The following paper was originally published
in the Proceedings of the
Tenth USENIX System Administration Conference (LISA X)
Chicago, IL, USA, Sept. 29 - Oct. 4, 1996

Abstract Yourself With Modules

John L. Furlani, Sun Microsystems, Inc.
Peter W. Osel, Siemens Components, Inc.

For more information about USENIX Association contact:

1. Phone: 510 528-8649
2. FAX: 510 548-5738
3. Email: office@usenix.org
4. WWW URL: http://www.usenix.org

Abstract Yourself With Modules

John L. Furlani – Sun Microsystems, Inc.
Peter W. Osel – Siemens Components, Inc.

ABSTRACT

Modules abstracts the activation of applications from the details of their installation. It provides a uniform interface for selecting applications and for applying the necessary changes to the environment.

Five years ago, the first paper on Modules was published [1]. Since then, Modules has been written in C, uses Tcl [2] as its extension language, has seen acceptance and use at a rich variety of sites and has acquired several features for supporting the management of hundreds of software packages across large and diverse intranets. With Modules' proven combination of features and flexibility, we believe it has the potential to become the preferred standard for software management and activation.

In this paper, we compare the Modules package with several systems that have appeared in the years since its introduction. We also present some real-world examples of how the Modules package is being applied. This paper covers some of the new features in the current implementation. Finally, we discuss how the Modules concept can be applied elsewhere, including the problem of loading and installing on-demand applets and applications.

Introduction

The problems surrounding software installation and software distribution for heterogeneous distributed networks continue to be an area of great interest to systems administrators and software developers alike. Because of this level of interest in recent years, a plethora of solutions are available for solving many of the problems with installing and distributing software for large networks. Beyond software installation and distribution is software activation and software management. We define *software activation* as the problems surrounding how a user learns about installed software as well as how the software, once installed, is enabled by the user. We define *software management* as the set of problems around maintaining an ever-growing, ever-revising array of software packages. Modules solves the problem of software activation and software management for the user and the systems administrator. In essence, Modules is about abstracting the user from traditional dependencies on the location, the architecture and the issues surrounding version migration for installed and distributed software packages.

In this paper, we present a few interesting case studies which exemplify how the Modules package can be and has been applied. We follow up the case studies with a section comparing Modules as it is implemented today with other packages and approaches that attempt to solve related problems. Once compared with other solutions, we describe in more detail the currently available implementation of the Modules concept done in Tcl and C. The final portion of the paper is dedicated to new features in future versions of the Modules package, to ideas for new implementations of the Modules concept and to other areas where the concepts behind Modules can and should be applied – specifically applet deployment.

It is important at this point to separate the Modules package from the Modules concept. The *concept* behind Modules is grounded in applying object oriented techniques such as methods and data abstraction to the problems of software management and activation. The Modules *package* is a particular implementation of the Modules concept for UNIX systems using Tcl and C.

Modules Basics

Modules abstracts the activation of applications from the details of local software installation and distribution. It frees the user from clumsy manipulation of environment variables by providing high level methods for software activation.

If "object oriented" wasn't an overloaded buzzword, we would be even more inclined to advertise Modules as the "object oriented" solution to software management and activation. In object oriented parlance, installed software packages are the objects. Modulefiles or some other application-specific database information contains the implementations for a set of methods, e.g., activate, deactivate, show information. The program behind the *module*(1) command implements a command-line user interface for triggering the execution of methods on the objects (deactivate *software1*, activate *software2*). The *module*(1) user interface does not differ between platforms or shells.

Dependencies and conflicts between applications and versions of an application can be configured into the modulefiles. Modules also affords the user query abilities which include listing the available software packages, information about a particular package and package-specific help.

Case Studies

The Modules package is helping numerous sites of different sizes, ranging from a single workstation to large intranets with several hundreds or thousands of workstations, to manage their software. This section describes four deployments of the Modules package and exemplifies its flexibility.

Siemens AG

At the Semiconductor Division of Siemens AG the Modules package is used for managing more than 250 software packages on more than 800 Sun and HP workstations worldwide. The software packages include commercial, freely available and proprietary tools. It is not unusual to have many (sometimes up to a dozen) versions of a software package installed and in use at the same time. For example, each version of their CAD system uses a different set of software packages. The software and configuration information, including modulefiles, are mirrored to development sites in Germany (München and Düsseldorf), Austria (Villach), The United States (Cupertino, CA), and Singapore using OpenDist [9]. Within their environment *module*(1) is called more than six thousand times a day. They also ship parts of their CAD system to more than twenty external design contractors. At the contractors' sites the Modules package integrates smoothly into a multitude of different existing system and network setups.

All sites use the same modulefiles. Differences in the systems' configuration are handled by common library functions returning domain-specific and system-specific configuration information. Thus, all system configuration can be done by a central library, thereby decreasing the maintenance effort while improving documentation of differences between their computing sites.

Every software package activation is logged using the usage tracing feature of the Modules package. Knowing the usage of software on the systems at each site helps them efficiently phase out old or unused software. Usage tracing also enables them to inform affected users of a particular software package if a mission-critical bug is found. This is more efficient and effective than posting to a general information news system or broadcasting to all users. They have found that users bombarded by e-mail or repeated news about software packages they don't use end up ignoring all of the informational messages. They claim the timely and effective distribution of information to those affected saves thousands of dollars.

Sun Microsystems, Inc.

The Modules package originates from Furlani's work as a systems administrator for the North Carolina Development Center. Its use unified software management and activation for about seventy users on over one hundred workstations. Novice users found it simpler to learn about and activate new applications. Advanced users quickly embraced its wide range of capabilities and its immense flexibility.

Often, software developers must test against different releases of many different source code trees. The Modules package permits a team-organized mechanism for simple switching between daily builds, weekly builds, earlier releases, personal clones and other developer's clones.

Today, the Modules package is heavily utilized by the Solaris XIL Image and Video Library development team. The team shares a common directory of modulefiles specifying the location of common directories like daily, weekly and release builds. Each developer has his or her own directory of modulefiles augmenting or overriding the commonly available modulefiles. Often a developer isolates a bug or adds a new feature in his or her development tree that other developers want to test or try prior to integrating the changes into a master source tree. Switching to a new development tree occurs with a single *module*(1) command referencing the target developer's modulefile.

Also, some groups inside of Sun use the Modules package for the more commonly stated software activation and management features.

Cray Research, Inc.

The Modules package is released with Cray's Programming Environment 2.0 [10] product. Cray supplies modulefiles with the product for managing the dependencies between the various pieces within the Programming Environment as well as the cross-compiling capabilities of their compilers. Using the Modules package also allows local systems administrators an easy method for managing more than one version of the product at the same time. Cray also provides some tools for customers to create their own modulefiles as well as a specific modulefile for each version of the product.

Auburn University

The College of Engineering at Auburn University uses the Modules package to manage a large number of applications for many users. They use the Modules package as the foundation for a self-developed menu-driven application initialization and activation package called user-setup [4]. Since the introduction of user-setup Version 2 in May 1992, over 5,000 users have benefited from the easy menu-driven setup and manipulation of their environment afforded by user-setup.

The systems administrators found an immediate and dramatic reduction in their help-desk's work-load after the introduction of user-setup Version 2. Some unexpected side-effects of introducing this Modules-based solution include simplified instructions for those describing how to access software applications, a reduction in requests to install new software packages, users easily locating and using a wider variety of software on their own and enabling the use of UNIX systems to non-UNIX users.

Related Tools

We divide the software distribution cycle into five distinct steps as follows:

- *Development* encompasses the creation of the bits comprising an application.
- *Packaging* focuses on the problem of getting the bits onto a distribution media (be that magnetic, optical or a network).
- *Installation* (the inverse of the packaging problem) is getting the bits off of the distribution media onto a local media in an organized fashion.
- *Integration* is making the bits available on the user's system.
- *Activation* includes how users learn about the availability of the application as well as how they enable the new application once it is integrated on their system.

Each stage of the software distribution cycle requires solving a different set of problems. Solutions and tools often cover more than a single step. In fact, they often imply or depend upon attributes of the neighboring steps.

Tools for software development, software packaging and software installation are beyond the scope of this paper. Although, it is important to note that Modules operates independently of the solutions used for packaging and installation. Modules aims to solve the final two steps of this software distribution cycle. The next two sub-sections focus on how Modules can be used to integrate and activate software in a distributed environment.

Integration

Conceptually, files on a user's system start at the root directory and live locally or remotely. Integration involves integrating the application into the filesystem hierarchy available to each user.

There are a number of solutions which provide the mechanics for organizing and distributing the application's bits to a large number of machines. Modules works independently of these software installation and integration solutions. But we have found that Modules better solves the distributed software management problem if Modules' object oriented characteristics are taken into consideration when choosing or utilizing an integration scheme.

Typically, solutions to the software integration problem provide policies dictating how software packages are to be integrated onto the system. These solutions often define a package's location within the filesystem hierarchy whereas solutions to the software installation problem often define a package's internal directory structure. Often, both of these solutions include tools to help systems administrators install, distribute and remove packages according to a set of policies.

Policies may account for the efficient storage and distribution of software packages for a highly distributed heterogeneous network of computers or for a single computer.

Many take advantage of transparent remote network filesystem access such as NFS [26] and AFS [13]. Some use tools like *rdist*(1) [8] or the OpenDist package to duplicate software within a network.

Our experience with the Modules package indicates integrating software in discrete packages works best. Fortunately, it is becoming common practice that integrating software into a distributed network is done as separate packages. The clean separation of software packages and versions help to manage an ever-growing and fast-changing ocean of software.

Installation of software as separate packages has several advantages. Installation and removal of software becomes cleaner and errors or name conflicts become less probable. Multiple versions can be installed in parallel and an be selected individually. Software discovery becomes easier if a standard directory is used to install all packages. However software activation becomes more difficult because a user has to change the environment prior to starting an application.

Providing merged access directories (e.g., bin, lib, man) by linking all files from separate packages is one common approach to solve this problem. Users include these merged access directories in their path and have access to all installed software. However, if this merged directory is configured centrally, the user looses control over name conflict resolution or version selection. If he or she wants to use a different version, he or she still has to know where this software is installed and change his or her environment accordingly.

Also, a merged directory for commands quickly becomes huge. This increases the start-up time for shells with hash tables (like C shells) and the search time for commands in shells without hash tables. Symbolic links carry a performance penalty as well and require tedious maintenance.

Systems providing merged access directories to separately installed software packages include the "Corporate Software Bank" [24], LUDE [23], Depot (CMU) [15], its extension "Local Disk Depot" [17], and Xhier [22].

Xhier uses a user-owned merged access directory such that users can select which software packages are linked. Users can select packages and versions.

The Depot (NIST) [21], Depot-Lite [16], opt_depot [18] install software as separate packages without specifying merged access directories. They rely on the user or other tools to provide for software activation. In fact, Depot-Lite uses the Modules package.

Also see John P. Rouillard and Richard B. Martin, "Depot-Lite: A Mechanism for Managing Software" for a discussion of several software integration solutions (such as a comparison of Depot-Lite with NIST Depot, CMU, CMU Ext, Ericsson, and Xhier).

The System V software management utilities (*pkgadd*(1), *pkginfo*(1), *pkgmk*(1), ...) [12] and the emerging POSIX standard for software administration [25] address packaging and installation but not distributed integration of software.

No policies of how and where the software is to be integrated into the system are defined. Though, they recommend it should be possible to install software at any point in the filesystem hierarchy.

The main focus of LUDE is software integration. But, it also touches on installation and packaging of software. Software packages can be retrieved from LUDE ftp servers or tapes, and can be integrated into your installation by LUDE commands.

Activation

Software activation should hide integration details from the user. Users should be able to find out what software is available, what the software does and should have the ability to enable the software without knowing the details of its integration on the system. Of course, the user's favorite shell should be supported.

There are several tools purporting to solve the software activation problem. However, most of them solve only part of the problem. None of them provides the user with as flexible and as rich a feature set as the Modules package does. Most of the tools do not support software configuration management features like querying for available packages, dependency declarations, etc. See Table 1 for a comparison of features.

All of the solutions support initial setup of the user's environment during login by setting or altering environment variables. BNR [6], login-shared [5], and the Modules package use exactly one file per application to store the description of environment changes. login-shared stores the files with the applications, while BNR uses a configuration file to list their locations. The Modules package supports single or multiple arbitrarily nested directory structures to store configuration files.

Here we look at each solution independently in more detail.

Features	Modules Package	BNR	Ini	Login-Shared	Soft
Supported shells	sh, csh perl, emacs	sh, csh	sh, csh	csh	sh, csh
Freely Available	YES	NO	YES	YES	YES
On-the-fly Activate/Deactivate	YES	NO	YES	NO	NO
On-the-fly Reload/Refresh	YES	NO	YES	NO	YES
On-the-fly Exchange/Swap	YES	NO	NO	NO	NO
Define Alias/Shell Functions	YES	NO	NO	YES	NO
Set Local Shell Variables	NO	YES	NO	YES	NO
X Resource Manipulation	YES	NO	NO	NO	NO
Package Dependency Control	YES	NO	NO	NO	NO
Standard Configuration Language	YES	NO	NO	NO	NO
Hierarchical Package Naming	YES	NO	NO	NO	NO
Query Available and Active Packages	YES	NO	YES	NO	NO
Query Information About Packages	YES	NO	NO	NO	NO
Inherent Versioning Support	YES	NO	LIMITED	NO	NO
Symbolic Version Management	YES	NO	NO	NO	NO
Usage Tracing	YES	NO	NO	NO	NO

Table 1: Features of Software Activation Solutions

ini

ini [20] is for loading, unloading packages on-the-fly, as well as for querying available and loaded packages. Bourne and C shells are supported. *ini* uses a single directory to store configuration files describing environment changes needed to activate an application. Configuration information can be stored in a single file or on a per package basis. There is no hierarchical grouping of configuration files. The configuration files do not use a standard language. You can select to make packages available based on operating system, machine architecture or hostname. *ini* is implemented in *perl*(1) [11]. It is available free of charge for non-commercial use.

envv

envv [7] manipulates environment and local variables in a shell-independent manner. It supports Bourne and C shells. You can set and alter environment variables. When adding a component to a path list, you can define the position of the new component. You can also move components within existing path lists. *envv* has been used to write start-up scripts for applications that can be sourced by both Bourne and C shells. Once an application is activated by sourcing its start-up script, you cannot deactivate it unless you write a separate deactivation script that reverses the effects of the start-up script. *envv* is available free of charge.

Soft

Soft [3] uses a configuration file listing the packages a user plans to activate. For each entry in the user's configuration file, a database lists the necessary directories and variables to activate the application. Macros can band applications or other macros together. Shell script caches for both Bourne and C shells are created. These are sourced during login to construct the user's environment. Users modify their environment by editing the configuration file, executing an update tool, and finally sourcing the caches. There is no on-the-fly activation or deactivation of individual packages. *Soft* does not support querying for available packages. It uses its own language for its selection file. It is available free of charge.

login-shared

login-shared provides mechanisms for rapidly initializing an user's environment during login. C shell scripts for activating applications are kept with each software package (there is not a central repository). Applications can be activated interactively. A cache is created asynchronously for applications activated in the user's shell start-up file. The cache ensures during login that each environment variable is only set once. There are no commands to deactivate a package on-the-fly. A directory containing links for every command from every available package helps users locate packages. *login-shared* encourages users to activate many packages at login time. Because of this effect, a variant of the C shell was created for supporting a 4k-character search path. With the Modules package's convenient on-the-fly activation and deactivation, the user's environment can remain small and thus requires no changes to system programs. *login-shared* only supports the C shells. It is available free of charge.

BNR Standard Login

BNR Standard Login supports the setup of the user's initial environment during login for Bourne and C shells. It uses configuration files (tables) to define how the environment is modified when activating an application. Environment and local variables can be set, unset or altered. Programs can be executed and the user can be queried to correctly set the TERM variable. Users can exclude applications from initialization. Also the *BNR Standard Login* supports delayed application configuration in order to make initial login faster. It does not support on-the-fly package activation and deactivation or software package discovery. It is not publicly available.

Our Modules Implementation

Initializing the Modules package and the *module*(1) command is accomplished by sourcing a shell-specific script into the shell. The script either creates an alias or a function in the user's shell which becomes the *module*(1) command. When invoked, the alias or function instructs the shell to evaluate the output of a program. The program is called *modulecmd*(1) and converts the user requests into shell directives.

Modulefiles are written shell-independently in Tcl and are interpreted by the *modulecmd*(1) program. Modulefiles can be loaded, unloaded or switched on-the-fly. Each modulefile describes the necessary changes to a user's environment in order to activate an application.

Typically, a modulefile is a small amount of code that sets or alters a few key shell environment variables such as PATH, MANPATH, etc. Using a rich programming language like Tcl allows for arbitrarily complex modulefiles that resolve issues like complex application dependencies, resource acquisition and application access and use policies.

From the user's perspective, changing the environment for one type of shell is exactly the same as changing the environment for another type of shell. One set of information takes care of every shell type.

Often, a pool of system modulefiles are shared by many users. The Modules package enables users to maintain their own collection of modulefiles that supplement or replace shared modulefiles. A search path for modulefiles controls locating modulefiles and can be dynamically manipulated.

It has become common practice to have one modulefile per revision of a software package. These modulefiles are stored in a directory named after the software package name. Referencing the directory structure selects a default version, yet a specific version may be specified as well. These directories can be kept in single or multiple modulefile repositories. Arbitrarily nested directory structures can be utilized for storing modulefiles.

The Modules package source code is freely available. It is written in ANSI C and can be built on any modern UNIX machine with an ANSI C compiler (like the GNU C compiler).

Standard Features

Our Modules package supports the following features:

- Set or alter environment variables (pre-pending or appending path components)
- Environment variable optimization
- Alias definition (mapped to shell aliases or functions where appropriate)
- X-Resource manipulation
- A variety of shells and languages, e.g., Bourne shell, C shell, perl, and emacs
- Activate (load) / deactivate (unload) one or several packages in a single operation
- Unload all currently loaded packages (purge)
- Exchange (swap) packages or versions
- Refresh (reload) all currently loaded packages
- Display effects of loading a package on the environment
- Query available or loaded packages
- Per-package help and information
- Centralized and distributed repositories
- Alter modulefile repository search path
- Hierarchical application categorization
- Dependency declaration (conflicts and pre-requisites)
- Tcl as activation specification language
- Modules-specific Tcl functions to manipulate environment variables, aliases, and X-Resources in a shell independent manner to query the current operation mode (load, unload, info) and to query system information (machine name, operating system release, ...)

The Modules package can be used nearly everywhere for application activation. It is not limited to interactive use. Use it in your scripts (*sh*(1), *csh*(1), *perl*(1), or *emacs*(1)), in X11 startup scripts, X11 menu files, etc.

As an aside, we have decided not to implement environment caches for setup of the user's environment during login. The Modules package optimizes environment setting when activating multiple software packages with a single *module*(1) command.

Key New Features

Last year we began a major revision of the Modules package to add features that help the largest installations manage software. This revision is version 3.0 of the Modules package and is backward compatible with earlier releases.

Active Dependency Resolution

Previously, the loading or unloading of a modulefile is aborted if a conflict or a dependency is not met.

With Active Dependency Resolution, modulefiles can force modulefiles to be loaded and unloaded such that all conflicts and dependencies are resolved. This enables the activation of complete software systems containing several packages that are considered a set. Thus guaranteeing all software packages are loaded with tested and compatible versions by automatically unloading any conflicting packages and loading any missing packages.

Circular and conflicting dependencies are detected. In these cases, the user is informed of the error and the modulefiles are not loaded. Modulefile writers are responsible for ensuring these cases don't occur when specifying dependencies. Much like header file creation in C or C++, it is possible to create modulefiles with circular and invalid dependencies.

Symbolic Versions

A symbolic version can be assigned to a specific revision of a software package. The symbolic version turns a meaningless revision like 4.2.17.8 into a descriptive name like "beta", "current" or "old". For systems with multiple software packages assigning a symbolic version to the current revision reduces the maintenance effort for configuring software systems with many version of each package.

The user can easily spot which revision is released, which revision is beta, which revisions are old and which revision belongs to a specific system. The current assignment of a symbolic version to each software revision can be listed for future reference in case the user wants to exactly reproduce an environment after the symbolic names change.

Users have control of the revision selection as well as the assignment of symbolic versions to software revisions.

Autoload

Tcl provides for the autoloading of functions. If a function is called but is not defined in the current Tcl script, a list of directories is searched for Tcl libraries containing the function.

Autoloading eases the creation of libraries containing common functions. Modulefiles shrink since code is not duplicated leading to an overall reduction in maintenance costs. Displaying standard information for packages and platform detection are examples of common functions.

Apropos and Whatis

Similar to the UNIX *apropos*(1) and *whatis*(1) commands for manual pages, a one line description is printed for every available modulefile matching the given argument. With the number of packages installed at some sites reaching staggering heights, it is no longer useful to retrieve information for every package one at a time. The new commands make it convenient to browse all of the installed software at large installations.

Logging and Usage Tracing

A network-wide logging of *module*(1) invocations using *syslog*(3) enables systems administrators to collect statistics on the usage of individual software packages. Knowing when a software package is no longer in use helps make the transition from old to new version of a software package smooth. It also helps track users of a software package for notification or interdepartmental charges.

Miscellaneous

- Oft-used package names or combinations can be abbreviated by defining a modulefile alias. For example, you can map OpenWindows/3.0 to ow3 to save keystrokes.
- Error messages are generated based on user-selectable levels. This helps novice users receive corrective information and keeps experts from complaining about verbosity.
- Configuration variables control enabling many of the new features without recompiling source code. For example, error and information messages can be configured to be printed to stdout, stderr, appended to files or passed to *syslog*(3).
- A new test suite for the Modules package provides regression testing of most features. This helps maintain quality levels through the development of new features.

Experiences

There are some not so obvious issues that you might experience when introducing Modules. The ones we present here are culled from our own experiences and those expressed by other systems administrators using the Modules package.

“Continuum Breakdown”

The Modules package does not guarantee that the user’s environment is restored to a previous state when deactivating modulefiles. Modulefiles may be very complex scripts that make it difficult or infeasible to exactly reverse their effect without storing the environment prior to their activation or storing the exact environment changes at each step. Altering the sequence of loading and unloading modulefiles adds yet more complexity.

In practice, the absence of an environment continuum has caused few or no problems for users.

“It’s a Small World”

Our studies have found that most modulefiles are very small. Out of 250 different modulefiles, 75% contain less than 20 lines and only eight have between 100 and 150 command lines.

“You Win Some, You Lose Some”

A good software activation solution makes it convenient to install more software while software installation and maintenance is easier. In most cases, a user’s PATH environment variable becomes shorter too. However, some users tend to activate every software package they might ever use in their shell’s startup file. The environment is a limited resource. Especially the length of the PATH environment variable in some shells (notably the C shell). This limits the number of packages that can be active at the same time.

To avoid the limitations of the PATH environment variable, aliases that activate a software package using the *module*(1) command before calling the application can be defined. The same method can be applied for package bundles that load several packages with a single command. This is a form of autoloading that can reduce the number of packages that users need to load. It can also provide many of the features afforded by maintaining single directory with every command linked to its location without many of the pitfalls.

“Coins Have Two Sides”

Being able to efficiently manage large installations of numerous software packages in separate locations makes it convenient to provide several versions of the same software package. This does help users migrate to newer versions. They can switch back to an old version with a single *module*(1) command if the new version does not satisfy their needs (i.e., has new bugs). Although, this capability tends to make version transitions take more time.

Given this, there is less pressure for user’s to migrate to new versions of the software by a given date. Some users stick with old versions (knowing how to work around bugs) so increased administrative pressure may need to be applied in order to get them to switch. However, we believe the flexibility gained far outweighs this side-effect.

“Start a New Shell, Do Not Pass...”

Though environment variables are inherited when starting a new shell, local variables and aliases are not. To work around this limitation, we added the update sub-command which resets the environment and reloads all currently active modulefiles.

An alternative solution we are pursuing is to have the Modules package reload those modulefiles with alias definitions when being initialized in a new shell.

Performance Considerations

There is a negligible performance penalty incurred at login when using the Modules package. On a SPARCstation 10/51 running Solaris 2.4 and version 3.0 of the Modules package, the average elapsed time to activate seven popular software packages (X11/R6, gnu, pbmplus, mtools, TeX, pgp, and Adobe’s

Acrobat) is 0.6 seconds. Activating 21 software packages takes an average of 1.5 seconds. All of the software packages are activated with a single *module*(1) load command. We use the timing facilities of *tcsh*(1). The times include any overhead from the usage trace feature.

Some users have experienced a new performance gain because not all applications must be activated during login. The dynamic activation features of the Modules package make it easy to activate less frequently used packages as they're needed.

We observe the performance penalties incurred by poorly configured or inattentive machines and networks are much higher than the overhead caused by using the Modules package.

"Think about the future..."

The Modules package based on Tcl is just one example of how the Modules concept can be applied to software management and activation. In this section, we venture into ideas which border on plans for future implementations of the Modules concept. We finish this section with a discussion of how Modules might be applied to the problem of configuring dynamically loadable Java [14] applications.

A Distributed Modules Server

The current implementation of Modules requires a separate filesystem distribution mechanism that permits all users of the Modules package to gain access to a common database of modulefiles. As the number of machines and the distance between the machines grows, keeping the database up-to-date becomes more difficult. We have begun preliminary work on the design and the development of a distributed Modules server. Such a server would assist with the problems of updating and distributing modulefiles to a wide range of machines and domains.

Software Discovery

Beyond the software discovery mechanisms available in the Modules package, we see a Modules server permitting collaboration between different administrative domains within a company. Individual domains run their own server which communicates with other servers within a company. Queries for available software within one domain can be modified to extend to other cooperating domains. Through NFS or some other file access method, an application in one domain may be discovered and made available to the user. Note that software discovery and activation in this manner is much like a selecting an HTML link on a Web page.

Software Information

We expect the Modules server to act as a software information and documentation tool like csdsdb [19]. Users should be able to query for information about each software package like:

- Maintainer's contact address
- Licensing information
- Potential source code availability
- Revision history
- Record of applied patches
- Planned upgrade information

Activating or retrieving information about the software can use the same user interface.

Programmers Interface

We have already seen the development of a wide array of tools that use and augment the Modules package. With the expanded capabilities of a Modules server, we see such tools becoming more prevalent. Providing an application programmer interface to the query capabilities, environment manipulation mechanisms and an ability to programmatically setup a new program's environment via the Modules server is an important goal.

For example, building Modules interaction into a UNIX shell reduces the need to form a traditional filesystem-based search path. When the shell is initialized, the Modules server can download application names into the shell's hash table directly without having to mount and search each filesystem. Upon invocation of any of these names (or aliases), a wide variety of environment preparation can take place. Furthermore, the shell can dynamically receive new program names and information from the Modules server as they become available or change.

User Profiles and Release Management

Based upon a user's organization or a user's profile, the list of available software and information can be customized. For example, certain applications (oh, say *salarytool*(1)) may be available to managers in an organization but not to every employee.

Profiles can also be used to remove clutter. Some users may not be interested in knowing about the wide assortment of CAD software available on the network. Being able to configure this software out of the viewable list of software assists novice users who may be overwhelmed by the large amount of software available on the network at a typical large company.

Often, applications are not available on all the hardware platforms within a company. The server can flag software packages that are unavailable on the client's hardware platform or can make them appear unavailable to the client.

Hiding software packages that are not-yet-released is another variation. A list of users, e.g., beta-testers, will see these software packages listed while other users will not be tempted by potentially unstable software.

Performance and Complexity

We expect the Modules server to increase the performance of processing modulefiles because commonly used modulefiles can be cached and potentially pre-compiled for commonly used shells. Redundant servers can ensure availability of the Modules service.

Dependencies between modulefiles can be stored in the servers' database to discover indirect dependencies or conflicts. Dependency and conflict matrices can be provided as well, e.g., to assist in discovering conflicting dependencies.

Using DNS as Modules Server

Another path we are considering is using the existing Internet Domain Name System (DNS) protocol and query mechanisms to provide some of the aforementioned Modules server capabilities. DNS permits storing arbitrary amounts of text and other information for a given domain name. We are investigating the potential of using a reserved domain name within an intranet to provide the existing *module*(1) capabilities. Special domain names can specify query requests for available modulefiles. Once the list has been received, the client can query the DNS server for more information about specific packages.

Due to the static nature of DNS records, a Modules server may still be necessary for some of the more advanced query capabilities. Although, it may be possible to use DNS to help locate the appropriate Modules server for a given application or domain.

"The Taming of Java"

Much of Modules is about preparing the system's environment prior to the execution of an application. Currently, modulefiles contain the steps necessary to activate an application. This includes problems like resource allocation and verification, environment modification and dependency and conflict resolution. As more complex applications are developed for downloading over the Internet or over intranets, the traditional problems of integrating software onto the system arise.

Applications should not be responsible for handling configuration issues on the local host. As it stands today, Java applets request additional resources and other applets which may or may not be available to the local machine. The execution of the applet fails if these capabilities are not obtained – potentially after much of the applet has been downloaded. For example, certain types of or portions of applications will not work through a firewall. A user may begin loading such an application over the network and even begin executing it before discovering the missing dependency. If activating a particular feature of the application invokes an unsupported access to outside the firewall, then it depends on how well the application handles the failure and the nature of the work as to whether the failure will have a minimal or catastrophic affect on the user's work.

We envision a small modulefile-like Java program being downloaded into the local host prior to downloading and executing an application. Using a set of well-defined system configuration interfaces and tests as well as all the features of Java, the modulefile can verify the capabilities of the local host. Also, the modulefile can verify resolution of dependencies on other applets for the application. Once the modulefile has verified the host is configured such that it is likely to successfully execute the application does the transfer of the application begin.

Some sites may have different network restrictions and configurations as well as different policies with regard to downloading Java applications. The modulefile would be able to scope out these configuration issues prior to downloading the application in addition to potentially modifying the behavior of the application to suit local configuration parameters. With network bandwidth at a premium and Java applications tackling larger problems, using Modules in this fashion reduces the likelihood of wasting network bandwidth on attempting to load applications that won't execute.

As Java applications become more complex they will have the same tendency to depend on particular versions of other applets or Java libraries. The modulefile would be able to verify the local host doesn't have incompatible versions loaded. For a while, other dependencies may include the installation of a particular set of native methods on the local host.

The modulefile doesn't have to be static. It can interface with the user to suggest modifications to the local host prior to loading the application.

Like the modulefiles used in the Modules package, the pre-loaded modulefiles can provide information about the application that helps the user determine if he or she wants to continue the download. Furthermore, these pre-loaded modulefiles can handle issues like licensing and monetary exchange prior to downloading the application. In such a situation, the user might enjoy an indication of whether the application will execute on his or her machine.

We have just begun to scratch the surface of an implementation based on the ideas presented in this section. Some number of security and certification issues will need to be resolved as well as a more concrete understanding of the necessary configuration parameters a local host can make available to the modulefile before any implementation is complete.

Summary

In abstracting the user from the details of software installation, distribution and integration, Modules is a compelling solution to the software activation and management problem for large-scale distributed networks. We have contrasted the Modules package with other solutions to the software activation problem. In addition, we have introduced the reader to the features of the latest release of the Modules package as well as ideas for utilizing future capabilities of a Modules server.

Lastly, we have presented how the Modules concept can be exploited to manage the burgeoning configuration problems surrounding a swelling sea of increasingly complex Java applications.

Availability

Visit Modules' home page at http://www.modules.org/ for latest information on Modules.

Modules is freely available from ftp://ftp.modules.org/pub/Modules/ or from your friendly Tcl archive in your neighborhood, like ftp://ftp.neosoft.com/tcl/.

You can subscribe to the Modules interest mailing list, by sending the line "subscribe modules-interest <your-email-address>" in the message body to majordomo@modules.org.

Acknowledgments

We would like to thank a number of individuals who have made significant contributions in their support of the Modules package: Tony Bennett, Maureen Chew, Richard Elling, Leif Hedstrom, Doug Kubel, Don Libes, Ken Manheimer, Marty McLean, Phillip Moore and John Rouillard.

Development of the 3.0 version of the Modules package has been funded in part by the Semiconductor Group of Siemens AG. Much of the new features were implemented by Jens Hamisch, Strawberry EDV-Systeme GmbH, Munich. We are especially thankful for his contributions during the specification of the new features and his solid implementation. We also thank Connect! GmbH, Munich for providing ftp and web space.

John thanks all of his co-workers through the years for supporting the use of the Modules package and putting up with the bugs and problems during its development. John thanks his parents for their continuing support of all his travels and endeavors.

Peter thanks SAM for their *Never Ending Story*. Also, he would like to thank Johnny Clegg & Savuka for their album "Heat, Dust & Dreams" which helped him to keep working on Modules and this paper.

Author Information

John L. Furlani is currently technical lead of the Solaris XIL Imaging and Video development team at SunSoft, Inc. in Mt. View, CA. John received a Bachelor of Science in Electrical and Computer Engineering from the University of South Carolina at Columbia in 1990. While employed by Sun in North Carolina, he received a Masters of Science in Computer Science from Duke University in 1994. He was a systems administrator at USC and the Naval Research Laboratory in Washington, D.C. during his undergraduate college years. Upon graduation, John joined Sun Microsystems, Inc. as the systems administrator for Sun's North Carolina Development Center in Research Triangle Park, North Carolina. John enjoys playing the keyboard and the bassoon, hiking, traveling, snow skiing, cycling, rollerblading, cooking, eating great food and wine tasting. John can be reached via e-mail at j.furlani@ieee.org.

Peter W. Osel received his diploma in electrical engineering from the Technische Universität München (TUM) in 1985. For three years he worked at corporate research of Siemens AG, where he developed tools for ECAD of Integrated Circuits. From 1988 until June 1996 he was working for the Semiconductor Division of Siemens. He was responsible for worldwide integration and distribution of the CAD system, as well as the development of central tools, and the coordination of the development sites' system environments. Since July 1996 he is working for Siemens Business Services GmbH & Co OHG in the department that administrates all workstations of the Semiconductor Division of Siemens. In September 1996 he moved from Munich, Germany, to Cupertino, CA, where he designs, implements and administrates the compute environment for the development and marketing of Siemens' new microcontroller family. Reach Peter at Siemens Business Services GmbH & Co OHG, SBS DS 33, Postfach 801709, D-81617 München, Germany; or at Siemens Components, Inc., 10950 North Tantau Avenue, Cupertino, CA 95014; or by e-mail at pwo@HL.Siemens.DE, or see his Web page at http://www.ConnectDE.NET/˜pwo/.

References

[1] John L. Furlani, "Modules: Providing a Flexible User Environment", *Proceedings of the Fifth Large Installation Systems Administration Conference (LISA V)*, pp. 141-152, San Diego, CA, September 30 – October 3, 1991.

[2] John K. Ousterhout, *Tcl and the Tk Toolkit*, Addison Wesley Publishing Company, Inc., ISBN 0-201-63337-X, 1994.

[3] Rémy Evard and Robert Leslie, "Soft: A Software Environment Abstraction Mechanism", *Proceedings of the Eighth Systems Administration Conference (LISA VIII)*, pp. 65-74, San Diego, CA, September 19-23, 1994.

[4] Richard Elling, Matthew Long, "user-setup: A system for Custom Configuration of User Environments, or Helping Users Help Themselves", *Proceedings of the Sixth Systems Administration Conference (LISA VI),* pp. 215-223, Long Beach, CA, October 19-23, 1992.

[5] Carl Hauser, "Speeding Up UNIX Login by Caching the Initial Environment", *Proceedings of the Eighth Systems Administration Conference (LISA VIII)*, pp. 117-124, San Diego, CA, September 19-23, 1994.

[6] Christopher Rath, "The BNR Standard Login (A Login Configuration Manager)", *Proceedings of the Eighth Systems Administration Conference (LISA VIII)*, pp. 125-138, San Diego, CA, September 19-23, 1994.

[7] David F. Skoll, "envv – manipulate environment

variables in a shell-independent manner", UNIX man page and source code for version 1.6 of the application, July 1995.

[8] Michael A. Cooper, "Overhauling Rdist for the '90s", *Proceedings of the Sixth Systems Administration Conference (LISA VI)*, pp. 175-188, Long Beach, CA, October 19-23, 1992.

[9] Peter W. Osel, Wilfried Gänsheimer, "OpenDist – Incremental Software Distribution", *Proceedings of the Ninth Systems Administration Conference (LISA IX)*, pp. 181-193, Monterey, CA, September 17-22, 1995.

[10] Cray Research, Inc. Products. http://www.cray.com/PUBLIC/product-info/sw/ .

[11] Larry Wall and Randal L. Schwartz, *Programming perl*, O'Reilly & Associates, Inc., Sebastopol, CA, 1991.

[12] System V software management utilities, Solaris 2.5 manual pages *pkgadd*(1), *pkgmk*(1), *pkgtrans*(1) etc.

[13] *AFS distributed filesystem FAQ*. http://www.cis.ohio-state.edu/hypertext/faq/usenet/afs-faq/faq.html .

[14] James Gosling, Henry McGilton, "The Java Language Environment", A White Paper from Sun Microsystems, Inc., October 1995. http://java.sun.com/doc/language_environment/ .

[15] Wallace Colyer and Walter Wong, "Depot: A Tool for Managing Software Environments", *Proceedings of the Sixth Systems Administration Conference (LISA VI)*, pp. 151-162, Long Beach, CA, October 19-23, 1992.

[16] John P. Rouillard and Richard B. Martin, "Depot-Lite: A Mechanism for Managing Software", *Proceedings of the Eighth Systems Administration Conference (LISA VIII)*, pp. 83-91, San Diego, CA, September 19-23, 1994.

[17] Walter C. Wong, "Local Disk Depot – Customizing the Software Environment", *Proceedings of the Seventh Systems Administration Conference (LISA VII)*, pp. 51-55, Monterey, CA, November 1-5, 1993.

[18] opt_depot – http://www.arlut.utexas.edu/opt_depot/opt_depot.html .

[19] *csdsdb – Computer Science Division Software Database*, http://www.arlut.utexas.edu/csd/csdsdb/ .

[20] ini – GeNUA GmbH, ftp://www.genua.de/tools/ini.tar.gz .

[21] Kenneth Manheimer, Barry A. Warsaw, Stephen N. Clark, Walter Rowe, "The Depot: A Framework for Sharing Software Installation Across Organizational and UNIX Platform boundaries", *Proceedings of the Fourth Large Installation Systems Administrator's Conference*, pp. 37-46, Colorado Springs, CO, October 18-19, 1990.

[22] John Sellens, "Software Maintenance in a Campus Environment: The Xhier Approach", *Proceedings of the Fifth Large Installation Systems Administration Conference (LISA V)*, pp. 21-28, San Diego, CA, September 30 – October 3, 1991.

[23] Michel Dagenais, Stéphane Boucher, Robert Gérin-Lajoie, Pierre Laplante, Pierre Mailhot, "LUDE: A Distributed Software Library", *Proceedings of the Seventh Systems Administration Conference (LISA VII)*, pp. 25-32, Monterey, CA, November 1-5, 1993.

[24] Steven W. Lodin, "The Corporate Software Bank", *Proceedings of the Seventh Systems Administration Conference (LISA VII)*, pp. 33-42, Monterey, CA, November 1-5, 1993.

[25] Barrie Archer, "Towards a POSIX Standard for Software Administration", *Proceedings of the Seventh Systems Administration Conference (LISA VII)*, pp. 67-79, Monterey, CA, November 1-5, 1993.

[26] Sandberg, R., D. Goldberg, S. Kleiman, D. Walsh, B. Lyon, "Design and Implementation of the Sun Network Filesystem," *USENIX Conference Proceedings,* USENIX Association, Berkeley, CA, Summer 1985.