Automatic Deployment Set Generation

15th June 2005

1 Introduction

Software configuration management has become an increasingly difficult problem in the enterprise. As more and more business processes are computerized, the number of different configurations that need to be maintained increase in number. As the number of different software packages used by programs rise, the complexity of each individual configuration increases. And as these packages are distributed across more and more computers, the number of deployments of these configurations increase as well.

The technology for managing and understanding this increasing number of complex configurations is still immature. Fortunately, the management of a large number of interacting and complex source code files can be likened to the management of a large number of interacting and complex software configurations. As such, the related field of source code management can offer a great deal of insight into what tools are needed and how they should work.

For example, the Vesta system [1] is an advanced source code management system. One of its features is the automatic detection of build dependencies. By exporting an entire file system using an NFS interface, it can track the usage of all files used during the build process, and hence have complete knowledge of build dependencies.

A similar tool for automatically detecting the set of files needed for a certain software deployment configuration would be similarly useful in a configuration management system. One attempt at designing such a tool is examined here.

2 Use Case

During the development of a software application, a programmer may make ad hoc use of a wide variety of different software packages. Different pieces of middleware might be experimented with and discarded, or they might be kept but repeatedly reconfigured and updated. Once the application is finished and can run successfully on a programmer's machine, it might no longer be clear how to deploy the application on another machine. The specific set of packages that need to be installed on a new machine and the specific modifications needed to them might no longer be obvious. And simply replicating the exact configuration of the programmer's machine might also be infeasible because of the security risks of copying unnecessary developer files and programs

onto a server and because some of these unnecessary developer files and programs may require expensive licensing.

In these situations, it might be useful to have a tool that can automatically determine the minimum set of files, hereafter referred to as the *deployment set*, that need to be replicated from a programmer's machine onto a new machine in order to get the software application to run there. Knowing this information, one can archive up all these files into a single package, making deployment relatively easy.

This report investigates the feasibility of designing such a tool. The issues and methods encountered during the design of a prototype are documented, and the effectiveness of the prototype in handling two test cases are used to infer whether the strategies used by the tool can be scaled up to handle this general class of problem.

3 Possible Approaches

Determining the set of files needed by an application is not entirely straight-forward. Not only is it necessary to trace the file usage of a particular application, but since an application may rely on other applications and services, application dependencies and the file usage of other applications must also be tracked.

Fortunately, the source code management field explored similar issues many years ago. For example, tools have existed for Java [2] and Smalltalk for some time that can calculate the minimum set of methods and classes needed to run a particular application. By looking at a specific "main" class and tracking all possible method invocations from it and descendant invoked methods, these tools can calculate the maximally reachable graph of methods and classes. This set of code is then a good candidate for being the minimum set needed to run a particular application.

Unfortunately, extracting file and application dependency information is much more difficult that extracting code dependency information. Whereas it is well-known how to modify a compiler to output all method invocations of a program's source code, it is not obvious how to extract all file and inter-process interactions from a program's executable. It might be possible to examine strings from the executable and elsewhere in order to try to piece together possible file names, but that approach is extremely error prone. In fact, code dependency tools also tend to fail when having to deal with reflection where method invocations are implicit in the code as opposed to explicit.

Instead, it is easier to trace the file usage and application dependencies of a running program than to extract this information from a program's binary. By gathering a trace of a program's execution, one can easily observe all of its inter-process communication with other programs and all files used. Unfortunately, a trace only describes the files used during a particular execution run as opposed to describing all files that can possibly be used during any execution run, meaning that if certain functionality isn't exercised, this functionality will not be described in the trace. Practically, though, as long as a user runs a comprehensive set of tests against their program, the execution traces should exhibit reasonable coverage.

The deployment set tool uses runtime traces as opposed to analysis of binaries.

4 Gathering Deployment Set Information

It is possible to gather traces of application file usage at many different levels:

- Collecting information at the *application level* provides the most detailed information about application file usage, but doing so requires the modification of many different programs and can be quite tedious. A system based on such a design would also be unable to detect OS configuration information.
- Probing *system calls* requires only the modification of the OS (as opposed to the modification of several applications), but it can only detected a limited amount of file usage information such as whether a file has been opened for reading or writing. For applications based on Java, for example, where the Java VM may search through many jar files looking for code, system call information would not provide enough information to differentiate between jar files that are actually used and jar files that are merely searched in.
- Finally, monitoring accesses to individual *i-nodes* provides the most accurate information about which files are being used while requiring the fewest modifications to OSes and applications, but it is unable to gather any useful information about the usage patterns of those files.

The prototype deployment set tool uses system call probing to gather its configuration information. Since Linux is widely used in servers and has available source code, it was chosen as the basis for the tool. Although Linux has many systems calls related to file usage and IPC, only a small subset of system calls and internal functions were modified to log usage information: execve, fork, vfork, exit, open, close, open_exec, socket, listen, bind, accept, and connect. A small subset was chosen because modifying all systems calls would have been prohibitive and unnecessary to judging the feasibility of a prototype tool. Each log entry encoded information about accessed files, current working directory, process id, and any new or expiring process ids.

In order to make the logging of system call information easier, the tool makes use of User Mode Linux [3]. User Mode Linux is a virtual instance of Linux that can be run in user-mode on a Linux computer. It is similar to virtual machine technology such as VMWare or Xen. In order to use the tool, a user with a working configuration on a certain machine, would copy the contents of the machine hard drive onto a virtual drive. The user would then start User Mode Linux, using the kernel with the system call logging modifications, from the virtual drive. While the user interacts with their application on the virtual Linux instance, all information about processes and file usage is logged to a file on the real Linux instance. This log file is then analyzed by the tool to determine which files from the original machine need to be copied to new machines to run the same applications. The use of User Mode Linux means that a user does not need to modify their own OS to use the tool. Since the tool attempts to emulate the original environment of the user's machine in an instance of User Mode Linux as closely as possible, applications should behave identically under User Mode Linux as on the user's machine.

As noted previously, the user must "exercise" their application such that the application in such way that all desired functionality is used. So, for example, supposing a user has a web service running on their machine, they must then run the web service within the tool and make use of the web service. If the user doesn't make use of some remote administration features of the service, the tool will not be able to detect which files are needed for those administration features and might not include those files in the deployment set.

5 Tool Evaluation

In order to evaluate the effectiveness of the tool at determining the deployment set, a simple set of tests were created.

A Fedora Core 3 system, configured with Apache, Postgresql, MySql, Php, programming tools, and desktop tools, was installed on a system. The size of the installation was 3.96 GB.

Then, the effectiveness of the tool at detecting the set of files needed to run two different test sequences were evaluated:

- *ls:* The system is started, the root user logs into the system from a terminal, and this user runs ls followed by halt.
- *Linux-Apache-MySQL-PHP (LAMP):* The system is started, and the root user logs into the system from a terminal, and configures the networking on the system. Another user then uses a web browser to access a page running on the web server of the system. This web page is within a user account and makes use of php to read some data from a MySql database. Afterwards, the root user runs halt.

The prototype tool is first used to gather file and process dependency information during the running of those test sequences. Then, the tool is instructed to calculate the deployment set needed to run a specified set of processes. For the first test sequence, the tool is instructed to calculate the deployment set needed to run ls and halt. For the second test sequence, the tool is instructed to calculate the deployment set for ifconfig, httpd, and halt. The effectiveness of the tool can then be evaluated by the size of the deployment set and by whether the deployment set is viable configuration or not (a deployment set is considered viable if the set of steps from the test sequence can be repeated without errors).

Due to incompatibilities between User-Mode Linux and Fedora Core 3, the User-Mode Linux environment is configured slightly differently than the original Fedora Core 3 environment. In particular, the /lib/tls libraries are removed and SELinux enforcement is disabled.

6 Straight-Forward Algorithm

To generate baseline results, the prototype tool was initially set-up to use a simple algorithm for determining the deployment set. The algorithm simply takes all files ac-

cessed during the execution of a test sequence as being the deployments set. “All files” includes objects mapped into the filesystem like device nodes and FIFO objects, symbolic links, and destinations of symbolic links. If a directory is accessed, the directory is included in the deployment set but none of its contents. Hard links are ignored, and files from certain special directories such as /sys and /proc are not copied. The entire contents of the /dev directory is also added to the deployment set.

For the ls test sequence, the deployment set was 138 MB in size. Some services such as cups, sshd, sendmail, and httpd could not start because of missing files and directories. The missing files and directories resulted from the fact that certain system calls such as readlink, chdir, and stat64 were not being logged to capture file dependency information. It was possible to complete the test sequence using the generated deployment set, however.

For the LAMP test sequence, the deployment set was also 138 MB in size. The same set of services that were not able to start in the ls test sequence were also not able to start in this test sequence. In order to allow Apache httpd service to start, it was necessary to create an empty /var/www/html directory. After the creation of this directory, it was possible to complete the LAMP test sequence using the deployment set.

The large sizes of the resulting deployment set for even the simple ls test sequence suggest that the sets includes many unnecessary services started during the Linux boot sequence; nonetheless, the 138 MB size of the deployment sets is much smaller than the 3.96 GB size of the complete installation. Reducing the size of the deployment set further requires aggressive pruning of unnecessary executables. A better algorithm would detect which processes are not essential to the running of the service of interest and remove those executable files from the deployment set.

7 Pruning Algorithms

7.1 Dependency Model

In order to build the pruning algorithm described in the previous section, the tool needs to be able to track dependencies between processes and files. To aid in this task, the algorithm first converts the log of system calls into a graph.

Figure 1 shows a sample graph. Nodes of the graph consist of either processes or files. Edges are directed and denote a dependency relationship between two nodes. The following are the different possible dependency relationships:

- When a process forks another process, the child process is considered dependent on the parent.
- When a process execs a different program, the new program is modelled as a new process node which is dependent on the original process (although technically, the new program is the same process as the original process)
- When a process opens a file for reading, the process is considered to be dependent on that file

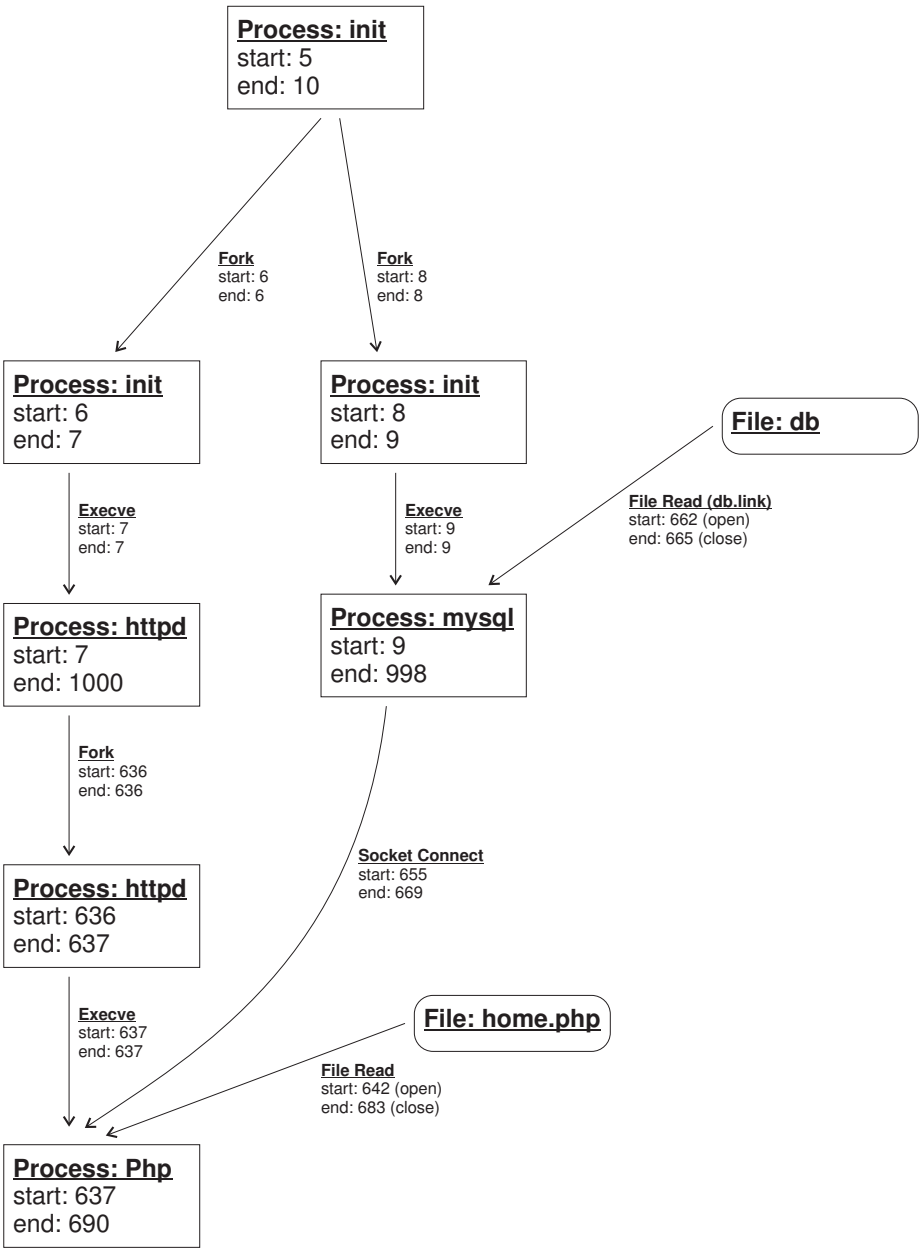

Figure 1: Sample graph of extracted dependencies

- When a process opens a file for writing, the file is considered to be dependent on the process
- When a process creates a TCP socket connection to another process, the initiator of the connection is considered to be dependent on the other process

File dependencies are annotated with the pathname used to access the file. All edges and process nodes are annotated with information about the time when they came into existence and when they were destroyed. Clock information is not recorded in the system call log file; instead, line number within the system call log is used as a substitute for time. These are the rules for deciding the start and end times of edges and process nodes:

- A process begins when it is forked or execed. It ends when it exits or when it execs a different program
- A socket connection begins when the connect system call is called and ends when one side of the connection calls close
- A file dependency begins when a file is opened and ends when a file is closed. For executable and library files, a file dependency begins and ends when the file is linked into the process
- A fork dependency begins and ends when the fork system call is made
- An execv dependency begins and ends when the exec system call is made
- When a new process node is created after a fork or execv, the new node does not inherit file or socket dependencies

Unfortunately, since not all file system calls are logged, it is not possible to create a completely accurate graph representation of file and process dependencies. In particular, although file opens and file closes are logged, system calls for duplicating file handles are not, so it is difficult to know whether a close system call will, in fact, close all references to a particular file within a process. It is possible to infer this information based on what file handles are closed during process exit, but this is not absolutely safe because other objects that are not tracked such as pipes might cause confusion. Fortunately, for this particular application, an accurate graph representation is not strictly necessary, so a good approximation of the representation is sufficient.

7.2 Pruning

Once the prototype tool has constructed a dependency graph, the user must mark some of the nodes as being important. The tool can then calculate which nodes and edges of the graph are not needed by the marked nodes. For the ls test sequence, the /bin/ls and /sbin/halt process nodes as well as all descendant nodes of the /sbin/halt process nodes were marked as important. For the LAMP test sequence, the /usr/sbin/httpd, /sbin/ifconfig, and /sbin/halt process nodes as well as all descendant nodes of the /sbin/halt process nodes were marked as important.

Initially, a Strict Dependencies algorithm was used for pruning. In this algorithm, all paths leading from marked nodes are followed, without regard to edge types, node types, edge timings, or node timings, until the full reachable subset is calculated. This subset is then selected as the deployment set. After a bit of experimentation, it was found that the /etc/mtab and /var/run/utmp files as well as the entire /dev directory subtree could not be included in the dependency calculations because those files were often read from and written to by many different processes but not actually used for data interchange between processes (e.g. many processes read and write to /dev/console but do not use the device to pass data between each other). Figure 2 shows the minimum set of processes needed by the ls test sequence as calculated using the Strict Dependencies algorithm.

Unfortunately, the deployment sets for both the ls and LAMP test sequences are non-functional. During the boot sequence, several processes are started that configure device drivers and the OS environment. Because these processes do not transfer data to or from other processes in the test sequences, the dependency graph does not capture their importance to the execution of other processes. As a result, the filesystems in these configurations do not mount correctly, and they end up being read-only.

To rectify this problem, it is necessary to mark the descendants of the /etc/rc.d/rc.sysinit and /sbin/runlevel processes as well as all descendants of processes with no recorded binary as being important in the Strict Dependencies algorithm.

After this change the ls test sequence is able to start and complete successfully. The LAMP test sequence is not able to start MySQL successfully. This failure results from the fact that MySQL invokes the /usr/bin/tee program, but doesn't actually use the output of the program. With the strict dependencies algorithm, if the output of a program isn't used, then the program will not be included in the deployment set. This causes problems because when MySQL attempts to exec /usr/bin/tee, the exec will fail, and MySQL will terminate because of this error even though /usr/bin/tee is not essential to the running of MySQL.

One possible solution is to include all invoked child processes in the deployment set regardless of whether or not they are deemed essential. Unfortunately, this results in all processes being included in the deployment set. However, because certain processes do not terminate if an attempt to execute a child process fails, the child processes of those processes can be excluded from the deployment set. In particular, the /etc/rc.d/rc start-up process satisfies these conditions. The /etc/rc.d/rc process invokes all services and daemons at start-up, so pruning children of this process essentially removes unnecessary Linux services. Unfortunately, more fine-grained pruning of unnecessary programs is not possible under such a scheme.

Other possible solutions include trying to use timing information in calculating process dependencies (so that child processes can be removed from the deployment set if the original process can fail during the execute without causing dependency problems elsewhere) or to create "fake" stub binaries that simply exit immediately after being executed.

After the prototype tool was modified to include all child processes, the deployment sets calculated for the two test sequences were, for the most part, functional. The LAMP test sequence needed only minor adjustments:

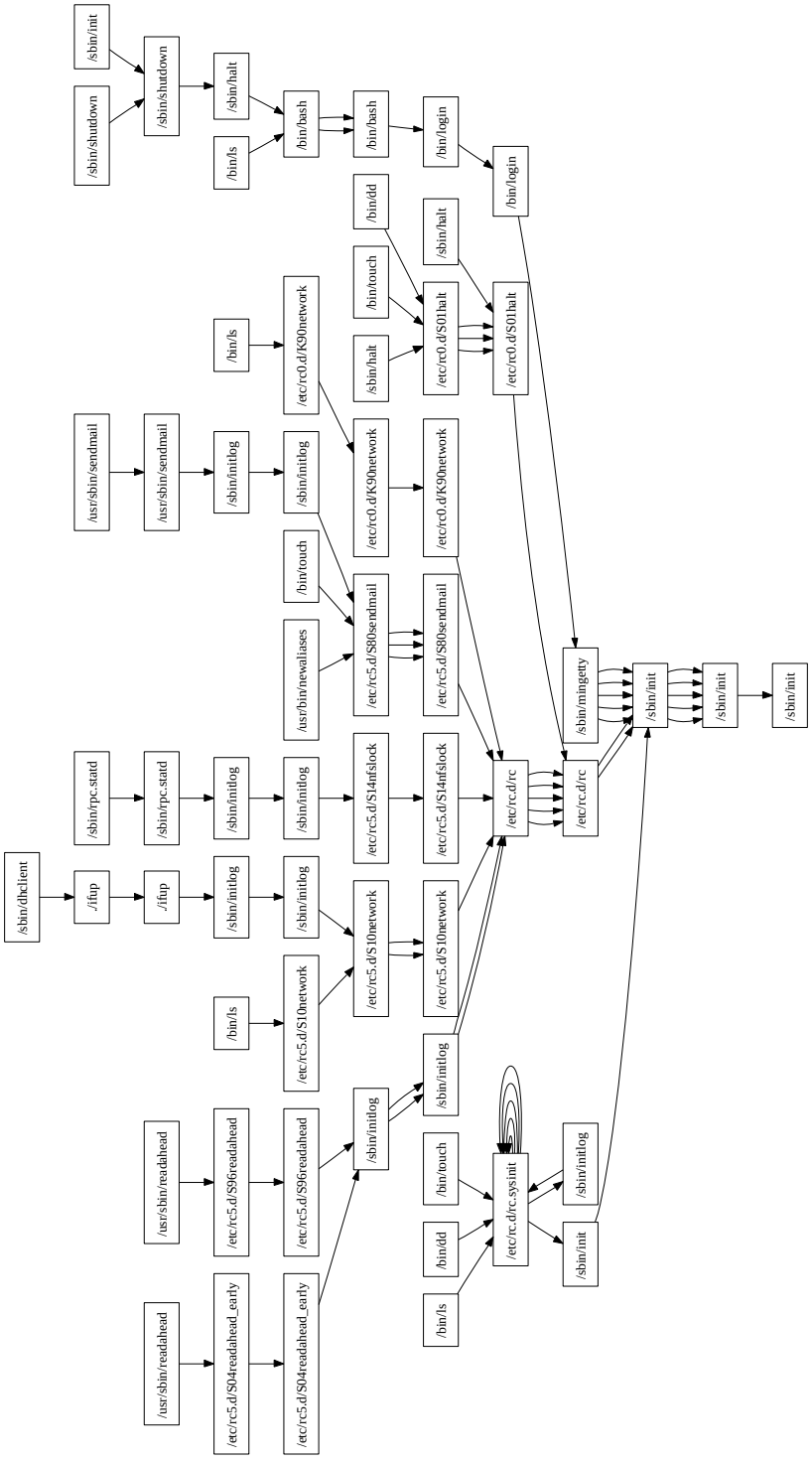

Figure 2: Sample graph of extracted process dependencies

- As before, it was necessary to create a /var/www/html directory
- Two files were written to by Apache but never read from: /etc/httpd/logs/error_log and /etc/httpd/run/httpd.pid. Since their contents were not necessary to the running of Apache, the files were not included in the deployment sets. Unfortunately, the directories holding these files were not included in the deployment sets either, so Apache could not create new copies of those files.

The prototype tool can be easily modified to automatically detect and correct the issues noted.

In both test sequences, the tool was able to detect that Postgres was not needed. The tool also did not include many shutdown scripts needed to properly shutdown services such as httpd and mysql during system shutdown. This is worrisome, but did not seem to cause problems with the test sequences.

Table 1 summarizes the behaviours of the different pruning algorithms. All of the pruning algorithms were able to create a smaller deployment set for the ls test sequence than for the LAMP test sequence. The inclusion of child processes in a deployment set only increased the size of the sets by 3 MB, which suggests that simply pruning out unnecessary services invoked at start-up is sufficient for removing the majority of unnecessary files.

Test Sequence	Size	Functional?
Strict Dependencies		
ls	86 MB	No
LAMP	99 MB	No
With System Files		
ls	94 MB	Yes
LAMP	107 MB	No
With Inclusion of Unnecessary Child Processes		
ls	97 MB	Yes
LAMP	110 MB	Yes*
All Accessed Files		
ls	138 MB	Yes
LAMP	138 MB	Yes*
Complete Installation		
ls	3.96 GB	Yes
LAMP	3.96 GB	Yes

* with minor tweaks to the deployment sets

Table 1: Pruning Results

8 Related Work

Although there are other papers dealing with deployment of services in data centres [4], these papers focus on the actual act of copying files to data centre computers and the configuring of services. They inherently assume that the developers know which set of files are necessary for a service.

Like the prototype tool, there are other tools for better understanding the Linux boot process such as the BootChart project [5]. The BootChart project profiles the time spent in various processes during system start-up in an attempt to identify possible bottlenecks. Unlike the prototype tool, this project focuses more on boot optimization than on understanding dependencies and pruning unnecessary services.

The technique of tracing of system calls used by the prototype tool to gain a greater understanding of system behaviour is well-known. One common use of system call tracing is for anomaly detection in intrusion detection systems [6]. By looking for sequences of system calls that differ from the normal behaviour of applications, one can identify possible attacks by an intruder.

Also, the examination of IPC system calls to understand process dependencies has been examined before in the context of trying to improve process scheduling [7]. By tracking the IPC data flows between various processes at a very fine-grained level, it's possible to make more intelligent scheduling decisions that prevent high priority processes from having to wait for lower priority processes, even during very complex interactions.

9 Conclusion

The prototype tool described in this report is very effective at generating deployment sets and removing unnecessary services from those deployment sets. The tool is not entirely automatic though, and requires user intervention and interaction to be truly effective. Because the tool operates at the system call level, it is not able to understand subtle process interactions, such as devices or files that are both read and written to by different processes but not used to exchange data or such as child processes that do not serve any useful purpose to a parent process, but which might cause a failure of the parent process if they are excluded. In the most complex cases, the tool can simply suggest using the entire set of accessed files as the deployment set, but much smaller deployment sets are possible if the user help the tool in its analysis.

The amount of user expertise needed to use the tool is likely less than the amount of expertise needed to manually create a deployment set. With the tool, the user merely needs to run the tool, identify unnecessary or missing processes, find the errors in the dependency graph, and iterate until a suitable deployment set is created. When manually creating a deployment set, the user must deeply understand the interactions of the service they want to deploy in order to recall its dependencies, find the locations of all the files needed by the service and its dependencies, gather all of these files into a deployment set, and iterate if the user has forgotten important details.

In the future, the tool should be revamped to trace more system calls and to provide more facilities for identifying and correcting deployment set errors. The tool should

also be ported to other systems such as Windows or Java. The tool might also benefit from being extended to deal with configuration issues such as identifying files that need to be changed for a deployment set to run on a different machine, merging different deployment sets, or automatically adjusting a deployment set to handle differences between the deployment machine and the original machine.

References

[1] Allan Heydon, Roy Levin, Timothy Mann, and Yuan Yu. The vesta approach to software configuration management. Technical Report 168, Compaq Systems Research Center, March 2001.

[2] Frank Tip, Chris Laffra, Peter F. Sweeney, and David Streeter. Practical experience with an application extractor for java. In *OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 292–305. ACM Press, 1999.

[3] Jeff Dike. The user-mode linux kernel home page. *http://user-mode-linux.sourceforge.net/*.

[4] Vanish Talwar, Dejan Milojicic, Qinyi Wu, Calton Pu, Wenchang Yan, and Gueyoung Jung. Approaches for service deployment. *IEEE Internet Computing*, 9(2):70–80, 2005.

[5] Ziga Mahkovec. Bootchart: Boot process performance visualization. *http://www.bootchart.org*.

[6] Stephanie Forrest, Steven A. Hofmeyr, Anil Somayaji, and Thomas A. Longstaff. A sense of self for Unix processes. In *Proceedinges of the 1996 IEEE Symposium on Security and Privacy*, pages 120–128. IEEE Computer Society Press, 1996.

[7] Haoqiang Zheng and Jason Nieh. Swap: A scheduler with automatic process dependency detection. In *Proceedings of the First USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI-2004)*, San Francisco, CA, March 29–31 2004. Usenix Association.