CHAPTER

The Design of Naming Schemes

3

CHAPTER CONTENTS

OVERVIEW

In the previous chapter we developed an abstract model of naming schemes. When the time comes to design a practical naming scheme, many engineering considerations—constraints, additional requirements or desiderata, and environmental pressures—shape

Principles of Computer System Design: An Introduction

DOI: 10.1016/B978-0-12-374957-4.00012-8

the design. One of the main ways in which users interact with a computer system is through names, and the quality of the user experience can be greatly influenced by the quality of the system's naming schemes. Similarly, since names are the glue that connects modules, the properties of the naming schemes can significantly affect the impact of modularity on a system.

This chapter explores the engineering considerations involved in designing naming schemes. The main text introduces a wide range of naming considerations that affect modularity and usability. A case study of the World Wide Web Uniform Resource Locator (URL) illustrates both the naming model and some problems that arise in the design of naming schemes. Finally, a war stories section explores some pathological problems of real naming schemes.

3.1 CONSIDERATIONS IN THE DESIGN OF NAMING SCHEMES

We begin with a discussion of an interaction between naming and modularity.

3.1.1 Modular Sharing

Connecting modules by name provides great flexibility, but it introduces a hazard: the designer sometimes has to deal with preexisting names, perhaps chosen by someone else over whom the designer has no control. This hazard can arise whenever modules are designed independently. If, in order to use a module, the designer must know about and avoid the names used within that module for its components, we have failed to achieve one of the primary goals of modularity, called *modular sharing*. Modular sharing means that one can use a shared module by name without knowing the names of the modules it uses.

Lack of modular sharing shows up in the form of *name conflict*, in which for some reason two or more different values compete for the binding of the same name in the same context. Name conflict can arise when integrating two (or more) independently conceived sets of programs, sets of documents, file systems, databases, or indeed any collection of components that use the same naming scheme for internal interconnection as for integration. Name conflict can be a serious problem because fixing it requires changing some of the uses of the conflicting names. Making such changes can be awkward or difficult, for the authors of the original subsystems are not necessarily available to help locate, understand, and change the uses of the conflicting names.

The obvious way to implement modular sharing is to provide each subsystem with its own naming context, and then work out some method of cross-reference between the contexts. Getting the cross-reference to work properly turns out to be the challenge.

Consider, for example, the two sets of programs shown in Figure 3.1—a word processor and a spelling checker—each of which comprises modules linked by name and each of which has a component named INITIALIZE. The designer of the procedure WORD_PROCESSOR wants to use SPELL_CHECK as a component. If the designer tries to combine the two sets of programs by simply binding all of their names in one naming

context, as in the figure (where the arrows show the binding of each name), there are two modules competing for binding of the name INITIALIZE. We have a name conflict.

So the designer instead tries to create a separate context for each set of programs, as in Figure 3.2. That step by itself doesn't completely address the problem because the program interpreter now needs some rule to determine which context to use

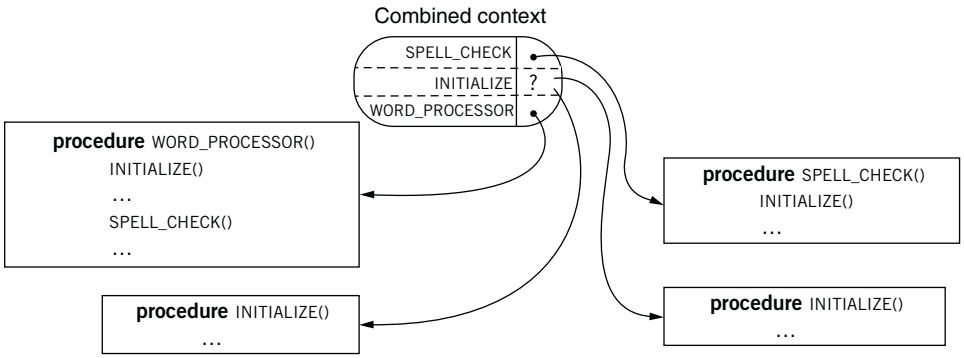

FIGURE 3.1

Too-simple integration of two independently written sets of programs by just merging their contexts. Procedure WORD_PROCESSOR calls SPELL_CHECK, but SPELL_CHECK has a component that has the same name as a component of WORD_PROCESSOR. No single set of bindings can do the right thing.

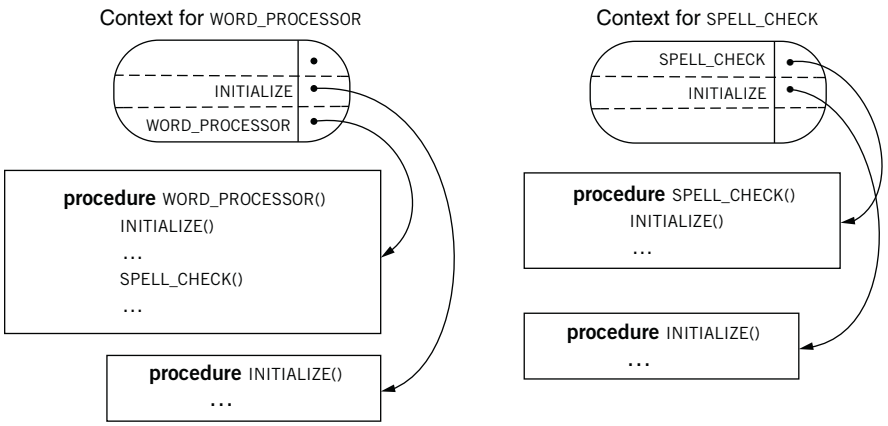

FIGURE 3.2

Integration of the same two programs but using separate contexts. Having a separate context for SPELL_CHECK eliminates the name conflict, but the program interpreter now needs some basis for choosing one context over the other.

for each use of a name. Suppose, for example, it is running WORD_PROCESSOR, and it encounters the name INITIALIZE. How does it know that it should resolve this name in the context of WORD_PROCESSOR rather than the context of SPELL_CHECK?

Following the naming model of Chapter 2, and the example of the e-mail system, a direct solution to this problem would be to add a binding for SPELL_CHECK in the WORD_PROCESSOR context and attach to every module an explicit context reference, as in Figure 3.3. This addition would require tinkering with the representation of the modules, an alternative that may not be convenient or even not allowed if some of the modules belong to someone else.

Figure 3.4 suggests another possibility: augment the program interpreter to keep track of the context in which it originally found each program. The program interpreter would use that context for resolving all names found in that program. Then, to allow the word processor to call the spell checker by name, place a binding for SPELL_CHECK in the WORD_PROCESSOR context as shown by the solid arrow numbered *1* in that figure. (imagine that the contexts are now file system directories).

That extra binding creates a subtle problem that may produce a later surprise. Because the program interpreter found SPELL_CHECK in the word processor's context, its context selection rule tells it (incorrectly) to use that context for the names it finds inside of SPELL_CHECK, so SPELL_CHECK will call the wrong version of INITIALIZE. A solution is to place an indirect name (the dashed arrow numbered *2* in Figure 3.2) in the word processor's context, bound to the name of SPELL_CHECK in SPELL_CHECK's own context. Then, the interpreter (assuming it keeps track of the context where it actually found each program) will correctly resolve names found in both groups of programs.

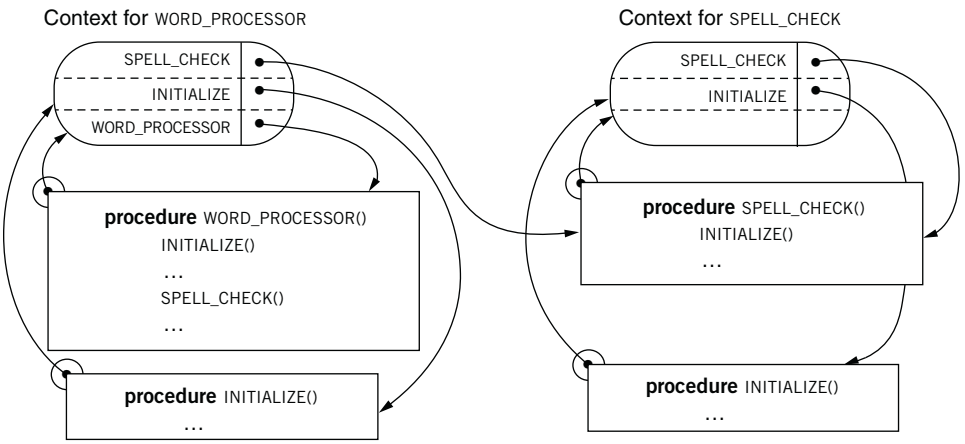

FIGURE 3.3

Modular sharing with explicit context references. The small circles added to each program module are context references that tell the name interpreter which context to use for names found in that module.

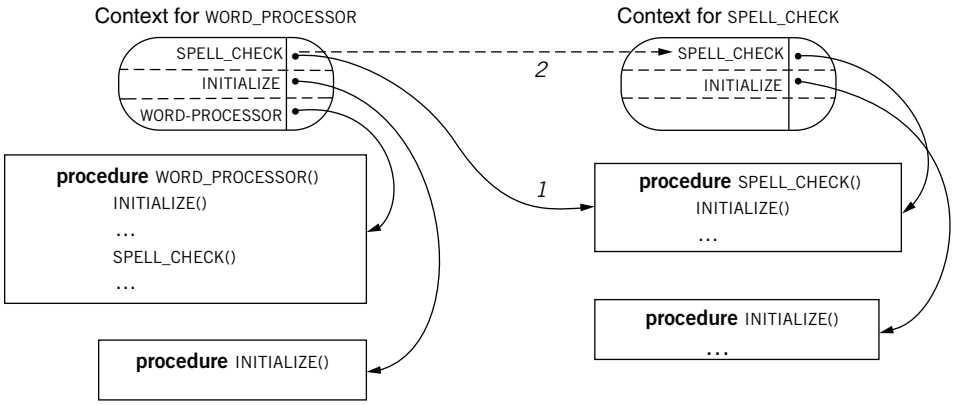

FIGURE 3.4

Integration with the help of separate contexts. Having a separate context for spell-check eliminates the name conflict, but the program interpreter still needs some basis for choosing one context over the other. Adding the solid arrow numbered *1* doesn't quite work, but the dashed arrow numbered *2*, an indirect name, does.

Keeping track of contexts and using indirect references (perhaps by using file system directories as contexts) is commonplace, but it is a bit *ad hoc*. Another, more graceful, way of attaching a context reference to an object without modifying its representation is to associate the name of an object not directly with the object itself but instead with a structure that consists of the original object plus its context reference. Some programming languages implement just such a structure for procedure definitions, known as a "closure", which connects each procedure definition with the naming context in which it was defined. Programming languages that use static scope and closures provide a much more systematic scheme for modular sharing of named objects within the different parts of a large application program, but comparable mechanisms are rarely found* in file systems or in merging applications such as the word processing and spell-checking systems of the previous example. One reason for the difference is that a program usually contains many references to lots of named objects, so it is important to be well organized. On the other hand, merging applications involves a small number of large components with only a few cross-references, so *ad hoc* schemes for modular sharing may seem to suffice.

*An ambitious attempt to design a naming architecture with all of these concepts wired into the hardware was undertaken by IBM in the 1970s, documented in a technical report by George Radin and Peter R. Schneider: *An architecture for an extended machine with protected addressing*, IBM Poughkeepsie Laboratory Technical Report TR 00.2757, May, 1976. Although the architecture itself never made it to the market, some of the ideas later appeared in the IBM System/38 and AS/400 computer systems.

3.1.2 Metadata and Name Overloading

The name of an object and the context reference that should be associated with it are two examples of a class of information called *metadata*—information that is useful to know about an object but that cannot be found inside the object itself (or if it is inside may not be easy to find). A library bibliographic record is a collection of metadata, including title, author, publisher, publication date, date of acquisition, and shelf location of a book, all in a standard format. Libraries have a lot of experience in dealing with metadata, but failure to systematically organize metadata is a design shortcoming frequently encountered in computer systems.

Some common examples of metadata associated with an object in a computer system are a user-friendly name, a unique identifier, the type of the object (executable program, word processing text, video stream, etc.), the dates it was created, last modified, and last backed up, the location of backup copies, the name of its owner, the program that created it, a cryptographic quality checksum (known as a *witness*—see Sidebar 7.1 [on-line]) to verify its integrity, the list of names of who is permitted to read or update the object, and the physical location of the representation of the object. A common, though not universal, property of metadata is that it is information about an object that may be changed without changing the object itself.

One strategy for maintaining metadata in a file system is to reserve storage for the metadata in the same file system structure that keeps track of the physical location of the file and to provide methods for reading and updating the metadata. This strategy is attractive because it allows applications that do not care about the metadata to easily ignore it. Thus, a compiler can read an input file without having to explicitly identify and ignore the file owner's name or the date on which the file was last backed up, whereas an automatic backup application can use the metadata access method to check those two fields. The UNIX file system, described in Section 2.5.1, uses this strategy by storing metadata in inodes.

Computer file systems nearly always provide for management of specialized metadata about each file such as its physical location, size, and access permissions, but they rarely have any provision for user-supplied metadata other than the file name. Because of this limitation, it is common to discover that file names are *overloaded* with metadata that has little or nothing to do with the use of the name as a reference.* The naming scheme may even impose syntax rules on allowable names to support overloading with metadata. A typical example of name overloading is a file name that ends with an extension that identifies the type of the file, such as text, word processing document, spreadsheet, binary application program, or movie. Other examples are illustrated in Figure 3.5. A physical address is another example of name overloading that is so common that the next section explores its special properties. Names that have no overloading whatever are known as *pure names*. The only operations it makes sense to apply to a pure name are COMPARE, RESOLVE, BIND, and UNBIND; one cannot

*Use of the word "overloading" to describe names that carry metadata is similar to, but distinct from, the use of the same word to describe symbols that stand for several different operators in a programming language.

Name	Some of the things that overload this name
solutions.txt	solutions = file content; txt = file format
solutions.txt.backup 2	backup 2 = this is the second backup copy
businessplan 10-26-2007.doc	10-26-2007 = when file was created
executive summary v4	v4 = version number
image079.large.jpg	079 = where file fits in a sequence; large = image size
/disk-07/archives/Alice/	disk-07 = physical device that holds file; Alice = user id
OSX.10.5.2.dmg	OSX = program name; 10.5.2 = program version
IPCC_report_TR-4	IPCC = author; TR-4 = technical report series identifier
cse.pedantic.edu	cse = department name; pedantic = university name; edu = registrar name
ax539&ttiejh!90rrwl	no (apparent) overloading

FIGURE 3.5

Some examples of overloaded names and a pure name.

extract metadata from it by applying a parsing operation. An overloaded name, on the other hand, can be used in two distinct ways:

1. As an identifier, using COMPARE, RESOLVE, BIND, and UNBIND.
2. As a source from which to extract the overloaded metadata.

Path names are especially susceptible to overloading. Because they describe a path through a series of contexts, the temptation is to overload them with information about the route to the physical location of the object.

Overloading of a name can be harmless, but it can also lead to violation of the principles of modular design and abstraction. The problem usually shows up in the form of a *fragile name*. Name fragility appears, for example, when it is necessary to change the name of a file that moves to a new physical location, even though the identity and content of the file have not changed. For example, suppose that a library program that calculates square roots and that happens to be stored on `disk05` is named `/disk05/library/sqrt`. If `disk05` later becomes too full and that library has to be moved to `disk06`, the path name of the program changes to `/disk06/library/sqrt`, and someone has to track down and modify every use of the old name. Name fragility is one of the reasons that World Wide Web addresses stop working. The case study in Section 3.2 explores that problem in more detail.

The general version of this observation is that overloading creates a tension between the goal of keeping names unchanged and the need to modify the overloaded information. Typically, a module that uses a name needs the name to remain unchanged for at least as long as that module exists. For this reason, overloading must be used with caution and with understanding of how the name will be used.

Finally, in a modular system, an overloaded name may be passed through several modules before reaching the module that actually knows how to interpret the overloading. A name is said to be *opaque* to a module if the name has no overloading

that the module knows how to interpret. A pure name can be thought of as being opaque to all modules except RESOLVE.

There are also more subtle forms of metadata overloading. Overloading can be less obvious if the user's mind, rather than the computer system, performs the metadata extraction. For example, in the Internet host name "CityClerk.Reston.VA.US", the identifier of the context, "Reston.VA.US", is also recognizable as the identifier of a real place, a town named Reston, Virginia, in the United States. Each component of this name is being used to name two different real-world things: the name "Reston" identifies both a town and a table of name/value pairs that acts as a context in which the name of a municipal department may be looked up. Because it has mnemonic value, people find this reuse by overloading helpful—assuming that it is done accurately and consistently. (On the other hand, if someone names a World Wide Web service in Chicago "SaltLakeCity.net" people seeing that name are likely to assume—incorrectly—that it is actually located in Salt Lake City.)

3.1.3 Addresses: Names that Locate Objects

In a computer system, an *address* is the name of a physical location or of a virtual location that maps to a physical location. Computer systems are constructed of real physical objects, so they abound in examples of addresses: register numbers, physical and virtual memory addresses, processor numbers, disk sector numbers, removable media volume numbers, I/O channel numbers, communication link identifiers, network attachment point addresses, pixel positions on a display—the list seems endless.

Addresses are not pure names. The thing that characterizes an address is that it is overloaded in such a way that parsing the address provides a guide to the location of the named object in some virtual or real coordinate system. As with other overloaded names, addresses can be used in two ways, in this case:

1. As an identifier with the usual naming operations.
2. As a locator.

Thus, "Leonardo da Vinci" is an identifier that was once bound to a physical person and is now bound to the memory of that Leonardo. This identifier could have been used in comparisons to avoid confusion with Leonardo di Pisa when both of them were visiting Florence.* Today, the identifier helps avoid mixing up their writings. At the same time, "Leonardo da Vinci" is also a locator; it indicates that if you want to examine the birth record of that Leonardo, you should look in the archives of the town named Vinci.

Since access to many physical devices is geometric, addresses are often chosen from compact sets of integers in such a way that address adjacency corresponds to physical adjacency, and arithmetic operations such as "add 1" or subtracting one

*Actually, they could not have both visited Florence at the same time. The mathematician Leonardo di Pisa (also known as Fibonacci) lived three centuries before the artist Leonardo da Vinci.

address from another have a useful, physical meaning. For example, a seek arm finds track #1079 on a magnetic disk by counting the number of tracks it passes, and a disk arm scheduler looks at differences in track addresses to decide the best order in which to perform seeks. For another example, a memory chip contains an array of bits, each of which has a unique integer address. When a read or write request for a particular address arrives at the chip, the chip routes individual bits of that address to selectors that guide the flow of information to and from the intended bit of storage.

Sometimes it is inappropriate to apply arithmetic operations to addresses, even when they are chosen from compact sets of integers. For example, telephone numbers (known technically as "directory numbers") are integers that are overloaded with routing information in their area and exchange codes, but there is no necessary physical adjacency of two area codes that have consecutive addresses. Similarly, there is no necessary physical adjacency of two telephones that have consecutive directory numbers. (In decades past, there was physical adjacency of consecutive directory numbers inside the telephone switching equipment, but that adjacency was so constraining that it was abandoned by introducing a layer of indirection as part of the telephone switch gear.)

The overloaded location information found in addresses can cause name fragility. When an object moves, its address, and thus its name, changes. For this reason, system designers usually follow the example of telephone switching systems: they apply the design principle *decouple modules with indirection* to hide addresses. Adding a layer of indirection provides a binding from some externally visible, but stable, name to an address that can easily be changed when the object moves to a new location. Ideally, addresses never need to be exposed above the layer of interpretation that directly manipulates the objects. Thus, for example, the user of a personal computer that has a communication port may be able to write programs using a name such as `COM1` for the port, rather than a hexadecimal address such as $4D7C_{hex}$, which may change to $4D7E_{hex}$ when the port card is replaced.

When a name must be changed because it is being used as an address that is not hidden by a layer of indirection, things become more complicated and they may start to go wrong. At least four alternatives have been used in naming schemes:

- Search for and change all uses of the old address. At best, this alternative is a nuisance. In a large or geographically distributed system, it can be quite painful. The search typically misses some uses of the name, and those users, on their next attempted use of the name, either receive a puzzling not-found response for an object that still exists or, worse, discover that the old address now leads to a different object. For that reason, this scheme may be combined with the next one.
- Plan that users of the name must undertake an attribute-based search for the object if they receive a not-found response or detect that the address has been rebound to a different object. If the search finds the correct object, its new address can replace the old one, at least for that user. A different user will have to do another search.

- If the naming scheme provides either synonyms or indirect names, add bindings so that both the old and new addresses continue to identify the object. If addresses are scarce and must be reused, this alternative is not attractive.
- If the name is bound to an active agent, such as a post office service that accepts mail, place an active intermediary, such as a mail forwarder, at the old address.

None of these alternatives may be attractive. The better method is nearly always for the designer to hide addresses behind a layer of indirection. Section 3.3.2 provides an example of this problem and the solution using indirection. Exercise 2.1 explores some interesting indirection-related naming problems in the telephone system related to the feature known as call forwarding.

One might suggest avoiding the name fragility problem by using only pure names, that is, names with no overloading. The trouble with that approach is that it makes it difficult to locate the object. When the lowest-layer name carries no overloaded addressing metadata, the only way to resolve that name to a physical object is by searching through an enumeration of all the names. If the context is small and local, that technique may be acceptable. If the context is universal and widely distributed, name resolution becomes quite problematic. Consider, for example, the problem of locating a railway car, given only a unique serial number painted on its side. If for some reason you know that the car is on a particular siding, searching may be straightforward, but if the car can be anywhere on the continent, searching is a daunting prospect.

3.1.4 Generating Unique Names

In a unique identifier name space, some protocol is needed to ensure that all of the names actually are unique. The usual approach is for the naming scheme to *generate* a name for a newly created object, rather than relying on the creator to propose a unique name. One simple scheme for generating unique names is to dole out consecutive integers or sufficiently fine timestamp values. Sidebar 3.1 shows an example. Another scheme for generating unique names is to choose them at random from a sufficiently large name space. The idea is to make the probability of accidentally choosing the same name twice (a form of name conflict called a *collision*) negligibly small. The trouble with this scheme is that it is hard for a finite-state machine to create genuine randomness, so the chance of accidentally creating a name collision may be much higher than one would predict from the size of the name space. One must apply careful design, for example, by using a high-quality pseudorandom number generator and seeding it with a unique input such as a timestamp that was created when the system started. An example of such a design is the naming system used inside the Apollo DOMAIN operating system, which provided unique identifiers for all objects across a local-area network to provide a high-degree of transparency to users of the system; for more detail, see Suggestions for Further Reading 3.2.1.

Yet another way to avoid generated name collisions, for an object that has a binary representation and that already exists when it is being named, is to choose as its unique name the contents of the object. This approach assigns two objects with the same content the same name. In some applications, however, that may be a feature—it provides

Sidebar 3.1 Generating a Unique Name from a Timestamp Some banking systems generate a unique character-string name for each transaction. A typical name generation scheme is to read a digital clock to obtain a timestamp and convert the timestamp to a character string. A typical timestamp might contain the number of microseconds since January 1, 2000. A 50-bit timestamp would repeat after about 35 years, which may be sufficient for the bank's purpose. Suppose the timestamp at 1:35 P.M. on April 1, 2007, is

00010111110110101101001100111001100010111010011001

To convert this string of bits to a character string, divide it into five-bit chunks and interpret each chunk as an index into a table of 32 alphanumeric characters. The five-bit chunks are:

00010–11111–01101–01101–00110–01110–01100–01011–10100–11001

Next, reinterpret the chunks as index numbers:

2 31 13 13 6 16 12 11 20 25

Then look those numbers up in this table of 32 alphanumeric characters:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
B	C	D	F	G	H	J	K	L	M	N	P	Q	R	S	T	U	V	W	X	Y	Z	1	2	3	4	5	6	7	8	9	0

The result is the 10-character unique name `"D9RRJ-UQTYP"`. You may have seen similar unique names in transactions performed with an on-line banking system.

a way of discovering the existence of unwanted duplicate copies. That name is likely to be fairly long, so a more practical approach is to use as the name a shorter version of its contents, known as a *hash*. For example, one might run the contents of a stored file through a cryptographic transformation function whose output is a bit string of modest, fixed length, and use that bit string as the name. One version of the Secure Hash Algorithm (SHA, described in Sidebar 11.8 [on-line]) produces, for any size of input, an output that is 160 bits in length. If the transforming function is of sufficiently high quality, two different files will almost certainly end up with different names.

The main problem with any naming scheme that is based on the contents of the named object is that the name is overloaded. When someone modifies an object whose name was constructed from its original contents, the question that arises is whether to change its name. This question does not come up in preservation storage systems that do not allow objects to be modified, so hash-generated unique names are sometimes used in those systems.

Unique identifiers and generated names can also be used in places other than unique identifier name spaces. For example, when a program needs a name for a temporary file, it may assign a generated name and place the file in the user's working directory. In this case, the design challenge for the name generator is to come up with an algorithm that will not collide with the names of already existing names chosen

by people or generated by other automated name generators. Section 3.3.1 gives an example of a system that failed to meet this challenge.

Providing unique names in a large, geographically distributed system requires careful design. One approach is to create a hierarchical naming scheme. This idea takes advantage of an important feature of hierarchy: delegation. For example, a goal of the Internet is to allow creation of several hundred million different, unique names in a universal name space for attachment points for computers. If one tried to meet that goal by having someone at the International Telecommunications Union coordinating name assignment, the immense number of name assignments would almost certainly lead to long delays as well as mistakes in the form of accidental name collisions. Instead, some central authority assigns the name "edu" or "uk" and delegates the responsibility for naming things ending with that suffix to someone else—in the case of "edu", a specialist in assigning university names. That specialist accepts requests from educational institutions and, for example, assigns the name "pedantic" and thereby delegates the responsibility for names ending with the suffix ".pedantic.edu" to the Pedantic University network staff. That staff assigns the name "cse" to the Computer Science and Engineering Department, further delegating responsibility for names ending with the suffix ".cse.pedantic.edu" to someone in that department. The network manager inside the department can, with the help of a list posted on the wall or a small on-line database, assign a name such as "ginger" that is locally unique and at the same time can be confident that the fully qualified name "ginger.cse.pedantic.edu" is also globally unique.

A different example of a unique identifier name space is the addressing plan for the commercial Ethernet. Every Ethernet interface has a unique 48-bit *media access control* (MAC) *address*, typically set into the hardware by the manufacturer. To allow this assignment to be made uniquely, but without a single central registry for the whole world, there is a shallow hierarchy of MAC addresses. A standards-setting authority allocates to each Ethernet interface manufacturer a block of MAC addresses, all of which start with the same prefix. The manufacturer is then free to allocate MAC addresses within that block in any way that is convenient. If a manufacturer uses up all the MAC addresses in a block, it applies to the central authority for another block, which may have a prefix that has no relation to the previous prefix used by that same manufacturer.

One consequence of this strategy, especially noticeable in a large network, is that the MAC address of an Ethernet interface does not provide any overloading information that is useful for physically locating the interface card. Even though the MAC address is assigned hierarchically, the hierarchy is used only to delegate and thus decentralize address assignment, and it has no assured relation to any property (such as the physical place where the card attaches to the network) that would help locate it. Just as in locating a railway car knowing only its unique identifier, resolving a MAC address to the particular physical device that carries it is difficult unless one already has a good idea where to start looking.

People struggling to figure out how to tie a software license to a particular computer sometimes propose to associate the license with the Ethernet MAC address of that computer because that address is globally unique. Apart from the problem that some computers have no Ethernet interface and others have more than one, a trouble

with this approach is that if an Ethernet interface card on the computer fails and needs to be replaced, the new card will have a different MAC address, even though the location of the system, the software, and its owner are unchanged. Furthermore, if the card that failed is later repaired and reinstalled in another system, that other system will now have the MAC address that was previously associated with the first system. The MAC address is thus properly viewed only as the unique name of a specific hardware component, not of the system in which it is embedded.

Deciding what constitutes the unique identity of a system that is constructed of replaceable components is ultimately a convention that requires an arbitrary choice by the designer of the naming scheme. This choice is similar to the question of establishing the identity of wooden ships. If, over the course of 300 years, every piece of wood in the ship has been replaced, is it still the same ship? Apparently, ship registries say "yes". They do not associate the name of the ship with any single component; the name is instead associated with the ship as a whole. Answering this identity question can clarify which of the three meanings of the COMPARE operation that was discussed in Section 2.2.5 is most appropriate for a particular design.

3.1.5 Intended Audience and User-Friendly Names

Some naming schemes are intended to be used by people. Names in such a name space are typically user-chosen and user-friendly strings of characters with mnemonic value such as "economics report", "shopping list", or "Joe.Smith" and are widely used as names of files and e-mailboxes. Ambiguity (that is, non-uniqueness) in resolving user-friendly names may be acceptable because in interactive systems the person using the name can be asked to resolve the ambiguity.

Other naming schemes are intended primarily for use by machines. In these schemes, the names need not have mnemonic value, so they are typically integers, often of fixed width designed to fit into a register and to allow fast and unambiguous resolution. Memory addresses and disk sector addresses are examples. Sometimes the term *identifier* is used for a name that is not intended to be intelligible to people, but this usage is by no means universal. Names intended for use by machines are usually chosen mechanically.

When a name is intended to be user-friendly, a tension arises between a need for it to be a unique, easily resolvable identifier and a need to respect other, non-technical values such as being easy to remember or being the same as some existing place or personal name. This tension may be resolved by maintaining a second, machine-oriented identifier, in addition to the user-friendly name—thus billing systems for large companies usually have both an account name and an account number. The second identifier can be unique and thus resolve ambiguities and avoid problems related to overloading of the account name. For example, personal names are usually overloaded with family history metadata (such as the surname, a given middle name that is the same as a mother's surname, or an appended "Jr." or "III"), and they are frequently not unique. Proposals to require that personal names be chosen uniquely always founder on cultural and personal identity objections. To avoid these problems, most systems

that maintain personal records assign distinct unique identifiers to people, and include both the user-friendly name and the unique identifier in their metadata.

Another example of tension in the choice of user-friendly names is found in the use of capital and small letters. Up through the mid-1960s, computer systems used only capital letters, and printed computer output always seemed to be shouting. There were a few terminals and printers that had lower-case letters, but one had to write a device-dependent application to make use of that feature, just as today one has to write a device-dependent application to use a virtual reality helmet. In 1965, the designers of the Multics time-sharing system introduced lower-case alphabetics to names of the file system. This being the first time anyone had tried it, they got it wrong. The designers of the UNIX file system copied the mistake. In turn, many modern file systems copy the UNIX design in order to avoid changing a widely used interface. The mistake is that the names "Court docket 5" and "Court Docket 5" can be bound to different files. The resulting violation of the ***principle of least astonishment*** can lead to significant confusion, since the computer rigidly enforces a distinction that most people are accustomed to overlooking on paper. Systems that enforce this distinction are called *case-sensitive*.

A more user-friendly way to allow upper- and lower-case letters in names is to permit the user to specify a preferred combination of upper- and lower-case letters for storage and display of a name, but coerce all alphabetic characters to the same case when doing name comparisons. Thus, when another person types the name, the case does not have to precisely match the display form. Systems that operate this way are called *case-preserving*. Both the Internet Domain Name System (described in Section 4.4) and the Macintosh file system provide this more user-friendly naming interface. A less satisfactory way to reduce case confusion is *case-coercing*, in which all names are both coerced to and stored in one case. A case-coercing system constrains the appearance of names in a way that can interfere with good human engineering.

The case studies in Section 3.2 and the war stories in Section 3.3 describe some unusual results when a system design mixes case-sensitive and case-preserving naming systems.

User-friendly names are not always strings of characters. In a graphical user interface (GUI), the shape (and sometimes the position) of an icon on the display is an identifier that acts exactly like a name, even if a character string is not associated with it. What action the system undertakes when the user clicks the mouse depends on where the mouse cursor was at that instant, and in a video game the action may depend on what else is happening at the same time. The identifier is thus bound to a time and a position on the screen, and that combination of values is in turn an identifier that is bound to some action.

Another, similar example of a user-friendly name that does not take the form of a string of characters is the cross-linking system developed by the M.I.T. Shakespeare Project. In that system, hypertext links say where they are coming from rather than where they are going to. Resolution starts by looking up the identifier of the place where the link was found. The principle is identical to that of the GUI/mouse example, and the system is described in Sidebar 3.2.

Sidebar 3.2 Hypertext Links in the Shakespeare Electronic Archive There are many representations of all of Shakespeare's plays: a modern text, the sixteenth-century folios, and several movies. In addition, a huge amount of metadata is available about each play: commentaries, stage directions, photographs and sketches of sets, directors' notes, and so on. In the study of a play, it would be helpful if these various representations could be linked together, so that, for example, if one were interested in the line "Alas, poor Yorick! I knew him, Horatio" from *Hamlet*, one could quickly check the wording in the several editions, compare different movie clips of the presentation of that line, and examine commentaries and stage directions that pertain to that line.

The M.I.T. Shakespeare Project has developed a system intended to make this kind of cross-reference easy. The basic scheme is first to assign a line number to every line in the play and then index every representation of the play by line number. A user displays one representation, for example, the text of a modern edition, and selects a line. Because the edition is indexed by line number, that selection is a reference that is bound to the line number. The user then clicks on the selection, causing the system to look up the associated line number in one of several contexts, each context corresponding to one of the other representations. The user selects a context, and the system can immediately resolve the line number in that context and display that representation in a different window on the user's screen.

3.1.6 Relative Lifetimes of Names, Values, and Bindings

If names must be chosen from a name space of short, fixed-length strings of bits or characters, they are by nature *limited* in number. The designer may permanently bind the names of a limited name space, as in the case of the registers of a simple processor, which may, for example, run from zero to 31. If the names of a limited name space can be dynamically bound, they must be reused. Therefore, the naming scheme usually replaces the BIND and UNBIND operations with some kind of name allocation/deallocation procedure. In addition, the naming scheme for a limited name space typically assigns the names, rather than letting the user choose them. On the other hand, if the name space is *unlimited*, meaning that it does not significantly constrain name lengths, it is usually possible to allow the user to choose arbitrary names. Thus, the telephone system in North America uses a naming scheme with short, fixed-length names such as 208–555–0175 for telephone numbers, and the telephone company nearly always assigns the numbers. (Section 3.3.5 describes some of the resulting problems.) On the other hand, names in most modern computer file systems are for practical purposes unlimited, and the user gets to choose them.

A naming scheme, a name, the binding of that name to a value, and the value to which the name is bound can all have different lifetimes. Often, both names and values are themselves quite long-lived, but the bindings that relate one to the other are somewhat more transient. Thus, both personal names and telephone numbers are typically long-lived, but when a person moves to a different city, the telephone company will

usually bind that personal name to a new telephone number and, after some delay, bind a new personal name to the old telephone number. In the same way, an application program and the operating system interfaces it uses may both be long-lived, but the binding that connects them may be established anew every time the program runs. Renewing the bindings each time the program is launched makes it possible to update the application program and the operating system independently. For another example, a named network service, such as `PostOffice.gov`, and a network attachment point, such as the Internet address 10.72.43.131, may both be long-lived, but the binding between them may change when the Post Office discovers that it needs to move that service to a different, more reliable computer, and it reassigns the old computer to a less important service.

When a name outlives its binding, any user of that name that still tries to resolve it will encounter a *dangling reference*, which is a use of a previously bound name that resolves either to a not-found result or to an irrelevant value. Thus, an old telephone number that rings in the wrong house or leads to a message saying "that number has been disconnected" is an example of a dangling reference. Dangling references are nearly always a concern when the name space is limited because names from limited name spaces must be reused. An object that incorrectly uses old names may make serious mistakes and even cause damage to an unrelated object that now has that name (for example, if the name is a physical memory address). In some cases, it may be possible to deal with dangling references by considering names to be simply hints that require verification. Thus when looking up the telephone number of a long-lost friend in a distant city, the first question when someone answers the phone at that number is something such as "are you the James Wilson who attended high school in . . . ?"

When a name space is unlimited and names are never reused, dangling references affect only the users of names that have for some reason been unbound from their former values. These dangling references can be less disruptive. For example, in a file system, an indirect name is one that is bound to some other (target) file system name. The indirect name becomes a dangling reference if someone removes the target name. Because an unbound indirect name simply produces a not-found result, it is more likely to be a nuisance than a source of damage. However, if someone accidentally or maliciously reuses the target name for a completely different file, the user of the indirect name could be in for a surprise.

When systems are large or distributed, however, a name, once bound and exported, tends to be discovered and remembered in widely dispersed places. That dispersion creates a need for stable bindings. This effect has been particularly noticed in the World Wide Web, whose design encourages the creation of cross-references to documents whose names are under someone else's control, with the result that cross-references often evolve into dangling references.

There is a converse to the dangling reference: when an object outlives every binding of a name to it, that object becomes what is known as an *orphan* or a *lost object* because no one can ever refer to it by name again. Lost objects can be a serious problem because there may be no good way to reclaim the physical storage they occupy. A system that regularly loses track of objects in this way is said to have a *storage leak*. To avoid lost

objects, some naming schemes keep track of the number of bindings to each object, and, when an UNBIND operation causes that number to reach zero, the system takes the opportunity to reclaim the storage occupied by the object. We saw this reference counting scheme used for links in the case study in Section 2.5. It contrasts with *tracing garbage collection*, an alternative technique used in some programming languages that involves occasional exploration of the named connections among objects to see which objects can and cannot be reached. The UNIX file system, described in Section 2.5, uses reference counting for file objects.

3.1.7 Looking Back and Ahead: Names are a Basic System Component

In this and the previous chapter, we have explored both the underlying principles of, and many engineering considerations surrounding, the use of names, but we have only lightly touched on the applications of names in systems. Names are a fundamental building block in all system areas. Looking ahead, almost every chapter will develop techniques and methods that depend on the use of names, name spaces, and binding:

- In modularizing systems with clients and services (Chapter 4), clients need a way to name services.
- In modularizing systems with virtualization (Chapter 5), virtual memory is an address naming system.
- In enhancing performance (Chapter 6), caches are renaming devices.
- Data communication networks (Chapter 7 [on-line]) use names to identify nodes and to route data to them.
- In transactions (Chapter 9 [on-line]) it is frequently necessary to modify several distinct objects "at the same time", meaning that all the changes appear to happen in a single program step, an example of atomicity. One way to obtain this form of atomicity is by temporarily grouping copies of all the objects that are to be changed into a composite object that has a temporary, hidden name, modifying the copies, and then rebinding the composite object to a visible name. In this way, all of the changed components are revealed simultaneously.
- In security (Chapter 11 [on-line]), designers use *keys*, which are names chosen randomly from a very large and sparsely populated address space. The underlying idea is that if the only way to ask for something is by name, and you don't know and can't guess its name, you can't ask for it, so it is protected.

Name discovery, which was introduced in the preceding chapter, will reappear when we discuss information protection and security. When one user either tries to identify or grant permission to another named user, it is essential to know the authentic name of that other user. If someone can trick you into using the wrong name, you may grant permission to a user who shouldn't have it. That requirement in turn means that one needs to be able to trace the name discovery procedure back to some terminating

direct communication step, verify that the direct communication took place in a credible fashion (such as examining a driver's license), and also evaluate the amount of trust to place in each of the other steps in the recursive name discovery protocol. Chapter 11 [on-line] describes this concern as the *name-to-key binding problem*.

Discovery of user names is one example in which authenticity is clearly of concern, but a similar authenticity concern can apply to any name binding, especially in systems shared by many users or attached to a network. If anyone can tinker with a binding, a user of that binding may make a mistake, such as sending something confidential to a hostile party. Chapter 11 [on-line] addresses in depth techniques of achieving authenticity. The User Internet Architecture research project uses such techniques to provide a secure, global naming system for mobile devices based on physical rendezvous and the trust found in social networks. For more detail, see Suggestions for Further Reading 3.2.5.

There is also a relation between uniqueness of names and security: If someone can trick you into using the same supposedly unique name for two different things, you may make a mistake that compromises security. The Host Identity Protocol addresses this problem by creating a name space of Internet hosts that is protected by cryptographic techniques similar to those described in Chapter 11 [on-line]. For more detail, see Internet Engineering Task Force *Request for Comments RFC 4423*.

This look ahead completes our introduction of concepts related to the design of naming systems. The next two sections of this chapter provide a case study of the relatively complex naming scheme used for pages of the World Wide Web, and a collection of war stories that illustrate what can go wrong when naming concepts fail to receive sufficient design consideration.

3.2 CASE STUDY: THE UNIFORM RESOURCE LOCATOR (URL)

The World Wide Web [see Suggestions for Further Reading 3.2.3] is a naming network with no unique root, potentially many different names for the same object, and complex context references. Its name-mapping algorithm is a conglomeration of several different component name-mapping algorithms. Let's fit it into the naming model.*

3.2.1 Surfing as a Referential Experience; Name Discovery

The Web has two layers of naming: an upper layer that is user-friendly, and a lower layer, which is also based on character strings, but is nevertheless substantially more mechanical.

*This case study informally introduces three message-related concepts that succeeding chapters will define more carefully: *client* (an entity that originates a request message); *server* (an entity that responds to a client's request); and *protocol* (an agreement on what messages to send and how to interpret their contents.) Chapter 4 expands on the client/service model, and Chapter 7 [on-line] expands the discussion of protocols.

At the upper layer, a Web page looks like any other page of illustrated text, except that one may notice what seem to be an unusually large number of underlined words, for example, Alice's page. These underlined pieces of text, as well as certain icons and regions within graphics, are labels for *hyperlinks* to other Web pages. If you click on a hyperlink, the browser will retrieve and display that other Web page. That is the user's view. The browser's view of a hyperlink is that it is a string in the current Web page written in HyperText Markup Language (HTML). Here is an example of a text hyperlink:

```
<a href="http://web.pedantic.edu/Alice/www/home.html">Alice's page</a>
```

Nestled inside this hyperlink, between the quotation marks, is a *Uniform Resource Locator* or, in Webspeak, a *URL*, which in the example is the name of another Web page at the lower naming layer. We can think of a hyperlink as binding a name (the underlined label) to a value (the URL) that is itself a name in URL name space. Since a context is a set of bindings of names to values, any page that contains hyperlinks can be thought of as a context, albeit not of the simple table-lookup variety. Instead, the name-mapping algorithm is one carried on in the mind of the user, matching ideas and concepts to the various hyperlink labels, icons, and graphics. The user does not usually traverse this naming network by typing path names, but rather by clicking on selected objects. In this naming network, a URL plays the role of a context reference for the links in the page fetched by the URL.

In order to retrieve a page in the World Wide Web, you need its URL. Many URLs can be found in hyperlinks on other Web pages, which helps if you happen to know the URL of one of those Web pages, but somewhere there must be a starting place. Most Web browsers come with one or more built-in Web pages that contain the URL of the browser maker plus a few other useful starting points in the Web. This is one way to get started on name discovery. Another form of name discovery is to see a URL mentioned in a newspaper advertisement.

3.2.2 Interpretation of the URL

In the example hyperlink above, we have an absolute URL, which means that the URL carries its own complete, explicit context reference:

```
http://web.pedantic.edu/Alice/www/home.html
```

The name-mapping algorithm for a URL works in several steps, as follows.

1. The browser extracts the part before the colon (here, `http`), considers it to be the name of a network protocol to use, and resolves that name to a protocol handler using a table-lookup context stored in the browser. The name of that context is built in to the browser. The interpretation of the rest of the URL depends on the protocol handler. The remaining steps describe the interpretation for the case of the hypertext transfer protocol (http) handler.
2. The browser takes the part between the `//` and the following `/` (in our example, that would be `web.pedantic.edu`) and asks the Internet Domain Name System (DNS) to resolve it. The value that DNS returns is an Internet address.

Section 4.4 is a case study of DNS that describes in detail how this resolution works.

3. The browser opens a connection to the server at that Internet address, using the protocol found in step 1, and as one of the first steps of that protocol it sends the remaining part of the URL, `/Alice/www/home.html`, to the server.
4. The server looks for a file in its file system that has that path name.
5. If the name resolution of step 4 is successful, the server sends the file with that path name to the client. The client transforms the file into a page suitable for display.

(Some Web servers perform additional name resolution steps. The discussion in Section 3.3.4 describes an example.)

The page sent by the server might contain a hyperlink of its own such as the following:

```
<a href="contacts.html">How to contact Alice.</a>
```

In this case the URL (again, the part between the quotation marks) does not carry its own context. This abbreviated URL is called a *relative* or *partial* URL. The browser has been asked to interpret this name, and in order to proceed it must supply a default context. The URL specification says to derive a context from the URL of the page in which the browser found this hyperlink, assuming somewhat plausibly that this hypertext link should be interpreted in the same context as the page in which it was found. Thus it takes the original URL and replaces its last component (`home.html`) with the partial URL, obtaining

```
http://web.pedantic.edu/Alice/www/contacts.html
```

It then performs the standard name-mapping algorithm on this newly fabricated absolute URL, and it should expect to find the desired page in Alice's `www` directory.

A page can override this default context by providing something called a *base element* (e.g., `<base href="some absolute URL">`). The absolute URL in the base element is a context reference to use in resolving any partial URL found on the page that contains the base element.

3.2.3 URL Case Sensitivity

Multiple naming schemes are involved in the Web naming algorithm, as is clear by noticing that some parts of a URL are case sensitive and other parts are not. The result can be quite puzzling. The host name part of a Uniform Resource Locator (URL) is interpreted by the Internet Domain Name System, which is case-insensitive. The rest of the URL is a different matter. The protocol name part is interpreted by the client browser, and case sensitivity depends on its implementation. (Check to see if a URL starting with `"HTTP://"` works with your favorite Web browser.) The Macintosh implementation of Firefox treats the protocol name in a case-preserving fashion, but the now-obsolete Macintosh implementation of Internet Explorer is case-coercing.

The more interesting case-sensitivity questions come after the host name. The Web specifies that the server should interpret this part of the URL using a scheme that depends on the protocol. In the case of the HTTP protocol, the URL specification is insistent that this string is *not* a UNIX file name, but it is silent on case sensitivity. In practice, most systems interpret this string as a path name in their file system, so case-sensitivity depends on the file system of the server. Thus if the server is running a standard UNIX system, the path name is case-sensitive, while if the server is a standard Macintosh, the path name is case-preserving. There are examples that mix things up even further; Section 3.3.4 describes one such example.

3.2.4 Wrong Context References for a Partial URL

The practice of interpreting URL path names as path names of the server's file system can result in unexpected surprises. As described earlier, the Web browser supplies a default context reference for relative names (that is, partial URLs) found in Web pages. The default context reference it supplies is simply the URL that the browser used to retrieve the page that contained the relative name, truncated back to the last slash character. This context reference is the name of a directory at the server that should be used to resolve the (first component of) the relative name.

Some servers provide a URL name space by simply using the local (for example, UNIX) file system name space. When the local file system name space allows synonyms (symbolic links and the Network File System mounts described in Section 4.5 are two examples) for directory names, the mapping of local file system name space to the URL name space is not unique. Thus, several different URLs can have different path names for the same object. For example, suppose that there is a UNIX file system with a symbolic link named `/alice/home.html` that is actually an indirect reference to the file named `/alice/www/home.html`. In that case, the URLs

1 `<http://web.pedantic.edu/alice/home.html>`

and

2 `<http://web.pedantic.edu/alice/www/home.html>`

refer to the same file. Trouble can arise when the object that has multiple URLs is a directory whose name is used as a context reference. Continuing the example, suppose that file `home.html` contains the hyperlink `<a href = "contacts.html">`. Both `home.html` and `contacts.html` are stored in the directory `/alice/www`. Suppose further that the browser obtained `home.html` by using the URL *1* above.

Now, the user clicks on the hyperlink containing the partial URL `contacts.html`, asking the browser to resolve it. Following the usual procedure, the browser materializes a default context reference by truncating the original URL to obtain:

`http://web.pedantic.edu/alice/`

and then uses this name as a context by concatenating the partial URL:

```
http://web.pedantic.edu/alice/contacts.html
```

This URL will probably produce a not-found response because the file we are looking for actually has the path name `/alice/www/contacts.html`. Or worse, this request could return a different file that happens to be named `contacts.html` in the directory `/alice`. The confusion may be compounded if the different file with the same name turns out to be an out-of-date copy of the current `contacts.html`. On the other hand, if the user originally used URL *2,* the browser would retrieve the file named `/alice/www/contacts.html`, as the Web page designer expected.

A similar problem can arise when interpreting the relative name "..". This name is, conventionally, the name for the parent directory of the current directory. The UNIX system provides a semantic interpretation: look up the name ".." in the current directory, where by convention it evaluates (in inode name space) to the parent directory. In contrast, the Web specifies that ".." is a syntactic signal that means "modify the default context reference by discarding the least significant component of the path name." Despite these drastically different interpretations of "..", the result is usually the same because the parent of an object is usually the thing named by the next-earlier component of that object's path name. The exception (and the problem) arises when the Web's syntactic modification rule is applied to a path name with a component that is an indirect name for a directory. If the path name in the URL does not traverse the directory's parent, syntactic interpretation of ".." creates a default context reference different from the one that would be supplied by semantic interpretation.

Suppose, in our example, that the file `home.html` contains the hyperlink `<a href="../phone.html">`. If the user who reached home.html via URL *1* clicks on this hyperlink, the browser will truncate that URL and concatenate it with the partial URL, to obtain

```
http://web.pedantic.edu/alice/../phone.html
```

and then use the syntactic interpretation of " . . " to produce the URL

```
http://web.pedantic.edu/phone.html
```

another non-existent file. Again, if the user had started with URL *2*, the result of syntactic interpretation of " . . " would be to request the file

```
http://web.pedantic.edu/alice/phone.html
```

as originally intended.

This problem could be fixed in at least three different ways:

1. Arrange things so that the default context reference always works.
   a. Always install a UNIX link to the referenced page in the directory that held the referring page. (Or never use UNIX links at all.)
   b. Never use ".." in hyperlinks.

2. Do a better of job of choosing a default context reference.
   a. The client sends the original URL plus the Web link to the server and lets the server figure out what context to use.
3. Provide an explicit context reference.
   a. The server places an absolute URL in a location field of the protocol header.
   b. The client uses that URL as the context reference.

One might suggest that the implementer of the server (or the writer of the pages containing the relative links) failed to heed the following warning in the Web URL specification* for path names: "The similarity to unix and other disk operating system filename conventions should be taken as purely coincidental, and should not be taken to indicate that URIs should be interpreted as file names."

This warning is technically correct, but the suggestion is misleading. Unfortunately, the problem is built in to the Web naming specifications. Those specifications require that relative names be interpreted syntactically, yet they do not require that every object have a unique URL. Unambiguous syntactic interpretation of relative names requires that the context reference be a unique path name. Since the browser derives the context reference from the path name of the object that contained the relative name, and that object's path name does not have to be unique, it follows that syntactic interpretation of relative names will intrinsically be ambiguous. When servers try to map URL path names to UNIX path names, which are not unique, they are better characterized as exposing, rather than causing, the problem.

That analysis suggests that one way to conquer the problem is to change the way in which the browser acquires the context reference. If the browser could somehow obtain a canonical path name for the context reference, the same canonical path name that the UNIX system uses to reach the directory from the root, the problem would vanish.

3.2.5 Overloading of Names in URLs

Occasionally, one will encounter a URL that looks something like

```
http://www.amazon.com/exec/obidos/ASIN/0670672262/o/qid=
  921285151/sr=2-2/002-7663546-7016232
```

or perhaps

```
http://www.google.com/search?hl=en&q=books+about+systems&btnG=
  Google+Search&aq=f&oq=
```

Here we have two splendid examples of overloading of names. The first example is of a shopping service. Because the server cannot depend on the client to maintain any state about this shopping session other than the URL of the Web page currently being displayed, the server has encoded the state of the shopping session, in the form of an identifier of a state-maintaining file at the server, in the path name part of the URL.

*Tim Berners-Lee, *Universal Resource Identifiers: Recommendations*.

The second example is of a search service; the browser has encoded the user's search query into the path name part of the URL it has submitted. The tip-off here is the question mark in the middle of the name, which is a syntactic trick to alert the server that the string up to the question mark is the name of a program to be launched, while the string after the question mark is an argument to be given to that program. To see what processing www.google.com does to respond to such a query, see Suggestions for Further Reading 3.2.4.

There is another form of overloading in many URLs: they concatenate the name of a computer site with a path name of a file, neither of which is a particularly stable identifier. Consider the following name for an earthquake information service:

```
http://HPserver14.pedantic.edu/disk05/science/geophysics/
  quakes.html
```

This name is at risk of change if the HP computer is replaced by a Sun server, if the file server is moved to `disk04`, if the geophysics department is renamed "geology" or moves out of the school of science, or if the responsibility for the earthquake server moves to the Institute for Scholarly Studies. A URL such as this example frequently turns out to be unresolvable, even though the page it originally pointed to is still out there somewhere, perhaps having moved to a different site or simply to a different directory at the original site.

One way to avoid trapping the name of a site in the URLs that point to it is to choose a service name and arrange for DNS to bind that service name as an indirect name for the site. Then, if it becomes necessary to move the Web site to a different computer, a change to the binding of the service name is all that is needed for the old URLs to continue working. Similarly, one can avoid trapping an overloaded path name in a URL by judicious use of indirect file names. Thus the name

```
http://quake.org/library/quakes.html
```

could refer to the same Web page, yet it can remain stable through a wide variety of changes.

Considerable intellectual energy has been devoted to inventing a replacement for the URL that has less overloading and is thus more robust in the face of changes of server site and file system structure. Several systems have been proposed: *Permanent URL* (PURL), *Universal Resource Name* (URN), *Digital Object Identifier* (DOI)®, and *handle*. To date, none of these proposals has yet achieved wide enough adoption to replace the URL.

3.3 WAR STORIES: PATHOLOGIES IN THE USE OF NAMES

Although designing a naming scheme seems to be a straightforward exercise, it is surprisingly difficult to meet all of the necessary requirements simultaneously. The following are several examples of strange and sometimes surprising results that have been noticed in deployed naming schemes.

3.3.1 A Name Collision Eliminates Smiling Faces

A west coast university provides a "visual class list" Web interface that instructors can use to obtain the names and photos of all the students enrolled in a particular section of a class. At the beginning of the fall 2004 teaching term, instructors noticed that their classes had several photographs of the same individual. One might believe a section includes a set of triplets, but not triskaidekatuplets.

What went wrong: When there is no picture available for a student, the system inserts an image of a smiley face with the words "No picture available". The system designer stored the image in a file named "smiley.jpg". That fall a new freshman whose last name was Smiley registered the user name "smiley". As one might expect, the freshman's photograph was named "smiley.jpg", and it became the "No picture available" image.

3.3.2 Fragile Names from Overloading, and a Market Solution

Internet mailbox names such as `Alice@Awesome.net` can be viewed as two-component addresses. The component before the @-sign identifies a particular mailbox, and the component after the @-sign is an Internet domain name that identifies the Internet service provider (ISP) that provides that mailbox. When two ISPs (say, `Awesome.net` and `Awful.net`) merge, the customers of one of them (and sometimes both) typically receive a letter telling them that their mailbox address, which contained some representation of the name of their former ISP, will have to change. The new ISP may automatically forward mail addressed to the old address, or it may require that the user notify all of his or her correspondents of the new mailbox address. The reason for the change is that the second component of the old mailbox name was overloaded with a trademark. The new provider does not want to continue using that old trademark, and the old provider may not want to see the trademark used by the new provider.

Alice may also find, to her disappointment, that not only does the domain name of her mailbox change from `Awesome.net` to `Awful.net`, but that in `Awful.net`'s mailbox name space, another customer has already captured the personal mailbox name `Alice`, so she may even have to choose a new personal mailbox name, such as `Alice24`.

As the Internet grows, some ISPs have prospered and others have not, so there have been many mergers and buyouts. The resulting fragility of e-mail service provider names has created a market for indirect domain names. The customers in this market are users who require a stable e-mail address, such as people who run private businesses or who have a large number of correspondents. For an annual fee, an indirect name provider will register a new domain name, such as `Alice.com`, and configure a DNS name server so that the mailbox name `Alice@Alice.com` becomes a synonym for `Alice@Awesome.Net`. Then, upon being notified of the ISP merger, Alice simply asks the indirect name provider to rebind the mailbox name `Alice@Alice.com` to `Alice24@Awful.net`, and her correspondents don't have to know that anything happened.

3.3.3 More Fragile Names from Overloading, with Market Disruption

The United States Post Office assigns postal delivery codes, called Zip codes, hierarchically, so that it can take advantage of the hierarchy in routing mail. Zip codes have five digits. The first digit identifies one of 10 national areas; New England is area 0 and California, Washington, and Oregon comprise area 9. The next two digits identify a section. The South Station Postal Annex in Boston, Massachusetts, is the headquarters of section 021. All Zip codes beginning with those three digits have their mail sorted at that sectional center. Zip codes beginning with 024 identify the Waltham, Massachusetts, section. The last two digits of the Zip code identify a specific post office (known as a station), such as Waban, Massachusetts, 02468. Zip codes can also have four appended digits (called Zip + 4) that are used to sort mail into delivery order for each mail carrier. Although they are numerical, adjacent zip codes are not necessarily assigned to adjacent stations or adjacent sections, so they are really names rather than physical addresses. Despite not being interpretable as physical addresses, these names are overloaded with routing information.

Although routing is hierarchical, apparently the 10 national areas have no routing significance; everything is done by section. It is reported that if you walk into the South Station Postal Annex in Boston, you will find that outgoing mail is being sorted into 999 bins, one for each sectional center, nationwide. In addition, for mail addressed with Zip codes beginning with 021 (that is, within the South Station section) there are 99 bins, one for each station within the section. The mail in the outgoing bins goes into bags, with each bag containing mail for one section. Then all the bags for Southern California sections, for example, go into the same truck to the airport, where they go onto a plane to Los Angeles. As they come off the plane in Los Angeles, they are loaded onto different trucks that go to the various Southern California sections. The mail in the 99 bins for section 021 also goes into bags, with each bag destined for a different post office within the 021 section.

Mail that originates at a post office and is destined for the same post office still goes to the sectional center for sorting because individual post offices don't have the automatic sorting machines that can put things into delivery order. There used to be many exceptions to the rule that all mail goes to a sectional center, but the number of exceptions has been gradually reduced over the years.

When the volume of mail handled by the South Station Postal Annex began to exceed its capacity in the late 1990s, the Post Office decided to transfer part of that section's work to the newer Waltham, Massachusetts, section. Since the first three digits of the Zip code are overloaded with routing information, to accomplish this change it announced that about half of the Zip codes that began with 021 would, on July 1, 1998, change to 024. The result, as one might expect, was rather chaotic. The Post Office tried to work with large mailers to have them automatically update their address records, but loose ends soon appeared.

For example, American Express, a credit card company, installed a Zip code translator in its mail label printing system, so that its billing statements would go directly to the Waltham section, but it did not change its internal customer address

records because its computer system flags all address changes as "moves", which affect verification procedures as well as credit ratings. So everything that American Express mailed was addressed properly, but their internal records retained the old Zip codes.

Now comes the problem: some Internet vendors will not accept a credit card unless the shipping address is identical to the credit card address. Customers began to encounter situations in which the Internet vendor rejected the Zip code 02168 as being an invalid delivery address, and American Express rejected the Zip code 02468 because it did not match its customer record. When this situation arose, it was not possible to complete a purchase without human intervention.

Despite the vendor check that identifies 02168 as invalid, mail addressed with that Zip code continued to be correctly delivered to addresses in Waban for several years. It just took an extra day to be delivered because it went first to the South Station Postal Annex, which simply forwarded it to the Waltham sectional center. The renaming was done not because the post office was running out of Zip codes, but rather because the sorting capacity of one of its sectional centers was exceeded.

3.3.4 Case-Sensitivity in User-Friendly Names

Even though, as described on page 128, the UNIX system propagated case-sensitive file names to many other file systems, not all widely used naming schemes are case-sensitive. The Internet generally is case-preserving. For example, in the Internet Domain Name System described in Section 4.4, one can open a network connection to `cse.pedantic.edu` or to `CSE.Pedantic.edu`; both refer to the same destination. The Internet mail system is also specified to be case-preserving, so you can send mail to `alice@pedantic.edu`, `Alice@pedantic.edu`, and `aLiCe@pedantic.edu`, and all three messages should go to the same mailbox.

In contrast, the Kerberos authentication system (described in Sidebar 11.6 [on-line]) is case-sensitive, so the names "alice" and "Alice" can identify different users. The rationale for this decision is muddy. Requiring that the case accurately match makes it harder for an intruder to guess a user's name, so one can argue that this decision enhances security. But allowing "alice" and "Alice" to identify different users can lead to serious mistakes in setting up permissions, so one can also argue that this decision weakens security. This decision comes to a head, for example, in the implementation of a mail delivery service with Kerberos authentication. It is not possible to correctly do a direct mapping of Kerberos user names to mailbox names because the necessary coercion might merge the identities of two distinct users.

A mixed example is a service-naming service developed at M.I.T. and called Hesiod, which uses the Internet Domain Name System (DNS) as a subsystem. One of the kinds of services Hesiod can name is a remote file system. DNS (and thus Hesiod) is case-insensitive, while file system names in UNIX systems are case-sensitive. This difference leads to another example of a user interface glitch. If a user asks to attach a remote file

system, specifying its Hesiod name, Hesiod will locate the file system using whatever case the user typed, but the UNIX `mount` command mounts the file system using the name coerced to lower case. Thus if the user says, for example, to mount the remote file system named `CSE`, Hesiod will locate that remote file system, but the UNIX system will mount it using the name `cse`. To use this directory in a file name, the user must then type its name in lower case, which may come as a surprise.

Hesiod is used as a subsystem in larger systems, so the mixing of case-sensitive and case-insensitive names can become worse. For example, the current official M.I.T. Web server responds to the URL

```
http://web.mit.edu/Alice/www/home.html
```

by first trying a simple path name resolution of the string `/Alice/www/home.html`. If it gets a `NOT-FOUND` result from the resolution of that path name, it extracts the first component of the path name (`Alice`) and presents it to the Hesiod service naming system, with a request to interpret it as a remote file system name. Since Hesiod is case-insensitive, it doesn't matter whether the presented name is `Alice`, `alice`, or `aLiCe`. Whatever the case of the name presented, Hesiod coerces it to a standard case, and then it returns the standard file system path name of the corresponding remote file system directory, which for this example might be

```
/afs/athena/user/alice
```

The Web server then replaces the original first component (`Alice`) with this path name and attempts to resolve the path name:

```
/afs/athena/user/alice/www/home.html
```

Thus for the current M.I.T. Web server, the first component name after the host name in a URL is case-insensitive, while the rest of the name is case-sensitive.

3.3.5 Running Out of Telephone Numbers

"Nynex is Proposing New '646' Area Code for Manhattan Lines"

— headline, *Wall Street Journal,* March 3, 1997

The North American telephone numbering plan name space is nicely hierarchical, which would seem to make it easy to add phone numbers. Although this appears to be an example of an unlimited name space, it is not. It is hierarchical, but the hierarchy is rigid—there is a fixed number of levels, and each level has a fixed size.

Much of Europe does it the other way. In some countries it seems that every phone number has a different number of digits. A variable-length numbering plan has a downside. The telephone numbers are longest in the places that grew the most and thus have the most telephone calls. In addition, because the central exchange can't find the end of a variable-length telephone number by counting digits, some other scheme is necessary, such as noticing that the user has stopped dialing for a while.

A European-style solution to the shortage of phone numbers in Manhattan would be to simply announce that from now on, all numbers in Manhattan will be 11 digits long. But since the entire American telephone system assumes that telephone numbers are exactly 10 digits long, the American solution is to introduce a new area code.

A new area code can be introduced in one of two ways: *splitting* and *overlay*. Traditionally, the phone companies have used only splitting, but overlay is beginning to receive wider attention.

Splitting (sometimes called partition) is done by drawing a geographical line across the middle of the old area code—say 84th street in Manhattan—and declaring that everyone north of that line is now in code 646 and everyone south of that line will remain in code 212. When splitting is used, no one "changes" their seven-digit number, but many people must learn a new number when calling someone else. For example,

- Callers from Los Angeles who used to dial (212)-xxx-xxxx must now dial (646)-xxx-xxxx if they are calling to a phone north of 84th street, but they must use the old area code for phones located south of 84th street.
- Calling from one side of 84th street to the other now requires adding an area code, where previously a seven-digit number was all one had to dial.

In the alternative scheme, overlay, area code 212 would continue to cover all of Manhattan, but when there weren't any phone numbers left in that area code, the telephone companies would simply start assigning new numbers with area code 646. Overlay places a burden on the switching system, and it wouldn't have worked with the step-by-step switches developed in the 1920s, in which the telephone number described the route to that telephone. When the Bell System started to design the crossbar switches introduced in the 1940s, it realized that this inflexibility was a killer problem, so it introduced a number-to-route lookup—a name-resolving system called a *translator*—as part of switch design. With modern computer-based switches, translation is easy. So there is now nothing (but old software) to prevent two phones served by the same switch from having numbers with different area codes.

Overlay sounds like a great idea because it means that callers from Los Angeles continue to dial the same numbers they have always dialed. However, as in most engineering trade-offs, someone loses. Everyone in Manhattan now has to dial a 10-digit number to reach other places in Manhattan. One no longer can tell what the area code is by the geographic location of the phone. One also can't pinpoint the location of the target by its area code because the area code has lost its status as geographic metadata. This could be a concern if people become confused as to whether or not they were making a toll call.

Another possibility would be to use as a default context the area code of the originating phone. If calling from a 212 phone, one wouldn't have to dial an area code to call another 212 number. The prevailing opinion—which may be wrong—is that people can't handle the resulting confusion. Two phones on the

same desk, or two adjacent pay phones, may have different area codes, and thus to call someone in the next office one might have to dial different numbers from the two phones.

Here is one way of coping: BankBoston (long since merged into larger banks) once arranged that the telephone number 788–5000 ring its customer service center from every area code in the state of Massachusetts. The nationwide toll-free number (800) 788–5000 also rang there. Although that arrangement did not completely eliminate name translation, it reduced it significantly and made the remaining name translation simple enough that people could actually remember how to do it.

Requiring that all numbers be dialed with all 10 digits encourages a more coherent model: the number you dial to reach a particular target phone does not depend on the number from which you are calling. The trade-off is that every North American number dialed would require 10 digits, even when calling the phone next door. The North American telephone system has been gradually moving in this direction for a long time. In many areas, it was once possible to call people in the same exchange simply by dialing just the last four digits of their number. Then it took five digits. Then seven. The jump to 10 would thus be another step in the sequence.

The newspaper also reports that at the rate telephone numbers are being used up in Manhattan, another area code will be needed within a few years. That observation would seem to affect the decision. Splitting is disruptive every time, but overlay is disruptive only the first time it is done. If there is going to be another area code needed that soon, it might be better to use overlay at the earliest opportunity, since adding still more area codes with overlay will cause no disruption at all.

Overlay is already widely used. Manhattan cell phones and beepers have long used area code 917, and little confusion resulted. Also, in response to an outcry over "yet another number change", in 1997 the Commonwealth of Massachusetts began requiring that future changes to its telephone numbering plan be done with overlay.

EXERCISES

3.1 Alyssa asks you for some help in understanding how metadata is handled in the UNIX file system, as described in Section 2.5.

3.1a Where does the UNIX system store system metadata about a file?

3.1b Where does it store user metadata about the file?

3.1c Where does it store system metadata about a file system?

2008-0-1

3.2 Bob and Alice are using a UNIX file system as described in Section 2.5. The file system has two disks, mounted as `/disk1` and `/disk2`. A system administrator

creates a "home" directory containing symbolic links to the home directories of Bob and Alice via the commands:

```
mkdir /home
ln -s /disk1/alice /home/alice
ln -s /disk2/bob /home/bob
```

Subsequently, Bob types the following to his shell:

```
cd /home/alice
cd ../bob
```

and receives an error.

Which of the following best explains the problem?

A. The UNIX file system forbids the use of ".." in a cd command when the current working directory contains a symbolic link.

B. Since Alice's home directory now has two parents, the system complains that ".." is ambiguous in that directory.

C. In Alice's home directory, ".." is a link to `/disk1`, while the directory "bob" is in `/disk2`.

D. Symbolic links to directories on other disks are not supported in the UNIX file system; their call-by-name semantics allows their creation but causes an error when they are used.

2007-1-7

3.3 We can label the path names in the previous question as *semantic* path names. If Bob types `"cd .."` while in working directory `d`, the command changes the working directory to the directory in which `d` was created. To make the behavior of `"..."` more intuitive, Alice proposes that `"..."` should behave in path names *syntactically*. That is, the parent of a directory `d`, `d/..` is the same directory that would obtain if we instead referred to that directory by removing the last path name component of `d`. For example, if Bob's current working directory is `/a/b/c` and Bob types `"cd .."`, the result is exactly as if Bob had executed `"cd /a/b"`

3.3a If the UNIX file system were to implement syntactic path names, in which directory would Bob end up after typing the following two commands?

```
cd /home/alice
cd ../bob
```

3.3b Under what circumstances do semantic path names and syntactic path names provide the same behavior?

A. When the name space of the file system forms an undirected graph.

B. When the name space of the file system forms a tree rooted at "/".

C. When there are no synonyms for directories.

D. When symbolic links, like hard links, can be used as synonyms only for files.

3.3c Bob proposes the following implementation of syntactic names. He will first rewrite a path name syntactically to eliminate the "..", and then resolve the rewritten path name forward from the root. Compared to the implementation of semantic path names as described in Section 2.5, what is a disadvantage of this syntactic implementation?

A. The syntactic implementation may require many more disk accesses than for semantic path names.

B. This cost of the syntactic implementation scales linearly with the number of path name components.

C. The syntactic implementation doesn't work correctly in the presence of hard links.

D. The syntactic implementation doesn't resolve "." correctly in the current working directory.

2007-0-1

3.4 The inode of a file plays an important role in the UNIX file system. Which of these statements is true of the inode data structure, as described in Section 2.5?

A. The inode of a file contains a reference count.

B. The reference count of the inode of a directory should not be larger than 1.

C. The inode of a directory contains the inodes of the files in the directory.

D. The inode of a symbolic link contains the inode number of the target of the link.

E. The inode of a directory contains the inode numbers of the files in the directory.

F. The inode number is a disk address.

G. A file's inode is stored in the first 64 bytes of the file.

2005-1-4, 2006-1-1, and 2008-1-3

3.5 Section 3.3.1 describes a name collision problem. What could the designer of that system have done differently to eliminate (or reduce to a negligible probability) the possibility of this problem arising?

2008-0-2

Additional exercises relating to Chapter 3 can be found in the problem sets beginning on page 425.