A Networked Machine Management System

Dave Roth
rothd@roth.net
Roth Consulting
http://www.roth.net

1 Purpose

Out of the box, Win32 machines lack the ability to be efficiently managed. It is up to an administrator to implement some form of system management. For many administrators with relatively small networks this means walking from machine to machine to install new software or check to see how many CDROM drives exist on the computer. Considering the sheer size of some networks it becomes a nightmare to do something such as deploying an updated driver or service pack. Even a simple task as discovering which computers have enough free hard drive space for a software upgrade can take many hours of an administration teams time.

To be sure, this is not an isolated problem. It is ubiquitous enough to create a market where there are several commercial products available to help administrators manage this issue. Microsoft's Systems Management Service (SMS) is one example. However a product such as this can become quite time consuming just to maintain the system as well as costly for full licensing.

With a little bit of forethought administrators can implement a simple and efficient system for performing the same functionality as a full-blown SMS implementation. By using off-the-shelf products and some of Win32's standard services a network of almost any size can be managed.

This paper provides a case study on one particular implementation.

2 Needs Assessment

As with any project a needs assessment is a wise beginning. This paper focuses on a large cellular service provider which we had the pleasure of consulting with. We shall call this company *CellBell*.

2.1 Background

CellBell had a network of approximately 1700 NT boxes scattered across the state. The machines were running a combination of NT 3.51 and NT 4.0. Some had different service packs installed while others had none. Some machines had 32 megs of ram while others had only 16. Some had large hard drives with plenty of free space while others had practically no space available. Some had 486 processors while others were Pentiums. The point is that the network was non-homogenous and there was no documentation to help sort out how each machine was configured. To make matters worse the IT team was competent but under funded. There was no money to send the team to SMS training let alone money to purchase SMS licenses.

From time to time CellBell corporate headquarters would decide that all machines on the network would have to have a software update. The team, loyal to a fault, would painstakingly walk from machine to machine, log off any current user, log on as administrator, check if the machine satisfied any ram and drive memory requirements, connect to a network server, install the software then reboot. If the capacity of the machine did not meet the required spec then it would be added to a list then later parts would be procured and installed. This could take days to accomplish – requiring overtime and travel.

2.2 Needs

We decided that there was a need for a system which would:

- Gather information about each machine in a timely basis
- Effectively install software on a machine with minimal human interaction
- Perform routine machine maintenance (update drivers, remove temporary files, etc)
- Alert the IT team when a machine was having problems
- Be inexpensive
- Be flexible enough to adopt to any future need
- Be easy to use

Considering these basic issues and a rather tight budget the team set out to develop an infrastructure that could effectively provide all these needs.

3 The Infrastructure

After much deliberation an infrastructure was devised. To be sure it is far from a perfect design but for a tight financial and time budget it worked well.

3.1 Design Overview

The entire process is quite simple. It makes use of what we called *scripts*. A script could be anything executable such as a program or a batch file, for example.

At some predetermined time each machine on the network would spawn a process which and check to see if there were any scripts in a predetermined location (presumably on a network server). Each of the scripts would then be executed. Once all scripts have been run the process ends.

That is it; quite simple indeed. The system would be only as complex as the administration staff decided to make it.

3.2 Scripts

Each script can do anything that the administrator may need. One may look for a given ODBC driver on the machine and install it if need be. Another script may install some new software package. Another script may examine the local event log for errors and warnings. Yet another script may download and implement an anti-virus programs updated virus signature files.

So this simple system could have scripts written which modify registry settings, copy directories, apply permissions to files, synchronize the clock with a server or anything else one may need.

3.3 Script development

The management system was going to be home grown to avoid the costs associated with commercial software. Since nobody wanted to even consider using DOS like batch files a programming language would have to be chosen. Due to it's remarkable abilities and reasonable price (free) the Perl language was elected.

Perl is well known as the Swiss army knife of languages. It is almost as powerful as C++ but has a much faster development cycle. And if the language has any limitations they can be eliminated with Perl extensions (plug-ins developed in another language such as C).

One other compelling reason to use Perl is the vast number of modules and extensions that have already been created for it. If there is a need to access web servers, ftp files, search for files and directories or even query databases you will most likely find that someone has already written an extension to do just that. Of which you can download from the net.

It was decided that the system engine and its subsequent scripts would be written in Perl.

3.3.1 Script Execution Tracking

Undoubtedly some scripts may be designed to only run once. If, for example, a software package is to be installed the install script needs to only run once. It would be simple enough to create the script and let every machine run it then remove the script. However, if a new machine is brought online it, too, would need to have the software package installed.

If each install script added an entry to some configuration file on the local machine the script could check to see if it had already been run. This execution tracking permits scripts to remain active in the repository but not a menace to machines already having been affected by it.

3.4 The Script Repository

The infrastructure was designed around a central script repository. This is simply a network share and directory where all the scripts are located. By locating the scripts in one central location the burden up altering scripts is greatly reduced – there is no need to alter a script on every computer or on several servers. This directory can be part of a replication tree so that any change made will be replicated to other backup repository machines.

CellBell had a WAN that extended across the state with multiple subnets. Since many of the subnets were connected together using a slow frame relay link it was crucial that the network design include a repository on each subnet. Ideally there would be one repository on each backup domain controller this way the name of the machine can be discovered by locating a domain controller.

The repository was to be the hub of the management system. The bulk of the files accessed were to be through this repository. The good news was that since the access consisted of simple reading of files there would be very little overhead for the repository machine. Since the BDC's were configured to only act as a domain controller (for user authentication) the overhead of serving such files made only a minimal impact.

By utilizing domain controllers CellBell also provided redundancy. If the repository goes offline then the system engine would locate another domain controller, which may be on different subnet.

3.5 Database Services

One crucial component of this system was a database server. Our team of administrators needed a reliable way of generating an accurate and up-to-date list of attributes regarding the machines. Attributes such as hard drive sizes (physical and available), amounts of ram, pagefile size, processor speed and service pack level just to name a few. A database server was installed to maintain this information.

Since the database was populated with accurate information we could submit queries to discover which machine had less than 64 megs of ram, for example. This provided an effective way to access a wealth of administrative information from across the network. There was no longer a need for a remote administrator to call into the help desk to obtain such information; she could just simply submit a query to the database server. This proved to be absolutely invaluable when management needed fast answers regarding our current state of affairs.

3.6 Information Server

Since we had an accurate database of information the next step was to install an intranet web server. We had written CGI scripts that provided access into the database. It was this way that we empowered non-technically inclined folks (management in particular) to query information regarding the network.

This tool immediately became indispensable for both management and the administration team. Administrators were able to logon when they came in and view reports of the network that were generated earlier in the morning. If an administrator was not able to make it into the office she could dial in remotely and peruse the logs.

4 The Details of the Design

The actual engine and scripts would be fairly quick to prototype but it is the other non programming aspects that would prove to be more difficult. The details such as how to schedule the engine to be run on a timely basis and how to secure the system were the issues that took the most time to work out.

4.1 The Scheduler Service

The Scheduler service works very much as its UNIX cousin the CRON daemon. Basically speaking the Scheduler service allows a job to be scheduled for a given time. A job is simply a command that can be spawned as a process.

The management system will make copious use of the Scheduler service. A job is added to the service that will execute the main Perl script (the system engine). The job is scheduled for a time in which it will not interfere with a user since occasionally a script may need the computer to reboot.

At CellBell we configured each machine to run the engine between 3:00 and 4:00 in the morning. When the job was scheduled a random time was chosen so that there would be less of a burden to both the network and the repository as all the machines woke up.

4.2 Security Concerns

One of the primary concerns regarding the implementation was that of security. After all it is quite easy to write a script that deletes all files on a hard drive, and if some disgruntled employee slipped such a script into the repository there would be quite a problem. Securing this system was absolutely imperative.

4.2.1 Scheduler Account

By creating a special user account that the Scheduler service would run under we were able to guarantee security. In our case we created a user account called “SchedulerAccount”. The decision for this name was quite arbitrary but self-descriptive.

The “SchedulerAccount” account was made a member of the “Domain Admins” group. Since each machine in the domain included the “Domain Admins” global group in its local “Administrators” group this gave the Scheduler service administrative abilities on every machine.

4.2.2 Repository Permissions

The script repository was limited so that only “Domain Admins” and “Administrators” had access to modify the files. To protect entry into the repository the permissions were applied to the sharepoint. To also protect against backdoor access the files and directories in the repository tree were also protected with the same permissions as the share.

There was a debate as to whether or not permissions for users should be applied. Some administrators felt read only administators should have access whereas others did not foresee a problem with users having read only permissions. A decision to prevent users any access was made so that a script could have hard coded userids and passwords. Whereas any file containing userids and passwords is a potential security risk it was decided that in some instances it was necessary.

Considering that users are usually more curious than they need to be a decision was made to hide the share. If a user does not see a share there will be no desire to play with it. The share was hidden by appending the share name with a dollar sign such as “Repository$”.

4.3 Variations Of Scripts

A multitude of Perl scripts were written which performed an amazing array of functions. Some of the more useful ones included:

- A script which compared the machines copy of Perl against the repositories copy. Any updates were then replicated down to the local machine.
- A script checking for files in the temp directory. If files exist that were older than 2 weeks they were removed.
- A script that retrieved all IP addresses hard coded on the machine and submitted them to the database. At the time CellBell was using hard coded IP addresses, none of which were mapped to physical machines. This script helped the IT team track IP address with userids and machine names.
- On script modified each local machines registry to change a hard coded IP address to use DHCP.

There were three scripts in particular, though, that proved to be most useful. These performed ODBC client database configuration (setting up the data source name so that the scripts could talk to a database), collecting machine information and checking the event logs.

4.3.1 ODBC Databases

Since our implementation of the system made use of a SQL Server database it was necessary that there was a verification that the SQL Server ODBC driver was installed and that the correct data source name (DSN) was configured.

The file _ODBC.PL (*Script 3*) performed this task. Notice that the file name begins with an underscore. This was to guarantee that it was run before any other script. Since many of the scripts would attempt to connect to the database it was important that this script was the first (or at least one of the first) scripts to be executed.

4.3.2 System Info

One of the most important scripts that were created was the OS.PL script (*Script 4*). This script collects information regarding the physical machine and the operating system. This information is submitted to the database for later processing.

The script collected information about the computer such as:

- Lists of all local hard drives, CDROMs and removable drives, their total drive size and how much space is available.
- The amount of physical RAM.
- The total pagefile size.
- The OS type (Win NT or Win 95/98).
- The OS version.
- The OS service pack.
- The class and speed of the processor.

Once this information had been collected it was submitted to the database where the information could later be processed.

4.3.3 Event Log Scanning

With thousands of machines on the network it became virtually impossible to keep an eye on each computer. With the EVENTS.PL script (*Script 5*) the local event log was scanned for warnings and errors. These events were then submitted to the database. In this way the administration team could generate reports consolidating multiple errors and alert the staff which of the machines were having problems.

4.4 Installation

Installing the system on each computer promised to take time. In the case of CellBell the machines on the network were not setup with much forethought.

One of the first things we needed to do on each machine was to setup the Scheduler service to logon using the domain account “SchedulerAccount”. Additionally we needed to copy the Perl tree onto the local machine then secure it such that users only had read permissions. This required that the drive be formatted as NTFS. If it was a FAT drive the installation routine would need to run the NTFS conversion utility.

A plan was devised in which our team would walk and touch each machine, logging on with a temporary administrative account we created for this sole purpose called “InstallAdmin”. This account was added to the “Domain Admins” global group. Since each machine included the “Domain Admins” group in its local

“Administrators” group the logon would have administrative authority over the machine.

The “InstallAdmin” account was configured with a logon script which acted as the install script (refer to *Script 3*). The actual logon script command was set to:

“\\server\perlshare$\perl\bin\perl.exe \\server\perlshare$\install.pl”

This would run both Perl and the install script from the repository.

Since the script would configure the machine then reboot all that the administrator had to do was logon with the “InstallAdmin” account then walk to the next machine.

The installation script would register the Scheduler service to logon using the userid of “SchedulerAccount” and the provided password. Since the password would be hard coded in the Perl script this particular file was set with read permissions for only the Administrators and no access for everyone else.

Since all of the functions that the installation script performed could be accomplished remotely it is possible to have a machine or machines on the net walking through the list of online computers and remotely configure them. In the case of CellBell none of the machines had NTFS partitions and since the convert.exe utility must run as a process local to the machine we decided to physically log onto each computer. This way we could also boot any computer that had been powered down.

5 Conclusion

Systems management programs can make life easier for any administrator who has a large network of Windows NT machines. But since the cost and learning time is not necessarily something that is affordable for everyone it makes sense to get the most out of what is available.

The infrastructure presented here not only works but it also works well. The ability to update files, clean directories, apply permissions and other such things is enough to justify the effort.

For CellBell, the total time to create the infrastructure took one person roughly 2 days. Once the installation account was created and configured the system was up and running on all machines in just a couple of days.

Within the first month of use the team was given the task to rollout quite a large and new application. The amount of money in overtime that was saved ranged in the thousands of dollars. At that point the system had by far paid for itself.

6 Appendix A: Resources

This paper covers many topics, mostly related the Perl language. More information on them can be found in the following references:

6.1 Files

The files that are referenced in this paper are available online at the Roth Consulting web site:

http://www.Roth.Net/Conference/LisaNT/1999

6.2 Win32 Perl

Win32 ports of Perl come in both binary and source form. The most reliable sources for Win32 Perl are:

- ActiveState Tool Corp., source code and binaries: http://www.ActiveState.com
- The Perl home page, source code: http://www.Perl.com

6.3 Win32 Perl Extensions

Perl is an extendable language allowing users to create extensions that supplement the built in capabilities. Since Perl is also a cross platform language it is not capable of utilizing many of Win32's unique abilities. This is where the Win32 extensions come in.
The following Win32 Perl extensions come standard with Win32 Perl from ActiveState:

- Win32
- Win32::NetAdmin
- Win32::Registry
- Win32::Service

The following Win32 Perl extensions can be found at Roth Consulting, http://www.roth.net/

- Win32::ODBC
- Win32::AdminMisc

The following Win32 Perl extension can be found at Aldo Calpini's web site, http://www.divinf.it/dada/perl/

- Win32::API

The following Win32 Perl extension was authored by Jens Helberg <jens.helberg@bosch.com>. Its source code can be found at ftp://ftp.roth.net/pub/ntperl/others/lanman/lanman_1_01.zip

- Win32::Lanman

6.4 Perl Reference Books

There has been many books written on Perl. Two such books that are especially applicable to this paper are:

- Larry Wall, Tom Christiansen, Randal L. Schwartz, Stephen Potter, *Programming Perl*, Second Ed., O'Reilly & Associates, 1996.
- Dave Roth, *Win32 Perl Programming: The Standard Extensions*, Macmillan Technical Publishing, 1999.

One particularly useful book regarding WinNT administration is:

- Æleen Frisch, *Essential Windows NT Administration*, O'Reilly & Associates, 1998.

7 Appendix B: Scripts

The management system allowed for scripts that would be executed. The order of execution was alphabetical so a desired order could be achieved by changing names.

7.1 Script 1: An installation script.

This script will setup a machine so that it is ready to use the management system. This script will:

1. Add the global “Domain Admins” group to the local “Administrators” group.
2. Configure the Scheduler service to logon with a given userid.
3. Grant the specified userid the privilege to logon as a service.
4. Convert the Perl drive to be NTFS if it is not already.
5. Set permissions on the Perl directory such that Administrators have full control and everyone else has read only access.
6. Copy the Perl tree to the local machine.
7. Update the PATH environment variable to include the Perl \bin directory if it is not already.
8. Remove the “last logged on user” so that the next time the machines logon box is displayed there is no specified userid.
9. Reboot the machine.

This script requires the following extensions:

- Win32
- Win32::API
- Win32::Perms
- Win32::Lanman
- Win32::Service
- Win32::NetAdmin
- Win32::Registry
- Win32::AdminMisc

```
# Install.pl
# ----------
# This is an installation script designed to configure
# a machine for use with the network management system.
#
#   Dave Roth
#   Roth Consulting
#   http://www.roth.net
#

use Win32;
use Win32::API;
use Win32::Perms;
use Win32::Lanman;
use Win32::Service;
use Win32::NetAdmin;
use Win32::Registry;
use Win32::AdminMisc;

$Machine = Win32::NodeName();
$Domain = "MyDomain";
$User = "$Domain\\SchedulerAccount";
$Password = ’’;

# Even though it’s called the "Scheduler" the proper service name is "Schedule"
$Service = 'Schedule';

$PerlDrive = 'm:';
$PerlPath  = '\perl';
$LogFile   = '\\\\server\perlshare$\install.log';

$REMOTE_PERL_PATH = '\\\\server\perlshare$\perl';
$NTFS_CONVERSION_APP = "convert.exe";
$NTFS_CONVERSION_PARAM = "/FS:NTFS /V";

ConfigGroup( $Domain );
ConfigService( $Service, $User, $Password );
if( ConfigDrive( $PerlDrive ) )
{
    SecureDir( "$PerlDrive$PerlPath" );
```

```
    CopyDir( $REMOTE_PERL_PATH, "$PerlDrive$PerlPath" );
    ConfigPath( "$Dir\\bin" );
}
ConfigLastUser( "" );
RebootMachine();
Log( $LogFile );
print "Finished.\n";

sub Log
{
  my( $LogFile ) = @_;

  if( open( LOG, ">+ $LogFile" ) )
  {
    flock(LOG, 2);
    seek(LOG, 0, 2);
    print LOG Win32::NodeName(), "\t", scalar( localtime() ), "\n";
    flock(LOG, 8);
    close(LOG);
  }
}

sub RebootMachine
{
    Win32::AdminMisc::ExitWindows( EWX_REBOOT | EWX_FORCE );
}

sub ConfigLastUser
{
    my( $User ) = @_;
    my $Key;

    if( $HKEY_LOCAL_MACHINE->Create( 'SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon',
$Key ) )
    {
        $Key->SetValueEx( $Key, 0, REG_SZ, $User );
        $Key->Close();
    }
}

sub ConfigGroup
{
    my( $Domain ) = @_;

    $Domain .= "\\" if( '' ne $Domain );

    # Make sure that we add the Domain Admins to
    Win32::NetAdmin::LocalGroupAddUsers( '', 'Administrators', "$Domain" . "Domain Admins" );
}

sub ConfigPath
{
    my( $Dir ) = @_;
    my $RegexDir = "$Dir";
    my $Path;

    # First check to see if the $Dir is already in the path...
    $Path = Win32::AdminMisc::GetEnvVar( 'PATH' );

    # Prepare $Dir for a regex...
    $RegexDir =~ s/([.\\\$])/\\$1/g;
    if( $Path !~ /$Dir/i )
    {
        # Add $Dir to the system (not user) PATH
        Win32::AdminMisc::SetEnvVar( 'PATH', "$Path;$Dir" );
    }
    return( 1 );
}

sub CopyDir
{
```

```
    my( $RemoteDir, $DestDir ) = @_;

    print "Copying files from '$RemoteDir' to '$DestDir' ...\n";

    # XCOPY.EXE will autocreate the destination dir
    `xcopy "$RemoteDir\\*.*" "$DestDir\\*.*" /s`;
}

sub SecureDir
{
    my( $Dir ) = @_;
    my( $Perm, $Result );

    print "Securing the $Dir directory...\n";

    `md "$Dir"`;

    if( $Perm = new Win32::Perms( $Dir ) )
    {
        # Remove *all* entries...
        $Perm->Remove( -1 );

        $Perm->Allow( 'Administrators', FULL_CONTROL_FILE, FILE );
        $Perm->Allow( 'Administrators', FULL_CONTROL_DIR, DIR );

        $Perm->Allow( 'Everyone', READ_FILE, FILE );
        $Perm->Allow( 'Everyone', LIST_DIR, DIR );

        $Perm->Owner( 'Administrators' );

        $Result = $Perm->Set();
    }
    else
    {
        print "  Unable to create security descriptor. Directory is not secured.\n";
    }

    return( 0 != $Result );
}

sub ConfigDrive
{
    my( $Drive ) = @_;
    my %Info;

    print "Configuring the $Drive drive ...\n";

    # Make sure that the drive exists
    if( join( ' ', Win32::AdminMisc::GetDrives() ) !~ /$Drive/i )
    {
        print "  The $Drive drive does not exist.\n";
        return( 0 );
    }

    # Do we have a fixed hard drive?
    if( DRIVE_FIXED != Win32::AdminMisc::GetDriveType( $Drive ) )
    {
        print "  The $Drive drive is not a local fixed hard drive. Skipping drive formatting.\n";
        return( 0 );
    }

    # If the drive is not NTFS then convert it into NTFS
    %Info = Win32::AdminMisc::GetVolumeInfo( $Drive );
    if( 'NTFS' ne $Info{FileSystemName} )
    {
        print "  The $Drive drive is not an NTFS drive.\n";
        print "  Converting from the $Info{FileSystemName} format...\n";
        @{$Log{ntfs_conversion}} = `$NTFS_CONVERSION_APP $Drive $NTFS_CONVERSION_PARAMS`;
    }

    return( 1 );
```

```
}

sub ConfigService
{
  my( $Service, $User, $Password ) = @_;

  my $OpenSCManager = new Win32::API( 'advapi32.dll', 'OpenSCManager', [ P,P,L ], L );
  my $OpenService = new Win32::API( 'advapi32.dll', 'OpenService', [ L,P,L ], L );
  my $CloseServiceHandle = new Win32::API( 'advapi32.dll', 'CloseServiceHandle', [ L ], I );
  my $SCHandle, $ServiceHandle;
  my $ServiceString, $UserString, $PasswordString;
  my %ServiceStatus;
  my $UnicodeFiller = "";

  print "Configuring the $Service service ...\n";

  $Result = Win32::Service::GetStatus( '', $Service, \%ServiceStatus );

  # Service status 1 == Service has stopped
  $Result = Win32::Service::StopService( '', $Service ) if( 1 != $ServiceStatus->{CurrentState}
);

  # Convert the string to UNICODE if needed
  $UnicodeFiller = "\x00" if( Win32::API::IsUnicode() );
  ( $ServiceString = $Service ) =~ s/(.)/$1$UnicodeFiller/g;
  ( $UserString = $User ) =~ s/(.)/$1$UnicodeFiller/g;
  ( $PasswordString = $Password ) = s/(.)/$1$UnicodeFiller/g;

  # The flag 0x00F003F requests to open the service manager with full access
  if( $SCHandle = $OpenSCManager->Call( 0, 0, 0x000F003F ) )
  {
    # The flag 0xC0000000 requests GENERIC_READ and GENERIC_WRITE
    if( $ServiceHandle = $OpenService->Call( $SCHandle, $ServiceString, 0xC0000000 ) )
    {
        my $ChangeServiceConfig = new Win32::API( 'advapi32.dll', 'ChangeServiceConfig', [
L,L,L,L,P,P,P,P,P,P,P ], I );

        # The 0x00000010 flag represents that the service will logon as a user account
        # AND it will create it's own process (not share process space with other services)
        # The 0xFFFFFFFF flag represents no change to this attribute.
        # The 0x00000002 flag represents the service is to auto start
        my $Result = $ChangeServiceConfig->Call( $ServiceHandle,
                                                 0x00000010,
                                                 0x00000002,
                                                 0xFFFFFFFF,
                                                 0,
                                                 0,
                                                 0,
                                                 0,
                                                 $UserString,
                                                 $PasswordString,
                                                 0 );
        if( $Result )
        {
            print "Granting the $User account the privilege to logon as a service\n";

            # Grant the service account with the privilege to logon as a service...
            $Result = Win32::Lanman::GrantPrivilegeToAccount( '', 'SeServiceLogonRight', [$User]
);
            if( ! $Result )
            {
                print "  Could not grant the privilege: ";
                print Win32::FormatMessage( Win32::Lanman::GetLastError() );
            }
        }
        else
        {
            print "  Could not modify the '$Service' service: ";
            print Win32::FormatMessage( Win32::GetLastError() );
        }
```

```
        $CloseServiceHandle->Call( $ServiceHandle );
    }
    else
    {
        print "Could not open the '$Service' service: ";
        print Win32::FormatMessage( Win32::GetLastError() );
        }
    $CloseServiceHandle->Call( $SCHandle );
  }
  else
  {
    print "  Could not open the service manager: ";
    print Win32::FormatMessage( Win32::GetLastError() );
  }

  Win32::Service::StartService( '', $Service );
}
```

7.2 Script 2: The System management engine.

This script is the main engine. This is the script that is executed once a day. From this script all the other scripts will be loaded and executed.
This script requires the following extension:

- Win32::NetAdmin

```
# SysMan.PL
# ----------
# This is the management system engine script. This is the
# core to the entire management system. This script will
# locate any domain controller and begin running scripts
# from there.
#
#   Dave Roth
#   Roth Consulting
#   http://www.roth.net
#

use Win32::NetAdmin;

$REMOTE_PERL_SHARE = 'perlshare$';
$REMOTE_PERL_DIR   = 'perl';
$REPOSITORY_DIR    = 'scripts';

if( Win32::NetAdmin::GetAnyDomainController( '', '', $Server ) )
{
  my @Scripts, $Script;

  $ScriptDir = "$Server\\$REMOTE_PERL_SHARE\\$REPOSITORY_DIR";
  @Scripts = glob( "$ScriptDir\\*.pl" );
  foreach $Script ( sort( @Scripts ) )
  {
    # We only will process scripts that exist and are files
    next unless( -f $Script );

    print "Loading and running '$Script'...\n";
    require $Script;
  }
}
else
{
  print "Unable to locate remote repository.\n";
}
```

7.3 Script 3: An ODBC configuration script.

This script installs the specified ODBC driver and configures a data source name so that scripts will have database access. This script will:

10. Install and configure the specified ODBC driver if it is not already installed.

11. Configure a new data source name (DSN) if it is not already configured.

This script requires the following extension:

- Win32::ODBC

```
# _ODBC.PL
# ----------
# This is a management system script designed to check for and
# install the MS SQL Server ODBC driver. It also checks
# for a particular ODBC DSN. If it does not exist it will
# be created.
# THIS SCRIPT assumes that ODBC has been already installed
# on the machine.
# Notice that the script name has been prepended with an underscore.
# This will cause this script to be run before other scripts.
#
#   Dave Roth
#   Roth Consulting
#   http://www.roth.net
#

use Win32::Registry;
use Win32::ODBC;

$Dsn = "ConfigServer";
$ODBCDriver = "SQL Server2";

$DestDir = "$ENV{WINDIR}\\System32";
$REMOTE_DRIVER_DIR = '\\\\server\perlshare$\ODBC\SQLServerDriver';

%DriverList = Win32::ODBC::Drivers();
if( ! defined $DriverList{$ODBCDriver} )
{
  my $Key;

  # Copy all the SQL Server ODBC Driver files...
#  `xcopy "$REMOTE_DRIVER_DIR\*.*" "$DestDir\*.*" /s`;

  # We need to install the SQL Server driver.
  if( $HKEY_LOCAL_MACHINE->Create( "SOFTWARE\\ODBC\\ODBCINST.INI\\$ODBCDriver", $Key ) )
  {
    my $SubKey;

    $Key->SetValueEx( 'APILevel', 0, REG_SZ, "2" );
    $Key->SetValueEx( 'ConnectFunctions', 0, REG_SZ, "YYY" );
    $Key->SetValueEx( 'CPTimeout', 0, REG_SZ, "60" );
    $Key->SetValueEx( 'Driver', 0, REG_SZ, "$DestDir\\SQLSRV32.DLL" );
    $Key->SetValueEx( 'DriverODBCVer', 0, REG_SZ, "03.50" );
    $Key->SetValueEx( 'FileUsage', 0, REG_SZ, "0" );
    $Key->SetValueEx( 'Setup', 0, REG_SZ, "$DestDir\\SQLSRV32.DLL" );
    $Key->SetValueEx( 'SQLLevel', 0, REG_SZ, "1" );

    if( $Key->Create( 'FileList', $SubKey ) )
    {
      $SubKey->SetValueEx( 'CTL3D32.DLL', 0, REG_SZ, "$DestDir\\CTL3D32.DLL" );
      $SubKey->SetValueEx( 'DBNMPNTW.DLL', 0, REG_SZ, "$DestDir\\DBNMPNTW.DLL" );
      $SubKey->SetValueEx( 'DRVSSRVR.HLP', 0, REG_SZ, "$DestDir\\DRVSSRVR.HLP" );
      $SubKey->SetValueEx( 'MSVCRT40.DLL', 0, REG_SZ, "$DestDir\\MSVCRT40.DLL" );
      $SubKey->SetValueEx( 'SQLSRV32.DLL', 0, REG_SZ, "$DestDir\\SQLSRV32.DLL" );

      $SubKey->Close();
    }
    $Key->Close();

    # Update the installed driver list...
    if( $HKEY_LOCAL_MACHINE->Create( 'SOFTWARE\ODBC\ODBCINST.INI\ODBC Drivers', $Key ) )
    {
      $Key->SetValueEx( $ODBCDriver, 0, REG_SZ, "Installed" );
      $Key->Close();
    }
  }
```

```
}

%DSNList = Win32::ODBC::DataSources();
# We should probably perform a better test to take into account
# mixed case...
if( ! defined $DSNList{$Dsn} )
{
    if( ! Win32::ODBC::ConfigDSN( ODBC_ADD_DSN, $ODBCDriver, ("DSN=$Dsn", "Description=The Configuration Database", "Server=dbserver", "Trusted_Connection=Yes", "Database=ConfigLogs" ) ) )
    {
        print "Unable to create DSN: " . Win32::ODBC::Error() . "\n";
    }
}
print "Finished.\n";
# We must return a 1 value to indicate this script was successfully loaded
return( 1 );
```

7.4 Script 4: A script to update a database with computer statistics.

This script will setup a machine so that it is ready to use the management system. This script will:

12. Gather information about the computer, processor and operating system.
13. Gather information about each drive.
14. Update a database with the gathered information.

This script requires the following extensions:

- Win32::ODBC
- Win32::AdminMisc

```
# OS.PL
# ----------
# This is a management system script designed to discover
# information about both the computer and OS. Once the
# information has been obtained it is then set to a database
# where it is stored.
#
# This script assumes that the database has 2 tables:
#   1) Computer:
#       ID          int(), primary key, autoincrimenting
#       Name        char
#       Processor   char
#       Speed       int
#       OS          char
#       ServicePack char
#       Version     float
#       Build       int
#       MMX         int (used as a boolean or bit)
#       Ram         int
#       PageFile    int
#
#   2) Drives:
#       ID          int, primary key, autoincrimenting
#       Size        int
#       Free        int
#       Drive       char
#       Type        char
#       Computer    int
#
#   Dave Roth
#   Roth Consulting
#   http://www.roth.net
#

use Win32::AdminMisc;
use Win32::ODBC;

$DSN = "Machines";
$Machine = Win32::NodeName();

%DriveType = (
```

```
    &DRIVE_REMOTE    => "remote",
    &DRIVE_REMOVABLE => "removable",
    &DRIVE_FIXED     => "fixed",
    &DRIVE_CDROM     => "cdrom",
    &DRIVE_RAMDISK   => "ramdisk",
);

# Get memory info...
%Mem = Win32::AdminMisc::GetMemoryInfo();

# Get processor info...
%Processor = Win32::AdminMisc::GetProcessorInfo();

# Get OS info then fix the servicepack and platform values...
%OS = Win32::AdminMisc::GetWinVersion();
( $OS{ServicePack} ) = ( $OS{CSD} =~ /(\d*?)$/ );
$OS{Platform} =~ s/win32_//i;

# Get drive info...
foreach $Drive ( Win32::AdminMisc::GetDrives() )
{
    my $Type = Win32::AdminMisc::GetDriveType( $Drive );

    if( DRIVE_REMOTE != $Type )
    {
        $Drives{$Drive} = { size => 0, free => 0 };

        if( DRIVE_FIXED == $Type )
        {
            ($Drives{$Drive}->{size}, $Drives{$Drive}->{free} ) =
Win32::AdminMisc::GetDriveSpace( $Drive );
        }

        $Drives{$Drive}->{type} = $Type;
        if( DRIVE_REMOVABLE != $Type && DRIVE_CDROM != $Type )
        {
            my %Volume = Win32::AdminMisc::GetVolumeInfo( $Drive );
            $Drives{$Drive}->{filesystem} = $Volume{FileSystemName};
        }
    }
}

if( $db = new Win32::ODBC( $DSN ) )
{
    $Id = GetDbId( $db, $Machine );
    if( ! $Id )
    {
        $db->Sql( "INSERT INTO Computer (Name) VALUES ('$Machine')" );
        $Id = GetDbId( $db, $Machine );
    }

    if( $Id )
    {
        $Update = "UPDATE Computer SET " .
               "  ServicePack = '$OS{ServicePack}', " .
               "  Version     = $OS{Major}.$OS{Minor}, " .
               "  OS          = '$OS{Platform}', " .
               "  Build       = $OS{Build}, " .
               "  Processor   = '$Processor{ProcessorType}', " .
               "  Speed       = $Processor{Win32ProcessorSpeed}, " .
               "  MMX         = $Processor{MMX}, " .
               "  Ram         = $Mem{RAMTotal}, " .
               "  PageFile    = $Mem{PageTotal} " .
               "WHERE ID = $Id";
       if( $db->Sql( $Update ) )
       {
         print "Could not update computer info: " . $db->Error() . "\n";
       }

       if( $db->Sql( "DELETE Drives WHERE Computer=$Id" ) )
       {
```

```
            print "Could not delete drive info: " . $db->Error() . "\n";
         }

         foreach $Drive ( sort( keys( %Drives ) ) )
         {
           my $Insert = "INSERT INTO Drives " .
                        "(Drive,Size,Free,Type,Computer) " .
                        "VALUES ( '$Drive', $Drives{$Drive}->{size}, " .
                        "$Drives{$Drive}->{free}, '$DriveType{$Drives{$Drive}->{type}}', " .
                        "$Id )";
           if( $db->Sql( $Insert ) )
           {
              print "Could not insert drive $Drive info: " . $db->Error() . "\n";
           }
         }
    }
    $db->Close();
}
else
{
    print "Unable to connect to the database: " . Win32::ODBC::Error() . "\n";
}

print "Finished.\n";

sub GetDbId
{
    my( $db, $Computer ) = @_;
    my( %Data );
    if( ! $db->Sql( "SELECT DISTINCT * FROM Computer WHERE Name like '$Computer' " ) )
    {
        if( $db->FetchRow() )
        {
            %Data = $db->DataHash( 'ID' );
        }
    }
    return( $Data{ID} );
}
# We must return a 1 value to indicate this script was successfully loaded
return( 1 );
```

7.5 Script 5: A script to report event log warnings and errors.

This script will setup a machine so that it is ready to use the management system. This script will:

15. Gather information from the local event log.
16. Update a database with the gathered information.

This script requires the following extensions:

- Win32::ODBC
- Win32::EventLog

```
# EVENTS.PL
# ----------
# This is a management system script designed to discover
# eventlog errors and warnings. Such data is procured and
# submitted to a database where it is stored.
#
# This script assumes that the database has 2 tables:
#   1) Computer:
#       ID          int(), primary key, autoincrimenting
#       Name        char
#
#   2) Events:
#       Source      char
#       Event       char
#       Type        int
#       Userid      char
#       Computer    int
#       Time        datetime
#
#   Dave Roth
```

```
#   Roth Consulting
#   http://www.roth.net
#

use Win32::EventLog;
use Win32::ODBC;

$DSN = "Machines";
$Machine = Win32::NodeName();

$SecPerDay = 24 * 60 * 60;
$Now = time();
$TimeLimit = $Now - ( $SecPerDay * 1 );

if( $db = new Win32::ODBC( $DSN ) )
{
  my $Id = GetDbId( $db, $Machine );
  if( $Id )
  {
    foreach $Source ( 'System', 'Application', 'Security' )
    {
      my $Event = new Win32::EventLog( $Source, $Machine );
      if( $Event )
      {
        if( $Event->GetNumber( $Num ) )
        {
          my %EventData;

          $Flag = EVENTLOG_BACKWARDS_READ | EVENTLOG_SEQUENTIAL_READ;
          do
          {
            if( $Event->Read( $Flag, $Num, \%EventData ) )
            {
              # Escape all ' chars for the sake of SQL syntax...
              map{ $EventData{$_} =~ s/'/''/g; } ( keys( %EventData ) );

              # We are only going to log errors...
              if( ( $EventData{EventType} & EVENTLOG_ERROR_TYPE )
                  || ( $EventData{EventType} & EVENTLOG_WARNING_TYPE ) )
              {
                my $Insert;
                my @Date = localtime( $EventData{TimeGenerated} );
                my $DateStamp = sprintf( "{ts '%04d-%02d-%02d %02d:%02d:%02d' }",
                                         $Date[5] + 1900,
                                         $Date[4] + 1,
                                         $Date[3],
                                         $Date[2],
                                         $Date[1],
                                         $Date[0] );
                my $Insert = "INSERT INTO Events " .
                             "(Source, Event, Type, Userid, Time, Computer) " .
                             "VALUES ('$EventData{Source}', '$EventData{Event}', " .
                             "$EventData{EventType}, '$EventData{User}', " .
                             "$DateStamp, $Id )";
                if( $db->Sql( $Insert ) )
                {
                  print "Unable to insert event data: " . $db->Error() . "\n";
                }
              }
            }
            else
            {
              undef %EventData;
            }
            # This will cause the next reading of the registry to move to the
            # next record automatically.
            $Num = 0;
          } while( $TimeLimit < $EventData{TimeGenerated} );

          Win32::EventLog::CloseEventLog( $Event->{handle} );
        }
```

```
      }
    }
  }
  $db->Close();
}
else
{
  print "Unable to connect to database: " . Win32::ODBC::Error() . "\n";
}

sub GetDbId
{
    my( $db, $Computer ) = @_;
    my( %Data );
    if( ! $db->Sql( "SELECT DISTINCT * FROM Computer WHERE Name like '$Computer' " ) )
    {
        if( $db->FetchRow() )
        {
            %Data = $db->DataHash( 'ID' );
        }
    }
    return( $Data{ID} );
}
# We must return a 1 value to indicate this script was successfully loaded
return( 1 );
```