# 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.)

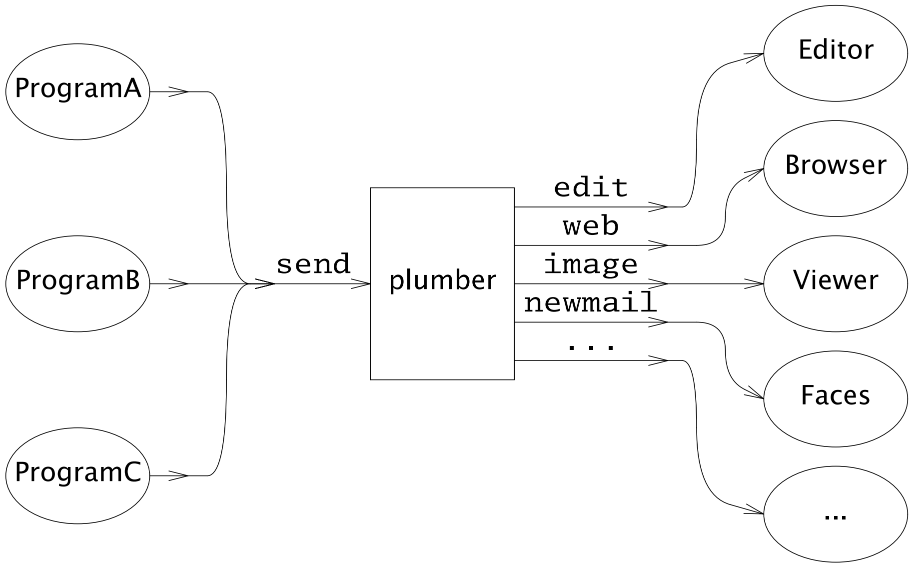


Figure 1. The plumber controls the flow of messages between applications. Programs write to the file `send` and receive on 'ports' of various names representing services such as `edit` or `web`. Although the figure doesn't illustrate it, some programs may both send and receive messages, and some ports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