A Distributed Implementation of Many-to-Many Synchronous Channels

Maarten VAN STEEN and Mark POLMAN
Erasmus University, Department of Computer Science
P.O. Box 1738, 3000 DR, Rotterdam, {steen,polman}@cs.few.eur.nl

Abstract. Within the ESPRIT project Hamlet, we have developed a graphical-based *Application Design Language* (ADL). This language allows a developer to primarily focus on the high-level *design* of parallel applications in terms of processes communicating by means of message-passing. ADL has been tailored to allow for automated generation of efficient parallel target code. In most cases, this goal can be relatively easily met. However, ADL also supports some high-level communication constructs which may be quite difficult to implement in the general case. In this paper, we discuss one particular implementation aspect, namely that of synchronous channels that allow communication between multiple senders and multiple receivers. Our attention focuses on a distributed, scalable solution for transputer-based systems. This solution is, in fact, also applicable to occam-like languages that permit guarded output statements.

1. Introduction

1.1. Background

The graphical Hamlet Application Design Language (ADL) has been developed to support the construction of parallel applications [4, 5, 7]. The language is based on a notion of processes communicating by means of message-passing. Its primary goal is to support the logical and physical *design* of applications that are to be executed on transputer-based systems. In particular, advanced constructs are provided by which a developer can easily devise the intricate communication structures that are inherent to parallel applications, without being limited to specific support for expressing communication by the actual target language. Among these constructs are so-called **replicators** for specifying geometrically structured collections of communicating processes and various **communication media** with rich semantics for expressing the exchange of messages between processes.

Although focusing on the design rather than the implementation of a parallel application may be important from a software engineering point of view, it does not alleviate development problems if there is hardly or no support for deriving efficient implementations. To that aim, ADL has been carefully tailored to allow efficiently executable code to be derived *automatically* from a design. This is particularly easily established in those cases when communication constructs have an evident counterpart in target languages. Automated generation of efficient code may become a problem when the relation between an ADL construct and the target language is less obvious. In the case of ADL, this problem arises with the general implementation of so-called **synchrounous channels** for transputer-based systems.

Synchronous channels in ADL are very similar to occam channels [2], with the exception that in ADL a channel can be shared between multiple senders and multiple receivers. As we shall explain, this in fact is equivalent to allowing alternative selection of output channels, which is not permitted in occam. In this paper, we describe an efficient distributed solution of these synchronous channels for transputer-based systems. In the remainder of this section we shall first specify the semantics of ADL synchronous channels, and show that most forms can indeed be implemented efficiently. The core of the paper is presented in Section 2 where we outline a distributed implementation as a general solution. The computational complexity of our solution is discussed in an informal manner in Section 3. We conclude by putting our present research into context in Section 4.

1.2. Problem description

An ADL synchronous channel connects 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, $\min\{M, N\}$ (sender,receiver)-pairs are selected nondeterministically and a message is transferred from sender to receiver. If a process cannot immediately communicate, it will block until communication is possible (assuming that the process is willing to communicate *only* by means of a single synchronous channel). Consequently, if $M > N$ a total of $M - N$ nondeterministically selected senders will remain blocked, and likewise, if $M < N$ a total of $N - M$ nondeterministically selected receivers will remain blocked.

The relationship between these semantics and those of occam channels is more obvious than one might intitially suspect. For example, when $M = N = 1$, there is no difference between the two languages. This also means that we can efficiently implement ADL channels directly by means of occam channels. When $M > 1$ and $N = 1$, our semantics are the same as that of an occam program in which a single receiver can alternatively select amongst M input channels, one for each sender. The opposite situation occurs when $M = 1$ and $N > 1$: in that case, a single sender should select betweeen N *output* channels, one for each receiver. An implementation in this case requires only two channels per receiver, adding up to a total of $2N$ occam channels. Per receiver, there is one channel from the sender to the receiver for passing the actual message, and one channel from the receiver to the sender by which the receiver can *announce* its willingness to communicate. The sender then first selects the communicating receiver by means of an alt-statement on the channels for announcements, and then proceeds by sending its message through the regular channel which connects it to the selected receiver.

But when $M > 1$ and $N > 1$, we may find ourselves in a difficult spot. When M and N are not too large, a centralized solution by which an additional process is responsible for (1) selecting a (sender,receiver)-pair, and (2) subsequently forwarding the message, may be acceptable (see [1] for further details). But as soon as the number of senders and receivers increase, the central process may turn out to be a bottleneck. Our goal at this point, is therefore to present a completely distributed and scalable solution that is suited for transputer-based systems.

2. A distributed solution

In order to come to an efficient distributed 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 1.

Each process essentially consists of two components. The subprocess (called the **main**

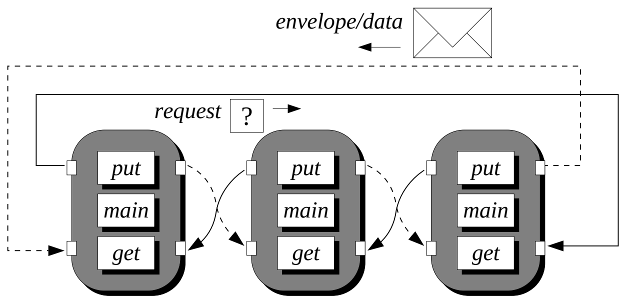

Figure 1: The global architecture of a distributed implementation for ADL synchronous channels.

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 are implemented), jointly comprising an actual process.

2.1. 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**.

2.1.1. A 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 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. As soon as it knows that there is someone waiting for an empty envelope, the envelope is forwarded to its right-hand neighbor.
2. The envelope is empty: in this case, the receiver is in possession of an envelope that needs yet to be filled by a sender. Hence, it will first need to wait for a request from its right-hand neighbor, and then subsequently forward the envelope. In order to ensure that the envelope will eventually return, the receiver *marks* it.

The mechanisms for forwarding an envelope will be described below. An important observation is that we never forward the envelope unless there is a good reason to do so. In particular, 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.

2.1.2. A sender

The sender's situation is almost symmetrical to that of 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*.

2.1.3. An inactive process

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 a sender on the ring, or when it is full forwarding only takes place when there is a receiver. Note, by the way, that an active process will always forward the envelope: the only reason it got to the process was because there was a request for it.

2.2. Forwarding the envelope

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.

2.2.1. 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.

2.2.2. 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. The following three situations need to be distinguished.

Case 1: The process itself is currently inactive. Assume the envelope is empty (the case of a full envelope is analogous), 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 when there is also an outstanding 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.

Case 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).

Case 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.

2.2.3. *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 if no request for a filled envelope had arrived from its righthand neighbor. 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 when there are no outstanding requests. Then, the envelope is filled and behavior proceeds as mentioned above.

2.3. *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 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.

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 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.

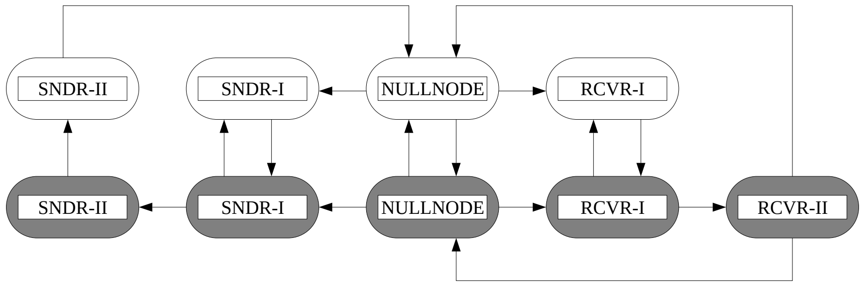

Figure 2: The state-transition diagram for a process.

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.
- **SNDR-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 it is suited to be filled by the main thread.
- **SNDR-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.
- **RCVR-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.
- **RCVR-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.

The actual implementation of the algorithm is now straightforward. The only aspect that needs some attention is how we can combine the communication between threads (which is through shared data), and the communication between processes (which is across links by means of message-passing). More specifically, we need to find a way by which a get thread in a particular process can synchronize by means of a single mechanism on (1) information from one of the other threads in that process, and (2) information from other processes. A solution is found by using a local channel between the main thread and the get thread. This channel is used to inform the get thread that the main thread wants to either communicate, or that it is finished with communication. Using an alt-statement, it is then possible for the get thread to selectively wait on any incoming information.

We shall not further discuss implementation details, as these are now straightforward. Instead, the interested reader is referred to [6] where detailed skeleton code is presented[1].

3. Complexity 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 be low.

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:

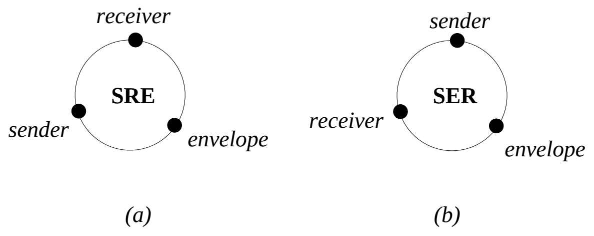

Figure 3: The two possible situations in a lightly loaded system: SRE (a) and SER (b).

SRE: In this case, we assume that, if we travel the ring clockwise starting at the sender, we encounter the receiver before the envelope, as shown in Figure 3(a).

[1] The report is available on ftp-site ftp.cs.few.eur.nl.

SER: In this case, we assume the envelope is located between the sender and the receiver, as shown in Figure 3(b).

Let $\delta^+(R \to E)$ ($\delta^-(R \to E)$) denote the distance between the receiver and the envelope, expressed in the number of links that need to be crossed when we travel (counter-)clockwise from the receiver to the envelope. Similarly, we use the notations $\delta^+(S \to R)$ ($\delta^-(S \to R)$) and $\delta^+(S \to E)$ ($\delta^-(S \to E)$) to denote the (counter-)clockwise measured distance from the sender to the receiver, and from the sender to the envelope, respectively. 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

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

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 $\delta^+(S \to E) = \frac{2}{3}N_{\text{proc}}$, whereas the receiver's request for the envelope traveled a distance of $\delta^+(R \to S) = \frac{2}{3}N_{\text{proc}}$. 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 $\delta^-(E \to S) = \frac{2}{3}N_{\text{proc}}$ from its original location to the sender, and, after the receiver entered the ring, another distance of $\delta^-(S \to R) = \frac{2}{3}N_{\text{proc}}$ 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 $\delta^+(R \to E) = \frac{1}{3}N_{\text{proc}}$ 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 $\delta^+(S \to E) = \frac{2}{3}N_{\text{proc}}$. As soon as the request arrives, the envelope will be transferred over a total distance of $\delta^-(E \to S) = \frac{2}{3}N_{\text{proc}}$, 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 $\delta^-(S \to R) = \frac{2}{3}N_{\text{proc}}$ links to the receiver adding up to a total distance of $\frac{4}{3}N_{\text{proc}}$.

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), we can summarize our analyses as shown in Table 1.

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.

Table 1: Average number of hops for requests and the envelope per message exchange.

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

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

3.2. Analysis of heavily loaded systems

In the case of heavily 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. How these simulations have been conducted is discussed in [6]. Here, we shall only briefly discuss their results.

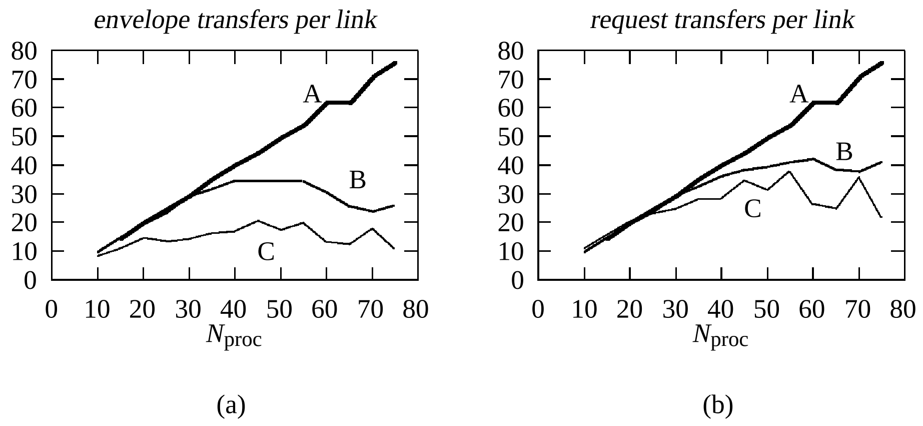

Figure 4: Simulation results for various loaded systems with *sendprob* = 0.50.

Figure 4 shows the result of our simulations when the number of senders and receivers was equally balanced. In Figure 4(a) we have visualized how many links the envelope will, on average, travel per message exchange between a sender and a receiver. The curve marked “A” shows the situation in a lightly loaded system. As can be seen, this result corresponds with our informal analysis given above. Curves “B” and “C” reflect the situation when network traffic increases. Note that in extremely heavily loaded systems, the number of envelope transfers per link tends to be almost constant. Figure 4(b) shows a similar case for the number of request transfers per link, for each message exchange.

4. Discussion

In this paper we have concentrated on the implementation of synchronous channels that are shared between multiple senders and receivers. In particular, attention has focused on a distributed, scalable solution for transputer-based systems. This solution is, in fact, also a solution for an implementation of occam-like languages that support guarded output statements. In particular, it is not difficult to see that if our target lanaguage supported the alternative selection of output statements, we could have easily derived an implementation in the form of the following skeleton code (we adopt an occam-like notation; the pseudo-variable self refers to the identity of the process in which it is used):

```
process receiver ( [M][N] chan of message channel )
  alt i = 0 for M
     channel[i][self] ? data

process sender ( [M][N] chan of message channel )
  alt j = 0 for M
     channel[self][j] ! data
```

The question whether or not guarded output statements are to be provided by a language has generally been a difficult one to answer. Due to the fact that a general *efficient* implementation is hard to derive, most language designers have omitted them. To date, only a few languages support guarded output statements (e.g. Joyce [3]). The reason for including a similar concept in ADL is motiviated by the fact that from the perspective of application *design*, the availability of high-level communication constructs is a desirable feature. Later, when a developer is putting more effort into the derivation of an efficient implementation, he or she might choose to alter a design in such a way that communication constructs that are difficult to implement are avoided all together. This is the right thing to do during the implementation; it is not something that should bother a developer during the design phase.

A first version of ADL has been implemented as part of the Hamlet Design Entry System. This version allows for full generation of *simulation code*, by which the overall behavior of an application can be simulated for various hardware configurations. At present, our attention is directed towards the generation of actual parallel target code. Because many implementation decisions are application-dependent, code generation is supported in such a way that a developer can highly influence the generation process. The solution presented in this paper for the implementation of synchronous channels, will thus only be one out of several that can be selected when generating code.

References

[1] G.R. Andrews. *Concurrent Programming: Principles and Practice.* Benjamin/Cummings, 1991.

[2] G. Jones and M. Goldsmith. *Programming in Occam*, Prentice-Hall. 1988.

[3] Brinch Hansen P. "Joyce – A Programming Language for Distributed Systems." *Software – Practice and Experience*, 17(1):29–50, January 1987.

[4] M.R. van Steen, T. Vogel, and A. ten Dam. "ADL: A Graphical Design Language for Parallel Real-Time Applications." In R. Grebe et al., (eds.), *Transputer Applications and Systems*, IOS Press, Amsterdam, 1993.

[5] M.R. van Steen. "The Hamlet Application Design Language, Introductory Definition Report." Techn. Rep. EUR-CS-93-16, Erasmus University Rotterdam, December 1993.

[6] M.R. van Steen. "The Hamlet Application Design Language: On the Implementation of Synchronous Channels." Techn. Rep. EUR-CS-94-03, Erasmus University Rotterdam, May 1994.

[7] M.R. van Steen, A. ten Dam, and T. Vogel. "Computer-Aided Support for Designing Parallel Real-time Solutions on Transputer-based Systems." *Proceedings IEEE Conference on Control Applications*, Glasgow, August 1994, .