The Styx Architecture for Distributed Systems

Rob Pike
Dennis M. Ritchie

Computing Science Research Center
Lucent Technologies, Bell Labs
Murray Hill, New Jersey
USA

ABSTRACT

A distributed system is constructed from a set of relatively independent components that form a unified, but geographically and functionally diverse entity. Examples include networked operating systems, Internet services, the national telephone switching system, and in general all the technology using todays diverse digital networks. Nevertheless, distributed systems remain difficult to design, build, and maintain, primarily because of the lack of a clean, perspicuous interconnection model for the components.

Our experience with two distributed operating systems, Plan 9 and Inferno, encourages us to propose such a model. These systems depend on, advocate, and generally push to the limit a fruitful idea: to present their resources as files in a hierarchical name space. The objects appearing as files may represent stored data, but may also be devices, dynamic information sources, interfaces to services, and control points. The approach unifies and provides basic naming, structuring, and access control mechanisms for all system resources. A simple underlying network protocol, Styx, forms the core of the architecture by presenting a common language for communication within the system.

Even within non-distributed systems, the presentation of services as files advantageously extends a familiar scheme for naming, classifying, and connecting to system resources. More important, the approach provides a natural way to build distributed systems, by using well-known technology for attaching remote file systems. If resources are represented as files, and there are remote file systems, one has a distributed system: resources available in one place are usable from another.

Introduction

The Styx protocol is a variant of a protocol called *9P* that was developed for the Plan 9 operating system [9man]. For simplicity, we will use the name Styx throughout this paper; the difference concerns only the initialization of a connection.

The original idea behind Styx was to encode file operations between client programs and the file system, to be translated into messages for transmission on a computer network. Using this technology, Plan 9 separates the file server a central repository for permanent file storage both from the CPU server a large shared-memory multiprocessor and from the user terminals. This physical separation of function was central to the original

Originally appeared in *Bell Labs Technical Journal*, Vol. 4, No. 2, April-June 1999, pp. 146-152.

design of the system; what was unexpected was how well the model could be used to solve a wide variety of problems not usually thought of as file system issues.

The breakthrough was to realize that by representing a computing resource as a form of file system, many of the difficulties of making that resource available across the network would disappear naturally, because Styx could export the resource transparently. For example, the Plan 9 window system, 8½ [Pike91], is implemented as a dynamic file server that publishes files with names like /dev/mouse and /dev/screen to provide access to the local hardware. The /dev/mouse file, for instance, may be opened and read like a regular file, in the manner of UNIX™ device files, but under 8½ it is multiplexed: each client program has a private /dev/mouse file that returns mouse events only when the client s window is the active one on the display. This design provides a clean, simple mechanism for controlling access to the mouse. Its real strength, though, is that the representation of the window system s resources as files allows Styx to make those resources available across the network. For example, an interactive graphics program may be run on a CPU server simply by having 8½ serve the appropriate files to that machine.

Note that although the resources published by Styx behave like files they have file names, file permissions, and file access methods they do not need to exist as standard files on disk. The /dev/mouse file is accessed by standard file I/O mechanisms but is nonetheless a transient object fabricated dynamically by a running program; it has no permanent existence.

By following this approach throughout the system, Plan 9 achieves a remarkable degree of transparency in the distribution of resources [PPTTW93]. Besides interactive graphics, services such as debugging, maintenance, file backup, and even access to the underlying network hardware can be made available across the network using Styx, permitting the construction of distributed applications and services using nothing more sophisticated than file I/O.

The Styx protocol

Styxs place in the world is analogous to Sun NFS [RFC] [NFS] or Microsoft CIFS [CIFS], although it is simpler and easier to implement [Welc94]. Furthermore, NFS and CIFS are designed for sharing regular disk files; NFS in particular is intimately tied to the implementation and caching strategy of the underlying UNIX file system. Unlike Styx, NFS and CIFS are clumsier at exporting dynamic device-like files such as /dev/mouse.

Styx provides a view of a hierarchical, tree-shaped file system name space [Nee89], together with access information about the files (permissions, sizes, dates) and the means to read and write the files Its users (that is, the people who write application programs), don t see the protocol itself; instead they see files that they read and write, and that provide information or change information.

In use, a Styx *client* is an entity on one machine that establishes communication with another entity, the *server*, on the same or another machine. The client mechanisms may be built into the operating system, as they are in Plan 9 or Inferno [INF1] [INF2], or into application libraries; a server may be part of the operating system, or just as often may be application code on a separate server machine. In any case, the client and server entities communicate by exchanging messages, and the effect is that the client sees a hierarchical file system that exists on the server. The Styx protocol is the specification of the messages that are exchanged.

At one level, Styx consists of messages of 13 types for

> Starting communication (attaching to a file system);
>
> Navigating the file system (that is, specifying and gaining a handle for a named file);

Name	Description
attach	Authenticate user of connection; return FID
clone	Duplicate FID
walk	Advance FID one level of name hierarchy
open	Check permissions for file I/O
create	Create new file
read	Read contents of file
write	Write contents of file
close	Discard FID
remove	Remove file
stat	Report file state: permissions, etc.
wstat	Modify file state
error	Return error condition for failed operation
flush	Disregard outstanding I/O requests

Table 1. Summary of Styx messages.

Reading and writing a file; and

Performing file status inquiries and changes

However, application writers simply code requests to open, read, or write files; a library or the operating system translates the requests into the necessary byte sequences transmitted over a communication channel. The Styx protocol proper specifies the interpretation of these byte sequences. It fits, approximately, at the OSI Session Layer level of the ISO standard classification. Its specification is independent of most details of machine architecture and it has been successfully used among machines of varying instruction sets and data layout. The protocol is summarized in Table 1.

In use, an operation such as

```
open("/usr/rob/.profile",       O_READ);
```

is translated by the underlying system into a sequence of Styx messages. After establishing the initial connection to the file server, an attach message authenticates the user (the person or agent accessing the files) and returns an object called a FID (file ID) that represents the root of the hierarchy on the server. When the open() operation is executed, it proceeds as follows.

A clone message duplicates the root FID, returning a new FID that can navigate the hierarchy without losing the connection to the root.

The new FID is then moved to the file /usr/rob/.profile by a sequence of walk messages that step along, one path component at a time (usr, rob, .profile).

Finally, an open message checks that the user has permission to read the file, permitting subsequent read and write operations (messages) on the FID.

Once I/O is completed, the close message will release the FID.

At a lower level, implementations of Styx depend only on a reliable, byte-stream Transport communications layer. For example, it runs over either TCP/IP, the standard transmission control protocol and Internet protocol, or Internet link (IL), which is a sequenced, reliable datagram protocol using IP packets. It is worth emphasizing, though, that the model does not require the existence of a network to join the components; Styx runs fine over a Unix pipe or even using shared memory. The strength of the approach is not so much how it works over a network as that its behavior over a network is identical to its behavior locally.

Architectural approach

Styx, as a file system protocol, is merely a component in a more encompassing approach to system design: the presentation of resources as files. This approach will be discussed using a sequence of examples.

Example: networking

As an example, access to a TCP/IP network in Inferno and Plan 9 systems appears as a piece of a file system, with (abbreviated) structure as follows [PrWi93]:

```
/net/
        dns/
        tcp/
                clone
                stats
                0/
                        ctl
                        status
                        data
                        listen
                1/
                        ...
                ...
        ether0/
                0/
                        ctl
                        status
                        ...
                1/
                        ...
        ...
```

This represents a file system structure in which one can name, read, and write files with names like `/net/dns`, `/net/tcp/clone`, `/net/tcp/0/ctl` and so on; there are directories of files `/net/tcp` and `/net/ether0`. On the machine that actually has the network interface, all of these things that look like files are constructed by the kernel drivers that maintain the TCP/IP stack; they are not real files on a disk. Operations on the files turn into operations sent to the device drivers.

Suppose an application wishes to establish a connection over TCP/IP to `www.bell-labs.com`. The first task is to translate the domain name `www.bell-labs.com` to a numerical internet address; this is a complicated process, generally involving communicating with local and remote Domain Name Servers. In the Styx model, this is done by opening the file `/dev/dns` and writing the literal string `www.bell-labs.com` on the file; then the same file is read. It will return the string `204.178.16.5` as a sequence of 12 characters.

Once the numerical Internet address is acquired, the connection must be established; this is done by opening `/net/tcp/clone` and reading from it a string that specifies a directory like `/net/tcp/43`, which represents a new, unique TCP/IP channel. To establish the connection, write a message like `connect 204.178.16.5` on the control file for that connection, `/net/tcp/43/ctl`. Subsequently, communication with `www.bell-labs.com` is done by reading and writing on the file `/net/tcp/43/data`.

There are several things to note about this approach.

> All the interface points look like files, and are accessed by the same I/O mechanisms already available in programming languages like C, C++, or Java. How-

ever, they do not correspond to ordinary data files on disk, but instead are creations of a middleware code layer.

Communication across the interface, by convention, uses printable character strings where feasible instead of binary information. This means that the syntax of communication does not depend on CPU architecture or language details.

Because the interface, as in this example with /net as the interface with networking facilities, looks like a piece of a hierarchical file system, it can easily and nearly automatically be exported to a remote machine and used from afar.

In particular, the Styx implementation encourages a natural way of providing controlled access to networks. Lucent, like many organizations, has an internal network not accessible to the international Internet, and has a few gateways between the inside and outside networks. Only the gateway machines are connected to both, and they implement the administrative controls for safety and security. The advantage of the Styx model is the ease with which the outside Internet can be used from inside. If the /net file tree described above is provided on a gateway machine, it can be used as a remote file system from machines on the inside. This is safe, because this connection is one-way: inside machines can see the external network interfaces, but outside machines cannot see the inside.

Example: debugging

A similar approach, borrowed and generalized from the UNIX system [Killi], is useful for controlling and discovering the status of the running processes in the operating system. Here a directory /proc contains a subdirectory for each process running on the system; the names of the subdirectories correspond to process IDs:

```
/proc/
        1/
                status
                ctl
                fd
                text
                mem
                ...
        2/
                status
                ctl
                ...
        ...
```

The file names in the process directories refer to various aspects of the corresponding process: status contains information about the state of the process; ctl, when written, performs operations like pausing, restarting, or killing the process; fd names and describes the files open in the process; text and mem represent the program code and the data respectively.

Where possible, the information and control are again represented as text strings. For example, one line from the status file of a typical process might be

```
samterm  dmr  Read  0  20  2478910  0  0  ...
```

which shows the name of the program, the owner, its state, and several numbers representing CPU time in various categories.

Once again, the approach provides several payoffs. Because process information is represented in file form, remote debugging (debugging programs on another machine) is possible immediately by remote-mounting the /proc tree on another machine. The machine-independent representation of information means that most operations work properly even if the remote machine uses a different CPU architecture from the one doing the debugging. Most of the programs that deal with status and control contain no machine-dependent parts and are completely portable. (A few are not, however: no attempt is made to render the memory data or instructions in machine-independent form.)

Example: PathStar™ Access Server

The data shelf of Lucents PathStar Access Server [PATH] uses Styx to connect the line cards and other devices on the shelf to the control computer. In fact, Styx is the protocol for high-level communication on the backplane.

The file system hierarchy served by the control computer includes a structure like this:

```
/trip/
        config
        admin/
                ospfctl
                ...
        boot/
                0/
                        ctl
                        eeprom
                        memory
                        msg
                        pack
                        alarm
                        ...
                1/
                        ...
/net/
        ...
```

The directories under /net are similar to those in Plan 9 or Inferno; they form the interface to the external IP network. The /trip hierarchy represents the control structure of the shelf.

The subdirectories under /trip/boot each provide access to one of the line cards or other devices in the shelf. For example, to initialize a card one writes the text string reset to the ctl file of the card, while bootstrapping is done by copying the control software for the card into the memory file and writing a reset message to ctl. Once the line card is running, the other files present an interface to the higher-level structure of the device: pack is the port through which IP packets are transferred to and from the card, alarm may be read to discover outstanding conditions on the card, and so on.

All this structure is exported from the shelf using Styx. The external element management software (EMS) controls and monitors the shelf using Styx operations. For example, the EMS may read /trip/boot/7/alarm and discover a diagnostic condition. By reading and writing the other files under /trip/boot/7/, the card may be taken off line, diagnosed, and perhaps reset or substituted, all from the system running the EMS, which may be elsewhere in the network.

Another example is the implementation of SNMP in the PathStar Access Server. The functionality of SNMP is usually distributed through the various components of a network, but here it is a straightforward adaption process, running anywhere in the network, that translates SNMP requests to Styx operations in the network element. Besides dramatically simplifying the implementation, the natural ability for aggregation permits a single process to provide SNMP access to an arbitrarily complex network subsystem. Yet the structure is secure: the file-oriented nature of the operations make it easy to establish standard authentication and security controls to guarantee that only trusted parties have access to the SNMP operations.

There are local benefits to this architecture, as well. Styx provides a single point in the design where control can be separated from the details of the underlying fabric, isolating both from changes in the other. Components become more adaptable: software can be upgraded without worrying about hidden dependencies on the hardware, and new hardware may be installed without updating the control software above.

Security issues

Styx provides several security mechanisms for discouraging hostile or accidental actions that injure the integrity of a system.

The underlying file-communication protocol includes user and group identifiers that a server may check against other authentication. For example, a server may check, on a request to open a file, that the user ID associated with the request is permitted to perform the operation. This mechanism is familiar from general-purpose operating systems, and its use is well-known. It depends on passwords or stronger mechanisms for authenticating the identity of clients.

The Styx approach of providing remote resources as file systems over a network encourages genuinely secure access to the resources in a way transparent to applications, so that authentication transactions need not be provided as part of each. For example, in Inferno, the negotiation of an initial connection between client and server may include installation of any of several encrypting or message-digesting protocols that supervise the channel. All application use of the resources provided by the server is then protected against interference, and the server has strong assurance that its facilities are being used in an authorized way. This is relevant both for general-purpose file servers, and, in the telephony field, is especially useful for safe remote administration.

Summary

Presentation of resources as a piece of a possibly remote file system is an attractive way of creating distributed systems that treads a path between two extremes:

1 All communication with other parts of the system is by explicit messages sent between components. This communication differs in style from applications' use of local resources.

2 All communication is by means of closely shared resources: the CPU-addressable memory in various parts is made directly available across a big network; applications can read and write far-away objects exactly as they do those on the same motherboard as their own CPU.

Something like the first of these extremes is usually more evident in todays systems, although either the operating system or software layered upon it usually paper over some of the rough spots. The second remains more difficult to approach, because networks (especially big ones like the Internet) are not very reliable, and because the machines on them are diverse in processor architecture and in installed software.

The design plan described and advocated in this paper lies between the two extremes. It has these advantages:

A simple, familiar programming model for reading and writing named files. File systems have well-defined naming, access, and permissions structures.

Platform and language independence. Underlying access to resources is at the file level, which is provided nearly everywhere, instead of depending on facilities available only with particular languages or operating systems. C++ or Java classes, and C libraries can be constructed to access the facilities.

A hierarchical naming and access control structure. This encourages clean and well-structured design of resource naming and access.

Easy testing and debugging. By using well-specified, narrow interfaces at the file level, it is straightforward to observe the communication between distributed entities.

Low cost. Support software, at both client and server, can be written in a few thousand lines of code, and will occupy only small space in products.

This approach to building systems is successful in the general-purpose systems Plan 9 and Inferno; it has also been used to construct systems specialized for telephony, such as Mantra [MAN] and the PathStar Access Server. It supplies a coherent, extensible structure both to the internal communications within a single system and external communication between heterogeneous components of a large digital network.

References

[NFS] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon, Design and Implementation of the Sun Network File System, *Proc. Summer 1985 USENIX Conf.*, Portland, Oregon, June 1985, pp. 119-130.

[RFC] Internet RFC 1094.

[9man] *Plan 9 Programmer's Manual*, Second Edition, Vol. 1 and 2, Bell Laboratories, Murray Hill, N.J., 1995.

[Kill84] T. J. Killian, Processes as Files, *Proc. Summer 1984 USENIX Conf.*, June 1984, Salt Lake City, Utah, June 1984, pp. 203-207.

[PPTTW93] R. Pike, D. L. Presotto, K. Thompson, H. Trickey, and P. Winterbottom, The Use of Name Spaces in Plan 9, *Op. Sys. Rev.*, Vol. 27, No. 2, April 1993, pp. 72-76.

[PrWi93] D. L. Presotto and P. Winterbottom, The Organization of Networks in Plan 9, *Proc. Winter 1993 USENIX Conf.*, San Diego, Calif., Jan. 1993, pp. 43-50.

[Nee89] R. Needham, Names, in *Distributed systems*, edited by S. Mullender, Addison-Wesley, Reading, Mass., 1989, pp. 89-101.

[CIFS] Paul Leach and Dan Perry, CIFS: A Common Internet File System, Nov. 1996, *http://www.microsoft.com/mind/1196/cifs.htm*.

[INF1] *Inferno Programmer's Manual*, Third Edition, Vol. 1 and 2, Vita Nuova Holdings Limited, York, England, 2000.

[INF2] S. M. Dorward, R. Pike, D. L. Presotto, D. M. Ritchie, H. Trickey, and P. Winterbottom, The Inferno Operating System, *Bell Labs Technical Journal* Vol. 2, No. 1, Winter 1997.

[MAN] R. A. Lakshmi-Ratan, The Lucent Technologies Softswitch Realizing the Promise of Convergence, *Bell Labs Technical Journal*, Vol. 4, No. 2, April-June 1999, pp. 174-196.

[PATH] J. M. Fossaceca, J. D. Sandoz, and P. Winterbottom, The PathStar Access Server: Facilitating Carrier-Scale Packet Telephony, *Bell Labs Technical Journal*, Vol. 3, No. 4, October-December 1998, pp. 86-102.