pdating an Application over the Internet, Part II

Doug Hennig

This second of a two-part series looks at several strategies for automating the process of distributing application updates by downloading an application's files from an FTP site.

Last month, I presented a set of classes that download an application's files from an FTP site. To refresh your memory, SFUpdate is an abstract class that provides the logic of deciding which files to copy from a server to a workstation, copying only those files that've been updated on the server. SFInternetUpdate is a subclass that's specific for downloading files from an FTP site. It collaborates with SFFTP, a subclass of West Wind Technologies' wwFTP class (part of the shareware wwIPStuff; www.west-wind.com), which handles the Internet communication and actual file transfer.

These classes provide the basics for an Internet update process. This 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 the STARTAPP utility I presented in my October 1997 column, while another will implement the majority of the functionality in the application's EXE itself, delegating only the final step to a separate EXE.

STARTAPP.PRG

Rather than running your application's EXE, the user runs a "loader" EXE, which checks if a newer version of the application EXE is available, downloads it if so, and then runs it. The reason for using a loader EXE is that we can't overwrite the running program, so we can't have the application EXE download a new version of itself.

The loader EXE uses a configuration table, CONFIG.DBF, to specify how it should do its job: what server and where on that server to look for the updated files, what EXE to run after it's done, if messages should be displayed, etc. Table 1 shows the structure of this table. CONFIG.DBF is built into the loader EXE so it's one less file to get deleted or corrupted, but could be provided as a separate file as well.

Field	Description
UpdateFile	The name of a text file containing a list of files (delimited with carriage returns) to update from server copies. Note: the first line of this file could be "VERSION=" followed by a version number to compare to the current version of the application
Quiet	.T. to not display any progress messages
Prompt	.T. to prompt if the user wants to download the update
ServerName	The name of the server where application files are found. Note: this can be left blank for LAN/WAN updates; it's only required for Internet updates
ServerPath	The server directory where application files are found
UserName	The user name for the server. Note: this can be left blank for LAN/WAN updates
Password	The password for the server. Note: this can be left blank for LAN/WAN updates
Title	The title for MESSAGEBOX dialogs; usually the full name of the application as the user knows it
AppName	The name of the application to run after the update
UpdClass	The update class to instantiate
UpdLibrary	The library the class specified in UpdClass is contained in

Table 1. Fields in CONFIG.DBF.

The main program for the loader EXE is STARTAPP.PRG. Let's look at it in pieces (we'll skip the mundane stuff at the start like header comments and locals declarations). First, the program tries to find the configuration table; if it can't, it displays an error and exits. If it can, it opens the table, reads the contents into variables, and closes it again.

```
lparameters tcParameters
lcConfig = 'CONFIG.DBF'
if not file(lcConfig)
```

```
  lcDirectory = justpath(sys(16))
  lcConfig    = lcDirectory + lcConfig
  if not file(lcConfig)
    messagebox('Configuration file cannot be located.', ;
      MB_OK + MB_ICONSTOP, 'Application Error')
    return .F.
  endif not file(lcConfig)
endif not file(lcConfig)
select 0
use (lcConfig) again shared
scatter memvar memo
use
```

Next, STARTAPP instantiates an updater object (SFLANUpdate for a LAN/WAN server, SFInternetUpdate for an FTP server) and configures it to copy a text file containing a list of the application's files from the server. lPromptForDownload is set to .F. so the user isn't prompted, lCheckVersion is set to .F. to force the file to be copied without checking if the server version is more current, and lQuiet is set to .T. so no progress messages are displayed. These settings allow this file to be retrieved with nothing shown to the user at all. Finally, UpdateLocal is called to retrieve the file.

```
loUpdate = newobject(m.UpdClass, m.UpdLibrary)
with loUpdate
  dimension .aFiles[1]
  .aFiles[1]          = m.UpdateFile
  .cServerName        = m.ServerName
  .cServerDirectory   = m.ServerPath
  .cUserName          = m.UserName
  .cPassword          = m.Password
  .cTitle             = m.Title
  .lPromptForDownload = .F.
  .lCheckVersion      = .F.
  .lQuiet             = .T.
  llReturn            = .UpdateLocal()
```

If we received the file, its contents are read into the laFiles array, the file is deleted, and the first line is examined. If it contains "VERSION=", the version number of the application on the server is read from the rest of the line into lcVersion, that line is removed from the array (since it isn't the name of a file to possibly copy), and llVersion is set to .T. so we know we'll do a version number check. Why check version numbers when, as we saw last month, the updater object can compare file sizes and DateTimes to determine if we should copy files or not? It's just another level of control you have over the update process.

```
  do case
    case llReturn
      lnFiles = alines(laFiles, filetostr(m.UpdateFile))
      erase (m.UpdateFile)
      if upper(laFiles[1]) = 'VERSION='
        lcVersion = upper(alltrim(substr(laFiles[1], 9)))
        adel(laFiles, 1)
        lnFiles = lnFiles - 1
        dimension laFiles[lnFiles]
        llVersion = .T.
      endif upper(laFiles[1]) = 'VERSION='
```

If the main application program exists, STARTAPP tries to read its version number using AGETFILEVERSION() (see the sidebar "Assigning a Version Number to a VFP EXE"). If version information exists for the file, STARTAPP formats the version number and puts it into lcCurrVersion. The reason for formatting the number is that even though I enter "05" for the minor build portion of the version number, it shows up as just "5", so I left-pad the value with zeros to two places so it's formatted as I like; adjust this code to suit your own needs.

```
      if file(m.AppName)
        lnItems = agetfileversion(laVersion, m.AppName)
        if lnItems >= 4
          lnPos1 = at('.', laVersion[4])
```

```
        lnPos2 = at('.', laVersion[4], 2)
        lcCurrVersion = left(laVersion[4], lnPos1) + ;
          padl(substr(laVersion[4], lnPos1 + 1, ;
            lnPos2 - lnPos1 - 1), 2, '0')
      endif lnItems >= 4
    endif file(m.AppName)
```

If we didn't get a version number for the server or workstation files or if the server copy's version number is newer than the workstation's, we have to copy files from the server, so the laFiles array is put into the aFiles array of the updater object and the lQuiet and lPromptForDownload properties are set to the configuration settings. lCheckVersion is set to .T. (meaning we'll check file sizes and DateTimes to determine if we should copy files or not) if we didn't get a version number for the server or workstation files; that way, if we did get version information, we won't bother checking these things since we know we have to copy the files. Then, STARTAPP calls UpdateLocal to retrieve the files.

```
    if not llVersion or empty(lcVersion) or ;
      empty(lcCurrVersion) or lcVersion > lcCurrVersion
      dimension .aFiles[lnFiles]
      acopy(laFiles, .aFiles)
      .lQuiet              = m.Quiet
      .lPromptForDownload = m.Prompt
      .lCheckVersion       = not llVersion or ;
        empty(lcCurrVersion)
      llReturn             = .UpdateLocal()
```

If UpdateLocal failed for some reason, STARTAPP displays a simple error message with the option to display more detailed information (which, like the "details" section of a GPF dialog, isn't useful to the user but may be to the support person that user will contact). Due to a lack of space, we won't look at this code.

We're done the update process, so whether files were copied or not, it's time to run the "real" EXE or APP for the application. If any parameters were passed to STARTAPP, it passes them to the application.

```
if not empty(m.AppName) and (file(m.AppName) or ;
  file(forceext(m.AppName, 'EXE')) or ;
  file(forceext(m.AppName, 'APP')))
  lcRunDir = justpath(m.AppName)
  if not empty(lcRunDir)
    cd ('"' + lcRunDir + '"')
  endif not empty(lcRunDir)
  if empty(tcParameters)
    do (m.AppName)
  else
    do (m.AppName) with &tcParameters
  endif empty(tcParameters)
endif not empty(m.AppName) ...
```

To use STARTAPP as the loader EXE for an application, create a new project and add STARTAPP.PRG as the main program. Add CONFIG.DBF (marked as included and with the field values set as desired), SFIPSTUFF.VCX, SFUPDATE.VCX, WWIPSTUFF.VCX, and WWUTILS.PRG (the latter two files are part of wwIPStuff, so they aren't included with this month's Subscriber Downloads). Build an EXE and use it as the startup executable for your application (don't forget to include WWIPSTUFF.DLL in the application directory). If you want to create a generic startup EXE that can be used with any application, exclude CONFIG.DBF from the project and ship it as a separate file, configured differently for each application.

This month's Subscriber Downloads includes a set of sample files for STARTAPP (they're in the STARTAPP subdirectory). This sample requires that you have a copy of wwIPStuff, since it's not included with the Subscriber Downloads. It also requires an FTP site you have access to. First, copy the following wwIPStuff files to the directory where you installed the sample files (that is, the parent directory of the STARTAPP subdirectory): WWIPSTUFF.VCX, WWIPSTUFF.VCT, WUTILS.PRG, and WCONNECT.H. Next, copy WWIPSTUFF.DLL into the STARTAPP subdirectory. Open the CONFIG table, fill in the FTP site, user name, and password, then build STARTAPP.EXE from the STARTAPP project. Open the MAIN project, click on Build, click on Version, enter "1.1" for the version number ("1"

in the major number and "1" in the minor number), and build MAIN.EXE. Upload FILES1.TXT (which contains references to the version number and file name) and MAIN.EXE to the FTP site. Build MAIN.EXE again with a version number of 1.0, then run it; you should see "1.0.0" displayed for the version number. Now run STARTAPP.EXE. After displaying the progress of the download, you should see "1.1.0" displayed, indicating that the updated EXE was downloaded.

UPDATE.EXE

While STARTAPP works great, there are a few little irritants about it. First, the user has to run the loader EXE rather than the "real" EXE for your application (you can minimize this issue if you name the loader EXE what you would normally name the real EXE and name the real EXE something like "MAIN.EXE"). Second, notice the one thing that can't be updated is the loader EXE itself; since it's the currently running program, it can't be overwritten with a newer version. While it's unlikely you'll change the loader EXE often, you might want to add additional functionality to it, such as displaying an application-specific splash screen, after it's already installed on your client's machines. Finally, while this process works well for an automatic update, what if you'd rather have your users manually choose to look for an updated version (maybe they don't always have an Internet connection available), perhaps by choosing an "Internet Update" function from the Tools menu of your application?

Because of these issues, I decided to try a different approach: make the application's real EXE responsible for checking for and downloading the update files. Of course, the fly in the ointment is that we can't overwrite the running EXE. However, what if the download was a ZIP file instead of an EXE? That would eliminate the problem of overwriting the EXE and give us the additional benefit of a small download. For this to work, we have to delegate the responsibility of unzipping the downloaded ZIP file to another EXE. The idea is to call the other EXE using RUN rather than DO (so it runs in a separate process), then QUIT the current EXE. The other EXE will unzip the downloaded file, which likely contains a new version of the application EXE, then RUN the application EXE and quit.

To use this technique, have your application's main PRG call CHECKUPDATE.PRG early in the startup process. That routine has almost the same code as STARTAPP.PRG (since they have similar responsibilities), with the following differences:

- It obviously doesn't run the main EXE at the end.
- It checks the version number of the currently running EXE rather than the EXE named in AppName.
- The EXE named in AppName is the name of the unzipping program to call.
- After successfully downloading the files, it executes the following code to call the unzipping program and terminate:

```
lcFiles = ''
for lnI = 1 to lnFiles
  lcFile = laFiles[lnI]
  if upper(justext(lcFile)) = 'ZIP'
    lcFiles = laFiles[lnI] + chr(13)
  endif upper(justext(lcFile)) = 'ZIP'
next lnI
lcUpdateFile = sys(2015) + '.txt'
strtofile(lcFiles, lcUpdateFile)
lcCaption = _screen.Caption
run /n1 "&AppName" "&lcCaption" "&lcUpdateFile" ;
  "&lcCurrEXE"
on shutdown
quit
```

Before calling the unzipping program, it creates a text file with the names of the ZIP files that were downloaded (I used a file to transfer the list of file names rather than a parameter in case there were issues that would cause problems with parameters passed to an EXE, such as spaces in the file names). It then runs the unzipping program, passing it the _SCREEN caption (we'll see why we need that in a moment; if _SCREEN isn't visible in your application, change this code accordingly), the name of the text file, and the name of the current EXE to the unzipping program (with quotes around all parameters in case there are spaces in them), and then terminates the application.

The unzipping program, UPDATE.EXE, is a very small because it consists of just three PRGs: UNZIPDOWNLOAD.PRG, WAITFORAPPTERMINATION.PRG, and KILLPROCESS.PRG. It uses DynaZip, a commercial zipping utility available from Inner Media (www.innermedia.com), to do the dirty work. It could easily be changed to work with a different unzipping utility if necessary.

The main program, UNZIPDOWNLOAD.PRG, waits until the application that called it has terminated; if it doesn't terminate after about ten seconds, it kills that application. It then opens the text file containing the list of files to unzip and unzips each one, using the UnZip function in WWIPSTUFF.DLL (which in turn calls DynaZip's DUNZIP32.DLL; I could've called the DynaZip function directly but wwIPStuff was already being used for the FTP transfer anyway). It then re-runs the EXE that called it, which is likely a newer version after the unzipping process.

```
lparameters tcCaption, ;
  tcZipFile, ;
  tcAppName
local llReturn, ;
  laFiles[1], ;
  lnFiles, ;
  lnI, ;
  lcZipFile

* Wait until the application that called us has
* terminated. If it doesn't do so voluntarily, we'll
* kill it.

llReturn = WaitForAppTermination(tcCaption)
if not llReturn
  llReturn = KillProcess(tcCaption)
endif not llReturn

* If the application terminated, read the file containing
* the list of files to unzip into an array, erase the
* list file, and unzip each file.

if llReturn
  lnFiles = alines(laFiles, filetostr(tcZipFile))
  erase (tcZipFile)
  declare integer UnZip ;
    in wwIPStuff.dll ;
    string ZipFile,;
    string Destination,;
    string FileSpec
  for lnI = 1 to lnFiles
    lcZipFile = laFiles[lnI]
    if file(lcZipFile)
      UnZip(fullpath(lcZipFile), sys(5) + curdir(), ;
        '*.*')
      erase(lcZipFile)
    endif laFileSizes[lnI] > 0
  next lnI
endif llReturn

* Start the application if one was specified and it
* exists.

if not empty(tcAppName) and (file(tcAppName) or ;
  file(forceext(tcAppName, 'EXE')))
  run /n1 "&tcAppName"
endif not empty(tcAppName) ...
```

WAITFORAPPTERMINATION.PRG waits for up to ten seconds for the specified application to terminate. It uses the FindWindow API function to check if the application is running. FindWindow expects to be passed the title bar caption of the application; that's why we passed _SCREEN.Caption from the application EXE to the unzipping EXE.

```
lparameters tcCaption
local lnCounter, ;
```

```
  llReturn
declare integer FindWindow in Win32API ;
  string @cClassName, string @cWindowName
declare Sleep in Win32API ;
  integer nMilliseconds
lnCounter = 0
llReturn  = .F.
do while not llReturn and lnCounter < 10
  Sleep(1000)
  lnCounter = lnCounter + 1
  llReturn  = FindWindow(0, tcCaption) = 0
enddo while not llReturn ...
return llReturn
```

KILLPROCESS.PRG, which is only used if the application doesn't terminate by itself, kills the specified application. It uses the FindWindow and SendMessage API functions to send a terminate message to the application.

```
lparameters tcCaption
local lnhWnd, ;
  llReturn
#define WM_DESTROY 0x0002
declare integer FindWindow in Win32API ;
  string @cClassName, string @cWindowName
declare integer SendMessage in Win32API ;
  integer hWnd, integer uMsg, integer wParam, integer lParam
lnhWnd   = FindWindow(0, tcCaption)
llReturn = lnhWnd = 0
if not llReturn
  SendMessage(lnhWnd, WM_DESTROY, 0, 0)
  llReturn = WaitForAppTermination(tcCaption)
endif not llReturn
return llReturn
```

By the way, in case you're wondering where I found the information about these API functions, it was on the Windows FAQ and API sections of the Universal Thread (www.universalthread.com). Thanks to George Tasker, Erik Moore, and Ed Rauh for posting this information. If you're not using the Universal Thread to find or share information with other FoxPro developers, you're working too hard!

You may wonder how UPDATE.EXE itself gets updated on the workstation when a new version is released. That's easy: add UPDATE.EXE to the list of files to download from the FTP site. CHECKUPDATE.PRG will download it, then call the new version to unzip the rest of the files.

This month's Subscriber Downloads includes a set of sample files for this process (they're in the UPDATE subdirectory). This sample requires that you have a copy of both DynaZip and wwIPStuff, since they're not included with the Subscriber Downloads. First, if you didn't do it for the STARTAPP sample, copy the wwIPStuff files to the parent directory of the UPDATE subdirectory. Next, copy the DynaZip DUNZIP32.DLL and WWIPSTUFF.DLL into the UPDATE subdirectory. Open the MAIN project, open the CONFIG table, and fill in the FTP site, user name, and password. Click on Build, click on Version, enter 1.1 for the version number, and build MAIN.EXE. Zip MAIN.EXE into MAIN.ZIP, then upload FILES2.TXT (which contains references to the version number and file name) and MAIN.ZIP to the FTP site. Build MAIN.EXE again with a version number of 1.0, then run it; you should see "1.0.0" displayed for the version number, followed by the progress of the download, then after a pause, you should see "1.1.0" displayed, indicating that the updated EXE was downloaded.

Customizing

Several changes could be made to STARTAPP or UPDATE, depending on your needs. For example, instead of checking for a more current version every time the application starts, you could look in CONFIG.DBF for the number of days between checks and look in the Registry for last date checked (you must, of course, update that date in the Registry when you do the check). You could even look in the Registry for a user-defined application option: do the startup check at all (if you don't update the application often or if the user doesn't always have an Internet connection available, they may want to turn

this feature off). Instead of using a configuration table, the settings could be stored in the Registry or (horrors!) an INI file.

Conclusion

Both STARTAPP and UPDATE provide a valuable resource for your toolbox: a way to automate the distribution of new releases of your applications. I'm using the UPDATE process in a commercial application and it works great. I hope you find it as useful.

Assigning a Version Number to a VFP EXE

You can assign a version number to a VFP EXE in a couple of ways. One, when you click on the Build button in the Project Manager, the Build dialog has a Version button that brings up a dialog in which you can enter version information, including the version number. Another way is to set the VersionNumber property of the Project object that exists when the project is open using code such as _VFP.ActiveProject.VersionNumber = '6.05' (you can set other version properties using similar code; see the VFP help topic for Project Object for a list of the properties).

VFP can automatically increment an EXE's version number every time you build an EXE by checking the Auto Increment checkbox in the version dialog or setting the Project object's AutoIncrement property to .T. I prefer not to do this but instead to set the build number myself, because I often do a lot of test builds prior to a release build and don't want those tests to affect the version number. However, if you're like me, you often forget little things like changing the version number before doing a release build, and then you get support calls from customers asking why the version number hasn't changed in the new release <g>. To automate the process of incrementing the version number only for release builds, I created a ProjectHook object (see my September 1998 column, "The Happy Project Hooker", for information on ProjectHooks) with an lReleaseBuild property and code in its BeforeBuild method that increment the VersionNumber property of the Project object if lReleaseBuild is .T. When I want to do a release build, I simply set _ VFP.ActiveProject.ProjectHook.lReleaseBuild to .T. prior to building the EXE (actually, the ProjectHook displays a toolbar with a checkbox for the property so it's right there in my face so I don't forget <g>). In fact, my ProjectHook even takes it a step further: after building a release version, it updates the version number in the update text file, zips the EXE, and uploads the zip and text file to my FTP site so I don't have to do it manually.

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.