Maintaining Node Symmetry

In a Networked Windows NT Environment

Scott M. Lewandowski
Senior Honors Thesis
May 1999

Department of Computer Science
Brown University
Providence, RI 02912-1910
scl@cs.brown.edu

1. Introduction

One of the problems with the Windows NT operating system is the lack of a framework for installing applications or system updates to nodes on a network. When system administrators want to install applications or update system software, they must go to each node to perform the installation, request that the user perform the installation, or prepare a complex installation script that allows unattended installation. These solutions are all with substantial fault; they require either substantial effort by the administrator or the user to be trusted to perform the installation correctly. A system that could effectively generate an automatic installation script by example would not suffer from any of the problems that the solutions above entail.

To this end, we have developed a system that can "watch" an application installation and generate an installation package that can easily be installed over the network by the administrator from any node on the network. If an application or system software update is to be installed on all nodes on the network, all nodes on the network are kept in a consistent state. If an application is installed on only a subset of the nodes, it keeps software at the same revision level on those nodes and helps identify and resolve conflicts that may occur with other installed applications.

The work described herein is the result of the author's collaborative work with Professor Robert H.B. Netzer in his pursuit of a degree in Computer Science with Honors in May 1999.

2. Problem Definition

We endeavor to solve the application installation problem by developing an application that is able to maintain symmetry across nodes in a Windows NT software environment. Using such a system allows administrators to make changes on any node on the network and then have all other nodes updated to reflect the changes they just effected. When a node is added to the network, an administrator can quickly bring the new node to the same state as the other nodes on the network by batching the software installations to date.

The need for some of the node state to remain distinct to a given node complicates matters substantially. For example, each node will have a unique name and IP address; these must be preserved when an installation is applied to a node. Another example is that some nodes may have applications installed on them that are not to be made available on all nodes in the network. To be useful, an application like the one we propose will clearly need to account for such issues.

Maintaining node state involves both the filesystem and the registry. The registry is a filesystem-like system for storing small amounts of data in an organized fashion. The registry is well suited to store certain types of data because it provides very efficient data access.

While such a tool is useful in and of itself, in the course of analyzing the problem domain, we realized that the system components needed to build such an application could be leveraged to provide much more robust node management services. For example, we wanted to provide the ability to checkpoint a node so it could be restored to a consistent state should the software configuration become corrupted. The components comprising the system can be used to create an "rdist" type of application or to synchronize folders. Finally, the components would support a robust application uninstallation feature that is actually able to restore a node to its state immediately preceding application installation instead of just removing the files installed by the application.

3. Current "Solutions"

The difficulties in maintaining a large Windows NT network are well documented, and only recently have vendors released solutions to tackle some of the issues that administrators are faced with when trying to orchestrate support for a large network. Good solutions to these problems are available for other platforms, such as Solaris; in fact, the Brown University Computer Science Department uses a solution that can reinstall a node without user intervention and handles node-specific data.

Windows NT has an option for unattended installation, which involves preparing a complex installation script and having the Windows NT media accessible, either on a networked share or local to the machine (e.g., on a CD-ROM). This technique is acceptable for performing basic installations of NT, but falls short when complex options are required. Furthermore, some of the prompts are difficult, if not impossible, to remove, making the installation somewhat less than "unattended". The process is not well documented, either. An unattended installation of Windows NT does not install any applications beyond those included in a standard Windows NT installation (e.g., clock, telnet, notepad, etc.).

The official option for unattended and automated application installation that is supported by Microsoft is to use a utility called "sysdiff". Sysdiff was designed to allow applications to be included with Windows NT installations performed on machines using the unattended installation feature of Windows NT. Sysdiff allows an administrator to create and apply "difference packages" from installation directories that it creates. To use sysdiff, a user snapshots the system before installing applications. The user then installs the applications, and asks sysdiff to generate a difference package, which reflects the changes made to the computer. This difference package is then applied on other nodes, by duplicating the changes stored in the difference file on those nodes.

Sysdiff requires that the computer the "diff" is performed on be identical to the one that will have the package applied to it. This is not suitable for many environments, since it precludes its use when some nodes have applications that others do not. Sysdiff also requires a special .ini file to work correctly, and fine tuning this file can be tedious. To use sysdiff, code must be executed on the target machine, which complicates administration and debugging of installation issues, and prevents an administrator from installing applications from a single node.

Furthermore, our testing of sysdiff revealed that the program is bug-ridden. For example, on several systems we tested on, the differencing procedure halted prematurely with an error code of 32. It is possible that this error results from sysdiff's inability to handle files that are in use, which is a major problem that a system like the one we propose must deal with.

The initial setup of Windows NT can actually be accomplished much more efficiently using third-party utilities. Norton's Ghost product is one example of such a product. It can take a drive image and send it to one or more nodes via the network and replicate the image on that system's disk. This is a decent alternative to NT's default unattended installation scheme, and can be used to start a node off with some applications preinstalled. This solution falls short, however, when updates are to be performed on nodes that have been customized through the addition of software, user settings, and the like. The only way that Ghost can update nodes is by replacing the disk contents with the new image file; in the process, all changes a system's user has made are lost.

None of these solutions allow for the uninstallation of applications. The only commercial product we are aware of that allows for uninstallation of applications is WinInstall from Seagate. WinInstall is useful for managing a network from a single node, but it is not very useful for performing unattended application installations since it requires a great deal of user interaction and cannot handle many special cases.

4. Key Features and Benefits

Our system provides many benefits over currently available installation technologies.

Our system allows the administrator to define logical networks of machines that can be treated independently. For example, some machines on the physical network could be designated as being lab machines and others could be designated as research machines. Each group would be treated as its own "miniature network", and changes to one group of nodes would be kept separate from changes made to the other set of nodes. Furthermore, both groups could be administrated from a single workstation. The logical networks that the administrator defines can be changed at any time without disrupting installation services.

Installation and removal is performed in such a way that it is robust even when various failures occur, such as node crashes, or network problems. In contrast, the products discussed above can fail under many circumstances; in most cases, failure is unrecoverable, and the node must be reformatted and installation must begin from scratch, causing the loss of any applications or data unique to that node.

The issue of fault-tolerance extends to other capabilities of our system, as well. Since past application installs are stored by our system, new nodes can be added to the network and can be brought to the current level of application installation and customization. This also means that when a node has a problem, the administrator can simply reformat it, reinstall Windows NT (using one of the products described above), and ask our system to update the node. This is a sharp contrast with current system administration techniques, which

would cause the administrator to spend a long time trying to "fix" a node with the hope of saving application reinstallation and configuration time before finally giving up on it. Our system also allows administrators to revert to past system states by uninstalling applications. This too can prevent disaster when an application deployed on a network is later found to be undesirable for some reason.

Finally, the simple customization model enables our system to serve multiple roles. An administrator can use the system to maintain a network as described above, and can also use it for other purposes, such as keeping notebook users' desktop and laptop data directories synchronized. No products available today have the flexibility to provide such rdist-like capabilities.

5. Required Functionality

There are two primary areas of functionality to be provided by our system:

- Application installation. An administrator installs an application onto one node, the application is propagated to other nodes in the system. The administrator can request propagation to all nodes, to a subset of nodes, or to no other nodes. Both local files and registry keys are propagated; our system does not concern itself with data that applications write to remote shares.
- Application removal. Removal from one or more nodes occurs by restoring the node to the state it was in just prior to application installation. This removal capability is value-added over what typical application installers provide: uninstallers do not restore the previous state of files (for example, a DLL that was overwritten with a different version). In addition, not all applications provide an uninstaller, and of those that do, many do not uninstall correctly; often files are left on the node, and files that other applications rely on can be removed. Our application removal mechanism overcomes these difficulties.

6. System Design

Our system consists of the following functional components, which will be integrated into a single executable.

6.1 Core Application

A GUI-based application will coordinate the installation and removal of user applications as well as NT service packs, hotfixes, and other system software updates. This application can be run from any node on the network.

6.2 Application installation

Installs of an application is accomplished by coordinating several high-level functions:

- *Delta.* A delta approach is used to compute what local changes are made to a node by an application's installation or removal. A *delta* consists of the files and registry settings created, deleted, or changed by application installation or removal. The information in a delta is sufficient to install the application on another node. Conceptually, a delta is computed by comparing the state of the file system and registry before and after an installation, as follows:
  - Scan the system to collect information about its current state. Such information includes data about each file on the local system disk and a list of registry keys and their current values.
  - Perform the actual installation of the application to be installed. This may include rebooting the node on which the installation is taking place. The administrator can take this opportunity to configure the new application if they want to propagate configuration settings to other nodes.
  - Re-scan the system to determine the files and registry keys that were changed, added, or deleted. The state of these files and keys at this point make up the delta.
- *Propagation.* After the delta has been computed, the files and registry settings included in the delta are propagated to zero or more other nodes as desired.
  - If a file or registry key to be updated on a target node is not busy, the update is directly applied to the node and a reboot is not required[1]. If the file or registry key to be updated is in use, the change will be scheduled to occur on the next node reboot (see below).
  - When nodes cannot be completely updated due to busy files, our system will inform the user and optionally begin a reboot of all nodes.
  - The delta is saved so it can be applied to new nodes that are later added to the network, or existing nodes that were down for more than one update. This ensures that new nodes can be made identical to other nodes on the network. These nodes will be updated by a single batch update containing the aggregation of all missed updates. Registry updates can also be applied in batch.
  - Our system provides an exclusion list mechanism to specify that certain files and registry settings are node specific, allowing each node to retain a certain amount of distinctiveness.

6.3 Application removal

The application removal function is similar to application installation.

- *Regression delta.* A *regression delta* consists of the original state of the files and registry settings contained in the corresponding delta (i.e., the state immediately before the installation took place). This information allows nodes to be brought back to earlier software revision levels.

[1]Note that some registry settings may require a reboot to take effect. A database of known scenarios where this is the case could be consulted and, if appropriate, the user could be prompted to reboot the node or to defer the changes.

- *Propagation.* The regression delta will be propagated using the same propagation engine used to apply deltas (described above). This reverts each node to its state immediately preceding installation of the application.

7. Assumptions and Limitations

- All application installation and removal must be performed through our system. There will be no mechanism to enforce this; it will be up to the system administration follow this guideline. If an installation is not performed through our system, it has no way of knowing that the administrator really meant to do an installation. This can cause the following problems:
  - The correct versions of files may not be retained on future installations when applications installed on only a subset of nodes are involved[2].
  - The application could not be uninstalled, since there was no time to perform a filesystem scan before installation.
  - Propagation would not occur.
- Because our propagation mechanism copies the updated files and registry keys, there are two cases where an application can fail to run on the nodes to which it is propagated:
  - The application uses information specific to the node on which it is installed, such as the CPU ID's or the node's device configuration. This information is invalid for other nodes. Esoteric applications such as copy protection schemes or disk defragmenters are examples.
  - The application adds to files or registry keys that the operating system (or other applications) also modify. For example, an application may modify the system.ini file, which all 16-bit applications and the OS use to store configuration information. Our system is not able to merge such changes into the corresponding files on other nodes.
- Multiple file streams are not supported. Few, if any, commercially available applications use this feature of NTFS.
- When installing an application across all nodes, our system relies on the application's installer to flag and resolve file-version conflicts, as is done in normal NT application setup. However, an exception is made for applications that conflict with others installed on only a subset of nodes, because in this case there can be different conflicts on the various target nodes and application installers are unable to detect those (because they are looking only at the local node). Our system will flag two types of file-version conflicts:
  - Two or more applications install and require a file with the same name but that are of a different version or are completely unrelated. Modern installers typically

[2]For example, a node starts with version 1 of a given DLL installed. Suppose a user installs an application on that node without using our system and that the application installs version 2 of the same DLL. An application that installs version 3 of the DLL is then installed onto another node and propagated, overwriting version 2 of the DLL. If the application that was propagated is uninstalled, the node in question should be left with version 2 of the DLL, since that is the DLL that was installed immediately prior to installation. However, version 1 will be restored, since our system has no knowledge of version 2; this could cause the application to malfunction.

do not take steps to resolve this issue, although such conflicts are not prevalent in commercial software.[3]

- An application wants to overwrite a newer version of a DLL with an older version, potentially "breaking" applications that rely on the functionality of the newer DLL. Some installers do not check the versions of the files they are overwriting, even though this violates the Windows NT installation guidelines.

- If the installation of one application requires one or more node reboots, then these reboots must be performed before another application is installed. There will be no mechanism to enforce this. This requirement is currently imposed by the normal Windows NT installation mechanism. Installing two applications consecutively without rebooting after the first can lead to unpredictable results[4].
- It is assumed that the node our system is run from can access other nodes' system disks. This requires full administrative access to all nodes and can be implemented using the default system shares setup by Windows NT.
- Our system relies on the availability of sufficient space on a shared NTFS partition that it can use for temporary files, delta storage, etc.

8. Component Design

Our system is implemented as a user-runnable application and a collection of cooperating modules providing packaged functionality.

8.1 Core Application

When a system administrator wishes to perform an install, they will run the our core application. It will have a simple GUI that guides them through the installation procedure. It will coordinate the following functions:

1. Pre-installation step:
   - Ensure that installations are not taking place elsewhere on the network.
   - Perform any pre-installation scanning of the system required to compute the delta (see below).

[3]For example, two applications may both create a file c:\log.txt; another potential conflict are two applications that require two different versions of DLLs (in this case, the application that requires the older DLL relies on functionality in the DLL not included in the newer version, and the application using the newer DLL relies on functionality not included in previous versions).

[4]An example of this occurs when two applications try to update the same DLL to a version newer than what the user currently has installed. Suppose v2.0 of the DLL is on the system and the user installs application A, which requires and ships with v2.2 of the DLL. The installer checks to see if the installed DLL must be updated, and it finds that it must be. However, the DLL may be in use (and subsequently cannot be replaced), so the installer makes the required registry entry to have the DLL updated the next time the machine is restarted. Suppose a user then installs application B, which requires and ships with v2.1 of the DLL. It checks the currently installed DLL for the version number, and seeing that it is only v2.0, makes a registry entry to have v2.1 installed on next reboot. When the machine is rebooted, v2.2 of the DLL replaces v2.0, and then v2.1 replaces v2.2, leaving application A non-functional. Had the machine been rebooted after installing application A, this problem would not have occurred since the application B installer would have detected that a newer version of the DLL was already installed.

2. Wait in the background (perhaps across one or more node reboots) while the administrator installs new software or otherwise changes the node's configuration.

3. After the installation completes, if the installation process requires the machine to be rebooted, a reboot must be performed before the installation can proceed. This decision was made so that when the post-installation step is performed, the application will be fully installed[5]. Our application is automatically re-invoked after reboot.

4. Once the application installation has completed (including any required reboots), the administrator is given the opportunity to configure or partially configure the application. This provides the ability to propagate configuration settings along with the application, perhaps avoiding the need to manually configure each node in some cases. Finally, the administrator is given the option of continuing with the installation and updating the system revision level, or aborting the installation, in which case the regression delta can be computed and applied to revert the node to its state before the installation began.

5. Post-install step:
   - Perform any post-install scanning of the system required to compute the delta and regression delta (see below).
   - Propagate the delta to all other running nodes (see below).
   - Save the delta and regression delta to an NTFS-formatted administrative area for later use.

8.2 Delta and Regression Delta Computation

8.2.1 Delta computation

To determine which files on the local system disk are updated by an install, our system will perform a file system scan. The Win32 API will be used to traverse the entire local file system. For each file visited, a checksum can be generated; by comparing checksums, two versions of a file can be compared to see if they differ (it is possible to also use other file attributes such as version, timestamp, and size). This approach only identifies changed files; the type of change is not known. Certain files or directories can be left untouched by excluding them from the scan, or the results of the scan can be post-processed to remove records for files or directories that are to be excluded.

[5]The alternative is to propagate before a reboot. Many installers schedule file moves and copies to occur on the first reboot after installation using MoveFileEx(). If propagation occurred before rebooting the node, the scheduled move or copy as well as the original file would be replicated on each node, and then each node would need to be rebooted to effect the change. This is not a substantial problem, since each target node will likely need to be rebooted since the propagation mechanism is based on the same principles as MoveFileEx(). However, two advantages are derived by propagating after a reboot: there is no need to resolve conflicting temporary filenames; and it provides the opportunity for the administrator to configure the application. This would not be possible if we were to propagate before the reboot, since changes made when configuring the application would not be propagated to other nodes.

To determine which registry entries are updated, a scan very similar to the filesystem scan will be used[6]. This approach provides the same benefits as the filesystem scan approach, including the ability to exclude certain registry keys.

Note that this approach means that changes effected by applications or the system while an installation is taking place will be included in the delta, and will therefore be propagated to all nodes and undone if the regression delta is applied. This should not be an issue under normal conditions, since the Windows NT guidelines recommend that all applications be closed before application installation occurs; this means that there should be no running applications to make changes between file scans that are not part of the installation process. There is no resolution to this problem, even if an alternate technique is used to detect changes files and registry entries (there is never a reliable way to determine who effected the change).

8.2.2 Regression delta computation

An application installation modifies a set of files and registry keys. The regression delta contains the original state of these files and keys at the point just before the installation began. Having the regression delta allows an application to be removed from a node. This feature is useful since Windows NT does not provide a standardized system for uninstalling applications.

A filesystem regression delta is computed as follows:

- A filesystem scan performed before application installation generates a system snapshot, which consist of images of the files changed since the last time a scan was performed.
- The application is installed by the administrator.
- A filesystem scan is performed immediately following application installation. The difference between this snapshot and the one generated immediately preceding the installation is the regression delta.
- The state of the files in the regression delta immediately preceding the installation can be found by checking system snapshots, from the most recent snapshot to the oldest snapshot, to see if any contain the file in question. The first occurrence of the file that is found in a snapshot is the state of the file immediately before installation; this is the image to be restored if regressing a node.

The registry regression delta will be computed in the same manner as the filesystem regression delta, using multiple scans. Like filesystem snapshots, registry snapshots can be used to restore previous configurations of the registry.

[6]The original design (in v1 of the document) called for using Global Registry hooks to receive notification of changes to the registry. This approach is not appropriate for our system since it will pass all changes (even temporary changes and multiple changes to the same key) to the registry to our system. This would require extensive processing to determine which changes should actually be stored. Furthermore, this approach does not allow our system to receive notification of registry changes that occur when our system is not running; this is important when changes are effected on node reboot, for example.

8.2.3 Alternatives to Filesystem Scan

Two alternatives were identified but determined to be inferior to the file scan solution. To be complete, we detail the arguments underlying this decision; readers uninterested in these details can skip forward to the next subsection.

- *Filter driver.* It is possible to install an NT device driver that monitors all file system activity. This file system driver would create and attach a filter device driver object to the actual file system's driver object so that all IRP and FastIO requests directed at the file system are captured. Using information stored internally, file handles can be mapped to file path names. With this approach, filtering to exclude certain activity or certain files/folders must happen as a post-processing phase (after all data is collected). Because of the performance penalty it imposes, the filter driver would be started and stopped when application install was to take place, and a mechanism to keep it installed during a node reboot would need to be implemented. The filter driver approach could fail since it would be unable to resolve file handles that were opened before the filter driver was started. However, such handles represent files that the install process did not itself open, so it is probably safe to assume that they have not been updated by the installation.
- *Changed file synchronization object.* The Win32 API features a synchronization object that can be used to identify additions to and deletions from a directory as well as changes to files contained in the directory. This function can be set for recursive application to monitor all files in a given directory hierarchy. The programmer can specify what changes to files and directories should affect the synchronization object. No explicit exclusion option is provided, but one can be simulated rather trivially by using multiple synchronization objects. A race condition may occur immediately following a detected file update; further investigation of this issue is required.

The synchronization object approach was rejected since synchronization objects were designed to monitor a small subset of the overall directory tree for changes and are therefore not suitable for our application. This approach would also be unable to detect changes effected on reboot.

The Filter Driver approach is not as desirable as the File System Scan for several reasons.

- File system scan would need to be implemented anyway in order to get the initial system baseline. This indicates duplicated development effort.
- Changes to the filesystem made between application installations will go undetected since the filter driver would not be active between installations due to performance overhead and stability concerns.
- The point in the boot that the filter driver would be loaded is not well documented, and may be subject to change in future releases of NT. In particular, it is essential that the driver be loaded before changes deferred until reboot are made (such as those effected by MoveFileEx()). Furthermore, the time at which the driver is loaded may be earlier than when it actually becomes active and able to monitor activity, since drivers are allowed some time to initialize. Commercially available products that rely

on filter drivers are plagued by this problem, and developers have been unable to resolve the issue.

- Communicating the information from kernel mode to a user mode process is resource intensive. A sample filesystem filter driver does this by moving information into a buffer that is periodically sent to a user mode process. This procedure requires a significant degree of processor time, and relies on synchronous transfer, which reduces overall system throughput. Furthermore, under periods of intensive activity, buffer overflows can occur, causing lost data. While it is probably possible to resolve this limitation, the developers of the utility were unable to do so without imposing unreasonable resource requirements or performance penalties.
- Information reported by a filter driver is very fine grained in nature. For example, the act of copying a modestly sized file could generate tens of transactions. Making sense of such a quantity of data would require extensive post processing. Numerous special cases would need to be accounted for (e.g., the reassignment of file handles during a transaction).
- Filter drivers are a very poorly documented (and in some areas, undocumented) aspect of NT, which is something which should be avoided. Furthermore, the driver structure for Windows NT 4 and Windows 2000 are different, so different versions of the driver would need to be maintained for each version of the OS. Future changes to NT are likely to require more modifications to the driver.

8.3 Propagation Engine

The propagation mechanism reliably applies local changes made to one node to other nodes in the system. It will handle node-specific files through a user-extensible update mechanism.

If new nodes later join the network, they can be updated to the current revision level upon their first boot into NT (the updates will be batched to make the operation as efficient as possible).

8.3.1 Update mechanisms

Updates that occur at reboot will be performed by an updated version of MoveFileEx(). Windows NT uses a Win32 function called MoveFileEx() to schedule updates to files that cannot be updated at that time to occur on node reboot; for example, open files cannot be updated, so updates are deferred until the next node reboot since files are guaranteed to be closed upon reboot. This function does not work for remote nodes (which are accessed through UNC names or mapped shares) since NT networking is not available when it performs the requested file copies; however, a replacement that eliminates this limitation can be developed. This replacement for MoveFileEx() will serve as an interface for copying and deleting files over the network. Other limitations that the MoveFileEx() replacement will overcome include checking to see if a file copy can occur immediately instead of on the next reboot (thus preventing unnecessary machine reboots) and verifying the syntax of the source and destination file specifications (not doing this could be problematic in maintaining high system availability and

stability). The updated version of MoveFileEx() will store information in the registry on the remote node so the node knows to do an update. The replacement function will also copy the file to the remote node, so the original master node need not be present; MoveFileEx() does not do this, which is not adequate to meet our requirements, since the file could change again before the remote machine is rebooted.

8.3.2 Node distinctiveness

Some files are node dependent; that is, they contain information unique to a certain node and should not be propagated to other nodes as part of the update process. A file exclusion facility will modify the list generated by the delta facility to account for items that should vary from node to node. The administrator can update a database of known unique files that are known to vary from node to node under normal circumstances.

8.3.3 Applications installed on only one node or a subset of nodes

Applications that are to be installed on only a subset of nodes pose a challenge for our system. Let us call such an application a "unique app." For the sake of illustration, assume that a unique app is installed only on node X. File-version conflicts can occur between applications installed on node X and other applications installed on other nodes. Our system will detect these conflicts, on a node-by-node basis, with checks similar to those performed by modern installers. Two types of conflicts can occur:

- Future installations may update files and local registry keys on node X that the unique-app install has already updated. Since this can cause applications to fail to function or jeopardize system stability, our system will include a mechanism to detect it. We will provide this by computing a delta for unique app installations. Then, future (normal) installations can compare their deltas with those of past unique-app installs and report any intersections.
- Past installations could have updated files or local registry keys that the unique app's install process subsequently updates. This could cause previously installed apps to behave differently on node X than on other nodes. Our system will detect such situations by comparing all past deltas to a unique app's delta and flagging any intersections.

It is up to the system administrator to determine whether conflicts are benign. They will be presented with a dialog box showing information about the conflicting files and they must choose which copy to keep. This is analogous to the dialogs presented by most application installers when file-version conflicts are detected locally. The difference is that here there can be different conflicts on different nodes.

8.3.4 Propagation Difficulties

During the course of normal propagation, several difficulties can be encountered, all of which can be resolved – all nodes are always updated correctly.

- A node is unavailable, perhaps because it is crashed, purposely inactive, or not yet installed. It will be batch-updated upon its next reboot and brought up to the most current software revision level.
- Files that need to be updated are in use. The updates will be scheduled to take place on the next reboot, and a node reboot will then be scheduled.

9. Implementation Notes

Most of the effort required to build our system involved designing the components and analyzing the application installation process to identify areas that needed attention. Determining all of the possible conflicts between installed applications and the proper resolution for each conflict, for example, was a time consuming process because of the number of variables involved. Development of the project is still in progress. Working prototypes of all basic filesystem features have been developed, and are being integrated into a final product.

Despite the fact that most of the effort was spent on design and analysis, a number of interesting implementation issues arose. Below we detail some of those issues and give an overview of the problem encountered and the solution we devised.

9.1 MoveFileEx()

The Win32 API provides a Windows NT-only function, MoveFileEx(), that is able to move a file or directory. Using the appropriate flags, a programmer can use it to move files and directories to and from any valid storage location, including a name specified using the Universal Naming Convention (UNC) format. Since sharing violations can occur in such scenarios, the MOVEFILE_DELAY_UNTIL_REBOOT flag can be specified; this causes the action to take place the next time the node is rebooted, eliminating the possibility that the file to be overwritten will be in use. Note that when the MOVEFILE_DELAY_UNTIL_REBOOT flag is specified, the system does not attempt to effect the file move at that time; the move is always deferred until reboot.

A successful call to MoveFileEx() that is made with the MOVEFILE_DELAY_UNTIL_REBOOT flag sets the PendingFileRenameOperations value in the registry. This value is located in the HKEY_LOCAL_MACHINE hive, and is a value used by the Session Manager (maintained under SYSTEM\CurrentControlSet\Control). When Windows NT is restarted, the session manager (smss.exe) initializes and completes the AUTOCHK routines, and then the file pairs in PendingFileRenameOperations are moved.

While MoveFileEx() solves some substantial problems inherent in the other file copy mechanisms provided by Win32 (e.g., CopyFileEx()), it has its own problems. The parameters are not checked, so the source or destination files specified may be invalid. The return value from MoveFileEx() just confirms that the information was written to the registry, not that the values are valid. Another problem is that MoveFileEx() cannot deal with UNC names when the MOVEFILE_DELAY_UNTIL_REBOOT flag is specified

since networking is not available when the moves are processed. Even were networking available, MoveFileEx() always updates the local registry instead of the registry on the remote computer the file is being moved to.

To overcome these problems with MoveFileEx(), we had to develop a custom version of the function to address these limitations. Our version takes the exact same arguments as MoveFileEx(), but handles the case when the source or destination file is specified using UNC or is on a mapped drive correctly. When the file to be moved should end up on another node, our replacement for MoveFileEx() copies the file to a temporary location on the target computer and updates that node's registry so the move will occur the next time that node is rebooted.

Our replacement function is able to determine if a remote file is involved by checking for a UNC filename (easy to detect, since it starts with \\) and by checking the drive type using GetDriveType(). If a networked file is involved, it is copied to be local to the node that the destination file will reside on. After the file is local, the registry must be modified, but the values passed into the function cannot be used—they may contain mapped drive letters, for example, which need to be resolved to something local to the destination machine. This may involve resolving multiple levels of indirection. For example, a drive Z: could be mapped to \\remotenode\share, which could be the D: drive of the node named "remotenode". Once the files are resolved, the registry value can be updated. The format is null-terminated strings representing source and destination files, ended by a double null. However, deleting a file is accomplished by specifying null as the destination string, so encountering a double null does not necessarily signify the end of the string. Fortunately, we need only append to the values already written, which is comparatively simple to handle.

Our replacement function detects invalid source and destination specifications and returns failure when these cases are encountered. It also handles failures at other points in its execution, such as copying files to make them local to the destination node, and reports them accordingly.

9.2 Persistence Layer

It is necessary to store trees on disk so that they can be retrieved later to compare states of the filesystem, batch updates, and the like. This can be done in many ways, but it was determined that it would be useful to design a reusable persistence layer that could be reused in other areas of the system and for future projects. A persistent object retains its state longer than the lifetime of the program that originally created it.

The Microsoft Foundation Class (MFC) library includes an implementation of a persistence layer, but there were several problems in using this implementation. The implementation we devised is able to coexist with MFC, but does not rely on it in any way. It uses templates and RTTI so that a remarkably small amount of code is required.

One of the problems with MFC is that it is not very object oriented. Modeling an application to use MFC can often lead to poor designs. In addition, if a design is dependent on MFC, a superior class library cannot be used easily in the future.

Another problem is that modeling a program around MFC leads to spurious inheritance; for example, to use MFC's persistence layer, your classes must inherit from CObject. The need to inherit from CObject can also lead to problems involving multiple inheritance, since all of the classes from which a class inherits most likely inherit from CObject, causing CObject to be inherited twice. This is a problem when providing persistence, since it makes it difficult to have the two superclasses store their state correctly. This problem cannot be solved using virtual base classes, since an ambiguity as to which constructor to use to initialize the common superclass results, and can be resolved only by violating encapsulation.

A third problem with MFC's serialization system is that it will not work with templates. Programmers desiring to use the flexible STL classes cannot use MFC's persistence mechanism since they all rely on templates. The reason MFC's persistence mechanism does not work with templates is that it relies on macros, which require that the class names be known at preprocessing-time.

9.2.1 Dynamic Object Creation

A basic component of any persistence system is the ability to create a default object of the correct type. When the object is restored, this default object is populated with the state stored when the object was written to disk. The mechanism for creating objects should be able to create objects of the correct type without the object that is trying to recreate the object knowing what type of object it is creating; this allows objects to serialize objects they have pointers to but do not know the exact type of.

Our implementation of dynamic creation relies on a "database" that can be indexed by class name. Each entry in the database can manufacture an object of the required type, simply by calling its default constructor. In our implementation, the database accepts a class name and returns a new object of that type if it is able to do so; otherwise, null is returned. Our approach relies on templates to create factories for the types of objects that will need to be created. The factories are needed only when reading objects from disk—they are not required once the objects have been recreated. This design eliminates the overhead present in some other persistence systems. The "database" used in our implementation is a simple linked list, which has proven to be plenty efficient given the number of factory objects that need to be managed in a typical application.

9.2.2 File I/O

To provide persistence, the objects must be written to storage. To provide maximum flexibility, we have abstracted this functionality, so that the store can be anything the user would like: a Win32 HANDLE, an ANSI-C file, or even a network stream. We created a base class that provides a standardized interface for an I/O store, and then derived a class from it that implements the interface for the specific I/O system we wanted to use.

The store interface consists of read and write functions, which take pointers to void* and length as parameters, and are also overloaded to handle many common types (e.g., ints, STL strings, etc.). The overloading occurs at the abstract class level so that the derived classes only need to override 2 functions.

9.2.3 Serialization

Classes that are to be persistent inherit from the Persistent superclass, which contains two functions that must be overridden. The first is version(), which should return a different version number every time the class is updated and the serialize() method changes. The methods that actually load and store the object check these version numbers to report conflicts. The serialize() method should be implemented to both read and write the file. It can interact directly with the persistent store using the read and write methods since the store is passed as a parameter. The serialize() method determines whether to read an object in or write to disk using the "direction" parameter. If a class inherits from another class, it must remember to call serialize() on that superclass.

9.2.4 Persistence

All objects that should be persistent are derived from Persistent, which provides a great deal of functionality, freeing the programmer from dealing with most of the issues encountered in writing objects to disk. For example, the Persistent superclass provides version-number support; an error is returned if you try to read a version from the persistent store that does not match the class definition being used at runtime.

When objects are to be written to disk, flush() should be called on the object. Flush() writes out information needed to recreate the object, such as type and version information, and then calls the object's serialize() function to get the relevant data written to disk. The load() function is used to restore an object from disk, and works by reading in the class name, dynamically creating an object of that type, verifying the version information, and then asking the created object to serialize() from the persistent store.

9.3 Data Structures

Trees that are generated from the filesystem scans are composed of file and directory nodes. File nodes contain all of the information pertinent to a file, and directory nodes contain similar information, with the addition of a linked list to hold the contents of the directory (both files and directories). STL was used for all required data structures.

We designed a class that is able to traverse a directory and call functions when it reaches new files or directories. Several functions are called when key events occur (e.g., move back up a directory level) so that subclasses can build robust data structures or perform complex operations as a directory is navigated. The tree is generated by a class derived from this class that creates the appropriate node types as files and directories are encountered. The tree is designed to promote the use of iterators, which can move throughout a tree and perform operations on nodes, or operations based on nodes. For example, an iterator to delete all entries in the tree from disk was written.

The operators + and – were overloaded for trees. This allows differences between trees to be computed. This is an important feature when trying to bring tree B to the state of tree A, since the difference between tree A and B is what needs to be added to B, and the difference between B and A is what needs to be deleted from B. The sum of tree is used to batch updates. To facilitate batching and differencing of trees, directories support these operations as well. Operator == was also overloaded to compare directories and files. For efficiency, files are compared using checksums instead of a byte-by-byte comparison. These checksums are generated only when needed, or when the file object is serialized to disk (this prevents expensive checksums from being generated without need). For directories to be equal, both directories much contain the exact same set of files and directories.

9.4 Fault Tolerance

Our system minimizes the necessity of rebooting a node by accurately determining when a node must be rebooted.

Our system is fully recoverable in the event of failure. Successful resolution of a problem may involve one or more node reboots, or correction of the underlying problem (e.g., making a share available again). Fault tolerance mandates that we be able to restore a node to its state immediately before the problematic installation was begun. Some potential failures include:

- System administrative becoming unavailable in the middle of propagation or checkpointing
- Our system or application installer crashes
- Node being propagated to or being checkpointed crashes
- A rebooted node fails to come back up
- Propagation fails due to lack of disk space, network unavailability, etc.

9.5 Performance

There are functions that our system must perform that can substantially affect the performance of the application as a whole. The most significant such functions are the propagation of changes to nodes and the file scans required to generate trees reflecting the state of the filesystem. Other functions that will take a noticeable amount of time to perform include the computation of deltas and the storage of deltas and the corresponding files on disk.

The storage of deltas on disk involves traversing the in-memory data structure and writing its contents to the shared administrative area, so this operation cannot be substantially optimized. Storing the files on disk involves copying files from the local system partition to this share, and likewise cannot be substantially optimized. Propagating changes involves reading files from the share and copying them across the network to remote systems. Like the other operations, this cannot be substantially optimized; this operation is also subject to the performance limitations of the remote

share and the network. File scans involve reading files over the network or from a local system partition and generating an in-memory data structure. There is no substantial optimization of such a file scan operation.

10. Future Work

Our system is reasonably complete, but could be extended in several ways to provide a higher degree of usability or performance.

10.1 Extensible Support For Node-Specific Information

A more advanced feature might be to have our application invoke a DLL that modifies files unique to one node so that they work on another node. For example, when installed, an application may store the CPU's serial number in a file; a DLL could be developed to encapsulate this knowledge and regenerate a file for each node containing that node's serial number.

10.2 Handle File Attributes

Future versions of our system could be extended to preserve file attributes (such as "hidden" status) and security information. File attributes are available as part of a data structure maintained for each filesystem entry, but security information would need to be retrieved through the Win32 security API and stored in the delta file. The security information would need to be translated across nodes, since user and file identifiers would change (rendering the old identifiers unusable).

10.3 Optimization and Tuning

An interesting, but complex, addition to our system would be the ability to recognize renamed files and apply the name changes to nodes on the network instead of propagating the file with the new name. This would require matching internal file identifiers or other file attributes that uniquely identify a file. Another optimization would be to recognize files that are substantially the same, and only propagate enough information to change the portion of the file that did change, leaving the rest intact. This could be especially useful when propagating application updates.

11. Conclusion

Using information derived from analyzing the application installation process, a system to maintain symmetry across nodes in a Windows NT environment has been built from simple building blocks. The most difficult issues to resolve are those involving applications that are not to be available on all nodes in the network. Such applications create a multitude of special cases to be dealt with.

Although this system can be adapted to provide other services, it is best suited to fulfilling its purpose of making application installation and removal on the nodes of a

network as simple as possible. By allowing network administrators working at an arbitrary node to install applications to and remove applications from any other node on the network, our system makes system administration a less burdensome task.

12. Acknowledgments

This work would not have been possible without the advice, guidance, and patience of Robert H.B. Netzer. His contributions to the quality of this work and my personal growth as a computer scientist are immeasurable. I would also like to thank Mark D. Handy, who was able to provide valuable feedback on the design of my data structures and the algorithms that operate on them and answer my (often annoying) questions about the STL.