MASTER

# Functional design for a TCP/IP implementation in the Waterloo Port environment

Bezem, J.P.M.

*Award date:*
1988

5318

# Functional design for a TCP/IP implementation in the Waterloo Port environment.

by J.P.M. Bezem.

EU 145

Eindhoven University of Technology
Department of Electrical Engineering
Group: Digital Systems (EB)

# Functional design for a TCP/IP implementation in the Waterloo Port environment.

by J.P.M. Bezem.

Graduation report by J.P.M. Bezem.

By order of: Prof. Ir. M.P.J. Stevens,
University of Technology, Eindhoven.
Supervisors: Ir. J.P. Dubbelman,
PC Robo Automation BV, Valkenswaard.
Ing. L.A. van Bokhoven,
PC Robo Automation BV, Valkenswaard.
Ing. P.H.A. van der Putten,
University of Technology, Eindhoven.

Eindhoven, The Netherlands, August 1988.

## A B S T R A C T

In the first stages of preparation, working towards a design of a communication channel between Waterloo Port and Unix, a survey was made to compile an inventory of communication protocols in general use throughout the networking community.

Aside from the proprietary protocols from IBM (SNA) and Digital (DNA), the main protocols available were X.25 from OSI, and TCP/IP from DARPA.
TCP/IP was chosen because X.25 already is available for the Waterloo Port environment, and because X.25 is not yet very widely used throughout the Unix world.

The functionality of the principal protocols from the TCP/IP set is important, because a functional design should incorporate all functions present in the definitions.
IP is seen to offer a datagram oriented (Connectionless) service, fragmentation and reassembly to accomodate networks with a small maximum frame size, Internet-wide addressing, and, most important, routing.
The ICMP definition provides error reporting and internetwork management within the network layer, a separate protocol because of the connectionless orientation of IP.
ARP bridges the gap between Internet (IP) addressing and hardware addressing (eg. Ethernet), but only if the hardware is capable of broadcasting.
Finally, TCP does the rest: Connection management, acknowledgements and retransmissions, end-to-end flow control, and stream-oriented delivery.

In the final sections of this report a functional design for the TCP/IP protocol suite is presented using the access-graph method. The design is discussed in a top-down fashion, the interface functions are bundled in an appendix.

The functional design is a first step towards the eventual implementation. An important conclusion has to be that TCP/IP is a very complex structure, even for a set of communication protocols, and if this report can contribute to a better understanding of TCP/IP, it serves a good purpose.
The functional design itself serves another purpose: It is intended to be first in a series of design stages, each being more detailed than the previous one. Eventually a set of programs can be written (eg. for Waterloo Port) as an implementation of TCP/IP.

CONTENTS

## LIST OF FIGURES

## LIST OF TABLES

# 1. INTRODUCTION

As the need increased to provide a means of communication between the Waterloo Port network system and the Unix environment, PC Robo Automation, the Dutch distributor of the Waterloo Port products, started looking for a solution. The target for a solution should be something like 'an open communication channel between the Waterloo Port network system and a Unix system.'

In designing a communication channel the choice of which protocols to use is a major one. Therefore a study has been made of the protocol sets available, in order to make a sound choice, both technically and commercially (Chapters 3 and 4).
Chapter 3 discusses the OSI reference model and the related protocols, while in chapter 4 two large network architectures apart from the OSI structure are covered in some detail: SNA (IBM) and DNA (Digital).
On technical as well as commercial grounds a choice has been made (chapter 5), the TCP/IP protocol suite seemed to qualify best.

A short overview of the history of ARPANET is presented next (chapter 6), to provide some background on TCP/IP.
In order to understand a design for TCP/IP it is necessary to have some insight in the functionality offered by the various protocols. For this reason the Internet Protocol is discussed in chapter 7. Two supplemental network layer protocols, ARP and ICMP, are covered in chapter 8, and finally, in chapter 9, the functionality of the Transmission Control Protocol is elaborated on.

The functional design presented in chapter 10 is the first major step towards the realization of a communication channel between the Unix and Waterloo Port environments.

# 2. THE ASSIGNMENT

The assignment for this graduation project came from a need to connect a Waterloo Port network to a Unix host, in addition to the modem connection already possible.
Waterloo Port, or Port for short, also offers a general solution based on the X.25 protocol set. However, many Unix hosts have no X.25 link installed, so suggesting X.25 as a solution can present a commercial disadvantage: For an X.25 link to a Unix host both Port and the host need an extension in the form of an X.25 driver set.
A more viable option as far as the Unix side is considered would be Ethernet. The problem in this possibility lies with Port: There is no Ethernet support for Port.
This line of reasoning resulted in an assignment for a graduation project. The project was to have a somewhat broader scope than simply creating an Ethernet implementation for Waterloo Port, the reason why the original problem was chosen as the basis for the assignment.
The goal for this project can thus be formulated:

> Design a communication channel for Waterloo Port, to enable the Port network to gain access to a wide range of Unix computers.

The effort on the Unix side should eventually be minimal, so established (industry-)standards seem to be the most viable building blocks because of the degree of acceptance in the market.

The broader scope of the project was meant to avoid other feasible solutions from being overlooked. A small study, about networking and standards, seemed appropriate.

# 3. THE OSI REFERENCE MODEL

A short summary of the OSI reference model will be presented here. It is assumed that the reader is familiar with the layered structure of the model, and that he/she also has some basic knowledge of the terminology used in network discussions.

## 3.1 The OSI layering

The seven defined layers in the OSI reference model are pictured below. (Fig. 1)
A short discussion of each of those layers will be given.

| | | |
|---|---|---|
| 7 | Application | message |
| 6 | Presentation | message |
| 5 | Session | message |
| 4 | Transport | message |
| 3 | Network | packet |
| 2 | Data Link | frame |
| 1 | Physical | bit |

Fig. 1 The OSI layers and corresponding communication units

<u>Physical layer:</u>

This layer has the task to get the bits over in good enough shape to be recognized on the other side. Typical issues here are voltage assignments, bit-duration, full- or half-duplex, connector shapes, pin assignments and so on. The details dealt with here usually are of mechanical, electrical or procedural nature regarding the subnet.

Data Link layer:

Frames are the information units dealt with by the data link layer. Its task is to ensure error free transmission over the physical channel. To achieve that goal it usually breaks up the incoming data into frames, and then adds extra information to each frame, like checksums, to be able to distinguish a damaged frame from a correctly received one. This layer also deals with acknowledgements, retransmissions, duplicate frames and lost ones. This way the next layer can assume to be working on an error-free (virtual) line. Other issues addressed here are synchronization of receiver and transmitter, and acknowledgement piggybacking.

Network layer:

The network layer accepts messages, converts them into packets (the unit of information dealt with by the network layer) and sees to it that the packets get to the other side undamaged. The main issue here is routing, with an additional issue for larger networks, namely accounting functions. The routing functions also deal with congestion on the network, evading bottlenecks in some cases, while the accounting functions provide for billing the customer using the network.

Transport layer:

Multiplexing several virtual links onto one real link and vice versa can be done at several levels in the structure, but the transport layer is the highest layer in which it is allowed to occur. It is the first true end-to-end layer, where the previous lower layers deal only with their immediate neighbors, this layer communicates directly with its equivalent on the destination side. The transport layer also is the layer determining what functions will ultimately be available to users of the network. Typical transport protocols offer Connection Oriented Network Services (CONS), meaning that for all the layers above this one the communication seems to be over a dedicated link, permanently accessible for as long as the connection is left intact, like a human telephone call. The other possibility is ConnectionLess Network Service (CLNS), where isolated messages are guaranteed to arrive at the other end, but no commitment on the order of arrival is made. Another feature can be broadcasting messages (to multiple destinations). Only when the connection is established is the type of service determined.

Session layer:

The session layer provides the basic functionality of the user's interface to the network, ie. it determines what the user can and cannot do with the network. The issue of userfriendliness is addressed by the presentation layer, which only transforms the data going back and forth from the user to the session layer. Session layers are address-translators converting user-provided addresses to the required transport addresses. In communicating with their equivalent on the other side, session layers are negotiators, starting on a basic level of communication, and from there they ask the other session layer what extensions to the basic communication set can be used. This way the user can specify how the communication link must be used, if possible: Full- or half-duplex, reverse billing (only if the other side agrees, therefore negotiators) among others. This setting up a session is sometimes called binding.

Presentation layer:

This layer provides some commonly needed functions that have little or nothing to do with the network itself, but are in themselves useful in combination with the network. Prerequisite for inclusion in the presentation layer is that the functions will be used often enough to warrant a general solution instead of leaving it up to the user. Examples can be text compression, encryption/decryption, and several kinds of conversions such as character set conversion, file format conversion and terminal handling conversion.

Application layer:

The application layer can be seen as the network interface routines needed in an application, to be able to access the network facilities. The application alone determines what messages are allowed, in combination with the destination side's application layer of course, and also what actions will be taken upon reception of a message. Network transparency can be an issue here, hiding the network for the user so that, apart from the extended delays, if any, the user might think to be accessing

local resources. Other issues here are in the field of distributed storage and processing.
For more information on the OSI reference model see [Tan 1981].

These short descriptions indicate a strong tendency to separate functions into strictly functional groups: Each layer represents a different level of abstraction, performs a set of well defined functions and provides consistent interfaces towards the layers above and below.
Strict definitions offer a lot of advantages. One of the most promising is that different implementations for the same layer may be interchangeable if the interfaces are adhered to. One of the disadvantages apparent here is that all existing protocols will experience problems trying to fit in this tight scheme.
We will take a look now at some OSI protocols.

## 3.2 Some protocols in perspective

The OSI reference model has been established by the International Standardization Organization (ISO) in the late seventies, without any existing protocol available to demonstrate its feasibility: None of the existing protocols fitted in the reference model. Since that date many protocols have been adapted or specially defined for use in an OSI environment.
The first protocols fitting in the OSI structure were protocols for the lower layers, most notably the data link layer.
Later other layers were filled in, like layers three and four, network and transport. These extensions enabled some network architectures to conform to the only partially complete OSI standards, while filling in the higher layers with specific protocols. This increased the acceptance of the OSI standards even before the definitions were complete.
Only very recently some definitions have been approved of for the higher layers, and even these definitions are subject to constant revision as technology offers steady improvements over the existing network options. This is one of the main areas of concern with standardization efforts as big as these: To be fully compatible

some sacrifices have to be made in respect to using the latest technology available.
We will now discuss some well known standards adopted by the ISO OSI committees as OSI standards. Please be aware however, that the information presented here, about protocols being standard to OSI, is subject to constant change.

### 3.2.1 Ethernet

The Ethernet definition is an implementation of the physical and data link layers according to the OSI standard, although Ethernet$^{TM}$ (Ethernet is a trademark of Xerox Corporation) does not itself belong to the OSI standards. It is, however, widely used in the corporate environment, and sufficiently well defined to be called a standard in the usual sense. Furthermore, Ethernet is on the list of OSI protocols-to-be, in a slightly different format. It is relatively easy to make it fit the OSI structure.
An Ethernet connection consists of coaxial cable operated in baseband with a data rate of 10 Mb/s. Connections to Ethernet are standardized, and hardware is available from various manufacturers and resellers, often including the circuitry to provide CSMA/CD as the data link layer. Although a standard, Ethernet is available in two main forms, thick Ethernet and thin Ethernet, differing only in the wiring and the connectors. Conversion boxes are available, as are repeaters, making the number of stations that can be connected considerable.

### 3.2.2 Media Access Control

Media Access Control (MAC) is a general expression covering a number of protocol definitions. Each of these definitions is a protocol definition for part of the data link layer, the part that interfaces with the physical layer. They exist in a mutually exclusive fashion: If one is used the others are not.
Some of the currently adopted OSI standards were originally defined by the standardization committee of the IEEE, the Institute of Electrical and Electrotechnical Engineers. Standards are IEEE 802.3, IEEE 802.4, IEEE 802.5, meaning CSMA/CD, Token Bus, and

Token Ring respectively. CSMA/CD is most frequently used in combination with Ethernet or derivatives, Token Ring mostly on shielded twisted pair at 1 or 4 Mb/s. Token Bus is known under several different names, with ARCNET as a prominent example. The terminology used mostly in this context is: 'Operating with "the" token-passing mechanism.' This can be confusing at the very least. The most common implementations use coaxial cable at a bit rate of 5 Mb/s.
The ISO has added some other definitions, especially for fiber optic networks, and improved some of the IEEE definitions to fit into the OSI structure. These are of minor importance for now.

### 3.2.3 Logical Link Control

The Logical Link Control (LLC) definitions, in combination with the MAC definitions, form a full definition for OSI layer 2, the data link layer: Where the MAC deals with the hardware, the LLC deals with the data reliability. This could mean that the MAC should belong to the physical layer, but that would be a wrong conclusion. The hardware is perfectly capable to transmit bits on the medium, but if no MAC protocol is used, no one would know when to send a message, so chances are that anyone will start at random with no one to supervize and control the access to the medium. This function does not belong in the physical layer, therefore the MAC is present. The LLC protocol then provides for frame acknowledgements, duplicate frame suppression, and retransmission. If the interfaces to the various MAC protocols are consistent, the LLC can be identical no matter what MAC definition is used.
The LLC definition offers three distinct modes of operation:
LLC type 1 offers connectionless service, type 2 connection-oriented service, and type 3 single frame service. Types 1 and 2 are most often used, which one is dependent on the use of the network and on the specific choice of the designers. Type 3 is useful only in very specific environments.

### 3.2.4 Link Access Procedure

There are a couple of variations on the basic Link Access Procedure (LAP) protocol, depending on the way they are used, but the basic LAP protocol is a full featured data link layer definition, mainly associated with X.25 access to a telephone network (circuit-switched physical layer), or to a dedicated Packet Data Network (PDN).
LAP and LAPB (LAP - Balanced) are derived from IBM's SDLC (Synchronous Data Link Control), used with SNA (Systems Network Architecture). These 'standards' and many other derivatives are all almost mutually compatible, so: Not compatible at all.
All are bit-oriented protocols applying bit-stuffing for data-transparency. Frames can be of arbitrary bit-length, with checksums and flag sequences as delimiters.
This relatively simple definition of the data link layer uses a control field per frame to accomodate sequence numbers, acknowledgements and other information, and uses a sliding window technique size 8 to prevent frame loss from causing excessive performance degradation, because retransmission of all frames since the lost one would be necessary.
The LAP protocol can be used in combination with the MultiLink Procedure (MLP) to provide multiplexing of several channels over one link.

### 3.2.5 X.25

The X.25 definition effectively spans layers 1, 2 and 3 of the OSI model. The layer 1 definition in X.25 is not an actual new definition but rather a reference to some existing definitions. The CCITT (Consultative Committee on International Telephony and Telegraphy) had previously defined two standards, X.21 and X.21bis, for digital and analog interfaces respectively between hosts and directly connected intermediate nodes. The protocols used between two intermediate nodes are not defined. The data link protocols used for X.25 connections are LAP and LAPB as discussed before.
Most standards belonging to the X-series are to this date not

adopted as OSI standards, except for both LAP standards and the X.25 network layer definition.
The underlying structure of an X.25 network, layers 1 and 2, make up a datagram based network, the network layer definition offers both connectionless as connection-oriented service. Permanent virtual circuits are possible with X.25.

### 3.2.6 Some other OSI protocols

The ISO has defined several other protocols. Many more at the layers discussed; they are omitted because the general acceptance of these protocols is still low. Some more at higher levels; detailed discussion is avoided about them because these adopted standards are relatively new, still subject to a lot of changes and not generally accepted in wider areas. However, I will name a few, just for reference.
The OSI transport Protocol provides transport encryption and five classes of operation: Simple (type 0), basic error recovery (1), multiplexing (2), error recovery and multiplexing (3), and error detection and recovery (4). Provisions have been made for both connectionless as connection-oriented operation.
The most visible achievements of ISO have been on the application layer. Several protocol-using applications have been adopted, and some have even been tested - successfully - in a multivendor environment. Among them are:
MHS (Message Handling System), corresponding to X.400 of CCITT;
FTAM (File Transfer, Access and Management);
VTS (Virtual Terminal Service).

# 4. SOME OTHER NETWORK ARCHITECTURES

The OSI reference model is relatively young, it was established only in the late seventies. Some larger computer companies already had established their own networking architectures, with large installed bases. These architectures could not be abandoned as OSI appeared, because extended support would remain necessary for current customers. Notably, IBM has claimed that the OSI reference model is not going to take the place of its own SNA because that was not the primary objective for the OSI standards: OSI is going to be the glue between multivendor networks, but the internal architecture of one network is totally up to the manufacturer of that network as long as the possibility of an OSI gateway is provided.

Because of the impact of other architectures besides OSI, and because of the fact that these architectures have been fully functional for quite some time now, it seems appropriate to devote some effort to a short description of two major contenders besides OSI, IBM's SNA and DEC's DNA (Digital Network Architecture). TCP/IP has been seriously considered as well, but the TCP/IP discussion has been expanded beyond the scope of this chapter, see chapter 6 et seq. This description will not be more than an overview, in relation to the OSI reference model.

## 4.1 Systems Network Architecture

The IBM network architecture SNA has a layered structure like the OSI reference model. With some effort of will a correspondence can be seen between the two models, looking somewhat like pictured in figure 2.

| | | |
|---|---|---|
| 7 | Application | End user |
| 6 | Presentation | NAU services |
| 5 | Session | Data flow control |
| | | Transmission control |
| 4 | Transport | Path control |
| 3 | Network | |
| 2 | Data Link | Data link control |
| 1 | Physical | Physical |
| | OSI | SNA |

Fig. 2 The relation between OSI and SNA

The correspondences indicated here are, and should be, approximations. Especially the layers 3, 4, and 5 do not match well. Other people might map SNA differently.
The physical links used for SNA are not standardized.

IBM's Data link control layer consists of SDLC, a bit-oriented protocol, later adopted by several other manufacturers and standardization committees. (See chapter 3.4.2). To achieve data transparency SDLC and all derivatives use bit-stuffing.
The Path control layer in SNA offers virtual circuits to the layers above, reason why the layer cannot be fully equivalent to the OSI network layer; it also offers some functions of the OSI transport layer.

The Transmission control and the Data flow control layers extend this functionality by providing session control, as equivalent to the OSI Session layer. The role of the Data flow control layer is not to control the flow of data in the usual sense, but more with arbitration of which side is the next one to talk during a session. Also error recovery is an important task.
The NAU services layer provides two classes of service to user processes, in addition to the network services for dealing with the network itself: Presentation services like in the OSI Presentation layer, and session services for setting up connections.

## 4.2 The Digital Network Architecture

The correspondence between DNA and the OSI reference model is illustrated graphically in figure 3, below. Like with SNA, the indicated correspondences are approximations.

| | | |
|---|---|---|
| 7 | Application | Application |
| 6 | Presentation | |
| 5 | Session | (None) |
| 4 | Transport | Network services |
| 3 | Network | Transport |
| 2 | Data Link | Data link control |
| 1 | Physical | Physical |

OSI DNA

Fig. 3 The relation between OSI and DNA

The DNA definition is provided so that individual customers can construct their own network using Digital products. In this respect DNA is equivalent to SNA. DNA, however, is considerably simpler than SNA, resulting in smaller implementation modules with less options.

The first four layers of DNA correspond perfectly with the OSI layers, although Digital calls layer 3 the Transport layer instead of layer 4. DNA has no session layer, and the Application layer covers aspects of both the OSI Presentation layer and the OSI Application layer.
The options for the physical link supported by DNA are almost the same as for SNA. The DNA Data link control layer serves the same objectives as IBM's, but the implementations differ in several ways. For instance, Digitals implementation uses frames with a multiple of eight bits for length, the actual length is recorded in the header, while SNA uses bit-sequences to determine the actual frame length.
DNA's Transport layer (in OSI terms the Network layer) uses independent routing for all packets.
Because of this the implementation of the Network services layer is in no way similar to, say, SNA's Path control / Transmission control layers, although the resulting sets of services offered to higher layers are comparable.
As stated before DNA has no session layer. The presentation services offered by DNA are very basic, not like SNA at all. The only information conversion offered is with some file transfer option, where conversions are called upon when necessary.

It is important to notice that both the IBM and the Digital architectures have been developed with a specific idea about what customers would want, in the early seventies, with commercial motives. The OSI standards have been designed to be(come) everything for everyone, in as far as this was considered feasible. Main issue there was to provide network designers with a set of basic specifications to enable them to connect their network to others adhereing to the same set of rules. This divergence in objectives makes it possible that IBM does not consider OSI as opponent to their SNA, but as a valued addition to their own internetworking capabilities.

## 4.3 TCP/IP

The TCP/IP protocol suite is an industry standard not belonging to the OSI definitions. For a full discussion of TCP/IP see chapter 6 and subsequent chapters. It is mentioned here only because it was one of the protocols considered seriously; it is not discussed here because the extensive discussion in chapter 6 et seq would be necessary anyhow.

# 5. THE SELECTION PROCESS

In this chapter a choice will be made as to which protocol(s) are best suited for internetworking between Waterloo Port and a Unix host.

## 5.1 Selecting a set of protocols; requirements

A first selection has been made on grounds of the original target: Waterloo Port is a networking system for IBM PC's and compatibles, so protocol sets used exclusively in mainframe to mainframe communication are of little relevance here.
At the same time a Unix host is the target machine. This implies that if a protocol set is chosen for which no direct support is available for Unix machines, this protocol set must be implemented on the Unix host as well as on the Port system; such solutions are not discussed any further.
A third criterion in the first rough selection is an estimate of the installed base of the chosen protocol set: If only a very small amount of Unix users has installed the protocol set being considered for (inter-)network communication the choice must be a wrong one because the market potential for a Port gateway using those protocols is very low.
These three selection rules forbid any protocol designed from scratch, however optimal such protocol may be.
Let's see what options we may have.

### 5.1.1 Protocol options

Only the lower four layers of the OSI model need be considered, since the session, presentation and application layers will have to be adapted to the specific target Unix environment. If we start at the lowest level we can summarize the overview presented in the previous chapters as follows:

### 5.1.1.1 physical and data link layers

The choice of the physical layer is a choice of hardware. Several options are available. The most feasible choices will be Ethernet and X.25. All other options can be excluded on grounds of speed restrictions (serial lines) or costs (fiber optic cable). The two token-passing systems (Token ring and token bus) are not very well known outside the LAN-environment.
Both Ethernet and X.25 define some of the higher layers of the OSI model.
If a choice has to be made with only the physical layer in mind, the choice is Ethernet for several reasons:

- Ethernet is heavily used in Unix environments.
- X.25 is already available for the Waterloo Port environment.
- X.25 is not used much in Unix environments.

The data link layer is not very special, the only requirement is that it should fit in with the choice of the hardware used, since also the media access layer is part of the data link layer.

### 5.1.1.2 network layer and higher

If the physical layer offers only two realistic solutions, with one clear winner, the only issue to deal with at the network layer level and higher up is whether this solution is a winner in the higher range of protocol levels too.
In this case: Yes. Ethernet is used extensively in a Unix environment, mainly with TCP/IP on layers three and four. TCP/IP is an established industry standard, and the specifications are in the public domain. This more than compensates the possible drawback that TCP/IP is not an OSI standard.
In fact, it's even better. TCP/IP has always been under development following a simple rule:

> Use existing protocol standards whenever such standards apply; invent new protocols only when existing standards are insufficient, but be prepared to migrate to international standards when they become available. [Com 1988; p. 8]

## 5.2 Conclusions

The conclusion can be simple: Ethernet - TCP/IP is the best combination in the given case.
This general solution has three components, Ethernet, TCP/IP and the Port interface to both.
In view of this graduation project is seems unrealistic to suppose that all three components can be realized, if any at all, in the restricted amount of time available. Choices must be made.

### 5.2.1 How to proceed

In the next chapter an overview will be presented of the TCP/IP environment, ARPANET, and of the protocol suite itself. In subsequent chapters the functionality of the main protocols will be discussed.
After a first understanding of the TCP/IP basics has been acquired, a functional design of the TCP/IP protocol suite will be the next step.

# 6. ARPANET

The ARPANET network is one of the oldest networks still in operation. It served as a testbed for operational research since 1969, and through the years several protocol sets in many versions have been used on the network. The TCP/IP protocol suite is now the required set of protocols for all of ARPANET as well as for all networks connected. A little history will show how TCP/IP is a natural step in the continuous development of ARPANET.

## 6.1 The ARPANET protocol family

The structure of the ARPANET architecture as compared to the OSI structure can been seen in figure 4, below. The boundaries are known to be rough approximations, and the IMP-IMP layer does not fit into the OSI reference model too well.

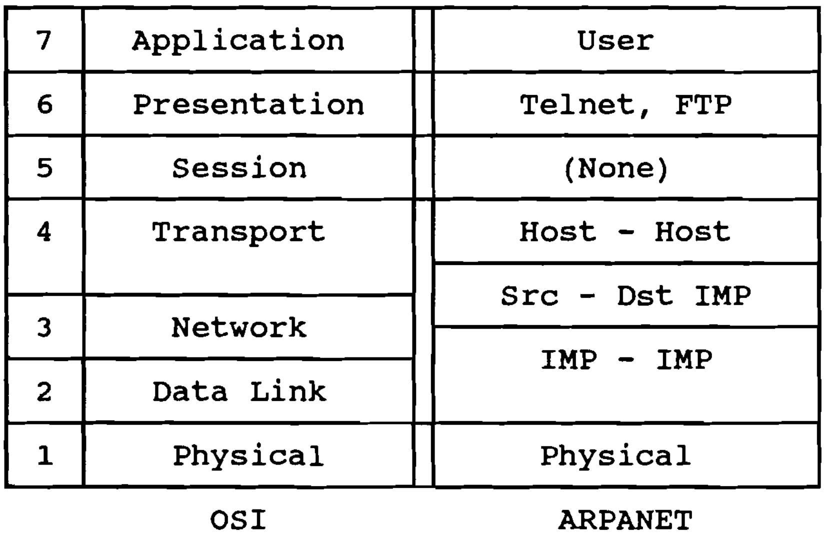


Fig. 4 The relation between OSI and ARPANET

This view represents the ARPANET in general. The protocols used in implementations differ, and their subdivision in layers can lead to different pictures than the general one in figure 4.

The physical layer can be implemented in many ways, to be chosen by the implementor. The IMP-IMP protocol (IMP - Interface Message Processors, the nodes in the ARPANET) does not fit in the OSI reference model. It comprises the Data link layer and also extensive routing information belonging to the Network layer. The protocol that is related to the IMP-IMP protocol is the Source-to-Destination-protocol, it verifies the correct reception of every packet sent by the source IMP at the destination IMP.
The Host to Host protocol is more or less equivalent to the OSI Transport protocol. In figure 4 this protocol is usually implemented with NCP (Network Control Protocol). NCP offers an error-free virtual circuit to higher levels, requiring a perfect subnet from the lower levels.
ARPANET does not have any session layer, and no specific presentation layer either, although some presentation services are available in the application layer protocols: Telnet for remote login, FTP (File Transfer Protocol) for file transfer over the network.

## 6.2 TCP/IP

In the early days the scheme devised functioned perfectly, the 50 kbps leased lines offered good quality connections, and the assumption of a perfect subnet made by the NCP protocol was valid.
However, as the ARPANET started expanding, and other types of networks had to be connected, the perfect subnet assumption became too restrictive. A new family of protocols evoluated from the original ARPANET protocols, offering the same error-free virtual circuits to layers above the transport layer, but using a transport protocol capable of using imperfect subnetworks.
Along with this TCP protocol (Transmission Control Protocol), a new network layer protocol was developed. The new functionality implemented in the new transport protocol required a rearrangement of functionality in the lower layers too. The IP protocol (Internet Protocol) corresponds more to the OSI network layer, offering the possibility to use any physical layer with an appropriate data link layer for the basic communication facilities needed.

These arrangements will be explained in some more detail using Ethernet as an example, since that is going to be our target configuration.

## 6.3 TCP/IP and Ethernet

The correspondence between OSI and TCP/IP with Ethernet is pictured in figure 5, below.

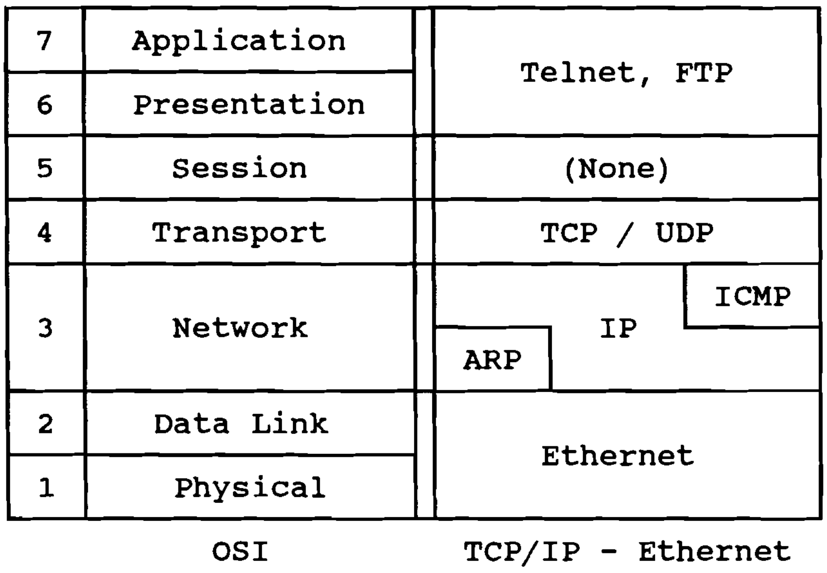


Fig. 5 The relation between OSI and TCP/IP-Ethernet

In figure 5, ignore the blocks ARP and ICMP for the moment. They will be explained a little later on.
The new figure then fits the OSI reference model rather well.
Ethernet covers two layers with one name, but even in Ethernet the two-layer functionality is easily separated, as the IEEE has indicated with its version of the original Ethernet, embedded in an IEEE 802 definition.
The other difference is the upper layer, covering presentation layer services in network applications.

Both extra blocks in the network layer are present out of a necessity.
The IP protocol was not designed specifically for use with Ethernet. A problem with interfacing Ethernet and IP (one of the very few) has been the address mismatch: Ethernet uses 48-bit addresses, IP uses 32-bits per address. To solve this problem, a new protocol has been introduced.
ARP, the Address Resolution Protocol, is a quite general protocol for network address mapping and can be used on any network hardware supporting broadcasting. It is not restricted to address lengths of 48 or 32 bit, but only used that way with Ethernet. In later chapters the functionality of ARP will be covered.
The ICMP protocol (Internet Control Message Protocol) offers a means of communication for the subnet. The subnet may not be able to transmit packets reliably, so it is possible that packets are lost. The transport protocol has the functionality to counter such loss with acknowledgements and retransmissions.
However, the routing mechanism is part of the unreliable subnet. If some intermediate router discovers an error, for instance an address cannot be reached, the packet can be discarded without further thought because of the assumption of an unreliable subnet. The transport protocol will then, after a timeout, decide that the packet must be lost, and retransmit the packet. This will lead to aborting the connection, eventually, but not after a considerable delay and a high network load.
It would be desirable to have a mechanism to notify the sender when such an event occurs. The ICMP protocol is used for such purposes.

A few highlights from ICMP:

- Notification if the destination cannot be reached.
- Subnet flow control; end-to-end flow control can be (and is) implemented in TCP, but if some intermediate router cannot handle the load, ICMP provides a way to notify the sender(s) to slow down the traffic on that link.
- Notification of the source when inefficient routes are being used.
- Notification when invalid routes (routing loops) or routes that are too long are being used.
- Notification when reassembly timeouts (after fragmentation) occur.

In short, many problems that can occur within the subnet and lead to discarding of packets, can be reported back to the sender using ICMP, especially when retries will not solve the problem. For a more detailed discussion of the ICMP protocol, refer to section 8.2.

In the next chapter the functionality of the Internet Protocol will be discussed in more detail. The Ethernet functionality is rather simple, and it will be discussed briefly. This only to offer a full overview for reference.

# 7. THE TCP/IP INTERNET PROTOCOL[1]

In this chapter a short overview will be presented of the Ethernet functionality; the IEEE 802.3 definition is functionally the same, the details are not covered here.
The main emphasis of this chapter will be the IP functionality, with references to the next chapter for the separate ARP and ICMP protocols. The IP functionality will be clarified by explaining the use of the IP-header fields.

## 7.1 Ethernet as a basis for the IP protocol

The interface of Ethernet with higher protocol levels provides a relatively modest set of services:

- Ethernet provides its own addressing scheme, 48-bits addresses. 47 bits are used for node-addressing. When the most significant bit zero is, the other 47 bits indicate a node-address.
  When not, the address is a multi-cast (some zero's) or a broadcast (all one's) address.
- An Ethernet frame contains a 2-byte type-field, used to indicate the level three protocol using Ethernet. This makes it possible to use several network layer protocols on the same Ethernet link in the same host: The type field indicates the destination network layer protocol.
- A 4-byte CRC check is appended to every Ethernet frame for error detection.

Except for the multi-cast facilities, the features mentioned are used by the Internet Protocol, and a sufficient basis. The CSMA/CD principle used in the Ethernet Media Access Control implements 15 retries after a collision detect. When that is not sufficient, the transmission attempt is aborted with an error interrupt or equivalent.

## 7.2 The IP header

The Internet Protocol header is a block of data, 160 through 480 bits in length (20 through 60 bytes), appended to every IP packet. The data block contains information necessary for the correct operation of the Internet Protocol layer. A representation of the various fields contained in the header is given in figure 6.
Some fields of the header are meant to assist in offering a specific functionality feature; these fields will be discussed in one of the subsections of this chapter.
Other fields are useful to the IP protocol in general; they will be discussed first.

The official definition (the latest) of the Internet Protocol can be found in RFC 791[1][2].

| | | | | |
|---|---|---|---|---|
| VERS | LEN | TYPE OF SERVICE | TOTAL LENGTH | |
| IDENTIFICATION | | | FLAGS | FRAGMENT OFFSET |
| TTL | | PROTOCOL | HEADER CHECKSUM | |
| SOURCE IP ADDRESS | | | | |
| DESTINATION IP ADDRESS | | | | |
| OPTIONS | | | | PADDING |

Fig. 6 Internet Protocol header

VERS (4 bit)

Indicates the version number of the used IP implementation. If the version number in an incoming packet does not correspond with the supported version(s), the message is better discarded than incorrectly processed. The current version number is 4.

---

1) References to RFC's can always be outdated because of new issues. When the information is of vital importance, always check the validity of a reference.

2) Indicated are the main references, eg. containing the specifications. For full details, more RFC's can be of importance.

LEN (4 bit)

Indicates the length of the IP header, in 32-bit words. The OPTIONS field has a variable length, but always a multiple of 32 bits using PADDING bits. The value of this field can vary from 5 up to 15.

TOTAL LENGTH (16 bit)

Indicates the length of the IP datagram in bytes (8 bits, octets), including both the header and the data portion of the datagram.

HEADER CHECKSUM (16 bit)

Contains a simple checksum of all header bytes. The header is divided in 16-bit words, and the checksum is then calculated as follows: The one's complements of all words are added, then the one's complement of the result is the checksum.

### 7.2.1 Type Of Service field

The Type Of Service field is meant as a hint to the routing algorithm, to provide a means of discriminating between two or more equally valid routes. The 8-bit field consists of five sub-fields, as indicated in figure 7.

| | | | | |
|---|---|---|---|---|
| Precedence | D | T | R | (Unused) |

Fig. 7 Type of Service field

- Precedence (3 bits) provides a way to distinguish important messages from less important ones. The US Department of Defense (DoD) has defined all eight levels, the amateur community uses four of those under different names. The top two levels are intended for internal use within the network; see table 1. Fact is that only very few routers have implemented precedence routing, although the precedence field could gain some importance in the future: It is possible to develop a congestion control protocol using the two topmost precedence levels, without disturbing the protocol with the congestion it is trying to control.

Table 1. Precedence levels

| Value | DoD names | Amateur names |
|---|---|---|
| 000 | Routine | Routine |
| 001 | Priority | Welfare |
| 010 | Immediate | Priority |
| 011 | Flash | Emergency |
| 100 | Flash Override | N/A |
| 101 | Critical | N/A |
| 110 | Internet Control | |
| 111 | Network Control | |

- Delay (1 bit) indicates a desire for a low delay in favor of high throughput or reliability. A value of 0 is 'off', 1 is 'on'. Keystrokes entered in a terminal emulator program, sent in an IP datagram, typically have this bit set to 1.
- Throughput (1 bit) is used (set to 1) in situations where a bulk of data is to be transmitted, eg. file transfer. The overall delay is not too important, neither is reliability in the case of a large text-file, whereas the total transmission time is proportional to the cost.
- Reliability (1 bit) requests a highly reliable connection. This is useful in the case where a code-file is transmitted: Every bit-error can be fatal to the program. Again, 1 is 'on', 0 is 'off'.

The two least significant bits are not used.

It is important to remember the TOS-byte is only a **hint** to the routing algorithm; if the router cannot comply with the request, it may simply ignore it (as many routers do). It should, however, copy the TOS-byte into all routed datagrams, to indicate the requested service to every subsequent router capable of providing such service.

### 7.2.2 Time To Live field

In a datagram based subnetwork providing routing functionality, every datagram may be routed individually. This can present problems if some available route is not correct. (It could occur theoretically that two routers transmit a particular datagram back and forth to each other because of a faulty routing table.) In such a case a datagram could stay in the network forever without ever being delivered.
The Time To Live field in the IP header is meant to aid in the destruction of such a runaround packet. It is simply a counter, set to the number of seconds a packet will be allowed to exist in the network before being delivered or destroyed. That means that every router checks for the TTL-field being zero, and every packet with a TTL value of zero is removed from the subnetwork. (See chapter 8.2 for the notification method used.)
One problem has risen this way: How can a unit of time, in this case a second, be determined accurately over a number of routers, without an enormous amount of overhead? For the time being the answer is not given. Effectively another unit of 'time' is used, equivalent to the "hop-count": Every router in the internet that receives a message, decrements the TTL-field before checking whether it is zero or not, and discarding the packet if it is. So the expression 'seconds' as a unit of measurement is at least confusing.
The overall effect, however, is the same: After a certain amount of time a packet still not delivered is destroyed.

### 7.2.3 Protocol field

If in a host several transport level protocols are active at the same time, the IP protocol needs some method to separate incoming messages and lead them on to the correct level four protocol. The protocol field is intended for this purpose.
A request for IP service by, say, TCP must specify the TCP protocol ID. This ID (6) is included in the transmitted IP header's protocol field. The receiving IP process can determine from this protocol ID that the TCP handling routines are the target protocol for this

message, and forward the data correctly. The principle is analogous to the Ethernet type-field (Chapter 7.1), and also to the TCP ports (Chapter 9).

## 7.3 Internet addressing

The addressing scheme used with IP is a simple one, basically. However, the possible expansions on this basic scheme are manyfold, and so are the possibilities of using them. Only the most important features of IP addressing will be covered.

### 7.3.1 Network-based addressing

In order to achieve hierarchy within the 32-bit address space, the 32 bits are subdivided. The subdivision permits three formats to accomodate both small networks with few hosts (Class C) and large networks with many hosts (Class A). An intermediate format for medium-size networks with a considerable number of hosts is provided also (Class B).
To determine the format the first few bits of the address are used, as indicated in figure 8.

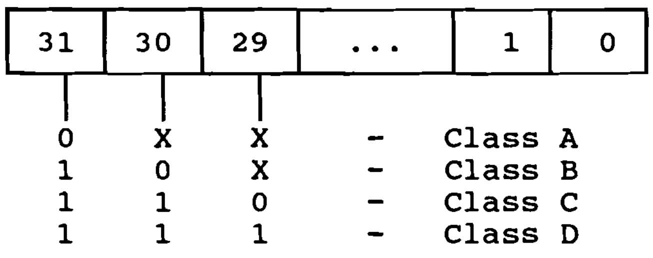


Fig. 8 Network types

In figure 8 the bit-values for the three most significant bits are shown to be the indicators for the rest of the IP address layout.
Class D is included only for completeness. Network class D is not used yet except in experiments considering the possibility of internet broadcasting. No standards have emerged yet.

The subdivision of the IP address in network number and host number is different for all three classes to accomodate for the different number of hosts to be expected in a network of the specific class. Table 2 lists the differences.

Table 2. IP address fields

| Class | Initial bits | Number of network bits | Number of host bits | Network mask 32-bit |
|---|---|---|---|---|
| A | 0XX | 7 | 24 | FF000000 |
| B | 10X | 14 | 16 | FFFF0000 |
| C | 110 | 21 | 8 | FFFFFF00 |

This division of the address space has some consequences, mainly concerning routing (obvious) and gateways (less obvious). The routing issues will be pursued first.

### 7.3.2 Internet routing

The way routing tables are set up within the Internet Protocol arrange for routing on network number only outside the target network, and host specific routing solely within the bounds of that network. Default routing further helps to restrict the size of routing tables.
For networks capable of broadcasting on the physical/data link levels, an additional method is given in chapter 8.1.

### 7.3.3 Internet gateways

The introduction of networks and hosts within a network poses an addressing problem for hosts connected to more than one network at once.
A host might be addressed by two or more node-numbers, with different network numbers. The routing mechanisms using network numbers first, the efficiency of routing a packet to such a node may depend on which node-number is used.

Even more important is the fact that such a host must keep track of the interface on which a message arrived, to provide the correct source address for a reply, if any. Also, a message can arrive at an interface with a different target address that the address of interface indicates, but still intended for that host. Such a message may not be forwarded onto the other network.

This consideration, while far from complete, serves to indicate that the overview presented here must inevitably be lacking detail. Interested readers may refer to the literature overview.

## 7.4 Fragmentation and reassembly

Physical networks tend to restrict their frame size. For instance, Ethernet imposes a strict frame size maximum of 1518 bytes.
If a message is routed over various networks with different physical characteristics, chances are that a packet arriving at a router is too big for the network it should proceed on. The source host cannot possibly know this, because every packet may be routed differently.
A possible solution adopted by the Internet protocol designers is fragmentation and reassembly.

The fields in the IP header involved in this functionality are the following:

- The identification field (16 bits) is meant to identify all the parts of the original datagram (may be only one). All fragments, if more than one, have the same identification number.
- The fragment offset field (13 bits) indicates the relative offset of the first byte in the fragment in respect to the original datagram. Unit of measurement is 8 bytes, so that with 13 bits (range 0 - 8191) the largest offset is 65,528. This is consistent with the maximum datagram size of 64k bytes.

- The flags field (3 bits) contains two flags and an unused bit. The "don't fragment"-flag, when set, indicates that the datagrams must not be fragmented, when, for example, the target host is not capable of reassembling the original datagram. If there is no possibility of transmission without fragmentation, an error is signalled to the source of the message (See chapter 8.2).

The "more fragments"-flag indicates that more fragments follow this one, in sequence rather than in time. The order of arrival of the various fragments cannot be guaranteed. Therefore the MF-flag, when reset (no more fragments follow), indicates that the fragment offset of this fragment is the highest FO occurring with this ID number, so with that and the length of the fragment the length of the original datagram can be calculated.

Fragmentation should be avoided if possible, because there are great disadvantages involved:

- From the point of fragmentation onward, all fragments, being sincere datagrams themselves, are routed separately to their common destination, to be reassembled only at the target host. The overhead for the network is increased. This would not be a very convincing argument on its own, but
- When only one of the fragments making up the original datagram is lost, the whole datagram must be discarded, because the IP protocol is datagram based, and does not provide for acknowledgements or retransmissions. This could increase the overhead considerably.

To avoid fragmentation, usually the TCP layer adjusts the segment size it uses to submit data for transmission to the maximum datagram size of IP.

## 7.5 Internet options

The option field in the IP header is of variable length, from absent (0) to 40 bytes. The length can be determined from the internet header length, in units of four bytes. If the actual length is not a multiple of four bytes, the free space is padded with zero's.

Every internet option is a sequence of one or more bytes. The two options of one byte length effectively do nothing, 0 meaning "end of option list", used for padding, and 1 being the "no operation" code.
The other options consist of an opcode of one byte, a byte indicating the length of an option in bytes, and further data.
The opcode contains a field, the most significant bit, indicating whether the option should be copied to all fragments when fragmentation is necessary (1), or only to the first fragment sequentially (0).

In short, options are defined for the following functions:

- Security enforcement
- Routing along predefined paths
- Recording the followed route for debugging purposes
- Timestamping datagrams for debugging purposes
- Identifying a stream (This is used to allocate resources when a certain amount of traffic is expected to use the same route, and the resources needed can be claimed only with difficulty, as in satellites)

# 8. ADDITIONAL NETWORK LAYER PROTOCOLS

In addition to the Internet Protocol two other protocols are of importance in the network layer. One is a required protocol for every IP implementation offering subnet error signalling. The other is an address mapping protocol for linking IP addresses with hardware addresses, optional, and requiring a physical/data link layer capable of broadcasting.
The latter will be discussed first, considering an implementation on Ethernet.

## 8.1 The ARP protocol

Ethernet addresses of 48 bits length are not easily mapped to IP addresses or vice versa: The Ethernet addresses are fixed for every controller, while the IP addresses of the nodes using these controllers are dependent on the network in which they are used. What is more, when a controller is replaced the Ethernet address changes, but the IP address does not.
Therefore the mapping must be made in a table, linking every known local IP address to their respective Ethernet addresses. (This does not mean that some central site, or worse: every host, should contain all the mappings for every host in the global Internet, because of the network-based routing algorithms.)
For large Ethernet networks this could result in large tables resident in every host, with most of the information unused for most of the time.
The Address Resolution Protocol was created to make the table maintenance dynamic: Information needed is obtained, from the table if possible, from the network if necessary. If the information is obtained from the network, it is stored in the table for later reference, because it is reasonable to suppose that the information will be needed again in the near future. On the other hand, information that has not been used for a specified amount of time is removed from the table.

## 8.1.1 ARP messages

The ARP protocol is implemented as a protocol directly on top of the data link layer. This means that for an Ethernet implementation, a special Ethernet type number is assigned for use in Ethernet frames: $0806_{16}$. The protocol can therefore send and receive its own frames.
The message format used has no fixed header-data separation. Refer to figure 9 for the message format used with IP on Ethernet.

| | | |
|---|---|---|
| HARDWARE | | PROTOCOL |
| HLEN | PLEN | OPERATION |
| SENDER HARDWARE ADDRESS (Bytes 0 - 3) | | |
| SENDER HA (Bytes 4 - 5) | | SENDER IA (Bytes 0 - 1) |
| SENDER IA (Bytes 2 - 3) | | TARGET HA (Bytes 0 - 1) |
| TARGET HARDWARE ADDRESS (Bytes 2 - 5) | | |
| TARGET INTERNET ADDRESS (Bytes 0 - 4) | | |

Fig. 9 ARP message format for use of IP on Ethernet

The fields in figure 9 have functions as follows:
- HARDWARE (16 bits) indicates the physical/data link subnetwork used. A value of 1 indicates the 10 Mbit/s Ethernet.
- PROTOCOL (16 bits) indicates the layer 3 protocol using ARP for address resolution. The values used in this field are dependent of the way the hardware uses to identify the protocol. In the case of Ethernet, the type value $0800_{16}$, as used by Ethernet, indicates the IP protocol.
- HLEN (8 bits) and PLEN (8 bits) indicate the lengths of the hardware address and the protocol address respectively, measured in bytes. In this case the HLEN is 6 for Ethernet addresses, the PLEN value is 4 for IP addresses.

- OPERATION (16 bits) indicates the aim of the message. It may indicate a request (Value 1) or a response (Value 2). Other values are possible too, but these values are not part of the ARP definition; the Reverse ARP (RARP) protocol uses the same message format, however, on Ethernet a different type value of $8035_{16}$ is used. See chapter 8.1.2.
- SENDER HARDWARE ADDRESS (Size dependent on HLEN) contains the hardware (Ethernet) address of the sender of the message.
- SENDER INTERNET ADDRESS contains the layer 3 protocol address of the sender. (If known, see chapter 8.1.2.)
- TARGET HARDWARE ADDRESS contains the Ethernet address of the receiver. If this ARP packet is a request, this field is empty.
- TARGET INTERNET ADDRESS contains the IP address of the receiver. It should contain a value always.

In case of a request, the TARGET HA is empty and should be provided by the host identified with the TARGET IA. To reach the intended target machine, a physical broadcast is used: Stations with a different IP address simply ignore the request, and only the target issues a reply.
In case a reply is issued, the OPERATION field should be changed to reflect this, and the two TARGET fields should be swapped with the two SENDER fields, to reflect the direction in which the message is passed.

The previously mentioned RARP protocol has a functional link with ARP, and in implementation the similarities are great. A brief discussion is justified.

### 8.1.2 The RARP protocol

For diskless workstations in a network using TCP/IP as network protocol it may present a problem to find out its own internet address: It is the basis for all useful network communication.
The Reverse Address Resolution Protocol (RARP) provides a solution.

As stated previously, the RARP protocol uses the same packet layout as the ARP protocol, refer to figure 9. The means to differentiate the two protocols, both being self-contained protocols on top of a service like Ethernet, is the Ethernet type field. RARP uses a value of $8035_{16}$.
The values of the OPERATION field (figure 9) are request (3) and reply (4). A request contains only the SENDER **HARDWARE** ADDRESS field, the other fields are empty. A request is broadcast over the network to all stations.
At least one RARP server should exist on every network using RARP. The RARP knows the physical address to IP address link, completes the message and returns the reply. After that the diskless station can communicate using its own IP address to communicate with the host offering the RARP service, since its IP address and Ethernet address can be learned from the RARP reply too.

## 8.2 The ICMP protocol

The very nature of the datagram-oriented IP protocol implies that no guarantees can be made regarding delivery of packets. As soon as a packet is transmitted successfully over to the first intermediate router, it is out of the hands of the source host. The network layer protocol will never know whether the message was received by the destination host, or not. And if not, why not.
The source host may know why, but only if the target host detects the problem on a higher protocol level than the network layer, and notifies the source. If a router in between detects a problem, IP itself does not provide a way to report that.
The Internet Control Message Protocol (ICMP) does.

### 8.2.1 Message format and delivery

The ICMP protocol was designed to provide network control and error reporting within the network sublayer. A source host needs to know that a destination cannot be reached, or it will keep sending retries, using network resources senselessly.

A network layer protocol can only use the functionality offered by the data link layer, as can be seen with the ARP protocol, chapter 8.1; in that situation nothing else is needed.
However, if the last router in a series detects the error to be reported, the error message needs to be transferred over one or more routers, functionality offered by the IP protocol, but most definitely not by the data link layer.
That is why the ICMP protocol is functionally a part of the network layer, though implemented as if it were a layer four protocol using IP for network layer services.
One important difference is that if an ICMP packet causes an error to occur, preventing delivery, no ICMP message is generated. This keeps routing errors from generating ICMP messages in a circular fashion, not shut down by the regular use of the Time-To-Live field within IP, or congestion reporting messages from generating a flood of similar messages adding to the congestion, instead of reporting them.

Although all ICMP messages have their own format, they all start with the same three fields:
- a TYPE field, 8 bits, indicating the function of the message;
- a CODE field, 8 bits, further specifying the function if necessary;
- a 16-bit checksum field covering the complete ICMP message.

For the ICMP messages reporting an error the IP header of the packet causing the error is part of the message, as is the first part (8 bytes) of the data contained in the message. This can be of help to identify the source of the error more precisely.
The format of the rest of the message is dependent of the value contained in the type field. The defined numbers are listed in table 3.

Table 3. ICMP defined type numbers

| Type Number | ICMP Message |
|---|---|
| 0 | Echo reply |
| 3 | Destination unreachable |
| 4 | Source quench |
| 5 | Redirect |
| 8 | Echo request |
| 11 | Time exceeded |
| 12 | Parameter problem |
| 13 | Timestamp request |
| 14 | Timestamp reply |
| 15 | Information request |
| 16 | Information reply |
| 17 | Address mask request |
| 18 | Address mask reply |

The functions of these ICMP messages will be discussed in short.

The Echo Request (8) and Echo Reply (0) messages are used to test whether a destination can be reached. The CODE field is zero always, and aside from TYPE and checksum fields the sent and received messages should be identical.

The Destination Unreachable (3) message is returned whenever a packet cannot be delivered to the ultimate destination. The CODE field is used to specify the entity that cannot be reached, or the reason why the delivery has failed:

| | | |
|---|---|---|
| 0 | Network | |
| 1 | Host | |
| 2 | Protocol | (Transport layer; see chapter 9.) |
| 3 | Port | (See chapter 9.) |
| 4 | Fragmentation needed, DF set on packet | |
| 5 | Source route failed (when a routing option was specified) | |

The Source Quench (4) message is sent to the source of a packet when the router is too busy to handle it properly. This should indicate that the transmission rate of the transport protocol should be lowered for that connection because of congestion.

The Redirect (5) message is sent to the source of a packet if a router discovers that the host could use a more efficient path to the destination. It indicates the router (TCP/IP-gateway) to which all packets for the original destination should be sent in the future. The CODE field indicates what 'part' of the destination should be involved in the routing change:

| | |
|---|---|
| 0 | Redirect for the network |
| 1 | Redirect for the host only |
| 2 | Redirect for the network, when this type-of-service is required |
| 3 | Redirect for the host only, when this type-of-service is required |

So 0 and 2 indicate routing changes for all hosts on the network (The original IP-header is included in the message and contains the original destination IP-address and the TOS-byte), while 1 and 3 refer to packets for the specific destination host only. At the same time, 0 and 1 are general updates, while 2 and 3 only apply to packets with a specific TOS-field.

The Time Exceeded (11) message is generated by a router when a datagram is discarded because of a zero TTL(time-to-live)-field. This hints at a faulty route. Also when a fragmented packet is not received intact (a fragment takes too long before arriving) a message type 11 is generated (by the receiving host, this time). The CODE field distinguishes the two cases: 0 for a zero TTL-field, 1 for an exceeded fragment reassembly time.

A Parameter Problem (12) message is returned whenever a problem is experienced in processing an IP-header. A pointer is provided as to where in the header the problem seems to be.

Other messages provide means to calculate the round-trip-time of a message (Timestamp Request and Reply), to obtain a network address from a router as an alternative to the RARP protocol with the Information Request and Reply (see chapter 8.1.2), and to obtain a subnet-mask (Address Mask Request and Reply).

For more information about the ICMP-messages, layout and usage, refer to [Com 1988], chapter 9. The subnetting principles are covered in more detail in [Com 1988], chapter 16.

# 9. THE TCP/IP TRANSMISSION CONTROL PROTOCOL

Layer four in the TCP/IP protocol suite offers a few options to choose from. The two main protocols in this layer are the Transmission Control Protocol (TCP), and the User Datagram Protocol (UDP).
UDP offers datagram service to higher protocol layers and users, is not connection oriented and offers no acknowledgements. The main feature UDP can offer in addition to the IP functionality (see chapter 7) is a mechanism to have many points of entry to the protocol, using the same technique as TCP for that: Ports are used in both TCP and UDP. A description of this mechanism is included in this chapter.
The main subject discussed in this chapter is TCP. It is the most widely used protocol within a TCP/IP environment, and is included in every TCP/IP implementation.
TCP offers a reliable, connection-oriented service between two sides of the link. As with the Internet Protocol in chapter 7, TCP will be discussed along the lines of the TCP header-structure.

## 9.1 The TCP header

A TCP protocol unit, usually called a 'segment', is packed completely in the data portion of an IP packet. The TCP header is just plain data as far as the IP protocol layer is concerned. For the transport layer, however, the TCP header contains valuable information concerning acknowledgements, final destination and connection state changes.
First, the fields used solely for the TCP protocol itself will be discussed, after which the TCP functionality offered to higher level protocols will be covered in several subsections, each referring to the fields of the header it uses for achieving such functionality.
The format of the TCP header is presented in figure 10.
The various flags in the FLAGS-field are laid out in figure 11.

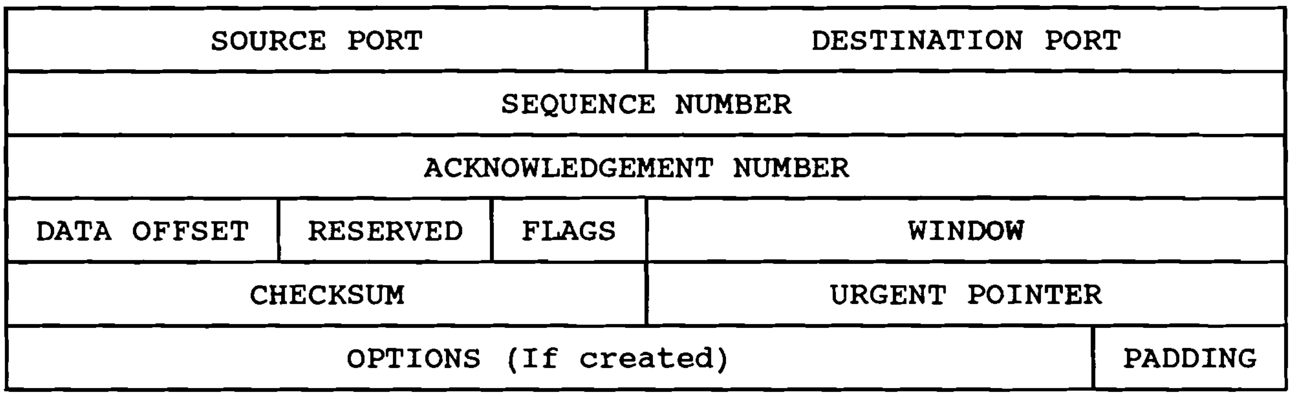


Fig. 10 The TCP header format

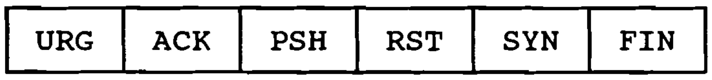


Fig. 11 The FLAGS-field layout

DATA OFFSET (4 bit)

Indicates the length of the TCP header in 32-bit units, including the options field. Thanks to the PADDING field after the options, the TCP header always is an integral number of 32 bits long. This field is used to compute the beginning of the data.

RESERVED (6 bit)

This area is reserved for future use. It should be zero always.

RST (FLAGS field, 1 bit)

The RST flag indicates a connection reset when '1'. This can be used in extraordinary situations, such as host crashes, to try to agree on a connection abort. It should only be used when resynchronization is bound to fail anyhow.

CHECKSUM (16 bit)

The CHECKSUM field covers the complete header and all data in the TCP message. In addition it also covers a so called 'pseudo-header' consisting of the source IP address, destination IP address, the PROTOCOL field value used by the Internet Protocol, and the TCP segment length. These data are placed as indicated in figure 12, and conceptually prefixed to the TCP header in calculating the checksum.

The pseudo-header is not transmitted with the TCP segment, it is just used in the calculation of the checksum, while its fields are to be extracted from the IP header of the packet carrying this segment.

The purpose of such a pseudo-header is to enable the TCP layer to verify that the segment has arrived at its correct destination; that destination consists of an IP address and a port number (see chapter 9.5), but also a protocol identification, since the UDP protocol uses the same port numbering mechanism.

| | | |
|---|---|---|
| SOURCE IP ADDRESS | | |
| DESTINATION IP ADDRESS | | |
| ZERO (0) | PROTOCOL | TCP LENGTH |

Fig. 12 The TCP pseudo-header

The ZERO field consists of eight zero-bits, the PROTOCOL field is the same 8-bit value used in the IP header to signify the TCP layer four protocol, and the TCP LENGTH field simply contains the total length of the TCP segment.

In order to calculate the checksum, the CHECKSUM field is defined to be zero, and the pseudo-header is prefixed to the actual header. In addition to the PADDING field in the header to make the header an exact multiple of 32bits long, the field containing the data transmitted by TCP can be of arbitrary length. For the purpose of computing the checksum the data field is padded with zero bits until the length is an exact multiple of 16 bits (not 32 !), and

the checksum can be computed. These padding bits (a byte of value 0 is appended or not) are different from the PADDING field in the header in that they are not part of the data field, they are not transmitted at all. They are just a mechanism to aid in the computation of the checksum, just as the pseudo-header.
The algorithm used to compute the checksum is the same as used for the IP header checksum: The 16-bit one's complement of the one's complement sum of all 16-bit words in pseudo-header, header and data.

## 9.2 Reliable service and flow control

TCP offers reliable service to higher layer protocols or applications. To do this it uses the services of the Internet Protocol: Datagram oriented unreliable packet transfer, on a connectionless subnet.
This mismatch between services offered by IP and services expected from TCP determines most of the TCP functionality. This required functionality calls for some means of acknowledgements.

TCP uses a technique called the 'sliding window' concept to send and acknowledge data, or to retransmit data in case of failure.
This concept has been extended to provide a flow control mechanism. Both parts of the concept will be discussed separately.

### 9.2.1 The sliding window concept

The simplest form of 'Positive Acknowledgements with Retransmission' has at most one unit (message, byte, ...) waiting for an acknowledgement from the receiving side. The next unit is not sent until an ACK(nowledgement) for this one has been received. This method tends to result in a poor bandwidth efficiency, especially when packets are lost occasionally.
The sliding window principle tries to improve on the PAR-method. It establishes a window indicating the units that can be transmitted without receiving acknowledgements. See figure 13; the double-lined box indicates the window.

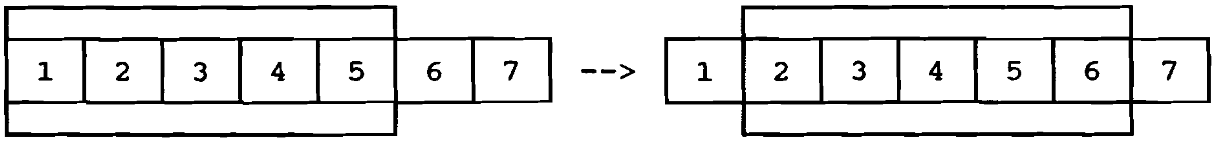


Fig. 13 A 'sliding window'

The window size here is five units. This means that units 1 through 5 can be sent without receiving any acknowledgements. As soon as an ACK for unit 1 has been received, the window slides, taking unit 1 out and bringing unit 6 in. It is now waiting for ACKs on units 2, 3, 4, 5, and 6 (after transmitting unit 6 of course).
This concept can be enhanced further by accepting cumulative acknowledgements. This means that the receiver, after receiving units 1 and 2 correctly, can send an acknowledgement for both units in one time: It is implied that by acknowledging a specific unit, all units with sequence numbers lower than the acknowledged one are also ACK-ed. This can reduce the number of ACK-packets dramatically.

### 9.2.2 A variable-size window

The window size used in this sliding window concept is an important parameter in determining the resource consumption of a connection. The administration of the sliding window should keep track of acknowledgements for all units that have been sent but not yet ACK-ed, and what is more, the sending side should keep all units handy, just in case a retransmission is necessary.
This feature enables the designer of the communication protocol to include flow control in the sliding window concept: By making the window size a variable rather than a constant, flow control is rather easily implemented.

### 9.2.3 The TCP variant

The Transmission Control Protocol uses the methods indicated above.
The SEQUENCE NUMBER field indicates the sequence number of the first byte contained in the data field following the header. Every byte in the data stream has its own sequence number, so the unit in the TCP case is a byte. This SN field is used to regulate the traffic towards the receiving side.
The WINDOW field and the ACKNOWLEDGEMENT NUMBER field take part in controlling the communication towards the sender of this message, in addition to the ACK flag in the FLAGS field of the header.
The ACK flag indicates whether the ACKNOWLEDGEMENT NUMBER field contains a valid ACK number.
The ACK number signifies, in the numbering scheme used by the other party, the sequence number of the first byte that was not yet received (correctly), implying that all bytes before that were: Cumulative acknowledgement.
The WINDOW field represents the number of bytes this side is willing to accept, starting with the one indicated in the ACKNOWLEDGEMENT NUMBER field. This window size is thus made variable, successfully implementing flow control (Only end-to-end; for flow control on a lower level see chapter 8.2 on the ICMP protocol).

## 9.3 Connection management

The TCP protocol covers a small part of the OSI session layer in that it covers connection (= session) management.
TCP uses the SYN and FIN flags in the FLAGS field for this purpose.
Also the SEQUENCE NUMBER field and the ACKNOWLEDGEMENT NUMBER field are used, to implement the three-way handshake synchronization procedure.
The three-way handshake serves two purposes:
- It synchronizes both sender and receiver, verifying that both are ready to start a connection.
- It establishes the initial sequence numbers both sides will use in transmitting and acknowledging data.

To terminate a connection the FIN (for final) flag is used.

Connection setup and connection breakdown will be discussed separately.

### 9.3.1 Connection setup

The three-way handshake used in connection setup with TCP can be used when one side initiates the connection, the other one listening, but also when both sides try to initiate a connection at the same time. The basic principles of the three-way handshake will be covered here, the reader is referred to the protocol specification for the details.[3)]

A synchronization message is identified by a SYN flag equal to 1. The sequence number in its TCP header is the number of the SYN message, the data in that message starts with a sequence number that is one higher. In this way the SYN flag itself can be acknowledged.
The sequence numbers used to start the connection are not fixed, because delayed duplicate messages from other connection attempts might be mistaken for belonging to this attempt, thereby leaving one side confused as to the connection status.

The simplest exchange of messages to establish a connection looks as follows:

[3)] The TCP specification can be found in RFC 793.

```
Site 1:
Send SYN with sequence number X.
                                \
                                 Site 2:
                                 Receive  SYN,  record  sequence
                                 number X.
                                 Send SYN with sequence number Y,
                                 include ACK for X+1.
                                /
Site 1:
Receive  SYN,  record  sequence
number Y.
Accept ACK for X+1.
Send ACK for Y+1.
                                \
                                 Site 2:
                                 Accept ACK for Y+1.
```

In this way both sides know the sequence numbers the other side will use, and they know (from receiving the correct acknowledgements) that the other one knows and is ready to send and receive data. The connection has been established.
Keep in mind that and ACK for X+1 means that the sending side has received everything including X correctly, and is expecting X+1 as the next sequence number.

The fact that SYNs have their own sequence numbers makes it possible even for the handshake messages to contain data. If that is the case, the data should be held until the connection has successfully been established. Details can be found in the protocol specification.

### 9.3.2 Connection breakdown

A TCP connection is a full duplex link: Both sides can transmit data to the other on the same link.
The TCP connection, however, can best be viewed as a double half-duplex link: Both sides can terminate their side of the connection without affecting the other side.
If one side has no more data to send, and does not need the connection any longer, it can close the link. The other side can continue to send data (the data being received in the normal way) indefinitely. The connection is not fully closed until also the other side has decided to close its side of the connection.
It is not possible, after having closed down one side of the connection, to change your mind and reopen it, not even when the link in the other direction is still busy.

The mechanism used in closing down one side of the connection is quite simple: The FIN flag from the FLAGS field in the header is used to indicate that no more data is being sent over the link, after the data contained in the message of the active FIN flag has been correctly received.
To achieve a clean connection close, the FIN flag itself has been assigned a sequence number: One more than the sequence number of the last data byte in the message. The FIN flag can thus be acknowledged.
If an ACK for the FIN flag is received, that side of the connection is closed. The other side continues to transmit, expecting ACKs in return, until a FIN is sent.
Keep in mind, that once a FIN is sent, the side wishing to close the connection should wait for the corresponding ACK, because it might be necessary to retransmit some data, or even the FIN flag itself.

## 9.4 Forcing data delivery

As far as the application using TCP is concerned, the protocols transmit a stream of bytes from one host to another. How the protocol does it is not important.
However, there are some situations where the application wants some control over the TCP buffering mechanisms, especially in interactive use (eg. TCP terminal emulation).
For this purpose the TCP protocol uses the PSH flag in the FLAGS field (PSH for push). It forces TCP to transmit the data received so far immediately. But it does more: It also forces the receiving TCP host to deliver the data in that segment to the higher layer protocols or applications immediately (that is the reason why a flag in the TCP header is used: The other side should get the same note, to force delivery).

In addition to the PSH flag, an application can send urgent data using TCP. The feature is not implemented very often.
The URG flag in the FLAGS field (URG for urgent) indicates whether the URGENT POINTER field contains a valid pointer or not. This pointer, when valid, indicates the last urgent byte in the segment. It is to be interpreted as a positive offset from the SEQUENCE NUMBER field value.
It can be used to increase the priority with which the data is being processed.

## 9.5 TCP ports

On small microcomputers operated under a single-tasking operating system such as MS-DOS, the need to have several TCP connections open at one time may not be great. On large multi-user multi-tasking hosts, however, it would be a grave shortcoming if such would not be possible.
TCP does have the capability for maintaining multiple links, if necessary with as many different hosts as there are connections.
TCP communicates with higher layer protocols and applications through the use of ports, abstractions of entry points to the protocol. A port can be seen as the number of an application within

the host. Numbers are assigned to applications on request, if and when the number is not currently assigned to another application. (A higher layer protocol can be seen as an application as far as TCP is concerned.)
The application opens a connection in the following way:

- Request permission to use local TCP port <l> for communication.

  If the port <l> is not currently involved in active communication for another application process, the request is granted.

- Initiate the three-way handshake to synchronize with a remote port. The remote host needs be specified by issuing a remote IP address <ipr>, and a remote port number <r>, a port local to the remote host indicated by <ipr>.

  If the remote host is within reach using <ipr>, the remote port <r>, local to host <ipr>, is not engaged in active communication but listening to any incoming traffic, (the port must be willing to passively initiate communications, otherwise the connection request is denied!) the three-way handshake is completed and a connection established.

A connection is established between two hosts using both IP addresses and ports: An IP address for the local host plus a corresponding port number, the combination called a (local) socket, and an IP address plus a port number for the remote application, also called a (remote) socket.
Since the IP addresses are contained in the IP header and not directly part of the TCP protocol definition, only the source and destination ports are contained in the TCP header.
Once a connection is established between two ports, it could occur that a stray TCP segment on the network would mistakenly be accepted as belonging to that connection, just because the port numbers match: The IP layer has mis-routed the message.
To avoid this, the concept of the pseudo-header has been invented: A message transmitted by a UDP protocol, using the same sockets for source and target, is singled out by the PROTOCOL field of the pseudo-header. TCP has a different protocol-id than UDP, therefore

the message has to have a different checksum when calculated by a different protocol. (UDP uses the same pseudo-header as TCP does.) For the same reason both IP addresses (source and target) are part of the pseudo-header.
Another problem in the usage of ports as indicated above can be the establishment of a connection when two newly acquainted hosts want to communicate: What ports do they use ? Misunderstandings are all around ...
A way to solve the problem - used by TCP and UDP - is to standardize some of the port numbers for some equally standard applications: A mail server, no matter on what kind of host, always listens on port number 25 (SMTP, decimal) for requests in respect to electronic mail; the SMTP definition takes care of what requests can be honored, and how they should be formatted; a terminal emulation server always listens to port number 23 (TELNET).
A list of preassigned port numbers (only numbers between 0 and 255 inclusive are assigned; 256 and above are free for use) can be found in [Com 1988], p. 149, figure 12.11.

## 9.6 The 'Maximum Segment Size' option

The Maximum Segment Size (MSS-) option is the only TCP option generally accepted (Apart from the End-Of-Option-List, one byte, 0x00; No-Operation, one byte, 0x01).
It simply consists of the Option number (0x02), an option length byte (always 4), and the actual value of the requested maximum segment size. The value used is negotiated between communication partners, details can be found in the TCP specifications.
Segment size negotiation can be useful when a host has only very little resources available for connections: Buffer space can be at a premium there. An absolute minimum MSS of 576 is recommended; less is possible, though efficiency is at risk then.

# 10. A FUNCTIONAL DESIGN FOR TCP/IP

The descriptions of the various protocols, as presented in the previous chapters, have served as a set of specifications; the official specifications of TCP and associated protocols were not available at the time.
According to these specifications a functional design has been constructed, to be expanded later, up to the implementation level eventually.

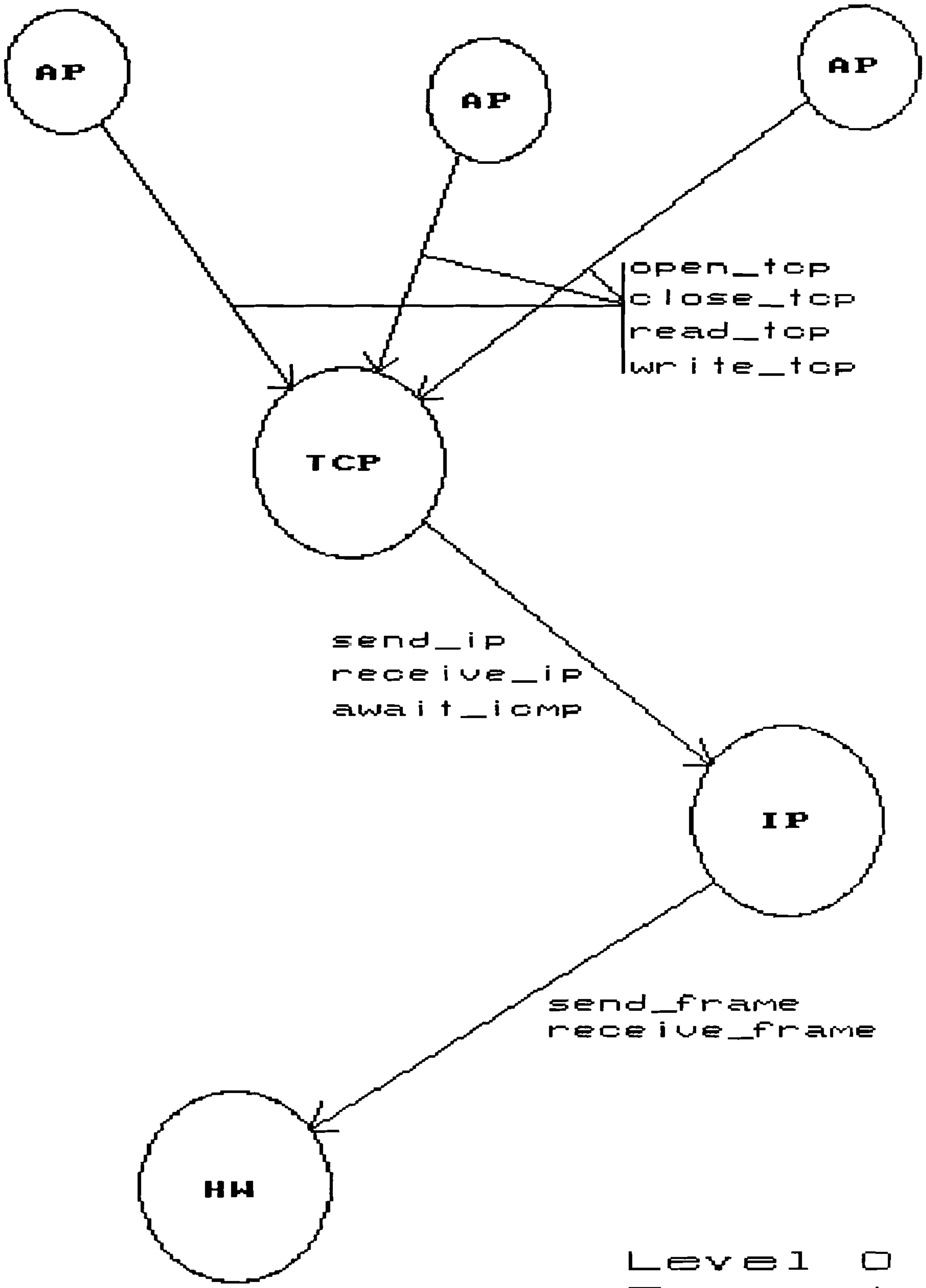
The design is presented here using the so-called 'access-graph'-method, a simple though not formal way of describing a design. Time constraints have been the main reason for not choosing a formal method.
The figures representing the design (Fig's 14 through 24) are explained in plain English here; interface definitions and functionality overviews in algorithmic form can be found in appendix B.
As to how the link can be made from this design to a Waterloo Port-specific detailed design, simplified examples are presented in appendix C.

## 10.1 Top-level design

The first level of design (figure 14) clearly shows the layered structure of the TCP/IP communication protocol suite: Every main protocol a separate functional entity, with a few simple interface functions.
The interface offered to applications (AP) using TCP is similar to the interface of a Unix pipe: A channel must be opened before it can be used, then an indeterminate number of reads/writes may occur, after which the connection should be closed.

One important difference is that the close_tcp function[4] only closes the sending-side of the connection; if the application is designed that way, it is perfectly legal to read from a closed channel.
The other interface functions simply represent send- and receive-primitives for the respective layers.

[4] This terminology does not imply implementation details. All interface entities are referred to as functions.
Regardless of how the modules actually communicate over the interfaces, all interfaces can be packaged in library functions. To the programmer using the interface a set of functions is available. Those functions are meant here.



ig. 14

*Global Design*

## 10.2 First level of expansion: Hardware module

The functional blocks in figure 14 can be expanded further, even without the need to make implementation dependent assumptions. The size of the pictures is not sufficient however to expand the whole global design in one step.

The first expanded module is the 'HW' module from figure 14 (see figure 15).
A new concept is introduced here in respect to the 'access-graph' design method. The circles indicate 'active' parts (eg. software performing some function), while the rectangles indicate 'passive' entities controlled by actions from the circles.
In figure 15 the 'HWline' module is a rectangle. This means that the functionality offered by HWline is fixed; the module cannot do anything without directions from other modules, and when directed, performs some predefined, unalterable function. In this case the Ethernet functionality is fixed, and contained in HWline.
The same goes for 'Sem', a semaphore with two interface functions corresponding to a P-operation and a V-operation as indicated.

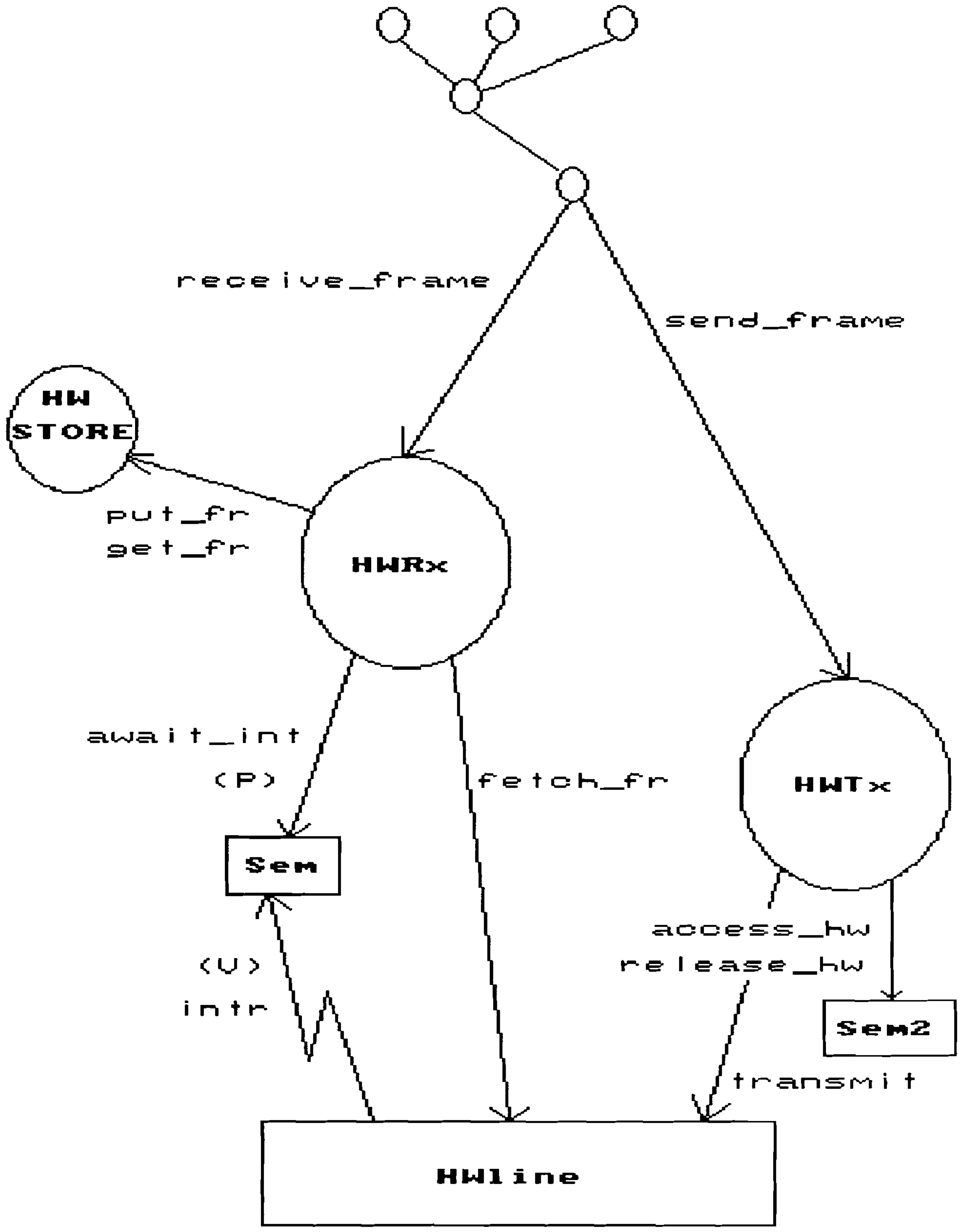
The small circles at the top of the picture are just meant to indicate the relative position of this expanded picture in the picture representing the previous level.

A functional design can be expanded up to the level where implementation details become important; expanding further violates the purely functional aspect of the design.
This is the case with the 'HWstore' module: The functionality of the two interface functions can be described in a more or less algorithmic form (see appendix B), but expansion of the module itself is only feasible if more details are taken into account regarding the nature of the storage method that will be used.
So this module is the end of a line.

The 'HWTx' transmit module is also not expanded further, but for different reasons: HWTx can be expressed in a (reentrant) function

(This time a real function, not just a library-version of something different), so there is no need to expand it further.

The last module, 'HWRx', has to be expanded further.
The reason for that is not simple, because the principal functionality can be expressed in algorithmic form just as with HWTx. The difference, however, is a protocol-discrimination issue: The layer 3 protocol calling a real function 'receive_frame' cannot know beforehand that the first frame arriving over the hardware will actually be directed to it: An ARP frame may arrive if IP is waiting for one. This separation of protocols is hard to achieve when HWRx is not expanded into smaller modules.



ig. 15

Level 1
Form 1

3

HW: Hardware Module

Analog lines of reasoning have determined whether or not other modules needed expansion; I will not elaborate on these issues as widely as with this example, an expansion will be presented if it seems logical or inevitable.

### 10.2.1 The HWRx receiver module

In figure 16 the 'HWRx' module from figure 15 has been expanded. The functionality will be presented here.

'receive_fr' is a process (not a procedure or function), repeating a steady cycle:
- Wait for an interrupt from the hardware; this indicates that a frame has been received over the communication line. The HWline module only has limited buffer space, so:
- Transfer the frame from HWline to main memory, using 'fetch_fr'.
- Immediately transfer the frame to the 'HWstore' module using 'put_fr'; this function (real) stores the frame temporarily.
- Signal the semaphore corresponding to the value of the Ethernet type-field; this takes care of notifying the consumer of the frame. The 'Sem' module (note the dots) is one of the possibly many semaphores, one for every layer 3 protocol expecting frames.

After this cycle the process starts again.

On the other side the 'take_fr' module should provide the functionality required by 'receive_frame'. It can be a reentrant procedure if and only if the function is called only once at the same time per layer 3 protocol: If two processes call 'receive_frame' from the IP protocol, both start waiting for the same semaphore; the decision for whom the frame will be is a hard one.
In this case the precondition holds true (see figures 17, 18, 19), so 'take_fr' can be (is) a reentrant procedure.

A last remark about the 'get_fr' functionality: In order not to nullify the effect of having multiple semaphores, 'get_fr' should only get messages from storage with matching type-fields/type-identifications. (See appendix B.)

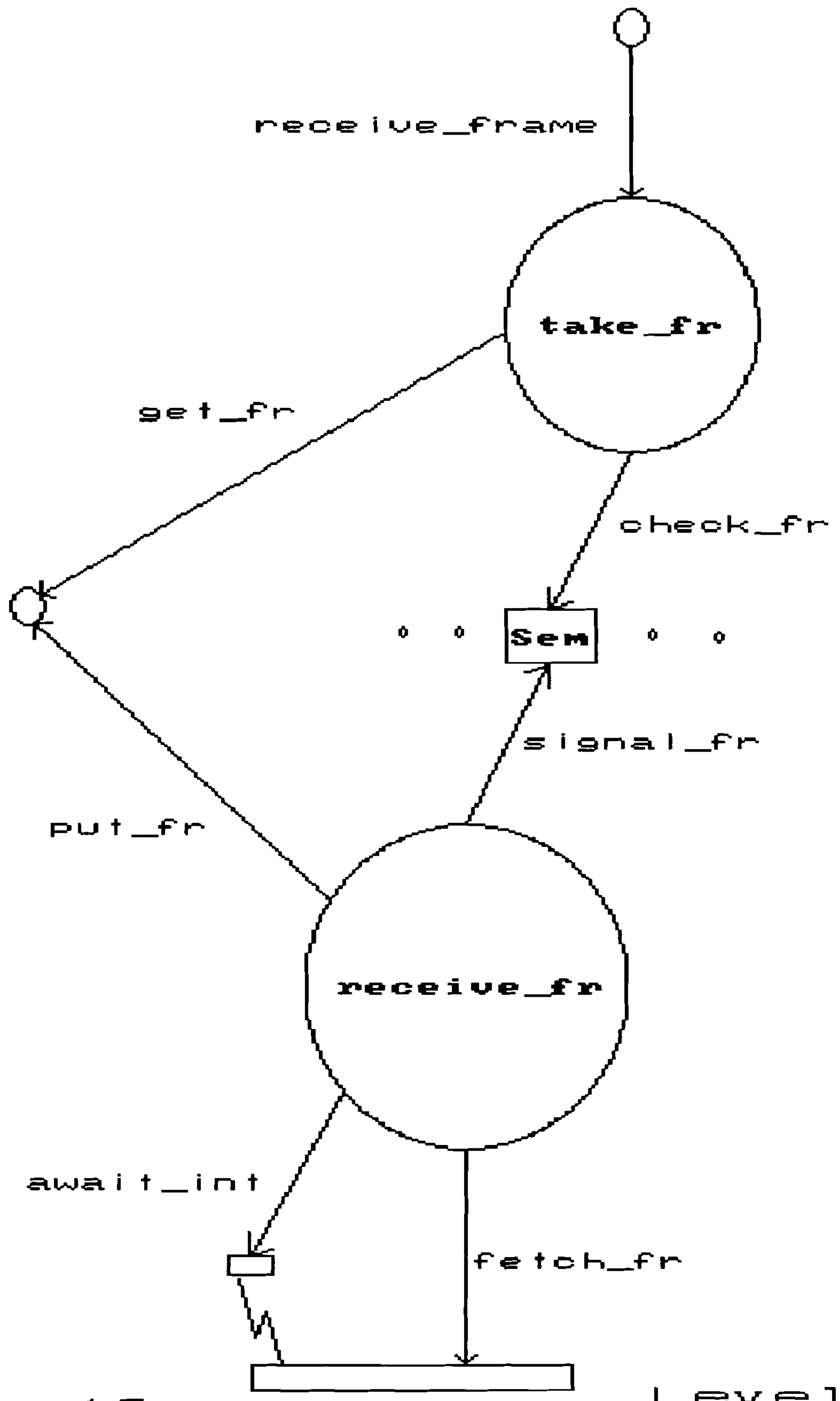


Fig. 16

Level 2
Form 1

50 *HW Rx: HW Receiver*

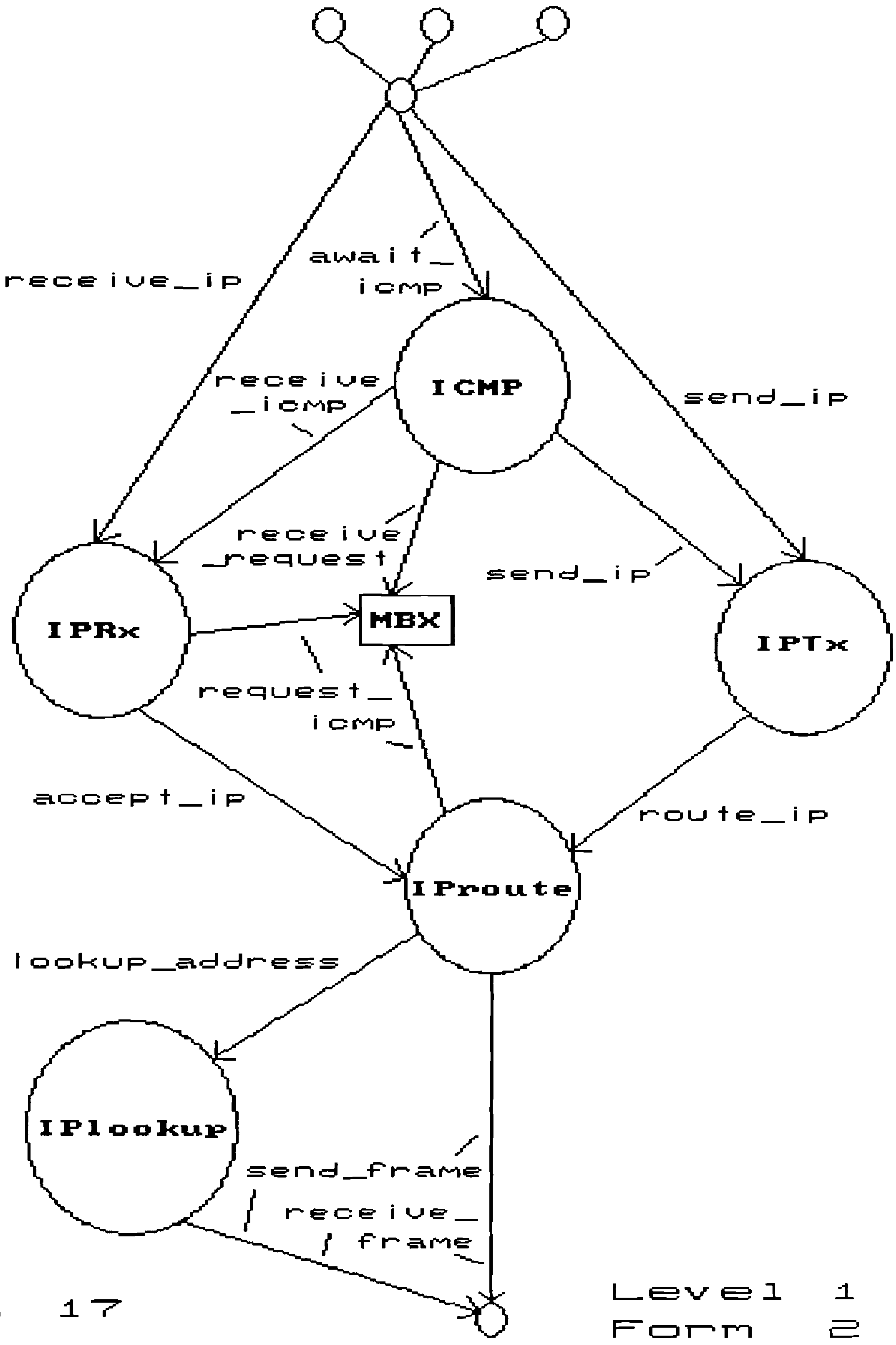
## 10.3 First level of expansion: IP module

The IP module seems somewhat complex. It is not really.
One new concept is introduced, a mailbox ('MBX' module). A mailbox is a sort of synchronizing mechanism: Reading an empty mailbox blocks the reader, as does writing to a full one. The mailbox stores messages passed between writer and reader, so that a writer does not block usually, nor does a reader.
A mailbox can be used to synchronize reading and writing messages. Messages are the unit of information contained in a mailbox.

The functionality of the 'IP' module can be described as follows:
* 'IProute' determines the destination of a packet by looking at the destination IP address, and uses 'lookup_address' to determine the hardware address to be used in calling 'send_frame'.
* 'IPlookup' implements a lookup-table, as well as the ARP protocol in the cases where the table does not contain the information.
* The 'MBX' module is used to mediate requests for reporting an error (through ICMP). The 'ICMP' module monitors this mailbox and sends ICMP messages using IP where necessary.
* The 'IPRx' module is the local receiver of IP packets. ('IProute' accepts all packets, rerouting packets for other hosts, and passing on the packets for its own host to 'IPRx'.)
* The 'ICMP' module performs two functions: It accepts incoming ICMP messages indicating errors reported on by other hosts or routers; and it packages ICMP messages to report errors found by its own host, and transmits them using IP.

Aside from the passive module 'MBX', the 'IPTx' module is the only one not expanded in the pictures that follow. It can be implemented as a reentrant procedure, compiling an IP header and transferring the packet to IProute.

Details will surface in the discussion of the various expanded modules.



ig. 17

Level 1
Form 2

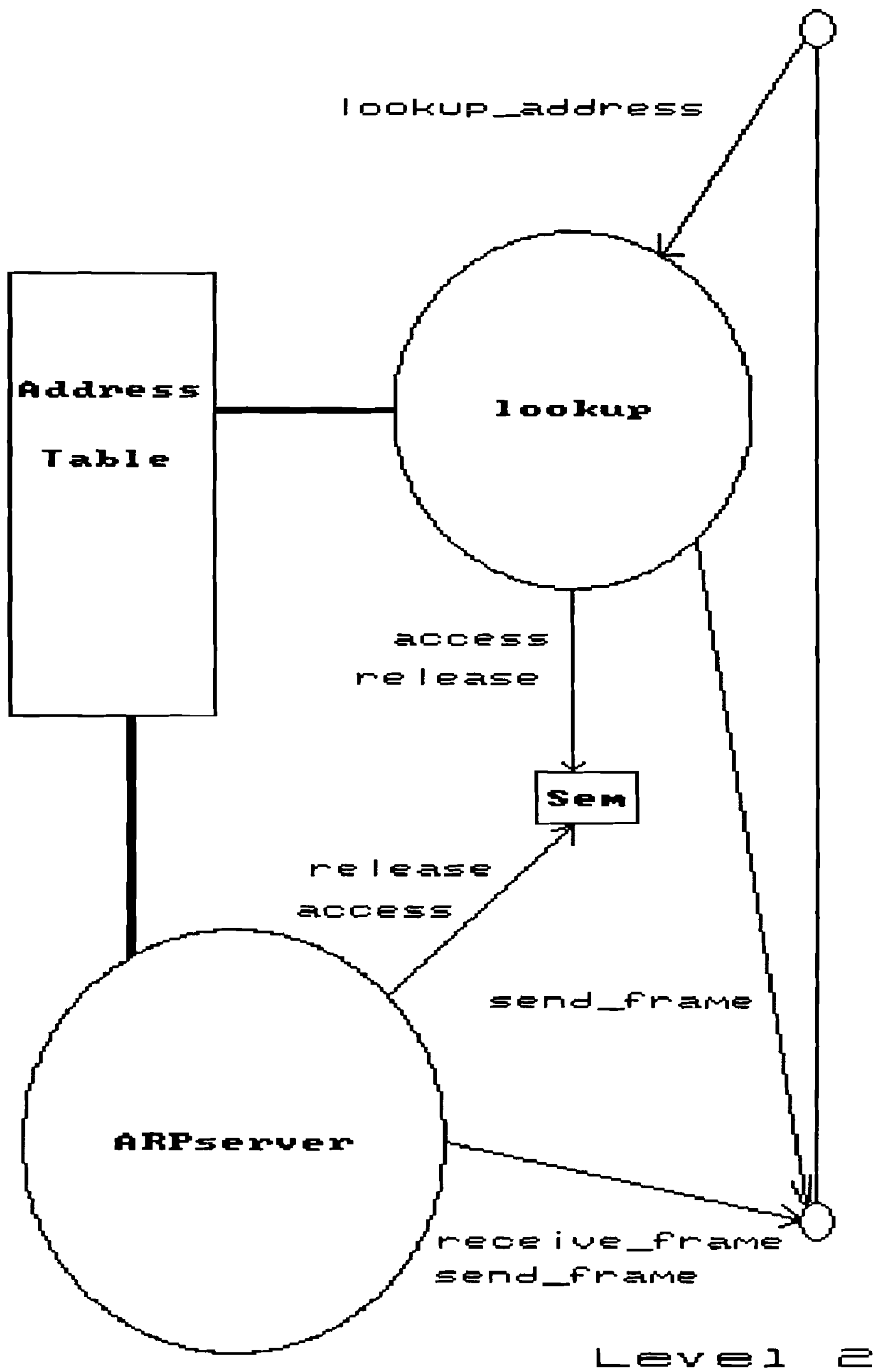
*IP: Internet Protocol*

### 10.3.1 The IPlookup module

The 'IPlookup' module is relatively simple:

- The 'ARPserver' waits for incoming ARP messages (Thus a process). If an ARP request is encountered, the server replies using ARP if our host is the target of the broadcast message. If an ARP reply arrives, the data contained in the message is entered in the 'Address-Table', ensuring mutual exclusion through using the semaphore 'Sem'.
- The 'lookup' module uses the 'Address-Table' to find the hardware address (Ethernet) of a target, given an IP address. If the address cannot be found, it issues an ARP request with 'send_frame'. (The reply will be picked up by 'ARPserver', making the address available when the next request for it is made.)

'ARPserver' must be a process (it waits for ...), while 'lookup' can be a real function (not even necessarily reentrant).



ig. 18

*IPlookup: ARP*

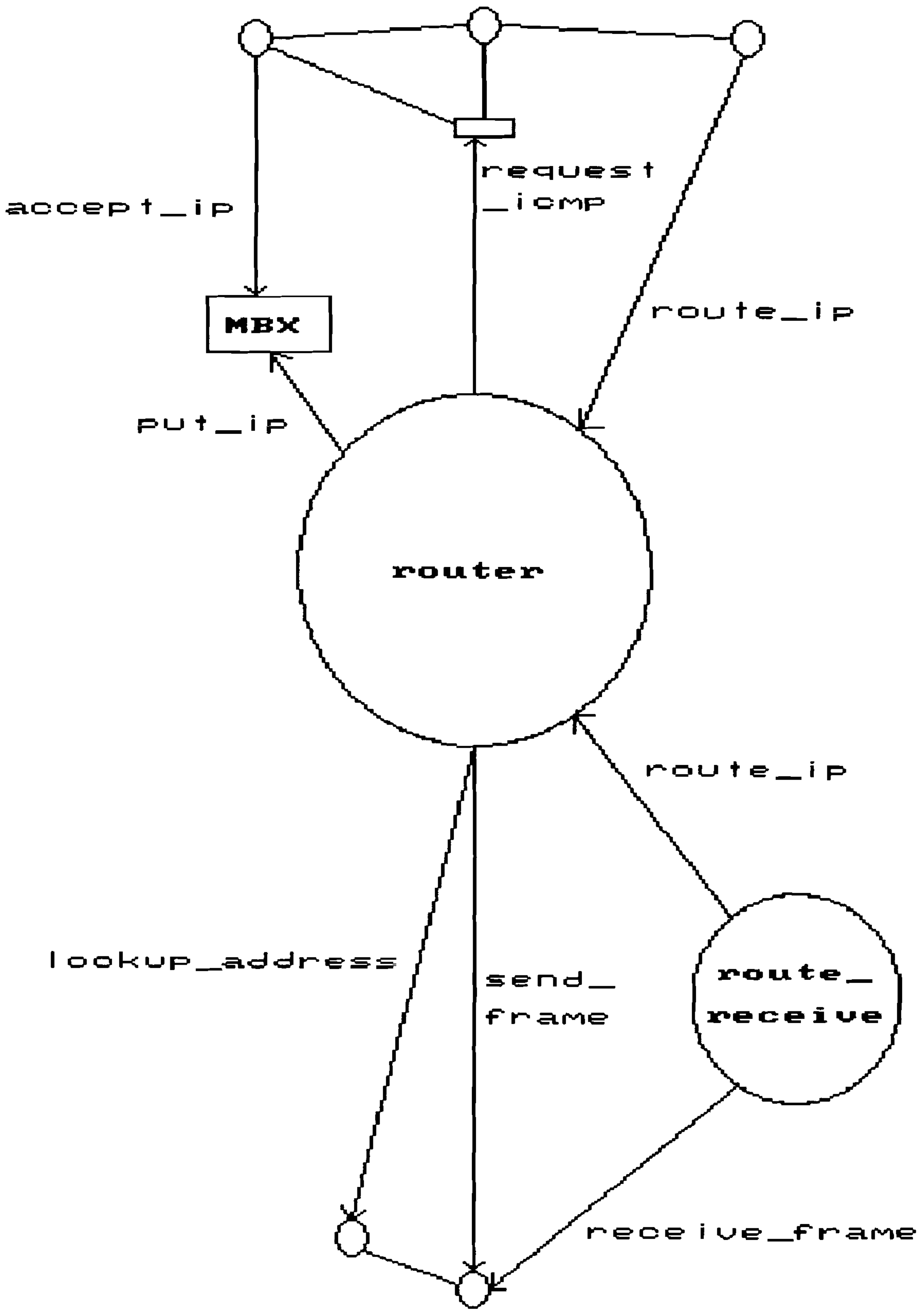
### 10.3.2 IProute: Router functionality

The 'route_receiver' module is a process waiting for an IP packet to arrive. It performs no other function than to forward the packet to the 'router' module.
The 'router' module can be a reentrant procedure. It should do the following when a packet comes in:

- Look if the packet is intended for this host; if so, put it in the mailbox 'MBX' using 'put_ip' and quit.
- Otherwise, use 'lookup_address'. If an address is returned, use it to send the packet in a frame; if not, discard the packet (retries and ARP will take care of it).
- Quit.

If an incoming packet triggers an error, use 'request_icmp' to report the error to the originating host using ICMP.

'route_ip' can be called for all packets, packets from layer 4 protocols, as well as packets arriving over the (a) network.
The functionality of 'router' is somewhat more extensive than indicated here: Only the functionality relevant for explaining the pictures is mentioned; for more details, see appendix B.



ig. 19

Level 2
Form 3

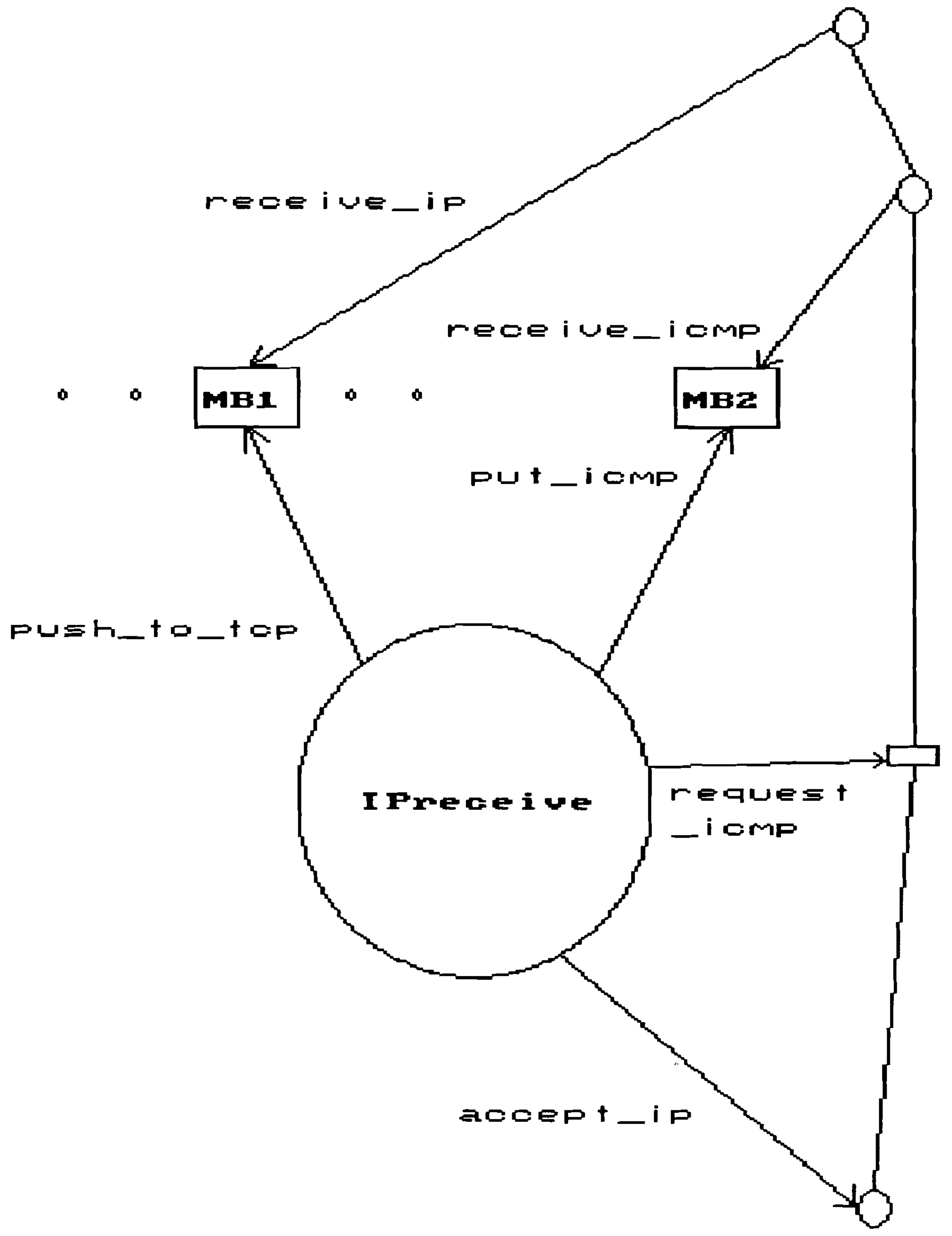
56

IProute

### 10.3.3 IP receiver

The 'IPreceive' module performs protocol separation on the network level. It is a process accepting all IP packets intended for this host, trying to determine what level four protocol should be triggered, reporting a non-existent protocol using 'request_icmp', and finally placing the packet in one of the two mailboxes, depending on the IP header's PROTOCOL field.
This interface is not general enough to serve other layer four protocols, but a series of mailboxes, one for every protocol, is easy to imagine; the underlying principles do not differ at all.

In order to achieve separation of (TCP-) ports on a higher level it may be necessary to create multiple mailboxes for TCP (one for ICMP). Every TCP-mailbox then corresponds to a TCP port.
The strict functional separation between IP and TCP would be slightly violated, but not unacceptably so.
This method is indicated by the dots around the 'MB1' module in figure 20.



ig. 20

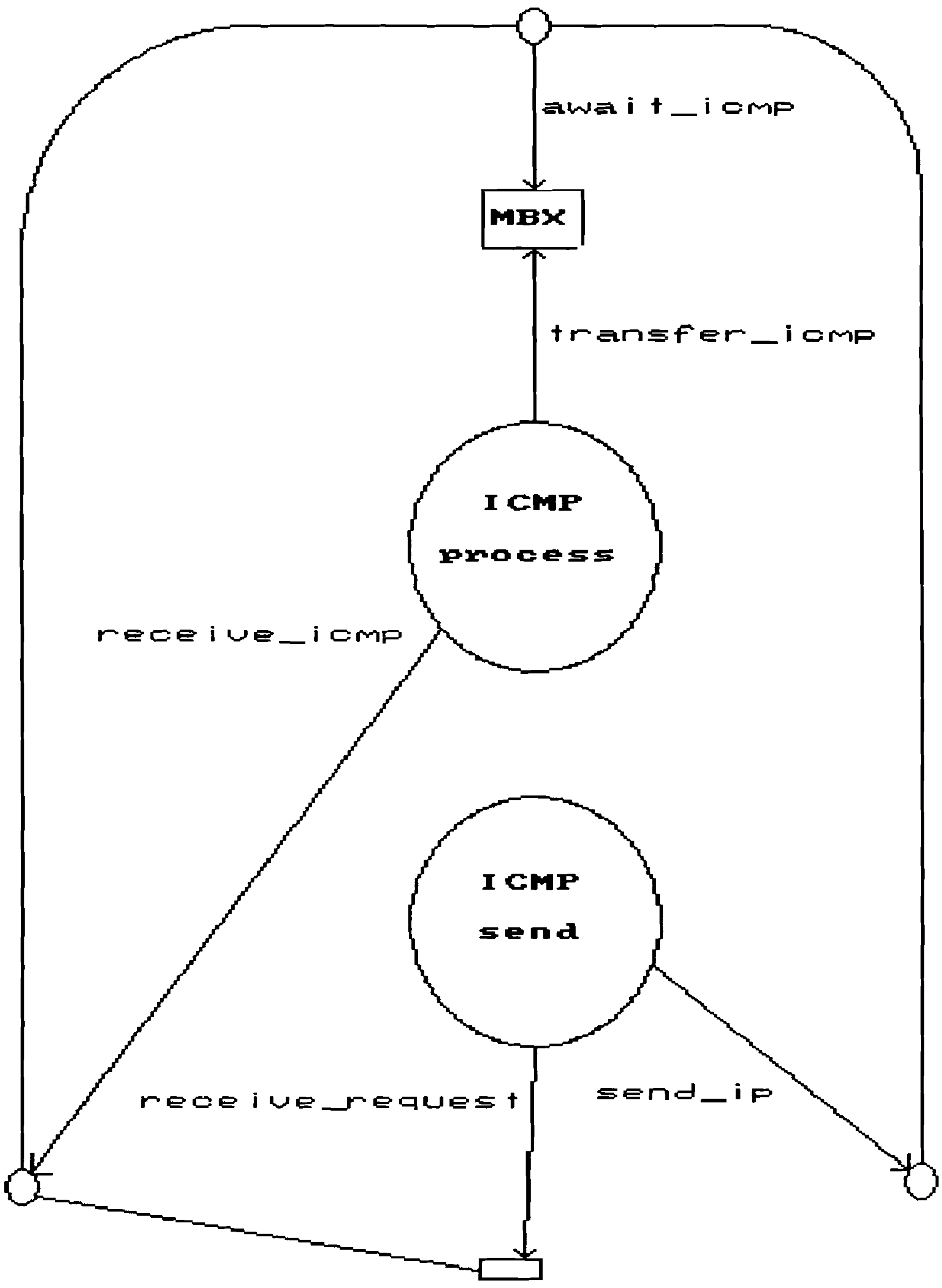
Level 2
Form 4

58 *IPRx: IP receive*

### 10.3.4 ICMP error reporting

The expanded 'ICMP' module effectively consists of two separate submodules, instead of linked in some way.
One ('ICMPprocess') receives the incoming ICMP messages, processes them or passes them up to TCP (should be generalized to include UDP, NVP, ...) if the message is intended for a specific protocol or port-connection.
It should be implemented as a process (It has to wait for a message).

The other, 'ICMPsend', acts on behalf of other parts of the Internet Protocol. It monitors a mailbox, and sends ICMP messages to hosts responsible for a detected error, using send_ip.



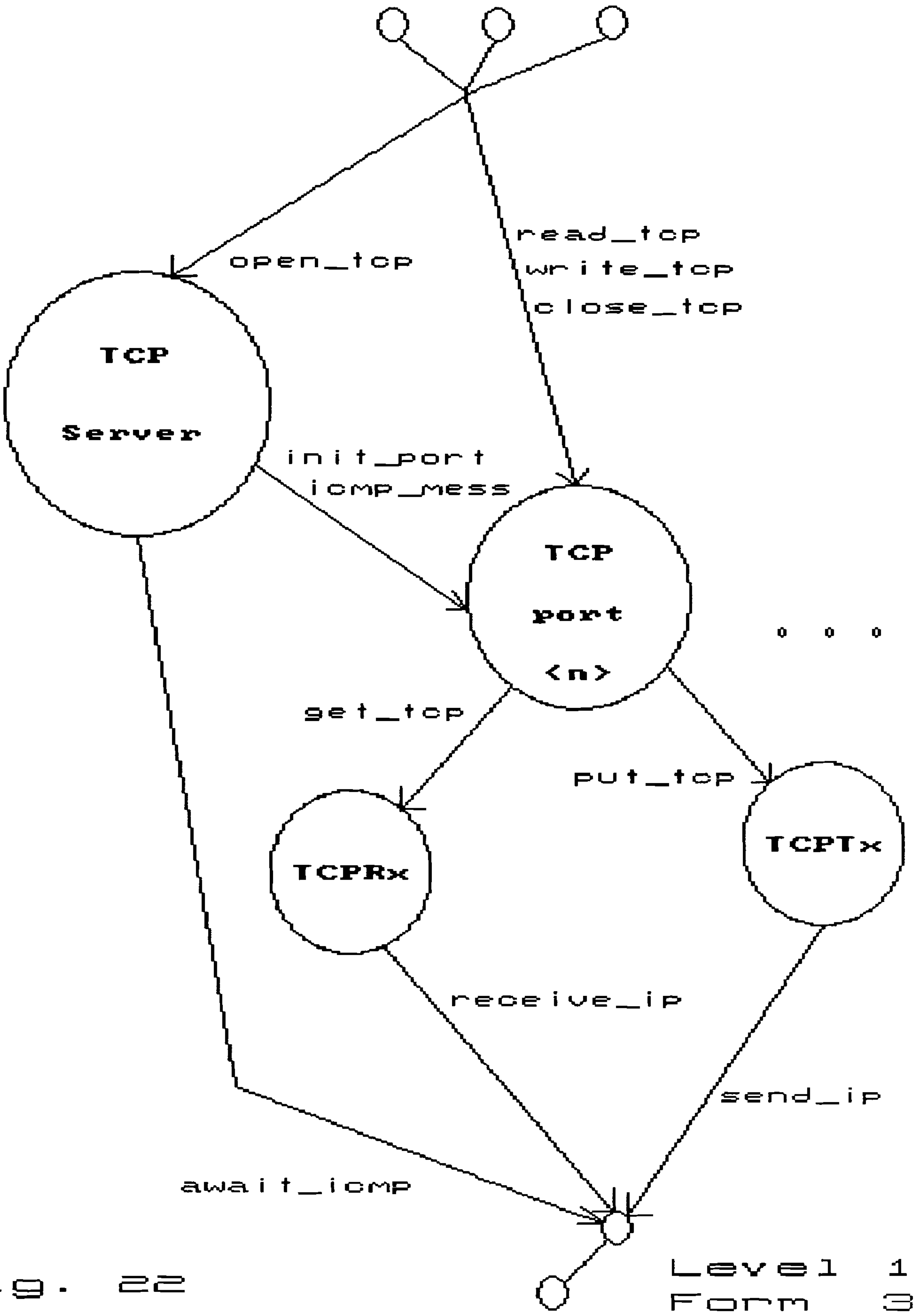
Level 2
Form 5

ig. 21

70 *ICMP: Control*

## 10.4 Transmission Control Protocol

Expansion of the TCP module in the global design (figure 14) is shown in figure 22.
The 'TCPserver' module organizes TCP functionality. For every connection (all using a different TCP port) it creates a new process, the 'TCPport' module, and initializes it. A pointer to this process is then passed back to the process calling 'open_tcp'. This pointer is then used to issue further requests (read, write, close).
The 'TCPRx' and 'TCPTx' modules perform simple transmission and reception functions; procedures are sufficient to implement them.
The other two modules are expanded further in the next sections.



ig. 22

2

*TCP: Transmission Control Protocol*

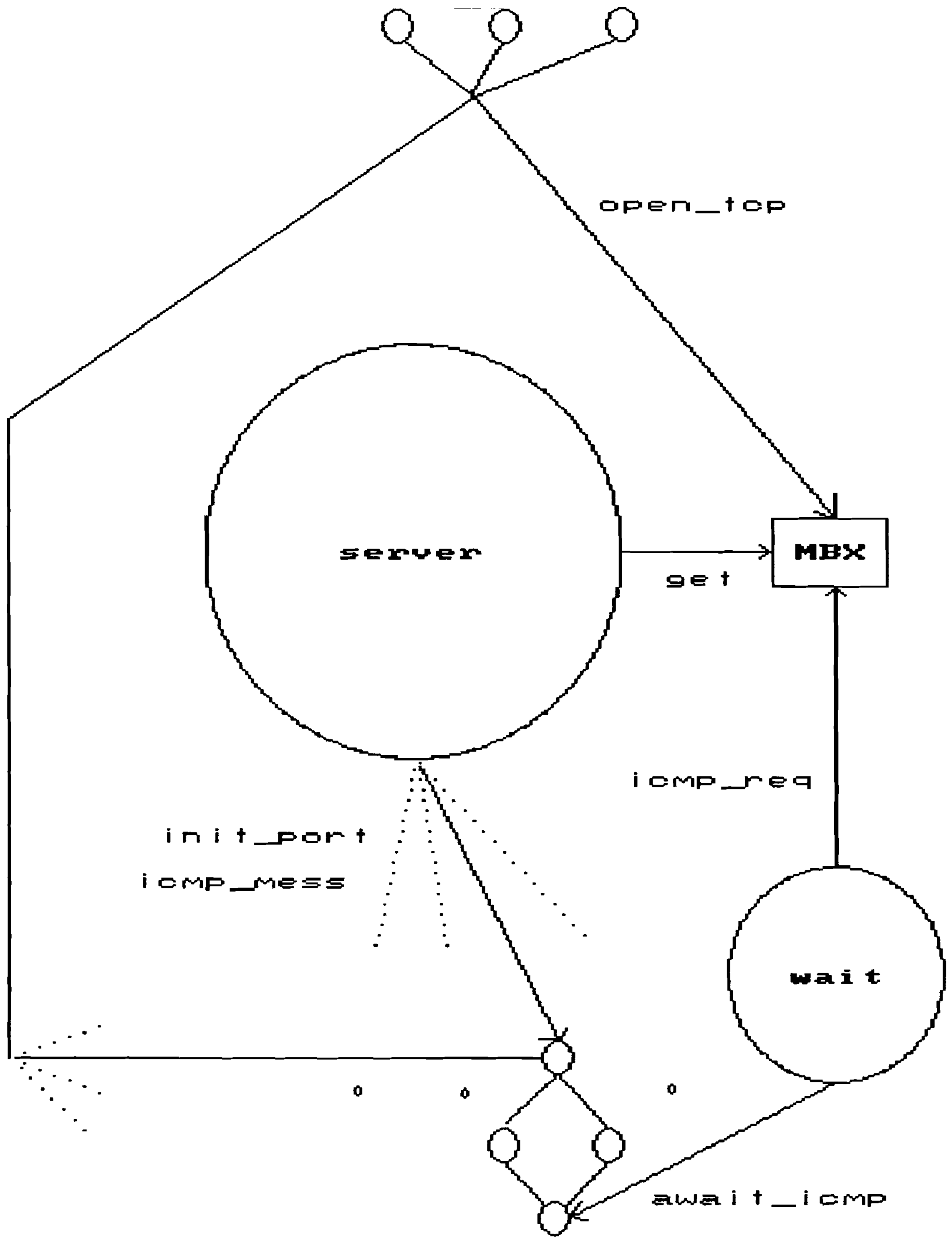
### 10.4.1 The TCPserver module

The 'TCPserver' module is expanded in figure 23.

The 'server' module is a true server: It accepts requests for service by monitoring the mailbox 'MBX', then processes them and continues.
The kinds of requests fall into two categories:
- Requests to pass on ICMP messages to a specific TCPport process.
- Requests to open a new connection.

These are processed using calls to 'init_port' and 'icmp_mess'.

The 'wait' module is also a process. It waits for ICMP messages coming in that the 'ICMP' module cannot handle by itself.
As soon as such a message comes in, it is passed to 'server'.



ig. 23

74

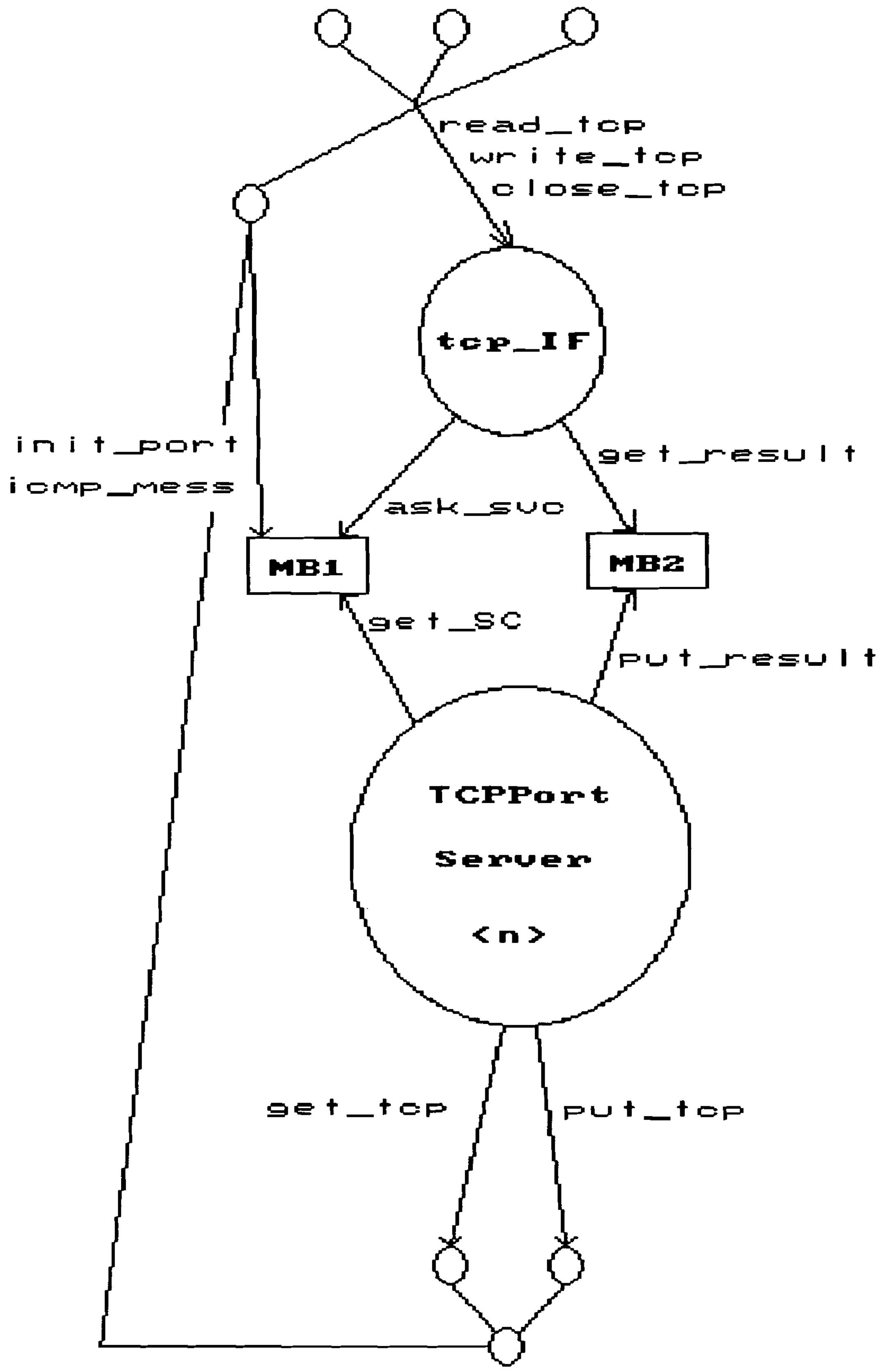
TCPserver

### 10.4.2 The TCPport module

The 'TCPport server' module in figure 24 is once again a true server process:

- Use 'get_SC' (Service Code) to get a request
- Process it using 'get_tcp' and 'put_tcp'
- Then quit if the request came from 'TCPserver', otherwise return the result in 'MB2' using 'put_result'

The operations read, write, and close, belonging to the module 'tcp_IF' (InterFace) can be implemented as true functions, issuing the request and waiting for the result.
The method of waiting for the result is important in this part of the design: It means that the function 'read_tcp' actually blocks until the response from 'TCPport server' is received.
And 'tcp_read' blocking means that 'get_tcp' is allowed to block, because no other 'read_tcp', 'write_tcp' or 'close_tcp' calls can occur in that time.
If that is not appreciated, the design should be adapted correspondingly.



Level 2
Form 7

ig. 24

76

TCP port

## 10.5 Final remarks

The design presented here is explained in algorithmic form with much more detail in appendix B.
If some assumptions made in this design are not valid for the target environment, the design should be adapted to suit different needs.
An example is the blocking 'read_tcp': See figure 24 and chapter 10.4.2.
Finally, in appendix C an example is given suggesting ways how to implement certain constructs (a semaphore and a mailbox) in the Waterloo Port development environment.

# 11. CONCLUSIONS

With the design presented in this report a first step in the direction of a TCP/IP implementation is taken.
The Waterloo Port angle is a little undervalued, as it should be when a functional design is being discussed. Appendix C is intended to hint at the environment where TCP/IP should eventually become operational, and to offer a first impression concerning the translation of this functional design into a detailed design specifically for Waterloo Port.
When another environment is chosen for a target, this design could be used as a basis. The assumptions should be examined, and if invalid the design may need some modifications. Finally the synchronization primitives, semaphore and mailbox, may not be present. It is then necessary to find some mapping to the mechanisms available in the new environment, as is done with Waterloo Port (Appendix C).

It might be a good idea, depending on the development time available to implement this design, to translate this design first to a similar design using a formal design method. And if the time constraints are strict I would strongly advise to do so.
The access-graph method can represent a good design in an easy to understand way, but it cannot give any indication whether a design is good or bad; this is similar to structured programming in assembly language: It is possible but requires a lot of discipline.

## ACKNOWLEDGEMENTS

Without the help of many people I wouldn't be where I am now. Thanks to my parents, for insisting; to Peter Dubbelman and Piet van der Putten, for patience; to Rob Oostveen, for giving me a chance; to Carla, Josje, Cor, Frank, Wim, Onno, Gerard, Jean-Pierre, Ferdinand, Bart, and Willem-Barent, for accepting me as one of them; to Hans van Keulen (and Berry), for friendship; to Leon Stok, for providing information when no information was available; to professor Stevens, for having had more influence than he may realize.

And a very special word of thanks to Leo van Bokhoven; he really works wonders as a coach:
Leo, thanks!

Johan Bezem
Eindhoven, August 14th, 1988.

# Appendix A TERMINOLOGY ISSUES

The confusion concerning network interconnection options is certainly not alleviated by a consistent vocabulary. To illustrate this, and to provide a framework for this report, a separate chapter may be useful.

## 1.1 Definitions

The word "network" has no rigid definition to help determine what is still a network, and what should be called an "internetwork" (two or more interconnected networks). The boundaries are not trivial.
To start bottom up:

A basic network segment is a collection of hosts directly connected to one another by physical wiring only.

This means that a direct link is available for all possible communication between hosts within the basic network segment.
A single configuration of only one basic network segment is certainly a network, and not an internetwork.
The difficulties arise when a single basic network segment is no longer sufficient in terms of geographical span, number of hosts or network capacity: Interconnecting two or more basic network segments can be done in various ways; a natural way to group the interconnections is the highest level of the OSI Reference Model they cover.

### 1.1.1 Repeaters

Repeaters are the simplest way to interconnect two basic network segments. They operate at the OSI physical level, and can be seen as two-way signal boosters. They do not need any local intelligence, and are pure hardware devices.
Every frame presented on one side of the repeater is transmitted onto the other side, no partitioning other than electrical is created between the two connected segments. It simply extends the

range over which the signals can be transmitted without too much deterioration of signal quality.
A repeater may connect more than two basic network segments.

Other common words found for repeaters are "hub" or "active hub"; the expression "hub" does not always indicate a repeater, it can also be an even simpler impedance-matching device. Two or more basic network segments connected with these impedance-matching "passive hubs" are still considered one basic network segment.
Two or more basic network segments connected with repeaters only are considered one "network segment".
A single configuration of basic network segments interconnected with repeaters only is definitely not an internetwork.

### 1.1.2 Bridges

Bridges operate at the data link level (Media Access Control) of the OSI Reference Model. They not only boost the signal power, but they also provide partitioning of the network.
A bridge should forward all packets destined for all network segments connected to another side of the bridge, directly or indirectly. All other packets should be discarded.
This required functionality involves some sort of routing. One of the big differences is that a bridge should perform its forwarding as transparent as possible, so that the host protocols do not need to know bridges are used, not even the host data link protocols.
Routers are discussed in the next section.

The transparency requirement leads to other demands:
- A bridge should monitor all traffic on all its lines to determine what frames need to be forwarded on another link.
- A bridge should, if possible, operate at the same speed as the network segments it connects.
- A bridge should not introduce extra duplicates of frames being delivered (in case of two bridges parallel !).

The first two requirements indicate the performance a bridge should ideally offer. The latter requirement indicates clearly the need for some routing facilities and a bridge-management protocol.

More on these subjects can be found in [Bac 1988], discussing the so called 'spanning tree' algorithm, [Dix 1988] and [Ham 1988] on a non-transparent source routing mechanism, and [Zha 1988] comparing the two approaches.

The functionality of a bridge is easily extended to include some features belonging to the network layer; this has caused a lot of discussions recently. The dividing line between the layers two and three of the reference model is not very clear.
Sometimes a bridge may be called a level two gateway or a data link gateway, bridge is a commonly accepted expression, however.
In this report a set of network segments connected with bridges will be called a "bridged network segment." It is still considered one network.

### 1.1.3 Routers

Routers connect bridged network segments to other bridged network segments; those segments may have bridges internally, but even basic network segments fall into the bridged network segment category.
Routers are addressed directly, in contrast to bridges, and execute the network layer protocol. Their function is known to (some) hosts in the networks they connect to, so that hosts can send their packets for other networks through these routers.
Routers have to process only the packets that need forwarding thanks to the direct addressability of a router, and can thus handle a specific load with less computing power.
They need more storage space than bridges, because they have to accomodate for larger differences in transmission speeds between the connected networks (eg. 10 Mbit/s Ethernet vs. 56 kbit/s leased lines).
The terminology used for routers is the most confusing: A gateway (TCP/IP terminology), an intermediate system (OSI terminology), a network relay, and an interface message processor (IMP, dated ARPAnet terminology) are expressions for the router. Here gateway is the most confusing because the word gateway is generally used

for connections on the higher protocol levels, levels four to seven.
The expression gateway is also used to indicate a simple store-and-forward system, like in "mail-gateway."

[Bos 1988] discusses the problems of routers (and bridges) in large LANs, with general examples from experience with TCP/IP networks. It may be very enlightening.
[Sei 1988] compares the functionality and requirements for bridges and routers, with examples from TCP/IP, DECnet (DNA), and XNS, the Xerox Network System.
[Per 1988] clearly draws the picture and suggests various approaches, while [Pad 1988] offers a refreshing look at the subject.

Various bridged network segments interconnected by routers are called an internetwork, or internet. However, when someone talks about an Internet (capitalized) while discussing TCP/IP or ARPAnet, she/he will probably mean the collection of networks connected to the ARPAnet using TCP/IP.

### 1.1.4 Gateways

Gateways are hosts that perform a special task in connecting two or more networks (if the expression is not used to indicate a router; see Appendix A, 1.1.3). This means that gateways operate on levels four through seven of the OSI reference model.
They can connect to two completely different networks, offering protocol conversion when necessary (eg. IBM's SNA to OSI protocols). This type of connection is not very clearly described, and therefore it can mean various things.
A common expression is, for instance, a mail-gateway, usually indicating a host capable of sending and receiving messages on behalf of its users, and sometimes also forwarding on behalf of other mail-servers.

## 1.2 Terminology used in this report

In this report the following expressions are used without further explanation:

Repeater
Bridge
Router

A gateway is a host providing internetwork connections on level four or higher (OSI reference model).
A TCP/IP-gateway is a router, as is an IP-router.
Deviations from these definitions will be clearly indicated.
Be aware, however, that in TCP/IP terminology a router is usually called a gateway, most definitely in the literature too.

# Appendix B ALGORITHMIC DESCRIPTION OF THE FUNCTIONAL DESIGN

## 2.1 Index to functions, procedures and processes

In this subsection an overview is presented of the functions, procedures and processes of which an algorithmic description is included in this section.

## 2.2 Application - TCP interface

```
function open_tcp(
  local_socket     : socket,      { IP address plus port no. }
  remote_socket    : socket,      { Destination }
  active           : boolean,     { Active or listening }
  window           : integer,     { Window size, send & receive, in
                                    bytes }
  type_of_service : byte          { Bit values }
                 ): tcp_handle;  { For future reference }

Begin {open_tcp}

    { Ask TCPserver to open connection }
  put_in_mailbox(OPEN_TCP, local_socket, remote_socket, active,
    window, type_of_service);
    { Wait for it to do it }
  delay(3000);        { milliseconds }
    { Get process ID of TCPport process, to communicate with }
  tmp_handle.pid := ask_global_name_server( TCP_PORT, local_socket,
    remote_socket);
    { Check if I am really connected, by sending empty message }
  result := write_tcp(tmp_handle, NULLPOINTER, 0, FALSE, 0);
    { If all OK return process ID in handle, else NULL }
  if result = OK then open_tcp := tmp_handle
  else open_tcp := NULLHANDLE;

End; {open_tcp}
```

---

```
function close_tcp(
  circuit_handle : tcp_handle   { Returned by first open_tcp() }
                ): integer;     { Error code }

Begin {close_tcp}

    { Submit request for closing connection (svc : service) }
  ask_svc(CLOSE_TCP, circuit_handle, NULLPOINTER, 0);    { One sided
                                                           close
                                                           only }
    { Wait for the result and return the code }
  close_tcp := get_result(circuit_handle, NULLPOINTER, 0);

End; {close_tcp}
```

---

```
function read_tcp(
  circuit_handle : tcp_handle,  { Returned by first open_tcp() }
  buffer_pointer : pointer,     { Storage space }
  buffer_length  : integer      { Length of available storage space
                                  }
                 ): integer;    { Error code }

Begin {read_tcp}

    { Submit request for a read }
  ask_svc(TCP_READ, circuit_handle, NULLPOINTER, buffer_length);
    { Wait for the result and copy the data }
  read_tcp := get_result(circuit_handle, buffer_pointer,
    buffer_length);

End; {read_tcp}
```

---

```
function write_tcp(
  circuit_handle : tcp_handle,  { Returned by first open_tcp() }
  buffer_pointer : pointer,     { Data buffer }
  buffer_length  : integer,     { No. of bytes }
  push_flag      : boolean,     { Data needs be PUSHed }
  urgent_data    : integer      { First <n> bytes are urgent }
                  ): integer;   { Error code }

Begin {write_tcp}

    { Submit write request }
  ask_svc(TCP_WRITE, circuit_handle, buffer_pointer,
    buffer_length);
    { Indicate urgent data }
  ask_svc(URGENT, circuit_handle, NULLPOINTER, urgent_data);
    { Indicate PSH or not }
  if push_flag then ask_svc(PUSH, circuit_handle, NULLPOINTER, 0)
  else ask_svc(NOPUSH, circuit_handle, NULLPOINTER, 0);

    { Then wait for the result and return it }
  result := get_result(circuit_handle, NULLPOINTER, no_of_bytes);
    { and check the transmitted number of bytes }
  if no_of_bytes = buffer_length then write_tcp := result
  else write_tcp := result || NOT_ALL;

End; {write_tcp}
```

---

---

## 2.3 TCP - IP interface

```
procedure send_ip(
  source                 : ip_address,   { (one   of)   our   own
                                           address(es) }
  destination            : ip_address,   { Target IP address }
  protocol_id            : byte,         { ID-byte from transmitting
                                           protocol }
  type_of_service        : byte,         { Bit-field indicating the
                                           services requested }
  time_to_live           : integer,      { Max. no. of router hops }
  fragmentation_allowed  : boolean,      { Router      fragmentation
                                           allowed }
  data_pointer           : pointer,      { Where to find it ... }
  length                 : integer       { ... and how much it is }
                   );

Begin {send_ip}

  assemble_ip_packet();
  route_ip(packet_pointer, length, destination);

End; {send_ip}
```

---

```
procedure receive_ip(
      circuit_handle   : tcp_handle,     { To determine the correct
                                           mailbox       for      this
                                           connection }

  var data_pointer     : pointer,
  var length           : integer,
  var rxbroadcast      : boolean,        { Received      packet     is
                                           hardware-broadcast }
      protocol_id      : byte,           { Caller's protocol-ID }
                    );

Begin {receive_ip}

    { Read mailbox for this connection (address, protocol, port) }
  get_message_from_my_mailbox(circuit_handle, protocol_id,
    data_pointer, length, rxbroadcast);

End; {receive_ip}
```

---

```
procedure await_icmp(
  var icmp_type    : byte,       { ICMP message type }
  var icmp_code    : byte,       { Elaboration on type }
  var icmp_pointer : pointer     { Pointer to first 12 bytes of ICMP
                                   message }
  var iph_included : boolean,    { Validates iph_pointer variable }
  var iph_pointer  : pointer,    { Points to offending IP header
                                   plus 8 bytes of IP data }
                        );

Begin {await_icmp}

    { Read the ICMP message ... }
  read_icmp_mailbox(icmp_pointer, tmp_length);
    { ... and dissect it }
  extract_variables(icmp_pointer, tmp_length, icmp_type, icmp_code,
    iph_included, iph_pointer);

End; {await_icmp}
```

---

---

## 2.4 IP - HW interface

```
procedure send_frame(
  destination  : hw_address,     { Ethernet address }
  source       : hw_address,     { Us }
  type         : word,           { Layer 3 protocol ID }
  broadcast    : boolean,        { HW broadcast message ? }
  data_pointer : pointer,
  length       : integer
                        );

Begin {send_frame}

  assemble_frame();
    { Mutual exclusion }
  access_hw();
  transmit(frame_pointer, length);
  release_hw();

End; {send_frame}
```

---

```
procedure receive_frame(
  var source         : hw_address,    { Them }
  var destination    : hw_address,    { Our interface HW address, or
                                        broadcast }
  var type           : word,          { Their    sending    level    3
                                        protocol, should match ours }
  var data_pointer : pointer,
  var length         : integer,
  var broadcast      : boolean        { Incoming broadcast message ?
                                        }
                           );

Begin {receive_frame}

  check_fr(type);
    { A frame for us has arrived; take it }
  get_fr(type, data_pointer, length);
    { Extract data items }
  extract_variables();

End; {receive_frame}
```

---

---

2.5 HWRx - HWstore interface

```
function put_fr(
  frame_pointer : pointer,
  length         : integer,
  type           : word          { Layer 3 protocol }
               ): integer;       { Error code }

Begin {put_fr}

    { First-Come-First-Served storage, but per type ! }
  result := store_per_type_fcfs(frame_pointer, length, type);
    { Report success or failure }
  if result = OK then put_fr := OK
  else put_fr := OUT_OF_MEMORY;

End; {put_fr}
```

---

```
function get_fr(
  var frame_pointer : pointer,
  var length        : integer,
  var type          : word          { Retrieve frame for protocol
                                      ID <type> }
             ): integer;            { Error code }

Begin {get_fr}

    { Get a frame with type-field equal to the given 'type' }
  result := fetch_next_frame(type, frame_pointer, length);
    { If result is OK, return frame; otherwise return INVALID }
  if result = OK then get_fr := OK
  else get_fr := INVALID;

End; {get_fr}
```

---

---

2.6 HWRx - HWline interface

```
procedure await_int(
  semaphore_id : semaphore      { HW module semaphore 'Sem' }
                  );

Begin {await_int}

    { Wait for interrupt driven semaphore }
  P_operation(semaphore_id);
    { Terminate interrupt processing directly }
  end_of_interrupt();

End; {await_int}
```

---

```
function fetch_fr(
  var frame_pointer : pointer,  { Provide storage space }
  var length        : integer,
              ): status;        { Return status }

Begin {fetch_fr}

    { Interrupt could have meant error: so check status first }
  if Rx_status = OK then retrieve_frame_from_hwline(frame_pointer,
                                 length);
    else process_status();
  fetch_fr := Rx_status;

End; {fetch_fr}
```

---

---

## 2.7 HWTx - HWline interface

```
function transmit(
  frame_pointer : pointer,      { Data to be transmitted }
  length        : integer
                 ): status;     { Did it work ? }

Begin {transmit}

    { Prepare hardware for transmission }
  setup_hardware();
    { Offer frame to hardware }
  setup_frame();
    { Make hardware transmit the frame }
  start_transmission();
    { Return the hardware status }
  transmit := Tx_status;

End; {transmit}
```

---

---

## 2.8 ICMP - IPRx interface

```
procedure receive_icmp(
  var message_pointer : integer,    { ICMP message ... }
  var length          : integer,    { ... with length }
  var source          : ip_address,     { Source of ICMP message }
  var destination     : ip_address,     { And destination (We) }
  var type_of_service : byte    { TOS can be useful in processing }
        );

Begin {receive_icmp}

    { Get ICMP message, maximally one IP packet: no fragments }
  read_icmp_mailbox(message_pointer, length);
    { Extract addresses and type-of-service, check protocol ID }
  if protocol_field(message_pointer, length) <> ICMP_PROT then
  begin
    message_pointer := NULLPOINTER;     { Indicates error }
    length := 0;
  end
  else extract(source, destination, type_of_service,
    message_pointer, length);

End; {receive_icmp}
```

---

```
procedure request_icmp(
  icmp_type    : byte,       { ICMP message type }
  icmp_code    : byte,       { Elaboration on type }
  connection   : tcp_conn_structure,    { Contains      connection
                                          specific data }
  iph_included : boolean,    { Validates iph_pointer variable }
  iph_pointer  : pointer,    { Points to offending IP header plus 8
                               bytes of IP data }
                        );

Begin {request_icmp}

    { Pass request on to ICMP }
  place_available_data_in_mailbox(icmp_type, icmp_code, connection,
    iph_included, iph_pointer);

End; {request_icmp}
```

---

```
procedure receive_request(
  var icmp_type    : byte,       { ICMP message type }
  var icmp_code    : byte,       { Elaboration on type }
  var connection   : tcp_conn_structure,    { Contains   connection
                                              specific data }
  var iph_included : boolean,    { Validates iph_pointer variable }
  var iph_pointer  : pointer,    { Points  to  offending  IP  header
                                   plus 8 bytes of IP data }
                            );

Begin {receive_request}

  get_available_data_from_mailbox(icmp_type, icmp_code, connection,
    iph_included, iph_pointer);

End; {receive_request}
```

---

---

## 2.9 IPRx - IProute interface

```
procedure accept_ip(
  var packet_pointer : pointer,     { IP packet intended for us }
  var length         : integer      { ... }
                      );

Begin {accept_ip}

  read_mailbox_from_iprouter(packet_pointer, length);

End; {accept_ip}
```

---

---

## 2.10 IPTx - IProute interface

```
function route_ip(
  packet_pointer : pointer,
  length         : integer,
  target_address : ip_address   { To make routing possible }
                 ): integer;

Begin {route_ip}

    { Decide if packet is for us }
  if target_address = OUR_IP_ADDRESS then
  begin
    put_ip(packet_pointer, length);
    route_ip := OK;
  end
  else
  begin
    HW_addr := lookup_address(target_address);
    if HW_addr <> NULL_HW_ADDR then
    begin
      if HW_addr = DELAYED then route_ip := OK;
    { Let TCP (...) decide that message has not arrived properly }
      else
      begin
        if length <= MAXIMUM_PACKET_LENGTH then
          send_frame(HW_addr, OUR_HW_ADDR, IP_TYPE, FALSE,
            packet_pointer, length)
        else
        begin
    { Take care of fragmentation }
          loosen_header_from_data();
          total_length := 0;
          while total_length < length do
          begin
            assemble_fragment();
            set_fragment_flag();
            send_frame(HW_addr, OUR_HW_ADDR, IP_TYPE, FALSE,
              fragment_pointer, fragment_length);
    { Update total_length of transmitted fragments }
            total_length := total_length + fragment_length;
          end; {while}
        end; {else}
        route_ip := OK;
      end; {else}
    end {if}
    { No possible way to transmit the packet ! }
    else route_ip := INVALID_TARGET;
  end; {else}

End; {route_ip}
```

## 2.11 IProute - IPlookup interface

```
function lookup_address(
  ip_target : ip_addr;      { Address to look up }
                          ): hw_addr;      { Result of search }

Begin {lookup_address}

    { Mutual exclusion }
  access();
    { Look for address in Address Table }
  tmp_addr := search_table();
    { Release table for use by others }
  release();
    { If table contains address, search is over, ... }
  if tmp_addr <> NULLHWADDR then lookup_address := result;
  else
    { ... otherwise an ARP search is required: }
  begin
    assemble_arp_request();
    send_frame(HW_BROADCAST_ADDRESS, OUR_HW_ADDRESS, ARP_TYPE,
    { Broadcast is true }
      TRUE, data_pointer, length);
    { Special return value to IProute, indicating the ARP search }
    lookup_address := DELAYED;
  end;

End; {lookup_address}
```

## 2.12 TCPserver - TCPport interface

```
procedure init_port(
  local_port       : tcp_port,    { To identify the correct process }
  remote_socket    : socket,      { Destination }
  type_of_service  : byte,        { Parameters to open the connection
                                    with }
  active           : boolean,
  window           : integer
                    );

Begin {init_port}

    { Find the requests-mailbox of the TCPport process managing
      'local_port' }
  mailbox_id := mailbox_dehash(local_port);
    { Then dump the information in that mailbox }
  write_to_mailbox_tcpport(mailbox_id, INIT, remote_socket, active,
    window, type_of_service);

End; {init_port}
```

```
procedure icmp_mess(
  local_port   : tcp_port,      { To identify the process' mailbox
                                      }
  icmp_type    : byte,      { ICMP information }
  icmp_code    : byte,
  iph_included : boolean,   { IP header present or not ? }
  iph_pointer  : pointer    { Length is known }
                    );

Begin {icmp_mess}

    { Find the requests-mailbox of the TCPport process managing
      'local_port' }
  mailbox_id := mailbox_dehash(local_port);
    { Copy data in mailbox }
  write_to_mailbox_tcpport(mailbox_id, ICMP, icmp_type, icmp_code,
    iph_included, iph_pointer);

End; {icmp_mess}
```

## 2.13 TCPport - TCPRx interface

```
procedure get_tcp(
  var tcp_pointer : pointer,    { On return it contains a pointer
                                  to a complete TCP segment }
  var tcp_length  : integer
              );

Begin {get_tcp}

    { First get an IP packet }
  receive_ip(our_circuit_handle, tcp_pointer, tcp_length,
    broadcast, TCP_PROT);
    { As long as a series of fragments is not completely received,
      assemble them }
  if fragmented(tcp_pointer, tcp_length) then
  begin
    start_timer();
    { Here the actions on timer-overrun are specified: }
    on timer_overrun do
    begin
      discard_all_fragments();
      notify_icmp();
    end;
    { And here the timer overrun instructions end }
    fragment_complete := FALSE;
    while not fragment_complete do
    begin
    { Assemble all fragments }
      temporarily_store_fragment();
    { Get next packet }
      receive_ip(our_circuit_handle, tcp_pointer, tcp_length,
        broadcast, TCP_PROT);
    { Check for completeness }
      fragment_complete := all_fragments_assembled();
    end; {while}
    compose_fragments_to segment();
  end;
  fill_out_variables();

End; {get_tcp}
```

## 2.14 TCPport - TCPTx interface

```
procedure put_tcp(
  connection      : tcp_conn_structure,     { Contains data fixed
                                              for the duration of
                                              the connection },
  segment_pointer : pointer,
  length          : integer
                  );

Begin {put_tcp}

    { Simply extract the correct data and call send_ip: }
  send_ip(OUR_IP_ADDR, connection.remote_ip, TCP_PROT,
    connection.type_of_service, connection.time_to_live,
    connection.fragmentation_allowed, segment_pointer, length);

End; {put_tcp}
```

## 2.15 take_fr - receive_fr interface

```
procedure check_fr(
  semaphore_id : semaphore         { The 'Sem' in HWRx }
               );

Begin {check_fr}

    { The HWRx semaphores are counting semaphores, initialized to
      the number of frame slots available in HWstore }
  P_operation(semaphore_id);

End; {check_fr}
```

```
procedure signal_fr(
  semaphore_id : semaphore
               );

Begin {signal_fr}

  V_operation(semaphore_id);

End; {signal_fr}
```

## 2.16 ARPserver - lookup interface

This includes access to the IPlookup semaphore 'Sem' only; both modules have access to the 'Address Table', subject to the use of the semaphore, but both use their own routines for such access.

```
procedure access(
  semaphore_id : semaphore
              );

Begin {access}

    { The IPlookup semaphore is a binary semaphore, controlling the
      mutual exclusion of 'Address Table' }
  P_operation(semaphore_id);

End; {access}
```

---

```
procedure release(
  semaphore_id :semaphore
                );

Begin {release}

  V_operation(semaphore_id);

End; {release}
```

---

---

## 2.17 MBX - router interface

```
procedure put_ip(
  packet_pointer : pointer,     { IP packet intended for this host
                                  }
  length         : integer      { ... }
                   );

Begin {put_ip}

  write_to_mailbox_MBX_in_IProute(packet_pointer, length);

End; {put_ip}
```

---

---

## 2.18 IPreceive - MB interfaces

```
procedure push_to_tcp(
  port    : tcp_port,    { To identify the mailbox }
  pointer : pointer,
  length  : integer
                    );

Begin {push_to_tcp}

    { Find the right mailbox first }
  mailbox_id := iptcp_mailbox_dehash(port);
    { Then dump the data in mailbox }
  write_mailbox(mailbox_id, pointer, length);

End; {push_to_tcp}
```

---

```
procedure put_icmp(
  data_pointer : pointer,
  length       : integer
                 );

Begin {put_icmp}

    { Transfer the data to ICMP through 'MB2' of IPRx }
  write_mailbox_MB2(data_pointer, length);

End; {put_icmp}
```

---

---

## 2.19 ICMPprocess - MBX interface

```
procedure transfer_icmp(
  message_pointer : integer,    { ICMP message ... }
  length          : integer,    { ... with length }
  source          : ip_address,     { Source of ICMP message }
                        );

Begin {transfer_icmp}

    { Put ICMP message in mailbox for TCP to process }
  write_mailbox_MBX_in_ICMP(message_pointer, length, source);

End; {transfer_icmp}
```

---

---

## 2.20 server - wait interface

```
procedure get(
  var item_pointer : pointer    { Supposed to point to an item in
                                  the mailbox, not exactly messages
                                  }
             );

Begin {get}

    { Read one item from mailbox and have 'item_pointer' point at
      it. Caller should know what it should be; first in a series
      is a byte always. }
    { Accepted are: TCP_OPEN, ICMP }
  item_pointer := ^read_mailbox_MBX_in_TCPserver();

End; {get}
```

---

```
procedure icmp_req(
  icmp_type    : byte,        { ICMP message type }
  icmp_code    : byte,        { Elaboration on type }
  icmp_pointer : pointer      { Pointer to first 12 bytes of ICMP
                                message }
  iph_included : boolean,     { Validates iph_pointer variable }
  iph_pointer  : pointer,     { Points to offending IP header plus 8
                                bytes of IP data }
                  );

Begin {icmp_req}

        { Copy the data into the mailbox as items, not as one
          message; code ICMP indicates an ICMP message }
  write_to_MBX_as_items(ICMP, icmp_type, icmp_code, icmp_pointer,
    iph_included, iph_pointer);

End; {icmp_req}
```

---

---

## 2.21 tcp_IF - TCPport interface

```
procedure ask_svc(      { Ask service }
  code          : byte,      { Code of requested operation }
  handle        : tcp_handle,    { To identify the right mailbox }
  data_pointer : pointer,
  length        : integer
                 );

Begin {ask_svc}

    { Find the mailbox belonging to the 'handle' TCPport process }
  mailbox_id := mailbox_dehash(handle.local_port);
    { Transfer a message into MB1 of TCPport (the request-mailbox)
    }
  write_to_mailbox_MB1_of_TCPport(mailbox_id, code, data_pointer,
    length);

End; {ask_svc}
```

---

```
procedure get_SC(                        { Get Service Code }
  var code         : byte,               { Code of requested operation }
  var data_pointer : pointer,            { Data, if any }
  var length       : integer
                 );

Begin {get_SC}

    { Find my mailbox }
  mailbox_id := mailbox_dehash(local_port);
    { Transfer a message from MB1 (the request-mailbox) }
  read_from_mailbox_MB1(mailbox_id, code, data_pointer, length);

End; {get_SC}
```

---

```
procedure put_result(
  result        : integer,   { Result code of operation }
  data_pointer : pointer,   { Data, if any }
  length        : integer
                     );

Begin {put_result}

    { Find my mailbox }
  mailbox_id := mailbox_dehash(local_port);
    { Transfer the results into it }
  write_to_mailbox_MB2(mailbox_id, result, data_pointer, length);

End; {put_result}
```

---

```
function get_result(
      handle        : tcp_handle,
  var data_pointer  : pointer,
  var length        : integer
                    ): integer;

Begin {get_result}

    { Find the mailbox of the right TCPport process }
  mailbox_id := mailbox_dehash(handle.local_port);
    { Then read the results from that mailbox }
  read_from_mailbox_MB2_of_TCPport(mailbox_id, result,
    data_pointer, length);
  get_result := result;

End; {get_result}
```

## 2.22 receive_fr functionality

```
process receive_fr();

  repeat
    { Wait for an interrupt to occur }
    await_int(HWSem);
    { Then take the frame }
    if fetch_fr(frame_pointer, length) = OK then
    begin
    { Find out protocol type }
      type := type_field(frame_pointer, length);
    { Store frame }
      put_fr(frame_pointer, length, type);
    { And signal the higher protocols (Semaphores arranged in an
      array, per protocol. Maybe hashing is cheaper, but not
      secure. }
      signal_fr(Sem[type]).
    end;    { Discard if an error occurred }
    { And then again: }
  until forever;

End; {receive_fr}
```

## 2.23 ARPserver functionality

```
process arpserver();

  repeat
    { Receive an ARP frame }
    type := ARP_TYPE;
    receive_frame(source, destination, type, data_pointer, length,
      broadcast);
    { Check it }
    if (type = ARP_TYPE) then
    begin
    { A request is always a broadcast ... }
      request := request_field(data_pointer, length);
      if (broadcast and (request = ARP_REQUEST)) or
    { and a reply is never a broadcast ... }
        (not broadcast and (request = ARP_REPLY)) then
      begin
    {  Reply or request ? }
        case request of
          ARP_REQUEST:
            begin
                { Check if they mean us }
              if target_ip_address(data_pointer, length) =
                OUR_IP_ADDRESS then
              begin
                assemble_response();
                send_frame( {destination is} source,
                            {source is} OUR_HW_ADDRESS,
                            FALSE, ARP_TYPE, data_pointer, length);
              end;
            end;
          ARP_REPLY:
            begin
                { Mutual exclusion in accessing Address Table }
              access();
                { Update table with new information }
              update_table();
                { And release the table for other users }
              release();
            end;
          otherwise  ;        { ERROR: do nothing }
        end; {case}
      end;
    end;
    { Then keep at it ... }
  until forever;

End; {arpserver}
```

## 2.24 route_receive functionality

```
process route_receive();

  repeat
    { Get IP frame }
    type := IP_TYPE;
    receive_frame(source, destination, type, data_pointer, length);
    { Check if really IP_TYPE }
    if type = IP_TYPE then
    begin
    { Convert frame into packet }
      extract_ip_packet();
    { Examine IP target address }
      target_ip_address := get_target_address(ip_pointer,
        ip_length);
    { Then pass it on to the router }
      route_ip(ip_pointer, ip_length, target_ip_address);
    { And go on ... }
  until forever;

End; {route_receive}
```

## 2.25 IPreceive functionality

```
process ipreceive();

  repeat
    { First get a packet }
    accept_ip(packet_pointer, length);
    { Then check it; ICMP request on error }
    if check_ip(packet_pointer, length) = ERROR then
    begin
      assemble_icmp_message();
      tmp_connection.remote_ip := source_field(packet_pointer,
        length);
      request_icmp(type, code, tmp_connection, iph_included,
        iph_pointer);
    end
    else
    begin
    { Extract protocol field }
      protocol := protocol_field(packet_pointer, length);
      if protocol = ICMP then put_icmp(packet_pointer, length)
      elseif protocol = TCP then
      begin
    { Extract local port number; TCP functionality }
        port := tcp_port_field(packet_pointer, length);
    { Then use the corresponding mailbox to drop the packet }
        push_to_tcp(port, packet_pointer, length);
      end;
    { Other layer 4 protocols go here }
    end;
    { Then start again: }
  until forever;

End; {ipreceive}
```

## 2.26 ICMPprocess functionality

```
process icmpprocess();

  repeat
    { Get ICMP message }
    receive_icmp(data_pointer, length, source, destination, tos);
    { Process the message if possible }
    process_icmp_message();
    { Transfer upward if necessary }
    if tcp_processing_needed then transfer_icmp(data_pointer,
                                                length, source);

    { An incoming ICMP message is never allowed to trigger another
      ICMP message ! }
    { Repeat this cycle forever }
  until forever;

End; {icmpprocess}
```

## 2.27 ICMPsend functionality

```
process icmpsend();

  repeat
    { Accept a request }
    receive_request(icmp_type, icmp_code, connection, iph_included,
      iph_pointer);
    { And execute it }
    assemble_icmp_message();
    if iph_included then append_ip_header(iph_pointer);
    { Send off in an IP packet }
    send_ip(OUR_IP_ADDR, connection.remote_ip, ICMP_PROT,
      connection.type_of_service, connection.time_to_live, FALSE,
      frame_pointer, length );
    { And look for new requests }
  until forever;

End; {icmpsend}
```

## 2.28 wait functionality

```
process wait();

  repeat
    { Wait for an ICMP message coming in }
    await_icmp(icmp_type, icmp_code, icmp_message_pointer,
      iph_included, iph_pointer);
    { And pass it on to the TCP server through a mailbox }
    icmp_req(icmp_type, icmp_code, icmp_message_pointer,
      iph_included, iph_pointer);
    { Forever and ever ... }
  until forever;

End; {wait}
```

## 2.29 server functionality

```
process server();

  repeat
    { Get a request }
    get(item_ptr);
    { Process it }
    case ^item_ptr of
      TCP_OPEN:
        begin
    { Try to open a TCP-connection }
          in_use := FALSE;
    { Is local socket in use ? }
    { ^ : Fill variable with value of item pointed at by the result
      of get() }
          get(local_socket^);
          if not local_socket_in_use(local_socket) then
    { If not create the process to handle a connection, and
      initialize it }
          begin
            create_tcp_port(local_socket);
            get(remote_socket^);
            get(active^);
            get(window^);
            get(type_of_service^);
            init_port(local_socket.port, remote_socket, active,
              window, type_of_service);
          end; {if}
        end;
      ICMP:
        begin
    { Get the items }
          get(icmp_type^);
          get(icmp_code^);
          get(iph_included^);
          get(iph_pointer^);
    { Get local port number from IP packet }
          lport := extract_local_port_from_tcp_bytes(iph_pointer);
    { And transfer: }
          icmp_mess(lport, icmp_type, icmp_code, iph_included,
            iph_pointer);
        end;
      otherwise:    { Error: ignore ! }
    end; {case}
  until forever;

End; {server}
```

---

These are the algorithmic descriptions. They have not been tested in any way, their sole purpose (for now) is to augment the pictures given in chapter 10.

# Appendix C WATERLOO PORT

## 3.1 Introduction

Waterloo Port is a network-oriented operating system, evoluated from the Thoth system (See [Che 1982]). It contains a real-time multitasking kernel for use on IBM PC and AT systems.
The computers can be connected into a Local Area Network using various different types of network hardware. Using the interprocess communication facilities offered, also capable of interprocess communication over the network, the system has successfully implemented a distributed computing environment.
The process structures of the Waterloo Port system will be introduced here, to give a basis for understanding the implementation examples presented in a later section of this appendix. The implementation examples themselves have been chosen in such a way as to provide a link between the functional design presented in chapter 10 and an eventual implementation in Waterloo Port.
The main point of discussion will be how to map the synchronization and communication structures used in the design to the structures available in Waterloo Port.

## 3.2 Waterloo Port process structures

The design of the Waterloo Port system is based on the concept of message passing as an interprocess communication method.
The specific variant uses blocked message passing. This implies that once an initiative to send a message to another process is taken, the sending process blocks until the receiving process issues a reply message. This is illustrated in figure 25.
When a process expects a message from someone else and initiates a receive from that process, the receiver will become blocked until the expected message is actually sent. This is illustrated in figure 26. Again, the sender of that message will get blocked until the receiver, who becomes ready at receiving the message, completes the task requested from him, and returns a reply.

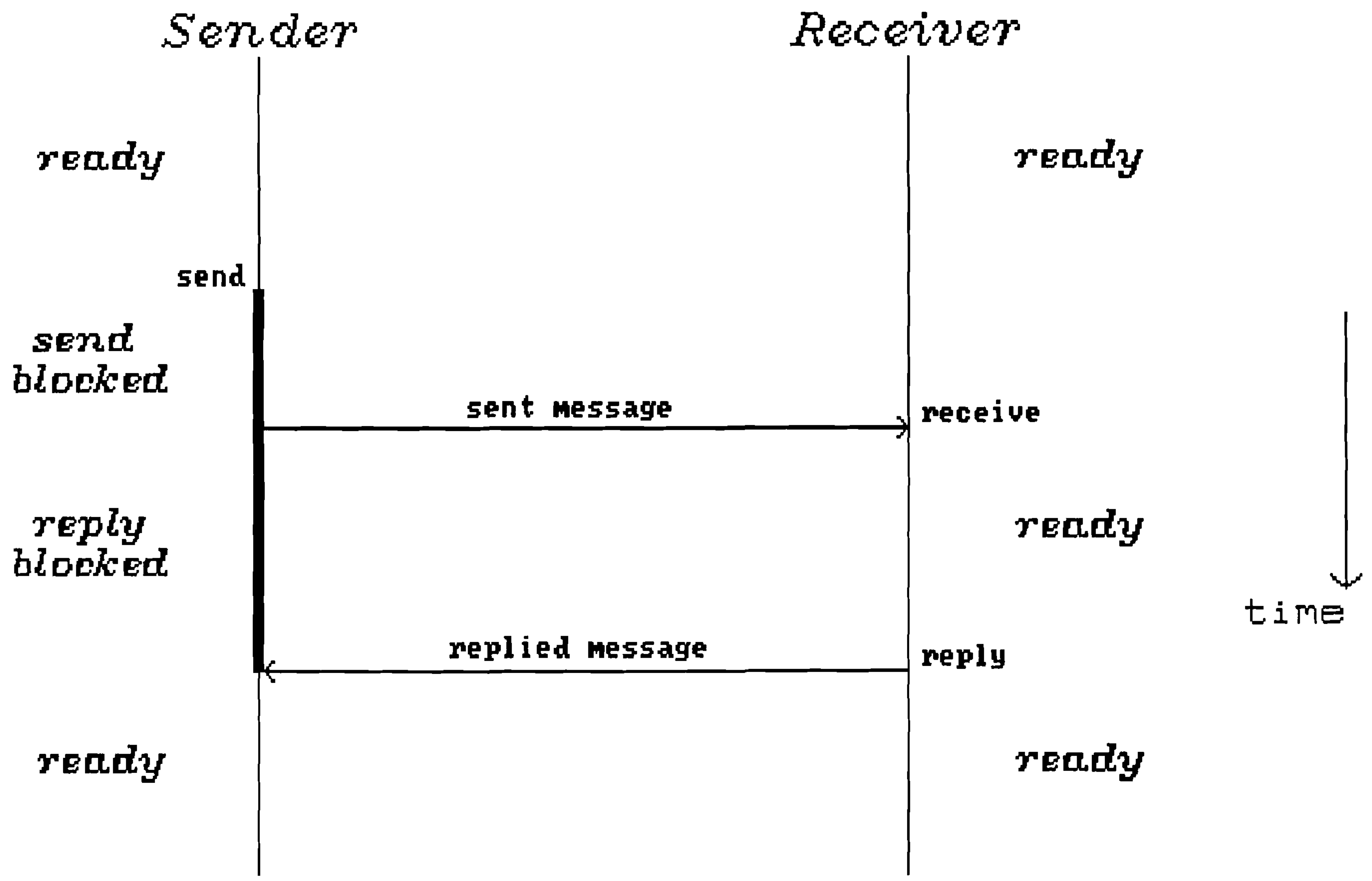


Fig. 25

Message pass (send first)

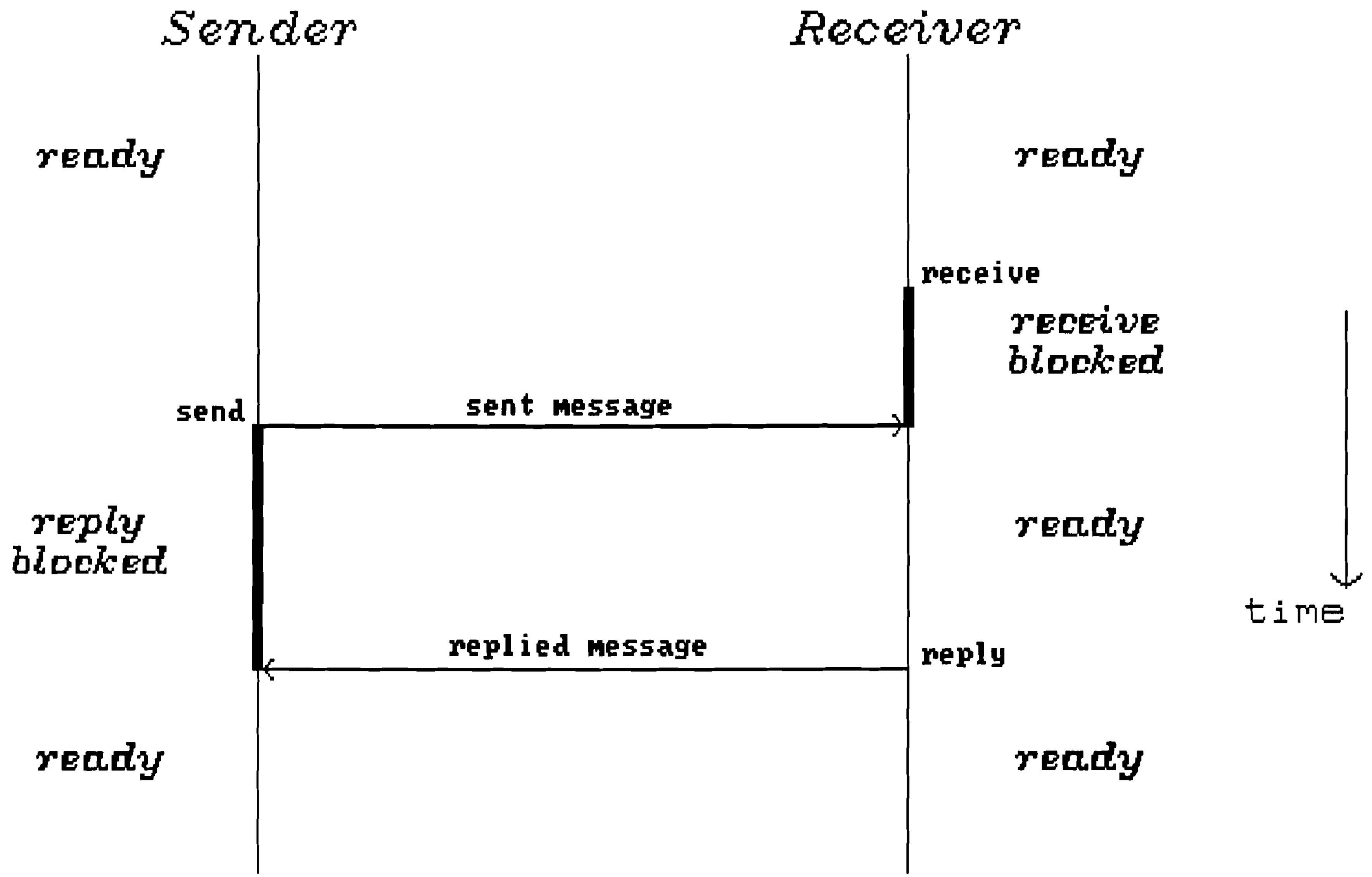


Fig. 26 Message pass (receive first)

This blocking principle when used with message passing has two main advantages:

- The transfer of the message does not require extra data space to store messages in transit: The sender has to prepare a message in its own memory in order to send it anyhow, as has the receiver to provide some data space in its memory to store the incoming message. The message passing primitives are operations built into the kernel of the Waterloo Port (or Port, for short) system, so the kernel can pass the message using the respective data spaces of the processes involved, without the need of intermediate storage.
- A second big advantage of the sender of the message being blocked while the receiver handles the incoming message is that its data space is frozen between the send and the corresponding reply. The kernel can (and does) provide a way for the receiver to transfer data from the sender's data space into the receiver's data space, and vice versa. This means that the messages themselves can be relatively small, even when large chunks of data need to be transferred between the processes.

Processes that receive all sorts of messages will benefit:
As long as simple messages are received, no space is being held in reserve for an occasional large message coming in. Instead, a large message is divided in a control part with all transfer information, and the actual information resides some place else. Only the control part is sent in the message, the data is then transferred by the receiver, if required. Only if and when such a message arrives, the receiver claims the data space needed (the actual amount needed is passed along in the message) and transfers the data. Allocate-on-demand is much more economical this way.
The principle of data transfer in bulk is graphically illustrated in figure 27.

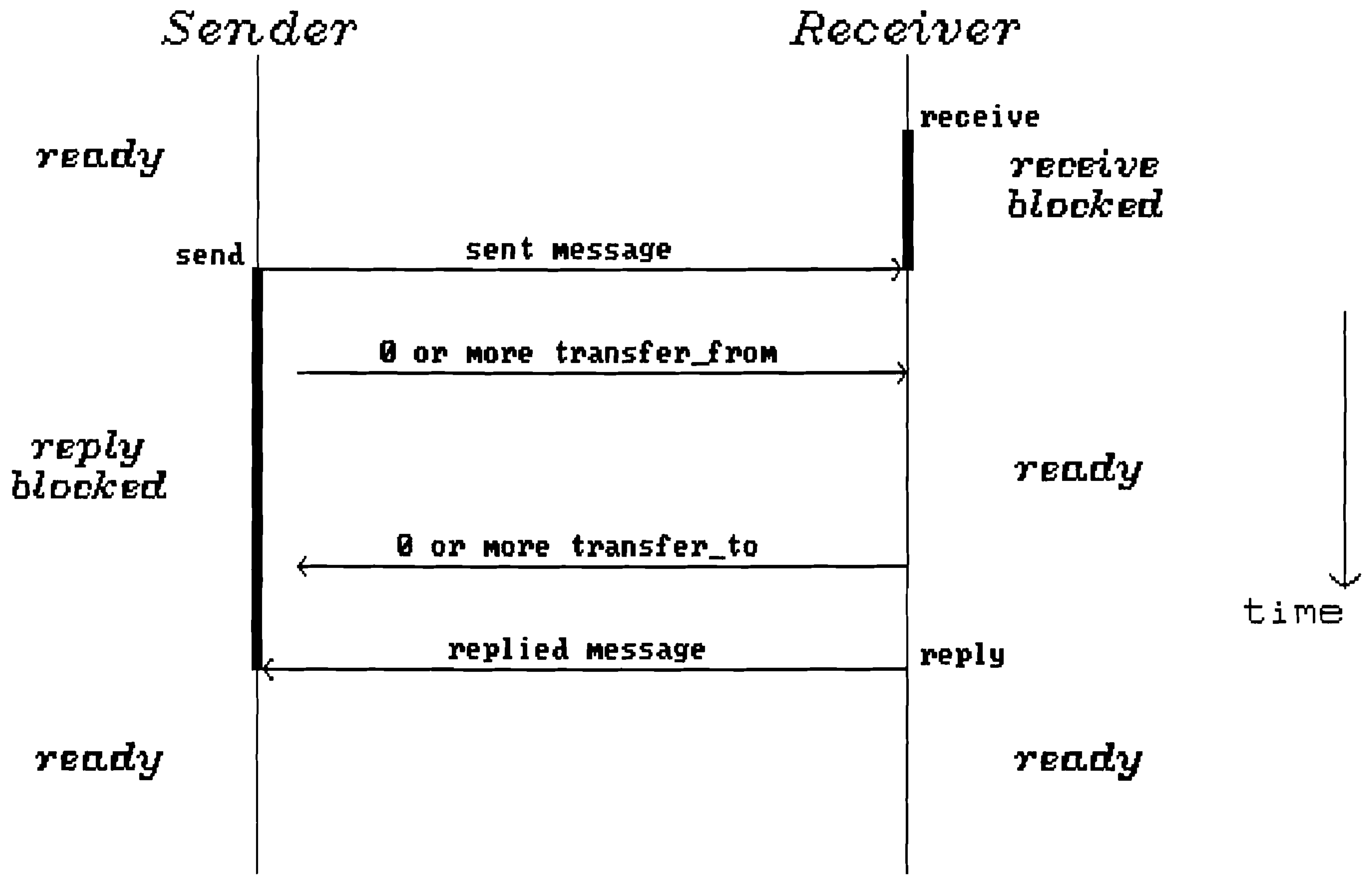


Fig. 27

Bulk data transfer

One last feature of the message passing system is of importance. The message passing primitives of Port return a result-code on completion. This result-code distinguishes two cases: Success or failure.

Failure can occur in two cases:

- The process the message is targetted at does not exist on the moment of the request. The primitives do not block in that case, because it would result in some sort of deadlock, creating unrecoverable resources.
- The targetted process did exist at the moment of the request, but was destroyed before the corresponding send/receive was executed. This implies that the requesting process has been blocked on completion of the message passing primitive, but the primitive is aborted because the other partner died prematurely.

These failure possibilities apply for all but one of the primitives within Port: The regular primitives 'send', 'receive', and 'reply' can all be aborted. (Reply can fail because some external event may have wiped out the blocked sender to which a reply is addressed.)

One primitive exists that cannot fail: 'receive_any'

The receive_any returns a process identification instead of a result-code, the process-id of the sending process.

This primitive is provided for server-like purposes: A server executes one 'receive_any' in its main working cycle, notes the process-id of the requesting process to be able to return a reply later, executes the request on behalf of the sender, replies and starts again. Or:

```
repeat
  pid = receive_any( <args> );
  execute();
  reply( pid, <args> );
until FOREVER;
```

This kind of offering service to all could not be accomplished easily without some 'receive_any' primitive.

## 3.3 Port process types

Port distinguishes several different process types. These subdivisions are made on the interrelationships of these processes with their direct environment, and for some cases they are too vague. But all in all they provide a useful classification mechanism.
In figure 28 a legend is presented for use with the next series of pictures. The double-arrow and its function will be explained in the chapter on the 'vulture' process type. The initialization arrow indicates the hierarchy of process creation and has no other function.
The send arrow points in the 'direction' of the send primitive: The process pointed at executes a receive, and later a reply, the other executes a send.
The rest will be clear when the text to the pictures is examined.

| | |
|---|---|
| ○ | Process |
| → | Send |
| ⇒ | Receive specific |
| ⇸ | Send (Initialization) |

Fig. 28 *Legend*

### 3.3.1 Proprietor

The proprietor is the simplest classification for a server. It accepts requests from client processes (client is not in the classification system, but indicates that the process called 'client' uses the services of another process), uses an algorithm of the kind presented at the end of section 3.2 of this appendix to service all requests, and replies. In this way a proprietor can handle only one request at a time. See figure 29.

### 3.3.2 Administrator and worker

The administrator is somewhat more complex. The administrator can handle more than one request at a time; for this purpose worker processes are used. Operation is generally as follows:
(See figure 30)
The administrator accepts requests from the clients, one at a time. To handle the request it creates a worker process, initializes it with the information from the client, the worker replies immediately, thereby freeing the administrator for other work.
At this moment the worker can do its job, and the administrator can proceed with accepting requests.
When the worker has the results, it sends a special request to the administrator, say WORKER_READY, whereupon the administrator accepts the results from the worker and transfers them to the waiting client. The worker then destroys itself.

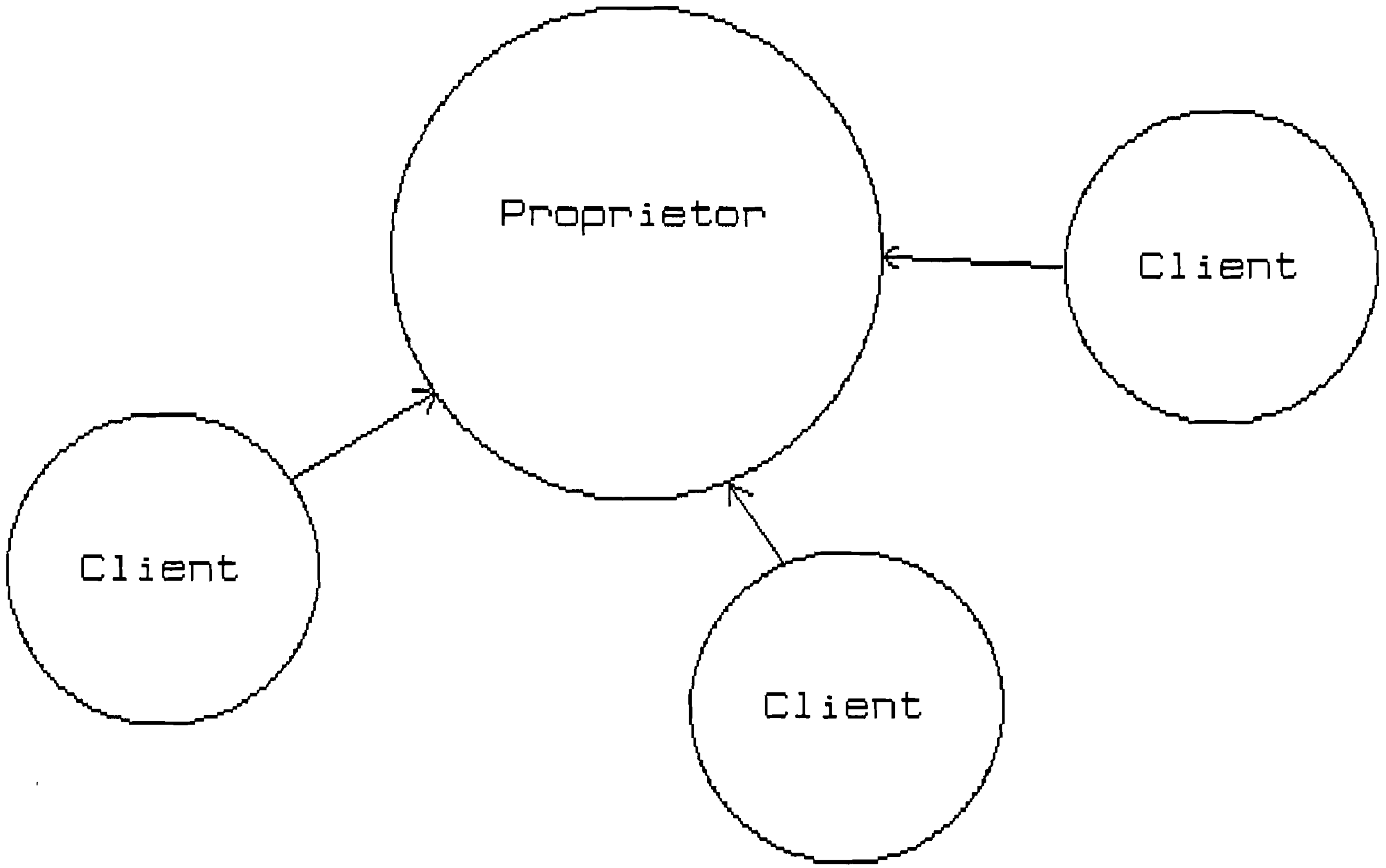


Fig. 29

# *Proprietor*

A prototype administrator could look like this:

```
  repeat
    pid = receive_any( <args> );    { This  comes  from  client  or
                                      worker }
    case request of
      CLIENT_REQUEST:
        begin
          wid = create_worker();    { Use   worker   to   support
                                      multiple clients }
          send( wid, <init_data> );
        end;
      WORKER_READY:
        begin
          transfer_from( wid, <args> );     { Accept results }
          reply( wid, <args> );             { Kill worker }
          transfer_to( pid, <args> );       { Report results }
          reply( pid, <args> );             { Free client }
        end;
    end; {case}
  until FOREVER;
```

Largely simplified, of course. One of the things missing here is a way to distinguish more than one client; important in practice, but not for our purpose.

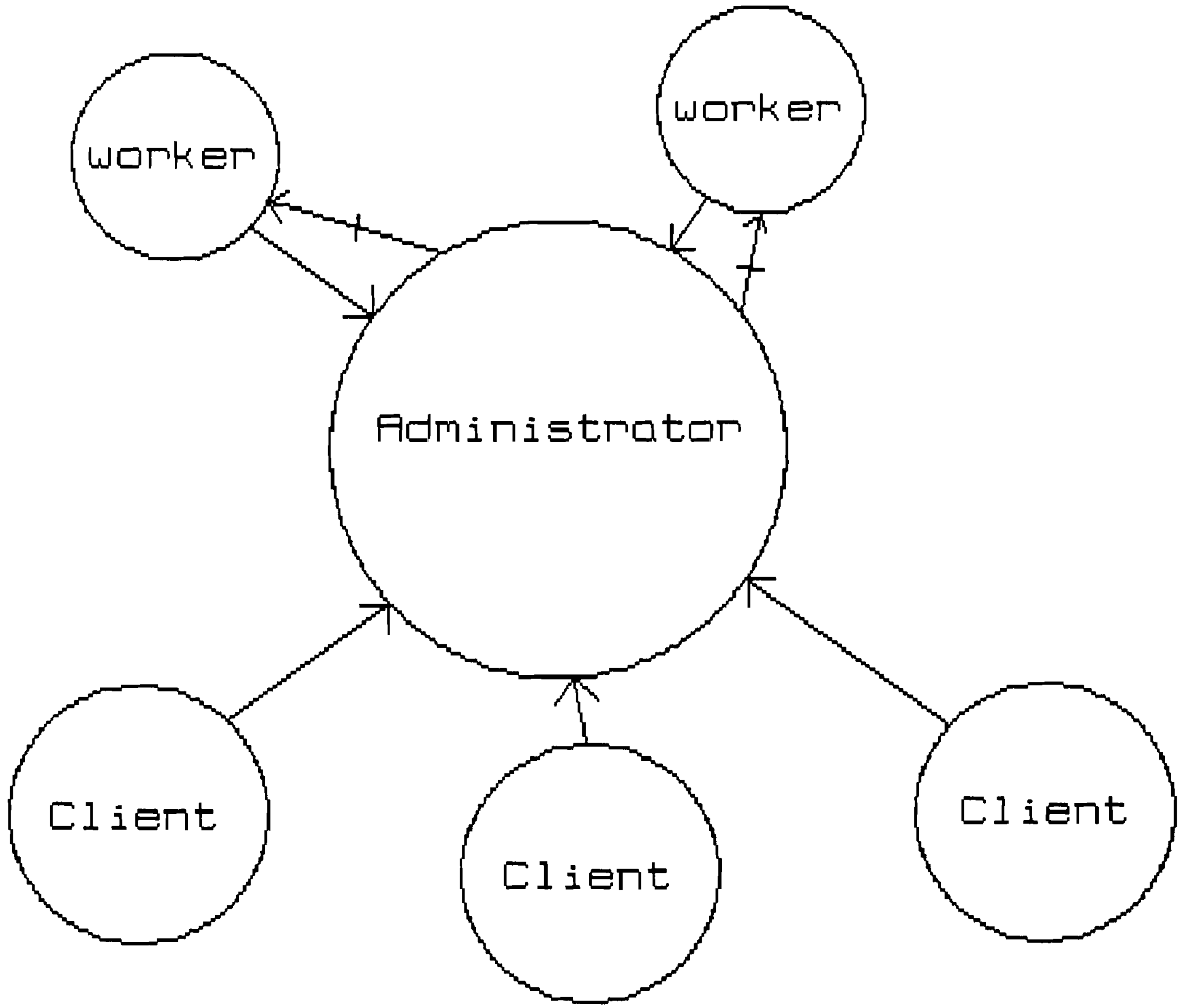


Fig. 30

Administrator

A worker could look like this:

```
  begin
    mid =  receive_any( <args> );    { How could he know his master
                                       yet ? }
    reply( mid );                    { First release master }
    work();                          { Perform      actions      for
                                       administrator }
    send( mid, WORKER_READY, <args> );
              { Report results are ready, and offer possibility
                to transfer them from worker to administrator }
    suicide();  { When ready, destroy yourself }
  end;
```

These two server types, proprietor and administrator, form the basis for all servers. More complicated variants exist, in which for example the worker is allowed to communicate with the client it services, but the principles are the same.
Some process types exist with a special function; they will be discussed in the next few subchapters.

### 3.3.3 Courier

A courier provides an indirection mechanism for data transfer.
When two servers have to communicate, it is not desirable to have one server block on a message pass to the other, because servers in general have a community-serving function: They should be available whenever possible. That is also the reason why servers receive messages rather than send them, because senders block, but never a receive_any.
To let two servers communicate, a courier is inserted (see figure 31). The courier is initialized by the server wishing to communicate, and both process-id's of the server processes are given to it. The courier then sends a special request to the sending server, asking for information to transfer. When that has arrived, the courier sends a related request to the other server indicating that information has arrived.

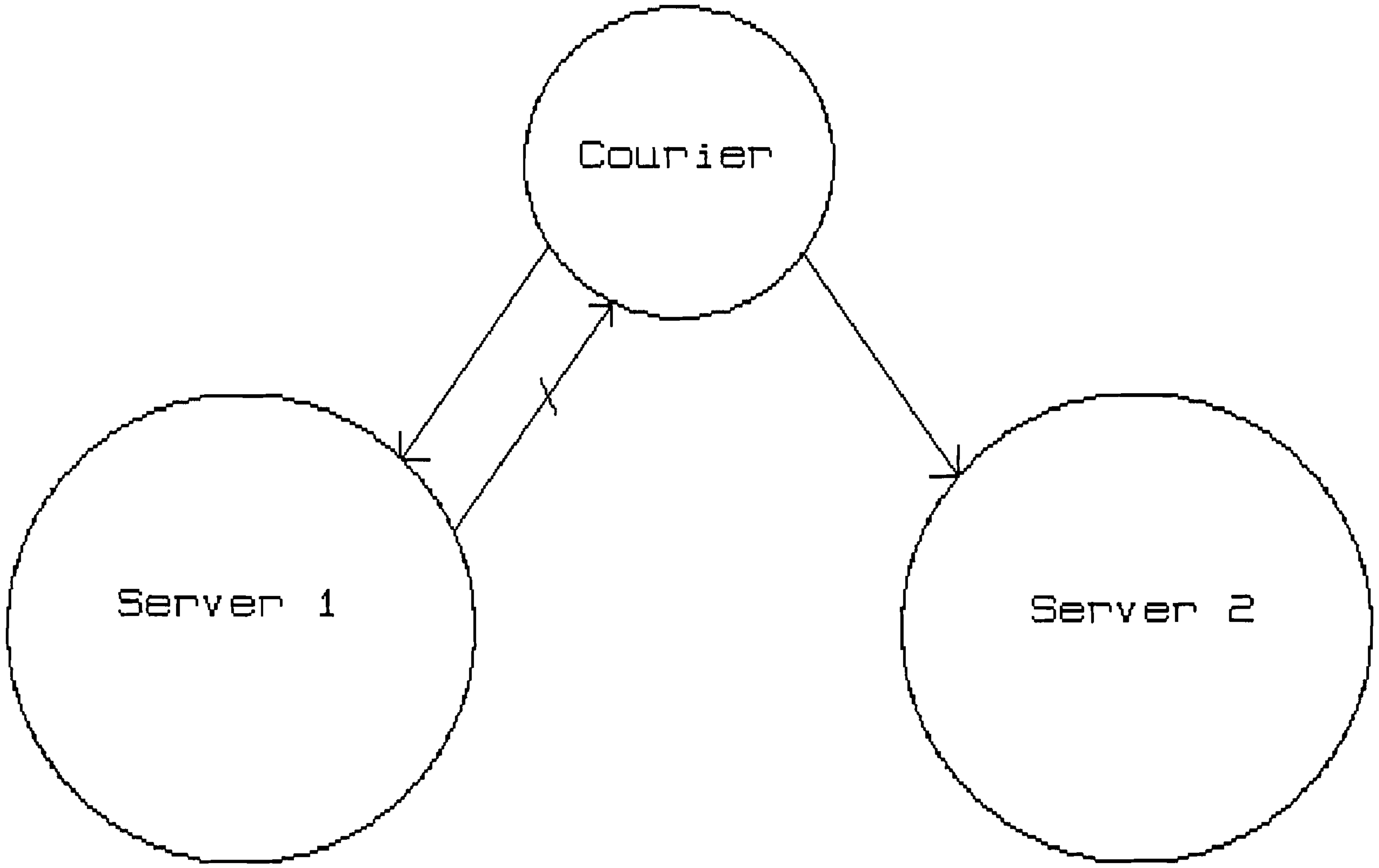


Fig. 31

Courier

Somewhat like this:

```
  begin
    mid = receive_any( target_server_id, <args> );
                { To receive both process id's }
    repeat
      send( mid, <args> );
                { To get the data }
      send( target_server_id, <args> );
                { To pass the data on }
    until FOREVER;
  end;
```

When the services of the courier are not needed anymore, the courier is destroyed.

### 3.3.4 Notifier and vulture

Two special couriers have their own classification. See figure 32.
The notifier serves as a synchronized version of an interrupt. As stated previously, the server cannot be expected to block on some event, whether it be a message pass or an external event.
The message pass block is circumvented with the courier, in the most general case.
The special case of an external event is circumvented with the notifier. A notifier is initialized by a server by passing the server's process-id on to the notifier. The notifier starts waiting on an event (implemented in the system as a kernel operation; the notifier becomes event-blocked), and as soon as the event occurs, the notifier performs a send to his server, with a special request.

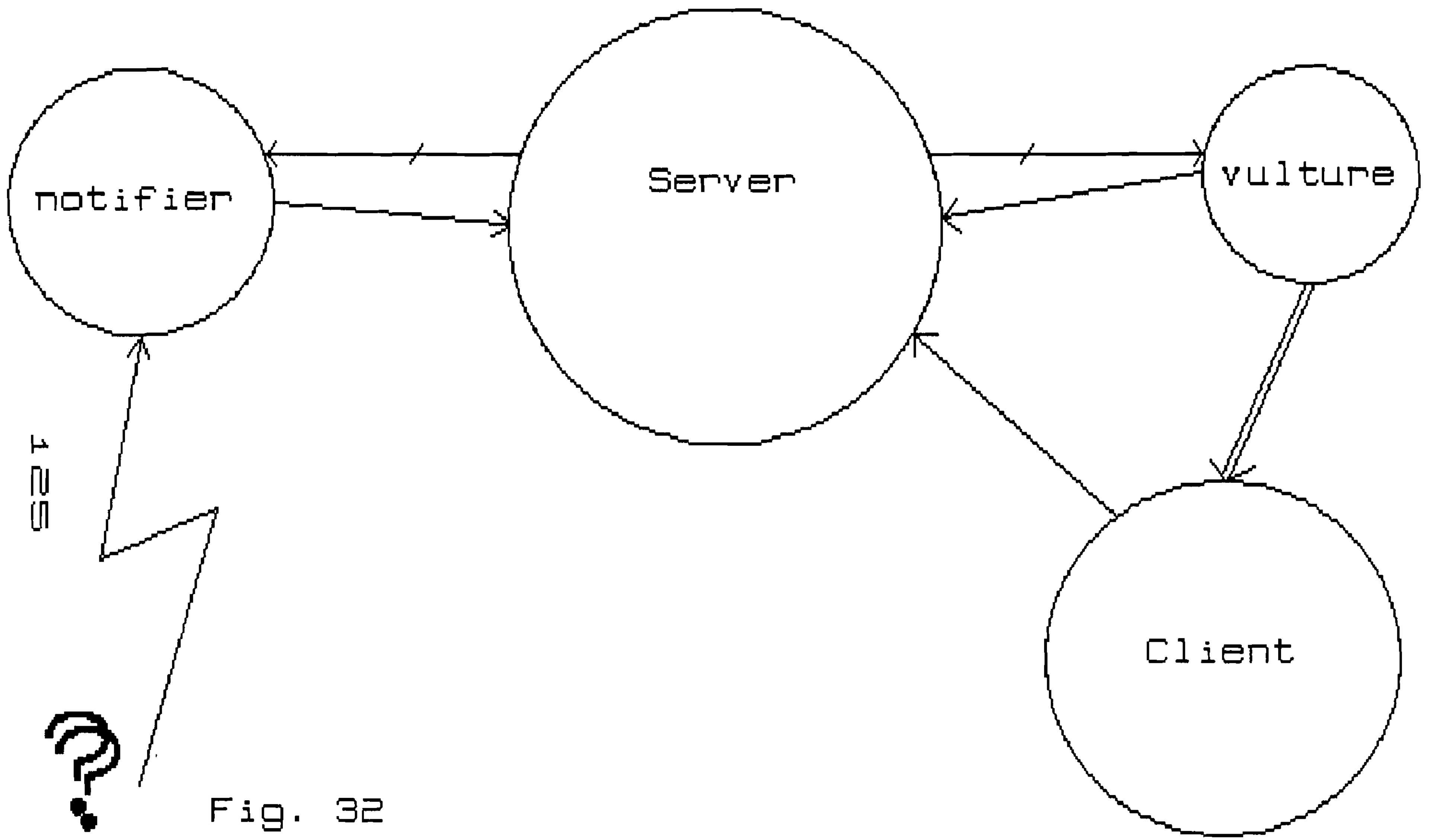


Fig. 32

Notifier/Vulture

Like this:

```
begin
  mid = receive_any( <args> );
  event_wait( <event> );
  send( mid, EVENT_OCCURRED, <args> );
  suicide();
end;
```

This notifier is intended for a one-time event. Others may repeat forever, like a simple courier.

A vulture is an insurance policy for a server. It checks whether a client, on whose behalf the server has allocated expensive resources, is still alive and kicking. It may work like this:
A server receives a request from a client, and with the request comes the process-id of that client. The server knows that expensive resources have to be allocated for this request, and therefore calls upon a vulture. This vulture gets the process-id of the client, and of course of the server itself, and then blocks on a receive from the client.
Now a nice feature works for the server: Normally the client does not know that someone is interested in receiving something from him, so the client does not send anything. Strangely enough, the vulture doesn't expect anything being sent.
Only, when the client happens to kill himself, during a prolonged client-server relationship in which the client continues to use the server's facilities, or is killed by someone else, the receive executed by the vulture fails, unblocking the vulture.
The vulture then sends a special request, say CLIENT_DIED, to the server, whereupon the server can harmlessly free the resources taken by that specific client.
So a vulture is a sort of a death courier.

## 3.4 Implementing semaphores in Waterloo Port

The most general way to implement semaphores in Port could be to create a 'semaphore server'. This server is implemented as a process to which requests can be sent. Some possible requests could be, for instance:

- CREATE_SEMAPHORE, indicating a binary/counting semaphore, initialization value (0/1 for binaries, non-negative for counting semaphores); the request would return a 'semaphore ID', for later reference in all the other kinds of requests.
- P_OPERATION, to decrement the semaphore if the current value is greater than zero, or to block until an increment is done when not.
- V_OPERATION, to increment the semaphore.
- DESTROY_SEMAPHORE, to free the semaphore resource when not needed anymore.

The most interesting request in this structure is the P_OPERATION.
A process sending such a request will be blocked until the semaphore server replies. If the current semaphore value is zero, the server should not reply until another process has completed a V_OPERATION. So, instead of replying, it remembers the process-id of the requesting process in a First-Come-First-Served structure, and resumes operations. As soon as a V_OPERATION is executed on behalf of another process, the semaphore value could be incremented from zero to one; however, if the structure of waiting processes contains any valid entries, the first process in line will get its reply, while the semaphore value stays zero. The latter process becomes unblocked and can assume correctly that the P_OPERATION request is finally honored.
If the server maintains integrity at all times, the preemptive scheduling of processor time cannot interfere with the operations based on the use of semaphores.
A big advantage of this approach can be that once the server is written and operational, everyone who needs a semaphore can request one from the server. This can very well be a good solution for all those cases where multiple semaphores are used for every protocol involved, see for the design pictures chapter 10.
An algorithm for such a server could look something like this:

```
repeat
  pid := receive_any( request, <args> );
  case request of
    CREATE_SEMAPHORE:
      begin
        semID := create_semaphore_structure( type );
        initialize_sem( semID, value );
        reply( pid, semID );
      end;
    P_OPERATION:
      if value( semID ) > 0 then
      begin
        decrement_value( semID );
        reply( pid );
      end
      else store_id_of_requestor( pid );
    V_OPERATION:
      begin
        if (value( semID ) > 0) and (value( semID ) < semID.max )
        then begin
          increment_value( semID );
          reply( pid );
        end
        elseif value( semID ) = 0 then
        begin
          next_id := retrieve_and_delete_next_P_request();
          reply( next_id );    { Unblock next P_operation request
                                 }
          reply( pid );    { The report success of V_operation }
  { This works because the V_OPERATION requestor is blocked as
    long as the second reply is not given; it can thus not take
    advantage of the fact that officially it is still in control
    of the protected resource (for instance) }
        end
        else
          do_something_about_exceeding_the_maximum_value();
      end;
      DESTROY_SEMAPHORE:
        begin
          remove_semaphore_structure();
          reply( pid );      { ALWAYS REPLY to unblock the process
                               friendly   enough   to   indicate   an
                               unnecessary semaphore }
        end;
      otherwise     { Illegal request }
        reply( pid );    { So issue a dummy reply ! }
    end;
  until FOREVER;
```

The picture of such a semaphore server would be identical to the proprietor picture in section 3.3.1 of this appendix. Please refer back.

Another way to implement a semaphore in Waterloo Port is to omit it altogether. This may seem rather strange, but in fact it is not.
The blocking message passing mechanism is ideal for synchronizing two processes, so if the sole purpose of a semaphore is to synchronize, it is cheapest to implement when a semaphore is not used at all. Silent assumption is that the semaphore in the design is a binary one, counting semaphores are mostly not used for synchronizing.
The process waiting for the other one with a P-operation on the semaphore could perform a receive from the other process. The equivalent of the V-operation would then be a send to the first process; both sides block if the other side is not ready, both sides unblock when the other side becomes ready, and the synchronizing is in effect.
A different way of solving the problem can be found in the notifier process: A notifier waits for some external event, and then proceeds (with notifying its master). This structure can replace a synchronizing semaphore as seen in the hardware module presented in chapter 10 (figure 15). In that picture it is assumed that the operating system can convert an interrupt into a semaphore action. The semaphore action can be reacted upon by the software.

In the Waterloo Port environment the possibilities are different but not less powerful: Software can react directly on a hardware interrupt, thereby eliminating the need for an extra semaphore.

## 3.5 Implementing mailboxes

Some kinds of mailboxes are as easy to implement as semaphores: Just leave them out. In this case, however, that has some consequences.
The 'receive_any' primitive of Port can be used to serve as a mailbox. Only, the storage capacity usually associated with a mailbox is zero. That means that everyone who has a message for a mailbox has to wait his turn. When that is not a disadvantage the solution can be used.
When it is a disadvantage and another solution has to be found, problems arise: An extra process has to be provided to supply extra storage facilities.
That process could be a server accepting two kinds of requests:

- A MAILBOX_DEPOSIT request from clients who want to store a message for another server to process them; and
- A MAILBOX_RETRIEVAL request from that server to process the message.

An algorithm for such an intermediate process will be presented below. The example illustrates a single mailbox; extensions can be made to create a general process administering all mailboxes in the system if that is desired, but the principles are similar to the semaphore case and therefore we will not discuss that further.
Here is the algorithmic description:

```
repeat
  pid := receive_any( request, <args> );
  case request of
    MAILBOX_DEPOSIT:
      begin
        store_message_in_FCFS_structure();
        reply( pid );
      end;
    MAILBOX_RETRIEVAL:
      begin
        get_next_message_from_storage();
        reply( pid );
      end;
  otherwise   { Illegal request }
    reply( pid );     { Dummy reply }
  end;
until FOREVER;
```

If mailboxes are used to transfer data between processes, while at the same time synchronizing them, again it is possible to omit the actual mailboxes at no additional cost (although the 'receive_any' primitive is not necessarily used, since we are not talking about a server accepting requests through mailboxes; just two processes knowing eachother and with a desire to synchronize and transfer data at the same time).

## 3.6 Conclusions

Waterloo Port offers a set of interprocess communication primitives that can be used to simulate virtually every other sort of IPC.
The implementation of semaphores and mailboxes can be done in a very general way using semaphore respectively mailbox servers; these servers are modelled analogous to the proprietor process structure.

Very often, however, the mailboxes or semaphores can be omitted from the implementation, because the blocking message passing system used incorporates many of the essential features of both mailbox and semaphore:

- Synchronization
- Data transfer
- Mutual exclusion facilities, in some cases at least

This will make it easy to make a detailed, Waterloo Port oriented design from a general one.

## Appendix D ABBREVIATIONS

ARP
  Address Resolution Protocol

CCITT
  Consultative Committee on International Telephony and Telegraphy

CONS
  Connection Oriented Network Services

CSMA/CD
  Carrier Sense Multiple Access with Collision Detect

DNA
  Digital Network Architecture

FTAM
  File Transfer and Access Management

FTP
  File Transfer Protocol

ICMP
  Internet Control Message Protocol

IEEE
  Institute of Electrical and Electrotechnical Engineers

IMP
  Interface Message Protocol

IP
  Internet Protocol

IPC
  InterProcess Communication

ISO
  International Standards Organization

LAP
  Link Access Procedure

LAPB
  Link Access Procedure - Balanced

LLC
  Logical Link Control

MAC
  Medium Access Control

| | |
|---|---|
| MHS | Message Handling System |
| MLP | MultiLink Procedure |
| NCP | Network Control Protocol |
| OSI | Open Systems Interconnection |
| PDN | Packet Data Network |
| SDLC | Synchronous Data Link Control |
| SMTP | Small Mail Transfer Protocol |
| SNA | System Network Architecture |
| TCP | Transmission Control Protocol |
| UDP | User Datagram Protocol |
| VTS | Virtual Terminal Service |
| XNS | Xerox Networking System |

## Appendix E LITERATURE

[Asc 1986]
Aschenbrenner, J.S.; Open Systems Interconnection;
1986, IBM Systems Journal, Vol. 25, No. 3/4, pp. 369 - 379;
Reprint order no. G321-5281

[Bac 1988]
Backes, F.; Transparent bridges for interconnection of IEEE 802 LANs;
1988, IEEE Network, Vol. 2, No. 1 (January), pp. 5 - 9;

[Bos 1988]
Bosack, L., C. Hedrick; Problems in large LANs;
1988, IEEE Network, Vol. 2, No. 1 (January), pp. 49 - 56;

[Che 1982]
Cheriton, D.R.; The Thoth system: Multi-process structuring and portability;
1982, North Holland Publishing Company;
<Computer Science Library>
<Operating and programming systems series: 8>

[Com 1988]
Comer, D.E.; Internetworking with TCP/IP; Principles, protocols, and architecture;
1988, Prentice-Hall Inc., Englewood Cliffs;
ISBN 0-13-470-188-7

[Dav 1988]
Davidson, J.; An introduction to TCP/IP;
1988, Springer-Verlag, New York;
ISBN 0-387-96651-X
3-540-96651-X

[Dix 1988]
Dixon, R.C., D.A. Pitt; Addressing, bridging, and source routing;
1988, IEEE Network, Vol. 2, No. 1 (January), pp. 25 - 32;

[Ham 1988]
Hamner, M.C., G.R. Samsen; Source routing bridge implementation;
1988, IEEE Network, Vol. 2, No. 1 (January), pp. 33 - 36;

[Har 1988]
Hart, J.; Extending the IEEE 802.1 MAC bridge standard to remote bridges;
1988, IEEE Network, Vol. 2, No. 1 (January), pp. 10 - 15;

[Pad 1988]
Padlipsky, M.A.; At last, the last word;
1988, IEEE Network, Vol. 2, No. 1 (January), pp. 92 - 93;

[Per 1988]
Perlman, R., A. Harvey, G. Varghese; Choosing the appropriate ISO layer for LAN interconnection;
1988, IEEE Network, Vol. 2, No. 1 (January), pp. 81 - 86;

[Rou 1987]
Routt, The.J.; Distributed SNA: A network architecture gets on track;
1987, Data Communications, Vol. 16, February, pp. 116 - 134;

[Sei 1988]
Seifert, W.M.; Bridges and routers;
1988, IEEE Network, Vol. 2, No. 1 (January), pp. 57 - 64;

[Tan 1981]
Tanenbaum, A.S.; Computer networks;
1981, Prentice-Hall Inc., Englewood Cliffs;
ISBN 0-13-164699-0

[Zha 1988]
Zhang, L.; Comparison of two bridge routing approaches;
1988, IEEE Network, Vol. 2, No. 1 (January), pp. 44 - 48;

Appendix F INDEX

[1]:
The TCP/IP protocol suite is a research project. Intermediate results are used for implementation, and many people regard the suite as an industry standard.
The intermediate results are published in the so called 'Request For Comment' papers, in a quite informal way. Some of these RFC's contain the definitions of the current protocols, others just some test results. The whole history of the ARPANET can be found in RFC's. In [Com 1988] a considerable number of pages is devoted to some descriptions and overviews of the available RFC's. Ordering information by phone, mail or E-mail can be found there too.
For completeness sake:

```
      DDN Network Information Center       Phone: 1 (800) 235-3155
          SRI International      Outside USA/CDN: 1 (415) 859-3695
      333, Ravenswood Avenue
  USA - Menlo Park, CA 94025
```

This center can provide the RFC's as well as the DDN Protocol Handbook, which contains many of the relevant RFC's (December 1985).