acm

OPERATING SYSTEMS REVIEW

A Quarterly Publication of the
Special Interest Group on Operating Systems

Volume 17, Number 5 Special Issue

Proceedings of the Ninth ACM Symposium on Operating Systems Principles

10-13 October 1983
Mount Washington Hotel
Bretton Woods, New Hampshire

ACM ORDER NO. 534830

Proceedings of
the Ninth ACM Symposium
on
Operating Systems Principles

10-13 October 1983
Mount Washington Hotel
Bretton Woods, New Hampshire

Sponsored by

Special Interest Group on Operating Systems (SIGOPS)

Association for Computing Machinery (ACM)

The Association for Computing Machinery, Inc.
11 West 42nd Street
New York, New York 10036

ISBN 0-89791-115-6

Additional copies may be ordered prepaid from:

ACM Order Department
P.O. Box 64145
Baltimore, MD 21264

Price:
Members: $11.00
All others: $15.00

ACM Order Number: 534830

Ninth ACM Symposium on Operating Systems Principles

General Chairman

Jerome Saltzer *Massachusetts Institute of Technology*

Program Co-Chairmen

Roy Levin *Xerox*
David Redell *Xerox*

Program Committee

David Clark *Massachusetts Institute of Technology*
James Gray *Tandem Computers*
Keith Lantz *Stanford University*
John Ousterhout *University of California, Berkeley*
Fred Schneider *Cornell University*
Alfred Spector *Carnegie-Mellon University*
Kenneth Thompson *Bell Laboratories*
John Zahorjan *University of Washington*

Symposium Staff

Secretary: Muriel Webber *Massachusetts Institute of Technology*
Treasurer: David Clark *Massachusetts Institute of Technology*
Meeting Site: Robert Graham *University of Massachusetts*
Printed Materials: David Reed *Massachusetts Institute of Technology*
Registration: Butler Lampson *Xerox*
European Liaison: Liba Svobodova *IBM*

Referees

Unlike most previous SOSPs, the 9th SOSP has relied almost entirely on the Program Committee to review submitted papers. There were some cases in which outside referees were used, however, and we wish to acknowledge their important contribution to the Symposium.

Guy Almes
Ozalp Babaoglu
Sayed Banawan
Ken Birman
Hans Boehm
Jordan Brower
Susan Cady
Alan Demers
Robert Fowler
John Gilbert
David Jacobson
Warren Jessup
Eric Jul
Ed Lazowska
Bruce Lindsay
Barry McCord
Dennis Ritchie
Alan Shaw
Peter Weinberger
David Wright

FOREWORD

The Ninth Symposium on Operating System Principles continues the tradition of its eight predecessors by reporting on significant original work in the field. The papers in this proceedings range from design proposals to implementation lessons, from storage allocation to security verification, from data storage to communication protocols. While many focus on topics and issues in distributed systems, the papers as a whole are indicative of the true breadth of the field.

The strong technical history of these symposia and their relative infrequency (bi-annual) exerted considerable influence on the makeup of the program committee and on the reviewing process for the submitted papers. Ten committee members, displaying wide technical diversity, were chosen from six academic institutions and three industrial organizations. More than half were serving on an SOSP program committee for the first time. Departing from the approach followed in recent years, the committee members decided to review most of the papers themselves, consulting outside referees in those cases where special expertise was needed. A two round reviewing process was employed, ensuring that at least three committee members read each submission and that most members read most of the papers. The committee was thus very well informed when it met and selected sixteen papers from a field of eighty-three for inclusion in the symposium. Three outstanding papers were nominated and subsequently accepted for publication in a special issue of the ACM Transactions on Computer Systems. A fourth paper was independently submitted to and accepted by TOCS. In accordance with ACM policy, only extended abstracts of these papers appear in the proceedings.

A good technical program is necessary but not sufficient for a successful conference. In addition to the program committee and other symposium officials, we wish to thank Kathi Anderson, Shannon McElyea, and Muriel Webber, whose diligence and care have greatly contributed to the quality of this conference.

Roy Levin and Dave Redell
Program Committee Co-Chairmen
August, 1983

Ninth ACM Symposium on Operating Systems Principles

11 October 1983

Session I: Communication

Session II: Resource Management

Session III: Special Session

12 October 1983

Session IV: LOCUS

Session V: Work in Progress

A series of short presentations describing projects currently in progress.

Session VI: Recovery and Reconfiguration

Session VII: Debate

RESOLVED: *That network Transparency is a Bad Idea*

13 October 1983

Session VIII: Distributed File Access

Session IX: Experience

* In keeping with ACM publication policy, papers that will appear in the ACM Transactions on Computer Systems are reprinted here in extended abstract form only.

Computation & Communication in R*: A Distributed Database Manager

by

Bruce G. Lindsay, Laura M. Haas, C. Mohan, Paul F. Wilms, & Robert A. Yost

IBM San Jose Research Laboratory
5600 Cottle Road
San Jose, CA 95193

EXTENDED ABSTRACT

R* is an experimental prototype distributed database management system. The computation needed to perform a sequence of multi-site user transactions in R* is structured as a *tree* of processes communicating over virtual circuit communication links. Distributed computation can be supported by providing a server process per site which performs requests on behalf of remote users. Alternatively, a new process could be created to service each incoming request. Instead of using a shared server process or using the process per request approach, R* creates a process associated with the computation of the user on the first request to the remote site. This process is incorporated into the tree of processes serving a single user and is retained for the duration of the user computation. This approach allows R* to factor some of the request execution overhead into the process creation phase, and simplifies the retention of user and transaction context at the multiple sites of the distributed computation.

R* uses virtual circuit communication links to connect the tree of processes in an R* computation. Virtual circuits provide message ordering, flow control, and error detection and reporting. Especially important in the distributed transaction processing environment is the ability of the virtual circuit facility to detect and report any process, processor, or communication failures to the end points of the virtual circuit. Error detection and reporting by the virtual circuit facility is used to manage the tree of processes comprising a user computation and to handle correctly the resolution of distributed transactions in the presence of various kinds of failures.

R* uses the communication facility in a variety of ways. Many functions use a syn-

chronous, remote procedure call protocol to perform work at remote sites. Site authentication, user identification, data definition, and database catalog management are all implemented using this remote procedure activation protocol. Query planning, on the other hand, distributes query execution plans in parallel to the sites involved. Parallel plan distribution allows server sites to overlap the computation needed to validate and store query execution plans. The query execution plans often involve passing data streams from site, to site with each site transforming the data stream in some way. The execution of data access requests exploits virtual circuit flow controls to allow overlapped execution at data producer and data consumer sites.

Finally, distributed transaction management in R* uses the virtual circuits connecting the process tree to exchange the messages of the two-phase commit protocol. However, if a failure occurs during the commit protocol, the virtual circuits and process of the original computation may be lost. When failures interrupt the distributed commit protocol, R* reverts to a datagram-oriented protocol to transfer the messages needed to resolve the outstanding transaction. R* also uses datagrams to communicate the information needed to detect multi-site deadlocks.

The R* approach to distributed computation may be contrasted with datagram-based and server-oriented distributed systems. The retention of remote processes, and the virtual circuits connecting them, for the duration of the user computation improves execution performance whenever repeated accesses are made to a remote site. The retention of remote processes is also helpful for maintaining user and transaction context between requests to the remote site. The use of virtual circuits allows several concerns, such as message ordering and flow control, to be relegated to the network and virtual circuit implementation. The ability of the virtual circuit implementation to report failures is fundamental to the management of the R* distributed computation.

Currently, R* is running on multiple processors and is able to perform any SQL statement on local or remote data. This includes not only data definition and catalog manipulation statements, but also n-way joins, subqueries, and data update statements. Besides the SQL language constructs, the transaction management and distributed deadlock detection protocols are implemented and running. The tree structure of the R* computation and the use of virtual circuits have proved to be quite well adapted to the problems of implementing and controlling the complex, distributed computations needed to support the execution of a distributed database management system.

Implementing Remote Procedure Calls

Andrew D. Birrell and Bruce Jay Nelson
(Abstract written by Andrew D. Birrell)
Xerox Palo Research Center,
Palo Alto, CA 94304, USA

Remote procedure calls (*RPC*) are a useful paradigm for providing communication across a network between programs written in a high level language. This paper describes a package, written as part of the *Cedar* project, providing a remote procedure call facility. The paper describes the options that face a designer of such a package, and the decisions we made. We describe the overall structure of our RPC mechanism, our facilities for binding RPC clients, the transport level communication protocol, and some performance measurements. We include descriptions of some optimisations we used to achieve high performance and to minimize the load on server machines that have many clients.

Our primary aim in building an RPC package was to make the building of distributed systems easier. Previous protocols were sufficiently hard to use that only members of a select group of communication experts were willing to undertake the construction of distributed systems. We hoped to overcome this by providing a communication paradigm as close as possible to the familiar facilities of our high level languages. To achieve this aim, we concentrated on making remote calls efficient, and on making the semantics of remote calls as close as possible to those of local calls.

To use the package, a programmer designs an interface module (just as he would for a single-machine program), then uses a translator called *Lupine* to produce *stub program modules* which are responsible for converting local calls into calls on a package which provides node-to-node packet transport. A program wishing to make a remote call just makes a local call to the appropriate stub module. The stub module causes the RPC runtime system to transport the appropriate packets to the corresponding stub module on the destination machine, where the packets are unpacked and a local call is made to the programmer's module implementing the desired procedure.

We provide facilities for binding the parts of a distributed computation together at runtime. These facilities use the Grapevine distributed database to locate the appropriate nodes on our internet, then make an RPC call to the appropriate node to obtain the binding information. The binding is performed at the level of an interface module.

Our packet transport protocol concentrates on two techniques to achieve efficiency. Firstly, we are careful to minimize the number of packets transmitted for simple calls. The simplest calls take just one packet in each direction. Secondly, we designed our connnection management to ensure that connection establishment and take down are cheap, and that maintenance of large numbers of connections does not impose an undue load on the server nodes. When a connection is idle, the only state maintained by a caller is the binding information and a machine-wide sequence number; the state maintained by a callee is just the identity and last sequence number of the caller. Furthermore, the callee (which we expect to be typically a server machine shared amongst many users) can discard the state information on a connection after a suitable interval.

The package is fully implemented, and we are in the early stages of acquiring experience with its use. The paper includes some measurements of the performance of the system on test cases. We believe the parts of our RPC package that we discuss are of general interest in several ways. They represent a particular point in the design spectrum of RPC. We believe that we have achieved very good performance without adopting extreme measures, and without sacrificing useful call and parameter semantics. The techniques for managing transport level connections so as to minimize the communication costs and the state that must be maintained by a server are important in our experience of servers dealing with large numbers of users. Our binding semantics are quite powerful, but conceptually simple to a programmer familiar with single machine binding. They were easy and efficient to implement.

An Asymmetric Stream Communication System

Andrew P. Black

Department of Computer Science, FR-35,
University of Washington,
Seattle, WA 98115

Abstract

Input and output are often viewed as complementary operations, and it is certainly true that the direction of data flow during input is the reverse of that during output. However, in a conventional operating system, the direction of *control* flow is the same for both input and output: the program plays the active rôle, while the operating system transput primitives are always passive. Thus there are *four* primitive transput operations, not two: the corresponding pairs are passive input and active output, and active input and passive output. This paper explores the implications of this idea in the context of an object oriented operating system.

This work is supported in part by the National Science Foundation under Grant No. MCS-8004111. Computing equipment and technical support are provided in part under a cooperative research agreement with Digital Equipment Corporation.

0. Introduction

In most operating systems the primitives for transput (i.e. input and output) appear as system calls. Programs almost always take the initiative in interactions with the system. The most notable exception to this generalisation is that usually there exists some kind of primitive interrupt facility whereby the operating system kernel can notify a program that a certain event has occurred.

The Eden system currently under construction at the University of Washington is radically different from this norm. In Eden it is quite usual for one program to ask another for a service, *via* a mechanism called *invocation*. This design naturally leads to a system in which most services are provided by "programs" rather than by the system itself, and each program is a provider as well as a consumer of services.

One of the consequences of this design is that each program is prepared to receive invocations as well as to send them. Communication with the outside world is no longer the perogative of the program; the "outside world" is able to take the initiative in communication. This capability allows a transput system for Eden to be built in a rather novel way, which this paper explores. However, before continuing it is necessary to provide some background about Eden itself.

1. The Eden System

The Eden Project [11] (currently in its third year) is a five-year experiment in the design, construction and use of an integrated, distributed computing environment.

The distribution aspects of Eden are not particularly relevant to this paper. The significant aspect of Eden is that it is usual for programs both to provide and consume services. Using the term object in a sense very similar to that of the Smalltalk programming language [5], we refer to Eden as an "object oriented system".

To distinguish our particular flavour of object from that of other systems and languages, we refer to them as Ejects (for *E*den Ob*jects*). An Eject has the following characteristics.

- Each Eject has a unique unforgeable identifier (*UID*); one Eject may communicate with another only by knowing its UID. It is not necessary to know the physical location of an Eject within the Eden system.
- Ejects may receive and reply to *invocations* from other Ejects. An invocation is a request to perform some named operation, and may be thought of as a kind of remote procedure call.
- Each Eject has a concrete *type*, that is, a fixed piece of code that defines the set of invocations to which the Eject will respond. Eden types are similar to the collection of methods that make up a Smalltalk Class.
- An Eject may perform a *Checkpoint* operation. The effect of Checkpointing is to create a *Passive Representation*, a data structure designed to be durable across system crashes. The data in a passive representation should be sufficient to enable the Eject they represent to re-construct itself in a consistent state. The checkpoint primitive is the only mechanism provided by the Eden kernel whereby an Eject may access "stable storage" (i.e. the disk).
- Each Eject has its own thread of control and may be thought of as active at all times. The sending of an invocation does not suspend the execution of the sending Eject: the sender is free to perform other tasks. The programming language used within Eden is an extension of Concurrent Euclid [8], [9], and encourages such a programming style. Each Eject

is provided with multiple processes, of which some may be waiting for incoming invocations, some may be waiting for replies to invocations, and some may be running. This is in contrast to the Smalltalk language, where the act of sending a message transfers control to the receiver.

In practice, Ejects are not always active, either because they (or their computers) have crashed, or because they have explicitly deactivated themselves. However, if a passive eject is sent an invocation, the Eden kernel will activate it. When an Eject is activated by the kernel it will normally attempt to put its internal data structures into a consistent state. If the Eject had previously Checkpointed, it can use the data in its Passive Representation to define this state.

Ejects and invocations are the only entities in the Eden system. Eden is obviously well-suited to the server model of computation, where progress is made by one Eject requesting another to perform some service. For example, the interface to a data-base system could be represented by an Eject which responds to invocations of the form "List the records that match the following pattern." What is not immediately clear is how conventional operating system services like a filing system and redirectable device independent transput fit into the Eden model of computation. These topics are explored in the next section.

2. Files and Transput in Eden

In Eden, files are Ejects: they are active rather than passive entities. An Eden file would itself be able to respond to open, close, read and write invocations rather than being a mere data structure acted upon by operating system primitives. Once a file has been written, the data is committed to stable storage by Checkpointing. Management of the underlying storage medium is performed by the Eden kernel, not by the filing system itself.

Once a file has been created, it is usual to enter it into some directory and associate a meaningful string with that entry, so that the information contained in the file can be conveniently accessed. In Eden directories are also Ejects; they respond to invocations like *Lookup*, *DeleteEntry*, *AddEntry* and *List*. Each entry in a directory Eject is in principle a pair consisting of a mnemonic lookup string and the Unique Identifier of the Eject. It is, of course, possible to enter the UID of *any* Eject in a directory, so arbitrary networks of directories can be constructed.

From the point of view of an Eject trying to perform a Lookup operation, any Eject which responds in the appropriate way is a satisfactory directory. For example, it is possible to provide a *Directory Concatenator* type which is initialised with a list of directories and which yields the same result as would be obtained from performing the lookup on all of the directories in turn until the name is found. Such a concatenator provides a facility rather like that offered by the Unix® shell and the PATH environment variable. It may be implemented either by actually performing the multiple lookups, or by maintaining some sort of table which represents the concatenation of the directories.

There are thus two notions of "type" in Eden. The ***behaviour*** of an Eject is the only aspect that is important to its users. The Eden type of the Eject, i.e. the identity of the particular piece of type-code which defines that behaviour, is irrelevant. Each Eject may be thought of as an abstract machine. The type-code of the Eject defines the transitions of the machine; the inputs are the invocations it receives, and the outputs are the replies to those invocations. Since this pattern of invocation and reply is all that other entities can observe about the Eject, all Ejects with equivalent state machines provide the same functionality. Because many pieces of code can define the same transitions, it is quite possible for several distinct Eden types to behave in the same way. In such a case the Eden types provide alternative implementations of the same abstract machine.

The notion of behavioural compatibility can be further extended. If a client Eject E assumes that some server Eject behaves according to an abstract specification S, then not only will E be satisfied by any implementation of S, but also by any implementation of S', where S' is a superset of S. In other words, provided that S' contains all the operations of S and that their semantics are the same, it does not matter to E that S' contains other operations in addition.

A tree of abstract machines similar to the above can be constructed with Simula Classes [2] and Smalltalk Objects [5]. Observe, however, that the behaviour of a given Eden type may include that of more than one other type, so the situation in Eden is more general than in these languages. In fact, in some ways it resembles Smalltalk with a multiple class inheritance hierarchy [3]. However, our implementation does not currently enforce recompilation when inherited code is changed.

3. Filters and Pipes

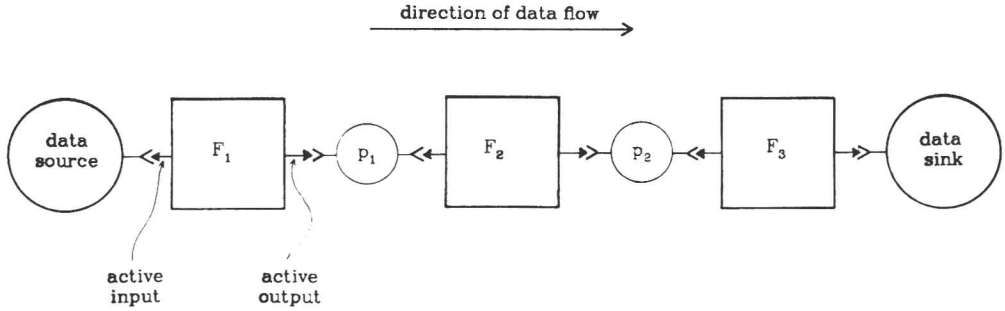
A large number of utilities in a typical operating system may be described as filters. A filter is a program which takes a single stream of input and produces a single stream of output; the output is some transformation of the input. A simple example of a filter is a program whose output is a copy of its input except that all lines beginning with "C" have been omitted. Such a filter might be used to strip comment lines from a Fortran program. Most filters may be parameterised: a more useful program is one which deletes all lines matching a pattern given as an argument. Text formatters, stream editors, spelling checkers, prettyprinters and paginators are all filters.

In a conventional operating system, a filter F performs two functions. In addition to applying a transformation to the data stream, it acts as a ***data pump***, that is, it causes data to flow from the input to the output. The pumping function arises because both input and output are performed ***actively***. By this I mean that F takes the initiative in both input and output; it is F which calls the *Read* and *Write* operations. The rôle of the operating system is merely to respond to the requests made by the filter. If F calls a *Read* operation, the response of the operating system is in some sense a kind of output, because data flows from the system to F. However, the system does not itself call a *Write* operation: it responds to the *Read* that is already in progress. I will call this response ***passive output***. The adjective ***passive*** indicates that the operating system is responding to an initiative of F's; passive output is by definition the complement of active input. In general, data will flow from entity A to entity B if B performs active input and A responds with passive output. Because they communicate with each other I will refer to active input and passive output as ***corresponding*** operations.

When F performs active output, the response from the operating system is ***passive input***. Thus data can also flow from entity A to entity B if A performs active output and B responds with passive input. Passive input and active output are also corresponding operations.

One very useful facility provided by the Unix operating system is the ability to connect filters F_1, F_2, ..., F_n together so that the output of F_i becomes the input of F_{i+1}. This is done by interposing an entity called a ***pipe***

® Unix is a trademark of Bell Laboratories.



F_1, F_2 and F_3 are filters. The shape of the connectors on the filters indicate that they are performing active input and active output. The circles represent facilities provided by the Unix kernel. p_1 and p_2 are pipes; *data source* and *data sink* may be files or devices.

Figure 1: A Pipeline in Unix.

between F_i and F_{i+1}; Unix refers to the whole arrangement as an n-stage *pipeline*. The function of a pipe is to perform passive transput in response to the active transput operations of the filters. When F_i performs a *Write* operation, the pipe to which it is connected responds by accepting the data, i.e. it performs passive input. When F_{i+1} performs a *Read* operation, the pipe responds by supplying data it has previously received from F_i, i.e. the pipe performs passive output (see Figure 1). Because entities like Unix pipes perform both buffering and passive transput, I will refer to them as *passive buffers*.

It should now be clear why passive buffers are necessary. Even though filter F_i performs active output, and filter F_{i+1} performs active input, they cannot be connected directly because these operations are not complementary. The passive buffer provides the active transput operations with the necessary correspondents. In a conventional operating system, the only transput operations made available to user programs are the active ones. The passive transput operations are always performed by the system itself.

In Eden the invocation of the read or write operation of some other Eject represents an active transput operation. Responding to such an invocation is a passive transput operation. All four operations are thus available to any Eject. As was observed above, data can be made to flow from one entity to another using only two of the operations, provided that they form a corresponding pair. Thus data can moved from Eject A to Eject B either by A initiating a *Write* invocation to which B responds, or by B initiating a *Read* invocation to which A responds. It thus seems to be possible to construct a transput system in which there is no active output, just passive output and active input. In other words, the write primitive is apparently unnecessary.

It is interesting to compare this implementation with input and output in Hoare's CSP [7] and in Browning's Tree Machine Notation [4]. In these languages transput occurs when one process executes an output (*!*) operation and its correspondent executes an input (?) operation. This interaction may be regarded in several different ways. Both *!* and ? may be regarded as active, and the (software or hardware) interpreter as the passive connection which transfers data from one to the other. Alternatively, input may be regarded as active ("get me data!") and output as passive ("wait until I am asked for data"). The converse interpretation is also possible: input may be regarded as a passive wait for data, and output as the active operation which generates data. This last interpretation corresponds to Hoare's decision to allow input commands in guards but to exclude output commands.

4. Programming with Read-Only Transput

It is worthwhile considering just how a transput system without active output be constructed and used. I will refer to such an arrangement as a "read only" transput system.

Output devices such as terminals and printers would provide a potentially infinite supply of *Read* invocations. Connecting a terminal to a filter Eject would be rather like starting a pump; it would suck data through the filter and generate a partial vacuum (in the form of outstanding read invocations) on the far side. A file opened for input would respond to read invocations with the appropriate data, and eventually with an indication that the end of the file had been reached. A file opened for output would immediately issue a *Read* invocation, and would continue reading until it received an end of file indicator. It is possible to create pipelines of arbitrary length without any need for intermediate buffering; the only requirement is that each pipeline must start with a data source and end with a data sink.

As should be apparent from the discussion of Eden types, any Eject which responds to *Read* invocations is by definition a source, and any Eject which generates them is a sink. The null sink is an Eject which reads indiscriminately and ignores the data it is given. An Eject which responds to a read invocation by returning the current date and time is a source. Eden Directories also behave as sources; in addition to *Lookup* and *DeleteEntry*, they respond to an invocation called *List*. The effect of a *List* invocation is to prepare the directory to receive a number of *Read* invocations, which transfer a printable representation of the directory's contents to the reader.

There is a certain similarity between a transput system constructed in this way and a lazy implementation of Lisp [6]. In both cases no computation need be done until the result is requested. There is, of course, a difference in the origin of the laziness; in the case of an applicative language it is designed into the implementation, whereas in the case of the transput system each Eject may be programmed so as not to do any work until it is asked for output. A consequence of this is that the filter Ejects are pure transformers: they do not also

pump data (unlike Unix programs). No data flows until a sink is connected to the pipeline.

Laziness, however, is not desirable in a system which permits parallel execution. Instead, one would prefer that each Eject does a certain amount of computation in advance, in anticipation that it will eventually be asked for the fruits of it's labours. Typically, each Eject in a pipeline should read some input and buffer-up some output, and then suspend processing pending a request for output. In this way all the Ejects in a pipeline can run concurrently.

The interconnexion of the elements of the pipeline is easily accomplished in Eden. A filter is initialised by an invocation which supplies it with arguments. Most of these arguments parameterise the behaviour of the filter in the usual way, but one of them is the Unique Identifier of the Eject from which it is to obtain its input. Note that it is not necessary to tell a filter where the output is to go: it will be sent to whatever Eject requests it (by performing a *Read*). A file could be printed simply by requesting the printer server to read from the file. If a paginated listing were required, the printer server would be requested to read from the paginator, and the paginator to read from the file. Since files are active entities, there is no distinction between input redirection from a file and from a program. (This is not so in Unix, for example, where the shell uses different syntax and a different implementation in the two cases.)

One advantage of the "read only" system just outlined is that a sequence of n filters, a source and a sink can all be implemented by $n+2$ Ejects. This means that only $n+1$ invocations are needed to transfer a datum from one end of the pipeline to the other. Conversely, if each filter were to perform active output as well as active input, $2n+2$ invocations would be needed, as would $n+1$ passive buffer Ejects. Thus considerable savings of communications overhead and process switching can be realised with long pipelines. Figure 2 illustrates the same pipeline as Figure 1, but constructed according to the "read only" model.

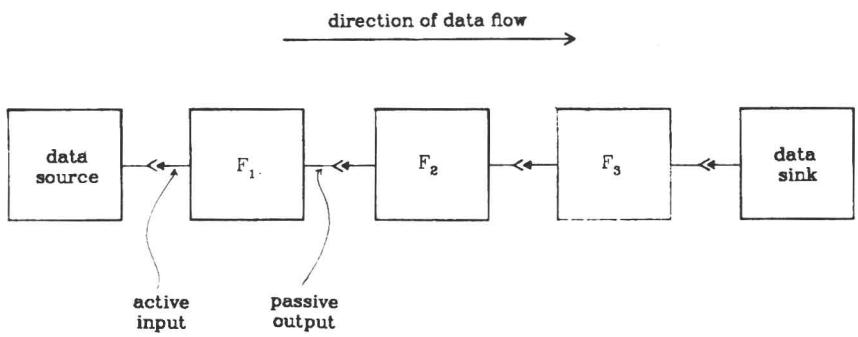
One way of visualising the origin of these savings is as a merging of each passive buffer with its source. In doing this merge, two Ejects are turned into one, and the inter-Eject communication link between them is turned into internal communication. Without any further refinement, this implies that the filter must be written so that it looks for incoming *Read* invocations pending from other Ejects instead of performing write operations.

It is possible to adopt a more conventional style of programming by adding an extra process to the filter. The standard IO module obtained from a library would implement the usual *Write* operations that put characters into a buffer. However, that buffer would be shared with a process that receives invocations which request data and services them. The filter process itself would be programmed in the conventional way and make use of the *Write* operations whenever necessary.

In some sense, then, the cost of "read only" transput is that the programmer (or her language implementor) is given the burden of providing the processes and communication primitives that are no longer necessary at the system level. Is this good or bad? Answering this question requires more experience with "read only" transput than we currently have, but the following observations are relevant.

- The programming language used in the construction of Ejects needs to support parallelism regardless of the transput protocol. An Eject which provides a set of services to clients will typically be organised as a "coordinator" process that receives incoming invocations, and a number of "worker" processes that actually perform the processing necessary to satisfy them.† The use of a separate process to service read requests from the next stage of the pipeline is only a special case of a more general programming methodology.
- Processes provided within the programming language are likely to be more efficient than the processes of the underlying machine or system on which the Ejects are based. Similarly, interprocess communication within an Eject is likely to be much more efficient than invocation.
- By eliminating active output and passive input from the system (at the level of inter-Eject interfaces, if not internally to the Ejects), a considerable simplification of Eject interfaces has been achieved.
- In comparison with the obvious design incorporating passive buffers between each pair of active Ejects, roughly half as many invocations are required to move data from one end of the pipeline to the other. The cost of an invocation must inevitably be higher than that of a system call in an ordinary operating

† Such an organisation is described in [11], where the Eden kernel was assigned responsibility for its maintenance. Our current implementation provides processes at the language level; see [1]



Each box represents an Eject. The filters F_i all perform active input and passive output. The sink actively inputs and the source passively outputs.

Figure 2: The same Pipeline in Eden with "read only" Transput.

system (because invocation is location-independent), so such saving may be significant in Eden.

5. Write-Only Transput; Multiple Inputs and Outputs

The system described so far uses active input and passive output as its only transput primitives. The dual arrangement should also be considered; in this case only passive input and active output would be available. Data sources would continually attempt to perform write invocations, and sinks would always be ready to accept them. An Eject would explicitly send data to the next Eject in a pipeline, but would not in general be concerned with the origin of the data it processed. Within an Eject, a conventional *Read* routine could be implemented by extracting data from an internal buffer; another process would respond to incoming *Write* invocations and use the data thus obtained to fill the same buffer.

Because the "read only" and "write only" models are exact duals, both are equally convenient in the case of a pipeline of pure filters. The differences between the models become apparent when we start to relax the assumptions that introduced this discussion. One assumption that must be examined is that pure filters occur frequently amongst the utilities of the average operating system. In fact it is very common for filters to be impure: many useful programs require multiple inputs or generate multiple outputs. Examples of programs with multiple inputs include file comparison programs and stream editors that have a command input as well as a text input. It is also common for a program to produce a stream of *Reports* (i.e. monitoring messages) in addition to its main output stream.

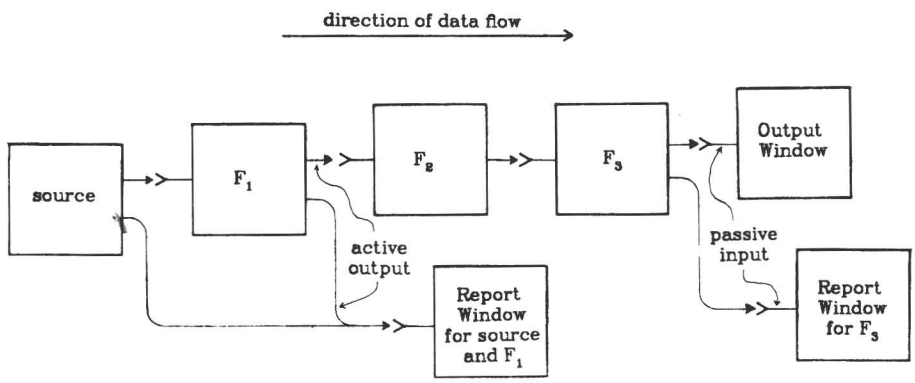
In the "read only" transput scheme the filter Eject F knows the Unique Identifier of the Eject from which it requests input data. Because of this feature it is easy to generalise the "read only" scheme to allow an arbitrary number of inputs. If F needs n inputs, it maintains n UIDs, each referring to an Eject which responds to read requests. In contrast, it is difficult to have multiple outputs with the "read only" scheme, because output occurs only in response to an external request. Arranging for two or more Ejects to make *Read* invocations on F does not help: F cannot distinguish this from one Eject making the same total number of *Read* invocations. As we have described it so far, "read only" transput allows arbitrary fan-in but no fan-out.

The dual situation exits with "write only" transput. Each filter has (or appears to have) a single source, but can direct output to as many sinks as is convenient. There is arbitrary fan-out, but no fan-in. Conventional transput allows arbitrary fan-in and fan-out because both reads and writes are active. (However, some operating systems place restrictions on the number of streams which may be redirected.)

One might attempt to remedy this failing by permitting F to examine the UID of the originator of the request; however, this introduces more problems than it solves. Although these UIDs are present in the invocation message (so that the reply may be returned correctly) they are in principle private to the Eden kernel. This is because the effect of a particular invocation ought to depend only its parameters, and not on the identity of the invoker. Doing otherwise would prohibit dynamic re-direction of transput streams. A parallel may be drawn with programming languages: the effect of a particular procedure call should not depend on who makes it. Even though the return address is on the execution stack and could easily be accessed, procedural programming languages do not provide a primitive to do so. The semantics of procedure call would be greatly complicated by such a provision.

Let us consider how multiple outputs may be accommodated within the "read only" model. One possibility is to designate one output stream as the "primary" output, and make all the others "secondary". Primary output is supplied in response to *Read* invocations in the way previously discussed, but now secondary output is volunteered in *Write* invocations. When such impure filters are initialised, they must be informed of the destination of their secondary outputs. Typically these outputs will be directed into passive buffers, which will then be sources for other pipelines. This amounts to abandoning the "read only" nature of the transput system for all filters with multiple outputs — and a large number of filters produce reports.

On the assumption that more filters have multiple outputs than multiple inputs, the dual arrangement may be preferable. In a "write only" transput system each filter would have a primary input, which is supplied by a source Ejects performing *Write* invocations, and a number of secondary inputs, which are actively read. These secondary inputs will typically be passive buffers, filled by the active output of some pipeline, file or device. Multiple outputs present no difficulty; Figure 3 shows a



Once again, each box represents an Eject. The source, F_1 and F_3 produce reports as well as normal output. The reports from source and F_1 are directed to a common destination, perhaps a window on a display.

Figure 3: An Eden pipeline in the write-only discipline, with Report Streams