TECHNICAL RESEARCH REPORT

Managing File Subsystem Data Streams for Databases on Networked Systems

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

CSHCN T.R. 96-13
(ISR T.R. 96-60)

The Center for Satellite and Hybrid Communication Networks is a NASA-sponsored Commercial Space Center also supported by the Department of Defense (DOD), industry, the State of Maryland, the University of Maryland and the Institute for Systems Research. This document is a technical report in the CSHCN series originating at the University of Maryland.

Web site http://www.isr.umd.edu/CSHCN/

Managing file subsystem data streams for databases on networked systems*

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

July 2, 1996

Abstract

One important activity for networked database systems that distribute data across several workstations is moving data between the file and network subsystems. It is possible to create data streams in the operating system kernel. If provided on a system, they allow user level processes to request transfer of data without having to copy it into the user space. This is particularly useful for data whose content or format is not modified during the transfer. In this paper we present a conservative criterion for access and control for the management of such data streams for databases in a networked environment, and define the implementation requirements for achieving the criterion. The approach is to maintain atleast the current level of access management. We define the specific implementation semantics that this criterion entails.

[†]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, by the University of Maryland Institute of Advanced Computer Studies, by ARPA under Grant No. F30602–93–C–0177.

1 Introduction

Distributing database files over workstations connected by a network is a common concept. It is useful for reliability, scaling performance, and resource sharing. Unix DBMS over a network of workstations would typically require to move a lot of data among the systems. The system calls needed to move data from the file subsystem to the network requires data to be moved in and out of the kernel, into the user space. Typically, large transfers are done piecemeal by the user process, using read and write system calls. The copying and the scheduling this way, costs more than it could, and can be improved.

It is possible to create data streams in the operating system kernel, using several abstractions [2], [1], [4] and [5]. These are useful for improving system performance while transferring data whose content or format is not modified. Such data streams over the Unix file subsystem [2] provide good performance while retaining the process and file descriptor abstraction. The implementation abstraction called Cue, performs data transfer over the file subsystem on request from user processes. The end-points of the transfer are specified using standard file descriptors. A control descriptor is provided to the user process when it creates a Cue ADT using a new system call (`cue()`). This descriptor is used to manage the data stream. Since the input to `cue()` are descriptors, it is possible to design the semantics of the call such that the control descriptor obtained from an invocation of such a call may be used as one input to another invocation. This allows the data streams to be managed as a distributed resource. In this report, we discuss the requests that are useful on this ADT and those that may be permitted in a networked database environment. Certain semantics are required of the implementation in such a setup, and in this paper we present the requirements for managing these data streams.

2 Environment

The environment for this work is the storage subsystem for a networked database system called Top [6], on Unix workstations. The secondary storage for the database may be spread over more than one system. Each participating system maintains a cache of pages from database files retrieved from one or more systems. Database files are striped over several systems. The transfers are for large amounts of data, and may often be individual pages from files. The files contain database tables and large images. The discussion in the report will be with this application in mind.

This application brings two new requirements over the implementation reported in [2]. First, this is general purpose environment, and the call may have to be provided as an unrestricted service. Providing the call as an open service entails additional safeguards that should come with providing a kernel service. Second, maintaining the advantage of striping requires initiating and using data retrievals from several systems, simultaneously. Considering them as a local as well as a remote resource requires access management.

3 Criterion for managing the data streams

As mentioned above, one of the key features of `cue()` is the abstraction that allows composition of a kernel path to remotely direct transfers from the client. Before providing this call for general use, the first requirement is to protect the system that provides data streaming service in the kernel. The case for `cue()` is premised on providing a service that offers less load for a frequent activity on an important set of systems (Data servers). Consequently, by default the remote system should not be able to request any more load from the Cue than it could using a standard user process

servicing its requests. In general, the remote client should not be able to request transfers at a rate any greater than that the server specifies. Even the local user process should not be able to configure, or allow configuration of the transfer at a rate greater than that available to the process. Similarly, the remote system should not be able to request any data that it would not be able to request in the standard configuration. Accounting should be possible on the local system, and the server should be able to request notification of data transfers in progress, at a level it could, were it to use `read`, `write`.

To summarize, in our view the evaluation criterion for introducing a service in an existing operational system, would best be conservative for each new feature: `cue()` should not make the system's resources, namely processor time, files, and data any more vulnerable than with the existing system calls. The calling process should be able to retain accounting and notification capability as with existing calls, as is applicable. In the next section we present the requirements for supporting this criterion.

4 Supporting manageable semantics

The scope of applying the criterion mentioned in the previous section is defined with respect to data transfer requirements of networked DBMS. In the current implementation, the semantics are for simple data transfers from the disk to network. Cues, as it is, may not be extended for general purpose transfers. Providing this as a service used by a remote system other than in a testing environment requires enhancement of the basic semantics.

The networked database data transfer environment requires consideration on resource protection, enhancements for notification of progress of transfers, accounting during the transfers, and for managing multiple streams. The application environment cannot be as tightly controlled as the test environment. Consequently, data transfers may take longer than expected, and may be interrupted. Appropriate semantics for recovering from interrupted transfers needs to be added.

4.1 User Interface

Three operations supported by Cues. These are requests to (a) set the size of data transfer buffer, (b) initiate transfer of a given size on the kernel data stream, and (c) set offset into the *source file*. Since we are moving this to an application, the operations need to be augmented. The operations in [2] are unacknowledged. The execution is assumed to be correct, since simple tests routines and reliable connections were used to evaluate the concept. In the case of (a) and (b), the return from the call must be an appropriate error value. In the third case, in addition to the error value, it must return the number of bytes actually transferred in response to the request. For this discussion, the notation will be SETBUFSZ(W), TRANSFERBYTES(N), and SETOFFSET(B) respectively. Parameters w,n,b are specified in bytes. The Cue object that is set up in the kernel will be referred to as Cue. These requirements will be reflected in the subsequent discussion.

4.2 Resource Protection

The resources to be protected are all kernel resourcece that are visible to the Cue, from processes / connections accessing it. Cue operations use parameters that may be set by the user process. It is important that access to these parameters be controlled. The standard resource to be protected on the host is access to the files and processing time. Retaining the current level of protection needs to be viewed from the following perspectives:

Local permissions available to the process on the system implementing Cues, and

Remote permissions available to the remote process when the control descriptor of a Cue is connected to a peer process using another Cue. (Remotely controlled transfers.)

Remote access may be through a socket with a remote system, or via a Unix domain pipe on the local system. Kernel resource access is managed by the layer over the file subsystem. It is the responsibility of this layer to maintain access control to the files on behalf of users and processes. We will use the following definitions.

Definition 4.1 *A Cue is said to be* **active** *if the process that owns it is in the kernel mode, and the Cue will perform a data transfer before this process returns to user mode.*

Definition 4.2 *An active Cue will respond to requests or perform data transfers only when it is scheduled during the time it is active. At such time, using the process terminology, a Cue is said to be* **running**, *or* **executing** *a specific transfer request.*

Requirement 4.1 *Verify permissions at each access to any protected object during the execution of* TRANSFERBYTES( ).

Maintaining local access permission to files is simpler as the abstraction is over the file subsystem. Data transfer by a Cue is done entirely in kernel mode, in the context of the user process that set it up. Consequently, per process file flags are available for examination, for each read/write on the source and destination file descriptors. These can also be examined when the Cue is set up. The permissions on the file are subject to change by the file owner, even if they have been granted access at the time the file was opened by the process using it. This may happen between two TRANSFERBYTES( ) requests as well. Therefore they should be checked before *each* kernel access to the file for read/write. This is even though subsequent requests may be part of the same TRANSFERBYTES( ), and the transfer may still be in progress, in the kernel.

Remote file access permissions go through the same routines for data transfer. Remote access to data can be at any offset, and may be initiated multiple times. The read/write permission checks for each transfer ensure that no part of the file is read after the access to it has been changed.

In addition to the change in permission, it is also possible that the process's file descriptors used in the initial invocation may point to different objects for two different TRANSFERBYTES( ) requests. This is discussed next.

Requirement 4.2 *Verify the value of input parameters during the execution of each operation.*

The parameters to the call *cue(s,d)* made by a process are descriptors to the files opened earlier. These are verified to be valid, and converted to pointers to the kernel file table at the time the Cue is set up. As mentioned following requirement 4.1, it is not guaranteed that the file descriptor used in the call to set up the Cue points to the same object between successive requests. The user process requests to set up a Cue, and it owns the two file descriptors. As a consequence, if the value of these descriptors were to change, the correct semantics would require that this change be reflected. The owner of the file may let another process change these permissions at any time during this transfer. This entails that a Cue store the state in terms of the user file descriptors provided to it by *cue(s,d)* , and these be converted to kernel file descriptors before each access by the Cue operations. This will ensure there is no access to a closed file, and if the user file descriptor points to a different file, that is used. The same checks will apply for descriptors with remote requests.

The descriptors should not change during the execution of a transfer. This would make the semantics, particularly the interpretation of the offset into the file, hard to define meaningfully. This is because the offset cannot be made available to the user process without switching back to user mode. It is not possible to restrict the user from trying to make this call without changing the code associated with other system calls for files. Instead, to keep the call simple, and changes to other system calls minimal, we restrict the request for initiating data transfer on the stream to block until the transfer is completed. Thus,

Requirement 4.3 *All* TRANSFERBYTES(N) *requests should block until n bytes have been transferred, except when such transfers are interrupted.*

As an immediate consequence of the above restriction,

Requirement 4.4 *If either s or d in a call* cue(s,d) *are set to non-blocking I/O, the system call should not set up the Cue.*

Other than the data transfer operation, which sets the size of the transfer, SETBUFSZ(W) sets the buffer size for each individual read and write to w bytes. The value of w is checked against a fixed maximum and minimum defined at compile time. SETOFFSET(O) makes a change in the file offset field. Local permissions for change in offset are exactly the same as that using the `lseek` system call which is standard.

Setting the buffer size on the local system can be justified by precedent, where BSD Unix Networking [3] is used. The system call to set socket options, `setsockopt` can be used to change the size of a kernel buffer at an even lower abstraction in the kernel. Similarly, the transfers are scheduled to ensure they use the process's cpu share, so their length can be specified locally.

It is hard to justify permissions to remotely modify these values for a Cue, based on a precedent. With BSD networking software, though it is common practice to use a permissions file for defining remote hosts that can execute programs for a given user. Similarly file system access permissions are granted using such configuration files. These two put together can be used to access per process resources. We do not consider such use common, and it is not desirable to keep these resources open, by default. In contrast to simple files, Cue is resource local to the process, like the socket buffer size on a BSD socket.

Of the three parameters, transfer size, offset, and buffer size, the buffer size would not modified often, once a Cue is set up. Requests from the file server in networked database applications is typically in groups of pages. Changing the offset is particularly useful, as pages may be requested out of order. Similarly, specifying the size will allow multiple sequential pages to be grouped in one request. In the current implementation, the schedule for reading and writing pages was set after empirically measuring the time taken by the read/write from the file subsystem. The schedule is fixed, though it can be changed by the user process, if it has superuser privileges. Allowing a remote process to specify the transfer and buffer size indirectly amounts to remotely requesting a workload about which the process that set up the data stream will not know. This needs to be prevented in the default case. We need to classify the operations.

Definition 4.3 *The messages for operations on the Cue can be classified into two types, using the above discussion. Those, that attempt to* **configure** *the Cue, and those that send* **auxilliary** *information that does not have any affect on the load offered to the system which runs the Cue.*

One way of ensuring that remote systems may not send configuration messages by default, is to block these altogether, and only allow superusers to set up a Cue that can be controlled remotely.

This would restrict its usefulness for general application processes. It is possible for the Cue to distinguish between an invocation for data path from one that connects the control descriptor of an existing Cue to a remote socket. This can be done because the descriptor returned by `cue()` can be distinguished in the kernel, just as kernel descriptors for sockets, files, and on some systems pipes, can be distinguished from each other.

Using this distinction, it is possible to ensure that no configuration messages are forwarded on the Cue. When these messages are received, they can be passed up to process via the control descriptor of Cue. Subsequently, these messages may be sent back to the Cue on the same descriptor. When a message has been redirected to the control descriptor, the data stream should be stopped. The call (`cue()`) needs to return to the process with an indication that data transfer has been interrupted and a (*privileged*) configuration message has been received. This can be summarized in the next four requirements.

Definition 4.4 *A Cue set up for data transfer is called* **data Cue**, *and that set up for controlling an existing Cue from a remote connection is called* **control Cue**. *The same adjectives may be used for distinguishing between the corresponding file subsystem streams.*

Requirement 4.5 *In the default set up, a control stream Cue should not forward the configuration messages. These are passed up vertically on the control descriptor. These messages may be sent down on the same control descriptor, by the user process.*

Requirement 4.6 *An new operation is required to send configuration messages to a Cue via the control descriptor of it's control Cue.*

We add the following:

Requirement 4.7 *From the perspective of the process that creates a Cue, the* data *stream is one way only.*

Requirement 4.7 pertains to identifying the eventual destination of an intercepted configuration message, i.e., the ones forwarded vertically. It is possible to add complexity to the messages and drop this requirement, but it does not seem particularly useful right now. The order of parameters to the `cue()` call determine the source and destination for the cue.

Descriptors may be passed among processes on a system. The Cue control descriptor points to a Cue object in the kernel which contains references to two other descriptors. One of these may be a Cue descriptor. There is a system call to pass descriptors across unrelated processes. One alternative is to overload this system call and pass the Cue object along with some state information from the process, recursively. It complicates the semantics, and requires modification to an unrelated system call. Cue requires process related state. At this point, there seems to be little use for passing a Cue descriptor. Thus, every request to the Cue service routines in the kernel needs explicit or implicit assurance that a Cue descriptor refers to one created by the process responsible for this execution of the Cue. This requires the Cue to store the original process identifier at the time of creation.

Requirement 4.8 *The process id should be checked with the original process id, for every invocation of Cue service routines in the kernel.*

Recall that for most of these calls, we are considering the default case, and we want to define the permissions such that system to may not be any more vulnerable than that using standard options. It may be possible to override some of these restrictions.

The data and control streams are also resources that need to be protected. Protecting the kernel streams requires monitoring them. This is discussed in the next subsection.

4.3 Notification of Progress

A request to initiate large data transfers on a Cue differs from read/write based transfers in two respects. Individual instances of the read and write system calls return to the user process. The time taken by the individual read and write call is determined by the amount of data to be read. This in turn is dependent on how large a buffer the user process can allocate for the data to be read. The amount of data transferred by a TRANSFERBYTES(N) can be as large as MAXINT bytes, provided there is that much data in the file. Since TRANSFERBYTES(N) blocks until completion, there is no way for a user process to figure out the status of a transfer while it is in progress. Obviating returns to the user mode while performing large transfers on behalf of the user process is a *feature* of this system call. With just these semantics it means that the user process cannot know if the data transfer is stalled.

The return from the call can be used to provide the number of bytes actually transferred. The call uses the standard error values that the file subsystem returns to the user process. If the entire request has been serviced, the return values can indicate success, locally. A process controlling a remote Cue cannot access this value directly. It cannot it interrogate the process that has set up the remote Cue, as it would be blocked until the transfer size set up for the remote Cue is actually complete. In general, a control stream can be set up for servicing multiple requests.

Returning a message on the control descriptor solves this problem. Since the abstraction is over the file subsystem, to maintain identical semantics for local as well as remote control requires each of these messages to be read from the control descriptor. The reads can be asynchronous. This is unlike system call based requests which return the value *synchronously* and maintain the results of only the last request made from the kernel. It is a consequence of the fact that remote control requests use sockets, which buffer messages and new messages do not overwrite messages received earlier.

Using this method, a process can monitor a local stream by forking another process which reads the requests serviced. Then it would be useful to set intermediate notification of transfers as well, after a fixed size of bytes have been transferred. This will allow a user process to monitor the status of the transfer.

With this much, when a transfer is being controlled remotely, the process that set up the stream will not be able to monitor the data stream. The data stream is a local resource, and it should be possible for the process to monitor it. This will be especially useful if a forked process runs or monitors the Cue. (An example of such a use will be seen in subsection 4.4.) It can be taken care of, by the following:

Requirement 4.9 *A message should be returned on the control descriptor at the completion of each* TRANSFERBYTES(N) *request, returning the exact count transferred. When the call is interrupted, the exact count of bytes transferred should be returned, along with a message indicating that the transfer was interrupted.*

Transfers can be interrupted by software interrupts. An active Cue relinquishes the processor periodically, and raises the priority level to ensure other processes can be scheduled. The raised priority level would allow software interrupts (signals) to be received.

To monitor the transfer more closely than the entire length requested, the natural choice is the size of the transfer buffer set by SETBUFSZ(W). This keeps the code for tracking the transfers simple.

Definition 4.5 *The messages returned on the control descriptor in response to an operation performed need to be classified as as* **status** *messages.*

Requirement 4.10 *A new operation is required to set granularity of the status message for transfers in progress, in terms of the the number of transfers (of size w).*

These messages can be distinguished by the Cue service routines. The desired default requirements for these messages are:

Requirement 4.11 *Status messages have to buffered at the control descriptor until they are read.*

Requirement 4.12 *A control stream should copy status messages vertically, in addition to forwarding them.*

4.4 Managing multiple streams

Only one Cue can be active for streaming per process, and it runs only when the process is scheduled. The execution is in kernel mode. With the given design and this mechanism for scheduling, it is not possible for a single process to have several Cues simultaneously active or remotely controlled by other machines.

Two workarounds are possible with the current set up. Multiple pairs of data and / or control Cues can be set up by the user process and they can be scheduled in round-robin or another suitable manner. Alternatively a process can be forked for each pair of data and control Cues, and the parent process can remotely control the appropriate cues via a socket, if required. This is an additional level of indirection. The performance of scheduling using a single process will depend on the workload.

The unix `select()` call is designed to help multiple transfers in parallel. Currently the Cue ADT can accomodate only one transfer stream. Since `cue()` leaves the descriptors accessible by the user process, it would be possible to use select with multiple cues. This would not be efficient, as it would require control to return to the user process (for each return of `select()`) before each transfer is initiated. On the other hand allocating a completely new data structure per stream will be wasteful. One of the main reasons for efficiency of kernel data streams is that the user/kernel address space switch is not needed for each quantum of transfer. The modification for multiple transfer implementation needs to be designed with the above in mind.

5 Further work

The scheduling in the kernel ensures that the priority of the process is raised to let other processes be scheduled, and ensure the delivery of signals. The call does not keep any state that would be required to restart the transfer on the stream, if a signal were received and the call was interrupted. This needs work.

One of the most needed features, from the point of view to simplify programming and the model is the support for concurrent cues per process. One of the reasons for not attempting this at present is that scheduling of the Cues is still naive. Determining the schedule for the transfer, on the fly, as mentioned in [2] still remains on the wish list. Most of the requirements are simple permission checks. The performance impact of these will be considered once the scheduling is mature.

There has been no discussion on use of remote Cues on a heterogenous set of workstations. In particular, the control messages require a presentation wrapper to map between machine data formats. Another rare possibility in this setup that will need consideration is the recovery from partly received control messages on a remote Cue, if an application decides to use connectionless sockets. At this point, we require remote Cues to use reliable connections only.

Another important detail not considered yet, is a simpler and familiar problem. With remote Cues on heterogenous workstations, the byte order of the native machine matters. The evaluation tests in [2] used machines with similar byte order for integers. To extend this to heterogenous machines the format of the remote requests needs to take care of the possibility of difference in byte order. Currently, Cue control descriptor uses integers for operation codes, as well as parameter values. Before providing this as a general purpose service, the remote requests need to be converted to the format on the host machine. This can use the same solution as used by networking and RPC software, of mapping the bytes in integers to network order and then back to the order on the host. In addition, verifying the requests, and recovering from incomplete requests needs to be implemented. The number of operations from remote Cues is small, and the operation codes are integers. The codes can be selected such that any change in the order of integers results in an invalid code. The opcode can thus be the preamble of the message. A timeout based mechanism would be useful for recovering from errors.

6 Conclusion

If the `cue()` call is used on special purpose systems, such as a standalone file servers for DBMS, etc., it can be reserved for use by applications started by a superuser `uid` or a privileged group `gid`. In a networked DBMS environment, the call needs several features that were not critical for this configurations, or for the initial tests [2] to test the concept. Managing the data streams as a resource introduces several resource protection considerations. This report defines a conservative criterion for resource protection and the requirements to implement this criterion. The requirements are built by analogy with similar features in the system. The feasibility of implementing them is also discussed.

7 Acknowledgements

A discussion with Dennis M. Ritchie helped define initial ideas for the design of abstractions for these data transfers; some of the approach to safety considerations in subsection 4.2 resulted out of issues pointed out by David L. Presotto. [Both at Bell Labs.]

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, Institute of Systems Research, Univesity of Maryland at College Park, 1996.

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

[4] Dennis M. Ritchie. A Stream Input-Output System. *AT&T Bell Laboratories Technical Journal*, 63(8):1897–1910, 1984.

[5] Debanjan Saha, Dilip Kandlur, and Marc Willebeek-LeMair. Protocol architecture for multimedia applications over ATM networks. Technical report, IBM T.J. Watson Research Lab, 1994.

[6] Michael Tan, Nick Roussopoulos, and Stephen Kelley. The tower of pizzas. Technical Report 95-52, Institute of Advanced Computer Studies, Univesity of Maryland at College Park, 1995.