Port directed communication*

A. Silberschatz

Department of Computer Sciences, The University of Texas, Austin, Texas 78712, USA

In his paper, Communicating Sequential Processes, Hoare has introduced a clean concept of a process, in which the states of processes are isolated from one another, except at explicit communication points. Communication and synchronisation in this system is accomplished through the I/O facility and Dijkstra's guarded command. Hoare insisted in his proposal that every input and output command must name its source and destination explicity. In this paper we propose an alternative to Hoare's scheme. We suggest that communication and synchronisation should be handled through port names. We define the notion of ownership of a port, and require that each port will have a single owner and as many users as needed. These concepts provide the means for:
(*a*) clean handling of subroutines
(*b*) allowing both input and output commands to appear in guards
(*c*) obtaining an efficient implementation of the communication and synchronisation constructs.
These issues and concepts will be dealt with in this paper, and their usefulness will be demonstrated providing solutions to a variety of familiar programming exercises.

(Received October 1979)

1. Introduction

In his paper Communicating Sequential Processes (CSP), Hoare (1978) has introduced a language concept for concurrent processing which is suitable for a microcomputer network environment with distributed storage. As with any new concept, there are several aspects that seem to deserve further explanation and discussion. The purpose of this paper is to examine Hoare's proposal in greater detail with a view toward gaining a better understanding as to how communication and synchronisation should be handled in distributed systems.

We start by presenting a brief survey of the essential features of CSP. Central to the language are the following concepts:

1. A CSP program consists of a fixed number of sequential processes, that are started simultaneously. A program terminates when each of its processes reaches the end of its execution.
2. Each sequential process contains a private data structure and a list of commands (instructions) to manipulate the data. One process cannot operate on the data of another process.
3. Communication and synchronisation are accomplished through the Input/Output facility. In order to exchange messages, the source and destination processes must name each other explicitly.
4. The sequential control structures are based on Dijkstra's guarded commands (Dijkstra, 1975). A guard may include a single input command; output commands may not appear in guards.

These features allow one to specify systems that are composed of a finite number of disjoint processes which are driven by their inputs and synchronised by their outputs.

Hoare emphasised in his paper that his proposal is at best only a partial solution to the problem of finding an appropriate language for concurrent programming. In particular, Hoare noted that the following topics need further study and development:

(*a*) the question of relaxing the requirement for explicit naming (e.g. that every input or output command must name its source and destination explicitly)

(*b*) the question of permitting output commands to appear in guards

(*c*) the question of efficient implementation of his constructs.

Some of these issues have recently been addressed in the literature. The problem of allowing output commands to appear in guards and the question of an efficient implementation were considered in Silberschatz (1979). The approach taken there was to impose a strict order among each pair of communicating processes. Thus for each such pair, it was required that one process be the master of the other (or conversely, one process be the slave of the other). This replaces the asymmetry of the I/O constructs by an asymmetry between the various processes. With this approach it is possible to obtain a simple deadlock free implementation of the I/O commands. Moreover it allows both input and output commands to appear in guards with the restriction that a slave process can have I/O commands in guards which involve only those processes that are his masters. This scheme suffers from the fact that there is no complete symmetry between the various processes. It also does not address the issue of the explicit naming requirement.

Related work was performed in connection with the development of the US Department of Defense high level language (1977). The preliminary Green language, which was influenced by CSP, attempted to resolve the difficulties associated with the explicit naming requirement by making naming one sided only. Tasks in that language were characterised as services and users. Communication between these tasks was carried out through *boxes*. A user of a service task was required to specify the name of the task (as well as the name of a box) whenever he wished to communicate with that task. On the other hand, the service task was required only to mention the name of a box in order to communicate with its users. Although this strategy handles issue (*a*) above, it does not address the question of permitting output commands to appear in guards (issue (*b*)). This shortcoming has led to the development of the *entry* concept in the final version of the language Green (Ada reference manual, 1979). Entries are structurally similar to procedures; they accept parameters and are called in the same manner. The main difference lies in the way calls are internally handled. Thus the input and output primitives of CSP have been replaced by a much higher level primitive. A similar approach has been taken by Brinch Hansen (1978) in his work on distributed processes.

In this paper we wish to examine the three issues posed above by sticking as closely as possible to the original CSP constructs. In Section 2 we present another approach for handling com-

*This research was supported in part by the National Science Foundation under Grant No. MCS 7702463, and in part by the Office of Naval Research under Contract N 00014-80-C-0592.

CCC-0010-4620/81/0024-0078 $02.50

munication and synchronisation by introducing the concept of a *communication port*. In Sections 3 and 4 we provide illustrative examples and further developments. Finally, in Section 5 we outline a general method for implementing our proposed constructs.

2. Communication ports

An alternative to Hoare's explicit naming requirement would be to name *ports* through which communication is to take place. The port names would be local to the system processes, and the manner in which processes are to be connected to the ports would be declared locally in each individual process. The concept of ports as a communication mechanism is not new (Balzer, 1971; Walden, 1972). What is new about our proposal is that it pertains specifically to Hoare's work. Moreover, it has a different semantic interpretation.

Each CSP process can declare in its address space a set of port names. This declaration is accomplished via the statement:

⟨list of port names⟩: *port*;

For example, a process P may declare

A, B, C: *port*;

We will say that process P is the *owner* of the ports A, B and C. The notion of ownership of a port will play an important role in our discussion concerning synchronisation and communication.

A process can use a port that it does not own, by declaring:

use (⟨list of port names⟩);

For example, a process Q may declare

use (A, C);

This declaration specifies that process Q may communicate with process P through the ports A and C. We will denote this fact by saying that Q is the *user* of these two ports.

These concepts can be illustrated via the use of directed graphs. Circles correspond to processes, squares correspond to ports, and arcs correspond to user/owner relationship. Thus an arc from process Q to a port A indicates that process Q is the user of the port A. Similarly, an arc from port A to process P indicates that process P is the owner of port A. In general, a port has one and only one owner and several users. Thus the general graph (called *port graph*) corresponding to one port is as shown in **Fig. 1.**

It should be pointed out that in order for this scheme to work all port names need to be distinct. This is necessary so that no confusion will arise when a process declares its intention for the use of ports. An alternative is to extend the *use* primitive to allow the inclusion of owner names with each port name defined in the use list. Since this is semantically equivalent to the present proposal, we will restrict our attention in this paper to the case where all port names are distinct.

Thus far, we have described the manner by which processes can be connected to a common port. We now describe how communication can take place through ports. Suppose that two processes wish to communicate. In order to accomplish this the processes need to be connected to a common port. Moreover, one of these processes needs to be the owner of that port. For example, in Fig. 1, the only communication that can take place through port A is between the n pairs of processes P and Q_i. Through port A, Q_i *cannot* communicate with Q_j.

As with Hoare's proposal, communication through ports is accomplished through the input and output commands. The only difference is that port names are used instead of process names. The input command thus has the form:

⟨port-name⟩?⟨target variables⟩

while the output command has the form:

⟨port-name⟩!⟨expression⟩

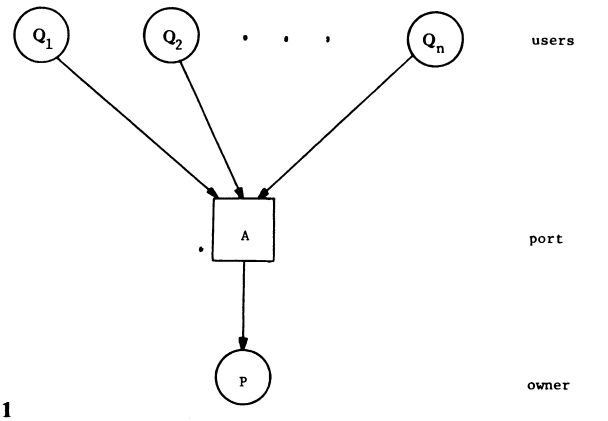

Fig. 1

There are two main semantic differences between Hoare's proposal and ours. First, in Hoare's proposal communication occurs when one process names another as destination for output and the second process names the first as source for input. In our proposal communication occurs only when:

(*a*) two processes name the same port in their I/O commands

(*b*) one of these processes performs an input while the other output

(*c*) one of these processes is the owner of the port.

In both Hoare's and our proposal we require that target variables of the input command should match the value denoted by the expression of the output command.

The second semantic difference is that in Hoare's proposal an I/O command specifies exactly one communicating partner while in our proposal, an I/O command may involve several processes; however, only one of them will be selected for communication. More precisely, suppose that processes Q_1, Q_5 and Q_7 of Fig. 1 are ready to do I/O (e.g. they have each invoked an I/O command involving port A). Further suppose that process P (the owner of port A) has also invoked an I/O command which matches the ones of Q_1 and Q_7. In this case communication occurs between either the pair (P, Q_1) or the pair (P, Q_7) but *not both*. The choice as to which pair will be selected is not known. This is another means by which non-determinism is introduced in our proposal (the other one being Dijkstra's guarded command (Dijkstra, 1975) to be discussed below).

In contrast to Hoare's proposal we allow both input and output commands to appear in guards. This however is restricted to the case where a process may have I/O commands in guards, but these commands may involve only those ports that it *owns*. The reason for this restriction is that it allows one to obtain a uniform efficient implementation of the I/O commands (see Section 5). We note however, that our proposal is as powerful as the one presented by Hoare, since for each pair of communicating processes, say P_1 and P_2, one can declare ports A_1 and A_2 such that P_1 is the owner of A_1 and P_2 is the owner of A_2.Communication between P_1 and P_2 can thus be handled through either A_1 or A_2. We feel however that our scheme allows a more structured approach to the problem of writing concurrent programs because it allows the simplification of a large class of algorithms and reduces the number of I/O commands that are needed in the implementation of these algorithms (see Section 3).

We note that the concepts of port directed communication and of ownership of ports allow us to relax the requirement of explicit naming and permit one to include output commands in guards. Thus we have succeeded in presenting solutions to issues (*a*) and (*b*) posed in the introduction. In Section 5, we

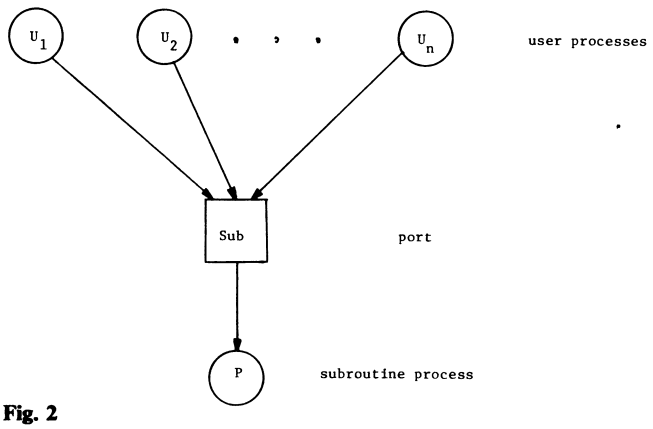

Fig. 2

examine the question of efficient implementation of the I/O command structures (i.e. issue (*c*) posed in the introduction). This is done by presenting a uniform abstract implementation of our constructs.

3. Illustrative examples

In this section we demonstrate the usefulness of our concepts by providing different solutions to some of the programming exercises presented by Hoare (1978). We only discuss those algorithms in which our solutions differ widely from those of Hoare's.

3.1 *Conventional subroutines*

A subroutine in CSP can be implemented as a process that is operating concurrently with its user processes. Parameter passing is done by value/result. Hoare's scheme requires that the subroutine process lists all of its user processes. With our scheme the subroutine process needs only to declare a local port, say Sub. The user processes import (e.g. declare *use* (Sub)) this port and use it in order to pass parameters to and from the subroutine. Thus, the general port graph corresponding to such a scheme is shown in **Fig. 2.** Note that with such a scheme one can write subroutines without the prior knowledge as to which processes will invoke it. This corresponds to the standard way subroutines are handled.

Central to the subroutine process is the repetitive command

*[Sub?(value-parameters) . . . Sub!(result-parameters)]

where . . . computes the results from the value parameters. The user processes invoke the subroutine via:

Sub!(arguments) . . . Sub?(results);

Note that this scheme works only if a user process invokes the subroutine in the above specified sequence. This is because only one port is used for communication, and we have implicitly assumed when designing this algorithm that only one user process at a time is executing Sub?(results). More specifically, all users except one are delayed at Sub!(arguments) and hence at most one process may be delayed at Sub?(results). Thus when Sub!(result-parameters) is executed it matches exactly the one process that successfully executed Sub!(arguments).

It is worthwhile pointing out that with our scheme the decision as to which user process will be served next (e.g. the nondeterminism) is handled by the port mechanism, while in Hoare's scheme this is done via the guarded command mechanism.

3.2 *Restricted subroutines*

There are cases where subroutines are called by result only. However, with Hoare's scheme this could not be effectively handled because Hoare forbids output commands to appear in guards. To illustrate this, consider a subroutine process S with processes U_1 and U_2 its users. Central to S is the repetitive guarded command

$$*[U_1?(\ ) \rightarrow \ldots U_1!(\ )$$
$$[]\ U_2?(\ ) \rightarrow \ldots U_2!(\ )\ ]$$

for handling subroutine calls, where $U_1!(\ )$ and $U_2!(\ )$ are optional depending on whether a result value is expected.

The user processes (e.g. U_1 and U_2) simulate subroutine calls to S by the sequence;

$$S!(\ ) \ldots S?(\ )$$

The problem is that this sequence must be observed even when no output is required from the user process to the simulated subroutine S. In this case S!( ) is nothing more than a pure signal to avoid loss of efficiency (as an example consider the bounded buffer solution of Hoare's). This problem could be remedied if output commands were allowed in guards. With our scheme this is possible.

Central to the subroutine process S is the repetitive command:

*[Sub!(value-result) → . . .]

central to U_1 and U_2 is the command:

Sub?(results)

As an example of this is a subroutine that returns to the calling process some unique value. This value might be a unique number for identification purpose (something that is frequently used in operating systems), or value from a random number generator, current-time value (time stamp (Lamport 1978)), etc.

3.3 *Multiple-entry/exit subroutines*

Multiple-entry/exit subroutines (Hoare, 1972) are handled in our scheme in a manner similar to the one of Hoare's. The main difference is that we use port names rather than process names in the I/O commands. Thus we also achieve here the same advantages as we have obtained with conventional subroutines (as discussed above). As an example, consider a subroutine process S with two entry points:

*[Sub?entry$_1$(value-params) → . . .
[] Sub?entry$_2$(value-params) → . . .]

Where again Sub is a port name declared locally to process S. Note, that in contrast to single entry subroutines nondeterminism here is handled by the port mechanism as well as the guarded command mechanism.

3.4 *Bounded buffer*

A pair of processes Producer and Consumer wish to communicate via a buffering process X, that contains up to ten portions. The Producer process outputs to X, while the Consumer inputs from X.

The process X could be implemented as follows:

```
X::
  Bound: port;
  Buffer: (0 . . 9) portion;
  In, Out: integer;
  In := 0;
  Out := 0;
  *[In < Out + 10; Bound?Buffer (In mod 10) → In :=
                                          In + 1;
  [] [Out < In; Bound!Buffer (Out mod 10) → Out :=
                                          Out +1;
  ]
```

In this example, X is the owner of the port Bound. Therefore, X can have both input and output commands in guards involving that port. In contrast, since Hoare does not allow output commands to appear in guards, he had to introduce the command X!more to be executed by the Consumer process to

signal the fact that it is ready to execute X?p.

It is interesting to note that our solution to the bounded buffer problem is quite analogous to the solution using conventional monitors. This is precisely what Hoare commented on in Section 7.8 of his paper.

3.5 *Integer semaphore*

A general semaphore is to be implemented as a CSP process S. The process S is to be shared among N user processes.

```
S::
  Sem: port
  Val: integer;
  Val := 0;
  *[Sem?V( ) → Val := Val + 1;
  [] Val > 0; Sem?P ( ) → Val := Val − 1
  ]
```

Each user process increments the semaphore by executing Sem!V( ) or decrements it by executing Sem!P( ).

In contrast to Hoare's solution we do not use an array of processes to represent the user processes. In addition, process S need not indicate which process can communicate with it. This corresponds to the general way subroutines/monitors are used.

4. Further development

In this section we discuss several more issues concerning Hoare's proposal and ours.

4.1 *Port restrictions*

There is one more issue concerning ports that is worth discussing here. It concerns the question of providing additional declarative mechanisms to allow the owner of a port to state restrictions on the way the port is used. Such restrictions might be:

(*a*) limit the number of processes that can use the port

(*b*) list the processes that are allowed to use the port

(*c*) restrict the user of the port to either input or output via the port but *not* both

(*d*) restrict the type of messages that can be handled through the port.

Such restriction can be verified to hold at compile time, and thus reduce some runtime overhead. We shall elaborate only on the last of these in this paper.

In many cases one would like to restrict the type of message that can be transferred through a particular port, to a single unique type. This could be accomplished by allowing the owner of a port to declare the message type that can be associated (transferred) through the port. This will eliminate the need for run time checking that the target variables of the input command match the value denoted by the expression of the output command. Instead, the compiler can verify that this restriction is indeed observed. For example, consider the Bounded Buffer problem (Section 3.4). One could declare:

Bound: *port of* portion;

The compiler then can verify that via port Bound one can only input and output variables of type portion.

4.2 *Arrays of Ports*

In some applications the concept of an array of ports seems appropriate. For example, the handling of recursive subroutines and the scheduling of identical server processes among user processes. The problem is that in order to handle these cases effectively one needs to distribute such an array over several owners. In order to achieve this we will allow the declaration of an array of ports to be distributed over several processes. We only require that the same element of such an array will have a unique owner. This can be easily verified by the compiler. Let us illustrate this concept by an example. Suppose that we have 50 identical server processes that are to be allocated to user processes upon demand.

```
Server (i: 1 . . 50)::
                A[i] : port;
                  .
                  .
                  .
                  .
                  .
```

In this case we have declared 50 processes each of which owns one element out of the array of ports A.

User processes may require a service to be performed by one of the server processes. Since all of them are identical any one of them will suffice as long as it is not busy performing the service for another user process. To achieve this, one can provide a Scheduler process that distributes to the users indices to the array A. Thus a user first acquires an index say *i*, from the Scheduler process; it then uses it by communicating with the Server process it has been allocated via the port A(*i*); finally it releases the Server process by calling upon the Scheduler process and returning (outputting) the index *i*. Note that this scheme is quite similar to the conventional resource scheduling scheme with monitors.

5. Implementation notes

In this section we outline a general method for implementing the input and output commands. This will be done by presenting an abstract implementation of these constructs. Our approach here is similar to the one taken in Silberschatz (1979). We restrict our attention here to a simplified version of Hoare's constructs, namely, that processes can exchange only fixed size messages. We also do not discuss here the question as to how information is transferred from one process to another.

When processes wish to communicate, some information about the state of each process must be exchanged in the course of executing an I/O command, in order to determine when and whether communication can take place. A state information exchange may be viewed as a *signal* which carries no information relevant to a particular program, but is needed for synchronisation purposes. When looking for an efficient implementation of the I/O commands one seeks an algorithm that will reduce the number of such synchronisation signals. Moreover, the signals should not result in deadlocks which could not otherwise occur.

In the following we present one such algorithm for our proposed I/O constructs. The approach we take in presenting the abstract implementation will be to discuss these issues in terms of owner and users of a single port, say A. We adopt the rule that a user of a port initiates an I/O command by sending a message (or signal) to the owner of the port requesting the I/O service.

In order to handle communication it is necessary to define certain data structures. These may be implemented as memory locations, hardware registers, hardware buffers, etc. In this paper we describe these structures in terms of Pascal notations (Wirth, 1971). For each locally declared port A, the following data structures are needed so that users of A can notify their intention for communication.

```
var UserA = (⟨list of all users of port A⟩);
    ReadA: array [UserA] of Boolean;
    WriteA: array [UserA] of Boolean;
```

These are declared locally to the process that owns port A.

In addition to these data structures which are declared for each individual port, additional structures are needed for each

individual process, to allow internal buffering and the handling of transfer of messages from one process to another. The following structures are declared locally to each process:

```
var Buffer: Message;
    Flag:   Boolean;
    Pname: Process-name;
```

These data structures are intended for the following use:

(*a*) ReadA (⟨source⟩) is set to the value True by the process ⟨source⟩ whenever that process is ready to output a message to the process that owns port A

(*b*) WriteA (⟨destination⟩) is set to the value True by the process ⟨destination⟩ whenever that process is ready to input a message from the owner of port A

(*c*) Buffer is a variable of type Message that is used by process P for internal buffering, so that communication can take place between it and its communicating processes

(*d*) The variable Flag of process Q is used as follows. Let Q be the user of port A, owned by process P. Flag is set to the value True by process P, to indicate that P has completed an I/O service requested by Q

(*e*) Pname is used in process P to record the name of the process whose I/O request is currently being fulfilled by process P.

These data structures will be used in the implementation of the input and output commands.

The system provides the following primitives for handling the transfer of data:

(*a*) Put (Q, M) ≡ transfer message M to the Buffer of process Q

(*b*) Get (Q, M) ≡ transfer content of Buffer of process Q to M

(*c*) Signal (Q, F) ≡ set the Boolean variable F of process Q to the value True (where F is either the variables Read, Write, or Flag)

(*d*) Wait (F) ≡ wait until value of F is True. The waiting can be carried out by either busy waiting (if the process runs on a dedicated processor), or by process suspension (in case the processor is multiplexed)

(*e*) Select (A, Pname) ≡ A is either the array Read or the array Write. Select is a function procedure that returns the value True if there exists a process-name Q such that A(Q) = True. If this is the case, then in addition Pname = Q. If there are several such Q's then an arbitrary Q is selected to be assigned to Pname.

We emphasise again that for the purpose of this paper we ignore the problem as to how these primitives are handled.

We are in a position now to describe how the input and output commands are handled. We will do so by describing their implementation in terms of an owner and user of a common port A. Let P and Q be a pair of communicating processes such that P is the owner of A, and Q the user of A. The translation of the input and output commands (which do not appear in guards) in P and Q is presented below:

P (owner of port A)	**Q (user of port A)**
A!*a*	*A*?*b*
Wait (Select (WriteA, Pname))	Signal (P, WriteA(Q))
Put (Pname, a)	Wait (Flag)
WriteA(Pname) := false	b := Buffer
Signal (Pname, Flag)	Flag := false
A?*a*	*A*!*b*
Wait (Select (ReadA, Pname))	Buffer := b
Get (Pname, a)	Signal (P, ReadA(Q))
ReadA(Pname) := false	Wait (Flag)
Signal (Pname, Flag)	Flag := false

Note that the owner always waits for the user to initiate I/O.

How are I/O commands handled in guards? Recall that we require that a process can have I/O commands in guards, but these commands may involve only those ports that it owns. This implies that a process executing a guard is always waiting for the initiation of I/O requests in some other process. Thus one can easily extend the translation algorithm described above to the case where I/O commands appear in guards. We do not elaborate on this issue any further here.

We note that our implementation did not take into account the fact that processes may terminate. This can be easily handled by adding another data structure in each process. This data structure will be used by the process to record and remember which of its communicating partners have terminated.

6. Conclusion

We have presented an alternative to Hoare's scheme that required that every input and output command must name its source and destination explicitly. Instead, we suggested the use of port names through which communication is to take place. We have defined the notion of ownership of a port and have shown that with this notion one can obtain uniform efficient implementation of the communication and synchronisation constructs. Moreover, it provided the means for allowing both input and output commands to appear in guards. We have demonstrated the usefulness of our concepts by providing solutions to a variety of familiar programming exercise.

References

BALZER, R. M. (1971). Ports—a Method for Dynamic Interprogram Communication and Job Control, AFIPS Conference.

BRINCH HANSEN, P. (1978). Distributed Processes: A Concurrent Programming Concept, *CACM*, Vol. 21 No. 11, pp.934-941.

DIJKSTRA, E. W. (1975). Guarded Commands, Non-determinancy and Formal Derivation of Programs, *CACM*, Vol. 18 No. 8, pp. 453-457.

HOARE, C. A. R. (1972). Proof of Correctness of Data Representation, *Acta Informatica*, Vol. 1 No. 4, pp. 271-281.

HOARE, C. A. R. (1978). Communicating Sequential Processes, *CACM*, Vol. 21 No. 8, pp. 666-677.

LAMPORT, L. (1978). Time, Clocks, and the Ordering of Events in a Distributed System, *CACM*, Vol. 21 No. 7, pp. 558-565.

Preliminary ADA Reference Manual, *SIGPLAN Notices*, Vol. 14 No. 6.

SILBERSCHATZ, A. (1979). Communication and Synchronization in Distributed Systems, *IEEE Trans. on Software Engineering*, Vol. 5 No. 6, pp. 542-547.

US DEPARTMENT OF DEFENSE (1977). Department of Defense Requirement for High Order Computer Programming Languages, *SIGPLAN Notices*, Vol. 12 No. 12, pp. 39-54.

WALDEN, D. C. (1972). A System for Interprocess Communication in a Resource Sharing Computer Network, *CACM*, Vol. 15 No. 4, pp. 221-230.

WIRTH, N. (1971). The Programming Language PASCAL, *Acta Informatica*, Vol. 1 No. 1, pp. 35-63.