Using Content-Derived Names for Package Management in Tcl

Ethan L. Miller & Kennedy Akala
Computer Science & Electrical Engineering Department
University of Maryland Baltimore County
1000 Hilltop Circle
Baltimore, MD 21250
{elm,kakala1}@csee.umbc.edu

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

ABSTRACT

Managing different versions of library routines has long been a problem, both for Tcl and for other languages that permit code reuse and modification (i.e., all computer languages that the authors are aware of). This problem is particularly difficult for Tcl because it allows libraries (in the form of packages) to be dynamically loaded as needed. While this feature is very convenient — users need only keep a single copy of each library to use it in many programs — it can lead to code compatibility and distribution problems.

This paper presents a solution for this problem — using content-derived names (CDNs) to name Tcl packages. Using this solution, a program can simultaneously use two different versions of a single package. In addition, the Tcl interpreter can easily find instances of a missing package over the network and download them, making them available to a running application. 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. Thus, a user may download missing packages from any server willing to provide them without fear of virus or trojan horse attacks.

1 Introduction

The proliferation of complex software libraries has made development easier in Tcl as well as in other languages 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. While Tcl can use the `package` statement to deal with this problem, there are still many shortcomings that need to be addressed. This paper discusses an extension to traditional Tcl packages that eases the distribution of large Tcl 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, the `tcdn` (Tcl Content-Derived Name) mechanism permits applications to automatically download needed packages over the network, even from untrusted hosts, and insert them into running code. Using this package, application distribution changes from a `tar` file with dozens of Tcl files and a `README` list of required packages to a single "root" Tcl file. All of the remaining code can be dynamically fetched as needed.

In additional to making distribution of applications much easier, `tcdn` permits many versions of packages to coexist peacefully on a single machine. The Tcl `package` mechanism currently allows this, but only supports "exact" or "later than" testing for package version numbers. In our experience, however, the "later than" approach is dangerous; often, changes in a library from version 2.x to 2.(x+1) break some applications that used the earlier version. In such a case, the application designer is deluged with requests to squash bugs. Using the `tcdn` system, however, 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, she is better able to test the application's behavior.

The `tcdn` package provides a third benefit: security. Content-derived names are computed by hashing the contents of a Tcl package using a secure hash such as MD5 [6] or SHA-1 [1] and recording the (relatively short) result in the file that uses the package. It is a simple matter for a Tcl file to recompute the hash value before importing a file, guaranteeing that the application is indeed using the appropriate file.

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. The `tcdn` package builds on this work as well as research in secure hash functions.

This paper appeared in the 6th Annual Tcl/Tk Conference, September 1998, San Diego, CA, pages 171–179.

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 programs' use of standard code. With script-based languages such as Tcl, though, distribution of applications as dozens or hundreds of individual files is practically guaranteed. Managing these files can cause big problems, as the authors 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. Tcl provides a mechanism to avoid this, but its use requires detailed knowledge of package search paths. Additionally, file names can become a problem because different versions usually use the same file names, and package search paths can become exceedingly long. van der Hoek, et. al. [3] 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 Package Versions in Tcl

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.

While this approach can work with "well-behaved" packages, it presents several difficulties. First, users must make sure that all files are available before running the application. While attendees at the Tcl/Tk conference may have no difficulty doing this, average users will have somewhat more trouble. Package 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 about their code working should use the `exact` option to the `package require` statement, 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. In Tcl, this conflict cannot be easily resolved because only one of the `package require` statements will be able to load the desired package. The developer of "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.

2.3 Secure Hash Functions

A key feature of `tcdn` 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 [6] 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

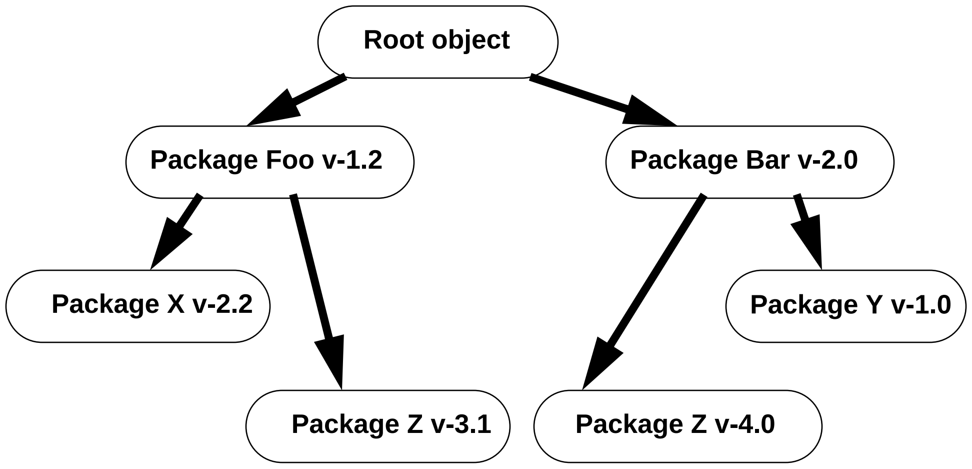

Figure 1. Package requirement conflicts in a complex application.

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} having the same signature is approximately 10^{-9}. By increasing the signature length to 256 bits, the chance of collision drops to 10^{-14} for 10^{30} unique objects [4].

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 in [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 `Tcdn` Design

The basic concept underlying `tcdn` is that a complex software installation can be thought of as a directed graph of procedure calls, and that procedures are grouped together into Tcl 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 — just a word to be typed into a `package require` statement.

Tcdn provides all of its benefits by converting package names from a name and version number meaningful to a developer into a content-derived name that ca n be used to check package integrity and support secure remote retrieval. Since this name is probabilistically guaranteed not to conflict with other package names, it may be shared between different computers without fear of name duplication.

3.1 High-Level Design

The goal of tcdn is to convert a directed package graph such as that in Figure 1 into a package graph using content-derived names, such as that shown in Figure 2. Unlike the graph in Figure 1, which might have two `z.tcl` files (one for version 3.1 and one for 4.0), the graph in Figure 2 has unique names for all packages. Moreover, the code for `foo-1.2` includes `tcdn-package require` statements that reference the packages that it uses — in this case,
`1ee91c2024d8dbe901a33bf3b3200afe`
and
`42faca939af96f68ac164858cffdbc96`.
Because the content-derived name for `foo-1.2` is a cryptographic hash over its entire code, including the statements that reference the packages used by `foo-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 `foo-1.2`.

More generally, the user need only trust a single Tcl package, that which 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 (a

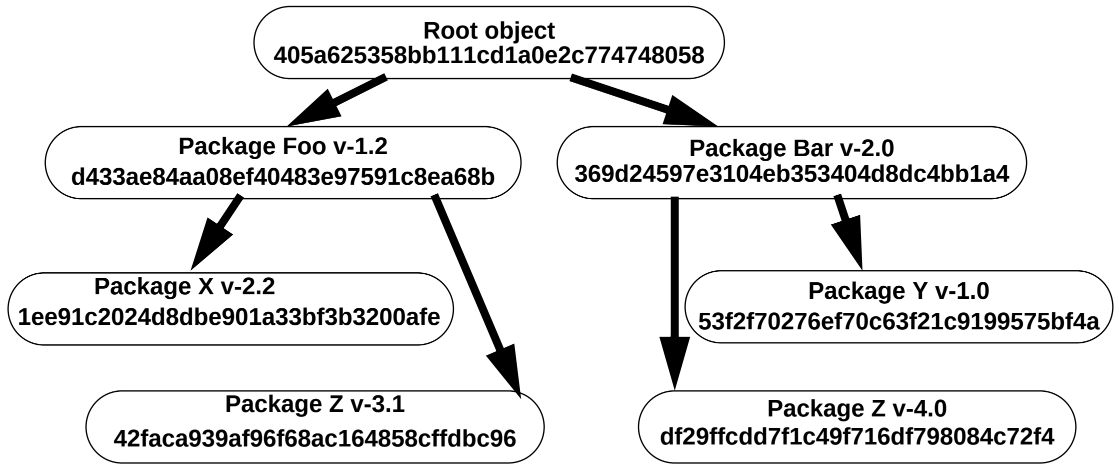
Root object
405a625358bb111cd1a0e2c774748058
Package Foo v-1.2
d433ae84aa08ef40483e97591c8ea68b
Package Bar v-2.0
369d24597e3104eb353404d8dc4bb1a4
Package X v-2.2
1ee91c2024d8dbe901a33bf3b3200afe
Package Y v-1.0
53f2f70276ef70c63f21c9199575bf4a
Package Z v-3.1
42faca939af96f68ac164858cffdbc96
Package Z v-4.0
df29ffcdd7f1c49f716df798084c72f4

Figure 2. Packages converted to `tcdn` format. Note that each package has a unique 32 character name.

wise precaution today), she can check the cryptographic hash of a downloaded object against the name she provided. If they do not match, the object is faulty.

3.2 What the Developer Sees

A developer need not radically change the way she writes code to benefit from tcdn. Instead, she needs to follow a few simple rules. First, 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 version 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.

The second restriction on developers is that 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.

If the programmer follows the above guidelines, she may use the tools described in Section 4.1 to convert her code into tcdn packages, making them available over the Web.

3.3 What the User Sees

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, he simply requests 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.

4 Tcdn Details

The `tcdn` package includes two pieces: code run by an application developer to generate the content-derived names and rewrite packages, and runtime functions necessary to locate and load packages named by content.

4.1 Developing Code for `tcdn`

Packages to be turned into tcdn packages are 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:

- All `package require` statements must be placed in the appropriate namespace.
- Each package must be contained in a single file. As a result, each file must have a `package provide` statement.
- There cannot be any circular dependencies between packages.
- All package code should be enclosed in a namespace with a unique name. This can be done by appending the version number to the "original" namespace name. If this is not done, most of the `tcdn` functionality will still be available, including the ability to fetch missing packages from a remote server. However, a single program will not be able to simultaneously use different versions of a particular 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 `pkg_mkIndex`. The routine to perform name conversion is called as follows:

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

This call operates on 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, and this option is necessary if the destination will not have Web access. A much 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 your Web server or any other Web server that has a copy of the file. The user is assured of receiving the correct file because her computer can compute an MD5 hash over the downloaded file; only if the file matches is it used.

4.2 The `tcdnify` Process

Packages in `tcdn` are named using a secure hash run over the entire body of the package. This name is then embedded into all files that require the package.

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

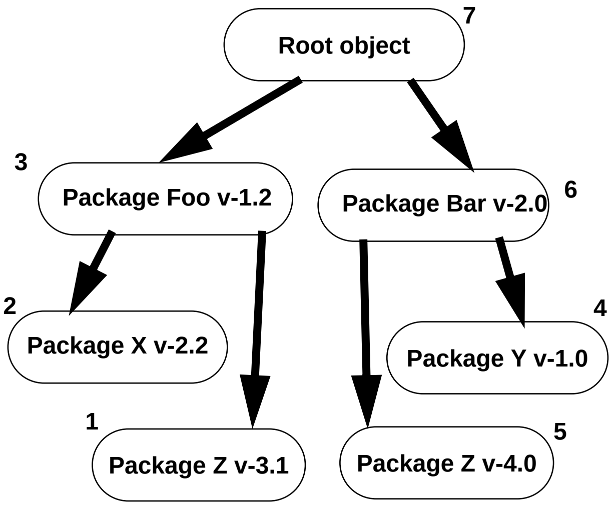

Figure 3. Sample package dependency graph.

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.

4.3 File System Independence

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. Of course, the efficiency of a file system may drop when handling directories with potentially thousands of files, but this problem has been solved in the SGI XFS file system [7] and elsewhere. This arrangement eliminates the need for users to specify information about where the software packages will reside, and makes it simpler for a designer to test the software because she no longer has to worry about users with different package search paths. The authors have had difficulties with Tcl software that refuses to work until the ordering of a search path is changed; we believe that this approach to software distribution is flawed because it limits usage to those who are relatively good at software installation.

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 `tcdn` system has a single directory (or list of directories, if desired) in which it looks for components. This directory (list) is stored in `tcdn::tcdndirs`, but it is relatively immune to user error. Should the user specify an incorrect (but readable and writable) directory, `tcdn` will merely download “missing” packages, even if they are stored elsewhere in the local system.

Unfortunately, this approach does not work for software developers who need to be able to modify packages because it forces name changes when the contents of a package change. However, it *can* be used for developers who wish to use other packages unchanged, and works very well for the majority of users who simply want their application to work without the need for painful installation.

4.4 Locating Packages

Perhaps the best feature of the `tcdn` package is that it permits automatic downloading of missing packages. 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.

When tcdn attempts to load a package, it first looks in the directories specified by tcdndirs. If the package is not found there, it proceeds to query each of the URLs contained in the list variable `tcdn::tcdnservers`. This list is searched in order, so it is likely that a site may put its own package cache server first before the “home” site or more comprehensive, but more distant caches. Additionally, the application itself can append values to `tcdnservers`, 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 Tcl packages for use by many machines at the site. If the object is not available there, tcdn can go to either a public server with many packages (the equivalent of `sunsite`, perhaps) or to the developer’s site to download the object directly from its source.

The integrity of packages found on the Web is of utmost concern because it is far too easy to implement a Tcl trojan horse. Thus, tcdn checks the integrity of any downloaded package (and, optionally, *any* `tcdn` package including those found locally) by hashing it and comparing its hash to its name. Files that do not hash properly are simply discarded, though it would be a simple extension to add Tcl code to send mail to a system administrator noting that a “bad `tcdn` package file” was received, alerting her of potential dangers. Note that if an integrity check fails, the package is treated as if it were never there. Thus, tcdn can go on to other servers listed in `tcdnservers` and check them to find a good package. If no valid package is found locally or on any server, `tcdn` throws an error.

Another advantage of this scheme is that it is not strictly necessary to even be able to store the file in order to use it. Instead, Tcl can dynamically load in the downloaded file but never store it on disk. This approach is poor for machines with disks that can cache the file locally. However, it may be advantageous for Tcl interpreters with no persistent local storage, such as those that run inside a Web browser.

5 Using TCDN

This section describes the actual installation and usage of the `tcdn` package. Because the package is simple, and places relatively few limitations on its use, it should be straightforward to use it with existing code. However, it will work best if developers follow a few simple guidelines for writing packages.

5.1 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. Usually, a system wide installation will entail placing the package in directories where writing is restricted, much as with any other normal package. This is only required for the initial installation of `tcdn`. Since the `tcdn` package is itself a CDN-based package, all 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 run time by any newer version if one is available.

The initial installation will also require a small amount of setup. The most important (and so far only) step of this setup is deciding where the downloaded CDN-named packages will reside. The surprising answer here is that 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.

5.2 Creating a CDN-named Package

The `tcdn` library was designed so that it would be easy for new 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 be as unobtrusive as possible, 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. Here the programmer should just use `package require` 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.

Of course, 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. `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.

5.3 Distributing CDN-named Code

One of the primary design goals of `tcdn` was to make distribution much easier. The current package mechanism requires users to manually find and install packages as needed. `Tcdn` will automatically find and install CDN-named packages on demand. `Tcdn` can be configured with the location of CDN-named package servers. When a CDN-named package is requested and not found on the local system these servers are searched in turn. The package is then downloaded from the server and installed. This means that in order to distribute a package the programmer needs to upload it to a server or several servers. The programmer must also make public the content-derived name of the package. This name is all anyone else needs to know in order to be able to use the package. Application programmers do not even need to know the names of the servers on which the package has been stored. The end user doesn't need to know anything at all. Once the package has been uploaded and the content-derived name has been publicized, the entire process is automatic.

5.4 Using a CDN-named Package

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 plan to build similar functionality into the dynamic library loaders for Windows 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 applets [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.

7 Conclusions

This paper has presented a Tcl package, `tcdn`, that allows Tcl developers to create distributions of their code that have several advantages over current Tcl distribution 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 package will enable Tcl-based applications to reach a wider audience by simplifying the installation process as well as the upgrade process. 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 `tcdn` provides integrity checking and the ability to fetch missing packages from remote server sites, we believe it will be essential for developers who wish to make Tcl software available via the Web. By providing both integrity and ease of use, `tcdn` enables even novice users to run complex Tcl applications without the need for complex installations or the fear of trojan horse packages.

Code Availability

Further information about content-derived naming is available on the Web at:
`http://www.csee.umbc.edu/~elm/Projects/CDN/`.
This page contains references to other work on content-derived names as well as the Tcl source code and documentation for `tcdn`.

References

[1] *Secure Hash Standard*, FIPS-180-1, National Institute of Standards and Technologies, U.S. Department of Commerce, April 1995.

[2] J. Gosling, B. Joy, and G. Steele, *The Java Language Specification*, 1996 (Addison-Wesley).

[3] A. van der Hoek, R. S. Hall, D. Heimbiger, and A. L. Wolf, "Software Release Management," CU-CS-806-96, University of Colorado, August 1996.

[4] J. K. Hollingsworth and E. L. Miller, "Using Content-Derived Names for Configuration Management," 1997 Symposium on Software Reusability (SSR '97), Boston, MA, May 1997.

[5] J. W. Moore, "The Use of Encryption to Ensure the Integrity of Reusable Software Components," International Conference on Software Reuse, Rio de Janeiro, November 1994, pages 118-123.

[6] R. L. Rivest, "The MD5 Message-Digest Algorithm," RFC 1321, Network Working Group, April 1992.

[7] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck, "Scalability in the XFS File System," Proceedings of the Winter 1996 USENIX Conference (San Diego, CA), January 1996, pages 33-44.