
3

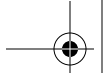
Remote Script Management

In the previous chapter we described three ways of installing a script on a workstation to allow it to run either on bootup or at a predetermined time without any user intervention. We have also suggested techniques for automating the installation process itself, for example, by scripting the steps that need to be carried out in order to set up a Win32-based service. We start this chapter by taking the concept of automation further by writing a script that, once installed, on a workstation will never need to be updated; it will be given the power to update itself automatically whenever it becomes obsolete. In the second half of the chapter, we demonstrate how this functionality can be adapted to create a generic stub script, whose only function is to automatically install, update, and run any maintenance or configuration scripts that are thrown at it.

Self-Updating Scripts

Surely, there is no easier way to manage changes to scripts that are sitting on remote workstations than to let the scripts do it themselves. The rationale is crystal clear: it saves a huge amount of the system administrator's time and, unlike an overworked sysadmin, a script will not forget to run or have to delay for a week before updating itself due to sudden priority changes. Once the mechanism for automatic script updating is in place, the system administrator can cheerfully make changes or implement some new functionality to deal with new or changed circumstances; deploying these changes on a fleet of workstations is never an issue to worry about.

The good news is that a mechanism for automatic updates is surprisingly simple to implement: all that is needed is a way for the script to connect to a file server, compare itself with an "authoritative" version, and overwrite itself if required. In



order to be useful, a self-updating script for workstation maintenance has to have the following characteristics:

- It must be able to run on the workstations for which it is intended.
- It must be triggered to run automatically.
- Its successful operation should not require any user intervention.
- It should recognize when it is obsolete and update itself accordingly.
- It must be able to deal with network disruption.

We will now look at each of these requirements in more detail, highlighting the most significant issues. This will give us a general indication of the overall mechanism that will need to be implemented in our self-updating script. The first three requirements are, of course, common to any script that is meant to run unattended on a workstation. By contrast, the requirements to recognize the concept of “out-of-dateness” and to deal with network disruptions are peculiar to self-updating scripts.

Runs on the Intended Workstation

This is an obvious point but still worth mentioning. If a script is to run at all, each workstation must be equipped with all the necessary baggage that allows it to run. For example, any Perl script would be remarkably ineffectual if Perl is not installed! If the script invokes command-line utilities that do not form part of the standard NT workstation distribution, then they should also be present.

Runs Automatically

We’ve dealt with the mechanics for this in some detail in the previous chapter. However, to summarize, you can either configure automatic logon to have the script run at startup, have it triggered by a scheduler, or implement it as a service.

Requires No User Intervention

It is essential to ensure that the script does not hang around waiting for user input. While this may seem obvious, it is easy to get caught out. For instance, a wrapped command-line utility may require confirmation before it carries out a given action; even worse, the utility might do this only in certain circumstances.* If such eventualities are not foreseen and solutions written in, the script will sit idle, waiting for input that just won’t turn up.

* A classic example is the `net share` command (used to manage shares). If you attempt to create a share with too long a name, it will warn that MS-DOS workstations will be unable to access it and request confirmation that you really want to create it.



Recognizes That It Is Obsolete

The script needs a reliable method to determine when it is obsolete. An obvious check would be to use timestamps, but the obvious solution is not always the most reliable. After all, workstations do not necessarily agree on time and date, especially if the end user has a reasonable level of control over them; even if network time protocols are used, it is rarely possible to guarantee that a workstation's concept of time ties with reality. Unless you are convinced that you really can guarantee synchronized dates across your network, it is far better not to rely on them at all. A much more reliable way to determine the obsolescence of a script is to use version numbering. Whenever the master server-side script is amended, its version number should be incremented; this allows a workstation script simply to compare its version number with that of the server version. Such a solution is unambiguous, simple to implement, and strongly recommended!*

Deals with Network Disruption

A workstation script has to rely on the network to update itself; if the network is unavailable, the ability to check itself disappears. An important decision to make when implementing a self-updating script is to determine behavior in such a situation: should the script run without checking itself or should it die quietly. In many cases, when it may be performing some quite innocuous task, it will not matter a great deal if an obsolete script runs. However, if the result of running the script has an influence on the behavior of other workstations, for instance, changing IP numbers or NetBIOS names, then failure to execute the most recent version could cause havoc in the local network community. Apply a suitable rule to match the function of the script.

Writing the Script

Now it is time to put our thoughts into action and write the script itself. Taking into account the preceding discussion, we can lay out a narrative of how the script should behave:

1. The script starts up and attempts to connect to a dedicated server share.
2. If the share is not available, the script continues to run, ignoring the possibility that it is out of date. Alternatively, it halts immediately and exits.

* The use of version numbering suggested here is hardly revolutionary; in fact, it is common practice in many situations. For example, DNS servers (such as Bind) rely on version numbering to determine the validity of cached DNS database files. For details, see *DNS and Bind*, by Paul Albitz and Cricket Liu, (O'Reilly & Associates, 1998). There are, of course, other possible approaches such as checksumming or using MD5 signatures.

3. If a connection to the share is made, it attempts to find a file on the server with the same name as itself. If no matching file is found, the script continues to run, assuming that it is the most up-to-date version. Another possibility would be to change this behavior so that if a version is not found on the server, the script would assume it is superfluous to requirements and delete itself. We implement the latter scenario when discussing stub scripts, later in the chapter.
4. If a connection to the share is made and a match is found, the script must determine which version is the most up-to-date one. It does this by opening and reading the first two lines of the server-side script. The second line contains its version number. The running script then makes a comparison of its version number with the one found in the server copy. If the server copy has a later version number, then an update is required. The script copies over the server file, overwriting itself in the process, launches its updated self as a new process, and exits. If the server version is older, then there is no need for an update; the script continues to run as usual.

The following Perl program implements the preceding specification. To run it, type `self.pl` at the NT command line with no parameters. More usefully, install the script so that it runs at startup or at regular intervals with a scheduler. After all, if you're carrying out a once-only operation, you probably don't need a self-updating script.

First, we set the variable `$currentversion` and load the `Win32::Process` module that will be used to launch the new script if an update is required (this is explained in more detail later). It is essential that `$currentversion` should appear on the second line of the source file, because that is where we look for it on the server version when we do the version comparison.* Then we set a series of variables needed to connect to the network share and find the server file. Although it would be possible to hardcode this information where it is needed further down the script, modifications tend to be far simpler and less error prone if you set up strings in advance.

```
#Self.pl
$currentversion=8;
use Win32::Process;

$selfshare='\\kanchenjunga\test$';
$selfdrive='Z:';
$selfname='self.pl';
$selfpath="$selfdrive\\$selfname";
```

* It would be a simple task to alter the code so that it scoured the whole source file for this variable, but we leave that as an exercise for you.

Now we attempt to connect to the share. You will notice that no username and password parameters are passed to `net use`, so the script will try to connect with the security context in which it is running. It may well be appropriate to add two parameters to change this behavior; the change would be trivial. The rather strange group of punctuation marks `!$?` probably deserves some explanation for the reader who is not too well versed in Perl peculiarities: whenever a system command is executed (i.e., through backticks), the error return code is stored in `$?` ; conventionally, a return code of 0 means that the command executed successfully. Therefore, `if (!$?)` means “if the command executed successfully . . .”

```
`net use $selfdrive $selfshare`;  
if (!$?) {
```

Now for the check. First, we see if the file exists on the server; if it does, we open it for reading, discard the first line, and read the second. A quick regular expression extracts the version number from the line, and we close the file. You will notice that we don't bother to check that the second line contains a valid version number; typing errors aside, it always should! It would be simple to implement this functionality, but you would need to decide what action the script would take if the version number is invalid.

```
    if (-e $selfpath) {  
        open (CHECK,$selfpath);  
        <CHECK>;  
        chomp ($version = <CHECK>);  
        $version =~ s/\$currentversion=(\d+)/$1/;  
        close (CHECK);
```

Now we check the version number of the server-side script with that of the one currently running. If the server script is newer, we make a local copy of it (overwriting the running script), disconnect from the share, and launch the new script with the `Process` module.

We use the module function (rather than launching the new script with backticks or a `system()` call) because it allows us to launch the new script cleanly as a genuinely new process; without the module function, the old script would have to hang around in the background waiting for the new one to finish its work before exiting. Such a situation may not cause a problem, but it is hardly elegant. (The process module is explained more fully in the Appendix, *Perl Module Functions*.) Having launched the new script, we exit with a return code of 0; after all, the process has been successful!

```
    if ($version > $currentversion) {  
        `xcopy $selfpath $selfname`;  
        `net use $selfdrive \delete`;  
        Win32::Process::Create($process, "C:\\Perl\\5.00502\\  
bin\\MSWin32-x86-object\\perl.exe", "perl $selfname",  
0, CREATE_NO_WINDOW, ".") || die "Create: $!";
```

```
        exit(0);  
    }  
}
```

If the script could not be found on the server, or if the server copy is older than the version that is running, we disconnect from the network drive. Finally, we get on with running the payload of the script, the bit that actually carries out the maintenance or configuration task.

```
`net use $selfdrive \delete`;  
print "doing something useful here onwards\n";
```

And that's all there is to it. Getting this script to work for the first time will require a little thought. It will need an appropriate share and directory hierarchy to be set up on a file server within your domain; some thought will also have to be given to permissions and user accounts (discussed in Chapter 2, *Running a Script Without User Intervention*, and again in Chapter 8, *Script Safety and Security*). However, once you do this, all that it will take for a script to become self-updating will be to include the preceding code at the start of it, set the initial variables appropriately, and place a copy on the server share. Once the script has initially been installed on a workstation, manual updates will be a thing of the past.

Actually, the implications of installing a single self-updating script on a workstation are wider than this. While it is certainly true that once a script is installed, modifications are updated automatically, the real beauty of the system is that once a self-updating script is installed on a workstation, you *never* have to install *any* script on that workstation manually again. If you wish to install a script on all your workstations to carry out a completely new and unforeseen task, you simply append code to the self-updating script that will automatically install the *new* scripts for you. (The Perl needed to automatically install a new script is discussed at length in Chapter 2.) Of course, in the interests of future-proofing, you could always make sure that all new scripts include the self-updating code.

As you can see, self-updating scripts constitute a very powerful workstation management tool that, in conjunction with automatic script installation, can greatly reduce the time needed to deploy maintenance and configuration tasks on your workstations. However, we have not quite reached the end of the story, because although they are powerful, self-updating scripts in their current guise are not wholly satisfactory in all situations. A few pitfalls and caveats are listed here:

- Every self-updating script has to contain a substantial chunk of code that has nothing directly to do with the primary function of the script. Not only does that make code more bulky and less clear, it also provides a perfect opportu-

nity for the introduction of typographic errors, which can cause bugs that may be very hard to detect.*

- In the very likely scenario that cut-and-paste is employed to insert self-update functionality into a script, it is highly likely that (at least once) someone will forget to change the variables configuring server share, path, and script name. This will lead to a script potentially updating itself with another, totally different, one. Again, the effects may well go undetected for quite some time.
- If circumstances require that your server setup changes and the server-side scripts need to be moved, every single self-updating script will have to be modified to reflect the changes.
- Depending on your domain security model and the security context within which your scripts are running, it is possible that each self-updating script will need to store username and password information with which it can connect to the server share. The more self-updating scripts that are present, the more likely it is that one of them will be inadvertently left in a state in which an unauthorized user can read this privileged information.
- If a script finds that it is obsolete, it downloads and runs the new version, which will also presumably be self-updating. This leads to inefficiency, as each script has to check itself against the server twice (first the old script does it, then the new one does, only to discover that it is up-to-date anyway).
- There is no mechanism for removing scripts altogether (rather than updating them) when they become genuinely obsolete. As previously mentioned, however, this can be dealt with by changing the update mechanism so that scripts delete themselves if there is no equivalent on the server.

None of these points on its own is necessarily that serious, and certainly if you never need to run but one or two self-updating scripts on workstations, it is unlikely that too many problems will be encountered. However, in a situation in which several maintenance and configuration scripts are running on a workstation, each of which needs to be self-updating, it would be far better to have a single script that took the responsibility for updating *all* installed scripts. This would allow others just to get on with the business of doing their jobs, without worrying about versions and all that paraphernalia.

Not only is it possible to write a script that takes care of all the script updates on a workstation, but if you understand how a self-updating script works, it is also reasonably straightforward. We call these things stub scripts, and by the end of the chapter, you will know how to write one.

* This problem is mitigated substantially if the self-updating code is moved to a general module that can be invoked by all scripts that wish to update themselves (see Chapter 9, *A Custom Module*).

Using a Stub Script

A stub script uses a very similar mechanism to a self-updating script, but instead of checking its own version number against that of a server-based equivalent, it compares a whole directory of Perl scripts with an equivalent directory on the server. Once it is satisfied that its local directory is a faithful replica of the server directory, it runs each of the local scripts in turn, exiting when all are complete. The stub can, of course, be fired by any of the methods discussed in Chapter 2.

A major advantage of a stub over a self-updating script is that update code is stored in only one location and is applied globally, so it can be changed easily without worrying about incompatibilities and conflicts. The only requirement for a script that wants to be updated is that it has a version number stored in the second line of the source code.

An even more important advantage of the stub is that it allows complete control over what scripts run on workstations. If you use a self-updating script to install a new script (as described earlier) and that new script becomes obsolete, it may be impossible to remove it remotely from the workstation and prevent it from running (especially if that script is *not* self-updating). Even if this does not happen, a workstation may have several Perl scripts running as services, at logon or sparked by schedulers in a general higgledy-piggledy mess. At best, such a situation can be considered aesthetically unpleasing; at worst, a workstation could be left in an awful mess, with all sorts of scripts running all over the place, possibly when they are not even required. By contrast, if you use a stub, the stub itself is the *only* script that *ever* need be installed directly to the workstation and configured to run automatically;* it is the stub that is responsible for running all other scripts. This makes for a far neater setup, as there is only one “point of contact” between the operating system and your management scripts. To illustrate the point, once a stub is installed, consider the steps needed to add a new script to all of your workstations:

1. Write the script, not worrying about any update or installation issues.
2. Put it in the right server directory. The stub script on each workstation will ensure that the script is downloaded and run regularly.

To update a script on all your workstations:

1. Modify the server-based script.
2. Increment the version number.

* By this, we mean that it is the only script that will have to be installed to a scheduler, in an autologon situation, or as a service. Clearly, all scripts will still be “installed” in that they will reside on the local machine’s hard disk.

And to remove a script from all your workstations:

1. Remove the script from the server directory. The stub script on each workstation will ensure that it is never run again.
2. Have coffee?

We hope you will agree that this is about as painless as it gets! Of course, there is no reason why a stub script cannot update itself while it is at it.

Writing the Stub

Before presenting the code for the stub script itself, it is worth considering (as we did for self-updating scripts) exactly what the script needs to do. A narrative might look something like this:

1. Start up and connect to a share on the server.
2. If the share is not available, report the error and die. An alternative scenario would be to assume everything is up-to-date if the server cannot be reached and run all scripts anyway.
3. For each file on the server, check to see if there is an equivalent on the workstation. If there is, check the version numbers.
4. If the workstation does not contain a copy of the server file or if the server copy is newer, copy the new file to the local workstation.
5. If any files exist on the workstation that have no equivalents on the server, consider them to be obsolete and delete them.
6. Run each of the scripts in turn before exiting.

Most of these steps are nonproblematic: connecting to a network share, version checking, and sparking new scripts have all been seen in the self-updating script code. In fact, running other scripts is easier here, as we do not need to use the `Win32::Process` module; we *want* the stub to carry on running, waiting for other scripts to return. One issue that does require a little thought, however, is the business of comparing the contents of the server and workstation script directories. It is easy, using Perl, to read the contents of a directory. Therefore, it should not be too hard to go through each file in the server directory and check for its equivalent on the workstation; this would give us a way of finding out if there is anything on the server the workstation does not know about. The hard part is that we also need to keep tabs on what files on the *workstation* are not on the *server*; this would involve repeating the process in reverse (checking each file in the workstation's script directory to see if it exists on the server). To be sure, this would work, but it can hardly be considered efficient! We need to find a better way of matching contents in two directions.

There are, in fact, several solutions to this problem. The one we use involves the use of a hash table. If you haven't come across hash tables before, for this particular purpose you can consider them to be special kinds of arrays that have the following rather useful features:

- You can easily add or remove elements at will.
- Elements are stored in pairs of keys and values.
- You can use *keys* to search through a hash table without writing a loop.

There is actually slightly more to hash tables than this, as you will see when they crop up again in the next chapter.*

So how do hash tables solve our current problem? We create a hash table containing keys corresponding to all the Perl scripts in the workstation script directory. For each item in the server directory, we look up in the hash table to see whether the workstation equivalent exists. If it does, we do the version checking, copy the file if necessary, and *delete the entry from the hash table*. This last step is crucial: once we have gone through every script on the server directory, the hash table will contain only those scripts that reside on the workstation but are *not* on the server. The files corresponding to those entries can then be deleted from the workstation directory.

Next, we present the script itself. As you can probably guess, you can run it by typing *stub.pl* at the NT command prompt or, more usefully, setting up the workstation so the stub runs automatically. This stub script does not include the self-updating code, but it clearly could, should this be required.

First, we set up a few variables to specify the local path for updatable scripts, the server share name and the share path. Unlike the previous script, we do not specify a local drive letter to use for connection; instead, *Z:* is hard-wired. The change, should this be required, is totally trivial; you simply add an extra variable name and use it when needed throughout the script.

```
#STUB SCRIPT
$localpath = 'd:\updatable';
$remoteshare = '\\annapurna\test$';
$remotepath = '\updates';
```

Next, we open a handle to the workstation script directory and create an array of all the files in it. We then go through each file in this array; each time we come across a file that ends in *.pl*, we add it to the hash table `%local_files`. Note that `%local_files` never has to be mentioned explicitly, as the syntax `$local_files{$file} = "x"` means "add an element to the hash table with the key

* For a simple explanation of general hash table usage, see *Learning Perl on Win32 Systems*, by Randal L. Schwartz, Erik Olson, and Tom Christiansen, (O'Reilly & Associates, 1997).

`$file` and the value `x`.” In our example, we are not interested in the value, so we set it to something small and arbitrary; it is the key that is of importance to us. Finally, we reset the array to null (to save a bit of memory) and close the handle to the directory.

```
opendir LOCAL, $localpath;
@local_files = readdir LOCAL;
foreach $file(@local_files)
{
    if($file =~ /\.pl$/)
    {
        $local_files{$file} = "x";
    }
}
@local_files = "";
closedir LOCAL;
```

Now, we attempt to connect to the remote share. (We start by disconnecting anything that might be connected to `Z:`, so we can use this letter. Note that we do not care about the return code from this first invocation of `net use`). If we are able to mount the share, we open the server script directory and create an array of all the files in it. Here, we don't bother with removing files that do not contain a `.pl` suffix; we assume that the administrator won't put such files there. (If you wish to add this check, you can incorporate the regular expression from the section of script shown earlier). If we cannot connect to the server, the script dies—functionality that you may wish to change, so that the script runs existing local scripts instead.

```
`net use Z: /delete`; # just to be sure
`net use Z: $remoteshare`;
if(! $?) # if we got the share mapped successfully
{
    opendir REMOTE, "Z:$remotepath";
    @remote_files = readdir REMOTE;
    closedir REMOTE;
}
else
{
    die "Unable to connect to server";
}
```

The following chunk of code contains the main functionality of the script. For each file in the array of remote files, we look it up in the hash table of local files. If it exists here, we check the version numbers using more or less identical code to that used in the self-updating scripts. If the server version is newer, the old one is overwritten and the associated hash table element is removed; if the file does not exist in the workstation directory, the new one is copied over.

```
foreach $remote_file(@remote_files)
{
```

```

if($local_files{$remote_file} ne undef)
{
    print "Found a match on $remote_file\n";
    open FILE, "$localpath\\$remote_file";
    <FILE>; $localversion = <FILE>;
    $localversion =~ s/\\$version=(\\d+)/$1/;
    close FILE;
    open FILE, "Z:$remotepath\\$remote_file";
    <FILE>; $remoteversion = <FILE>;
    $remoteversion =~ s/\\$version=(\\d+)/$1/;
    close FILE;
    if($remoteversion > $localversion)
    {
        `xcopy Z:\\$remotepath\\$remote_file
        $localpath\\$local_file`;
    }
    delete $local_files{$remote_file}; # delete hash entry
}
else `xcopy Z:\\$remotepath\\$remote_file $localpath\\$remote_file`;
}

```

Now we have a new copy of everything that is on the server. All that remains in the update process is to delete from the workstation files that are now obsolete (that is, the ones that were not present on the server). This involves going through each element that remains in the hash table and deleting the associated file.

```

@deletables = keys(%local_files);
foreach $deletable(@deletables)
{
    print "deleting $deletable";
    `del $localpath\\$deletable`;
}

```

Finally, we go through each script in the local directory, running all the scripts. This involves creating a new array of filenames and running each in turn. It goes without saying that here it is essential that we only run files with a *.pl* suffix; hence the regular expression.

```

opendir LOCAL, $localpath;
@local_files = readdir LOCAL;
closedir LOCAL;
print "Running the scripts...";
foreach $file(@local_files)
{
    if($file =~ /\.pl$/) # We're interested only in .pl files
    {
        `$localpath\\$file`;
    }
}

```

And that is all there is to it. It is possible to make the stub far more sophisticated by, for example, defining a configuration file format, so that scripts could be run at different times of day or at system startup. Rather than checking the contents of

the script directory, the stub could check the server-based configuration file and act accordingly to add or remove services, items to or from a scheduler, and so forth. In some situations, this level of sophistication might be very useful, and if you want to write it, go ahead! However, we believe that in this sort of situation, simplicity has the great advantage of transparency. If you need different scripts to run at different times of day, there is no reason why you cannot have a series of stubs running on each workstation, each looking after a different script directory, and each running at a different time.



A word of warning: If you need the stub script to be self-updating, use the self-update code presented in the first half of this chapter. Do *not* store the stub script file in the directory that the script itself monitors in the hope that this will deal with automatic updates. If you do, consider what will happen: The stub will update (possibly) and then run itself. It will then update (possibly) and then run itself. It will then update (possibly) and then run itself . . . ad infinitum!

Summary

We started this chapter by suggesting a way in which scripts running on NT workstations can be written so as to update themselves automatically whenever a change in circumstance renders them obsolete. We have intimated that with a bit of planning, such scripts could save an enormous amount of administration time and frustration. However, we have also pointed to the dangers of using such scripts, the potential for confusion, errors, and obfuscation. The second half of the chapter introduced stub scripts, a more elegant solution to the general problem of keeping workstation maintenance and configuration scripts up-to-date and running smoothly. We have suggested that in most circumstances, a stub-script-based approach to managing scripts on multiple workstations is far preferable to using a large number of self-updating scripts. In both this and the previous chapter, we have concentrated on techniques for installing, maintaining, and running Perl scripts on Windows NT workstations. In the next chapter we start to give you some idea as to what these running scripts might actually do!