pdating an Application over the Internet

Doug Hennig

Ever wanted to update your client's applications with minimal effort? This month's article, the first of two on this topic, presents a set of classes that download an application's files from an FTP site, automating the process of distributing application updates.

Based on the emails I received and messages I've seen in various FoxPro forums, my October 1997 column, which presented a utility I called STARTAPP, hit a nerve. To refresh your memory, STARTAPP is an application "loader"; rather than running your application's EXE from a server, the user runs STARTAPP.EXE (which could be renamed to something like MYAPP.EXE) on their workstation. STARTAPP compares the application's files (such as EXEs and DLLs) on the workstation with those in a defined directory on a server, and copies newer files from the server to the workstation. It then runs the main program (APP or EXE) for your application. There are several advantages to using STARTAPP, the most important of which are the ability to run the application on a workstation rather than a server (for performance but also perhaps portability reasons) while providing an easy way to update the workstations (avoiding going to each one to update the files manually), and an easy way to install a new version of an application without kicking all the users out first (which would be required if they were running the application off the server). I've used STARTAPP with nearly every application I've delivered since I wrote it.

Since writing STARTAPP, I've encountered a similar yet different need: the ability to update an application's files remotely. For clients too far away to visit on-site, I usually zip the files making up a new revision of an application and email the ZIP file to the application's "administrator" at the client site. There are several common problems I've encountered, however:

- *File size*: even zipped, the files making up an application can be several MB. Some clients have their email systems configured to reject messages larger than a certain size.
- *Too many complicated steps*: the user has to save the ZIP file attachment to a directory, open Windows Explorer, navigate to that directory, find the file they saved, double-click on it to run an unzipping program like WinZip (assuming they have such a program), then extract the files. Believe it or not, in this day and age, there are still many people who have no idea how to do any of these steps. And if the user makes one mistake in this process, the update won't be properly installed.
- *Too busy, too lazy, or too scared*: you send the update, go over all the installation steps with the user, and a week later, they report bugs you know you've fixed. When you have them check the file size, date, or version number, it turns out they never got around to installing the update.

A different approach is to burn a CD and ship it to the client. Except for the media (CD vs. floppy), this is how we've done it since the 80's, and it has all the frailties you'd expect: the CD wasn't burned properly or is incompatible with the client's drive, the CD was damaged in shipping, the CD took forever to get there, the CD never arrived …

A better way to install an updated application would be to have an Internet version of STARTAPP. That is, an application loader would automatically check an FTP site to see if newer versions of the application's files are available, download them if so, and then run the main program for the application. The beauty of this approach is that when you want to ship a new version of an application, you simply install some files on an FTP server. The next time your client runs the application, they'll get the new version automatically.

That's what this and next month's articles are about. This month, we'll look at some classes that handle the process of copying files from a server (LAN/WAN or FTP), and next month, we'll look at several strategies for driving these classes and updating your application's files. Before we dig into this month's code, I'd like to thank Willie van Schalkwyk for suggesting this idea and helping flesh out the basic concepts.

SFUpdate

SFUpdate, in SFUPDATE.VCX, is based on the VFP Custom base class rather than SFCustom in SFCTRLS.VCX (as would normally be the case) to avoid problems with several instances of SFCTRLS floating around (one in the loader program and another in the main program). SFUpdate is an abstract class; we'll look at a couple of subclasses that implement updates from an LAN or WAN server and from an FTP site. Table 1 shows the custom properties of SFUpdate.

Property	Description
aFiles	An array of files to update from the server.
aServerFiles	An array of information about files on the server (Protected).
cErrorMessage	The error message if something went wrong.
cPassword	The password to access the server.
cScriptFile	The name of an optional VFP program that "scripts" the behavior of various stages of the updating process.
cServerDirectory	The directory on the server where the files can be found.
cServerName	The name of the server.
cTitle	The caption for any status window.
cUserName	The user name to access the server.
lCheckVersion	.T. (the default) to compare the workstation and server versions of a file; .F. to force the files to be copied from the server.
lPrompted	.T. if the user has been prompted for the download (Protected).
lPromptForDownload	.T. (the default is .F.) to prompt the user if they want to proceed with the file transfer.
lQuiet	.T. (the default is .F.) to suppress status display.
lTerminateOnError	.T. (the default is .F.) to stop the update process if an error occurs.
nCurrentServerFile	An index into aServerFiles for the current server file (Protected).

Table 1. Properties of SFUpdate.

UpdateLocal is the main method in SFUpdate. It checks each file in the aFiles property and if necessary, copies it to the workstation. It's mostly a template method; that is, it directs other methods about how this process should work. Let's look at it in logical sections. First, if the cScriptFile property contains the name of a procedure file (this property should include the path and an FXP extension), that file is opened with SET PROCEDURE TO. This allows you to "script" certain parts of the update process; in other words, without having to create a subclass, you can tell an instance of SFUpdate how to behave at certain stages by simply providing a procedure file with the appropriate code. This can be used for client-specific code, for example. You could even generate the PRG and compile it on the fly if desired.

```
local lcScriptFile, ;
  llScriptFile, ;
  llReturn, ;
  lnFiles, ;
  lnI, ;
  lcFile, ;
  llOK
with This
  if not empty(.cScriptFile) and file(.cScriptFile)
    if '\' $ .cScriptFile
      lcScriptFile = upper(.cScriptFile)
    else
      lcScriptFile = '\' + upper(.cScriptFile)
    endif '\' $ .cScriptFile
    lcScriptFile = forceext(lcScriptFile, '.')
    if not lcScriptFile $ upper(set('PROCEDURE'))
      llScriptFile = .T.
      set procedure to (.cScriptFile) additive
    endif not lcScriptFile $ upper(set('PROCEDURE'))
  endif not empty(.cScriptFile) ...
```

Next, the BeforeUpdate method is called. BeforeUpdate, AfterUpdate, BeforeCopyFile, and AfterCopyFile are hook methods that allow you to perform additional processing in the script file or in a

subclass of SFUpdate (we'll see how this might be used when we look at the SFInternetUpdate class later). In SFUpdate, these methods run the function of the same name in the script file if one is being used, and return .T. if everything went OK. The Before methods also allow you to terminate the process under some conditions (such as if an error occurs); SFUpdate won't continue processing if the BeforeUpdate method returns .F., and will stop if the BeforeCopyFile method returns .F. and the lTerminateOnError property is .T.

If BeforeUpdate returns .T., GetServerFileInfo is called to put information about the files in the server directory (specified in cServerDirectory) into the aServerFiles array property. If that method (which is empty in this class) returns .T., each file in the aFiles array is processed in a loop. If the lQuiet property is .F., which means we should display status messages, the DisplayStatus method is called to display information about the process.

```
llReturn = .BeforeUpdate()
if llReturn
  llReturn = .GetServerFileInfo()
  if llReturn
    lnFiles  = alen(.aFiles)
    for lnI = 1 to lnFiles
      lcFile = .aFiles[lnI]
      if not empty(lcFile)
        if not .lQuiet
          .DisplayStatus('Checking server and ' + ;
            'workstation versions of ' + lcFile + ;
            '...')
        endif not .lQuiet
```

Next, CheckVersion is called to determine if we're supposed to copy the file from the server to the workstation; we'll look at that method later. If it returns .F., meaning the workstation file is not current, and the server copy of the file exists (in which case, the nCurrentServerFile property will contain the index to the file in the aServerFiles array), we'll copy the file. DisplayStatus is called to update the status display and if we're supposed to prompt the user (lPromptForDownload is .T.) and we've haven't already done so (lPrompted is .F.), the PromptForDownload method is called. If that method returns .F., the user decided not to update to the latest version, so we'll terminate the process. Otherwise, the BeforeCopyFile hook method is called (which allows you to cancel the process for any file-specific reason or perform any other pre-copy processing) and then the CopyFile method (which is empty in this class) is used to perform the actual file copy.

```
        if not .CheckVersion(lcFile) and ;
          .nCurrentServerFile <> 0
          if not .lQuiet
            .DisplayStatus('Updating workstation ' + ;
              'version of ' + lcFile + '...')
          endif not .lQuiet
          if .lPromptForDownload and not .lPrompted
            .lPrompted = .T.
            llReturn   = .PromptForDownload()
            if not llReturn
              exit
            endif not llReturn
          endif .lPromptForDownload ...
          llOK     = .BeforeCopyFile(lcFile) and ;
            .CopyFile(lcFile)
          llReturn = llReturn and llOK
          do case
```

If the file was successfully copied, the AfterCopyFile hook method is called. This method (or the AfterCopyFile procedure in a script file that it calls) could, for example, unzip a ZIP file that was copied or update a status display such as a progress thermometer. If the file copy failed and the lTerminateOnError property is .T., the process is terminated.

```
            case llOK
              .AfterCopyFile(lcFile)
            case .lTerminateOnError
```

```
            exit
          endcase
        endif not .CheckVersion(lcFile) ...
      endif not empty(lcFile)
    next lnI
```

After all files have been processed successfully, the AfterUpdate hook method is called, and if lQuiet is .F., ClearStatus is used to clear any status display. Finally, if a script file was opened at the beginning of UpdateLocal, it's closed with RELEASE PROCEDURE.

```
    if llReturn
      .AfterUpdate()
    endif llReturn
  endif llReturn
endif llReturn
if not .lQuiet
  .ClearStatus()
endif not .lQuiet
if llScriptFile
  release procedure (.cScriptFile)
endif llScriptFile
endwith
return llReturn
```

The CheckVersion method is called by UpdateLocal to determine if we should copy the file from the server to the workstation or not. If lCheckVersion is .T. (set it to .F. to force all workstation files to be updated regardless of whether they need it or not), it calls GetLocalFileInfo to get information about the local copy of the file, and compares the file sizes (which must be the same) and last modified DateTimes (the workstation version must be the same or newer than the server, because installing a file on the workstation sets the file to the current DateTime). If the files are the same, it returns .T., indicating that we won't copy the file from the server. This method also sets the nCurrentServerFile property to the index for the current file into the aServerFiles array so other methods can retrieve information about the file from the array.

```
lparameters tcFile
local laLocal[1], ;
  lnLocalFiles, ;
  llLocalFileExists, ;
  lnServerFile, ;
  llServerFileExists, ;
  llReturn
with This

* Get the version info for the server and workstation
* copies of the file (the server file information is in
* This.aServerFiles).

  lnServerFile        = ascan(.aServerFiles, ;
    upper(tcFile))
  llServerFileExists  = lnServerFile > 0
  .nCurrentServerFile = iif(llServerFileExists, ;
    asubscript(.aServerFiles, lnServerFile, 1), 0)
  if .lCheckVersion
    lnLocalFiles      = .GetLocalFileInfo(tcFile, ;
      @laLocal)
    llLocalFileExists = lnLocalFiles > 0
  endif .lCheckVersion

* Return .T. if the file exists on both the server and
* workstation, they are the same size, and the
* workstation version is the same or a later DateTime
* than the server version.

  llReturn = llServerFileExists and ;
    llLocalFileExists and ;
    .aServerFiles[.nCurrentServerFile, 2] = ;
```

```
      laLocal[1, 2] and ;
    .aServerFiles[.nCurrentServerFile, 3] <= ;
      laLocal[1, 3]
endwith
return llReturn
```

DisplayStatus and ClearStatus are abstract methods that, in a subclass, can show status information about the process. DisplayStatus is called by UpdateLocal at various places in the processing, and is passed a message about what process is about to occur. ClearStatus is called at the end of UpdateLocal to clear any message displayed by DisplayStatus.

GetLocalFileInfo is called by CheckVersion. Its purpose is to fill the passed array with information about the specified file. It uses ADIR() to get the file name, size, date, and time, and calls GetDateTime to convert the date and time into a DateTime value.

```
lparameters tcFile, ;
  taFiles
local laFiles[1], ;
  lnFiles, ;
  lnI
lnFiles = adir(laFiles, tcFile)
if lnFiles > 0
  dimension taFiles[lnFiles, 3]
  for lnI = 1 to lnFiles
    taFiles[lnI, 1] = laFiles[lnI, 1]
    taFiles[lnI, 2] = laFiles[lnI, 2]
    taFiles[lnI, 3] = This.GetDateTime(laFiles[lnI, 3], ;
      laFiles[lnI, 4])
  next lnI
endif lnFiles > 0
return lnFiles
```

The final method, PromptForDownload, simply asks the user if they want to download the updated files for the application, and returns .T. if they chose Yes.

That's it for SFUpdate. As I mentioned, it's an abstract class, so I created a couple of subclasses. SFLANUpdate (which we won't look at here) is for updating a workstation from a LAN or WAN server (similar to the purpose of STARTAPP) and SFInternetUpdate is for updating from an FTP site over the Internet.

SFInternetUpdate

SFInternetUpdate only overrides a few methods of SFUpdate, but that's because it collaborates with another class to do the actual work of downloading information and files from an FTP site. I originally intended to use the Microsoft Internet Transfer ActiveX control; however, after fussing with it for a while, I decided it wasn't worth the effort, given that an inexpensive third party utility I was already using was much easier to implement. wwIPStuff, from West Wind Technologies (www.west-wind.com), is a shareware ($99 to register) set of classes that provide a wide variety of Internet utilities. I discussed the email services in my March 2000 article ("Spam, Wonderful Spam"); this month, we'll use the FTP services.

Before we look at SFInternetUpdate, let's take a peek at a subclass I created of the wwFTP class that comes with wwIPStuff. SFFTP, in SFIPSTUFF.VCX, overrides a few methods. aFTPDir, which is the FTP equivalent of ADIR(), returns 0 rather than an error code if the method failed; this makes it consistent with the success return value, which is the number of files matching the desired file specification. FTPGetFileEx, which retrieves a file from an FTP site, accepts both source and target file names. In SFFTP, if the target name isn't specified, it defaults to the same name as the source file. Also, although wwFTP has a method, UnzipFiles, to unzip a ZIP file (using DynaZip from Inner Media, www.innermedia.com) and a property, lUseZip, that indicates if that method should be used, it doesn't actually call UnzipFiles after downloading a file. In SFFTP, FTPFileGetEx calls UnzipFiles if the file was successfully downloaded, has a ZIP extension, and lUseZip is .T., then erases the ZIP file if it was successfully unzipped. Finally, OnFTPBufferUpdate, which is called periodically as a file is uploaded or downloaded, displays a WAIT WINDOW NOWAIT message showing the number of bytes transferred so far and a percentage of the total.

This method uses two new properties, lQuiet (.T. to suppress the status display) and nFileSize (the size of the file being transferred).

Back to SFInternetUpdate. BeforeUpdate, which is called before the update process begins, instantiates an object from the class specified in the cFTPClass and cFTPLibrary properties (which contain "SFFTP" and "SFIPSTUFF.VCX" by default) into the oFTP property, sets several properties for the object, and tries to connect to the FTP server. It returns .T. if it succeeded or .F. if not; the cErrorMessage property contains information about what went wrong if it failed.

```
local llReturn, ;
  lnResult
with This
  if vartype(.oFTP) = 'O'
    llReturn = dodefault()
  else
    if not '\WWUTILS.FXP' $ upper(set('PROCEDURE'))
      set procedure to wwUtils additive
    endif not '\WWUTILS.FXP' $ upper(set('PROCEDURE'))
    .oFTP = newobject(.cFTPClass, .cFTPLibrary)
    .oFTP.cUserName = .cUserName
    .oFTP.cPassword = .cPassword
    .oFTP.cTitle    = .cTitle
    lnResult = .oFTP.FTPConnect(.cServerName)
    llReturn = lnResult = 0
    if llReturn
      llReturn = dodefault()
    else
      .cErrorMessage = .cErrorMessage + ;
        iif(empty(.cErrorMessage), '', chr(13)) + ;
        'BeforeUpdate: ' + iif(empty(.oFTP.cErrorMsg), ;
        'Cannot connect to Internet update site', ;
        .oFTP.cErrorMsg)
    endif llReturn
  endif vartype(.oFTP) = 'O'
endwith
return llReturn
```

GetServerFileInfo uses the aFTPDir method of the oFTP object to fill the aServerFiles property with the files on the FTP site and returns .T. if it succeeded.

```
local laFiles[1], ;
  lnFiles, ;
  llReturn, ;
  lnI
with This
  if empty(.aServerFiles[1, 1])
    lnFiles  = .oFTP.aFTPDir(@laFiles, ;
      .cServerDirectory + '*.*')
    llReturn = lnFiles > 0
    if llReturn
      dimension .aServerFiles[lnFiles, 3]
      for lnI = 1 to lnFiles
        .aServerFiles[lnI, 1] = upper(laFiles[lnI, 1])
        .aServerFiles[lnI, 2] = laFiles[lnI, 2]
        .aServerFiles[lnI, 3] = laFiles[lnI, 4]
      next lnI
    else
      .cErrorMessage = .cErrorMessage + ;
        iif(empty(.cErrorMessage), '', chr(13)) + ;
        'GetServerFileInfo: ' + .oFTP.cErrorMsg
    endif llReturn
  else
    llReturn = .T.
  endif empty(.aServerFiles[1, 1])
endwith
return llReturn
```

CopyFile first sets the nFileSize property of the oFTP object to the size of the file to be downloaded (which it gets from aServerFiles) and erases the workstation copy of the file (because wwIPStuff gives an error if the file already exists), then uses the FTPGetFileEx method of the oFTP object to download the file.

```
lparameters tcFile
local lnResult, ;
  llReturn
with This
  .oFTP.lQuiet    = .lQuiet
  .oFTP.nFileSize = .aServerFiles[.nCurrentServerFile, 2]
  erase (tcFile)
  lnResult = .oFTP.FTPGetFileEx(.cServerDirectory + ;
    justfname(tcFile))
  llReturn = lnResult = 0
  do case
    case not llReturn
      .cErrorMessage = .cErrorMessage + ;
        iif(empty(.cErrorMessage), '', chr(13)) + ;
        'CopyFile: ' + .oFTP.cErrorMsg
    case .oFTP.lCancelDownload
      llReturn = .F.
  endcase
endwith
return llReturn
```

Trying it Out

TESTUPDATE.PRG demonstrates how the SFInternetUpdate class works. It downloads the DIRMAP.TXT file from Microsoft's FTP site; running it a second time won't retrieve the file again since it already exists on the workstation and hasn't been updated on the server. Notice that some properties, such as cServerDirectory, cUser, and cPassword, aren't set because they aren't needed in this case; they might be required for your FTP site, however. Before running this program, copy the following wwIPStuff files to the directory where TESTUPDATE.PRG is located: WWIPSTUFF.VCX, WWIPSTUFF.VCT, WUTILS.PRG, and WCONNECT.H.

Conclusion

So far, we have the basics for an Internet update process. Using code similar to that in TESTUPDATE.PRG, you could implement an application update utility right now. However, next month, we'll look at a few strategies for driving SFInternetUpdate and downloading an application's files. One will involve using a loader program similar to STARTAPP, while another will implement the majority of the functionality in the main program itself, delegating only the final step to a separate EXE (since we can't overwrite the running EXE).

Doug Hennig is a partner with Stonefield Systems Group Inc. in Regina, Saskatchewan, Canada. He is the author or co-author of Stonefield's add-on tools for FoxPro developers, including Stonefield Database Toolkit, Stonefield Query, and Stonefield Reports. He is also the author of "The Visual FoxPro Data Dictionary" in Pinnacle Publishing's "The Pros Talk Visual FoxPro" series. Doug has spoken at the 1997, 1998, and 1999 Microsoft FoxPro Developers Conferences (DevCon) as well as user groups and regional conferences all over North America. He is a Microsoft Most Valuable Professional (MVP) and Microsoft Certified Professional (MCP). He can be reached at dhennig@stonefield.com.