Network communication in the Oberon environment

ETH

Eidgenössische
Technische Hochschule
Zürich

Departement Informatik
Institut für
Computersysteme

Clemens A. Szyperski

Network Communication in the Oberon Environment

February 1990

Authors' address:

Computersysteme
ETH-Zentrum
CH-8092 Zurich, Switzerland

e-mail: szyperski@inf.ethz.ch

Network Communication in the Oberon Environment

Clemens A. Szyperski

Abstract

The Oberon system has several environmental and conceptual aspects that make it rather different from other systems supporting workstations interconnected by a local area network. This report concentrates on the integration of flexible communication primitives into the Oberon system. The major design goal was to simultaneously support simple applications using the primitives directly and higher-level communication services building on top of the primitives. The design decisions and the resulting implementation for the Ceres workstation environment are described. To illustrate effectiveness and efficiency of the chosen communication primitives, a typical example is developed, and some comparative performance figures are given. A concise interface documentation is added as a separate reference part.

1. Introduction

The available set of communication primitives has significant influence on the complexity of applications for loosely coupled distributed systems. There are numerous publications on the modelling and implementation issues of communications in such systems (for an overview see [Ta81]). In this report a set of communication primitives is presented. However, it is distinguished in the assumptions made for the underlying environment. In the following the environment and its influence on the design of the communication primitives is examined.

1.1 Environment: The Oberon System

The environment is made up of a set of Ceres workstations [Eb87, He88] connected via a low-cost local area network, the Ceres-Net [Wi89], and operated by the Oberon system [WiGu89]. The operating software i.e. the Oberon system had the most influence on the decisions taken and described in this paper.

The Oberon system is tailored to support single users on single-user workstations. It implements multi-tasking using a sort of a cooperative *non-preemptive scheduling* policy, as explained below. The term cooperative scheduling indicates that a task switch cannot take place unless the participating applications cooperate by returning control (as executing applications are never preempted). When the system is idle it cycles through a central loop polling various event queues. Typically, such events are input activities from the user (keyboard, mouse) and network requests, causing the system to call some event handler. The called handler may in turn update some data structures (such as inserting a character into a displayed text) or it may cause execution of some command. The Oberon system has no notion of a running application, as the user may initiate any command available in the system at any time the system is idle. Information maintained in global variables (or in the automatically garbage-collected heap) is preserved between execution of commands. Oberon is extensible: New commands that access existing data structures may be added at any time without disturbing the running system. Even on a low level there is no notion of static system configuration: device drivers may be added freely.

Assuming that individual commands given by the user should execute in a short time, the system has been designed to dedicate all its processing power to the execution of such a command. Once the command has completed, the system falls back into the central loop (becoming idle again) and waits for new events to come. Typical commands issued by the user call the compiler, open, modify, or close some documents, or request some service from a remote machine.

In order to make the central loop mechanism of Oberon extensible, it is possible to install idle event handlers, called tasks. Such tasks are called sequentially whenever the system is idle. For example, a standard task of the system is used to periodically collect garbage in the heap. Added tasks can poll device buffers and state information to take certain actions when required.

To support critical devices the Oberon system allows interrupt handlers to be installed without modifying the existing system. Interrupts may occur at any time and are serviced immediately unless some other interrupt service is still in progress. By convention, interrupt handlers add information to buffers which in turn are polled by the central loop or a task.

Although the Oberon workstations are designed to operate autonomously, the benefits of a network connection are not ignored. Currently, the network support covers various servers, including print–, file–, and mail–services. So far, the network support is hard–wired into the packages accessing the mentioned services [Wi89]. A goal of the project presented in this paper is to provide a simple yet efficient networking package that hides most of the hassles of (low–level) network protocols from the higher–level packages that use the network.

The Oberon system does not provide (preemptively scheduled) processes. Therefore, it is impossible to impose strict upper time bounds on the reaction time of an Oberon system to some occuring event. To put it in other words, the user might invoke some command that executes for an arbitrarily long time span. For user events (like keyboard or mouse activities) this is quite appropriate. However, network events, i.e. the arrival of some packet, have intrinsic properties that make them very different from local user events.

First of all, the network is not inherently reliable. Due to its access mechanism (CSMA *without* CD [MeBo76]) packets may get lost at any time. Furthermore, packets may get lost due to overflowing receiver buffers, and (with a far smaller probability, though) due to interrupt conflicts.

Secondly, event source and sink are located on separate machines (henceforth called sites), which in turn may fail separately. As there is no hardware support to distinguish between a remote machine that is just busy and hence not responding, and a site that has failed, the only means remaining to separate such cases is the use of timeouts. However, timeouts will not help unless a correctly received packet at a non–failed site can enforce some reactions within a given time bound.

Another orthogonal issue is the Oberon system's extensibility. While the local structure is quite extensible, this is not so for the current network related services. These are integrated into the implementations of a few, fixed commands accessing a set of predefined services. Instead, it should be possible to add new network related applications that do not interfere with existing ones and that can be used together on the same machine.

1.2 Outline of Chosen Approach

There are two quite different classes of clients that will use the communication primitives. On one hand, it should be possible and straightforward to design standard (usually point–to–point) applications like file transfer services that directly make use of the primitives. On the other hand, higher–level communication models building on the primitives such as point–to–multipoint message passing should be possible as well. Both classes of client programs should be able to coexist on the same site without getting in the way

of each other. In the following, the requirements of each of the client classes will be looked at separately, leading to a synthesis of communication primitives.

Simple applications that use the primitives directly usually need to 1) find a partner site (e.g. a server machine) by name for some particular task at hand and 2) interact with that partner site by exchanging information. To add fault–tolerance aspects, an application might also wish to monitor a remote site that it is expecting to receive an answer from, although this is usually less important for simple applications.

To simplify such applications it is important that the primitives guarantee consensus at both involved sites on the success of the information exchange performed. While it can be shown that this is generally not possible in the presence of site failures [FiLyPa85], the primitives should guarantee the consensus at a probability sufficiently high to ignore other cases. This will be discussed in detail later.

In addition to the above requirements, higher–level communication services often need to 1) monitor remote sites to detect certain kinds of site failures (especially fail–stop cases where the remote site crashed and does no longer respond to network events) and 2) to efficiently send small asynchronous messages. The former is required to implement higher–level abstractions that are resilient to site failures. The latter avoids synchronizing sites (as is caused by the reliable information exchange primitive described above) in the case of small protocol messages.

As a result, four classes of basic communication primitives can be derived: Reliable two–way information exchange, name binding and lookup, efficient signalling of small messages, and site monitoring. Each of these classes will be covered in the second section. The derived primitives impose certain requirements on the link access mechanism, as will be explained in the third section. To demonstrate the ease of writing simple communicating applications using these primitives, section four details a file transfer application including some comparative performance figures. A concluding reference part of this report describes the actual interfaces.

2. Communication Primitives

Four classes of communication primitives are discussed: Reliable two-way information exchange, name binding and lookup, efficient signalling of small messages, and site monitoring. Taking the special nature of the Oberon environment into account, special solutions for each of these have been developed. Each of the following sections starts by first explaining the chosen design and then discusses the reasons behind the major design decisions that were taken.

2.1 Synchronous Sessions – The Phone Call Protocol

The goal is to support two-way exchange of potentially large amounts of data between two sites. The Oberon system is either idle and waiting for some event to occur, or it is busy in servicing such an event. Hence, traditional communication models implemented using a set of concurrent (preemptively scheduled) processes are not implementable. Instead, the session model is used: two Oberon systems may execute freely unless they have both accepted some communication session between each other. Once the session has been established, both machines are fully dedicated to it until the session is closed again. A session consists of a series of strictly alternating simplex phases. To clearly denote the roles taken during a session, *caller*, *callee*, *sender*, and *receiver* are distinguished. Caller and callee are distinguished by the asymmetric way of opening a communication session. During each individual simplex phase, the two sites are clearly distinguished as sender and receiver. Initially, the caller is sender, and the callee receiver.

The situation where a site tries to communicate with another one can be compared to the very similar situation of a normal phone call. Initially, both sites are independent, i.e. are executing fully asynchronously (or have failed). If one of the sites tries to get in contact with the other one, it calls that site requesting a communication session. The caller tries for some time, not knowing whether the called site is busy or down. If the callee is engaged in an ongoing communication with some other site, the caller will receive some special signal from the communication service. If the callee accepts the request a service handler is invoked. To distinguish several handlers on the same site a special *service id* is used.

Upon establishing a communication between caller and callee, both sites are in a special mode and dedicate all their computing resources to the communication. The flow of information between the two engaged sites is assumed to being half-duplex. That is, during a session between caller and callee, both sites agree upon strictly alternating simplex phases. (This corresponds to a phone session between polite participants.) For obvious reasons, this model is called Phone Call model, and the protocol implementing it is called Phone Call Protocol (or PCP for short).

As long as the connection between caller and callee is established, both sites communicate synchronously. Finally, both sites agree to hang up. By hanging up, the session closes and the tight connection between the two participants is released. Hence, both sites are free again to return to some local event processing or to open up new communication connections. A question that arises is how both sites can agree upon when to switch phases, or when to hang up. Like in a real phone call the information required to perform this kind of high-level synchronization between communicating sites is contained in the semantics of the information exchanged until a certain point in time. Communicating applications must agree upon some conventions on how to interpret the information exchanged in a session anyway. It is only natural to exploit this existing consensus to deduce the required synchronization information. Later, it will be explained how to do this by sketching a typical application.

It is essential that the session mechanism eventually terminates under all circumstances to deal with failures, e.g. remote site crashes. PCP provides this with timeouts which indicate that the partner site did not sent the expected packet within a given time bound. To reduce complexity, the functions of reliably

detecting remote site failures and reliably communicating with remote sites have been separated completly. Hence, a timout abortion of a PCP session should not be interpreted as a signal that the partner site is down, but as a hint that it might be down and that a failure service should be consulted if appropriate. In the case that a request timed out, it is most likely that the partner site is busy and thus does not respond. To allow PCP to enforce these timeouts, the application using it executes under certain timing constraints. (This will be discussed in detail in section 2.1.6, below.)

2.1.1 Protocol Overview

The following simple EBNF syntax gives an overview of actual packet sequences on the channel pair between two sites using PCP (after removing duplicate packets). In the case of site failures (or too many lost packets), the actual packet sequence will be a prefix of a legal sentence produced by the syntax. By convention, the direction from caller to callee is *positive*, while the opposite direction from callee to caller is *negative*. A sign in front of some symbol denotes its direction, a plus sign is omitted. A sign in front of a non–terminal symbol distributes over all symbols of its production, while a sign in front of a terminal denotes the actual direction of the represented packet. (As usually, two consecutive minus signs cancel each other out.)

Interaction	=	REQ [–REJ | –REP Session].
Session	=	{Phase –Phase} (FinalPhase | Phase –FinalPhase).
Phase	=	{DAT –ACK} STP.
FinalPhase	=	{DAT –ACK} HUP –AAK.

The all–capital symbols are terminals representing network packets. The packet symbol meanings are:

REQ	request a session (carries information indicating which service is requested)
REP	request reply: the callee accepts the request
REJ	request reject: the callee rejects the request for some reason (i.e. the requested site is engaged, the requested service is not available, or the name binding is not valid)
DAT	basic data packet (*carries user data*), requires to be acknowledged
ACK	acknowledge last received DAT and request next one
STP	last data packet within a phase (*carries user data*)
HUP	last data packet within a session (*carries user data*)
AAK	special auto–acknowledgement packet (used to acknowledge last received HUP)

The first production captures the request mechanism, that may either fail to get a response for a request, may get some rejecting response, or may get a positive reply leading to a session. The second production describes such a session, where the last phase (called a final phase) is modelled separately. A normal (i.e. non–final) phase consists of a (possibly empty) sequence of data/acknowledgement packet pairs, followed by a stop packet. A final phase closes with a special stop packet, called hang–up, which requires the current receiver to answer with an auto–acknowledgement. (Other than normal acknowledgements, auto–acknowledgements are sent by the mechanism invoked upon physical arrival of a packet; this is explained below.) For further illustration, the following explains some typical PCP interactions, as observed when looking at the channel (without failures):

REQ.
The request has not been answered within some time bound, i.e. the callee is busy.

REQ –REJ.
The request has been answered but for some reason not accepted.

REQ –REP DAT –ACK DAT –ACK HUP –AAK.
A simple session sending user data in three packets (DAT, DAT, HUP) in one direction (i.e. using a single phase).

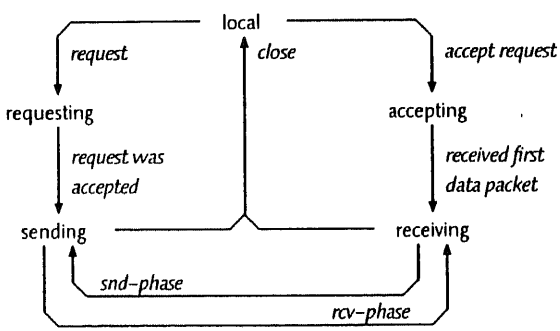
REQ –REP STP –DAT ACK –STP HUP –AAK.
A session consisting of three simplex phases, the first carrying a single user data packet (STP), the secong carrying two (DAT STP), and the last one carrying a single one, again (HUP).

Note that (except for repetitions to cover lost packets) the packets within a PCP interaction strictly alternate (which, of course, follows directly from the syntax above). As a result the thread of control of the PCP implementation is greatly simplified compared to a protocol supporting full duplex communication.

The PCP is composed of a request and a session mechanism described in the following. The request mechanism is used to establish sessions, while the session mechanism controls the exchange of data in alternating simplex phases.

2.1.2 Request Mechanism

An application wishing to request a session calls PCP and passes the information which other site it wishes to call and which service on that other site should be connected to. If there is no current session the local site is engaged in, the caller sends a request to the callee and waits for some answer (state *requesting*). Upon receiving such a request the callee decides whether it wishes to accept the request (i.e. whether the requested service is available, the name binding check (see below) succeeded, and no active session is currently running). It either rejects or accepts the request. In the latter case the callee awaits the first data from the caller (state *accepting*). Upon receiving a positive reply from the callee, the caller opens the session (state *sending*). Once the first data from the caller arrives at the callee, the callee opens the session (state *receiving*), too. While waiting in states requesting and accepting, both sites use timeouts to prevent endless waiting in the case of some failure. The timeout in state requesting also covers the case of a callee site that is busy and hence not responding. (The timeout state changes have been omitted from the diagram for clarity.)



If a site was busy for some time many outdated requests might have queued up in the link access buffer. To deal with such problems in an adequate way the link access module is required to automatically timestamp arriving packets. When fetching the next pending packet from the link access module the timestamp is used to discriminate obviously outdated packets.

2.1.3 Session Mechanism

Once caller and callee have cooperated to open a session, data transfer in the direction from caller to callee begins. The implementation uses a lazy send approach to collect as much data from an application as possible before sending an actual packet, i.e., a packet is sent only after PCP collected enough data to send a packet of maximum size allowed by the link access module, or after the application decides to switch phases, or to close the session. If a PCP call requires PCP to send some packet, the call blocks until the packet actually arrives and is acknowledged.

Once all data sent up to a certain point within a session has arrived, the sender may ask for a phase switch to become the new receiver. This causes all outgoing data still in the buffer to be sent as a special packet (STP: stop) – if there is no data in the buffer an empty STP packet is sent. A receiver uses either the semantics of what it received or a special end-of-data condition returned by PCP's receive calls to detect the end of a phase or session. Explicitly signalling end-of-data to the application is sometimes useful. For example, when the data sent has no easily known length and a terminator would be hard to code. Upon detecting the end of a phase, the receiver asks for a phase switch to become the new sender and the phase switch completes. An STP packet need not be acknowledged, as its arrival causes the first packet of the next phase to be sent, which is interpreted as having the STP packet's acknowledgement piggy-back.

Finally, the current sender of a session may decide to close the session. A special packet (HUP: hang-up) is sent which is comparable to STP packets. Its arrival causes signalling end-of-data to the receiver and also closes the session. The problem at this point is the reliable (consistent) close, also called "the last ack problem". If a HUP arrives, it makes no sense to send a standard acknowledgement packet upon receiving it! If the acknowledgement gets lost, the receiver has closed the session and may be busy again, while the sender repeats its last packet (the HUP) waiting for the acknowledgement, and eventually aborts. Hence, a single packet loss (the last acknowledgement) causes the whole session to be terminated inconsistently! This violates the requirement that consensus on session termination should be guaranteed at a very high probability.

To attack the reliable close problem, the HUP packet requests a special auto-acknowledgement packet (AAK) from its destination site (which is to be sent by the physical packet arrival interrupt handler). The HUP may be repeated if the AAK got lost, as a duplicate HUP is easily detected at the receiver's site (it cannot be another data packet, and it is not a request), and an AAK is sent even if the receiver's site is busy again. An AAK must arrive before the sender times out. It is claimed that this can be met with acceptable probability, as AAK packets are sent *immediately*. The crucial point is that the acknowledgement for a normal data packet has dual semantics, namely that the last packet arrived and that the next packet may be sent. For the last data packet (the HUP), far simpler semantics are in order, simply stating that the HUP has arrived at the destination. This solution allows the session close to be in the same order of reliability as all other parts of the session, only depending on the probability of a packet loss reduced by the maximum number of retries performed.

[Remark: There is a small probability that the receiver crashes after it successfully closed the session and caused some changes to stable storage, but before the sender had a chance to do all its HUP repeats. The argumentation above assumes that this probability is significantly smaller than that of losing a packet. At this point the Impossibility Result proved in [FiLyPa85] is reached.]

All session protocol parts except for the final acknowledgement are restricted to function only during an established session. The final acknowledgement has been moved down into the link access module and the PCP module has been implemented without processes.

2.1.4 Dealing with Unreliable Channels

The major assumption taken for the underlying physical communication channels (the network) is that messages sent from one site to some other fixed site are not reordered. However, the physical channel does lose packets. To deal with lost packets a retransmission strategy must be used. Retransmitting packets opens the possibility that a packet arrives more than once, i.e. packet duplicates must be detected and discarded.

The PCP uses the well-known Alternating Bit Protocol (ABP, also called stop-and-wait protocol). To cover lost packets, the ABP sends packets repeatedly until an acknowledgement arrives. To detect duplicates, data and acknowledgement packets are labeled with a sequence number (modulo 2: the alternating bit). In principle, both sites could actively repeat their current data or acknowledgement packet until the expected acknowledgement or the next data packet arrives, respectively. In practice, a sender or receiver driven model is used, where one of the two engaged sites drives the protocol by actively repeating packets. The former has been chosen for the session mechanism, since the request mechanism is necessarily sender driven.

Since packets between two engaged sites strictly alternate, the Alternating Bit Protocol may be used for the entire session independently of phase switches in between. To achieve the strict alternation, the implementation uses piggy-back acknowledgements whensoever possible, i.e. at the border between two phases (where the first packet of the new phase acknowledges the arrival of the last packet of the previous phase).

An important goal of the PCP design was to guarantee that if both engaged sites do not fail and both cooperating application parts behave consistently, then the PCP sessions should be reliable. Clearly, for physical channels that may lose a packet at any time, there is no possible algorithm to guarantee perfectly reliable transmission within bounded time. The PCP implementation's reliability depends on certain assumptions, i.e. that the probabilities that the channel delivers a corrupted packet or that it loses more than a certain number of packets within a critical path are neglectable.

2.1.5 The PCP Interface

The full PCP interface (together with the interfaces of other discussed modules) can be found in the reference part of this report. The following is a projection sketching the relevant parts.

```
CONST
    Done = 0;
    Rejected = -1; BadID = -2; BadName = -3; Timeout = -4; BadReceive = -5; NoRequest = -6;

TYPE
    Site = SHORTINT; ServiceID = INTEGER;
    Serve = PROCEDURE(s: Site);
    RejectFilter = PROCEDURE(id: ServiceID; s: Site);

VAR
    res: INTEGER; (*set to Done on entry of Install, Remove, Request, Accept, and before calling a service*)
    eod: BOOLEAN; (*"end of data" - reset by ReceivePhase and before calling a service; set by Receive*)

(*server installation*)
PROCEDURE Install(id: ServiceID; S: Serve);
PROCEDURE Remove(id: ServiceID);
```

```
(*session primitives*)
PROCEDURE Request(id: ServiceID; to: Site; tries: INTEGER; name: ARRAY OF CHAR);
PROCEDURE Accept(Rej: RejectFilter; VAR id: ServiceID; VAR from: Site);
PROCEDURE SendPhase;
PROCEDURE Send(x: BYTE);
PROCEDURE ReceivePhase(dt: LONGINT);
PROCEDURE Receive(VAR x: BYTE);
PROCEDURE Close;
```

The server installation/removal calls allow for maintaining up–call handlers [Cl85] associated with certain service numbers. Upon accepting a session request, PCP calls the appropriate handler (after the transition from state *accepting* to state *receiving*). The fact that the PCP may abort at any time (due to some timeout) is reflected by a global result variable (res). To simplify writing applications using PCP, the convention has been adopted that res = Done is established by "opening" calls to PCP, and that PCP session primitives (except for Request and Accept) do nothing after res ≠ Done is established, thus keeping that result value stable. Therefore, PCP applications do not need complicated nested structures to catch errors, while still being safe from "overlooking" some error. The second global variable, *eod* for end–of–data, signals that the last receive call passed the end of the current phase.

2.1.6 Rationales for Design Decisions

Alternating Bit Protocol vs. Sliding Window Protocols

As indicated above, the PCP session protocol is based on the simple Alternating Bit Protocol. A sliding window protocol could have been used instead. Within the Ceres/Oberon environment the physical network has never more than one bit in transmission. The logical channel between two sites which includes the link access receiver buffers can be thought of as having multiple packets "in transmission". However, PCP dedicates the full processing power of the communicating sites to the session and sending and receiving site are expected to be loaded about equally by a session. Hence, there is no site–determined reason for a window size greater than one, and a sliding window protocol would have the sole effect of reducing the number of acknowledgement packets sent. As PCP tries to send data packets of maximum size whensoever possible and as redundant acknowledgements during phase switches are sent piggy–back, the gained speedup is too small to justify the significant increase in implementation complexity. (For example, a back–of–the–envelope calculation shows that under the assumption that the transmission time is proportional to the number of bytes sent and that network contention effects are neglectable, the speedup for a window size of 8 is in the order of 10%.)

Implicit versus Explicit Request Acceptance

In a processless system, the complete absence of preemptive context switches restricts the system to a single thread of control. Therefore special program structures are required. Consider a system reacting to external events (like user input or arriving network packets). As the ordering of these events is nondeterministic, the system has to expect any event at any time. In the Oberon system the resulting structure is the central loop. The thread of control cycles through the central loop, called tasks, and called event handlers. Tasks and events handlers are procedures installed at run–time and invoked using up–calls [Cl85]. Each time the system returns to the central loop it is ready to pass control to one of the installed tasks or handlers.

The PCP module installs an Oberon task and periodically checks the link access module for packet arrivals. As long as no request packet is detected, the PCP tasks falls back to the central loop immediately. Otherwise, the request is checked and the requested service is looked up. If the request is acceptable and

the requested service is available, a session is started and the found service handler is up-called. As soon as the service handler terminates, it returns control to the PCP task, which in turn returns to the central loop.

The implicit acceptance of session requests from within the Oberon task conforms to the Oberon system's design, but in order to be able to accept a session request at all, the system must return to the central loop first. Then the PCP task eventually gets control, detects a current request and performs an up-call to the requested service, thereby accepting the request. This leads to an inverted style of programming also found in the non-distributed applications of Oberon, where input from the keyboard is normally accepted from the central loop by performing an up-call of some handler.

For certain applications it may be inconvenient or even impossible to return to the central loop just to accept some *awaited* session request. For example, an application might want to use a PCP session to pass parameters to some remote service. Then, it awaits a call-back from just that service to receive some results after an arbitrary delay. (Would the delay be short and bounded, a single PCP session could be used.) Using implicit request acceptance, this means: Update the global variables of the application to remember what it was doing, return to the central loop, reject every incoming request (directed to this application) except for the awaited one, and eventually handle the call-back. For typical client-server models this is not convenient. For RPC style [BiNe84] communication, it is even impossible to return to the central loop, as the remote call might have happened within some nested procedure calls.

To handle such situations, PCP allows for explicitly accepting session requests using the Accept procedure. To keep control within PCP when rejecting or accepting some pending request, Accept takes a reject filter procedure as parameter. If there is no pending request, Accept returns immediately, setting res to NoRequest. If a request is pending for which the reject filter returns TRUE, Accept causes rejection of that request, and otherwise behaves as if there was no pending request. Finally, if Accept detects a pending request for a desired service, it accepts the request and opens the session. Hence, if Accept returns with res = Done set, the awaited request has been accepted and the session is already established.

Control Flow and Application Constraints

In the PCP implementation the up-call technique is used at a single point to invoke a requested service. One could argue that a more symmetric implementation could use conventional calls to deal with the sending side, while using up-calls to deal with the receiving side. The straightforward way leads to an up-call for every single byte delivered to the receiving PCP client. This is certainly easy to do within the PCP module. However, the client code would have to keep track of what is going on in some global variables. As a result, the client code loses structure and becomes complicated and hard to understand. More modest approaches could try to use up-calls on a per-phase basis. However, the principle structuring problem remains.

Another issue is the way data is transferred during an ongoing session. The PCP implementation limits copying to an absolute minimum. Data is collected into a single transmission buffer large enough for composing packets of the maximum size. As soon as enough bytes have been collected they are sent. Arriving data is kept in the low level data link buffer until it is actually consumed by the PCP client. A different design might consider phases as atomic messages that are passed to PCP and delivered by PCP as a whole. However, the larger the transfer unit becomes, the more buffering needs to take place. Furthermore, it is very likely that this strategy leads to some restriction on the maximum message size. For example, file transfer applications for arbitrarily large files would then need to use their own higher-level protocol to slice up a file into such messages and recompose a file from the received messages.

The multi-phase sessions of PCP, its model of control flow, and its data transfer policy have their advantages when writing PCP based applications. However, they impose constraints on the application code that might not always be acceptable. In particular, the application code is allowed to produce and consume data "on the fly" for an ongoing session. Therefore, the application's producer and consumer code logically belongs to the timing critical network software. If an application takes too long to produce the next or consume the current byte, the partner site eventually times out and aborts the session. Hence, PCP applications must be designed with this timing constraint in mind.

The precise timing constraint imposed on a PCP application depends on the timeout constant settings in the used PCP module, as well as on the actual network access and transmission timing. The effect of these timing constraints depends on the relative speed of the producer/consumer code. The former tends to depend on the network bandwidth and the current network load, while the latter depends on the speed of the used machines. While in principle, it is possible to set all PCP timeout constants to rather large values, for performance reasons, these constants should be set to the smallest values possible to allow for quick recovery from intermediate packet losses. (The values chosen for the current implementation may be found in the reference part.)

An application expecting a certain computation delay at the receiving site before the data is ready to be sent back, the sending site can set an upper bound to the time it is willing to wait for a responce. If the application has no control over the time it takes to produce or consume a message, it could collect the message into some local data structure to avoid processing overhead within a PCP session. If the processing is the result of some previous session phase, a call-back model could be used. Here, the application closes the session, does the required computation, and calls back to return the results. A standard module for off-line message composition called Msgs (described in the reference part) simplifies the design of such applications.

2.2 Name Service

To support addressing remote sites by name, primitives controlling a simple distributed name service are provided. Each site has a local table of locally bound names. A name lookup is performed using a physical broadcast. To avoid repeated broadcasts in the case of repeated uses of the same name, found names and their associated machines are locally cached. The otherwise "empty" request packets are used to perform a cache consistency check "on the fly" [ChMa89]: A name may be passed together with the request and if the addressed site detects that the given name is no longer bound to it, the request is rejected. This in turn may be used by the requesting site to invalidate its cache entry.

As for the session primitives, the following interface is a projection of the important parts. The full interface is described in the reference part.

```
CONST
    Done = 0; BadName = -3;

TYPE
    Site = SHORTINT;

VAR
    res: INTEGER;  (*set to Done on entry of BindSite and FindSite*)
```

```
(*name service*)
PROCEDURE OwnSite(): Site;
PROCEDURE BindSite(name: ARRAY OF CHAR);
PROCEDURE UnbindSite(name: ARRAY OF CHAR);
PROCEDURE FindSite(name: ARRAY OF CHAR; VAR s: Site);
PROCEDURE InvalCachedName(name: ARRAY OF CHAR);
PROCEDURE InvalCachedSite(s: Site);
```

Name bindings are expected to be managed externally to the system, i.e. the name service does not guarantee that a name is bound to at most one machine, although the bind call tries to detect whether that happened. The Oberon system allows for "personalizing" an operating machine by setting a current user name (and password). By default, the name service considers the current user name to be bound to that machine. Additionally, further names may be bound to a machine. This is important if a collection of named services may or may not reside on the same machine: in the latter case the services need to have different names. An example might be a set of different services (like printer, mailer, and file distribution).

Caching multiple located names, as is done in the implementation, is important if several names are frequently looked up. If named objects may migrate from one machine to another while at least one machine still has the old location in its cache, the cache consistency check becomes important. As migration of such named objects is expected to be rather infrequent in the given environment, the chosen cache consistency check on the fly when using a cached location introduces minimal overhead. (The costs for a global cache invalidation algorithm upon changing a name binding are not justified in this case.) Therefore, the name service answers name searches with a "good guess" if the name binding is in the local cache. (A true lookup may be enforced, however, by first invalidating a potential cache entry before issuing the name search.)

[For a classification of name services and their integration into distributed systems, see [CoPe89]. In their terms, the PCP name service supports a single name space with synonyms (nicknames) and absolute names, but neither unique nor relative names.]

2.3 Asynchronous Markers

Sessions are ideal for potentially massive data transfer but at the costs of synchronizing the engaged sites and some significant packet overhead for small single-phase sessions. Certain applications might wish to unreliably send single packets (often called datagrams) to some remote site. For example, an application might periodically send some update information where only the latest is of interest and infrequent losses are acceptable. Another example is a higher-level communication service (providing some higher abstraction and involving some own protocols) which uses unreliable packets for its protocol.

In principle the underlying link access module (see third section) can be used to send such datagrams. To avoid conflicts with the PCP session primitives using the same module, multiple ports on the link access level could be introduced. Consider the following scenario. Two sites are engaged in a session, and a third site sends a datagram to one of these sites (using a non-session port). If there is only one packet buffer, the datagram must be discarded by the session primitives to avoid blocking forthcoming session packets. (As there are no processes the datagram cannot be fetched from the link access module while the session protocol has control.) To avoid discarding many datagrams during sessions, the link access module could maintain a separate buffer for each port, increasing the complexity of the link access interrupt driver. Also, using separate ports would create multiple logical channels between sites and hence destroy the often useful intrinsic property of the physical channel not to reorder messages between pairs of sites.

All these problems can be avoided by integrating datagram primitives into the PCP implementation. For the sake of having a name for it, these integrated datagrams are called markers. The following interface projection shows the marker primitives. Like session requests, markers are sent to some service on a

remote site. Upon arrival a marker is either delivered by performing an up-call, or it is queued if some ongoing session prevents immediate delivery. As the queue is of limited size and might be full, a marker might get lost even if it arrived. Markers that are queued are delivered before the next session is accepted. For an up-called service there is no way to determine whether a marker needs to be received or a session to be handled. The service must be dedicated to receiving markers since a special call ReceiveMark is used to actually take over the marker data. (This avoids a multitude of different cases that otherwise need to be considered in the session receive primitives.)

```
TYPE
    Site = SHORTINT;  ServiceID = INTEGER;

(*name service*)
PROCEDURE SendMark(id: ServiceID; to: Site; VAR marker: ARRAY OF BYTE; len: INTEGER);
PROCEDURE ReceiveMark(VAR marker: ARRAY OF BYTE; VAR len: INTEGER; VAR time: LONGINT);
```

As markers are meant to be fully asynchronous, they are never really "expected" at any one time. Hence, there is no explicit receive call (comparable to the session Accept call), but markers pending while Accept is called are delivered by up-calling the appropriate handler. This is fully transparent to the application calling Accept unless it has some marker receiving service installed. (Accept should not be called from within a marker handler, as this may cause recursive calls to the handler.)

2.4 Site Monitoring

The last set of primitives helps an application to monitor some remote site. A typical site monitoring algorithm either periodically polls a remote machine by sending some probe packet requesting a response, or periodically announces its own liveness to other sites by sending "up" packets. The major problem here is that a remote site might be busy for some long period of time. Unless special provisions are taken, a site monitor would require very large timeout constants to avoid frequent failure declarations of sites that are perfectly up and running.

The requirement for the link access level to be able to send an auto-acknowledgement upon arrival of certain packet types has already been isolated in the session primitives discussion. By adding a special probe packet type that is never delivered to the destination site but answered by an auto-acknowledgement, polling of remote sites with very short latency times becomes feasible. For simple detection of fail-stop crashes of a remote site, where arriving packets are no longer answered, this is already sufficient. If a remote site fails to respond after a certain number of probe packet tries, it can be considered to have crashed at a very high probability.

```
CONST
    Done = 0;  Timeout = -4;

VAR
    res: INTEGER;

TYPE
    Site = SHORTINT;

(*site monitoring*)
PROCEDURE ChangeState(new, mask: SET);          (* state := (state - mask) + (new * mask) *)
PROCEDURE Monitor(s: Site; tries: INTEGER; VAR state: SET; VAR time: LONGINT);
PROCEDURE Introduce(VAR n: INTEGER; VAR s: ARRAY OF Site; VAR state: ARRAY OF SET);
```

For failures that do not belong to the fail-stop class, i.e. where the failed site crashed in a way that left the interrupt-level intact, problems arise. To cover such cases auto-acknowledgement packets are augmented by a state set and a timestamp. Monitor sends a probe packet to a remote site and either

returns that site's current state set and timestamp, or times out. By inspecting state set and timestamp the liveness of a remote application may be judged. Applications executing on some site can use ChangeState to modify individual flags of the local state set any time. A call to ChangeState always causes the associated timestamp to be updated.

A recovering site can either contact some other site that is already up, or it can perform a local recovery. To decide what to do, it is important to be able to find other active sites. An easy way to do this is provided by the Introduce call which physically broadcasts a poll packet and collects all auto-acknowledgement packets that arrive within a certain time window after that. The resulting set of remote site states can be taken as a hint on where to find sites of interest.

3. Link Access

The lowest abstraction level in the networking software considered is the link access module (called SCC, reflecting the name of the link access chip [Scc82, Scc85]). The standard SCC module [Wi89] supports unreliable packet (datagram) transfer between sites. Packets consist of an SCC defined header plus some arbitrary data body. Sending requires passing the whole packet (header plus body) to the SCC module. As specified by the header, packets are either sent to some addressed site or physically broadcasted. The link access module contains the interrupt handler called upon arrival of a packet. If the module's packet buffer is full, the packet is discarded. Otherwise packet header and packet body are written to the buffer. To fetch a buffered packet from the SCC module a client program repeatedly polls the module until a packet is available which is signalled by returning the packet's header (and removing it from the buffer). Then it may be decided to skip the packet, or to fetch it bytewise from the SCC module.

As shown in the second section, the link access module should be augmented to additionally timestamp certain packets upon arrival and send back auto-acknowledgement packets on demand. To implement time-stamping and auto-acknowledging of arriving packets, the SCC module has been refined. The new SCC module (called SCC4) inspects the type field of arriving packets. If the type falls into a certain range, the packet is time stamped. If it falls into some other (overlapping) range, an auto-ack is sent back to the packet's source. The actual auto-acknowledgement packet can be freely customized. For failure service purposes, where it is important to monitor the liveness of some remote machine, independent of its local state and therefore independent of its buffer, a special probe packet type is supported that triggers an auto-acknowledgement, but it is always discarded.

A problem occurs when considering client programs of the standard SCC module (as the existing standard networking packages) to coexist with client programs of the new SCC module on the same machine. To solve this the SCC4 module has been implemented in a (perhaps temporary) version that supports both client classes at the same time. To do so, packet types are divided into positive and negative valued ones, where positive packet types are handled just as done by the standard SCC module, while negative ones have the additional semantics given above.

The following interface projection shows the essential parts of the SCC4 interface. The fully documented interface can be found in the reference part.

```
TYPE
    Header = RECORD
        valid: BOOLEAN;
        dadr, sadr, typ: SHORTINT;  (*destination site, source site, packet type*)
        len: INTEGER;  (*of data following header*)
        destLink, srcLink: INTEGER  (*spare*)
    END;

(*standard channel – positive packet types; compatible with standard module SCC*)
PROCEDURE Start(filter: BOOLEAN);
PROCEDURE SendPacket(VAR head, buf: ARRAY OF BYTE);  (*fills out head.sadr*)
PROCEDURE Available(): INTEGER;
PROCEDURE Receive(VAR x: BYTE);
PROCEDURE ReceiveHead(VAR head: ARRAY OF BYTE);
PROCEDURE Skip(m: INTEGER);
PROCEDURE Stop;
```

```
(*extended channel – negative packet types; supports auto-acks and real-time-stamps*)
PROCEDURE XStart(filter: BOOLEAN);
PROCEDURE XSendPacket(VAR head: Header; VAR buf: ARRAY OF BYTE); (*fills out head.sadr*)
PROCEDURE XSetAutoAck(len, destLink, srcLink: INTEGER; VAR buf: ARRAY OF BYTE);
PROCEDURE XReceive(VAR x: BYTE);
PROCEDURE XReceiveHead(VAR head: Header; VAR time: LONGINT);
PROCEDURE XSkip(m: INTEGER);
PROCEDURE XStop;
```

To avoid mutual blocking or interference, the two packet types are buffered in two separate buffers. For efficiency reasons, SCC4 has two disjunct interface procedure sets, referred to as standard and extended channel. The procedures defined for the standard channel have (except for the special meaning of negative packet types) the identical semantics and naming as those defined in standard SCC. (While in principle one could recompile all SCC clients to use SCC4, a simple stub module exists to redirect SCC calls to SCC4.)

4. Typical PCP Applications

The communication primitives presented in the second section are designed to support direct applications as well as higher-level communication abstractions. This section gives two examples, developed in full detail, to illustrate each of these program classes. The first example has been augmented with some comparative performance figures.

4.1 File Transfer Service – Example for a Direct PCP Application

To illustrate the correlation between PCP and an application using it, a typical PCP application is sketched below. Assume that a module should be developed supporting simple file transfer between named sites. The following is the definition of such a module:

```
DEFINITION FileTransfer;
   CONST
      Ok = 0; Unavail = 1; Busy = 2; NoPerm = 3; NotFound = 4; Failed = 5; (*result values*)

   VAR
      res: INTEGER;

   PROCEDURE Permissions(read, write: BOOLEAN);
   PROCEDURE Send(site, file: ARRAY OF CHAR);
   PROCEDURE Receive(site, file: ARRAY OF CHAR);
END FileTransfer.
```

The *Permissions* procedure allows for enabling/disabling remote reading and/or writing of files. Initially both are disabled. Procedures *Send* and *Receive* try to send a file to resp. receive a file from a remote site. The possible result values signal that a file transfer has completed successfully, that the named remote site cannot be found or that it has not answered, that it is busy, that the permission to perform the requested transfer is not granted, or that the transmission failed for some other reason.

4.1.1 Session Syntax and Transfer Primitives

The first step in designing a PCP client module is the definition of an appropriate session syntax. For the file transfer example, the following syntax is used: (The sign conventions are as explained in section 2.)

```
Send     = filename (–NOPERM | –OK File).
Receive  = filename (–NOTFOUND | –NOPERM | –OK –File).
File     = {byte}.
```

The filename is a zero terminated string. The all-capital symbols correspond to the result codes defined as constants above and are sent as single bytes. From the syntax follows that four basic object types are to be transfered: single bytes, strings, and files.

To start with, primitives to send and receive zero-terminated character strings and arbitrary files are defined using the PCP send and receive byte primitives. Normally, such standard transfer operations will be packaged into a service module residing on top of PCP (like the module Msgs described in the last section). For the sake of clarity, the presented transfer primitives are somewhat over-simplified: first of all, they do not check for file system errors, and secondly, the file to PCP data exchange happens bytewise. The primitives given in the completed file transfer module in the appendix have been refined to remove both shortcomings, by catching file system errors and by transferring large blocks of bytes.

```
PROCEDURE SendStr(VAR s: ARRAY OF CHAR);
    VAR
        i: INTEGER;
BEGIN
    i := -1; REPEAT INC(i); PCP.Send(s[i]) UNTIL s[i] = 0X
END SendStr;

PROCEDURE SendFile(F: Files.File);
    VAR
        R: Files.Rider;  buf: BYTE;
BEGIN
    Files.Set(R, F, 0); Files.Read(R, buf);
    WHILE ~R.eof & (PCP.res = PCP.Done) DO
        PCP.Send(buf); Files.Read(R, buf)
    END
END SendFile;

PROCEDURE ReceiveStr(VAR s: ARRAY OF CHAR);
    VAR
        i: INTEGER;
BEGIN
    i := -1; REPEAT INC(i); PCP.Receive(s[i]) UNTIL (s[i] = 0X) OR (PCP.res # PCP.Done)
END ReceiveStr;

PROCEDURE ReceiveFile(VAR name: ARRAY OF CHAR);
    VAR
        R: Files.Rider;  F: Files.File;  buf: BYTE;
BEGIN
    F := Files.New(name);
    IF F # NIL THEN
        Files.Set(R, F, 0); PCP.Receive(buf);
        WHILE ~PCP.eod & (PCP.res = PCP.Done) DO
            Files.Write(R, buf); PCP.Receive(buf)
        END;
        IF PCP.res = PCP.Done THEN Files.Register(F) END
    END
END ReceiveFile;
```

4.1.2 Application Services – The Callee Part

The syntax suggests to have two different PCP services. One for sending a requested file, and the other for receiving a sent file. The server procedures are designed to handle the callee part of the file transfer sessions.

```
CONST
    SendID = 41;  ReceiveID = 43;  (*some unique service ids*)

VAR
    readPerm, writePerm: BOOLEAN;  (*remote file read/write permissions*)
```

```
PROCEDURE *SendServer(site: PCP.Site);
    VAR
        F: Files.File;  name: ARRAY 32 OF CHAR;
BEGIN
    ReceiveStr(name); PCP.SendPhase;
    IF readPerm THEN
        F := Files.Old(name);
        IF F # NIL THEN PCP.Send(Ok); SendFile(F)
        ELSE PCP.Send(NotFound)  (*file not found*)
        END
    ELSE PCP.Send(NoPerm)  (*read permission denied*)
    END;
    PCP.Close
END SendServer;

PROCEDURE *ReceiveServer(site: PCP.Site);
    VAR
        name: ARRAY 32 OF CHAR;
BEGIN
    ReceiveStr(name); PCP.SendPhase;
    IF writePerm THEN PCP.Send(Ok); PCP.ReceivePhase(0); ReceiveFile(name)
    ELSE PCP.Send(NoPerm)  (*write permission denied*)
    END;
    PCP.Close
END ReceiveServer;

BEGIN (*module body*)
    readPerm := FALSE; writePerm := FALSE;
    PCP.Install(SendID, SendServer); PCP.Install(ReceiveID, ReceiveServer)
END FileTransfer.
```

4.1.3 Application Drivers – The Caller Part

Finally, the two file transfer procedures are developed to remotely invoke the corresponding server. These two procedures handle the caller part of the file transfer sessions. (Note the simplification possible due to the implicit guard PCP.res = PCP.Done of the PCP session primitives.)

```
CONST Tries = 3;
VAR res: INTEGER;

PROCEDURE Send*(site, file: ARRAY OF CHAR);
    VAR
        s: PCP.Site;  F: Files.File;  stat: SHORTINT;
BEGIN
    F := Files.Old(file);
    IF F # NIL THEN
        PCP.FindSite(site, Tries, s);
        IF PCP.res = PCP.Done THEN
            PCP.Request(ReceiveID, s, Tries, site); SendStr(file);
            PCP.ReceivePhase(0); PCP.Receive(stat);
            IF (PCP.res = PCP.Done) & (stat = Ok) THEN PCP.SendPhase; SendFile(F) END;
            PCP.Close;
            IF PCP.res = PCP.Done THEN res := stat ELSE res := Failed END
        ELSE res := Unavail
        END
    ELSE res := NotFound
    END
END Send;
```

```
PROCEDURE Receive*(site, file: ARRAY OF CHAR);
    VAR
        s: PCP.Site; stat: SHORTINT;
BEGIN
    PCP.FindSite(site, Tries, s);
    IF PCP.res = PCP.Done THEN
        PCP.Request(SendID, s, Tries, site); SendStr(file);
        PCP.ReceivePhase(0); PCP.Receive(stat);
        IF (PCP.res = PCP.Done) & (stat = Ok) THEN ReceiveFile(file) END;
        PCP.Close;
        IF PCP.res = PCP.Done THEN res := stat ELSE res := Failed END
    ELSE res := Unavail
    END
END Receive;
```

To keep the example compact, the presented module does not include user authentification, which could be added easily.

4.1.4 Timing Constraints

As explained in the second section, a PCP application executes under timing constraints. When inspecting the application code developed above, the timing critical operations are localized in the transfer primitives, or, more precisely, in the file transfer primitives. The current PCP implementation uses timeout constants that easily cover the transfer of files within ongoing sessions, including the occuring file accesses. To generally simplify the design of PCP applications, it is useful to collect such transfer primitives with known timing behaviour into a service module. (The module Msgs described in the reference part of this report is an example for such a service module.)

4.1.5 Complexity and Performance Measures

A comparison of the file service in implementation size and execution times may be in order. It is compared to the existing dedicated file service. (*FileServ* is a version of the developed module FileTransfer which provides an Oberon command interface instead of a procedural interface.)

Module sizes

	SCC4	*PCP**	*FileServ*	*SCC*	*Net*
Lines of source code	240	450	160	150	460
Size of code (bytes)	1600	3800	1300	1100	4600

**version supporting sessions, only*

Performance Measures

	Ceres–1 (32032, 10MHz)			Ceres–2 (32532, 25MHz)		
Memory to memory						
repeat–loop	17.7			2.2		s/MByte
processor move inst.	2.5			0.3		s/MByte
File reading/writing						
in 512 byte blocks	23.6			9.4		s/MByte
byte–wise	74.3			23.5		s/MByte
Raw network speed			36.5			s/MByte
File transmission[+]	*PCP*[°]	*std*[*]		*PCP*[°]	*std*[*]	
in 512 byte blocks	80.8	n/a		58.6	n/a	s/MByte
byte–wise	196.6	115.2		91.8	75.5	s/MByte

[°]*SCC4 + PCP + FileServ* [*]*SCC + Net* [+]*for large files*

The performance advantage of the PCP solution using block operations over the standard Net module which operates strictly byte–wise is noticeable although PCP introduces a generic protocol and its own module layer. However, this result is rather sad as it forces one to conceive, implement, and use the "right" block operations without gaining any additional functionality. (There are two causes for the high costs of bytewise operations: external procedure calls are rather overpriced and processor supported block moves are relatively fast, rf. to memory–to–memory measurements, above.)

4.2 Communicating Logical Nodes – Example for a PCP based Communication Abstraction

A logical node is a communicating entity residing on exactly one site at a time. More than one node may exist on a single site. Therefore, communication between nodes falls into two classes: between two nodes on the same site and between two nodes on different sites. The logical node abstraction makes this difference transparent. As long as the only means of interaction between such nodes is through messages sent using the abstract communication primitives, performance changes are the only differences resulting from a migration of a logical node from one site to another.

The construction of higher–level abstractions is usually rather involved. The abstraction developed in the following should be considered as a mere example, hence leaving out important details of a fully functional implementation of a communication abstraction. The following module definition is used; the full implementation is given in the appendix.

```
DEFINITION Comm;
    IMPORT
        PCP, Msgs;

    CONST
        BadNode = –10;  (*other result values as defined by PCP module*)

    TYPE
        Node = LONGINT;
        Receiver = PROCEDURE(from, to: Node; M: Msgs.Msg);
        Notifier = PROCEDURE(from, to: Node; M: Msgs.Msg);

    VAR
        res: INTEGER;
```

```
    PROCEDURE Start;
    PROCEDURE Stop;

    PROCEDURE Find(name: ARRAY OF CHAR; VAR node: Node);
    PROCEDURE Home(node: Node): PCP.Site;
    PROCEDURE New(name: ARRAY OF CHAR; R: Receiver; N: Notifier; VAR node: Node);
    PROCEDURE Close(node: Node);

    PROCEDURE Send(from, to: Node; M: Msgs.Msg);
END Comm.
```

Module Comm makes use of abstract messages defined by the module Msgs (described in the reference part). The PCP module restricts data items to be transported within a session's phase to sequences of bytes. Also, PCP does not support sessions with caller and callee being the same site. Furthermore, PCP imposes real-time constraints on applications producing or consuming such items. To simplify applications, module Msgs ("messages") provides for means to produce, send, receive, and consume messages composed of typed items. In the current implementation, such items are integer numbers, strings, sequences of bytes, file pieces, and other messages (cyclic inclusion not possible). The composition of messages follows a simple syntax:

```
Message     = { number | string | bytes | Piece | Message }.
Piece       = Filename beg end.
```

As for PCP sessions, the message module expects clients to use the semantics of what they have read from a message up to a certain point to deduce what needs to be read next; a message contains nearly no overhead data to describe the items written to it. Module Msgs could be a basis to support communication in a heterogeneous network, as it might translate to and from some external data representation format (like ASN.1 [Iso87] or XDR [Sun88]).

Instead of developing Comm in full detail as was done in the first example, only the main ideas are sketched. To start with, the intended semantics of the interface is defined. (Procedures *Start* and *Stop* are meant to start and stop the automatic delivery of messages. They do not clear internal queues nor stop attempts to deliver queued messages to remote sites.)

Comm defines an abstraction called (logical) node. Nodes have (network-wide) unique identifiers of type *Node*. Comm maintains an association list L of quadruples (*n, nm, R, N*). Each quadruple binds a receiver *R* and a notifier *N* to a node *n*, both being procedure variables. Furthermore, it binds a name *nm* to node *n*. Upon receiving a message, Comm adds it to its local delivery queue. The head of the delivery queue is delivered to its destination node *n* by calling n's receiver. If *n* tries to send a message to some other node it might happen that the other node is unavailable. If so, Comm calls *n*'s notifier. A notifier may inspect the global result variable res to get a clue of what went wrong with the message. The name *nm* of node *n* is never used to address it, but may be used to initially find some node. All operations of Comm work transparently for nodes residing on the local site as well as for nodes residing on remote sites.

Find(nm, n) – locate node *n* named *nm*. Returns *n* = –1 and res = PCP.BadNode if no node with that name could be found. As Find uses the PCP name service this does not necessarily mean that such a node does not exist (remember, this is a small sample application), but that node *nm*'s home site is temporarily not available, i.e. busy or down.

Home(n) – return home site of node *n*. This is a local operation which always works for valid node identifiers.

New(nm, R, N, n) – create new node with name *nm*, receiver *R*, and notifier *N*. The new node's identifier *n* is returned.

Close(n) – close node *n*. Traps if *n* is not bound to the local site, i.e. it is not possible to close nodes on behalf of remote sites.

Send(n, n', m) – send message *m* from node *n* to node *n'*. Traps if *n* is not bound to the local site, or if *m* is not in state MReady (cf. description of Msgs, reference part).

Comm maintains a delivery queue and a transmission queue. A message sent to a local node is immediately appended to the delivery queue. The same happens to messages arriving from remote sites. Messages sent to remote sites are added to the transmission queue. If the transmission queue is not empty, Comm tries to transmit the head message to its destination site. If the head message cannot be delivered, Comm calls the originator node's notifier (this fails if the originator has been closed in the meantime). The notified node may then decide to retransmit or discard the message. (Note that a retransmission leads to a new relative ordering of the message with respect to others already in the transmission queue.)

To solve the binding problem of nodes to their home sites, a special encoding for node identifiers has been chosen. For node *n* its site identifier is *n* MOD Mask (for some implementation dependent Mask). A problem with this is the missing indirection. A node cannot migrate to another site without aquiring a new node identifier and thereby making it unreachable for former clients. To allow for transparent node migration a client can use an indirect naming scheme via the name service (*Find*).

5. Conclusions

A set of simple yet flexible communication primitives has been introduced covering both the needs of applications using them directly and of applications using them to provide some higher-level communication abstraction. The primitives cover four areas: the synchronous and reliable two-way exchange of potentially large amounts of data (sessions), the binding and lookup of names, efficient signalling of small messages, and monitoring of remote sites. To illustrate the interrelation between the primitives and applications using it, two typical applications have been sketched and analyzed.

For the session mechanism a simple communication protocol, the Phone Call Protocol, has been introduced. It has been shown how this protocol can be implemented for the Oberon architecture, and how its specific characteristics match characteristics of the Oberon system.

It has been demonstrated that an efficient and effective abstraction can be provided supporting network based applications in the Oberon environment. Experience shows that writing applications that use the primitives is easy and yields compact and concise code. By carefully designing the network application's thread of control, it has been possible to completely avoid introducing preemptively scheduled processes without sacrificing clarity or usefulness. Only in the case of unsuccessfully awaiting some information from a remote site (e.g. a session or name service request) does the local site perform true busy waiting. (On a far smaller scale the same happens when recovering from packet losses.)

To keep the PCP simple but flexible, the thread of control extends through client code during an active session. In other words, the producing and consuming parts of the client code calling the PCP send and receive procedures logically belong to the protocol implementation and therefore run under certain timing constraints. On the other hand, they belong to the client implementation as they provide for the semantics of the transmitted data and thereby control the session and its phases. While the timing constraints are in an order that allows for most data access methods (including file accesses), they are clearly not acceptable when computations are required in between. To aid in writing applications of the latter type, a module supporting abstract messages has been provided. Such messages are composed off-line, sent and received under control of that module, and finally consumed off-line.

On the link access level (the SCC module), two core features have been isolated that are a prerequisite to implementing network protocols in the Oberon system: auto-acknowledgement packets and arrival real-time-stamps. In the former case, certain packet types should be automatically acknowledged upon arrival. This is necessary to allow for reliable close in the PCP at a high probability, as well as for implementing site monitoring facilities: By sending a special probe packet which gets automatically acknowledged, but otherwise ignored, allows to distinguish between machines that are down and others that are just busy.

In the latter case, it should be possible to place a real-time stamp onto certain packet types upon arrival. Having time stamped packets is *useful* to avoid unnecessary waiting after detecting outdated packets (especially requests), and it is *necessary* to implement algorithms that wish to exploit the passing of time to derive information about global state without actually communicating [La84]. The key idea here is that placing a time stamp onto a packet just after it arrived happens physically almost at the same time as the end of the packet's transmission happens on the sending site. Hence, both sites mark (almost) the same physical point in time relative to their local clocks. As the used crystal clocks have a very small relative drift, the passing of (local) time on one site allows to deduce information about the passing of (local) time on the other site.

References

[BiNe84] A.D. Birrel, B.J. Nelson. *Implementing Remote Procedure Calls.* ACM TOCS 2:1, pp.39–59, 1984.

[ChMa89] D.R. Cheriton, T.P. Mann. *Decentralizing a global naming service for improved performance and fault tolerance.* ACM TOCS 7:2, pp. 147–183, 1989.

[Cl85] D.D. Clark. *The Structuring of Systems using Upcalls.* ACM Operating System Reviews 19:5, pp. 171–180, 1985.

[CoPe89] D.E. Comer, L.L.Peterson. *Understanding naming in distributed systems.* Distributed Computing 3, pp. 51–60, 1989.

[Eb87] H. Eberle. *Hardware description of the workstation Ceres.* ETH Zürich Departement Informatik Technical Report 70, 1987.

[FiLyPa85] Fischer, Lynch, Paterson. *Impossibility of Distributed Consensus with One Faulty Process.* J. ACM 32:2, pp. 374–382, 1985.

[Gu89] J. Gutknecht. *The Oberon Guide.* (revised edition) ETH Zürich Departement Informatik Technical Report 119, 1989.

[He88] B. Heeb. *Design of the processor board for the Ceres–2 workstation.* ETH Zürich Departement Informatik Technical Report 93, 1988.

[Iso87] International Organization for Standardization. *ISO–OSI Specification of Abstract Syntax Notation One (ASN.1).* ISO 8824:1987(E)

[La84] L. Lamport. *Using time instead of timeout for fault–tolerant distributed systems.* ACM TOPLS 2:1, pp. 254–280, 1984.

[MeBo76] R.M. Metcalfe, D.R. Boggs. Ethernet: *Distributed packet switching for local computer networks.* Comm. ACM 19, pp. 395–404, 1976.

[Scc82] *Z8530/Z8030 Serial Communications Controller.* Technical Manual, Advance Micro Devices, 1982.

[Scc85] *Z8530 SCC Serial Communications Controller.* Datasheet, Zilog, 1985.

[Sun88] Sun Microsystems. *Network Programming Manual: External Data Representation (XDR).* 800–1779–10 Rev. A, 1988.

[Ta81] A.S. Tanenbaum. *Network protocols.* Computing Surveys 13:4, pp 453–489, 1981.

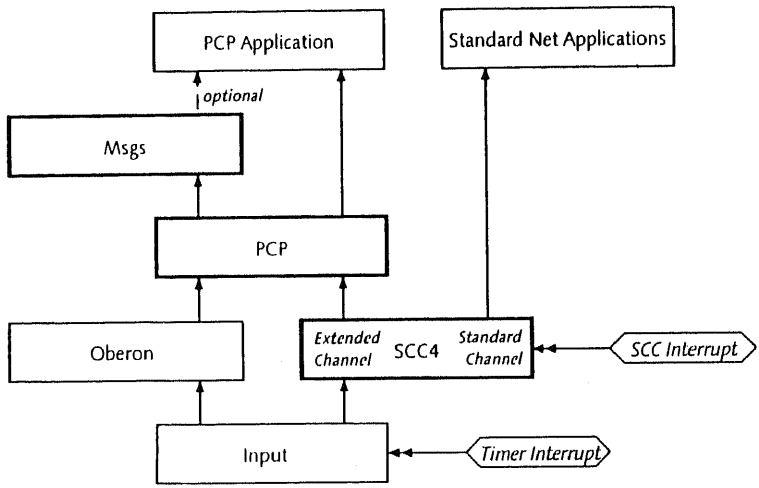
[Wi89] N. Wirth. *Ceres–Net: A low–cost computer network.* Software – Practice and Experience 20:1, pp. 13–24, 1989.

[WiGu89] N. Wirth, J. Gutknecht. *The Oberon system.* Software – Practice and Experience 19:9, pp. 857–893, 1989.

Part II – Reference Manual

1. Module Structure and Overview

The following diagram shows the module dependency graph of the modules described in this reference part: SCC4, PCP, and Msgs. Potential application modules and some relevant standard modules have been incorporated to clarify the overall picture.



Module Sizes

	SCC4	SCC4*	PCP	Msgs
Lines of source code	240	290	600	330
Size of code (bytes)	1600	2200	5300	2900

**including standard channel support*

The link access module SCC4 provides for the interface to the physical network. Module PCP contains several sets of communication primitives. Module Msgs supports the off-line preparation of structured messages and their controlled transmission. Each of the module interfaces is described in the following sections. For a discussion on the chosen design refer to the corresponding sections in the first part of this report.

A major design decision was *not* to decompose the set of communication primitives described in the second chapter into several modules. The rationale for this is the rather tight interconnection of the primitive's implementations and the moderate size of the resulting monolithic module PCP. A modular decomposition of PCP would lead to, say four modules (a protocol base module, a session service module, a marker plus site monitoring module, and a name server module). Typical clients would have to import (directly or indirectly) almost always nearly all of them, leading to a fifth (closure) module.

2. SCC4 Module

```
DEFINITION SCC4;
    TYPE
        Header = RECORD
            valid: BOOLEAN;
            dadr, sadr, typ: SHORTINT;
            len: INTEGER;  (*of data following header*)
            destLink, srcLink: INTEGER  (*spare*)
        END;

    (*standard channel - positive packet types; compatible with standard module SCC*)
    PROCEDURE Start(filter: BOOLEAN);
    PROCEDURE SendPacket(VAR head, buf: ARRAY OF BYTE);
    PROCEDURE Available(): INTEGER;
    PROCEDURE Receive(VAR x: BYTE);
    PROCEDURE ReceiveHead(VAR head: ARRAY OF BYTE);
    PROCEDURE Skip(m: INTEGER);
    PROCEDURE Stop;

    (*extended channel - negative packet types; supports auto-acks and real-time-stamps*)
    PROCEDURE XStart(filter: BOOLEAN);
    PROCEDURE XSite(): SHORTINT;
    PROCEDURE XSendPacket(VAR head: Header; VAR buf: ARRAY OF BYTE);
    PROCEDURE XSetAutoAck(len, destLink, srcLink: INTEGER; VAR buf: ARRAY OF BYTE);
    PROCEDURE XReceive(VAR x: BYTE);
    PROCEDURE XReceiveBytes(VAR x: ARRAY OF BYTE; beg, len: INTEGER);
    PROCEDURE XReceiveHead(VAR head: Header; VAR time: LONGINT);
    PROCEDURE XSkip(m: INTEGER);
    PROCEDURE XStop;
END SCC4.
```

For the definition of the standard channel interface refer to the Oberon Guide [Gu89]. The following describes the differences found when using SCC4 and especially when using the extended channel part.

The extended channel of SCC4 adds auto-acknowledgements and real-timestamping. Auto-acknowledgements have packet type -64. Packet types in the range [-128..-65] are auto-acknowledged, packet types in the range [-128..-1] are timestamped. The packet type -128 is always auto-acknowledged and then discarded (probe), while other packets are auto-acknowledged if they fit into the buffer, only.

Start and *XStart* clear the packet buffer of the corresponding channel. The physical site id of the local machine can be retrieved using *XSite*. The actual auto-acknowledgement packet can be fully customized using *XSetAutoAck* which sets the remaining header fields and the body part of auto-acknowledgements sent afterwards. Besides *XReceive*, *XReceiveBytes* has been added to efficiently receive a block of bytes directly into some buffer. Finally, *XReceiveHead* has been augmented with an additional result parameter returning the actual arrival time of the packet corresponding to the returned header. (As in SCC, *XReceiveHead* returns with head.valid = FALSE if no packet is currently pending.)

Both channels may be separately turned on and off. Upon module initialization, the auto-acknowledgement packet is set to a header with head.len = 0 (i.e. no body part), and both channels are started using *Start*(TRUE) and *XStart*(TRUE).

Warning: when starting a channel with parameter false all packets on the network will be received which is useful for monitoring tools. As such tools need to take over the whole machine anyway (to keep up with the potentially very high input rate) it is expected that either both channels are monitored or one is turned off. The latest call to one of the Start procedures determines the actual filter mode of both channels. (Note that this is the only possible point of interference between the two SCC4 channels.)

3. PCP Module

```
DEFINITION PCP;
    CONST
        Done = 0;
        Rejected = -1; BadID = -2; BadName = -3; Timeout = -4; BadReceive = -5; NoRequest = -6;

    TYPE
        Site = SHORTINT;
        ServiceID = INTEGER;
        Serve = PROCEDURE(s: Site);
        RejectFilter = PROCEDURE(id: ServiceID; s: Site);

    VAR
        res: INTEGER;    (*set to Done on entry of procedures marked with res! and before up-call
                          - guard for procedures marked with res? - ignored by other procedures*)
        eod: BOOLEAN; (*"end of data" - reset by Request, Accept, ReceivePhase - set by Receive, ReceiveBytes*)

    PROCEDURE Start;
    PROCEDURE Stop;

    (*server installataion*)
    PROCEDURE Install(id: ServiceID; S: Serve);                                    (*res!*)
    PROCEDURE Remove(id: ServiceID);

    (*marker primitives*)
    PROCEDURE SendMark(id: ServiceID; to: Site; VAR marker: ARRAY OF BYTE; len: INTEGER);   (*res!*)
    PROCEDURE ReceiveMark(VAR marker: ARRAY OF BYTE; VAR len: INTEGER);             (*res?*)

    (*site monitoring*)
    PROCEDURE ChangeState(new, mask: SET);
    PROCEDURE Monitor(s: Site; tries: INTEGER; VAR state: SET; VAR time: LONGINT);  (*res!*)
    PROCEDURE Introduce(VAR n: INTEGER; VAR s: ARRAY OF Site; VAR state: ARRAY OF SET);  (*res!*)

    (*session primitives*)
    PROCEDURE Request(id: ServiceID; to: Site; tries: INTEGER; name: ARRAY OF CHAR);  (*eod! res!*)
    PROCEDURE Accept(Rej: RejectFilter; VAR id: ServiceID; VAR from: Site);          (*eod! res!*)
    PROCEDURE SendPhase;                                                             (*res?*)
    PROCEDURE Send(x: BYTE);                                                         (*res?*)
    PROCEDURE SendBytes(VAR x: ARRAY OF BYTE; beg, len: INTEGER);                    (*res?*)
    PROCEDURE ReceivePhase(dt: LONGINT);                                             (*eod! res?*)
    PROCEDURE Receive(VAR x: BYTE);                                                  (*eod? res?*)
    PROCEDURE ReceiveBytes(VAR x: ARRAY OF BYTE; beg, len: INTEGER);                 (*eod? res?*)
    PROCEDURE Close;

    (*name service*)
    PROCEDURE OwnSite(): Site;
    PROCEDURE BindSite(name: ARRAY OF CHAR);                                         (*res!*)
    PROCEDURE UnbindSite(name: ARRAY OF CHAR);
    PROCEDURE FindSite(name: ARRAY OF CHAR; tries: INTEGER; VAR s: Site);            (*res!*)
    PROCEDURE InvalCachedName(name: ARRAY OF CHAR);
    PROCEDURE InvalCachedSite(s: Site);
END PCP.
```

PCP defines id types for sites and services local to some site. It maintains an association list L of pairs (service id, up-call handler), a list of locally bound names N, and an association cache C of pairs (name, site id), which contains names that were found to be bound to a remote site.

Two exported global variables of PCP open part of its state to client programs. Both should be treated *read-only*. The result variable *res* takes one of the values listed in the constants list and has a dual functionality.

For procedures P marked with *res!* the semantics is:

P: [| P.action; res := P.result |]

For procedures Q marked with *res?* the sematics is:

Q: [| if res = Done → P.action; res := P.result [] res ≠ Done → skip fi |]

For procedures neither marked with *res!* nor with *res?* this variable is neither changed nor inspected.

The second variable *eod* signals that within a session's phase the last byte sent has already been received. Otherwise the procedure annotation with *eod!* and *eod?* follows analogous conventions as for *res*.

Procedures *Start* and *Stop* control the task inside of PCP polling SCC4 to receive packets. They do not affect L, N, or C and except for controlling the task do not affect the modules functionality. (Idempotent operations.)

3.1 Server installation and removal

Install(id, S) – if no pair (id, S') ∈ L, add pair to L; if pair (id, S) ∈ L the call is ignored, and if pair (id, S') ∈ L with S' ≠ S, res = BadID is returned. (Idempotent operation.)

Remove(id) – a pair (id, S) ∈ L is removed. (Idempotent operation.)

3.2 Asynchronous markers (datagrams)

Markers are blocks of arbitrary data and limited size. For the Ceres implementation this limit is 512 bytes.

SendMark(id, s, m, l) – unreliably send a marker *m*[0..*l*–1] to service *id* on site *s*. Traps for *l* ≥ 512.

ReceiveMark(m, l, T) – to be called from within a service handler, returns the received marker *m*[0..*l*–1] and its arrival time stamp *T*. (*T* is comparable to values returned by function *Oberon.Time*().)

3.3 Site monitoring

PCP maintains a local state set S (in the Ceres implementation, 32 flags where flags [0..3] are reserved for PCP) associated with a timestamp T. This timestamp is comparable to values returned by function *Oberon.Time*().

ChangeState(N, M) – S, T := (S – *M*) ∪ (*N* ∩ *M*), *Oberon.Time*(). The idea is to select a subset of the state flags using a mask set *M* and to give new values *N* for these flags. For example ChangeState({5}, {5, 6}) sets flag 5 and clears flag 5.

Monitor(s, t, S, T) – probes site *s* and awaits the current (*S*, *T*) pair of that site. At most *t* tries are performed, then res = Timeout is returned if still no answer arrived.

Introduce(n, s, S) – broadcasts a probe (once) and collects all answers arriving during a certain time window. It returns the vector of sites *s*[0..*n*–1] that answered as well as the corresponding vector of states *S*[0..*n*–1].

3.4 Synchronous sessions

PCP sessions are invoked by a caller site and accepted by a callee site. Once opened, the caller is the sender and the callee the receiver of the session. A phase switch happens when both sites agree to switch from receive to send and from send to receive mode, respectively. Empty phases transmitting no data are

legal. Data transmissions within a phase are buffered to produce optimally filled packets. Timeouts (res = Timeout) due to some failure may occur at any time during a session.

The application code calling the session primitives executes under timing constraints. For the current Ceres implementation 512 bytes within a session must be composed and sent resp. received and consumed within 200ms. A typical transmission of a 512 byte packet (data) takes less than 20ms, a typical transmission of a header only packet (ack) takes less than 2ms. Hence, at least 150ms are available per 512 bytes of data. (Opening a file takes less than 70ms. Also see table in section 4.1.5.)

The procedures *SendBytes* and *ReceiveBytes* add nothing to PCP's functionality but are essential for efficient applications. The additional begin parameter (as opposed to the block operations provided by the standard Files module) has been added to allow for partial transmission of buffers.

Request(id, s, t, n) – request a session with service *id* on site *s* and await an answer. At most *t* tries are performed, then res = Timeout is returned if still no answer arrived. If an answer arrives it affects the result: res = Rejected indicates that *s* is involved in some other session (or awaits a different request) and hence rejected the request; res = BadID indicates that the service *id* is currently not installed on *s*; res = BadName indicates that the name n is currently not bound to *s* (rf. to name service section below). Traps if *id* = *OwnSite*() (see below).

Accept(R, id, s) – explicitly accept session requests. A pending request is rejected if the reject–filter *R(id, s)* returns TRUE, and accepted otherwise. If a request got accepted, res = Done is returned, the session is opened, and the local site is in receive mode; in all other cases res = NoRequest results. Note that pending asynchronous markers may be delivered without notice (by up–calling the appropriate service handlers) when calling *Accept*. *Accept* should not be called from within a marker handler, as this may cause recursive calls to the handler.

SendPhase – switch phase by turning a site engaged in a session from receive to send mode. Traps if not currently in send mode.

Send(x) – send a single byte *x*

SendBytes(x, b, l) – send bytes *x*[*b*..*b*+*l*–1]; same semantics as repeated call to *Send*. Degenerated cases with *l* ≤ 0 are ignored. Index ranges are checked and may cause a trap.

ReceivePhase(dt) – switch phase by turning a site engaged in a session from send to receive mode. If a delay time *dt* > 0 is passed, ReceivePhase requests an AAK for the last STP packet and allows additional *dt*/300 seconds before it times out. This gives the other site more time to compute an answer before sending it back. Traps if not currently in receive mode.

Receive(x) – receive a single byte *x*. Returns *eof* = TRUE if no more byte available in current phase.

ReceiveBytes(x, b, l) – receive bytes *x*[*b*..*b*+*l*–1]; same semantics as repeated call to *Receive*. Note that this implies the possibility of a partial receive *x*[*b*..*e*], *e* < *b*+*l*–1 if *eof* = TRUE results. Degenerated cases with *l* ≤ 0 are ignored. Index ranges are checked and may cause a trap.

Close – closes current session. Should be called even if some error result has been signalled during the session. May return with res = BadReceive if the receiver closes a session before having received all data of that phase.

3.5 Integrated name service

PCP maintains a local name binding list N of names bound to the local site. The name u currently set by the user (variable *Oberon.User*, set by *System.SetUser*) is always bound, i.e. u ∈ N. This binding cannot be

changed other than by changing the set user name. Additionally, an association cache C of pairs (name, site) is maintained for names that have been found to be bound to some remote site. (For a way to automatically keep this cache consistent, refer to Request above.)

OwnSite() – return site id of local site.

BindSite(n) – try to bind *n* to local site. If $n \in N$: return res = BadName; if $n \notin N$: Include(N, *n*), then check if some other site on the network can be found with that n bound to it, if one is found: Exclude(N, *n*) again and return res = BadName. This test is not perfectly reliable. If globally unique names are absolutely required and cannot be guaranteed administratively, a higher–level protocol should be used.

UnbindSite(n) – unbind name *n* from local site, i.e. Exclude(N, *n*).

FindSite(n, t, s) – lookup to which site *s* the name *n* is bound where at most *t* tries are performed. If the name is locally bound, i.e. $n \in N$, *s* = OwnSite() is returned. If the name is not found at all, res = Timeout results. A name that has been found to be bound to some remote site is locally cached together with that site's id. Hence, repeated calls to FindSite tend to be efficient.

InvalCachedName(n) – a pair $(n, s) \in C$ is removed. Idempotent operation.

InvalCachedSite(s) – all pairs $(n, s) \in C$ are removed. This is typically done after deciding that remote site s has failed. Idempotent operation.

4. Msgs Module

```
DEFINITION Msgs;
    IMPORT
        Files;

    CONST
        MInvalid = -1;  MNew = 0;  MCreate = 1;  MReady = 2;  MOpen = 3;  (*message states*)

    TYPE
        Msg = POINTER TO MsgDesc;
        MsgDesc = RECORD
            len: LONGINT
        END;

    (*create message*)
    PROCEDURE OpenNew(M: Msg);
    PROCEDURE Write(M: Msg; x: BYTE);
    PROCEDURE WriteBytes(M: Msg; VAR x: ARRAY OF BYTE; beg, len: LONGINT);
    PROCEDURE WriteInt(M: Msg; n: LONGINT);
    PROCEDURE WriteString(M: Msg; s: ARRAY OF CHAR);
    PROCEDURE AttachPiece(M: Msg; f: Files.File; name: ARRAY OF CHAR; beg, end: LONGINT);
    PROCEDURE AttachMsg(M: Msg; m: Msg);

    (*read messages*)
    PROCEDURE OpenOld(M: Msg);
    PROCEDURE Read(M: Msg; VAR x: BYTE);
    PROCEDURE ReadBytes(M: Msg; VAR x: ARRAY OF BYTE; beg, len: LONGINT);
    PROCEDURE ReadInt(M: Msg; VAR n: LONGINT);
    PROCEDURE ReadString(M: Msg; VAR s: ARRAY OF CHAR);
    PROCEDURE FetchPiece(M: Msg; VAR f: Files.File; VAR beg, end: LONGINT);
    PROCEDURE FetchMsg(M: Msg; VAR m: Msg);
    PROCEDURE Close(M: Msg);

    (*handle messages*)
    PROCEDURE State(M: Msg): INTEGER;
    PROCEDURE Compact(M: Msg): BOOLEAN;
```

```
    (*message transmission*)
    PROCEDURE Send(M: Msg);
    PROCEDURE Receive(M: Msg);
    PROCEDURE SendMark(M: Msg; id: PCP.ServiceID; to: PCP.Site);
    PROCEDURE ReceiveMark(M: Msg; VAR time: LONGINT);
END Msgs.
```

Module Msgs implements an abstract data type Msg useful for creating, sending, receiving, and consuming structured messages. The plain data size in bytes contained within a message m is indicated by its public field m.len (read only). A message is any one of five possible states. A message (or an extension of it) freshly allocated in the heap has state MNew. Once opening the message to write into it, it is in state MCreate. After completing a message and closing it, the state becomes MReady. Such a message may be sent or opened for reading (state MOpen). When receiving a message fails, the resulting state is MInvalid. After reading a message it may be closed again to allow for repeated reading of the same message. State violations cause a trap.

4.1 Creating messages

OpenNew(m) – open message *m* for writing to it.

Write(m, x) – write byte *x* to message *m*.

WriteBytes(m, x, b, l) – write bytes *x*[*b*..*b*+*l*–1] to message *m*. Degenerated cases with *l* ≤ 0 are ignored. Index ranges are checked and may cause a trap.

WriteInt(m, n) – write integer *n* to message *m*.

WriteString(m, s) – write zero-terminated string *s* to message *m*.

AttachPiece(m, f, n, b, e) – attach file piece <*f*, *b*, *e*> referring to file *f*, range [*beg*..*end*) to message *m*. If transmitted to a remote site, the file will be created using name *n*.

AttachMsg(m, m') – attach nested message *m'* to message *m*. (As message *m'* must be in state MReady, while message *m* must be in state MCreate, cyclic inclusion is not possible.)

4.2 Reading messages

OpenOld(m) – open message *m* for reading from it.

Read(m, x) – read byte *x* from message *m*.

ReadBytes(m, x, b, l) – read bytes *x*[*b*..*b*+*l*–1] from message *m*.

ReadInt(m, n) – read integer *n* from message *m*.

ReadString(m, s) – read zero-terminated string *s* from message *m*.

FetchPiece(m, f, b, e) – fetch file piece <*f*, *b*, *e*> refering to file *f*, range [*beg*..*end*) from message *m*. If *m* got transmitted from a remote site, the file has been created using name *n* but *not* registered with the directory.

FetchMsg(m, m') – fetch nested message *m'* from message *m*.

4.3 Message attributes

State(m) – return current state of *m*.

Compact(m) – return whether *m* could be send as a marker (rf. SendMark below).

4.4 Message transmission

Send(m) – send message *m*. Assumes active PCP send phase.

Receive(m) – receive message *m*. Assumes active PCP receive phase.

SendMark(m, id, s) – unreliably send message *m* as marker to the service *id* on site *s*.

ReceiveMark(m, T) – receive message *m* as marker. The marker arrival time is *T*. To be called within service handler (rf. to PCP.ReceiveMark).

Appendix

A.1 File Transfer Service Example

The file transfer sample application is given in full source form. Compared to the version developed in the fourth section, it contains faster and more solid file transfer primitives that make use of block operations provided by modules Files and PCP. In order to do this the length of the file is transmitted just before transmitting the file. This leads to a slight modification of the syntax given in 4.1.1.

File = filelength {byte}.

Here, filelength is a four byte integer.

```
MODULE FileTransfer;
    IMPORT
        Files, PCP;

    CONST
        Ok* = 0; Unavail* = 1; Busy* = 2; NoPerm* = 3; NotFound* = 4; Failed* = 5; (*result codes*)
        SendID = 41; ReceiveID = 43; (*some unique service ids*)
        Tries = 3;

    VAR
        readPerm, writePerm: BOOLEAN; (*remote file read/write permissions*)


    (* transport primitives *)

    PROCEDURE SendStr(VAR s: ARRAY OF CHAR);
        VAR
            i: INTEGER;
    BEGIN
        i := -1; REPEAT INC(i) UNTIL s[i] = 0X;
        PCP.SendBytes(s, 0, i)
    END SendStr;

    PROCEDURE SendFile(F: Files.File; VAR done: BOOLEAN);
        VAR R: Files.Rider; buf: ARRAY 512 OF BYTE; len: LONGINT; blen: INTEGER;
    BEGIN
        Files.Set(R, F, 0); len := Files.Length(F);
        PCP.SendBytes(len, 0, SIZE(LONGINT)); blen := 512;
        LOOP
            IF len < 512 THEN blen := SHORT(len) END;
            Files.ReadBytes(R, buf, blen); IF R.res # 0 THEN EXIT END;
            PCP.SendBytes(buf, 0, blen); IF PCP.res # PCP.Done THEN EXIT END;
            DEC(len, 512); IF len <= 0 THEN EXIT END
        END;
        done := (R.res = 0) & (PCP.res = PCP.Done)
    END SendFile;


    PROCEDURE ReceiveStr(VAR s: ARRAY OF CHAR);
        VAR
            i: INTEGER;
    BEGIN
        i := -1; REPEAT INC(i); PCP.Receive(s[i]) UNTIL (s[i] = 0X) OR (PCP.res # PCP.Done)
    END ReceiveStr;
```

```
PROCEDURE ReceiveFile(VAR name: ARRAY OF CHAR; VAR done: BOOLEAN);
    VAR
        R: Files.Rider;  F: Files.File;  buf: ARRAY 512 OF BYTE;  len: LONGINT;  blen: INTEGER;
BEGIN
    F := Files.New(name); PCP.ReceiveBytes(len, 0, SIZE(LONGINT)); blen := 512;
    LOOP
        IF len < 512 THEN blen := SHORT(len) END;
        PCP.ReceiveBytes(buf, 0, blen); IF PCP.res # PCP.Done THEN EXIT END;
        Files.WriteBytes(R, buf, blen); IF R.res # 0 THEN EXIT END;
        DEC(len, 512); IF len <= 0 THEN EXIT END
    END;
    done := (R.res = 0) & (PCP.res = PCP.Done)
END ReceiveFile;


(* servers *)

PROCEDURE *SendServer(site: PCP.Site);
    VAR
        F: Files.File;  name: ARRAY 32 OF CHAR;  done: BOOLEAN;
BEGIN
    ReceiveStr(name); PCP.SendPhase;
    IF readPerm THEN
        F := Files.Old(name);
        IF F # NIL THEN PCP.Send(Ok); SendFile(F, done)
        ELSE PCP.Send(NotFound)  (*file not found*)
        END
    ELSE PCP.Send(NoPerm)  (*read permission denied*)
    END;
    PCP.Close
END SendServer;

PROCEDURE *ReceiveServer(site: PCP.Site);
    VAR
        name: ARRAY 32 OF CHAR;  ch: CHAR;  done: BOOLEAN;
BEGIN
    ReceiveStr(name); PCP.SendPhase;
    IF writePerm THEN PCP.Send(Ok); PCP.ReceivePhase(0); ReceiveFile(name, done)
    ELSE PCP.Send(NoPerm)  (*write permission denied*)
    END;
    PCP.Close
END ReceiveServer;


(*file transfer*)

PROCEDURE Permissions*(read, write: BOOLEAN);
BEGIN
    readPerm := read; writePerm := write
END Permissions;
```

```
    PROCEDURE Send*(site, file: ARRAY OF CHAR; VAR res: INTEGER);
        VAR
            s: PCP.Site;  F: Files.File;  stat: SHORTINT;  done: BOOLEAN;
    BEGIN
        F := Files.Old(file);
        IF F # NIL THEN
            PCP.FindSite(site, Tries, s);
            IF PCP.res = PCP.Done THEN
                PCP.Request(ReceiveID, s, Tries, site); SendStr(file);
                PCP.ReceivePhase(0); PCP.Receive(stat);
                IF (PCP.res = PCP.Done) & (stat = Ok) THEN
                    PCP.SendPhase; SendFile(F, done)
                END;
                PCP.Close;
                IF done & (PCP.res = PCP.Done) THEN res := stat ELSE res := Failed END
            ELSE res := Unavail
            END
        ELSE res := NotFound  (*file not found*)
        END
    END Send;

    PROCEDURE Receive*(site, file: ARRAY OF CHAR; VAR res: INTEGER);
        VAR
            s: PCP.Site;  stat: SHORTINT;  done: BOOLEAN;
    BEGIN
        PCP.FindSite(site, Tries, s);
        IF PCP.res = PCP.Done THEN
            PCP.Request(SendID, s, Tries, site); SendStr(file);
            PCP.ReceivePhase(0); PCP.Receive(stat);
            IF (PCP.res = PCP.Done) & (stat = Ok) THEN
                ReceiveFile(file, done)
            END;
            PCP.Close;
            IF done & (PCP.res = PCP.Done) THEN res := stat ELSE res := Failed END
        ELSE res := Unavail
        END
    END Receive;

BEGIN
    readPerm := FALSE; writePerm := FALSE;
    PCP.Install(SendID, SendServer); PCP.Install(ReceiveID, ReceiveServer)
END FileTransfer
```

(This source has 120 lines and on Ceres compiles into 1300 bytes object code.)

A.2 Communicating Logical Nodes Example

The Comm sample module is given in full source form.

```
MODULE Comm;
    IMPORT
        PCP*, Msgs*, Oberon;

    CONST
        BadNode* = -10;  (*other result values as defined by PCP module*)

    TYPE
        Node* = LONGINT;
        Receiver* = PROCEDURE(from: Node; to: Node; M: Msgs.Msg);
        Notifier* = PROCEDURE(from: Node; to: Node; M: Msgs.Msg);

    VAR
        res*: INTEGER;
```

```
CONST
    NameServID = 5; TransportServID = 6;
    Mask = 256;  Tries = 3;

TYPE
    Name = ARRAY 32 OF CHAR;

    Entry = POINTER TO EntryDesc;
    EntryDesc = RECORD
        next: Entry;
        Receive: Receiver;
        Notify: Notifier;
        name: Name;
        node: Node
    END;

    QItem = POINTER TO QItemDesc;
    QItemDesc = RECORD
        next: QItem;
        from, to: Node;
        M: Msgs.Msg
    END;

    Q = RECORD  (*empty iff head=NIL*)
        head, tail: QItem
    END;

VAR
    started: BOOLEAN;
    ownSite: PCP.Site;
    nextNode: INTEGER;
    task: Oberon.Task;
    names: Entry;  (*ring with head*)
    out, deliver: Q;


PROCEDURE ASSERT(pred: BOOLEAN);
BEGIN
    IF ~pred THEN HALT(99) END
END ASSERT;

PROCEDURE FindName(VAR name: ARRAY OF CHAR; VAR entry: Entry);    (*not found iff entry = NIL*)
BEGIN
    entry := names↑.next; WHILE (entry # names) & (entry↑.name # name) DO entry := entry↑.next END;
    IF entry = names THEN entry := NIL END
END FindName;

PROCEDURE FindNode(node: Node; VAR entry: Entry);              (*not found iff entry = NIL*)
BEGIN
    names↑.node := node; entry := names↑.next; WHILE entry↑.node # node DO entry := entry↑.next END;
    IF entry = names THEN entry := NIL END
END FindNode;

PROCEDURE Put(VAR queue: Q; item: QItem);
BEGIN
    IF queue.head = NIL THEN queue.head := item ELSE queue.tail↑.next := item END;
    queue.tail := item
END Put;
```

```
(* servers *)

PROCEDURE *NameServer(from: PCP.Site);
    VAR
        name: Name; node: Node; entry: Entry;
BEGIN
    PCP.ReceiveBytes(name, 0, SIZE(Name)); node := -1;
    IF PCP.res = PCP.Done THEN
        FindName(name, entry);
        IF entry # NIL THEN node := entry↑.node END
    END;
    PCP.SendPhase; PCP.SendBytes(node, 0, SIZE(Node));
    PCP.Close
END NameServer;

PROCEDURE *TransportServer(from: PCP.Site);
    VAR
        item: QItem; entry: Entry;
BEGIN
    NEW(item); res := PCP.Done;
    PCP.ReceiveBytes(item↑.from, 0, SIZE(Node)); PCP.ReceiveBytes(item↑.to, 0, SIZE(Node));
    IF PCP.res = PCP.Done THEN FindNode(item↑.to, entry) ELSE entry := NIL END;
    PCP.SendPhase; IF entry # NIL THEN PCP.Send(PCP.Done) ELSE PCP.Send(BadNode) END;
    PCP.ReceivePhase(0);
    NEW(item↑.M); Msgs.Receive(item↑.M);
    IF PCP.res = PCP.Done THEN Put(deliver, item) END;
    PCP.Close
END TransportServer;


(* task *)

PROCEDURE *Task;
    VAR
        item: QItem; entry: Entry; to: Node; dest: PCP.Site; code: SHORTINT;
BEGIN
    item := deliver.head;
    IF item # NIL THEN (*deliver first message in queue*)
        FindNode(item↑.to, entry);
        IF entry # NIL THEN entry↑.Receive(item↑.from, item↑.to, item↑.M) END;
        deliver.head := deliver.head↑.next
    END;
    item := out.head;
    IF item # NIL THEN (*send first message in queue*)
        dest := SHORT(SHORT(item↑.from MOD Mask)); res := PCP.Done;
        PCP.Request(dest, TransportServID, Tries, "");
        PCP.SendBytes(item↑.from, 0, SIZE(Node)); PCP.SendBytes(item↑.to, 0, SIZE(Node));
        PCP.ReceivePhase(0); PCP.Receive(code);
        IF (PCP.res = PCP.Done) & (code = PCP.Done) THEN Msgs.Send(item↑.M) END;
        PCP.Close;
        IF PCP.res # PCP.Done THEN res := PCP.res ELSE res := code END;
        IF res # PCP.Done THEN (*try to notify that destination is unreachable*)
            FindNode(item↑.from, entry);
            IF entry # NIL THEN res := code; entry↑.Notify(item↑.from, item↑.to, item↑.M) END
        END;
        out.head := out.head↑.next
    END
END Task;
```

```
(* commands *)

PROCEDURE Start*;
BEGIN
    IF ~started THEN
        Oberon.Install(task);
        PCP.Install(NameServID, NameServer);
        PCP.Install(TransportServID, TransportServer);
        started := TRUE
    END
END Start;

PROCEDURE Stop*;
BEGIN
    IF started THEN
        Oberon.Remove(task);
        PCP.Remove(NameServID);
        PCP.Remove(TransportServID);
        started := FALSE
    END
END Stop;


PROCEDURE Find*(name: ARRAY OF CHAR; VAR node: Node);
    VAR
        entry: Entry;  site: PCP.Site;  i: INTEGER;
BEGIN
    PCP.FindSite(name, Tries, site);
    IF PCP.res = PCP.Done THEN
        IF site = ownSite THEN
            FindName(name, entry);
            IF entry # NIL THEN node := entry↑.node; res := PCP.Done
            ELSE res := BadNode
            END
        ELSE
            PCP.Request(NameServID, site, Tries, name);
            IF PCP.res = PCP.BadName THEN
                PCP.FindSite(name, Tries, site); PCP.Request(NameServID, site, Tries, name)
            END;
            PCP.SendBytes(name, 0, SIZE(Name)); PCP.ReceivePhase(0);
            PCP.ReceiveBytes(node, 0, SIZE(Node)); PCP.Close
        END
    END;
    IF (PCP.res # PCP.Done) OR (node = -1) THEN node := -1; res := BadNode END
END Find;

PROCEDURE Home*(node: Node): PCP.Site;
BEGIN
    RETURN SHORT(SHORT(node MOD Mask))
END Home;


PROCEDURE New*(name: ARRAY OF CHAR; R: Receiver; N: Notifier; VAR node: Node);
    VAR
        entry: Entry;
BEGIN
    PCP.BindSite(name);
    IF PCP.res # PCP.Done THEN node := -1; res := BadNode
    ELSE
        NEW(entry); entry.Receive := R; entry.Notify := N;
        COPY(name, entry↑.name); entry↑.node := nextNode; INC(nextNode, Mask);
        entry↑.next := names↑.next; names↑.next := entry;
        node := entry↑.node; res := PCP.Done
    END
END New;
```

```
    PROCEDURE Close*(node: Node);
        VAR
            prev, entry: Entry;
    BEGIN  ASSERT( node MOD Mask = ownSite );
        names↑.node := node; prev := names; WHILE prev↑.next↑.node # node DO prev := prev↑.next END;
        IF prev # names THEN
            entry := prev↑.next; prev↑.next := entry↑.next; PCP.UnbindSite(entry↑.name)
        END
    END Close;


    PROCEDURE Send*(from, to: Node; M: Msgs.Msg);
        VAR
            item: QItem;prev, entry: Entry;
    BEGIN  ASSERT( (from MOD Mask = ownSite) & (Msgs.State(M) = Msgs.MReady) );
        FindNode(from, entry);  ASSERT( entry # NIL );
        NEW(item); item↑.next := NIL; item↑.from := from; item↑.to := to; item↑.M := M;
        IF to MOD Mask = ownSite THEN Put(deliver, item) ELSE Put(out, item) END
    END Send;

BEGIN
    ownSite := PCP.OwnSite(); nextNode := ownSite;
    NEW(task); task.handle := Task;
    NEW(names); names↑.next := names;
    out.head := NIL; deliver.head := NIL;
    started := FALSE; Start
END Comm.
```

(This source has 200 lines and on Ceres compiles into 950 bytes object code.)

Gelbe Berichte des Departements Informatik

115	W. Gander G.H. Golub D. Gruntz	Solving Linear Equations by Extrapolation
116	B. Sanders	Stepwise Refinement of Mixed Specifications of Concurrent Programs
117	N. Wirth	Modula-2 and Object-Oriented Programming. Drawing Lines, Circles, and Ellipsis in a Raster. Flintstone.
118	H.-J. Schek H.-B Paul M.H. Scholl G. Weikum	The DASDBS Project: Objectives, Experiences, and Future Prospects
119	J. Gutknecht	The Oberon Guide
120	D. Mey	A Predicate Calculus with Control of Derivations
121	H.P. Frei P. Schäuble M.F. Wyle	The Assessment of Information Retrieval Algorithms
122	P. Läuchli	An Elementary Theory for Planar Graphs
123	B. Wüthrich	Detecting Inconsistencies in Deductive Databases
124	C. Pfister	The Graphics Editor Condor The Layout System Pedro
125	R. Crelier	OP2: A Portable Oberon Compiler
126	A. Szyperski	Network Communication in the Oberon Environment