8½, the Plan 9 Window System

Rob Pike
rob@plan9.bell-labs.com

ABSTRACT

The Plan 9 window system, 8½, is a modest-sized program of novel design. It provides textual I/O and bitmap graphic services to both local and remote client programs by offering a multiplexed file service to those clients. It serves traditional UNIX files like `/dev/tty` as well as more unusual ones that provide access to the mouse and the raw screen. Bitmap graphics operations are provided by serving a file called `/dev/bitblt` that interprets client messages to perform raster operations. The file service that 8½ offers its clients is identical to that it uses for its own implementation, so it is fundamentally no more than a multiplexer. This architecture has some rewarding symmetries and can be implemented compactly.

Introduction

In 1989 I constructed a toy window system from only a few hundred lines of source code using a custom language and an unusual architecture involving concurrent processes [Pike89]. Although that system was rudimentary at best, it demonstrated that window systems are not inherently complicated. The following year, for the new Plan 9 distributed system [Pike92], I applied some of the lessons from that toy project to write, in C, a production-quality window system called 8½. 8½ provides, on black-and-white, grey-scale, or color displays, the services required of a modern window system, including programmability and support for remote graphics. The entire system, including the default program that runs in the window — the equivalent of `xterm` [Far89] with 'cutting and pasting' between windows — is well under 90 kilobytes of text on a Motorola 68020 processor, about half the size of the operating system kernel that supports it and a tenth the size of the X server [Sche86] *without* `xterm`.

What makes 8½ so compact? Much of the saving comes from overall simplicity: 8½ has little graphical fanciness, a concise programming interface, and a simple, fixed user interface. 8½ also makes some decisions by fiat — three-button mouse, overlapping windows, built-in terminal program and window manager, etc. — rather than trying to appeal to all tastes. Although compact, 8½ is not ascetic. It provides the fundamentals and enough extras to make them comfortable to use. The most important contributor to its small size, though, is its overall design as a file server. This structure may be applicable to window systems on traditional UNIX-like operating systems.

The small size of 8½ does not reflect reduced functionality: 8½ provides service roughly equivalent to the X window system. 8½'s clients may of course be as complex as they choose, although the tendency to mimic 8½'s design and the clean programming interface means they are not nearly as bloated as X applications.

Originally appeared, in a slightly different form, in *Proc. of the Summer 1991 USENIX Conf.,* pp. 257–265, Nashville. Note that 8½ has been replaced by `rio` (see *rio*(1)).

User's Model

8½ turns the single screen, mouse, and keyboard of the terminal (in Plan 9 terminology) or workstation (in commercial terminology) into an array of independent virtual terminals that may be textual terminals supporting a shell and the usual suite of tools or graphical applications using the full power of the bitmap screen and mouse. Text is represented in UTF, an encoding of the Unicode Standard [Pike93]. The entire programming interface is provided through reading and writing files in `/dev`.

Primarily for reasons of history and familiarity, the general model and appearance of 8½ are similar to those of `mux` [Pike88]. The right button has a short menu for controlling window creation, destruction, and placement. When a window is created, it runs the default shell, `rc` [Duff90], with standard input and output directed to the window and accessible through the file `/dev/cons` ('console'), analogous to the `/dev/tty` of UNIX. The name change represents a break with the past: Plan 9 does not provide a Teletype-style model of terminals. 8½ provides the only way most users ever access Plan 9.

Graphical applications, like ordinary programs, may be run by typing their names to the shell running in a window. This runs the application in the same window; to run the application in a new window one may use an external program, `window`, described below. For graphical applications, the virtual terminal model is extended somewhat to allow programs to perform graphical operations, access the mouse, and perform related functions by reading and writing files with suggestive names such as `/dev/mouse` and `/dev/window` multiplexed per-window much like `/dev/cons`. The implementation and semantics of these files, described below, is central to the structure of 8½.

The default program that runs in a window is familiar to users of Blit terminals [Pike83]. It is very similar to that of `mux` [Pike88], providing mouse-based editing of input and output text, the ability to scroll back to see earlier output, and so on. It also has a new feature, toggled by typing ESC, that enables the user to control when typed characters may be read by the shell or application, instead of (for example) after each newline. This feature makes the window program directly useful for many text-editing tasks such as composing mail messages before sending them.

Plan 9 and 8½

Plan 9 is a distributed system that provides support for UNIX-like applications in an environment built from distinct CPU servers, file servers, and terminals connected by a variety of networks [Pike90]. The terminals are comparable to modest workstations that, once connected to a file server over a medium-bandwidth network such as Ethernet, are self-sufficient computers running a full operating system. Unlike workstations, however, their role is just to provide an affordable multiplexed user interface to the rest of the system: they run the window system and support simple interactive tasks such as text editing. Thus they lie somewhere between workstations and X terminals in design, cost, performance, and function. (The terminals can be used for general computing, but in practice Plan 9 users do their computing on the CPU servers.) The Plan 9 terminal software, including 8½, was developed on a 68020-based machine called a Gnot and has been ported to the NeXTstation, the MIPS Magnum 3000, SGI Indigos, and Sun SPARCstations—all small workstations that we use as terminals—as well as PCs.

Heavy computations such as compilation, text processing, or scientific calculation are done on the CPU servers, which are connected to the file servers by high-bandwidth networks. For interactive work, these computations can access the terminal that instantiated them. The terminal and CPU server being used by a particular user are connected to the same file server, although over different networks; Plan 9 provides a view of the file server that is independent of location in the network.

The components of Plan 9 are connected by a common protocol based on the

sharing of files. All resources in the network are implemented as file servers; programs that wish to access them connect to them over the network and communicate using ordinary file operations. An unusual aspect of Plan 9 is that the *name space* of a process, the set of files that can be accessed by name (for example by an `open` system call) is not global to all processes on a machine; distinct processes may have distinct name spaces. The system provides methods by which processes may change their name spaces, such as the ability to *mount* a service upon an existing directory, making the files of the service visible in the directory. (This is a different operation from its UNIX namesake.) Multiple services may be mounted upon the same directory, allowing the files from multiple services to be accessed in the same directory. Options to the `mount` system call control the order of searching for files in such a *union directory.*

The most obvious example of a network resource is a file server, where permanent files reside. There are a number of unusual services, however, whose design in a different environment would likely not be file-based. Many are described elsewhere [Pike92]; some examples are the representation of processes for debugging, much like Killian's process files for the 8th edition [Kill84], and the implementation of the name/value pairs of the UNIX `exec` environment as files. User processes may also implement a file service and make it available to clients in the network, much like the 'mounted streams' in the 9th Edition [Pres90]. A typical example is a program that interprets an externally-defined file system such as that on a CD-ROM or a standard UNIX system and makes the contents available to Plan 9 programs. This design is used by all distributed applications in Plan 9, including 8½.

8½ serves a set of files in the conventional directory `/dev` with names like `cons`, `mouse`, and `screen`. Clients of 8½ communicate with the window system by reading and writing these files. For example, a client program, such as a shell, can print text by writing its standard output, which is automatically connected to `/dev/cons`, or it may open and write that file explicitly. Unlike files served by a traditional file server, however, the instance of `/dev/cons` served in each window by 8½ is a distinct file; the per-process name spaces of Plan 9 allow 8½ to provide a unique `/dev/cons` to each client. This mechanism is best illustrated by the creation of a new 8½ client.

When 8½ starts, it creates a full-duplex pipe to be the communication medium for the messages that implement the file service it will provide. One end will be shared by all the clients; the other end is held by 8½ to accept requests for I/O. When a user makes a new window using the mouse, 8½ allocates the window data structures and forks a child process. The child's name space, initially shared with the parent, is then duplicated so that changes the child makes to its name space will not affect the parent. The child then attaches its end of the communication pipe, `cfd`, to the directory `/dev` by doing a `mount` system call:

```
mount(cfd, "/dev", MBEFORE, buf)
```

This call attaches the service associated with the file descriptor `cfd` — the client end of the pipe — to the beginning of `/dev` so that the files in the new service take priority over existing files in the directory. This makes the new files `cons`, `mouse`, and so on, available in `/dev` in a way that hides any files with the same names already in place. The argument `buf` is a character string (null in this case), described below.

The client process then closes file descriptors 0, 1, and 2 and opens `/dev/cons` repeatedly to connect the standard input, output, and error files to the window's `/dev/cons`. It then does an `exec` system call to begin executing the shell in the window. This entire sequence, complete with error handling, is 33 lines of C.

The view of these events from 8½'s end of the pipe is a sequence of file protocol messages from the new client generated by the intervening operating system in response to the `mount` and `open` system calls executed by the client. The message generated by the `mount` informs 8½ that a new client has attached to the file service it

provides; 8½'s response is a unique identifier kept by the operating system and passed in all messages generated by I/O on the files derived from that `mount`. This identifier is used by 8½ to distinguish the various clients so each sees a unique `/dev/cons`; most servers do not need to make this distinction.

A process unrelated to 8½ may create windows by a variant of this mechanism. When 8½ begins, it uses a Plan 9 service to 'post' the client end of the communication pipe in a public place. A process may open that pipe and `mount` it to attach to the window system, much in the way an X client may connect to a UNIX domain socket to the server bound to the file system. The final argument to `mount` is passed through uninterpreted by the operating system. It provides a way for the client and server to exchange information at the time of the `mount`. 8½ interprets it as the dimensions of the window to be created for the new client. (In the case above, the window has been created by the time the mount occurs, and `buf` carries no information.) When the `mount` returns, the process can open the files of the new window and begin I/O to use it.

Because 8½'s interface is based on files, standard system utilities can be used to control its services. For example, its method of creating windows externally is packaged in a 16-line shell script, called `window`, the core of which is just a `mount` operation that prefixes 8½'s directory to `/dev` and runs a command passed on the argument line:

```
mount -b $'8½serv' /dev
$* < /dev/cons > /dev/cons >[2] /dev/cons &
```

The `window` program is typically employed by users to create their initial working environment when they boot the system, although it has more general possibilities.

Other basic features of the system fall out naturally from the file-based model. When the user deletes a window, 8½ sends the equivalent of a UNIX signal to the process group — the clients — in the window, removes the window from the screen, and poisons the incoming connections to the files that drive it. If a client ignores the signal and continues to write to the window, it will get I/O errors. If, on the other hand, all the processes in a window exit spontaneously, they will automatically close all connections to the window. 8½ counts references to the window's files; when none are left, it shuts down the window and removes it from the screen. As a different example, when the user hits the DEL key to generate an interrupt, 8½ writes a message to a special file, provided by Plan 9's process control interface, that interrupts all the processes in the window. In all these examples, the implementation works seamlessly across a network.

There are two valuable side effects of implementing a window system by multiplexing `/dev/cons` and other such files. First, the problem of giving a meaningful interpretation to the file `/dev/cons` (`/dev/tty`) in each window is solved automatically. To provide `/dev/cons` is the fundamental job of the window system, rather than just an awkward burden; other systems must often make special and otherwise irrelevant arrangements for `/dev/tty` to behave as expected in a window. Second, any program that can access the server, including a process on a remote machine, can access the files using standard read and write system calls to communicate with the window system, and standard open and close calls to connect to it. Again, no special arrangements need to be made for remote processes to use all the graphics facilities of 8½.

Graphical input

Of course 8½ offers more than ASCII I/O to its clients. The state of the mouse may be discovered by reading the file `/dev/mouse`, which returns a ten-byte message encoding the state of the buttons and the position of the cursor. If the mouse has not moved since the last read of `/dev/mouse`, or if the window associated with the instance of `/dev/mouse` is not the 'input focus', the read blocks.

The format of the message is:

'm'
1 byte of button state
4 bytes of x, low byte first
4 bytes of y, low byte first

As in all shared data structures in Plan 9, the order of every byte in the message is defined so all clients can execute the same code to unpack the message into a local data structure.

For keyboard input, clients can read `/dev/cons` or, if they need character-at-a-time input, `/dev/rcons` ('raw console'). There is no explicit event mechanism to help clients that need to read from multiple sources. Instead, a small (365 line) external support library can be used. It attaches a process to the various blocking input sources — mouse, keyboard, and perhaps a third user-provided file descriptor — and funnels their input into a single pipe from which may be read the various types of events in the traditional style. This package is a compromise. As discussed in a previous paper [Pike89] I prefer to free applications from event-based programming. Unfortunately, though, I see no easy way to achieve this in single-threaded C programs, and am unwilling to require all programmers to master concurrent programming. It should be noted, though, that even this compromise results in a small and easily understood interface. An example program that uses it is given near the end of the paper.

Graphical output

The file `/dev/screen` may be read by any client to recover the contents of the entire screen, such as for printing (see Figure 1). Similarly, `/dev/window` holds the contents of the current window. These are read-only files.

To perform graphics operations in their windows, client programs access `/dev/bitblt`. It implements a protocol that encodes bitmap graphics operations. Most of the messages in the protocol (there are 23 messages in all, about half to manage the multi-level fonts necessary for efficient handling of Unicode characters) are transmissions (via a write) from the client to the window system to perform a graphical operation such as a `bitblt` [PLR85] or character-drawing operation; a few include return information (recovered via a read) to the client. As with `/dev/mouse`, the `/dev/bitblt` protocol is in a defined byte order. Here, for example, is the layout of the `bitblt` message:

'b'
2 bytes of destination id
2x4 bytes of destination point
2 bytes of source id
4x4 bytes of source rectangle
2 bytes of boolean function code

The message is trivially constructed from the `bitblt` subroutine in the library, defined as

```
void bitblt(Bitmap *dst, Point dp,
            Bitmap *src, Rectangle sr, Fcode c).
```

The 'id' fields in the message indicate another property of 8½: the clients do not store the actual data for any of their bitmaps locally. Instead, the protocol provides a message to allocate a bitmap, to be stored in the server, and returns to the client an integer identifier, much like a UNIX file descriptor, to be used in operations on that bitmap. Bitmap number 0 is conventionally the client's window, analogous to standard input for file I/O. In fact, no bitmap graphics operations are executed in the client at all; they are all performed on its behalf by the server. Again, using the standard remote file

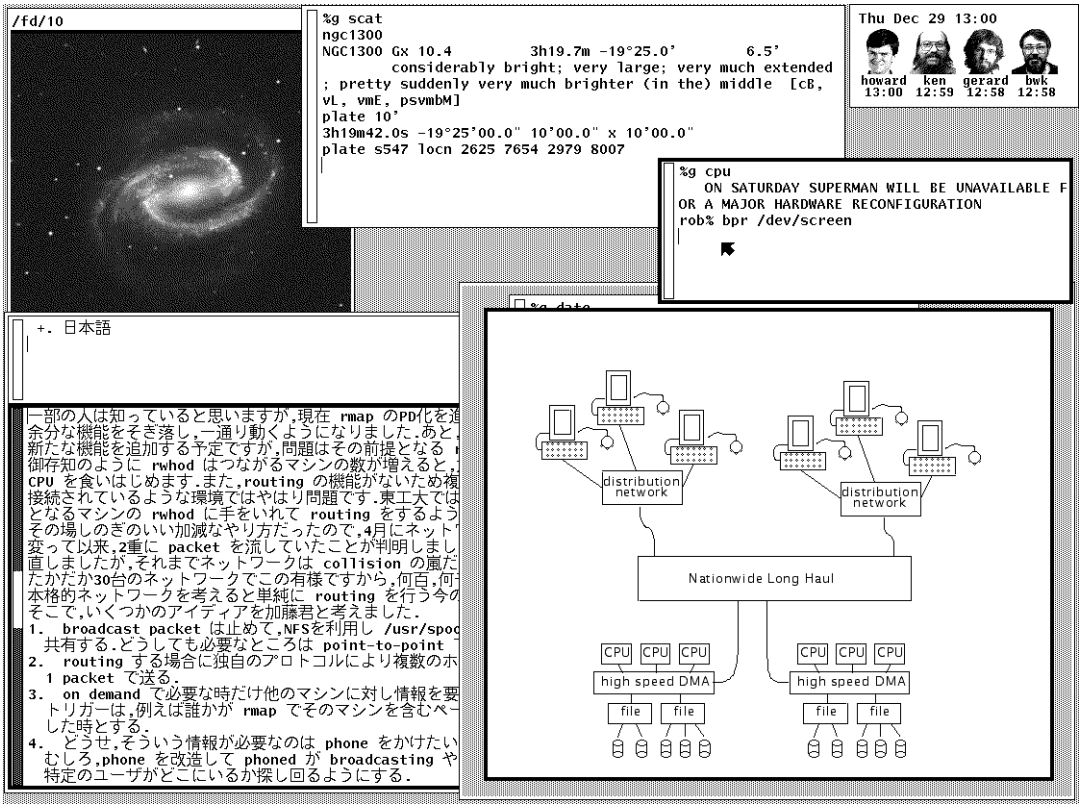

Figure 1. A representative 8½ screen, running on a NeXTstation under Plan 9 (with no NeXT software). In the upper right, a program announces the arrival of mail. In the top and left are a broswer for astronomical databases and an image of a galaxy produced by the browser. In the lower left there is a screen editor, `sam` [Pike87], editing Japanese text encoded in UTF, and in the lower right an 8½ running recursively and, inside that instantiation, a previewer for `troff` output. Underneath the faces is a small window running the command that prints the screen by passing `/dev/screen` to the bitmap printing utility.

operations in Plan 9, this permits remote machines having no graphics capability, such as the CPU server, to run graphics applications. Analogous features of the original Andrew window system [Gos86] and of X [Sche86] require more complex mechanisms.

Nor does 8½ itself operate directly on bitmaps. Instead, it calls another server to do its graphics operations for it, using an identical protocol. The operating system for the Plan 9 terminals contains an internal server that implements that protocol, exactly as does 8½, but for a single client. That server stores the actual bytes for the bitmaps and implements the fundamental bitmap graphics operations. Thus the environment in which 8½ runs has exactly the structure it provides for its clients; 8½ reproduces the environment for its clients, multiplexing the interface to keep the clients separate.

This idea of multiplexing by simulation is applicable to more than window systems, of course, and has some side effects. Since 8½ simulates its own environment for its clients, it may run in one of its own windows (see Figure 1). A useful and common application of this technique is to connect a window to a remote machine, such as a CPU server, and run the window system there so that each subwindow is automatically on the remote machine. It is also a handy way to debug a new version of the window system or to create an environment with, for example, a different default font.

Implementation

To provide graphics to its clients, 8½ mostly just multiplexes and passes through to its own server the clients' requests, occasionally rearranging the messages to maintain the fiction that the clients have unique screens (windows). To manage the overlapping windows it uses the layers model, which is handled by a separate library [Pike83a]. Thus it has little work to do and is a fairly simple program; it is dominated by a couple of switch statements to interpret the bitmap and file server protocols. The built-in window program and its associated menus and text-management support are responsible for most of the code.

The operating system's server is also compact: the version for the 68020 processor, excluding the implementation of a half dozen bitmap graphics operations, is 2295 lines of C (again, about half dealing with fonts); the graphics operations are another 2214 lines.

8½ is structured as a set of communicating coroutines, much as discussed in a 1989 paper [Pike89]. One coroutine manages the mouse, another the keyboard, and another is instantiated to manage the state of each window and associated client. When no coroutine wishes to run, 8½ reads the next file I/O request from its clients, which arrive serially on the full-duplex communication pipe. Thus 8½ is entirely synchronous.

The program source is small and compiles in about 10 seconds in our Plan 9 environment. There are ten source files and one `makefile` totaling 5100 lines. This includes the source for the window management process, the cut-and-paste terminal program, the window/file server itself, and a small coroutine library (`proc.c`). It does not include the layer library (another 1031 lines) or the library to handle the cutting and pasting of text displayed in a window (960 lines), or the general graphics support library that manages all the non-drawing aspects of graphics — arithmetic on points and rectangles, memory management, error handling, clipping, — plus fonts, events, and non-primitive drawing operations such as circles and ellipses (a final 3051 lines). Not all the pieces of these libraries are used by 8½ itself; a large part of the graphics library in particular is used only by clients. Thus it is somewhat unfair to 8½ just to sum these numbers, including the 4509 lines of support in the kernel, and arrive at a total implementation size of 14651 lines of source to implement all of 8½ from the lowest levels to the highest. But that number gives a fair measure of the complexity of the overall system.

The implementation is also efficient. 8½'s performance is competitive to X windows'. Compared using Dunwoody's and Linton's `gbench` benchmarks on the 68020, distributed with the ''X Test Suite'', circles and arcs are drawn about half as fast in 8½ as in X11 release 4 compiled with `gcc` for equivalent hardware, probably because they are currently implemented in a user library by calls to the `point` primitive. Line drawing speed is about equal between the two systems. Unicode text is drawn about the same speed by 8½ as ASCII text by X, and the `bitblt` test is runs four times faster for 8½. These numbers vary enough to caution against drawing sweeping conclusions, but they suggest that 8½'s architecture does not penalize its performance. Finally, 8½ boots in under a second and creates a new window apparently instantaneously.

An example

Here is a complete program that runs under 8½. It prints the string `"hello world"` wherever the left mouse button is depressed, and exits when the right mouse button is depressed. It also prints the string in the center of its window, and maintains that string when the window is resized.

```
#include <u.h>
#include <libc.h>
#include <libg.h>

void
ereshaped(Rectangle r)
{
    Point p;

    screen.r = r;
    bitblt(&screen, screen.r.min, &screen, r, Zero); /* clear */
    p.x = screen.r.min.x + Dx(screen.r)/2;
    p.y = screen.r.min.y + Dy(screen.r)/2;
    p = sub(p, div(strsize(font, "hello world"), 2));
    string(&screen, p, font, "hello world", S);
}

main(void)
{
    Mouse m;

    binit(0, 0, 0);     /* initialize graphics library */
    einit(Emouse);      /* initialize event library */
    ereshaped(screen.r);
    for(;;){
        m = emouse();
        if(m.buttons & RIGHTB)
            break;
        if(m.buttons & LEFTB){
            string(&screen, m.xy, font, "hello world", S);
            /* wait for release of button */
            do; while(emouse().buttons & LEFTB);
        }
    }
}
```

The complete loaded binary is a little over 26K bytes on a 68020. This program should be compared to the similar ones in the excellent paper by Rosenthal [Rose88]. (The current program does more: it also employs the mouse.) The clumsiest part is `ereshaped`, a function with a known name that is called from the event library whenever the window is reshaped or moved, as is discovered inelegantly but adequately by a special case of a mouse message. (Simple so-called expose events are not events at all in 8½; the layer library takes care of them transparently.) The lesson of this program, with deference to Rosenthal, is that if the window system is cleanly designed a toolkit should be unnecessary for simple tasks.

Status

As of 1992, 8½ is in regular daily use by almost all the 60 people in our research center. Some of those people use it to access Plan 9 itself; others use it as a front end to remote UNIX systems, much as one would use an X terminal.

Some things about 8½ may change. It would be nice if its capabilities were more easily accessible from the shell. A companion to this paper [Pike91] proposes one way to do this, but that does not include any graphics functionality. Perhaps a textual version of the `/dev/bitblt` file is a way to proceed; that would allow, for example, `awk` programs to draw graphs directly.

Can this style of window system be built on other operating systems? A major part of the design of 8½ depends on its structure as a file server. In principle this could be

done for any system that supports user processes that serve files, such as any system running NFS or AFS [Sun89, Kaza87]. One requirement, however, is 8½'s need to respond to its clients' requests out of order: if one client reads `/dev/cons` in a window with no characters to be read, other clients should be able to perform I/O in their windows, or even the same window. Another constraint is that the 8½ files are like devices, and must not be cached by the client. NFS cannot honor these requirements; AFS may be able to. Of course, other interprocess communication mechanisms such as sockets could be used as a basis for a window system. One may even argue that X's model fits into this overall scheme. It may prove easy and worthwhile to write a small 8½-like system for commercial UNIX systems to demonstrate that its merits can be won in systems other than Plan 9.

Conclusion

In conclusion, 8½ uses an unusual architecture in concert with the file-oriented interprocess communication of Plan 9 to provide network-based interactive graphics to client programs. It demonstrates that even production-quality window systems are not inherently large or complicated and may be simple to use and to program.

Acknowledgements

Helpful comments on early drafts of this paper were made by Doug Blewett, Stu Feldman, Chris Fraser, Brian Kernighan, Dennis Ritchie, and Phil Winterbottom. 8½'s support for color was added by Howard Trickey. Many of the ideas leading to 8½ were tried out in earlier, sometimes less successful, programs. I would like to thank those users who suffered through some of my previous 7½ window systems.

References

[Duff90] Tom Duff, ''Rc - A Shell for Plan 9 and UNIX systems'', Proc. of the Summer 1990 UKUUG Conf., London, July, 1990, pp. 21-33, reprinted, in a different form, in this volume.

[Far89] Far too many people, XTERM(1), Massachusetts Institute of Technology, 1989.

[Gos86] James Gosling and David Rosenthal, ''A window manager for bitmapped displays and UNIX'', in Methodology of Window Management, edited by F.R.A. Hopgood et al., Springer, 1986.

[Kaza87] Mike Kazar, ''Synchronization and Caching issues in the Andrew File System'', Tech. Rept. CMU-ITC-058, Information Technology Center, Carnegie Mellon University, June, 1987.

[Kill84] Tom Killian, ''Processes as Files'', USENIX Summer Conf. Proc., Salt Lake City June, 1984.

[Pike83] Rob Pike, ''The Blit: A Multiplexed Graphics Terminal'', Bell Labs Tech. J., V63, #8, part 2, pp. 1607-1631.

[Pike83a] Rob Pike, ''Graphics in Overlapping Bitmap Layers'', Trans. on Graph., Vol 2, #2, 135-160, reprinted in Proc. SIGGRAPH '83, pp. 331-356.

[Pike87] Rob Pike, ''The Text Editor `sam`'', Softw. - Prac. and Exp., Nov 1987, Vol 17 #11, pp. 813-845, reprinted in this volume.

[Pike88] Rob Pike, ''Window Systems Should Be Transparent'', Comp. Sys., Summer 1988, Vol 1 #3, pp. 279-296.

[Pike89] Rob Pike, ''A Concurrent Window System'', Comp. Sys., Spring 1989, Vol 2 #2, pp. 133-153.

[Pike91] Rob Pike, ''A Minimalist Global User Interface'', USENIX Summer Conf. Proc., Nashville, June, 1991.

[Pike92] Rob Pike, Dave Presotto, Ken Thompson, Howard Trickey, and Phil Winterbottom, Operating Systems Review Vol 27, #2, Apr 1993, pp. 72-76 (reprinted from Proceedings of the 5th ACM SIGOPS European Workshop, Mont Saint-Michel, 1992, Paper nº 34, and reprinted in this volume).

[Pike94] Rob Pike and Ken Thompson, ''Hello World or Καλημέρα κόσμε or こんにちは 世界'', USENIX Winter Conf. Proc., San Diego, Jan, 1993, reprinted in this volume.

[PLR85] Rob Pike, Bart Locanthi and John Reiser, ''Hardware/Software Tradeoffs for Bitmap Graphics on the Blit'', Softw. - Prac. and Exp., Feb 1985, Vol 15 #2, pp. 131-152.

[Pres90] David L. Presotto and Dennis M. Ritchie, ''Interprocess Communication in the Ninth Edition Unix System'', Softw. - Prac. and Exp., June 1990, Vol 20 #S1, pp. S1/3-S1/17.

[Rose88] David Rosenthal, ''A Simple X11 Client Program -or- How hard can it really be to write ''Hello, World''?'', USENIX Winter Conf. Proc., Dallas, Jan, 1988, pp. 229-242.

[Sche86] Robert W. Scheifler and Jim Gettys, ''The X Window System'', ACM Trans. on Graph., Vol 5 #2, pp. 79-109.

[Sun89] Sun Microsystems, NFS: Network file system protocol specification, RFC 1094, Network Information Center, SRI International, March, 1989.

Rc — The Plan 9 Shell

Tom Duff
td@plan9.bell-labs.com

ABSTRACT

Rc is a command interpreter for Plan 9 that provides similar facilities to UNIX's Bourne shell, with some small additions and less idiosyncratic syntax. This paper uses numerous examples to describe *rc*'s features, and contrasts *rc* with the Bourne shell, a model that many readers will be familiar with.

1. Introduction

Rc is similar in spirit but different in detail from UNIX's Bourne shell. This paper describes *rc*'s principal features with many small examples and a few larger ones. It assumes familiarity with the Bourne shell.

2. Simple commands

For the simplest uses *rc* has syntax familiar to Bourne-shell users. All of the following behave as expected:

```
date
cat /lib/news/build
who >user.names
who >>user.names
wc <file
echo [a-f]*.c
who | wc
who; date
vc *.c &
mk && v.out /*/bin/fb/*
rm -r junk || echo rm failed!
```

3. Quotation

An argument that contains a space or one of *rc*'s other syntax characters must be enclosed in apostrophes (').

```
rm 'odd file name'
```

An apostrophe in a quoted argument must be doubled:

```
echo 'How''s your father?'
```

4. Patterns

An unquoted argument that contains any of the characters * ? [is a pattern to be matched against file names. A * character matches any sequence of characters, ? matches any single character, and [*class*] matches any character in the `class`, unless the first character of *class* is ~, in which case the class is complemented.

The *class* may also contain pairs of characters separated by −, standing for all characters lexically between the two. The character / must appear explicitly in a pattern, as must the path name components . and .. . A pattern is replaced by a list of arguments, one for each path name matched, except that a pattern matching no names is not replaced by the empty list; rather it stands for itself.

5. Variables

UNIX's Bourne shell offers string-valued variables. *Rc* provides variables whose values are lists of arguments — that is, arrays of strings. This is the principal difference between *rc* and traditional UNIX command interpreters. Variables may be given values by typing, for example:

```
path=(. /bin)
user=td
font=/lib/font/bit/pelm/ascii.9.font
```

The parentheses indicate that the value assigned to `path` is a list of two strings. The variables `user` and `font` are assigned lists containing a single string.

The value of a variable can be substituted into a command by preceding its name with a $, like this:

```
echo $path
```

If `path` had been set as above, this would be equivalent to

```
echo . /bin
```

Variables may be subscripted by numbers or lists of numbers, like this:

```
echo $path(2)
echo $path(2 1 2)
```

These are equivalent to

```
echo /bin
echo /bin . /bin
```

There can be no space separating the variable's name from the left parenthesis; otherwise, the subscript would be considered a separate parenthesized list.

The number of strings in a variable can be determined by the $# operator. For example,

```
echo $#path
```

would print 2 for this example.

The following two assignments are subtly different:

```
empty=()
null=''
```

The first sets `empty` to a list containing no strings. The second sets `null` to a list containing a single string, but the string contains no characters.

Although these may seem like more or less the same thing (in Bourne's shell, they are indistinguishable), they behave differently in almost all circumstances. Among other things

```
echo $#empty
```

prints 0, whereas

```
echo $#null
```

prints 1.

All variables that have never been set have the value `()`.

Occasionally, it is convenient to treat a variable's value as a single string. The elements of a string are concatenated into a single string, with spaces between the elements, by the `$"` operator. Thus, if we set

```
list=(How now brown cow)
string=$"list
```

then both

```
echo $list
```

and

```
echo $string
```

cause the same output, viz:

```
How now brown cow
```

but

```
echo $#list $#string
```

will output

```
4 1
```

because `$list` has four members, but `$string` has a single member, with three spaces separating its words.

6. Arguments

When *rc* is reading its input from a file, the file has access to the arguments supplied on *rc*'s command line. The variable `$*` initially has the list of arguments assigned to it. The names `$1`, `$2`, etc. are synonyms for `$*(1)`, `$*(2)`, etc. In addition, `$0` is the name of the file from which *rc*'s input is being read.

7. Concatenation

Rc has a string concatenation operator, the caret ^, to build arguments out of pieces.

```
echo hully^gully
```

is exactly equivalent to

```
echo hullygully
```

Suppose variable `i` contains the name of a command. Then

```
vc $i^.c
vl -o $1 $i^.v
```

might compile the command's source code, leaving the result in the appropriate file.

Concatenation distributes over lists. The following

```
echo (a b c)^(1 2 3)
src=(main subr io)
cc $src^.c
```

are equivalent to

```
echo a1 b2 c3
cc main.c subr.c io.c
```

In detail, the rule is: if both operands of ^ are lists of the same non-zero number of strings, they are concatenated pairwise. Otherwise, if one of the operands is a

single string, it is concatenated with each member of the other operand in turn. Any other combination of operands is an error.

8. Free carets

User demand has dictated that *rc* insert carets in certain places, to make the syntax look more like the Bourne shell. For example, this:

```
cc -$flags $stems.c
```

is equivalent to

```
cc -^$flags $stems^.c
```

In general, *rc* will insert ^ between two arguments that are not separated by white space. Specifically, whenever one of $ ' ` follows a quoted or unquoted word, or an unquoted word follows a quoted word with no intervening blanks or tabs, an implicit ^ is inserted between the two. If an unquoted word immediately following a $ contains a character other than an alphanumeric, underscore or *, a ^ is inserted before the first such character.

9. Command substitution

It is often useful to build an argument list from the output of a command. *Rc* allows a command, enclosed in braces and preceded by a left quote, `{...}, anywhere that an argument is required. The command is executed and its standard output captured. The characters stored in the variable `ifs` are used to split the output into arguments. For example,

```
cat `{ls -tr|sed 10q}
```

will concatenate the ten oldest files in the current directory in temporal order, given the default `ifs` setting of space, tab, and newline.

10. Pipeline branching

The normal pipeline notation is general enough for almost all cases. Very occasionally it is useful to have pipelines that are not linear. Pipeline topologies more general than trees can require arbitrarily large pipe buffers, or worse, can cause deadlock. *Rc* has syntax for some kinds of non-linear but treelike pipelines. For example,

```
cmp <{old} <{new}
```

will regression-test a new version of a command. < or > followed by a command in braces causes the command to be run with its standard output or input attached to a pipe. The parent command (`cmp` in the example) is started with the other end of the pipe attached to some file descriptor or other, and with an argument that will connect to the pipe when opened (e.g., `/dev/fd/6`). Some commands are unprepared to deal with input files that turn out not to be seekable. For example `diff` needs to read its input twice.

11. Exit status

When a command exits it returns status to the program that executed it. On Plan 9 status is a character string describing an error condition. On normal termination it is empty.

Rc captures command exit status in the variable `$status`. For a simple command the value of `$status` is just as described above. For a pipeline `$status` is set to the concatenation of the statuses of the pipeline components with | characters for separators.

Rc has a several kinds of control flow, many of them conditioned by the status returned from previously executed commands. Any `$status` containing only 0's and |'s has boolean value *true*. Any other status is *false*.

12. Command grouping

A sequence of commands enclosed in {} may be used anywhere a command is required. For example:

```
{sleep 3600;echo 'Time''s up!'}&
```

will wait an hour in the background, then print a message. Without the braces,

```
sleep 3600;echo 'Time''s up!'&
```

would lock up the terminal for an hour, then print the message in the background.

13. Control flow — `for`

A command may be executed once for each member of a list by typing, for example:

```
for(i in printf scanf putchar) look $i /usr/td/lib/dw.dat
```

This looks for each of the words `printf`, `scanf` and `putchar` in the given file. The general form is

`for(`*name* `in` *list*`)` *command*

or

`for(`*name*`)` *command*

In the first case *command* is executed once for each member of *list* with that member assigned to variable *name*. If the clause "`in` *list*" is missing, "`in $*`" is assumed.

14. Conditional execution — `if`

Rc also provides a general if-statement. For example:

```
for(i in *.c) if(cpp $i >/tmp/$i) vc /tmp/$i
```

runs the C compiler on each C source program that cpp processes without error. An 'if not' statement provides a two-tailed conditional. For example:

```
for(i){
    if(test -f /tmp/$i) echo $i already in /tmp
    if not cp $i /tmp
}
```

This loops over each file in `$*`, copying to `/tmp` those that do not already appear there, and printing a message for those that do.

15. Control flow — `while`

Rc's while statement looks like this:

```
while(newer subr.v subr.c) sleep 5
```

This waits until `subr.v` is newer than `subr.c`, presumably because the C compiler finished with it.

If the controlling command is empty, the loop will not terminate. Thus,

```
while() echo y
```

emulates the *yes* command.

16. Control flow — `switch`

Rc provides a switch statement to do pattern-matching on arbitrary strings. Its general form is

```
switch(word){
case pattern ...
    commands
case pattern ...
    commands
...
}
```

Rc attempts to match the word against the patterns in each case statement in turn. Patterns are the same as for filename matching, except that / and . and .. need not be matched explicitly.

If any pattern matches, the commands following that case up to the next case (or the end of the switch) are executed, and execution of the switch is complete. For example,

```
switch($#*){
case 1
    cat >>$1
case 2
    cat >>$2 <$1
case *
    echo 'Usage: append [from] to'
}
```

is an append command. Called with one file argument, it appends its standard input to the named file. With two, the first is appended to the second. Any other number elicits an error message.

The built-in ~ command also matches patterns, and is often more concise than a switch. Its arguments are a string and a list of patterns. It sets `$status` to true if and only if any of the patterns matches the string. The following example processes option arguments for the *man*(1) command:

```
opt=()
while(~ $1 -* [1-9] 10){
    switch($1){
    case [1-9] 10
        sec=$1 secn=$1
    case -f
        c=f s=f
    case -[qwnt]
        cmd=$1
    case -T*
        T=$1
    case -*
        opt=($opt $1)
    }
    shift
}
```

17. Functions

Functions may be defined by typing

```
fn name { commands }
```

Subsequently, whenever a command named *name* is encountered, the remainder of

the command's argument list will assigned to `$*` and *rc* will execute the *commands*. The value of `$*` will be restored on completion. For example:

```
fn g {
    grep $1 *.[hcyl]
}
```

defines `g` *pattern* to look for occurrences of *pattern* in all program source files in the current directory.

Function definitions are deleted by writing

`fn` *name*

with no function body.

18. Command execution

Rc does one of several things to execute a simple command. If the command name is the name of a function defined using `fn`, the function is executed. Otherwise, if it is the name of a built-in command, the built-in is executed directly by *rc*. Otherwise, directories mentioned in the variable `$path` are searched until an executable file is found. Extensive use of the `$path` variable is discouraged in Plan 9. Instead, use the default (`. /bin`) and bind what you need into `/bin`.

19. Built-in commands

Several commands are executed internally by *rc* because they are difficult to implement otherwise.

`. [-i]` *file ...*
Execute commands from *file*. `$*` is set for the duration to the reminder of the argument list following *file*. `$path` is used to search for *file*. Option `-i` indicates interactive input — a prompt (found in `$prompt`) is printed before each command is read.

`builtin` *command ...*
Execute *command* as usual except that any function named *command* is ignored. For example,

```
fn cd{
    builtin cd $* && pwd
}
```

defines a replacement for the `cd` built-in (see below) that announces the full name of the new directory.

`cd [`*dir*`]`
Change the current directory to *dir*. The default argument is `$home`. `$cdpath` is a list of places in which to search for *dir*.

`eval [`*arg ...*`]`
The arguments are concatenated (separated by spaces) into a string, read as input to *rc*, and executed. For example,

```
x='$y'
y=Doody
eval echo Howdy, $x
```

would echo

```
Howdy, Doody
```

since the arguments of `eval` would be

```
echo Howdy, $y
```

after substituting for `$x`.

`exec [`*command ...*`]`
Rc replaces itself with the given *command*. This is like a *goto* — *rc* does not wait for the command to exit, and does not return to read any more commands.

`exit [`*status*`]`
Rc exits immediately with the given status. If none is given, the current value of `$status` is used.

`flag` *f* `[+-]`
This command manipulates and tests the command line flags (described below).

```
flag f +
```

sets flag *f*.

```
flag f -
```

clears flag *f*.

```
flag f
```

tests flag *f*, setting `$status` appropriately. Thus

```
if(flag x) flag v +
```

sets the `-v` flag if the `-x` flag is already set.

`rfork [nNeEsfF]`
This uses the Plan 9 *rfork* system entry to put *rc* into a new process group with the following attributes:

Flag	Name	Function
`n`	RFNAMEG	Make a copy of the parent's name space
`N`	RFCNAMEG	Start with a new, empty name space
`e`	RFENVG	Make a copy of the parent's environment
`E`	RFCENVG	Start with a new, empty environment
`s`	RFNOTEG	Make a new note group
`f`	RFFDG	Make a copy of the parent's file descriptor space
`F`	RFCFDG	Make a new, empty file descriptor space

Section *fork*(2) of the Programmer's Manual describes these attributes in more detail.

`shift [`*n*`]`
Delete the first *n* (default 1) elements of `$*`.

`wait [`*pid*`]`
Wait for the process with the given *pid* to exit. If no *pid* is given, all outstanding processes are waited for.

`whatis` *name ...*
Print the value of each *name* in a form suitable for input to *rc*. The output is an assignment to a variable, the definition of a function, a call to `builtin` for a built-in command, or the path name of a binary program. For example,

```
whatis path g cd who
```

might print

```
path=(. /bin)
fn g {gre -e $1 *.[hycl]}
builtin cd
/bin/who
```

~ *subject pattern ...*

The *subject* is matched against each *pattern* in turn. On a match, `$status` is set to true. Otherwise, it is set to `'no match'`. Patterns are the same as for filename matching. The *patterns* are not subjected to filename replacement before the ~ command is executed, so they need not be enclosed in quotation marks, unless of course, a literal match for * [or ? is required. For example

```
~ $1 ?
```

matches any single character, whereas

```
~ $1 '?'
```

only matches a literal question mark.

20. Advanced I/O Redirection

Rc allows redirection of file descriptors other than 0 and 1 (standard input and output) by specifying the file descriptor in square brackets [ ] after the < or >. For example,

```
vc junk.c >[2]junk.diag
```

saves the compiler's diagnostics from standard error in `junk.diag`.

File descriptors may be replaced by a copy, in the sense of *dup*(2), of an already-open file by typing, for example

```
vc junk.c >[2=1]
```

This replaces file descriptor 2 with a copy of file descriptor 1. It is more useful in conjunction with other redirections, like this

```
vc junk.c >junk.out >[2=1]
```

Redirections are evaluated from left to right, so this redirects file descriptor 1 to `junk.out`, then points file descriptor 2 at the same file. By contrast,

```
vc junk.c >[2=1] >junk.out
```

redirects file descriptor 2 to a copy of file descriptor 1 (presumably the terminal), and then directs file descriptor 1 to a file. In the first case, standard and diagnostic output will be intermixed in `junk.out`. In the second, diagnostic output will appear on the terminal, and standard output will be sent to the file.

File descriptors may be closed by using the duplication notation with an empty right-hand side. For example,

```
vc junk.c >[2=]
```

will discard diagnostics from the compilation.

Arbitrary file descriptors may be sent through a pipe by typing, for example,

```
vc junk.c |[2] grep -v '^$'
```

This deletes blank lines from the C compiler's error output. Note that the output of `grep` still appears on file descriptor 1.

Occasionally you may wish to connect the input side of a pipe to some file descriptor other than zero. The notation

```
cmd1 |[5=19] cmd2
```

creates a pipeline with `cmd1`'s file descriptor 5 connected through a pipe to `cmd2`'s file descriptor 19.

21. Here documents

Rc procedures may include data, called ''here documents'', to be provided as input to commands, as in this version of the *tel* command

```
for(i) grep $i <<!
...
tor 2T-402 2912
kevin 2C-514 2842
bill 2C-562 7214
...
!
```

A here document is introduced by the redirection symbol <<, followed by an arbitrary EOF marker (`!` in the example). Lines following the command, up to a line containing only the EOF marker are saved in a temporary file that is connected to the command's standard input when it is run.

Rc does variable substitution in here documents. The following command:

```
ed $3 <<EOF
g/$1/s//$2/g
w
EOF
```

changes all occurrences of `$1` to `$2` in file `$3`. To include a literal `$` in a here document, type `$$`. If the name of a variable is followed immediately by ^, the caret is deleted.

Variable substitution can be entirely suppressed by enclosing the EOF marker following << in quotation marks, as in `<<'EOF'`.

Here documents may be provided on file descriptors other than 0 by typing, for example,

```
cmd <<[4]End
...
End
```

If a here document appears within a compound block, the contents of the document must be after the whole block:

```
for(i in $*){
        mail $i <<EOF
}
words to live by
EOF
```

22. Catching Notes

Rc scripts normally terminate when an interrupt is received from the terminal. A function with the name of a UNIX signal, in lower case, is defined in the usual way, but called when *rc* receives the corresponding note. The *notify*(2) section of the Programmer's Manual discusses notes in some detail. Notes of interest are:

`sighup`
The note was 'hangup'. Plan 9 sends this when the terminal has disconnected from *rc*.

`sigint`
The note was 'interrupt', usually sent when the interrupt character (ASCII DEL) is typed on the terminal.

`sigterm`
The note was 'kill', normally sent by *kill*(1).

`sigexit`
An artificial note sent when *rc* is about to exit.

As an example,

```
fn sigint{
    rm /tmp/junk
    exit
}
```

sets a trap for the keyboard interrupt that removes a temporary file before exiting.

Notes will be ignored if the note routine is set to `{}`. Signals revert to their default behavior when their handlers' definitions are deleted.

23. Environment

The environment is a list of name-value pairs made available to executing binaries. On Plan 9, the environment is stored in a file system named `#e`, normally mounted on `/env`. The value of each variable is stored in a separate file, with components terminated by zero bytes. (The file system is maintained entirely in core, so no disk or network access is involved.) The contents of `/env` are shared on a per-process group basis – when a new process group is created it effectively attaches `/env` to a new file system initialized with a copy of the old one. A consequence of this organization is that commands can change environment entries and see the changes reflected in *rc*.

Functions also appear in the environment, named by prefixing `fn#` to their names, like `/env/fn#roff`.

24. Local Variables

It is often useful to set a variable for the duration of a single command. An assignment followed by a command has this effect. For example

```
a=global
a=local echo $a
echo $a
```

will print

```
local
global
```

This works even for compound commands, like

```
f=/fairly/long/file/name {
    { wc $f; spell $f; diff $f.old $f } |
      pr -h 'Facts about '$f | lp -dfn
}
```

25. Examples — *cd, pwd*

Here is a pair of functions that provide enhanced versions of the standard `cd` and `pwd` commands. (Thanks to Rob Pike for these.)

```
ps1='% '        # default prompt
tab='	'       # a tab character
fn cd{
  builtin cd $1 &&
  switch($#*){
  case 0
    dir=$home
    prompt=($ps1 $tab)
  case *
    switch($1)
    case /*
      dir=$1
      prompt=(`{basename `{pwd}}^$ps1 $tab)
    case */* ..*
      dir=()
      prompt=(`{basename `{pwd}}^$ps1 $tab)
    case *
      dir=()
      prompt=($1^$ps1 $tab)
    }
  }
}
fn pwd{
  if(~ $#dir 0)
    dir=`{/bin/pwd}
  echo $dir
}
```

Function `pwd` is a version of the standard `pwd` that caches its value in variable `$dir`, because the genuine `pwd` can be quite slow to execute. (Recent versions of Plan 9 have very fast implementations of `pwd`, reducing the advantage of the `pwd` function.)

Function `cd` calls the `cd` built-in, and checks that it was successful. If so, it sets `$dir` and `$prompt`. The prompt will include the last component of the current directory (except in the home directory, where it will be null), and `$dir` will be reset either to the correct value or to `()`, so that the `pwd` function will work correctly.

26. Examples — *man*

The *man* command prints pages of the Programmer's Manual. It is called, for example, as

```
man 2 sinh
man rc
man -t cat
```

In the first case, the page for *sinh* in section 2 is printed. In the second case, the manual page for *rc* is printed. Since no manual section is specified, all sections are searched for the page, and it is found in section 1. In the third case, the page for *cat* is typeset (the `-t` option).

```
cd /sys/man || {
  echo $0: No manual! >[1=2]
  exit 1
}
NT=n  # default nroff
s='*' # section, default try all
for(i) switch($i){
case -t
  NT=t
case -n
  NT=n
case -*
  echo Usage: $0 '[-nt] [section] page ...' >[1=2]
  exit 1
case [1-9] 10
  s=$i
case *
  eval 'pages='$s/$i
  for(page in $pages){
    if(test -f $page)
      $NT^roff -man $page
    if not
      echo $0: $i not found >[1=2]
  }
}
```

Note the use of `eval` to make a list of candidate manual pages. Without `eval`, the `*` stored in `$s` would not trigger filename matching — it's enclosed in quotation marks, and even if it weren't, it would be expanded when assigned to `$s`. Eval causes its arguments to be re-processed by *rc*'s parser and interpreter, effectively delaying evaluation of the `*` until the assignment to `$pages`.

27. Examples — *holmdel*

The following *rc* script plays the deceptively simple game *holmdel*, in which the players alternately name Bell Labs locations, the winner being the first to mention Holmdel.

This script is worth describing in detail (rather, it would be if it weren't so silly.)

Variable `$t` is an abbreviation for the name of a temporary file. Including `$pid`, initialized by *rc* to its process-id, in the names of temporary files insures that their names won't collide, in case more than one instance of the script is running at a time.

Function `read`'s argument is the name of a variable into which a line gathered from standard input is read. `$ifs` is set to just a newline. Thus `read`'s input is not split apart at spaces, but the terminating newline is deleted.

A handler is set to catch `sigint`, `sigquit`, and `sighup`, and the artificial `sigexit` signal. It just removes the temporary file and exits.

The temporary file is initialized from a here document containing a list of Bell Labs locations, and the main loop starts.

First, the program guesses a location (in `$lab`) using the `fortune` program to pick a random line from the location list. It prints the location, and if it guessed Holmdel, prints a message and exits.

Then it uses the `read` function to get lines from standard input and validity-check them until it gets a legal name. Note that the condition part of a `while` can be a compound command. Only the exit status of the last command in the

```
t=/tmp/holmdel$pid
fn read{
        $1=`{awk '{print;exit}'}
}
ifs='
'       # just a newline
fn sigexit sigint sigquit sighup{
        rm -f $t
        exit
}
cat <<'!' >$t
Allentown
Atlanta
Cedar Crest
Chester
Columbus
Elmhurst
Fullerton
Holmdel
Indian Hill
Merrimack Valley
Morristown
Neptune
Piscataway
Reading
Short Hills
South Plainfield
Summit
Whippany
West Long Branch
!
while(){
   lab=`{fortune $t}
   echo $lab
   if(~ $lab Holmdel){
      echo You lose.
      exit
   }
   while(read lab; ! grep -i -s $lab $t) echo No such location.
   if(~ $lab [hH]olmdel){
      echo You win.
      exit
   }
}
```

sequence is checked.

Again, if the result is Holmdel, it prints a message and exits. Otherwise it goes back to the top of the loop.

28. Design Principles

Rc draws heavily from Steve Bourne's `/bin/sh`. Any successor of the Bourne shell is bound to suffer in comparison. I have tried to fix its best-acknowledged shortcomings and to simplify things wherever possible, usually by omitting inessential features. Only when irresistibly tempted have I introduced novel ideas. Obviously I have tinkered extensively with Bourne's syntax.

The most important principle in *rc*'s design is that it's not a macro processor. Input is never scanned more than once by the lexical and syntactic analysis code

(except, of course, by the `eval` command, whose *raison d'être* is to break the rule).

Bourne shell scripts can often be made to run wild by passing them arguments containing spaces. These will be split into multiple arguments using `IFS`, often at inopportune times. In *rc*, values of variables, including command line arguments, are not re-read when substituted into a command. Arguments have presumably been scanned in the parent process, and ought not to be re-read.

Why does Bourne re-scan commands after variable substitution? He needs to be able to store lists of arguments in variables whose values are character strings. If we eliminate re-scanning, we must change the type of variables, so that they can explicitly carry lists of strings.

This introduces some conceptual complications. We need a notation for lists of words. There are two different kinds of concatenation, for strings — `$a^$b`, and lists — `($a $b)`. The difference between `()` and `''` is confusing to novices, although the distinction is arguably sensible — a null argument is not the same as no argument.

Bourne also rescans input when doing command substitution. This is because the text enclosed in back-quotes is not a string, but a command. Properly, it ought to be parsed when the enclosing command is, but this makes it difficult to handle nested command substitutions, like this:

```
size=‘wc -l \‘ls -t|sed 1q\‘‘
```

The inner back-quotes must be escaped to avoid terminating the outer command. This can get much worse than the above example; the number of \'s required is exponential in the nesting depth. *Rc* fixes this by making the backquote a unary operator whose argument is a command, like this:

```
size=‘{wc -l ‘{ls -t|sed 1q}}
```

No escapes are ever required, and the whole thing is parsed in one pass.

For similar reasons *rc* defines signal handlers as though they were functions, instead of associating a string with each signal, as Bourne does, with the attendant possibility of getting a syntax error message in response to typing the interrupt character. Since *rc* parses input when typed, it reports errors when you make them.

For all this trouble, we gain substantial semantic simplifications. There is no need for the distinction between `$*` and `$@`. There is no need for four types of quotation, nor the extremely complicated rules that govern them. In *rc* you use quotation marks when you want a syntax character to appear in an argument, or an argument that is the empty string, and at no other time. `IFS` is no longer used, except in the one case where it was indispensable: converting command output into argument lists during command substitution.

This also avoids an important UNIX security hole. In UNIX, the *system* and *popen* functions call `/bin/sh` to execute a command. It is impossible to use either of these routines with any assurance that the specified command will be executed, even if the caller of *system* or *popen* specifies a full path name for the command. This can be devastating if it occurs in a set-userid program. The problem is that `IFS` is used to split the command into words, so an attacker can just set `IFS=/` in his environment and leave a Trojan horse named `usr` or `bin` in the current working directory before running the privileged program. *Rc* fixes this by never rescanning input for any reason.

Most of the other differences between *rc* and the Bourne shell are not so serious. I eliminated Bourne's peculiar forms of variable substitution, like

```
echo ${a=b} ${c-d} ${e?error}
```

because they are little used, redundant and easily expressed in less abstruse terms. I deleted the builtins `export`, `readonly`, `break`, `continue`, `read`, `return`, `set`, `times` and `unset` because they seem redundant or only marginally useful.

Where Bourne's syntax draws from Algol 68, *rc*'s is based on C or Awk. This is harder to defend. I believe that, for example

```
if(test -f junk) rm junk
```

is better syntax than

```
if test -f junk; then rm junk; fi
```

because it is less cluttered with keywords, it avoids the semicolons that Bourne requires in odd places, and the syntax characters better set off the active parts of the command.

The one bit of large-scale syntax that Bourne unquestionably does better than *rc* is the `if` statement with `else` clause. *Rc*'s `if` has no terminating `fi`-like bracket. As a result, the parser cannot tell whether or not to expect an `else` clause without looking ahead in its input. The problem is that after reading, for example

```
if(test -f junk) echo junk found
```

in interactive mode, *rc* cannot decide whether to execute it immediately and print `$prompt(1)`, or to print `$prompt(2)` and wait for the `else` to be typed. In the Bourne shell, this is not a problem, because the `if` command must end with `fi`, regardless of whether it contains an `else` or not.

Rc's admittedly feeble solution is to declare that the `else` clause is a separate statement, with the semantic proviso that it must immediately follow an `if`, and to call it `if not` rather than `else`, as a reminder that something odd is going on. The only noticeable consequence of this is that the braces are required in the construction

```
for(i){
    if(test -f $i) echo $i found
    if not echo $i not found
}
```

and that *rc* resolves the ''dangling else'' ambiguity in opposition to most people's expectations.

It is remarkable that in the four most recent editions of the UNIX system programmer's manual the Bourne shell grammar described in the manual page does not admit the command `who|wc`. This is surely an oversight, but it suggests something darker: nobody really knows what the Bourne shell's grammar is. Even examination of the source code is little help. The parser is implemented by recursive descent, but the routines corresponding to the syntactic categories all have a flag argument that subtly changes their operation depending on the context. *Rc*'s parser is implemented using *yacc*, so I can say precisely what the grammar is.

29. Acknowledgements

Rob Pike, Howard Trickey and other Plan 9 users have been insistent, incessant sources of good ideas and criticism. Some examples in this document are plagiarized from [Bourne], as are most of *rc*'s good features.

30. Reference

S. R. Bourne, UNIX Time-Sharing System: The UNIX Shell, Bell System Technical Journal, Volume 57 number 6, July-August 1978

The Text Editor sam

Rob Pike
rob@plan9.bell-labs.com

ABSTRACT

Sam is an interactive multi-file text editor intended for bitmap displays. A textual command language supplements the mouse-driven, cut-and-paste interface to make complex or repetitive editing tasks easy to specify. The language is characterized by the composition of regular expressions to describe the structure of the text being modified. The treatment of files as a database, with changes logged as atomic transactions, guides the implementation and makes a general 'undo' mechanism straightforward.

Sam is implemented as two processes connected by a low-bandwidth stream, one process handling the display and the other the editing algorithms. Therefore it can run with the display process in a bitmap terminal and the editor on a local host, with both processes on a bitmap-equipped host, or with the display process in the terminal and the editor in a remote host. By suppressing the display process, it can even run without a bitmap terminal.

This paper is reprinted from Software—Practice and Experience, Vol 17, number 11, pp. 813-845, November 1987. The paper has not been updated for the Plan 9 manuals. Although Sam has not changed much since the paper was written, the system around it certainly has. Nonetheless, the description here still stands as the best introduction to the editor.

Introduction

Sam is an interactive text editor that combines cut-and-paste interactive editing with an unusual command language based on the composition of regular expressions. It is written as two programs: one, the 'host part,' runs on a UNIX system and implements the command language and provides file access; the other, the 'terminal part,' runs asynchronously on a machine with a mouse and bitmap display and supports the display and interactive editing. The host part may be even run in isolation on an ordinary terminal to edit text using the command language, much like a traditional line editor, without assistance from a mouse or display. Most often, the terminal part runs on a Blit[1] terminal (actually on a Teletype DMD 5620, the production version of the Blit), whose host connection is an ordinary 9600 bps RS232 link; on the SUN computer the host and display processes run on a single machine, connected by a pipe.

Sam edits uninterpreted ASCII text. It has no facilities for multiple fonts, graphics or tables, unlike MacWrite,[2] Bravo,[3] Tioga[4] or Lara.[5] Also unlike them, it has a rich command language. (Throughout this paper, the phrase *command language* refers to textual commands; commands activated from the mouse form the *mouse language*.) Sam developed as an editor for use by programmers, and tries to join the styles of the UNIX text editor ed[6,7] with that of interactive cut-and-paste editors by providing a

comfortable mouse-driven interface to a program with a solid command language driven by regular expressions. The command language developed more than the mouse language, and acquired a notation for describing the structure of files more richly than as a sequence of lines, using a dataflow-like syntax for specifying changes.

The interactive style was influenced by `jim`,[1] an early cut-and-paste editor for the Blit, and by `mux`,[8] the Blit window system. `Mux` merges the original Blit window system, `mpx`,[1] with cut-and-paste editing, forming something like a multiplexed version of `jim` that edits the output of (and input to) command sessions rather than files.

The first part of this paper describes the command language, then the mouse language, and explains how they interact. That is followed by a description of the implementation, first of the host part, then of the terminal part. A principle that influenced the design of `sam` is that it should have no explicit limits, such as upper limits on file size or line length. A secondary consideration is that it be efficient. To honor these two goals together requires a method for efficiently manipulating huge strings (files) without breaking them into lines, perhaps while making thousands of changes under control of the command language. `Sam`'s method is to treat the file as a transaction database, implementing changes as atomic updates. These updates may be unwound easily to 'undo' changes. Efficiency is achieved through a collection of caches that minimizes disc traffic and data motion, both within the two parts of the program and between them.

The terminal part of `sam` is fairly straightforward. More interesting is how the two halves of the editor stay synchronized when either half may initiate a change. This is achieved through a data structure that organizes the communications and is maintained in parallel by both halves.

The last part of the paper chronicles the writing of `sam` and discusses the lessons that were learned through its development and use.

The paper is long, but is composed largely of two papers of reasonable length: a description of the user interface of `sam` and a discussion of its implementation. They are combined because the implementation is strongly influenced by the user interface, and vice versa.

The Interface

`Sam` is a text editor for multiple files. File names may be provided when it is invoked:

```
sam file1 file2 ...
```

and there are commands to add new files and discard unneeded ones. Files are not read until necessary to complete some command. Editing operations apply to an internal copy made when the file is read; the UNIX file associated with the copy is changed only by an explicit command. To simplify the discussion, the internal copy is here called a *file*, while the disc-resident original is called a *disc file.*

`Sam` is usually connected to a bitmap display that presents a cut-and-paste editor driven by the mouse. In this mode, the command language is still available: text typed in a special window, called the `sam` *window,* is interpreted as commands to be executed in the current file. Cut-and-paste editing may be used in any window — even in the `sam` window to construct commands. The other mode of operation, invoked by starting `sam` with the option `-d` (for 'no download'), does not use the mouse or bitmap display, but still permits editing using the textual command language, even on an ordinary terminal, interactively or from a script.

The following sections describe first the command language (under `sam -d` and in the `sam` window), and then the mouse interface. These two languages are nearly independent, but connect through the *current text,* described below.

The Command Language

A file consists of its contents, which are an array of characters (that is, a string); the *name* of the associated disc file; the *modified bit* that states whether the contents match those of the disc file; and a substring of the contents, called the *current text* or *dot* (see Figures 1 and 2). If the current text is a null string, dot falls between characters. The *value* of dot is the location of the current text; the *contents* of dot are the characters it contains. Sam imparts to the text no two-dimensional interpretation such as columns or fields; text is always one-dimensional. Even the idea of a 'line' of text as understood by most UNIX programs — a sequence of characters terminated by a newline character — is only weakly supported.

The *current file* is the file to which editing commands refer. The current text is therefore dot in the current file. If a command doesn't explicitly name a particular file or piece of text, the command is assumed to apply to the current text. For the moment, ignore the presence of multiple files and consider editing a single file.

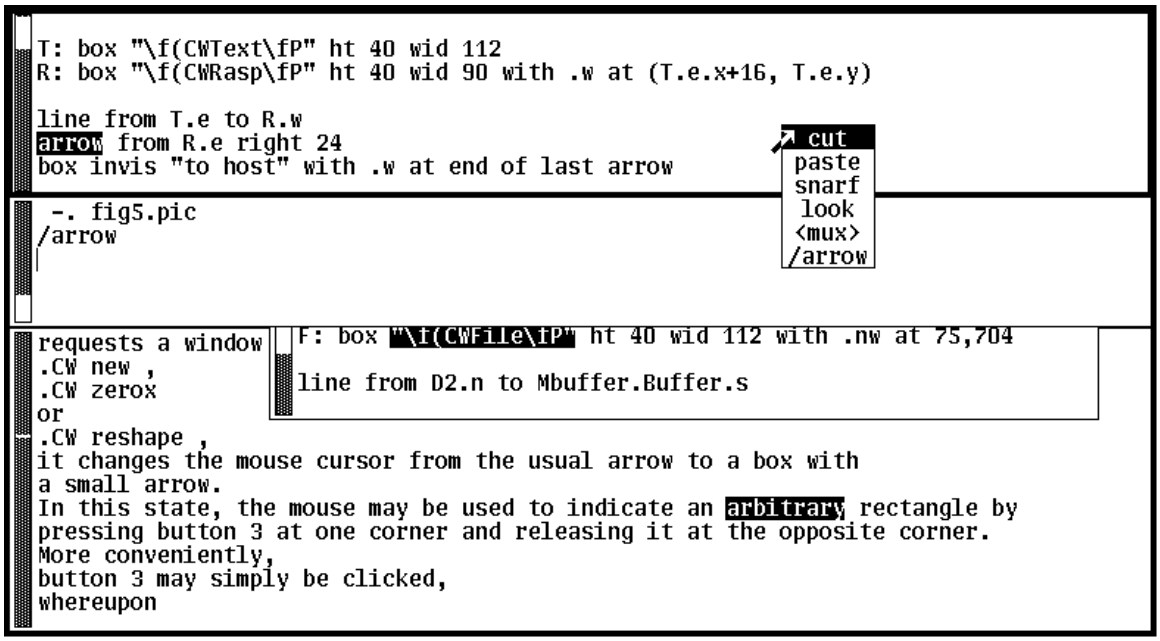

Figure 1. A typical sam *screen, with the editing menu presented. The* sam *(command language) window is in the middle, with file windows above and below. (The user interface makes it easy to create these abutting windows.) The partially obscured window is a third file window. The uppermost window is that to which typing and mouse operations apply, as indicated by its heavy border. Each window has its current text highlighted in reverse video. The* sam *window's current text is the null string on the last visible line, indicated by a vertical bar. See also Figure 2.*

Commands have one-letter names. Except for non-editing commands such as writing the file to disc, most commands make some change to the text in dot and leave dot set to the text resulting from the change. For example, the delete command, `d`, deletes the text in dot, replacing it by the null string and setting dot to the result. The change command, `c`, replaces dot by text delimited by an arbitrary punctuation character, conventionally a slash. Thus,

```
c/Peter/
```

replaces the text in dot by the string `Peter`. Similarly,

```
a/Peter/
```

(append) adds the string after dot, and

```
i/Peter/
```

(insert) inserts before dot. All three leave dot set to the new text, `Peter`.

Newlines are part of the syntax of commands: the newline character lexically terminates a command. Within the inserted text, however, newlines are never implicit. But since it is often convenient to insert multiple lines of text, `sam` has a special syntax for that case:

```
a
some lines of text
to be inserted in the file,
terminated by a period
on a line by itself
.
```

In the one-line syntax, a newline character may be specified by a C-like escape, so

```
c/\n/
```

replaces dot by a single newline character.

`Sam` also has a substitute command, `s`:

s/*expression*/*replacement*/

substitutes the replacement text for the first match, in dot, of the regular expression. Thus, if dot is the string `Peter`, the command

```
s/t/st/
```

changes it to `Pester`. In general, `s` is unnecessary, but it was inherited from `ed` and it has some convenient variations. For instance, the replacement text may include the matched text, specified by &:

```
s/Peter/Oh, &, &, &, &!/
```

There are also three commands that apply programs to text:

< *UNIX program*

replaces dot by the output of the UNIX program. Similarly, the > command runs the program with dot as its standard input, and | does both. For example,

```
| sort
```

replaces dot by the result of applying the standard sorting utility to it. Again, newlines have no special significance for these `sam` commands. The text acted upon and resulting from these commands is not necessarily bounded by newlines, although for connection with UNIX programs, newlines may be necessary to obey conventions.

One more command: `p` prints the contents of dot. Table I summarizes `sam`'s commands.

The value of dot may be changed by specifying an *address* for the command. The simplest address is a line number:

```
3
```

refers to the third line of the file, so

```
3d
```

deletes the third line of the file, and implicitly renumbers the lines so the old line 4 is now numbered 3. (This is one of the few places where `sam` deals with lines directly.)

Table I. Sam commands

Text commands	
a/*text*/	Append text after dot
c/*text*/	Change text in dot
i/*text*/	Insert text before dot
d	Delete text in dot
s/*regexp*/*text*/	Substitute text for match of regular expression in dot
m *address*	Move text in dot after address
t *address*	Copy text in dot after address
Display commands	
p	Print contents of dot
=	Print value (line numbers and character numbers) of dot
File commands	
b *file-list*	Set current file to first file in list that sam has in menu
B *file-list*	Same as b, but load new files
n	Print menu lines of all files
D *file-list*	Delete named files from sam
I/O commands	
e *filename*	Replace file with named disc file
r *filename*	Replace dot by contents of named disc file
w *filename*	Write file to named disc file
f *filename*	Set file name and print new menu line
< *UNIX-command*	Replace dot by standard output of command
> *UNIX-command*	Send dot to standard input of command
| *UNIX-command*	Replace dot by result of command applied to dot
! *UNIX-command*	Run the command
Loops and conditionals	
x/*regexp*/ *command*	For each match of regexp, set dot and run command
y/*regexp*/ *command*	Between adjacent matches of regexp, set dot and run command
X/*regexp*/ *command*	Run command in each file whose menu line matches regexp
Y/*regexp*/ *command*	Run command in each file whose menu line does not match
g/*regexp*/ *command*	If dot contains a match of regexp, run command
v/*regexp*/ *command*	If dot does not contain a match of regexp, run command
Miscellany	
k	Set address mark to value of dot
q	Quit
u *n*	Undo last *n* (default 1) changes
{ }	Braces group commands

Line 0 is the null string at the beginning of the file. If a command consists of only an address, a p command is assumed, so typing an unadorned 3 prints line 3 on the terminal. There are a couple of other basic addresses: a period addresses dot itself; and a dollar sign ($) addresses the null string at the end of the file.

An address is always a single substring of the file. Thus, the address 3 addresses the characters after the second newline of the file through the third newline of the file. A *compound address* is constructed by the comma operator

address1,*address2*

and addresses the substring of the file from the beginning of *address1* to the end of *address2*. For example, the command 3,5p prints the third through fifth lines of the file and .,$d deletes the text from the beginning of dot to the end of the file.

These addresses are all absolute positions in the file, but sam also has relative addresses, indicated by + or −. For example,

```
$-3
```

is the third line before the end of the file and

```
.+1
```

is the line after dot. If no address appears to the left of the + or −, dot is assumed; if nothing appears to the right, 1 is assumed. Therefore, .+1 may be abbreviated to just a plus sign.

The + operator acts relative to the end of its first argument, while the − operator acts relative to the beginning. Thus .+1 addresses the first line after dot, .− addresses the first line before dot, and +− refers to the line containing the end of dot. (Dot may span multiple lines, and + selects the line after the end of dot, then − backs up one line.)

The final type of address is a regular expression, which addresses the text matched by the expression. The expression is enclosed in slashes, as in

/expression/

The expressions are the same as those in the UNIX program egrep,[6,7] and include closures, alternations, and so on. They find the *leftmost longest* string that matches the expression, that is, the first match after the point where the search is started, and if more than one match begins at the same spot, the longest such match. (I assume familiarity with the syntax for regular expressions in UNIX programs.[9]) For example,

```
/x/
```

matches the next x character in the file,

```
/xx*/
```

matches the next run of one or more x's, and

```
/x|Peter/
```

matches the next x or Peter. For compatibility with other UNIX programs, the 'any character' operator, a period, does not match a newline, so

```
/.*/
```

matches the text from dot to the end of the line, but excludes the newline and so will not match across the line boundary.

Regular expressions are always relative addresses. The direction is forwards by default, so /Peter/ is really an abbreviation for +/Peter/. The search can be reversed with a minus sign, so

```
-/Peter/
```

finds the first `Peter` before dot. Regular expressions may be used with other address forms, so `0+/Peter/` finds the first `Peter` in the file and `$-/Peter/` finds the last. Table II summarizes `sam`'s addresses.

Table II. `Sam` addresses

Simple addresses	
#n	The empty string after character *n*
n	Line *n*.
/regexp/	The first following match of the regular expression
−/regexp/	The first previous match of the regular expression
$	The null string at the end of the file
.	Dot
'	The address mark, set by `k` command
"*regexp*"	Dot in the file whose menu line matches regexp
Compound addresses	
a1+*a2*	The address *a2* evaluated starting at right of *a1*
a1−*a2*	*a2* evaluated in the reverse direction starting at left of *a1*
a1,*a2*	From the left of *a1* to the right of *a2* (default 0,$)
a1;*a2*	Like , but sets dot after evaluating *a1*

The operators + and − are high precedence, while , and ; are low precedence. In both + and − forms, *a2* defaults to 1 and *a1* defaults to dot. If both *a1* and *a2* are present, + may be elided.

The language discussed so far will not seem novel to people who use UNIX text editors such as `ed` or `vi`.[9] Moreover, the kinds of editing operations these commands allow, with the exception of regular expressions and line numbers, are clearly more conveniently handled by a mouse-based interface. Indeed, `sam`'s mouse language (discussed at length below) is the means by which simple changes are usually made. For large or repetitive changes, however, a textual language outperforms a manual interface.

Imagine that, instead of deleting just one occurrence of the string `Peter`, we wanted to eliminate every `Peter`. What's needed is an iterator that runs a command for each occurrence of some text. `Sam`'s iterator is called `x`, for extract:

```
x/expression/  command
```

finds all matches in dot of the specified expression, and for each such match, sets dot to the text matched and runs the command. So to delete all the `Peters`:

```
0,$ x/Peter/ d
```

(Blanks in these examples are to improve readability; `sam` neither requires nor interprets them.) This searches the entire file (`0,$`) for occurrences of the string `Peter`, and runs the `d` command with dot set to each such occurrence. (By contrast, the comparable `ed` command would delete all *lines* containing `Peter`; `sam` deletes only the `Peters`.) The address `0,$` is commonly used, and may be abbreviated to just a comma. As another example,

```
, x/Peter/ p
```

prints a list of `Peters`, one for each appearance in the file, with no intervening text (not even newlines to separate the instances).

Of course, the text extracted by `x` may be selected by a regular expression, which complicates deciding what set of matches is chosen — matches may overlap. This is resolved by generating the matches starting from the beginning of dot using the leftmost-longest rule, and searching for each match starting from the end of the previous one. Regular expressions may also match null strings, but a null match adjacent to a non-null match is never selected; at least one character must intervene. For example,

```
, c/AAA/
x/B*/ c/-/
, p
```

produces as output

```
-A-A-A-
```

because the pattern `B*` matches the null strings separating the `A`'s.

The `x` command has a complement, `y`, with similar syntax, that executes the command with dot set to the text *between* the matches of the expression. For example,

```
, c/AAA/
y/A/ c/-/
, p
```

produces the same result as the example above.

The `x` and `y` commands are looping constructs, and `sam` has a pair of conditional commands to go with them. They have similar syntax:

`g/`*expression*`/ `*command*

(guard) runs the command exactly once if dot contains a match of the expression. This is different from `x`, which runs the command for *each* match: `x` loops; `g` merely tests, without changing the value of dot. Thus,

```
, x/Peter/ d
```

deletes all occurrences of `Peter`, but

```
, g/Peter/ d
```

deletes the whole file (reduces it to a null string) if `Peter` occurs anywhere in the text. The complementary conditional is `v`, which runs the command if there is *no* match of the expression.

These control-structure-like commands may be composed to construct more involved operations. For example, to print those lines of text that contain the string `Peter`:

```
, x/.*\n/ g/Peter/ p
```

The `x` breaks the file into lines, the `g` selects those lines containing `Peter`, and the `p` prints them. This command gives an address for the `x` command (the whole file), but because `g` does not have an explicit address, it applies to the value of dot produced by the `x` command, that is, to each line. All commands in `sam` except for the command to write a file to disc use dot for the default address.

Composition may be continued indefinitely.

```
, x/.*\n/ g/Peter/ v/SaltPeter/ p
```

prints those lines containing `Peter` but *not* those containing `SaltPeter`.

Structural Regular Expressions

Unlike other UNIX text editors, including the non-interactive ones such as sed and awk,[7] sam is good for manipulating files with multi-line 'records.' An example is an on-line phone book composed of records, separated by blank lines, of the form

```
Herbert Tic
44 Turnip Ave., Endive, NJ
201-5555642

Norbert Twinge
16 Potato St., Cabbagetown, NJ
201-5553145

...
```

The format may be encoded as a regular expression:

```
(.+\n)+
```

that is, a sequence of one or more non-blank lines. The command to print Mr. Tic's entire record is then

```
, x/(.+\n)+/ g/^Herbert Tic$/ p
```

and that to extract just the phone number is

```
, x/(.+\n)+/ g/^Herbert Tic$/ x/^[0-9]*-[0-9]*\n/ p
```

The latter command breaks the file into records, chooses Mr. Tic's record, extracts the phone number from the record, and finally prints the number.

A more involved problem is that of renaming a particular variable, say n, to num in a C program. The obvious first attempt,

```
, x/n/ c/num/
```

is badly flawed: it changes not only the variable n but any letter n that appears. We need to extract all the variables, and select those that match n and only n:

```
, x/[A-Za-z_][A-Za-z_0-9]*/ g/n/ v/../ c/num/
```

The pattern [A-Za-z_][A-Za-z_0-9]* matches C identifiers. Next g/n/ selects those containing an n. Then v/../ rejects those containing two (or more) characters, and finally c/num/ changes the remainder (identifiers n) to num. This version clearly works much better, but there may still be problems. For example, in C character and string constants, the sequence \n is interpreted as a newline character, and we don't want to change it to \num. This problem can be forestalled with a y command:

```
, y/\\n/ x/[A-Za-z_][A-Za-z_0-9]*/ g/n/ v/../ c/num/
```

(the second \ is necessary because of lexical conventions in regular expressions), or we could even reject character constants and strings outright:

```
,y/'[^']*'/ y/"[^"]*"/ x/[A-Za-z_][A-Za-z_0-9]*/ g/n/ v/../ c/num/
```

The y commands in this version exclude from consideration all character constants and strings. The only remaining problem is to deal with the possible occurrence of \' or \" within these sequences, but it's easy to see how to resolve this difficulty.

The point of these composed commands is successive refinement. A simple version of the command is tried, and if it's not good enough, it can be honed by adding a clause or two. (Mistakes can be undone; see below. Also, the mouse language makes it unnecessary to retype the command each time.) The resulting chains of commands are somewhat reminiscent of shell pipelines.[7] Unlike pipelines, though, which pass along modified *data*, sam commands pass a *view* of the data. The text at each step of the

command is the same, but which pieces are selected is refined step by step until the correct piece is available to the final step of the command line, which ultimately makes the change.

In other UNIX programs, regular expressions are used only for selection, as in the sam g command, never for extraction as in the x or y command. For example, patterns in awk[7] are used to select lines to be operated on, but cannot be used to describe the format of the input text, or to handle newline-free text. The use of regular expressions to describe the structure of a piece of text rather than its contents, as in the x command, has been given a name: *structural regular expressions.* When they are composed, as in the above example, they are pleasantly expressive. Their use is discussed at greater length elsewhere.[10]

Multiple files

Sam has a few other commands, mostly relating to input and output.

```
e discfilename
```

replaces the contents and name of the current file with those of the named disc file;

```
w discfilename
```

writes the contents to the named disc file; and

```
r discfilename
```

replaces dot with the contents of the named disc file. All these commands use the current file's name if none is specified. Finally,

```
f discfilename
```

changes the name associated with the file and displays the result:

```
'-. discfilename
```

This output is called the file's *menu line,* because it is the contents of the file's line in the button 3 menu (described in the next section). The first three characters are a concise notation for the state of the file. The apostrophe signifies that the file is modified. The minus sign indicates the number of windows open on the file (see the next section): − means none, + means one, and * means more than one. Finally, the period indicates that this is the current file. These characters are useful for controlling the X command, described shortly.

Sam may be started with a set of disc files (such as all the source for a program) by invoking it with a list of file names as arguments, and more may be added or deleted on demand.

```
B discfile1 discfile2 ...
```

adds the named files to sam's list, and

```
D discfile1 discfile2 ...
```

removes them from sam's memory (without effect on associated disc files). Both these commands have a syntax for using the shell[7] (the UNIX command interpreter) to generate the lists:

```
B <echo *.c
```

will add all C source files, and

```
B <grep -l variable *.c
```

will add all C source files referencing a particular variable (the UNIX command grep −l lists all files in its arguments that contain matches of the specified regular

expression). Finally, D without arguments deletes the current file.

There are two ways to change which file is current:

```
b filename
```

makes the named file current. The B command does the same, but also adds any new files to sam's list. (In practice, of course, the current file is usually chosen by mouse actions, not by textual commands.) The other way is to use a form of address that refers to files:

"*expression*" *address*

refers to the address evaluated in the file whose menu line matches the expression (there must be exactly one match). For example,

```
"peter.c" 3
```

refers to the third line of the file whose name matches peter.c. This is most useful in the move (m) and copy (t) commands:

```
0,$ t "peter.c" 0
```

makes a copy of the current file at the beginning of peter.c.

The X command is a looping construct, like x, that refers to files instead of strings:

X/*expression/ command*

runs the command in all files whose menu lines match the expression. The best example is

```
X/'/ w
```

which writes to disc all modified files. Y is the complement of X: it runs the command on all files whose menu lines don't match the expression:

```
Y/\.c/ D
```

deletes all files that don't have .c in their names, that is, it keeps all C source files and deletes the rest.

Braces allow commands to be grouped, so

```
{
        command1
        command2
}
```

is syntactically a single command that runs two commands. Thus,

```
X/\.c/ ,g/variable/ {
        f
        , x/.*\n/ g/variable/ p
}
```

finds all occurrences of variable in C source files, and prints out the file names and lines of each match. The precise semantics of compound operations is discussed in the implementation sections below.

Finally, the undo command, u, undoes the last command, no matter how many files were affected. Multiple undo operations move further back in time, so

```
u
u
```

(which may be abbreviated u2) undoes the last two commands. An undo may not be undone, however, nor may any command that adds or deletes files. Everything else is undoable, though, including for example e commands:

```
e filename
u
```

restores the state of the file completely, including its name, dot, and modified bit. Because of the undo, potentially dangerous commands are not guarded by confirmations. Only D, which destroys the information necessary to restore itself, is protected. It will not delete a modified file, but a second D of the same file will succeed regardless. The q command, which exits sam, is similarly guarded.

Mouse Interface

Sam is most commonly run connected to a bitmap display and mouse for interactive editing. The only difference in the command language between regular, mouse-driven sam and sam −d is that if an address is provided without a command, sam −d will print the text referenced by the address, but regular sam will highlight it on the screen — in fact, dot is always highlighted (see Figure 2).

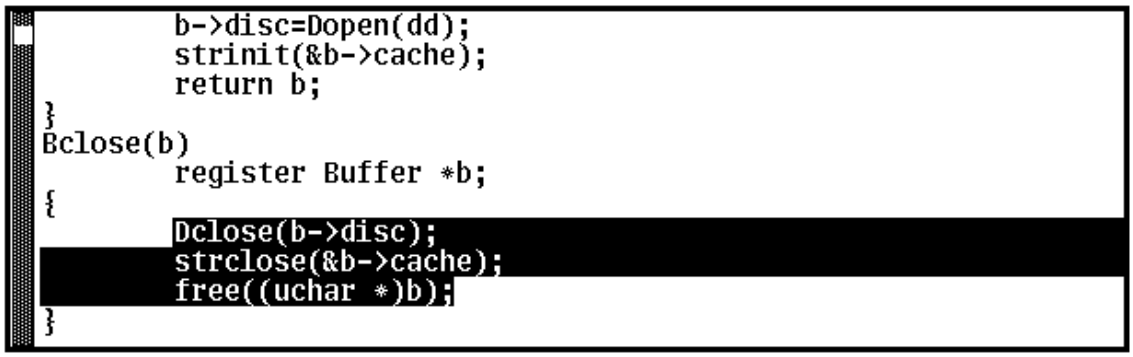

Figure 2. A sam *window. The scroll bar down the left represents the file, with the bubble showing the fraction visible in the window. The scroll bar may be manipulated by the mouse for convenient browsing. The current text, which is highlighted, need not fit on a line. Here it consists of one partial line, one complete line, and final partial line.*

Each file may have zero or more windows open on the display. At any time, only one window in all of sam is the *current window,* that is, the window to which typing and mouse actions refer; this may be the sam window (that in which commands may be typed) or one of the file windows. When a file has multiple windows, the image of the file in each window is always kept up to date. The current file is the last file affected by a command, so if the sam window is current, the current window is not a window on the current file. However, each window on a file has its own value of dot, and when switching between windows on a single file, the file's value of dot is changed to that of the window. Thus, flipping between windows behaves in the obvious, convenient way.

The mouse on the Blit has three buttons, numbered left to right. Button 3 has a list of commands to manipulate windows, followed by a list of 'menu lines' exactly as printed by the f command, one per file (not one per window). These menu lines are sorted by file name. If the list is long, the Blit menu software will make it more manageable by generating a scrolling menu instead of an unwieldy long list. Using the menu to select a file from the list makes that file the current file, and the most recently current window in that file the current window. But if that file is already current, selecting it in the menu cycles through the windows on the file; this simple trick avoids a special menu to choose windows on a file. If there is no window open on the file, sam changes the mouse cursor to prompt the user to create one.

The commands on the button 3 menu are straightforward (see Figure 3), and are like the commands to manipulate windows in `mux`,[8] the Blit's window system. `New` makes a new file, and gives it one empty window, whose size is determined by a rectangle swept by the mouse. `Zerox` prompts for a window to be selected, and makes a clone of that window; this is how multiple windows are created on one file. `Reshape` changes the size of the indicated window, and `close` deletes it. If that is the last window open on the file, `close` first does a `D` command on the file. `Write` is identical to a `w` command on the file; it is in the menu purely for convenience. Finally, `~~sam~~` is a menu item that appears between the commands and the file names. Selecting it makes the `sam` window the current window, causing subsequent typing to be interpreted as commands.

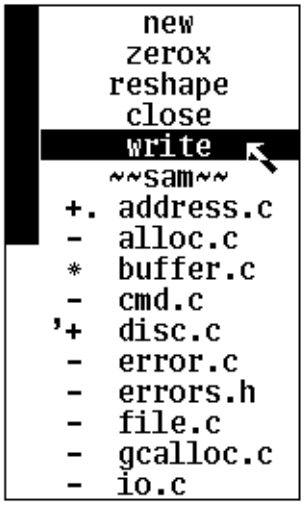

Figure 3. The menu on button 3. The black rectangle on the left is a scroll bar; the menu is limited to the length shown to prevent its becoming unwieldy. Above the `~~sam~~` *line is a list of commands; beneath it is a list of files, presented exactly as with the* `f` *command.*

When `sam` requests that a window be swept, in response to `new`, `zerox` or `reshape`, it changes the mouse cursor from the usual arrow to a box with a small arrow. In this state, the mouse may be used to indicate an arbitrary rectangle by pressing button 3 at one corner and releasing it at the opposite corner. More conveniently, button 3 may simply be clicked, whereupon `sam` creates the maximal rectangle that contains the cursor and abuts the `sam` window. By placing the `sam` window in the middle of the screen, the user can define two regions (one above, one below) in which stacked fully-overlapping windows can be created with minimal fuss (see Figure 1). This simple user interface trick makes window creation noticeably easier.

The cut-and-paste editor is essentially the same as that in Smalltalk-80.[11] The text in dot is always highlighted on the screen. When a character is typed it replaces dot, and sets dot to the null string after the character. Thus, ordinary typing inserts text. Button 1 is used for selection: pressing the button, moving the mouse, and lifting the button selects (sets dot to) the text between the points where the button was pressed and released. Pressing and releasing at the same point selects a null string; this is called clicking. Clicking twice quickly, or *double clicking,* selects larger objects; for example, double clicking in a word selects the word, double clicking just inside an opening bracket selects the text contained in the brackets (handling nested brackets correctly), and similarly for parentheses, quotes, and so on. The double-clicking rules reflect a bias toward programmers. If `sam` were intended more for word processing, double-clicks would probably select linguistic structures such as sentences.

If button 1 is pressed outside the current window, it makes the indicated window current. This is the easiest way to switch between windows and files.

Pressing button 2 brings up a menu of editing functions (see Figure 4). These mostly apply to the selected text: `cut` deletes the selected text, and remembers it in a hidden buffer called the *snarf buffer,* `paste` replaces the selected text by the contents of the snarf buffer, `snarf` just copies the selected text to the snarf buffer, `look` searches forward for the next literal occurrence of the selected text, and `<mux>` exchanges snarf buffers with the window system in which `sam` is running. Finally, the last regular expression used appears as a menu entry to search forward for the next occurrence of a match for the expression.

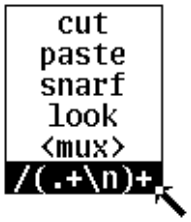

Figure 4. The menu on button 2. The bottom entry tracks the most recently used regular expression, which may be literal text.

The relationship between the command language and the mouse language is entirely due to the equality of dot and the selected text chosen with button 1 on the mouse. For example, to make a set of changes in a C subroutine, dot can be set by double clicking on the left brace that begins the subroutine, which sets dot for the command language. An address-free command then typed in the `sam` window will apply only to the text between the opening and closing braces of the function. The idea is to select what you want, and then say what you want to do with it, whether invoked by a menu selection or by a typed command. And of course, the value of dot is highlighted on the display after the command completes. This relationship between mouse interface and command language is clumsy to explain, but comfortable, even natural, in practice.

The Implementation

The next few sections describe how `sam` is put together, first the host part, then the inter-component communication, then the terminal part. After explaining how the command language is implemented, the discussion follows (roughly) the path of a character from the temporary file on disc to the screen. The presentation centers on the data structures, because that is how the program was designed and because the algorithms are easy to provide, given the right data structures.

Parsing and execution

The command language is interpreted by parsing each command with a table-driven recursive descent parser, and when a complete command is assembled, invoking a top-down executor. Most editors instead employ a simple character-at-a-time lexical scanner. Use of a parser makes it easy and unambiguous to detect when a command is complete, which has two advantages. First, escape conventions such as backslashes to quote multiple-line commands are unnecessary; if the command isn't finished, the parser keeps reading. For example, a multiple-line append driven by an `x` command is straightforward:

```
x/.*\n/ g/Peter/ a
one line about Peter
another line about Peter
.
```

Other UNIX editors would require a backslash after all but the last line.

The other advantage is specific to the two-process structure of sam. The host process must decide when a command is completed so the command interpreter can be called. This problem is easily resolved by having the lexical analyzer read the single stream of events from the terminal, directly executing all typing and mouse commands, but passing to the parser characters typed to the sam command window. This scheme is slightly complicated by the availability of cut-and-paste editing in the sam window, but that difficulty is resolved by applying the rules used in mux: when a newline is typed to the sam window, all text between the newline and the previously typed newline is made available to the parser. This permits arbitrary editing to be done to a command before typing newline and thereby requesting execution.

The parser is driven by a table because the syntax of addresses and commands is regular enough to be encoded compactly. There are few special cases, such as the replacement text in a substitution, so the syntax of almost all commands can be encoded with a few flags. These include whether the command allows an address (for example, e does not), whether it takes a regular expression (as in x and s), whether it takes replacement text (as in c or i), which may be multi-line, and so on. The internal syntax of regular expressions is handled by a separate parser; a regular expression is a leaf of the command parse tree. Regular expressions are discussed fully in the next section.

The parser table also has information about defaults, so the interpreter is always called with a complete tree. For example, the parser fills in the implicit 0 and $ in the abbreviated address , (comma), inserts a + to the left of an unadorned regular expression in an address, and provides the usual default address . (dot) for commands that expect an address but are not given one.

Once a complete command is parsed, the evaluation is easy. The address is evaluated left-to-right starting from the value of dot, with a mostly ordinary expression evaluator. Addresses, like many of the data structures in sam, are held in a C structure and passed around by value:

```
typedef long Posn;    /* Position in a file */
typedef struct Range{
        Posn    p1, p2;
}Range;
typedef struct Address{
        Range   r;
        File    *f;
}Address;
```

An address is encoded as a substring (character positions p1 to p2) in a file f. (The data type File is described in detail below.)

The address interpreter is an Address-valued function that traverses the parse tree describing an address (the parse tree for the address has type Addrtree):

```
Address
address(ap, a, sign)
        Addrtree *ap;
        Address a;
        int sign;
{
        Address a2;
        do
                switch(ap->type){
                case '.':
                        a=a.f->dot;
                        break;
                case '$':
                        a.r.p1=a.r.p2=a.f->nbytes;
                        break;
                case '"':
                        a=matchfile(a, ap->aregexp)->dot;
                        break;
                case ',':
                        a2=address(ap->right, a, 0);
                        a=address(ap->left, a, 0);
                        if(a.f!=a2.f || a2.r.p2<a.r.p1)
                                error(Eorder);
                        a.r.p2=a2.r.p2;
                        return a;
                /* and so on */
                }
        while((ap=ap->right)!=0);
        return a;
}
```

Throughout, errors are handled by a non-local `goto` (a `setjmp/longjmp` in C terminology) hidden in a routine called `error` that immediately aborts the execution, retracts any partially made changes (see the section below on 'undoing'), and returns to the top level of the parser. The argument to `error` is an enumeration type that is translated to a terse but possibly helpful message such as '?addresses out of order.' Very common messages are kept short; for example the message for a failed regular expression search is '?search.'

Character addresses such as `#3` are trivial to implement, as the `File` data structure is accessible by character number. However, `sam` keeps no information about the position of newlines — it is too expensive to track dynamically — so line addresses are computed by reading the file, counting newlines. Except in very large files, this has proven acceptable: file access is fast enough to make the technique practical, and lines are not central to the structure of the command language.

The command interpreter, called `cmdexec`, is also straightforward. The parse table includes a function to call to interpret a particular command. That function receives as arguments the calculated address for the command and the command tree (of type `Cmdtree`), which may contain information such as the subtree for compound commands. Here, for example, is the function for the `g` and `v` commands:

```
int
g_cmd(a, cp)
        Address a;
        Cmdtree *cp;
{
        compile(cp->regexp);
        if(execute(a.f, a.r.p1, a.r.p2)!=(cp->cmdchar=='v')){
                a.f->dot=a;
                return cmdexec(a, cp->subcmd);
        }
        return TRUE;    /* cause execution to continue */
}
```

(Compile and execute are part of the regular expression code, described in the next section.) Because the parser and the File data structure do most of the work, most commands are similarly brief.

Regular expressions

The regular expression code in sam is an interpreted, rather than compiled on-the-fly, implementation of Thompson's non-deterministic finite automaton algorithm.[12] The syntax and semantics of the expressions are as in the UNIX program egrep, including alternation, closures, character classes, and so on. The only changes in the notation are two additions: \n is translated to, and matches, a newline character, and @ matches any character. In egrep, the character . matches any character except newline, and in sam the same rule seemed safest, to prevent idioms like .* from spanning newlines. Egrep expressions are arguably too complicated for an interactive editor — certainly it would make sense if all the special characters were two-character sequences, so that most of the punctuation characters wouldn't have peculiar meanings — but for an interesting command language, full regular expressions are necessary, and egrep defines the full regular expression syntax for UNIX programs. Also, it seemed superfluous to define a new syntax, since various UNIX programs (ed, egrep and vi) define too many already.

The expressions are compiled by a routine, compile, that generates the description of the non-deterministic finite state machine. A second routine, execute, interprets the machine to generate the leftmost-longest match of the expression in a substring of the file. The algorithm is described elsewhere.[12,13] Execute reports whether a match was found, and sets a global variable, of type Range, to the substring matched.

A trick is required to evaluate the expression in reverse, such as when searching backwards for an expression. For example,

```
-/P.*r/
```

looks backwards through the file for a match of the expression. The expression, however, is defined for a forward search. The solution is to construct a machine identical to the machine for a forward search except for a reversal of all the concatenation operators (the other operators are symmetric under direction reversal), to exchange the meaning of the operators ^ and $, and then to read the file backwards, looking for the usual earliest longest match.

Execute generates only one match each time it is called. To interpret looping constructs such as the x command, sam must therefore synchronize between calls of execute to avoid problems with null matches. For example, even given the leftmost-longest rule, the expression a* matches three times in the string ab (the character a, the null string between the a and b, and the final null string). After returning a match for the a, sam must not match the null string before the b. The algorithm starts execute at the end of its previous match, and if the match it returns is null and abuts

the previous match, rejects the match and advances the initial position one character.

Memory allocation

The C language has no memory allocation primitives, although a standard library routine, `malloc`, provides adequate service for simple programs. For specific uses, however, it can be better to write a custom allocator. The allocator (or rather, pair of allocators) described here work in both the terminal and host parts of `sam`. They are designed for efficient manipulation of strings, which are allocated and freed frequently and vary in length from essentially zero to 32 Kbytes (very large strings are written to disc). More important, strings may be large and change size often, so to minimize memory usage it is helpful to reclaim and to coalesce the unused portions of strings when they are truncated.

Objects to be allocated in `sam` are of two flavors: the first is C `structs`, which are small and often addressed by pointer variables; the second is variable-sized arrays of characters or integers whose base pointer is always used to access them. The memory allocator in `sam` is therefore in two parts: first, a traditional first-fit allocator that provides fixed storage for `structs`; and second, a garbage-compacting allocator that reduces storage overhead for variable-sized objects, at the cost of some bookkeeping. The two types of objects are allocated from adjoining arenas, with the garbage-compacting allocator controlling the arena with higher addresses. Separating into two arenas simplifies compaction and prevents fragmentation due to immovable objects. The access rules for garbage-compactable objects (discussed in the next paragraph) allow them to be relocated, so when the first-fit arena needs space, it moves the garbage-compacted arena to higher addresses to make room. Storage is therefore created only at successively higher addresses, either when more garbage-compacted space is needed or when the first-fit arena pushes up the other arena.

Objects that may be compacted declare to the allocator a cell that is guaranteed to be the sole repository of the address of the object whenever a compaction can occur. The compactor can then update the address when the object is moved. For example, the implementation of type `List` (really a variable-length array) is:

```
typedef struct List{
        int     nused;
        long    *ptr;
}List;
```

The `ptr` cell must always be used directly, and never copied. When a `List` is to be created the `List` structure is allocated in the ordinary first-fit arena and its `ptr` is allocated in the garbage-compacted arena. A similar data type for strings, called `String`, stores variable-length character arrays of up to 32767 elements.

A related matter of programming style: `sam` frequently passes structures by value, which simplifies the code. Traditionally, C programs have passed structures by reference, but implicit allocation on the stack is easier to use. Structure passing is a relatively new feature of C (it is not in the standard reference manual for C[14]), and is poorly supported in most commercial C compilers. It's convenient and expressive, though, and simplifies memory management by avoiding the allocator altogether and eliminating pointer aliases.

Data structures for manipulating files

Experience with `jim` showed that the requirements of the file data structure were few, but strict. First, files need to be read and written quickly; adding a fresh file must be painless. Second, the implementation must place no arbitrary upper limit on the number or sizes of files. (It should be practical to edit many files, and files up to megabytes in length should be handled gracefully.) This implies that files be stored on disc, not in

main memory. (Aficionados of virtual memory may argue otherwise, but the implementation of virtual memory in our system is not something to depend on for good performance.) Third, changes to files need be made by only two primitives: deletion and insertion. These are inverses of each other, which simplifies the implementation of the undo operation. Finally, it must be easy and efficient to access the file, either forwards or backwards, a byte at a time.

The `File` data type is constructed from three simpler data structures that hold arrays of characters. Each of these types has an insertion and deletion operator, and the insertion and deletion operators of the `File` type itself are constructed from them.

The simplest type is the `String`, which is used to hold strings in main memory. The code that manages `Strings` guarantees that they will never be longer than some moderate size, and in practice they are rarely larger than 8 Kbytes. `Strings` have two purposes: they hold short strings like file names with little overhead, and because they are deliberately small, they are efficient to modify. They are therefore used as the data structure for in-memory caches.

The disc copy of the file is managed by a data structure called a `Disc`, which corresponds to a temporary file. A `Disc` has no storage in main memory other than bookkeeping information; the actual data being held is all on the disc. To reduce the number of open files needed, `sam` opens a dozen temporary UNIX files and multiplexes the `Discs` upon them. This permits many files to be edited; the entire `sam` source (48 files) may be edited comfortably with a single instance of `sam`. Allocating one temporary file per `Disc` would strain the operating system's limit on the number of open files. Also, spreading the traffic among temporary files keeps the files shorter, and shorter files are more efficiently implemented by the UNIX I/O subsystem.

A `Disc` is an array of fixed-length blocks, each of which contains between 1 and 4096 characters of active data. (The block size of our UNIX file system is 4096 bytes.) The block addresses within the temporary file and the length of each block are stored in a `List`. When changes are made the live part of blocks may change size. Blocks are created and coalesced when necessary to try to keep the sizes between 2048 and 4096 bytes. An actively changing part of the `Disc` therefore typically has about a kilobyte of slop that can be inserted or deleted without changing more than one block or affecting the block order. When an insertion would overflow a block, the block is split, a new one is allocated to receive the overflow, and the memory-resident list of blocks is rearranged to reflect the insertion of the new block.

Obviously, going to the disc for every modification to the file is prohibitively expensive. The data type `Buffer` consists of a `Disc` to hold the data and a `String` that acts as a cache. This is the first of a series of caches throughout the data structures in `sam`. The caches not only improve performance, they provide a way to organize the flow of data, particularly in the communication between the host and terminal. This idea is developed below, in the section on communications.

To reduce disc traffic, changes to a `Buffer` are mediated by a variable-length string, in memory, that acts as a cache. When an insertion or deletion is made to a `Buffer`, if the change can be accommodated by the cache, it is done there. If the cache becomes bigger than a block because of an insertion, some of it is written to the `Disc` and deleted from the cache. If the change does not intersect the cache, the cache is flushed. The cache is only loaded at the new position if the change is smaller than a block; otherwise, it is sent directly to the `Disc`. This is because large changes are typically sequential, whereupon the next change is unlikely to overlap the current one.

A `File` comprises a `String` to hold the file name and some ancillary data such as dot and the modified bit. The most important components, though, are a pair of `Buffers`, one called the transcript and the other the contents. Their use is described in the next section.

The overall structure is shown in Figure 5. Although it may seem that the data is touched many times on its way from the `Disc`, it is read (by one UNIX system call) directly into the cache of the associated `Buffer`; no extra copy is done. Similarly, when flushing the cache, the text is written directly from the cache to disc. Most operations act directly on the text in the cache. A principle applied throughout `sam` is that the fewer times the data is copied, the faster the program will run (see also the paper by Waite[15]).

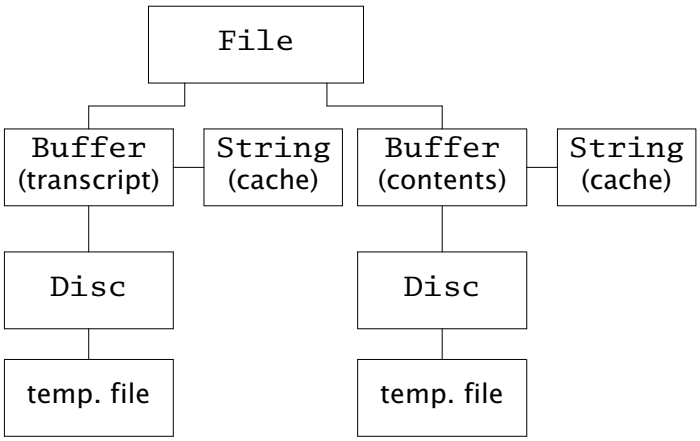

Figure 5. File data structures. The temporary files are stored in the standard repository for such files on the host system.

The contents of a `File` are accessed by a routine that copies to a buffer a substring of a file starting at a specified offset. To read a byte at a time, a per-`File` array is loaded starting from a specified initial position, and bytes may then be read from the array. The implementation is done by a macro similar to the C standard I/O `getc` macro.[14] Because the reading may be done at any address, a minor change to the macro allows the file to be read backwards. This array is read-only; there is no `putc`.

Doing and undoing

`Sam` has an unusual method for managing changes to files. The command language makes it easy to specify multiple variable-length changes to a file millions of bytes long, and such changes must be made efficiently if the editor is to be practical. The usual techniques for inserting and deleting strings are inadequate under these conditions. The `Buffer` and `Disc` data structures are designed for efficient random access to long strings, but care must be taken to avoid super-linear behavior when making many changes simultaneously.

`Sam` uses a two-pass algorithm for making changes, and treats each file as a database against which transactions are registered. Changes are not made directly to the contents. Instead, when a command is started, a 'mark' containing a sequence number is placed in the transcript `Buffer`, and each change made to the file, either an insertion or deletion or a change to the file name, is appended to the end of the transcript. When the command is complete, the transcript is rewound to the mark and applied to the contents.

One reason for separating evaluation from application in this way is to simplify tracking the addresses of changes made in the middle of a long sequence. The two-pass algorithm also allows all changes to apply to the *original* data: no change can affect another change made in the same command. This is particularly important when evaluating an `x` command because it prevents regular expression matches from stumbling over changes made earlier in the execution. Also, the two-pass algorithm is

cleaner than the way other UNIX editors allow changes to affect each other; for example, ed's idioms to do things like delete every other line depend critically on the implementation. Instead, sam's simple model, in which all changes in a command occur effectively simultaneously, is easy to explain and to understand.

The records in the transcript are of the form ''delete substring from locations 123 to 456'' and ''insert 11 characters 'hello there' at location 789.'' (It is an error if the changes are not at monotonically greater positions through the file.) While the update is occurring, these numbers must be offset by earlier changes, but that is straightforward and local to the update routine; moreover, all the numbers have been computed before the first is examined.

Treating the file as a transaction system has another advantage: undo is trivial. All it takes is to invert the transcript after it has been implemented, converting insertions into deletions and vice versa, and saving them in a holding Buffer. The 'do' transcript can then be deleted from the transcript Buffer and replaced by the 'undo' transcript. If an undo is requested, the transcript is rewound and the undo transcript executed. Because the transcript Buffer is not truncated after each command, it accumulates successive changes. A sequence of undo commands can therefore back up the file arbitrarily, which is more helpful than the more commonly implemented self-inverse form of undo. (Sam provides no way to undo an undo, but if it were desired, it would be easy to provide by re-interpreting the 'do' transcript.) Each mark in the transcript contains a sequence number and the offset into the transcript of the previous mark, to aid in unwinding the transcript. Marks also contain the value of dot and the modified bit so these can be restored easily. Undoing multiple files is easy; it merely demands undoing all files whose latest change has the same sequence number as the current file.

Another benefit of having a transcript is that errors encountered in the middle of a complicated command need not leave the files in an intermediate state. By rewinding the transcript to the mark beginning the command, the partial command can be trivially undone.

When the update algorithm was first implemented, it was unacceptably slow, so a cache was added to coalesce nearby changes, replacing multiple small changes by a single larger one. This reduced the number of insertions into the transaction Buffer, and made a dramatic improvement in performance, but made it impossible to handle changes in non-monotonic order in the file; the caching method only works if changes don't overlap. Before the cache was added, the transaction could in principle be sorted if the changes were out of order, although this was never done. The current status is therefore acceptable performance with a minor restriction on global changes, which is sometimes, but rarely, an annoyance.

The update algorithm obviously paws the data more than simpler algorithms, but it is not prohibitively expensive; the caches help. (The principle of avoiding copying the data is still honored here, although not as piously: the data is moved from contents' cache to the transcript's all at once and through only one internal buffer.) Performance figures confirm the efficiency. To read from a dead start a hundred kilobyte file on a VAX-11/750 takes 1.4 seconds of user time, 2.5 seconds of system time, and 5 seconds of real time. Reading the same file in ed takes 6.0 seconds of user time, 1.7 seconds of system time, and 8 seconds of real time. Sam uses about half the CPU time. A more interesting example is the one stated above: inserting a character between every pair of characters in the file. The sam command is

```
,y/@/ a/x/
```

and takes 3 CPU seconds per kilobyte of input file, of which about a third is spent in the regular expression code. This translates to about 500 changes per second. Ed takes 1.5 seconds per kilobyte to make a similar change (ignoring newlines), but cannot undo it. The same example in ex,[9] a variant of ed done at the University of California at

Berkeley, which allows one level of undoing, again takes 3 seconds. In summary, sam's performance is comparable to that of other UNIX editors, although it solves a harder problem.

Communications

The discussion so far has described the implementation of the host part of sam; the next few sections explain how a machine with mouse and bitmap display can be engaged to improve interaction. Sam is not the first editor to be written as two processes,[16] but its implementation has some unusual aspects.

There are several ways sam's host and terminal parts may be connected. The first and simplest is to forgo the terminal part and use the host part's command language to edit text on an ordinary terminal. This mode is invoked by starting sam with the −d option. With no options, sam runs separate host and terminal programs, communicating with a message protocol over the physical connection that joins them. Typically, the connection is an RS-232 link between a Blit (the prototypical display for sam) and a host running the Ninth Edition of the UNIX operating system.[8] (This is the version of the system used in the Computing Sciences Research Center at AT&T Bell Laboratories [now Lucent Technologies, Bell Labs], where I work. Its relevant aspects are discussed in the Blit paper.[1]) The implementation of sam for the SUN computer runs both processes on the same machine and connects them by a pipe.

The low bandwidth of an RS-232 link necessitated the split between the two programs. The division is a mixed blessing: a program in two parts is much harder to write and to debug than a self-contained one, but the split makes several unusual configurations possible. The terminal may be physically separated from the host, allowing the conveniences of a mouse and bitmap display to be taken home while leaving the files at work. It is also possible to run the host part on a remote machine:

```
sam −r host
```

connects to the terminal in the usual way, and then makes a call across the network to establish the host part of sam on the named machine. Finally, it cross-connects the I/O to join the two parts. This allows sam to be run on machines that do not support bitmap displays; for example, sam is the editor of choice on our Cray X-MP/24. Sam −r involves *three* machines: the remote host, the terminal, and the local host. The local host's job is simple but vital: it passes the data between the remote host and terminal.

The host and terminal exchange messages asynchronously (rather than, say, as remote procedure calls) but there is no error detection or correction because, whatever the configuration, the connection is reliable. Because the terminal handles mundane interaction tasks such as popping up menus and interpreting the responses, the messages are about data, not actions. For example, the host knows nothing about what is displayed on the screen, and when the user types a character, the message sent to the host says ''insert a one-byte string at location 123 in file 7,'' not ''a character was typed at the current position in the current file.'' In other words, the messages look very much like the transaction records in the transcripts.

Either the host or terminal part of sam may initiate a change to a file. The command language operates on the host, while typing and some mouse operations are executed directly in the terminal to optimize response. Changes initiated by the host program must be transmitted to the terminal, and vice versa. (A token is exchanged to determine which end is in control, which means that characters typed while a time-consuming command runs must be buffered and do not appear until the command is complete.) To maintain consistent information, the host and terminal track changes through a per-file data structure that records what portions of the file the terminal has received. The data structure, called a Rasp (a weak pun: it's a file with holes) is held and updated by both the host and terminal. A Rasp is a list of Strings holding those

parts of the file known to the terminal, separated by counts of the number of bytes in the interstices. Of course, the host doesn't keep a separate copy of the data (it only needs the lengths of the various pieces), but the structure is the same on both ends.

The `Rasp` in the terminal doubles as a cache. Since the terminal keeps the text for portions of the file it has displayed, it need not request data from the host when revisiting old parts of the file or redrawing obscured windows, which speeds things up considerably over low-speed links.

It's trivial for the terminal to maintain its `Rasp`, because all changes made on the terminal apply to parts of the file already loaded there. Changes made by the host are compared against the `Rasp` during the update sequence after each command. Small changes to pieces of the file loaded in the terminal are sent in their entirety. Larger changes, and changes that fall entirely in the holes, are transmitted as messages without literal data: only the lengths of the deleted and inserted strings are transmitted. When a command is completed, the terminal examines its visible windows to see if any holes in their `Rasps` intersect the visible portion of the file. It then requests the missing data from the host, along with up to 512 bytes of surrounding data, to minimize the number of messages when visiting a new portion of the file. This technique provides a kind of two-level lazy evaluation for the terminal. The first level sends a minimum of information about parts of the file not being edited interactively; the second level waits until a change is displayed before transmitting the new data. Of course, performance is also helped by having the terminal respond immediately to typing and simple mouse requests. Except for small changes to active pieces of the file, which are transmitted to the terminal without negotiation, the terminal is wholly responsible for deciding what is displayed; the host uses the `Rasp` only to tell the terminal what might be relevant.

When a change is initiated by the host, the messages to the terminal describing the change are generated by the routine that applies the transcript of the changes to the contents of the `File`. Since changes are undone by the same update routine, undoing requires no extra code in the communications; the usual messages describing changes to the file are sufficient to back up the screen image.

The `Rasp` is a particularly good example of the way caches are used in `sam`. First, it facilitates access to the active portion of the text by placing the busy text in main memory. In so doing, it provides efficient access to a large data structure that does not fit in memory. Since the form of data is to be imposed by the user, not by the program, and because characters will frequently be scanned sequentially, files are stored as flat objects. Caches help keep performance good and linear when working with such data.

Second, the `Rasp` and several of the other caches have some *read-ahead;* that is, the cache is loaded with more information than is needed for the job immediately at hand. When manipulating linear structures, the accesses are usually sequential, and read-ahead can significantly reduce the average time to access the next element of the object. Sequential access is a common mode for people as well as programs; consider scrolling through a document while looking for something.

Finally, like any good data structure, the cache guides the algorithm, or at least the implementation. The `Rasp` was actually invented to control the communications between the host and terminal parts, but I realized very early that it was also a form of cache. Other caches were more explicitly intended to serve a double purpose: for example, the caches in `Files` that coalesce updates not only reduce traffic to the transcript and contents `Buffers`, they also clump screen updates so that complicated changes to the screen are achieved in just a few messages to the terminal. This saved me considerable work: I did not need to write special code to optimize the message traffic to the terminal. Caches pay off in surprising ways. Also, they tend to be independent, so their performance improvements are multiplicative.

Data structures in the terminal

The terminal's job is to display and to maintain a consistent image of pieces of the files being edited. Because the text is always in memory, the data structures are considerably simpler than those in the host part.

Sam typically has far more windows than does mux, the window system within which its Blit implementation runs. Mux has a fairly small number of asynchronously updated windows; sam needs a large number of synchronously updated windows that are usually static and often fully obscured. The different tradeoffs guided sam away from the memory-intensive implementation of windows, called Layers,[17] used in mux. Rather than depending on a complete bitmap image of the display for each window, sam regenerates the image from its in-memory text (stored in the Rasp) when necessary, although it will use such an image if it is available. Like Layers, though, sam uses the screen bitmap as active storage in which to update the image using bitblt.[18,19] The resulting organization, pictured in Figure 6, has a global array of windows, called Flayers, each of which holds an image of a piece of text held in a data structure called a Frame, which in turn represents a rectangular window full of text displayed in some Bitmap. Each Flayer appears in a global list that orders them all front-to-back on the display, and simultaneously as an element of a per-file array that holds all the open windows for that file. The complement in the terminal of the File on the host is called a Text; each connects its Flayers to the associated Rasp.

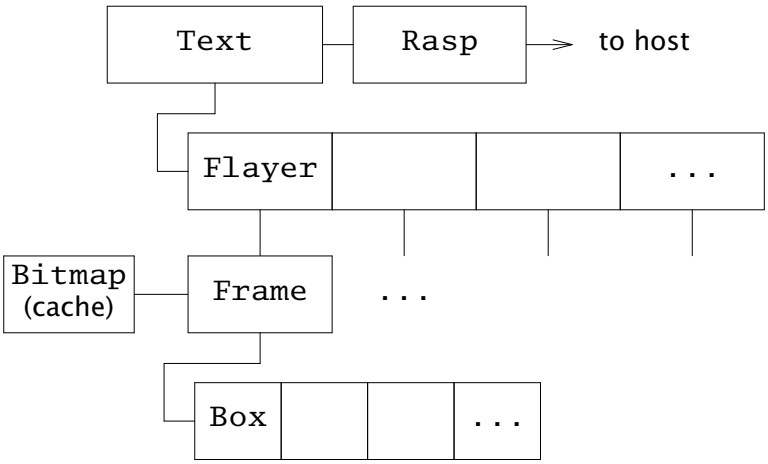

Figure 6. Data structures in the terminal. Flayers *are also linked together into a front-to-back list.* Boxes *are discussed in the next section.*

The Bitmap for a Frame contains the image of the text. For a fully visible window, the Bitmap will be the screen (or at least the Layer in which sam is being run), while for partially obscured windows the Bitmap will be off-screen. If the window is fully obscured, the Bitmap will be null.

The Bitmap is a kind of cache. When making changes to the display, most of the original image will look the same in the final image, and the update algorithms exploit this. The Frame software updates the image in the Bitmap incrementally; the Bitmap is not just an image, it is a data structure.[18,19] The job of the software that updates the display is therefore to use as much as possible of the existing image (converting the text from ASCII characters to pixels is expensive) in a sort of two-dimensional string insertion algorithm. The details of this process are described in the next section.

The Frame software has no code to support overlapping windows; its job is to keep a single Bitmap up to date. It falls to the Flayer software to multiplex the various Bitmaps onto the screen. The problem of maintaining overlapping Flayers is easier than for Layers[17] because changes are made synchronously and because the contents of the window can be reconstructed from the data stored in the Frame; the Layers software makes no such assumptions. In sam, the window being changed is almost always fully visible, because the current window is always fully visible, by construction. However, when multi-file changes are being made, or when more than one window is open on a file, it may be necessary to update partially obscured windows.

There are three cases: the window is fully visible, invisible (fully obscured), or partially visible. If fully visible, the Bitmap is part of the screen, so when the Flayer update routine calls the Frame update routine, the screen will be updated directly. If the window is invisible, there is no associated Bitmap, and all that is necessary is to update the Frame data structure, not the image. If the window is partially visible, the Frame routine is called to update the image in the off-screen Bitmap, which may require regenerating it from the text of the window. The Flayer code then clips this Bitmap against the Bitmaps of all Frames in front of the Frame being modified, and the remainder is copied to the display.

This is much faster than recreating the image off-screen for every change, or clipping all the changes made to the image during its update. Unfortunately, these caches can also consume prohibitive amounts of memory, so they are freed fairly liberally — after every change to the front-to-back order of the Flayers. The result is that the off-screen Bitmaps exist only while multi-window changes are occurring, which is the only time the performance improvement they provide is needed. Also, the user interface causes fully-obscured windows to be the easiest to make — creating a canonically sized and placed window requires only a button click — which reduces the need for caching still further.

Screen update

Only two low-level primitives are needed for incremental update: bitblt, which copies rectangles of pixels, and string (which in turn calls bitblt), which draws a null-terminated character string in a Bitmap. A Frame contains a list of Boxes, each of which defines a horizontal strip of text in the window (see Figure 7). A Box has a character string str, and a Rectangle rect that defines the location of the strip in the window. (The text in str is stored in the Box separately from the Rasp associated with the window's file, so Boxes are self-contained.) The invariant is that the image of the Box can be reproduced by calling string with argument str to draw the string in rect, and the resulting picture fits perfectly within rect. In other words, the Boxes define the tiling of the window. The tiling may be complicated by long lines of text, which are folded onto the next line. Some editors use horizontal scrolling to avoid this complication, but to be comfortable this technique requires that lines not be *too* long; sam has no such restriction. Also, and perhaps more importantly, UNIX programs and terminals traditionally fold long lines to make their contents fully visible.

Two special kinds of Boxes contain a single character: either a newline or a tab. Newlines and tabs are white space. A newline Box always extends to the right edge of the window, forcing the following Box to the next line. The width of a tab depends on where it is located: it forces the next Box to begin at a tab location. Tabs also have a minimum width equivalent to a blank (blanks are drawn by string and are not treated specially); newlines have a minimum width of zero.

The update algorithms always use the Bitmap image of the text (either the display or cache Bitmap); they never examine the characters within a Box except when the Box needs to be split in two. Before a change, the window consists of a tiling of Boxes; after the change the window is tiled differently. The update algorithms

	for(i=0; i<NL; i++){		/* for each element */	

Figure 7. A line of text showing its Boxes. *The first two blank* Boxes *contain tabs; the last contains a newline. Spaces are handled as ordinary characters.*

rearrange the tiles in place, without backup storage. The algorithms are not strictly optimal — for example, they can clear a pixel that is later going to be written upon — but they never move a tile that doesn't need to be moved, and they move each tile at most once. Frinsert on a Blit can absorb over a thousand characters a second if the strings being inserted are a few tens of characters long.

Consider frdelete. Its job is to delete a substring from a Frame and restore the image of the Frame. The image of a substring has a peculiar shape (see Figure 2) comprising possibly a partial line, zero or more full lines, and possibly a final partial line. For reference, call this the *Z-shape.* Frdelete begins by splitting, if necessary, the Boxes containing the ends of the substring so the substring begins and ends on Box boundaries. Because the substring is being deleted, its image is not needed, so the Z-shape is then cleared. Then, tiles (that is, the images of Boxes) are copied, using bitblt, from immediately after the Z-shape to the beginning of the Z-shape, resulting in a new Z-shape. (Boxes whose contents would span two lines in the new position must first be split.)

Copying the remainder of the Frame tile by tile this way will clearly accomplish the deletion but eventually, typically when the copying algorithm encounters a tab or newline, the old and new x coordinates of the tile to be copied are the same. This correspondence implies that the Z-shape has its beginning and ending edges aligned vertically, and a sequence of at most two bitblts can be used to copy the remaining tiles. The last step is to clear out the resulting empty space at the bottom of the window; the number of lines to be cleared is the number of complete lines in the Z-shape closed by the final bitblts. The final step is to merge horizontally adjacent Boxes of plain text. The complete source to frdelete is less than 100 lines of C.

frinsert is more complicated because it must do four passes: one to construct the Box list for the inserted string, one to reconnoitre, one to copy (in opposite order to frdelete) the Boxes to make the hole for the new text, and finally one to copy the new text into place. Overall, though, frinsert has a similar flavor to frdelete, and needn't be described further. Frinsert and its subsidiary routines comprise 211 lines of C.

The terminal source code is 3024 lines of C, and the host source is 5797 lines.

Discussion

History

The immediate ancestor of sam was the original text editor for the Blit, called jim. Sam inherited jim's two-process structure and mouse language almost unchanged, but jim suffered from several drawbacks that were addressed in the design of sam. The most important of these was the lack of a command language. Although jim was easy to use for simple editing, it provided no direct help with large or repetitive editing tasks. Instead, it provided a command to pass selected text through a shell pipeline, but this was no more satisfactory than could be expected of a stopgap measure.

Jim was written primarily as a vehicle for experimenting with a mouse-based interface to text, and the experiment was successful. Jim had some spin-offs: mux, the second window system for the Blit, is essentially a multiplexed version of the terminal part of jim; and the debugger pi's user interface[20] was closely modeled on jim's. But after a couple of years, jim had become difficult to maintain and limiting to use,

and its replacement was overdue.

I began the design of sam by asking jim customers what they wanted. This was probably a mistake; the answers were essentially a list of features to be found in other editors, which did not provide any of the guiding principles I was seeking. For instance, one common request was for a ''global substitute,'' but no one suggested how to provide it within a cut-and-paste editor. I was looking for a scheme that would support such specialized features comfortably in the context of some general command language. Ideas were not forthcoming, though, particularly given my insistence on removing all limits on file sizes, line lengths and so on. Even worse, I recognized that, since the mouse could easily indicate a region of the screen that was not an integral number of lines, the command language would best forget about newlines altogether, and that meant the command language had to treat the file as a single string, not an array of lines.

Eventually, I decided that thinking was not getting me very far and it was time to try building. I knew that the terminal part could be built easily — that part of jim behaved acceptably well — and that most of the hard work was going to be in the host part: the file interface, command interpreter and so on. Moreover, I had some ideas about how the architecture of jim could be improved without destroying its basic structure, which I liked in principle but which hadn't worked out as well as I had hoped. So I began by designing the file data structure, starting with the way jim worked — comparable to a single structure merging Disc and Buffer, which I split to make the cache more general — and thinking about how global substitute could be implemented. The answer was clearly that it had to be done in two passes, and the transcript-oriented implementation fell out naturally.

Sam was written bottom-up, starting from the data structures and algorithms for manipulating text, through the command language and up to the code for maintaining the display. In retrospect, it turned out well, but this implementation method is not recommended in general. There were several times when I had a large body of interesting code assembled and no clue how to proceed with it. The command language, in particular, took almost a year to figure out, but can be implemented (given what was there at the beginning of that year) in a day or two. Similarly, inventing the Rasp data structure delayed the connection of the host and terminal pieces by another few months. Sam took about two years to write, although only about four months were spent actually working on it.

Part of the design process was unusual: the subset of the protocol that maintains the Rasp was simulated, debugged and verified by an automatic protocol analyzer,[21] and was bug-free from the start. The rest of the protocol, concerned mostly with keeping menus up to date, was unfortunately too unwieldy for such analysis, and was debugged by more traditional methods, primarily by logging in a file all messages in and out of the host.

Reflections

Sam is essentially the only interactive editor used by the sixty or so members of the computing science research center in which I work. The same could not be said of jim; the lack of a command language kept some people from adopting it. The union of a user interface as comfortable as jim's with a command language as powerful as ed's† is essential to sam's success. When sam was first made available to the jim community, almost everyone switched to it within two or three days. In the months that followed, even people who had never adopted jim started using sam exclusively.

†The people who criticize ed as an interactive program often forget that it and its close relative sed[7] still thrive as programmable editors. The strength of these programs is independent of their convenience for interactive editing.

To be honest, ed still gets occasional use, but usually when something quick needs to be done and the overhead of downloading the terminal part of sam isn't worth the trouble. Also, as a 'line' editor, sam −d is a bit odd; when using a good old ASCII terminal, it's comforting to have a true line editor. But it is fair to say that sam's command language has displaced ed's for most of the complicated editing that has kept line editors (that is, command-driven editors) with us.

Sam's command language is even fancier than ed's, and most sam customers don't come near to using all its capabilities. Does it need to be so sophisticated? I think the answer is yes, for two reasons.

First, the *model* for sam's command language is really relatively simple, and certainly simpler than that of ed. For instance, there is only one kind of textual loop in sam — the x command — while ed has three (the g command, the global flag on substitutions, and the implicit loop over lines in multi-line substitutions). Also, ed's substitute command is necessary to make changes within lines, but in sam the s command is more of a familiar convenience than a necessity; c and t can do all the work.

Second, given a community that expects an editor to be about as powerful as ed, it's hard to see how sam could really be much simpler and still satisfy that expectation. People want to do ''global substitutes,'' and most are content to have the recipe for that and a few other fancy changes. The sophistication of the command language is really just a veneer over a design that makes it possible to do global substitutes in a screen editor. Some people will always want something more, however, and it's gratifying to be able to provide it. The real power of sam's command language comes from composability of the operators, which is by nature orthogonal to the underlying model. In other words, sam is not itself complex, but it makes complex things possible. If you don't want to do anything complex, you can ignore the complexity altogether, and many people do so.

Sometimes I am asked the opposite question: why didn't I just make sam a real programmable editor, with macros and variables and so on? The main reason is a matter of taste: I like the editor to be the same every time I use it. There is one technical reason, though: programmability in editors is largely a workaround for insufficient interactivity. Programmable editors are used to make particular, usually short-term, things easy to do, such as by providing shorthands for common actions. If things are generally easy to do in the first place, shorthands are not as helpful. Sam makes common editing operations very easy, and the solutions to complex editing problems seem commensurate with the problems themselves. Also, the ability to edit the sam window makes it easy to repeat commands — it only takes a mouse button click to execute a command again.

Pros and cons

Sam has several other good points, and its share of problems. Among the good things is the idea of structural regular expressions, whose usefulness has only begun to be explored. They were arrived at serendipitously when I attempted to distill the essence of ed's way of doing global substitution and recognized that the looping command in ed was implicitly imposing a structure (an array of lines) on the file.

Another of sam's good things is its undo capability. I had never before used an editor with a true undo, but I would never go back now. Undo *must* be done well, but if it is, it can be relied on. For example, it's safe to experiment if you're not sure how to write some intricate command, because if you make a mistake, it can be fixed simply and reliably. I learned two things about undo from writing sam: first, it's easy to provide if you design it in from the beginning, and second, it's necessary, particularly if the system has some subtle properties that may be unfamiliar or error-prone for users.

Sam's lack of internal limits and sizes is a virtue. Because it avoids all fixed-size tables and data structures, sam is able to make global changes to files that some of our other tools cannot even read. Moreover, the design keeps the performance linear when doing such operations, although I must admit sam does get slow when editing a huge file.

Now, the problems. Externally, the most obvious is that it is poorly integrated into the surrounding window system. By design, the user interface in sam feels almost identical to that of mux, but a thick wall separates text in sam from the programs running in mux. For instance, the 'snarf buffer' in sam must be maintained separately from that in mux. This is regrettable, but probably necessary given the unusual configuration of the system, with a programmable terminal on the far end of an RS-232 link.

Sam is reliable; otherwise, people wouldn't use it. But it was written over such a long time, and has so many new (to me) ideas in it, that I would like to see it done over again to clean up the code and remove many of the lingering problems in the implementation. The worst part is in the interconnection of the host and terminal parts, which might even be able to go away in a redesign for a more conventional window system. The program must be split in two to use the terminal effectively, but the low bandwidth of the connection forces the separation to occur in an inconvenient part of the design if performance is to be acceptable. A simple remote procedure call protocol driven by the host, emitting only graphics commands, would be easy to write but wouldn't have nearly the necessary responsiveness. On the other hand, if the terminal were in control and requested much simpler file services from the host, regular expression searches would require that the terminal read the entire file over its RS-232 link, which would be unreasonably slow. A compromise in which either end can take control is necessary. In retrospect, the communications protocol should have been designed and verified formally, although I do not know of any tool that can adequately relate the protocol to its implementation.

Not all of sam's users are comfortable with its command language, and few are adept. Some (venerable) people use a sort of ''ed subset'' of sam's command language, and even ask why sam's command language is not exactly ed's. (The reason, of course, is that sam's model for text does not include newlines, which are central to ed. Making the text an array of newlines to the command language would be too much of a break from the seamless model provided by the mouse. Some editors, such as vi, are willing to make this break, though.) The difficulty is that sam's syntax is so close to ed's that people believe it *should* be the same. I thought, with some justification in hindsight, that making sam similar to ed would make it easier to learn and to accept. But I may have overstepped and raised the users' expectations too much. It's hard to decide which way to resolve this problem.

Finally, there is a tradeoff in sam that was decided by the environment in which it runs: sam is a multi-file editor, although in a different system there might instead be multiple single-file editors. The decision was made primarily because starting a new program in a Blit is time-consuming. If the choice could be made freely, however, I would still choose the multi-file architecture, because it allows groups of files to be handled as a unit; the usefulness of the multi-file commands is incontrovertible. It is delightful to have the source to an entire program available at your fingertips.

Acknowledgements

Tom Cargill suggested the idea behind the Rasp data structure. Norman Wilson and Ken Thompson influenced the command language. This paper was improved by comments from Al Aho, Jon Bentley, Chris Fraser, Gerard Holzmann, Brian Kernighan, Ted Kowalski, Doug McIlroy and Dennis Ritchie.

REFERENCES

1. R. Pike, 'The Blit: a multiplexed graphics terminal,' *AT&T Bell Labs. Tech. J.,* **63**, (8), 1607–1631 (1984).
2. L. Johnson, *MacWrite,* Apple Computer Inc., Cupertino, Calif. 1983.
3. B. Lampson, 'Bravo Manual,' in *Alto User's Handbook,* pp. 31–62, Xerox Palo Alto Research Center, Palo Alto, Calif. 1979.
4. W. Teitelman, 'A tour through Cedar,' *IEEE Software,* **1** (2), 44–73 (1984).
5. J. Gutknecht, 'Concepts of the text editor Lara,' *Comm. ACM,* **28**, (9), 942–960 (1985).
6. Bell Telephone Laboratories, *UNIX Programmer's Manual,* Holt, Rinehart and Winston, New York 1983.
7. B. W. Kernighan and R. Pike, *The Unix Programming Environment,* Prentice-Hall, Englewood Cliffs, New Jersey 1984.
8. *Unix Time-Sharing System Programmer's Manual, Research Version, Ninth Edition, Volume 1,* AT&T Bell Laboratories, Murray Hill, New Jersey 1986.
9. *Unix Time-Sharing System Programmer's Manual, 4.1 Berkeley Software Distribution, Volumes 1 and 2C,* University of California, Berkeley, Calif. 1981.
10. R. Pike, 'Structural Regular Expressions,' *Proc. EUUG Spring Conf., Helsinki 1987,* Eur. Unix User's Group, Buntingford, Herts, UK 1987.
11. A. Goldberg, *Smalltalk-80 – The Interactive Programming Environment,* Addison-Wesley, Reading, Mass. 1984.
12. K. Thompson, 'Regular expression search algorithm,' *Comm. ACM,* **11**, (6), 419–422 (1968).
13. A. V. Aho, J. E. Hopcroft and J. D. Ullman, *The Design and Analysis of Computer Algorithms,* Addison-Wesley, Reading, Mass. 1974.
14. B. W. Kernighan and D. M. Ritchie, *The C Programming Language,* Prentice-Hall, Englewood Cliffs, New Jersey 1978.
15. W. M. Waite, 'The cost of lexical analysis,' *Softw. Pract. Exp.,* **16**, (5), 473–488 (1986).
16. C. W. Fraser, 'A generalized text editor,' *Comm. ACM,* **23**, (3), 154–158 (1980).
17. R. Pike, 'Graphics in overlapping bitmap layers,' *ACM Trans. on Graph.,* **2**, (2) 135–160 (1983).
18. L. J. Guibas and J. Stolfi, 'A language for bitmap manipulation,' *ACM Trans. on Graph.,* **1**, (3), 191–214 (1982).
19. R. Pike, B. Locanthi and J. Reiser, 'Hardware/software trade-offs for bitmap graphics on the Blit,' *Softw. Pract. Exp.,* **15**, (2), 131–151 (1985).
20. T. A. Cargill, 'The feel of Pi,' *Winter USENIX Conference Proceedings, Denver 1986,* 62–71, USENIX Assoc., El Cerrito, CA.
21. G. J. Holzmann, 'Tracing protocols,' *AT&T Tech. J.,* **64**, (10), 2413–2434 (1985).

Acme: A User Interface for Programmers

Rob Pike
rob@plan9.bell-labs.com

ABSTRACT

A hybrid of window system, shell, and editor, Acme gives text-oriented applications a clean, expressive, and consistent style of interaction. Traditional window systems support interactive client programs and offer libraries of pre-defined operations such as pop-up menus and buttons to promote a consistent user interface among the clients. Acme instead provides its clients with a fixed user interface and simple conventions to encourage its uniform use. Clients access the facilities of Acme through a file system interface; Acme is in part a file server that exports device-like files that may be manipulated to access and control the contents of its windows. Written in a concurrent programming language, Acme is structured as a set of communicating processes that neatly subdivide the various aspects of its tasks: display management, input, file server, and so on.

Acme attaches distinct functions to the three mouse buttons: the left selects text; the middle executes textual commands; and the right combines context search and file opening functions to integrate the various applications and files in the system.

Acme works well enough to have developed a community that uses it exclusively. Although Acme discourages the traditional style of interaction based on typescript windows—teletypes—its users find Acme's other services render typescripts obsolete.

History and motivation

The usual typescript style of interaction with Unix and its relatives is an old one. The typescript—an intermingling of textual commands and their output—originates with the scrolls of paper on teletypes. The advent of windowed terminals has given each user what amounts to an array of teletypes, a limited and unimaginative use of the powers of bitmap displays and mice. Systems like the Macintosh that do involve the mouse as an integral part of the interaction are geared towards general users, not experts, and certainly not programmers. Software developers, at least on time-sharing systems, have been left behind.

Some programs have mouse-based editing of text files and typescripts; ones I have built include the window systems `mux` [Pike88] and 8½ [Pike91] and the text editor Sam [Pike87]. These have put the programmer's mouse to some productive work, but not wholeheartedly. Even experienced users of these programs often retype text that could be grabbed with the mouse, partly because the menu-driven interface is imperfect and partly because the various pieces are not well enough integrated.

Originally appeared in *Proc. of the Winter 1994 USENIX Conf.,* pp. 223-234, San Francisco, CA

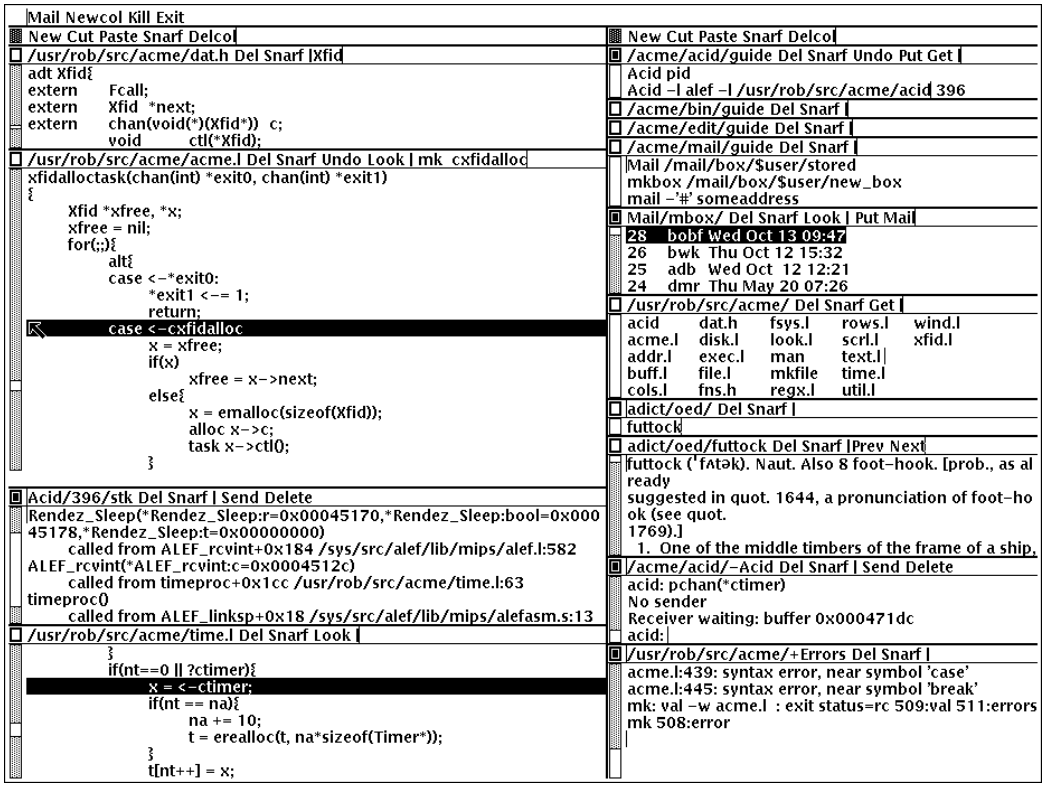

Figure 1. A small Acme screen—normally it runs on a larger display—demonstrating some of the details discussed in the text. The right column contains some guide files, a mailbox presented by Acme's mail program, the columnated display of files in Acme's own source directory, a couple of windows from the OED browser, a debugger window, and an error window showing diagnostics from a compilation. The left column holds a couple of source files (`dat.h` and `acme.l`), another debugger window displaying a stack trace, and a third source file (`time.l`). `Time.l` was opened from the debugger by clicking the right mouse button on a line in the stack window; the mouse cursor landed on the offending line of `acme.l` after a click on the compiler message.

Other programs—EMACS [Stal93] is the prime example—offer a high degree of integration but with a user interface built around the ideas of cursor-addressed terminals that date from the 1970's. They are still keyboard-intensive and dauntingly complex.

The most ambitious attempt to face these issues was the Cedar system, developed at Xerox [Swei86]. It combined a new programming language, compilers, window system, even microcode—a complete system—to construct a productive, highly integrated and interactive environment for experienced users of compiled languages. Although successful internally, the system was so large and so tied to specific hardware that it never fledged.

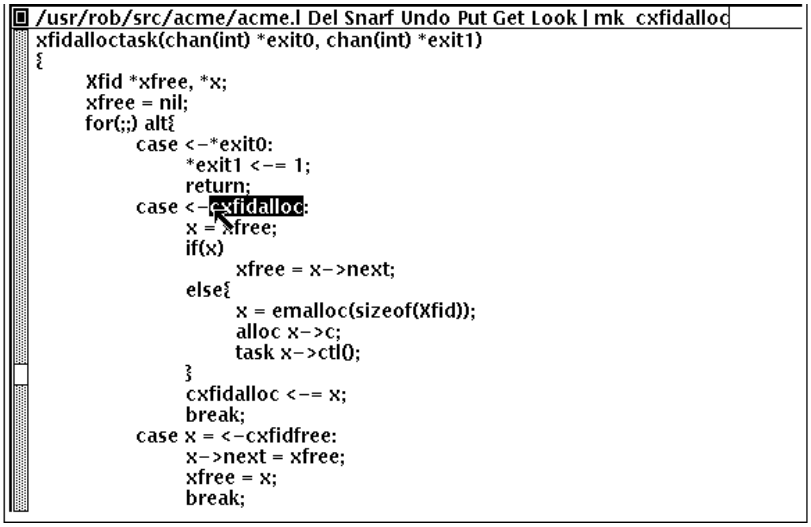

Figure 2. An Acme window showing a section of code. The upper line of text is the tag containing the file name, relevant commands, and a scratch area (right of the vertical bar); the lower portion of the window is the body, or contents, of the file. Here the scratch area contains a command for the middle button (`mk`) and a word to search for with the right button (`cxfidalloc`). The user has just clicked the right button on `cxfidalloc` and Acme has searched for the word, highlighted it, and moved the mouse cursor there. The file has been modified: the center of the layout box is black and the command `Put` appears in the tag.

Cedar was, however, the major inspiration for Oberon [Wirt89], a system of similar scope but much smaller scale. Through careful selection of Cedar's ideas, Oberon shows that its lessons can be applied to a small, coherent system that can run efficiently on modest hardware. In fact, Oberon probably errs too far towards simplicity: a single-process system with weak networking, it seems an architectural throwback.

Acme is a new program, a combined window system, editor, and shell, that applies some of the ideas distilled by Oberon. Where Oberon uses objects and modules within a programming language (also called Oberon), Acme uses files and commands within an existing operating system (Plan 9). Unlike Oberon, Acme does not yet have support for graphical output, just text. At least for now, the work on Acme has concentrated on producing the smoothest user interface possible for a programmer at work.

The rest of this paper describes Acme's interface, explains how programs can access it, compares it to existing systems, and finally presents some unusual aspects of its implementation.

User interface

Acme windows are arrayed in columns (Figure 1) and are used more dynamically than in an environment like X Windows or 8½ [Sche86, Pike91]. The system frequently creates them automatically and the user can order a new one with a single mouse button click. The initial placement of a new window is determined automatically, but the user may move an existing window anywhere by clicking or dragging a *layout box* in the upper left corner of the window.

Acme windows have two parts: a *tag* holding a single line of text, above a *body* holding zero or more lines (Figure 2). The body typically contains an image of a file

being edited or the editable output of a program, analogous to an EMACS shell window. The tag contains the name of the window (usually the name of the associated file or directory), some built-in commands, and a scratch area to hold arbitrary text. If a window represents a directory, the name in the tag ends with a slash and the body contains a list of the names of the files in the directory. Finally, each non-empty body holds a scroll bar at the left of the text.

Each column of windows also has a layout box and a tag. The tag has no special meaning, although Acme pre-loads it with a few built-in commands. There is also a tag across the whole display, also loaded with helpful commands and a list of active processes started by Acme.

Typing with the keyboard and selecting with the left button are as in many other systems, including the Macintosh, 8½, and Sam. The middle and right buttons are used, somewhat like the left button, to 'sweep' text, but the indicated text is treated in a way that depends on the text's location—*context*—as well as its content. This context, based on the directory of the file containing the text, is a central component of Acme's style of interaction.

Acme has no single notion of 'current directory'. Instead, every command, file name, action, and so on is interpreted or executed in the directory named by the tag of the window containing the command. For example, the string `mammals` in a window labeled `/lib/` or `/lib/insects` will be interpreted as the file name `/lib/mammals` if such a file exists.

Throughout Acme, the middle mouse button is used to execute commands and the right mouse button is used to locate and select files and text. Even when there are no true files on which to operate—for example when editing mail messages—Acme and its applications use consistent extensions of these basic functions. This idea is as vital to Acme as icons are to the Macintosh.

The middle button executes commands: text swept with the button pressed is underlined; when the button is released, the underline is removed and the indicated text is executed. A modest number of commands are recognized as built-ins: words like `Cut`, `Paste`, and `New` name functions performed directly by Acme. These words often appear in tags to make them always available, but the tags are not menus: any text anywhere in Acme may be a command. For example, in the tag or body of any window one may type `Cut`, select it with the left button, use the middle button to execute it, and watch it disappear again.

If the middle button indicates a command that is not recognized as a built-in, it is executed in the directory named by the tag of the window holding the text. Also, the file to be executed is searched for first in that directory. Standard input is connected to `/dev/null`, but standard and error outputs are connected to an Acme window, created if needed, called *dir*`/+Errors` where *dir* is the directory of the window. (Programs that need interactive input use a different interface, described below.) A typical use of this is to type `mk` (Plan 9's `make`) in the scratch area in the tag of a C source window, say `/sys/src/cmd/sam/regexp.c`, and execute it. Output, including compiler errors, appears in the window labeled `/sys/src/cmd/sam/+Errors`, so file names in the output are associated with the windows and directory holding the source. The `mk` command remains in the tag, serving as a sort of menu item for the associated window.

Like the middle button, the right button is used to indicate text by sweeping it out. The indicated text is not a command, however, but the argument of a generalized search operator. If the text, perhaps after appending it to the directory of the window containing it, is the name of an existing file, Acme creates a new window to hold the file and reads it in. It then moves the mouse cursor to that window. If the file is already

loaded into Acme, the mouse motion happens but no new window is made. For example, indicating the string `sam.h` in

```
#include "sam.h"
```

in a window on the file `/sys/src/cmd/sam/regexp.c` will open the file `/sys/src/cmd/sam/sam.h`.

If the file name is followed immediately by a colon and a legal address in Sam notation (for example a line number or a regular expression delimited in slashes or a comma-separated compound of such addresses), Acme highlights the target of that address in the file and places the mouse there. One may jump to line 27 of `dat.h` by indicating with the right button the text `dat.h:27`. If the file is not already open, Acme loads it. If the file name is null, for example if the indicated string is `:/^main/`, the file is assumed to be that of the window containing the string. Such strings, when typed and evaluated in the tag of a window, amount to context searches.

If the indicated text is not the name of an existing file, it is taken to be literal text and is searched for in the body of the window containing the text, highlighting the result as if it were the result of a context search.

For the rare occasion when a file name *is* just text to search for, it can be selected with the left button and used as the argument to a built-in `Look` command that always searches for literal text.

Nuances and heuristics

A user interface should not only provide the necessary functions, it should also *feel* right. In fact, it should almost not be felt at all; when one notices a user interface, one is distracted from the job at hand [Pike88]. To approach this invisibility, some of Acme's properties and features are there just to make the others easy to use. Many are based on a fundamental principle of good design: let the machine do the work.

Acme tries to avoid needless clicking and typing. There is no 'click-to-type', eliminating a button click. There are no pop-up or pull-down menus, eliminating the mouse action needed to make a menu appear. The overall design is intended to make text on the screen useful without copying or retyping; the ways in which this happens involve the combination of many aspects of the interface.

Acme tiles its windows and places them automatically to avoid asking the user to place and arrange them. For this policy to succeed, the automatic placement must behave well enough that the user is usually content with the location of a new window. The system will never get it right all the time, but in practice most windows are used at least for a while where Acme first places them. There have been several complete rewrites of the heuristics for placing a new window, and with each rewrite the system became noticeably more comfortable. The rules are as follows, although they are still subject to improvement. The window appears in the 'active' column, that most recently used for typing or selecting. Executing and searching do not affect the choice of active column, so windows of commands and such do not draw new windows towards them, but rather let them form near the targets of their actions. Output (error) windows always appear towards the right, away from edited text, which is typically kept towards the left. Within the column, several competing desires are balanced to decide where and how large the window should be: large blank spaces should be consumed; existing text should remain visible; existing large windows should be divided before small ones; and the window should appear near the one containing the action that caused its creation.

Acme binds some actions to chords of mouse buttons. These include `Cut` and `Paste` so these common operations can be done without moving the mouse. Another is a way to apply a command in one window to text (often a file name) in another, avoiding the actions needed to assemble the command textually.

Another way Acme avoids the need to move the mouse is instead to move the cursor to where it is likely to be used next. When a new window is made, Acme moves the cursor to the new window; in fact, to the selected text in that window. When the user deletes a newly made window, the cursor is returned to the point it was before the window was made, reducing the irritation of windows that pop up to report annoying errors.

When a window is moved, Acme moves the cursor to the layout box in its new place, to permit further adjustment without moving the mouse. For example, when a click of the left mouse button on the layout box grows the window, the cursor moves to the new location of the box so repeated clicks, without moving the mouse, continue to grow it.

Another form of assistance the system can offer is to supply precision in pointing the mouse. The best-known form of this is 'double-clicking' to select a word rather than carefully sweeping out the entire word. Acme provides this feature, using context to decide whether to select a word, line, quoted string, parenthesized expression, and so on. But Acme takes the idea much further by applying it to execution and searching. A *single* click, that is, a null selection, with either the middle or right buttons, is expanded automatically to indicate the appropriate text containing the click. What is appropriate depends on the context.

For example, to execute a single-word command such as `Cut`, it is not necessary to sweep the entire word; just clicking the button once with the mouse pointing at the word is sufficient. 'Word' means the largest string of likely file name characters surrounding the location of the click: click on a file name, run that program. On the right button, the rules are more complicated because the target of the click might be a file name, file name with address, or just plain text. Acme examines the text near the click to find a likely file name; if it finds one, it checks that it names an existing file (in the directory named in the tag, if the name is relative) and if so, takes that as the result, after extending it with any address that may be present. If there is no file with that name, Acme just takes the largest alphanumeric string under the click. The effect is a natural overloading of the button to refer to plain text as well as file names.

First, though, if the click occurs over the left-button-selected text in the window, that text is taken to be what is selected. This makes it easy to skip through the occurrences of a string in a file: just click the right button on some occurrence of the text in the window (perhaps after typing it in the tag) and click once for each subsequent occurrence. It isn't even necessary to move the mouse between clicks; Acme does that. To turn a complicated command into a sort of menu item, select it: thereafter, clicking the middle button on it will execute the full command.

As an extra feature, Acme recognizes file names in angle brackets <> as names of files in standard directories of include files, making it possible for instance to look at `<stdio.h>` with a single click.

Here's an example to demonstrate how the actions and defaults work together. Assume `/sys/src/cmd/sam/regexp.c` is open and has been edited. We write it (execute `Put` in the tag; once the file is written, Acme removes the word from the tag) and type `mk` in the tag. We execute `mk` and get some errors, which appear in a new window labeled `/sys/src/cmd/sam/+Errors`. The cursor moves automatically to that window. Say the error is

```
main.c:112: incompatible types on assignment to 'pattern'
```

We move the mouse slightly and click the right button at the left of the error message; Acme makes a new window, reads `/sys/src/cmd/main.c` into it, selects line 112 and places the mouse there, right on the offending line.

Coupling to existing programs

Acme's syntax for file names and addresses makes it easy for other programs to connect automatically to Acme's capabilities. For example, the output of

```
grep -n variable *.[ch]
```

can be used to help Acme step through the occurrences of a variable in a program; every line of output is potentially a command to open a file. The file names need not be absolute, either: the output appears in a window labeled with the directory in which `grep` was run, from which Acme can derive the full path names.

When necessary, we have changed the output of some programs, such as compiler error messages, to match Acme's syntax. Some might argue that it shouldn't be necessary to change old programs, but sometimes programs need to be updated when systems change, and consistent output benefits people as well as programs. A historical example is the retrofitting of standard error output to the early Unix programs when pipes were invented.

Another change was to record full path names in the symbol table of executables, so line numbers reported by the debugger are absolute names that may be used directly by Acme; it's not necessary to run the debugger in the source directory. (This aids debugging even without Acme.)

A related change was to add lines of the form

```
#pragma src "/sys/src/libregexp"
```

to header files; coupled with Acme's ability to locate a header file, this provides a fast, keyboardless way to get the source associated with a library.

Finally, Acme directs the standard output of programs it runs to windows labeled by the directory in which the program is run. Acme's splitting of the output into directory-labeled windows is a small feature that has a major effect: local file names printed by programs can be interpreted directly by Acme. By indirectly coupling the output of programs to the input, it also simplifies the management of software that occupies multiple directories.

Coupling to new programs

Like many Plan 9 programs, Acme offers a programmable interface to other programs by acting as a file server. The best example of such a file server is the window system 8½ [Pike91], which exports files with names such as `screen`, `cons`, and `mouse` through which applications may access the I/O capabilities of the windows. 8½ provides a *distinct* set of files for each window and builds a private file name space for the clients running 'in' each window; clients in separate windows see distinct files with the same names (for example `/dev/mouse`). Acme, like the process file system [PPTTW93], instead associates each window with a directory of files; the files of each window are visible to any application. This difference reflects a difference in how the systems are used: 8½ tells a client what keyboard and mouse activity has happened in its window; Acme tells a client what changes that activity wrought on any window it asks about. Putting it another way, 8½ enables the construction of interactive applications; Acme provides the interaction for applications.

The root of Acme's file system is mounted using Plan 9 operations on the directory `/mnt/acme`. In that root directory appears a directory for each window, numbered with the window's identifier, analogous to a process identifier, for example `/mnt/acme/27`. The window's directory contains 6 files: `/mnt/acme/27/addr`, `body`, `ctl`, `data`, `event`, and `tag`. The `body` and `tag` files contain the text of the respective parts of the window; they may be read to recover the contents. Data written to these files is appended to the text; `seeks` are ignored. The `addr` and `data` files

provide random access to the contents of the body. The `addr` file is written to set a character position within the body; the `data` file may then be read to recover the contents at that position, or written to change them. (The tag is assumed small and special-purpose enough not to need special treatment. Also, `addr` indexes by character position, which is not the same as byte offset in Plan 9's multi-byte character set [Pike93]). The format accepted by the `addr` file is exactly the syntax of addresses within the user interface, permitting regular expressions, line numbers, and compound addresses to be specified. For example, to replace the contents of lines 3 through 7, write the text

```
3,7
```

to the `addr` file, then write the replacement text to the `data` file. A zero-length write deletes the addressed text; further writes extend the replacement.

The control file, `ctl`, may be written with commands to effect actions on the window; for example the command

```
name /adm/users
```

sets the name in the tag of the window to `/adm/users`. Other commands allow deleting the window, writing it to a file, and so on. Reading the `ctl` file recovers a fixed-format string containing 5 textual numbers—the window identifier, the number of characters in the tag, the number in the body, and some status information—followed by the text of the tag, up to a newline.

The last file, `event`, is the most unusual. A program reading a window's `event` file is notified of all changes to the text of the window, and is asked to interpret all middle- and right-button actions. The data passed to the program is fixed-format and reports the source of the action (keyboard, mouse, external program, etc.), its location (what was pointed at or modified), and its nature (change, search, execution, etc.). This message, for example,

```
MI15 19 0 4 time
```

reports that actions of the mouse (`M`) inserted in the body (capital `I`) the 4 characters of `time` at character positions 15 through 19; the zero is a flag word. Programs may apply their own interpretations of searching and execution, or may simply reflect the events back to Acme, by writing them back to the `event` file, to have the default interpretation applied. Some examples of these ideas in action are presented below.

Notice that changes to the window are reported after the fact; the program is told about them but is not required to act on them. Compare this to a more traditional interface in which a program is told, for example, that a character has been typed on the keyboard and must then display and interpret it. Acme's style stems from the basic model of the system, in which any number of agents—the keyboard, mouse, external programs writing to `data` or `body`, and so on—may change the contents of a window. The style is efficient: many programs are content to have Acme do most of the work and act only when the editing is completed. An example is the Acme mail program, which can ignore the changes made to a message being composed and just read its body when asked to send it. A disadvantage is that some traditional ways of working are impossible. For example, there is no way 'to turn off echo': characters appear on the screen and are read from there; no agent or buffer stands between the keyboard and the display.

There are a couple of other files made available by Acme in its root directory rather than in the directory of each window. The text file `/mnt/acme/index` holds a list of all window names and numerical identifiers, somewhat analogous to the output of the `ps` command for processes. The most important, though, is `/mnt/acme/new`, a directory that makes new windows, similar to the `clone` directory in the Plan 9 network

devices [Pres93]. The act of opening any file in `new` creates a new Acme window; thus the shell command

```
grep -n var *.c > /mnt/acme/new/body
```

places its output in the body of a fresh window. More sophisticated applications may open `new/ctl`, read it to discover the new window's identifier, and then open the window's other files in the numbered directory.

Acme-specific programs

Although Acme is in part an attempt to move beyond typescripts, they will probably always have utility. The first program written for Acme was therefore one to run a shell or other traditional interactive application in a window, the Acme analog of `xterm`. This program, `win`, has a simple structure: it acts as a two-way intermediary between Acme and the shell, cross-connecting the standard input and output of the shell to the text of the window. The style of interaction is modeled after `mux` [Pike88]: standard output is added to the window at the *output point;* text typed after the output point is made available on standard input when a newline is typed. After either of these actions, the output point is advanced. This is different from the working of a regular terminal, permitting cut-and-paste editing of an input line until the newline is typed. Arbitrary editing may be done to any text in the window. The implementation of `win`, using the `event`, `addr`, and `data` files, is straightforward. `Win` needs no code for handling the keyboard and mouse; it just monitors the contents of the window. Nonetheless, it allows Acme's full editing to be applied to shell commands. The division of labor between `win` and `Acme` contrasted with `xterm` and the X server demonstrates how much work Acme handles automatically. `Win` is implemented by a single source file 560 lines long and has no graphics code.

`Win` uses the middle and right buttons to connect itself in a consistent way with the rest of Acme. The middle button still executes commands, but in a style more suited to typescripts. Text selected with the middle button is treated as if it had been typed after the output point, much as a similar feature in `xterm` or 8½, and therefore causes it to be 'executed' by the application running in the window. Right button actions are reflected back to Acme but refer to the appropriate files because `win` places the name of the current directory in the tag of the window. If the shell is running, a simple shell function replacing the `cd` command can maintain the tag as the shell navigates the file system. This means, for example, that a right button click on a file mentioned in an `ls` listing opens the file within Acme.

Another Acme-specific program is a mail reader that begins by presenting, in a window, a listing of the messages in the user's mailbox, one per line. Here the middle and right button actions are modified to refer to mail commands and messages, but the change feels natural. Clicking the right button on a line creates a new window and displays the message there, or, if it's already displayed, moves the mouse to that window. The metaphor is that the mailbox is a directory whose constituent files are messages. The mail program also places some relevant commands in the tag lines of the windows; for example, executing the word `Reply` in a message's tag creates a new window in which to compose a message to the sender of the original; `Post` then dispatches it. In such windows, the addressee is just a list of names on the first line of the body, which may be edited to add or change recipients. The program also monitors the mailbox, updating the 'directory' as new messages arrive.

The mail program is as simple as it sounds; all the work of interaction, editing, and management of the display is done by Acme. The only difficult sections of the 1200 lines of code concern honoring the external protocols for managing the mailbox and connecting to `sendmail`.

One of the things Acme does not provide directly is a facility like Sam's command language to enable actions such as global substitution; within Acme, all editing is done manually. It is easy, though, to write external programs for such tasks. In this, Acme comes closer to the original intent of Oberon: a directory, `/acme/edit`, contains a set of tools for repetitive editing and a template or 'guide' file that gives examples of its use. Acme's editing guide, `/acme/edit/guide`, looks like this:

```
e file | x '/regexp/' | c 'replacement'
e file:'0,$' | x '/.*word.*\n/' | p -n
e file | pipe command args ...
```

The syntax is reminiscent of Sam's command language, but here the individual one-letter commands are all stand-alone programs connected by pipes. Passed along the pipes are addresses, analogous to structural expressions in Sam terminology. The `e` command, unlike that of Sam, starts the process by generating the address (default dot, the highlighted selection) in the named files. The other commands are as in Sam: `p` prints the addressed text on standard output (the `-n` option is analogous to that of `grep`, useful in combination with the right mouse button); `x` matches a regular expression to the addressed (incoming) text, subdividing the text; `c` replaces the text; and so on. Thus, global substitution throughout a file, which would be expressed in Sam as

```
0,$ x/regexp/ c/replacement/
```

in Acme's editor becomes

```
e 'file:0,$' | x '/regexp/' | c 'replacement'
```

To use the Acme editing commands, open `/acme/edit/guide`, use the mouse and keyboard to edit one of the commands to the right form, and execute it with the middle button. Acme's context rules find the appropriate binaries in `/acme/edit` rather than `/bin`; the effect is to turn `/acme/edit` into a toolbox containing tools and instructions (the guide file) for their use. In fact, the source for these tools is also there, in the directory `/acme/edit/src`. This setup allows some control of the file name space for binary programs; not only does it group related programs, it permits the use of common names for uncommon jobs. For example, the single-letter names would be unwise in a directory in everyone's search path; here they are only visible when running editing commands.

In Oberon, such a collection would be called a *tool* and would consist of a set of entry points in a module and a menu-like piece of text containing representative commands that may be edited to suit and executed. There is, in fact, a tool called `Edit` in Oberon. To provide related functionality, Acme exploits the directory and file structure of the underlying system, rather than the module structure of the language; this fits well with Plan 9's file-oriented philosophy. Such tools are central to the working of Oberon but they are less used in Acme, at least so far. The main reason is probably that Acme's program interface permits an external program to remain executing in the background, providing its own commands as needed (for example, the `Reply` command in the mail program); Oberon uses tools to implement such services because its must invoke a fresh program for each command. Also, Acme's better integration allows more basic functions to be handled internally; the right mouse button covers a lot of the basic utility of the editing tools in Oberon. Nonetheless, as more applications are written for Acme, many are sure to take this Oberon tool-like form.

Comparison with other systems

Acme's immediate ancestor is Help [Pike92], an experimental system written a few years ago as a first try at exploring some of Oberon's ideas in an existing operating system. Besides much better engineering, Acme's advances over Help include the actions of the right button (Help had nothing comparable), the ability to connect long-running

programs to the user interface (Help had no analog of the `event` file), and the small but important change to split command output into windows labeled with the directory in which the commands run.

Most of Acme's style, however, derives from the user interface and window system of Oberon [Wirt89, Reis91]. Oberon includes a programming language and operating system, which Acme instead borrows from an existing system, Plan 9. When I first saw Oberon, in 1988, I was struck by the simplicity of its user interface, particularly its lack of menus and its elegant use of multiple mouse buttons. The system seemed restrictive, though—single process, single language, no networking, event-driven programming—and failed to follow through on some of its own ideas. For example, the middle mouse button had to be pointed accurately and the right button was essentially unused. Acme does follow through: to the basic idea planted by Oberon, it adds the ability to run on different operating systems and hardware, connection to existing applications including interactive ones such as shells and debuggers, support for multiple processes, the right mouse button's features, the default actions and context-dependent properties of execution and searching, and a host of little touches such as moving the mouse cursor that make the system more pleasant. At the moment, though, Oberon does have one distinct advantage: it incorporates graphical programs well into its model, an issue Acme has not yet faced.

Acme shares with the Macintosh a desire to use the mouse well and it is worth comparing the results. The mouse on the Macintosh has a single button, so menus are essential and the mouse must frequently move a long way to reach the appropriate function. An indication that this style has trouble is that applications provide keyboard sequences to invoke menu selections and users often prefer them. A deeper comparison is that the Macintosh uses pictures where Acme uses text. In contrast to pictures, text can be edited quickly, created on demand, and fine-tuned to the job at hand; consider adding an option to a command. It is also self-referential; Acme doesn't need menus because any text can be in effect a menu item. The result is that, although a Macintosh screen is certainly prettier and probably more attractive, especially to beginners, an Acme screen is more dynamic and expressive, at least for programmers and experienced users.

For its role in the overall system, Acme most resembles EMACS [Stal93]. It is tricky to compare Acme to EMACS, though, because there are many versions of EMACS and, since it is fully programmable, EMACS can in principle do anything Acme does. Also, Acme is much younger and therefore has not had the time to acquire as many features. The issue therefore is less what the systems can be programmed to do than how they are used. The EMACS versions that come closest to Acme's style are those that have been extended to provide a programming environment, usually for a language such as LISP [Alle92, Lucid92]. For richness of the existing interface, these EMACS versions are certainly superior to Acme. On the other hand, Acme's interface works equally well already for a variety of languages; for example, one of its most enthusiastic users works almost exclusively in Standard ML, a language nothing like C.

Where Acme excels is in the smoothness of its interface. Until recently, EMACS did not support the mouse especially well, and even with the latest version providing features such as 'extents' that can be programmed to behave much like Acme commands, many users don't bother to upgrade. Moreover, in the versions that provide extents, most EMACS packages don't take advantage of them.

The most important distinction is just that EMACS is fundamentally keyboard-based, while Acme is mouse-based.

People who try Acme find it hard to go back to their previous environment. Acme automates so much that to return to a traditional interface is to draw attention to the extra work it requires.

Concurrency in the implementation

Acme is about 8,000 lines of code in Alef, a concurrent object-oriented language syntactically similar to C [Alef]. Acme's structure is a set of communicating processes in a single address space. One subset of the processes drives the display and user interface, maintaining the windows; other processes forward mouse and keyboard activity and implement the file server interface for external programs. The language and design worked out well; as explained elsewhere [Pike89, Gans93, Reppy93], user interfaces built with concurrent systems can avoid the clumsy top-level event loop typical of traditional interactive systems.

An example of the benefits of the multi-process style is the management of the state of open files held by clients of the file system interface. The problem is that some I/O requests, such as reading the `event` file, may block if no data is available, and the server must maintain the state of (possibly many) requests until data appears. For example, in 8½, a single-process window system written in C, pending requests were queued in a data structure associated with each window. After activity in the window that might complete pending I/O, the data structure was scanned for requests that could now finish. This structure did not fit well with the rest of the program and, worse, required meticulous effort to guarantee correct behavior under all conditions (consider raw mode, reads of partial lines, deleting a window, multibyte characters, etc.).

Acme instead creates a new dedicated process for each I/O request. This process coordinates with the rest of the system using Alef's synchronous communication; its state implicitly encodes the state of the I/O request and obviates the need for queuing. The passage of the request through Acme proceeds as follows.

Acme contains a file server process, F, that executes a `read` system call to receive a Plan 9 file protocol (9P) message from the client [AT&T92]. The client blocks until Acme answers the request. F communicates with an allocation process, M, to acquire an object of type `Xfid` ('executing fid'; fid is a 9P term) to hold the request. M sits in a loop (reproduced in Figure 2) waiting for either a request for a new `Xfid` or notification that an existing one has finished its task. When an `Xfid` is created, an associated process, X, is also made. M queues idle `Xfids`, allocating new ones only when the list is empty. Thus, there is always a pool of `Xfids`, some executing, some idle.

The `Xfid` object contains a channel, `Xfid.c`, for communication with its process; the unpacked message; and some associated functions, mostly corresponding to 9P messages such as `Xfid.write` to handle a 9P write request.

The file server process F parses the message to see its nature—open, close, read, write, etc. Many messages, such as directory lookups, can be handled immediately; these are responded to directly and efficiently by F without invoking the `Xfid`, which is therefore maintained until the next message. When a message, such as a write to the display, requires the attention of the main display process and interlocked access to its data structures, F enables X by sending a function pointer on `Xfid.c`. For example, if the message is a write, F executes

```
x->c <-= Xfid.write;
```

which sends the address of `Xfid.write` on `Xfid.c`, waking up X.

The `Xfid` process, X, executes a simple loop:

```
void
Xfid.ctl(Xfid *x)
{
    for(;;){
        (*<-x->c)(x);       /* receive and execute message */
        bflush();           /* synchronize bitmap display */
        cxfidfree <-= x;    /* return to free list */
    }
}
```

Thus X will wake up with the address of a function to call (here `Xfid.write`) and execute it; once that completes, it returns itself to the pool of free processes by sending its address back to the allocator.

Although this sequence may seem complicated, it is just a few lines of code and is in fact far simpler than the management of the I/O queues in 8½. The hard work of synchronization is done by the Alef run time system. Moreover, the code worked the first time, which cannot be said for the code in 8½.

Undo

Acme provides a general undo facility like that of Sam, permitting textual changes to be unwound arbitrarily. The implementation is superior to Sam's, though, with much higher performance and the ability to 'redo' changes.

Sam uses a multi-pass algorithm that builds a transcript of changes to be made simultaneously and then executes them atomically. This was thought necessary because the elements of a repetitive command such as a global substitution should all be applied to the same initial file and implemented simultaneously; forming the complete transcript before executing any of the changes avoids the cumbersome management of addresses in a changing file. Acme, however, doesn't have this problem; global substitution is controlled externally and may be made incrementally by exploiting an observation: if the changes are sorted in address order and executed in reverse, changes will not invalidate the addresses of pending changes.

Acme therefore avoids the initial transcript. Instead, changes are applied directly to the file, with an undo transcript recorded in a separate list. For example, when text is added to a window, it is added directly and a record of what to delete to restore the state is appended to the undo list. Each undo action and the file are marked with a sequence number; actions with the same sequence number are considered a unit to be undone together. The invariant state of the structure is that the last action in the undo list applies to the current state of the file, even if that action is one of a related set from, for example, a global substitute. (In Sam, a related set of actions needed to be undone simultaneously.) To undo an action, pop the last item on the undo list, apply it to the file, revert it, and append it to a second, redo list. To redo an action, do the identical operation with the lists interchanged. The expensive operations occur only when actually undoing; in normal editing the overhead is minor. For example, Acme reads files about seven times faster than Sam, partly because of this improvement and partly because of a cleaner implementation.

Acme uses a temporary file to hold the text, keeping in memory only the visible portion, and therefore can edit large files comfortably even on small-memory machines such as laptops.

Future

Acme is still under development. Some things are simply missing. For example, Acme should support non-textual graphics, but this is being deferred until it can be done using a new graphics model being developed for Plan 9. Also, it is undecided how Acme's style of interaction should best be extended to graphical applications. On a

smaller scale, although the system feels smooth and comfortable, work continues to tune the heuristics and try new ideas for the user interface.

There need to be more programs that use Acme. Browsers for Usenet and AP News articles, the Oxford English Dictionary, and other such text sources exist, but more imaginative applications will be necessary to prove that Acme's approach is viable. One that has recently been started is an interface to the debugger Acid [Wint94], although it is still unclear what form it will ultimately take.

Acme shows that it is possible to make a user interface a stand-alone component of an interactive environment. By absorbing more of the interactive functionality than a simple window system, Acme off-loads much of the computation from its applications, which helps keep them small and consistent in their interface. Acme can afford to dedicate considerable effort to making that interface as good as possible; the result will benefit the entire system.

Acme is complete and useful enough to attract users. Its comfortable user interface, the ease with which it handles multiple tasks and programs in multiple directories, and its high level of integration make it addictive. Perhaps most telling, Acme shows that typescripts may not be the most productive interface to a time-sharing system.

Acknowledgements

Howard Trickey, Acme's first user, suffered buggy versions gracefully and made many helpful suggestions. Chris Fraser provided the necessary insight for the Acme editing commands.

References

[Alef] P. Winterbottom, ''Alef Language Reference Manual'', *Plan 9 Programmer's Manual,* AT&T Bell Laboratories, Murray Hill, NJ, 1992; revised in this volume.
[Alle92] *Allegro Common Lisp user Guide, Vol 2,* Chapter 14, "The Emacs-Lisp Interface". March 1992.
[AT&T92] Plan 9 Programmer's manual, Murray Hill, New Jersey, 1992.
[Far89] Far too many people, XTERM(1), Massachusetts Institute of Technology, 1989.
[Gans93] Emden R. Gansner and John H. Reppy, ''A Multi-threaded Higher-order User Interface Toolkit'', in *Software Trends, Volume 1, User Interface Software,* Bass and Dewan (Eds.), John Wiley & Sons 1993, pp. 61-80.
[Lucid92] Richard Stallman and Lucid, Inc., *Lucid GNU EMACS Manual,* March 1992.
[Pike87] Rob Pike, ''The Text Editor sam'', Softw. - Pract. and Exp., Nov 1987, Vol 17 #11, pp. 813-845; reprinted in this volume.
[Pike88] Rob Pike, ''Window Systems Should Be Transparent'', Comp. Sys., Summer 1988, Vol 1 #3, pp. 279-296.
[Pike89] Rob Pike, ''A Concurrent Window System'', Comp. Sys., Spring 1989, Vol 2 #2, pp. 133-153.
[PPTTW93] Rob Pike, Dave Presotto, Ken Thompson, Howard Trickey, and Phil Winterbottom, ''The Use of Name Spaces in Plan 9'', Op. Sys. Rev., Vol. 27, No. 2, April 1993, pp. 72-76, reprinted in this volume.
[Pike91] Rob Pike, ''8½, the Plan 9 Window System'', USENIX Summer Conf. Proc., Nashville, June, 1991, pp. 257-265, reprinted in this volume.
[Pike92] Rob Pike, ''A Minimalist Global User Interface'', Graphics Interface '92 Proc., Vancouver, 1992, pp. 282-293. An earlier version appeared under the same title in USENIX Summer Conf. Proc., Nashville, June, 1991, pp. 267-279.
[Pike93] Rob Pike and Ken Thompson, ''Hello World or Καλημέρα κόσμε or こんにちは 世界'', USENIX Winter Conf. Proc., San Diego, 1993, pp. 43-50, reprinted in this volume.
[Pres93] Dave Presotto and Phil Winterbottom, ''The Organization of Networks in Plan 9'', Proc. Usenix Winter 1993, pp. 271-287, San Diego, CA, reprinted in this volume.

[Reis91] Martin Reiser, *The Oberon System,* Addison Wesley, New York, 1991.
[Reppy93] John H. Reppy, ''CML: A higher-order concurrent language'', Proc. SIGPLAN'91 Conf. on Programming, Lang. Design and Impl., June, 1991, pp. 293-305.
[Sche86] Robert W. Scheifler and Jim Gettys, ''The X Window System'', ACM Trans. on Graph., Vol 5 #2, pp. 79-109.
[Stal93] Richard Stallman, *Gnu Emacs Manual, 9th edition, Emacs version 19.19,* MIT.
[Swei86] Daniel Sweinhart, Polle Zellweger, Richard Beach, and Robert Hagmann, ''A Structural View of the Cedar Programming Environment'', ACM Trans. Prog. Lang. and Sys., Vol. 8, No. 4, pp. 419-490, Oct. 1986.
[Wint94], Philip Winterbottom, ''Acid: A Debugger based on a Language'', USENIX Winter Conf. Proc., San Francisco, CA, 1993, reprinted in this volume.
[Wirt89] N. Wirth and J. Gutknecht, ''The Oberon System'', Softw. - Prac. and Exp., Sep 1989, Vol 19 #9, pp 857-894.

Plumbing and Other Utilities

Rob Pike

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Plumbing is a new mechanism for inter-process communication in Plan 9, specifically the passing of messages between interactive programs as part of the user interface. Although plumbing shares some properties with familiar notions such as cut and paste, it offers a more general data exchange mechanism without imposing a particular user interface.

The core of the plumbing system is a program called the *plumber*, which handles all messages and dispatches and reformats them according to configuration rules written in a special-purpose language. This approach allows the contents and context of a piece of data to define how it is handled. Unlike with drag and drop or cut and paste, the user doesn't need to deliver the data; the contents of a plumbing message, as interpreted by the plumbing rules, determine its destination.

The plumber has an unusual architecture: it is a language-driven file server. This design has distinct advantages. It makes plumbing easy to add to an existing, Unix-like command environment; it guarantees uniform handling of inter-application messages; it off-loads from those applications most of the work of extracting and dispatching messages; and it works transparently across a network.

Introduction

Data moves from program to program in myriad ways. Command-line arguments, shell pipe lines, cut and paste, drag and drop, and other user interface techniques all provide some form of interprocess communication. Then there are tricks associated with special domains, such as HTML hyperlinks or the heuristics mail readers use to highlight URLs embedded in mail messages. Some systems provide implicit ways to automate the attachment of program to data—the best known examples are probably the resource forks in MacOS and the file name extension 'associations' in Microsoft Windows—but in practice humans must too often carry their data from program to program.

Why should a human do the work? Usually there is one obvious thing to do with a piece of data, and the data itself suggests what this is. Resource forks and associations speak to this issue directly, but statically and narrowly and with little opportunity to control the behavior. Mechanisms with more generality, such as cut and paste or drag and drop, demand too much manipulation by the user and are (therefore) too error-prone.

We want a system that, given a piece of data, hands it to the appropriate application by default with little or no human intervention, while still permitting the user to override the defaults if desired.

The plumbing system is an attempt to address some of these issues in a single, coherent, central way. It provides a mechanism for formatting and sending arbitrary

messages between applications, typically interactive programs such as text editors, web browsers, and the window system, under the control of a central message-handling server called the *plumber*. Interactive programs provide application-specific connections to the plumber, triggering with minimal user action the transfer of data or control to other programs. The result is similar to a hypertext system in which all the links are implicit, extracted automatically by examining the data and the user's actions. It obviates cut and paste and other such hand-driven interprocess communication mechanisms. Plumbing delivers the goods to the right place automatically.

Overview

The plumber is implemented as a Plan 9 file server [Pike93]; programs send messages by writing them to the plumber's file `/mnt/plumb/send`, and receive messages by reading them from *ports*, which are other plumber files in `/mnt/plumb`. For example, `/mnt/plumb/edit` is by convention the file from which a text editor reads messages requesting it to open and display a file for editing. (See Figure 1.)

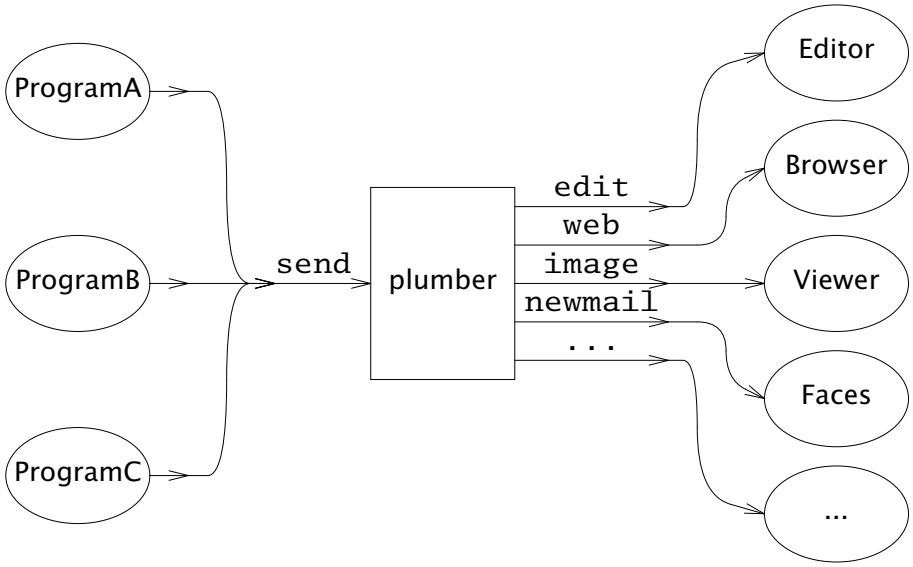

Figure 1. The plumber controls the flow of messages between applications. Programs write to the file `send` and receive on 'ports' of various names representing services such as `edit` or `web`. Although the figure doesn't illustrate it, some programs may both send and receive messages, and some ports are read by multiple applications.

The plumber takes messages from the `send` file and interprets their contents using rules defined by a special-purpose pattern-action language. The language specifies any rewriting of the message that is to be done by the plumber and defines how to dispose of a message, such as by sending it to a port or starting a new process to handle it.

The behavior is best described by example. Imagine that the user has, in a terminal emulator window, just run a compilation that has failed:

```
% make
cc -c rmstar.c
rmstar.c:32: syntax error
...
```

The user points the typing cursor somewhere in the string `rmstar.c:32:` and executes the `plumb` menu entry. This causes the terminal emulator to format a plumbing message containing the entire string surrounding the cursor, `rmstar:32:`, and to

write it to `/mnt/plumb/send`. The plumber receives this message and compares it sequentially to the various patterns in its configuration. Eventually, it will find one that breaks the string into pieces, `rmstar.c`, a colon, `32`, and the final colon. Other associated patterns verify that `rmstar.c` is a file in the current directory of the program generating the message, and that `32` looks like a line number within it. The plumber rewrites the message, setting the data to the string `rmstar.c` and attaching an indication that `32` is a line number to display. Finally, it sends the resulting message to the `edit` port. The text editor picks up the message, opens `rmstar.c` (if it's not already open) and highlights line 32, the location of the syntax error.

From the user's point of view, this process is simple: the error message appears, it is 'plumbed', and the editor jumps to the problem.

Of course, there are many different ways to cause compiler messages to pop up the source of an error, but the design of the plumber addresses more general issues than the specific goal of shortening the compile/debug/edit cycle. It facilitates the general exchange of data among programs, interactive or otherwise, throughout the environment, and its architecture—a central, language-driven file server—although unusual, has distinct advantages. It makes plumbing easy to add to an existing, Unix-like command environment; it guarantees uniform handling of inter-application messages; it off-loads from those applications most of the work of extracting and dispatching messages; and it works transparently and effortlessly across a network.

This paper is organized bottom-up, beginning with the format of the messages and proceeding through the plumbing language, the handling of messages, and the interactive user interface. The last sections discuss the implications of the design and compare the plumbing system to other environments that provide similar services.

Format of messages

Since the language that controls the plumber is defined in terms of the contents of plumbing messages, we begin by describing their layout.

Plumbing messages have a fixed-format textual header followed by a free-format data section. The header consists of six lines of text, in set order, each specifying a property of the message. Any line may be blank except the last, which is the length of the data portion of the message, as a decimal string. The lines are, in order:

The source application, the name of the program generating the message.

The destination port, the name of the port to which the messages should be sent.

The working directory in which the message was generated.

The type of the data, analogous to a MIME type, such as `text` or `image/gif`.

Attributes of the message, given as blank-separated *name*=*value* pairs. The values may be quoted to protect blanks or quotes; values may not contain newlines.

The length of the data section, in bytes.

Here is a sample message, one that (conventionally) tells the editor to open the file `/usr/rob/src/mem.c` and display line 27 within it:

```
plumbtest
edit
/usr/rob/src
text
addr=27
5
mem.c
```

Because in general it need not be text, the data section of the message has no terminating newline.

A library interface simplifies the processing of messages by translating them to and from a data structure, `Plumbmsg`, defined like this:

```
typedef struct Plumbattr Plumbattr;
typedef struct Plumbmsg  Plumbmsg;

struct Plumbmsg
{
   char       *src;    /* source application */
   char       *dst;    /* destination port */
   char       *wdir;   /* working directory */
   char       *type;   /* type of data */
   Plumbattr  *attr;   /* attribute list */
   int        ndata;   /* #bytes of data */
   char       *data;
};

struct Plumbattr
{
   char       *name;
   char       *value;
   Plumbattr  *next;
};
```

The library also includes routines to send a message, receive a message, manipulate the attribute list, and so on.

The Language

An instance of the plumber runs for each user on each terminal or workstation. It begins by reading its rules from the file `lib/plumbing` in the user's home directory, which in turn may use `include` statements to interpolate macro definitions and rules from standard plumbing rule libraries stored in `/sys/lib/plumb`.

The rules control the processing of messages. They are written in a pattern-action language comprising a sequence of blank-line-separated *rule sets*, each of which contains one or more *patterns* followed by one or more *actions*. Each incoming message is compared against the rule sets in order. If all the patterns within a rule set succeed, one of the associated actions is taken and processing completes.

The syntax of the language is straightforward. Each rule (pattern or action) has three components, separated by white space: an *object*, a *verb*, and optional *arguments*. The object identifies a part of the message, such as the source application (`src`), or the data portion of the message (`data`), or the rule's own arguments (`arg`); or it is the keyword `plumb`, which introduces an action. The verb specifies an operation to perform on the object, such as the word '`is`' to require precise equality between the object and the argument, or '`isdir`' to require that the object be the name of a directory.

For instance, this rule set sends messages containing the names of files ending in `.gif`, `.jpg`, etc. to a program, `page`, to display them; it is analogous to a Windows association rule:

```
# image files go to page
type is text
data matches '[a-zA-Z0-9_\-./]+'
data matches '([a-zA-Z0-9_\-./]+)\.(jpe?g|gif|bit|tiff|ppm)'
arg isfile $0
plumb to image
plumb client page -wi
```

(Lines beginning with # are commentary.) Consider how this rule handles the following message, annotated down the left column for clarity:

```
src     plumbtest
dst
wdir    /usr/rob/pics
type    text
attr
ndata   9
data    horse.gif
```

The `is` verb specifies a precise match, and the `type` field of the message is the string `text`, so the first pattern succeeds. The `matches` verb invokes a regular expression pattern match of the object (here `data`) against the argument pattern. Both `matches` patterns in this rule set will succeed, and in the process set the variables `$0` to the matched string, `$1` to the first parenthesized submatch, and so on (analogous to &, \1, etc. in ed's regular expressions). The pattern `arg isfile $0` verifies that the named file, `horse.gif`, is an actual file in the directory `/usr/rob/pics`. If all the patterns succeed, one of the actions will be executed.

There are two actions in this rule set. The `plumb to` rule specifies `image` as the destination port of the message. By convention, the plumber mounts its services in the directory `/mnt/plumb`, so in this case if the file `/mnt/plumb/image` has been opened, the message will be made available to the program reading from it. Note that the message does not name a port, but the rule set that matches the message does, and that is sufficient to dispatch the message. If on the other hand a message matches no rule but has an explicit port mentioned, that too is sufficient.

If no client has opened the `image` port, that is, if the program `page` is not already running, the `plumb client` action gives the execution script to start the application and send the message on its way; the `-wi` arguments tell `page` to create a window and to receive its initial arguments from the plumbing port. The process by which the plumber starts a program is described in more detail in the next section.

It may seem odd that there are two `matches` rules in this example. The reason is related to the way the plumber can use the rules themselves to refine the *data* in the message, somewhat in the manner of Structural Regular Expressions [Pike87a]. For example, consider what happens if the cursor is at the last character of

```
% make nightmare>horse.gif
```

and the user asks to plumb what the cursor is pointing at. The program creating the plumbing message—in this case the terminal emulator running the window—can send the entire white-space-delimited string `nightmare>horse.gif` or even the entire line, and the combination of `matches` rules can determine that the user was referring to the string `horse.gif`. The user could of course select the entire string `horse.gif`, but it's more convenient just to point in the general location and let the machine figure out what should be done. The process is as follows.

The application generating the message adds a special attribute to the message, named `click`, whose numerical value is the offset of the cursor—the selection point—within the data string. This attribute tells the plumber two things: first, that the regular expressions in `matches` rules should be used to identify the relevant data; and second, approximately where the relevant data lies. The plumber will then use the first `matches` pattern to identify the longest leftmost match that touches the cursor, which will extract the string `horse.gif`, and the second pattern will then verify that that names a picture file. The rule set succeeds and the data is winnowed to the matching substring before being sent to its destination.

Each `matches` pattern within a given rule set must match the same portion of the string, which guarantees that the rule set fails to match a string for which the second pattern matches only a portion. For instance, our example rule set should not execute if the data is the string `horse.gift`, and although the first pattern will match

`horse.gift`, the second will match only `horse.gif` and the rule set will fail.

The same approach of multiple `matches` rules can be used to exclude, for instance, a terminal period from a file name or URL, so a file name or URL at the end of a sentence is recognized properly.

If a `click` attribute is not specified, all patterns must match the entire string, so the user has an option: he or she may select exactly what data to send, or may instead indicate where the data is by clicking the selection button on the mouse and letting the machine locate the URL or image file name within the text. In other words, the user can control the contents of the message precisely when required, but the default, simplest action in the user interface does the right thing most of the time.

How Messages are Handled in the Plumber

An application creates a message header, fills in whatever fields it wishes to define, attaches the data, and writes the result to the file `send` in the plumber's service directory, `/mnt/plumb`. The plumber receives the message and applies the plumbing rules successively to it. When a rule set matches, the message is dispatched as indicated by that rule set and processing continues with the next message. If no rule set matches the message, the plumber indicates this by returning a write error to the application, that is, the write to `/mnt/plumb/send` fails, with the resulting error string describing the failure. (Plan 9 uses strings rather than pre-defined numbers to describe error conditions.) Thus a program can discover whether a plumbing message has been sent successfully.

After a matching rule set has been identified, the plumber applies a series of rewriting steps to the message. Some rewritings are defined by the rule set; others are implicit. For example, if the message does not specify a destination port, the outgoing message will be rewritten to identify it. If the message does specify the port, the rule set will only match if any `plumb to` action in the rule set names the same port. (If it matches no rule sets, but mentions a port, it will be sent there unmodified.)

The rule set may contain actions that explicitly rewrite components of the message. These may modify the attribute list or replace the data section of the message. Here is a sample rule set that does both. It matches strings of the form `plumb.h` or `plumb.h:27`. If that string identifies a file in the standard C include directory, `/sys/include`, perhaps with an optional line number, the outgoing message is rewritten to contain the full path name and an attribute, `addr`, to hold the line number:

```
# .h files are looked up in /sys/include and passed to edit
type is text
data matches '([a-zA-Z0-9]+\.h)(:([0-9]+))?'
arg isfile /sys/include/$1
data set /sys/include/$1
attr add addr=$3
plumb to edit
```

The `data set` rule replaces the contents of the data, and the `attr add` rule adds a new attribute to the message. The intent of this rule is to permit one to plumb an include file name in a C program to trigger the opening of that file, perhaps at a specified line, in the text editor. A variant of this rule, discussed below, tells the editor how to interpret syntax errors from the compiler, or the output of `grep -n`, both of which use a fixed syntax *file*`:`*line* to identify a line of source.

The Plan 9 text editors interpret the `addr` attribute as the definition of which portion of the file to display. In fact, the real rule includes a richer definition of the address syntax, so one may plumb strings such as `plumb.h:/plumbsend` (using a regular expression after the `/`) to pop up the declaration of a function in a C header file.

Another form of rewriting is that the plumber may modify the attribute list of the

message to clarify how to handle the message. The primary example of this involves the treatment of the `click` attribute, described in the previous section. If the message contains a `click` attribute and the matching rule set uses it to extract the matching substring from the data, the plumber deletes the `click` attribute and replaces the data with the matching substring.

Once the message is rewritten, the actions of the matching rule set are examined. If the rule set contains a `plumb to` action and the corresponding port is open—that is, if a program is already reading from that port—the message is delivered to the port. The application will receive the message and handle it as it sees fit. If the port is not open, a `plumb start` or `plumb client` action will start a new program to handle the message.

The `plumb start` action is the simpler: its argument specifies a command to run instead of passing on the message; the message is discarded. Here for instance is a rule that, given the process id (pid) of an existing process, starts the `acid` debugger [Wint94] in a new window to examine that process:

```
# processes go to acid (assuming strlen(pid) >= 2)
type is text
data matches '[a-zA-Z0-9.:_\-/]+'
data matches '[0-9][0-9]+'
arg isdir /proc/$0
plumb start window acid $0
```

(Note the use of multiple `matches` rules to avoid misfires from strings like `party.1999`.) The `arg isdir` rule checks that the pid represents a running process (or broken one; Plan 9 does not create `core` files but leaves broken processes around for debugging) by checking that the process file system has a directory for that pid [Kill84]. Using this rule, one may plumb the pid string printed by the `ps` command or by the operating system when the program breaks; the debugger will then start automatically.

The other startup action, `plumb client`, is used when a program will read messages from the plumbing port. For example, text editors can read files specified as command arguments, so one could use a `plumb start` rule to begin editing a file. If, however, the editor will read messages from the `edit` plumbing port, letting it read the message from the port insures that it uses other information in the message, such as the line number to display. The `plumb client` action is therefore like `plumb start`, but keeps the message around for delivery when the application opens the port. Here is the full rule set to pass a regular file to the text editor:

```
# existing files, possibly tagged by address, go to editor
type is text
data matches '([.a-zA-Z0-9_/\-]*[a-zA-Z0-9_/\-])('$addr')?'
arg isfile $1
data set $1
attr add addr=$3
plumb to edit
plumb client window $editor
```

If the editor is already running, the `plumb to` rule causes it to receive the message on the port. If not, the command '`window $editor`' will create a new window (using the Plan 9 program `window`) to run the editor, and once that starts it will open the `edit` plumbing port as usual and discover this first message already waiting.

The variables `$editor` and `$addr` in this rule set are macros defined in the plumbing rules file; they specify the name of the user's favorite text editor and a regular expression that matches that editor's address syntax, such as line numbers and patterns. This rule set lives in a library of shared plumbing rules that users' private rules can build on,

so the rule set needs to be adaptable to different editors and their address syntax. The macro definitions for Acme and Sam [Pike94,Pike87b] look like this:

```
editor=acme
# or editor=sam
addrelem='((#?[0-9]+)|(/[A-Za-z0-9_\^]+/?)|[.$])'
addr=:($addrelem([,;+\-]$addrelem)*)
```

Finally, the application reads the message from the appropriate port, such as `/mnt/plumb/edit`, unpacks it, and goes to work.

Message Delivery

In summary, a message is delivered by writing it to the `send` file and having the plumber, perhaps after some rewriting, send it to the destination port or start a new application to handle it. If no destination can be found by the plumber, the original write to the `send` file will fail, and the application will know the message could not be delivered.

If multiple applications are reading from the destination port, each will receive an identical copy of the message; that is, the plumber implements fan-out. The number of messages delivered is equal to the number of clients that have opened the destination port. The plumber queues the messages and makes sure that each application that opened the port before the message was written gets exactly one copy.

This design minimizes blocking in the sending applications, since the write to the `send` file can complete as soon as the message has been queued for the appropriate port. If the plumber waited for the message to be read by the recipient, the sender could block unnecessarily. Unfortunately, this design also means that there is no way for a sender to know when the message has been handled; in fact, there are cases when the message will not be delivered at all, such as if the recipient exits while there are still messages in the queue. Since the plumber is part of a user interface, and not an autonomous message delivery system, the decision was made to give the non-blocking property priority over reliability of message delivery. In practice, this tradeoff has worked out well: applications almost always know when a message has failed to be delivered (the `write` fails because no destination could be found), and those occasions when the sender believes incorrectly that the message has been delivered are both extremely rare and easily recognized by the user—usually because the recipient application has exited.

The Rules File

The plumber begins execution by reading the user's startup plumbing rules file, `lib/plumbing`. Since the plumber is implemented as a file server, it can also present its current rules as a dynamic file, a design that provides an easily understood way to maintain the rules.

The file `/mnt/plumb/rules` is the text of the rule set the plumber is currently using, and it may be edited like a regular file to update those rules. To clear the rules, truncate that file; to add a new rule set, append to it:

```
% echo 'type is text
data is self-destruct
plumb start rm -rf $HOME' >> /mnt/plumb/rules
```

This rule set will take effect immediately. If it has a syntax error, the write will fail with an error message from the plumber, such as 'malformed rule' or 'undefined verb'.

To restore the plumber to its startup configuration,

```
% cp /usr/$user/lib/plumbing /mnt/plumb/rules
```

For more sophisticated changes, one can of course use a regular text editor to modify

`/mnt/plumb/rules`.

This simple way of maintaining an active service could profitably be adopted by other systems. It avoids the need to reboot, to update registries with special tools, or to send asynchronous signals to critical programs.

The User Interface

One unusual property of the plumbing system is that the user interface that programs provide to access it can vary considerably, yet the result is nonetheless a unifying force in the environment. Shells talk to editors, image viewers, and web browsers; debuggers talk to editors; editors talk to themselves; and the window system talks to everybody.

The plumber grew out of some of the ideas of the Acme editor/window-system/user interface [Pike94], in particular its 'acquisition' feature. With a three-button mouse, clicking the right button in Acme on a piece of text tells Acme to get the thing being pointed to. If it is a file name, open the file; if it is a directory, open a viewer for its contents; if a line number, go to that line; if a regular expression, search for it. This one-click access to anything describable textually was very powerful but had several limitations, of which the most important were that Acme's rules for interpreting the text (that is, the implicit hyperlinks) were hard-wired and inflexible, and that they only applied to and within Acme itself. One could not, for example, use Acme's power to open an image file, since Acme is a text-only system.

The plumber addresses these limitations, even with Acme itself: Acme now uses the plumber to interpret the right button clicks for it. When the right button is clicked on some text, Acme constructs a plumbing message much as described above, using the `click` attribute and the white-space-delimited text surrounding the click. It then writes the message to the plumber; if the write succeeds, all is well. If not, it falls back to its original, internal rules, which will result in a context search for the word within the current document.

If the message is sent successfully, the recipient is likely to be Acme itself, of course: the request may be to open a file, for example. Thus Acme has turned the plumber into an external component of its own operation, while expanding the possibilities; the operation might be to start an image viewer to open a picture file, something Acme cannot do itself. The plumber expands the power of Acme's original user interface.

Traditional menu-driven programs such as the text editor Sam [Pike87b] and the default shell window of the window system 8½ [Pike91] cannot dedicate a mouse button solely to plumbing, but they can certainly dedicate a menu entry. The editing menu for such programs now contains an entry, `plumb`, that creates a plumbing message using the current selection. (Acme manages to send a message by clicking on the text with one button; other programs require a click with the select button and then a menu operation.) For example, after this happens in a shell window:

```
% make
cc -c shaney.c
shaney.c:232: i undefined
...
```

one can click anywhere on the string `shaney.c:232`, execute the `plumb` menu entry, and have line 232 appear in the text editor, be it Sam or Acme—whichever has the `edit` port open. (If this were an Acme shell window, it would be sufficient to right-click on the string.)

[An interesting side line is how the window system knows what directory the shell is running in; in other words, what value to place in the `wdir` field of the plumb message. Recall that 8½ is, like many Plan 9 programs, a file server. It now serves a new file, `/dev/wdir`, that is private to each window. Programs, in particular the Plan 9 shell, `rc`, can write that file to inform the window system of its current directory. When a `cd`

command is executed in an interactive shell, `rc` updates the contents of `/dev/wdir` and plumbing can proceed with local file names.]

Of course, users can plumb image file names, process ids, URLs, and other items—any string whose syntax and disposition are defined in the plumbing rules file. An example of how the pieces fit together is the way Plan 9 now handles mail, particularly MIME-encoded messages.

When a new mail message arrives, the mail receiver process sends a plumbing message to the `newmail` port, which notifies any interested process that new mail is here. The plumbing message contains information about the mail, including its sender, date, and current location in the file system. The interested processes include a program, `faces`, that gives a graphical display of the mail box using faces to represent the senders of messages [PiPr85], as well as interactive mail programs such as the Acme mail viewer [Pike94]. The user can then click on the face that appears, and the `faces` program will send another plumbing message, this time to the `showmail` port. Here is the rule for that port:

```
# faces -> new mail window for message
type is text
data matches '[a-zA-Z0-9_\-./]+'
data matches '/mail/fs/[a-zA-Z0-9/]+/[0-9]+'
plumb to showmail
plumb start window edmail -s $0
```

If a program, such as the Acme mail reader, is reading that port, it will open a new window in which to display the message. If not, the `plumb start` rule will create a new window and run `edmail`, a conventional mail reading process, to examine it. Notice how the plumbing connects the components of the interface together the same way regardless of which components are actually being used to view mail.

There is more to the mail story. Naturally, mail boxes in Plan 9 are treated as little file systems, which are synthesized on demand by a special-purpose file server that takes a flat mail box file and converts it into a set of directories, one per message, with component files containing the header, body, MIME information, and so on. Multi-part MIME messages are unpacked into multi-level directories, like this:

```
% ls -l /mail/fs/mbox/25
d-r-xr-xr-x M 20 rob rob     0 Nov 21 13:06 /mail/fs/mbox/25/1
d-r-xr-xr-x M 20 rob rob     0 Nov 21 13:06 /mail/fs/mbox/25/2
--r--r--r-- M 20 rob rob 28678 Nov 21 13:06 /mail/fs/mbox/25/body
--r--r--r-- M 20 rob rob     0 Nov 21 13:06 /mail/fs/mbox/25/cc
...
% mail
25 messages
: 25
From: presotto
Date: Sun Nov 21 13:05:51 EST 1999
To: rob

Check this out.

===> 2/ (image/jpeg) [inline]
        /mail/fs/mbox/25/2/fabio.jpg
:
```

Since the components are all (synthetic) files, the user can plumb the pieces to view embedded pictures, URLs, and so on. Note that the mail program can plumb the contents of `inline` attachments automatically, without user interaction; in other words, plumbing lets the mailer handle multimedia data without itself interpreting it.

At a more mundane level, a shell command, plumb, can be used to send messages:

```
% cd /usr/rob/src
% plumb mem.c
```

will send the appropriate message to the edit port. A surprising use of the plumb command is in actions within the plumbing rules file. In our lab, we commonly receive Microsoft Word documents by mail, but we do not run Microsoft operating systems on our machines so we cannot view them without at least rebooting. Therefore, when a Word document arrives in mail, we could plumb the .doc file but the text editor could not decode it. However, we have a program, doc2txt, that decodes the Word file format to extract and format the embedded text. The solution is to use plumb in a plumb start action to invoke doc2txt on .doc files and synthesize a plain text file:

```
# rule set for microsoft word documents
type is text
data matches '[a-zA-Z0-9_\-./]+'
data matches '([a-zA-Z0-9_\-./]+)\.doc'
arg isfile $0
plumb start doc2txt $data | \
    plumb -i -d edit -a action=showdata -a filename=$0
```

The arguments to plumb tell it to take standard input as its data rather than the text of the arguments (-i), define the destination port (-d edit), and set a conventional attribute so the editor knows to show the message data itself rather than interpret it as a file name (-a action=showdata) and provide the original file name (-a filename=$0). Now when a user plumbs a .doc file the plumbing rules run a process to extract the text and send it as a temporary file to the editor for viewing. It's imperfect, but it's easy and it beats rebooting.

Another simple example is a rule that turns man pages into hypertext. Manual page entries of the form plumber(1) can be clicked on to pop up a window containing the formatted 'man page'. That man page will in turn contain more such citations, which will also be clickable. The rule is a little like that for Word documents:

```
# man index entries are synthesized
type is text
data matches '([a-zA-Z0-9_\-./]+)\(([0-9])\)'
plumb start man $2 $1 | \
    plumb -i -d edit -a action=showdata -a filename=/man/$1($2)
```

There are many other inventive uses of plumbing. One more should give some of the flavor. We have a shell script, src, that takes as argument the name of an executable binary file. It examines the symbol table of the binary to find the source file from which it was compiled. Since the Plan 9 compilers place full source path names in the symbol table, src can discover the complete file name. That is then passed to plumb, complete with the line number to find the symbol main. For example,

```
% src plumb
```

is all it takes to pop up an editor window on the main routine of the plumb command, beginning at line 39 of /sys/src/cmd/plumb/plumb.c. Like most uses of plumbing, this is not a breakthrough in functionality, but it is a great convenience.

Why This Architecture?

The design of the plumbing system is peculiar: a centralized language-based file server does most of the work, while compared to other systems the applications themselves contribute relatively little. This architecture is deliberate, of course.

That the plumber's behavior is derived from a linguistic description gives the system

great flexibility and dynamism—rules can be added and changed at will, without rebooting—but the existence of a central library of rules ensures that, for most users, the environment behaves in well-established ways.

That the plumber is a file server is perhaps the most unusual aspect of its design, but is also one of the most important. Messages are passed by regular I/O operations on files, so no extra technology such as remote procedure call or request brokers needs to be provided; messages are transmitted by familiar means. Almost every service in Plan 9 is a file server, so services can be exported trivially using the system's remote file system operations [Pike93]. The plumber is no exception; plumbing messages pass routinely across the network to remote applications without any special provision, in contrast to some commercial IPC mechanisms that become significantly more complex when they involve multiple machines. As I write this, my window system is talking to applications running on three different machines, but they all share a single instance of the plumber and so can interoperate to integrate my environment. Plan 9 uses a shared file name space to combine multiple networked machines—compute servers, file servers, and interactive workstations—into a single computing environment; plumbing's design as a file server is a natural by-product of, and contributor to, the overall system architecture [Pike92].

The centrality of the plumber is also unusual. Other systems tend to let the applications determine where messages will go; consider mail readers that recognize and highlight URLs in the messages. Why should just the mail readers do this, and why should they just do it for URLs? (Acme was guilty of similar crimes.) The plumber, by removing such decisions to a central authority, guarantees that all applications behave the same and simultaneously frees them all from figuring out what's important. The ability for the plumber to excerpt useful data from within a message is critical to the success of this model.

The entire system is remarkably small. The plumber itself is only about two thousand lines of C code. Most applications work fine in a plumbing environment without knowing about it at all; some need trivial changes such as to standardize their error output; a few need to generate and receive plumbing messages. But even to add the ability to send and receive messages in a program such as text editor is short work, involving typically a few dozen lines of code. Plumbing fits well into the existing environment.

But plumbing is new and it hasn't been pushed far enough yet. Most of the work so far has been with textual messages, although the underlying system is capable of handling general data. We plan to reimplement some of the existing data movement operations, such as cut and paste or drag and drop, to use plumbing as their exchange mechanism. Since the plumber is a central message handler, it is an obvious place to store the 'clipboard'. The clipboard could be built as a special port that holds onto messages rather than deleting them after delivery. Since the clipboard would then be holding a plumbing message rather than plain text, as in the current Plan 9 environment, it would become possible to cut and paste arbitrary data without providing new mechanism. In effect, we would be providing a new user interface to the existing plumbing facilities.

Another possible extension is the ability to override plumbing operations interactively. Originally, the plan was to provide a mechanism, perhaps a pop-up menu, that one could use to direct messages, for example to send a PostScript file to the editor rather than the PostScript viewer by naming an explicit destination in the message. Although this deficiency should one day be addressed, it should be done without complicating the interface for invoking the default behavior. Meanwhile, in practice the default behavior seems to work very well in practice—as it must if plumbing is to be successful—so the lack of overrides is not keenly felt.

Comparison with Other Systems

The ideas of the plumbing system grew from an attempt to generalize the way Acme acquires files and data. Systems further from that lineage also share some properties with plumbing. Most, however, require explicit linking or message passing rather than plumbing's implicit, context-based pattern matching, and none has the plumber's design of a language-based file server.

Reiss's FIELD system [Reis95] probably comes the closest to providing the facilities of the plumber. It has a central message-passing mechanism that connects applications together through a combination of a library and a pattern-matching central message dispatcher that handles message send and reply. The main differences between FIELD's message dispatcher and the plumber are first that the plumber is based on a special-purpose language while the FIELD system uses an object-oriented library, second that the plumber has no concept of a reply to a message, and finally that the FIELD system has no concept of port. But the key distinction is probably in the level of use. In FIELD, the message dispatcher is a critical integrating force of the underlying programming environment, handling everything from debugging events to changing the working directory of a program. Plumbing, by contrast, is intended primarily for integrating the user interface of existing tools; it is more modest and very much simpler. The central advantage of the plumber is its convenience and dynamism; the FIELD system does not share the ease with which message dispatch rules can be added or modified.

The inspiration for Acme was the user interface to the object-oriented Oberon system [WiGu92]. Oberon's user interface interprets mouse clicks on strings such as `Obj.meth` to invoke calls to the method `meth` of the object `Obj`. This was the starting point for Acme's middle-button execution [Pike94], but nothing in Oberon is much like Acme's right-button 'acquisition', which was the starting point for the plumber. Oberon's implicit method-based linking is not nearly as general as the pattern-matched linking of the plumber, nor does its style of user-triggered method call correspond well to the more general idea of inter-application communication of plumbing messages.

Microsoft's OLE interface is another relative. It allows one application to *embed* its own data within another's, for example to place an Excel spreadsheet within a Frame document; when Frame needs to format the page, it will start Excel itself, or at least some of its DLLs, to format the spreadsheet. OLE data can only be understood by the application that created it; plumbing messages, by contrast, contain arbitrary data with a rigidly formatted header that will be interpreted by the pattern matcher and the destination application. The plumber's simplified message format may limit its flexibility but makes messages easy and efficient to dispatch and to interpret. At least for the cut-and-paste style of exchange OLE encourages, plumbing gives up some power in return for simplicity, while avoiding the need to invoke a vestigial program (if Excel can be called a vestige) every time the pasted data is examined. Plumbing is also better suited to other styles of data exchange, such as connecting compiler errors to the text editor.

The Hyperbole [Wein] package for Emacs adds hypertext facilities to existing documents. It includes explicit links and, like plumbing, a rule-driven way to form implicit links. Since Emacs is purely textual, like Acme, Hyperbole does not easily extend to driving graphical applications, nor does it provide a general interprocess communication method. For instance, although Hyperbole provides some integration for mail applications, it cannot provide the glue that allows a click on a face icon in an external program to open a mail message within the viewer. Moreover, since it is not implemented as a file server, Hyperbole does not share the advantages of that architecture.

Henry's `error` program in 4BSD echoes a small but common use of plumbing. It takes the error messages produced by a compiler and drives a text editor through the steps of looking at each one in turn; the notion is to quicken the compile/edit/debug cycle. Similar results are achieved in EMACS by writing special M-LISP macros to parse the error messages from various compilers. Although for this particular purpose they may be

more convenient than plumbing, these are specific solutions to a specific problem and lack plumbing's generality.

Of course, the resource forks in MacOS and the association rules for file name extensions in Windows also provide some of the functionality of the plumber, although again without the generality or dynamic nature.

Closer to home, Ousterhout's Tcl (Tool Command Language) [Oust90] was originally designed to embed a little command interpreter in each application to control interprocess communication and provide a level of integration. Plumbing, on the other hand, provides minimal support within the application, offloading most of the message handling and all the command execution to the central plumber.

The most obvious relative to plumbing is perhaps the hypertext links of a web browser. Plumbing differs by synthesizing the links on demand. Rather than constructing links within a document as in HTML, plumbing uses the context of a button click to derive what it should link to. That the rules for this decision can be modified dynamically gives it a more fluid feel than a standard web browsing world. One possibility for future work is to adapt a web browser to use plumbing as its link-following engine, much as Acme used plumbing to offload its acquisition rules. This would connect the web browser to the existing tools, rather than the current trend in most systems of replacing the tools by a browser.

Each of these prior systems—and there are others, e.g. [Pasa93, Free93]—addresses a particular need or subset of the issues of system integration. Plumbing differs because its particular choices were different. It focuses on two key issues: centralizing and automating the handling of interprocess communication among interactive programs, and maximizing the convenience (or minimizing the trouble) for the human user of its services. Moreover, the plumber's implementation as a file server, with messages passed over files it controls, permits the architecture to work transparently across a network. None of the other systems discussed here integrates distributed systems as smoothly as local ones without the addition of significant extra technology.

Discussion

There were a few surprises during the development of plumbing. The first version of plumbing was done for the Inferno system [Dorw97a,Dorw97b], using its file-to-channel mechanism to mediate the IPC. Although it was very simple to build, it encountered difficulties because the plumber was too disconnected from its clients; in particular, there was no way to discover whether a port was in use. When plumbing was implemented afresh for Plan 9, it was provided through a true file server. Although this was much more work, it paid off handsomely. The plumber now knows whether a port is open, which makes it easy to decide whether a new program must be started to handle a message, and the ability to edit the rules file dynamically is a major advantage. Other advantages arise from the file-server design, such as the ease of exporting plumbing ports across the network to remote machines and the implicit security model a file-based interface provides: no one has permission to open my private plumbing files.

On the other hand, Inferno was an all-new environment and the user interface for plumbing was able to be made uniform for all applications. This was impractical for Plan 9, so more *ad hoc* interfaces had to be provided for that environment. Yet even in Plan 9 the advantages of efficient, convenient, dynamic interprocess communication outweigh the variability of the user interface. In fact, it is perhaps a telling point that the system works well for a variety of interfaces; the provision of a central, convenient message-passing service is a good idea regardless of how the programs use it.

Plumbing's rule language uses only regular expressions and a few special rules such as `isfile` for matching text. There is much more that could be done. For example, in the current system a JPEG file can be recognized by a `.jpg` suffix but not by its

contents, since the plumbing language has no facility for examining the *contents* of files named in its messages. To address this issue without adding more special rules requires rethinking the language itself. Although the current system seems a good balance of complexity and functionality, perhaps a richer, more general-purpose language would permit more exotic applications of the plumbing model.

In conclusion, plumbing adds an effective, easy-to-use inter-application communication mechanism to the Plan 9 user interface. Its unusual design as a language-driven file server makes it easy to add context-dependent, dynamically interpreted, general-purpose hyperlinks to the desktop, for both existing tools and new ones.

Acknowledgements

Dave Presotto wrote the mail file system and `edmail`. He, Russ Cox, Sape Mullender, and Cliff Young influenced the design, offered useful suggestions, and suffered early versions of the software. They also made helpful comments on this paper, as did Dennis Ritchie and Brian Kernighan.

References

[Dorw97a] Sean Dorward, Rob Pike, David Leo Presotto, Dennis M. Ritchie, Howard W. Trickey, and Philip Winterbottom, ''Inferno'', *Proceedings of the IEEE Compcon 97 Conference*, San Jose, 1997, pp. 241-244.

[Dorw97b] Sean Dorward, Rob Pike, David Leo Presotto, Dennis M. Ritchie, Howard W. Trickey, and Philip Winterbottom, ''The Inferno Operating System'', *Bell Labs Technical Journal*, **2**, 1, Winter, 1997.

[Free93] FreeBSD, Syslog configuration file manual *syslog.conf*(0).

[Kill84] T. J. Killian, ''Processes as Files'', *Proceedings of the Summer 1984 USENIX Conference*, Salt Lake City, 1984, pp. 203-207.

[Oust90] John K. Ousterhout, ''Tcl: An Embeddable Command Languages'', *Proceedings of the Winter 1990 USENIX Conference*, Washington, 1990, pp. 133-146.

[Pasa93] Vern Paxson and Chris Saltmarsh, "Glish: A User-Level Software Bus for Loosely-Coupled Distributed Systems" , *Proceedings of the Winter 1993 USENIX Conference*, San Diego, 1993, pp. 141-155.

[Pike87a] Rob Pike, ''Structural Regular Expressions'', *EUUG Spring 1987 Conference Proceedings*, Helsinki, May 1987, pp. 21-28.

[Pike87b] Rob Pike, ''The Text Editor sam'', *Software - Practice and Experience*, **17**, 5, Nov. 1987, pp. 813-845.

[Pike91] Rob Pike, ''8½, the Plan 9 Window System'', *Proceedings of the Summer 1991 USENIX Conference*, Nashville, 1991, pp. 257-265.

[Pike93] Rob Pike, Dave Presotto, Ken Thompson, Howard Trickey, and Phil Winterbottom, ''The Use of Name Spaces in Plan 9'', *Operating Systems Review*, **27**, 2, April 1993, pp. 72-76.

[Pike94] Rob Pike, ''Acme: A User Interface for Programmers'', *Proceedings of the Winter 1994 USENIX Conference*, San Francisco, 1994, pp. 223-234.

[PiPr85] Rob Pike and Dave Presotto, ''Face the Nation'', *Proceedings of the USENIX Summer 1985 Conference*, Portland, 1985, pg. 81.

[Reis95] Steven P. Reiss, *The FIELD Programming Environment: A Friendly Integrated Environment for Learning and Development*, Kluwer, Boston, 1995.

[Wein] Bob Weiner, *Hyperbole User Manual*, `http://www.cs.indiana.edu/elisp/hyperbole/hyperbole_1.html`

[Wint94] Philip Winterbottom, ''ACID: A Debugger based on a Language'', *Proceedings of*

the USENIX Winter Conference, San Francisco, CA, 1994.

[WiGu92] Niklaus Wirth and Jurg Gutknecht, *Project Oberon: The Design of an Operating System and Compilers*, Addison-Wesley, Reading, 1992.