The Hideous Name †

Rob Pike
P.J. Weinberger

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

The principles of good naming in computing have been known for decades. The invention of new facilities in computing systems can be guided by these principles. For example, the introduction of networking need not require any change to the majority of system utilities, because objects such as files on remote machines can be given syntactically familiar names within the local machine's name space. Indeed, the implementers of networks often do well by these standards by striving to make remote files essentially indistinguishable from local ones. Unfortunately, the situation with internetwork mail addresses is not as satisfactory. The practitioners of internetworking would profit by understanding the benefits of simple, uniform syntax.

research!ucbvax!@cmu-cs-pt.arpa:@CMU-ITC-LINUS:dave%CMU-ITC-LINUS@CMU-CS-PT
- Carnegie-Mellon University mailer

I cannot tell what the dickens his name is.
- Shakespeare, *Merry Wives of Windsor*, II. ii. 20.

Introduction

Any object relevant to computation — file, process, user, computer, network or whatever — needs a name. The name determines the access: it is by interpreting the name, within the *name space* it inhabits, that a program or person is given access to the object. The manner in which names are constructed affects not only how objects are named but also how they are used.

The form of names and name spaces is the subject of this essay. We will use file names from several operating systems as examples to illustrate criteria for distinguishing good names from bad. The same criteria may be applied to network mail names, pointing out some of the shortcomings of the current *ad hoc* systems for internetworking.

The criteria are not new, and seem to be generally accepted, but are not applied in practice. This paper is an attempt to re-establish their use.

Principles of Names and Name Spaces

What's in a name? A string of characters, encoded by some convention (ASCII or EBCDIC), that identifies an object. If the function of a name stopped there, this definition would be sufficient. In IBM's MVS, for example, a file name is an at most 44-character string, largely uninterpreted by system software. The name space is 'flat,' or linear.

Systems designed more recently use names to help *organize* as well as identify. For instance, although in MVS the disk containing the file is specified separately from the name, in MS-DOS part of the

---

† An earlier version of this appeared in Summer 1985 Usenix Conference Proceedings, Salt Lake City, Utah.

name of a file is a string that identifies the disk drive holding the file. Syntax separates these components of the name; MS-DOS uses a colon following the disk name, a single character:

```
A:FILE
```

is a file on disk drive `A`, while

```
B:FILE
```

is a file on disk drive `B`.

The advantage of putting such information in names is that software need not know about disks to manipulate files. Internally, of course, system software must use the syntax of the name to locate the file, but this is largely transparent to applications software, and users.

The MVS system uses independent information (stored in a catalog) to find a file given its name, while MS-DOS exposes location information in the name. MS-DOS users may put related files on the same disk, thus using a distinguished piece of the name to help organize their world. MVS users must instead adapt *ad hoc* strategies (conventions for the syntax of names) to the same end. Thus each system does both more and less for the user: MVS provides no help in organization and a naming independent of the physical location of the file, while MS-DOS provides the opposite. Instead, if names had multiple components (that is, syntax), where the components did not necessarily correspond to physical devices, the name space would have the advantages of that of both MVS and MS-DOS, with the disadvantages of neither. Such a name space exists.

A good example: UNIX®

UNIX file name space is a tree, with file names that specify a path from one node to another. The representation of the name is a simple ASCII string, with slashes `/` separating node identifiers. The name

```
/usr/rob/bin/cat-v
```

is a path from the root of the tree (denoted by the leading slash), through nodes called `usr`, `rob` and `bin`, to a file called `cat-v`. Nodes intermediate in the tree are called *directories*. The system uses the components to find the file, which the user can use as syntax to organize sets of files. For example, although on most UNIX systems, the string `/usr` specifies a separate disk drive, this is irrelevant to both software and users; it is merely a string that identifies the directory beneath which users' files may be found.

The structure of the name space (a directory tree) is reflected in the style of the name (a path through the tree). Were the file system arranged differently, say as a flat array, the form and interpretation of file names would also be different; for example, UNIX processes are named by small integers.

Properties of Name Spaces

Name spaces have some general properties. First, names within the space may be absolute or relative. Absolute names specify an object by position with respect to a single fixed point, such as the root of the UNIX file system (named `/`); relative names, with respect to a local point (the 'current working directory' in UNIX, named `.` (dot)). Also, an operating system typically has operators to manipulate its name space, such as system calls to create and remove files. UNIX also provides a system call (named `mount`) to join together two name spaces by attaching the root of one space, resident on a separate disk drive, to a leaf of another.

Finally, a name space has syntax — how a name is constructed — and semantics — the nature of the object a name identifies.

A UNIX file name, for example, is a sequence of slash-separated strings that identifies a formatless byte stream. External conventions may provide further semantics: the UNIX file system contains objects that are not ordinary files. Simply by having ordinary file names, though, these objects have ordinary file properties such as protection. Some examples from our research version of UNIX, called the Ninth Edition:

Device files. With names conventionally prefixed (that is, residing in the directory) `/dev`, these files provide direct access to devices. The name `/dev/mt`, for example, identifies a magnetic tape drive.

Processes. The directory `/proc` contains files with names that are process numbers. Opening such a file provides access to processes for purposes such as interactive debugging. Although processes have integers that identify them, it is convenient to provide names for them in the file system as well. For example, listing the directory containing the process files is a simple way to identify running processes.

Databases. Some databases are conveniently represented as name-value pairs in a hierarchy, and such databases may be mapped into the file system name space. For example, the directory `/n/face` contains a hierarchically-structured database that associates digitized images of people's faces with the people's electronic mail addresses.

Other files. UNIX has the notion of *standard input*: the input connected to a (typically) interactive program. The name `/dev/stdin` identifies a file that, when opened, connects to the standard input of the program. This allows files that demand a file name (such as the file comparison program) to be given input directly from the interactive terminal or from a pipe.

Because these unusual objects have regular names, existing tools can treat them as files, so standard software can provide services for them that would otherwise require special handling.

Some of the Ninth Edition examples above have different names in other UNIX systems. `/dev/stdin` is often represented by the single character `-`, as an argument to commands, but this convention is capriciously followed. Because it must be provided explicitly by each program, it is only available in some programs. By providing `/dev/stdin` in the global name space, it is available uniformly for all programs, always. As another example, processes are represented by an integer process identifier, which is only meaningful to a few process-specific system calls. These calls implement their own protection mechanism, although the protection provided by the file system suits perfectly (these system calls predate the process file system). Finally, virtual terminals implemented using the multiplexed files of the Seventh Edition (an earlier research version of the system) have no external name, so it is impossible to open one for I/O. The Ninth Edition provides a name in the file system that is available, without prearrangement or special protocol, to any program.

Connecting Name Spaces

When machines are connected together, their name spaces may be joined to facilitate the sharing of files. If the name spaces have the same clean structure, that structure can be extended simply to describe the larger space. The Newcastle Connection names a file on another machine, say `ucbvax`, as `/../ucbvax/usr/rob/bin/cat-v`; the Ninth Edition notation is `/n/ucbvax/usr/rob/bin/cat-v`. In the former the name space has been extended by making it a subspace of a larger space, in the latter a new name subspace has been grafted on using `mount`, but in neither case has the *syntax* of names been changed; any program that understands a file name will understand a network file name without change, and relative names for files (those that don't begin with `/`) are unchanged. As a spectacular example, we might see on which machines user `wnj` has a login by searching (using a program called `grep`) through the system administration files (called `/etc/passwd`) on all the machines:

```
grep wnj /n/*/etc/passwd
```

The file name 'wild card' character `*` matches all files within a directory. Here, it happens to match all machines reachable from the local machine, although `grep` is oblivious of this distinction. We could even investigate those machines connected to `ucbvax` by

```
grep wnj /n/ucbvax/n/*/etc/passwd
```

The file system that is the union of these name spaces might have no global root, so the meaning of an absolute name may become ambiguous because of the presence of multiple reference points. In fact, there might be no single point to which all names can be fixed. In practice, though, this ambiguity is unimportant.

A bad example: VAX/VMS

Unfortunately, not everyone chooses naming conventions in accord with these guidelines. On VAX/VMS our canonical file might be called `UCBVAX::SYS$DISK:[ROB.BIN]CAT_V.EXE;13`. The VMS file naming scheme provides a distinct syntax for each level in the name: `UCBVAX::` is a machine; `SYS$DISK:` is a disk (actually a macro that expands to a disk name such as `DUA0:`); `[ROB.BIN]` is a directory; `CAT_V` is a file 'base' name; `.EXE` is a file 'type'; and `;13` is a version number.

Although this syntax may seem unnecessarily cumbersome, it has a precedent: it is analogous to expressions in programming languages. Consider a C expression such as `*structure[index].field->ptr`. If `*` were postfix and `/` the only dereferencing operator, the expression might be written `structure/index/field/ptr/`. Functionally-minded programmers might use the notation `contents(ptr(field(index(structure))))`. (A single character cannot be used in C because it could not distinguish `X[Y]` and `X->Y`, with `X` a structure pointer and `Y` an integer or structure element respectively, but this ambiguity could be eliminated in a different language.) C and VMS use syntax to distinguish the types of the components of a name. Instead, the UNIX file system deliberately hides the distinctions. Aside from the obvious advantages such as simplicity of syntax and the usurping of only a single character, the uniformity also makes the name space easier to manipulate: the `mount` system call aliases a disk and a directory.

VMS has no true name space manipulation operator. Although one could be constructed, it would be limited in scope: how could a disk be mounted atop `SYS$DISK:[ROB.BIN]` when disks are always before directories in the name? Instead, VMS has macros such as `SYS$DISK` to hide the manner in which the space is assembled, and even to provide the concept of a local name by automatically inserting an expanded macro before an unqualified name.

The problems with dynamic evaluation of macros are well known. For example, the VMS service to set the reference point for local names (the equivalent of UNIX `chdir`) sets the default prefix for file names, but the prefix will only be evaluated and so checked for validity, when a file name is interpreted, which may be arbitrarily and confusingly long after the prefix was set. In fact, the default prefix macro is handled in a special way, because directories are not constructed by simple concatenation; subdirectory `[.BIN]` in directory `[ROB]` is named `[ROB.BIN]`. Also, these local names are not really local at all; the prefix implicitly binds them to a root of the name space. This implies that all names are always attached to some root, and therefore if the root changes, the name must also change, invisibly.

Another problem with VMS names is that one cannot do the equivalent of searching the VMS password files (`SYSUAF.LIS`) on various machines with `*::SYS$SYSTEM:SYSUAF.LIS`; the `*` operator doesn't apply to that portion of a name. This is an example of the general problem that whenever the name syntax is changed all programs that interpret names must be modified. More subtly, although if the machine `ucbvax` were a gateway we could access files on a distant machine as `UCBVAX::KREMVAX::file`, it is only because the semantics of `::` explicitly permit such access. The `::` operator is implemented by passing the string after it to the remote machine, but first checking its syntax, so the file name parser must have special code for multiple `::`'s.

A Quibble about Cedar

The Cedar file system has a uniform naming syntax, just like UNIX, except that files have version numbers, separated from the file name by an exclamation mark `!`. The implementers thought that version numbers are fundamentally different components of file names and therefore deserved different syntax. But the change in syntax requires new rules to define the meaning of file names. A good test of naming schemes is whether arbitrary names constructed by the syntactic rules make sense within the rules of the system or whether their interpretation requires new semantic rules. In Cedar file, `/usr/rob/bin/cat-v!3` is clearly version 3 of `cat-v`, but what is `/usr/rob!3/bin/cat-v`?

Connecting to other machine's file systems

The IBIS remote file system on UNIX 4.2BSD names a remote file as `ucbvax:file`. Many programs don't understand this syntax; the shell (command interpreter) must be modified to make `*:file` behave as we expect, because the shell expects a slash to separate name components. Worse, by changing the syntax, the implicit semantics of the original naming scheme is lost. In the Ninth Edition name `/n/ucbvax/file` it is obvious what `file` refers to: a file in the root directory of `ucbvax`. But what is it in `ucbvax:file`? It *might* be a file in the root, but it isn't. It is a file in the *initial working directory* on the *destination* machine (`ucbvax`) of the user invoking the name on the source machine (unless it begins with `/`); its meaning depends on who is asking. The extra semantics of `:` complicate attempts to patch the syntactic problems. We might try creating a connection (called a symbolic link in UNIX) from the name `/n/ucbvax` to the name `ucbvax:`, but `/n/ucbvax/file` would then still point to a file in someone's home directory, and `/n/ucbvax/usr/wnj/file` would refer to `/usr/wnj/usr/wnj/file`. If the link evaluates to `ucbvax:/`, things work as expected, but the slash-less form of IBIS naming is made unavailable.

Part of the problem in the IBIS file system is that it is implemented outside the name space. By using a variant of the standard system call `mount`, the Ninth Edition remote file system guarantees that the syntax and semantics of names are free of surprises. For example, it is clear what `/n/ucbvax/n/kremvax/file` refers to, but what about the IBIS name `ucbvax:kremvax:file`? Where does `kremvax:file` get interpreted?

There are other ways to interpret file names like `ucbvax:file`. When using the UNIX program `uucp` to copy a local file to a remote machine, the name `ucbvax!file` refers to the file on `ucbvax` whose name is `file` prefixed by the current directory on the *source* machine. The prize goes to DECNET, however: `ucbvax::file` refers to `file` in the home directory of the 'default network user' on the destination machine, and `ucbvax"wnj password"::file` refers to `file` in `wnj`'s home directory. It is inexcusable that the password is in the file name, let alone that it is in clear text.

The story so far

In summary, there are some guidelines for constructing naming conventions, particularly for objects in a network. There should be both relative names and absolute names. Relative names are more important because, among other reasons, the root of the name space may be unknown or non-unique. The syntax should be clean and uniform; every new syntactic rule requires at least one, and usually many, semantic rules to resolve peculiarities introduced by the new syntax. If the name space is a tree or any other kind of graph, a single character should be used to separate nodes in a name.

If these guidelines are followed, names of objects in a network of machines will be easy to construct and interpret; difficult problems of networking will be completely hidden to the users and programs accessing objects in the network. If they are ignored, both users and programs must be aware of and understand the details of naming locally, globally and everywhere in between.

Principles of mail names

Now consider the other common name space, mail names. Mail names are more complex than file names, for both syntactic and semantic reasons. There are conflicting syntactic traditions, the most familiar two being the UNIX tradition and the ARPANET tradition. Also, mail names are interpreted by user programs only, with no operating system to enforce semantics. Thus, the interpretation of the name space is subject to arbitrary hackery.

Even a trivial case like the name `pjw` in the command

```
mail pjw
```

has no clear meaning. When electronic mail was invented, the name `pjw` referred to a mailbox on the local machine — the only machine to which mail could be sent. The local space of mailbox names was a small flat space. Later, when systems were connected together, there were two ways to generalize. If the computers were closely connected (that is, sharing administration), one could extend the flat name space over the whole set of machines, so that saying `mail pjw` on any of the machines gets `pjw`'s mailbox on `pjw`'s home machine. If the machines were instead loosely connected, a more attractive scheme would be to use machine names to qualify the local mailbox names: `pjw@system` in the ARPANET tradition, or `system!pjw` in the UNIX tradition. The two methods differ only in the naming and how the software decides to find the destination. In the first alternative, it looks up `pjw` in a database, while in the second it looks up `system`. In both cases, the software on the machines involved must also have a protocol for delivering mail, but that's irrelevant here. Note that neither naming scheme has anything to do with routing the message.

At this level, either of these two schemes is fairly convenient. But when we try to connect lots of systems with these flat name spaces, names must either conflict or be decorated artificially to disambiguate them. We should apply the principles of good naming to find a better solution.

Mail names specify paths within a large name space populated by systems and mailboxes rather than files, but the basic idea is the same. The question is what a path denotes. The answer depends on how the software determines what to do with the mail.

Imagine we are on machine `ucbvax` and want to send mail to `pjw@system`. There are two methods to negotiate the transaction. The first method, used by UNIX, views `system` as the name of an authority that the mail and mail address are passed to. That is, sending the mail involves a message to `system` of the form, ''I am machine `ucbvax`, here is mail for `pjw`.'' The second, that of the ARPANET, interprets `system` as the name of an authority that will say where to send the mail, as in, ''I am machine `ucbvax`, where do I send mail for `pjw`?'' The destination of these messages is found by looking up `system` in a database. (The details of sending the message are outside this discussion.) All naming schemes for mail follow some combination of these alternatives. The UNIX method uses the same mechanism to resolve names and to transmit mail; the ARPANET method resolves the names with one mechanism and uses some other, not associated with the name at all, to send the mail.

Given these two models, how do we generalize mail delivery in larger networks? For ARPANET, a mail address `user@world3.world2.world1` is interpreted by looking up `world1` and then asking it where to send mail for `world3.world2`. For UNIX, a mail address `world1!world2!world3!user` is interpreted by looking up `world1` and then sending the mail and `world2!world3!user` to it. Although the two forms sound similar, they have different problems. (And, why does ARPA use two characters when one is sufficient?)

The most common interpretation of the UNIX name is as a route, but it need have nothing to do with a route. Once the name is handed off to `world1`, it can be rewritten to correspond to the syntax of `world1`'s name space; in fact, UNIX mailers rewrite names freely. Because ARPANET names are handled differently, they cannot be rewritten: the answer to the routing question must produce the four-byte binary network address of the destination mailbox. (At least in principle, this defect may be circumvented. The response to the routing question might be a little program: ''Send the mail to A and tell it to use protocol P to send it to B'' and so forth. However, that's not how it's done in practice.)

Consider again the relation between mail names and file names. When the operating system interprets the name `/n/ucbvax/n/kremvax/file` it discovers that the directory `/n/ucbvax` refers to a remote machine, finds the server on that machine, and sends it the name `/n/kremvax/file` expecting back a handle to use the file. It does not care what the server does with the name. It does not expect to get back instructions for finding the file. It is asking for file service, not name service. Indeed were `ucbvax` a VMS machine, the server might invisibly translate `/n/kremvax/file` immediately into `KREMVAX::SYS$DISK:[NETUSER]file` to discover it on its local system. Remote file access would be harder to implement using the ARPANET scheme.

Name servers† considered speculative

Name servers don't scale well, for precisely the reason that the ARPANET name scheme doesn't scale well: the name server must understand all possible name syntaxes. When a system with a different naming convention is connected, the name server must suddenly interpret all the different syntaxes, instead of

† ''Name server'' is a noun phrase that is ambiguous. You hand a name server a name, it hands you connection information. Thus it serves connections, not names. Consider the difference between air pollution and noise pollution.

leaving the job to the new system itself. Worse, how do you connect two networks, each with its own name server? Even if the servers use the same data formats and algorithms, they might use unique identifiers that become non-unique when they are joined.

Name servers have problems on other levels, too. Who administers a name server's database? If the database is not audited frequently much of the data will be obsolete, while if the controls are too onerous, people won't bother keeping the database current. What does the database contain? Most name servers produce network addresses, but no single network reaches everywhere.

Why are mail names such a mess?

Because people keep gluing name spaces together without smoothing the syntactic differences. The result is the mail name equivalent of bastardized file names like `/n/ucbvax/UCBVAX::KREMVAX:/rob/bin/A:dos-file` where different conventions are mixed in a single string.

Relative names are important

The ARPANET people define their names to have the form

local-part`@`*domain*

where both *local-part* and *domain* are dot-separated lists of words. Domains are the generalization of what we have been calling systems; the local part is anything understood by the leftmost domain name. According to RFC 882 (''Domain Names − Concepts and Facilities''), the domains are all absolute. The dot signifying the root of the hierarchy is implicit at the right of the list of names, which makes it impossible to connect disjoint name spaces since all interpreters of names must know all names at the top level of the hierarchy. Also, for backwards compatibility, RFC 822 (''Standard for the Format of ARPA Internet Text Messages'') allows all but the leftmost of the domain names to be elided, since ''specification of a fully qualified address can become inconvenient.''

What happens in practice?

As long as software continues to deliver mail, people are unwilling to improve the state of affairs. Mailers just butt together names with their own rewriting rules, producing names like:

```
IJQ3SRA%UCLAMVS.BITNET%SU-LINDY@SU-CSLI.ARPA
```

This is the name of user `IJQ3SRA` on machine `UCLAMVS`, accessible through BITNET from machine `SU-LINDY`, which is known to `SU-CSLI` on the ARPANET. Each program that touched this name rewrote it by its own rules, although the domains proposal is intended to prevent this.

There are two domains in this name (although the syntax is wrong): `BITNET` and `ARPA`. However, `BITNET` is not a registered name, so the gateway service between BITNET and ARPANET must be made explicit in the name, requiring the invention of a new syntax character (`%`) which is translated to `@` at the gateway, because ARPANET names can only contain a single `@`. Despite the words in the standard about hierarchy, the domain space is nearly flat, so the local parts of the names carry source routing and domain transitions explicitly. To worsen matters, machines that advertise adherence to the standard in fact do not; instead the name translations that occur at gateways (such as converting `@` to `%` and rearranging the components) are at best *ad hoc*. By legislating away bad names, ARPANET has reduced the problem of networking to a still-unsolved problem. But the mailers plod resolutely on.

Standards?

It is clear that standards are necessary for electronic mail to be delivered reliably across network boundaries. What needs to be standardized is the interpretation of names, especially at network boundaries. Until such a standard exists; is syntactically and semantically clean; distributes the interpretation of names across the systems that understand them; and is adhered to, the network mail situation will not improve.

Conclusions

Doug McIlroy has observed that

> ... bad notations can stifle progress. Roman numerals hobbled mathematics for a millennium but were propagated by custom and by natural deference to authority. Today we no longer meekly accept individual authority. Instead, we have ''standards,'' impersonal imprimaturs on convention. Some standards are sound and indispensable; some simply celebrate bureaucratic littleness of mind. A harvest of gimmicks to save appearances within the standard has grown up, then gimmicks to save the appearances within the appearances. You know how each one got there: an overnight hack to paste another tumor onto a wild cancerous growth. The concern was with method, regardless of results. The result is extravagantly worse than Roman numerals: you can't read the notation right to left or left to right. As an amalgam of languages, it can't be deciphered by a native speaker of any one of them, much as if we were to switch at random places in a number between Roman and Arabic signs and between big-endian and little-endian order. But now that it all ''works'' — at least for the strong of stomach — the tumors themselves are being standardized.

> I fled, and cry'd out ''Death'';
> Hell trembled at the hideous name, and sigh'd
> From all her caves, and back resounded, ''Death.''
>
> - Milton, Paradise Lost