Using Content-Derived Names for Configuration Management*

Jeffrey K. Hollingsworth
University of Maryland College Park
hollings@cs.umd.edu

Ethan L. Miller
University of Maryland Baltimore County
elm@cs.umbc.edu

Abstract

Configuration management of compiled software artifacts (programs, libraries, icons, etc.) is a growing problem as software reuse becomes more prevalent. For an application composed from reused libraries and modules to function correctly, all of the required files must be available and be the correct version. In this paper, we present a simple scheme to address this problem: content-derived names (CDNs). Computing an object's name automatically using digital signatures greatly eases the problem of disambiguating multiple versions of an object. By using content-derived names, developers can ensure that only those software components that have been tested together are permitted to run together.

1. Introduction

As software modularity and code reuse have evolved from academic concept to accepted practice, programs have changed from self-contained monolithic files to a complex collection of interrelated files. A typical application today might consist of a main executable file and several configuration files, and require tens of libraries to be installed. Although this evolution has many positive aspects including software reuse and reduced disk storage, it has greatly complicated the process of installing and configuring software packages. Instead of loading a single file, an application could require hundreds of files when all libraries and icons are included. For example, Microsoft Office Pro 95 consists four major applications (Word, Excel, Powerpoint, and Access) plus several smaller applications and totals over 1,200 files. Many of these files are shared by several of the applications. To make matters worse, previously installed software may use incompatible versions of shared files. As a result, installing one application can often cause previously installed applications to fail.

This paper addresses many of these concerns with a simple mechanism: providing a name for objects based solely on their content. A content-derived name is the same for any two objects with identical content, regardless of their location or origin. However, it is different for different versions of the same package or library. Since an object's name does not depend on a manually assigned identifier such as a file name, a program requiring a specific version of a library is guaranteed to get the correct version because different versions have different contents and thus different names.

We first describe previous work on object code configuration management and some of the theory underlying digital signatures. We then describe how content-derived names provide benefits for software configuration management. We conclude with a look at the implications of using content-derived names, and some possible directions for future work.

* This work support in part by DOE grant DE-FG02-93ER25176, NASA grant NAG 2-1094, NIST CRA award 70-NANB-5H0055, and a grant from the Department of Defense.

2. Background

While the idea of using content-derived names (CDNs) for configuration management of object code is a new one, there has been previous work in the areas of using explicitly managed version numbers to provide configuration management. This paper builds on that work, as well as previous work in digital signatures.

2.1 Configuration Management

Most of the research in configuration management has concentrated in the area of managing the building of software objects from a source code repository. If a build produces a single, monolithic object (i.e., an executable program), then configuration management stops at this point and the object can be distributed[1]. However, if instead a family of related objects is produced (i.e., several communicating programs or a program plus separately stored libraries), then configuration management needs to be continued through the installation of the software onto each end-user's computer. To date, little research has addressed this second aspect of configuration management. One notable exception is van der Hoek et al.[8]. They address the related problem of "software release management" by proposing a system to support software acquisition and to ensure that the correct versions of dependent packages are acquired with the primary package. Their approach relies on a centralized software repository and explicit admini-

[1] We might archive configuration information to document the pedigree of the released artifact.

stration of version numbers for all software packages. In contrast, our approach is completely decentralized and permits anyone to release a new software package.

Explicit specification of software components based on filename and version number is also used in the UNIX operating system for dynamically linking shared libraries. This scheme permits compatible and incompatible changes to libraries by assigning a major and minor version number to each file. Files with different major version numbers are assumed to be incompatible. Files with the same major version number and different minor version numbers are assumed to be compatible. The decision of whether a change to a library is a "compatible" or "incompatible" is the responsibility of the library supplier. No provision is made to let application developers specifically indicate if a particular library instance is compatible with their application or not. As a result, a software vendor can not ensure that their product is being used with a known (and tested) configuration.

2.2 Secure Hash Functions

A key feature of CDNs is the use of a secure hash functions to assign a unique name to an object based on its content. Digital signature algorithms such as MD5[6] and SHA-1[1] are one-way functions that take an arbitrary sequence of bytes and produce a result that is likely to be different from that of any other (different) input sequence. MD5 is well suited to generate content-derived names. The MD5 algorithm produces a 128 bit signature, and Rivest[6] claims that it is NP-hard to find another document with an identical signature. Touch[7] has reported that it is possible to compute MD5 in software at the rate of over 10 MB/second on current RISC workstations; we feel this rate is more than adequate for our proposed use of MD5.

In order for a content-derived name system to work, it must probabilistically guarantee that two different objects will not share the same object name (i.e., the probability of a hash collision must be sufficiently small). To trust a probabilistic guarantee, we must ensure that the probability of failure is sufficiently small compared to other probabilistic guarantees already built into computer systems such as an undetected parity error in a disk read request. The space of all compiled software artifacts is very large, potentially billions of objects. Fortunately, even in this large space, the probability of such a failure is small. The probability that m numbers chosen randomly from a pool of n will be unique is $e^{-m(m-1)/2n}$ [4], where $n = 2^{128}$ for MD5. For 10^{15} objects, the probability of success (no two objects with different content have the same name) is $e^{-2^{-29}}$ assuming that object names (hash values) are uniformly distributed (which MD-5 ensures). Since $1-e^{-x} \approx x$ for small x, the chance of failure is approximately 2^{-29}, or 10^{-9}. We believe this chance of failure is sufficiently low because it is below the probability that there would be an undetectable failure in a disk or network link during that time. Thus, undetectable hardware failure is more likely to cause the use of an "incorrect" object than content-derived naming.

If a larger number of objects is required or lower probability of failure is needed, secure hash functions can still be used. There is no theoretical limit to the length of a digital signature. While MD5 produces 128 bit signatures, a similar algorithm could be constructed to produce a signature with 256 bits, allowing the creation of 10^{30} unique objects with the chance of collision dropping to below 10^{-17}. This is sufficient to allow each of ten billion computers to create ten million unique objects per second for over three hundred years.

3. Using CDNs for Configuration Management

The current trend in creating software is to reuse components such as classes, dynamic libraries, icons, and sound bites. For technical (size of the objects) or legal (different component vendors) reasons, different objects are stored as separate files. For a software product to work correctly, however, the different components must be compatible with each other. A single package may involve hundreds of individual files, each of which must be the right file, and be installed in the right place in the directory structure. An incorrect version of a particular library or configuration file, or even the right file in the wrong place, can render the entire package useless. This situation is complicated by the evolution of software and the interdependence of software packages. A single computer often has dozens of software packages, some of which may require different versions of the same software library. Maintaining such systems is difficult at best, and installing new software is often a challenge.

The popularity of personal computers and the WWW has further complicated software configuration management by facilitating the distribution of software over the Internet. No longer is the installer a computer expert; instead, complex systems must be "installed" by less-experienced users. Automatic install programs provide some assistance in this regard, but ensuring compatibility with previously installed applications is not currently supported. Also, languages such as Java[2] allow users to pull classes from many different locations, yet there is no guarantee that the files obtained in this way will actually work together. Some files may not work with the latest version of a Java class, instead requiring an older version. How can the software publisher specify a particular version of a Java class or similar object?

3.1 Ensuring Version Consistency

Currently, names, or names combined with a version string, are used to identify external components (such as dynamically linked library). However, using names as the basis of compatibility is problematic. Software quality assurance requires that product be tested with all compatible components prior to shipping. As a result, many products include copies of the tested components as part of their distribution. When the product is installed, the included components are copied onto the target machine with a name assigned by the developer. This ensures that the last installed product will have the correct components. However, any previously installed software may now break because it may rely on older (or newer) versions of components that have been replaced by more recently installed software.

Using digital signatures provides a good solution to finding consistent versions of objects. Each library, icon, or sound bite will have an object identifier computed using a digital signature. This signature uniquely identifies an object solely based on its content. We term this digital signature a content-derived name (CDN) since it can be used to fully specify a requested object. When an application (or a library) wants to reference or load an external component, it simply specifies the CDN for the desired object. A library (or perhaps the file system) then locates the requested object and loads it. Since different versions of the same software component will have different CDNs, each application will get the desired version.

At software installation, each component included with the distribution is loaded only if its CDN is not already installed. On the other hand, a new version of a software package may leave unmodified many of the files that it uses. These objects will retain the same object identifiers as the older version of the software, allowing the user to load only the files that have changed since the last version was released. Likewise, objects shared by several applications need only be installed once. Objects can refer to other objects by their CDNs forming a graph of object dependencies. By recursively traversing this graph, it is possible to ensure that all required components for an application are installed.

Traditionally, shared objects have been distributed in relatively large units (libraries) because maintaining consistent versions was so difficult. However, the use of content-derived naming allows the sharing of objects at a much finer granularity since the verification of consistent versions can be automated. This will (hopefully) encourage fine grained object code reuse.

Sometimes it is possible for an application to be able to use more than one version of a library. For example, an application might be compatible and have been tested with either of two similar releases of an object library. This case can easily be accommodated by lists of equivalent CDNs. As long as one of the objects specified in the equivalence list is present, the installation process does not need to load an object.

It is also possible that an application could be customized during installation or by the user at a latter time. For example, users might add custom macros to their word processing system. Customizations could be applied to either the application or to individual objects. However, customizations could potentially change the content of an object (and thus its CDN). To accommodate this situation, each object should contain a customization region containing fields that can be changed. This part of the object would not be used in computing its CDN.

The overall structure of an object in our scheme is shown in Figure 1. 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.

3.2 File Hierarchy Independence

A second benefit from content-derived naming is file system location independence. Many packages require extensive per-site customization to tell the software where to find files it needs. With content-derived naming, there is no need to explicitly enumerate the location of desired files. A package that refers to an object using its CDN only needs to look it up in a database of signatures and objects. Objects can still be assigned human-usable names; however such names are not required.

A file system to store objects based on their CDN could be built. Such a file system would support efficient storage and linkage of CDN-based objects. In addition to objects referring to other objects, a user-visible namespace could be provided similar to the way a UNIX directory structure provides a user access to numbered files (inodes). Objects would not be explicitly deleted from such a filesystem, but would be implicitly deallocated. Thus users would not have to worry about deleting files used by old versions of software, the filesystem can do this automatically. To implement implicit deletion, each stored object would have a reference count. When a reference count was decreased to zero, the object would be deleted. Due to the possibility of mutually referential objects creating unreachable cycles, a periodic garbage collection of the object space will also be required.

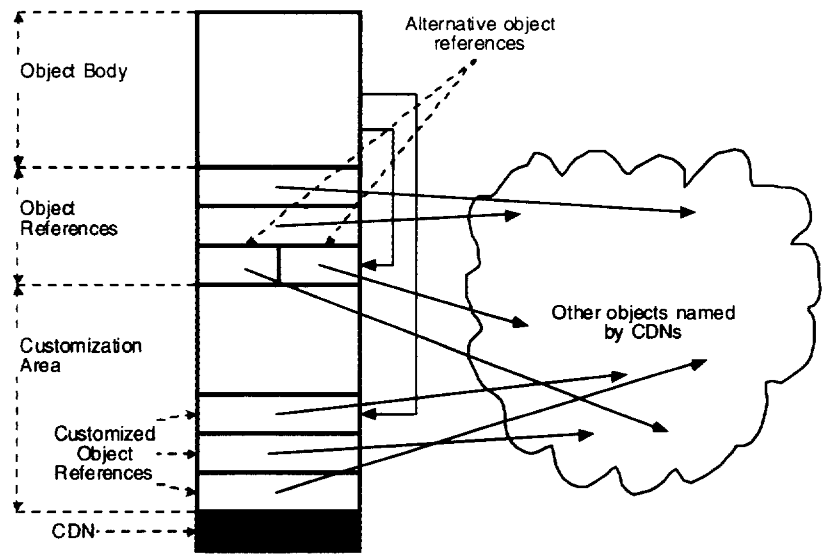

Alternative object references
Object Body
Object References
Customization Area
Customized Object References
CDN
Other objects named by CDNs

Figure 1: Layout for an object using CDNs.

The ability to locate objects by an identifier based solely on their content could simplify software distribution via the Internet. Already, programs such as Netscape use a single directory to cache retrieved network objects. Currently, the identifiers for these objects in the cache bear no relation to the objects' contents. Using content-derived names instead would allow the "cache" to grow to the size of the entire disk. At that point, hierarchical directory structure could be used as an overlay, providing a user-friendly interface to a cache of objects fetched from the Internet. Any piece of software could be distributed in this way; the user could assign names to those pieces of software that they wished to access directly, such as the main executable for a word processing program. Other objects would be fetched as necessary. To allow disconnected operation (e.g., when a laptop computer is being used on the road), a package might arrive with a self-installer that contains all files necessary to run the program.

3.3 Example of Content-Derived Naming

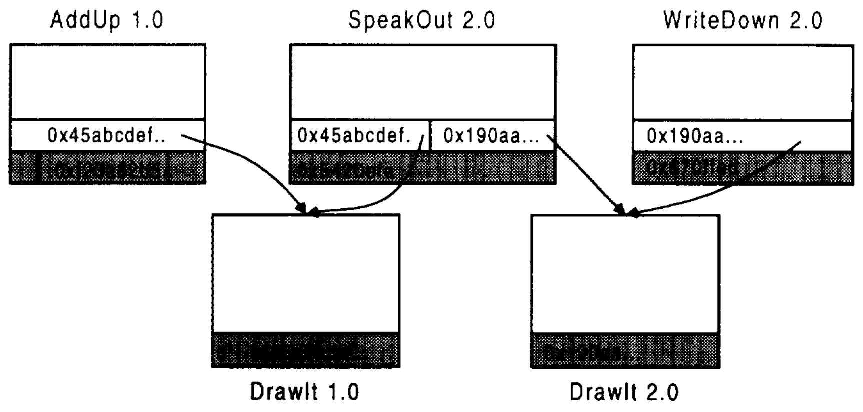
In this section we provide a concrete example of how content-derived naming can be used to manage multiple versions of a library on a single machine. Consider three applications a word processor (writeDown), a spreadsheet (addUp), and a presentation tool (speakOut). Each of these is a large package the contains many parts, and they all use a common set of routines for creating illustrations (drawIt). All three applications and the drawing library are supplied by different software vendors. When a new version of drawIt is released, the publishers of the applications test their application with the new version (to ensure the two packages still work together) and then as part of their next software release distribute a copy of the new version of drawIt.

WriteDown version 1.0 works with drawIt version 1.0, but writeDown 2.0 requires drawIt version 2.0. AddUp versions before 3.0 require drawIt version 1.0. Also, speakOut version 1.0 requires drawIt 1.0, but speakOut 2.0 can use drawIt 1.0 or 2.0. Figure 2 shows the content names and objects for this example. In the currently shown state, speakOut and writeDown have been upgraded to version 2.0. However, since addUp is still version 1.0, both versions of the drawIt library are installed. When addUp 3.0 is installed, drawIt 1.0 could be deleted because it is no longer required to run any installed application.

3.4 Handling Software Updates

Distributing new versions of software that fixes bugs in previous versions is currently a Herculean task, requiring software authors to ship entire new releases to users. This task has been shrunk somewhat by using patch programs that only modify the "broken" pieces of code, but updaters must still update **every** piece of software that needs to be fixed. In modern systems, however, individual programs share many libraries; when one is fixed, each software supplier must provide a patch that checks to see if a buggy library has been updated to the latest version.

Using CDNs, however, provides a simple solution to the problem of proliferating patches to buggy libraries. Rather than have each individual software provider patch the necessary libraries, the provider can simply supply a new "root" object that specifies the use of the updated libraries. Section 3.1 notes that each application has a single object that serves as the root of an object reference graph in which links are specified by CDNs, as shown Figure 3. Merely changing the root can cause the creation of a whole new graph because high-level objects don't specify the names of **all** of the objects in the graph - only those they



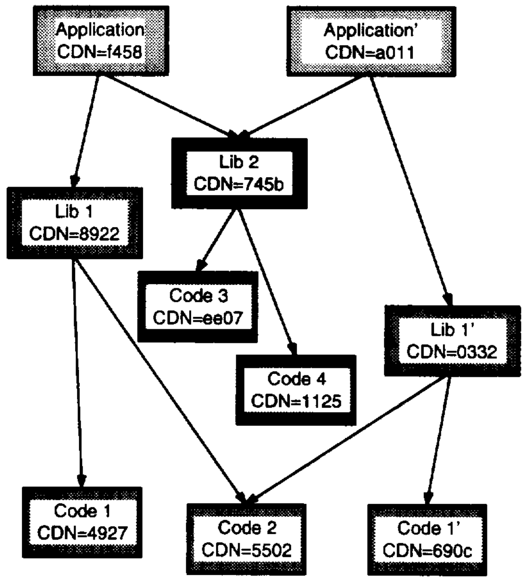
The example shows three applications and two versions of the shared DrawIt library. For each object, the last item (in gray) is its content-derived name.

Figure 2: Example of Using Content Naming.

access directly. Thus, an update to the root object may cause it to reference the updated version of a library, which in turn references updated objects, and so on through the rest of the graph. Distribution of software updates is simple because the process is started by distributing just a single updated object with references to more recent objects. Once this is done, the system automatically loads updated libraries and code as specified by CDN references from higher in the graph. As with code being loaded for the first time, the new version of the application must know where to find new version of objects it uses; however, this is no different from finding them for a "brand new" object and can use the same mechanism.

4. Related Work

Digital signatures combined with authentication services have been proposed by Moore[3]. His approach permits verifying the integrity and source of a particular component. In contrast, our system permits verifying that a reference to a software component is the desired one. Since our scheme is based solely on a hash function, we do not need to use an authentication server to obtain author, and public key combinations. In addition, our approach requires only a hash function rather than a hash function followed by asymmetric encryption. However, we don't provide verification of the author's identity, but rather verify that the component is the one that the creator of the referring component intended. Our two approaches are compatible and complementary. Moore's authentication could be used to verify the top-level component (i.e., the based application), and our scheme could then be used to verify version and identity of the separate components (i.e., libraries, icons, fonts, etc.). Moore's approach would need to be applied once when an object is loaded onto the system; CDN are used every time the application is invoked to locate and bind the correct component and version of that component.



Note that when a library developer fixes a bug in code object 1 in library 1, an application developer need only update the reference to library 1 in the application; they need not know which objects in library 1 were actually changed.

Figure 3: Object reference graph for CNDs.

In some ways, a CDN-based file system would be similar to the PILOT file system used on the Altos [5]. In the Pilot file system, all stored objects had a globally unique object identifier. However, this object identifier was created by concatenating the host identifier of the server where the object was created and a server relative identifier. The Pilot file system also supported linking files together based on their object identifiers. However, a program using the file system needed to explicitly mark an object immutable before its object identifier was frozen. With a CDN, no explicit designation is required.

5. Future Directions

Mobile computing can also benefit from CDNs. One of the problems in mobile computing is the ability to operate a computer away from its "base," since requests for objects must eventually be sent through the network to the mobile computer's home file server. CDNs present an attractive alternative: a mobile computer can merely ask the local file servers for an object using its CDN. The mobile computer need not follow the same file pathname conventions as the local server, since the object is identified solely by its content. Moreover, the mobile computer can check that it received the object it requested by computing the digital signature on the object, so it need not even trust the local server.

Eventually, if network-based software distribution replaces physical distribution, most of the disk space on client computers could be turned into a cache for network objects. When a commercial software package is purchased, a request for an object would fetch it from the Internet. There would be no need to explicitly delete old software or for garbage collection; an object would simply be removed from the cache to make room for new objects. If that discarded object were needed again, it could be re-fetched. A significant amount of research on caching strategies and other issues is necessary before this goal can become a reality, but its implementation would greatly simplify the operation of computers in an environment where access to the Internet is constant and omnipresent.

6. Conclusion

In this paper we have presented a new approach to object code configuration and naming. Rather than using user-assigned names to identify objects, we proposed to derive object names automatically based on the content of an object. Our scheme makes it possible to identify references to the same object even if the objects have completely different names and to differentiate variations of the same object. Our approach is helpful for standalone systems with physical media distribution, and for connected systems using network-based software distribution.

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. W. Moore, "The Use of Encryption to Ensure the Inbtegrity of Reusable Software Components", *International Conference on Software Reuse*. Nov. 1994, Rio de Janeiro, pp. 118-123.
4. R. Motwani and P. Raghavan, *Randomized Algorithms*. 1995: Cambridge University Press.
5. D. D. Redell, Y. K. Dalal, T. R. Horsley, H. C. Lauer, W. C. Lynch, P. R. McJones, H. G. Murray, and S. C. Purcell, "Pilot: An Operating System for a Personal Computer", *Communications of the ACM*, Feb 1980. **23**(2), pp. 81-92.
6. R. L. Rivest, *The MD5 Message-Digest Algorithm*, RFC 1321, Network Working Group, April 1992.
7. J. D. Touch, "Performance Analysis of MD5", *SIGCOMM*. Aug 1995, Cambridge, MA, pp. 77-86.
8. A. van der Hoek, R. S. Hall, D. Heimbiger, and A. L. Wolf, *Software Release Management*, CU-CS-806-96, University of Colorado, Aug. 1996.