Connecting Oberon

Guideline and Programming Interfaces

Markus Hof

Report 7
April 1996

Author's address:

Institut für Informatik (Systemsoftware)
Johannes Kerpler Universität Linz
A- 4040 Linz/Auhof

email:
hof@ssw.uni-linz.ac.at

© 1996 Institut für Informatik (Systemsoftware),
Johannes Kepler Universität Linz

Abstract

Oberon [WiGu89] is a modular, single-threaded operating system for single-user operation of stand-alone workstations. It is used in daily work as well as in programming courses. As a stand-alone system, most Oberon implementations lack network support. This report describes two modules, which extend Oberon with concurrency and network capabilities.

Module *Threads* implements the abstraction of threads, as well as mechanisms for thread synchronisation. Threads are scheduled preemptively and are useful to run network servers in the background.

Module *Network* offers an abstract way to access networks. It is an abstraction corresponding to the transport layer of the OSI model [Tan89]. *Network* offers multiple logical hosts on one physical site and channels for communications between hosts. Actual network protocols have to be supplied in other modules. Channels can be extended and adapted to the needs of clients and servers.

Contents

1 Introduction
2 Threads
  2.1 Basic characteristics
  2.2 Programming Interfaces
  2.3 Exception handling
  2.4 Restrictions
  2.5 Modifications to the Oberon System
  2.6 Example: Producer Consumer
3 Network
  3.1 Overview
  3.2 System Model
  3.3 Connecting two Hosts
  3.4 Programming Interface
  3.5 Failure Semantics
  3.6 Example: Time Server
  3.7 Adding a Protocol
4 Related Work
  4.1 Hermes
  4.2 Amoeba Microkernel
  4.3 Chorus
  4.4 DCE
  4.5 Mach Microkernel
  4.6 Summary
5 Conclusions
Appendix A: How to get modules *Threads* and *Network*
Appendix B: References

1 Introduction

The results described in this report are part of a research project, which adds distributed objects to the Oberon system. The report describes the module *Network* introducing network access and the module *Threads* providing preemptive processes, respectively. Together these modules form the basic building blocks for distributed programming and distributed objects. Our system was designed with the following goals in mind:

- It should be as simple as possible without restricting clients.
- It should be extensible.
- It should conform to the original Oberon system.
- It should keep platform dependencies to a minimum and hide them from clients.
- It should be adaptable to arbitrary networks and protocols.
- It should provide transparent access to other hosts regardless of actual protocol and network topology.
- It should decouple logical hosts from physical sites.

Together with Oberon's memory management and I/O system we obtain the support necessary for distributed computing.

2 Threads

The ETH Oberon System [WiGu89] is a single user multitasking system. It was not designed to support multi user operation and preemptive multitasking. Still the system allows a user to work on multiple tasks simultaneously, i.e. it supports cooperative multitasking. With cooperative multitasking, task switches are triggered by user actions. A global loop waits for input events and dispatches them to the respective applications. Applications handle the events and return control to the input loop as soon as possible. As long as an application has control no further user input is accepted and all other applications are suspended. This scheme for multitasking is practical as long as all user-initiated tasks are short. However for longer tasks such as formatting disks, compiling several modules, or accessing a database it is an unsatisfactory solution. Besides blocking further user input, it effectively removes all computing power from other tasks for an indefinite amount of time.

In single user mode this may be acceptable since the user can normally predict the response times. However, there are situations where more powerful mechanisms are needed. In a network several users may access the system concurrently asking for information, processor time or other local resources. Remote access is not predictable but may occur at any time. Cooperative multitasking would allow one user to lock out all other users effectively turning down their requests. A remote user cannot tell whether a request remains unanswered because it was lost, because the receiver is busy, or because the receiver is just not reachable. This behaviour is not acceptable in a network. We need a more predictable behaviour. Therefore we introduce the notion of threads. Threads are scheduled preemptively, allowing a system to respond to incoming requests immediately.

2.1 Basic characteristics

We designed the threads subsystem to be as simple as possible. We refrained from adding goodies such as priorities or memory protection.

- Memory protection was omitted mainly for two reasons: Firstly, Oberon is a type safe language, which guarantees that no thread corrupts memory belonging to other threads. Secondly, we wanted to extend the Oberon System while remaining compatible with it. Since all Oberon tasks run in the same address space, introducing protected memory would have forced us to rewrite large portions of the system.

- Thread priorities, while useful, add considerably to the complexity of a threads package. Since our threads are mainly used for network services, any local timing considerations are quite obsolete, as the network delay and latency overshadow most delays introduced locally.

Of course one could argue that our threads are too simple. However, for offering network services this simple package is sufficient.

The threads package adds two abstractions to the Oberon System: Preemptively scheduled threads and semaphores for synchronisation of threads. Unfortunately scheduling is system dependent. Therefore the module *Threads* is not portable. Up to now, there exist versions for the PowerMacintosh and for Windows NT.

2.2 Programming Interface

```
DEFINITION Threads;

    TYPE
        Semaphore = RECORD
            PROCEDURE (VAR s: Semaphore) Init (val: INTEGER);
            PROCEDURE (VAR s: Semaphore) Down;
            PROCEDURE (VAR s: Semaphore) Up;
            PROCEDURE (VAR s: Semaphore) TryDown (): BOOLEAN;
        END;

        Thread = POINTER TO ThreadDesc;
        ThreadDesc = RECORD END;

    (* thread handling *)
    PROCEDURE Start (thread: Thread; proc: PROCEDURE; stackLen: LONGINT);
    PROCEDURE ActiveThread (): Thread;
    PROCEDURE OberonThread (): Thread;
    PROCEDURE Kill (thread: Thread);
    PROCEDURE Sleep (milliSeconds: LONGINT);

    (* compatibility support *)
    PROCEDURE RegisterReentrant (modName: ARRAY OF CHAR);
    PROCEDURE Schedule (allowed: BOOLEAN);

END Threads.
```

Operations for thread handling

- *Start (thread, proc, stackLen)* initialises and starts a new thread with a stack of *stackLen* bytes. Execution is initiated by calling procedure *proc*. *thread* must point to an already allocated thread structure.
- *thread := ActiveThread ()* returns the currently running thread.
- t*hread := OberonThread ()* returns the main Oberon thread (the one which exists when the Oberon System is started).
- *Kill (thread)* terminates the specified thread. The Oberon thread must not be killed.
- *Sleep (milliSeconds)* puts the calling thread to sleep for the specified amount of time. No guarantee on the precision is given (see Section 2.4 Restrictions), except that the thread sleeps for at least the given interval. *Sleep(0)* can be used to yield the processor and pass control to another thread.

Compatibility support

We have to make sure that thread switches only occur when no global state can be invalidated and the currently executing module is reentrant. In Oberon, most system modules are not reentrant. Therefore we have to tell the system which modules are reentrant so that thread switches may occur within them.

- *RegisterReentrant (modName)* allows preemptive thread switches within this module.
- *Schedule (allowed)* allows thread switches if *allowed* is TRUE, otherwise no further thread switches take place until *Schedule (TRUE)* is called.

Synchronisation

A *Semaphore* allows the management of shared resources. It is a protected counter that can be used to restrict the number of threads operating concurrently in a critical region.

- *sem.Init (val)* initialises *sem* to the value *val* which must be greater or equal to 0.
- *sem.Down*. If the value of *sem* is greater than 0 it is decremented by 1, otherwise the calling thread is suspended waiting for *sem* to go up.
- *sem.Up.* If threads are waiting for *sem*, the first thread in the queue is resumed. Otherwise the value of *sem* is incremented by 1.
- *b := sem.TryDown ().* If the value of *sem* is greater than 0 it is decremented by 1 and the operation returns with TRUE. Otherwise the operation returns with FALSE.

2.3 Exception handling

When an exception (programmed halt, division by zero, NIL-reference, ...) is raised, the Oberon system displays a trap viewer with some trap information and returns control to the global input loop. With multiple threads this behaviour has to be changed, since only the Oberon thread is allowed to run the global input loop. Therefore we distinguish two cases: If a trap occurred in the Oberon thread we maintain the behaviour of the Oberon system. If a trap occurred in some other thread, we display the trap information and then kill this thread.

2.4 Restrictions

The Oberon System was designed for cooperative multitasking. Therefore it does not protect its resources (e.g. global variables) or states against race conditions. To avoid race conditions one has to disable multi-threading within modules of the Oberon system. The threads package handles modules as non-reentrant by default. As long as a thread stays in a non-reentrant module, scheduling is disabled until the thread runs into a reentrant module. The same applies to user modules. If a user module is not specified as reentrant, no thread switches may occur within it. As long as the current thread stays in this module, other threads are blocked.

However, if a non-reentrant module calls procedures from a reentrant module, it can happen that it is reentered nevertheless. In particular, one or several schedules may occur, whenever an operation of a system module is called. It is the programmer's task to ensure correct execution when calling procedures from other modules.

We encourage programmers to write reentrant modules, by avoiding global resources or by guarding them with semaphores.

Threads does not check for stack overflows. It is the responsibility of the programmer to specify sufficient stack space.

2.5 Modifications to the Oberon System

The Oberon System has an automatic garbage collector that periodically collects all unreferenced memory blocks. In order to do so, it marks all blocks that can be reached from life pointers. Such pointers may reside on the procedure stack and since there is a stack for every thread, the garbage collector has to traverse all of them. Therefore the garbage collector had to be modified to scan multiple stacks instead of just one.

Whenever a thread is started its stack is registered for garbage collection. When a thread is killed its stack is removed. The Oberon module *Kernel* had to be modified as follows:

```
DEFINITION Kernel;
    .........
    PROCEDURE AddStack (beg, end: LONGINT);
    PROCEDURE RemoveStack (pos: LONGINT);
END Kernel.
```

AddStack defines a new stack in the specified memory area, and *RemoveStack* removes the stack containing the address *pos*.

2.6 Example

We show how to implement a producer consumer system. The producer generates random numbers and writes them into a buffer, which can hold at most *n* numbers. The consumer reads the numbers from the buffer in first-in-first-out order. If no number is available, the consumer blocks. There are three points for synchronisation: The consumer may only read if there are full slots in the buffer; the producer may only write if there are empty slots in the buffer; the buffer should not be modified by both threads at the same time in order to avoid race conditions.

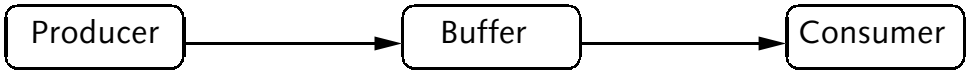

Figure 1. Overview

We implement a ring buffer with an array of *n* elements and protect it with a semaphore allowing only one access at any time. To avoid buffer overflow or underflow we introduce two semaphores *fullSlots* and *emptySlots* with 0 and *n* as initial values, respectively.

First we initiate the whole scenario by creating the semaphores and starting the producer and consumer threads:

```
CONST
    n = ...;
VAR
    emptySlots, fullSlots, lock: Threads.Semaphore;
    consumer, producer: Threads.Thread;
BEGIN
    fullSlots.Init(0); (* no full slots *)
    emptySlots.Init(n); (* all slots free *)
    lock.Init(1);

    NEW (producer); Threads.Start (producer, Produce, 2000);
    NEW (consumer); Threads.Start (consumer, Consume, 2000)
```

The two threads (*Produce* and *Consume*) are started and produce respectively consume the numbers, while protecting the buffer with *lock* and waiting for *fullSlots* respectively *emptySlots*.

```
PROCEDURE Produce;
BEGIN
   LOOP
      r := new random number;
      emptySlots.Down;
      lock.Down;
      ... put r into buffer
      lock.Up;
      fullSlots.Up
   END
END Produce;
```

```
PROCEDURE Consume;
BEGIN
   LOOP
      fullSlots.Down;
      lock.Down;
      ... get r from buffer
      lock.Up;
      emptySlots.Up;
      ... use r
   END
END Consume;
```

3 Network

Module *Network* is an abstraction layer for network access. It does not implement a particular network protocol, but offers a general view on networks regardless of their actual topologies and protocols. It builds a logical layer (hosts connected through channels) on top of the physical layer (sites connected through links). The logical layer offers the following abstractions:

- A *Host* is a logical computer reachable via *Network*. Any physical computer (called *site*) may contain several hosts. Two hosts can connect transparently without knowledge of the underlying protocol and the sites hosting them. As a special case both hosts may reside on the same site.
  Note: The terms used in literature for logical and physical computers are inconsistent (site, host, node, location, ...). Therefore we chose terms which we considered appropriate.

- A *Protocol* connects hosts for sending and receiving information. *Protocols* may be added at run-time to support different types of networks (Ethernet, ATM, ...) and network protocols (TCP, AppleTalk, ...) [Coul94]. *Network* implements only a protocol to connect two local hosts (two hosts on same site).

- A *Channel* connects two hosts bi-directionally. It achieves this by choosing automatically the suiting protocol, which handles the network dependent details of the actual connection. The hosts may reside on the same site, or on different sites connected through physical links. Channels guarantee error-free data transmission. After an unrecoverable error the channel state is set accordingly and all channel operations will return immediately (for details see Section 3.5 about failure semantics). In order to re-establish a faulty channel for communication, it has to be re-connected to the other host.

We distinguish between the physical layer of sites connected through links and the abstract layer of hosts connected through channels. Network servers and clients need not know about sites and links, but only about abstract hosts and channels. Only protocol implementations, actually managing the network, know about physical sites and links.

Network offers functionality for four different kinds of clients:

- New network protocols. These protocols manage their connections, register newly reachable hosts, and remove old hosts.

- Network servers offering services to other hosts.

- Network clients accessing services on other hosts.

- Hosts broadcasting to all hosts.

3.1 Overview

To actually transmit data over a network and to handle incoming data, one has to extend the abstractions offered by module *Network*. Clients and servers view all hosts as a homogeneous set independent of the actual protocol needed to access them.

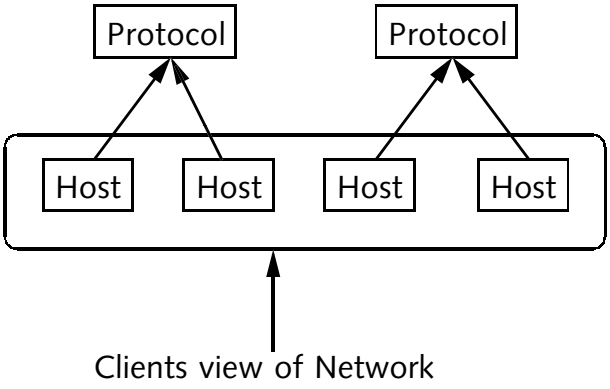

Figure 2. Host set

Network is built on top of the Oberon system heavily using object-oriented features of Oberon-2 [MöWi91]. It is not system dependent and may be ported without any changes. However, it imports the module *Threads* which has to be available on the target platform.

3.2 System Model

Terms used in literature are not consistent. Therefore we explicitly define the terms as used by us:

- *Sites* are the physical units of distribution. Different sites can fail independently.
- *Hosts* are the logical units of distribution. Different hosts do not share memory.
- *Links* are physical communication connections (site to site).
- *Channels* are logical connections (host to host). A channel connecting two host may be the same as a physical link but may also contain a series of links.
- A *Server* is a passive object registered to a host. Servers handle requests received by their hosts. Each host may have several registered servers. Only one server may accept a request.
- A *Request* is generated either by sending a broadcast or by setting up a new channel. *Network* chooses the appropriate protocol to pass the request to the destination host.
- *Request types* define the kind of a request. They are zero terminated strings with up to twenty characters (including zero).
- *Protocols* implement the actual communication over specific networks, e.g. a protocol for communication with TCP on the Internet or a protocol for communication with AppleTalk. They mask all network and protocol specific details.
- *Protocol channels* constitute each protocol's interface to its clients. Each client-server channel communicates over a corresponding protocol channel (see also figure 5). Protocol channels are extensions of normal channels extended to their respective protocol's needs.
- During a *Broadcast* data is sent from a host to all other hosts (incl. itself). The recipients give no acknowledge. Each broadcast has a type by which servers may accept or refuse it.

- Each site has a *default host*. This host is defined automatically when the module *Network* is loaded on a site. Default hosts are bound to their sites. Therefore, servers registered to the default host are specific to the site they are running on.

We view the system as a set of hosts, which are completely connected (every host with every other host) through reliable FIFO communication channels. Security is not guaranteed by *Network*. We assume *fail stop hosts*, which are defined as follows:

- A host either behaves properly or crashes and does nothing, i.e. it doesn't send any messages.
- A host's failure can be detected by other hosts.

On each site, *Network* manages the set of all known hosts located on this site or on other sites. There are three different clients which may scan the host set: network clients, network servers, and protocols. The hosts included in the set are not predefined, but registered dynamically either by an application or by a protocol. With each newly registered host one has to define the protocol which connects the local site with the site of the new host.

An application may register only local hosts (protocol is NIL), in order to add servers to these hosts. A protocol is responsible to register the set of hosts reachable using itself. The protocol has to pass a reference to itself with each registered host.

At each host several servers may be registered. Each of these servers may accept one or several request types and may choose to communicate via different extensions of *Network.Channel*.

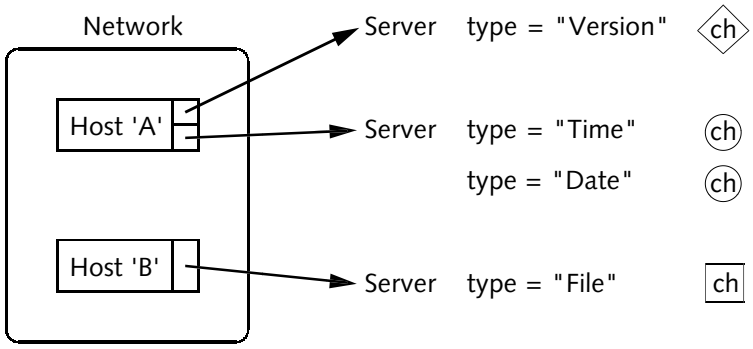

Figure 3. Example setup with two local hosts 'A' and 'B'

When we compare the OSI (open systems interconnection) model with our implementation, module *Network* matches the transport layer. It offers a uniform view to the session level above, regardless of the actual implementation of the lower four levels.

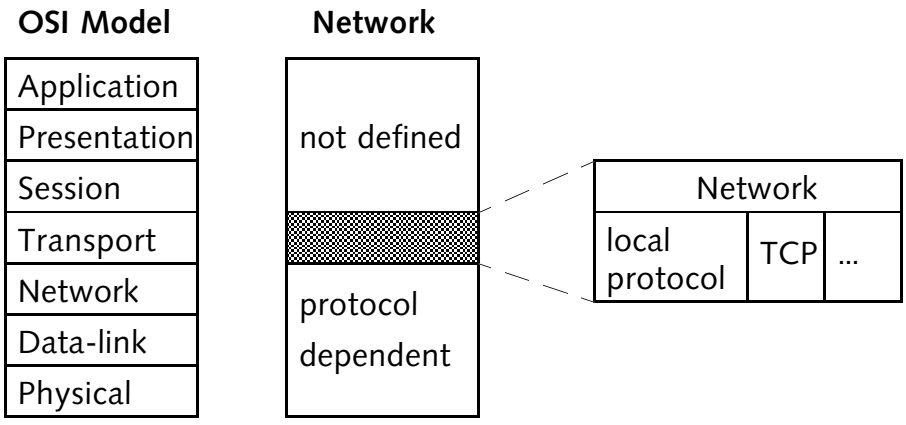

Figure 4. *Network* in the OSI model

Communication over networks is inherently unreliable. Four kinds of errors may occur:

- Data may be corrupted through electrical disturbance, noise or collisions. This may be detected with checksums. Every protocol has to cope with this problem. It either implements its own error correction algorithm or reports the error to the application and invalidates the channels.

- Data may be lost. Again the protocol may handle this error itself or report it to the application.

- Data may arrive out of order. This is mainly a problem with wide area networks (WAN), where different packets of data may be transmitted over different routes. But it may even exist on local area networks, if the receiver requests the retransmission of corrupted data. The protocol should mask out of order arrival by delaying received data until all preceding data are delivered to the application.

- Data may be delayed too long resulting in a time-out at the receiver. This error may even occur if both the sender and the receiver work correctly (e.g. broken link or slow medium). Each protocol has to define an appropriate time-out, after which it assumes the channel to be broken and sets the state accordingly.

Hosts communicate with each other through channels, which should be extensible to satisfy different requirements of clients and protocols. Since both a client and a protocol may want to extend a channel in different ways, it is split into two parts (a client channel and a protocol channel), which are connected in sequence using the Decorator pattern [GHJV95]. The client channel and the protocol channel can be extended individually to achieve a combination of both requirements.

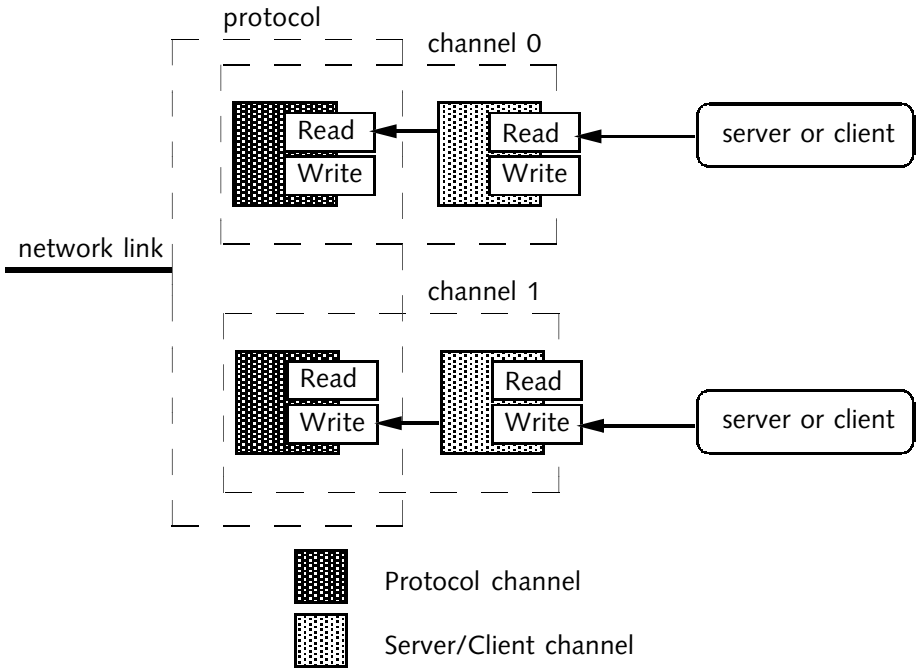

Figure 5. User and protocol channel

For a more detailed description of system models, fail stop hosts, or communication errors refer to [Jalote94, BirRen94].

3.3 Connecting two Hosts

In this section we explain step by step how *Network* establishes a channel *ch* from a client on host 'A' to a server on host 'B'. With each step goes a piece of source code showing some further implementation details. For more information on the interface of module *Network* see section 3.4.

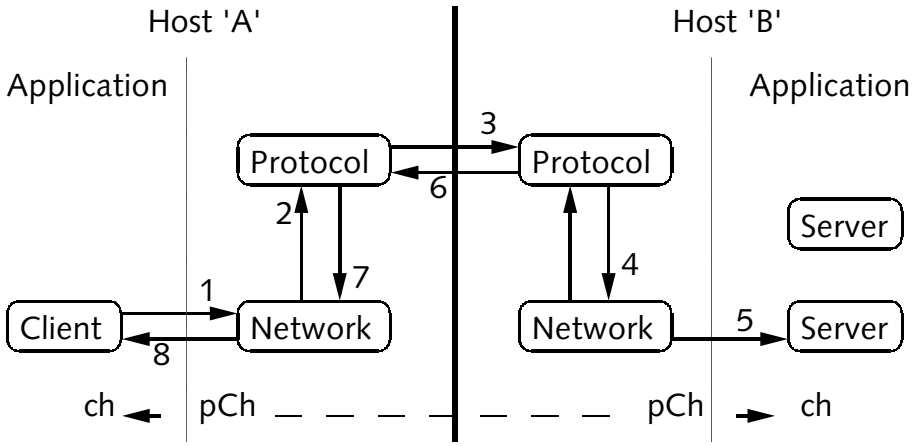

Figure 6. Establishing a channel

1. *Client* allocates a new instance *ch* of type *Network.Channel* or of a self-defined extension, and tries to open the connection by calling *ConnectChannel (ch, 'A', 'B', type, err)*.

```
NEW (ch);
err := Network.ConnectChannel (ch, Network.GetHost ("A"), Network.GetHost ("B"), type, err)
```

2. *Network* delegates the request to the protocol handling connections to host '*B*' (called *to*).

```
to.protocol.Connect (from, to, type, ...)
```

3. *Protocol* transfers the request over the network to its counterpart.

4. *Protocol* on 'B' creates a protocol-specific channel *pCh* for communication with 'A' and calls *Network.Request* to look for an accepting server.

```
Network.Request (from, to, type, pCh, error)
```

5. *Network.Request* tries to find a server handling the received request type. It asks all registered servers to create a server specific channel *ch* which will handle the request. It connects this channel *ch* with the protocol-specific channel *pCh* (using the *Decorator* pattern [GHJV95]). Finally the server is started so that it may handle communication over *ch*.

```
FOR all servers s of host to DO
    ch := s.NewChannel (from, to, type);
    IF ch # NIL THEN
        connect ch and pCh;
        s.Run (ch);
        RETURN
    END
END
```

6. *Protocol* transfers the results of *Network.Request* back over the network to host '*A*'.

7. *Protocol* on '*A*' creates a protocol-specific channel *pCh*, fills it with the information received from '*B*', and returns *pCh* to *Network*.

8. *Network* associates the client-specific channel *ch* with the protocol-specific channel *pCh* and returns control back to the client.

Executing these steps, *Network* connects the client with the server using the correct protocol. Further communications bypass module *Network*. They are handled completely by the chosen protocol and the channel instances allocated by the client and the server, respectively. Note, that all site-dependent handling is done exclusively within the chosen protocol.

3.4 Programming Interface

```
DEFINITION Network;

   IMPORT SYSTEM;

   CONST
      ok = 0;                        (* error codes *)
      hostTabFull = 1;
      doubleHost = 2;
      requestRefused = 3;

   TYPE
      Channel = POINTER TO ChannelDesc;
      ChannelDesc = RECORD
         PROCEDURE (ch: Channel) Read (VAR buf: ARRAY OF SYSTEM.BYTE; n: INTEGER);
         PROCEDURE (ch: Channel) Write (VAR buf: ARRAY OF SYSTEM.BYTE; n: INTEGER);
         PROCEDURE (ch: Channel) Res (): INTEGER;
         PROCEDURE (ch: Channel) Local (): Host;
         PROCEDURE (ch: Channel) Remote (): Host;
         PROCEDURE (ch: Channel) Connected (): BOOLEAN;
         PROCEDURE (ch: Channel) AvailableBytes (): LONGINT;
         PROCEDURE (ch: Channel) FreeBufferSize (): LONGINT;
         PROCEDURE (ch: Channel) Close;
      END;

      HostName = ARRAY 32 OF CHAR;

      Host = POINTER TO HostDesc;
      HostDesc = RECORD
         local-: BOOLEAN;
         name-: HostName;
         PROCEDURE (h: Host) AddServer (s: Server);
      END;

      HostIterator = RECORD
         PROCEDURE (VAR iterator: HostIterator) FirstHost (onlyLocal: BOOLEAN): Host;
         PROCEDURE (VAR iterator: HostIterator) NextHost (): Host;
      END;

      RequestType = ARRAY 20 OF CHAR;

      Protocol = POINTER TO ProtocolDesc;
      ProtocolDesc = RECORD
         PROCEDURE (p: Protocol) Connect (from, to: Host; type: RequestType; VAR ch: Channel;
                                          VAR error: INTEGER);
         PROCEDURE (p: Protocol) Broadcast (from: Host; type: RequestType;
                                            VAR buf: ARRAY OF SYSTEM.BYTE; n: INTEGER);
         PROCEDURE (p: Protocol) HostRegistered (host: Host);
         PROCEDURE (p: Protocol) HostRemoved (host: Host);
      END;
```

```
        Server = POINTER TO ServerDesc;
        ServerDesc = RECORD
            PROCEDURE (s: Server) NewChannel (from, to: Host; type: RequestType): Channel;
            PROCEDURE (s: Server) Run (ch: Channel);
        END ;

    (* connecting *)
    PROCEDURE ConnectChannel (ch: Channel; from, to: Host; type: RequestType;
                                        VAR err: INTEGER);
    PROCEDURE Broadcast (from: Host; type: RequestType; VAR buf: ARRAY OF SYSTEM.BYTE;
                                        n: INTEGER);

    (* hosts *)
    PROCEDURE DefaultHost (): Host;
    PROCEDURE ThisHost (name: ARRAY OF CHAR): Host;
    PROCEDURE RegisterHost (host: Host; protocol: Protocol; name: ARRAY OF CHAR;
                                    VAR err: INTEGER);
    PROCEDURE RemoveHost (host: Host);

    (* incoming requests  *)
    PROCEDURE Request (from, to: Host; type: RequestType; protocolChannel: Channel;
                                VAR err: INTEGER);

END Network.
```

Operations on Hosts

- *host := DefaultHost ()* returns the default host for the local site. The default host is generated automatically when module *Network* is loaded. The host name is taken from the file 'Network.Host'.

- *host := ThisHost (name)* returns the host specified by *name*.

- *host := iterator.FirstHost (local)* returns the first host known to the local site. The receiver *iterator* is prepared for further calls to *NextHost*. If *local* is TRUE, *FirstHost* and *NextHost* return only hosts located on the calling site.

- *host := iterator.NextHost ()* returns the next known host. If no more hosts exist *NextHost* returns NIL and *iterator* is no longer valid. Hosts returned by *FirstHost* and *NextHost* are not ordered.

- *RegisterHost (host, protocol, name, err)* registers a new host with the given name as reachable via *protocol*. *host* is registered as locally reachable (located at the calling site), if *protocol* is NIL. Only protocols should register non-local hosts. *err* indicates the success of the operation.

- *RemoveHost (host)* removes *host* from the host list. No new channels can be connected to *host* and broadcasts will no longer be delivered to it. However, existing channels are not influenced. A host should only be removed by the program that registered it.

Connecting

- *ConnectChannel (ch, from, to, type, err)* opens a new channel between *from* and *to*. *from* has to be a local host. The channel *ch* must already be allocated. Module *Network* uses the passed request type to establish the channel. If *to* has no server accepting the request type or if an error occurs, an error code is returned.

- *Broadcast (from, type, buf, n)* broadcasts *n* bytes in buffer *buf* from host *from* to all registered hosts. No acknowledgement is given. All servers will see the broadcast as a request of type *type* preceded by "BCast" (e.g. a broadcast of type "MyType" is seen as "BCastMyType"). In order to receive the data, they have to allocate a channel with NEW. Such channels (opened in response to a broadcast) are read-only (the write behaviour is not defined). A server may pass the same channel instance to more than one broadcast request. Several broadcasts issued simultaneously are not guaranteed to be delivered in the same order on all hosts.

Handling Incoming Requests

- *host.AddServer (s: Server)* registers a new server *s* on *host*. Registered servers are called whenever a request arrives for *host*. If a server accepts the request it should return a valid channel. Otherwise NIL should be returned. Servers may only be added to hosts situated on the same site as the caller of *AddServer*, i.e. on hosts registered locally with the protocol NIL.

- *Request (from, to, type, protocolChannel, err)* should be called by protocols, whenever a new request is received over the network. *Request* tries to establish the

channel using the previously registered servers.

- *ch := server.NewChannel (from, to, type)* is called for all servers registered at *to* whenever a request arrives. A server accepting the request type should return a valid channel. A server refusing the request should return NIL.

- *server.Run (ch)* is called by module *Network* after *server* accepted a request with *server.NewChannel*. The server may now communicate through *ch*. For lengthy tasks the server should start a thread handling the communication through channel *ch*.

Protocols

- *protocol.Connect (from, to, type, ch, error)* is called by *Network* whenever *protocol* should open a new channel. *from* is always a local host and *to* is always a host registered by *protocol*. The operation should return the newly opened channel in *ch* and set *error* accordingly.

- *protocol.Broadcast (from, type, buf, n)* is called by module *Network* whenever a broadcast is initiated. The data in *buf* should be forwarded to all hosts registered by *protocol*. No acknowledgement is expected.

- *protocol.HostRegistered (host)* is called by *Network* in order to notify *protocol* that a local host has been added.

- *protocol.HostRemoved (host)* is called by *Network* in order to notify *protocol* that a local host has been removed.

Channels

- *ch.Close* closes the channel *ch*. Closing a channel is not necessarily something both hosts have previously agreed upon. However, the protocol should at least try to inform both channel ends, in order to update the channel states at both ends. This, so-called 'Last Ack Problem' is proven to be unsolvable. An actual protocol may transmit several close channel messages and assume that one will arrive with high probability.

- *count := ch.AvailableBytes ()* returns the number of bytes readable from *ch* without blocking the reading thread.

- *count := ch.FreeBufferSize ()* returns the number of bytes writeable to *ch* without blocking the writing thread.

- *connected := ch.Connected ()* returns TRUE if *ch* still connects two hosts, respectively FALSE if the channel is broken or has been terminated.

- *host := ch.Local ()* returns the local end (host) of channel *ch*.

- *host := ch.Remote ()* returns the host at the other end of channel *ch* (remote host).

- *state := ch.Res ()* returns the result of the most recent operation on *ch*, i.e. the state of *ch*. Once an error occurs no more information can be transmitted over *ch*. All further calls to *ch.Res* yield the same return value, which is either an error code or the number of bytes which could not be read or written.

- *ch.Read (buf, n)* reads *n* bytes into *buf*. *Read* blocks until *n* bytes could be read. If an error occurs *ch.Res* will return the number of bytes which could not be read, otherwise it returns *ok*.

- *ch.Write (buf, n)* writes *n* bytes onto *ch*. *Write* blocks if the output buffer would overflow. If an error occurs *ch.Res* will return the number of bytes which could not be written. Otherwise it returns *ok*.

Network supports interoperability on an abstract level. However, when transmitting typed data (e.g. INTEGER, REAL, ...) via *ch.Read* or *ch.Write,* one has to ensure that the remote host has the same byte ordering. We recommend that higher level modules cope with this problem, e.g. by extending the base channel type of module *Network*.

3.5 Failure Semantics

Distributed systems introduce many advantages such as information and resource sharing, parallelism, or scalability. However, they give as well rise to significant new problems, e.g. failure proneness. At first sight, distributed systems seem to be more secure and more stable, because they are distributed over several physical sites. As sites have independent failure modes, a failing site allows the remaining sites to keep running. This is correct but has significant consequences for system designers and programmers. A system running on just one site does not have to cope with site failures (power failure, head crash, ...). Distributed systems, however, have to deal with failing sites in order to allow the remaining sites to go on with their work. One or several failing sites should not disable the whole system. We pay for the advantage of independent failures with a substantial increase in complexity.

There is no single correct solution to the above problem. There are many different approaches. Sometimes a site should be informed as fast as possible of another site's failure (e.g., in a backup system for a flight control system). Sometimes this may be delayed until the failed site is actually used (e.g., in a backup system for a data server). The desired method for failure detection depends on the actual task at hand.

We distinguish two kinds of failures. Failures within a single connection and failures involving several connections depending on each other. For both kinds, failure semantics define how to react upon abnormal conditions. This is relatively easy for the first kind of failures, as there is only one entity (connection) which may fail. We have no possibilities to continue our operation. As an example: How does a program react to a broken ftp connection? Should the program terminate with an error, should it raise an exception, or should it try to re-establish the state as it was prior to the failure?

The second kind of failures are far more difficult to tackle. Several hosts on different sites communicate with each other. Failing sites and/or connections should not hinder the on-going calculation, but let it continue as smoothly as possible. They should be replaced transparently in order to allow a successful completion of the distributed calculation. If error correction is not possible (too many failures, no backups, network partitioning, ...), the calculation should not be aborted, but should be brought to a graceful end, which allows the system to later resume the calculation under - hopefully - better circumstances. This report does not describe semantics for these kinds of failures, as it is not part of the low-level communication mechanisms, but is highly application dependent. Only the application has all the knowledge needed to choose the appropriate solution.

Failure semantics define how failures are reported. What happens locally if the remote site fails? What if the received data is corrupt? What happens when the network link is lost? From the point of view of an application it would be desirable to offer the abstraction of unfailing connections. Unfortunately this is not possible. Certain failures can not be corrected. Only the application knows the correct steps to be taken (ultimately by calling upon the user). The 'End-to-End argument' says that, since error checking must definitely be done by all applications, do not bother to do it at a lower level. The application probably knows better what to do. On the other hand, it makes no sense to report failures to the application, which could be handled easily within the low-level network system. Every application would have to replicate the code necessary to handle them.

We have to steer a middle course. Some failures can and should be corrected automatically. Others have to be handled by the application. The chosen borderline is arbitrary and differs largely between systems. Every application has to know this borderline which is defined by the failure semantics.

Network services are accessed through a programming interface (here module *Network*). It offers functionality for opening and maintaining connections, writing and reading data, as well as ways to perceive and handle failure conditions, i.e. the interface is influenced by the failure semantics and vice versa. There are three different ways to notify the application about an abnormal condition:

- Explicit parameters indicate failure conditions. Every operation, which possibly leads to an error, returns an error indication (visible in the interface). Possible additional error information (e.g. number of bytes actually read) may be passed in additional parameters. Using this technique, the resulting source code tends to be cumbersome to read respectively to maintain. After every operation the error parameters have to be checked and corrective measures may have to be taken. This tends to be troublesome and ugly. It may even be that troublesome, that programmers start to ignore possible failures, which leads to erroneous programs.

- Applications ignore failures when they communicate over a channel. If an error occurs, an exception is raised by the failing network operation. The exception gets forwarded to the appropriate exception handler. This effectively centralises the handling of failures. With this scheme, there is no way to accidentally ignore a failure. On the other hand, exceptions have drawbacks as well:

  - Not all systems respectively programming languages support exceptions.

  - Fine grained error handling - different actions on different errors and at different locations - is cumbersome.

  - Exceptions contradict the idea of structured programming. All instructions may possibly jump into the exception handler. When entering the exception handler, control flow may come from any instruction ('come from' semantics).

- Special operations return state information. This approach is similar to the first one which introduced explicit parameters. After every operation (here with no additional parameters) the state has to be requested and tested for possible failures. Programmers may omit this, which may lead to erroneous programs. However, together with suitable failure semantics, this approach yields a possible new solution.

We chose the third possibility for our module *Network*. Explicit parameters lead to inflated and difficult-to-read source code. Oberon-2 does not include an exception mechanism. Additionally, the flexible manner, in which the third alternative handles errors, appealed to us.

We associate a state with each channel. The channel state is set to *ok*, after the channel has been established successfully. The channel state may be accessed by calling the operation *channel.Res*. Furthermore we define that all operations on channels with an illegal state (not *ok*) preserve the state and return control immediately to the caller (null operation). This allows the application to use a channel without checking its state after every single operation. The application checks the state only on so-called checkpoints. The application may adapt its error checking scheme (fine or coarse grained) to its own needs, using all its knowledge of the ongoing computation.

Our chosen approach has two drawbacks. First, the application has to ensure that the channel state is *ok*, before previously received data is actually used. Second, a failure is not handled immediately but it is delayed until the next checkpoint. This may decrease performance in failure scenarios. However, with today's highly reliable networks (ATM), the emphasis should be put on the optimisation of error-free scenarios.

An example for delayed error checking while reading from a channel *ch*:

```
VAR
    count, age: INTEGER;
    name1, name2: ARRAY 32 OF CHAR;

ch.Read (count, 2); (* read number of entries *)
ch.Read (name1, 32); ch.Read (name2, 32); ch.Read (age, 2);
WHILE (count > 0) & (ch.Res() = Network.ok) DO     (* checkpoint *)
    ... print name1, name2, age
    DEC (count);
    ch.Read (name1, 32); ch.Read (name2, 32); ch.Read (age, 2)
END
```

3.6 Example: Implementing a time server

We will now show how a client can request the current time from a time server on a host called "Time Server". By accessing the host via a name, we can easily move it to a different site without invalidating clients. First, we have to register the host "Time Server" on one of our sites. Then we have to install a server object in this host. This is shown in the following code:

```
NEW (host);
Network.RegisterHost (host, NIL, "Time Server", err);
IF err = Network.ok THEN
    NEW (server);
    host.AddServer (server)
END
```

Now we have to think about the implementation of the time server. It needs a method *NewChannel* that checks for the type of requests it accepts. In our case, the server accepts request of type "TIME". For communication, our server does not need a special channel extensions, so it uses a channel of type *Network.Channel*.

```
PROCEDURE (s: Server) NewChannel (from, to: Host; type: RequestType): Network.Channel;
VAR ch: Network.Channel;
BEGIN
    IF type = "TIME" THEN NEW (ch) ELSE ch := NIL END;
    RETURN ch
END NewChannel;
```

After a request has been accepted, *Network* calls the method *Run* of the accepting server. Typically *Run* dispatches a thread handling the communication. Our time server omits this, as it just sends four bytes.

```
PROCEDURE (s: Server) Run (ch: Network.Channel);
VAR time: LONGINT;
BEGIN
    time := Input.Time ();
    ch.Write (time, 4);
    ch.Close
END Run;
```

We have now completed the implementation of our time server. A possible client may now open a channel to the server and retrieve the current time. The client needs to know the host running the server ("Time Server"), as well as the accepted request type ("TIME").

```
PROCEDURE Client;
VAR res: INTEGER; ch: Network.Channel; time: LONGINT; timeHost: Network.Host;
BEGIN
    timeHost := Network.ThisHost ("Time Server");
    IF timeHost # NIL THEN
        NEW (ch);
        Network.ConnectChannel (ch, Network.DefaultHost (), timeHost, "TIME", res);
        IF res = Network.ok THEN
            ch.Read (time, 4);
            ch.Close;
            ... use time
        END
    END
END Client;
```

We emphasise, that both server and client work with a channel instance allocated by themselves. Both may extend the base channel type *Network.Channel* in order to adapt it to their needs.

3.7 Adding a Protocol

Module *Network* includes a protocol supporting communication between hosts located on the same site. For cross site communication additional protocols have to be supplied. These protocols have to provide

- an extension of the type *Network.Protocol*,
- an instance of this extension,
- a bi-directional notification mechanism for host registration and removal.

Extension of Network.Protocol

Module *Network* offers an abstract base type *Protocol*. This type has to be extended for new protocols. Whenever *Network* wants to communicate with a host reachable by a new protocol, it calls the appropriate methods of the protocol instance associated with this host.

Instance of Protocol

For every protocol type, there is a single instance that is registered with every host which is reachable using this protocol. All these hosts share the same protocol instance.

Host notification

The set of hosts (maintained by *Network*) may be changed in two different ways. Either an application adds/removes a host, or a protocol connects/disconnects to a new site. This newly connected site may lready contain hosts, which will then be added to the locally stored set of hosts. In either case all sites have to be notified of this change. Network delegates this notification to the appropriate protocol, as only protocols can do this efficiently (every site has to be notified but once).

When an application adds a host on its site, *Network* notifies all protocols registered together with a host. The protocols have to forward the notification to all their sites, in order to update their sites's instances of the host set. The same notification mechanism is necessary, when a local host is removed.

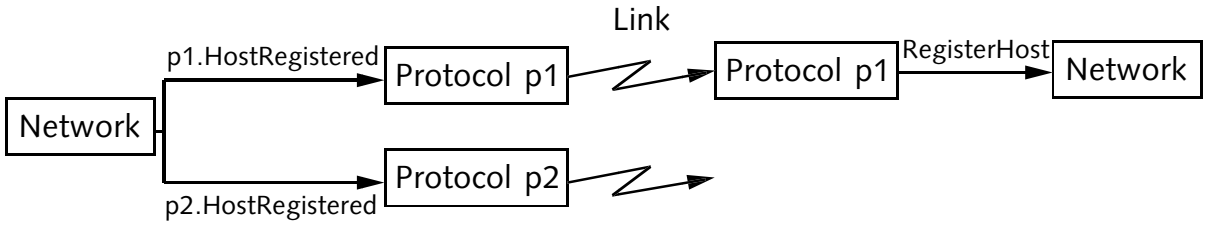

Figure 7. Notification of newly registered host

A protocol connects its site with other sites. When it is started, it has to collect the names of the hosts on its remote sites, and add them to the local set of hosts. Additionally, it has to broadcast local hosts to all its remote sites.

Example: We have the protocol 'P' which connects the sites 'A' and 'B'. On site 'B' the protocol is already loaded. As it was not able to connect to site 'A' it registered no hosts (Figure 8).

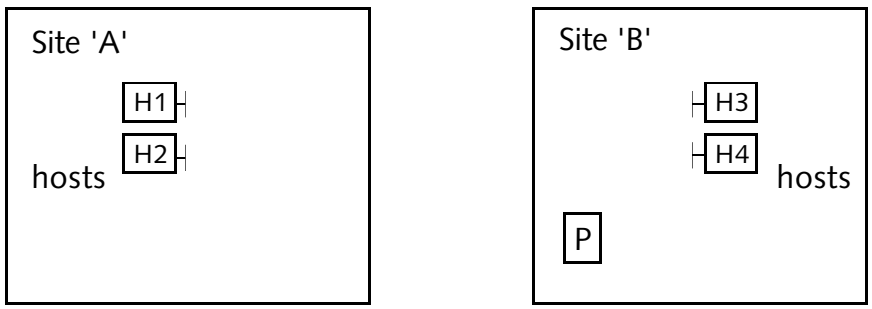

Figure 8. Sites 'A'and 'B' with protocol 'P' loaded on site 'B'

When started on site 'A', protocol 'P' tries to connect to site 'B'. If successfull, it collects the local hosts ("H1" and "H2"), sends them to site 'B', and waits for the list of hosts local to site 'B' ("H3" and "H4") (Figure 9).

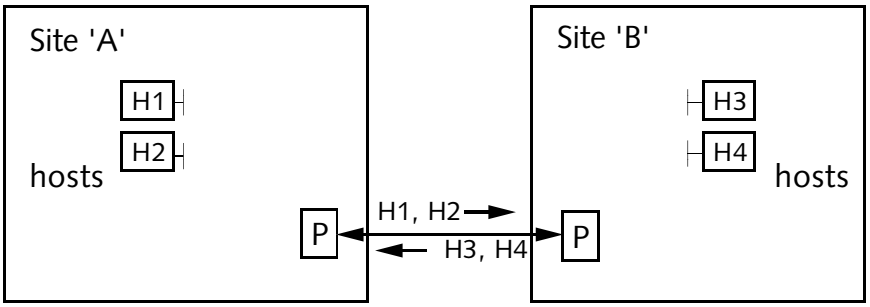

Figure 9. Exchange of local hosts

After 'P' received the hosts of site 'B' it registers them with module *Network*. With each host registered, it passes an instance of itself, in order to indicate, that these host may be reached via protocol 'P' (Figure 10).

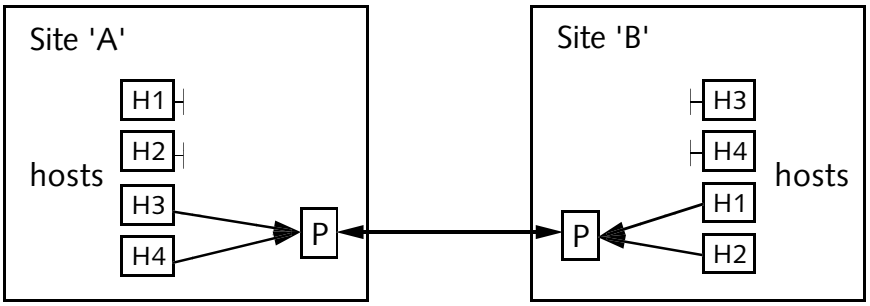

Figure 10. Startup of protocol 'P' completed

After this initialisation applications, running on site 'A' or on site 'B', can access the hosts "H1", "H2", "H3", and "H4". An application running on site 'A' will communicate with host "H1" locally. The same application running on site 'B' will communicate with host "H1" via protocol 'P'. However, this distinction is hidden within the module *Network,* and does not influence the application; i.e. the same application runs regardless of the actual location of host "H1".

In the future we will see, if the protocol interface offers enough information about hosts registered by protocols. Possibly we may have to extend the interface in order to enable clients of *Network* to choose suiting hosts. Which host has a compatible processor? Is a host tightly or loosely coupled?

4 Related Work

Most - if not all - of our work has already been done else where. We implemented our own network support mainly in order to remain compatible with the existing Oberon system. We now profit from the many existing Oberon implementations which considerably reduce porting efforts. In addition, we gain a run-time and developing environment. In this section we compare our work with some well known distributed systems, i.e. with their communication primitives and their process management. Of course, these systems contain much more support for distributed computing which is not included in our low-level modules. Our modules *Threads* and *Network* are designed to be extensible in order to be able to dynamically extend the offered features.

4.1 Hermes

Concurrency has already been added to Oberon running on the Ceres workstation [Ebe87]. This project [Lal94] discusses the implications of concurrency and network support on the design of Oberon. However most results - within the low-level modules - are applicable only to Oberon on Ceres workstations. The principal design of thread support is similar to our approach. The network support is specific to a particular network and does not support multiple protocols. It offers the abstraction of network messages.

4.2 Amoeba Microkernel

The Amoeba microkernel [TaKa94, Tan95 pp. 376-430] offers four services: Memory management, threads, network support, and low-level I/O. Of these services only the threads manager and the communication support are relevant for our comparison.

Processes are hierarchically structured. Each process may contain several threads. Child processes do not have to run on the same processor. They are distributed automatically by the run server. There are communication facilities for parents to contact their children. Processes are managed by a special server (run server). All threads within a process run on the same processor and share their memory. Threads may be synchronised with either signals, mutexes, or semaphores. Signals and mutexes are low-level facilities. The semaphores closely correspond to our semaphores.

Amoeba offers the two communication primitives: multicast and point-to-point communication. It offers primitives for remote procedure calls including support for security. All communication is based on the FLIP protocol. Our communication support is definitely inferior and has to be extended to be comparable. All communication is based on the FLIP protocol. However, with our notion of add-on protocols, we achieve more flexibility.

4.3 Chorus

Chorus [Brick94, Tan95 pp. 475-519] supports processes with one or several threads. Threads have priorities and are scheduled preemptively. Chorus allows the emulation of Unix and supports real-time applications. This complicates its threads model considerably.

Messages are passed between ports. Ports can be put together to form port groups. Naming is done with 64-bit unique identifiers. Messages have a header containing the sender, the receiver, and some additional flags. Our network support does not define a message format. It even does not presume that network messages exist. We have adopted a higher level of abstraction which leaves the actual communication details completely to the individual protocols.

4.4 DCE

DCE (Distributed Computing Environment) [John94, OSF95] is not a system, but the definition of a framework for building distributed systems. Threads are defined to support mutexes, condition variables (semaphores) and alerts. Alerts are used to provide graceful and reliable thread termination.

As a communication primitive only RPC's are defined (including an interface definition language (IDL)). The lower level of communication is left completely undefined. This approach is similar to our notion of attachable protocols, but it is done at a higher level (RPC instead of channels).

4.5 Mach Microkernel

Mach [BoyKir93, Tan95 pp. 431-474] provides a foundation for building other operating systems (e.g. Unix) which allow transparent access to network services. The details are left to the individual systems built on top of Mach.

The Mach kernel supports several independent tasks each with one or several threads of execution. Communication is done by sending messages between two ports. No higher level mechanism is defined. Mach offers user-level network servers similar to our installable protocols.

4.6 Summary

The following table shows the principal characteristics of the discussed systems compared to our modules *Threads* and *Network*.

System	Hermes	Amoeba	Chorus	DCE	Mach	Network
Basic Communication Mechanism	Messages	RPC Multicast	Messages	RPC	Messages	Channels
Process Management	Threads	Processes Threads	Processes Threads	Threads	Tasks Threads	Threads

Most of the mentioned systems include the process and communication support in their kernel. They run in kernel mode. In contrast, our network support runs as a user-level process and is built on top of the existing Oberon system.

5 Conclusions

The modules *Threads* and *Network* are just the first step towards distributed objects. They fit well into the standard Oberon System and offer the needed fundamental support and abstractions.

The module *Network* is system independent and supports arbitrarily networks and protocols. Communication between hosts is independent of the actual sites that host them. In contrast to other systems, we neither choose to offer low-level message passing facilities, nor high level mechanisms, e.g. RPC. We offer an abstraction which allows every protocol to choose its own method of data transmission.

The next steps will be to add the following points to our framework:

- *Linearisation*: Complex data structures (e.g. trees or lists) cannot be transmitted over a network in a straight forward way. The contained references must be reconstructed and cycles have to be restored on the receiving site. We need a general mechanism which allows the marshalling and unmarshalling of arbitrary data structures.

- Remote objects. Objects on arbitrary sites should be accessible transparently via ordinary Oberon-2 messages.

- *Stub generation*: For remote method invocation we need a stub generator similar to IDL compilers used for ordinary RPC calls.

- *Stub objects*: To offer transparent access to remote objects, we have to supply a local stub object which redirects method invocations to the corresponding server object. Such stub objects have to be allocated and linked to remote objects.

- *Name service*: An application accessing a remote object has to locate and identify the object, regardless of the physical site respectively logical host the object is actually located. This address resolution is done by a so called name service.

Appendix A: How to get modules Threads and Network

Both source modules (for PowerMac Oberon) as well as this report (as Oberon text and postscript file) are freely available via anonymous ftp from

ftp://ftp.ssw.uni-linz.ac.at/pub/Oberon/PowerMac/Network/

or via WWW from

http://www.ssw.uni-linz.ac.at/Projects/Network.html

Appendix B: References

[BirRen94] K. Birman, R. van Renesse:
"Reliable Distributed Computing with the Isis Toolkit"
IEEE Computer Society Press, 1994

[BoyKir93] J. Boykin, D. Kirschen: "Programming under Mach"
Addison-Wesley, 1993

[Brick94] A. Bricker, J. Lipkis:
"A New Look at Microkernel-Based UNIX Operating Systems"
Distributed Open Systems, IEEE Computer Society Press 1994, pp. 31-48

[Coul94] G. Coulouris: "Distributed Systems"
Addison-Wesley, 1994

[Ebe87] H. Eberle: "Development and Analysis of a Workstation Computer"
PhD thesis, Swiss Federal Institute of Technology (ETH Zürich), 1987.
Number 8431

[GHJV95] E. Gamma, R. Helm, R. Johnson, J. Vlissides: "Design Patterns"
Addison-Wesley, 1995

[Jalote94] P. Jalote: "Fault Tolerance in Distributed Systems"
Prentice Hall, 1994

[John94] B. Johnson:
"A Distributed Computing Environment Framework: An OSF Perspective"
Distributed Open Systems, IEEE Computer Society Press 1994, pp. 57-77

[Lal94] S. Lalis: "Hermes - Supporting Distributed Programming in a Network of Personal Workstations"
PhD thesis, Swiss Federal Institute of Technology (ETH Zürich), 1994.
Number 10736.

[MöWi91] H. Mössenböck, N. Wirth: "The Programming Language Oberon-2"
Structured Programming, 12, 4, 1991, pp. 179 - 195

[OSF95] Open Software Foundation: "Introduction to OSF DCE"
Prentice Hall, 1995

[TaKa94] A. Tanenbaum, M. Kaashoek: "The Amoeba Microkernel"
Distributed Open Systems, IEEE Computer Society Press 1994, pp. 11-30

[Tan89] A. Tanenbaum: "Computer Networks"
Prentice Hall, 1995

[Tan95] A. Tanenbaum: "Distributed Operating Systems"
Prentice Hall, 1995

[WiGu89] N. Wirth, J. Gutknecht: "The Oberon System"
Software-Practice and Experience, 19(9), 1989, pp. 857-893

Technical Reports

The following reports can be obtained as postscript files from the URL:

ftp.ssw.uni-linz.ac.at/pub/Reports/.

M. Knasmüller: Adding Persistence to the Oberon-System.
Report 6, 1996.

M. Hof: A Run Time Debugger for Oberon-2.
Report 5, 1995.

K. Koskimies, H. Mössenböck: Scenario-Based Browsing of Object-Oriented Systems.
Report 4, 1995.

H. Mössenböck, K. Koskimies: Active Text for Structuring and Understanding Source Code. Report 3, 1995.

K. Koskimies, H. Mössenböck: Designing a Framework for Language Implementation.
Report 2, 1995.

M. Knasmüller: Oberon Dialogs - Users' Guide and Progrmming Interfaces.
Report 1, 1994.