Network: A Multiprocessor Program[*]

(1978)

This paper explores the problems of implementing arbitrary forms of process communication on a multiprocessor network. It develops a Concurrent Pascal program that enables distributed processes to communicate on virtual channels. The channels cannot deadlock and will deliver all messages within a finite time. The operation, structure, text, and performance of this program are described. It was written, tested, and described in two weeks and it worked immediately.

1 Introduction

As an industrial programmer, I discovered that *real-time applications* require a much greater variety of process interactions than any "general-purpose" operating system can support. This experience led to the development of *Concurrent Pascal*—a programming language that makes it possible to implement arbitrary forms of process communication and resource scheduling by means of *monitors* (Brinch Hansen 1973, 1975; Hoare 1974).

Concurrent Pascal has been available on the PDP 11/45 computer since January 1975. It has been used to write three model operating systems for a single processor (Brinch Hansen 1976, 1977a, 1977b). This paper describes the first Concurrent Pascal program that controls process communication in a *multiprocessor network*.

The *Network* program enables distributed processes to communicate on virtual channels. These channels cannot deadlock and will deliver all messages within a finite time. The paper describes the operation, structure, text, and performance of this program. It was written, tested, and described in two weeks and it worked immediately.

[*]P. Brinch Hansen, Network: A multiprocessor program, *IEEE Transactions on Software Engineering 4*, 3 (May 1978), 194–199. Copyright © 1978, Institute of Electrical and Electronics Engineers, Inc.

2 Multiprocessor System

The multiprocessor system consists of a fixed number of processor *nodes* connected cyclically by unidirectional *bus links* (Fig. 1). Each node can receive input from its predecessor on one bus link and can send output to its successor on another bus link. An output operation in one node is delayed until its neighbor starts an input operation on the same bus link, and vice versa. (It is a sad comment on the complexity of the hardware that 500 words of machine code had to be added to the Concurrent Pascal kernel to implement these simple input/output operations.)

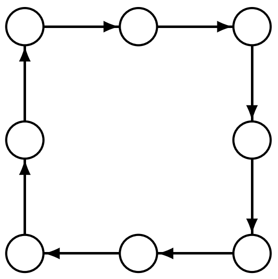

Figure 1 Data flow between processors connected by bus links.

The Network program assumes optimistically that the hardware works correctly.

3 Virtual Channels

The program uses the bus links to implement a fixed number of *virtual channels* connecting a fixed number of processes (Fig. 2). Each channel can transmit one data item at a time from a single sender process to a single receiver process. The sender and receiver processes of a channel may reside in the same or in different network nodes. The distribution of processes among the nodes and the connection of these processes by channels is fixed during program initialization. The restriction of one sender and one receiver per channel is assumed but is not enforced by the program.

Figure 3 shows an example of four processes A, B, C, and D connected by channels 1, 2, 3, and 4. Process A produces input for processes B and C

which, in turn, deliver output to process D. This abstract process configuration can be distributed in several ways. Figure 4 shows one possibility.

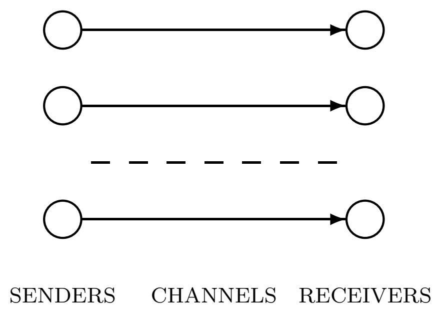

Figure 2 Data flow between processes connected by virtual channels.

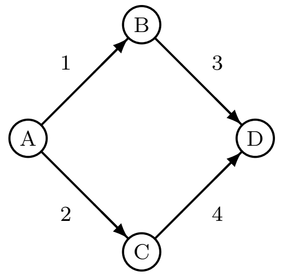

Figure 3 An example of processes connected by channels.

The Network program implements two operations

send(channel, item)   receive(channel, item)

A *send* operation on a channel delays the calling process until another process performs a *receive* operation on the same channel, and vice versa.

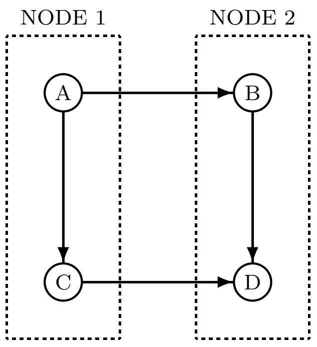

Figure 4 An example of a process distribution in a network with two nodes.

4 Network Operation

When two processes communicate on a virtual channel, the receiving node transmits a *request* to the sending node which then *responds* by transmitting a data item on the network (Fig. 5). So *a transmission on the network causes a message to pass through all nodes once.* The message begins as a request and ends as a response.

Transmission only takes place when a receiver is waiting for a message. Since each channel has only one receiver, it follows that *each channel can transmit only one message at a time.*

Each node is a first-come, first-served queuing system with a finite buffer capacity. A node receives input both from itself and its predecessor. The node accepts some of these messages as input to itself and outputs the rest to the next node (Fig. 6).

As long as its buffer is not full, a node will continue to receive input from itself and its predecessor in fair order. As long as a node is not delayed indefinitely by its successor, it will continue to output messages from its buffer in fair order (first come, first served).

5 Network Properties

The network has several pleasant properties:

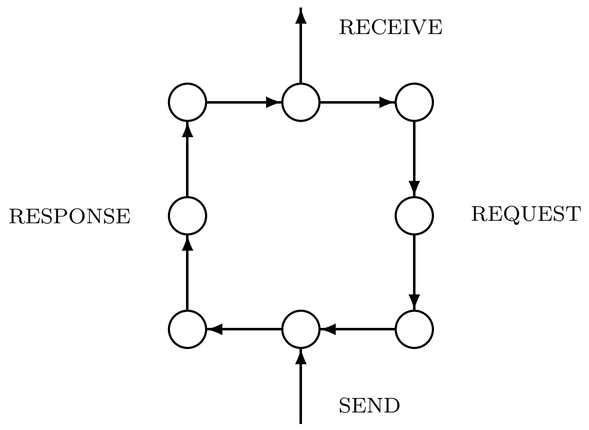

Figure 5 Transmission of a request and a response on the network.

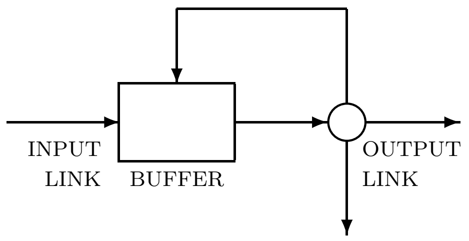

Figure 6 A network node viewed as a queuing system.

1. *No transmission takes place unless it is requested.* This means that the network only consumes machine time when processes are using it.

2. *Messages are never discarded due to buffer overflow.* A message cannot be sent until a process has provided a variable in which it can be stored and is waiting to receive it.

3. *The network cannot deadlock.* Each channel can transmit only one message at a time. If there are more buffer slots than channels in the network there will always be one or more empty buffer slots somewhere. So when there are messages on the net, at least one of the nodes will

always be able to receive a message from its predecessor and move it forwards towards its destination within a finite time.

4. *All messages are delivered within a finite time.* Suppose that a message M waits forever in a node N and is constantly being overtaken by other messages. Now, if the node is able to move some messages forward, it will do so in first-come, first-served order. And, if it continues to do so, it will eventually move the message M also. So, if one message gets stuck in a node, all messages arriving in that node will eventually get stuck there.

   Since all messages pass through all nodes, they would all sooner or later be stuck in the node N. But this cannot happen since we have shown that at least one message always can make progress somewhere. So the original assumption must be wrong: a message cannot wait forever in a node.

   The network will move each message forward with positive speed. And since a message passes through a finite number of nodes with finite buffer capacities it must arrive at its destination within a finite time. The network is fair.

5. *Transmission times are uniform.* All messages travel the same distance through all nodes. The uniform transmission times simplify the distribution of processes among the nodes.

   The only exception is a transmission between two processes in the same node. In this case, the request and the response both pass through all nodes.

6. *Space and time requirements are proportional to the size of the network.* Let n, c, and b be the number of nodes, channels, and buffer slots in the whole network. To avoid deadlock we must have $b > c$. The simplest choice is to give each node $c/n + 1$ slots, making

$$b = c + n$$

   Let t be the service time per message in a single node and let T be the total transmission time of a message on a single channel. Since a message must pass through all nodes once, we have $T \geq nt$. In the worst case, all channels may transmit simultaneously from the same node. It will then take nt for the first message to pass through all nodes. In addition, it will take the last node $(c-1)t$ to process the other $c-1$ messages. So we have

$$nt \leq T \leq (c+n-1)t$$

6 Program Structure

Each node contains a copy of the Network program that implements the virtual channels. The program consists of a fixed number of processes that communicate by monitors (Fig. 7).

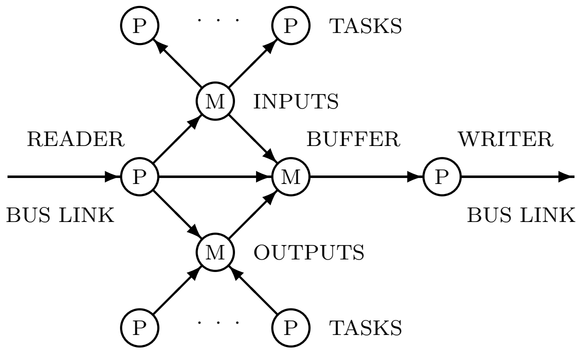

Figure 7 Data flow among processes and monitors in a single network node.

The *task* processes in a node will vary from one application to another. The other program components are fixed. A task process calls a local *input* monitor when it wishes to *receive* data on a channel. A *request* is now sent through the network to the other end of that channel and the process is delayed until a response comes back.

A task process calls a local *output* monitor when it wishes to *send* data on a channel. The process is delayed until a request for data arrives on that channel. A *response* is then sent through the network to the other end of the channel.

Each node contains a *reader* process that receives messages from the previous node through a *bus link*. If a message is a request or a response intended for its own node, the reader delivers it to the local output or input monitor. The remaining transit messages are sent directly to a local *buffer* monitor.

A *writer* process transmits messages from the local buffer through another *bus link* to the next node.

The Appendix contains the complete text of the Network program written in Concurrent Pascal.

7 Size and Performance

The Network program is about 250 lines long. It was written in less than a week and was tested systematically in another week using a method described in Brinch Hansen (1977b). No errors were found during testing. This paper was written in a few days making the total programming effort about two weeks.

The program has been running on two PDP 11/45 computers connected by bus links. It requires about 2900 words of core store in each computer (code 900 words and data 2000 words).

With 500 char/message the network has a maximum throughput of 30000 char/s. This rate is achieved only when the nodes spend all their time transmitting data. In practice, the speed of this multiprocesor system will be limited by the processing of messages performed by the task processes. The performance is also influenced by the configuration of processes and channels and their distribution among the network nodes.

8 Final Remarks

Many of these ideas are probably already described in the literature on computer networks (with which I am not familiar). The purpose of this paper, however, is not to advocate a particular method of network transmission. On the contrary, there are good reasons to believe that real-time programming can be simplified if process interactions can be tailored to the specific needs of each application.

To illustrate this point: It may be necessary to add recovery procedures to the present program to cope with transient hardware errors. Additional buffers must be added to make it possible for a process to poll several channels. And channels with many senders are much more convenient to use if distributed resources are scheduled among distributed processes. Finally, it seems clear that a completely different approach is needed to achieve high performance and cope with persistent hardware errors. This all depends on the requirements of particular applications.

This uncertainty about future needs makes it essential to have a methodology for the design of many different network programs. This paper shows one example of how such programs can be made simple and reliable at low cost by using an abstract language for modular multiprogramming.

Appendix: Program Text

This Appendix is intended for readers who are already familiar with the literature on Concurrent Pascal (Brinch Hansen 1975, 1976, 1977a, 1977b).

The Network program identifies procesor *nodes* and *virtual channels* by unique indices

```
type node = 1..nmax; channel = 1..cmax
```

The channels that originate or terminate in a node are identified by *channel sets*

```
type channelset = set of channel
```

The number of nodes and channels available and the type of data items transmitted through them may vary from one application to another

```
const nmax = ... ; cmax = ...
type item = ...
```

A network *message* is either a request or a response for a particular channel. If it is a response, the message includes a data item

```
type message = record
                 kind: (a_request, a_response);
                 link: channel;
                 contents: item
               end
```

A *task process* can *send* and *receive* data items on one or more more channels. These operations are implemented by output and input monitors described later.

```
type taskprocess =
process(inp: inputs; out: outputs; ...);
var a, b: channel; x, y: item;
begin
  ... inp.receive(a, x) ...
  ... out.send(b, y) ...
end
```

A *reader process* inputs one message at a time from the previous node through a bus link.

```
type readerprocess =
process(inpset, outset: channelset;
  inp: inputs; out: outputs;
  buf: buffer);
var m: message;
begin
  cycle
    input_from_buslink(m);
    with m do
      if (kind = a_response) & (link in inpset)
        then inp.response(m)
      else if (kind = a_request) & (link in outset)
        then out.request(m)
        else buf.send(m)
  end
end
```

The reader uses two constants defining the set of input channels and the set of output channels used by its node. If the node is the destination of a message, the reader performs a *response* or *request* operation on it. Otherwise, it *sends* the message through a local buffer to the next node. These operations are implemented by input, output, and buffer monitors. The details of input/output are described elsewhere (Brinch Hansen 1977b).

A *writer process* receives one message at a time from a local buffer and outputs it to the next node through a bus link.

```
type writerprocess =
process(buf: buffer);
var m: message;
begin
```

```
  cycle
    buf.receive(m);
    output_to_buslink(m)
  end
end
```

A *buffer monitor* implements two operations: *Send* delays a calling process as long as the buffer is full. It then puts a message into the buffer and continues the execution of another process (if there are any) waiting to receive the message. *Receive* delays a calling process as long as the buffer is empty. It then gets a message from the buffer and continues the execution of another process (if there are any) waiting to send a message.

Sequences and queues with several processes waiting to send or receive messages are not primitive concepts in Concurrent Pascal, but can be implemented in the language (Brinch Hansen 1977b).

```
type buffer =
monitor
const bmax = ... "cmax/nmax + 1";
var buf: sequence [bmax] of message;
  sender, receiver: queue;

procedure entry send(m: message);
begin
  if buf.full then delay(sender);
  buf.put(m);
  continue(receiver)
end;

procedure entry receive(var m: message);
begin
  if buf.empty then delay(receiver);
  buf.get(m);
  continue(sender)
end;

begin buf.reset end
```

An *input monitor* implements two operations: *Receive* sends a request through a local buffer and delays a calling process until a response arrives on a given channel. *Response* delivers a data item on a channel and continues the process that is waiting to receive it.

```
type inputs =
monitor(buf: buffer);
var receiver: array [channel] of queue;
  this: message;

procedure entry receive(c: channel; var v: item);
begin
  with this do
    begin kind := a_request; link := c end;
  buf.send(this); delay(receiver[c]);
  v := this.contents
end;

procedure entry response(m: message);
begin
  this := m; continue(receiver[m.link])
end;

begin end
```

An *output monitor* implements two operations: *Send* delays a calling process until a given channel is ready for transmission. It then sends a data item through a local buffer. *Request* makes a channel ready to send and continues a process (if there are any) waiting to send on that channel. *Initially* no channels are ready.

```
type outputs =
monitor(buf: buffer);
var list: array [channel] of
          record ready: boolean; sender: queue end;
  c: channel; this: message;

procedure entry send(c: channel; v: item);
begin
  with list[c] do
    if not ready then delay(sender);
  with this do
```

```
    begin
      kind := a_response; link := c;
      contents := v
    end;
  buf.send(this);
  list[c].ready := false
end;

procedure entry request(m: message);
begin
  with list[m.link] do
    begin ready := true; continue(sender) end
end;

begin
  for c := 1 to cmax do list[c].ready := false
end
```

All instances of these program components are declared and initialized by an *initial process* shown below. The definitions of channel sets and task processes may vary from node to node.

```
var inpset, outset: channelset;
  buf: buffer;
  inp: inputs; out: outputs;
  reader: readerprocess;
  writer: writerprocess;
  task1, task2, ... : taskprocess
begin
  inpset := [...]; outset := [...];
  init buf, inp(buf), out(buf),
    reader(inpset, outset, inp, out, buf),
    writer(buf),
    task1(inp, out, ...),
    task2(inp, out, ...),
    ...
end
```

Acknowledgements

The ideas in this paper were developed in discussions with B. Heidebrecht, D. Heimbigner, F. Stepczyk, and R. Vossler of TRW Systems, Redondo Beach, CA. The program was tested at TRW's Signal Processing Facility.

References

Brinch Hansen, P. 1973. *Operating System Principles.* Prentice Hall, Englewood Cliffs, NJ, (July).

Brinch Hansen, P. 1975. The programming language Concurrent Pascal. *IEEE Transactions on Software Engineering 1*, 2 (June), 199–207. *Article 7.*

Brinch Hansen, P. 1976. The Solo operating system. *Software—Practice and Experience 6*, 2 (April–June), 141–205. *Articles 8–9.*

Brinch Hansen, P. 1977a. Experience with modular concurrent programming. *IEEE Transactions on Software Enginering 3*, 2 (March), 156–159. *Article 11.*

Brinch Hansen, P. 1977b. *The Architecture of Concurrent Programs.* Prentice Hall, Englewood Cliffs, NJ, (July).

Hoare, C.A.R. 1974. Monitors: An operating system structuring concept. *Communications of the ACM 17*, 10 (October), 549–557.