# THE DISTRIBUTED OCCAM PROTOCOL

A New Layer On Top Of TCP/IP

To Serve occam Channels Over The Internet

Mario Schweigler

A dissertation submitted for the degree of

MSc in Distributed Systems and Networks

Supervised by Peter H. Welch

Computing Laboratory

University of Kent at Canterbury

Friday, 7th of September, 2001

13,000 words

# ABSTRACT

Networking, especially the internet, is playing a more and more important role in our lives. Many new applications are distributed on several computers which are connected over the internet.

The occam parallel processing language offers some features which make it particularly suited for writing applications which are distributed over the internet. Especially the channel paradigm which enables concurrent processes to communicate with each other, is a good basis for distribution.

This project deals with the development of a distributed version of occam channels which allow occam processes running on different computers to communicate with each other in a similar way as they would do if they were running on the same machine.

# CONTENTS

# 1. INTRODUCTION

occam is a parallel processing language which is based on the CSP calculus. This calculus is basically a neat mathematical way of modelling concurrency. There is a whole bunch of literature written about CSP, for instance [1], which the interested reader might investigate to get further insights into the calculus. In this paper I do not intend to get further into detail about the CSP calculus, as this is not necessary to understand the objectives of this project, and also because it was not a substantial part of the project work to deal with the calculus itself.

What is more important, in fact it is crucial, in order to be able to understand the aims of this project and the way it was carried out, is to understand the occam language. occam was originally developed by INMOS in 1985 to support their newly developed transputer [2].

The transputer was a new microchip which was specifically developed for parallel computing. It is able to carry out several instructions in parallel because it does not follow the von Neumann principle of sequential execution of each instruction and even each sub-instruction. The parallel processes are scheduled by a micro-coded scheduler which performs an effective management of the transputer's time slices. This is the reason why the transputer is much more efficient than even highly developed pipeline structures in conventional von Neumann architectures.

Furthermore, the transputer eliminates many of the bottlenecks which are caused by increased bus traffic in conventional microchips. Firstly, it has on-chip local memory which helps to eliminate the usual processor-to-memory bottlenecks. And secondly, it can communicate in parallel with other transputers using one of the four built-in high-speed serial links.

The concept of connecting transputers directly over independent links was a revolutionary idea. Now it was possible to let several, maybe thousands of transputers work in parallel and communicate with each other without the common problems of overloaded shared buses. The transputer-to-transputer links work autonomously form internal instructions the transputer is executing. Therefore, once a link communication is initialised, it proceeds concurrently with the internal processes. This is possible due to the use of direct memory access techniques.

This enables "real" concurrent communication with no more need to wait for a shared bus. It is possible to build large networks of transputers without the bandwidth becoming saturated as the systems increase in size. Furthermore, as the links are only thought for communication between two individual transputers, a large system will not suffer contention as it is the case when using a shared bus.

To use these high potentials, INMOS developed a new language – occam. The key feature of occam is that it was specifically developed for parallel processing. The concurrency features of occam were not added to an existing language by additional language constructs or by libraries. The support of concurrency in occam was a design issue.

There is a lot of literature written about occam. A good and compact introduction into occam is given in [5], for instance. In the following, I would like to point out some of the key features of occam which are necessary to understand how this project was done. However, I cannot give a complete introduction as this would exceed the scope of this dissertation. Readers who are not familiar with the occam syntax and semantics should therefore consult an occam manual.

The general idea of occam is that is enables the programmer to write sequential processes which can be executed in parallel. The parallel execution of processes is also a process which again might be executed in parallel with other processes and so on.

Processes running in parallel are independent from each other. They run autonomously. The only way to communicate with each other is over well-defined channels. This communication follows a handshake principle. This means that if one of the processes wants to write to a channel and the other process wants to read from it, the one who is first will be suspended until the other process is ready to perform the counterpart operation on the other side of the channel. This means that there is no buffering built into the channels.

These features offer a large range of possibilities how various processes can be allocated to different transputers [3]. The design of the transputer allows to execute several processes in parallel. When they communicate, the channel behaviour is achieved by block movements of data in the transputer's memory. I will refer to this kind of channel behaviour as "soft channels".

On the other hand, processes can also be distributed on different transputers. When two processes on two different transputers want to communicate, they have to do so over the physical links of the transputers. Each of a transputer's four links implements two occam channels – one for outputs and one for inputs.

The main program can assign an occam channel to a specific link address (i.e. the number of the link and whether it is the incoming or the outgoing channel of this link) on each of two transputers. These transputers have to be connected over those link addresses. Communication over such an occam channel now means physical communication from one transputer to another. I will refer to this kind of channel behaviour as "hard channels".

The key point is that the occam processes themselves do know at all whether they are communicating over a soft channel or a hard channel. For an occam process a channel is a channel, no matter whether it is emulated internally by the transputer moving data in its memory or whether it is a physical channel using one of the transputer's links.

Over the years, the occam language has been made available for other platforms too. An important step to achieve this was the Kent Retargetable occam Compiler (KRoC) [11], which was developed at the University of Kent at Canterbury. KroC is available, among others, for SunOS, Solaris, and Alpha DEC stations, as well as for Intel PCs under Linux.

KRoC performs two stages. At first, it compiles occam code into transputer assembly language. So the advantages of the transputer can be maintained. Among these advantages there are in particular the effective scheduling mechanisms for concurrent processes and the secure communication mechanisms between those processes which are used for the occam soft channels.

The second stage is to translate the transputer code into the assembly language of the target machine. That way, the target machines, or strictly speaking the resulting executables, are turned into virtual transputers. Those programs perform concurrency as a transputer would do. There is no need to involve the operating system's scheduling mechanisms. In fact, a running occam program is only one operating system process (one Linux process, for instance).

This is achieved by linking the compiled occam program with a runtime system that acts as a virtual transputer – the occam kernel – to an executable file. This fine-grained concurrency on application level makes occam programs that efficient. The advantages of transputers are so transferred to conventional processor architectures and operating systems.

What, however, has been lost in this process are the hard channels. As there are no real transputers but "only" virtual ones, hard channels have suddenly disappeared.

## 1.1. THE AIM OF THE PROJECT

Today's world is largely influenced by networking. Networks, above all the internet, are playing a more and more important role not only in the computing community but rather for the whole society. The aim of this project is to utilise the internet for occam.

The goal is to be able to distribute occam processes on different computers and enable them to communicate the same way as if they were running on the same machine – through occam channels. I will define this new kind of channels as "network channels". A network channel is a rather theoretical construct, which means that it is not a real channel but it emulates the behaviour of an occam channel, but with the two end points being located on different computers. In the following, I will call these end points "writer" and "reader" which means the sending and the receiving side of a network channel respectively.

Although hard channels and network channels are not really comparable, they have some aspects in common. They both are a contrast to soft channels, as they are not emulated by the same transputer or virtual transputer respectively by performing memory operations. What they both perform is some physical communication; in case of hard channels over transputer links, and in case of network channels over the internet.

## 1.2. OBJECTIVES

The objective of this project is to create an occam library which offers an interface to the internet. This interface shall support network channels of all occam standard data types and arrays thereof. The interface shall offer an array of normal occam channels (soft channels) to user processes. The user processes shall be connected to the interface via channels from this array. The interface shall communicate with other computers – which are running the same interface – via TCP socket connections.

A rather naïve way of doing this would be to assign a separate socket connection to each network channel. But this is not really feasible as it would mean a huge waste of resources. The number of socket connections supported by the operating system which can be used concurrently is restricted. Really necessary is only one socket connection between two computers. Should there exist more than one network channel between processes on these two computers, the data sent over these channels shall be multiplexed over the single socket connection between the two machines.

In order to ensure an easy use of the network channels, they must be easily identifiable. The easiest way to identify a channel is to give it a name. This is the way soft channels work, so why should not it be done the same way with network channels? Well, it is not that easy. For processes which are supposed to run on the same machine, in the main program a (soft) channel is defined by giving it a name, like with a variable. Then this channel is passed as a parameter to the processes which are supposed to communicate over it and these processes are executed in parallel.

In case of network channels there is no main program which can manage the names of network channels. We need a higher authority, which is located somewhere in the internet, to play this role. Here the Channel Name Server (CNS) comes into play. It is used to maintain information about the names of network channels and the related location of the channels in the internet.

A network channel itself is not located anywhere. What is located somewhere in the internet are the two end points of the network channel. So the CNS has to be a mediator between the end points. This is why the CNS shall be designed alike a Domain Name Server. At first the reader has to register with the CNS. The reader must tell the CNS the name of the network channel and its own location which the CNS will store in its database. When a writer wants to use this network channel, it has to make a request at the CNS, telling it the name of the network channel. Then the CNS will respond by telling the reader's location to the writer.

This way of implementing the CNS gives Any to One semantics for network channels, as there can be more than one writer requesting the location of the same reader by asking for the same network channel, (i.e. by telling the same name to the CNS). Nevertheless, it is still possible to use network channels as normal One to One channels (as conventional soft channels are). For that, it only has to be assured that only one writer makes a request to the same network channel.

# 2. THE BACKGROUND

There were several pieces of research work which built the fundament of this project. The occam socket library is the basis for the use of TCP socket connections under occam. Furthermore, there have been two undergraduate final year projects dealing with the distribution of CSP channels over the internet, which this project was building on.

## 2.1. THE OCCAM SOCKET LIBRARY

In 2000, the occam socket library [14] was created. It is an occam library which enables the use of socket connections for occam processes. This library is available for the Linux release of KRoC. This is also the main reason why this project was carried out under Linux.

Generally, socket communications, as other IO operations as well, are performed by user programs by making a system call to the operating system. The problem, however, is that these system calls cause the calling operating system level process, in our case the entire occam program, to block. This means that not only the occam process that wants to perform a socket operation would be blocked. In fact, all occam processes running in parallel with it as well as the occam microscheduler would be suspended as well until the socket operation is finished. As this would not be efficient at all – it would withdraw all the advantages of occam programs – another solution had to be found [13].

A naïve solution to avoid other occam processes being blocked when one occam process makes a Linux system call would be to run all occam processes as separate Linux processes and let them communicate with each other via the normal operating system level mechanisms. But this is not feasible either. When operating system level processes communicate over standard process communication, there are large communication overheads compared to the high performance of the occam kernel microscheduler.

The occam socket library therefore uses Linux clones. A clone in Linux is an operating system level process, but not a conventional one. The difference between a normal forked process and a clone is that the clone shares the same virtual memory and the same file descriptors with its parent.

When a blocking system call, such as for a socket operation, has to be performed, the occam kernel suspends the occam process which wants to do this operation and tells a clone of itself to make the blocking system call. Due to the fact that clones use the same shared memory, it is possible to use shared variables which enables an efficient way to communicate between the occam kernel and its clones. When the clone has finished the system call, it tells this to the occam kernel who then reschedules the suspended occam process.

The occam socket library offers the typical socket operations for creating sockets, reading, writing, resolving host names etc. I used the socket library to perform the necessary TCP/IP communication in the occam code which I created for the project.

## 2.2. THE JCSP.NET PROJECT

The JCSP.net project [18] was an undergraduate final year project which was done this academic year. The objectives of this project were pretty much the same as for mine, but for JCSP instead of occam. JCSP is a package for Java which offers CSP semantics. It enables users, such as occam does, to program sequential processes which are running in parallel and communicating over channels.

The advantages of JCSP are that it can utilise all the features Java provides, including objects. Furthermore, the resulting classes can be run on every platform which offers a Java Virtual Machine – as usual for Java programs. There are, however, also some disadvantages. Due to the fact that JCSP is added on top of Java, it firstly has some performance overheads compared to occam which was designed as a parallel processing language from the scratch. And secondly, in JCSP the CSP semantics are not so straightforward as in occam because they all had to be coded as Java objects.

Despite all the differences, the requirements for a distributed version of channels in occam and JCSP are quite similar. They both should use a Channel Name Server to store information about network channels. And they both should implement the network channels so that processes could use them the same way as they use soft channels.

The JCSP.net project, however, did more than the implementation of normal Any to One network channels. Two important additional features were the implementation of Connections and anonymous channels.

A Connection is a special network channel. The end points are not a normal reader and a normal writer, and the communication is also not one way as it is with normal channels. The end points of a Connection are called "client" and "server". Connections enable the development of client/server style applications. The server registers the Connection with the CNS (by name) and interested clients can make a request about that Connection at the CNS who will then return the location of the server. So far, this is not different from normal Any to One channels.

What is different, however, is the fact that in order to communicate, a Connection has to be opened first. When the server and a client have opened the Connection then they are really connected. This is different from normal Any to One channels as normal writers (all of them) can write to a normal reader as soon as they have the location of this reader.

The second difference is that Connections are there for two way communication between clients and servers rather than normal network channels which are only one way. So Connections are a good solution when we do not intend to establish a network with fixed network channels but we rather have several clients which might be interested in connecting to a server and it is not known at compile time when a client needs to talk to the server. If more than one client is requesting to open the Connection to the same server, it has to wait until the Connection to the previous client has been closed. Should several clients have tried to open the Connection to a server which is busy, they form a queue before this server.

Anonymous channels are another special kind of network channels. As the name indicates, they do not have a name associated with them and they are not established by use of the Channel Name Server. The usual way of establishing an anonymous channel is to first create an anonymous reader. The location of the anonymous reader is not registered with the CNS but rather sent from a Connection server to a Connection client or vice versa. Then an anonymous writer can be connected to the anonymous reader using this location which has just been received.

The use of anonymous channels is particularly useful when we want to increase the availability of a Connection server. When a server performs all operations itself which are necessary during a Connection with a client, it could happen that the next client has to wait quite long for the server to finish. On the other hand, the server could delegate the work to a worker process of which there might run several in parallel with the server. The client could communicate with the worker process over anonymous channels which it has exchanged with the server before. The Connection between the client and the server could be closed as soon as the anonymous locations have been exchanged and the server would be available immediately for the next client while the previous client could still be communicating with the worker process.

## 2.3. THE OCCAM NETCHANS PROJECT

The occam NetChans project [17] was also an undergraduate final year project which was done in parallel with the JCSP.net project. Originally, both projects were part of a bigger project, but after some time they departed from each other and were run more or less independently. The objectives of both projects were the same – developing distributed channels and the use of a Channel Name Server.

Whereas the JCSP.net project fulfilled all the original objectives and even achieved more, such as Connections and anonymous servers, the occam NetChans project did not really achieve its aims. Especially the Channel Name Server was not implemented in a satisfactory way.

The libraries which were written during the occam NetChans project provide several different interfaces instead of one. Firstly, there are different interfaces for sending and receiving data. Secondly, each interface supports only the sending or receiving of one data type. There are even different interfaces for single data types and for arrays of these data types. The size of the arrays is fixed, so when users want to use an interface to send an array of another size they have to recompile the library which is not very sensible.

The occam NetChans project supports only four of the occam standard data types. As described above, there are four different interfaces for each data type which makes a total of 16 different interfaces. This is far from being satisfactory. If a programmer would want to send and receive several different data types over network channels, they would have to run several of these interfaces in parallel which creates an overhead which is not necessary. If all eight occam standard data types would have been supported this would even have doubled the number of interfaces to 32.

(Actually, the number of interfaces in the occam NetChans project is 32. This is due to the fact that there are also interfaces which do not use the CNS but connect directly to each other.)

The implementation of the CNS was not done as set out in the objectives of the project. In fact the CNS which was written in the occam NetChans project is not even a Channel Name Server. Neither does it support channels nor does it support names. A better name for it would have been "socket connection number server".

Firstly, the occam NetChans CNS does not store information about network channels but about the location of a machine with a sending interface. This information can be requested by a machine with a receiving interface. Then the occam NetChans CNS removes the information about the sending interface from its database as soon as a reading interface has read it. This means that each machine which has been registered with the CNS can only connect to one other machine. That way, a programmer who wants to send (or receive) data to different machines has to run several interfaces in parallel, even if they use the same data type.

The individual channels of an interface have to be referenced by their array index. Moreover, the sender and the receiver have to be connected to the same array index on both the sending and the receiving machine's interfaces. This is not sensible as it requires the user processes on the different machines to know exactly at which array index they have to connect to the interface.

Secondly, even the information about sending machines are not referenced by a name, but rather by the index of the position where this information is stored in the database of the occam NetChans CNS. So instead of thinking "I want to connect to the network channel called 'fred'" a programmer would have to think something like "I want to connect to the network channel with the index number x on the machine which is stored by the CNS under index number y". This is not satisfactory. This solution lacks transparency as well as scalability.

Nevertheless, the occam NetChans project implements a sensible way of emulating occam channel semantics over socket connections. I adopted that way of communication, even if I had to implement it completely new as my interface is totally different from what has been done in the occam NetChans project.

When a writer wants to send data to a network channel, it has to send it to the interface on its machine. The interface sends the data over a socket connection to the remote machine. The remote machine receives the data from the network and places it in a buffer. Finally the reader receives the data from the buffer. The buffer is necessary to prevent the reader's machine from being blocked while waiting for the reader to read the data. As we use multiplexing techniques, it must be ensured that the reader's machine can already receive new data from the network (which might be destined for another reader) also when the previously sent data has not yet been read by the first reader. By placing the data in the buffer (which can read the data before a reader might be able to read it) the receiving machine does not have to wait for the reader.

For readers, the handshake principle is fulfilled automatically. When they want to read data from a network channel they are automatically suspended until the data can be read from the buffer. For writers, however, this is not so straightforward. To get real CSP semantics, a writer can only be allowed to continue its work when it knows for sure that the reader has received the sent data. For that, there is a need for an acknowledgement to be sent back to the writer.

This is done by the buffer. When the reader has read the data from the buffer, an acknowledgement is sent back to the writer's machine over the network. For that, the occam NetChans project uses another socket connection than the one over which the data had been sent. This deviates from the requirement to have only one socket connection between two computers. In my code, I removed the redundant socket and use only one socket for the communication in both directions.

The interface on the writer's machine finally sends the acknowledgement to the writer. This means, in order to get CSP semantics, a writer has always to await the acknowledgement before continuing. So where for a soft channel a writer would only have to make an output, for a network channel a writer has always to do two operations: the output itself and the input of an acknowledgement.

## 2.4. The Progress Of This Project – How The DoP Developed

When I started this project, my ideas about what was I was going to do were rather general. I understood what was expected, but I had not worked out yet a detailed plan about how I was going to achieve the requirements of the specification with my software.

At the beginning, I was not so familiar with occam. I had had about five introductory lessons about occam during one of the modules of my course, but this could only give me the very first contact with the language and the principles behind it. So the first thing to do for me was to consult literature about occam and to understand occam by heart. A particularly useful guide for me was [5] as it is short and easy to understand but very comprehensive as it covers all the important occam constructs.

Nevertheless, everything – also a programming language – is learnt best by doing. This is the reason why now, after having finished the project, my understanding of occam and even the understanding of how my code works is much bigger than at the beginning. If now I had to write everything from the scratch again, I would certainly need much less time then I did now.

Originally, I was a little naïve about what I could achieve in the time given. I was thinking of playing around with the compiler or the occam kernel; and I thought of many possible further improvements and extensions, for example the dynamic distribution of frozen processes [19] over the network. But this would have exceeded the limited time of this project, especially as I had a wrong imagination about how far the occam NetChans project had already progressed.

Before I started the project, I thought that both the JCSP.net project and the occam NetChans project would more or less have achieved to fulfil the objectives. But then, talking with my supervisor, I found out that the interfaces which were produced in the occam NetChans project were not really feasible. This is the reason why we agreed that for the time being the main issue was to achieve the same level on the occam side which had previously been achieved on the JCSP side by the JCSP.net project.

Since the JCSP.net project had even done more, after a while we agreed to amend the aims of my project and that I should also implement Connections and anonymous channels. Due to all this, there was no more time left to investigate the distribution of frozen processes or possible changes of the compiler or the occam kernel. All this is subject to further research work.

Nevertheless, what I think has been achieved is that the interface of the Distributed occam Protocol now has a standard which is comparable to what has been done in the JSCP.net project.

## 2.5. THE DOP FROM A NETWORKING PERSPECTIVE

An interesting question is what is the scope for distributed CSP channels from a networking point of view. Network models, for instance the ISO-OSI reference model, are usually built up in form of layers [20]. The lowest layer is the physical layer which is concerned with the physical sending and receiving of data. The highest layer is the application layer which abstracts the technical aspects of a network and is concerned with the communication between applications.

In the TCP/IP model there are no layers between the transport layer and the application layer. OSI's session layer and presentation layer are rather abstract, and in everyday use they are rarely necessary. This is why they have been omitted in the TCP/IP model. Therefore, the application layer is quite extensive in the TCP/IP model. In the end, everything which builds on TCP connections is said to be in the application layer. This covers telnet and ftp as well as HTTP.

The DoP, however, is a little different. It is obviously situated on top of TCP, i.e. on top of the transport layer. But is it therefore automatically in the application layer? One could argue that it is not, but rather something in between. As occam applications can be built on top of the DoP, it can be considered an own layer. This is, however, dependent on the point of view. Standardisation is always a matter of definitions, and every definition is influenced by personal attitudes of the people who makes it.

# 3. A User Guide To The DoP

## 3.1. The DoP Interface

The Distributed occam Protocol provides an occam library. In order to use this library, programs have to use and include the following files:

```
   #INCLUDE "doplib.inc"
   #USE "dop.lib"
```

The main part of the DoP library is its network interface:

```
PROC dop.interface([dop.count.channels]CHAN OF DOP.PACKET
   from.net, to.net,
   CHAN OF DOP.PACKET reconnection.channel,
   CHAN OF BOOL reconnection.ack.channel,
   VAL []BYTE my.ip, VAL INT my.port,
   VAL []BYTE cns.ip, VAL INT cns.port)
```

The interface provides two arrays of channels. One of these arrays is called `from.net` and is used to read data from the DoP interface. The other one is called `to.net` and it us used to send data to the DoP interface. All user processes that want to use the DoP interface have to run in parallel with it and to communicate over these channels. The end point of each network channel is a pair of `from.net` and `to.net` channels with the same index.

The DoP library, however, also offers a number of auxiliary processes which ease the interaction with the DoP interface as the programmers of the user processes do not have to cope with the structures of messages by whom they communicate with the interface.

The protocol of the `from.net` and `to.net` channels is called `DOP.PACKET`. It is a counted array protocol with `INT` as count type and `BYTE` as data type. This means that the user processes send and receive byte arrays of variable size to or from the DoP interface. This makes sense, as the data which is sent over socket connections also consists of byte arrays.

The other parameters of the DoP interface will be dealt with in the following sections of this chapter.

### 3.2. CONFIGURATION

There are two ways to configure the DoP interface. Most of the options have to be known at compile time. The options can be changed by changing the values of constants of the `doplib.inc` file (in the following called "configuration file") and recompiling the DoP library.

The number of channels provided by the DoP interface, i.e. the size of the `from.net` and `to.net` arrays is defined by `dop.count.channels`. The standard value is 100.

The number of socket connections provided by the DoP interface, i.e. the number of machines it can connect to is defined by `dop.count.sockets`. The standard value is 50.

The maximum size of a DoP packet i.e. the maximum number of bytes that can be sent over the network as one packet is defined by `dop.max.packet.size`. The standard value is 65536 which is 64K.

The other options that can be adjusted in the configuration file will be dealt with in the following sections of this chapter.

The second type of options of the DoP interface are local options. These are the values which are necessary to identify the location of a concrete running instance of the interface, namely the IP address or the host name of the machine it is running on and the socket port number it is listening on.

These two values are passed to the local DoP interface as parameters. `my.ip` is a string which contains the local IP address in the usual notation (“`x.x.x.x`”) or the local host name. `my.port` is an integer which contains the local socket port.

For ease of use, there exists a file called `dop.local.config.inc` (in the following called “local configuration file”). This file should be placed in the directory of each program which wants to use the DoP interface. It contains two constants, namely `dop.local.ip` and `dop.local.port` which store the local IP address or host name and the local socket port. Programmers should adjust those values in the local configuration file and include the file in their program. Then the two constants should be passed to the DoP as parameters.

## 3.3. Configuration Of The Channel Name Server

The Channel Name Server is configured similarly as the DoP interface. There are two files in the Channel Name Server’s directory called `dop.cns.config.inc` (in the following called “CNS configuration file”) and `dop.cns.config.export.inc` (in the following called “exportable CNS configuration file”).

The CNS configuration file contains a constant called `dop.cns.count.channels` which defines the number of channel locations the CNS can store in its database. The standard value is 500.

The exportable CNS configuration file contains two constants, namely `dop.cns.ip` which stores the IP address or host name of the machine where the CNS is supposed to be running on, and `dop.cns.port` which stores the socket port the CNS is listening on.

All three options can be changed by changing the values in the CNS configuration file or the exportable CNS configuration file respectively and recompiling the CNS program.

The IP address or host name and the socket port of the CNS are values which also have to be known by each local DoP interface. They are passed to the local DoP interface as parameters. `cns.ip` is contains the CNS IP address or host name. `cns.port` contains the local socket port.

For ease of use, the exportable CNS configuration file should be passed to all programmers who write programs which want to use this CNS (i.e. the CNS with the location stored in the exportable CNS configuration file). The file should be placed in the directory and included in the program. Then the two constants should be passed to the DoP as parameters.

### 3.4. How To Build A Distributed occam Network

Writing a program which runs on a single machine or a single transputer is quite easy. The main program just defines some channels over which the occam processes shall communicate. Then it runs the processes in parallel and passes the channels to the processes as parameters.

When occam processes shall be distributed to several transputers who are connected by links, the main program has to perform a `PLACED PAR` [3] which means that it has to assign (i.e. place) occam channels to transputer link addresses (i.e. the number of the link and whether it is the incoming or the outgoing channel of this link) prior to passing the channels to the processes as parameters.

In the case of network channels there is a similar approach. At first, the main program has first to define two arrays (of size `dop.count.channels`) of channels of the DoP packet protocol.

```
[dop.count.channels]CHAN OF DOP.PACKET from.net, to.net:
```

(Of course they need not be called `from.net` and `to.net`, this is just due to better understanding as it indicates to which parameters of the DoP interface these arrays have to be passed to.)

Then the DoP interface, which has been passed the `from.net` and `to.net` parameters, has to be run in parallel with a sequence which first registers the network channels with the Channel Name Server and then runs the user processes, which have also been passed the appropriate `from.net` and `to.net` channels, in parallel. Figure 3.1 shows the resulting structure.

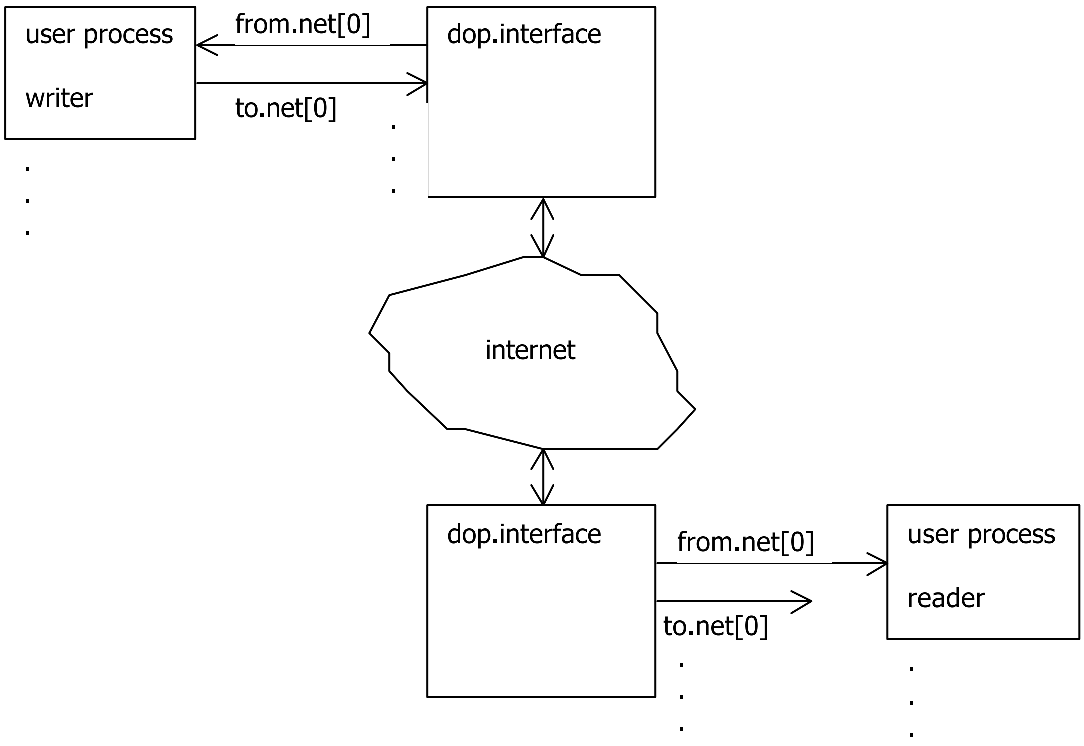

**Figure 3.1**

Usually, once registered (which is done in the main program) a reader need not send data to the DoP interface. Therefore, readers are usually not connected to the interface via a `to.net` channel but only via a `from.net` one.

As already mentioned above, in order to get CSP semantics, a writer always has to await an acknowledgement from the interface before it can continue. This is the reason why the writer is connected to the interface both via a `to.net` and a `from.net` one. For ease use, user processes can use the following process to receive an acknowledgement from the interface:

```
PROC dop.receive.ack(CHAN OF DOP.PACKET from.net)
```

The parameter `from.net` is the channel on which the acknowledgement is received from the interface, i.e. the `from.net` channel.

## 3.5. Registration

As mentioned in the previous section, before a user process can use a network channel, the network channel has to be registered. Please note that I use the term registration for both when readers register their location with the CNS and when writers make a request at the CNS about a reader.

The usual way to register a network channel is as described above – in the main program before starting the user process that uses the network channel. However, this is only a suggestion. Generally, the registration can take place any time before a network channel is used. So also a user process itself could register a network channel before using it. The user process could even wait and see and only register a network process when it is really needed the first time.

The registration of a reader or a writer takes place by sending a special message to the DoP interface over the `to.net` channel. The first message which is sent to the interface has to be a registration. When a process tries to send user data to the interface before the network channel is registered, this is very likely to produce an error which will cause the interface to crash.

For the sake of convenience, there are several registration processes which perform the sending of a registration message to the interface and react with the response. The basic registration processes are

```
PROC dop.reg.write(CHAN OF DOP.PACKET from.net, to.net,
   VAL []BYTE channel.name)
PROC dop.reg.read(CHAN OF DOP.PACKET from.net, to.net,
   VAL []BYTE channel.name,
   VAL INT buffer.size,
   BYTE result)
```

where dop.reg.write() is used to register as a writer and dop.reg.read() is used to register as a reader. from.net and to.net are the end point of the network channel. channel.name is a string which indicated the name of the network channel.

The registration process for a writer does not end until the CNS request for a reader with the relevant name has been successful. Should the CNS not know about the reader yet (because it has not yet registered) it will tell this the DoP interface which is connected to the writer. This will then wait for some time and retry the request. It will do so infinitely until finally it has been successful.

The time interval between two successive tries is a constant which is stored in the configuration file. The name of the constant is dop.reg.time.wait. It contains the waiting time in microceconds (as usual in occam). The standard value is 5000000 which means that the DoP interface will wait five seconds before retrying to register.

To prevent a livelock, the order of the registrations is important. Although this is not the only livelock-free solution, it is a good rule to first register all readers and then all writers. Otherwise the following scenario would be possible, for instance: Two machines want to establish two network channels between each other for either direction. They both first want to register their writer and their which causes infinite retries to request the corresponding readers' location because neither of the machines manages to register its reader.

One parameter of the registration process for a reader is buffer.size. This is needed to indicate the size of the buffer which is placed before the reader. As mentioned above, we need in any case one buffer in order to prevent the reader's machine from being blocked. In the following I will call this buffer "acknowledgement buffer" because (as mentioned above) it is the one who sends an acknowledgement back to the writer's machine as soon as the reader has read the data.

However, due to the Any to One semantics of network channels, a single buffer might not be enough to prevent a blocking of the reader's machine. If several writers should write to a network channel and the reader has not read them yet, the data of the different writers has to be queued before the reader. For that we need a buffer additional to the acknowledgement buffer. The `buffer.size` parameter indicates the size of that additional buffer. A value of 0 means that there will only be the acknowledgement buffer but no further one.

The maximum size of such an additional buffer is stored in the configuration file. The name of the constant is `dop.max.buffer.size`. The standard value is 20 which means that the `buffer.size` parameter can be passed values between 0 and 20.

The Channel Name Server's response to the registration of a reader will be returned by the registration process in the parameter `result`. This is an integer which can return three different values. The possible values are stored as constants in the configuration file and will therefore be available to the user program as it has `INCLUDE`d the configuration file. These are the possible values:

`DOP.CNS.OK` means that the registration of the reader was successful and the reader's location is now stored in the Channel Name Server's database.

`DOP.CNS.FULL` means that the registration was not successful because the database of the CNS was already full.

`DOP.CNS.NAME.EXISTS` means that the registration was not successful because another reader has registered before under the same name.

The result should be checked and considered somehow. Typically, it should be asserted that the registration was successful by calling ASSERT(result = DOP.CNS.OK) after the registration.

### 3.6. SUPPORT OF DIFFERENT DATA TYPES

Until this point, it has been assumed that the type of the data which a network channel can carry is a counted array of bytes. However, the objectives say that network channels of all standard occam data types and arrays thereof shall be supported by the DoP interface. As socket connections carry bytes, it is generally a good choice that a DoP packet also is a counted array of bytes. If we want to use other data types, we must retype the data from them to an array of bytes before sending it over the network and vice versa after having received it from the network.

There are two ways of doing that. Either a small conversion process has to be plugged between the writer and the DoP interface and between the remote interface and the reader, or the user processes have to do the retyping themselves.

The DoP library offers conversion processes to convert from and to all eight occam standard data types. It also offers conversion processes for counted array protocols of all eight occam standard data types with both INT and BYTE as count type. There is only one exception: As the DOP.PACKET protocol is defined as INT::[]BYTE, there is no need for conversion processes for this protocol, so these two conversion processes have been omitted.

This makes a total of 46 conversion process. As it is not useful to describe all of them, I will rather describe their usage in general, as apart from the different data type they support they are equal.

In the following, {TYPE} stands for one of the standard occam data types. {type} stands for the same, but written in small letters. So {TYPE} is a substitution for one of the following:

INT, INT16, INT32, INT64, BYTE, BOOL, REAL32, REAL64,

whereas {type} is a substitution for one of the following:

int, int16, int32, int64, byte, bool, real32, real64.

There are three variants of conversion processes from {TYPE}:

```
PROC dop.conv.from.{type}
   (CHAN OF {TYPE} from.proc,
   CHAN OF DOP.PACKET to.net)
PROC dop.conv.from.ca.i.{type}
   (CHAN OF INT::[]{TYPE} from.proc,
   CHAN OF DOP.PACKET to.net)
PROC dop.conv.from.ca.b.{type}
   (CHAN OF BYTE::[]{TYPE} from.proc,
   CHAN OF DOP.PACKET to.net)
```

The first process converts from {TYPE} to a DoP packet. The second and the third process convert from a counted array protocol with {TYPE} as data type to a DoP packet. The count type is INT in the second and BYTE in the third process.

The parameters are quite obvious: from.proc is the channel to be connected to the user process and to.net is the channel to be connected to the DoP interface.

The corresponding three variants of conversion processes to {TYPE} are the following:

```
PROC dop.conv.to.{type}
   (CHAN OF {TYPE} to.proc,
   CHAN OF DOP.PACKET from.net, to.net)
PROC dop.conv.to.ca.i.{type}
   (CHAN OF INT::[]{TYPE} to.proc,
   CHAN OF DOP.PACKET from.net, to.net)
PROC dop.conv.to.ca.b.{type}
   (CHAN OF BYTE::[]{TYPE} to.proc,
   CHAN OF DOP.PACKET from.net, to.net)
```

The naming conventions are the same as for the conversion processes from {TYPE}.

The parameters are also obvious: `from.net` is the channel to be connected to the DoP interface and `to.proc` is the channel to be connected to the user process.

There is, however, an exception. The conversion processes to {TYPE} also have a `from.net` parameter which is also the channel to be connected to the interface. This is necessary in order to keep CSP semantics. The acknowledgement is only allowed to be sent back to the writer when it is sure that the reader has read the data. If the reader is connected directly to the interface (without a conversion process), this is done by the acknowledgement buffer as soon as it has written the data out.

If there is a conversion process between the acknowledgement buffer and the reader, then the fact that the acknowledgement buffer has written the data out does not mean that the reader has read it. It rather means that the conversion process has read it. So in order to be able to send the acknowledgement back to the writer, the acknowledgement buffer has to a await an acknowledgement from the conversion process. The conversion process will send the acknowledgement to the interface (i.e. to the acknowledgement buffer) as soon as the reader has read the converted data. This is the reason why we also need a `from.net` parameter for conversion processes to `{TYPE}`.

In order to let the DoP interface know which kind of acknowledgement buffer it has to use, it is necessary to inform it whether there is a conversion process between a reader and the interface or not. This is done during registration. Therefore, we need a separate registration process for readers which use converted data:

```
PROC dop.reg.read.conv
   (CHAN OF DOP.PACKET from.net, to.net,
   VAL []BYTE channel.name,
   VAL INT buffer.size,
   BYTE result)
```

The parameters are the same as for the `dop.reg.read()` process. The difference is only internal as the DoP interface is told about the fact that there is a conversion process. So it knows which kind of acknowledgement buffer it has to use.

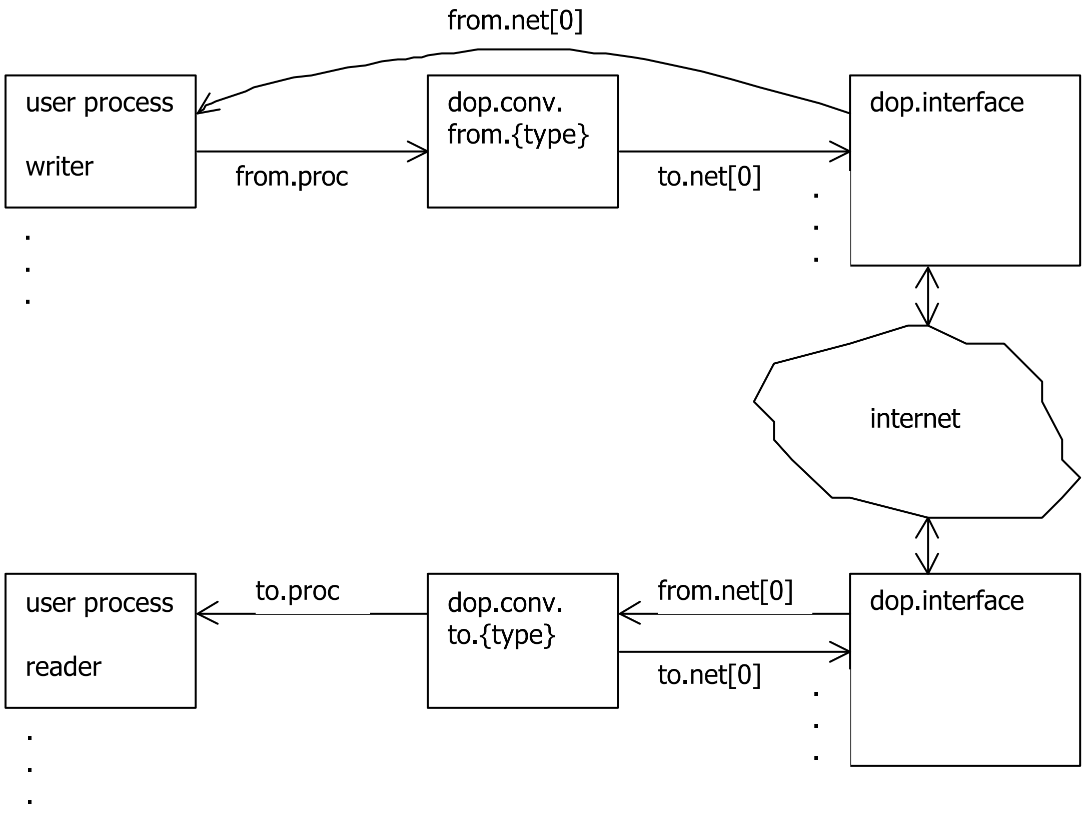

**Figure 3.2**

Figure 3.2 shows the structure when conversion processes are in use. As can be seen, the network is more complex now. This is a drawback of the use of conversion processes. If we do not want this complexity in our networks, the user processes have to perform the retyping themselves. For ease of use, the DoP library offers several processes which user processes can call to retype `{TYPE}` data to a DoP packet and send it to the interface or to receive a DoP packet from the interface and retype it to `{TYPE}` respectively.

There are processes to send and receive data of all occam standard data types as well as arrays of all types. The only exception are byte arrays as a DoP packet is already a counted byte array. This makes a total of 15 sending and 15 receiving process. These are the processes provided:

```
PROC dop.send.{type}(VAL {TYPE} data,
   CHAN OF DOP.PACKET to.net)
PROC dop.send.array.{type}([]{TYPE} data,
   CHAN OF DOP.PACKET to.net)
PROC dop.receive.{type}({TYPE} data,
   CHAN OF DOP.PACKET from.net)
PROC dop.receive.array.{type}([]{TYPE} data,
   INT size, CHAN OF DOP.PACKET from.net)
```

The parameters should be obvious. `data` is the data from the user process or the data which is returned to the user process respectively. `from.net` and `to.net` are the network channel end points of a reader or a writer respectively. Finally, `size` is the size of the `{TYPE}` array which was received. (After the retyping! Not the size of the DoP packet from which it was retyped.)

Please note that the use of sending processes does not exempt writers from the duty to await an acknowledgement (for which, as already mentioned, `dop.receive.ack()` can be used).

An advantage of sending and receiving processes over conversion processes is that several different data types can be sent over the same channel. This is particularly useful for Connections – where conversion processes cannot be used anyway (see next section).

## 3.7. Connections

As outlined in chapter 2, Connections are special network channels which allow two way communication between a client and a server. The advantages of Connections for the development of client/server style applications were also already mentioned.

The registration of Connection servers and Connection clients is more or less identical with the registration of normal readers and writers. The following registration processes are provided:

```
PROC dop.reg.conn.client
   (CHAN OF DOP.PACKET from.net, to.net,
   VAL []BYTE channel.name)
PROC dop.reg.conn.server
   (CHAN OF DOP.PACKET from.net, to.net,
   VAL []BYTE channel.name,
   VAL INT buffer.size, BYTE result)
```

The parameters are the same as for `dop.reg.write()` and `dop.reg.read()`. There is, however, a peculiarity with Connections. The `buffer.size` parameter of `dop.reg.conn.server()` means the size of the buffer which is used to form a queue of clients waiting for the server. For the actual communication between a client and the server there is no buffer necessary additional to the acknowledgement buffer. This is due to the fact that the communication over a Connection implements One to One semantics, i.e. as long as the Connection is open between a client and the server, no other client can communicate with the server.

But the request to open a Connection follows Any to One semantics, as several clients can make such a request at the same server. There is also an acknowledgement buffer which sends an acknowledgement to a client when the server has accepted to open the Connection. If there was only the acknowledgement buffer, it would have to be assured that maximum one client would try to connect to the server while the server is busy. Otherwise the server's machine would be blocked.

Therefore, it is likely to need an additional buffer. The `buffer.size` parameter indicates the size of that additional buffer. A value of 0 means that there will only be the acknowledgement buffer but no further one.

The maximum size of such an additional buffer is stored in the configuration file. The name of the constant is `dop.max.conn.buffer.size`. The standard value is 20 which means that the `buffer.size` parameter can be passed values between 0 and 20.

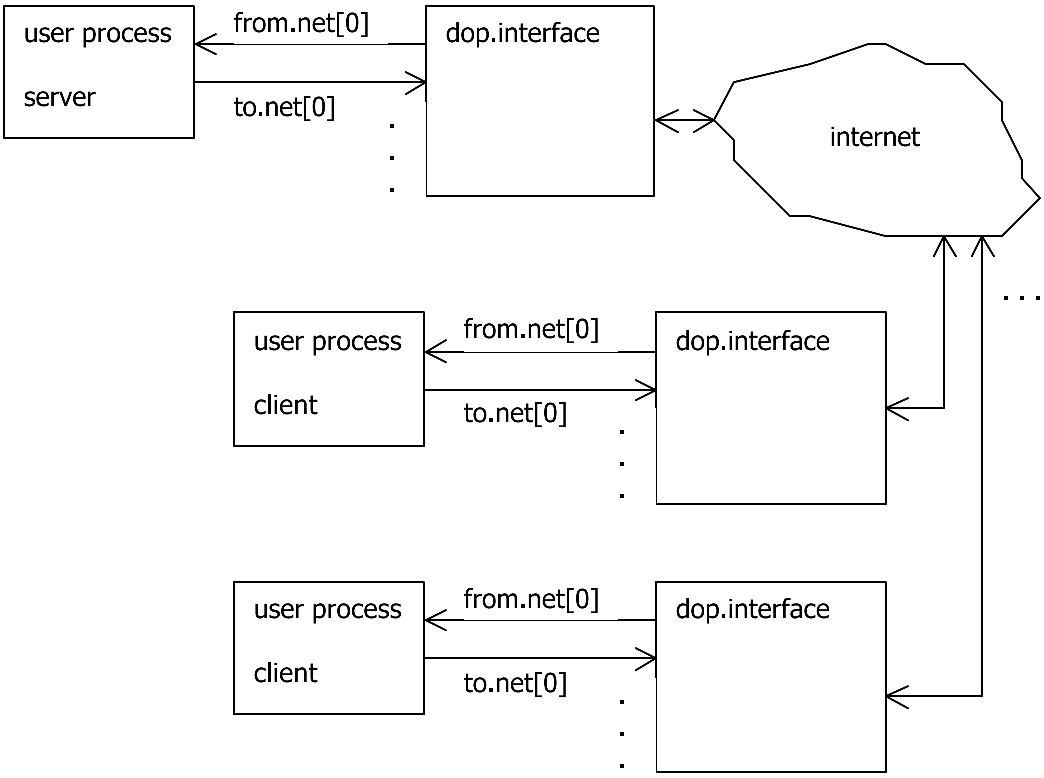

**Figure 3.3**

Figure 3.3 shows the structure of a Connection. As can be seen, both clients and servers are connected to the DoP interface via both `from.net` and `to.net`. This is necessary to allow two way communication between client and server. Please note that even if the Connection between the clients and the server is established, i.e. the clients know the server's location, communication between a client and a server is only possible when the Connection has been opened first.

As Connections are more sophisticated than normal Any to One channels, it is necessary to give the DoP interface instructions on how to use them. Firstly, it is necessary to tell the interface when to open and to close a Connection. And secondly, the interface has to know if the client wants to write to the server (which I will call "request" in the following) or vice versa (which I will call "response" in the following).

As these instructions are sent to the interface over the same channels as the data is sent (`from.net` and `to.net`), the interface must find a way to find out what is data and what is an instruction. This is done quite simple. Both client and server have to do the following: The first thing to send to the interface after registering a client or a server is a request to open the Connection. After that there are three possibilities: Either a request instruction is sent to the interface or a response instruction or an instruction to close the Connection. It is important that both client and server always send the same instructions.

When a client has sent a request instruction, the next step for it is to send data to the server. This is done the usual way as it is done with normal Any to One channels too. This means in particular that the client has to await the acknowledgement. When a client has sent a response instruction, the next step for it is to read data from the server, as well the usual way.

The server has to do this vice versa, i.e. it has to read after it has sent a request instruction and it has to write after a response instruction. What is important is that it does not matter which instruction is sent when, as long as server and client always send the same instructions. This means that the communication can absolutely start with a response or that there might be several requests or responses successively. Figure 3.4 shows the state diagrams for servers and Figure 3.5 shows the one for clients.

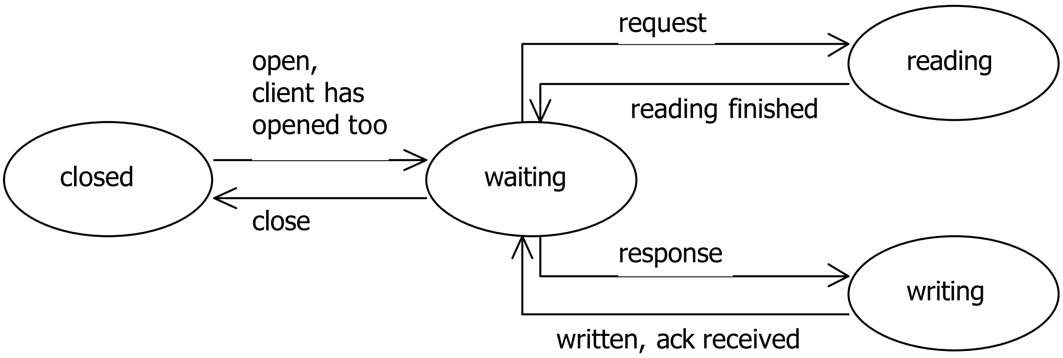

**Figure 3.4**

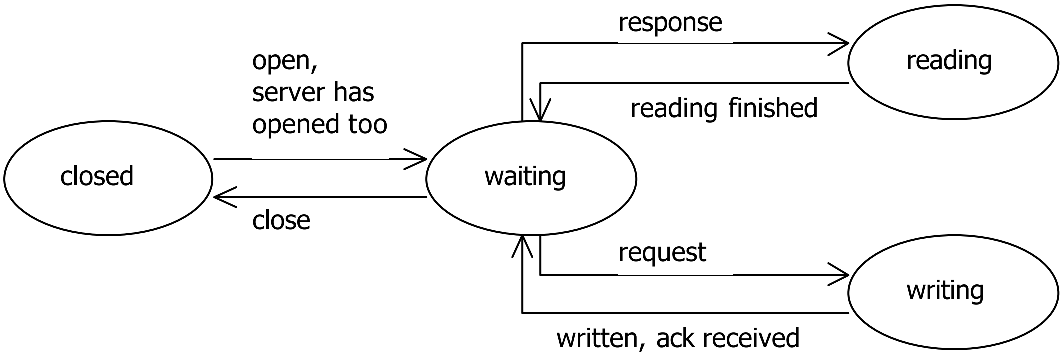

**Figure 3.5**

The DoP library offers four processes to send the instructions to the interface:

```
PROC dop.conn.open(CHAN OF DOP.PACKET from.net, to.net)
PROC dop.conn.close(CHAN OF DOP.PACKET to.net)
PROC dop.conn.request(CHAN OF DOP.PACKET to.net)
PROC dop.conn.response(CHAN OF DOP.PACKET to.net)
```

The parameters are the usual `from.net` and `to.net` channels. `dop.conn.open()` needs `from.net` because it has to wait for the acknowledgement that the other side of the Connection has also opened.

It is not possible to use conversion processes with Connections. There are two reasons. Firstly, both clients and servers can both read and write. This means that on each end both conversion processes would have to be plugged in: from {TYPE} and to {TYPE}. As both of them need to be connected over a to.net channel, this is not possible. And secondly, clients and servers send both data and instructions to the interface. If there was a conversion process plugged in between, the instructions would be converted as well, which hardly makes any sense.

## 3.8. ANONYMOUS CHANNELS

As outlined in chapter 2, anonymous channels are a neat way to reduce the workload of connection servers. The difference between normal Any to One channels and anonymous channels is that they are not established by name over the CNS but rather anonymously by exchanging the reader's location.

The registration processes for anonymous channels are the following:

```
PROC dop.reg.write.anonymous(CHAN OF DOP.PACKET to.net)
PROC dop.reg.read.anonymous
   (CHAN OF DOP.PACKET from.net, to.net,
   DOP.CHANNEL.LOCATION the.channel.location,
   VAL INT buffer.size)
PROC dop.reg.read.anonymous.conv
   (CHAN OF DOP.PACKET from.net, to.net,
   DOP.CHANNEL.LOCATION the.channel.location,
   VAL INT buffer.size)
```

When an anonymous reader is registered, we do not give it a name. The CNS is not involved, rather the location of the reader is returned to the caller of the registration process. The return value `the.channel.location` is of type `DOP.CHANNEL.LOCATION`. This is a record which stores the IP address, the port number and the channel index of an endpoint of a network channel. In the following, I will call data members of this type "channel locations". The other parameters are as with normal Any to One channels.

When an anonymous writer is registered, we do not need a name as well, as there is no request made at the CNS. The only thing that happens is that the DoP interface now knows that this network channel end point is an anonymous writer. In order to connect an anonymous writer to a reader, we need to get the reader's channel location.

The channel location of an anonymous reader is usually exchanged during a Connection. To do so, the DoP library offers two processes:

```
PROC dop.send.channel.location(DOP.CHANNEL.LOCATION
   the.channel.location, CHAN OF DOP.PACKET to.net)
PROC dop.receive.channel.location(DOP.CHANNEL.LOCATION
   the.channel.location, CHAN OF DOP.PACKET from.net)
```

The parameters are the channel location which shall be sent or received and the usual `from.net` and `to.net` channels.

Once we have the reader's channel location, the writer con be connected to the reader. This is done by the following process:

```
PROC dop.reconnect.anonymous(
   CHAN OF DOP.PACKET reconnection.channel,
   VAL INT channel.index,
   VAL DOP.CHANNEL.LOCATION the.channel.location,
   CHAN OF BOOL reconnection.ack.channel)
```

`reconnection.channel` and `reconnection.ack.channel` are two special channels to communicate with the DoP interface via whom reconnections of writers are done. These channels have also to be passed to the DoP interface as parameters. (Should we not need anonymous channels in a program, we can pass two dummy channels to the interface.) `channel.index` is the index of the anonymous writer which has to be connected. Finally, `the.channel.location` is the channel location of the anonymous reader which we just received.

`dop.reconnect.anonymous()` can be used several times to connect a writer to another anonymous reader. This means that the end poins of anonymous channels can be changed. Moreover, we can use `dop.reconnect.anonymous()` even to reconnect normal writers (which have previously been connected via the CNS) to an anonymous reader. This makes anonymous channels a powerful tool.

## 3.9. TESTING NETWORKS

During the project, I created two testing networks. One is for testing normal Any to One channels and the other one is for testing Connections and anonymous channels. Readers who want to test these networks should copy the contents of the enclosed CD-ROM to their hard drive and follow the instructions to compile the library and the programs.

Before a testing network can be started, it is necessary to start the CNS. It is the executable `dop.cns` in the `/dop/dop.cns` directory. The testing network for normal Any to One channels is started by starting the executables in the `machine.1`, `machine.2` and `machine.3` subdirectories of the `/dop/test.normal.channels` directory. The testing network for Connections and anonymous channels is started by starting the executables in the `server.machine`, `client.machine.1`, `client.machine.2` and `client.machine.3` subdirectories of the `/dop/test.connections.and.anonymous.channels` directory.

The local IP addresses in the local configuration files are set to the local host (“`127.0.0.1`”) which means that the different parts of the networks can be executed in different terminal windows of any Linux machine.

The first testing network consists of three machines. The first machine contains two processes, `a` and `b`. `a` is waiting for a request over a Boolean channel network channel. When it has got that request, it asks the user to type in an integer. Then it outputs it over an integer network channel. This all is repeated infinitely in a `WHILE TRUE` loop. `b` is a simple process which inputs a string over a network channel and outputs it to a channel which is connected to the standard output. The second machine contains two processes, `c` and `d`. `c` does the same as a, but for strings. d does the same as b, but for integers.

The third machine contains one process, e. e asks the user for a choice between machine one and two. According to that choice, it makes a request at the chosen machine and reads a string or an integer respectively from it which it then sends to the other of the two machines.

In the first network, the general network channel semantics were tested as well as special features like conversion processes.

The second testing network consists of one server machine and three client machines which are equal. The client’s task is to send strings to the server machine where they are displayed on the screen. The client offers the user two possibilities: to do it without workers or with workers. Then it opens a Connection to the server. Without workers, the Connection is open as long as the client gets string inputs because the displaying of the strings is done by the server itself. The user can end the inputs by entering an empty string. Then the Connection is closed.

With workers, the Connection is only open for a short while. until the server has sent the location of an anonymous reader to the client. The client then connects to this reader, and over this network channel it then sends the strings to one of ten worker processes, which run in parallel with the server, rather than to the server itself. As the worker processes are performing the task of displaying the strings, the server can in the meanwhile connect to another client. By playing around with this network, this difference will become recognisable as the server is blocked or not according to the option chosen.

# 4. How It Works – A Look Inside The DoP

## 4.1. General Ideas – The Parts Of The DoP Interface

In order to perform its work, the DoP interface needs two databases where it can store important values. The first one, called "socket location database", stores all machines to which the interface is connected and the socket which is used to communicate with that machines. The machines are identified by their IP address and port number.

The second database, called "channel destination database", stores for each pair of `from.net` and `to.net` channels the socket (by its index in the socket location database) to the remote machine where the other end point of the network channel is and the other end point's channel index on that machine.

The acknowledgement buffers and the additional buffers for Any to One channels and for waiting Connection clients are written as small internal processes which can be launched on demand.

There are four main processes which are running in parallel in the DoP interface: The acceptor, the receiver, the registrar and the reconnector. As these processes are using shared variables and channels (for instance the socket location database), we need `SEMAPHORE`s [15] to protect them.

The registrar is a parallel process which reads in parallel on all `to.net` channels to register. When it is requested to register a reader or a Connection server, the registrar connects to the Channel Name Server and registers the network channel there. Then it sets up the acknowledgement buffer and, if needed, the additional buffer in parallel.

When the registrar is requested to register a writer or a Connection client, it makes a request at the CNS and gets the reader's or server's channel location respectively. Then the registrar first looks in the socket location database if there is already a socket connection to the remote machine. If not, it establishes one by an algorithm which I called the "handshake algorithm". This algorithm is described in section 4.2. As soon as there is a socket connection to the remote machine, the registrar saves the socket index and the channel index of the remote end point in the channel destination database. Then, in case of writers, it starts an infinite loop that reads on the `to.net` channel and send any received data down the socket, preceded by a data message.

For anonymous channels, the registrar acts quite similar, but without contacting the CNS. For anonymous readers it returns the channel location of the reader to the caller of the registration. Then it sets up the buffers the same way as for normal readers.

For anonymous writers, the registrar sets up the infinite loop immediately without coping with the channel destination of that writer. Before this writer can be used, it has to be reconnected anonymously. The reconnector listens on the reconnection channel and when a request for a reconnection of a writer is made, it does the same thing that the registrar would do for a normal writer after it has got the reader's channel location from the CNS. This means that the reconnector updates the channel destination database for the writer it is requested to do so. This might be any writer, an anonymous one or a normal one which has been connected via the CNS before.

Messages which are sent over the network contain the message type, the channel index of the writer (on the writer's machine) and the channel index of the reader (on the reader's machine). The receiver reads from all sockets which are stored in the socket location database. When it gets a data message, it creates a new channel destination for the acknowledgement. The socket index is the socket index from where the data has been received and the channel index is the channel index of the remote writer, which is stored in the message. Then it passes the new channel destination and the sent data to the buffer of the reader. The acknowledgement buffer sends an acknowledgement message, using the new channel destination, as soon as the reader had read the data. When the receiver gets a message to open a Connection it does the same as if it would get a data message, but with the difference that it only sends the new channel destination (no data, as there was no data sent) to the Connection buffer. The new channel destination is used to update the server's channel destination database for the time the Connection is open.

When the receiver reads an acknowledgement message, it just sends an acknowledgement to the writer over the `from.net channel`. When it reads an acknowledgement message to a Connection open request, it does the same, but it sends it to a special channel on which clients are waiting for an acknowledgement to their open request.

For connection servers, the registrar acts quite similar as for readers. It sets up the Connection buffers (which are used to build a queue of clients). In parallel to that it reads from the server and waits for an open. Then it reads the channel destination of the first client which wants to open a Connection from the Connection acknowledgement buffer (which causes the Connection acknowledgement buffer to sends an acknowledgement message to the client) and updates the server's entry in the channel destination database. Then the registrar waits for instructions. When a request instruction comes, it does the same as an acknowledgement buffer would do (waiting for data from the receiver and sending an acknowledgement message as soon as the server has read the data), but only once instead of infinitely. When a response instruction comes, it acts as a writer, i.e. it sends a data message and the data to the remote machine. Then it waits for the next instruction and so on until a close instruction comes.

For Connection clients, the registrar reads from the client and waits for an open. Then it sends a message to the server to open a Connection. Then it waits for an acknowledgement from the receiver. After that, the client does the same as the server, only the reactions to request and response are the exact opposite.

In order to better understand the communication paths, I will explain step by step what happens when a writer sends data to a network channel. (For Connections this is corresponding.)

1) The writer sends data to its `to.net` channel.
2) The registrar creates a new data message which contains the channel index of the writer as source channel index and the channel index of the reader (which is stored in the channel destination database) as destination channel index.
3) The registrar sends this message and the data to the remote interface.

4) The remote receiver reads the message and the data.
5) The remote receiver creates a channel destination for the acknowledgement, which contains the socket index from where the message was received and the source channel index of the message (i.e. the channel index of our writer)
6) The remote receiver passes the channel destination and the data to the buffer before the remote reader.
7) When it is its turn, the channel destination and the data arrive at the remote acknowledgement buffer.
8) The remote acknowledgement buffer sends the data to the reader's `from.net` channel.

9) The remote acknowledgement buffer creates a new acknowledgement message which contains the channel index of the acknowledgement channel destination as destination index.

10)The remote acknowledgement buffer sends the acknowledgement message to the socket with the socket index from the acknowledgement channel destination.

11)Our receiver reads the message

12)Our receiver sends an acknowledgement to our writer’s `from.net` channel.

## 4.2. How To Ensure A Single Socket Connection Between Two Machines – The Handshake Algorithm

An important problem was how to assure that there is only one socket connection between two machines. The general idea is that at first a writer usually connects to the remote machine. The remote acceptor accepts that socket connection and both of them store their new socket in the socket location database, together with the machine details of the other machine each.

Please note that by “writer” I mean every part of the interface that might get information about a remote machine (IP address and port number) and want to connect to it. This could be the registrar where a writer or a Connection client has just been registered or the reconnector which wants to connect to a previously unknown machine.

Before a writer connects to the remote acceptor, it looks in its own socket location database whether there is already a socket to the concerning machine. If not, it connects to the remote acceptor in order to set up such a socket. But there is a trap, as at the same time there might a remote writer try to connect our acceptor. At the time of the database check, both our writer and the remote writer have no information about the other machine yet each. But then suddenly we have two socket connections between the two machines which is not suitable.

The naïve solution would be just not to allow the socket which comes second to be stored in the database. But this is not suitable as well as it might happen that on both machines the acceptors (say) are the first to store the new socket in the database, and the writers would not be able to do so and would have to close their sockets. This would result in two cut socket connections each of whom one socket is stored and the other one has disappeared.

Another naïve solution would be that a writer has to claim a semaphore for the database before the database check which would not be released until the newly established socket is stored in the database. The remote acceptor would have to claim the database as well in order to assure that there is no writer on the remote machine which might save an entry about our machine in the meanwhile. This would mean that the database would not be available during the whole handshake algorithm, which is not very good from a performance point of view. But this is not the main problem. The problem is that it could happen that again on both machines writers want to connect to the other machine each. So they would claim the database semaphore and try to connect to the remote acceptor. But none of the acceptors can get a lock on their database. So we would have produced a deadlock.

To make a long story short, I tried innumerable solution but they were all not suitable. They either produced a deadlock or multiple socket connections. The reason was what I called the "symmetry trap". As the same algorithm would have to be run on both machines, and it could always be possible that on both machines happens the same (e.g. a writer wants to connect to the other machine) at the same time, there was no way out.

The solution is simple. We just have to make all machines different, and there is no more symmetry trap. So I decided to distinguish between the what the machines were doing in the handshake algorithm according to their IP address and port number. I defined an order so that a machine is defined "bigger" as another one if its IP address is greater, and in case of equal IP addresses if its port is greater than the other one. Then I defined that an acceptor should be high priorised if it runs on a bigger machine and low priorised if it runs on a smaller machine. As for the writers it is the other way round: writers are high priorised if they run on a smaller machine and low priorised if they run on a bigger machine.

The key point now is that only high priorised writers and acceptors are allowed to store sockets in the database. So if a writer connects to the remote acceptor, the first thing to do it to tell it the own machine details. Then both the writer and the remote acceptor compare their IP addresses and port numbers and find out whether they both are high priorised or not. It is unavoidable that either both of them are high priorised or both of them are low priorised.

When they are high priorised, the remote acceptor has to look in its database whether there is already an entry to our machine. This is necessary because the following could have happened: Our writer could have checked our database and not have found a socket to the remote machine there. So it connected to the remote acceptor. But there was another writer on our machine already connected to the remote acceptor for whose completion our writer would have to wait. This is the reason why despite our writer's negative database check there could now be a socket connection between the two machines in the databases.

If this should be the case, our writer and the remote acceptor would close the socket connection and our writer would wait until a socket connection to the remote machine appears in our database, which will definitely be the case sooner or later as there must have been another writer on our machine who already established a socket connection to the remote machine.

If the remote acceptor does not find a socket to our machine in its database, the new socket connection can be saved in both databases. With this solution it is only necessary to lock the database for a very short period.

If our writer and the remote acceptor are low priorised, they would close the socket connection again. Then the remote acceptor would simulate a writer, i.e. it would connect to our acceptor. The rest is the same as explained above. So our (low priorised) writer would only have to wait until the remote machine appears in the database.

# 5. The Performance – Benchmarking Results

In order to get an imagination of the performance of the DoP interface, I wrote three small benchmarking programs. They can be found in the `benchmark.send` and `benchmark.receive` subdirectories of the `/dop/benchmark` directory on the enclosed CD-ROM.

All three programs benchmark the bandwidth [Bytes/s] against the size of the sent packet. The first program uses the DoP and is called `send.with.dop` and `receive.with.dop` respectively. The second program sends over normal socket connections and receives a one byte acknowledgement after each packet. It is called `send.socket.with.ack` and `receive.socket.with.ack` respectively. The third program only flushes the data to the network over raw sockets without receiving any acknowledgement. It is called `send.raw.socket` and `receive.raw.socket` respectively.

I ran all three programs locally on my computer **stue4b3** and on **kalgan** which is a Linux machine at the Computing Laboratory. Then I ran them over the university Ethernet in both directions **stue4b3** → **kalgan** and **kalgan** → **stue4b3**. The results can be found in the file `benchmark.txt` in the `/dop/benchmark/` directory.

Figures 5.1 - 5.3 show the four different locations for each of the three programs. In each of them **kalgan** shows the best performance, especially for high packet sizes. This must be due to some networking reasons, as **kalgan** is a slower machine than **stue4b3** and this is every time the case, independent from the computational workload (which is definitely higher for the DoP as for the socket with acknowledgement or even for the raw socket version.)

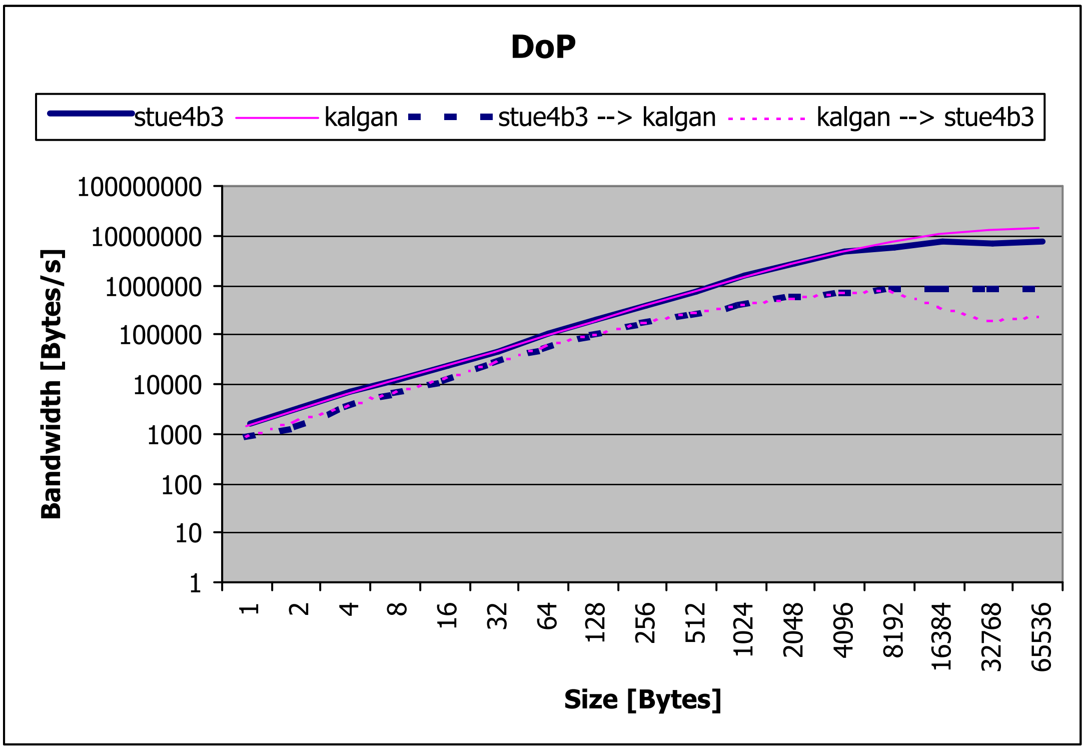

**Figure 5.1**

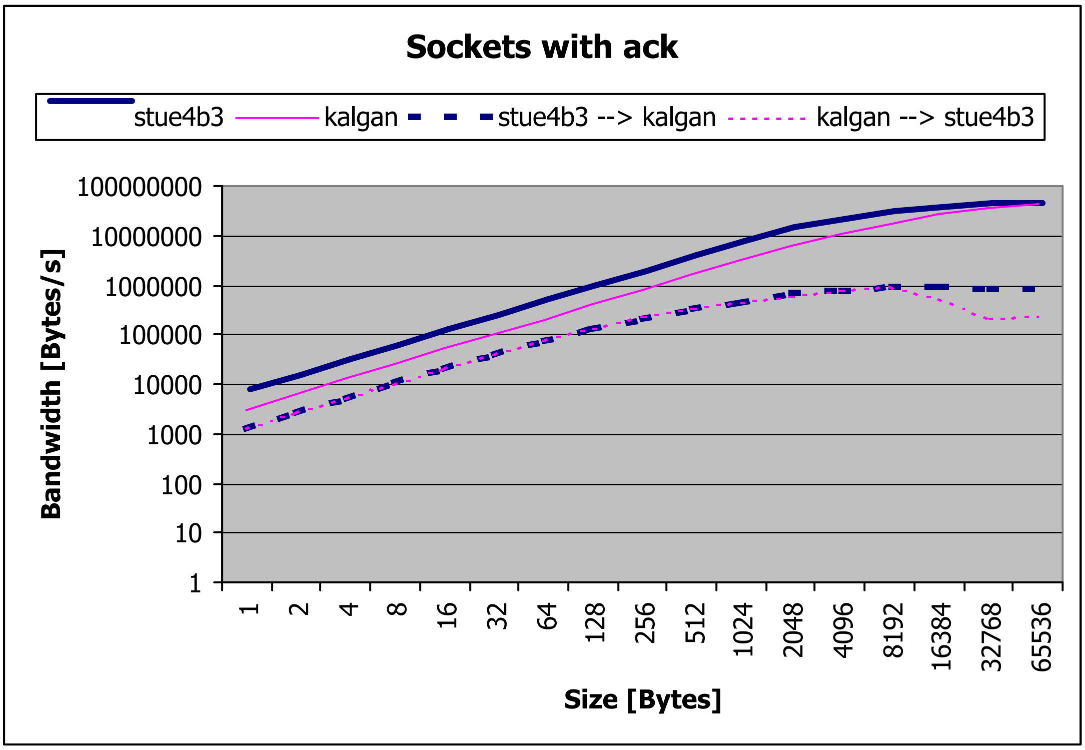

**Figure 5.2**

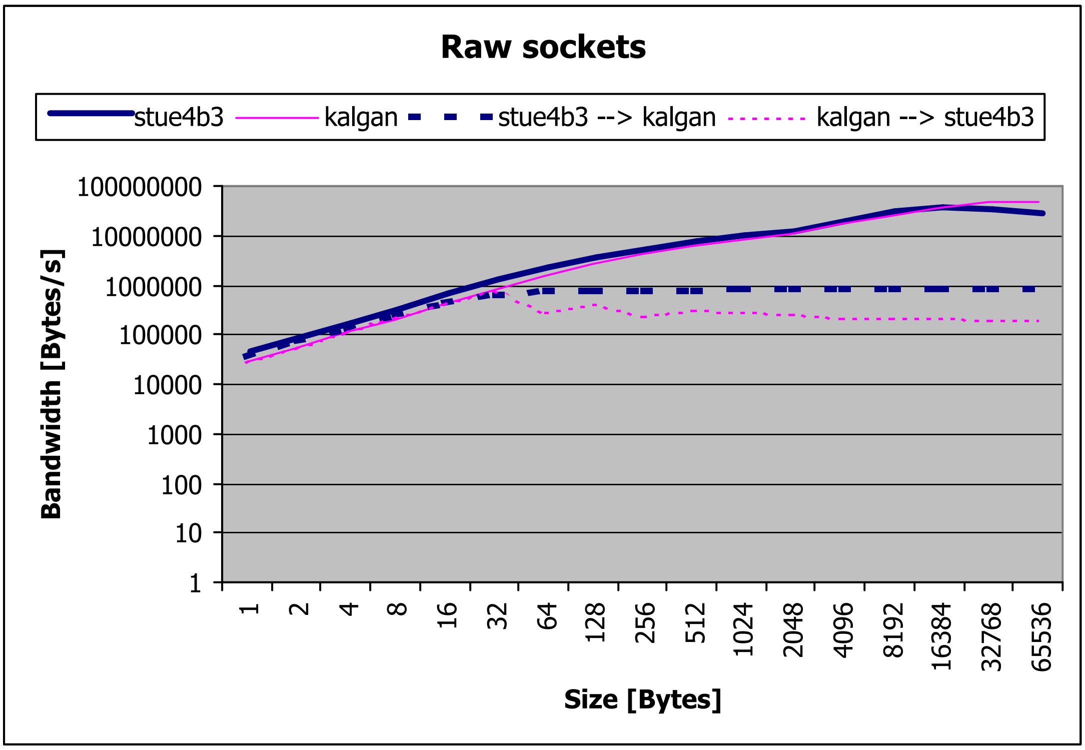

**Figure 5.3**

Figures 5.4 - 5.7 show the four different locations for each of the three programs.

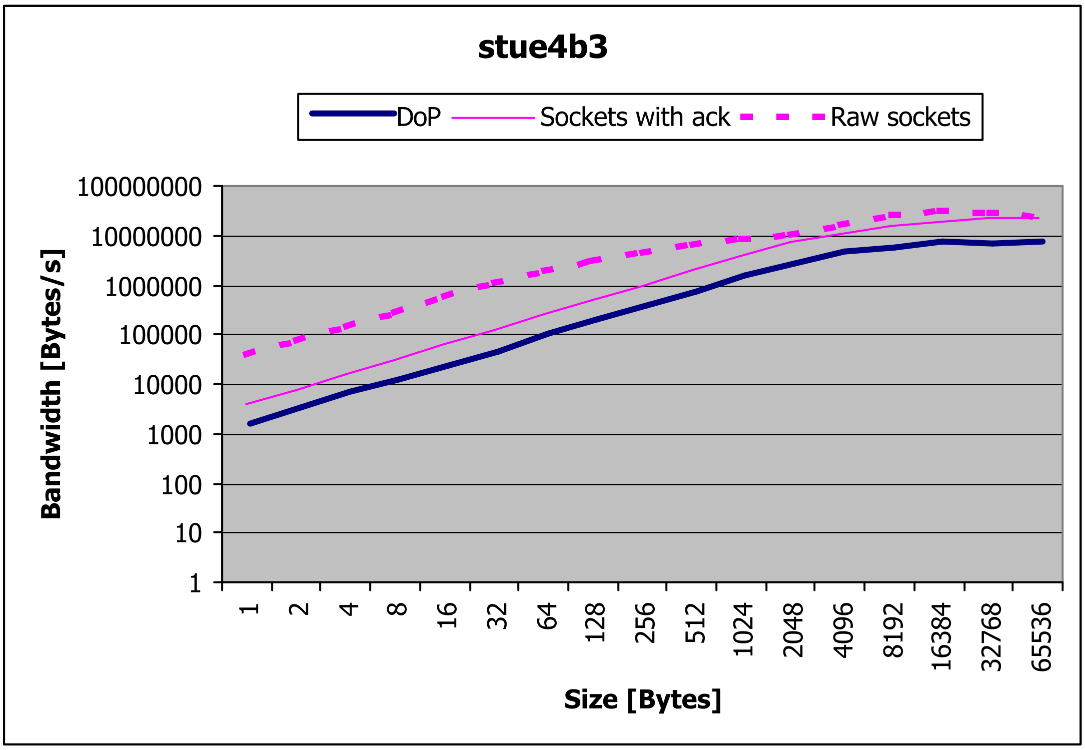

Figure 5.4

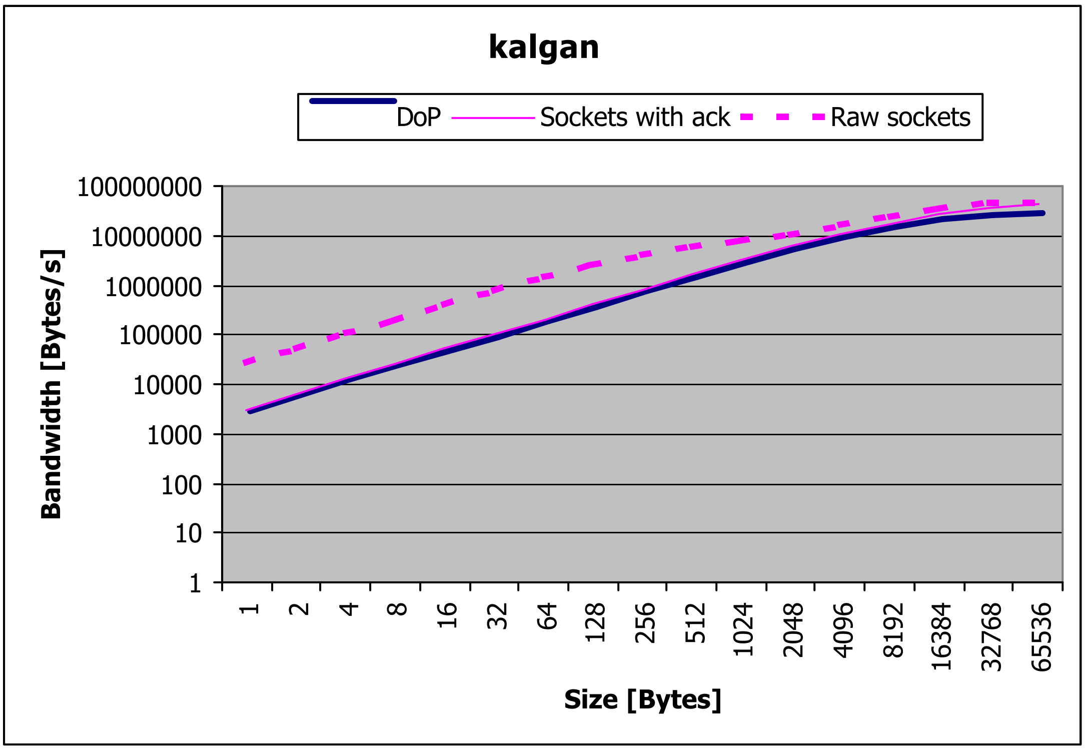

Figure 5.5

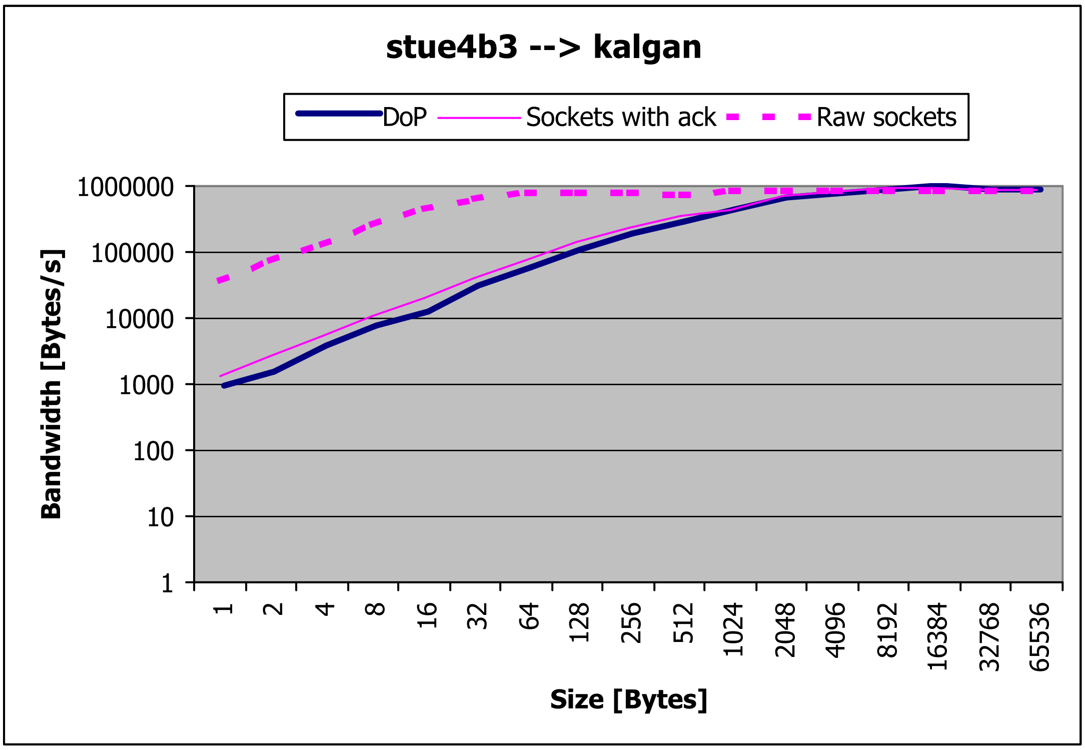

Figure 5.6

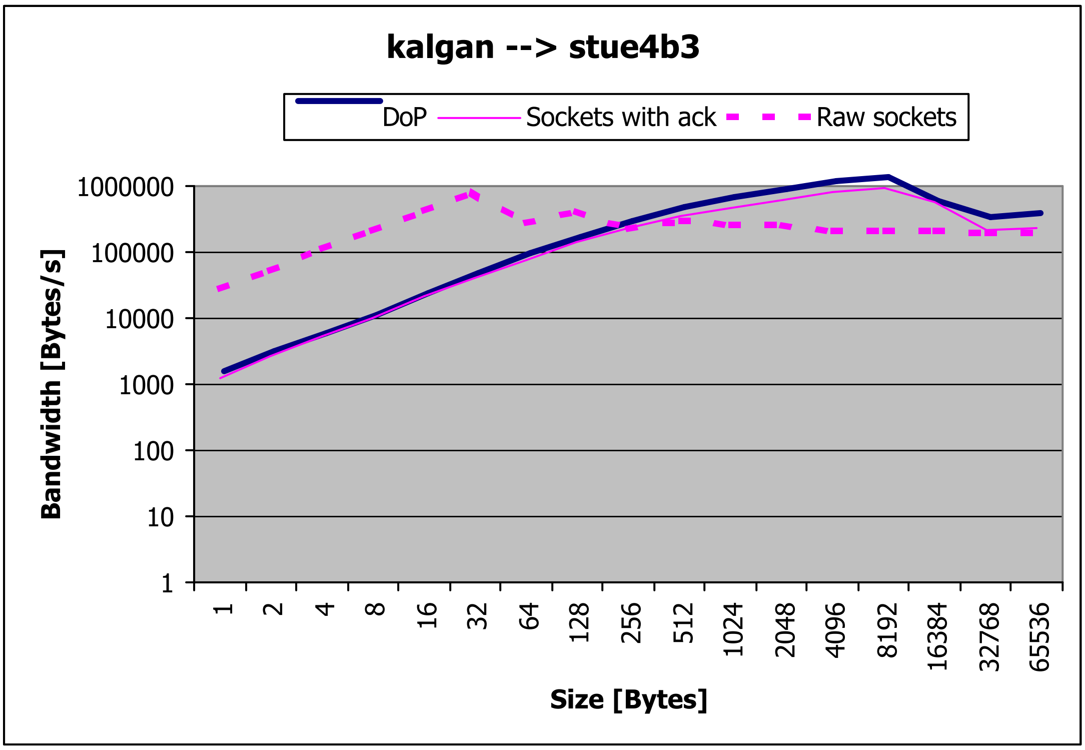

Figure 5.7

In all versions, except the **kalgan → stue4b3** version, the raw sockets are fastest, as one could expect.

The two local versions are, in principle, equal. The DoP is slower than the socket with acknowledgement version, but the factor is more or less constant around three. The raw socket version is fastest, but at a packet size of 64K the raw socket both non DoP versions reach about the same bandwidth.

The Ethernet versions show a much smaller difference between the DoP version and the socket with acknowledgement version. This is due to the fact that the socket operations are much slower so that the software overhead of the DoP is not that significant.

# 6. A Summary. And Which Steps Could Follow

This project has as a result an interface for distributed occam channels which is easy to use. After a rather slow start, I managed to get all the programming work done. It was also no problem to extend the objectives of the project and to add the support for Connections and anonymous channels.

During this project I learned a lot about occam and the way it works. I also could strengthen my knowledge about several networking topics, in particular about the usage of TCP sockets, and about several aspects of concurrent programming in general, especially about semaphores and the avoidance of deadlocks.

Most of the time the work went quite smoothly, although there were also some problems which had to be solved and mistakes that had to be corrected. For example did I first use two instead of one socket connection between two machines. This was mainly due to the fact that I had seen it that way in the occam NetChans code and therefore I thought that one could not read and write to the same socket in parallel. Nevertheless, it was not a big deal to change that into a single socket after my supervisor had explained me how it really worked.

The greatest problems did I have with the handshake algorithm. It took me quite a long time until I could sort that out, and if my mother would not have given me the tip to distinguish between the machines by their IP address, I would have struggled with it even longer.

Although the DoP has reached quite a good standard, there are of course lots of things which could be investigated in the future. One aspect that could be amended is that there could be the possibility added to terminate one interface without crashing the whole network.

Another feature which was added to JCSP.net were streaming channels. Streaming channels are used to send streams of data over a network channel without receiving acknowledgements after each single data item but only maybe after a larger amount of data. This could be particularly useful for multimedia applications and could be implemented in the DoP as well.

A research area will certainly be the extension of the DoP to support the dynamic distribution of frozen occam processes which are currently being dealt with by Fred Barnes.

All in all I can say that though it was really a lot of work, the project was fun, especially when I could see that the programs which I had written did really work! ☺

# BIBLIOGRAPHY

[1] C.A.R. Hoare, Communicating Sequential Processes, Prentice Hall, London, 1985

[2] A. Carling, Parallel Processing - The Transputer And occam, Sigma Press, Wilmslow, 1988

[3] J. Kerridge, occam Programming: A Practical Approach, Blackwell Scientific Publications, Oxford, 1987

[4] P.H. Welch, occam2 & Transputer Engineering, 1991 in CO516: Parallel And Imperative Programming, Lecture Notes, Computing Laboratory, University of Kent at Canterbury, 2000

[5] P.H. Welch, An occam2 Checklist in CO516: Parallel And Imperative Programming, Lecture Notes, Computing Laboratory, University of Kent at Canterbury, 2000

[6] INMOS Ltd., occam® 2 Reference Manual, Prentice Hall, Hertfordshire, 1988

[7] G. Jones and M. Goldsmith, Programming In occam® 2, Prentice Hall, Hertfordshire, 1988

[8] D. Pountain and D. May, A Tutorial Introduction To occam Programming, BSP Professional Books, Oxford, 1987

[9] P.H. Welch, An occam Approach To Transputer Engineering, 1988 in CO516: Parallel And Imperative Programming, Lecture Notes, Computing Laboratory, University of Kent at Canterbury, 2000

[10] P.H. Welch, GOTO (Considered Harmful)$^n$, n Is Odd, 1988, in CO516: Parallel And Imperative Programming, Lecture Notes, Computing Laboratory, University of Kent at Canterbury, 2000

[11] D.C. Wood and P.H. Welch, KRoC – The Kent Retargetable occam Compiler, Computing Laboratory, University of Kent at Canterbury, 1996

[12] P.H. Welch, Essentially KRoC, Computing Laboratory, University of Kent at Canterbury, 2000

[13] F.R.M. Barnes, Blocking System Calls in KroC/Linux, 2000 in Communication Process Architectures 2000 – WoTUG-23 Conference Proceedings, Computing Laboratory, University of Kent at Canterbury, 2000

[14] F.R.M. Barnes, The occam Socket Library, Computing Laboratory, University of Kent at Canterbury, 2000

[15] P.H. Welch and D.C. Wood, Higher Levels of Process Synchronisation, Computing Laboratory, University of Kent at Canterbury, 1997

[16] P.H. Welch and D.C. Wood, SEMAPHOREs, RESOURCEs, EVENTs and BUCKETs, Computing Laboratory, University of Kent at Canterbury, 1997

[17] I.N. Goodacre, occam NetChans, Project Report, Computing Laboratory, University of Kent at Canterbury, 2001

[18] C. Lahure and J.G. Foster, JCSP.net, Project Report, Computing Laboratory, University of Kent at Canterbury, 2001

[19] F.R.M. Barnes, Dynamic occam processes, Fringe talk during the CPA 2000 Conference, University of Kent at Canterbury, 2000

[20] A.S. Tanenbaum, Computer Networks, Prentice Hall, Upper Saddle River, 1996

# APPENDICES

## A. THE ORIGINAL PROJECT PLAN

21/03/2001 Mario Schweigler Project Plan

### Project Title: occam and Distribution

occam is a language which is based on the CSP calculus. The general idea behind CSP and occam is to provide a formal method respectively a language to enable the users to model concurrent systems. occam's parallelity features were a design issue. So they were not added to an already existing language; they are the general principle behind the language.

occam programs are built upon sequential processes that communicate with each other. The only way for processes to communicate with each other is over well-defined channels. So an occam process can be thought as a black box with input and output channels which are the interface to the outside world.

This project is concerned with the distribution of occam processes in a network. The occam paradigms, especially the channel paradigm, form a good basis for distribution. Several issues of distribution will be addressed during this project.

### What has been done so far

The project will build upon a range of research work which has already been done. The most important foundation is the new TCP Socket library that has been created recently. It has been created for the Linux version of the compiler; therefore the further development will be under Linux.

In his research paper “Blocking System Calls in KroC/Linux”, Fred Barnes describes the possibility of allowing occam processes to make a blocking system call without blocking the processes that are running in parallel with it. From this basis he created a new library which allows to utilise TCP sockets from occam processes.

The Socket library provides a number of processes to create TCP sockets, open them, read from them and write to them, close them, etc. The data format which can be sent over a socket is arrays of bytes. This library will be an important basis for the project.

Another research work this project will build on is the final year project of the BSc student Ian Goodacre. His work is primarily concerned with the construction of an interface from the occam world to the network. It consists of an interface for sending data, an interface for receiving data and a buffer on the receiving side.

These interfaces provide a certain level of transparency, as the users are no longer required to perform the socket operations themselves nor to cope with IP addresses or socket numbers. But there are still restrictions and a number of requirements have to be met in order to use this interfaces.

Another research work that has been done is Fred Barnes’ method to freeze occam processes and their state, saving them and restoring them in order to continue to run them. There is, however, no formal research paper about this but only a fringe talk during the Communicating Process Architectures 2000 conference.

## Objectives of the project

One target of the project is to provide a new level of transparency to the existing work.

The aim is to provide occam programmers with a transparency that in fact there is no more difference for them whether their processes which want to communicate, are running on the same machine or on different machines. The currently existing interfaces have a number of restrictions. This project will try to overcome them.

The Socket library only accepts arrays of bytes to be sent over a TCP socket. It will be an aim of this project to provide the possibility to sent also other data types, including protocols, by automatic retyping.

Ian Goodacre's interface provides the following processes: a process to input bytes coming from the network, a process to output bytes to the network and a process acting as an input buffer. Both input and output processes have as a parameter an array of channels of bytes which are used to input respectively output the data which has to be sent over the socket connection and an array of Booleans for sending confirmations. Different sending and receiving processes can connect to these channels; the channels are identified by their index within the array.

The handshake paradigm is implemented as follows: On the receiving side an input buffer is set between the receiving process and the input process. The data coming from the network is passed to the buffer by the input process. When the receiving process tries to read from the buffer it is suspended until there arrives something from the network. This means that the receiving process can be written the same way as if it would be for the use of local channels. The buffer sends a confirmation back to the input process after the receiving process has read the data. Only after the receiving process has read the data from the buffer, the buffer accepts new data from the input process.

On the sending side the sending process writes the data to the output process. After having passed over the network the input process sends the confirmation, which it gets from the buffer, back to the output process, which passes it to the sending process. This means that the sending process cannot be written as it would be the case for local channels. The sending process has to await the confirmation before it can continue.

Today it is possible to connect two processes which are running on the same machine over a channel by calling them (from another process, usually the main program) and giving them as parameter the same channel, which has been defined before. One process uses this channel as input channel and the other uses it as output channel.

The project's target is to provide the same technology over the network, i.e. to map occam channels on TCP socket connections. The aim is that in the end there is no more difference between local channels and channels using TCP sockets. The ideal solution would be that a socket channel could be created and identified by a name as today local channels are. On one machine this channel would be passed to a process as an input channel and on another machine the same channel would be passed to another process as an output channel. The communication would work by handshake as in the local occam world. There would be no additional work necessary to be done for the programmer.

The project will have to explore how far it can reach by creating occam code and to which extent the compiler will have to be adapted.

Another target of the project can be to distribute the process dynamics created by Fred Barnes. The aim could be to create the possibility to freeze processes and their state, pass the whole over the network to another machine, defrost it there and continue to run them.

## Planned phases of the project

- Background reading
  There will be need for me to read a good amount of background material (see bibliography).

- Implementation and documentation of the TCP socket channels
  It will be explored how to implement transparent TCP socket channels. The implementation will be done and documented.

- Conditional: Dynamic distribution of occam processes
  If there is enough time left, this area can be explored. Again, all implementations will be documented.

- Building a demonstration application
  In order to demonstrate the newly implemented capabilities, one ore more sample applications will be written and their behaviour will be documented.

- Final phase: Writing up
  The final phase will be concerned with bringing all the documentation that has been done during the project into a coherent form that meets the requirements for a dissertation.

## B. CHANGES TO THE ORIGINAL PLAN

All in all I think I can say that the project covered to a great extend what was purposed in the original plan. The result is an interface which is easy to use and, apart from having to wait for an acknowledgement when writing to a network channel, occam programmers can write their processes for distributed networks the same way as they would write them for local programs. Furthermore, Connections and anonymous channels are powerful extensions for client/server style applications. However, there were some minor changes to the original project plan.

The first change is quite obvious: it is the project title. At the time when the original plan was set out, I did not know yet that I would call my interface "Distributed occam Protocol". At that time my ideas about what I would do in the project were quite vague. This is the reason for the rather general original title.

A more substantial change was the fact that the project did not cope with the dynamic distribution of frozen processes. The main reason for that was the lack of time. Firstly, I agreed with my supervisor to concentrate on the aim to reach a similar level with the DoP as there had been achieved with the JCSP.net project before. Two important aspects of that were the implementation of Connections and anonymous channels which were not part of the original project plan.

And secondly, the occam code which was developed in the occam NetChans project was so far from the objectives of this project that it was hard for me to build upon that. In fact, I decided to build my interface from the scratch because there was not really code which I could reuse.

The project did also not cope with possible changes to the compiler or the occam kernel as this would have beaten the time constraints. It rather concentrated, agreed with my supervisor, on the development of a library which could possibly later be part of a KRoC release.

## C. The DoP Itself

On the enclosed CD-ROM there are the following directories and files (directories are printed in **`bold`**).

| | |
|---|---|
| **`dop`** | Main DoP directory |
| `dop.occ` | Source file for the DoP library |
| `doplib.inc` | Configuration include file |
| `dop.def.inc` | Definition include file |
| `dop.aux.inc` | Include file for auxiliary processes |
| `dop.make` | Shell script to compile all sources (change mode before executing!) |
| **`dop.cns`** | Channel Name Server directory |
| `dop.cns.occ` | CNS source file |
| `dop.cns.config.inc` | CNS configuration include file |
| `dop.cns.config.export.inc` | Exportable CNS configuration include file |
| **`test.normal.channels`** | Contains testing programs for normal Any to One channels |
| **`machine.1`** | Program for test machine 1 |
| `machine.1.occ` | Source file for test machine 1 |
| `dop.local.config.inc` | Local configuration include file |
| **`machine.2`** | Program for test machine 2 |
| `machine.2.occ` | Source file for test machine 2 |
| `dop.local.config.inc` | Local configuration include file |
| **`machine.3`** | Program for test machine 3 |
| `machine.3.occ` | Source file for test machine 3 |
| `dop.local.config.inc` | Local configuration include file |
| **`test.connections.and.anonymous.channels`** | Contains testing programs for Connections and for anonymous channels |
| **`server.machine`** | Program for the test server |
| `server.machine.occ` | Source file for the test server |
| `dop.local.config.inc` | Local configuration include file |
| **`client.machine.1`** | Program for test client 1 |

| | |
|---|---|
| `client.machine.occ` | Source file for test client 1 |
| `dop.local.config.inc` | Local configuration include file |
| **`client.machine.2`** | Program for test client 2 |
| `client.machine.occ` | Source file for test client 2 |
| `dop.local.config.inc` | Local configuration include file |
| **`client.machine.3`** | Program for test client 3 |
| `client.machine.occ` | Source file for test client 3 |
| `dop.local.config.inc` | Local configuration include file |
| **`benchmark`** | Contains benchmarking programs |
| `benchmark.txt` | The results of the benchmarking |
| **`benchmark.send`** | Contains benchmarking senders |
| `send.with.dop.occ` | Sender which uses the DoP |
| `dop.local.config.inc` | Local configuration include file |
| `send.socket.with.ack.occ` | Sender which uses sockets with acknowledgement |
| `send.raw.socket.occ` | Sender which uses raw sockets |
| **`benchmark.receive`** | Contains benchmarking receivers |
| `receive.with.dop.occ` | Receiver which uses the DoP |
| `dop.local.config.inc` | Local configuration include file |
| `receive.socket.with.ack.occ` | Receiver which uses sockets with acknowledgement |
| `receive.raw.socket.occ` | Receiver which uses raw sockets |
| **`dissertation`** | Contains this dissertation... |
| `dissertation.ps` | ... in Postscript format |
| `dissertation.pdf` | ... in Portable Document Format |
| **`api.documentation`** | Contains the HTML API documentation |
| `api.html` | HTML API documentation |
| `datatypes.html` | HTML documentation about data types |

The reader should copy the **dop** directory, including all files and subdirectories, to the hard drive and set the mode of the `dop.make` file to executable. Then they should execute the `dop.make` script. This will compile the library, copy the library and all necessary include files to the appropriate subdirectories and compile all programs.

The library (`dop.lib`, `libdop.a`) and the include file (the configuration file) (`doplib.inc`) might be placed into a directory which is part of the path which is defined in the `OCSEARCH` environmental variable. So when a new program is developed which uses the DoP, it could be avoided to copy these files to the new program's directory.

Please note that the values of the local configuration file (`dop.local.config.inc`) have to be adjusted individually for each new program which uses the DoP. The values of the exportable CNS configuration file (`dop.cns.config.export.inc`) have to be adjusted according to the location of the CNS and this file has to be given to programmers who write a program which wants to use the CNS at that location. This is the reason why these files usually have to be in the program directory for each program individually and therefore can hardly be put in a shared directory.

The HTML API documentation was created by the JoccDoc utility which had been developed during the occam NetChans project. However, I had to post-edit the produced HTML files because I was not fully saisfied with the result.

## D. GLOSSARY

**ACKNOWLEDGEMENT.** A writer has to wait for an acknowledgement that the reader has read the data the writer has sent. This is necessary to get CSP semantics.

**ANONYMOUS CHANNEL.** A network channel which is not connected over the Channel Name Server but anonymously after the exchange of a reader's channel location.

**ANONYMOUS READER.** The reading end point of an anonymous channel. Its channel location is sent to an anonymous writer which then connects to it.

**ANONYMOUS WRITER.** The writing end point of an anonymous channel. It connects to an anonymous reader after having received its channel location.

**ANY TO ONE CHANNEL.** See "Normal Any to One channel".

**CHANNEL LOCATION.** A record to identify the location of a channel. Consists of the IP address, the port number and the channel index.

**CHANNEL NAME SERVER.** A mediator between a writer and a reader or between a Connection server and a Connection client. Used to connect the end points of a network channel. Stores channel locations under a name and returns them on request.

**CLIENT.** See "Connection client".

**CLOSE.** An instruction to close a Connection between a client and a server.

**CNS.** See "Channel Name Server".

**CNS CONFIGURATION FILE.** The file `dop.cns.config.inc`. It contains options of the Channel Name Server program.

**CONFIGURATION FILE.** The file `doplib.inc`. It contains several options of the DoP interface. Programs which want to use this interface have to include this file.

**CONNECTION.** A special network channel for client/server style two way communication.

**CONNECTION CLIENT.** One end point of a Connection. Several clients can use the same server.

**CONNECTION SERVER.** One end point of a Connection. Several clients can use the same server.

**CONVERSION PROCESS.** An process which is plugged between the DoP interface and a user process and which converts network data.

**DOP.** The Distributed occam Protocol. The title of this project and the name of a protocol whose main part is a library which enables the mapping of occam channels on TCP socket connections.

**END POINT.** See "Network channel".

**EXPORTABLE CNS CONFIGURATION FILE.** The file `dop.cns.config.export.inc`. It contains the location of the machine the Channel Name Server is running on. This file should be given to programs which want to use the CNS. The values should be passed to the DoP interface as parameters.

**INTERFACE.** The DoP interface is connected to the end points of a network channel and emulates its behaviour over TCP socket connections.

The **LIBRARY.** The library to the DoP consists of the files `dop.lib` and `libdop.a`. It offers the DoP interface as well as several auxiliary processes.

**LOCAL CONFIGURATION FILE.** The file `dop.local.config.inc`. It contains the location of the machine a DoP interface is running on. These values should be passed to the DoP interface as parameters.

**NETWORK CHANNEL.** An emulation of an occam channel where the processes who are connected to it are distributed on different machines in the internet. These processes are called end points.

**NORMAL ANY TO ONE CHANNEL.** A network channel which implements Any to One semantics. This means that there can be several writers but only one reader.

**OPEN.** An instruction to open a Connection between a client and a server.

**READER.** An end point of a network channel which reads from it.

**RECONNECTION.** Connecting an anonymous writer to an anonymous reader after having exchanged its channel location.

**REGISTRATION.** The end points of a network channel have to tell the DoP interface of which type they are. This process is called registration.

**REQUEST.** An instruction to perform a writing operation from a Connection client to the Connection server.

**RESPONSE.** An instruction to perform a writing operation from the Connection server to a Connection client.

**RETYPING.** Converting the data type of network data. Done by conversion processes.

**SERVER.** See "Connection server".

**WRITER.** An end point of a network channel which writes to it.