1-1-1986

Task Queues: A General Model for the Implementation of Communications Protocols

Ann Kratzer
Dartmouth College

TASK QUEUES: A GENERAL MODEL FOR THE IMPLEMENTATION OF COMMUNICATIONS PROTOCOLS

Ann Kratzer

Technical Report PCS-TR86-119

Task Queues: A General Model for the Implementation of Communications Protocols

Ann Kratzer

Dartmouth College Hanover, New Hampshire 03755

Abstract

When any computer communications network is built, its communications protocol must always be implemented. The protocol is implemented on the switching nodes of the network. The node software must respond in real time to events generated external to the switching node. Thus, the software running on a switching node constitutes a concurrent program; this complicates the design, implementation and testing of the switching node software. The task queue model presented in this paper defines a structure for this software that facilitates the design, implementation and testing of communications protocols.

0) Introduction

The construction of any computer network includes the design and development of a communications operating system which runs on the switching nodes of the network. The communications operating system is an implementation of the communications protocol used in the computer network. The code of a communications operating system can be characterized as real time, asynchronous and concurrent. As such, it is difficult to design, implement and test. Given the specialized nature of communications operating systems, a specialized structure can greatly reduce the complexity of an implementation.

In this paper, we present a general model, which we call the task queue model, of a protocol machine. This model defines a structure for a communications operating system that facilitates its design, implementation and testing. In particular, the model limits the different kinds of concurrency that can occur and clearly defines critical

sections; furthermore, communications protocols map very naturally into the task queue model.

To the best of the author's knowledge, there has been no work reported on the general question of communications operating system design. Others have reported their work within the context of particular network implementations[1-3]. The research most closely related to this effort has been in three areas. The first has addressed communications protocols: the design and specification of standards[4,5], the analysis and measurement of protocol performance[6-8], the design of policies for flow control, routing etc.[6,7], and the verification of protocols[9,10]. Second, research in operating system construction is also related to this work[11-16]. Finally, work in concurrent and distributed programming is also relevant[17-25].

In Section 1, we give a general overview of a communications network and discuss the characteristics of a communications operating system. The job of a communications operating system is to provide communication services for users; the nature of these services is discussed in Section 2. In Section 3, the task queue model is presented. Two related issues, dispatching algorithms and task priority selection, are explored in Section 4; these issues are of central importance in the implementation of a task queue communications operating system. In Section 5, we outline the general form of the code for a task queue communications operating system. To illustrate how a protocol is mapped into the task queue model, we present in Section 6 a task queue implementation of a very small part of CCITT's protocol X.25[4]. Finally, the central concepts of the paper are summarized in Section 7.

1) Overview of a Communications Network

All computer networks include switching nodes that handle communication. The switching nodes typically implement the physical through network levels of the ISO Open Systems Model[26]. The lowest level of function in the ISO model, the physical level, includes the hardware needed to transmit and receive the electrical signals which represent digital information. Because the physical link is rendered unreliable due to noise, messages may be garbled in transmission or lost entirely. The second level, data link, is responsible for reliable communication of message packets between adjacent switching nodes. The data link level is defined by a protocol and is typically implemented partially in hardware and partially in software. Finally, reliable communication from one end of the communications network to the other is the responsibility of the network level. The network level is another layer of protocol and is usually

implemented in software. Higher levels of the ISO model define the protocols to be used between hosts and user processes.

Generally, a switching node interfaces one or more host computer(s) and possibly some number of terminals to the network. Communications line(s) connect the switching nodes together. There is an agreed-upon interface between hosts, terminals and the communications node; the data link and network level protocols govern communication between switching nodes.

The program running on the switching nodes must implement the interface to hosts and terminals in addition to the protocol. Thus, the node's software must respond in real time to requests generated external to the switching node. Implementation of the protocol entails managing I/O devices and other resources, most notably buffer space. Viewed in this light, the software running on a switching node constitutes a special purpose operating system. The primary design goal when implementing such an operating system is to achieve the protocol's inherent efficiency. This consideration affects the choice of resource management strategies and impacts the functional partitioning of the protocol into software modules. The task queue model defines a structure for a communications operating system.

2) Services Provided by a Switching Node

The mission of a switching node is to provide communication services. These services are available to three different kinds of customers: other switching nodes, host computers and terminals. It is the responsibility of the switching node to respond to requests from these customers. The nature of the requests depends on the customer.

The service, which a switching node provides to another switching node, is message relaying. The communications protocol defines how this is to be accomplished. To perform this service, functions such as flow control, routing, buffering and error control are required.

Two different paradigms for host computer service are common. They are distinguished by whether the host computer appears to the switching node as one or potentially many customers. In the first paradigm, the host is a single customer that requests communication services on behalf of its users. This is typical in datagram networks or if the host-to-host protocol multiplexes all user communication onto one logical host-to-host link. In the second model, the customers are individual host user processes. This paradigm is found in virtual circuit networks or if the

host-to-host protocol is demultiplexed. Regardless, the switching node performs essentially the same set of functions. These functions include creating a connection to a non local host or user process, destroying a connection, and sending and receiving information using an existing connection.

The primary service provided to a terminal is information transfer in both directions. A mechanism that allows a user to connect to a particular host is employed at the start of every terminal session. Ancillary services include support of options specific to terminals.

3) The Task Queue Model

The task queue model is developed in the sections that follow. In Section 3.0, the concepts of logical job, events and tasks are presented. In Section 3.1, we discuss the dispatcher which is responsible for sequencing the execution of tasks.

3.0) Logical Jobs, Events and Tasks

A communications operating system maps requests for service into the protocol used in the network. As such, the communications operating system is an interpreter, i.e. a dynamic translator. In the task queue model, this interpretation is carried out by tasks which service the requests made by the switching node's customers.

Each customer, supported by a communications operating system, constitutes a logical job. As noted in the previous section, requests for service can be made by customers. The arrival of a request for service is an event. Events are generated external to the switching node. (Usually, the arrival of an event is signalled by an interrupt.) Thus, a request for some communications service is synonymous with an event.

To process a request, a sequence of actions must be performed. In part, this sequence is defined by the interface used between the switching node and the customer. This interface stipulates what information is contained in the request and what return (if any) is made to the customer. The communications protocol also determines the actions which comprise this processing. Usually, the local switching node cannot completely process the request; message packets must either be sent to or received from other switching nodes in order to fulfill the request. The general form of the sequence of actions is:

build the appropriate packet, have it transmitted and wait for a response

or

wait for a packet (usually data.)

The key point is that the sequence of actions is broken at certain places by a wait. At these places, the node is obliged to wait for a packet from another switching node.

Several other points are noteworthy. The return to the customer may be made at the beginning of the processing; sending an acknowledgement (or arranging to send an acknowledgement) to an adjacent switching node on receipt of a packet is an example of such a return. Alternatively, the return may be made after all processing is complete; this is typically the case when the customer is a host or host user process. Also, there are some requests that can be processed locally by the switching node. In these cases, interaction with other switching nodes is not required, and thus no wait is needed.

The processing of a request is carried out by a sequence of tasks. A task is created when an event occurs and is specific to the particular event. The task must perform the processing necessary to respond to the event. The task terminates at the point where a wait is necessary or when servicing of the request is complete.

Because the processing of a request is decomposed into a sequence of tasks, it is necessary to remember where in the sequence of actions to resume processing. This information is maintained in the state associated with each logical job. Tasks modify the state of a logical job as appropriate to indicate what event is expected and what task should be created next to continue processing the request. The state of each logical job is contained in a data structure that is static.

For example, a host process might make a request to receive data when its logical job is in the idle state. If sufficient data is not available locally, the logical job enters the waiting-for-data state. When data arrives, the logical job enters the transfer-data state, and a transfer data task is created to return the data to the customer. If data is available locally at the time the request is made, then the logical job immediately enters the transfer-data state, and a data transfer task is created. In either case, when all the data has been transferred to the customer, the logical job enters the idle state. Should the connection be broken before sufficient data arrives, an error return is

made to the customer. In this case, the logical job enters the abend-receive-return state, and a task to handle the error return is created. After the error return, the logical job enters the idle state.

A task is described by a data structure that contains the name of the task or the address of the task code. In addition, the task descriptor may also record information about the event which spawned the task.

Although up to this point the discussion of tasks has been within the context of processing customer requests for service, the task model can be used for other purposes. Certain other functions are generally needed to implement communications protocols. Management of a timer (in generating time-outs) is a prime example of such a function. The example presented in Section 6 illustrates a function, which is not associated directly with any customer request, but which is needed to implement CCITT's protocol X.25.

3.1) The Dispatcher

The dispatcher is responsible for causing the execution of tasks. Associated with every task is the queue (or priority) to which it belongs. When a task is created, it is added to the end of the appropriate task queue. The dispatcher is responsible for scanning the task queues and selecting the next task to be run. In general, there may be many task queues; a total ordering of the task queues defines the priorities of the queues. The dispatcher scans the queues in priority order running down each queue in first-come-first-served order. The state of the dispatcher is defined by the priority of the queue currently being scheduled and the task currently running.

When a task is dispatched, it is removed from its queue and called as a subroutine of the dispatcher. Any relevant information about the event which caused the task's creation is passed to the task as arguments. The task then executes to completion. When complete, the task returns to the dispatcher which then schedules the next task.

Note that, logically, a task is never preempted by another task; this prevents critical sections between tasks. (A task may be interrupted when an event occurs. Such an interruption may create a task which is queued for later execution.) Preemption of a task is not necessary in a communications operating system because the processing performed by a task is generally simple. Thus, the time needed to complete a task is usually short. In cases where the processing is long, it can be decomposed into a sequence of tasks to create windows in which the dispatcher may run a more urgent task.

4) Dispatching Algorithms and Task Priority Selection

The performance of a task queue communications operating system is largely determined by: the algorithm used by the dispatcher in scanning queues, and the priorities assigned to the tasks. In this section, we attempt to identify the considerations relevant to both issues. (These issues are related.) However, given that the performance goals for computer communications networks are diverse, it is nearly impossible to be comprehensive, complete and general.

The two concerns in designing a dispatching algorithm are: time critical tasks must be run soon, and the scheduling algorithm must be fair, i.e. all tasks must be run in a timely fashion.

In general, a task should be regarded as critical if its delay would reduce the efficiency of the protocol. For example, many communications protocols employ time-outs and retransmission (until a response is received) on control messages. Thus, when such a control message or a response to such a control message is received, it should be processed as soon as possible so that the sender of the message does not time out and send another copy. Transmission and processing of flow control information are other examples. Tasks may also be considered critical if their delay would preclude meeting response time goals. This is the case in networks designed for real time response.

In addition to attempting to maximize protocol efficiency, the communications operating system must also attempt to minimize response time to customer requests. These to goals contradict each other. Noncritical tasks must not be delayed for a long time by the processing of critical tasks; starvation may result. Thus, in assigning priorities to tasks, distinctions between degrees of criticalness and fairness must be made.

Three main types of dispatching algorithm can be identified. Choice between them can only be made within the context of a particular implementation of a given protocol.

In the simplest dispatching algorithm, the dispatcher scans the queues in priority order, first-come-first-served within each queue. This is fair, but cannot respond in timely fashion to critical tasks.

Improving response to critical tasks can be accomplished very easily. When a critical task is created, the dispatcher is alerted. (A flag may be employed to accomplish this.) For the most part, the dispatcher still

considers the queues in priority order and first-come-first-served. However, when the dispatcher schedules the next task, it checks to see if a more critical task awaits execution. If so, then the more critical task is run. After processing the more critical task, the dispatcher should resume where it left off; this attempts to prevent starvation.

In certain circumstances for some protocols, critical tasks may be generated very frequently; this can lead to starvation which impacts response time. If this precludes meeting the response time goals of the computer network, then some mechanism to inflate task priorities is needed. Priority inflation can be accomplished by keeping track of the number of times processing is diverted to a critical task. When a task (or possibly an entire task queue) has weathered a certain predetermined number of such diversions, its priority is inflated to make it more critical. (We suspect that such an elaborate mechanism is probably never needed because communications time usually predominates over processing time in communications operating systems.)

5) Programming Considerations

The form of the code of a communications operating system is dependent on the language used to implement it. Nevertheless, its general form is:

Global Data: state information, buffer structures and queues

Mainline: initialization;
call dispatcher();

Subroutines for the tasks

Utility Routines for packet assembly, disassembly, buffer management etc.

Subroutines to respond to events

Certain information should be either global or accessible in many parts of the code. The logical state information is accessed by the initialization code, task subroutines and event code. Buffer structures are used by the initialization code, task subroutines and event code; the buffer free list need only be manipulated by the buffer management routines. The task queues are updated by the event code and the dispatcher.

After initialization, the dispatcher is invoked. The

dispatcher is in an infinite loop continuously dispatching tasks. Thus, a task queue communications operating system idles by running the dispatching loop.

The model presented above assumes that events are signalled by hardware interrupts that are processed by the event subroutines. If the hardware does not support interrupts then polling is necessary. (Interrupts are highly preferred because, with interrupts, it is easier to prevent second occurrences of an event before the first one is processed.) Polling is performed by the dispatcher. Rather than scan queues, the dispatcher polls devices to see what task should be run. The order in which the devices are polled is analogous to the order in which queues are dispatched; thus, the remarks of Section 4 apply to polling also. Although polling eliminates the overhead inherent in the event code, most notably the creation of tasks, overhead is incurred in polling.

The places in which critical sections can occur are limited. Critical sections on state information arise between the tasks and event code. In some implementations, it may be possible to have the event code examine but not change state information; this eliminates critical sections on state information. The task queues are subject to critical sections involving the dispatcher and the event code. If a single utility routine to insert a task on its queue is used by all event subroutines, then task queue critical sections are limited to two routines: the task queue insert subroutine and the dispatcher.

There is one other important issue to be addressed, deadlock. Deadlock can be caused in two different ways. First, a deadlock can occur locally within one switching node. Typically, this occurs when a critical task abends (possibly recreating itself or arranging to be recreated after a time out) while a less critical task holds the resource needed by the critical task. Such deadlocks can easily occur for buffer space, but can be prevented by using the protocol's flow control mechanism. Second, deadlock can occur between adjacent switching nodes. (Grid lock out is the classic example of such a deadlock.) Proper design of the protocol can eliminate these problems.

6) An Example

To illustrate how a protocol maps into the task queue model, we present an example of how a small part of CCITT's protocol X.25 can be implemented. Specifically, the implementation of the packet level link restart procedure is given. This part of the protocol does not directly service the request of any customer, but is an integral part of the

protocol.

Before presenting the example, it is necessary to discuss the part of X.25 covered by this example. X.25 is a multilevel protocol which supports virtual circuits. It is comprised of three levels that correspond to the lowest three levels of the ISO model. The part of X.25 used in this example is at level three of X.25, which CCITT refers to as the packet level. The packet level is responsible for implementing virtual circuits. The link restart procedure initializes the packet level link between two adjacent switching nodes.

In X.25 two different kinds of switching nodes are identified. Data Circuit Terminating Equipment, DCE, are the switching nodes in the communications network to which a user connects. The user's equipment is referred to as Data Terminal Equipment, DTE. X.25 defines the interface to be used between the DTE and the DCE. Before communication of information over virtual circuits can occur, the DTE and the DCE must both be brought to a known state. The procedure for this initialization, link restart, is best described by a state diagram - see Figure 1.

There are three states, indicated by boxes, in the restart procedure. Transitions between these states are indicated by directed edges. Each edge is labelled to indicate which party, the DCE or the DTE, causes the transition. Transitions occur when one party sends a packet to the other; the type of the packet is also shown on each edge.

At any time, either the DTE or the DCE may restart the link. When the restart is initiated, the link is in the packet level ready state. The actions which occur are essentially the same regardless of which party initiates the restart procedure. (Note that the state diagram is symmetric up to a renaming of packet types.) Thus, it suffices to discuss restart as initiated by one party.

When the DTE decides to restart the level three link, it indicates this by sending a restart request packet to the DCE. At this point, the DTE enters the DTE restart request state. The DCE can be in one of two states. First, if the DCE is in the packet level ready state, upon receipt of the DTE's restart request, the DCE acknowledges the restart by sending a restart confirmation packet to the DTE; the DCE reinitializes all virtual circuits to this DTE. Upon receipt of this restart confirmation, the DTE returns to the packet level ready state and initializes all virtual circuits to the DCE. Second, the DCE may itself be attempting to restart the packet level link at the time the DTE sends the restart request. In this case, a restart collision has occurred. Eventually, each party will realize that the other is also attempting a restart. The DTE receives a

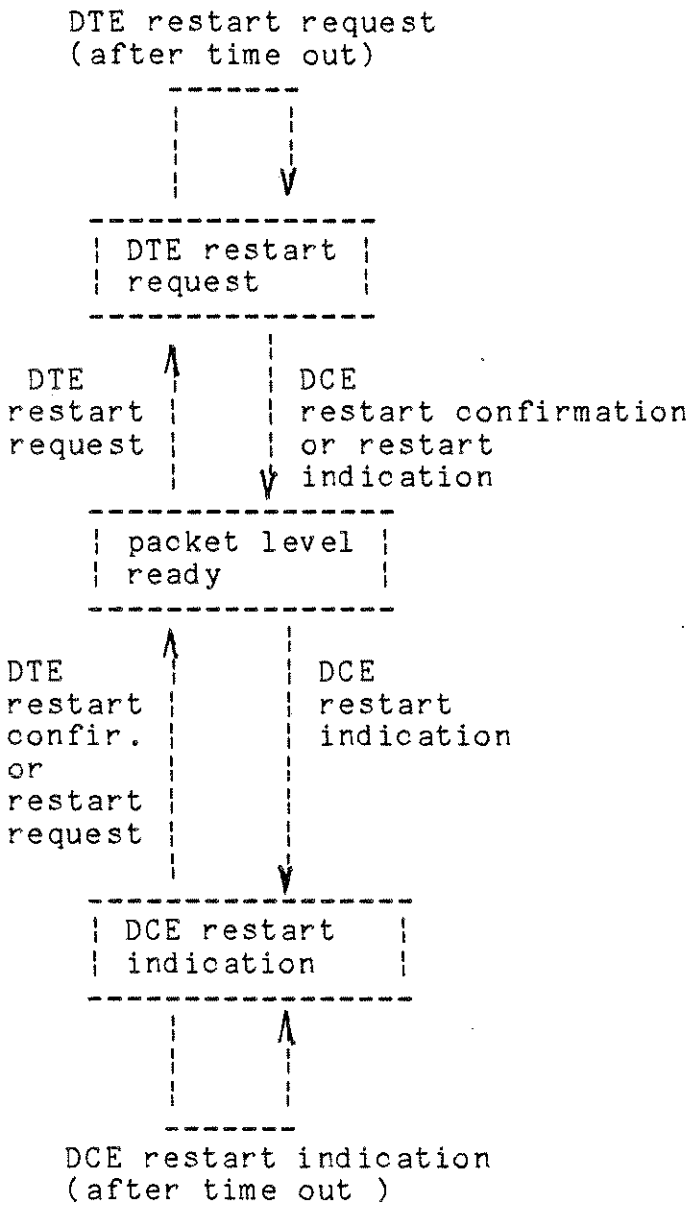

Figure 1: X.25 Restart Link Procedure

restart indication from the DCE (which is the way the DCE requests a restart,) and the DCE receives a restart request from the DTE. Both parties reinitialize the virtual circuits running between them and enter the packet level ready state.

In this example, we show the actions taken in the DTE. The DTE's link may be in one of two states: packet_level_ready or DTE_restart_request. The DTE has three tasks to handle the restart: one to send restart request packets, one to respond to a restart indication packet, and one to process a restart confirmation packet. All three tasks should be given high priority because the packets contain control information and because time outs are being used. An outline of each task is given in Figure 2. The important point is that the code can be written very easily from the state diagram; the code is an implementation of the state machine for the restart procedure.

```
procedure send_restart_request
begin
     build a restart request packet and arrange for it to be
     sent;
     link_state := DTE_restart_request;
     arrange for a time out;
end;


procedure when_receive_restart_indication
begin
     if link_state = DTE_restart_request then
          /* restart collision */
          stop the time out;
     else
          build and send a restart confirmation packet
     link_state := packet_level_ready;
     initialize all virtual circuits;
end;


procedure when_receive_restart_confirmation
begin
     stop the time out;
     link_state := packet_level_ready;
     initialize all virtual circuits;
end;
```

Figure 2: Tasks to Handle Restart in the DTE

The code to handle the time out event is shown in

Figure 3. Given that the time out was on the packet level link, the send_restart_request task is recreated to send another restart request packet; this task will be run by the dispatcher. We do not show the code invoked on the receipt of a restart confirmation or restart indication packet. This code is very similar to the timer event code; it creates either a when_receive_restart_indication or when_receive_restart_confirmation task as appropriate.

```
Timer Event Code:
     if the time out was on the packet level link then
          create a send_restart_request task;
```

Figure 3: Time Out Event Code

---

7) Summary

We have presented a general model of a protocol machine, the task queue model, that defines a structure for communications operating systems. This structure facilitates the design, implementation and test of communications operating systems. Concurrency is controlled and critical sections are well defined by this structure. Furthermore, communications protocol map very naturally into the task queue model. Presently, we are using the task queue model to design an implementation of the CCITT protocol X.25[4]; we feel that it has greatly simplified our work.

Acknowledgements

Special thanks go to Jonathan Panek whose fascination with computer networking prompted me to think about the construction of communications operating systems. I thank Robby Tan who established the utility of the task queue model by examining in detail its use in implementing CCITT's protocol X.25 Level 3. More recently, work done by Dick Schellens has extended our understanding of the more subtle aspects of task queues. Finally, I would like to thank my friends at the Kiewit Computational Center, most notably Dave Pearson and Stan Dunten, for sharing with me their

experiences and insights in designing and developing communications operating systems.

References

[1] "Improvements in the Design and Performance of the ARPA Network," J. M. McQuillan, W. R. Crowther, B. P. Cosell, D. C. Walden, F. E. Heart, AFIPS Fall Joint Computer Conference, p 741-754

[2] "A New Minicomputer/Multiprocessor for the ARPA Network," F. E. Heart, S. M. Ornstein, W. B. Barker, AFIPS Conference Proceedings Volume 42, 1973, National Computer Composition and Exposition, p 529-537

[3] "Packet Switching Using Concurrent Pascal in a Network Computer," Andre M. van Tilborg, Larry D. Wittie, COMPCON Fall 80, p 358-365

[4] Final Report on the Work of Study Group VII During the Period 1977-1980 Part III.2) Recommendation X.25, CCITT, June 1980

[5] "Ethernet: Distributed Packet Switching for Local Computer Networks," Robert M. Metcalfe, David R. Boggs, CACM, 26, 1, (January 1983,) p 90-95

[6] Queuing Systems Volume II: Computer Applications, Leonard Kleinrock, John Wiley and Sons, 1976

[7] Computer Networks, Andrew Tanenbaum, Prentice-Hall, 1981

[8] "Measured Performance of an Ethernet Local Network," J. F. Shoch, F. A. Hupp, CACM, 21,12, (December 1980,) p 711-721

[9] "The Certification of Data Communications Protocols," Keith Bartlett, D. Rayner, Proceedings, Trends and Applications: 1980, Computer Network Protocols, p 12-17

[10] "Verifying Network Protocols Using Temporal Logic," Brent Hailpern, Susan Owicki, Proceedings, Trends and Applications: 1980, Computer Network Protocols, p 18-28

[11] "The Structure of THE Multiprogramming System," E. W. Dijkstra, CACM, 11, 5, (May 1968,) p 341-346

[12] "Thoth, a Portable Real-Time Operating System," David R. Cheriton, Michael A. Malcolm, Lawrence S. Melen, Gary R. Sager, CACM, 22, 2, (February 1979,) p 105-115

[13] "The Roscoe Distributed Operating System," M. H. Solomon, R. A. Finkel, Proc. ACM SOSP7, December 1979, p 108-114

[14] "Medusa: An Experiment in Distributed Operating System Structure," John K. Ousterhout, Donald A. Scelza, Pradeep S. Sindhu, CACM, 23, 2, (February 1980,) p 92-104

[15] "SODS/OS: A Distributed Operating System for the IBM Series/1," W. D. Sincoskie, D. J. Farber, ACM SIGOPS, 14, 3, (July 1980,), p 46-54

[16] "The Cambridge Model Distributed System," M. V. Wilkes, R. M. Needham, ACM SIGOPS, 14, 1, (January 1980,) p 21-29

[17] "Monitors: An Operating System Structuring Concept," C. A. R. Hoare, CACM, 17, 10, (October 1974,) p 549-557

[18] "Communicating Sequential Processes," C. A. R. Hoare, CACM, 21, 8, (August 1978,) p 666-677

[19] The Architecture of Concurrent Programs, Per Brinch-Hansen, Prentice-Hall, 1977

[20] "Concurrent Programming Concepts," Per Brinch-Hansen, Computing Surveys, 5, 4, (December 1973,) p 223-245

[21] "The Programming Language Concurrent Pascal," Per Brinch-Hansen, IEEE Transactions on Software Engineering, SE-1, 2, (June 1975,) p 199-207

[22] "Distributed Processes: A Concurrent Programming Concept," Per Brinch-Hansen, CACM, 21, 11, (November 1978,) p 934-941

[23] "Modula: a Language for Modular Multiprogramming," N. Wirth, Software - Practice and Experience, 7 (1977,) p 3-35

[24] "Synchronizing Resources," Gregory R. Andrews, TOPLS, 3, 4, (October 1981,) p 405-430

[25] "High Level Programming for Distributed Computing," Jerome A. Feldman, CACM, 22, 6, (June 1979,) p 353-368

[26] Reference Model of Open Systems Interconnection (Version 4), ISO/TC97/SC16/N227