ZCP 7.1 (build 48315)

Zarafa Collaboration Platform

Python MAPI bindings

ZCP 7.1 (build 48315) Zarafa Collaboration Platform
Python MAPI bindings
Edition 1.0

The python MAPI bindings allow full access to Zarafa's messaging api MAPI.

[1] http://creativecommons.org/licenses/by-sa/3.0/

Introduction

The Python language binding provides a python interface to the messaging API used by Zarafa, MAPI. The original interface is an object-oriented C+\+ interface, and the interfaces provided to Python are designed to be completely analogous to the C+\+ interface.

Since the interface is otherwise identical to thes C+\+ interface, this document describes naming conventions, calling conventions and other Python-specific constructs, but does not document the API itself.

For the Python-programmer’s convenience, we provide several examples on how to use the Python Bindings.

Target audience

This document is mainly targeted at c+\+ developers wanting to do MAPI scripting through python, and python developers who want to start doing MAPI development.

It is a good idea to read some more about MAPI before proceeding with this document.

Conventions

3.1. Objects

There is a one-to-one relationship between objects in MAPI and the objects presented to Python. MAPI provides two main entrypoints in a classfactory-pattern to obtain references to the top-level objects.

These functions are:

```
MAPIAdminProfiles()
MAPILogonEx()
```

These functions can be used to create and manage MAPI profiles, and log on to one of those profiles, respectively. Each call returns a top-level object which can be used to subsequently open other objects:

```
import MAPI

profadmin = MAPI.MAPIAdminProfiles(0)
profadmin.CreateProfile('profile','password', 0, 0)

...

session = MAPI.MAPILogonEx(0, 'profile','password', 0)
session.GetMsgStoresTable(0)
```

3.2. Modules

The default *MAPI* package imports the c+\+ bindings, functions and methods. However, there are also some other modules that make life easier for the Python MAPI programmer:

3.2.1. MAPI.Struct

Python representations of MAPI structs. These contain data that cannot be described as a simple int or string. Naming of classes in this module mirrors the MAPI Struct as in c+\+ and provides convenient constructors to create data structures from Python

```
import MAPI.Struct
import MAPI.Tags

prop = SPropValue(MAPI.Tags.PR_SUBJECT, 'hello, world!')
```

3.2.2. MAPI.Time

Since MAPI uses a timestamp format that is not very common in the Python world called FILETIME (100-ns periods since 1 jan 1601), MAPI.Time offers an easy way to convert to and from the standard epoch-based timestamps normally used in Unix:

```
import MAPI.Time

t = FileTime(10000000000000)
print t.unixtime
```

```
t = unixtime(1234567890)
print t.filetime
```

All PT_SYSTIME values in MAPI will be converted to/from this format.

Note

FILETIME has a much wider time frame and greater precision than unix timestamps. Allthough in practice a precision of 1 second is usually fine, it may cause subtle problems if you are assuming the full FILETIME precision.

3.2.3. MAPI.Tags

The MAPI.Tags module contains constants for all well-known MAPI property tags:

```
import MAPI.Tags

print MAPI.Tags.PR_SUBJECT
print MAPI.Tags.PR_SUBJECT_A
print MAPI.Tags.PR_SUBJECT_W
```

The constants are identical to those used in c+\+ MAPI. By default, string tags use the PT_STRING8 (terminal charset) type. If you wish to use the PT_UNICODE variant, either use the _W type (eg PR_SUBJECT_W) or use the following construct:

```
import MAPI
MAPI.unicode = True
import MAPI.Tags

print PROP_TYPE(MAPI.Tags.PR_SUBJECT)
# outputs 31 (PT_UNICODE)
```

This will cause all properties to default to the PT_UNICODE property type.

3.2.4. MAPI.Defs

Provides convenience functions (also identical to their c+\+ counterparts) to create, test and modify property tags, and other utility functions:

```
import MAPI.Defs

PR_SUBJECT = MAPI.Defs.PROP_TAG(PT_STRING8, 0x0037)
type = MAPI.Defs.PROP_TYPE(PR_SUBJECT)
id = MAPI.Defs.PROP_ID(PR_SUBJECT)
```

3.2.5. MAPI.Util

MAPI.Util contains some useful functions for profile creation.

3.2.5.1. OpenECSession()

Opens a session for the given user/password pair on the specified path for a Zarafa server.

```
import MAPI
```

```
from MAPI.Util import *

session = OpenECSession('joe','password','file:///var/run/zarafa')
```

3.2.5.2. GetDefaultStore()

Opens the default store in a session.

```
import MAPI
from MAPI.Util import *

session = OpenECSession('joe','password','file:///var/run/zarafa')
store = GetDefaultStore(session)
```

3.3. Parameters

Parameters passed in python are generally equal to their c+\+ counterpart. For example, the function

```
MAPILogonEx(0, "profile", "password", 0, &lpStore)
```

is identical to the python call

```
store = MAPILogonEx(0, "profile", "password", 0)
```

There are some small differences:

- All count/arraypointer pairs are represented by a single list in python, e.g.:

  ```
  object->SetProps(2, lpSomePointer, NULL)
  ```

in python:

```
object.SetProps([prop1, prop2], None)
```

- NULL values in c+\+ are represented by *None* values in python
- All ULONG/LPENTRYID pairs are represented by a single string (containing the raw binary entryid)
- Return values in c+\+ are returned (in the order of the original c+\+ parameters) in the return value of the python call. If there is more than 1 return value, a list is returned

3.4. Errors

In the c+\+ interface, each method call returns a HRESULT containing a success or failure value. Since python is an exception-based language, all HRESULT errors are stripped from the call and ignored if there is no error. If an error has occurred, an exception of the MAPI.Exception type is raised.

Although this creates a pleasant look of the code, it does have one major drawback. In MAPI, there are *fatal* errors (which have the top bit set, so have values 0x8xxxxxxx) and *warning* errors (which do not have the top bit set). However, warnings actually behave more like success cases; the function still returns a value, but some minor part of the operation apparently failed.

Since there is no such thing as a *warning* in exceptions, currently warnings are treated exactly the same as success. Possibly future versions will change this, but will definitely not raise an exception.

3.5. Flags

Many MAPI calls have an *ulFlags* parameter that can be passed to control various things inside the call. The python interface supports all the flags that would normally be passed to the MAPI c +\+ interface, since it is *just* and int that is blindly transferred to MAPI. However there is one flag, MAPI_UNICODE, that has a direct affect on how the passed parameters are passed to MAPI. See the *character sets and unicode* part later in this document.

3.6. Methods

Method names in python are completely analogous to their c+\+ counterparts, including case.

3.7. Character sets and Unicode

Character sets are a little bit nasty. The reason for this is that we are working with three (variable) charsets in the Python bindings:

- Your terminal charset (depends on your locale, most modern OS's default to utf-8 but many still use iso-8859-1 or other local charsets)
- The sys.getdefaultencoding() charset (depends on your site.py settings but defaults to ascii)
- The internal MAPI charset (windows-1252)

What's more, a user can send information using a *string* or a *unicode* type in python.

This is the way that charsets are used:

- Since **sys.getdefaultencoding()** isn't easy to change for each application, it is not used. This in turn means that we never do any conversions between *string* and *unicode* in the python binding since this would require using **sys.getdefaultencoding()**. This would cause a lot of confusion since passing a *unicode* string without the MAPI_UNICODE flag would cause the unicode to be converted back to *string* (using ascii) and probably make python complain about the non asciiness of your unicode string, which is confusing to say the least, since the python binding itself would then have to convert from the *string* charset back into whatever charset MAPI was expecting.
- String input data is assumed to be in *Your terminal charset*
- Strings output by MAPI are in *Your terminal charset*
- When passing the MAPI_UNICODE flag in *flags* or when using the PT_UNICODE property type you **`must`** pass a unicode string (u'string'). Failure to do so will result in a raised exception.

The nice thing about this is that when you parse commandline arguments or when you are printing to the terminal, you never have to do any charset conversions. The drawback is that if you **know** that you are receiving, say, UTF-8 from some other library (eg. an XML reader), then you can do any of two things:

1. Make sure that the current locale is in utf-8 (use the *locale* command from the bash shell to check your locale)
2. Convert the utf-8 data from the other library to *unicode* strings and use the PT_UNICODE data types (and possibly MAPI_UNICODE flag, but this only affects strings in the argument list of a method call):

```
message = folder.CreateMessage(0)
s = 'some string from XML lib'
```

```
message.SetProps([SPropValue(PR_SUBJECT_W, s.decode('utf-8'))]);
```

Note

Since the release of version 7.0 Zarafa has server-wide support for unicode, but every older version only support the windows-1252 (almost identical to iso-8859-15 or Latin-1) charset internally. Which means that although using unicode strings in versions prior to 7.0 is supported, any character outside the windows-1252 charset will be converted to a questionmark symbol (*?*).

Note

The python interface has not changed with this internal change to unicode. Python programs written for Zarafa 6.30 or 6.40 will continue to work unchanged on 7.0 and upwards.

3.8. Memory management

Obviously you don't have to worry about memory management yourself. The python bindings will correctly release MAPI objects when their reference count in python reaches 0. Internal referencing inside MAPI allows for reverse-order releases:

```
folder = store.OpenEntry(None, None, 0)
message = folder.CreateMessage(0)

folder = None
message = None
```

Although the folder is released first, the fact that another message object was opened on it before prevents the actual MAPI object from being freed. Once all its children have been freed, the top-level object will free any resources used.

Getting Started

4.1. Starting a session

The first thing you do in a MAPI application is to log on to MAPI, thereby creating a session that you can use to communicate with the server. The session itself represents one user logged on to the server.

In windows, you can create *profile* via the MAPI profile editor. In linux this component is not available, and therefore you must create a profile before logging on.

The profile contains information like the type of service (Zarafa in this case), and variables like username, password, path, proxy options, etc. If you have an existing profile, it's easy to log on to such a profile:

```
import MAPI

session = MAPILogonEx(0, 'profilename', None, 0)
```

We simply pass the profile name and password (None in this case) and MAPI gives us a session object.

If you try this in linux, it will always fail; this is because linux does not keep a central log of profiles - each application starts fresh with no profiles. This means that you have to create a profile first. Since this is not a very simple operation, MAPI.Util provides us with a function to create a temporary profile, logon, and then delete the temporary profile:

```
import MAPI
from MAPI.Util import *

session = OpenECSession('username', 'password', 'http://localhost:236/zarafa')
```

This will work in both win32 and linux environments and will allow you to log on via HTTP, HTTPS or FILE (unix socket) connections.

Note

If your application is running as a trusted user (see local_admin_users setting in zarafa's server.cfg), then you can use the unix socket *file:///var/run/zarafa* to connect with zarafa without specifying a password. This allows you to run a task as the target user without knowing the user's password.

4.2. Opening a store

So now we have a session, it's time to open a store. A store is basically an entire tree of folders that the user can use. Normally each new profile has two stores: the user's own store, and the public store. Most applications will want to open the user's own store, known as the default store. Again, MAPI.Util provides a convenient function to do this, GetDefaultStore().

```
store = GetDefaultStore(session)
```

Now we have a store. The store has a set of properties like the name of the store and it's unique identifier. The store object itself is just a reference to the c+\+ object beneath so to get properties we have to use the GetProps() method to get information:

```
from MAPI.Tags import *

props = store.GetProps([PR_DISPLAY_NAME], 0)
```

You can see here that we need to import another module, MAPI.Tags. This module contains the symbolic names of all of the standard MAPI properties, like PR_DISPLAY_NAME. In reality, PR_DISPLAY_NAME is just an *int* specifying a property ID (a 16-bit integer identifier) and a type (also a 16-bit identifier). In this case, PR_DISPLAY_NAME is equal to 0x3001001e, with 0x3001 being the ID part, and 0x001e being the type (PT_STRING8).

GetProps() accepts a list of properties, but since we only want one for now, that's fine. Now we have the property, we can print the value:

```
print props[0].Value + '\n'
```

The returned value from GetProps() is a list of SPropValue() class instances, which is a simply object containing the two values of the c+\+ SPropValue structure:

- ulPropTag: The property tag of the property
- Value: The value of the property (can be int, string, double, MAPI.Util.FileTime)

The returned value in ulPropTag is the same as the requested property, PR_DISPLAY_NAME. This may seem redundant, but as we will see later, it can in some cases be different from the requested property (usually in error conditions).

The number of items in the list returned by GetProps() will always be exactly equal to the number of requested properties in the GetProps() call. You can also pass None as the property list, which will cause GetProps() to return all the properties in the object.

4.3. Opening the inbox

We said before that the store is basically a set of folders in a hierarchy. There is a function that will give us the unique identifier of the inbox: GetReceiveFolder():

```
result = store.GetReceiveFolder('IPM', 0)
inboxid = result[0]
```

The GetReceiveFolder() function actually returns two values: the unique id of the requested message class (*IPM*) and the message class that actually matched (*IPM*). There is not much point in explaining the exact meaning of these message classes since there is hardly ever any reason not to use *IPM* as the message class when you pass it to GetReceiveFolder().

Once you have that unique ID, you can use the generic OpenEntry() method of the store to open the object with that unique ID. In fact, OpenEntry() can open both folders and messages, purely depending on which unique identifier you request:

```
inbox = store.OpenEntry(inboxid, None, 0)
```

Now we have an open *inbox* object. Just like stores, an inbox also has a PR_DISPLAY_NAME property. You can get it in just the same way as the store:

```
props = inbox.GetProps([PR_DISPLAY_NAME], 0)
print props[0].Value + '\n'
```

This will show *Inbox* or your localized version of *Inbox*.

4.4. Using tables: listing messages

Most of the data inside MAPI can be viewed by using MAPI tables. MAPI tables are just what you would expect: a column/row view of a set of data. In this case, the table we want is the table of e-mail messages in the inbox. Each message in the inbox is one row in the table, and the columns are the properties of each message, for example the subject and the data.

The best way to think of the inbox's table is exactly the way that it is shown in your e-mail client: all the emails vertically and various properties of the emails horizontally.

Tables in MAPI are very flexible and allow setting different columns, they have a row cursor, allow reading arbitrary numbers of rows, support sorting and even grouping.

We can use a MAPI table to read the messages in the inbox (in fact, this is the **only** way to get the list of messages in the inbox):

```
table = inbox.GetContentsTable(0)
table.SetColumns([PR_ENTRYID, PR_SUBJECT], 0)
table.SortTable(SSortOrderSet( [ SSort(PR_SUBJECT, TABLE_SORT_ASCEND) ], 0, 0), 0);
rows = table.QueryRows(10, 0)
```

This needs some more explanation. The first line calls GetContentsTable() to get the table associated with the inbox. The SetColumns() call then selects two columns that we are interested in: PR_ENTRYID (the unique ID of each message) and PR_SUBJECT (the subject of each message). We could have requested much more properties, but requesting more properties is slower because it needs more bandwidth, i/o, cache, etc. from the server.

We then sort the table by creating an SSortOrderSet. The SSortOrderSet constructor accepts a list of the columns to sort by (note that the columns used for sorted are not necessarily in the viewable column set set by SetColumns()). Each property can be sorted ascending (TABLE_SORT_ASCEND) or descending (TABLE_SORT_DESCEND). The two zero's passed to the SSortOrderSet constructor are how many columns you wish to use for grouping, and how many of those groups should be expanded, but we're not using that right now.

The last line actually requests the data from the server. The QueryRows() call requests 10 rows. Since we just openen the table, it will give us the first 10 rows. A subsequent call to QueryRows() will give us the next 10 rows, etc.

The rows are returned as a two-dimensional array: an array of arrays of SPropValues. Processing them gives you an idea of what it looks like:

```
for row in rows:
  print row[1].Value + '\n'
```

It is important to note that the order of the properties in the row output is always exactly equal to the order of properties passed to SetColumns(). In this case we know that row[1] is the PR_SUBJECT property, and we can therefore print it in the way described above.

4.5. Opening a message

The table described above is great for getting information for lots of e-mails at once, but it has some (designed) limitations:

- You can only see properties of the email themselves, not the attachments or recipients of the e-mails
- Strings are truncated at 255 bytes

The reason for this is that tables are designed for summary overviews of e-mails, not for showing the details of a single message (eg when a user clicks on a message)

To get the message details, we need to open the message itself. We said before that we can use the OpenEntry() method of the store to open both folders and messages, and that's just what we're going to do now:

```
for row in rows:
  message = store.OpenEntry(row[0].Value, None, 0)
  props = message.GetProps(PR_SUBJECT)
  print props[0].Value + '\n'
```

This will open each message, and then use the GetProps() method again to get the PR_SUBJECT property. This is a little redundant since we already read the PR_SUBJECT from the table before. The only difference between this PR_SUBJECT and the PR_SUBJECT retrieved from the table is that this PR_SUBJECT is not truncated at 255 bytes.

We can now use the *message* object to do more interesting things.

4.6. Getting attachments

Each message (email) has a list of attachments. To get the list of attachments for an email, we can use the GetAttachmentTable() method on the message. As the name implies, this will return a MAPI table, exactly like the one we used before, but this time the rows will be the attachments, and the columns will be the properties (like before)

```
t = message.GetAttachmentTable(0)

t.SetColumns([PR_ATTACH_FILENAME, PR_ATTACH_NUM], 0)
rows = t.QueryRows(10,0)
```

As before, we can read the attachments by querying a certain column set, and then reading the rows with QueryRows(). The PR_ATTACH_NUM column can be used to open the actual attachment:

```
attach = message.OpenAttach(rows[0][1].Value, None, 0)

props = attach.GetProps([PR_ATTACH_FILENAME], 0)
```

Again, the attachment object itself can be queried with GetProps() to retrieve properties.

The data inside the attachment is just another property: PR_ATTACH_DATA_BIN. Logically you'd expect to be able to call GetProps() passing that property tag (PR_ATTACH_DATA_BIN) to get the attachment data. However, this would be a bad idea - if the attachment is big, you'd get a huge SPropValue value with several megabytes of data in the *Value* property of the SPropValue object.

To avoid this, MAPI doesn't allow you to get more than 8k of data via the GetProps() interface. If you have any more than that, you will get an error. Not just any error though, the GetProps() call will succeed as usual. However, the SPropValue returned will contain special information:

- ulPropTag will contain the same property ID, but the property type will be PT_ERROR. eg. If you request PR_SUBJECT (0x0037001e), and it is not available for any reason, you will get 0x0037000a (note that only the lower 16 bit, the property type, are different)
- Value will contain the MAPI error value. When the value was not returning because it was larger than 8k, the error will be MAPI_E_NOT_ENOUGH_MEMORY, if the property was not found at all, the error value will be MAPI_E_NOT_FOUND

To get the data, we have to use a *stream*. The MAPI stream is just like any other stream, allowing read/write/seek access to a sequential stream of data. We open the stream with the OpenProperty() call:

```
import MAPI.Guid
stream = attach.OpenProperty(PR_ATTACH_DATA_BIN, IID_IStream, 0, 0)
```

The OpenProperty() method takes the property tag you wish to open, an interface identifier (in this case we want to open a stream), and some flags.

We can then use the stream to read the attachment data in 4k blocks:

```
data = ''
while True:
  d = stream.Read(4096)
  if len(d) == 0: break
  data += d
```

This negates the effect of not having all that data in memory, but it illustrates the use of the stream pretty well.

4.7. Writing data

Until now we have only been reading data from MAPI. The most important ways of writing data to MAPI are:

- message.SetProps(): the opposite of GetProps()
- message.SaveChanges(): save changes done with GetProps()
- message.CreateAttach(): create a new attachment
- folder.CreateMessage(): create a new message

However, if you were to simply take the code we have created until now and added any of these functions, an exception would be raised. This is because all the flags we have been passing were 0, and most functions default to read-only access. The calls we have to change are:

- OpenEntry(entryid, None, **MAPI_MODIFY**) : Open a message or folder with write permissions
- OpenProperty(proptag, IID_IStream, 0, **MAPI_MODIFY**): Open an existing stream with write permissions
- OpenProperty(proptag, IID_IStream, 0, **MAPI_MODIFY|MAPI_CREATE**): Open a new stream with write permissions, or truncate the existing stream

This allows you to write to the objects:

```
message.SetProps([SPropValue(PR_SUBJECT, 'new subject')])
```

```
stream.Write('hello')
```

Note

Although changes on messages are not saved to disk until SaveChanges() is called on messages, changes on folders and stores are immediate and do not require a SaveChanges() call.

4.8. Recipients

Recipients, like attachments, can be accessed through a table. You can open a message's recipient table with the GetRecipientTable() method. Again, the columns are properties, but this time the rows are the recipients. Each recipient has a PR_DISPLAY_NAME (eg *Joe Jones*), a PR_EMAIL_ADDRESS (eg *joe@jones.com*) and a PR_RECIPIENT_TYPE (MAPI_TO, MAPI_CC, MAPI_BCC).

```
table = message.GetRecipientTable(0)
table.SetColumns([PR_DISPLAY_NAME, PR_EMAIL_ADDRESS], 0)
```

There are also some properties on the message that can be used to get a summary of the recipients. These are called PR_DISPLAY_TO, PR_DISPLAY_CC, PR_DISPLAY_BCC. They are simply PR_DISPLAY_NAME of all the recipients concatenated together. Since the properties are generated from the information in the recipient table, you cannot write these recipients.

If you wish to modify a row in the recipient table, you can do so with the ModifyRecipients() method:

```
message.ModifyRecipients(0, [ [
  SPropValue(PR_DISPLAY_NAME, 'Joe'),
  SPropValue(PR_EMAIL_ADDRESS, 'joe@domain'),
  SPropValue(PR_RECIPIENT_TYPE, MAPI_TO) ] ])
```

The format of the passed recipients array is exactly the same as a set of rows returned by QueryRows(): an array of arrays of SPropValues.

This will replace the recipient table with the passed recipients. This allows you to add multiple recipients at once, or clear the recipient table (by passing 0 recipients). The PR_DISPLAY_* properties on the message will then automatically be updated.

Like properties, changes in the recipient table are not actually saved until the SaveChanges() method is called.

4.9. Restrictions

Tables normally show rows for all the messages in a folder. MAPI also provides the possibility to filter the messages according to a restriction. A restriction is a tree of expressions including AND, OR and other expressions. Each message that matches the expression is included in the table, and each message that does not match is discarded.

Since the restriction only applies to a view of the messages, the restriction can be switched easily from a full view to a restricted view and back within the same table instance:

```
table.Restrict(SRestriction(RES_PROPERTY, RELOP_EQ, PR_SUBJECT, SPropValue(PR_SUBJECT, 'this
 subject')))
```

This will restrict the view to messages with a PR_SUBJECT of *this subject*. If there are no matching messages, no rows will be available.

4.10. Hierarchy

Until now we have only worked with the Inbox folder. MAPI supports a hierarchy of folders, including Inbox, Outbox, Sent Items, etc. The visible part of the tree (the part you see in the webaccess) is actually not the root of the folder structure, the part of the visible tree starts at a folder called the IPM_SUBTREE.

To get the entry id (unique identifier) of this folder, we can query a property of the store:

```
props = store.GetProps([PR_IPM_SUBTREE_ENTRYID], 0)
entryid = props[0].Value

ipmsubtree = store.OpenEntry(entryid)
```

The IPM_SUBTREE folder can now be used to get the folder hierarchy. This is accomplished by opening the hierarchy table of the folder. The hierarchy table is just like the contents table used earlier, except that the hierarchy table represents **folders** instead of **messages**.

```
table = ipmsubtree.GetHierarchyTable(0)
table.SetColumns([PR_DISPLAY_NAME, PR_ENTRYID], 0)
rows = table.QueryRows(10,0)
```

To get a view of all the folders, we need to open each folder, and for each folder loop through its children folders by examining the hierarchy table of each folder.

4.11. Default folders

Several folders like inbox, outbox, calendar, etc have a special status - they are *default folders*. MAPI clients will assign special behaviour to these folders. For example, the default folders cannot be renamed, deleted or moved. This limitation is completely client-enforced - MAPI itself doesn't really care if you delete your inbox.

The default folders can be identified by several properties that contain the unique ID for those folders:

```
props = store.GetProps([PR_IPM_OUTBOX_ENTRYID, PR_IPM_TRASH_ENTRYID], 0)
result = store.GetReceiveFolder('IPM', 0)
inboxid = result[0]
inbox = store.OpenEntry(inboxid, None, 0)
props = inbox.GetProps([PR_IPM_APPOINTMENT_ENTRYID, PR_IPM_TASKS_ENTRYID,
 PR_IPM_CONTACTS_ENTRYID,
                                                    PR_IPM_JOURNAL_ENTRYID, PR_IPM_NOTE_ENTRYID],
 0)
```

As you can see, some properties for the default folders are set on the store, while others are set on the inbox object. The inbox object itself cannot be found by using a PR_IPM_* property, but only by using the GetReceiveFolder() function.

Warning

Removing, moving or even renaming default folders can cause your MAPI applications to malfunction. It can also cause your MAPI client to generate a new set of folders, effectively hiding your old folders from the user.