OSI and other Layered Architectures: Principles and Implementation

Robin Sharp
Informatics and Mathematical Modelling, DTU.

February 2008

1 Layered Architectures

The idea of using layered architectures is a common one in communication systems, operating systems and other large system programs. The general principle is that each layer offers services for use by active objects in the layer above. The layered architecture described in the OSI Basic Reference Model [8] is a particular example of this, specifying which layers are conceptually to be found in a standard communication system, which services they conceptually offer, and which functions they are expected to be able to perform in order to offer these services.

The acronym OSI stands for *Open Systems Interconnection*, and is a general term covering everything which has to be considered when systems (especially computer systems) have to cooperate with one another, or – to use a more technical term – to *interwork* across a communication network in a manner which is independent of manufacturers' system-specific methods. The OSI Reference Model defines the most basic principles for this purpose.

In a layered architecture, each layer – by building on the facilities offered by the layer beneath – becomes able to offer the layer above a *service* which is different from the service which it itself is offered. As a rule it will be better (more reliable, more free of errors, better protected against "intruders", ...) or will offer more possibilities than the layer beneath. This is achieved by the active components in the N'th layer – in OSI parlance the *(N)-entities* – exchanging information with one another according to a set of rules which are characteristic for the layer concerned – the *(N)-protocol*. A simple example could be that they exchange information with a certain degree of redundancy, so that any errors which are introduced by the layer beneath – the *(N-1)-layer* – can be detected and

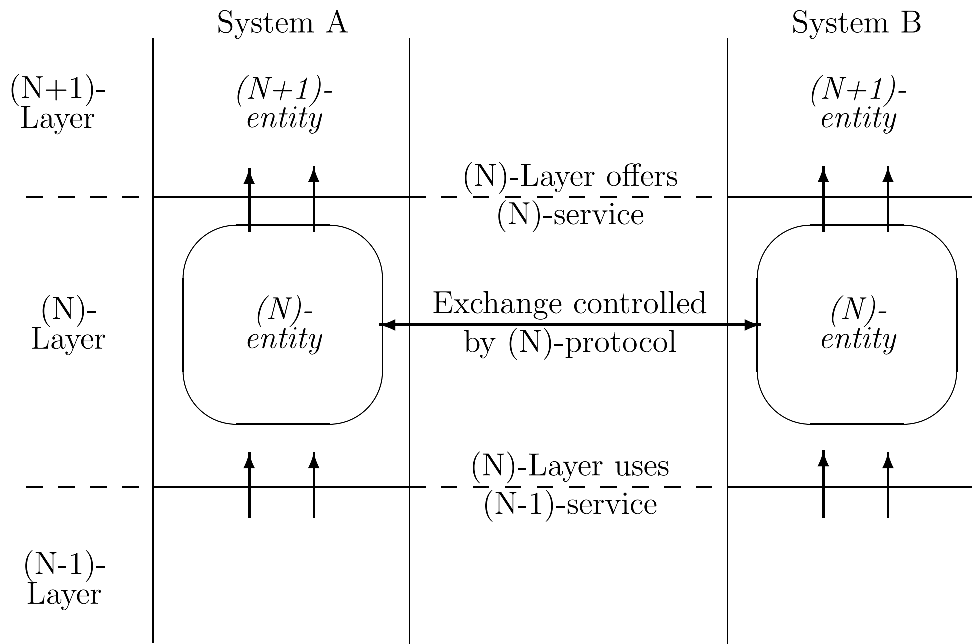

Figure 1: Principles in a layered architecture

corrected. The *(N)-service* which they offer to the active components in the *(N+1)-layer* – the *(N+1)-entities* – can thus be less disturbed by errors than the *(N-1)-service* which they are offered by the *(N-1)-layer*. This is illustrated in Figure 1.

The OSI Reference Model not only prescribes a layered architecture, but also defines which layers are conceptually to be found in a standard communication system, which services these layers offer and which functions the individual layers are expected to be able to perform in order to offer these services. As almost everybody knows, the OSI Reference Model specifically describes 7 layers, whose main functions are shown in Figure 2.

However, the important feature of the model is really that it introduced a *standard architecture* and a *standard notation* for many concepts related to data communication. The terms given in italics above are examples of terms introduced in the model. That, for example, there are seven layers is relatively unimportant, and the explanations of why there should be exactly seven are mostly entertaining rather than strictly technical. In practice, for descriptive purposes some of the layers (particularly the Data Link, Network and Application layers) are often divided into *sub-layers*, while implementations, on the other hand, often implement several layers as a single unit.

The OSI Reference Model architecture is not the only layered architecture which you may meet in communication systems. Several commercial manufacturers have developed products which are structured in a similar way. Well-known examples are IBM's SNA architecture and Digital's DECNET. Naturally, the protocols used are not in general the

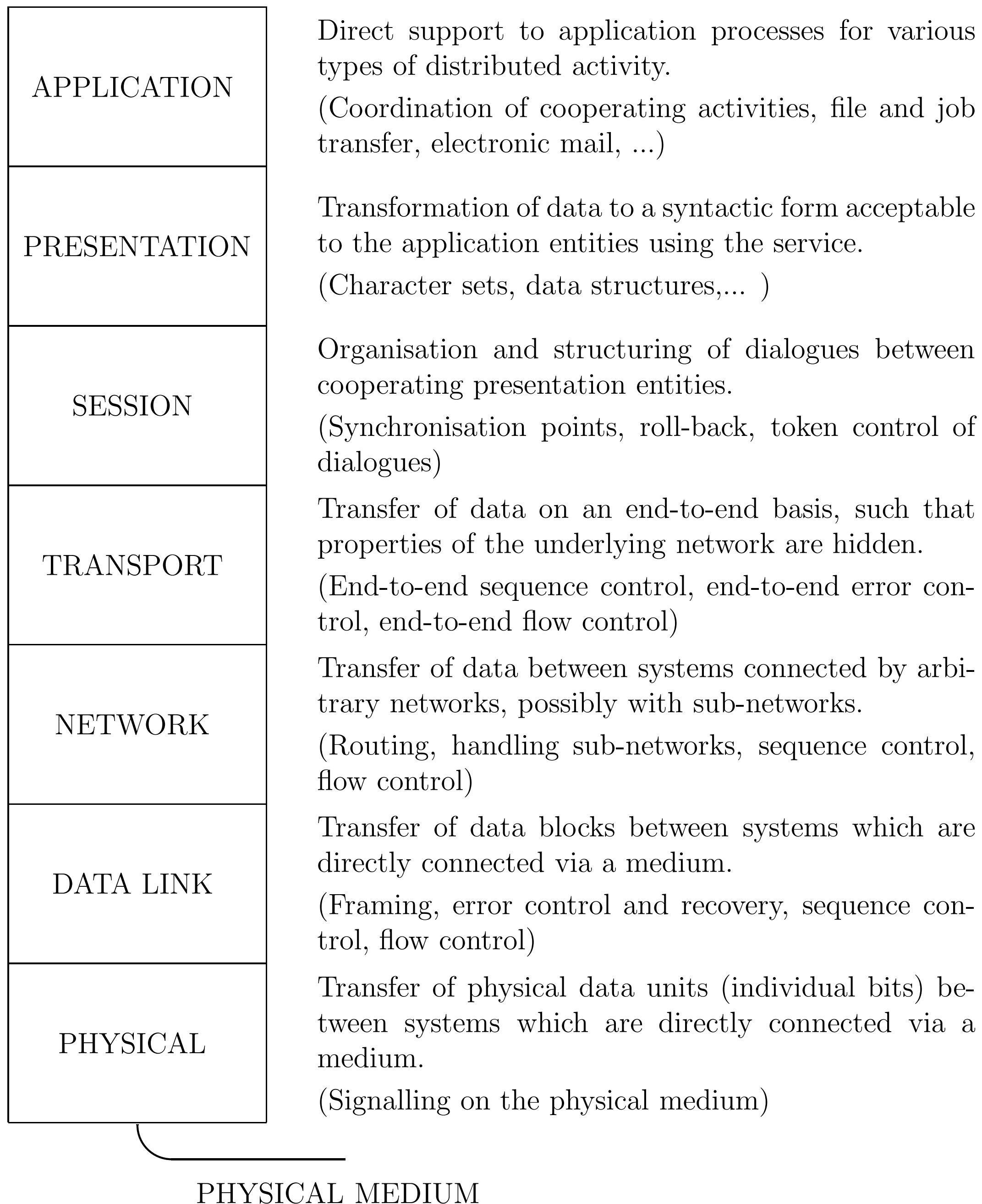

Figure 2: The OSI Reference Model: The 7 layers

same as OSI protocols, and the layers do not always correspond exactly to the OSI ones, especially in the so-called Upper Layers: the OSI Session, Presentation and Application layers.

A particularly common alternative arrangement is to consider the three upper layers as one unit, an 'Application-oriented layer' which depends directly on the Transport layer. A well-known example of this approach is found in the so-called *Internet protocols*, commonly used in Unix-based systems. Here, a whole series of application protocols – for example, for file transfer (FTP), electronic mail (SMTP), and handling virtual terminals (TELNET) – run directly over the Transport layer, which in such systems is based on the TCP protocol, while the standard OSI layer structure is used for the Network layer and below. Similar arrangements are often found in local area networks, where OSI protocols are used up to the Transport layer, while the architecture and choice of protocols in the Upper Layers deviates from the OSI standards.

Finally, in modern telecommunication systems, a somewhat different layered architecture can be found in systems based on ATM (Asynchronous Transfer Mode), a technology for supporting high-speed transfer of data over a local area or wide area network. This architecture is described by the *Broadband ISDN Protocol Reference Model (B-ISDN PRM)* [11]. In this model, although the layers roughly correspond to the OSI RM, there are several important technical differences, especially with respect to the way in which control and signalling information is transferred: In OSI, it forms part of the ordinary data flow; in B-ISDN, it is transferred over a separate connection.

2 Implementations

As we have seen, the OSI Reference Model takes a rather abstract view of communication systems, and the activities of the *(N)-layer* are described in terms of quantities such as the *(N)-entity*, which is an abstraction of the active element within the layer, the *(N)-SAP*s, which are abstractions of the mechanisms used to transfer data between the N'th and (N+1)'th layers, and *(N)-primitives*, which are abstractions of the individual interactions between these layers in connection with the use of the facilities available from the *(N)-service*. This abstract view of an OSI layer is illustrated in Figure 3.

Typical standards which describe the services and protocols in a particular layer continue to use these abstractions, with the obvious advantage that the service and protocol descriptions remain independent of possible ways of implementing them in practice. Thus the standards do not prescribe the answers to questions such as:

- Whether the *(N)-entity* is implemented in hardware or software – and, if it is software, whether an entity corresponds to a single process/task, or whether an entity

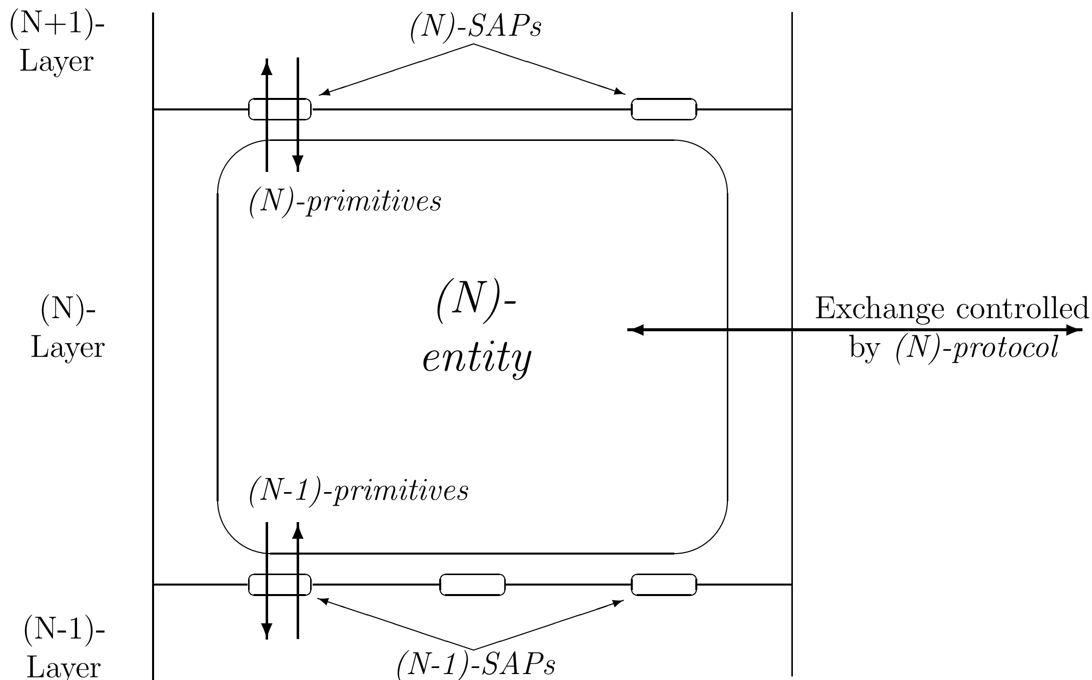

Figure 3: Abstract view of the N'th OSI layer in a single system

is composed of several processes or whether entities from several layers are collected together in a single process.

- Whether the *(N)-primitive*s are realised by means of signals, semaphores, message passing or subroutine calls, and how parameters for primitives are to be represented.
- Whether *SAP*s are to be implemented by means of control registers implemented in hardware, or by buffers or by queues.

Nor do standards concern themselves with matters such as where data are to be buffered, which size the buffers should have, how buffer management (allocation and deallocation) is to be organised, or what is to happen if no more buffers are available. Another open question is how many simultaneous *connnections* an implementation should be able to handle and what degree of *multiplexing* (if this is allowed in the layer in question) is to be permitted. The implementor has to find the answers to all these questions as part of the process of designing a particular implementation.

Naturally enough, this gives the implementor pretty free hands with respect to the implementation. In general, however, there is a tendency to implement the lower layers (the Physical layer up to and including the Network layer) in hardware, typically in the form of micro-programmed controllers or ASICs, in order to achieve maximum performance, while the upper layers (from the Transport layer and up) are typically implemented in software within the framework of a standard or specially designed operating system.

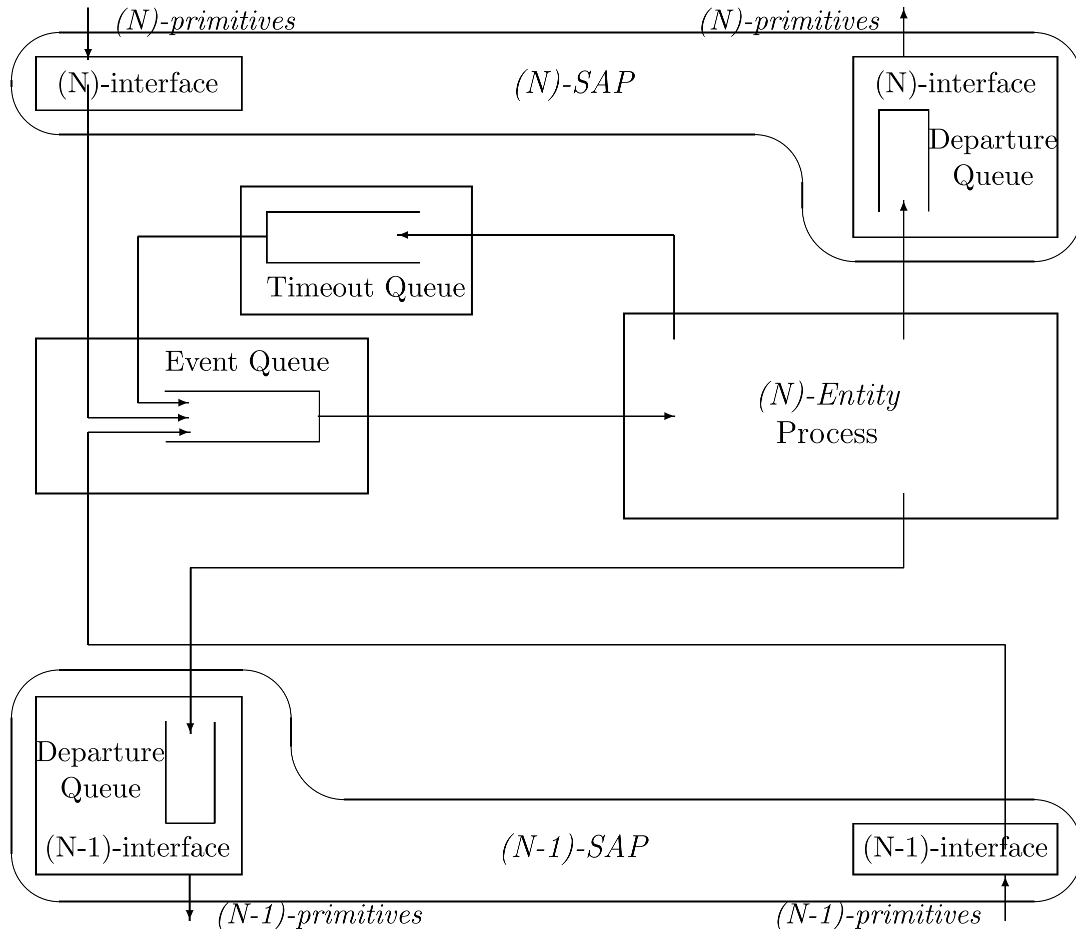

Figure 4: A possible implementation architecture for the N'th OSI layer

2.1 Implementation Architectures

Most implementations in fact follow the Reference Model's abstract architecture and have a modular structure, so that the individual layers are, to all intents and purposes, implemented separately. The places where you are most likely to see deviations from this modular structure are in the Physical layer, which in many hardware implementations is combined with the Data Link layer, and in the Presentation layer, which is often combined with the Application layer.

An example of a software architecture for use in implementing a single layer is shown in Figure 4. The central part of the implementation is the *(N)-Entity*, which is implemented as a state machine or sequential process which can react to incoming service primitives (from the $N-1$'th or $N+1$'th layer) and timeout events, and can generate outgoing service primitives in accordance with the rules of the protocol. Often this implementation is derived from a formal specification of the protocol in a process algebraic language (well-

known examples for this purpose are CSP [6], CCS [12] and LOTOS [9, 1]) or a language based on extended state machines (such as SDL [2] or ESTELLE [10]). We shall return in Section 3 to the question of how to transform such descriptions to code in a conventional sequential programming language.

Typically for a software implementation architecture, queues are used to buffer asynchronous events. Thus outgoing primitives are queued in the interface to the neighbouring layer, incoming events are queued in the Event Queue for sequential treatment by the *(N)-Entity*, and timer events are queued for delivery as timeout events to the Event Queue when the timer runs out.

If the neighbouring layers are also implemented in software, the interfaces between layers are trivial. If the neighbouring layers are implemented in hardware, the interfaces have the form of a hardware interface with associated driver. It should be obvious that a structure of this type can be stacked to produce an implementation of a group of contiguous layers.

2.2 Queues

An efficient implementation of the Event Queue has a considerable impact on the performance of the system. Particular attention needs to be paid to the mechanism used for getting the protocol entity to react to arrival of a new event. For example, the entity may poll the queue to see whether there are any new events, or may react to interrupts which are caused when an event is placed in the queue, or may wait at a semaphore which is signalled when an event arrives. The usual trade-offs apply here:

- Interrupts and semaphores generally give rise to context switching, which in many systems require a large number of instructions to be executed. On the other hand, these mechanisms enable a single CPU to be shared between several activities in an efficient manner.
- Polling the queue requires very few instructions, but has to be done continually in order to ensure a fast response time. This makes it difficult to use a single CPU for other activities at the same time.

As an example of the costs of interrupt handling, experiments with the Mach operating system on a DecStation 5000/200 show that handling an interrupt from an adaptor card takes $75\mu s$, a long time seen in relation to the $200\mu s$ needed to deal with all other functions involved in the receipt of an UDP/IP PDU [5]. An effective compromise is to cause an interrupt (or signal a semaphore) only if the queue is empty when an event has to be placed in it. The entity is then designed to take events out of the queue and deal with them until the queue becomes empty; after that it “sleeps” until woken by an interrupt (or signal).

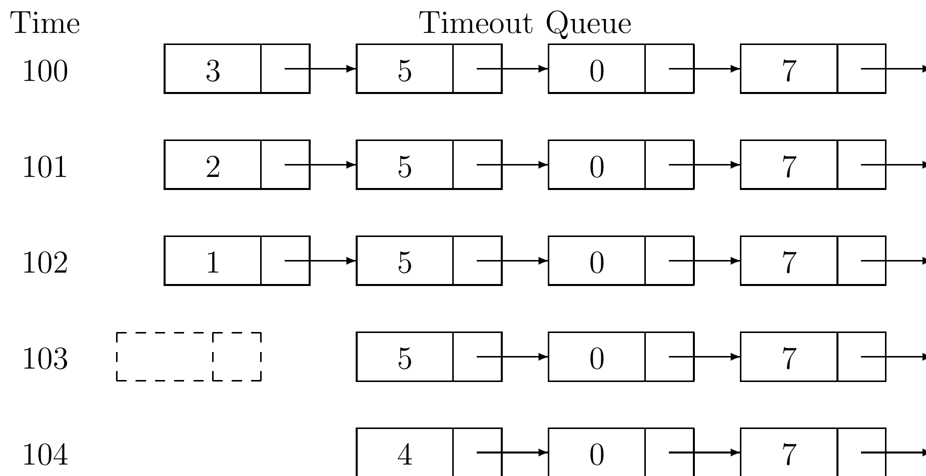

Figure 5: A Timeout Queue based on time differences

The primitives themselves are typically described by data structures with fields for the parameters of the primitive. Most service primitives only carry a few parameters. Nevertheless, the parameters themselves may take up a lot of room in memory – for example, they may be the data to be passed as an SDU, or long, multi-component addresses, or features such as Quality of Service, which have several components. Copying such parameters between layers is expensive, so transfer of pointers is the usual practice.

Many protocols require the use of several logical timers. A convenient way to organise the Timeout Queue is then as a queue of event descriptors, ordered according to the time at which the relevant logical timer will time out. Thus the first descriptor, say d_1, specifies how long there is to go before the first timeout event will take place, and so on. For efficiency, the following descriptors, $d_2, d_3, \ldots, d_n$, each specify the *difference* between the timeout instant for the corresponding logical timer and the previous logical timer. Although this makes *insertion* of a new logical timer in the queue comparatively expensive, when the physical timer indicates that a time unit has passed it only becomes necessary to count down the value of the element at the *head* of the queue. When this value reaches zero, the descriptor is removed from the Timeout Queue and a timeout event descriptor is placed in the main Event Queue. This is illustrated in Figure 5, which shows a Timeout Queue which at time 100 contains four logical timers, which will time out at time instants 103, 108, 108 and 115 respectively.

2.3 Multiplexing

Multiplexing in such an architecture is typically dealt with by setting up a data structure containing an entry for each opened connection. As in so many other situations, a statically

dimensioned data structure (an array with fixed dimensions) gives rapid insertion and lookup, but sets a natural limit for how many simultaneous connections can be set up; a dynamic data structure (such as a list) is only limited by the size of the memory, but generally gives a larger overhead for insertion and deletion of elements. Each entry in the data structure typically contains:

1. A *static* description of the connection, including such information as:
   - The *addresses* of the (N)-SAPs between which the connection is set up.
   - References (so-called *Connection end-point identifiers*) which uniquely identify the particular connection between these addresses, remembering that in a system with multiplexing there may be several such connections at any one time.
   - The *mapping* in the local system between the (N)-SAP which gives access to the layer above and the (N-1)-SAP which is used to give access to the layer below.
   - The required *Quality Of Service, QOS* for the connection.
   - The values to be used for any *timers* associated with the protocol.
2. A description of the current *dynamic state* of the connection, including such information as:
   - The current *state* of the protocol regarded as a sequential process or state machine.
   - The current positions of the *send and receive windows* in the sequence number space.
   - Descriptions of the *buffers* currently in use for sending and receiving PDUs.
   - Descriptions of the *timers* currently active on the connection.

For connectionless-mode protocols, it is obviously not necessary to preserve this type of information.

2.4 Buffers

In a layered architecture, an *(N)-SDU* passed from the (N+1)-layer as part of an *(N)-primitive* will be transmitted to the peer (N)-entity as one or more *(N)-PDU*s, where each (N)-PDU includes a fraction of the (N)-SDU as data, together with a header containing protocol control information for the (N)-protocol, *(N)-PCI*. This is illustrated in an abstract manner in Figure 6. The figure shows the unsegmented case where the entire SDU is sent in a single PDU. Note that as an SDU from an upper layer proceeds down the stack of layers, PCI has to be added and segmentation may have to take place in *each* layer.

Copying data between buffers is expensive in CPU bandwidth, and should preferably be completely avoided. Attention therefore needs to be paid to how to:

- Assemble the PCI and a segment of the SDU into a PDU in the sending system;

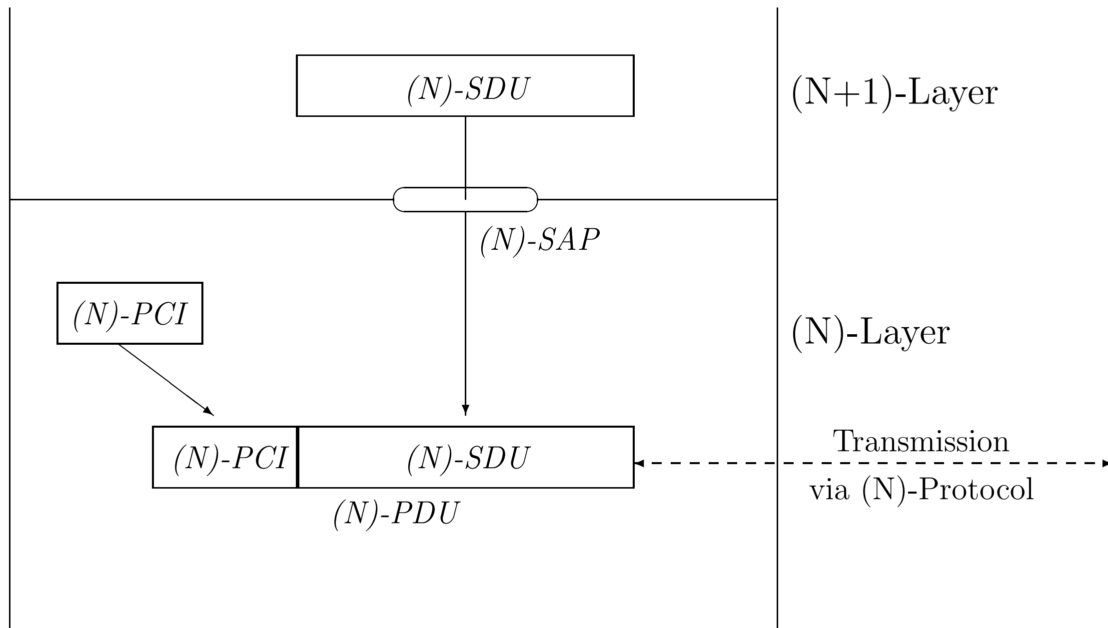

Figure 6: PCI and non-segmented data in a PDU

- Reassemble the parts of a segmented SDU into a complete SDU in the receiving system.

Whether this can be done efficiently or not depends on the capabilities of the operating system and its buffer allocation system.

As an example, consider a 16kbyte message buffer allocated contiguously in the store. Suppose this is to be transmitted as an SDU via a protocol which permits a maximal PDU size of 4kbyte, and suppose this is also the size of the pages used by the operating system in its memory management system. The segments of the SDU can only be allowed to fill rather less than 4kbyte, in order to leave room for PCI in each PDU, and most of them will not be page aligned (see Figure 7). The PCI for each PDU will in general come from a separate page. Thus the transfer of each PDU will in general require sending portions of three physical pages in the store; to transfer the entire SDU requires sending portions of 14 physical pages. A similar problem occurs on the receive side, where ideally the PCI has to be directed to a buffer of its own, while each segment of the SDU is placed after the previous segment in a contiguously allocated area of storage.

Modern operating systems typically support this style of operation by allowing the user to join one or more physical buffers logically together into an *aggregate*, to split an aggregate into its component buffers or to remove portions from the beginning or end of an aggregate. Concrete examples of this concept are the *mbufs* found in BSD Unix and the *messages* used in the *x*-kernel [7]. Modern physical interfaces sometimes support this by allowing the

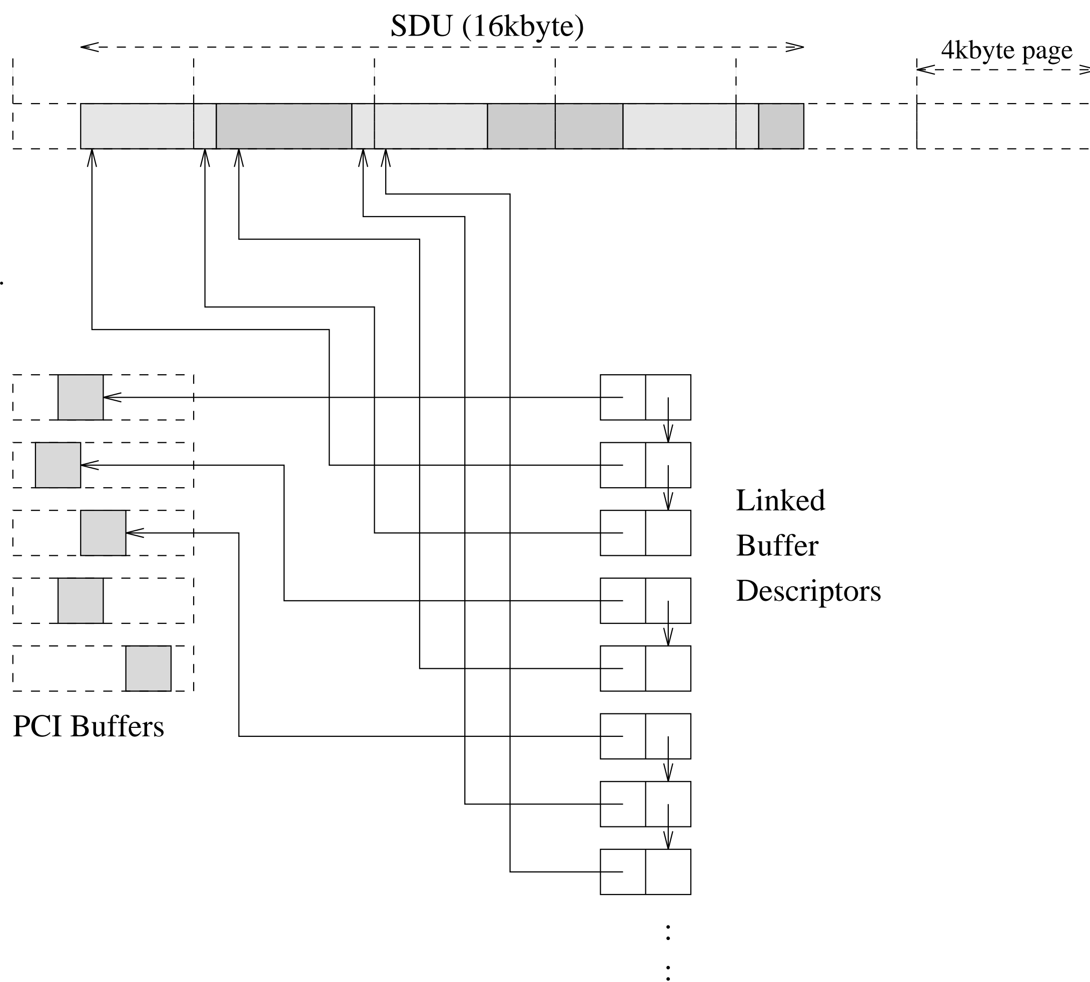

Figure 7: Linked descriptors for transmitting a segmented SDU

operating system to initiate DMA transfer of a sequence of physical buffers described by a linked structure of descriptors. This is illustrated in Figure 7. Or the system may support virtual address DMA via a hardware virtual-to-physical translation buffer (scatter/gather map); the software sets up the map to contain suitable mappings for all fragments of the buffer before starting the DMA transfer, which then takes place from a contiguous portion of *virtual* address space.

2.5 Protection domains

Modern operating systems generally use the concept of *protection domains*, and for protection reasons it is convenient if device drivers, protocol entities and application software all reside in different domains. Typically, the driver lies in the kernel domain, the protocol entity in the manager/server domain of the operating system, and the application software in the user domain. The problem is essentially how to transfer data between these domains without physically copying the buffers. Note that in communication systems, the data, once they have been placed in the buffers, are generally *immutable*, i.e. they do not subsequently need to be changed, but sometimes the sender needs to have further access to the data, for example for retransmission. Thus we distinguish between *move semantics*, where the buffer becomes inaccessible for the sending domain, and *copy semantics*, where the buffer becomes accessible for the receiving domain without becoming inaccessible for the sending domain.

Note that naive solutions such as statically sharing a pool of buffers between two or more domains are not acceptable, as they compromise protection and security by giving all domains read and write access to the buffers in the shared pool.

One possibility, used in the V-kernel [4] and DASH [14], is *page remapping*, where the same physical page in the storage is moved from one domain's *page map* to the other's, as illustrated in Figure 8. In other words, the map entry for the page is removed from the sending domain's map and placed in the receiving domain's map. This assumes, of course, that there are separate page maps for each protection domain, but this can reasonably be expected to be the case. Page remapping gives a *move semantics* for the operation, as the buffer becomes inaccessible for the sending domain.

Although data do not have to be copied between buffers, this technique introduces its own forms of overhead: In order to change the page map, it is necessary to change to supervisor mode, acquire necessary locks to virtual memory data structures (which nowadays typically consist not only of a hardware-implemented page map but also of higher-level data structures), change the mappings for each affected page, perform TLB/cache consistency operations and then return to user mode. Results for DASH running on a Sun 3/50 give an overhead of $208\mu s$/page, while on a DEC 5000/200 the overhead is about $22\mu s$/page. This value is typical for modern processors, where these operations are limited by memory

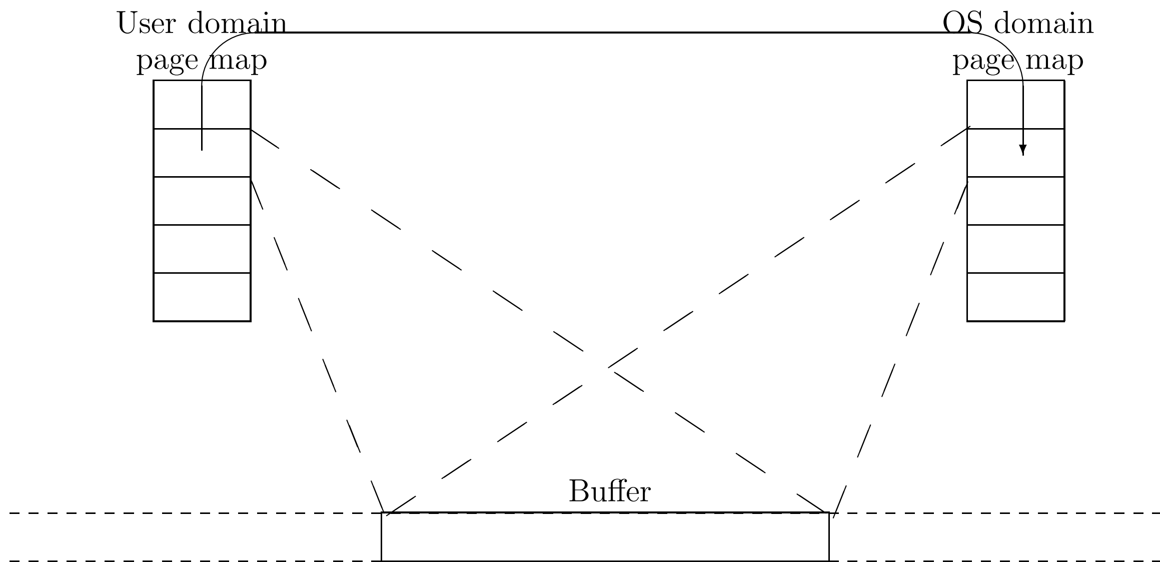

Figure 8: Page remapping between two domains

bandwidth rather than CPU speed.

An alternative technique, used in Accent [13] and Mach [3], is *copy-on-write (COW)*, which offers copy semantics. The technique is to allocate a new page frame, and to copy the data into this frame, only when *changes* are made to the data. As pointed out previously, protocol implementations are usually characterised by the data being immutable, so that once written they are not changed. Thus in practice only the originator of data writes into the buffer, while all other domains through which the buffer passes merely need read access. For this purpose, COW is an efficient technique.

3 Deriving Implementations from Specifications

To ensure that a protocol is correct in relation to some high-level requirements, it is a good idea to specify the protocol by using a formal specification language. This offers a number of well-known advantages, for example that such a formal specification is usually clear, unambiguous and concise, and that its properties can often be verified by some kind of formal proof technique. To produce a correct implementation it is then necessary to transform the formal specification in some systematic manner into a program in some (more or less) conventional programming language.

Most formal specification languages used in practice for protocol specification are based on the underlying idea that the system consists of a set of state machines (automata) which

interact by exchanging messages. At a given instant, a given automaton is in a particular state, and the occurrence of an event, such as a timeout or the receipt of a particular type of incoming message, causes it to change state and possibly to emit an outoing message. In some formal description languages, such as ESTELLE and SDL, this underlying model of the system is used directly: Each part of the system is specified as an extended finite state automaton, whose behaviour is described in terms of the actions to be performed when particular events occur at times when the system is in a particular state. For example, in ESTELLE, the specification is composed of a number of state transition descriptions, each of the general form:

```
from state
  to newstate
    when event
      provided condition
        begin Action end
```

where *state* gives the current state, *event* the event, *condition* a (possibly empty) Boolean condition for making the transition when this event occurs, *Action* the statements to be executed when the transition is made, and *newstate* the new state. Such formal descriptions can be transformed more or less directly into programs which simply react to incoming events in the manner described in the formal specification, following a skeleton in the style shown in Figure 9 (here given in a C-like syntax). It is here assumed that the event queue consists of a list of *descriptors* of events, and that the procedure `remove_from_event_queue` removes the descriptor at the head of the queue; if the queue is empty, execution is blocked until the queue is non-empty.

```
state      = S1;
terminated = FALSE;
initialise other state variables;
do
{ remove_from_event_queue( event );
  switch ( Action[state][event->type] )
  { case a1:  Action 1;  break;
    case a2:  Action 2;  break;
                   :
    case a99: Action 99; break;
  };
  state = Newstate[state][event->type];
} while (!terminated);
tidy up;
```

Figure 9: Program skeleton for implementation of finite state automaton

In languages such as CSP, CCS and LOTOS, on the other hand, the automata are disguised in the form of processes whose overall behaviour is formulated in terms of the basic events and composition operators of a process algebra. Implementation techniques for such languages commonly rely on the fact that the semantics of the process algebraic language can be given in terms of a *labelled transition system (LTS)*. In general terms, an LTS describes how a process whose behaviour is given by a process algebraic process expression of a particular syntactic form can perform a transition in which it takes part in an event and then behaves like a process with another form. This is expressed in terms of a set of *transitions* of the form:

$$P \xrightarrow{a} Q$$

where P and Q are processes (described in terms of behaviour expressions) and a is an event. For example, in CSP, the LTS might tell us that:

$$\begin{array}{rcl} (a \rightarrow b \rightarrow c \rightarrow P) & \xrightarrow{a} & (b \rightarrow c \rightarrow P) \\ (b \rightarrow c \rightarrow P) & \xrightarrow{b} & (c \rightarrow P) \\ (c \rightarrow P) & \xrightarrow{c} & P \end{array}$$

and similarly that:

$$(c!e \rightarrow Q) \parallel (c?x : \mathcal{D} \rightarrow P[x]) \xrightarrow{c!e} (Q \parallel P[e])$$

This is really just another way of thinking about an automaton: The state of the automaton corresponds to "where you are" in the execution of the process, i.e. to the form of the process expression describing the behaviour of the part of the process which has not yet been executed. The transitions in the LTS describe – for each event which is possible in a given state – how the state of the automaton changes when that event takes place.

A simple-minded implementation of a protocol entity described as a CSP process could thus be derived from the description by creating the state machine corresponding to the process, with each state corresponding to a position in the behaviour expression between two events. As an example, Figure 10 shows a CSP description of a sender entity for the Alternating Bit Protocol, with the individual states marked as (S1), (S2),.... Note that only the distinguishable states are marked; for example, after the event in QT in which a value is received for $a : E$ via channel *right*, the sender returns to state (S2) at the start of QT. The corresponding automaton is shown in Figure 11(a).

In practice, it would be more usual only to implement the part of the automaton which reacts to events corresponding to incoming messages, and to consider the output events as being part of the reaction. Likewise, the timer would typically not be implemented as a separate process, but as an object which could be started and stopped by (local or remote) procedure calls. This approach is illustrated in Figure 11(b). This style of implementation matches the implementation architecture shown in Figure 4. A sketch of the central part of a program derived from this automaton is shown in Figure 12.

$$
\begin{array}{ll}
\mathit{Sender} & \stackrel{\text{def}}{=} (S[1] \parallel \mathit{Timer}) \setminus \{up\} \\
S[n:\mathcal{Z}_2] & \stackrel{\text{def}}{=} (\boxed{\text{S1}}\,SAPA?x : \mathcal{M} \rightarrow QT[n,x]) \\
QT[n:\mathcal{Z}_2, x:\mathcal{M}] & \\
 & \stackrel{\text{def}}{=} (\boxed{\text{S2}}\,right!(n,x) \rightarrow (\boxed{\text{S3}}\,up!\textsc{set} \rightarrow \boxed{\text{S4}}\,(right?a:\mathcal{Z}_2 \rightarrow \boxed{\text{S5}} \\
 & \qquad (\textbf{if } (a = n) \\
 & \qquad \textbf{then } up!\textsc{reset} \rightarrow S[succ(n)] \\
 & \qquad \textbf{else } QT[n,x]\ ) \\
 & \quad \,[\!]\, right?a : E \rightarrow QT[n,x] \\
 & \quad \,[\!]\, up?t : \{\textsc{timeout}\} \rightarrow QT[n,x])\ )\ ) \\
 & \\
\mathit{Timer} & \stackrel{\text{def}}{=} (up?s : \{\textsc{set}\} \rightarrow \quad (up?r : \{\textsc{reset}\} \rightarrow \mathit{Timer} \\
 & \qquad [\!]\, up!\textsc{timeout} \rightarrow \mathit{Timer})\ )
\end{array}
$$

Figure 10: ABP sender entity

Here, $\mathcal{M}$ denotes the domain of messages, E denotes the domain of acknowledgment PDUs with erroneous checksums, and $succ(n)$ denotes $(n+1)$ mod 2.

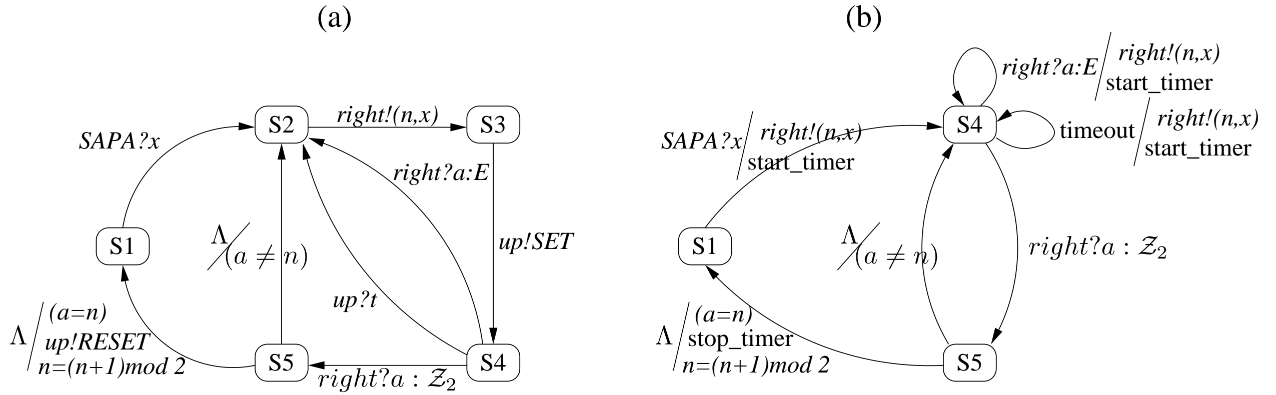

Figure 11: Finite state automata for ABP sender entity

(a) Directly derived from CSP specification
(b) Showing only reactions to incoming events

```
state      = S1;
terminated = FALSE;
// Initialise parameters and variables of processes S, QT
n          = 1;
x          = NULL;
a          = 0;
do
{ remove_from_event_queue( event );
  switch ( Action[state][event->type] )
  { case RECEIVE_MESS: // Action on SAPA?x:M
           x = event->data;
           y = assemble_PDU(n,x);
    case RECEIVE_EACK: // Action on right?a:E
    case TIMEOUT:      // Action on timeout
           add_to_departure_queue_down(y);
           start_timer();
           break;
    case RECEIVE_ACK: // Action on right?a:Z2
           a = *(event->data);
           if (a = n)
           { reset_timer();
             n = (n+1) % 2;
             state = S1;
             loop; }
           else
           { state = S4;
             loop; };
    case ERROR:        // Event not allowed in current state
           some suitable error reaction;
           terminated = TRUE;
           break;
  };
  state:= Newstate[state][event->type];
} while (!terminated);
tidy up;
```

Figure 12: A program derived from the FSA of Figure 11(b)

Declarations and code to initialise the two-dimensional arrays `Action` and `Newstate` are omitted.

For a small specification, the translation can be done quite easily "by hand", following a few simple rules: Incoming events (corresponding to CSP input, including receipt of timeouts) are removed from the event queue and dealt with according to their event type and the current state of the process. Outgoing events (corresponding to CSP output) are added to the relevant departure queue. Parameters and variables of the processes involved (in the example, `n`, `x` and `a`) become variables in the program. Recursive activations of the processes are replaced by the loop structure inherent in the program. To ensure that the implementation is *robust* – i.e. it does something sensible, even if unexpected events occur – the implementation should include well-defined actions even for incoming events which in principle should never occur in the current state of the process.

For more complex processes or sets of processes, the task of doing the translation by hand becomes insuperable. For some specification languages, an effort has therefore been put into developing software tools which can do the necessary translation, producing a program in some appropriate target language which can be executed or simulated on the desired target system. Finally, it should be mentioned that some more modern modular programming language environments may actually contain ready-made packages which make it possible to write and execute programs in a dialect of the actual specification language. Well-known examples are:

- The JCSP package offering a dialect of CSP within Java.
- The Concurrent SML facilities which offer CSP-like constructs within Standard SML.
- The CSP.NET package offering a subset of CSP in the .NET environment.
- The PythonCSP package offering a dialect of CSP in Python.

If such packages are available, the implementor obviously does not need to translate the specification, but merely to re-formulate it in the dialect appropriate to the package.

References

[1] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14:25–59, 1987.

[2] CCITT. *Recommendation Z.100: Specification and Description Language SDL*, 1988.

[3] J. B. Chen and B. N. Bershad. The impact of operating system structure on memory system performance. In *Proceedings of the Fourteenth ACM Symposium on Operating System Principles*, pages 120–133. ACM, December 1993.

[4] D. R. Cheriton. The V Kernel: A software base for distributed systems. *IEEE Software*, 1(2):19–42, April 1984.

[5] P. Druschel and L. L. Peterson. Experiences with a high-speed network adaptor: A software perspective. In *Proceedings of the SIGCOMM'94 Conference*, pages 2–13. ACM SIGCOMM, August 1994.

[6] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, 1985.

[7] N. C. Hutchinson and L. L. Peterson. The *x*-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, January 1991.

[8] International Standards Organisation. *International Standard ISO7498: Information Processing Systems – Open Systems Interconnection – Basic Reference Model*, 1984. This is identical to CCITT Recommendation X.200.

[9] International Standards Organisation. *International Standard ISO8807: Information Processing Systems – Open Systems Interconnection – LOTOS: A Formal Description Technique based on the Temporal Ordering of Observational Behaviour*, 1988.

[10] International Standards Organisation. *International Standard ISO9074: Information Processing Systems – Open Systems Interconnection – Estelle (Formal Description Technique based on an Extended State Transition Model)*, 1989.

[11] ITU-T. *Recommendation I.321: B-ISDN Protocol Reference Model*, 1972.

[12] R. Milner. *Communication and Concurrency*. Prentice-Hall International, 1989.

[13] R. Rashid and G. Robertson. Accent: A communication oriented network operating system. In *Proceedings of the Eighth Symposium on Operating System Principles*, pages 64–75. ACM, 1981.

[14] S.-Y. Tzou and D. P. Anderson. The performance of message passing and restricted virtual memory remapping. *Software – Practice and Experience*, 21:251–267, May 1991.