

Plumbing and Other Utilities

Rob Pike

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Overview

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 too much data flows from program to program because a human carries it there.

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.

The desideratum is a system that, given a piece of data, hands it to the appropriate application by default with little or no human intervention, without limiting the user’s options for more general behavior.

The plumbing system is an attempt to address some of these issues in a single, coherent, central way. It provides a mechanism for 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*. The plumber is implemented as a file server; programs send messages by writing them to the file `/mnt/plumb/send`, and receive messages by reading them from *ports*, which are other files in `/mnt/plumb`. For example, `/mnt/plumb/edit` is by convention the file from which the text editor reads messages requesting it to open and display a file for editing. The plumber takes messages from the `send` file and interprets their contents using 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 plumber itself is a language-driven file server that mediates interprocess communication as part of the user interface of the operating system. Interactive programs then 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 hyper-text system in which all the links are implicit, extracted automatically by examining the data and the user’s actions. It obviates the need for, without replacing, much cut and paste and other such hand-driven interprocess communication mechanisms. Like its namesake, plumbing delivers the goods to the right place automatically.

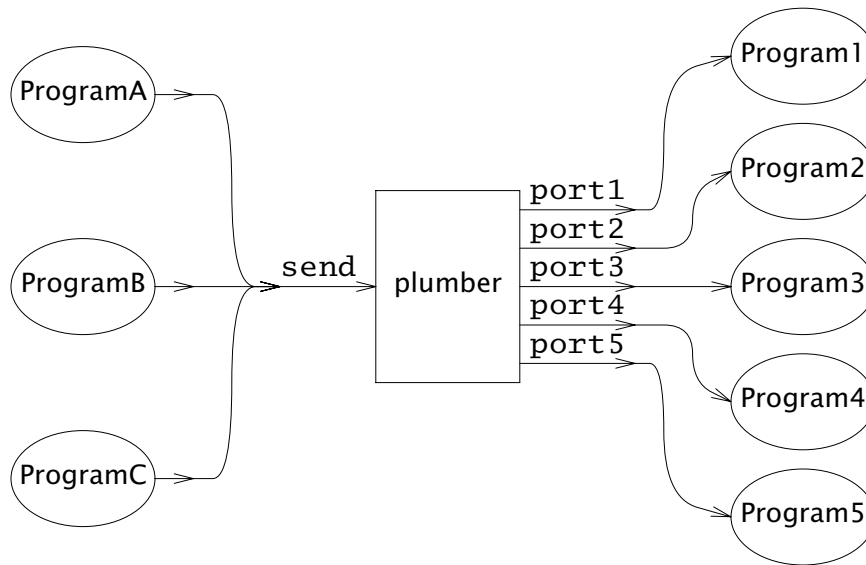


Figure 1. The plumber controls the flow of messages between applications. Programs write to the file `send` and receive on ‘ports’ of various names representing services such as `edit` or `web`. Although the figure doesn’t illustrate it, some programs may both send and receive messages.

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 section compares the plumbing system to other environments that provide similar services.

Format of messages

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`. For historical reasons, this is called the *kind* of the message.
- 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 message, 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
```

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      *kind;     /* kind 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 argument field of the pattern itself (`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 an extant directory.

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

```
# image files go to page
kind is text
data matches '[a-zA-Z0-9_\-./]+'
```

data matches '([a-zA-Z0-9_\-./]+)\.(jpg|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:

```
plumbtest

/usr/rob/pics
text

9
horse.gif
```

The `is` verb specifies a precise match, and the `kind` 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 `&` and `\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 no client has opened the `image` port, that is, if `page` is not already running, the `plumb client` action gives the execution script to start the application and send the message on its way. This process is described in the next section.

It may seem odd that there are two `matches` rules in this example. The combination guarantees that the rule fails to match a string for which the second pattern matches only a portion. For example, this rule set should not execute if the data is the string `horse.gift`. The way `matches` is defined, each `matches` pattern within a given rule set must match the identical string; in this example, the first pattern will match `horse.gift` while the second will match only `horse.gif` and the rule set will fail.

One could achieve this result using the `^` and `$` regular expression metacharacters, of course, but the multiple-pattern technique permits another, unusual use of regular expressions in the rules: they permit the expressions to *extract* valid substrings of the data, somewhat like structural regular expressions [XXX]. Imagine that a user has pointed at a string and asked to plumb it. The program generating the plumbing message may not know exactly what is being pointed at, but the plumbing rules can extract the information.

For example, consider what happens if the cursor is at the end of the command line

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

and the user asks to plumb what the cursor is pointing at. The program can send an arbitrary block of text containing the cursor, perhaps the white-space-delimited component `nightmare>horse.gif`, and set the special attribute `click` to say that the cursor was pointing to the null string at the end of the line. The plumber will use the

first matches pattern to identify the longest leftmost match that touches the cursor, and the second pattern will then verify that it is a picture file. (If a `click` attribute is not specified, all patterns must match the entire string.) The same approach 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.

The goal is to let the machine to do the work, to free the user from the need to do the fiddly selection of the exact string to plumb. In practice, programs that send plumbing messages generated by user actions like this will send the white-space-delimited string containing the cursor and set the `click` attribute. If the selection is non-null, `click` will not be set and the selected text is sent as the data segment of the message. This behavior allows the user to control the contents of the message precisely when required, while still guaranteeing that the common case is the simplest one to achieve in the user interface.

The Odyssey of a Message

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 original 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 will be 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.

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
kind 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 is the command to run upon receipt of the message, which is then discarded by the plumber. Here for instance is a rule that, given the process id (pid) of an existing process, starts the acid debugger [XXX] in a new window to examine that process:

```
# processes go to acid (strlen(pid) >= 2)
kind 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 [XXX]. 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 once it's started. 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 is 'plumbed'—if it reads 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. Here is the full rule set to pass a regular file to the text editor:

```
# existing files, possibly tagged by line number, go to editor
kind 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, the editor I use [XXX], look like this:

```
editor=acme
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.

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 'kind 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 the rules.

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

The user interface

One unusual property of the plumbing system is that the user interface 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 [XXX], 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 1) that Acme's rules for interpreting the text (that is, the implicit hyperlinks) were hard-wired; 2) that they were inflexible; and 3) 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 was a text-only system.

The plumber addresses all three of these limitations, even with Acme itself. Acme now uses the plumber to interpret the right button clicks for it; rather than using its own rules to determine what is being pointed at and how to interpret it, Acme now calls on the plumber. 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 it. 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 [XXX] and the default shell window of the window system 8½ [XXX] 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. (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—a 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. Interested processes include a program, `faces`, that gives a graphical display of the mail box using faces to represent the senders of messages [XXX], as well as interactive mail programs such as the Acme mail viewer [XXX]. The user can then click on (say) 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
kind      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 /mail/fs/mbox/1
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) [file]
      cp /mail/fs/mbox/25/2/body.jpg /usr/rob/fabio.jpg
:
```

(The `cp` command is printed by the mail program.) Since the components are all (synthetic) files, the user can plumb the pieces to view embedded pictures, URLs, and so on.

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 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
kind 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
```

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`). 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.

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.

Comparison with Other Systems

As described above, the ideas of the plumbing system grew from an attempt to generalize the way Acme acquires files and data. Systems further from that lineage share some properties with plumbing, however.

Reiss's Field system [XXX] 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 sending 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 difference is probably in the level of use. In Field, the message dispatcher is a key 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.

Henry's `error` program in 4BSD echoes a small but common use of plumbing. It takes the error message 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. Although for this particular purpose it may be more convenient than plumbing, it is a specific solution to a specific problem and lacks plumbing's generality.

Of course, the resource forks in MacOS and the associations 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) 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.

Most obvious, perhaps, is 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 system of replacing the tools by a browser.

Each of these prior systems 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.

Along the way, there were a few surprises. The first version of plumbing was done for

the Inferno system [XXX], 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.

On the other hand, Inferno was a new system and the user interface for plumbing was able to be made uniform for all applications. This was impractical for Plan 9, 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.

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. rsc sape