ETH

Eidgenössische
Technische Hochschule
Zürich

Departement Informatik
Institut für
Computersysteme

Niklaus Wirth

Ceres-Net: A Low-Cost Computer Network

Extending Ceres-Net by a Mail Service

September 1989

112

Authors' address:

Institut für Computersysteme
ETH-Zentrum
CH-8092 Zürich, Switzerland

Ceres–Net: A Low–Cost Computer Network

N. Wirth

Summary

This project report describes a local–area network, its hardware and software structure, and its protocol. The principal objective was to minimize the complexity of hard– and software, yet to offer an adequate functionality and performance for a system of up to 30 workstations with local file stores. The primary benefit of the achieved simple structure and low–overhead protocol is the system's reliability.

Introduction

Ceres is a workstation designed for practically autonomous use. It is based on a 32–bit microprocessor (NS 32000), a 2–4 MByte main store, and a 40–80 MByte disk store [1]. Ceres–Net, a local area network, augments its capabilities, making communication between workstations possible, and thereby allowing for the installation of dedicated machines providing file, print, and mail services.

The view of essentially autonomous workstations with support from servers via a network stands in constrast to the distributed computing system based on disk–less computers. We prefer the former, because it avoids bottlenecks and breakdown due to failure of a single component (disk server). The view of a net as an auxiliary rather than a crucial facility allows the consideration of low–cost solutions. The term "low–cost" applies to both hard– and software. In the former area it implies the need for a few interface components only and a twisted–pair wire instead of coaxial cable. In the area of software, the notion of low–cost implies a small set of relatively simple modules, a transparent structure guaranteeing effectiveness, reliability, and robustness.

Our willingness to restrict the network to specific functions (in contrast to including/integrating all connected stations into a single address space) should be rewarded by a structure of the network software that is reasonably straight–forward. In particular, it should avoid additional complexity primarily induced by generality–considerations. We refrain from introducing superfluous layers of abstraction that hardly make sense in a local–area network, and restrict the system to three layers: the signal–level (hardware) layer, the level incorporated by the (software) driver, and the application level defined by the data packet format.

In this paper, which intentionally bears traces of a tutorial as well as a project report, we start by presenting the hardware facilities on which Ceres–Net builds. This includes the SDLC packet format. We proceed by specifying the procedural interface of the network driver module. This is followed by a discussion of the Oberon system's metaphor which precludes the notion of indivudual processes, and of the integration of network services in the Oberon single–process system. A presentation of the communication protocol precedes the chapter on the file transfer service routines, the principal function of Ceres–Net.

And finally, we show how the network module can be extended, thus gradually converting a workstation into a server station, again without reliance on the concept of multiprocessing. The paper concludes with some performance figures.

The hardware level

Ceres–Net is based on the principle of a bus, i.e. of a wire to which all participants are connected in the same way by a transmitter and a receiver. Electrically, the net adheres to the RS–485 Standard, using a twisted wire pair to transmit the balanced signal (Fig. 1). The Standard postulates that each transmitter

be capable of driving at least 32 receivers, and it garantees immunity against large differences in electrical ground level between transmitter and receiver. At any time, only a single transmitter should be active. However, the Standard ensures that even in the case of several transmitters becoming active (collision) no physical damage may result. Unlike in the use of open-collector drivers, a collision cannot be detected by concurrently monitoring the signal on the wire. It will be distorted, however, and such distortion must be detectable by the receiver, making it necessary to provide redundancy that can be checked.

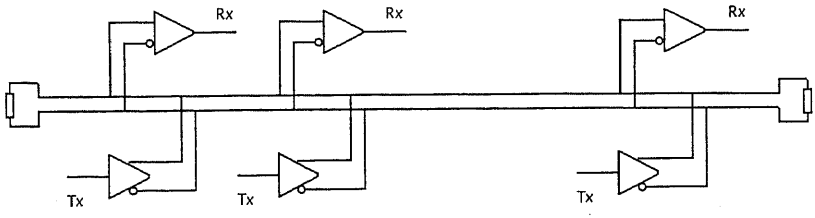

Fig. 1. RS 485 Bus

Since asynchronous, byte-by-byte transmission is inappropriate for data rates greater than 20 Kbit/s, redundant checking information cannot be attached to each byte, and hence a packet format is necessary. Packets are transmitted as a whole in synchronous mode. Ceres-Net adopts the standard SDLC-format (synchronous data link control).

SDLC packets may be of any length (not necessarily a multiple of 8). Beginning and end are marked by a "flag", consisting of 6 consecutive ones. If such a sequence is detected within the data packet, a zero is inserted by the transmitter's encoder and removed by the receiver's decoder. Furthermore, SDLC specifies that the first 8 bits of a packet be a destination address, and the last 16 bits be a cyclic redundancy checksum CRC (Fig. 2).

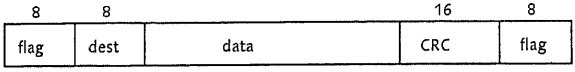

Fig. 2. SDLC Packet Format

The interface between the Ceres processor bus and the network consists of a Zilog 8530 Serial Communications Controller chip (SCC). The only additional components are drivers on both sides (Fig 3). The transmission rate is 250 Kbit/s.

The "cheapness" of this configuration rests on the following characteristics:

1. Data are transferred under control of the central processor. No facilities for direct memory access (DMA) are needed. This technique is appropriate for data rates less than 1 Mbit/s.

2. Serialization and deserialization of data as well as redundancy generation and check is performed by the SCC.

3. The SCC contains an address filter which, in accordance with the packet format, suppresses attention to packets not addressed to the particular station.

The SCC chip offers much flexibility that remains unused. It can be used in asynchronous, synchronous, SDLC, or SDLC loop mode; it contains two transmitter/receiver pairs, of which one remains idle. All this would not merit to be mentioned, were it not for the fact that this flexibility complicates both the chip's interior and its use. The unused facilities cannot merely be ignored; certainly not for the controller's initialization. They even cause serious difficulties, if incompletely documented.

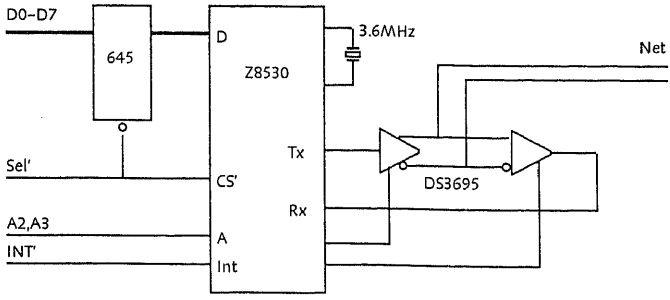

Fig. 3. SCC Interface

The SCC software interface

The purpose of the module called SCC is to provide a procedural interface to the network. On the one hand it should offer all the functionality given by the hardware, on the other hand it should shield the hardware from inadvertent use and provide the programmer with an appropriate abstraction.

The obvious abstraction here is the data packet and the operations to send an receive a packet. This implies the definition of a packet format as a data type. A Ceres-Net packet consists of two parts: the header and the data. The header is defined as a record type, the data as any sequence of bytes.

```
TYPE Header =
    RECORD valid: BOOLEAN;
        dadr, sadr, typ: SHORTINT;
        len, destLink, srcLink: INTEGER
    END
```

The procedure used to send a packet is defined as

```
PROCEDURE SendPacket(VAR head: Header; VAR data: ARRAY OF BYTE)
```

and causes the packet to be transmitted in the form shown in Fig. 4.

flag	dadr	sadr	typ	length	dstLink	srcLink	data	CRC	flag

Fig. 4. Ceres-Net Packet Format

The procedure requires the data to be passed as a single array parameter. This is necessary, because the sending of the packet occurs under strict timing constraints. A byte must be supplied (under processor control) every 35 ms. This implies that the processor must not be interrupted during transmission of a packet. We limit this period to 18 ms by postulating a maximum packet size of 512+9 bytes (header length = 9).

One might expect a similar procedure for receiving a packet. However, we chose a different scheme in recognition of the fact that sending and receiving are not entirely symmetric pictures of the same action. In particular, transmission of a packet must be regarded as a single action in which both the sending and receiving stations participate synchronously. The timing is dictated by the sender. As a consequence, the

receiver's attention must be evoked by an interrupt and the packet stored in a buffer. It is helpful to regard the interrupt handler as an extension of the transmitter process. Typically, the buffer is handled as a cyclic store without reflection of a packet structure, and it is the agent effectively decoupling the events of sending and ultimate receiving. Bytes can therefore be retrieved individually from the buffer without timing constraints.

A read procedure with an array parameter as packet destination would then have to move the data from the (hidden) cyclic buffer into the parameter. This would in many cases be an unnecessary action and constitute an avoidable overhead. The logical solution is to provide byte-wise access to the received data sequence, thereby avoiding another buffer and also leaving the choice of handling errors to the program receiving the data. The following set of procedures is provided for receiving:

```
PROCEDURE ReceiveHead(VAR head: Header);
PROCEDURE Receive(VAR x: BYTE);
PROCEDURE Available( ): INTEGER;
PROCEDURE Skip(n: INTEGER);
```

After the call ReceiveHead(h), *h.valid* means "a packet has been received and h is its header". *Receive(x) is* called to obtain the individual bytes of the data part, whose number is given by *h.len. Skip(n)* may be used to rapidly skip over unwanted packets.

A few details may be worth recording about the implementation of the SCC interface. Before sending, it must be established that the line is free. This is done by inspecting the hunt bit of the receiver status. If the line is busy, sampling is repeated after a certain delay. By making the delay station-dependent, the danger of collision is reduced, although not eliminated.

The sending process itself occurs after forcing the station's address into the *sadr* field and after barring interrupts, and it is highly time-critical. For each byte, the loading into the transmitter data buffer is preceded by polling the buffer-empty status bit. After sending the data part, the transmitter driver must remain enabled for about 0.2 ms in order to allow the SCC to transmit the CRC and the terminating flag (undocumented feature).

The first byte received after detecting the leading flag is compared by the SCC with the station's address. If they match, the processor is interrupted. This filtering process is essential to reduce the number of interrupts to those pertaining to the actually addressed station. (One address bypasses the filter and serves for broadcasts.) The most time critical part is the interrupt; the bytes following the address byte must be picked up in time in order to avoid the data overrun condition.

The end of a packet is detected when the end-of-frame bit of the receiver status is set for about 20 us. Then the CRC and overrun status bits are checked. If either an error is indicated, or if the packet would overflow the buffer, the buffer is reset and the packet is effectively ignored.

The SCC interface does neither interpret the source address, the type, nor the link fields of the header. From a logical point of view, the definition of those fields belongs to a higher level of the abstraction hierarchy. Also, it is worth noting that at this level no communication protocol is defined, packets are received (and possibly ignored) without acknowledgement. The entire driver module, including its time-critical parts, is programmed in the language Oberon extended by a statement to access device registers and a facility to install a procedure as an interrupt handler.

A facility for file transfer

When planning a remote access facility, one must consider the metaphor of the operating system in which the new function has to be embedded. The metaphor of the operating system Oberon [3] is that of a single process, sequentially interpreting (dialog-free) commands issued by its user. We now relax this scheme in the sense that commands may originate from more than one source, say, the mouse, the keyboard, and the *network.* This is realized by not only polling the mouse and the keyboard in Oberon's central loop, but by also allowing the installation of handlers that become part of the central loop and poll their input source, in this case the SCC's buffer. Remotely requested actions can thus be inserted in

the sequence of actions ordered by the "regular" operator.

The appropriate view of interaction over the net is thus the remote procedure call [4] . It supposedly can be considered like a regular procedure call (without the possibility of reference parameters), and is implemented by sending the procedure's identification and its (value) parameters, followed by receiving an acknowledgement with the remotely computed result. This concept fits the Oberon metaphor ideally, because each call constitutes an isolated action and does not presume a specific system state. Neither does the remote procedure call require a protocol, simply because protocols inherently are specifications of legal *sequences* of information exchanges and ensuing actions.

If we employ the remote procedure call metaphor to a file transfer facility, the individual commands' parameters may become rather voluminous: they are files. In particular, the length of a file may easily exceed the maximum length of a data packet. The individual call must obviously be broken up into a sequence of calls each carrying a piece of the file as its parameter. In order to synchronize sender and receiver, each call is acknowledged individually.

The information exchanged over the net evidently must obey a simple protocol. Since the communicating partners contribute information in strict alteration (without concurrency), their protocol can be specified by a syntax using Extended BNF–notation. We distinguish the contributions of the partners by italizising those of the requestor. A transaction for obtaining a file has the syntax

ReceiveTransaction = *SND username password filename*
(DAT0 data *ACK*1 {DATi data *ACK*i+1} | NAK | NPR).

The symbols SND, DAT, NAK, and NPR are transmitted in the type field of the packet headers; username, password, filename, and data are the parameters transmitted as the packet's data part. DAT and ACK are encoded as the packet's sequence number (modulo 8) and enable a check against missing packets. NAK signifies that the requested file does not exist, NPR that the requestor is not permitted to receive the file.

Note that the acknowledgement for the data packet i carries the number i+1, being at the same time the request for the next packet. With the exception of the last, an acknowledgement is simultaneously the request for the next data packet.

Analogously, a transaction for dispatching a file follows the syntax:

SendTransaction = *REC username password filename*
(ACK0 *DAT0 data* ACK1 {*DATi data* ACKi+1} | NAK | NPR).

In this case the partner receiving the request (sometimes called the target) effectively becomes the master: it requests a data packet by sending an ACK packet with the appropriate sequence number.

In a facility based purely on the remote procedure call metaphor, a failure to transmit a packet simply causes no answer to arrive at the requestor, which then may choose to repeat the request. In the case where the transaction is broken up into a sequence of transmissions, the failure of an individual transmission must be handled by the network facility through retransmission. We recall that collisions, although rare, lead to such failures: the packet in question does not appear to have arrived at the receiver (i.e. the detection of a CRC error caused the driver to suppress the packet).

We here follow the principle that a retransmission occurs *only if requested.* Hence, the receiver of the data will have to detect possible failure. This occurs through a timeout, i.e. a retry request is sent after a time span T0 during which no data were received.

```
PROCEDURE ReceiveData(F: Files.File; VAR done: BOOLEAN);
    VAR seqno, retry: INTEGER;
BEGIN seqno := 0; retry := 2;
    (*assume first data packet received*)
    LOOP
        IF head.typ = seqno THEN
            INC(seqno); retry := 2;
            Send(ACKseqno);
```

```
            Receive bytes and write file F;
            IF end THEN done := TRUE; EXIT END
        ELSE DEC(retry);
            IF retry = 0 THEN done := FALSE; EXIT END;
            Send(ACKseqno)
        END ;
        ReceiveHead(T0)
    END
END ReceiveData;
```

An acknowledgement, and thereby the request for the next packet, is emitted after receiving the head, but before transferring the respective data from the receiver buffer to the file. This makes it possible that the sender may read the next packet from the file into its transmitter buffer at the same time as the receiver transfers the previous packet to the file, hence doubling the overall transmission speed. This scheme relies on the assumption that producing a packet (reading from disk) and consuming a packet (writing on disk) take about the same amount of time.

```
PROCEDURE SendData(F: Files.File);
    VAR seqno: INTEGER;
        buf: ARRAY N OF BYTE;
BEGIN  seqno := 0;
    LOOP Read  from file into buffer;
        REPEAT Send(seqno, buf); ReceiveHead(T1)
        UNTIL head.typ # ACKseqno;
        INC(seqno);
        IF head.typ # ACKseqno THEN (*failure*) EXIT END;
        IF end THEN EXIT END
    END
END SendData;
```

The choice of the timeout values T0 and T1 depends on the expected time for a transmission (yielding a lower bound) and the actions of the partner that one wishes to allow before emitting the acknowledgement (yielding an upper bound). The value T1 should allow for the specified number R of retries of the receiver, before the sender abandons the interaction, i.e. T1 = R*T0.

A second complication due to the breaking up of a transaction into parts is the need to reserve the partner exclusively to the ongoing process. This implies that – in addition to the destination filter in the SCC – a source filter must be provided, which eliminates packets arriving from other sources. In our implementation, this filter is contained in the procedure *ReceiveHead*, which in turn calls *SCC.ReceiveHead*.

A simple, distributed name service

Individual stations in the network are characterized by a unique number. Evidently, a partner should be addressable by a name rather than a machine number. The following method allows to determine the address of a machine given its name without requiring a centralized name directory.

If, before a transaction, the partner's station number is unknown, a *name request* is broadcast, i.e. sent to all partners on the net (packet type = NRQ). A special address value is provided for this purpose which passes all address filters. The packet's parameter is the desired partner's name. Upon receiving a name request with a name matching the receiver's name, the receiver emits a response packet (NRS); its machine number is contained in the header's *sAdr* field.

NameTransaction = *NRQ name* [NRS].

In order to reduce the traffic due to name requests, each station may carry a table of name/number pairs established by name requests (a cache). In practice, it turns out that a single entry is quite sufficient, containing the identification of the last partner.

Integration of the server concept in the Oberon metaphor

The essence of the Oberon metaphor is the absence of concurrent processes (with the exception of interrupt-driven buffer handlers which are transparent to users). This apparently excludes the presence of a server process, unless the server is the only process in existence.

At the core of the Oberon system lies the central-loop, in which input devices – keyboard and mouse – are polled. If an input event is detected, control passes to an appropriate algorithm through calling an installed procedure in an object designated by either cursor or caret position. Additional procedures may, however, be installed in *handlers* which are also activated each time control passes through the central loop. Servers may thereby be included in the system without compromising the rule that individual actions are logically uninterruptible. Response time cannot be guaranteed, however, and we refrain from introducting even priorities among the input devices signalling events: the network simply becomes an event source in additon to keyboard and mouse. All time-critical actions are confined to hardware-driven interrupt handlers, which effectively decouple partners through their data buffers. The lack of arbitrary interruptability and of priorities is compensated by an increase in efficiency due to the absence of process switching (i.e. diversions of attention), and by a decrease of complexity because of absence of mutual locking mechanisms.

The handler for the file transfer facilities described follows the protocol syntax:

```
Transaction = {SendTransaction | ReceiveTransaction | NameTransaction} .

PROCEDURE Serve;
    VAR head: SCC.Header;
BEGIN SCC.ReceiveHead(head);
    IF head.valid THEN
        IF head.typ = SND THEN ... SendData ...
        ELSIF head.typ = REC THEN ... ReceiveData ...
        ELSIF head.typ = NRQ THEN
            IF receivedName = MyName THEN Send(NRS) END
        ELSE SCC.Skip(head.len)
        END
    END
END Serve
```

A central print server

The scheme described so far lets every station act both as a requestor (master) and a target (server), and all stations feature the same capability; they form a homogeneous society. In a typical laboratory or office environment it is, however, sensible to centralize certain services. For example, a station may be designated to serve as a file store for common system modules and/or data files, available to anyone at any time. It is probably economical to install a central printer fed by one machine but available to all through the network. A third example is a server storing and forwarding electronic mail.

The straight-forward solution to implement additional services in a distinguished station is to extend both the procedure *Serve* and the list of possible requests. In the case of requests for "small actions" such as passing a short message, indicating a central time and date, delivering a directory of a mailbox, etc., the straight-forward solution is not only simple but also quite adequate. In the case where the requested action is time-consuming, such as printing a page, it is inadequate. This is because under the Oberon metaphor the server's processor is not only unavailable when truly engaged in computing the print image, but also when waiting for the printer becoming ready for the next page. The acquisition of the processor for such lengthy time spans is clearly unacceptable, and fortunately also unnecessary.

The Oberon central printer server operates as follows: First, a requestor sends (i.e. requests the reception of) a file. This request characterises the file as a print file, and the (file) server accordingly inserts it into a queue instead of registering it in the general file directory. The (print) server, when recognizing the queue to be non-empty, starts processing the (next) element in the queue.

The queue effectively constitutes the interface between the two servers. A few additional global variables reflect the state of the printing activity, making it possible to release the processor while the printer processes the page. The variable *Pstat* assumes the following values, reflecting the successive phases of the printing activity:

0. ready for next printing task.
1. ready for processing next page.
   (reading file and computing print image)
2. ready for printing.
3. printing.

The processor is active in phase 1only, the printer in phase 3. Phase 2 extends in time, only if the printer is detected to be not ready (e.g. due to lack of paper). The server routine is now extended as sketched below:

```
PROCEDURE Serve;
BEGIN SCC.ReceiveHead(head);
    IF head.valid THEN (*serve net*)
        IF head.typ = SND THEN ...
        ...
        ELSIF head.typ = PRT THEN
            receive file and register it in print queue; INC(noftasks)
        ELSE SCC.Skip(head.len)
        END
    END ;
    IF noftasks > 0 THEN (*serve printer*)
        IF Pstat = 0 THEN (*ready for next print task*)
            IF printer ready THEN
                OpenFile(next in queue); Pstat := 1
            END
        ELSIF Pstat = 1 THEN (*ready for generating next page*)
            IF end of file THEN
                remove queue element; DEC(noftasks); Pstat := 0
            ELSE
                Read file and compute image up to start of next page;
                Set nofcopies; Pstat := 2
            ENDe
        ELSIF Pstat = 2 THEN (*ready for printing next page*)
            IF printer ready THEN Start printer; Pstat := 3 END
        ELSIF Pstat = 3 THEN (*printing*)
            IF printing done THEN
                DEC(nofcopies);
                IF nofcopies > 0 THEN Pstat := 2 ELSE Pstat := 1; ClearPage END
            END
        END
    END ;
    Other Servers
END Serve
```

The global variables needed to record the print server's state are (in addition to *Pstat* and the queue of print tasks) *nofcopies* and a file rider specifing the position up to which the file had been processed.

We emphasize that this simple scheme allows for quasi-concurrent file transfer and print image computation. Naturally, a file transfer can only take place while the print task is in phase 1 or phase 3. As the computation of a typical page takes less than a second, this constitutes no serious handicap.

Conclusions

The main purpose of the Ceres–Net project was to demonstrate that a reliable, simple–to–use network can be constructed at very low expense, if one is willing to concentrate on essential functions and to dispense with features and performance seldom used and rarely needed. The term low–cost is not restricted to material and financial aspects, but also to structure and complexity. A system based on perspicuous principles and justifyable features is easier to explain and understand, and hence needs no so–called maintenance. Extensions are feasible without undue complications and the danger of inadvertently impinging on the existing facilities.

The use of a relatively low–speed transmission line may surprise the reader. It was chosen because of the relative simplicity of the required hardware, and principally because it is evident that in an organization of moderate size equipped with powerful workstations using their own disk–stores, network utilization is quite low. It is virtually restricted to communication with the printer, the file distribution server, and the electronic mailer. Hence, the use of a more expensive transmission system would not be justified. A gain in actual data transfer speed would hardly materialize, because it is limited by the speed of file reading and writing. The currently used net is a good match, as shown by the following performance figures:

	Ceres–1	Ceres–2	
	(32032, 10MHz)	(32532, 25 MHz)	
File reading/writing			
in blocks of 512 bytes	15.2	8.1	s/MByte
byte–wise	80.0	24.0	s/MByte
Transmission of data stream (512 byte packets)	36.5		s/MByte
Transmission of files	115.2	75.5	s/MByte

The modular structure of the network software is shown in Fig. 5. A measure of its low complexity is evident from the following figures.

Module	lines of source code	bytes of object code
Net	315	3060
Server	400	4212
SCC	151	1108

(Note: *Server* is the version of *Net* extended by the printer facility.)

A central point of interest was also the integration of a network facility in the Oberon operating concept, which dispenses with the notion of multiprocessing, where task switching is left under the explicit control of the user issuing commands. Although the use of a network inherently involves several processors, the notion of conceptually uninterruptible commands proved to be useful and appropriate even in this case. The resulting simplification of the implemention is considerable. Our experience teaches that the genuine need for multiprocessing, i.e. for the possibility to freely spawn new processes and to switch the processor at arbitrary moments, should be carefully justified before that facility is introduced. If the need is not clearly established, multiprocessing may well belong to the problem set rather than the solution set. Inflating a system with rarely used features is easy and common practice. Optimizing the ratio of utility and reliability vs. required resources and complexity is much harder, but it remains the ultimate challenge of every design engineer.

References

1. H. Eberle. *Development and Analysis of a Workstation Computer.* Diss. ETH No. 8431 (Dec. 1987).

2. A. West and A. Davison. *CNet – A Cheap Network for Distributed Computing.* TR–120, Computer Systems Laboratory, Queen Mary College (1978).

3. N. Wirth and J. Gutknecht. The Oberon System. *Software – Practice and Experience, 19.*

4. B.J. Nelson. *Remote Procedure Call.* Report CLS-81-9. Xerox Palo Alto Research Center (1981).

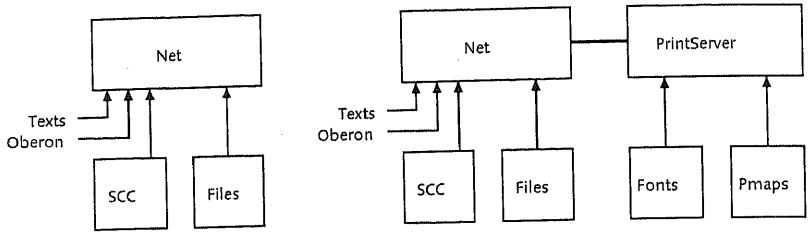

Fig. 5. Module Structures of Net and Servers

Extending Ceres–Net by a Mail Service

N. Wirth

1. Concept and Background

An important component in a modern, distributed computer system is the electronic mail server. The Ceres–Net service as described in [1] caters for a file distribution and a printing service. However, the server was designed to be easily extensible. Here we describe the addition of a mail server.

In the list of requirements, priority was given to quick response and robustness. This dictates a simple structure of mailboxes and a restriction to essential and fundamental functions. The complexity of the programs to send and recieve messages located in every workstation is minimized by integrating mail operations in the existing net service module and by the transmission of messages as plain texts. Thereby they automatically become editable objects in the Oberon System [2] and it is possible to reduce the set of mail commands to four functions only: Sending a text, receiving the mailbox directory, receiving a message selected from the displayed directory, and deleting a selected message. We consciously refrained from introducing any further bells and whistles.

The first step in the course of development was the addition of access to the existing (file and printer) server over a low–speed V24 (RS232) line, thereby bringing the server within reach of remote users via telephone connection (via the University–wide broadband network Kometh and central telephone interconnects). The same V24 line is used by the server, if messages are to be exchanged with the Department's central mail server; we call such messages "external" mail, in distinction to messages sent *directly* to the Ceres–Server (via Ceres–Net or telephone line), which are called *internal* mail. The global configuration of components is shown in Fig. 1.

Therefore, we first describe the extension of the system by the V24–line server. Then we discuss the addition of the internal mail service, and conclude with the description of the extension to cater for external mail.

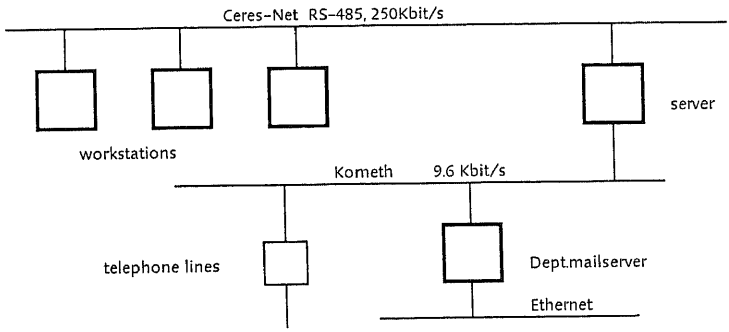

Fig.1. Network Configuration

2. The Line Server

The basic Ceres server system is shown in Fig. 2. The dark lines exhibit the additions of the print service facilities to the original system designed as file server only.

The additon of a server in the Oberon system implies the installation of a task. Such a task is called each time control circulates in the central loop. For each such activation, the task polls its command source (either a device such as the net or line interface, or a queue) and processes the request, if any is pending. Each installed server may hence be regarded as an interpreter of remote procedure calls.

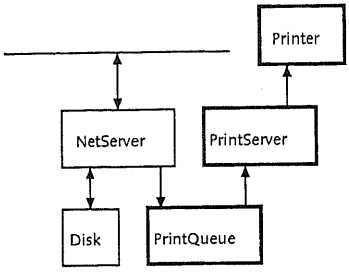

Fig. 2. Net- and Print Server Configuration

The configuration resulting from the addition of the Line Server is shown in Fig. 3. It suggests a strong similarity of the Net- and Line Servers which is supported by the fact that both servers offer the same functions of file transfer. The differences lie at the level of data transport.

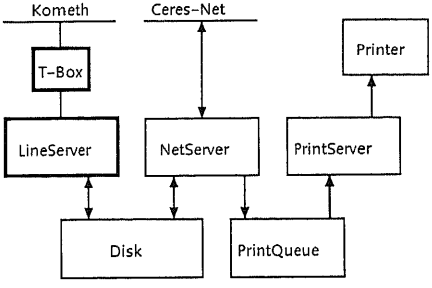

Fig. 3. Net-, Line-, Print Servers

The line is not only slower than the net connection by a factor of 25, it is also potentially much less secure, since it may involve telephone lines over wide distances. The basic, asynchronous mode of transmitting individual characters is therefore inapplicable. A packet mode is recommended, and this requires the definition of a packet format. The format used here was defined by M. Müller (for purposes ranging beyond the Ceres server connection) and is described as follows:

1. A stream of bytes is broken up into packets of at most 256 bytes; the last packet is identified by its length being less than 256.

2. Each packet starts with a header byte containing, apart from packet sequence numbers, a type tag. The following packet types are provided:

data packet data acknowledge

open packet	open acknowledge
close packet	close acknowledge
abort packet	

The packet types suggest the following protocol:

Transaction = Open OpenAck {Data DataAck} Close CloseAck.

Failures are, at this level, not indicated by specific NAK packets, but manifest themselves by timeouts or incorrect sequence numbers, in which case a retry is initiated by the sender.

Individual commands are encoded in (the data following the header byte of) the open packet. (In the Net Server, the commands are identified by the packet type tag itself.) In analogy to the Net Server, there exist the following commands with parameters as shown below:

SND username password filename
REC username password filename

The partner's reply follows not in its OpenAck packet (which is supposedly invisible at this level), but in the subsequent data packet, which consist either of a (single character) acknowledgement followed by further packets representing the requested data, or of a (single character) negative acknowledgement (for example, if the requested file does not exist). This, in comparison with the Net Server's protocol, increased complexity is due to adoption of the multilevel protocol dictated by the ISO Standard. Whereas in the case of a request for a non-existing file, the Net Server exchanges exactly 2 packets (SND, NAK), the Line Server exchanges six packets (Open, OpenAck, Data, DataAck, Close, CloseAck).

Considering the required robustness and the potential unreliability of lines over long distances, packets need to be checkable for correctness. For this purpose, they are encoded for actual transmission. First, we take into account that some data links interpret rather than transmit certain ASCII characters; these must be avoided as data. Hence, groups of 3 bytes are transmitted as 4 6-bit items, each encoded as an 8-bit byte in a way to avoid control character values. Second, a checksum is computed over the encoded data and appended. The precise encoding is specified in the Appendix. The maximal length of a transmitted packet is therefore not 256, but, including an additional header, 345 bytes. The effective data rate is 3/4 of the actual transmission speed. This is evidently the price to be paid for compliance with outdated but unremovable Standards.

The multilevel structure of the protocol definition is mirrored by the procedures to send and receive a data stream. In order to allow the reader a comparison with the scheme for the net [1], whose protocol design was under local control, the relevant procedures are listed below. As in the case of the net, procedure *SendData* is called after receiving a Send command and assurance that the file exists.

```
PROCEDEDURE SendData(F: Files.File; VAR res: INTEGER);
    VAR k: INTEGER; x: CHAR;
        R: Files.Rider;
        buf: ARRAY PakSize+2 OF CHAR:
BEGIN Files.Set(R, F, 0);
    LOOP k := 0;
        LOOP Files.Read(R, x);
            IF R.eof THEN EXIT END;
            buf[k] := x; INC(k)
        END;
        Send1(k, buf, res);
        IF(res # 0) OR (k < PakSize) THEN EXIT END;
    END
END SendData
```

Procedure *Send1* covers failures and retries:

```
PROCEDURE Send1(len: INTEGER; VAR buf: ARRAY OF CHAR; VAR res: INTEGER);
    VAR retries, typ, plen; INTEGER;
BEGIN mysno := 1-mysno; retries := 3;
    SendPacket(myrno*2+mysno+10H, len, buf);
    LOOP ReceivePacket(typ, plen);
        IF typ <= 0 THEN (*error*) DEC(retries);
            IF retries = 0 THEN res := 1; EXIT END;
            SendPacket(myrno*2+mysno+10H, len, buf)
        ELSIF typ DIV 10H = 7 THEN (*abort*) res := 2; EXIT
        ELSIF typ DIV 2 MOD 2 = mysno THEN res := 0; EXIT
        END
    END
END Send1
```

Procedure *SendPacket* performs the 8-to-6 bit/byte encoding, and procedure *ReceivePacket* the corresponding decoding. The first parameter indicates the packet type and whether a timeout or a checksum error had occurred (typ = 0). We note that, in contrast to the scheme used in the net server, the sender repeats transmission in the case where no acknowledgement is received (timeout). This may lead to confusion, if the receiver sends its acknowledgement too late (after the sender's timeout), and this possibility calls for a countermeasure. It is realized in terms of sequence numbers carried by each packet (in the header byte), one reporting the sender's, the other the receiver's current count represented by the (global) variables *mysno* and *myrno*. If suffices to use numbers modulo 2.

3. Internal Mail

Primarily, we expect an electronic mail system to be simple and reliable to use, and that service should be fast, practically instantaneous. We gladly accept the absence of enhancements offered to achieve so-called user-friendliness. This convenience must rather stem from a proper integration of mail service in the workstations environment, in particular its text editor. The Oberon system offers ideal conditions to realize this integration. Messages and directories appear as texts, and are thus editable and mergeable with all other texts. A welcome consequence is that the mail system can be restricted to perform the actions of pure transmission. The following four commands are made available on every workstation:

1. *Net.SendMail.* This command typically appears in a tool text and can then be activated by a single mouse click. The message to be sent is supposed to be displayed on the screen and is selected by placing Oberon's marker in the viewer containing the message text.

2. *Net.Mailbox.* Activating this command, which typically also appears in the net tool text, causes the directory to be fetched and displayed in a new viewer as a text. Each line corresponds to a message stored in the box, lists the name of the sender, and date and time of arrival.

3. *Net.ReceiveMail.* This command, displayed in the menu of the viewer showing the directory, fetches the message previously selected in the listed directory. The message is displayed in a new viewer.

4. *Net.DeleteMail.* This command serves to remove the message selected in the listed directory from the mailbox.

These commands can be regarded as remote procedure calls to the server, which retains no state influencing the interpretation of any subsequent command. A message to be sent must be headed by one or more lines specifying recipients. This is expressed by the following message syntax:

```
message =   recipient {recipient} messagetext.
recipient = "To:" name CR.
name =      letter {letter | digit | "."}.
```

3.1. Command protocol

The four mail service commands are added to the set of existing commands for file transfer and printing. Their detailed encoding differs slightly between net and line transmission for reasons indicated previously, and it is specified in the Appendix. The respective protocols are as follows (shown for the net):

SendMail:	*RML username,password* (ACK {*data* ACK} | NAK | NPR).
Mailbox:	*DIR username password* (ACK {data *ACK*} | NAK | NPR).
ReceiveMail:	*SML username password msgno* (ACK {data *ACK*} | NAK | NPR).
DeleteMail:	*DML username password msgno* (ACK | NAK | NPR).

The packets sent by the master (workstation) are printed in italics. NPR denotes a negative acknowledgement due to rejection of access permission. In the case of line transmission, the command identification (first packet) is transmitted as parameter of the open packet (service request).

3.2. Mailbox Structure

The implementation of the mailbox is determined by the following considerations:

- Robustness. In order to avoid loss of data in the case of equipment failure, all mail data are to be stored on a non–volatile medium (disk). The data associated with different users are to be appropriately disjoint.
- Speed. Access to a message must be fast, and is therefore restricted to serial reading of the entire message. In particular, a directory access is to be effectively instantaneous, and must therefore involve no access to individual message bodies. As a consequence, the directory is to be stored contiguously as a whole.
- The mailbox is considered as a temporary storage relay of messages on their way, and not as a permanent message archive. A limited, even small number of message slots, and a limited, relatively small storage capacity for message bodies is therefore acceptable.

From these premisses it was concluded, that the simplest and most effective way to represent the mailbox data is to associate with each user a single file. Its directory is contained as a header of fixed length which fits into the file's first two disk sectors. The directory has the following data format:

```
MailDir =   ARRAY 31 OF MailEntry;
MailEntry = RECORD pos, next: INTEGER;
                   len: LONGINT;
                   time, date: INTEGER;
                   originator: ARRAY 20 OF CHAR
            END
```

The subsequent part of the mail file is considered to consist of blocks (of 256 bytes), and each message occupies an integral number of blocks. Analysis of usage reveals that most messages require very few blocks only. Block occupation is recorded in the block reservation table. Accepting a maximal file length of 64k bytes, the table consists of 256 bits (8 words) and is placed at the very beginning of the file (in front of the directory). The index of the first block of a message is given by the *pos* field in its directory entry, and the message occupies a number of contiguous, adjacent blocks. The field *next* is used to link the directory entries in the order of message arrival.

3.3. System Structure

The foregoing explanations suggest to incorporate the four mail commands in the existing net and line server modules, in particular because they are interpreted as remote procedure calls and give no cause for the introduction of new processes. Due to the straightforward organization of the mailbox, the routines interpreting the commands are conveniently placed in those modules. The same holds for the programs residing in the workstations: The four mail routines are easily incorporated in the modules implementing file transfer.

The only reservation applies to the *SendMail* command. Not only is insertion of a new message somewhat

more complicated than its localization and reading, but it may involve distribution of a message to several, even many or all mailbox owners. A decoupling in time of message reception and distribution (called dispatching) appears as recommended. It implies the introduction of an additional server process (called *MailServer*) which is fed by a queue of tasks (called *MailQueue*), and whose only function is the dispatching of arrived messages. The resulting system structure is shown in Fig. 4.

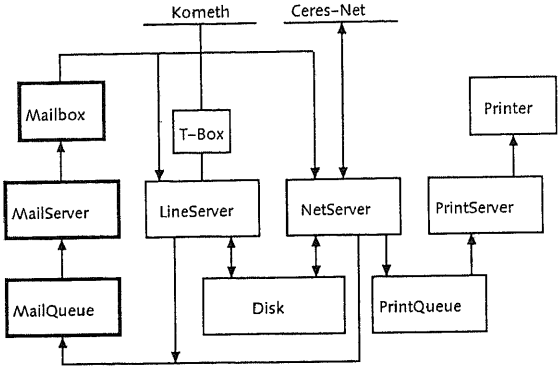

Fig. 4. Net-, Print-, Line-, and Mail Server

The mail queue has exactly the same structure as the print queue. It is a circular buffer of tasks, namely messages represented as text files to be dispatched.

```
Task =   RECORD file: Files.File;
           userno: INTEGER
         END
```

As an aside, we note that print- and mail tasks are temporary files. They are not registered in the file directory, and are purged after processing.

3.4. Access Permission

Every command is accompanied by an identification of the user (id) and a password (pw). The identification is an abbreviation, typically consisting of 2 or 3 letters, the password is encoded as a (long) integer. The server stores a table of id/pw pairs which allows to check the acceptability of a command.

In the case of mail service, abbreviated names are a (widespread) nuisance, because the sender must remember these codes. We instead impose that each user be addressed by his (her) full last name, and the table accordingly also registers that name (up to 20 characters). The table may also serve to hold accounting data.

4. External Mail

The last step in the sequence of extensions introduces the possibility to send messages to and to receive them from an external source, a central mail server. Since that source is accessed over the V24 (RS232) serial line, only the line- and the mail servers are affected.

Receiving messages from the external server proceeds in principle like receiving messages from a client; the message is inserted in the mail queue. However, the central server adheres to a Standard format for messages, in fact a subset of X400 called CX400. Hence, these messages must be decoded, in particular their header must be analysed. Apart from this, dispatching occurs like in the case of internal messages.

Sending messages to the external server requires that they be encoded according to CX400. Also, the server must now be able to act as a master, opening the connection over the line. As the line is relatively slow, and in particular because the line may be occupied, or because the external server may be (temporarily) unavailable, a decoupling in time of dispatching and transmission is indispensible. The consequence is the introduction of a third queue containing encoded messages to be exported to the external server. We call it *ExportQueue*, and it has exactly the same structure as the other queues. The resulting system structure is shown in Fig. 5, and the CX400 message encoding is specified in the Appendix.

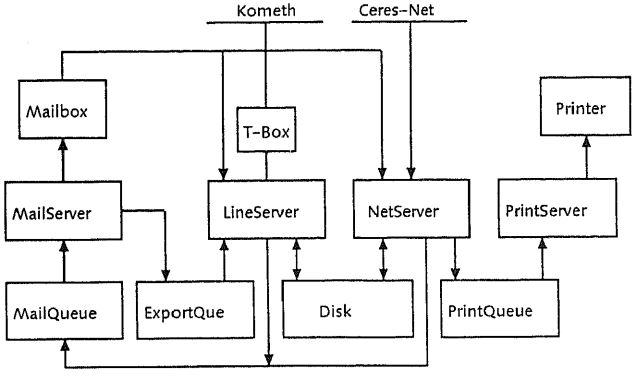

Fig. 5. Servers including External Mail

When dispatching a message, an external recipient is recognized by the MailServer when the recipient's name is followed by an address. Furthermore, a subject may be indicated which is recorded in the CX400 message header. The extended message syntax is as follows:

message = recipient {recipient} [subject] messagetext.
recipient = "To:" name ["@" address] CR.
address = {character}.
subject = "Re:" {character} CR.

Internal recipients cause a direct dispatch into the user's mailbox. External recipients cause encoding and insertion into the ExportQueue. (If several recipients contain an address, the message is forwarded once and carries all recipients in the encoded header.)

4.1. The External Mail Protocol

Establishing a connection between the Ceres server and the external server may be a lengthy operation. It is therefore advisable to seek connection in certain intervals of time only, and then not only to transmit, but to *exchange* accumulated mail. This is done according to the following protocol in which italics denote entities sent by the partner which established the connection (the master).

Exchange = *Open {Msg} Term* {Msg} Term *Close.*
Msg = Data {Data} Ack.

– *Msg* represents a mail message and consists of a sequence of non–empty data packets. Messages to be sent are taken from the export queue, messages received are inserted into the mail queue.

– *Ack* is a single (data) packet with length 0 (not to be confused with the DataAck packets following each data packet at the lower transport level). Each message is individually acknowledged at this level, in order to let the sender discard the transmitted message.

– *Term* is a single (data) packet with length 0 and signals that all messages had been transmitted.

– The *Open* packet specifies the mail exchange request and the requestor's identity.

If the external mail server requests an exchange, the request arrives through the line like any other request for service and is identified by its cx400 parameter. If the need for an exchange is recognised by the Ceres Server (export queue non–empty), the line connection must first be established. This process is fairly complicated and requires further explanation.

4.2. Establishing the Connection between the Servers

The server is tied to the univerisity–wide broad–band net, which offers individual 9600 bps lines via a tap called T–Box. The box is either in command mode or in data mode. In the former it accepts commands, such as "call nn", where nn is a port number. These commands are sequences of ASCII characters terminated with CR and evoke replies also consisting of characters. This scheme is suitable for text terminals operated by human operators, but ill–suited for computer–driven ports. Successful connection causes a reply ("call completed to nn") to appear at the partner's T–Box, whereupon both T–Boxes switch to data mode representing a transparent data line. The fact that a text rather than an encoded packet appears when a connection is established by a partner, requires that plain text must be "swallowed" by the packet receiver routine.

In passing, we note that the T–Box can be put into a quiet mode, where replies to commands are completely suppressed. Unfortunately, this includes replies of commands issued directly, such as "call". Since it is necessary to receive a reply in this case, the quiet mode cannot be used. In our particular case, the process of connecting is even more complicated, because the line leads through a gateway that must also be requested to grant a link. A mail exchange therefore requires three levels of "opening":

– T–Box: connect to gateway via Kometh,
– Gateway: connect through to mail server via Ethernet,
– Server: open link to mail service.

In order to keep the number of such connection processes within reasonable bounds – the Ceres server is blocked during this time – a clock (mailtime) records the time of the last exchange. The next connection is initiated after a message has been stored in the export queue and 5 minutes have elapsed since the last exchange.

If a user is already connected to the server via a (Kometh) line, the initiation of a mail exchange must be delayed. Otherwise the various opening commands would be transmitted as data to that user, disturbing communication. Therefore, a state must be associated with the line server (*Lstat*). The state indicates whether the line is active (in use) or not. The active state is assumed as soon as a request is received or initiated (open packet), the inactive state is assumed after receipt or dispatch of a close packet. Furthermore, a global variable *Lmode* records whether the line server is acting as a genuine server for the line, ready to accept requests – this is the normal mode – or whether it acts as the master, having initiated a mail exchange. This information determines the lead character of packets which serves to let the receiver determine whether a received packet came from the (correctly functioning) partner, or was a reflection (by an incorrectly operating T–Box or gateway) of its own last sent packet.

5. In Retrospect

This paper describes a project which is a good example of a modular design, progressing through a sequence of steps of successive extensions. The result is a system of surprisingly high funtionality in respect to its small size (see Appendix). Its efficiency and robustness is a direct benefit of its smallness, and the latter is the result of restriction to essential functions. During its development, the primary importance of the choice of the right basis for every successful modularization and extensibility became abundantly clear. It was also recognized that there is a significant difference between a system that is merely efficiently functional, and one that is usable. The latter must in addition be reliable and robust. This requirement is particularly essential for a server system, because it does not operate under constant

human control. The most remarkable activity in this respect is the establishment of a link between two servers. If it fails, attempts must be automatically repeated, and failure must certainly not infringe upon other activities.

A third fact also became evident: Only proper integration with existing facilities can lead to a maximal ratio between added functionality and added complexity. With regard to the described system, this implies for example that no mail service should have to cater for editing facilities (such as e.g. exchanging originator's and recipient's addresses). This is clearly an operation to be performed through existing editing facilities which, therefore, must be general and flexible.

And last but not least: a structured language is the best tool to enforce proper structure to the last detail. The entire server system described here is formulated in the language Oberon, including driver modules for the network and the line interfaces. Without the use of a strongly typed language and the aid of a watertight checker in the form of a compiler, the development of this system would have taken a multiple of the time and effort spent.

The use of a high-level language was in no way a hinderance to master certain real-time constraints inherent in every server system. On the one hand, there exist a few real-time constraints in the micro- and millisecond range. They are well isolated in the driver module SCC [1]. On the other hand, some real-time requirements in the range of seconds (timeouts) and even minutes are handled easily due to the availability of a system-wide timer that is permanently operating (resolution: 3ms), and whose value can be examined with a simple procedure call.

The entire file transfer, printer, and local mail server complex was completed by the author and programmer in a surprisingly short time. In contrast, the length of time taken for the implementation of the external mail service was rather disturbing. Development was fast for all those parts which were under complete control of the designer, and which therefore displayed appropriate structure and were fine-tuned to their purpose. In contrast, development was tedious – even extremely tedious – for those parts that had to be twisted to comply with existing components and imposed Standards. In many cases such development could not proceed according to widely heralded principles of orderly program design, but only by searching through "distributed documentation" and subsequent trial and error. One is tempted to conclude that existing, over-generalized standards and universally configurable software are the principal ferments of unprofessional work. It appears that the computer communications world is particularly plagued by its own artefacts and idiosyncrasies.

References

1. N. Wirth. Ceres-Net: A low-cost Local Area Computer Network. Companion Report. (to appear in *Software, Practice and Experience, 20*)

2. N. Wirth and J. Gutknecht. The Oberon System. *Software, Practice and Experience, 19*, 9 (Sept. 1989).

Appendix

1. Net Commands

The following commands are contained in module *Net* which is available on every workstation:

Net.SendFile partner file0 file1 ... ~

The first parameter identifies the partner to which the files are sent. The partner must have removed its write protection.

Net.ReceiveFile partner file0 file1 ... ~

Net.SendMail

The text contained in the marked viewer is dispatched to the mail server. In order to be acceptable as a message, the text must start with at least one address line of the form

To: *name address*

The name consists of letters, digits, and periods (no blanks). The address is optional. If omitted, the message is local. An address starts with the character @ (or %, or !). Each recipient is listed on a separate line starting with "To:". A subject may be indicated, headed by "Re:", also on one line.

Net.Mailbox

A viewer is opened displaying the mailbox directory fetched from the mail server. Each line contains a message number, the sender's name, and date, time, and length of the message.

Net.ReceiveMail

This command is taken from the mailbox viewer's menu after selecting a message in the same viewer (in order to select the line: double click). The message indicated in the selected line is fetched and displayed in a new viewer.

Net.DeleteMail

This command is taken from the mailbox viewer's menu after selecting a message in the same viewer. The selected message is then removed from the mailbox.

Net.GetTime name

The time and date is fetched from the specified server, and the workstation's time and date are set to the fetched values.

Net.SetPassword name password

The new password (specified as a string within quotes) is registered in the specified server.

Net.SendMsg name message

The message is sent to the named partner and displayed in the recipient's log text. The short message consists of any characters on one line.

Net.StartServer

This command enables the workstation to act as a server.

Net.Protect

The workstation is protected from receiving files by request from other stations (default mode).

Net.Unprotect

Net.StopServer

2. Line Commands

The following commands are contained in module *Line* which communicates over an RS-232 (V24) line:

Line.SendFile file0 file1 ... ~

The partner must have removed its write protection.

Line.ReceiveFile file0 file1 ... ~

Line.SendMail

The text contained in the marked viewer is dispatched to the mail server. (see *Net.SendMail*)

Line.Mailbox

A viewer is opened displaying the mailbox directory fetched from the mail server. Each line contains a message number, the sender's name, and date, time, and length of the message.

Line.ReceiveMail

This command is taken from the mailbox viewer's menu after selecting a message in the same viewer (in order to select the line: double click). The message indicated in the selected line is fetched and displayed in a new viewer.

Line.DeleteMail

This command is taken from the mailbox viewer's menu after selecting a message in the same viewer. The selected message is then removed from the mailbox.

Line.Start

Line.SendMsg

Sends the text specified on the same line. (Used to issue Kometh commands).

Line.Close

3. Server Commands

The following commands are contained in various server modules installed in a server station:

NetServer.Start	LineServer.Start	PrintServer.Start	MailServer.Start
NetServer.State	LineServer.State	PrintServer.State	MailServer.State
NetServer.Reset	LineServer.Reset	PrintServer.Reset	MailServer.Reset
NetServer.Stop	LineServer.Stop	PrintServer.Stop	MailServer.Stop
NetServer.SendFile	LineServer.StartTBox		
NetServer.ReceiveFile	LineServer.SendMsg		

The *Start* commands install the named server in the Oberon command interpretation loop, the *Stop* commands remove it. *State* indicates the state of the respective server and the number of tasks in its input queue. *Reset* clears the server's input queue.

4. Module Sizes

Module	lines of source code	Size of code (bytes)
Net	460	4600
Line	450	4700
NetServer	380	4800
LineServer	520	5900
PrintServer	150	1600
MailServer	300	4300
Core	100	1000
SCC	150	1100 + input buffer
V24	60	350 + input buffer
Printmaps	290 (assembler code)	

5. The Server's Module Structure

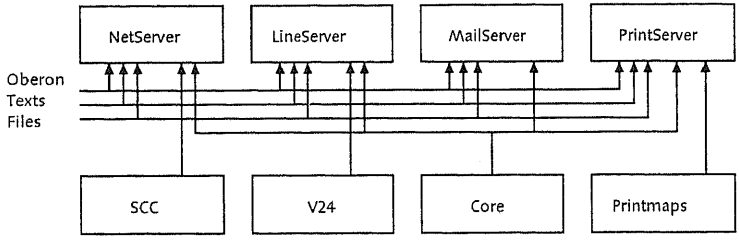

Fig. 6. Module Configuration

6. Interfaces of Modules SCC and V24.

The modules *SCC* and *V24* are used in workstations and server stations. They are the drivers of the respective net and line communication interfaces.

```
DEFINITION SCC;

    TYPE Header =
        RECORD valid: BOOLEAN; dadr, sadr, typ: SHORTINT;
            len: INTEGER; (*of data following header*)
            destLink, srcLink: INTEGER  (*link numbers*)
        END ;

    PROCEDURE Start(filter: BOOLEAN);
    PROCEDURE Send(VAR head, buf: ARRAY OF BYTE);
    PROCEDURE Available(): INTEGER;
    PROCEDURE ReceiveHead(VAR head: ARRAY OF BYTE);
    PROCEDURE Receive(VAR x: BYTE);
    PROCEDURE Skip(m: INTEGER);
    PROCEDURE Stop;
END SCC.


DEFINITION V24;
    (*interrupt-driven UART channel B, Signetics 2681*)

    PROCEDURE Start(CSR, MR2: CHAR);
    (* Clock Select Register: 66X: 1200 bps, 88X: 2400 bps, 0BBX: 9600 bps
        Mode Register 2: 7X: 1 stop bit, 0FX: 2 stop bits *)

    (*Output Port: 0 = DTR,  1 = RTS *)
    PROCEDURE SetOP(s: SET);
    PROCEDURE ClearOP(s: SET);

    (*Input Port: 0 = DCD, 1 = CTS, 2 = DSR*)
    PROCEDURE IP(n: INTEGER): BOOLEAN;

    (*Status Register. 0: Rx rdy, 2: Tx rdy, 4: overrun*)
    PROCEDURE SR(n: INTEGER): BOOLEAN;

    PROCEDURE Available(): INTEGER;
    PROCEDURE Receive(VAR x: BYTE);
    PROCEDURE Send(x: BYTE);

    PROCEDURE Break;
    PROCEDURE Stop;

END V24.
```

7. The Net and Line Protocol

In the following syntax, the initiator's packets are denoted in italics, those of the target in normal font. Codes written in capital letters represent a packet, words in lower case are parameters (data part).

Transaction =	SendFile | ReceiveFile | SendMail | Mailbox | ReceiveMail | DeleteMail | SendMsg | GetTime | SetPassword | NameRequest | PrintRequest.
SendFile =	*REC username password filename* OutTransaction.
ReceiveFile =	*SND username password filename* InTransaction.
SendMail =	*RML username password* OutTransaction.
Mailbox =	*DIR username password* InTransaction.
ReceiveMail =	*SML username password msgno* InTransaction.
DeleteMail =	*DML username password msgno* (ACK | NAK | NPR).
GetTime =	*TRQ* (TIM time date).
SendMsg =	*MSG string.*
SetPassword =	*SPW username password newpassword* (ACK | NAK | NPR).
PrintReq =	*PRT username password* OutTransaction.
MailTransfer =	*CX name* {*Data* {*Data*} ACK} *Term* {Data {Data} *ACK*} Term.
NameRequest=	*NRQ servername* [NRS].
InTransaction =	DAT_0 data ACK_1 {DAT_i data ACK_{i+1}} | NAK | NPR.
OutTransaction =	ACK_0 DAT_0 *data* ACK_1 {DAT_1 *data* ACK_{i+1}} | NAK | NPR.

In the Net protocol, the command is encoded as the packet's type. In the line protocol, all items shown appear as the data part of packets. The first packet, identifying the command (service request), is an open–packet (see Line Packet Format). All others are data–packets. Each transaction is terminated by an additional close–packet. The MailTransfer command is available on the line server only, and it serves to exchange mail with an external server.

SND	41H
REC	42H
PRT	43H
MSG	44H
TRQ	45H
TIM	47H
SPW	48H
DIR	4AH
SML	4BH
RML	4CH
DML	4DH
ACK	10H – 17H
NAK	25H
NPR	26H

username, filename, and servername, are strings of characters terminated by a 0X. Password, time and date are encoded into 4 bytes, and msgno is encoded in 2 bytes.

8. The Net Packet Encoding

Each packet consists of a header followed by a data part. The header contains the following fields:

dadr	1 byte	destination address (–1 = broadcast)
sadr	1 byte	source address
typ	1 byte	command code or sequence number (see App. 7)
len	2 bytes	length in bytes of data part
destLink	2 bytes	destination link number (unused)
srcLink	2 bytes	source link number (unused)

9. The Line Packet Encoding

Each packet consists of a header byte followed by a data part. The header byte contains the following fields:

type	bits 4–7	type at transport level
end	bit 2	unused
rsno	bit 1	receiver's sequence number (modulo 2)
ssno	bit 0	sender's sequence number (modulo 2)

Type values:

1	Data	2	DataAck
3	Open	4	OpenAck
5	Close	6	CloseAck
7	Abort		

Each packet is encoded for transmission in the following way:

1. A 2–byte checksum is computed and appended.
2. Byte triplets abc are formed and partitioned into 6–bit byte quadruples xyzw:

$$a_0 \ldots a_7\ b_0 \ldots b_7\ c_0 \ldots c_7 = x_0 \ldots x_5\ y_0 \ldots y_5\ z_0 \ldots z_5\ w_0 \ldots w_5$$

3. 33 is added to each 6–bit byte, which is extended to 8 bits.
4. A lead byte precedes the sequence obtained in steps 1–3. Its value is "{" for the partner which opened the connection (master), and "}" for the server.
5. A CR–byte (0DX) is appended.

10. The CX–400 Message Format

Message =	MSG	length envelope {body}.	
envelope =	ENV	length {field}.	
field =	MSGID	len string |	*message identification*
	ORIG	len ORname |	*originator*
	PRREC	len ORname |	*primary recipient*
	[COREC	len ORname] |	*copy recipient*
	[SUBJ	len string] |	*subject*
	[TITLE	len string] |	*title*
	SUBMI	len DateTime.	*submission date and time*
body =	ASCII	length {byte}.	
ORname =	list req conv show ntyp len string.		
DateTime =	year month day hour min sec.		
length:	4 bytes		

len:	2 bytes
string:	sequence of bytes terminated by 0X
list:	2 bytes (2010H or 2011H)
req:	2 bytes (2020H or 2021H)
conv:	2 bytes (2030H or 2031H)
show:	2 bytes (2040H or 2041H)
ntyp:	2 bytes (3010H, ETH name)
year:	1 byte, year – 1900

month, day, hour, min, sec: 1 byte each

MSG =	4000H
ENV =	5000H
MSGID =	1010H
ORIG =	1020H
PRREC =	1031H
COREC =	1032H
SUBJ =	1050H
TITLE =	1060H
SUBMI =	1090H
ASCII =	6020H