IID_IUnknown, (void**)&m_pServer);
	if(FAILED(hr))
		return false;

	//find connection point container
	IConnectionPointContainerPtr cpc;
	hr = m_pServer.QueryInterface(__uuidof(IConnectionPointContainer), 
&cpc);
	if(hr == E_NOINTERFACE)
	{
		AfxMessageBox("Could not find connection point container 
interface.");
		return false;
	}

	//find connection point interface
	hr = cpc->FindConnectionPoint(__uuidof(_IMyDeliveryServiceEvents), 
&m_mdsecp);
	if(hr == E_NOINTERFACE)
	{
		AfxMessageBox("Could not find mydelivery service events 
interface.");
		return false;
	}

	//create sink object
	m_sink = new CComObject<CMyDeliveryClient>;

	:

	//subscribe to event updates
	HRESULT hr = m_mdsecp->Advise(m_sink.GetInterfacePtr(), &m_cpc);
	if(hr != S_OK){
		AfxMessageBox("Could not subscribe to MyDelivery events");
	}

	return true;
}
```

### 3.5 MyDelivery Event Sink Implementation

When the MyDelivery client is updated with the events passed by the API platform it is handled by converting the method calls to deliveries. These deliveries are handled in the file MainFrm.cpp

```
                    Module: MyDelivery\MyDeliveryClient.cpp

STDMETHODIMP CMyDeliveryClient::NotifyNewDelivery(BSTR in_from,
                                                  BSTR in_subject,
                                                  BSTR in_message,
                                                  BSTR in_Key)
{
      :
      LPARAM lParam = (LPARAM)new NewDeliveryPacket(. . .)
      :
      ::PostMessage(m_hWnd, WM_SERVER_MESSAGE1, 0, lParam);
      :

      return S_OK;
}
```

```
                    Module: MyDelivery\MyDeliveryClient.cpp

STDMETHODIMP CMyDeliveryClient::NotifyDeliveriestatus(BSTR msgID, BSTR
status, LONG ack)
{
      :

      LPARAM lParam = (LPARAM)new DeliveriestatusPacket(msgID, status, ack);

      ::PostMessage(m_hWnd, WM_SERVER_MESSAGE2, 0, lParam);

      return S_OK;
}
```

```
                    Module: MyDelivery\MyDeliveryClient.cpp

STDMETHODIMP CMyDeliveryClient::NotifyDeliveryFailure(BSTR in_from,
                                                      BSTR in_subject,
                                                      BSTR in_message,
                                                      BSTR in_msgID)
{

      LPARAM lParam = (LPARAM)new DeliveryFailedPacket(. . . );

      ::PostMessage(m_hWnd, WM_SERVER_MESSAGE3, 0, lParam);

      return S_OK;
}
```

### 3.6 Software Updates

When the user enters his MyDeliveryID and password in the login dialog box, the process of authorization is started. The first step in the authorization process is checking if the MyDelivery user has the latest software. This is done by calling the server function GetVersion (MyDelivery\ComMgrDlg.cpp) in the AuthorizeThread (MyDelivery\ComMgrDlg.cpp).

The server function GetVersion is used by ComMgr.dll to verify the version of the installed software. This is the first function called by the ComMgr.dll and does not require prior authorization. This function is implemented by CComMgrApp::GetVersion (..)(ComMgr\ComMgr.cpp). This call is made before the authorization function Initialize is called.

The parameters to the function are:

- The version string returned by the server.
- The URL string returned by the server. This URL indicates the Internet address where the new software can be downloaded. This address is used by the MyDelivery.exe client to start the download process.
- The values of the version and the url are saved in CComMgrDlg::AuthorizeUser (MyDelivery\ComMgrDlg.cpp).

The version string returned by the server is evaluated (after step (3) above) in the function CMyDeliveryApp::CheckCompatibleVersion (MyDelivery\MyDelivery.cpp). The version string has two components: a major revision, and a minor revision component. The MyDelivery version is stored on the MyDelivery server's SQL Server database in the form major.minor.build, where a typical value is 0.9.31. A major revision component (change in "major") will require that the user update his MyDelivery installation to the newest available version. The location where the newest software is available is returned in the URL parameter. A minor revision (change in "minor" or "build") permits the client to operate the MyDelivery.exe client as is. However, the user will be prompted every 3 days to update the software.

If the version is no longer supported the user will be forced to close the client after being presented an option for immediate upgrade. If the user chooses to upgrade immediately then the download process is started by Opening the Internet URL above.

If the user did not choose to upgrade immediately then the version number obtained above is saved and the process exits. The next time the user runs MyDelivery the function CComMgrDlg::MigrateToNewVersion (MyDelivery\ComMgrDlg.cpp) is called by CComMgrDlg::OnInitDialog(). MigrateToNewVersion obtains the saved version number and will ask the user to upgrade.

## 4.0 Module: ComMgr.dll

The ComMgr.dll is responsible for all Internet communications and does the actual work of sending, downloading and sequencing attachments and its parts. All communication is carried out using SOAP over HTTP. Secure communications (HTTPS) can be used if an SSL or TLS certificate is stored on the MyDelivery server.

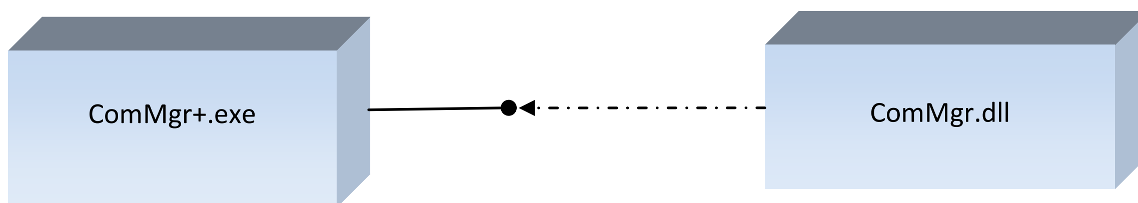
It uses several threads to accomplish this: a backup thread creates copies of any files that are necessary for the delivery; a compression thread works to create the smallest image possible for each file; the sender thread next creates a delivery plan and starts the upload to the recipient. Each attachment of 1MB is transmitted with a SHA1 hash code inserted in the payload. The receiving client uses the hash code to verify that the 1MB attachment has been received properly. During the process of sending, the sender thread may backtrack to a particular attachment that could not be verified, synchronize a previous transmission or transmit with no errors.

The downloader thread does the reverse of a sender and downloads each part whose checksum is verified with the onboard SHA1 value. The decompression thread collates these parts and pushes them to the client via the platform server.

All sending, downloading, compression, decompression and copy jobs are scheduled to its thread by a job manager that keeps track of the status of a delivery and does suitable check pointing to restart from where it left off if an unexpected event occurred.

The ComMgr.dll is controlled by a client status page (CSP) thread that must be fetched repeatedly. The frequency of fetch is controlled by the heartbeat duration indicated on the previous execution of the CSP, and it has a default period of five seconds.

The module, ComMgr.dll, communicates with the API server, ComMgr+.exe, via method calls by using an interface pointer that was provided at startup.




ComMgr.dll uses interface pointer

## 4.1 Initialization of ComMgr.dll

The ComMgr.dll is loaded at startup by ComMgr+.exe (API Server). The first function called by the API Server is `CComMgrApp::InitializeDll`. It is passed a handle of a synchronization object and a pointer to the API call back interface. The DLL saves this pointer in the application class member m_client . The InitializeDll function calls the CreateThreads function, which creates the Sender, DownloadThread, ClientStatusThread, ManagerThread, DecompressionThread and CopyThread. The sychronization object is used to synchronize access to the m_client inteface inside the DLL and must be signalled before a call to any API Server interface is placed.

```
                          Module: ComMgr\ComMgr.cpp

BOOL CComMgrApp::InitializeDll(IMyDeliveryService* pService, HANDLE hMutex)
{
      //mutex is created by ComMgr+
      m_hInterface = hMutex;

      //load address of the GetInactivity function
      GetInactivityMonitor();

      //lock service
      m_client.Attach(pService);

      CreateThreads();

      return TRUE;
}
```

```
                          Module: ComMgr\ComMgr.cpp

BOOL CComMgrApp::CreateThreads()
{
      //setup thread addresses
      PTHREAD_ADDRESS pthreads[] = {      SendThread,
                                          DownloadThread,
                                          ClientStatusThread,
                                          ManagerThread,
                                          CompressionThread,
                                          DecompressionThread,
                                          CopyThread    };

      unsigned threadID[7];
      unsigned threadstate = 0;      /* create running */
      unsigned i = 0;
      for(; i < sizeof(pthreads)/sizeof(pthreads[0]); i++)
      {
            m_threads.hWorkers[i] =
                  (HANDLE)_beginthreadex( NULL,
                  0, pthreads[i], NULL, threadstate, &threadID[i] );
      }
      :
}
```

## 4.2 JobManager

The job manager is a class (CJobManager) that is responsible for scheduling jobs to the various threads and updating any changes to a delivery such as status. It is defined in the files JobManager.h and JobManager.cpp.

The job objects managed by this class are outgoing deliveries and incoming deliveries. The outgoing deliveries are stored in the file %APPDATA%\MyDelivery\MyDeliveryID\Database\out.xml. The incoming deliveries are stored in the file %APPDATA%\MyDelivery\MyDeliveryID\Database\in.xml.

The methods of this class primarily update fields in the deliveries and do the formatting and IO required for writing and reading a delivery as it is being processed by the various threads.

## 4.3 CopyThread

The **CopyThread** (ComMgr\CopyFile.cpp) creates backup copies of selected attachments. The delivery is scheduled by calling the GetNextOutgoingJob (ComMgr\JobManager.cpp) method on the job manager. The job manager examines the list of outgoing jobs in the out.xml file and retrieves a delivery that is due for copy. The delivery that is due has a status value set to SCHEDULED_FOR_BACK_COPY. This function call also returns the list of files that need to be copied by filling in the renFiles vector .

Module: ComMgr\CopyFile.cpp

```
unsigned __stdcall CopyThread( void*  )
{
	COM_INIT thisThread;

	// wait for user to sign in
	theApp.m_events.WaitForUserAuthorization();
	theApp.m_events.WaitForCspStart();//CSP:WAIT

New_CopyJob:

	theApp.m_events.WaitForCopyJob();

	MESSAGE_OUT delivery, unused;
	bool bFlag = true;
	bool bstatusflag = true;

	if(GetFreeBytes() == 0)
		goto New_CopyJob;

	while(theApp.jobq.GetNextOutgoingJob(delivery,SCHEDULED_FOR_BACK_COPY))
	{
		:

		//prepare to copy each file
		vector<string>::iterator nameItor = delivery.renFiles.begin();
		vector<string>::iterator nameEnd = delivery.renFiles.end();
		vector<string>::iterator sourceItor =
		delivery.sourceFiles.begin();
		vector<string>::iterator sourceEnd = delivery.sourceFiles.end();
```

```
:

CAppStatus aps(PROCESSING);

while(nameItor != nameEnd)
{

      :

      string dest(_user_dir);
      dest += delivery.msgID;
      dest += "\\";
      dest += nameItor->c_str();
      dest += ".sent";

      :


      //if the source file is missing then process next copy job

      if(!bSrcFileExists)
      {
            //cancel processing on this delivery
            :
            :
            goto New_CopyJob;
      }

      _getfilesize(dest.c_str(), dstSize);

      //if zero byte source file, create empty file
      if(srcSize == 0){
            CFile file(dest.c_str(), CFile::modeCreate);
            nameItor++;
            sourceItor++;
            continue;
      }

      if(srcSize != dstSize)
      {
            //check size limitation per file
            if(within limit)
            {
                  Copy_File(sourceItor->c_str(), dest.c_str());
            }
            else
            {
                //notify over the limit
                :
            }
      }

      nameItor++;
      sourceItor++;
}

//schedule a compress job
```

```
            delivery.LastCompressed = 0;
            delivery.status = SCHEDULED_FOR_COMPRESSION;
            theApp.jobq.UpdateOutgoingJob(delivery);

            //signal compression thread
            delivery.sourceFiles.clear();
            theApp.m_events.SignalCompressionJob();

            delivery = unused;
        }
        :
}
```

### 4.4 Compression Thread

The **Compression Thread** (ComMgr\CompressFiles.cpp) compresses backup copies of any attachments. The delivery is scheduled for compression by calling the GetNextOutgoingJob (ComMgr\JobManager.cpp) method on the job manager. The job manager examines the list of outgoing jobs in the out.xml file and retrieves a delivery that is due for compression. The delivery that is due has a status value set to SCHEDULED_FOR_COMPRESSION. This function call returns the list of files that need to be copied by filling in the renFiles vector.

Not all files are compressed by the compression thread. If a file is smaller than 10KB, then compression is skipped for the file. Also, if the compressed file is bigger than the original file, it is not compressed.

Module: ComMgr\CompressFiles.cpp

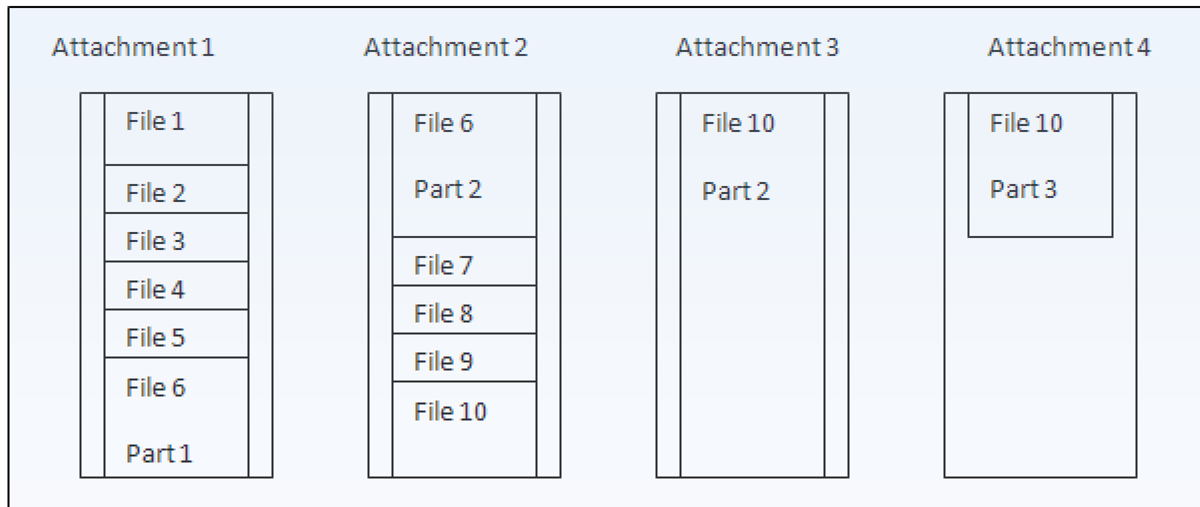
```
unsigned __stdcall CompressionThread( void*   )
{
	COM_INIT thisThread;

	:

	/* pull a job scheduled for compression */
	while(theApp.jobq.GetNextOutgoingJob(delivery, SCHEDULED_FOR_COMPRESSION))
	{
		:
		CAppStatus aps(PROCESSING);
		:

		for(;nameItor != nameEnd; nameItor++)
		{
			:

			//compress file
			if(def(path.c_str(), destfile.c_str(),
			Z_DEFAULT_COMPRESSION) != Z_OK)
				DeleteFile(destfile.c_str());
			else
			{
				:
			}

			//update the last file processed
			delivery.LastCompressed++;

			//set checkpoint
			theApp.jobq.UpdateOutgoingJob(delivery);
		}

		:
	}

	goto New_CompressionJob;
}
```

### 4.5 Delivery Layout

A delivery consists of *attachments*, *files* and *parts*. If a delivery is so large that it must be sent in separate pieces, each piece is called an "attachment". For simplicity, we will assume that MyDelivery permits only one DIME attachment per SOAP transmission. The size of the attachment stored temporarily on the server is the limiting factor, and is fixed at 1 megabyte. Here, a transmission is the same thing as an attachment. Each delivery can consist of one or more files. If a file is so large that it cannot be sent in a single attachment, it must be divided into "parts". It is possible for several files to fit into a single attachment. It is also possible for a file's parts to reside in more than one attachment.

Examples of a delivery:

1. No attachments.
2. One attachment containing one file.
3. One attachment containing two files.
4. One attachment containing three files.
5. Two attachments containing one file. The file is split into two parts; one part goes into each attachment.
6. Two attachments containing two files. The first file is split into two parts; one part goes into each attachment. The second file fits into the second attachment, along with the second part of the first file.
7. Four attachments containing ten files. The first five files fit into the first attachment. The sixth file is split into two parts: the first part fits into the first attachment and the second part fits into the second attachment. Files 7, 8 and 9 also fit into the second attachment. File 10 must be split into three parts: the first part fits into the second attachment, the second part fills up the entire third attachment, and the third part fits in the fourth attachment. See the figure below.




Delivery

Relationship between Delivery, Attachments, Files and Parts

## 4.6 Send Thread

The **Send Thread** (ComMgr\Sender.cpp) sends the delivery out to the designated recipient. The delivery is scheduled for send by calling the GetNextOutgoingJob (ComMgr\JobManager.cpp) method on the job manager. The job manager examines the list of outgoing jobs in the out.xml file and retrieves a delivery that is ready for send. The delivery that is due has a status value set to SCHEDULED_FOR_SEND. The function call GetNextOutgoingJob returns all attachment details, the subject, content and the recipient's MyDeliveryID.

The sender thread begins the process of sending out the delivery by calling PrepareDelivery(ComMgr\Layout.cpp). This function looks at the list of attachments and creates a list that holds the number of parts that make up this delivery and the attachments required for each part. More details of the layout are described in the section Delivery Layout.

The SenderThread processes a delivery by calling functions in this order

1. StartDeliveryUpload
2. UploadDeliveryHeader
3. UploadAttachments

StartDeliveryUpload (ComMgr\Sender.cpp) is the first function called when a delivery is ready to be transmitted. This returns with a delivery id.

The delivery id is then used in the next call to the server UploadDeliveryHeader1 (ComMgr\Sender.cpp). In this function call the subject, content, attachment descriptors and the computed check code for the parameters are uploaded. If the delivery contains no attachments then there is no further processing done on this delivery and the thread continues to the next delivery in the queue.

If the delivery contains attachments then UploadAttachments (ComMgr\Sender.cpp) gets called. In UploadAttachments the files are read in the same order as that was specified in the section Delivery Layout. Each attachment is formed by reading in the required file until a 1 megabyte buffer is full. The buffer is then uploaded to the server. Once this upload is complete the next attachment is similarly processed until we reach the end of the list of attachments. The last attachment may have a size less than one megabyte since there is no further data to be uploaded.

To verify the integrity of the attachments a SHA1 checksum is calculated for each attachment. When downloading the delivery, the recipient calculates the checksum for each downloaded attachment and compares it against the checksum uploaded by the sender. A correct transmission without errors will have a checksum that matches exactly.

A key behavior of the send thread is that when a delivery is being sent out a 'wait' value returned by the server is used to synchronize the sending thread with the receiving thread. When an attachment is uploaded to the server the sending thread sleeps for the number of milliseconds indicated in the wait.

If the server returns a SCHEDULED_FOR_RESYNC error code, usually returned when the recipient has received an attachment in error, then uploading further attachments is immediately stopped and the status of the delivery is set to SCHEDULED_FOR_RESYNC. The next time the sending thread begins to process this delivery it calls UpdateStatus to find a resume point for this delivery. The resume point indicates the attachment number the sender should retransmit from in his list of attachments. This mechanism ensures that errors are handled and recovered from as soon as they occur in the system.

Module: ComMgr\Sender.cpp

```
unsigned __stdcall SendThread( void*  )

{

       COM_INIT thisThread;

       // wait for user to sign in
       :

       //pull a job from the job queue
       while(theApp.jobq.GetNextOutgoingJob(delivery, SCHEDULED_FOR_SEND))
       {
              :

              // create guids, headers, size vectors and map
              theApp.sendClient.PrepareDelivery(delivery);

              CAppStatus aps(SENDING);

              :

              error = theApp.sendClient.StartDeliveryUpload(delivery);

              :
              :


              // upload the delivery header
              error = theApp.sendClient.UploadDeliveryHeader(delivery);

              :

              error = theApp.sendClient.SendDelivery(delivery, InitialValue);


              if(error == SUCCESS)
              {
                     :
                     // pop the job in the queue and process the next one
                     theApp.jobq.SetRequestStatus(delivery);

                     :
              }
       }

       goto New_SendJob;
}
```

## 4.7 DownloadThread

The **DownloadThread** (ComMgr\download.cpp) downloads the delivery from the sender. The delivery is scheduled for download by calling the GetNextIncomingJob (ComMgr\JobManager.cpp) method on the job manager. The job manager examines the list of outgoing jobs in the in.xml file and retrieves a delivery that is ready for download. The delivery that is due has a status value set to 0.

The DownloadThread processes a delivery by calling functions in this order

1. DownloadDeliveryHeader
2. DownloadDelivery

DownloadDeliveryHeader (ComMgr\download.cpp) is the first function called when a delivery is ready to be downloaded. This returns with the subject, message, list of attachments and a SHA1 checksum that was computed for this delivery by the sender. The SHA1 checksum for the received subject, message and list of attachments is recomputed and compared with the original. If there is an error the delivery is flagged as a failure and the server is informed of the failure and processing continues to the next download. The server code then assumes the job of notifying the sender when it calls UpdateStatus (ComMgr\manager.cpp). If the checksum matches and there are no attachments to download, then this delivery has finished processing. Next, the server is informed that the delivery has successfully been downloaded and is placed in the client inbox by calling the function PushNewDeliveries (ComMgr\JobManager.cpp).

If the delivery being processed has a list of attachments, then execution continues to the function DownloadDelivery (ComMgr\download.cpp). Here, the number of attachments that are to be downloaded is computed and the attachments themselves are downloaded serially. For each attachment downloaded the function GetSha1HashStringFromFileContent(ComMgr\download.cpp) is called and the checksum for the attachment is verified with the checksum sent by the sender. If the checksum matches then processing continues to the next attachment. If an error occurs then the server is notified of the failure via DeliveryStatus and the attachment number of the failure is provided.

On a failure to verify a downloaded attachment, resynchronization occurs. The server notifies the sender about the failure by returning the error code 6 and requests a retransmission from the failed attachment number. The sender backtracks to the attachment number after calling UpdateStatus and starts retransmitting the delivery from the failure point.

To account for different connection speeds by the sender and receiver a Wait interval is a parameter in the DownloadAttachment (ComMgr\download.cpp) and DownloadDeliveryHeader (ComMgr\download.cpp) functions. This allows the receiver to synchronize with the sender within a distance of one attachment. What this means is that the receiver is always one attachment behind the sender. If the receiver attempts to speed up it receives a wait interval from the server and proceeds to sleep for the indicated interval thereby allowing the sender to upload the attachment before the next attempt to download is made. If the sender has not yet finished uploading the next attachment then the Wait interval increases thereby allowing the sender more time to finish his upload. This process continues with the wait interval incrementally increasing or decreasing until the optimum rate of data transfer between the sender and receiver is achieved.

Module: ComMgr\download.cpp

```
unsigned __stdcall DownloadThread( void*  )
{
      COM_INIT thisThread;

      // wait for user authorization
      theApp.m_events.WaitForUserAuthorization();
      theApp.m_events.WaitForCspStart();//CSP:WAIT

      // start a new download
New_DownloadJob:

      :

      //pull first job from the delivery queue
      while(theApp.jobq.GetNextIncomingJob(delivery))
      {
            InitialValue = delivery;

Begin_DownloadHeader:

            CAppStatus aps(DOWNLOADING);

            //recover from checkpoint
            :

            error = theApp.dlClient.DownloadDeliveryHeader(delivery);

            :

            //download attachments
            error = theApp.dlClient.DownloadDelivery(delivery, InitialValue,
aps);

            if(error == SUCCESS)
            {
                  :
                  theApp.m_events.SignalDecompressionJob();
                  :
            }
      }

      goto New_DownloadJob;
}
```

## 4.8 Decompression Thread

The decompression thread (ComMgr\decompress.cpp) is the last thread to process a delivery before the delivery is inserted into the inbox of the recipient. The decompression thread is responsible for the collating the various downloaded files into a complete delivery before it is passed to the client for display.

The decompression thread periodically queries the job manager for a job by calling the function GetNextInboundJob (ComMgr\JobManager.cpp). The function call returns after calling the function ProcessHeader (ComMgr\download.cpp). The process header function loads all the details of the delivery such as the number of attachments, the number of parts in each attachment and size information of each file that was attached by the sender. It does this by reading the header that was saved by the DownloadDeliveryHeader (ComMgr\download.cpp) function and processing its contents. The result of the ProcessHeader function is a DeliveryMap list that describes each attachment.

The collation processing is done in the function ProcessBinFiles (ComMgr\decompress.cpp). The created delivery map is iterated to collate downloaded attachments by calling ProcessAttachment (ComMgr\decompress.cpp) for each entry.

The ProcessAttachment function processes the attachment and creates the list of files as they were sent by the sender. Each file is created and saved by progressively reading the downloaded attachments that make up the file and writing the part of the attachment buffer that makes up the file. For instance, if there were 2 files that are 3.5 MB and 1.5 MB when they were sent, then there would be at least 5 attachments that are 1MB in size that would complete the delivery. In this case the first 3 attachments would have 1 part of 1MB each and they would be first written, in the correct order, before the 4th attachment of 2 parts (0.5 MB part belonging to the first file and 0.5 MB part that would belong to the second file) is processed. In our example, only the first part of the $4^{th}$ attachment 0.5MB would be written to the end of the file to arrive at a complete file. The second file is next created and the remaining data of 0.5MB is written to disk before the next attachment of 1MB is appended to it. The processing continues in this fashion, part by part, attachment by attachment until the list of the files that are part of the delivery is fully assembled.

The assembled files are then required to be decompressed before being inserted in the inbox of the recipient. The decompression is a trivial matter now that we have a complete compressed file. The list of compressed files is iterated and each file is decompressed to arrive at the final file that is ready for presentation.

Module: ComMgr\decompress.cpp

```
unsigned __stdcall DecompressionThread( void*  )
{
      COM_INIT thisThread;

      // wait for user to sign in
      :

      /* pull a decompression job from the job queue */
      while(theApp.jobq.GetNextInboundJob(delivery,
SCHEDULED_FOR_DECOMPRESSION))
      {
            :

            CAppStatus aps(PROCESSING);

            :

            //process the downloaded attachments
            int error = ProcessBinFiles(delivery);

            :

            //get a pointer to each renamed file
            vector<string>::iterator nameItor = delivery.renFiles.begin();
            vector<string>::iterator nameEnd = delivery.renFiles.end();
            for(;nameItor != nameEnd; nameItor++)
            {
                  :

                  if(inf(path.c_str(), destfile.c_str()) == Z_OK)
                        DeleteFile(path.c_str());
                  :

                  :

            }

            :

            theApp.jobq.PushNewDeliveries();
      }

      goto New_DecompressionJob;
}
```

### 4.9 ClientStatusPage Thread

The client status page (CSP) thread (ComMgr\ClientStatusPage.cpp) is the key driver for deliveries in the MyDelivery system. The core function of the CSP thread is to

1. Keep the user logged into the MyDelivery system.
2. Detect when a delivery is ready for downloading and kick start the manager thread.
3. Detect other useful information about the server

The client status page (CSP) is the first thread to run in MyDelivery after initialization. The thread kick starts the manager thread and waits for the manager thread to finish calling UpdateStatus. It then sits in a loop and calls the ClientStatusPage web service after waiting for the interval that is indicated in the last fetch. It reads the downloaded data and calls UpdateStatus if a delivery is ready for download.

Every 10 minutes the CSP thread checks to see if the free disk space available has changed by greater than 10%. If the change has occurred then an UpdateStatus is triggered and the server is informed of the total disk space now available.

Module: ComMgr\ClientStatusPage.cpp

```
unsigned __stdcall ClientStatusThread( void*   )
{
      COM_INIT thisThread;
      :
      while(1)
      {
            :
3                 //fetch csp
                  theApp.GetClientStatusPage();

            :

            //on WAIT_TIMEOUT, signal an update on state change.
            if(csp != theApp.m_csp)
            {
                  //signal manager to call update status
                  theApp.m_events.SignalUpdateStatus();//UPDATE:ON
            }
            else if(tenMinutes <= 0)
            {
                  tenMinutes = 600000;

                  if(TenPercentChange(freeDiskSpace))
                  {
                        //signal manager to call update status
                        theApp.m_events.SignalUpdateStatus();//UPDATE:ON
                        freeDiskSpace = GetFreeBytes();
                  }
```

```
			}

			:

			Sleep(msWaitTime);
		}
		return 0;
}
```

### 4.10 Manager Thread

The main purpose of the manager thread (ComMgr\manager.cpp) is to call UpdateStatus when necessary. The manager thread is instructed to call UpdateStatus by the CSP thread.

The CSP thread as mentioned earlier repeatedly fetches a page on the server that describes the heartbeat duration and a status code. The status code indicates that a delivery is ready to be downloaded. When this bit is set the manager thread is started and it calls the UpdateStatus (ComMgr\manager.cpp) function.

The parameters sent in the UpdateStatus function are the list of delivery id's whose status needs to be known. If there are no deliveries whose status needs to be known then the parameters are not set. When UpdateStatus returns it lists the number of deliveries that are ready to be downloaded by the recipient. The manager thread then inserts these deliveries in the file in.xml.

The manager thread also performs the role of updating the status of the deliveries that were requested and related processing such as writing these changes to the file out.xml and in.xml. The relevant code is listed in the functions ReadDeliveryStatusResponse (ComMgr\Sender.cpp) and DeliveriesReadyForDownload (ComMgr\download.cpp).

Module: ComMgr\manager.cpp

```
unsigned __stdcall ManagerThread( void*   )
{
	COM_INIT thisThread;
	HRESULT hr = S_OK;

	//wait for user to sign in
	theApp.m_events.WaitForUserAuthorization();

	DWORD dwUpdateContext;

	while(1)
	{
	   dwUpdateContext = 0;

		theApp.m_events.WaitForUpdateStatus();//UPDATE:WAIT

		:
```

```
            //call update status
            error = theApp.UpdateStatus(..);

            :

            //start CSP
            theApp.m_events.SignalCspStart();//CSP:ON

            //clear for download
            if(DeliveriesReadyForDownload)
            {
                  //download delivery
                  theApp.m_events.SignalNewDownload();
            }

      }

      return 0;
}
```

### 4.11 Additional Files used by the Client

- **Common.xml** This file is located in the %APPDATA%\MyDelivery folder and used by the MyDelivery.exe client. It holds the settings common to the application. They are:
  - The last logged in user (<lastlogin>)
  - Version number(<newversion>)
  - New software url (<swurl>)
  - Timestamp(<timestamp>)

- **User.xml** This file is located in the %APPDATA%\MyDelivery*MyDeliveryID*\Database\ folder and is used by the MyDelivery.exe client. It holds information specific to each user. . They are:
  - Sort settings for each folder (<COL>),
  - Number of sort settings (<NUM_SORT_SETTINGS>),
  - Window position (<WindowPosition>),
  - View position(<WindowPositionCY>),
  - Control bar visibility (<CONTROLBARS>),
  - Delete on exit < DELETE_ON_EXIT >,
  - Last path where attachments were saved to (<SAVE_ATT_PATH >) and the folder hierarchy (<SAVE_ATT_TREEPATH >),
  - Last path where attachments were selected from (<ADD_ATT_PATH >).
  - Signature (<INCLUDE_SIGNATURE >). Specifies if a signature must be included when a delivery is sent.
  - Readtime (<RDTM>). This is the number of seconds that must elapse after a use has selected a delivery before it is marked as read.
  - File extract option (<FILE_EXTRACT_OPTION >). This saves the user selection when he chooses to extract the files with their folder names.

user.xml – sample file


```
<?xml version="1.0" ?>
- <MyDelivery>
<NewDeliveryAttachmentFolder />
- <WindowPosition>
   <top>242</top>
   <left>242</left>
   <bottom>991</bottom>
   <right>1502</right>
</WindowPosition>
- <WindowPositionCY>
   <left />
   <right />
   <width />
</WindowPositionCY>
- <INBOX>
   <COL1>0</COL1>
   <COL2>0</COL2>
   <COL3>0</COL3>
</INBOX>
- <OUTBOX>
   <COL1>0</COL1>
   <COL2>0</COL2>
   <COL3>0</COL3>
   <COL4>0</COL4>
</OUTBOX>
- <SENTITEMS>
   <COL1>0</COL1>
   <COL2>0</COL2>
   <COL3>0</COL3>
</SENTITEMS>
- <SORTSETTINGS>
<NUM_SORT_SETTINGS>0</NUM_SORT_SETTINGS>
</SORTSETTINGS>
- <ADDRBOOK_SORT_SETTINGS>
<COLUMN_NAME />
<SORT_ORDER>0</SORT_ORDER>
</ADDRBOOK_SORT_SETTINGS>
- <CONTROLBARS>
   <TOOLBAR>1</TOOLBAR>
   <STATUSBAR>1</STATUSBAR>
</CONTROLBARS>
<DELETE_ON_EXIT>0</DELETE_ON_EXIT>
<RDTM>1281</RDTM>
<INCLUDE_SIGNATURE>0</INCLUDE_SIGNATURE>
<FILE_EXTRACT_OPTION>DONOTUSE</FILE_EXTRACT_OPTION>
<ADD_ATT_PATH>C:\Documents and Settings\glingappa\Desktop</ADD_ATT_PATH>
<SAVE_ATT_TREEPATH />
<SAVE_ATT_PATH />
</MyDelivery>
```

**Files used for sending deliveries:**

- **.cmp** files are compressed files of attached files and they are stored in the GUID folder that is created when a delivery is placed in your outbox. They are temporary file and are deleted after the delivery has been sent. The location of this file is %APPDATA%\MyDelivery*MyDeliveryID*\Database\{GUID}\
- **.sent** files are copies of attached files and saved in the same folder as .cmp files. They are not deleted after the delivery is sent but only deleted when a delivery is deleted from the deleted items folder
- **attdetails.txt** This file describes the hierarchical order of selected attachments to the copy and compression threads. The CJobManager::LoadTreeInfo (ComMgr\JobManager.cpp) handles the parsing of this file. It is located in the same folder as the .sent files.

All the information that represents the files is written just in case the ComMgr.dll needs it for some future flexibility.

```
node_id,
parent_id,
ftCreationTime,
ftLastAccessTime,
ftLastWriteTime,
dwFileAttributes,
cAlternateFileName,
cFileName,
nFileSizeHigh,
nFileSizeLow,
OrigPath,
orig_nodeid
```

attdetails.txt – sample file

0*-1*0*0*0*16**MyDelivery*0*0**36831232
1*0*129040685540432531*129195444907062224*129195444907062224*16*STATUS~1*Status reports*0*0*C:*-1
2*1*129040685657617781*129195444755504574*129163365772030226*16**09*0*0*C:\Statu s reports*0
3*2*129109993381969911*129195445022214789*129036533426554005*128*USER-G~1.PDF*user-guide-a4.pdf*0*3131904*C:\Status reports\09*1214559
4*2*129109993381969911*129195445032058224*129055450983648709*128*USERGU~1.PDF* UserGuideInsight4.1.pdf*0*10838679*C:\Status reports\09*1214559
5*1*129095161542641040*129195444755504574*129145226566789354*16**10*0*0*C:\Statu s reports*0
6*5*129119327318840796*129195444769722869*129145226590226104*16**feb*0*0*C:\Statu s reports\10*1214559
7*6*129119327401807422*129158344310000000*129119343660453690*128*LINGAP~1.DOC*L ingappa.docx*0*15634*C:\Status reports\10\feb*1213227
8*5*129095161663574670*129195444769722869*129119327403369882*16**jan*0*0*C:\Statu s reports\10*1214559

9*8*129095161685605215*129158344310000000*129095213992681335*128*LINGAP~1.DOC*Lingappa.docx*0*17740*C:\Status reports\10\jan*1213227
10*5*129145226566789354*129195444769566624*129145233330479159*16**March*0*0*C:\Status reports\10*1214559
11*10*129145226588351164*129158344310000000*129145231789434724*128*LINGAP~1.DOC*Lingappa.docx*0*16037*C:\Status reports\10\March*1213227
12*5*129109993393222719*129158344310000000*129036533426554005*128*USER-G~1.PDF*user-guide-a4.pdf*0*3131904*C:\Status reports\10*1214559
13*5*129109993393222719*129158344310000000*129055450983648709*128*USERGU~1.PDF*UserGuideInsight4.1.pdf*0*10838679*C:\Status reports\10*1214559

- **attachments.txt** This file describes the hierarchical order of selected attachments to the recipient. It is sent by the UploadDeliveryHeader1 function after compression. It is stored in the same folder as .sent files. Not all the fields of the attdetails.txt are required for the recipient. The fields in the attachments.txt are listed below:

  node_id,
  parent_id,
  wfd.wfd.ftLastWriteTime,
  wfd.wfd.dwFileAttributes,
  wfd.wfd.cFileName,
  ui64

  The node_id is a number that is unique to this node and the parent_id is the node_id of the parent node. The rest of the fields are selected from the WIN32_FIND_DATA structure defined in WinBase.h (included in Windows.h). The ui64 is the file size.

  0*-1*0*16*MyDelivery*0
  1*0*129195444907062224*16*Status reports*0
  2*1*129163365772030226*16*09*0
  3*2*129036533426554005*128*user-guide-a4.pdf*3131904
  4*2*129055450983648709*128*UserGuideInsight4.1.pdf*10838679
  5*1*129145226566789354*16*10*0
  6*5*129145226590226104*16*feb*0
  7*6*129119343660453690*128*Lingappa.docx*15634
  8*5*129119327403369882*16*jan*0
  9*8*129095213992681335*128*Lingappa.docx*17740
  10*5*129145233330479159*16*March*0
  11*10*129145231789434724*128*Lingappa.docx*16037
  12*5*129036533426554005*128*user-guide-a4.pdf*3131904
  13*5*129055450983648709*128*UserGuideInsight4.1.pdf*10838679

**Files used for receiving deliveries:**

- **attachments.txt** This file describes the hierarchical order of selected attachments to the recipient. It is downloaded by the DownloadDeliveryHeader1 function and decompressed. It is

stored in the %APPDATA%\MyDelivery*MyDeliveryID*\Database\number\ folder. The number is randomly generated and is associated with the delivery.

- **.bin** This file is the downloaded attachment and is stored in the same folder as attachments.txt. It is an intermediate file that will be merged with other such files to form an attachment. The merge process is implemented by ProcessBinFiles (ComMgr\decompress.cpp).
- **header.txt.** It is downloaded by the DownloadDeliveryHeader1 function and contains header information sent by the UploadDeliveryHeader1 function. It contains the file sizes and the part information. It is stored in the same folder as the .bin files.

header.txt – sample file


```
<?xmlversion="1.0"?>
<MyDelivery>
<Files><NUM_ENTRIES>7</NUM_ENTRIES>
<COMPRESSION>fw==</COMPRESSION>
<FILEINFO>3,2696363,3:4,8618748,9:7,12648,1:9,14761,1:11,13054,1:12,2696363,4:13,8618748,9<
/FILEINFO>
</Files>
</MyDelivery>
```

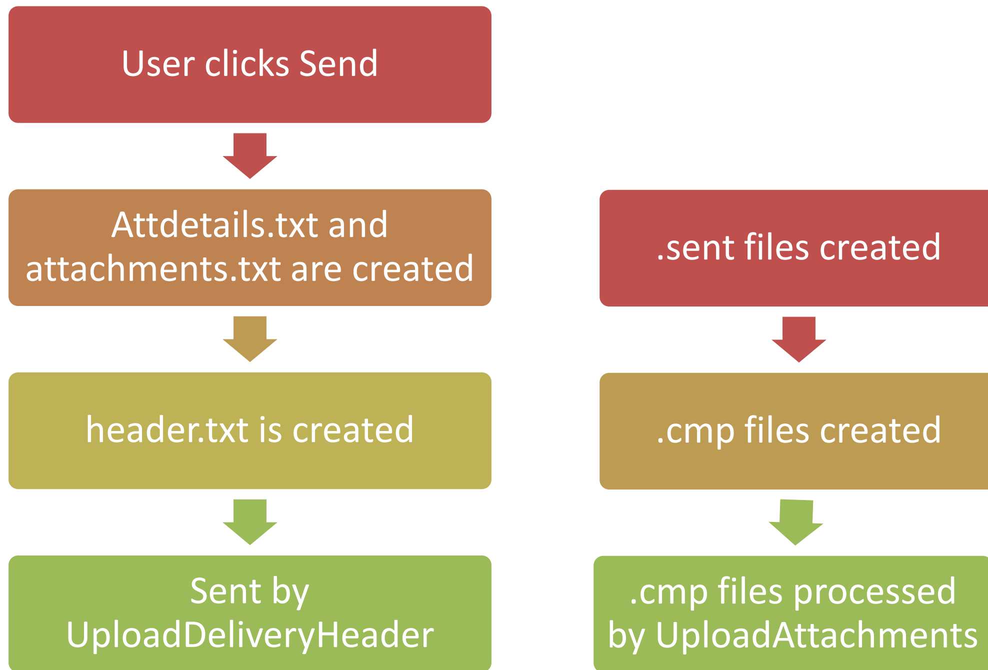

The header information is contained in the <COMPRESSION> and <FILEINFO> tags. The <COMPRESSION> tag holds a base 64 encoded value that describes which file is compressed. This is a bit string that has a value 1 if compressed 0 otherwise before being encoded. The encoding is done in the function CyoEncode::Base64Encode (ComMgr\Layout.cpp).

The <FILEINFO> tag specifies the files, file sizes and the parts. It is separated by a colon. The first value is a number that indicates which node id this file corresponds to. The second is the file size, the third value is the number of parts this file is split into. This information is necessary for the downloaded to assemble the files.

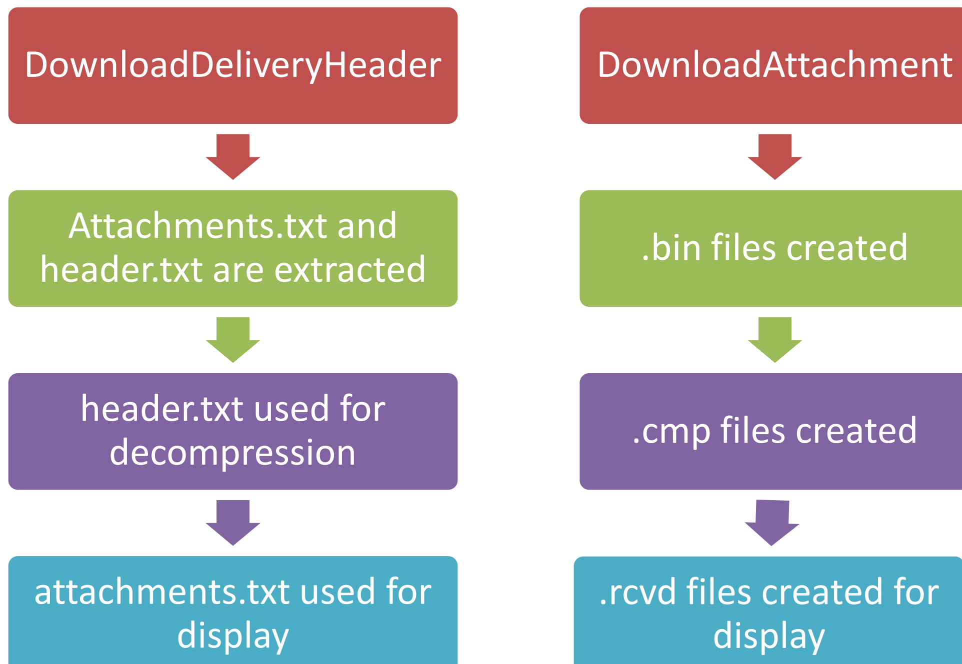
- **.cmp** files. These files are created after the ProcessBinFiles function has finished execution. They are compressed files of the original attachments and are decompressed by the decompression thread. They are intermediate files and are deleted after decompression. It is stored in the same folder as the .bin files.
- **.rcvd** files. These files are created by the decompressed thread after decompressing the .cmp files. They are files that were sent by the sender. It is only deleted after the user deletes the delivery from the deleted items folder. It is stored in the same folder as the .bin files.

The following illustrations show the order in which these files are created for sending and receiving.

**Order of File Creation While Sending**




**Order of File Creation While Receiving**



## 5.0 SOAP Functions Handled by ComMgr.dll

The SOAP function definitions are located in the file WSDL.cpp. These serve as definitions for the web services that are available on the server. This section describes how the client uses each of the SOAP functions. Please refer to the accompanying document, "The MyDelivery Server", for a detailed specification of each SOAP function.

### 5.1 GetVersion

The server function GetVersion is used by ComMgr.dll to verify the version of the installed software. This is the first function called by the ComMgr.dll and does not require prior authorization. This function is implemented by CComMgrApp::GetVersion (..)(ComMgr\ComMgr.cpp). This call is made before the authorization function Initialize is called.

The parameters to the function are:

- The **Version** string returned by the server. The version string is evaluated in the function CMyDeliveryApp::CheckCompatibleVersion (MyDelivery\MyDelivery.cpp). The version string has two components a major revision and a minor revision component. A major revision component will require that the user update his MyDelivery installation to the newest available. The location where the newest software is available is returned in the next parameter. A minor revision permits the client to operate the MyDelivery.exe client as is. However the user will be prompted every 3 days to update the software.
- The **NewSoftwareURL** string returned by the server. This URL indicates the Internet address where the new software can be downloaded. This address is used by the MyDelivery.exe client to start the download process.

### 5.2 ClientStatusPage (CSP)

The server function CSP is used by ComMgr.dll to fetch the client status page. This function is implemented by CComMgrApp::GetClientStatusPage(..)(ComMgr\ClientStatusPage.cpp). This function is used by the server to inform the client when a delivery is ready for download. It also is used by the server to determine when a client is online. Typically a client will execute CSP every five seconds, or the HeartBeat interval, which is returned to the client when it executes CSP.

The CSP takes a single parameter which is a **Key** value. This key value is returned by the server when the client calls Initialize.

### 5.3 Initialize

The server function Initialize is called in the method CComMgrApp::AuthorizeUser(…) (ComMgr\ComMgr.cpp) . This is responsible for the Authorization of a user and signing into MyDelivery. This is initiated at the client when the user logs in and is the second function called on the server (GetVersion is the first).

The parameters to the function are:

1. MyDeliveryID and MyDelivery password, these are used by the server to verify the user's credentials and are the values the user provided in the login dialog box. These are sent in the **UserID** parameter, concatenated together as MyDeliveryID/password.
2. The second parameter to MyDelivery is the device capacities parameter, **DeviceCaps**, that is an XML string that contains two tags. The first is the free disk space available on the machine and the second contains the largest file size that can be handled by the operating system (on FAT32 file system this maxes out at 4GB and is greater than that on NTFS file systems.)
3. The **ErrorStatus** parameter is a string returned by the server and it indicates the reason for any error generated by this call. This is used to provide feedback to the user like (an incorrect user name password combination)
4. The **UserName** (First And Last Name) parameter is returned by the server. This is the name used by the server when the user first registered with the system. This may be essential for any error reporting or web master messages that are to be placed in the inbox.
5. The **AddressBook** parameter is returned by the server if successfully authorized. This holds the compressed addressbook of the user and is returned to the client or the GetAddressBook API function after decompression.
6. The **Key** ID parameter is cached locally in the CUserInfo class. This value is used in all subsequent calls to web services including the CSP. This serves to identify the caller on the server.

## 5.4 ChangePassword

The web service ChangePassword is used by ComMgr.dll to change the current password for the user. It is implemented in the function CComMgrApp::ChangePassword(..) (ComMgr\ComMgr.cpp).

The parameters to the function are:

1. The **Key** ID parameter that was returned by the server when Initialize was called. This serves to identify the caller on the server.
2. The **OldPassword** parameter is used to verify the caller.
3. The **NewPassword** parameter is the new value that will be replacing the old password.
4. The ErrorStatus parameter, this is a string returned by the server and it indicates the reason for any error generated by this call. This is used to provide feedback to the user like (an incorrect user name password combination)

## 5.5 UploadUserInformation

The web service UploadUserInformation is used by ComMgr.dll to upload any changes to the current addressbook of the user. It is implemented in the function CComMgrApp:: UploadUserInformation(..) (ComMgr\ComMgr.cpp). This method is called exclusively by the Mydelivery.exe client. It is not available for the platform API.

The parameters to the function are:

1. The **Key** ID parameter that was returned by the server when Initialize was called. This serves to identify the caller on the server.
2. The compressed **AddressBook** of the user that replaces the addressbook stored on the server. Please note that the entire addressbook is uploaded each time this method is called.

3. The **ErrorStatus** parameter, this is a string returned by the server and it indicates the reason for any error generated by this call. This is used to provide feedback to the user like (a offline recipient)

## 5.6 CheckRecipient

The web service CheckRecipient is used by ComMgr.dll to verify the recipient's MyDeliveryID. It is implemented in the function CComMgrApp:: CheckRecipient(..) (ComMgr\ComMgr.cpp). This method is called both by the Mydelivery.exe client and the ComMgr+.exe API. This method is called when the user attempts to send a delivery to a recipient.

The parameters to the function are:

1. The **Key** ID parameter that was returned by the server when Initialize was called. This serves to identify the caller on the server.
2. The **ReceiverHandle** is the MyDeliveryID of the intended recipient, and it is used to verify that we have a valid recipient who is registered in the system.
3. The **TotalSize** is used to verify if the recipient has enough disk space to receive all the attachments of this delivery.
4. The **LargestFile** is used to verify that the largest file size can be handled. This is necessary since some systems cannot handle file sizes > 4GB.
5. The **ErrorStatus** parameter is a string returned by the server and it indicates the reason for any error generated by this call. This is used to provide feedback to the user like (insufficient disk space.)

## 5.7 DeliveryStatus

The web service DeliveryStatus is used by ComMgr.dll to inform the server if a delivery was received with success or failure. It is implemented in the function DeliveryStatus(..) (ComMgr\decompress.cpp). This method is never directly called by the Mydelivery.exe client and the ComMgr+.exe API. This method is called automatically when the recipient successfully downloads a delivery.

The parameters to the function are:

1. The **Key** ID parameter that was returned by the server when Initialize was called. This serves to identify the caller on the server.
2. The **DeliveryID** parameter is used to identify this delivery to the server.
3. The **DeliveryResult** parameter is used to inform the server of a success or failure.
4. The **ErrorStatus** parameter, this is a string returned by the server and it indicates the reason for any error generated by this call.

## 5.8 IsDeliveryValid

The web service IsDeliveryValid is used by ComMgr.dll to verify if a delivery is valid. The verification prevents the ComMgr.dll from unnecessarily downloading deliveries that have been terminated or which are no longer valid for some reason determined by the server. It is implemented in the function CDownload::IsDeliveryValid(..) (ComMgr\download.cpp) and in CUploadJobs::IsDeliveryValid(..) (ComMgr\sender.cpp). This method is never directly called by the Mydelivery.exe client and the ComMgr+.exe API. This method is called automatically before the recipient starts downloading a delivery or resuming uploading a delivery.

The parameters to the function are:

1. The **Key** ID parameter that was returned by the server when Initialize was called. This serves to identify the caller on the server.
2. The **DeliveryID** parameter is used to identify this delivery to the server.
3. The **DeliveryValid** parameter is used to determine whether a delivery is valid. If the delivery is not valid then the download thread / send thread stops further processing of this delivery and removes it from the download queue / send queue.
4. The **DeliveryTerminated** is used to determine whether a delivery has been terminated. If this delivery has been terminated then the download thread / send thread stops further processing of this delivery and removes it from the download queue / send queue.
5. The **ErrorStatus** parameter is a string returned by the server and it indicates the reason for any error generated by this call.

## 5.9 DownloadAttachment

The web service DownloadAttachment is used by ComMgr.dll to download an attachment. It is implemented in the function CDownload::DownloadAttachment(..) (ComMgr\download.cpp). This method is never directly called by the Mydelivery.exe client and the ComMgr+.exe API. This method is called repeatedly by the download thread to download all attachments that are part of a delivery.

The parameters to the function are:

1. The **Key** ID parameter that was returned by the server when Initialize was called. This serves to identify the caller on the server.
2. The **DeliveryID** parameter is used to identify this delivery to the server.
3. The **Attachment** parameter is the number of the attachment to be downloaded from the server. This is incremented each time by the client to progressively download all attachments that make up this delivery. The loop that increments this attachment number is located in the function CDownload::DownloadDelivery (ComMgr\download.cpp).
4. The **MaxAttachmentReady** is used to request the attachment number of the next attachment that is available for download.
5. The **Wait** parameter is used to request the server for a time interval in milliseconds that the client should wait before requesting the next attachment. This is used by the server to synchronize the sending client with the receiving client. Not doing so would mean that we are attempting to download at a rate that is faster than the rate at which the sender is capable of sending and would result in unnecessary consumption of bandwidth.
6. The **ErrorStatus** parameter is a string returned by the server and it indicates the reason for any error generated by this call.

## 5.10 DownloadDeliveryHeader1

The web service DownloadDeliveryHeader1 is used by ComMgr.dll to download the subject, message and the list of files in the delivery. It is implemented in the function CDownload::DownloadDeliveryHeader(..) (ComMgr\download.cpp). This method is never directly called

by the Mydelivery.exe client and the ComMgr+.exe API. This method is called once by the download thread before it attempts to download the rest of the delivery.

The parameters to the function are:

1. The **Key** ID parameter that was returned by the server when Initialize was called. This serves to identify the caller on the server.
2. The **DeliveryID** is used to identify this delivery to the server.
3. The **ErrorStatus** parameter is a string returned by the server and it indicates the reason for any error generated by this call.
4. The **DeliveryTerminated** parameter is used to request the server if a delivery has been terminated. If this delivery has been terminated then the download thread stops further processing of this delivery and removes it from the download queue.
5. The **SenderFirstName** parameter is used by the client when inserting this delivery into the inbox
6. The **SenderLastName** parameter is used by the client when inserting this delivery into the inbox.
7. The **SenderHandle** parameter is used by the client when inserting this delivery into the inbox.
8. The **Subject** is used by the client to retrieve the subject
9. The **TextMessage** parameter is used by the client to retrieve the text message portion of the delivery.
10. The **Files** parameter is used by the client to know the details of each attachment such as the file size.
11. The **Tree** parameter is used by the client to know the hierarchical order of the attachments
12. The **CheckCode** parameter is used by the client to determine if all the received parameters from this function call were received without error. This is computed by concatenating the received strings and then deriving a combined hash code for all the parameters (Subject, TextMessage, Files and Tree). The computed hash value should match the CheckCode.
13. The **Wait** parameter is used to request the server for a time interval in milliseconds that the client should wait. This is used by the client to skip processing this delivery as it is not available yet. If a wait value is received then the next delivery in the download queue is processed.

## 5.11 UpdateStatus

The web service UpdateStatus is used by ComMgr.dll to find the status of deliveries that have been sent so far and the list of deliveries that are ready for downloading. It is also used by the ComMgr.dll to update the server when the percentage of the free disk space available has changed by greater than 10%. It is implemented in the function CComMgrApp::UpdateStatus(..) (ComMgr\manager.cpp). This method is never directly called by the Mydelivery.exe client and the ComMgr+.exe API.

The parameters to the function are:

1. The **Key** ID parameter that was returned by the server when Initialize was called. This serves to identify the caller on the server.
2. The **DeliveryStatusRequest** parameter contains the list of deliveries whose status needs to be known by the sender. This is an XML string that holds the result for each of the deliveries that were requested. The result is then used by the sender to move any successful deliveries from the outbox to the sent items folder. The move occurs only if the status returned by the server is a success. The XML string also contains a list of deliveries that are ready for downloading. This

return string is parsed and the delivery ids are inserted into the in.xml file. After this is inserted into the in.xml file the download thread starts downloading the deliveries one after another.
3. **FreeDiskSpace** parameter is used to inform the server of the free space on the client
4. The **ErrorStatus** parameter, this is a string returned by the server and it indicates the reason for any error generated by this call.

### 5.12 UploadAttachment

The web service UploadAttachment is used by ComMgr.dll to upload an attachment. It is implemented in the function CUploadJobs:: UploadAttachments(..) (ComMgr\sender.cpp). This method is never directly called by the Mydelivery.exe client and the ComMgr+.exe API. This method is called repeatedly by the sender thread to upload all attachments that are part of a delivery.

The parameters to the function are:

1. The **Key** ID parameter that was returned by the server when Initialize was called. This serves to identify the caller on the server.
2. The **DeliveryID** is used to identify this delivery to the server.
3. The **attachment data (up to 1 MB)**, which is sent as a SOAP DIME attachment.
4. The derived **CheckCode** for the attachment
5. The **Attachment** is a parameter representing the particular attachment for the server. This is incremented each time by the client to progressively upload all attachments that make up this delivery. The loop that increments this attachment number is formed by a goto call to the label Upload_NextAttachment. CUploadJobs:: UploadAttachments (ComMgr\sender.cpp).
6. The **Wait** parameter is used to request the server for a time interval in milliseconds that the client should wait before uploading the next attachment. This is used by the client to control the speed of the upload. Not doing so would mean that we are attempting to upload at a rate that is faster than the rate at which the recipient is capable of sending and would result in unnecessary consumption of bandwidth.

### 5.13 UploadDeliveryHeader1

The web service UploadDeliveryHeader1 is used by ComMgr.dll to upload the subject, message and the list of files in the delivery. It is implemented in the function CUploadJobs::UploadDeliveryHeader(..) (ComMgr\sender.cpp). This method is never directly called by the Mydelivery.exe client and the ComMgr+.exe API. This method is called once by the sender thread before it attempts to send the rest of the delivery.

The parameters to the function are:

1. The **Key** ID parameter that was returned by the server when Initialize was called. This serves to identify the caller on the server.
2. The **DeliveryID** parameter is used to identify this delivery to the server.
3. The **ErrorStatus** parameter is a string returned by the server and it indicates the reason for any error generated by this call.
4. The **Subject** parameter is used by the client to send the subject
5. The **TextMessage** parameter is used by the client to send the message.
6. The **Files** parameter is used to send the details of each attachment such as the file size.
7. The **Tree** parameter is used to send the hierarchical order of the attachments.

8. The **CheckCode** parameter is a SHA1 hash code calculated for the concatenated value of all the parameters being sent (Subject, TextMessage, Files and Tree).

## 5.14 StartDeliveryUpload

The web service StartDeliveryUpload is used by ComMgr.dll to initiate a new delivery for a recipient. It is implemented in the function CUploadJobs:: StartDeliveryUpload(..) (ComMgr\sender.cpp). This method is never directly called by the Mydelivery.exe client and the ComMgr+.exe API. This method is called once by the sender thread before CUploadJobs:: UploadDeliveryHeader(..) (ComMgr\sender.cpp) is called.

The parameters to the function are:

1. The **Key** ID parameter that was returned by the server when Initialize was called. This serves to identify the caller on the server.
2. The **DeliveryID** parameter returned by the server is used to identify this delivery. It is saved in the out.xml file for this delivery.
3. The **ErrorStatus** parameter is a string returned by the server and it indicates the reason for any error generated by this call.
4. The **TotalSize** parameter is used by the server to verify if the recipient has enough disk space to receive all the attachments of this delivery.
5. The **LargestFile** parameter is used by the server to verify that the largest file size can be handled. This is necessary since some systems cannot handle file sizes > 4GB.

## 5.15 TerminateDelivery

The web service TerminateDelivery is used by ComMgr.dll to cancel processing and terminate one or more deliveries. It is implemented in the function CUploadJobs::TerminateDeliveryOnServer(..) (ComMgr\sender.cpp). This method is directly called by the Mydelivery.exe client alone.

The parameters to the function are:

1. The **Key** ID parameter that was returned by the server when Initialize was called. This serves to identify the caller on the server.
2. The **DeliveryID** parameter is a list of deliveries that need to be terminated. A delivery will be in this list if the user deletes it through the client user interface.

## 5.16 RequestToUpload

The web service RequestToUpload is used by ComMgr.dll to request permission before the next attachment is uploaded. This function is called before UploadAttachment is called and is a way the ComMgr.dll determines the server is ready to accept the delivery before the attachment is sent. It is implemented in the function CUploadJobs::RequestUpload(..) (ComMgr\sender.cpp). This method is never directly called by the Mydelivery.exe client or the ComMgr+.exe API.

The parameters to the function are:

1. The **Key** ID parameter that was returned by the server when Initialize was called. This serves to identify the caller on the server.
2. The **DeliveryID** parameter is used to identify this delivery to the server.
3. The **Attachment** parameter specifies a particular attachment for which the permission is requested.
4. The **Wait** parameter is used to request the server for a time interval in milliseconds that the client should wait before uploading the next attachment. This is used by the server to synchronize the sending client with the recipient. Not waiting would mean that we are attempting to upload at a rate that is faster than the rate at which the recipient is capable of sending and would result in unnecessary consumption of bandwidth. The client uses the return value to pause the sender thread execution for the indicated milliseconds before making a new RequestToUpload call. The main functional idea here is to keep the rates of send thread in pace with the recipient's download thread.

## 6.0 Sample Procedures Handled by ComMgr.dll

This section describes how the client handles some of its routine tasks.

### 6.1 Login

The login process is started when the user enters his MyDeliveryID and the MyDelivery password in the login dialog box and clicks on the login button. Then the MyDelivery client calls AuthorizeUser on the IMyDeliveryService interface. This is implemented in the function CComMgrApp::AuthorizeUser (ComMgr\ComMgr.cpp). This method calls the web service Initialize passing the parameters as defined in its SOAP function section earlier in this document.

The nature of COM by default on any call is blocking but his presents a problem if we need to cancel the Authorize call before it has finished. We solve this problem by issuing the call on a separate thread at address AuthorizeThread (MyDelivery\ComMgrDlg.cpp).

The modules that are used in the process are:

MyDeliveryClient->ComMgr+.exe->ComMgr.dll

### 6.2 Change Password

The change password process is started when the user enters his MyDeliveryID , the old MyDelivery password and the new MyDelivery password in the login dialog box and clicks on the login button. Then the MyDelivery client calls AuthorizeUser on the IMyDeliveryService interface. This is implemented in the function CComMgrApp::ChangePassword (ComMgr\ComMgr.cpp). This method calls the web service ChangePassword passing the parameters as defined in its SOAP function section earlier in this document.

The nature of COM by default on any call is blocking but his presents a problem if we need to cancel the ChangePassword call before it has finished. We solve this problem by issuing the call on a separate thread at address ChangePasswordThread (MyDelivery\ComMgrDlg.cpp).

The modules that are used in the process are:

MyDeliveryClient->ComMgr+.exe->ComMgr.dll

### 6.3 Addressbook Updates

The address book update process is started when the user clicks the OK button the address book dialog in the MyDelivery.exe client after making any changes. Then the MyDelivery client calls the method UploadUserInformation on the IMyDeliveryService interface. This is implemented in the function CComMgrApp:: UploadUserInformation (ComMgr\ComMgr.cpp). This method calls the web service UploadUserInformation passing the parameters as defined in its SOAP function section earlier in this document.

The modules that are used in the process are:

MyDeliveryClient->ComMgr+.exe->ComMgr.dll

### 6.4 Changes in Free Disk Space

The server keeps track of the amount of free disk space available on each client. This measurement helps the server from accepting and processing deliveries that cannot be handled by the recipient client due to lack of disk space. This not only saves the server bandwidth but also saves unnecessary processing on both the sender and receiver.

The change in the free disk space is computed every 10 minutes by the ClientStatusPage (CSP) thread (ComMgr\ClientStatusPage.cpp) in the function TenPercentChange (ComMgr\ClientStatusPage.cpp). If this change has occurred then UpdateStatus is signaled. The manager thread then executes UpdateStatus where the latest free disk space is computed and uploaded to the server.

Module: ComMgr\ClientStatusPage.cpp

```
unsigned __stdcall ClientStatusThread( void*   )
{
      COM_INIT thisThread;
      :
      while(1)
      {
            :
                  //fetch csp
                  theApp.GetClientStatusPage();

            :

            //on WAIT_TIMEOUT, signal an update on state change.
            if(csp != theApp.m_csp)
            {
                  //signal manager to call update status
                  theApp.m_events.SignalUpdateStatus();//UPDATE:ON
            }
            else if(tenMinutes <= 0)
            {
                  tenMinutes = 600000;

                  if(TenPercentChange(freeDiskSpace))
                  {
                        //signal manager to call update status
                        theApp.m_events.SignalUpdateStatus();//UPDATE:ON
                        freeDiskSpace = GetFreeBytes();
                  }
            }

            :
            Sleep(msWaitTime);
      }
      return 0;
}
```

## 6.5 Terminate Delivery

The terminate delivery process is started when the user deletes a delivery from the message list view. Then the MyDelivery client calls TerminateDeliveries on the IMyDeliveryService interface.

TerminateDeliveries terminates a delivery using a three step process

1. The deliveries that are not yet scheduled to be sent are deleted locally by deleting them from the out.xml file. This is implemented in the function CJobManager::TerminateDeliveries()(ComMgr\JobManager.cpp). All intermediate files such as compressed files are also deleted.
2. Deliveries that have already been sent or being sent to the server need to be deleted on the server and this is implemented in CUploadJobs::TerminateDeliveryOnServer (ComMgr\sender.cpp). The method called on the server is TerminateDelivery. The parameters are as defined as detailed in the SOAP function section earlier in this document. This is called after (1).
3. If terminate is requested on the delivery that is being sent.
    a. The event object m_CancelEvent is set. This is implemented in the function CUploadJobs::CancelCurrentDeliveryUpload(ComMgr\sender.cpp)
    b. The sender thread periodically checks to see if this delivery is being requested for cancellation by calling CUploadJobs::IsJobCancelled(ComMgr\sender.cpp) in the function CUploadJobs::UploadAttachments(ComMgr\sender.cpp).
    c. If set further processing is cancelled.

## 6.6 Error Handling and Management

When MyDelivery downloads

1. An attachment using DownloadAttachment ComMgr\download.cpp)
2. The header using DownloadDeliveryHeader1(ComMgrt\download.cpp)

It computes the SHA1 hash code for the downloaded attachment and compares it with the value sent by the sender. If the hash codes do not match then an error is generated at the recipient. The recipient informs the server that the delivery failed and provides the attachment number that failed to verify using the server function DeliveryStatus (this is implemented in the function DeliveryStatusOnChecksumFailure (ComMgr\decompress.cpp)). The recipient then goes on to process other outstanding jobs if any.

The sender thread is informed when it tries to upload the next attachment that it needs to resync this delivery. This information is passed as an error code. Once it receives the error the sender thread calls UpdateStatus (implemented in CComMgrApp::UpdateResync (ComMgr\manager.cpp)) to resync . The parameter returned by the server indicates where the client should start retransmitting from. The sender then back tracks to the correct attachment number and starts retransmitting the delivery. The recipient then downloads the retransmitted attachment the next time it calls DownloadAttachment (ComMgr\download.cpp) for this delivery. If the offset value is 0 then retransmission starts from UploadDeliveryHeader1.

## 7.0 Module: Usage.dll

The main function of the Usage.dll is to control the speed of the MyDelivery system. When extremely large files are being copied in the background or attached to a delivery, the system may possibly become unresponsive to user input from the keyboard or mouse. To prevent this, Usage.dll steps in by monitoring the keyboard and mouse and will immediately slow down the I/O operation of MyDelivery. This is accomplished by installing system hooks to the keyboard and mouse at Usage.dll initialization.

Module: Usage\usage.cpp

```
extern "C" __declspec(dllexport) void Initialize(void)
{
        :
        inactiveTicks = GetTickCount();
        hKeyHook = ::SetWindowsHookEx(WH_KEYBOARD, KeyboardProc, g_hModInstance, 0);
        hMouseHook = ::SetWindowsHookEx(WH_MOUSE, MouseProc, g_hModInstance, 0);
}
```

When the user moves his keyboard OR mouse the system calls the hook functions and the variable inactiveTicks is updated with the latest tick count.

Module: Usage\usage.cpp

```
LRESULT CALLBACK KeyboardProc(INT nCode, WPARAM wParam, LPARAM lParam)
{
        :
        inactiveTicks = GetTickCount();
        return CallNextHookEx(hKeyHook, nCode, wParam, lParam);
}
```

When any thread in MyDelivery.exe or ComMgr.dll performs I/O, it checks to see how busy the user is by calling the Usage.dll exported function GetInactivityWindow (Usage\usage.cpp). The function returns the difference between the current tick count and the last value updated by the hook callback. This gives us a variable that is inversely proportional to a systems usage. The value returned is used for controlling the speed of the thread by checking if it falls below 5000 ticks. The DELTA_T_DISKIO_MUL is a scaling multiplier and the value being multiplied is a function of the speed of the thread execution.

Module: ComMgr\CompressFiles.cpp

```
if(theApp.speed.LimitSpeedOnUserActivity())
{
        if(GetInactivity != 0 && ((*GetInactivity)() <= 5000))
                Sleep((etime-stime)*DELTA_T_DISKIO_MUL);
}
```

## 8.0 Module: ComMgr+.exe

The API server, ComMgr+.exe, is a single instance exe COM server that is a wrapper around the ComMgr.dll. The core function of the API server is to accept the jobs that are submitted via the command line tool and do the requested processing.

### 8.1 API Class

The MyDelivery API is an interface exposed by a COM server. The class definition of the service interface IMyDeliveryService is described in the ComMgr.idl. This will usually not be required to be known by the system integrators who just want to use the API platform unless they are planning to programmatically interface to and directly call ComMgr+.exe. Please note that this is a common interface to the MyDelivery client and the command line tool MDServer.exe. The interfaces that are related to the command line tool start with the letters cmd e.g. CmdSend(…). The interfaces that are used internally have the keyword [hidden] in the definition e.g. InitializeClient(). The rest of the interfaces are used by the MyDelivery client.

### 8.2 Event Management

The ComMgr+.exe platform has the dual role of servicing the MyDelivery.exe client interface and the command line tool. This is accomplished via the interface IMyDeliveryService. These dual roles impose certain requirements on the platform. For instance, the nature of the connection between the MyDelivery client and the platform is always 'ON', i.e. the platform should service all needs of the client when the user is logged in. However the command line tools are just processes that call the API and submit jobs and have no user interface. This means that they do not need any of the notifications like inbound delivery notifications.

The above dual requirements are so varied that supporting them in one system requires that we need to relay the client notifications only if the client is running but not otherwise. We thus define an event source dispatch interface called _IMyDeliveryServiceEvents. This source interface can thus be connected and disconnected from the API platform on demand without also disconnecting the main API interface. This ability is crucial since an API platform is required to continue to run and service clients even if the client is not running. Thus only the client will subscribe to this interface, where as the command line tool will not need to use this interface.

Module: ComMgr+\ComMgr.idl

```
[
	object,
	uuid(5E9A68E0-3902-46C2-B33A-398B589FDD8B),
	dual,
	nonextensible,
	helpstring("IMyDeliveryService Interface"),
	pointer_default(unique)
]

interface IMyDeliveryService : IDispatch{
	[id(1), helpstring("Set the service name and initialize")] HRESULT SetServiceName([in] BSTR name, [in]
LONG hasUI);
	[id(2), helpstring("Updates the users Address Book on the Server")] HRESULT UploadUserInformation([in]
BSTR addressBook, [in,out] BSTR* ErrorStatus);
	[id(3), helpstring("Sends a delivery.")] HRESULT SendDelivery([in] BSTR msgID, [in] BSTR to, [in] BSTR
subject, [in] BSTR message, [in] BSTR TotalSize, [in] LONG DelayCopy);
	[id(4), helpstring("Resends a delivery.")] HRESULT ResendDelivery([in] BSTR msgID);
	[id(5), helpstring("Terminates a delivery")] HRESULT TerminateDeliveries([in] BSTR msgIDs);
	[id(6), helpstring("Shuts down the communication server")] HRESULT ShutdownServer([in] LONG
exitcode);
	[id(7), helpstring("Checks recipient id")] HRESULT CheckRecipient([in] BSTR Recipient, [in] BSTR totalsize,
[in] BSTR largestFile, [in,out] LONG* response);
	[id(8), helpstring("Set proxy information")] HRESULT SetProxyInfo([in] BSTR username, [in] BSTR
password);
	[id(9), helpstring("Get Version")] HRESULT GetVersion([in,out] BSTR* ver, [in,out] BSTR* url, [in,out]
LONG* retVal);
	[id(10), helpstring("Authorizes a user")] HRESULT AuthorizeUser([in] BSTR username, [in] BSTR password,
[in,out] LONG* retVal);
	[id(11), helpstring("Changes the users password")] HRESULT ChangePassword([in] BSTR oldPassword, [in]
BSTR newPassword, [in,out] LONG* retVal);
	[id(12), helpstring("Acknowledges a notification")] HRESULT NotifyAck([in] BSTR msgID);
	[id(13), helpstring("Acknowledges a new delivery receipt")] HRESULT NewDeliveryAck([in] BSTR Key);
	[id(14), helpstring("Acknowledges an entry in the clients outbox")] HRESULT OutboxDeliveryAck([in] BSTR
msgIDs, [in,out] BSTR* Missing);
	[id(15), helpstring("Checks to see if a delivery can be sent")] HRESULT CanSendAttachments([in] BSTR
size);
	[id(16), helpstring("command line send")] HRESULT CmdSend([in] BSTR user, [in] BSTR password, [in] BSTR
proxyusername, [in] BSTR proxypassword, [in] BSTR dataxml, [in] BSTR statusxml, [in] LONG savecopy);
	[id(17), helpstring("command line version")] HRESULT CmdGetVer([in] BSTR proxyusername, [in] BSTR
proxypassword, [in] BSTR versionxml);
	[id(18), helpstring("command line terminate")] HRESULT CmdTerminate([in] BSTR user, [in] BSTR
password, [in] BSTR proxyusername, [in] BSTR proxypassword, [in] BSTR keyid);
	[id(19), helpstring("command line status")] HRESULT CmdGetStatus([in] BSTR user, [in] BSTR password,
[in] BSTR proxyusername, [in] BSTR proxypassword, [in] BSTR msgid, [in] BSTR statusxml);
	[id(20), helpstring("command line addressbook")] HRESULT CmdGetAddressBook([in] BSTR user, [in] BSTR
password, [in] BSTR proxyusername, [in] BSTR proxypassword, [in] BSTR path);
	[id(21), helpstring("Set path information")] HRESULT SetPath([in] BSTR appFolder, [in] BSTR dbFolder, [in]
BSTR dbFilePath);
	[id(22), helpstring("Request terminate clear")] HRESULT NotifyTerminate([in] BSTR msgid, [in,out] LONG*
ok);
```

```
        [id(23), helpstring("Release terminate lock")] HRESULT ReleaseTerminate(void);
        [id(24), helpstring("Get the service name")] HRESULT GetServiceName([in,out] BSTR* name);
        [id(25), helpstring("Set Active window")] HRESULT SetActiveWindow([in] BSTR hwnd);
        [id(26), helpstring("Get Active window")] HRESULT GetActiveWindow([in,out] BSTR* hwnd);
        [id(27), helpstring("Initializes the MyDelivery client."), hidden] HRESULT InitializeClient(void);
        [id(28), helpstring("Notifies an incoming delivery."), hidden] HRESULT NotifyNewDelivery([in] BSTR from,
[in] BSTR subject, [in] BSTR message, [in] BSTR Key);
        [id(29), helpstring("Notifies the sent status of a message."), hidden] HRESULT NotifyDeliveriestatus([in]
BSTR msgID, [in] BSTR status, [in] LONG ack);
        [id(30), helpstring("Notifies delivery failure."), hidden] HRESULT NotifyDeliveryFailure([in] BSTR from, [in]
BSTR subject, [in] BSTR message, [in] BSTR msgID);
        [id(31), helpstring("Shuts down the client interface."), hidden] HRESULT ShutdownClient([in] LONG
exitcode, [in] BSTR Reason);
        [id(32), helpstring("method CommStatus"), hidden] HRESULT CommStatus([in] ULONG status);
        [id(33), helpstring("method Sets the caption in the active window"), hidden] HRESULT
SetWindowMessage([in] BSTR text);
        [id(34), helpstring("method Gets the caption in the active window"), hidden] HRESULT
GetWindowMessage([in, out] BSTR* text);
        [id(35), helpstring("method Displays a message box"), hidden] HRESULT DisplayMessage([in] BSTR text,
[in] LONG style);
        [id(36), helpstring("method Saves the status of a delivery"), hidden] HRESULT UpdateStatusCache([in]
BSTR msgID, [in] BSTR status);
        [id(37), helpstring("method Signals that a send delivery has finished"), hidden] HRESULT
SendCompleted([in] BSTR msgID);
        [id(38), helpstring("method Resumes sending deliveries that were not sent.")] HRESULT CmdResume([in]
BSTR user, [in] BSTR password, [in] BSTR proxyusername, [in] BSTR proxypassword);
};
[
        uuid(341A6222-C3D3-4EC9-922F-C384A0BE5835),
        version(1.0),
        helpstring("MyDeliveryService 1.0 Type Library")
]
library MyDeliveryServiceLib
{
        importlib("stdole2.tlb");
        [
                uuid(B5A8B1EA-37E9-4A4E-BFA9-A7E53C490CE3),
                helpstring("_IMyDeliveryServiceEvents Interface")
        ]
        dispinterface _IMyDeliveryServiceEvents
        {
                properties:
                methods:
                        [id(1), helpstring("Initializes the MyDelivery client.")] HRESULT InitializeClient(void);
                        [id(2), helpstring("Notifies an incoming delivery.")] HRESULT NotifyNewDelivery([in]
BSTR from, [in] BSTR subject, [in] BSTR message, [in] BSTR Key);
                        [id(3), helpstring("Notifies the sent status of a message.")] HRESULT
NotifyDeliveriestatus([in] BSTR msgID, [in] BSTR status, [in] LONG ack);
                        [id(4), helpstring("Notifies delivery failure.")] HRESULT NotifyDeliveryFailure([in] BSTR
from, [in] BSTR subject, [in] BSTR message, [in] BSTR msgID);
                        [id(5), helpstring("Shuts down the client interface.")] HRESULT ShutdownClient([in]
LONG exitcode, [in] BSTR Reason);
                        [id(6), helpstring("method CommStatus")] HRESULT CommStatus([in] ULONG status);
```

```
				[id(7), helpstring("method Sets the caption in the active window")] HRESULT
SetWindowMessage([in] BSTR text);
				[id(8), helpstring("method Gets the caption in the active window")] HRESULT
GetWindowMessage([in, out] BSTR* text);
				[id(9), helpstring("method Displays a message box")] HRESULT DisplayMessage([in] BSTR
text, [in] LONG style);
				[id(10), helpstring("method Increments the unread count in the outbox.")] HRESULT
IncrementOutboxUnread(void);
	};
	[
		uuid(4292A28F-36D1-443E-8B20-0CDB71801E69),
		helpstring("MyDeliveryService Class")
	]
	coclass MyDeliveryService
	{
		[default] interface IMyDeliveryService;
		[default, source] dispinterface _IMyDeliveryServiceEvents;
	};
};
```

## 8.3 Intialization

The API server, ComMgr+.exe, is a multithreaded apartment model EXE server that implements the platform. The server is registered by running the command ComMgr+.exe /RegServer at the command prompt.

The API server limits itself to a single instance and creates 3 mutex objects at startup. These 3 mutexes are responsible for synchronizing all service calls ( m_hService ) , notifications (m_hNotify) and cache updates (m_hCache) . The synchronization is necessary as there are a number of threads that could be concurrent inside the API server. The use of these mutex objects orders contention among the threads that compete for the same resources.

The m_hService mutex guards all service alls and ensures that only one of these service calls are processed at a time by the platform. When the processing is complete the next service call is handled and so on until all outstanding calls are serviced. This mutex also ensures that login is atomic.

The m_hNotify mutex guards all notifications and ensures that only one notification is sent out at a time. This is necessary to prevent race conditions and deadlock by the ComMgr.dll notifications.

The m_hCache mutex supports fast cache updates and guards status structures in the application. This is necessary to prevent the cache from being updated at the same time a GetStatus call is being serviced.

## 8.4 Shutdown

The shutdown is handled by a separate thread called the StopService. This thread periodically runs when a job is removed from the list of jobs to check if it is okay for the server to shutdown. The server can shutdown when the application class member CComMgrModule::m_mapPC no longer owns any jobs and its size is zero.

Module: ComMgr+\stopservice.cpp

```
static unsigned __stdcall StopService(void* p)
{
	MTA thisThread;

	CStopService* ss = (CStopService*)p;

	try
	{
		while(1)
		{
			//wait for run
			SuspendThread(ss->m_hStopService.get());

			//acquire mutex
			std::auto_ptr<CAcquireMutex> lock(new CAcquireMutex(_AtlModule.m_hService));

			//if we have no more jobs to process, initiate shutdown
			if(_AtlModule.m_mapPC.size() == 0)
			{
				if(_AtlModule.m_pDLLi != NULL)
				{
					// shutdown processing
					:

					//stop application message pump
					::PostQuitMessage(0);

					break;
				}
			}
		}
	}
	catch(...)
	{
		// do not handle exception thrown when execution is cancelled
	}

	return 0;
}
```

## 8.5 Event Routing

When the application server starts up the client that has instantiated the server calls the SetServiceName method. This call is where the initialization of the ComMgr.DLL is performed. Among the parameters passed to the DLL a ‘this’ pointer of the caller is passed. This is saved by the DLL for any callbacks and notifications. Please see CComMgrModule::SetServiceName in the file startservice.cpp.

Module: ComMgr+\startservice.cpp

```
STDMETHODIMP CComMgrModule::SetServiceName(BSTR name, LONG hasUI, IMyDeliveryService* psi)
{
	std::auto_ptr<CAcquireMutex> lock(new CAcquireMutex(m_hService));

	:

		if(!m_bDllInitialized)
		{
			InitializeServer(psi, m_hService);

			m_bDllInitialized = true;

			:

			return S_OK;
		}

	:

	return S_OK;
}
```

## 8.6 Event Routing to Client

The MyDelivery client creates an 'event source' interface to which the 'user interface client' subscribes. The event source object saves the IUnknown of the 'event sink' (consumer). It then uses the cached pointer to report events to the client. This kind of a mechanism is necessary to avoid having to report events to 'send only' clients.

Send only clients do not wish to receive deliveries and are only concerned about sending deliveries to a recipient. An example of a send only delivery would be an X-ray machine that needs to send the patients X-ray to a radiologist.

Send only clients do not register themselves with the 'event source' (publisher). Please refer to the section **MyDelivery Event Sink Setup** described earlier**.**

When an event occurs, the platform passes the event to the client via the registered connection. This happens on the interface _IMyDeliveryServiceEvents this is implemented in the file _IMyDeliveryServiceEvents_CP.h. For example when a new delivery is to be shown in the inbox of the client the DLL calls CComMgrModule::NotifyNewDelivery which calls Fire_NotifyNewDelivery. The notification does a standard Invoke and calls the client.

Module: ComMgr+_IMyDeliveryServiceEvents_CP.h

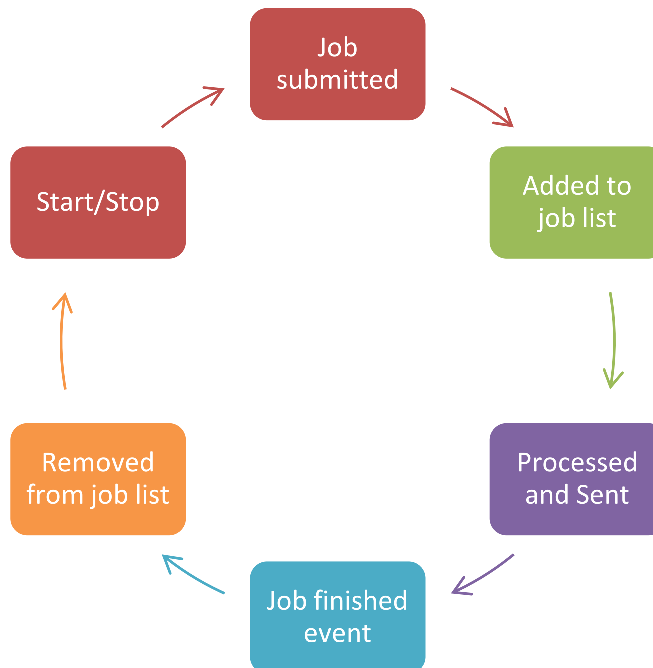
```
HRESULT Fire_NotifyNewDelivery( BSTR from,  BSTR subject,  BSTR message,  BSTR Key)
{
	HRESULT hr = S_OK;
	T * pThis = static_cast<T *>(this);
	int cConnections = m_vec.GetSize();

	for (int iConnection = 0; iConnection < cConnections; iConnection++)
	{
		pThis->Lock();
		CComPtr<IUnknown> punkConnection = m_vec.GetAt(iConnection);
		pThis->Unlock();

		IDispatch * pConnection = static_cast<IDispatch *>(punkConnection.p);

		if (pConnection)
		{
			//unmarshal dispatch interface
			:

			//invoke
			DISPPARAMS params = { avarParams, NULL, 4, 0 };
			hr = pConnection->Invoke(2, IID_NULL, LOCALE_USER_DEFAULT, DISPATCH_METHOD,
			&params, &varResult, NULL, NULL);
		}
	}
	return hr;
}
```

### 8.7 Process to Completion

The process to completion is performed by simply not allowing the ComMgr.DLL to shutdown until the submitted job has been processed till complete. This is accomplished by a simple linked list of job objects that are maintained by the main application class `CComMgrModule`. Every time a job is submitted it is added to this list and addref'ed and every time a job has completed processing it is removed from this list and released.

Module: ComMgr+\runtocompletion.cpp

```
void CComMgrModule::RunJobToCompletion(BSTR msgID, CMyDeliveryService* ps)
{
	if(msgID == NULL || ps == NULL)
		return;

	std::auto_ptr<CAcquireMutex> lock(new CAcquireMutex(m_hService));

	string id((LPCTSTR)_bstr_t(msgID));

	map<string,CMyDeliveryService*>::iterator ism = m_mapPC.find(id);
	if(ism == m_mapPC.end())
	{
		//insert entry
		m_mapPC.insert(map<string,CMyDeliveryService*>::value_type(id, ps));

		//call addref
		ps->AddRef();

	}
}
```



Jobs are submitted to the ComMgr+.exe server by instantiating it just as any other COM server and then calling methods on it. The methods themselves and its arguments are listed in the ComMgr.idl file included earlier.

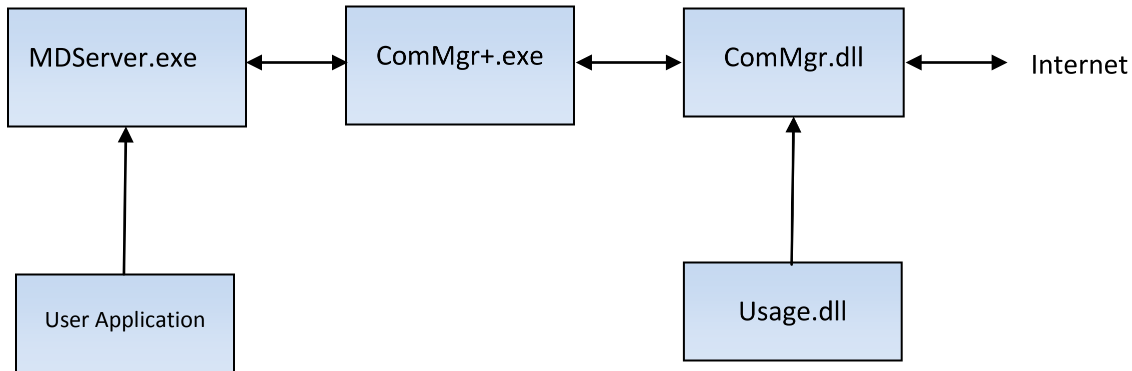
Once a job has finished processing, the event is reported back to the client if it submitted the job or discarded suitably. As each job moves through the system the resulting deliveries 'release' objects that submitted them originally on completion in the function `CComMgrModule::SendCompleted()`. When the count of the objects reaches zero the API server determines that all jobs have completed and initiates a shutdown.

Module: ComMgr+\runtocompletion.cpp

```
STDMETHODIMP CComMgrModule::SendCompleted(BSTR msgID)

{
	:

	std::auto_ptr<CAcquireMutex> lock(new CAcquireMutex(m_hService));
	string id((LPCTSTR)_bstr_t(msgID));

	map<string,CMyDeliveryService*>::iterator ism = m_mapPC.find(id);
	if(ism != m_mapPC.end())
	{
		//call addref
		ism->second->Release();

		//remove reference
		m_mapPC.erase(ism);
	}
	:

	//if we have no more jobs to process, stop service
	m_service.Stop();

	return S_OK;
}
```

## 9.0 Module: MDServer.exe – Using the API

The MDServer.exe module provides the Applications Programming Interface (API) for MyDelivery.




MyDelivery API Platform

Some potential MyDelivery users expressed the need for a MyDelivery Applications Programming Interface (API) system that third party devices could leverage to send files. A typical scenario would be a portable diagnostic device (such as an x-ray machine) that would plug into the local MyDelivery system and submit diagnostic information that is sent to a hospital over the internet. Another potential use would be interfacing an existing health record system to MyDelivery. Through the API it would be possible to send patient records to a remotely located MyDelivery client. Our view on addressing this need was that any such API should ideally plug into the existing MyDelivery delivery management structure and leverage existing code via COM. Hence the API was created to allow an easy way for other systems to interface to MyDelivery. This API is implemented in the MDServer.exe program.

The ComMgr+.exe platform can be directly interfaced to by a programmer and jobs submitted to it directly using the interface definitions stated in ComMgr.idl. However this involves some complexity and may exclude some systems that are restrictive.

To ease use and promote system integration the MDServer.exe was created. It is a command line tool that translates arguments and formats them suitably before passing them to the ComMgr+.exe API server.

It has a parser that breaks the command line parameters to arguments. It then instantiates platform API services as a COM server and invokes methods on it with the correct arguments. The individual commands are stated and implemented as follows:

## 9.1 Command: GetVersion

Downloads the latest version information to the file version.xml

*Use case 1:*
*MDServer --proxyusername pn --proxypassword pp --version  --file version.xml*

Case 1 applies in situations where a proxy server blocks access to the Internet.

*Use case 2:*
*MDServer --version  --file version.xml*

1. pn is the proxy username for the user to his gateway server
2. pp is the password for the proxyuser required by the gateway server
3. --version command will be used by a client to check the MyDelivery version
4. –version.xml is the path to an XML file that will receive the version number of the latest client software available from the MyDelivery server, and the new software URL where it can be obtained

Module: MDServer\version.cpp

```
 bool CVersion::GetVersion(void)
{
	//query any available server
	if(!theApp->CreateServiceInterface(NULL))
		return false;

	IMyDeliveryServicePtr p(theApp->GetServiceInterface(), false);

	if(p->CmdGetVer(m_proxyuserid.c_str(), m_proxypassword.c_str(), m_version_xml.c_str()) != 
S_OK)
	{
		//indicate error
		cout << "\nversion error:" << endl;
		DumpFile(m_version_xml);
		return true;
	}

	//dump version to stdout
	cout << "\nversion:" << endl;
	DumpFile(m_version_xml);

	return false;
}
```

### 9.2 Command: GetAddressBook

Downloads the user's addressbook on successful authorization to the path address.xml.

*Use case 1:*

*MDServer --user ID --password Password --proxyusername pn --proxypassword pp --address address.xml*

Case 1 applies in situations where a proxy server blocks access to the Internet.

*Use case 2:*

*MDServer --user ID --password Password --address address.xml*

1. ID is the MyDeliveryID
2. Password is the MyDelivery password
3. pn is the proxy username for the user to his gateway server
4. pp is the password for the proxyuser required by the gateway server
5. --address command will be used by a client to download their address book
6. address.xml specifies the path for the address book.

Module: MDServer\addressbook.cpp

```
bool CAddressBook::GetAddressBook(void)
{
	if(!theApp->CreateServiceInterface(m_userid.c_str()))
		return false;

	IMyDeliveryServicePtr p(theApp->GetServiceInterface(), false);

	if(p->CmdGetAddressBook(m_userid.c_str(), m_password.c_str(),
		m_proxyuserid.c_str(), m_proxypassword.c_str(), m_address_xml.c_str()) != S_OK)
	{
		cout << "\naddressbook error:" << endl;
		return false;
	}

	//indicate success
	cout << "\naddressbook:" << endl;

	DumpFile(m_address_xml);

	return true;
}
```

## 9.3 Command: Send

Sends a delivery to the recipient on successful authorization.

*Use case 1:*

*MDServer --user [ID] --password [Password] --send data.xml --out status.xml*

*Use case 2:*

*MDServer --user [ID] --password [Password] --send data.xml --out status.xml --savecopy*

*Use case 3:*

*MDServer --user [ID] --password [Password] --proxyusername pn --proxypassword pp --send data.xml --out status.xml --savecopy*

*Use case 4:*

*MDServer --user [ID] --password [Password] --proxyusername pn --proxypassword pp --send data.xml --out status.xml*

1. ID is the MyDeliveryID
2. Password is the MyDelivery password
3. pn is the proxy username for the user to his gateway server
4. pp is the password for the proxyuser required by the gateway server
5. –data.xml command specifies the path to an XML file that holds the following information:
   - The recipient id descriptor enclosed in tag <RECIPIENT>
   - The subject enclosed in tag <SUBJECT>
   - The content enclosed in tag <CONTENT>
   - An attachment descriptor enclosed in tag <ATTACHMENTS>

   Here is an example data.xml file that does not contain attachments:
   ```
   <?xml version="1.0"?><MyDelivery><RECIPIENT>john</RECIPIENT>
   <SUBJECT>test </SUBJECT>
   <CONTENT>test message</CONTENT>
   <ATTACHMENTS></ATTACHMENTS>
   </MyDelivery>
   ```

   Here is a sample data.xml file that has one file attached named 1a.cr2. Please refer to the description of UploadDeliveryHeader1 in the accompanying “The MyDelivery Server” for the specification of the “Tree” parameter. The “Attachments” parameter here is equivalent to the “Tree” parameter in UploadDeliveryHeader1.
   ```
   <?xml version="1.0"?><MyDelivery><RECIPIENT>john</RECIPIENT>
   <SUBJECT>test </SUBJECT>
   <CONTENT>test message</CONTENT>
   <ATTACHMENTS>0*-1*0*0*0*16**MyDelivery*0*0**15010
   ```

1*0*129098543678522132*129173704705760497*128726390320000000*32*1a.CR2*1a.CR2*
0*27115179*C:\Temp*-1
</ATTACHMENTS>
</MyDelivery>

6. –status.xml specifies a path to an XML file that receives the result and error codes returned, and also contains a msgid (32-bit number) that identifies this delivery.
7. --savecopy keeps a copy of this message in the MyDelivery clients sent items folder post send, if not specified the message will be auto deleted post send.

MDServer\send.cpp

```
bool CSend::SendDelivery(void)
{
	if(!theApp->CreateServiceInterface(m_userid.c_str()))
		return false;

	IMyDeliveryServicePtr p(theApp->GetServiceInterface(), false);

	if(p->CmdSend(m_userid.c_str(), m_password.c_str(),
		m_proxyuserid.c_str(), m_proxypassword.c_str(),
		m_dataXml.c_str(),
		m_statusXml.c_str(), bSaveCopy ? 1 : 0) != S_OK)
	{
		//indicate error
		cout << "\nsend error:" << endl;
		DumpFile(m_statusXml);
		return false;
	}

	//dump status to stdout
	cout << "\nsend:" << endl;
	DumpFile(m_statusXml);

	return true;
}
```

## 9.4 Command: GetStatus

The command GetStatus returns the status of a delivery that was submitted earlier for sending on successful authorization.

*Use case 1:*

*MDServer --user ID --password Password --proxyusername pn --proxypassword pp--getstatus --key msgid --out status.xml*

*Use case 2:*

*MDServer --user ID --password Password --getstatus --key msgid --out status.xml*

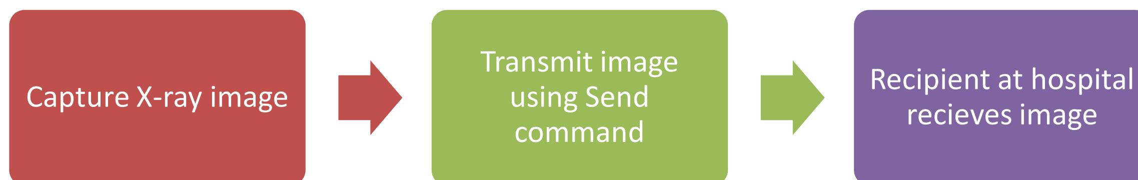
1. ID is the MyDeliveryID
2. Password is the MyDelivery password
3. pn is the proxy username for the user to his gateway server
4. pp is the password for the proxyuser required by the gateway server
5. --getstatus command is used by a client to determine the status of a delivery.
6. --msgid specifies the unique identification of a delivery that whose status is requested.
7. -- status.xml specifies the path of an XML file that receives the result of the request and will contain information as to how much of the delivery has been sent out so far.

MDServer\status.cpp

```
bool CStatus::GetStatus(void)
{
	if(!theApp->CreateServiceInterface(m_userid.c_str()))
		return false;

	IMyDeliveryServicePtr p(theApp->GetServiceInterface(), false);

	if(p->CmdGetStatus(m_userid.c_str(), m_password.c_str(),
		m_proxyuserid.c_str(), m_proxypassword.c_str(),
		m_msgid.c_str(), m_statusXml.c_str()) != S_OK)
	{
		//indicate error
		cout << "\ngetstatus error:" << endl;
		DumpFile(m_statusXml);
		return true;
	}

	//dump status to stdout
	:
	return false;
}
```

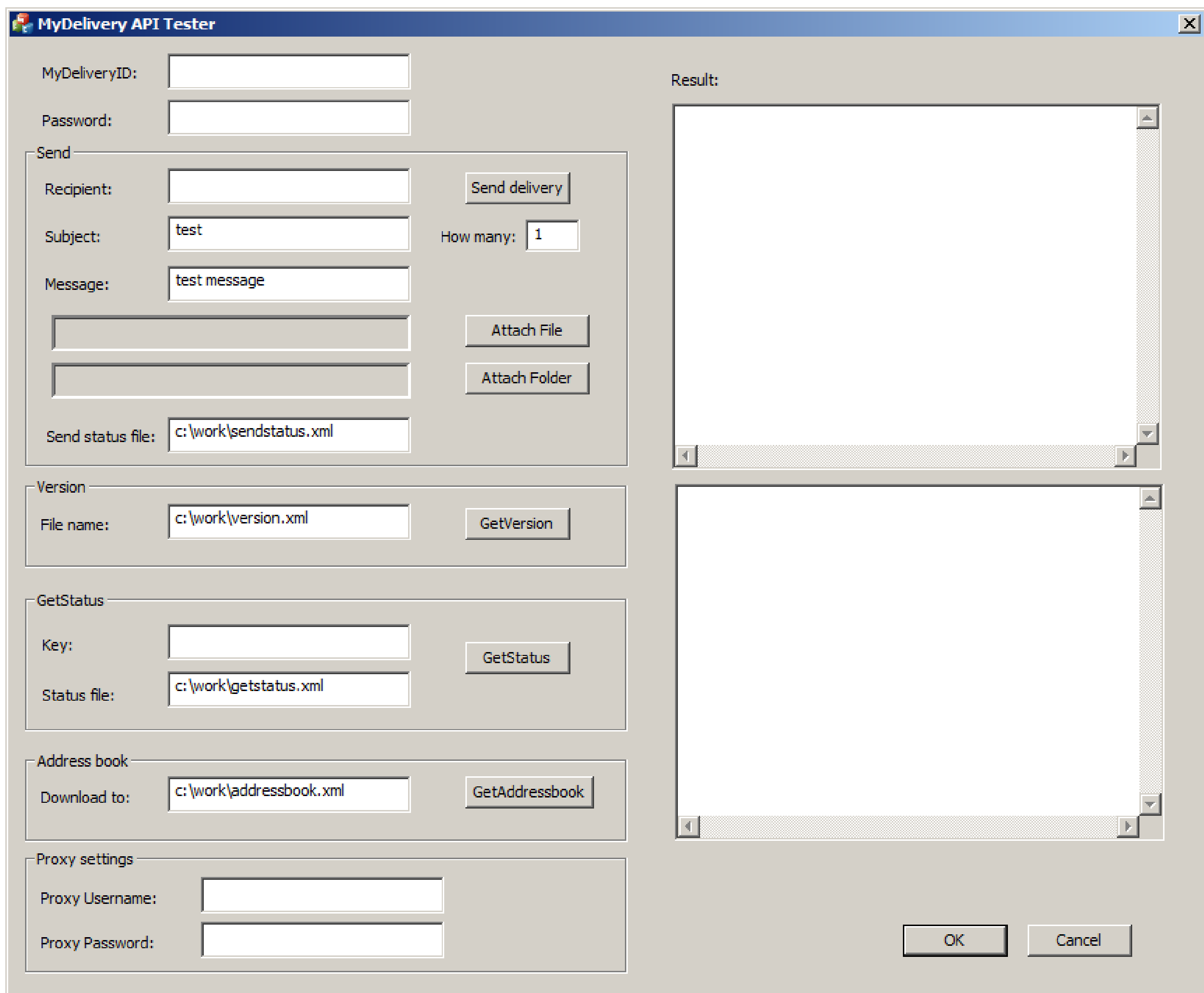
### 9.5 An Example of a Possible API Application

MyDelivery offers a flexible system where a device such as an X-ray machine or a CT scanner can send large files in an automated program to a radiologist anywhere on the Internet. For example an X-ray machine that periodically takes images could interface to the MyDelivery system through the command line interface and send the image to a recipient at the hospital (say a radiologist) using the send command described earlier. All the x-ray machine software would have to do is run the command line tool MDServer pointed to the image file. The recipient, of course, needs to run a MyDelivery client to receive the x-rays.



## 10.0 Module:APITest.exe (MyDelivery API Tester)

The MyDelivey API Test application, APITest.exe, which is included with the source distribution package, is intended to demonstrate how the MyDelivery platform API system can be used. Written in C++, it serves as a starting point for interfacing an existing system or a device to MyDelivery. To use it, run apitest.exe, and supply the appropriate arguments through the dialog interface. Please note that although the included MyDelivery client can send and receive deliveries, the API platform is limited to serve as a send only program that will need a client at the other end to receive and view the deliveries.




The APITest.exe user interface

For testing purposes, it is possible to have both a client (mydelivery.exe) and the apitest.exe running simultaneously on the same computer. For example, you may want to use apitest.exe to send deliveries to the client.

This is a description of the various components of the apitest.exe user interface dialog box:

| Field | Description |
|---|---|
| MyDeliveryID | The MyDeliveryID of the user |
| Password | The MyDelivery password of the user |
| Recipient | The MyDeliveryID of the recipient |
| Send | |
| Subject | The subject text |
| Message | The message text |
| Attach File button | Selects a file to attach |
| Attach Folder button | Selects a folder to attach |
| How many | How many deliveries should be sent |
| Send delivery button | Sends the delivery |
| Version | |
| File name | Points to the file that shall receive the version information |
| Get Version button | When clicked executes the version command |
| GetStatus | |
| Key | Holds the GUID value that identifies this delivery. It is auto filled when a delivery is sent by clicking the Send delivery button. |
| Status file | Points to the file that shall receive the status information |
| Address book | |
| Download to | Points to the file that shall receive the downloaded addressbook |
| GetAddressbook button | When clicked executes the addressbook command |
| Proxy Settings | |
| Proxy Username | The proxy username for the local gateway (if any) |
| Proxy Password | The proxy password for the local gateway |
| Result: | |
| Top right list box window | Holds the result of a send operation |
| Bottom right list box window | Holds the result of a GetVersion, GetAddressbook and GetStatus operation |

## 10.1 Proxy Settings

The Proxy Username and Proxy Password are used to enter the proxy information for MyDelivery if we are behind an Internet proxy server. The presence of a local proxy server can be usually be checked by typing the address of an external site in your Internet browser. If no firewall is present then you will be not be prompted by the browser to enter a username and password.

One should obtain the Proxy username and Proxy password from the local systems administrator if a proxy server is present. When no proxy server is present we do not need to set these values and they can be left blank.

The next four sections demonstrate four ways for using the API Test program:

1. Getting the current version of the MyDelivery system.
2. Downloading your address book.
3. Sending a delivery.
4. Getting the status of a delivery that was just sent.

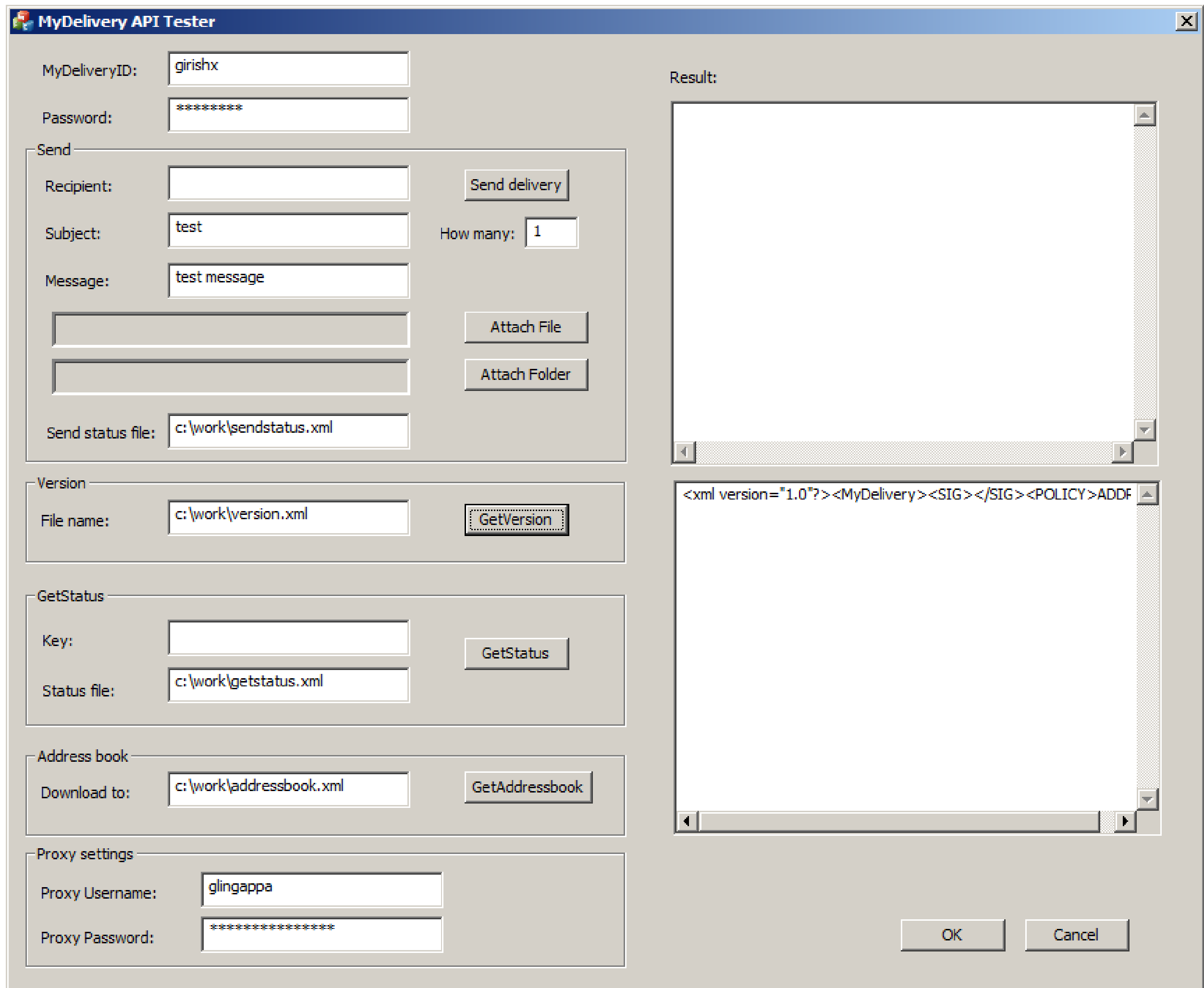
## 10.2 Get the current version of the MyDelivery system

Getting the correct version and checking for suitability is necessary to any system integration. The service returns the latest version and the URL for the latest software. The GetVersion button demonstrates this.

To get the version of MyDelivery:

1. Run the APITest.exe program.
2. Enter a desired path in the Version group box. (By default it is set to c:\work\version.xml)
3. If you have a proxy username and password enter them in the Proxy Settings group box. If you do not have a proxy server then these fields should be left blank.
4. Click the GetVersion button.

The capture below shows the version and the new software URL listed in the lower right window.



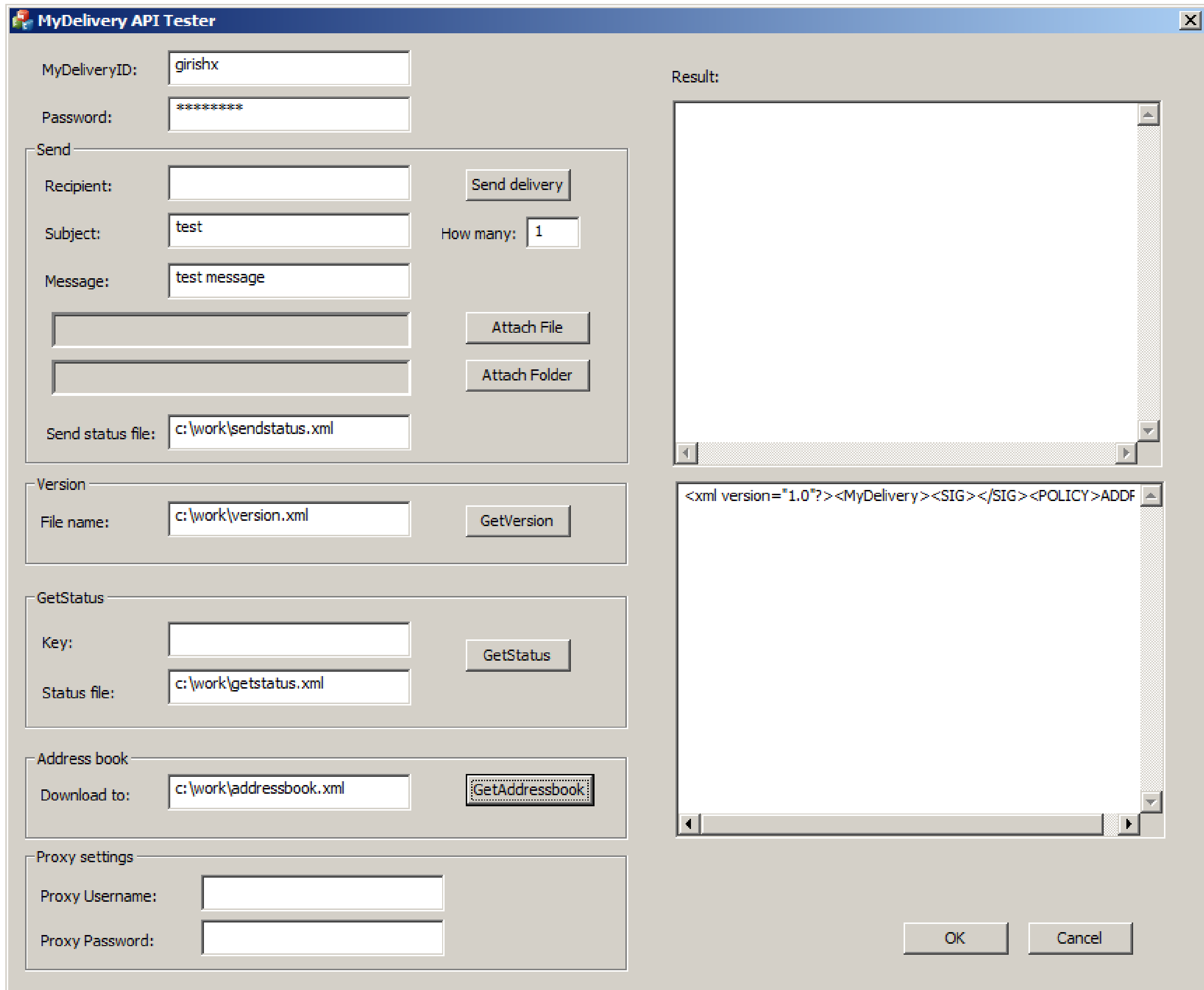

## 10.3 Downloading your AddressBook

To download an address book you need to first be registered with the MyDelivery system.

1. If not already registered, register with the MyDelivery system.
2. Run the MyDelivery.exe client
3. Add recipients to your address book.
4. Save and close the client.
5. Run the APITest.exe program.
6. Enter your MyDelivery ID and password in the MyDeliveryID and Password fields of the APITest.exe dialog.
7. If you have a proxy username and password enter them in the Proxy Settings group box. If you do not have a proxy server then these fields should be left blank.
8. Enter a desired path in the address book group box. (By default it is set to c:\work\addressbook.xml)
9. Click the GetAddressbook button.

One can only send deliveries to recipients who are registered with the MyDelivery system and who are listed in your address book.   In addition, each recipient must have your MyDeliveyID listed in their address book.

The screen capture below shows the address book listed in the lower right window.



## 10.4 Sending a delivery

To send a delivery, both the sender and recipient need to first be registered with the MyDelivery system.

1. If not already registered, register with the MyDelivery system.
2. Run the MyDelivery.exe client
3. Add the recipient to your address book. The recipient must also add the send to his address book
4. Close the client.
5. Run the APITest.exe program.
6. Enter your MyDelivery ID and password in the MyDeliveryID and Password fields of the APITest.exe dialog.
7. Enter a single recipient's MyDeliveryID, a suitable subject and a message.
8. You may also select a folder that you would like to attach with this delivery.
9. Enter the path to a send status file. By default this path is c:\Work\sendstatus.xml.
10. If you have a proxy username and password enter them in the Proxy Settings group box. If you do not have a proxy server then these fields should be left blank.
11. If you wish multiple deliveries be sent to the same recipient, enter the number in the "How many:" text box.
12. Click "Send delivery" and the API platform will send your delivery to the recipient. Please note that the recipient's client should be running to receive the delivery.

A delivery is identified by a large number called a Globally Unique ID (GUID), and it is displayed with the <MESSAGEID> tag. You will need to use this number to obtain the status of the delivery. When a delivery is sent by clicking the send delivery button the API Tester will automatically fill in the <MESSAGEID> value for the last delivery sent.

The <ERROR> tag will indicate the reason for any failures or a success. These are displayed in the Result window.

**MyDelivery API Tester**

MyDeliveryID: girishx

Password: *********

Send

Recipient: frank

Subject: test

Message: test message

C:\Perl\html

Send status file: c:\work\sendstatus.xml

Send delivery

How many: 1

Attach File

Attach Folder

Version

File name: c:\work\version.xml

GetVersion

GetStatus

Key: {C9168768-6981-BAC0-D0D-DOD(

Status file: c:\work\getstatus.xml

GetStatus

Address book

Download to: c:\work\addressbook.xml

GetAddressbook

Proxy settings

Proxy Username:

Proxy Password:

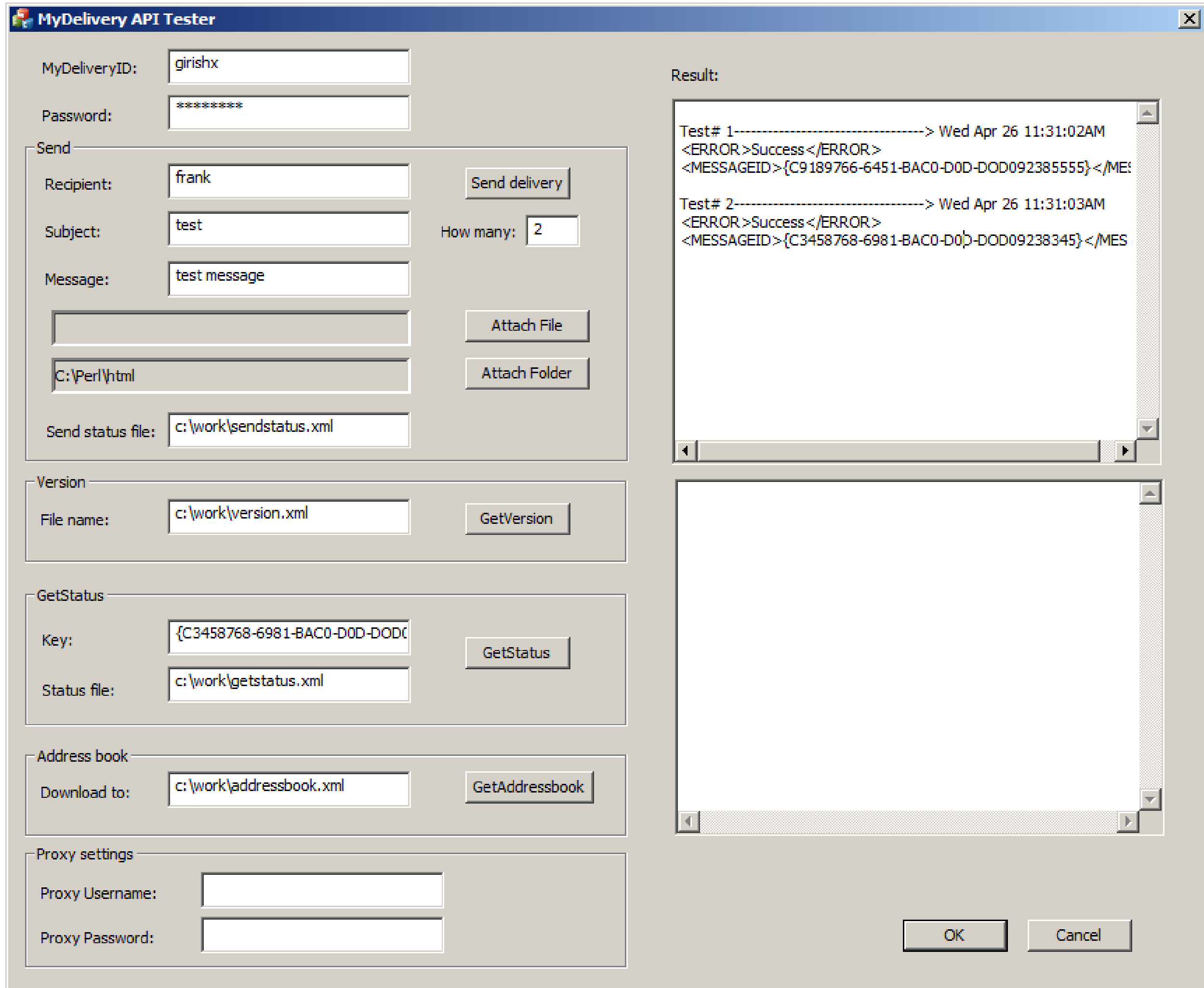
Result:

Test# 1--------------------------------> Wed Apr 26 11:30:02AM
<ERROR>Success</ERROR>
<MESSAGEID>{C9168768-6981-BAC0-D0D-DOD092384098}</ME

OK

Cancel

If multiple deliveries were chosen to be sent to the same recipient, the number of deliveries that were submitted for sending is listed in the Result window. The second image displays this in action. In this example, two deliveries were chosen to be sent.



## 10.5 Getting the status of a delivery

To get the status of a delivery:

1. Send a delivery using the steps detailed in the previous section.
2. When a delivery is sent by clicking the send delivery button the API Tester will automatically fill in the <MESSAGEID> value for the last delivery sent. One may also optionally enter the <MESSAGEID> tag value in the Key edit control inside the GetStatus group box. You will need to use this number to obtain the status of the delivery.
3. Click the GetStatus button to obtain the status of a delivery.

In the capture below the status indicates that at the time of capture about 76% of the attachments in the c:\Perl\html folder were sent to the recipient.

MyDelivery API Tester

MyDeliveryID: girishx

Password: ********

Send

Recipient: frank

Send delivery

Subject: test

How many: 2

Message: test message

Attach File

C:\Perl\html

Attach Folder

Send status file: c:\work\sendstatus.xml

Version

File name: c:\work\version.xml

GetVersion

GetStatus

Key: {C3458768-6981-BAC0-D0D-DOD(

GetStatus

Status file: c:\work\getstatus.xml

Address book

Download to: c:\work\addressbook.xml

GetAddressbook

Proxy settings

Proxy Username:

Proxy Password:

Result:

Test# 1---------------------------------> Wed Apr 26 11:31:02AM
<ERROR>Success</ERROR>
<MESSAGEID>{C9189766-6451-BAC0-D0D-DOD092385555}</ME

Test# 2---------------------------------> Wed Apr 26 11:31:03AM
<ERROR>Success</ERROR>
<MESSAGEID>{C3458768-6981-BAC0-D0D-DOD09238345}</MES

<STATUS>Sending ( 76% complete )</STATUS>

OK

Cancel