TECHNICAL RESEARCH REPORT

Simple Calls for Flexible Constructs Using the Traditional File API

by S. Gupta, J.S. Baras, S. Kelly, N. Roussopoulos

T.R. 96-56

Sponsored by
the National Science Foundation
Engineering Research Center Program,
the University of Maryland,
Harvard University,
and Industry

Simple calls for flexible constructs using the traditional file API*‖

Sandeep Gupta†, John S. Baras§, Stephen Kelley‡, Nick Roussopoulos†‡
Institute of Systems Research, University of Maryland, College Park, MD 20742
{sandeep,nick,skelley}@cs.umd.edu, baras@isr.umd.edu

Abstract

We present the *design* for a remote qos control interface to the transport protocol based on existing work for similar applications. This puts together the read/write calls from the traditional file system API and an additional primitive. The addition amounts to programming an operating system data-streaming service which may be provided as a system call or otherwise using the standard techniques. Put together these allow much more than the traditional call based control interface. The resulting interface simplifies the mechanisms for distributed control. Parts of this interface have also been implemented in our ongoing experiments with file transfer.

1 Introduction

In this note we make a case for simple kernel construct for transport control interface, based on closely similar implementations for a data transfers service on Unix. The use for this construct would be to remotely control protocols between machines that can establish authenticated connections. Using this design, it is possible to bypass the tradition user space based implementations.

The construct is simple, it uses Existing ideas, and *one* compatible extension. The extension can extend the interface very flexibly, as it is modular. The basic concept was implemented, in a very similar application, tested for control of file transfer.

2 Background

The traditional control interface for transport protocols on Unix uses the `ioctl()` system call (Described in the Unix online manual: `man` pages.). Examples of it's use may be found in [4] [7]. If a system is to receive a control request from a remote system, the way to set this up would be to have a process wait for these requests, and make a local ioctl call based on the control request. This amounts to an remote procedure call like mechanism. These will have a response time dictated by the speed at which the process can be scheduled. While that is of some importance, in a multiprocessing system it cannot be avoided all together. What can be avoided is the scheduling

†Also with Department of Computer Science.

‡Also with Institute of Advanced Computer Studies.

§Also with Department of Electrical Engineering.

*This material is based upon work supported in part by the National Science Foundation under Grant No. NSF EEC 94-02384, and by the Center for Satellite and Hybrid Communication Networks under NASA contract NAGW-2777 and the State of Maryland. Part of this work was done by the first author as a summer intern at AT&T Bell Laboratories, Murray Hill in Summer '94.

‖This writeup is based on the presentation made at the OPENSIG II Workshop: Open Signalling for Middleware and Service Creation at Columbia University, April 29-30, 1996, in the session: APIs for Transport.

of these requests entirely by user processes. If a number of connections are being serviced by a machine, a remote control would either entail user processes per connection or a centralized user space process for all these requests. It is possible to avoid this overhead and programming. It uses two key ideas. One is the use of the `write()` system call to send control information to the kernel. This is not a new idea, and has been in use [3] [5], and is also used in Plan 9 OS [6]. The second idea is to use a construct in the kernel, that can connect a remote socket to this write interface. This construct, is similar in functionality to [1] but differs in the implementation. Relevant to this description, the differences are in this implementation being at a higher level of abstraction, and the user interface being the abovementioned write call. Details of this construct and it's evaluation are reported in [2].

3 Required extension

The basic service expected from the extension is to allow read/write to proceed in *kernel* with the standard kernel end-point abstraction, namely the file descriptor. The interface to this service is given to the user process as the same standard end-point (but a different descriptor) to write control information to this in-kernel read write service. Since the control interface to this read/write service is another descriptor, remote control entails simply invoking the same service twice, the second time with a socket connected to the remote machine and the control descriptor from this service. We validated this concept by a prototype implementation on Unix.

4 Why is this interface interesting ?

There are several interesting features of this interface, other than streamlining the control path from a remote application. The invocation of the control path is per instance by the user process. It need not be be statically configured in the system afresh, for every new program as a RPC server or part of the program that polls requests from a remote machine. Since this concept is based on standard read/write, the concept is portable to other operating systems as well. Finally, since the kernel data transfer service returns a descriptor of the same type that it uses as it's input parameter, the service is composable as a construct, i.e., the control interface can use the read write service again.

5 Example of Use

Following is an example of an implementation tested on Unix. Variables `ctl, s, d` are standard descriptors. Of these, `s` would be obtained by opening a source file, and `d` by a connecting to a remote machine via a socket.

```
/* paraphrased */
ctl = readWriteService (s, d);
write (ctl, COMMANDS, T);
```

Here, `COMMANDS` is an array with requests from the service, and `T` is the size of the array, as usual in `write()`. These can be of an arbitrary length and type. The current implementation uses integers.

Alternatively, the commands can be sent from a remote machine, via a socket, say `r`:

```
ctl2 = readWriteService (r, ctl);
```

Following this call, a remote machine can send control requests on socket `r` exactly as they would be written to it from the local machine.

6 Conclusion

We presented the design for a remote control interface for the transport protocols. The design is simple, as it uses familiar constructs and one compatible extension. The tests have been limited to a data transfer application. Additional security considerations may have to be taken into account while using this design for other protocols. In any case, this should not be considered any more secure than the socket that is used to send control information. Finally, for each controlled protocol, there will be a need to supply, or otherwise configure checks to ensure that a remote system command is not in error, or is not otherwise misinterpreted.

We have tested this concept for data transfer applications. Such a construct will allow streamlining distributed control.

References

[1] Kevin Fall and Joseph Pasquale. Exploiting in-kernel data paths to improve i/o throughput and cpu availability. In *Usenix Winter Conference*, pages 327–333. University of California, San Diego, January 1993.

[2] Sandeep Gupta, John S. Baras, Stephen Kelley, and Nick Roussopoulos. Cues: File subsystem data streams. Technical Report 96-53, Univesity of Maryland at College Park, 1996.

[3] Tom Killian. Processes as files. In *Usenix Summer Conference*, Salt Lake City, May 1984. AT&T Bell Laboratories, Murray Hill, NJ 07974.

[4] Leffler, et al. *The Design and Implementation of the 4.3 BSD UNIX Operating System.* Addison-Wesley Publishing Company, 1989.

[5] Ashok V. Nadkarni. The processor file system in Unix SVR4.2. In *USENIX File Systems Workshop Proceedings*, Ann Arbor, MI, May 1992. Unix System Laboratories.

[6] Rob Pike, Dave Presotto, Ken Thompson, Howard Trickey, and Phil Winterbottom. The use of name space in Plan 9. Technical report, AT&T Bell Laboratories, Murray Hill, NJ 07974.

[7] W. Richard Stevens. *Unix Network Programming.* Prentice Hall, 1990.