Report EUR-CS-94-3
May 1994

The Hamlet
Application Design Language
On the Implementation of Synchronous Channels

Maarten R. van Steen
Erasmus University, Faculty of Economics
Department of Computer Science
P.O. Box 1738, 3000 DR Rotterdam
e-mail: steen@cs.few.eur.nl

Abstract

The graphical Hamlet Application Design Language (ADL) has been developed to support the construction of parallel applications. The language is based on a notion of processes communicating by means of message-passing. One of the goals of ADL is that its implementation should allow for automated parallel code generation. However, not every communication construct in ADL has an evident counterpart in present-day target languages. In this report, attention is paid to the implementation of synchronous message-passing between multiple senders and multiple receivers. It is shown how an arbitrary (sender,receiver) pair can be selected in the form of centralized algorithms. The main part of the report, however, focusses on an efficient distributed solution.

keywords: parallel computing, distributed algorithms, distributed synchronization.

Contents

1

Introduction

In this report we shall concentrate on some of the implementation issues of the Hamlet Application Design Language (ADL). In particular, we focus on the development of a distributed implementation for one of its communication mechanisms: so-called synchronous channels. The semantics of synchronous channels are described in Section 1.1. They are, in fact, the only aspects that need to be known about ADL in order to understand the material presented in this report. Further information on ADL can be found in:

> M.R. van Steen, "The Hamlet Application Design Language: Introductory Definition Report." Technical Report EUR-CS-93-16, Erasmus University Rotterdam, Department of Computer Science, December 1993.

In the remainder of this chapter we introduce a notation that allows us to express solutions in terms of this target programming model. We shall restrict ourselves to a rather informal introduction to our notation, anticipating that most constructs used in the examples are self-explanatory.

The report is further organized as follows. In Chapter 2 we discuss a number of alternative solutions which are all based on centralized decision making. The bulk of the report, however, can be found in Chapter 3. There, a fully distributed solution is explained, down to the level of skeleton code. The presented solution has been implemented on simulated hardware, which permitted us to obtain insight in the algorithm's complexity. the results of these simulations are discussde in Chapter 3 as well.

1.1 Problem description

Consider an ADL synchronous channel connecting a collection **S** of senders to a collection **R** of receivers. The semantics of synchronous channels are such that if at time instant T, $M = |\mathbf{S}_T|$ senders and $N = |\mathbf{R}_T|$ receivers want to communicate, $|M - N|$ (sender,receiver)-pairs are selected nondeterministically and a message is transferred from sender to receiver. Consequently, if $|\mathbf{S}_T| > |\mathbf{R}_T|$ a total of $|\mathbf{S}_T| - |\mathbf{R}_T|$ nondeterministically selected senders will remain blocked, and likewise, if $|\mathbf{S}_T| < |\mathbf{R}_T|$ a total of $|\mathbf{R}_T| - |\mathbf{S}_T|$ nondeterministically selected receivers will remain blocked. In this report we shall mainly concentrate on the problem of devising a distributed implementation

of a synchronous channel for the case that $|\mathbf{S}| > 1$ and $|\mathbf{R}| > 1$. In the next chapter, the case that either $|\mathbf{S}| = 1$ or $|\mathbf{R}| = 1$ is further examined, as well as centralized solutions for the general case.

ADL designs are eventually implemented into what we refer to as the *target language*. At present, it is anticipated that this target language bares strong resemblance to a CSP-like language such as occam or C augmented with occam-like constructs. Therefore, in order to express our implementations we employ a C-like programming notation that allows us to express processes that communicate by means of synchronous, unidirectional **links** adhering to semantics similar to those of channels used in CSP. In addition, we assume that the target language allows a process to be subdivided into a number of threads that can communicate by means of shared memory. The means to provide mutual exclusive access to shared data structures is assumed to be based on a combination of mutex and condition variables.

1.1.1 Interprocess message-based communication

We assume that the target language is based on the concept of processes communicating through synchronous unidirectional links operating in simplex mode. We assume that links of the target language are strongly typed and adhere to the semantics of synchronous message passing. In our notation, a link `data` for communicating messages of the type `MessageType` is declared by writing

```
link( MessageType ) data;
```

In order to declare links that are only used for synchronization purposes, i.e. those through which so-called null-messages are communicated, we use the special data type `void`:

```
link( void ) control;
```

Like any other data type, links can be indexed by constructing arrays. For example,

```
link( MessageType ) data[M];
```

declares an array of M identical links `data[0] ... data[M-1]` for communicating messages of type `MessageType`. Sending a message `message` across such a link is denoted in the usual way by means of a "shriek":

```
data ! message
```

Similarly, receiving a message is written as a query:

```
data ? message
```

A process is declared by using the reserved word `process`, and specifying as part of the signature the links through which it communicates with other processes. As an example, a simple producer-consumer pair of processes can be specified as follows:

```
link( MessageType ) data;

process producer ( data ){
  MessageType message;
  ...
  data ! message;
}

process consumer ( data ){
  MessageType message;
  ...
  data ? message;
}
```

In a similar fashion to data types, we can also declare arrays of identical processes. The index of a process that has been declared as an element of an array can be retrieved by means of the pseudo-variable `self`. To illustrate, consider the following declaration:

```
link( MessageType ) data[M];

process P[M]( data[self], data[(self + 1) % M] ){
  MessageType message;
  ...
  data[self] ? message;
  data[ (self + 1) % M ] ! message;
}
```

In this case, we have declared a ring of M processes where each process `P[i]` receives a message from its lefthand neighbor through link `data[i]`, and passes this message to its righthand neighbor through link `data[(i+1) % M]`. The fact that a process communicates with the outside world by means of an array of links is denoted by the pseudo-variable `ALL`:

```
link( MessageType ) data[M];

process server ( data[ALL] ){
  MessageType message;

  for( int i = 0; i < M; i++ ) data[i] ! message;
}
```

As in every CSP-based language, we assume that a process can non-deterministically select an incoming message. This is specified by means of a `select`-statement in combination with guarded commands expressed through so-called `when`-clauses:

```
link( MessageType ) data[M];

process receiver ( data[ALL] ){
  MessageType message;

  select(){
    when( data[0] ? message ){ ... };
    when( data[1] ? message ){ ... };
      ...
    when( data[M-1] ? message ){ ... };
  }
}
```

In this case, the receiver selects exactly one message from those links for which a

message is pending. If no messages have been sent, the process blocks until the first one arrives. A `when`-clause must contain precisely one input statement, but may be augmented with ordinary boolean expressions. Like occam, it is not permitted to include an output-statement in a `when`-clause.

The above specification can be given more concisely by introducing an index variable as parameter of the `select` operator:

```
link( MessageType ) data[M];

process receiver ( data[ALL] ){
  MessageType message;

  select( i : 0..M−1 ) when( data[i] ? message ){ ... };
}
```

1.1.2 Communication by shared data

As we have said, we assume that our target language supports a combination of mutex and condition variables that allows us to construct monitors. In particular, this will allow us to express mutual exclusive access to critical sections in a more or less structured fashion. A critical section of code is always associated with exactly one mutex variable. For example,

```
mutex cs;

process P(){
    enter( cs );
    ...
    leave( cs );
}
```

describes a process `P` that enters a critical section identified by the mutex variable `cs`, and later leaves this section again. Critical sections are assumed to be non-reentrant. In other words, if a process requires to enter a critical section in which it already is, the process will be postponed forever. In practice, this means that constructs such as:

```
mutex cs;

process P(){
    enter( cs ); /* first time - okay */
    ...
    enter( cs ); /* second time - deadlock */
    ...
    leave( cs );
    ...
    leave( cs );
}
```

are to be considered as erroneous.

Critical sections can have one or several associated condition variables. A condition variable can only be declared in combination with its associated critical section, for example, as in:

```
mutex     cs;
condition cv(cs);
```

which declares a condition variable `cv` associated with the critical section identified by the mutex variable `cs`. There are only two operations available for condition variables: `signal` and `wait`. These operations can only be called while the calling process is in the critical section associated with the condition variable to which the operations are applied.

Invoking the operation `wait(cv)` suspends the calling process and allows exactly one other process to enter the critical section associated with `cv`. If a process is also in any other critical section, then the latter will remain protected against access by any other process. A suspended process is reactivated by invocation of the operation `signal(cv)`. As with `wait(cv)`, this operation can only be invoked when a process, say *P*, has entered the critical section associated with `cv`. If several processes were suspended, only the one which has been suspended the longest, say *Q*, will be activated. The instant process *P* leaves the critical section, *Q* regains control and continues as the only process currently in the critical section. If several processes had been woken up, then only one of them will actually reenter the critical section. The others will remain suspended until the critical section is no longer occupied.

2

Centralized solutions

As we have mentioned, we shall only consider implementations of ADL designs which do not make use of select states in state-transition machines, and which are also based on blocked communication. In this chapter we take a look at centralized solutions for synchronous channels. The following three cases are distinguished: (1) there are multiple senders, but only a single receiver, (2) there is a single sender, but multiple receivers, and (3) there are multiple senders and receivers. As we shall show, the first two cases are relatively easy to implement efficiently. Using a centralized solution in the third case is also fairly straightforward, although this may not be an acceptable *general* solution. For this reason, we shall elaborate on a distributed version in the next chapter.

2.1 Multiple senders, single receiver

In the case that there are multiple senders and only a single receiver, we can directly employ the semantics of the target language by using a single communication link for each sender. The principle is shown in Figure 2.1.

The nondeterministic behavior introduced by the ADL synchronous channel is implemented by means of a select statement at the receiver's side. The general target code in the case that there are M senders, then takes the following form:

```
link( MessageType ) data[M];

process sender[M] ( data[self] ){
  MessageType message;

  data[self] ! message;
}

process receiver ( data[ALL] ){
  MessageType message;

  select( i : 0..M-1 ) when( data[i] ? message );
}
```

This implementation is straightforward. What we have done is replaced the single ADL synchronous channel by M separate links of the target language. Note that the semantics of ADL are completely preserved.

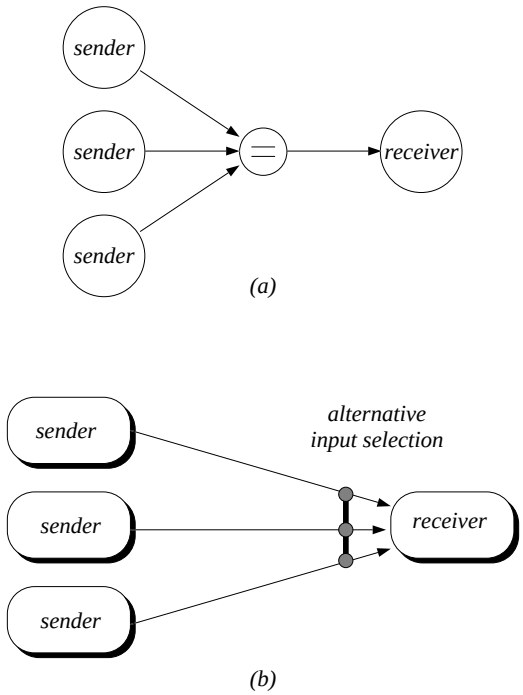


Figure 2.1: Translation of a synchronous link connecting multiple senders and a single receiver.

2.2 Single sender, multiple receivers

Now consider the case that there is only a single sender but possibly multiple receivers. Then, sending a message via a synchronous channel implies that a possible receiver should be selected non-deterministically. As our target language prohibits the use of output-statements in `when`-clauses, we devise a mechanism by which each receiver informs the sender when it is willing to receive a message. To this aim, we make use of an additional control link as shown in Figure 2.2(b).

In this case, the sender simply waits until there is a receiver that announces that it wants to communicate. The selection of the receiver is done by means of guarded input on all the control links. This leads to the following skeleton code.

```
link( MessageType ) data[N];
link( void ) ctrl[N];

process sender ( data[ALL], ctrl[ALL] ){
  void ready;
  MessageType message;

  select( i : 0..N-1 ) when( ctrl[i] ? ready ) { data[i] ! message; }
}
```

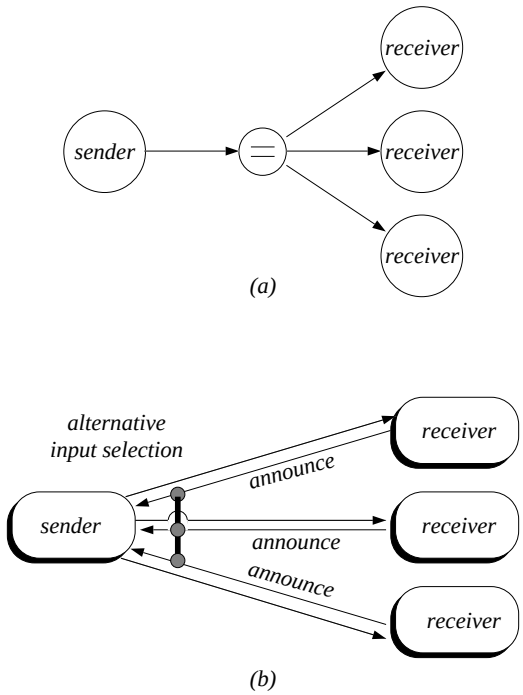


Figure 2.2: Translation of a synchronous channel with multiple senders and one receiver.

```
process receiver[N] ( data[self], ctrl[ALL] ){
  void ready;
  MessageType message;

  ctrl[self] ! ready;
  data[self] ? message;
}
```

2.3 Centralized solutions for multiple senders and receivers

In the case of multiple senders and receivers, there are two general approaches: solving the communication by means of either a centralized or distributed solution. In this section we shall take a look at two centralized solutions. A distributed solution is discussed in the next chapter. In the case of a centralized solution, we make use of a separate server process which matches a sender and a receiver. Two approaches can be followed. In the first case, the server can actually receive a complete message and pass it on to a waiting receiver. In the second case, we can use a server merely to find a suitable (sender,receiver)-pair. Let us start with considering the first situation.

2.3.1 A data-passing server

In this case, a sender simply sends a message to the server and subsequently waits until the server acknowledges the receipt of the message by a receiver. The communication

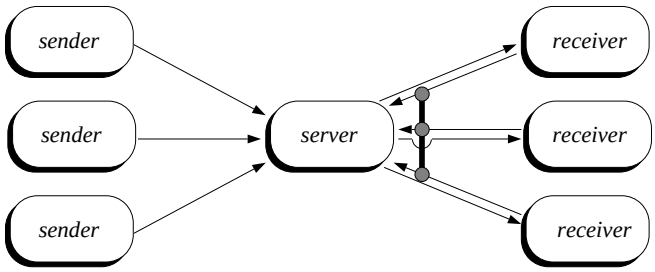


Figure 2.3: A centralized solution for many-to-many communication using a data server.

between the server and a receiver is similar to the communication in the case of a single sender and multiple receivers: the receiver starts with indicating that it is prepared to receive a message and subsequently waits until one is sent. As we shall discuss below, our solution requires that there is only a single data link between every sender and the data server. The architecture of this solution is sketched in Figure 2.3.

The implementation requires M data links for the senders, and N data links for the receivers. In addition, we need a total of N control links in order for the receivers to announce their willingness to communicate. The skeleton implementation code for the senders and receivers, respectively, is as follows.

```
link( MessageType ) senddata[M];
link( MessageType ) recvdata[N];
link( void ) recvctrl[N];

process sender[M]( senddata[self] ){
  senddata[self] ! message;
}

process receiver[N] ( recvdata[self], recvctrl[self] ){
  recvctrl[self] ! ready;
  recvdata[self] ? message;
}
```

The order in which the data server attempts to match a sender and a receiver is important. Let us assume that the data server first waits until a sender is prepared to transmit data. This can be implemented by means of a series of guarded input statements. However, as soon as the server has received data from a sender, the latter still needs to wait until a receiver has been identified as well. Consequently, we would need an additional control link to synchronize the sender when a receiver has been found. This control link can be omitted if we let the server identify a receiver first, and then wait until a sender is located. As soon as the sender transmits its data to the server, the latter can immediately forward the data to the previously identified receiver. These observations then lead to the following skeleton code for the data server.

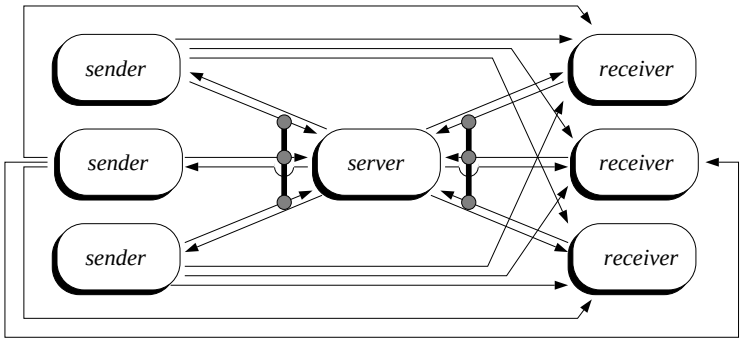


Figure 2.4: A centralized solution for many-to-many communication using a control server.

```
process server ( senddata[ALL], recvdata[ALL], recvctrl[ALL] ){
  while ( TRUE )
    select( i : 0..N−1 )
      when( recvctrl[i] ? ready )
        select( j : 0..M−1 )
          when( senddata[j] ? message )
            recvdata[i] ! message;
}
```

It is thus seen that we need a total of $M + 2N$ links. The number of communications per synchronization adds up to a total of 2 data communications and 1 control communication.

2.3.2 A control server

The second centralized solution is that of an arbiter: the server merely acts as a means to couple a sender and a receiver. To that aim, the identification of a sender and a receiver is passed on to the server, who, as soon as a pair has been found, exchanges the identifications. This will allow the sender to directly transmit its data to the identified receiver. The general architecture of this solution is depicted in Figure 2.4.

In order to exchange the identities of senders and receivers, each communicating process needs to communicate with the control process by means of two unidirectional links. In addition, a total of $M \cdot N$ links are required to transmit data from any sender to any receiver. This then leads to the following skeleton code.

```
link( MessageType ) data[M][N];
link( int ) send_id[M][2];
link( int ) recv_id[N][2];

process sender[M] ( data[self][ALL], send_id[self][ALL] ){
  MessageType message;
  int rcvr;

  send_id[self][0] ! self;
  send_id[self][1] ? rcvr;
  data[self][rcvr] ! message;
}

process receiver[N] ( data[ALL][self], recv_id[self][ALL] ){
  MessageType message;
  int sndr;

  recv_id[self][0] ! self;
  recv_id[self][1] ? sndr;
  data[sndr][self] ? message;
}

process server ( id_recv[ALL], send_id[ALL][ALL], recv_id[ALL][ALL] ){
  int sndr, rcvr;

  while( TRUE )
    select( i : 0..M-1 )
      when( send_id[i][0] ? sndr )
        select( j : 0..N-1 )
          when( recv_id[j][0] ? rcvr ){
            send_id[i][1] ! rcvr;
            recv_id[j][1] ! sndr;
          }
}
```

An alternative solution is to make the server completely symmetric by letting it wait for either a sender or receiver to initially identify itself. However, this will in no way affect the utilization of resources, for which reason we have chosen the solution shown above. It can easily be seen that the number of communications required per data exchange, adds up to 5, consisting of 4 communications with respect to exchanging identifications, and 1 for the actual data transmission.

2.3.3 Comparing the two solutions

With respect to the number of required links, and the number of communications, it would seem as if the centralized control server is worse than the centralized data server. However, one has to realize that the latter requires two data communications, whereas the control server solution requires only the exchange of identifications, and a single data transmission. In those situations in which the size of the actual data to be transmitted is large, the control server may outperform the data server.

3

A distributed solution

In this chapter we shall fully concentrate on a distributed solution for synchronous channels having multiple senders and multiple receivers attached to them. In our solution we organize the communicating parties as a logical ring. The outline of this solution is presented first, including detailed design aspects. The skeleton code provides a precise description, and is discussed as well. We shall end with a discussion and analysis of our solution.

3.1 The global architecture

In order to come to an efficient implementation of the semantics of ADL synchronous channels, we organize the senders and receivers into a logical ring and essentially adopt a token-based protocol for exchanging data. In particular, we let an **envelope** circulate counter-clockwise around the ring, whereas a process that requires the envelope will issue a **request** clockwise around the ring. The envelope can either be *full* or *empty*. This global architecture is shown in Figure 3.1.

Each process essentially consists of two components. The subprocess (called the *main thread*) represents the global behavior of either a sender or a receiver. The second component consists of two subprocesses (referred to as respectively the *put thread* and the *get thread*) and is responsible for getting the envelope and forwarding it at the appropriate time. These three subprocesses communicate by means of shared memory instead of links for reasons to be explained further below. For the sake of clarity, we shall denote each subprocess as a *thread* (as this is actually the way they will be implemented), which jointly comprise an actual process.

3.1.1 The behavior of a process

The global behavior of our solution can now be explained by taking a closer look at a receiver and sender, respectively. To that aim, we say that a process becomes *active* if it wants to either send or receive data. Otherwise, the process is said to be *inactive*.

Receiver. When a receiver becomes active, it is prepared to accept *any* incoming data. In our solution, this means that the receiver should get a hold of a *full* envelope. Assuming that the envelope is currently not at the receiver, the receiver will issue a

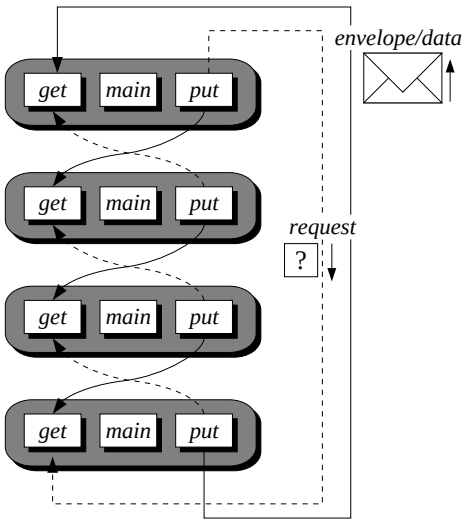


Figure 3.1: The global architecture of a distributed implementation for synchronous channels.

request for a full envelope to its lefthand neighbor. From that moment on, it simply waits until the envelope eventually arrives. As soon as the envelope arrives, or when the envelope was already located at the receiver, we need to distinguish two situations:

1. The envelope is full: in this case, the receiver can empty the envelope, and communication is considered to be finished.
2. The envelope is empty: in this case, the receiver will have to wait until a request for an empty envelope arrives (which can only come from a sender), forward the envelope to its righthand neighbor, and wait for it to arrive again. In order to ensure that the latter happens, the envelope will be *marked*.

The mechanisms for forwarding an envelope will be described in detail below. An important observation is that we never forward the envelope unless there is a good reason to do so. In other words, the receiver should know for certain that there is a sender on the ring willing to fill the envelope. In this way, we avoid the situation of a continuously circulating envelope – a solution generally adopted for many token-based protocols.

Sender. The sender's situation is almost symmetrical to a receiver. When becoming active, a sender should get hold of an *empty* envelope. Again, if we assume that the envelope is not at the sender's site, the sender will issue a request for the empty envelope to its lefthand neighbor and wait until the envelope arrives. When the envelope arrives, or when it was already available when the sender became active, two situations are to be considered:

1. The envelope is empty: in this case, the sender may fill the envelope, and subsequently wait until it receives a request for a full envelope (which can only

come from a receiver). As soon as the sender is certain that there is a receiver on the ring, it forwards the envelope and the communication is considered to be finished.

2. The envelope is full: this can only happen when there was another sender on the ring as well and which had previously filled the envelope. In that case, the sender should forward the envelope to its righthand neighbor. Similar to the case of a receiver, the sender *marks* the envelope and passes it on.

Again, note that we only forward the envelope if it is really needed. Another point that we shall explain further below, is that the sender will not issue a request when it finds the envelope already filled. Instead, the envelope is *marked.*

Of course, a process need not be active at all. In that case, the envelope is simply forwarded if there is a need to do so. In other words, if the envelope is empty there should be sender on the ring, or when it is full forwarding only takes place when there is a receiver. Let us now take a closer look at the way the envelope is circulated across the ring. To that aim, we consider how requests are forwarded, and how the envelope is forwarded. Also, we consider the actual acceptance of an envelope.

Forwarding requests. As we have mentioned, a process can issue two types of requests: one for an *empty* envelope, and one for a *full* envelope, respectively. Whenever a process receives a request (which can only come from its righthand neighbor), it will forward this request if and only if (1) the process currently does not have the envelope in its possession, and (2) it had not previously forwarded a similar request. In this way, it is seen that requests are not accumulated through the ring, but instead, if a process has recorded that the envelope is requested, there may be several processes actually requiring the envelope.

Forwarding the envelope. Whenever a process wants to forward the envelope, a necessary and sufficient condition is that the process is certain that someone is actually in need for the envelope. Let us first assume that this is the case so that the process will forward the envelope. Three situations are distinguished:

1. The process itself is currently inactive. Assume the envelope is empty and that the process knows there is a sender in need of the envelope. If a request has arrived at the process for a *full* envelope, the envelope is marked by setting a boolean variable `requestFull` to `true`. This variable is located on the envelope. Likewise, there is also a boolean variable `requestEmpty` which is set whenever a full envelope is forwarded, but there is also an outsanding request at the forwarding process for an empty envelope. The envelope is then forwarded, and all administration local to the forwarding process regarding previously issued requests is cleared.

2. The process is an active receiver. Again, let us first assume that the envelope is empty. In this case, the process will forward the envelope when it knows that there is a sender on the ring. However, it should also ensure that the envelope eventually returns, preferably having been filled in the meantime. To

that aim, it increments a counter `numOfRcvrs` located on the envelope, indicating the number of active receivers that have passed the envelope while it was empty. Consequently, as long as this counter is non-zero, it is known that there is a receiver somewhere on the ring, and which is in need of a full envelope.

When the envelope is full when it got to the receiver, the receiver will first empty it, and subsequently inactivate. Consequently, forwarding proceeds according to situation (1).

3. The process is an active sender. The special situation that we need to consider here, is when the sender has received an already filled envelope. This can only happen if there was already a receiver on the ring so that the envelope should always be forwarded. However, the sender should also indicate that it is still in need of an empty envelope. Analogously to the situation of an active receiver with an empty envelope, the sender will increment a counter `numOfSndrs` which is located at the envelope. This counter reflects the number of senders that have passed a filled envelope, but which are in need of an empty one.

   When an empty envelope was passed to the sender, it will subsequently fill it and wait for a receiver. The envelope is then forwarded and the process becomes inactive again.

The necessary and sufficient conditions for forwarding an envelope can now be stated more explicit: (1) the envelope is empty, and either `numOfSndrs` is non-zero, or `requestEmpty` is `true`, or (2) the envelope is full, and either `numOfRcvrs` is non-zero, or `requestFull` is `true`. After possibly updating the values for the four markers on the envelope, the envelope is forwarded and local administration with respect to outstanding requests is cleared.

Acceptance of an envelope. The last behavioral aspect we need to consider is the actual acceptance of the envelope and updates of its markers. Again, we make a distinction between receivers and senders. If a filled envelope arrives at a receiver, the receiver will first decrement the counter `numOfRcvrs` if it had previously incremented it. Also, the marker `requestFull` is set to `false`. The envelope can then be emptied, after which forwarding is considered as described above. Likewise, if an empty envelope arrives at a sender, the process will decrement `numOfSndrs` if it had previously incremented it, and also clear the marker `requestEmpty`. Then, the envelope is filled and behavior proceeds as mentioned above.

3.1.2 Design and implementation aspects

As mentioned, the distributed solution can be implemented by distinguishing three threads per process, cooperating by means of shared data. The *main thread* represents the general behavior of a sender or receiver, whereas the two threads named *put thread* and *get thread*, respectively, form the core of the algorithm. The *put thread* is responsible for transmitting any information on the ring. In particular, it takes care of forwarding requests and the envelope. By contrast, the *get thread* is reponsible for accepting the envelope from the process' lefthand neighbor, or for receiving requests

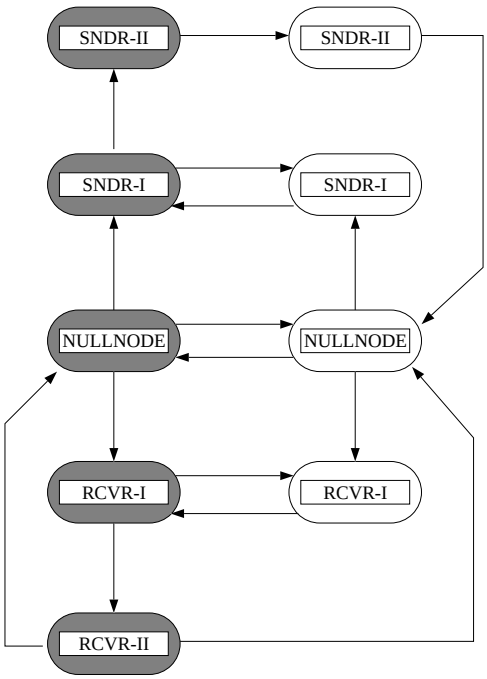


Figure 3.2: The state-transition diagram for a process.

from its righthand neighbor. The reason for making an explicit distinction between the two has everything to do with the synchronous nature of the communication links of our target language. To explain, consider the following situation.

Suppose a process P_i has just become active to which end it decides to issue a request for the envelope by sending a request to its lefthand neighbor P_{i-1}. If, by that time, process P_{i-1} has the envelope which it should forward on account of the fact that either `numOfRcvrs` or `numOfSndrs` was non-zero, P_{i-1} may simultaneously decide to forward the envelope to P_i. We will then find ourselves in the situation that P_i and P_{i-1} want to simultaneously send information to each other. Because we are dealing with synchronous communication, we will have created a deadlock. Deadlock in this case can be avoided if we implement a form of asynchronous communication by allowing a separate thread to deal with all incoming information.

Design of a process

The algorithm has been implemented as a state-transition machine on a per-process basis of which state information is maintained by cooperation of the get and put thread. The state-transition diagram is depicted in Figure 3.2. The shaded states represent the situation that a process has the envelope in its possession; the other states reflect that the envelope is somewhere else.

The following states and most important transitions are distinguished:

- **NULLNODE**: This represents an inactive process that will generally only forward requests and the envelope. The NULLNODE state is also the initial state of a

process. Only one process will, of course, start in the situation that it possesses the envelope.

- **SENDER-I**: A process enters this state the instant it becomes an active sender. The process remains in this state until the envelope is in its possession, and suited to be filled by the main thread.
- **SENDER-II**: A sending process currently filling the envelope, or otherwise waiting for a receiver to announce itself will remain in this state. This state can only be entered from state SENDER-I. As soon as a filled envelope can be forwarded to a receiver, will the process continue in state NULLNODE.
- **RECEIVER-I**: Similar to state SENDER-I, a receiver will enter this state the instant it becomes active. It will remain in state RECEIVER-I until the filled envelope has arrived.
- **RECEIVER-II**: While the envelope is being emptied, an active receiver will remain in this state. Regardless if the envelope can be forwarded or not, the process will continue in one of the two NULLNODE states.

Datastructures

In order to allow the three threads to communicate by shared data, each process has a separate critical region which is protected by means of single mutex variable. Assuming that there are a total of N_{proc} processes, we construct a data structure `node[i]` for each process P_i, and locate this structure at the same processor where P_i is placed. Each data object `node[i]` is protected against interleaved access by a mutex variable `data` by means of the following declaration:

```
typedef struct {
  mutex data;
  ...
} node[ NPROC ];
```

In order to keep track of the current state of a process, we add a boolean variable `have` to indicate whether or not a process is currently in possession of the envelope, as well as a variable `role` for distinguishing each of the above mentioned global states. In addition, we need to identify the period during which a main thread is actually processing the envelope, i.e. filling it when it is sender, or emptying the envelope when acting as a receiver, respectively. To that aim, we use a boolean variable `busy`. This leads to the following additional declarations:

```
typedef enum {NULLNODE, SENDER_1, SENDER_2, RECEIVER_1, RECEIVER_2} ROLE;

typedef struct {
  ...
  bool have; /* TRUE iff process is in possession of the envelope */
  bool busy; /* TRUE iff main thread is processing the envelope   */
  ROLE role; /* The present global state                          */
  ...
} node[ NPROC ];
```

Clearly, we have that if `busy` is `true`, then `have` should be `true` as well. The envelope itself is modeled by means of the following type declarations:

```
typedef enum { FULL, EMPTY } STATUS;

typedef struct {
  STATUS   status;
  bool     request[2];
  unsigned numOfWait[2];
} ENVELOPE
```

where `request[FULL]` and `request[EMPTY]` correspond to the previously mentioned boolean variables `requestFull` and `requestEmpty`, respectively, and the variables `numOfWait[FULL]` and `numOfWait[EMPTY]` correspond with the counters `numOfRcvrs` and `numOfSndrs`, respectively. The actual presence of the envelope at a process is captured by adding a field to each `node` structure:

```
typedef struct {
  ...
  ENVELOPE envelope; /* valid iff node[i].have == TRUE */
  ...
} node[ NPROC ];
```

Finally, the get thread and put thread will need to keep track of requests issued by their righthand neighbor, and the fact if they have already forwarded a request or not. Also, in order for a thread to justifiably decrement the counter `numOfWait` as found on the envelope, we'll have to keep track if a process previously incremented the counter or not. This leads to three additional boolean variables:

```
typedef struct {
  ...
  bool askedRight[2]; /* askedRight[FULL] (askedRight[EMPTY]) == TRUE   */
                      /* iff righthand neighbor requested full (empty)  */
                      /* envelope                                       */
  bool askedSelf[2];  /* askedSelf[FULL] (askedSelf[EMPTY]) == TRUE iff */
                      /* process forwarded request for full (empty)     */
                      /* envelope                                       */
  bool requested[2];  /* requested[FULL] (requested[EMPTY]) == TRUE iff */
                      /* process has incremented counter numOfWait.     */
  ...
} node[ NPROC ];
```

A question that remains to be answered is how exactly communication between the threads and processes takes place. If we were only dealing with inter-thread communication, it would suffice to use condition variables indicating that a certain event had taken place, after which data exchange could take place by means of manipulating the shared data. On the other hand, inter-process communication should be entirely based on the synchronous links of the target language. The two forms are hard, and practically impossible to combine when a thread should wait until either another process, or one of the threads to which it is associated, communicates. The solution is to let a thread waiting for *any* communication, wait on the presence of a message on a link. This implies that if the main thread wants to indicate its willingness to communicate to the associated get thread, it should do so by sending a message across a link that is *local* to the processor on which the two threads reside. For each process, we will now have a total of three links:

- a local link used by the main thread to indicate that it wants to either send or receive a message, or that it has finished processing the envelope.

- a link connecting a process to its lefthand neighbor, and through which the envelope is passed.
- a link connecting a process to its righthand neighbor, and through which it can accept requests for a full or empty envelope.

These three links are implemented by means of a separate common data structure (normally the accessibilty of the links would be declared by means of additional configuration information at start-up time).

```
typedef enum { WantToSend, WantToReceive, EnvelopeProcessed } MAINSIG;
typedef enum { GetFull, GetEmpty } REQUEST;

typedef struct {
  link ( MAINSIG  ) local;   /* signals sent by main thread           */
  link ( ENVELOPE ) ringEnv; /* envelope coming from lefthand neighbor */
  link ( REQUEST  ) ringReq; /* request coming from righthand neighbor */
} network[ NPROC ];
```

We assume that `network[i].local` refers to the local link at `node[i]`. Similarly, `network[right(i)].ringEnv` refers to the link between `node[i]` and its righthand neighbor for transmitting the envelope. Analogous interpretations are to be applied for `network[left(i)].ringEnv`, `network[right(i)].ringReq`, and `network[left(i)].ringEnv`.

Anticipating our discussion on actual implementation details, we need three condition variables to synchronize the threads that constitute a process. In the first place, in order for the put thread to eventually respond to any incoming information caught by the get thread, the latter will signal this event by means of a condition variable `newMessage`. The main thread will also have to be activated as soon as the envelope has arrived. This can be implemented by means of a condition variable `haveEnvelope`, which is signaled by the get thread as soon as it has received the envelope, and which is then suited for further processing by the main thread. Finally, whenever the main thread has just filled the envelope (in the case it behaves as a sender), it will need to be notified whenever a receiver has been detected. To that aim, we use a third condition variable `envelopeSent`. Together, this leads to the following expansion of each `node` structure:

```
typedef struct {
  ...
  unsigned  numOfMessages;
  condition newMessage( data );
  condition haveEnvelope( data );
  condition envelopeSent( data );
  ...
} node[ NPROC ];
```

The variable `numOfMessages` will be incremented by the get thread each time a new message arrives; accordingly, the put thread will decrement this variable each time it has responded to a signal from the get thread.

3.2 The skeleton target code

3.2.1 The main thread

The skeleton code for the main thread is actually quite simple. Assuming that a process exhibits an everlasting and alternating behavior consisting of a period of non-communication, and a period of communication, we can express this behavior by means of the following skeleton code.

```
process mainThread[NPROC]( node[self].local )
{
  bool sender;

  while( TRUE ){
    /* no communication */
    ...
    sender = ...; /* TRUE iff thread wants to send data */

    switch( sender ){
      TRUE  : network[self].local ! WantToSend;    break;
      FALSE : network[self].local ! WantToReceive; break;
    }

    enter( node[self].data );

    if(( !node[self].have ) ||
       ( sender  && node[self].envelope.status == FULL  ) ||
       ( !sender && node[self].envelope.status == EMPTY ) ){
      wait( node[self].haveEnvelope );
    }
    /* at this point the envelope is in possession of the main thread */

    ... /* fill or empty the envelope */

    node[self].envelope.status = ( sender ? FULL : EMPTY );
    network[self].local ! EnvelopeProcessed;
    if( sender ) wait( node[self].envelopeSent );
    leave( node[self].data );
  }
}
```

Of course, the details concerning the skeleton code above can only be given when the precise behavior of a process is known. With respect to ADL, this means that further information concerning the state-transition machine associated with the communicating process is required.

3.2.2 The get thread

The get thread per process can now also be detailed. The main role of the get thread is to receive any incoming message. In particular, it needs to react to messages sent by the main thread through the local link, the envelope which can be sent via its lefthand neighbor, and any requests from its righthand neighbors. As soon as a message has been received, the get thread will notify the put thread who is primarily responsible for further communication. The get thread can now be outlined as follows.

```
process getThread[ NPROC ](
    network[self].local, network[self].ringReq, network[self].ringEnv )
{
  MAINSIG  signal;    /* The signal as received   */
  REQUEST  request;   /* The request as received  */
  ENVELOPE envelope;  /* The envelope as received */

  while( TRUE ){
    select(){
      when( network[self].local ? signal ){
        /* The main thread is either requesting communication, or indicating
           that it has finished communication. The get thread will have to
           change the current role (when the main thread wants to
           communicate), or register that the envelope has been processed.
        */
        enter( node[self].data );
        switch( signal ){
        case WantToSend        : node[self].role = SENDER_1;   break;
        case WantToReceive     : node[self].role = RECEIVER_1; break;
        case EnvelopeProcessed : node[self].busy = FALSE;      break;
        }
      }

      when( network[left(self)].ringEnv ? envelope ){
        /* The envelope has arrived (through the lefthand neighbor). The get
           thread need merely register the fact and leave it up to the put
           thread to see what needs to be done.
        */
        enter( node[self].data );
        node[self].have = TRUE;
        node[self].envelope = envelope;
      }

      when( network[right(self)].ringReq ? request ){
        /* A request has been received (from the righthand neighbor). Again,
           only the fact needs to be registered; it is up to the put thread
           to see what the implications are.
        */
        enter( node[self].data );
        switch( signal ){
        case GetFull  : node[self].askedRight[FULL]  = TRUE; break;
        case GetEmpty : node[self].askedRight[EMPTY] = TRUE; break;
        }
      }
    }
    /* Just notify the put thread that a message has just been received. */

    node[self].numOfMessages++;
    signal( node[self].newMessage );
    leave( node[self].data );
  }
}
```

Note that the get thread is only responsible for recording the transition to either SENDER-I or RECEIVER-I. Also, because it is the only thread that can actually *receive* the envelope, it should also take care that the boolean variable `have` is set.

3.2.3 The put thread

The put thread actually forms the heart of our implementation. Before we take a closer look at its skeleton code, we consider some general functions that it should perform.

Passing requests

In the first place, the put thread is responsible for passing requests to its lefthand neighbor. We have chosen to encapsulate this functionality in a separate function as follows:

```
void issueRequest( unsigned caller, STATUS requiredStatus )
{
  if( !node[caller].askedSelf[requiredStatus] ){
    node[caller].askedSelf[requiredStatus] = TRUE;
    switch( requiredStatus ){
      case EMPTY: network[left(caller)].ringReq ! GetEmpty; break;
      case FULL:  network[left(caller)].ringReq ! GetFull;  break;
    }
  }
}
```

What is seen, is that if the put thread had not previously forwarded a similar request, it will now pass the required type of request to its lefthand neighbor.

Similarly, there is a function `forwardRequests` that simply checks if there are outstanding requests from a righthand neighbor which should be forwarded to the lefthand neigbor.

```
void forwardRequests( unsigned caller )
{
  STATUS  status  = FULL;
  REQUEST request = GetFull;

  do{
    if( node[caller].askedRight[status] &&
        !node[caller].askedSelf[status] ){
      node[caller].askedSelf[status] = TRUE;
      network[left(caller)].ringReq ! request;
    }
    status  = ( status  == FULL    ? EMPTY    : FULL    );
    request = ( request == GetFull ? GetEmpty : GetFull );
  } while( status ≠ FULL );
}
```

Passing the envelope

Whenever the put thread passes the envelope, it should do two things: (1) clear all its own local outstanding requests, and (2) update the administration on the envelope. To this aim, we distinguish two functions. The first one, `clearEnvelope`, merely decrements the counter `numOfWait` in case the calling process had previously incremented this counter. This leads to the following code:

```
void clearEnvelope( unsigned caller, STATUS requiredClearance )
{
  if( node[caller].requested[requiredClearance] ){
    node[caller].envelope.numOfWait[requiredClearance]--;
    node[caller].requested[requiredClearance] = FALSE;
  }
}
```

The second function describes the actual forwarding of the envelope, but only if this is actually supposed to happen. The actual decision is completely based on information that is available on the envelope itself. In particular, if there was either a general request, or the counter `numOfWait` is positive, then the envelope will be forwarded. This implies that the local administration of a process needs to be updated first, and possibly also the information on the envelope. Furthermore, a distinction should be made with respect to the *reason* the envelope is to be forwarded. In the general case, we need merely to look if there are other processes in need of the envelope. A special case, however, is when the process holding the envelope, finds that its present status is not the one it required. For example, a sender who has just received a filled envelope will need to forward it, but at the same time must ensure that the envelope will eventually come back. These considerations have led us to the following implementation:

```
typedef enum { GENERAL, SPECIAL } FORWARDING;

void forwardEnvelope( unsigned caller, FORWARDING forwarding )
{
  STATUS requiredStatus, currentStatus;

  currentStatus  = node[caller].envelope.status;
  requiredStatus = ( currentStatus == FULL ? EMPTY : FULL );

  /* Check the special situation: the process forwarding the envelope does
     require it, but with a different status than the present one. It must
     therefore ensure that the envelope eventually returns.
  */
  if( forwarding == SPECIAL && !node[caller].requested[requiredStatus] ){
    node[caller].requested[requiredStatus] = TRUE;
    node[caller].envelope.numOfWait[requiredStatus]++;
  }

  if( node[caller].askedRight[FULL]  )
    node[caller].envelope.request[FULL]  = TRUE;
  if( node[caller].askedRight[EMPTY] )
    node[caller].envelope.request[EMPTY] = TRUE;

  /* Now merely check if all criteria are met. If the envelope can be
     forwarded, the local administration must be cleared entirely.
  */
  if( node[caller].envelope.request[currentStatus] ||
      node[caller].envelope.numOfWait[currentStatus] > 0 ){
    node[caller].askedRight[FULL]  = FALSE;
    node[caller].askedRight[EMPTY] = FALSE;
    node[caller].askedSelf[FULL]   = FALSE;
    node[caller].askedSelf[EMPTY]  = FALSE;

    node[caller].have = FALSE;

    network[right(caller)].ringEnv ! node[caller].envelope;
  }
}
```

The main loop of the put thread

We have now come to the main loop of the put thread. In effect, it is actually built as the major part of the above mentioned state-transition machine. This means that all global states are inspected, making a further distinction between whether or not the envelope is presently in the process' possession. Again, the code has been commented in order to explain further details.

```
process putThread[ NPROC ](
   network[self].local, network[self].ringReq, network[self].ringEnv )
{
  /* Permanently access your own local monitor. Access to shared data is
     withdrawn each time the process is waiting on a condition variable.
  */
  enter( node[self].data );
  while( TRUE ){

    /* Test if there were any unprocessed, incoming messages. If not, wait
       until the next message arrives (to be signaled by the get thread).
    */
    if( node[self].numOfMessages ≤ 0 ) wait( node[self].newMessage );

    switch( node[self].role ){
    case NULLNODE :
      /* If the envelope is currently in possession of the process, forward
         it.  Otherwise, forward any outstanding requests.
      */
      if( node[self].have ) forwardEnvelope( GENERAL );
      else                  forwardRequests( self );
      break;

    case SENDER_1 :
      /* The process is waiting for the arrival of an empty envelope. If the
         envelope is in its possession, check if it can be filled (only if
         it is empty), or if it should be passed on (when it is full). If
         the envelope is not at the process, simply issue a request for it,
         and forward any outstanding requests as well.
      */
      if( node[self].have ){
        if( node[self].envelope.status == EMPTY ){
          node[self].busy = TRUE;
          node[self].role = SENDER_2;
          signal( node[self].haveEnvelope );
        }
        else forwardEnvelope( SPECIAL );
      }
      else{
        issueRequest( EMPTY );
        forwardRequests( self );
      }
      break;

    case SENDER_2 :
      /* The envelope is in possession of the process. As soon as the main
         thread has finished filling it, the envelope should be sent away as
         soon as possible.  The process remains in this state until the
         envelope could be passed on.
      */
      if( !node[self].busy ){
        clearEnvelope( EMPTY );
        forwardEnvelope( GENERAL );
        if( !node[self].have ){
          node[self].role = NULLNODE;
          signal( node[self].envelopeSent );
        }
      }
      break;
```

```
    case RECEIVER_1 :
      /* The process is waiting for a filled envelope. As soon as the
         envelope has arrived, it can be used whenever it is
         filled. Otherwise, it should be passed on as soon as there's a
         sender on the ring. If the envelope is not in the process'
         possession, it should issue a request, and forward any outstanding
         ones.
      */
      if( node[self].have ){
        if( node[self].envelope.status == FULL ){
          node[self].busy = TRUE;
          node[self].role = RECEIVER_2;
          signal( node[self].haveEnvelope );
        }
        else forwardEnvelope( SPECIAL );
      }
      else{
        issueRequest( FULL );
        forwardRequests( self );
      }
      break;

    case RECEIVER_2 :
      /* The envelope is in possession of the process. As soon as the main
         thread is finished, the envelope can simply be forwarded (if
         possible), but in any case, the global state changes to that of a
         null node.
      */
      if( !node[self].busy ){
        clearEnvelope( FULL );
        forwardEnvelope( GENERAL );
        node[self].role = NULLNODE;
      }
      break;
    }
    /* Record that you have processed an incoming message. */
    node[self].numOfMessages--;
  }
  leave( node[self].data );
}
```

3.3 Analysis

In this section we come to an informal and experimental complexity analysis of the algorithm. To that aim, we make a distinction with respect to the number of processes that are willing to communicate at a certain time. As we have explained above, we assume that each process generally resides in either a state in which it has no need to send or receive a message, or in a state in which it requires to communicate. When most processes are not willing to communicate, there will hardly be any network traffic. On the other hand, when the behavior of processes is predominated by the fact that they want to communicate, network traffic will be considerable but also rather unpredictable. For example, when there are many senders and receivers on the ring, it can be expected that the number of hops that a filled envelope has to make in order to deliver a message from a sender to a receiver is relatively low. Likewise, the number of hops an empty envelope has to make before it reaches a sender that can subsequently fill it, can also be expected to low.

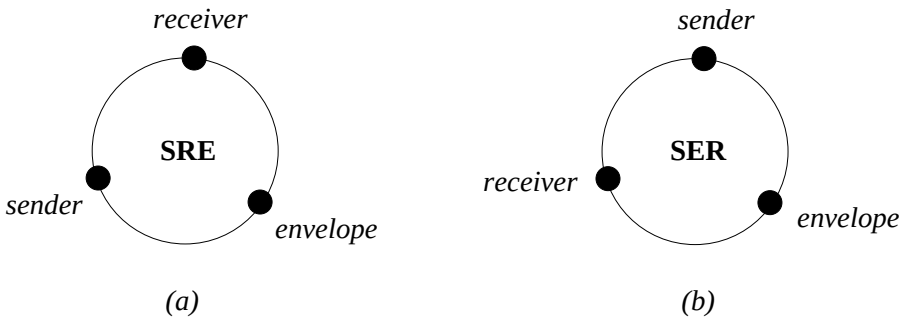


Figure 3.3: The two possible situations in a lightly loaded system: SRE (a) and SER (b).

3.3.1 Analysis of a lightly loaded system

In the case when processes are hardly ever willing to send or receive a message, the analysis of the complexity of the algorithm is rather straightforward. To that aim, denote by N_{proc} the total number of processes, which, of course, is also the length of the ring. Denote by $loc(E)$ the location of the envelope on the ring when there are initially no processes willing to communicate. Locations on the ring are clockwise numbered $0 \ldots N_{\text{proc}} - 1$. Similarly, we use the notations $loc(S)$ and $loc(R)$ to denote the locations of a sender and a receiver, respectively. We make a further distinction between the following two situations:

SRE: In this case, we assume that when traveling the ring clockwise starting at the sender, the receiver is located between the sender and the envelope, as shown in Figure 3.3(a).

SER: In the case, we assume the envelope is located between the sender and the receiver, as shown in Figure 3.3(b).

Let $\delta(R \rightarrow E)$ denote the distance between the receiver and the envelope, expressed in the number of links that need to be crossed when traveling *clockwise* from the receiver to the envelope. Similarly, we use the notations $\delta(S \rightarrow R)$ and $\delta(S \rightarrow E)$ to denote the (clockwise measured) distance from the sender to the receiver, and from the sender to the envelope, respectively. Note, as a matter of fact that for any two positions P and Q, we have that:

$$\delta(P \rightarrow Q) \quad = \quad (loc(Q) - loc(P)) \bmod N_{\text{proc}}$$

Because we are assuming that there are initially no senders and receivers on the ring, the envelope at first instance will be empty.

Let us first consider situation SRE. SRE reflects that both the sender and the receiver are now on the ring and that the envelope was *originally*, i.e. when there were no communicating processes on the ring, located between the receiver and the sender. Because the locations of either sender, receiver, and envelope are arbitrary (provided the ordering dictated by SRE), we may assume that

$$\delta(R \rightarrow E) = \delta(E \rightarrow S) = \delta(S \rightarrow R) = \frac{1}{3}N_{\text{proc}}$$

Two cases need to be considered further:

- **SRE-a:** *The sender arrived before the receiver.* In this case, we may assume that the envelope reached the sender before the receiver entered the ring. This implies that the sender's request for the envelope needed to travel a distance of $\delta(S \rightarrow E) = \frac{2}{3}N_{\text{proc}}$, whereas the receiver's request for the envelope traveled a distance of $\delta(R \rightarrow S) = \frac{2}{3}N_{\text{proc}}$. Consequently, the two requests jointly traveled a total distance of $\frac{4}{3}N_{\text{proc}}$ links. The envelope, on the other hand, needed to travel a total distance of $\delta(E \rightarrow S) = \frac{2}{3}N_{\text{proc}}$ from its original location to the sender, and, after the receiver entered the ring, another distance of $\delta(S \rightarrow R) = \frac{2}{3}N_{\text{proc}}$ links from the sender to the receiver.
- **SRE-b:** *The receiver arrived before the sender.* In this case, the receiver's request (for a *full* envelope) will first have to travel a distance of $\delta(R \rightarrow E) = \frac{1}{3}N_{\text{proc}}$ links where it arrives at the location of the envelope. At that point, nothing further happens due to the fact that the envelope is still empty. As soon as the sender enters the ring, its request for the empty envelope will have to travel a total distance of $\delta(S \rightarrow E) = \frac{2}{3}N_{\text{proc}}$. As soon as the request arrives, the envelope will be transferred over a total distance of $\delta(E \rightarrow S) = \frac{2}{3}N_{\text{proc}}$, marked with a request to forward it to the receiver as soon as it has been filled. After this has been done, it travels another $\delta(S \rightarrow R) = \frac{2}{3}N_{\text{proc}}$ links to the receiver adding up to a total distance of N_{proc}.

Summarizing, we have the following:

	total traveling distance	
	requests	*envelope*
SRE-a:	$\frac{4}{3}N_{\text{proc}}$	$\frac{4}{3}N_{\text{proc}}$
SRE-b:	N_{proc}	$\frac{2}{3}N_{\text{proc}}$

In a completely analogous way we can derive the complexity for the SER situation. Using the same distinction between cases SER-a (when the sender arrives before the receiver) and SER-b (when the receiver arrives before the sender), it can be readily verified that we have:

	total traveling distance	
	requests	*envelope*
SER-a:	$\frac{2}{3}N_{\text{proc}}$	$\frac{2}{3}N_{\text{proc}}$
SER-b:	N_{proc}	$\frac{2}{3}N_{\text{proc}}$

Either of the four cases (SRE-a, SRE-b, SER-a, and SER-b) can occur with equal probability. We can then draw the following conclusion:

> Each message exchange between a sender and a receiver in a lightly loaded system, requires on average a total of N_{proc} request transfers, and N_{proc} envelope transfers.

3.3.2 Analysis of heavy loaded systems

In the case of heavy loaded systems, we come to a completely different situation. On average, we may expect that the number of request and envelope transfers will decrease as more senders and receivers enter the ring. The reason is quite simple. In the first place, a request for an envelope need not always be forwarded to its initial destination. Instead, as soon as it reaches a process that had issued a similar request, its transfer halts. Likewise, the envelope may be successfully intercepted by a process that had entered the ring after the envelope had started to travel towards its initial destination.

Rather than providing a mathematical analysis, we have run a number of simulations in order to get an impression of the behavior of the algorithm. The results of these simulations are discussed in this section. In all cases we have measured the number of request transfers ($N_{\text{req}}^{\text{links}}$) envelope transfers ($N_{\text{env}}^{\text{links}}$) as a function of the total number of processes N_{proc}. In addition, we have varied the network traffic, but also varied the ratio between the number of senders and the number of receivers.

The network traffic was varied by letting processes reside in either a so-called null state σ_{null} in which no communication was required, or a communicating state σ_{comm}. While residing in state σ_{comm}, a process behaved either as a sender or a receiver. If the time to process the envelope, i.e., either filling it with data, or extracting its contents, is denoted by T_{env}, then the sojourn time T_{null} in state σ_{comm} was calculated as:

$$T_{\text{null}} = \mathit{delay} \cdot T_{\text{env}}$$

where *delay* was a simulation parameter varying between 1 and 50,000. In addition, the transition from state σ_{null} to σ_{comm} was probabilistic, adjusted by a simulation parameter *commprob*:

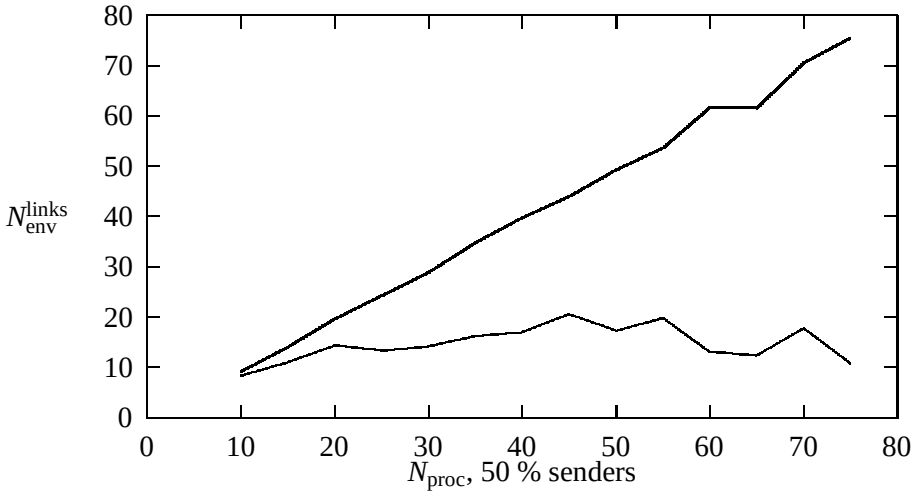
$$\mathit{commprob} = \mathcal{P}[\sigma_{\text{null}} \rightarrow \sigma_{\text{comm}}].$$

So, for example, a lightly loaded system was simulated by setting $\mathit{delay} = 50,000$ and $\mathit{commprob} = 0.50$. A very heavy loaded system was simulated by setting both these parameters to 1.

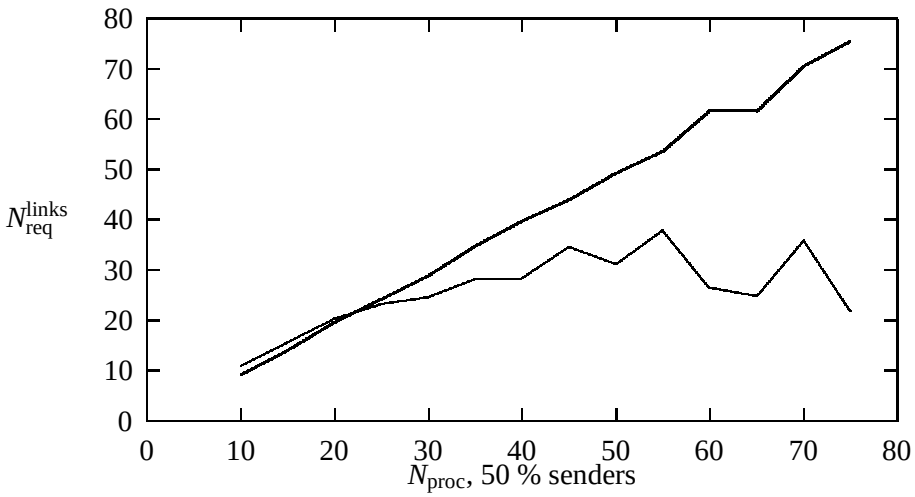
We also experimented with the ratio between senders and receivers as follows. If a process entered the ring to communicate while a total of N_{cproc} processes were already communicating, the newcoming process would become a sender with probability *sendprob* which could also be adjusted as simulation parameter. In effect, this meant that in a heavy loaded system, a fraction of approximately *sendprob* communicating processes would act as senders. This parameter did not have any affect in a lightly loaded system.

Figure 3.4 shows the result of our simulations when the number of senders and receivers was equally balanced. Two results are shown: in a lightly loaded system it is seen that the number of transfers is as expected in the sense that it linearly increases with the total number of processes. But as soon as the load increases, the number of transfers drops to an almost constant value in extremely heavy loaded networks.

The effect of systematically having less senders than receivers is shown in Figure 3.5. As was to be expected, the number of transfers increases less than in the case of a lightly loaded system. Perhaps surprisingly, this only holds for the number of envelope transfers: the number of request transfers is comparable to that in a lightly



(a)



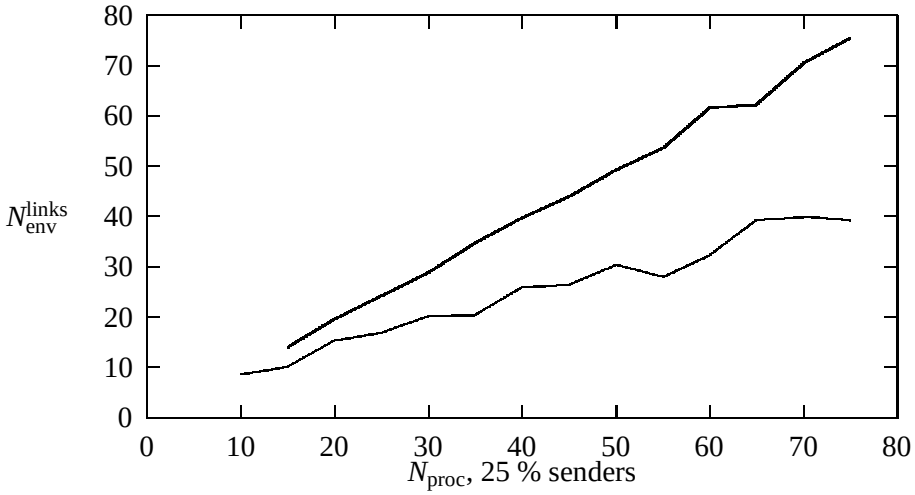
(b)

Figure 3.4: Simulation results for various loaded systems with *sendprob* = 0.50.

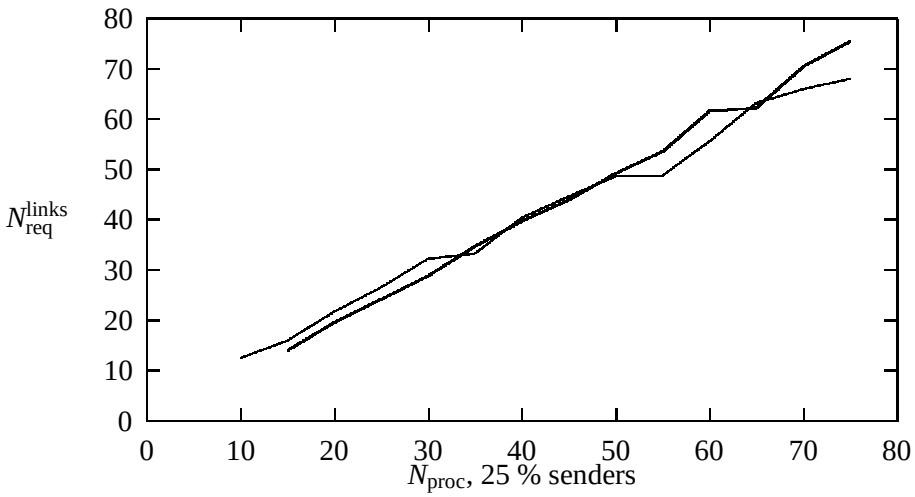
loaded system. This result is due to the fact that each time the envelope passes a process, any outstanding requests are cleared and administrated on the envelope. In effect, it can be shown that, on average, in a heavy loaded system with either very few senders, or very few receivers we have that:

- the number of request transfers is equal to N_{proc}, and
- the number of envelope transfers is equal to $\frac{1}{2}N_{\text{proc}}$.

We omit any further details concerning the actually analysis which is very like our analysis of lightly loaded systems.



(a)



(b)

Figure 3.5: Simulation results for various loaded systems with *sendprob* = 0.25.