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

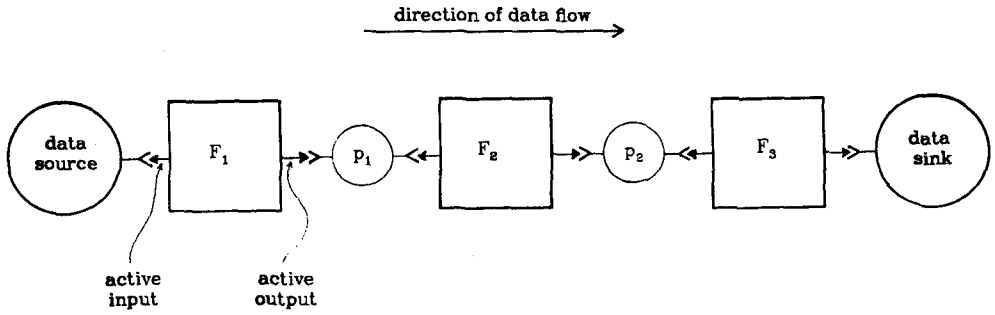
A large number of utilities in a typical operating system may be described as filters. A filter is a program which takes a single stream of input and produces a single stream of output; the output is some transformation of the input. A simple example of a filter is a program whose output is a copy of 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.

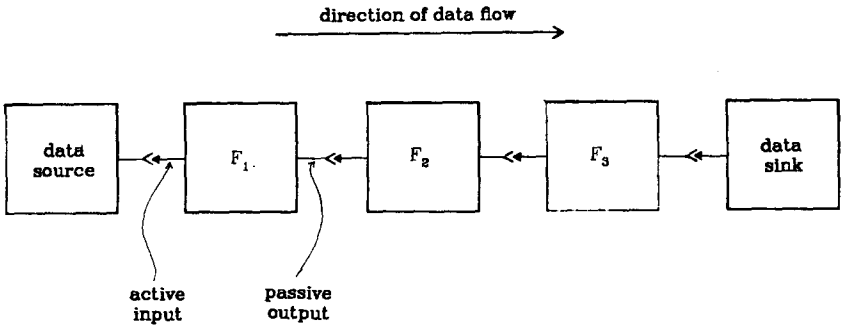
One way of visualising the origin of these savings is as a merging of each passive buffer with its source. In doing this merge, two Ejects are turned into one, and the inter-Eject communication link between them is turned into internal communication. Without any further refinement, this implies that 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.

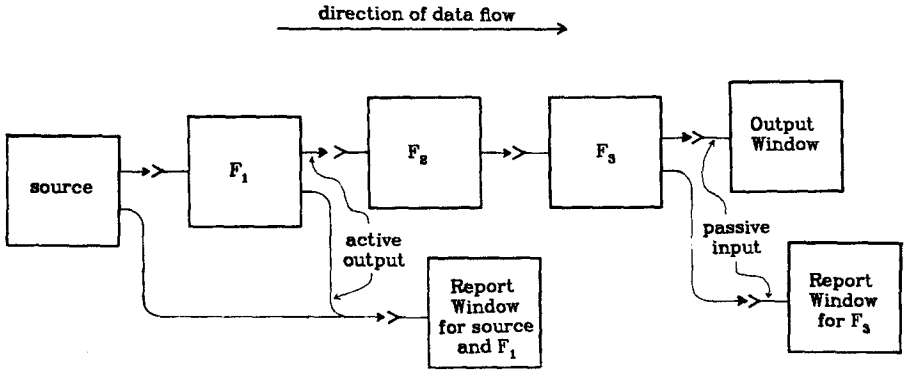
In the "read only" transput scheme the filter Eject F knows the Unique Identifier of the Eject from which it requests input data. Because of this feature it is easy to generalise the "read only" scheme to allow an arbitrary number of inputs. If F needs n inputs, it maintains n UIDs, each referring to 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

possible configuration.

Neither of these solutions is very satisfactory, as each involves re-introducing passive buffers and the other kind of active transput primitive. A better solution is to admit the existence of multiple inputs and outputs explicitly. In the "read only" model, a *channel identifier* is associated with each output stream, and each *Read* invocation is qualified by the appropriate identifier. For example, the specification of a filter *F* might state that it will respond to *Read* requests on channels *Report* and *Output*. When connecting sink Ejects to *F*, the sinks must be told not only *F*'s UID but also the channel identifier that should be used on each request. Figure 4 shows the same set of interconnections as Figure 3, but uses the "read only" discipline and channel identifiers.

The major disadvantage of this scheme is that the user who connects filters together now has the added burden of supplying the correct channel identifiers. However, this is very similar to the way transput is redirected in a conventional operating system, where the command language provides some primitive like ASSIGN OUTPUT CHANNEL *name* TO *file*, or like the Unix shell's "*n*>" syntax. It seems to me that once the user is aware of the existence of multiple channels, the requirement to provide channel identifiers does little to increase the perceived complexity of the system.

Because our channel identifiers are supplied to Ejects (i.e. user code) rather than system code, there is a risk that a dishonest programmer might read from someone else's channel. In other words, if *E* is told to read from *F*'s channel 1, nothing prevents it from reading from *F*'s channel 2 as well. One way of overcoming this problem is to use UIDs as channel identifiers: because UIDs cannot be forged, the only Ejects which are able to make valid *ReadonChannel* requests of *F* are those to which a channel identifier has been given explicitly. The cost of this additional security is that more work is now necessary to connect a sink to its source. Whoever sets up a pipeline must ask each filter for the UIDs of its channels, and then pass them on: UIDs cannot be given in the documentation in the same way as ordinary identifiers. The security of this scheme thus depends on the honesty of the Eject which performs the interconnections; in the last resort, a user can always convince himself of this by writing such an Eject himself. It does not depend on the honesty of the "system utility" Ejects. This is fortunate, because it is unreasonable for the user to have to rewrite all the utilities.†

If Eden addressed its messages to "Ports" rather than Ejects, arbitrary fan-out would be easy to achieve; there would simply be one port for each output channel. Using capabilities as channel identifiers may be regarded as a way of simulating multiple ports.

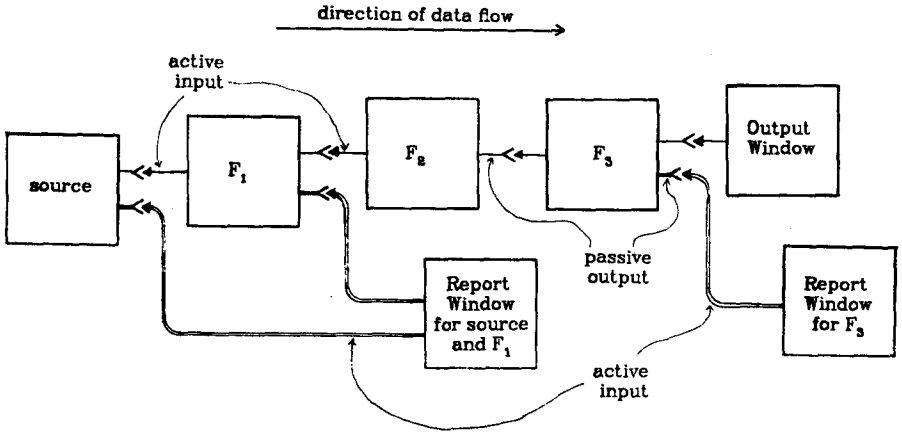
6. The Place of Stream Transput in Eden

The above discussion is solely concerned with input and output according to a stream protocol. In many operating systems, this is the only means whereby processes in different address spaces may communicate. The design of the Unix operating system is based on the assumption that (except for a primitive "software interrupt" facility) all programs communicate by byte-stream. Accordingly, all files are considered to be an unstructured sequence of bytes.

It is well to remember that the Eden System does not make this assumption. Any pair of Ejects which communicate by invocation need to establish a protocol which sets out what each may expect from the other. The Eden transput package is nothing more than such a protocol designed to support the abstraction of a Sequence, together with a collection of library routines which help user Ejects to obey it. Although the Eden transput protocol attempts to be sufficiently general to satisfy the input and output needs of most users, it is not intended to be universal. If two Ejects need to communicate in a way that is difficult or impossible with the transput package, they are free to create their own protocol (and perhaps make it available to other users as a library module). For example, the Transput protocol does not support random access; a disk file Eject (or an Eject with a large main store at its disposal) may wish to define a protocol which supports the abstraction of a Map. Such an Eject may not support the transput protocol at all, or it may support both protocols.

Nothing I have said about Eden transput constrains Eden streams to be streams of bytes. Streams of arbitrary records fit into the protocol just as well, provided only that they are homogeneous. In fact, because the Eden Programming Language lacks type parameterisa-

† This is a characteristic of all capability-based systems, not a special property of "read only" transput. Observe that in a conventional operating system there is nothing to stop an editor, say, from deleting all the files in a user's directory.



The same arrangement as Figure 3, but in the "read only" discipline. The double lines indicate *Read(ReportStream)* requests; the single lines indicate *Read(Output)* requests. It is assumed that the Report Window is designed to read from multiple sources.

Figure 4: The pipeline of Figure 3 in the read-only discipline

tion, it is a little inconvenient to allow streams of arbitrary types; it would be necessary to circumvent the type safety in the language. For a description of an operating system (written and programmed in an untyped language) which provided such streams, see [12].

Because invocation is itself a powerful inter-process communication mechanism, and because arbitrary protocols can be built on top of invocation, it is likely that byte-stream transput will be less important in Eden than in conventional operating systems. Nevertheless, we believe that the provision of a novel object-oriented environment will not by itself be sufficient for Eden to attract users who are familiar with a conventional operating system. Eden must also provide conventional operating system facilities in a way that compares favourably with systems such as Unix. Dynamically redirectable stream transput is an example of one such facility.

7. Current Status

At the time of writing (May 1983), a prototype distributed implementation of Eden is in service. The hardware substrate consists of several VAX processors connected together by 10 Mbit ethernet. The Unix operating system provides a development environment and supplies the underlying address spaces, processes and primitive disk access required to implement the Eden kernel.

We are experimenting with a "read only" transput system that uses integer channel identifiers as described in Section 5. Currently most data of interest is in the Unix file system, so a bootstrap Eden transput system has been constructed. This consists of a "Unix File System" Eject for each physical machine, which responds to two invocations, *NewStream* and *UseStream*. In outline (and ignoring the channel ids), the bootstrap transput system works as follows. *NewStream* takes as input a Unix path name, and returns as its result an Eden stream, i.e. a Capability. The Capability is actually the UID of a newly created Eject (of type *UnixFile*), whose purpose is to respond to *Transfer* invocations with the contents of the appropriate Unix file. When the user closes the stream, the *UnixFile* Eject deactivates itself and, since it has never Checkpointed, disappears. *UseStream* does the opposite; it takes as input a Unix path name and a Capability for a stream, and creates a *UnixFile* Eject which repeatedly invokes *Transfer* on the capability and records the data it receives. When an end of stream status is returned by *Transfer*, the appropriate Unix file is opened, written and closed.

The preliminary design for the full Eden file system incorporates nested transactions and atomic updates [10]. The implementation of a subset which excludes transactions is underway. The File system *Directory* type is in service, and supports the stream protocol as described in Section 2.

8. Conclusion

Four distinct transput primitives have been identified: passive input, active output, active input, and passive output. Passive input corresponds with active output, and active input corresponds with passive output. In a conventional operating system all these primitives exist, but the passive ones are not available to users: they are the responses of the system code which implements the transput system.

In an object oriented system such as Eden, all four primitives can be made available at the user level. Indeed, since users are able to generate whatever invocations they desire, there is no way of concealing any particular subset. Nevertheless, the corresponding pair passive output and active input with channel identifiers seem to be adequate for most applications of stream transput. Adopting such a "read only" convention reduces complexity and improves implementation efficiency.

Redirection of input and output can be provided very naturally in a system where each entity is referred to by means of a unique identifier. Special file or stream descriptors are not needed.

Programs which do not naturally fall into the byte-stream model exist in all operating systems. It is usually possible to coerce them into that mold if one tries hard enough, and generalises the stream model sufficiently. This need not happen in Eden, because stream transput is just a special use of the underlying invocation mechanism. Applications which do not fit this special case need not be distorted: they are free to use some other invocation protocol.

Acknowledgements

The Eden project currently involves about four faculty members, fifteen students and five staff members. Without the cooperation of these people there would be no Eden system on which to build the Transput protocol described in this paper. In particular, Jordan Brower has contributed to the evolution of these ideas by implementing the first Ejects which conform to the asymmetric stream protocol.

References

[1] Almes, G. A. *The Evolution of the Eden Invocation Mechanism.* Technical Report 83-01-03, Department of Computer Science, University of Washington. January 1983.

[2] Birtwhistle, G. M., Dahl, O-J., Myhrhaug, B., and Nygaard, K. *Simula BEGIN.* Auerbach, 1973.

[3] Borning, A. H. and Ingalls, D. H. *Multiple Inheritance in Smalltalk-80.* Technical Report 82-06-02, Department of Computer Science, University of Washington. June 1982.

[4] Browning, S. A. *The Tree Machine: A Highly Concurrent Computing Environment.* Technical Report (Ph. D. Thesis) Computer Science, California Institute of Technology. January 1980.

[5] Goldberg, A. J. and Robson, D. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[6] Henderson, P. *Functional Programming: Application and Implementation.* Prentice Hall, 1980.

[7] Hoare, C. A. R. *Communicating Sequential Processes.* Comm ACM, Vol 21 Nr 8 (August 1978) pp 666-677.

[8] Holt, R. C. *A Short Introduction to Concurrent Euclid.* SIGPLAN Notices Vol 17 Nr 5 (May 1982) pp 60-79.

[9] Holt, R. C. *Concurrent Euclid, The Unix System, and Tunis.* Addison-Wesley, 1983.

[10] Jessop, W. H., Noe, J. D., Jacobson, D. M., Baer, J-L. and Pu, C. *The Eden Transaction-Based File System.* Procs. 2nd Symp. Reliability in Distributed Software and Database Systems. Pittsburgh, PA. (July 1982).

[11] Lazowska, E. D., Levy, H. M., Almes, G. T., Fischer, M. J., Fowler, R. J. and Vestal, S.C. *The Architecture of the Eden System.* Procs. 8th Symp. Op. Sys. Principles. Asilomar, CA. pp 148-159 (December 1981).

[12] Stoy, J. E. and Strachey, C. *OS6: An Experimental Operating System for a Small Computer. Part 2: Input/output and filing system.* Comp. J, Vol 15 Nr 3 (August 1972) pp 195-203.