Parallel Processing Letters Vol. 9 No. 2 (1999) 215–225
© World Scientific Publishing Company

BINARY VERSION MANAGEMENT FOR COMPUTATIONAL GRIDS

JEFFREY K. HOLLINGSWORTH
Computer Science Department, University of Maryland, College Park, MD 20742
hollings@cs.umd.edu

ETHAN L. MILLER and KENNEDY AKALA
Computer Science & Electrical Engineering Department
University of Maryland Baltimore County, 1000 Hilltop Circle, Baltimore, MD 21250
{elm,kakala1}@csee.umbc.edu

Received September 1998
Revised March 1999
Accepted by B. Tourancheau and J. Dongarra

ABSTRACT

Applications are no longer monolithic files, but rather a collection of dynamically linked libraries, images, fonts, etc. For such applications to function correctly, all of the required files must be available and be the correct version. Missing files preclude application execution, and incorrect versions lead to mysterious and frustrating failures. This paper describes a simple scheme to address this problem: Content-Derived Names (CDNs). CDNs use digital signatures to automatically and uniquely name specific versions of files. Because Content-Derived Names are computed using a cryptographically strong hash over the text of a package, this process is safe from spoofing and other attacks based on providing the wrong library. We explain how CDNs ease the management of application distribution for clusters and grids. We also describe a prototype implementation of CDNs for the Tcl programming language.

1. Introduction

The proliferation of complex software libraries has made development easier by providing high-level functionality to application programmers. However, these libraries also complicate matters by introducing potential incompatibilities between an application and the packages that it wants to use. This paper discusses a technique to manage the components of applications that eases the distribution of large applications and allows the inclusion of two different versions of the same library, as may be required by large applications that include packages that themselves require packages. Additionally, Content-Derived Names permit applications to automatically download needed packages over the network, even from untrusted hosts, and insert them into running code.

In addition to making distribution of applications much easier, Content-Derived Naming permits many versions of packages to coexist on a single machine. The dynamic library facility of many Unix operating systems currently allows this, but only supports exact or later than testing for package version numbers. In our experience, however, the latter approach is dangerous; often, changes in a library from version 2.0 to 2.1 break some applications that used the earlier version. In such a case, the application designer is deluged with requests to squash bugs. Using our system, a developer can supply an application and specify exactly which packages and files must be used with it. Since the developer controls the environment more precisely, they are better able to test the application's behavior.

For a computational grid, maintaining software configurations becomes even more complicated. A single application might run on many different computers that are distributed both geographically and administratively. If even one site has the wrong version of the software, the application execution can fail and may waste significant computational resources at the same time. For this type of environment, a mechanism to verify and install missing application components is critical.

Content-Derived Names are computed by hashing the contents of a file using a secure hash such as MD5 [7] or SHA-1 [1] and recording the (relatively short) result in the file that uses the package. It is a simple matter for our tool to recompute the hash value before importing a file, guaranteeing that the application is indeed using the appropriate file. We presented the original concept for Content-Derived

Names in [3]. This paper provides the first description of an actual implementation of the idea. Also, we explain how the concept would be useful for a computational grid.

2. Background

While the idea of using Content-Derived Names (CDNs) for configuration management of executable code is new, there has been previous work in the areas of using explicitly managed version numbers to provide configuration management. Our work builds on this as well as research in secure hash functions.

2.1 Configuration Management

Most of the research in configuration management has concentrated on managing the construction of applications from a source code repository. This approach can work well with object code binaries, producing a single monolithic executable object that may be distributed. If this is the case, there is no need for further management of multiple versions of the same code. While the software developer must keep track of many versions of code, the end user need not. As a result, much commercial software is distributed this way.

Increasingly, however, software developers are providing their applications as a collection of code objects. The use of dynamically linked libraries in Unix, MacOS, and Windows facilitates this modular approach. With script-based languages such as Tcl and Java, though, distribution of applications as dozens or hundreds of individual files is practically guaranteed. Managing these files can cause subtle and frustrating failures, as many users have experienced when installing programs on their personal computers. Each application provides the libraries that it needs in the versions that it prefers, overwriting any existing versions of the libraries. Of course, this approach causes some existing programs to fail because their preferred version of the library has been erased by a later installation. Van der Hoek, et. al. [9] addressed this problem of "software release management" by suggesting a system to support software acquisition by ensuring that the correct versions of dependent packages are acquired with the primary package. However, their approach relies on a centralized software repository and explicit administration of version numbers for all packages. In contrast, our approach is completely decentralized and allows anyone to install a new application by simply entering a short (less than 50 characters) string of hexadecimal digits and a location from which to retrieve the object.

2.2 Secure Hash Functions

A key feature of Content-Derived Names is the use of a secure hash function to assign a unique name to an object based solely on its content. Digital signature algorithms such as MD5 [7] and SHA-1 [1] are one-way functions that take arbitrary data and produce a result that is very likely to be different from that of any other (different) input sequence. Our implementation uses MD5 to generate CDNs, but other algorithms could easily be substituted.

Because it is NP-hard to find another object that produces the same digital signature as a given object, it is unlikely that two objects will have the same signature, either by chance or by malicious construction of an object. For the 128-bit signature provided by MD5, the chance of two objects out of 10^{15} of having the same signature is approximately 10^{-19}. By increasing the signature length to 256 bits, the chance of collision drops to 10^{-14} for 10^{30} unique objects [3].

The use of secure hash functions provides another benefit beyond conflict-free naming, however. It allows applications to ensure that the code they are loading is authentic, preventing the introduction of Trojan horses. This concept is also discussed by Moore[5]. If a developer has a virus-free environment (and we hope that they do), the hash values that they compute will be those for correctly working code. If a virus later infects any piece of code, the secure hash will change and the loader will be able to reject the package, instead choosing to download a new version from the network.

3. Content Naming Explained

Many existing implementations of dynamic libraries have a mechanism to accommodate packages with different version numbers that permits applications to request a package with a specific version number or any version number later than a specific version. In fact, most of these systems include a two-level namespace based on major and minor version numbers. A major version number is incremented when an "incompatible" change is made to the library, and a minor version number is incremented with each release that is supposed to be compatible. A piece of code may request a library either implicitly (by having been linked with a specific dynamic library) or explicitly via a call to a `dlopen` statement. In either case, the system searches through a list of directories for an entry that provides the correct major version of library and then the latest minor version. While this approach can work with well-behaved libraries, it presents several difficulties. First, users must make sure that all files are available before running the application. While this might be clumsy, if manageable, on a single system, extending this model to a computational grid with many semi-autonomous nodes is problematic. Library availability is only the start, though. Another issue is version management. Use of versions greater than the one with which the application was tested can cause bugs in a program. While developers would like to think that version 2.2 is fully backward compatible with version 2.1, this is often not the case.

To address this problem, developers who care if their code works, only issue major version numbers, allowing them to test their code with all of the files that it will use. This approach introduces another problem, however. With complex code, it is possible that a single application may require two versions of a single package. The high-level code may not even be aware of this conflict if two packages themselves each require a different version of the same lower-level package, as shown in Figure 1. The developer of the root object may not even know of the conflict if she received the code for the two top-level packages from different sources. This can also introduce naming problems for unwary code designers because it requires that every version of a package have a unique file name. Of course, this can be done by appending the version number to every file in a package, leading to the problem of deleting old files when the package using them is gone. Figure 1 shows an example of an application built using two libraries: a mesh library and a sparse matrix library. Both these packages use versions of the a common linear algebra package. However, each one uses a different version of the library.

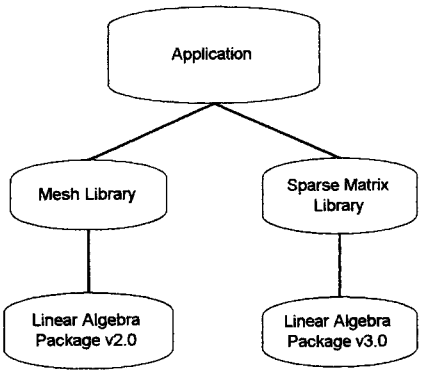

Figure 1 : Package requirement conflicts in a complex application.

The basic concept underlying content-naming is that a complex software installation can be thought of as a directed graph of procedure calls, and that procedures are grouped together into packages. The user does not care about internal package names; names are for the convenience of developers only.

While external function names are important to those using a package, the name of the package itself is still largely irrelevant. A CDN provides all of its benefits by converting a package name from a name and version number meaningful to a developer into a Content-Derived Name that can be used to check library integrity and support secure remote retrieval. Since this name is probabilistically guaranteed not to conflict with other library names, it may be shared between different computers without fear of name duplication.

The overall structure of an object in our scheme is shown in Figure 2. An object consists of the object body, external object references, customization region, and its CDN. The object body contains the majority of the object, including its executable code. References to other objects or customization data are represented as pointers to the appropriate section of the object. Each object reference can be a list of CDNs for equivalent objects. Although this information is immutable, it needs to be in a designated section of an object so that the object manipulation routines can identify an object's external object references. The customization region has two sections. One to store customized references to other objects and the second to store free format data. The only requirement is that pointers from the object body can't be modified due to customization since this would change the object's CDN.

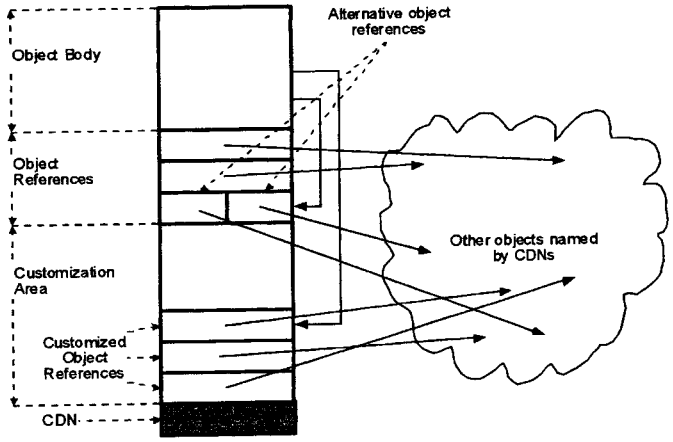

Figure 2: Layout for an object using CDNs.

4. Advantage of Content-Naming

In this section, we describe some of the advantage of using CDNs relative to using other approaches to version management and network software distribution.

4.1 File System Independence

A tedious aspect of installing software packages and libraries is managing the directories that they will be installed in. Many sites have developed their own (often incompatible) conventions for naming directories that contain software. For example, some sites put each package in a separate directory, others use common directories. Some sites install software with the version number as part of the directory name, or install different versions of software in different subdirectories grouped by application. By assigning Content-Derived Names, we guarantee that each version of each package has a unique name. Thus, we can store all packages in a single directory with no fear of name conflicts. Because all package

files reside in a single directory, the application will work regardless of what that directory is actually named. There is no need to embed directory information directly or indirectly into programs; instead, the CDN system has a single directory in which it looks for components.

Of course, the efficiency of a file system may drop when handling directories with potentially thousands of files, but this problem has been solved by using B+ trees for directory lookup in recent file systems such as SGI's XFS[8] and Microsoft's NTFS[6]. Placing all installed libraries in one directory eliminates the need for users to specify information about where the software packages will reside, and makes it simpler for designers to test the software because they no longer have to worry about users with different package search paths. For computational grids, eliminating the need to manage directory-naming conventions among different sites greatly eases the process of ensuring that applications can run at different locations.

4.2 Locating Software Components

One of the best feature of the CDN systems is that it permits automatic downloading of missing software compoents. If a required package is not found in the single directory that holds CDN-named packages, it may be fetched from a remote Web server using HTTP. This can be done without the user's knowledge; the only evidence that the network was consulted is the increased delay. For computational grids, this allows applications to automatically install packages when an application that needs them starts to execute on a node.[1]

When an attempt is made to load a library, the system first looks in the directories specified for conent named files on the local system. If the library is not found there, it proceeds to query each of the URLs contained in a list of likely download sites. This list is searched in order, so a site may put its own package cache server first before more comprehensive, but more distant caches. Additionally, applications can augment the site list, enabling an application to specify a Web server from which its component packages may be obtained.

By using this two-level approach, a site may maintain a cache of packages for use by many machines at the site. If the object is not available there, our system can go to either a public server with many packages (perhaps a version of netlib or the High-Performance Software Exchange) or to the developer's own site to download the required files.

The integrity of packages found on the Web is of utmost concern because it is far too easy to implement a trojan horse. Thus, our system checks the integrity of any downloaded. If an integrity check fails, the package is treated as if it were never there, and other servers will be consulted to determine if they contain a valid copy of the file. If no valid copy of the file is found locally or on any server, and error is reported.

5. Tcl Prototype

To demonstrate the ideas of Content-Derived Naming, we have created a prototype implementation that works with the Tcl programming language. Tcl is a widely used scripting language that has been used to build a variety of applications. In addition, many libraries (called packages) exist that provide useful functionality. As a result, package management in Tcl is similar that that of compiled binaries. We selected Tcl as our initial implementation platform since as a freely available scripting language, the details of implementation were easier than if we used native binary programs. Our prototype, called tcdn, has been used to convert several applications, and the software is available for downloading from the Internet (`http://www.csee.umbc.edu/~elm/Projects/CDN/`).

---

[1] Software licensing issues must be handled. However, they would also need to be handled if traditional naming were employed.

5.1 Existing Tcl Version Management

Tcl has a mechanism to accommodate packages with different version numbers that permits applications to request a package with a specific version number or any version number later than a specific version. A piece of code may request a package using a package require statement; this causes Tcl to search through the package index file for an entry that provides the package.

This index is built by searching files for `package provide` statements. The index file is constructed by looking through files in the order specified by a Tcl specific variable, and it must be constructed statically (though it could be run automatically when an unknown function is encountered). Nonetheless, the standard Tcl approach requires that a user install all required packages before running an application.

5.2 Design of tcdn

The goal of tcdn is to convert a directed package into a package graph using Content-Derived Names, such as that shown in Figure 3. In this figure, a hypothetical SNMP client is shown. The application consists of the root object (the application itself), and five packages. The graph has unique names for all packages. Notice that both the GraphIt package and the NetObjects package use the DrawingUtil package. However, the GraphIt package relies on version 3.0 and NetObjects uses version 5.0. By using tcdn, each package get the required version.

To see how tcdn ensures the correct versions of all packages transitively from the root object, consider the package GraphIt. The code for GraphIt v1.2 includes `tcdnpackage` require statements that reference the packages that it uses (`fb2729a8852d2c1c78a8d2a1f0d43f3` and `3ae6c72e98bb07f47ace823901328f6b`). Since the Content-Derived Name for GraphIt-1.2 is a cryptographic hash over its entire code, including the statements that reference the packages used by GraphIt-1.2, it is impossible for a malicious user to change the references to the two packages without changing the hash, and thus the CDN, of GraphIt-1.2. Thus, the CDN of each object ensures not only the correct version of that object, but of all objects used by that object.

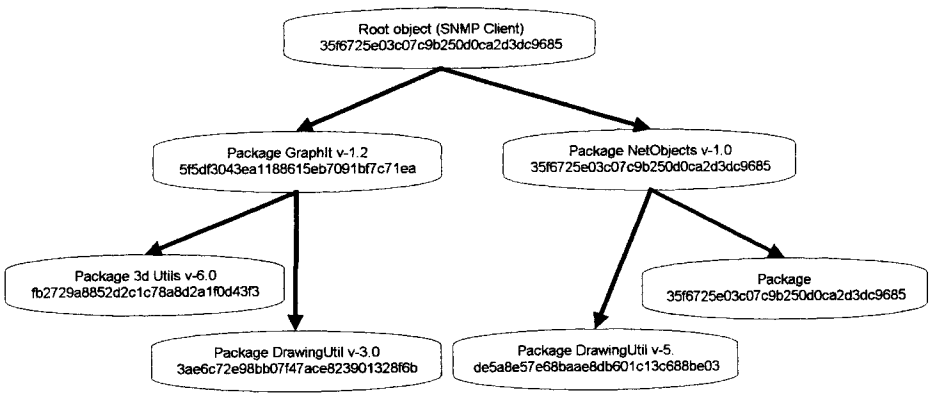

Figure 3: Packages converted to tcdn format.

More generally, the user need only trust a single Tcl package, that contains the routine that is called to start the whole application. If the name for that object is obtained from a trusted source (perhaps as part of a financial transaction in which a user purchases the software), the user can obtain the root object itself, as well as all objects it requires, from any computer willing to provide them. If the user does not trust other servers they can check the cryptographic hash of a downloaded object against the name she provided. If they do not match, the object is faulty.

5.3 What the Developer Sees

Developers need not radically change the way they write code to benefit from tcdn. Packages to be turned into tcdn packages are written in largely the same way as normal packages. There are, however, a few restrictions that must be followed to allow CDNs to work. The restrictions are:

- Each package must have its own namespace. This namespace must be named so that its name is different from that of every other package, including different versions of this package. If two versions of a package share the same namespace, they cannot each use different versions of underlying CDN-identified packages in a single program. Giving different versions of a package each a unique namespace is not difficult, however, because the version information can be appended to the namespace name to guarantee a unique name[2].
- Each package must be contained in a single file. As a result, each file must have a package provide statement.
- Packages may not use mutual recursion. In other words, if package A requires package B, package B may not in turn require package A. The simplest way around this problem is to break up one of the packages into two pieces, removing the cycle in the package graph. An alternative solution would be to combine packages A and B into a single, larger package.

Once the code is complete, a tool is used to rewrite all of the package names into Content-Derived Names. This is accomplished using a Tcl procedure with similar semantics to the function used to build the index files used by the current Tcl loader. The routine to perform name conversion is called as follows:

```
tcdn::tcdnify <destination> <source files...>
```

This command translates all of the source files named in the command, and places the resulting CDN files into the directory named by <destination>.

After converting files with tcdnify, packages may then be distributed to other developers who can use the package with tcdn::tcdnpackage require or to end users. Of course, this distribution may include all of the files if desired. A distribution would include all required files if the destination were likely to have poor or non-existent access to the Internet. A more attractive option, however, is to distribute the package by simply providing the Content-Derived Name (the entire object can be sent, but is not necessary) to the user. Future invocations will then automatically fetch the desired objects from either the developers Web server or any other Web server that has a copy of the file.

5.4 Installing tcdn

The tcdn package was written so that it can coexist and work with the existing package system. The tcdn package links the two worlds, and is both a tcdn package and a regular package. In fact, it is necessary to use the regular package mechanism to install and use the tcdn package. Traditionally, installing software to be used by all users on a system will entail placing the package in directories where

[2] Currently Tcl requires that every package have its own namespace to support its explicit version management. Our additional restriction could be accommodated easily with a pre-processor that attaches the version number onto the package name.

writing is restricted. However, with tcdn, this is only required for the initial installation of tcdn package itself. Since the tcdn package is itself a CDN-based package, later updates can be made automatically. Any package or application that uses the tcdn package needs to include a package require tcdn command. Following this, the package or application can use an upgraded tcdn package if it is available by including a tcdnpackage require statement. The semantics of using a CDN-named package will be covered shortly. For now, the important thing to know is that the original version of the tcdn package can be replaced at any time by a newer version if one is available.

The initial installation will also require a small amount of configuration. The only configuration is to specify the directory where the downloaded CDN-named packages will reside. Surprisingly, the CDN-named packages should be stored in a completely public directory, readable and writable by all. Usually, this would be a problem because it would open the end user up to all manner of trojan horse attacks. If tcdn were not in use and packages were stored in this manner then any user would be able to replace a package with whatever they wanted. This would be like making /bin world writable! Tcdn protects against this by making sure that the package an application is loading really is the right one. With tcdn, the offending package would simply be deleted and replaced with the correct one. The details of this process are covered in the next two sections.

Keeping packages in a public directory is an immense advantage because it allows the end user to use an application without having to get the access required to download and install all of the packages required by the application. It also allows the installation process to be more completely automated, thus making distribution much easier.

5.5 Creating a CDN-Named Package

The tcdn library was designed so that it would be easy for newly CDN-named packages to be created. The goal here is to make things simple so that the programmer will not have to go to unreasonable lengths to create a CDN-named package.

There are several conventions that must be followed when creating a package. CDN-named packages should contain a single package provide statement. The name of the package does not matter because it will be removed. This is necessary to allow for the conversion of multiple packages with multiple files each at the same time. Global variables should, of course, be avoided. Namespaces are not required, but they are recommended. If no namespaces are used, tcdn cannot ensure that the correct version of required packages are loaded for the same reason that vanilla Tcl cannot do so. Care must be taken when naming namespaces, as namespace collisions can still occur. The easiest method of assuring a unique namespace is to append the version number of the package to its name and use that as the namespace name. Following this short list of rules should be easy, as it allows the programmer to create packages in a more normal fashion.

CDN-named packages should also contain a command named tcdnInit. This command should exist in the global namespace. The purpose of this command is to allow the package to do initialization if it needs to. Remember, a package may not have existed on a system prior to the first time it is used. The tcdnInit command will allow the package to perform any setup that it needs to. If the package does not need to perform any special setup then the command can be left out. The command is executed right after the package is loaded and before control returns to the application. It is very important that the command have as few side affects as possbile, because it will be running in the context of the application.

CDN-named packages should store their configuration information in the user's directory. This is just like storing user options, and in fact just adds global options. The tcdnInit command should check for this configuration information before creating it or asking for it. The package programmer should make sure that this information will not take up too much space, and should also insure that any errors in its creation or reading will be handled without crashing the application. This requirement is not unique to tcdn, since no one would want to use a regular package that caused applications to crash.

Using another regular package in a CDN-named package is simple. The programmer just inserts package require statements as usual. Of course, the package being requested must exist on the system where the package is used or the package require command will fail. Using a package in this way may be necessary sometimes, especially since not all packages may be available as CDN-named packages. If CDN-named packages were used all around then the normal package mechanism could be replaced completely, but until then CDN-named packages may require a regular package every now and then.

Using a CDN-named package is just as simple. Tcdn provides a tcdnpackage require statement that handles the loading of CDN-named packages. In this case the requested package does not need to exist on the system at all, because tcdn will find and download it when it is needed. This frees the programmer from having to specify what packages are needed in order to be able to use their package. With tcdn the end user does not have to manually download the needed packages or even install them.

Once written, a CDN-named package must be converted. Tcdn was written as a library, so anyone can create an application that does the actual conversion. This was done so that the process could be as flexible as possible. To make simple conversion easy, tcdn provides a tcdnify command to do this conversion. The tcdnify command works on properly written regular packages that are to be converted to CDN-named packages. It strips the package of package provide statements and resolves any package interdependencies which may exist between packages being converted at the same time. It then outputs the CDN-named package with the correct Content-Derived Name as the file name. The current version of tcdn does this because it is assumed that programmers will be more comfortable with creating packages in the manner they have been used to. Future versions will provide a mechanism for simply generating the name of a package that has been written as a CDN-named package from the beginning.

5.6 The tcdnify Process

The tcdnify procedure has three steps. First, it creates a list of packages, resolving any source statements it finds. Next, it orders the packages by their dependencies on each other. If package A requires package B, then package B must be converted first because the secure hash for package A depends on the Content-Derived Name for package B. A sample dependency graph for the packages listed in Figures 1 and 2 is shown in Figure 3. The order in which files are processed is noted next to each file. Note that, in all cases, a package is processed after all of its children have been processed.

Once the files have been ordered, tcdnify runs through a loop for each package in order. For each package, all package require statements are converted to tcdnpackage require statements with the appropriate CDNs, and then the entire file is hashed with MD5. The result is stored in the specified destination directory.

Perhaps the most difficult part of this process is ordering the packages by their dependencies. While this could have been left out by simply requiring the user to convert a single package at a time, we felt that it was important to make the process as automatic as possible. As a result, a developer need only run tcdnify on an entire project to prepare it for distribution. Once this has been done, the resulting files can be made available for distribution via http or ftp, with only the root object distributed to potential users.

5.7 Using a CDN-Named Package

The user's view of a large Tcl application is greatly simplified using tcdn. Rather than having to download and place dozens of files, some of which may overwrite previous files, users simply request a single object that automatically fetches other objects over the Web. There is no longer a need to add a new directory to the package search path for the new application, and users who prefer the old package may continue to use it with no naming conflicts.

Using a CDN package is easy. Because tcdn is not the primary package mechanism the application will need to have a package require tcdn command. This will load tcdn and all of its commands. From here all that is needed is a `cdnpackage` require statement for each CDN-named package that will be used. After this the process is automatic.

If the CDN-named package is located on the system it is checked and then loaded. The check involves regenerating the Content-Derived Name. The generated name is then compared with the requested name. If the two match then the package has been located and verified and can be loaded. If the two do not match, then it is assumed that the package file is corrupt and it is thrown away. If there are other directories to search, then this process is repeated for each of them. If not, the package must be retrieved from a server.

The loading process for remote CDN-named packages is similar to the loading process for local packages. Each server is queried in turn for the desired package. Tcdn has been written so that different protocols can be used for each server. If none of the available servers returns the desired cdn package then the cdnpackage require command fails. This is not a normal situation, and would only happen if the network was unavailable or some other occurrence somehow prevented access. Usually, at least one server will return the requested package. The Content-Derived Name is then verified, just as it would be if it were local. If the generated CDN matches the requested CDN then the package file is usable and can be saved. If not, the package is discarded and the process continues.

6. Future Directions

Having demonstrated the usefulness of CDNs in Tcl, we hope to extend our work to other languages. In particular, we are building similar functionality into the dynamic library loaders for Windows NT and Linux, allowing them to reap the benefits of automatic installation of software packages. Doing so will also provide an additional benefit: the ability to dynamically load binary libraries into Tcl.

This technology should also be applicable to Java [2], providing additional security for complex applications at little overhead. Rather than authenticate all applets, requiring a relatively expensive check for each small piece of code, our system requires only that a root object be authenticated. Once this is done, the integrity of the objects immediately below the root is ensured because their names are embedded in the authenticated objects. This can transitively be applied to the entire dependency graph, allowing a computer to check most applet code locally without relying on external certificate providers.

Content-Derived Names are also useful in supporting software checkpoint and migration. For example, the Condor system[4] currently copies the full versions of all shared libraries when applications are migrated to new nodes. This is done to ensure that the correct version of the library is available on the target node. Since Condor uses checkpoints to move processes to non-idle nodes, the application may be stopped during execution of a call to a shared library. Also, libraries may contain state. As a result, when a program is restarted, the **exact** version must be used on the new node. CDNs provide exactly the semantics required for this environment, and could allow condor to migrate libraries only when they are not available at the target node. We are currently discussing the possibility of using CDNs as part of the Condor system.

7. Conclusions

This paper has presented Contend-Derived Naming, an approach to binary version management that has several advantages over current management methods: freedom from version conflicts, integrity checking for packages, and the ability to dynamically download needed modules from remote sites. It is our hope that this will ease the dynamic library miss-match problem that plagues Windows PCs, as well as help to ease the problem of configuration management for computational grids. All that is necessary to install an entire application is the Content-Derived Name of its root object and a location from which to

get it; from there, everything is handled automatically. If the software is upgraded, the user need only get a new root object from the developer, and the package dependencies are updated automatically.

Because CDNs can provide integrity checking and the ability to fetch missing packages from remote server sites, we believe it will be essential for application developers who wish to make software available via the Web.

References

1. *Secure Hash Standard*, FIPS-180-1, National Institute of Standards and Technology, U.S. Department of Commerce, April 1995.
2. J. Gosling, B. Joy, and G. Steele, *The Java Language Specification*. 1996: Addison-Wesley.
3. J. K. Hollingsworth and E. L. Miller, "Using Content-Derived Names for Configuration Management," *Symposium on Software Reusability (SSR)*. May 1997, Boston, ACM Press, pp. 104-109.
4. M. Litzkow, M. Livny, and M. Mutka, "Condor - A Hunter of Idle Workstations," *International Conference on Distributed Computing Systems*. June 1988, pp. 104-111.
5. J. W. Moore, "The Use of Encryption to Ensure the Integrity of Reusable Software Components," *International Conference on Software Reuse*. Nov. 1994, Rio de Janeiro, pp. 118-123.
6. S. M. Moore, ed. *Microsoft Windows NT Resource Kit*. 1996, Microsoft Press: Redmond, WA. 1350.
7. R. L. Rivest, *The MD5 Message-Digest Algorithm*, RFC 1321, Network Working Group, April 1992.
8. A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck, "Scalability in the XFS File System," *Winter 1996 USENIX Conference*. January 1996, San Diego, CA, pp. 33-44.
9. A. van der Hoek, R. S. Hall, D. Heimbigner, and A. L. Wolf, "Software Release Management," *6th European Software Engineering Conference*. Sept. 22-25, 1997, Zurich, Switzerland, pp. 159-175.