Routing messages through networks: an exercise in deadlock avoidance

by A.W. Roscoe

Oxford University Computing Laboratory,
8-11 Keble Road, Oxford OX1 3QD, U.K.

Abstract. *Protocols are developed for passing messages between the nodes of a transputer network. We see how mathematical principles allow us gradually to build up a complex system from its correctness and other requirements. In particular, much use is made of the author's earlier work on deadlock freedom.*

1. Introduction. We are given a network of processors, each of which might wish to send a message to any other. However each process can only afford to devote a relatively small amount of buffering space to relaying messages. It seems difficult to devise algorithms for this which are moderately efficient but, when heavily loaded, avoid both deadlock and infinite internal communication.

In this paper we show how to develop a protocol which avoids deadlock and guarantees delivery of all messages. It places only mild restrictions on routing (and we will show later how to avoid these restrictions in most cases). It does not generate any activity when no messages are being passed, as would probably be the case with a token-based protocol. The buffering space required in each node is independent of the size of the network.

We also show how to make use of the protocol to implement normal occam communication (handshaken with no output guards in ALTs) between all nodes of an arbitrary connected network.

The occam used is essentially occam-1 [inmos], with the occasional borrowing from occam-2 [inmos1]. Some of the development is carried out in CSP [H] rather than directly in occam. It is based heavily on the author's previous work on deadlock [BR,RD], the terminology of which is adopted without comment.

2. The problem. We will suppose that we are given a connected network of processors which communicate by posting messages to one another. Each message sent by a process carries the address of a single destination process. No process sends a message to itself. No process can indefinitely refuse delivery of a message from another one. It is conceptually simpler to think of this as the *only* mode of communication between the users of the network, though in practice it does not matter if there are other links as long as they obey the above regarding the acceptance of messages delivered by the service.

Clearly we should expect the service to deliver messages to their correct destinations in a finite time. Also, a node must not be able to refuse the input of a message from its user for ever. One would also hope that, allowing for nodes which become over-used or blocked, it will choose a sensible route for a message from its sender to destination, taking advantage of the full range of connections in the network.

An implementation of this service can be expected to consist of a server process running on each node in parallel with the user process. Normally the former will use all of the node's links within the network, while the latter will communicate only with its server and the outside world. The server should not consume an inordinate amount of space, and so should only offer a relatively small amount of buffering space for through messages. Bearing in mind the fact that the servers share processors with their users, they should not place unreasonable demands on running time. When the service is inactive (i.e., no messages are in transit) it should not consume any processor time. Equally, a process that is (hopefully, temporarily) prevented by its neighbours from passing on messages should wait non-busily.

Naive attempts to implement this service can fail for a number of reasons, some of which are described informally below.

a) All the nodes could fill up all their buffer space with messages. In this case, since no node can accept another message, no further progress can be made.

b) One might attempt to get round this by giving full nodes the ability to "swap" messages with a neighbour. However it is easy to construct examples of full networks (or subnetworks) where, whenever one process wishes to send a message to another, the latter has no message which it wants to swap.

c) Even if one manages to avoid a complete jam it is easy to let a message get trapped in a backwater and never arrive, even though the whole network continues to make progress. Equally, it is quite possible for a node to become unable ever to accept another message from its own user process because it is too busy transmitting messages for others.

Throughout the rest of this paper we will assume for simplicity that messages are carried in packets of a standard size, so that the number of messages a node is able to store is independent of their contents.

3.The ring protocol. In this section we introduce a protocol which meets our requirements but which applies only to simple ring networks. The author has already described a similar system in [BR]. First we establish two theorems on deadlock.

THEOREM 1 [BR,RD] If V is a busy, triple disjoint and strong conflict free network, then any deadlock state contains a cycle of ungranted requests (a cycle of at least three distinct processes, each of which is blocked by the next). □

We should recall (for it will be important later) that the definition of "ungranted request" in a network specifically excludes cases where either process (the blocker or the blocked) can communicate externally, for such cases cannot occur when the network is deadlocked! The following is a slightly stronger version of a result proved in [BR].

THEOREM 2 Suppose the network V is busy, triple disjoint and conflict free, and has the property that whenever any process P is blocked, it has an ungranted request to some other process Q such that Q has communicated with P, and has done so more recently than every communication with another process to which Q can have an ungranted request. Then V is deadlock free.

PROOF Suppose V can deadlock. Consider the cycle $\langle P_0, P_1, \ldots, P_n \rangle$ of ungranted requests generated by Theorem 1. The hypotheses of Theorem 2 allow us to assume that each of the ungranted requests in the cycle has the special form set out there. Each process P_i in the cycle is blocking P_{i-1} and blocked by P_{i+1} (arithmetic modulo $n+1$). The hypotheses are easily seen to imply that P_i's last communication with P_{i-1} must have come after its last with P_{i+1}. But, working this round the cycle gives a contradiction: the last communications between P_i and P_{i+1} must have been earlier with increasing i, but the conclusion that the last communication between P_n and P_0 preceded the last between P_0 and P_1 contradicts what we already know about P_0. □

So suppose that the nodes of our network are arranged in a ring and each has space for at least two messages in a fixed length buffer. Suppose further that no node will accept a message from its user process to another if this would fill the buffer. Otherwise it will accept any message for which it has room. Then the conditions of Theorem 2 are satisfied. A process can only be blocked when it is non-empty (for when empty it can communicate with its environment). Thus, whenever it is blocked it has an ungranted request to the next process round the ring (which play the role of "Q" from the statement of Theorem 2). A process can only be blocked by the next process round the ring when the latter is full. But when a process is full its last communication was with its predecessor, for its environment cannot fill it up.

There are other, similar, strategies for the ring which satisfy Theorem 2. The one essential feature is that no process can refuse an input from the ring unless it has previously accepted one and has done so since its last ring output (as opposed to outputs to its own user). Thus initially a slot must be reserved for ring input, and the slot created by a ring output must be reserved for ring input (though one need not reserve more than one at any time).

A CSP IMPLEMENTATION. Perhaps the simplest system satisfying the above constraints is one where each node contains a two-place buffer, will accept a message from the ring whenever it has a spare slot, but will only accept a message from its user when it is empty (i.e., when this message does not fill it).

The set of all messages is denoted by M. We assume that the destination of each message m can be extracted by applying the function d. The ring is then

$$\mathop{\Big\|}_{i=0}^{n-1} P_i$$

where

$$P_i = (i.m : M \to Q_i^m) \,\square\, (in_i.m : M \to Q_i^m)$$

$$\begin{aligned} Q_i^m &= out_i.m \to P_i && \text{if } d(m) = i \\ &= i.m' : M \to i{+}1.m \to Q_i^{m'} \\ &\quad \square\, i{+}1.m \to P_i && \text{otherwise} \end{aligned}$$

All arithmetic here is modulo n. Notice that P_i and P_{i+1} are linked by channel $i{+}1$, and that P_i is linked to its user by channels in_i (for inserting messages into the ring) and out_i (for the delivery of messages from other processes). It is easy to check that (for $N > 2$) the conditions of Theorem 2 are satisfied, so the network is deadlock free. (In fact it also works when $N = 2$.)

The network remains deadlock free if we insert arbitrary fixed-length buffers (with no connections to the outside world) between any or all of the nodes of this implementation. Like all rings this extended network has two possible cycles. All the nodes might be blocked on output (and no process can therefore be prepared to input) or *vice versa*. The former is excluded by exactly the same argument used in the proof of Theorem 2: no process can be blocked from outputting unless its successor's last communication was with it. The latter is impossible since the nodes which are not pure buffers can communicate externally when they can input, but not output, on the ring.

Notice also that the refusal by a user process to accept a message that is delivered to it will quickly bring much of the network to a halt. This is more or less inevitable when the nodes contain only finite buffering, and explains our assumption that user processes will never infinitely delay accepting a message. *Efficiency* might be improved by inserting buffering on channel out_i, but the only way to keep the ring moving when a process steadfastly refuses to accept its messages is to throw them away!

An occam implementation. The only real difficulty in translating the CSP program above to occam is the appearance of what amounts to an output guard in an **ALT** when the process is simultaneously willing to transmit one message on to one neighbour and to receive another from the other neighbour. Geraint Jones [J] has developed techniques for getting round this in cases like the present one where, once a program has the ability to perform an output, this ability can never disappear no matter what else the process does. They essentially rely on the **PAR** expansion laws of [RH] which state that the parallel composition of processes which can communicate initially is equivalent to a suitable **ALT**. Thus

```
                ALT
PAR               c?x
  c?x      =        d!2
  d!2             d!2
                    c?x
```

One of Jones' techniques is used below, and another slightly different one in a later example.

The following is one node from a ring satisfying the constraints of Theorem 2. In fact its buffering is rather smaller than that of the CSP example, being "normally" one, only expanding to two when it needs to (on inserting one of its own messages into the ring) and contracting as soon as it can (when it becomes empty or delivers a message to its own user). It has been pointed out to me that the system is quite similar to a "register insertion ring": see [T].

The process below has four external channels, **left** and **right** for respectively receiving and transmitting messages on the ring, and **in** and **out** for communicating with its local user. We assume that **local(x)** determines whether the message **x** from the ring is for the node's own user.

On receiving a message from its own user it splits into two so that it can still accept messages from the ring. The right hand half of this then initially transmits this message on round the ring, sends an acknowledgement to its partner and waits either for another message or the instruction to terminate. The left hand half listens either for this acknowledgement or for another message from the ring (not from its own user). If it receives the acknowledgement then it knows the whole node is now empty and can shrink back to size one (for that will not make it refuse a message from **left**). If it receives a message first which is not for its own user it waits for the acknowledgement (signifying that the right-hand process is empty) before passing that message on and going back to its initial state. If the message received is for its own user then it can safely shrink (after delivering the message and waiting for the other process to empty) as its last communication on the ring was with the node to its left.

```
VAR x:
WHILE TRUE
  ALT
    left ? x
      IF
        local(x)
          out ! x
        TRUE
          right ! x
    in ? x
      CHAN mid,ack,halt:
      PAR
        VAR going,y:
        SEQ
          going := TRUE
          WHILE going
            ALT
              ack ? ANY
                SEQ
                  halt ! ANY
                  going := FALSE
              left ? y
                IF
                  local(y)
                    PAR
                      SEQ
                        ack ? ANY
                        halt ! ANY
                      out ! y
                  TRUE
                    SEQ
                      ack ? ANY
                      mid ! y
        VAR going:
        SEQ
          going := TRUE
          WHILE going
            SEQ
              right ! x
              ack ! ANY
              ALT
                halt ? ANY
                  going := FALSE
                mid ? x
                  SKIP
```

Note that a node can only refuse input from the ring when *either* it can make an external communication *or* the most recent ring communication was an input. It can only refuse to output on the ring when it can make an external communication. Thus the conditions of Theorem 2 are satisfied.

Once again it is permissible to insert arbitrary fixed-length buffers between nodes. (In fact in either example it is permissible to introduce a variable length buffer provided the latter never, when its last communication was an output, shrinks enough to prevent it accepting an input. Thus a 6 item buffer with 2 things in it could then validly shrink to length 3 but not to length 2.)

Ensuring messages are delivered. The networks we have seen so far are deadlock free. When the networks are empty of messages they will accept any new message. Further, when a finite number of messages are inserted they will all be delivered correctly. Unfortunately, however, under heavy traffic one can guarantee neither that a message will be accepted by the network nor delivered even if it is. We will deal with the second problem now and leave the first till a little later.

The problem of an accepted message not being delivered can arise in either of our implementations because of one or more processes being infinitely excluded from executing (where processes are sharing processors) or because of unfairness by one of the nodes: it might accept messages from its own user infinitely without accepting any from the ring. From now on we will assume that the first of these never occurs. In other words every parallel process which is able to proceed (is not terminated, aborted or waiting for communication) is eventually enabled.

The second problem is of course very unlikely in the particular occam implementation above, but it is nevertheless allowed by the semantics. It can be avoided by making sure that nodes give priority to inputs from the ring over inputs from their own users. (In fact it is enough that they all give this priority for a non-zero proportion of inputs.) This can be achieved in our example by replacing the outermost **ALT** by **PRI ALT**. Since **PRI ALT** is a permitted implementation of **ALT** in the semantics, this change does not destroy the deadlock freedom of the network.

Since the ring is finite and messages pass round it in only one direction, to prove that each message reaches its destination it is enough to show that no message ever gets stuck infinitely in a node. So suppose this can happen. It is easy to see that no node can do an infinite amount of work without passing every message it receives onwards. This means that the node P where any message gets stuck eventually stops doing anything. We have assumed that no process is ever infinitely descheduled when it can do something, so must conclude that the node is infinitely blocked by its right-hand neighbour. But because of the **PRI ALT** now used in this neighbour and the assumption that no user can infinitely refuse a message offered to it we can conclude that it can only block P infinitely if it is in turn blocked by its own right hand neighbour and cannot communicate externally. Carrying this argument round the ring tells us it must be deadlocked, contradicting what we already know.

Ensuring that all messages are accepted. The implementations above can, in heavy traffic, easily prevent a node getting one of its messages into the ring. Even if the outermost **ALT** were alternately giving priority to ring and external inputs, this would not stop the node getting stuck in its "parallel" state where it can accept no external inputs. For example, if process P is continually sending messages to Q, some way down the ring, this can easily prevent all the users between them from inserting more than one message into the ring.

Observe that whenever a node receives a message for its own user it shrinks enough to accept a message *from* its user. (Indeed in the occam implementation, because the second place is only created for a locally generated message, it is bound to be able to accept such a message.) Thus if we were able to make sure that a node received messages for ever we would be sure it could accept messages for ever.

In the ring this could easily be achieved by making each message do a complete circuit: depositing its contents at its real destination but carrying its "token" property back to its original sender. There are two reasons why we do not adopt this simplest solution. Firstly it, on average, doubles the amount of traffic carried by the ring, and it can be expected that the ring will be significantly more effective when lightly loaded. More importantly this solution does not translate well to the cases of non-ring topologies covered below.

In fact we adopt a hybrid solution. We give each node two types of extra slots it can use when inserting its own messages. The type it normally uses will be for messages without this "boomerang" property. ("Normally", because we expect the ring to have enough capacity that events like the starvation of a node are unusual even without enhancements to avoid it.) But it has extra slot(s), for use *in extremis*, which can only be used for "boomerangs" (messages which carry on round the ring back to their sender after depositing their contents at their destinations). This has the advantage that, in normal operation, it is as efficient as the original system, but cannot starve any user.

We do not give a separate implementation here: one can easily be extracted from the more complex implementation below on general topologies.

We will see later that messages which always return to their senders have another use: implementing occam channels through a network such as the present one.

4. Non-ring topologies. Most networks are not rings, and very often imposing a ring structure on them would be very inefficient in terms of the length of path traversed by a message. The average length of path between two nodes on a ring is linear in the size of the network, whereas in an n-dimensional grid it is roughly the nth root of the number of nodes, and in certain other networks (e.g., butterfly or binary tree) it is logarithmic. It would be far better if we could make use of these much shorter paths when routing messages.

Suppose that, given an arbitrary network, we have a Hamiltonian circuit (a cycle including each node once). Then we can implement the ring protocol with all the disadvantages described above. Our aim will be to develop this into a relatively efficient system while retaining all the correctness of the original. The essential idea is to make use of the extra links in the system, but in a way which (a) any block on an extra link is of a trivial sort, which provably cannot appear in a deadlock state, (b) does not introduce livelock and (c) leaves the ring behaviour essentially unaltered.

We do this by treating messages received across these extra links, "crosslinks", in essentially the same way as messages received from the environment. When a process wants to send a message over a crosslink it must be prepared to accept rejection and, when this occurs, send the message down the ring: the important thing is that it should not *commit* itself to sending the message along a crosslink. In CSP this situation can be modelled as

$$((cl!m \rightarrow \mathit{SKIP}) \Box (\mathit{right}!m \rightarrow \mathit{SKIP})) \sqcap (\mathit{right}!m \rightarrow \mathit{SKIP})$$

or equivalently, but more suggestively, as

$$((cl!m \rightarrow \mathit{SKIP}) \Box (\mathit{wait} \rightarrow \mathit{right}!m \rightarrow \mathit{SKIP})) \backslash \mathit{wait}$$

where cl is a crosslink channel and right is the ring output.

We cannot do the same in occam because of the use of output guards above. However we can achieve a similar effect. Instead of refusing such a message when too full to insert it into the network, a process will always accept it and then send back either an acknowledgement (if there is space) or a rejection (if there is not). Of course, if messages are large, it may be more efficient to ascertain whether there is space before sending one across.

If desired a node can try several crosslinks for a given message, but in order to avoid livelock it is necessary that it never delays indefinitely committing the message to the ring. (It is sufficient that it never tries the same crosslink infinitely often.)

Provided we maintain the same condition as before, that unless its last communication with the ring was an input a node must be prepared to input a message from the ring, the network may be proved deadlock free by exactly the same argument. For the only links on which processes can be blocked in a potential deadlock state are those on the ring.

There is an important restriction we place on the use of crosslinks: A may only try to route a message to B via C (to which A has a crosslink) when C is between A and B in the underlying ring. If this constraint were ignored there would be the danger of a message being passed round the network for ever. This unfortunate possibility does not arise if the constraint is obeyed since whenever a message is passed it gets closer, in ring terms, to its destination: we can think of this distance as a variant in the usual sense. Later we will discuss some ways in which this constraint can be relaxed.

As before no message that has been accepted by the network can fail to reach its destination provided no process is infinitely excluded from executing. And this in turn is ensured by making sure each process sometimes gives priority to each of its inputs on non-external channels. (Note that this means that sometimes priority must be given to inputs on crosslinks, even when the received request will be turned down. We will discuss this issue more later.)

A slot for use by either local messages or crosslink messages can be recovered whenever a node succeeds in transmitting any message over a crosslink or receives a message (other than one which must be returned to its sender) for its own user.

Though we can think in a coarse sense that a node treats messages from its own user and from crosslinks in the same way, it cannot treat them the same way when it comes to "guaranteed acceptance". Note that though it would obviously be nice if each node could be guaranteed to accept messages over crosslinks for ever, this is not necessary to maintain correctness as no process ever waits indefinitely for it to accept one. It would be hard (if not impossible) to achieve this by the same method as before. For that method relies on the fact that the place where a message eventually terminates has a slot freed, and each message can only terminate at one place while it can pass across many crosslinks and might in any case be obliged to go back to its original sender. For these reasons we do not tackle these problems here.

Bearing the above in mind, we reserve each node's "boomerang" slots exclusively for messages originating there. There is an enhancement possible where there are crosslinks. Specifically there is no need for the "token" part of the message to go all the way to the addressee before being sent back to the sender. At some point, where it is safe, the message can split into two: the real message, to the addressee and the token, to the sender. This split should usually reduce both the total amount of communication in the system and the wait of the sender for the return of its token.

An occam implementation. There are many ways of implementing systems satisfying the above. The following is one fairly general solution. At this point we ignore the question of how to decide on the best way of routing a message.

We still have to get over the problem of output guards when a node is waiting to send a message down the ring. We adopt a slightly different solution this time: instead of creating a special parallel process when required and closing it down later we simply leave one running all the time. Which method is more efficient will depend on circumstances.

Thus the system we use is

```
CHAN mid,ack:
PAR
  P
  Q
```

where

```
Q  =  VAR x:
      WHILE TRUE
        SEQ
          mid ? x
          right ! x
          ack ! ANY
```

"right" is once again the channel used for outputting along the ring. **P** has external channels **left** for inputting messages from the ring, **left** and **ack** for communicating with **Q** and a pair **clin** and **clout** for each crosslink. (In the program below we assume there are two crosslinks, the channels on the first being as above and on the second being **clin1** and **clout1**. Increasing the number further requires only trivial modifications.) The pairs of channels implement a bidirectional crosslink: messages can pass in either direction.

P maintains a queue, implemented as a circular buffer, of messages waiting to be passed along the ring. There are a number of constants associated with **P**: **max** is one less than the upper bound on the size of the queue and is the most messages it can contain and still input; **max_for_cl** and **max_for_ring**, both of which must be positive and strictly less than **max** − **1** respectively represent the maximum contents at which **P** will normally accept an input respectively from a crosslink or from the ring. (The exceptions to these limits will be described later.)

There are also constants **yes** and **no**, distinct from all messages, which are used to indicate that a message sent over a crosslink has respectively been accepted or rejected.

The variable **acconring** records the node's obligation under the protocol to accept a message on the ring. Initially true, it is later true just when **P** has made a ring output more recently than a ring input. Note below that **P** cannot refuse a ring input when in its "main" state and **acconring** is true (this is the exception to the queue size limit for ring input). It will become apparent that **P** is never indefinitely prevented from returning to this state. Although the whole node (including the process on the right channel) can have output more recently than input on the ring when **acconring** is false, this can never cause refusal of a ring input: either the main process can make an output to the small one (causing **acconring** to be set), or the main process is empty so that the **max_for_ring** limit does not apply.

The variable **qsize** records the current number of messages in the queue awaiting output on the ring. It is used as the criterion for deciding whether or not to accept particular messages. Note that when **acconring** is set a place is reserved for ring input, but since it only is set to **TRUE** when a place has just been released this causes no problems of overfilling.

The overall shape of **P** is shown here. The sections named in italics are described below.

```
VAR x, qsize, next, acconring, queue[max+1], head, tail:
SEQ
  acconring := TRUE
  qsize := 0
  head := 0
  tail := max
  WHILE TRUE
    ALT
      (qsize > 0) & ack ? ANY
        SEQ
          qoutput
          mid ! next
          acconring := TRUE
      ((qsize < max) OR ((qsize = max) AND (NOT acconring))) & in ? x
        SEQ
          make boomerang
          route
      clin ? x
        IF
          qsize ≤ max_for_cl
            SEQ
              clout ! yes
              route
          TRUE
            clout ! no
      clin1 ? x
        IF
          qsize ≤ max_for_cl
            SEQ
              clout1 ! yes
              route
          TRUE
            clout1 ! no
      (acconring OR (qsize ≤ max_for_ring)) & left ? x
        SEQ
          route
          acconring := FALSE
```

The program is just an infinite loop which on each iteration performs an input from one of four sources. The first possibility is that it gets an acknowledgement from its "right" slave that a ring output has been performed: the structure of the latter then guarantees that the output of the next value on **mid** is not delayed. The acknowlegement, implying committal to output, is only accepted when there is something in the queue for ring output.

This program is willing to input from its own user whenever it possibly can: when there are at least two places available in the queue or when there is just one and there is no obligation to accept a ring input. Note that even though not every communication input is eventually put on the queue, input is only accepted when there is room. This is because one cannot tell that it will not have to be queued.

Requests over crosslinks are never refused, for we wish to create as few blocks over crosslinks as possible (see later for the actual rules which apply here). However a request is rejected if the node is too full.

An input over the ring may be accepted when the node is sufficiently empty or when an obligation exists under the protocol. Making such an input cancels any such obligation.

The segment *qoutput* removes an item from the queue and adjusts **qsize**.

```
SEQ
  next := queue[head]
  head := (head + 1) MOD (max + 1)
  qsize := qsize − 1
```

The segment *make boomerang* has the function of deciding whether or not a given message originating at the node will be a "boomerang" and, if so, marking it accordingly. Many strategies for this are possible and are omitted. To ensure that the node is infinitely often able to accept messages from its own user it is essential that this section never allows *both* the queue to be full (which includes the case where the only place is reserved for the ring) *and* there to be none of its boomerang messages presently in play.

route is the piece of code which decides what to do with a message **x** just input. For reasons which will become apparent we will sometimes have to repeat this routing procedure, so *route* has the form of a loop. However since we must ensure control returns infinitely often to the outer alternative, we cannot allow the procedure to remain in this loop forever. Thus there is a maximum tolerance for how many times the loop can be executed. (It can be expected that this bound comes into effect only very rarely.)

When a message arrives for the local user it is delivered unless it is a returned boomerang, in which case there is no need. If the message is another process' boomerang we have to deal with the return of the message (stripped of its contents) to its sender, so the variable **repeat** is set indicating that the routing routine should be repeated to start the return journey.

Otherwise we decide whether or not to try to send it over a crosslink (we assume the existence of predicates **try_cl** and **try_cl1**). The details of how such an attempts is made is given later.

```
VAR succ, repeat, newmess, tolerance:
SEQ
  repeat := TRUE
  tolerance := maxrepeats
  WHILE repeat
    SEQ
      repeat := FALSE
      IF
        local(x) AND returned(x)
          succ := TRUE
        local(x) AND (NOT must_return(x))
          SEQ
            out ! x
            succ := TRUE
        local(x) AND must_return(x)
          SEQ
            out ! mess(x)
            succ := TRUE
            repeat := TRUE
            newmess := ret(x)
        try_cl
          try sending x over cl
        try_cl1
          try sending x over cl1
        TRUE
          succ := FALSE
      IF
        repeat AND ((tolerance > 0) OR local(newmess))
          SEQ
            tolerance := tolerance − 1
            x := newmess
        repeat
          SEQ
            add newmess to queue
            repeat := FALSE
        succ
          SKIP
        TRUE
          add x to queue
```

If it has been decided not to send the message over a crosslink, or an attempt to do so has failed, it is added to the queue. If there has proved to be the need for a repeat call but there is no tolerance left for repeats, the new message is added to the queue. (Note that there will always be space in the queue for the message added, for no input is accepted at the initial stage unless there is at least one vacant space there. Also whenever a repeat call is generated this slot is still vacant.) There is one exception to this rule: when the new message is for the local user. But this cannot add more than one to the number of iterations since any repeat generated by a local message is the return of *another* node's boomerang and so is not local.

The main gap we still have to fill in is that to *try sending x over cl*. (Obviously the cl1 case is symmetric.) It is necessary that no deadlock inducing blocks should appear over crosslinks, so this piece of code has to be designed with particular care. We will make sure that the following rules are observed between each pair **N, M** of crosslinked nodes. (Here, "node" means the whole program there, including the small process sitting on the ring output link.) One process blocks another when (i) neither can communicate externally (with its own user), (ii) it cannot respond to the request for communication and (iii) neither can make any internal progress.

a) **N** cannot block a request from **M** unless **N** is awaiting a response to a request it has already made over a crosslink to a third process or is waiting to respond to a request made to it over a crosslink by a third process.

b) **N** cannot refuse to respond to a request it has accepted.

c) **M** cannot refuse to accept the response to a request it has made.

Clearly any block between a pair of processes which appears in the network because of (a) will soon disappear because of (b) and (c). Thus no such block can appear in a deadlock state.

Because (a) only allows a node to block a request over a crosslink when it has *already made* a request of its own, it must, when making such a request be prepared to accept others. Since the making of a request is an output, and there are no output guards, this must be done using a **PAR**. There are two distinct types of request that might come in: over a different crosslink to the one over which a request is being made, or over the same one in the opposite direction. (Remember that crosslinks are bidirectional.) The first variety is easily dealt with: we simply run a process in parallel with the one making the request which fields requests from third parties and (for simplicity) turns them down. The second variety is handled by making a request, and accepting the reponse, in parallel. The response received may be **yes** or **no**, or it may be a request from the other process. In the last case we note that the other node will receive our message as the "response" to our message, so the situation is quite symmetric and both processes are "aware" of the situation. We therefore deem the messages to have been accepted as a swap. In other words, if two nodes more or less simultaneously decide to try and send each other messages, these are swapped. When a node receives a swap message it makes a repeat call of the routing procedure to deal with it.

Note that a node will accept a swapped message even when full, this being the exception to the limit **max_for_cl** which normally applies to inputting messages from crosslinks. Of course it is safe in this circumstance because the swapped message is simply occupying the space vacated by the message traded for it.

The program which deals with requests on other links is an alternative between a message to close it down and all the requests it is fielding. (Note that by the second half of (a) above there is no need for it to run them all in parallel when there are more than one.) The message to terminate this component is generated by the component which makes the request, immediately after doing so. This means that requests over other crosslinks will be refused until the response to the present one is recieved. However such refusals are permitted by the first half of (a).

```
CHAN halt:
PAR
  SEQ
    clout ! x
    halt ! ANY
  VAR y:
  SEQ
    clin ? y
    IF
      y = yes
        succ := TRUE
      y = no
        succ := FALSE
      TRUE
        SEQ
          repeat := TRUE
          newmess := y
          succ := TRUE
  VAR going:
  SEQ
    going := TRUE
    WHILE going
      PRI ALT
        halt ? ANY
          going := FALSE
        clin1 ? ANY
          clout1 ! no
```

If desired one could use the **repeat** variable to cause the node to try again if once refused over a crosslink. But this has not been done here since it seems of doubtful value.

The gaps we still have to fill in are those concerned with adding elements to the queue: this is of course very easy. To add **x**:

```
SEQ
  tail := (tail + 1) MOD (max + 1)
  queue[tail] := x
  qsize := qsize + 1
```

There is no provision in the program above for splitting boomerang messages before they reach their destination. This would presumably be done only when the node was reasonably empty. It is easy to include at the expense of complicating the control of "repeats" a little more. Our program also allows only for trying one crosslink for each message: this also could easily be changed, of course. (Especial care will be required if it is decided to make repeated attempts to send a message down a single crosslink, for divergence could easily be introduced.)

5. The protocol in practice; possible efficiency improvements. Our protocol should have good performance when it is not overloaded. It places relatively few restrictions on how its capacity can be used: for example a single process can take up a large proportion of the network with its own messages, if the others will let it. On the other hand its worst-case performance is very bad, reducing to the ring. One should bear in mind that there is no way a communications protocol can overcome too little bandwidth in the a section network (perhaps too little capacity on a link joining two distinct parts), but this one may be rather badly affected by such faults except in cases where its loading is controlled as in section 7 below.

Therefore we should expect it to be used in circumstances where the communication capacity is, in some sense, comfortably enough for the demand that is put on it. One should perhaps think of the protocol we have designed primarily as a way of avoiding deadlocks and starvations that might be caused by occasional coincidences or a particularly busy part of a network.

It should therefore be expected that most messages will be sent over crosslinks (save when the ring is the most direct path).

One property that the protocol *does not* have is preserving the order of messages between two fixed processes. The reason is that later message may succeed in following a shorter route. If, for any reason, one requires this some higher-level protocol will need to be used, probably assigning consecutive numbers to messages before sending them and re-ordering upon receipt. See the next section for one case where this is necessary.

The following are suggestions for making communication, particularly over crosslinks, more effective. Whether a particular one is worth adopting will depend to a great extent on the circumstances of a particular implementation.

- Greater use might be made of the swapping mechanism developed to avoid conflicts over crosslinks. It could be made legal for a node to reply to a request over a crosslink by sending a swapped message (if there is a suitable one in its queue). Presumably, when this was possible, the usual limit for the acceptance of crosslink requests would be relaxed.
- In practice messages will probably be quite large and, unless the probability of a crosslink request being accepted is very high, it would be inefficient to make the request by sending the message. Better might be to make the request a simple synchronisation and to leave the actual transmission till after a positive acknowledgement. Of course it would be necessary to ensure that no block in either direction could occur during this phase. (Then the proof of deadlock freedom follows by exactly the same argument as before.)
- Even though all packets have nominally the same size, in practice many (in particular ones like returning boomerangs) will be short and transmittable quickly. This consideration may also influence the previous item.
- The requirement that messages only be passed over crosslinks which reduce ring distance can be annoying, particularly when the network is quiet. For it must, even in a well-designed system, increase the expected path length of messages moderately above the shortest (and certainly requires considerable ingenuity on the part of a network designer if the penalty is not to be too high). It will also have the effect that many crosslinks are much more used in one direction than the other. We can get round it in practice by supplying each message, when it is first sent out, with a number of “wild cards”. One wild card is lost when the message is passed the “wrong” way over a crosslink (or alternatively, and perhaps preferably, when it is passed in a way that does not decrease the distance from it destination). Wild cards can be taken away for other reasons, but never added. A message with no wild cards

left cannot be passed over a "wrong" crosslink. In either case it is easy to prove, using adjusted variants, that no message can be passed around infinitely, so each must reach its destination. Nodes might vary the number of wild cards given to messages, dependent possibly on distance from destination or traffic conditions. Wild cards have some similarity to the "hop counts" traditionally used in networking. But there is the obvious difference that in our system no message is lost (when a message's hop count becomes zero it is thrown away). (The idea of wild cards is due mainly to Michael Goldsmith.)

- Crosslinks were introduced as a way of moving data between nodes not linked by the ring. In this form they still provide no way for a node to pass a message directly back to its immediate predecessor in the ring. In fact there is no reason why crosslinks (presumably one directional ones) should not exist to do this. However we should note that such a situation requires a total of three channels between two adjacent processors, rather than the usual, and economically implementable, one each way. If using transputers with one two-way link, one would have to multiplex these, see [J]. Fortunately, for different reasons in the cases of the ring channel and the crosslink ones, we can use the more efficient multiplexing technique which replaces each channel with a one-place buffers.

- While a node it trying to send one of its own messages over a crosslink it automatically rejects all requests that come to it, even though it might have plenty of space. The "best" way round this is probably to make the program handling these requests accept in suitable circumstances, being careful never to accept too many messages or to get into an infinite loop. Another, simpler, approach is to delay (probably via the timer) the acceptance of requests on other crosslinks while the request was being made. This could not introduce deadlock, since such delays are transparent to the semantics where deadlock was analysed, and should usually have the desired effect.

When there is no Hamiltonian circuit. The protocol discussed above depends on the existence of a Hamiltonian path round the nodes. To get round this where none exists one can place more than one node in each physical processor. As an extreme example, given a spanning tree for any connected network, a "circuit" can be found which uses only the edges of the spanning tree: each processor contains as many nodes as there are edges incident on it in the spanning tree. Figure 1 illustrates this. Circles represent transputers, unbroken lines the structure of the tree, discs represent nodes and rows of arrowheads the ring connections. The difference in sizes between nodes is artistic licence. Crosslinks, both inter- and intra-transputer, are omitted.

One could either provide each user process with channels to all the nodes on his transputer or to only one (so that the other nodes have no external connections). The former is more efficient but requires the users to know about the detailed structure of the service.

Note that each directed edge is used precisely once. Since a ring connection only uses one channel this means that no multiplexing is required unless there are also crosslinks on the same edges. There would be little point in implementing the latter, since the ring provides for two-way communication over such edges.

There would presumably be crosslinks between all pairs of nodes on the same processor. Crosslinks between processors could either be between individual pairs of nodes (making use of internal crosslinks) or multiplexed between every pair. Which solution was more effective would

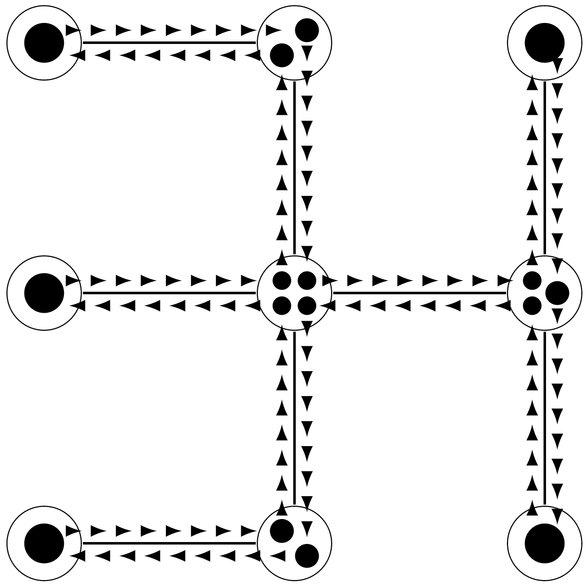

Figure 1: A tree converted into a ring

depend on such factors as whether "wild cards" were in operation. (If they were not the protocol would impose a constraint on the use of internal crosslinks.)

6. The "fair ALT" requirement. At several points in the program there are alternative statements where it is conceivable that one of the several inputs would always be possible. If this one were always accepted in preference to others it could result in a message not being delivered or a message never being accepted by the service. As mentioned earlier these aspects of correctness are dependent on each **ALT** polling each of its input channels every so often if it is executed an infinite number of times. (Of course this does not apply to **PRI ALT**.)

Unfortunately this fairness requirement is not part of the usual semantics or implementation of **ALT**: the former is nondeterministic and the latter is **PRI ALT**. However there are several ways in which it can be satisfied. Two alternative methods are given here, each of which is based on the idea that polling should begin at different points in the list each time, working cyclically round. One could either (a) advance the starting point by one each time, or (b) begin each time at the place after the successful guard of the last execution. Suppose we have an alternative

```
ALT
  g0
    P0
  g1
    P1
    . . .
    . . .
  gn-1
    Pn-1
```

which we wish to make fair in this sense. We then find new variable names which do not appear free in the whole program: two **i,j** for solution (a) and one **i** for solution (b). Outside the loop constructs with respect we want to attain fairness we declare the variable(s) and initialise **i** to 0 (or any other number less than n). Then we simply replace the alternative with, for (a), the code:

```
SEQ
  PRI ALT
    (i ≤ 0) & g_0
      j := 0
    (i ≤ 1) & g_1
      j := 1
       . . .
       . . .
    g_{n−1}
      j := n − 1
    (i > 0) & g_0
      j := 0
    (i > 1) & g_1
      j := 1
       . . .
       . . .
    (i > n − 2) & g_{n−1}
      j := n − 2
  CASE j
    0
      P_0
    1
      P_1
     . . .
     . . .
    n − 2
      P_{n−2}
    n − 1
      P_{n−1}
  i := (i + 1) MOD n
```

(If some of the original guards have boolean components one simply adds an appropriate conjunct to that component: thus **b & g** might become **((i** $\leq$ k**) AND b) & g**.)

Note here that, for each value of **i**, exactly n (consecutive) guards are enabled, one corresponding to each of the original guards. In the code above the indicator of where to start poling is simply advanced by one each time.

Solution (b) is arguably superior, since it guarantees that no guard can be activated twice while there is another one waiting. (Notice that in (a), it is possible that guard $\mathbf{g}_{n-2}$ can be activated on the first $n-1$ passes, even though $\mathbf{g}_{n-1}$ is continuously ready.

$$
\begin{array}{l}
\mathbf{SEQ} \\
\quad \mathbf{PRI\ ALT} \\
\quad\quad (\mathbf{i} \leq \mathbf{0})\ \&\ \mathbf{g}_0 \\
\quad\quad\quad \mathbf{i := 1} \\
\quad\quad (\mathbf{i} \leq \mathbf{1})\ \&\ \mathbf{g}_1 \\
\quad\quad\quad \mathbf{i := 2} \\
\quad\quad\quad \ldots \\
\quad\quad\quad \ldots \\
\quad\quad (\mathbf{i} \leq n-2)\ \&\ \mathbf{g}_{n-1} \\
\quad\quad\quad \mathbf{i :=}\ n-1 \\
\quad\quad \mathbf{g}_{n-1} \\
\quad\quad\quad \mathbf{i := 0} \\
\quad\quad (\mathbf{i} > \mathbf{0})\ \&\ \mathbf{g}_0 \\
\quad\quad\quad \mathbf{i := 1} \\
\quad\quad (\mathbf{i} > \mathbf{1})\ \&\ \mathbf{g}_1 \\
\quad\quad\quad \mathbf{i := 2} \\
\quad\quad\quad \ldots \\
\quad\quad\quad \ldots \\
\quad\quad (\mathbf{i} > n-2)\ \mathbf{g}_{n-1} \\
\quad\quad\quad \mathbf{i :=}\ n-1 \\
\quad \mathbf{CASE\ i} \\
\quad\quad \mathbf{1} \\
\quad\quad\quad \mathbf{P}_0 \\
\quad\quad \mathbf{2} \\
\quad\quad\quad \mathbf{P}_1 \\
\quad\quad \ldots \\
\quad\quad \ldots \\
\quad\quad n-1 \\
\quad\quad\quad \mathbf{P}_{n-2} \\
\quad\quad \mathbf{0} \\
\quad\quad\quad \mathbf{P}_{n-1}
\end{array}
$$

Simpler solutions to the problem of making alternatives fair are possible when the input channels are the components of an array: Jones [J1]. One could clearly modify the above ideas to provide for cases where, for example, one wishes to give one guard priority always, but to have fairness between the others.

Naturally, one should *develop* one's program without the above machinery and only apply the above transformations at the end, where necessary. In the program we are developing it is only necessary to apply either (a) or (b) to the main, outer alternative. (The only other alternative is already, and correctly, prioritised.)

7. Implementing channels and buffers through the service. Suppose we are given a set of processes running on transputers where it is not possible to map the logical channels onto the physical ones. Given a service satisfying the correctness conditions imposed earlier, it is relatively simple to implement *either* one place buffers in place of arbitrary sets of channels in a network *or*, if one is prepared to carry out a simple transformation on the code of the processes in the network, actual handshaken channels.

Jones [J] gives details of this. In outline the service described above is augmented by a process at each node which (a) accepts messages from the user and transmits them to the service (waiting for an acknowledgement message before accepting another on the same channel) and (b) accepts messages from the service (it has one space for each channel it handles), delivers them on appropriate channels to the user and sends an acknowledgement message back. The fact that it has a space for each channel and that all messages are acknowledged means that the requirement that all messages from the service are accepted, is satisfied *whether or not* the user process ever accepts a message. It is the intermediate process which is playing the role of "user" to the service, not the real user. Thus the requirement that each user process must accept every message that arrives is no longer necessary, which is just as well if we want these to be generally applicable "transformations".

Of course, replacing a channel by a one place (or any other length) buffer is not in general semantic-preserving: sometimes it can have disasterous results. (The discovery of situations where it *is* safe is a subject of ongoing research.) If it is desired to implement a particular "channel" as such, each output by a user on that channel is immediately followed by a wait for an acknowledgement. On such channels the acknowledgement used by the augmented service is transmitted back to the user.

One can, of course, mix these two modes of communication in the same network. One can safely combine communication of this form and normal communication along channels not used by the service.

To implement a longer buffer (i.e., more than one place) in place of a channel one does essentially the same thing, but is careful not to let the overtaking which can occur in the service be transmitted on to the user. (Perhaps all messages are given a number.) If one is prepared to make do with a buffer that can vary rather arbitrarily in length, one can avoid sending back an acknowledgement to *every* message.

It is tempting to let the acknowledgements take over the role of boomerangs. But it is not, in general, safe to do this. For if (as is now legal, see above) the ultimate user at the far end of a channel declines, forever, to accept the message that has been sent to him no acknowledgement would come back. This could mean that a second message, which the originator of the first wishes to send, may never be accepted. (There are other difficulties as well.)

Note that, in implementing channels (as opposed to buffers) we crucially used the asymmetry that no output guards may appear in alternatives.

In the type of use described in this section we have an easily obtained upper bound on the total number of messages that might be in transit at any time. This should allow the design of enough capacity into the network to stop it becoming too full as a whole. (One probably cannot prevent small parts of the system becoming gummed up occasionally, but should expect it to recover from this.)

8. Conclusions. This paper has hopefully illustrated how a complicated system can be developed in a logical way from its requirements and sound mathematical principles. In my opinion it would be practically impossible to understand how the eventual program works (consisting, as it does, of the parallel composition of a lot of far from simple agents interacting in subtle ways) without having seen its development.

As far as deadlock was concerned the main technique used was to first develop a relatively simple system which was deadlock free and then extend it in such a way that (essentially) no extra blocks were introduced and divergence was avoided. This meant that the old deadlock analysis still applied. Essentially the same ideas were used for the other aspects of correctness: for example the simple "variant" of moving a message closer round the ring to its destination.

Whether the particular system developed is useful in practice still has to be discovered. As has been pointed out there are many variations on the basic pattern which will affect efficacy.

Acknowledgements. This paper was inspired by a conversation with Geraint Jones, during which he gave me the problem and extracted from me some of the main ideas. Since then my ideas have been developed and refined during many conversations with him, Michael Goldsmith and Ben Potter.

References.

[BR] S.D. Brookes and A.W. Roscoe, *Deadlock analysis in networks of communicating processes, revised version*, to appear.

[H] C.A.R. Hoare, *Communicating sequential processes*, Prentice Hall, 1985.

[inmos] inmos Ltd., *The occam programming manual*, Prentice Hall, 1985.

[inmos1] inmos Ltd., *The occam-II programming manual*, Prentice Hall, to appear.

[J] G. Jones, *On guards*, this volume.

[J1] G. Jones, a contribution to the *occam* computer newsgroup, 1987.

[RD] A.W. Roscoe and N. Dathi, *The pursuit of deadlock freedom*, Oxford University Computing Laboratory technical monograph PRG-57 (also to appear in Information and Computation).

[RH] A.W. Roscoe and C.A.R. Hoare, *The laws of occam programming*, Oxford University Computing Laboratory technical monograph PRG-53.

[T] A.S. Tannenbaum, *Computer networks,* Prentice Hall 1981.