Communicating Process Architectures – 2003
Jan F. Broenink and Gerald H. Hilderink (Eds.)
IOS Press, 2003

Flexible, Transparent and Dynamic occam Networking With KRoC.net

Mario SCHWEIGLER
ms44@kent.ac.uk / research@informatico.de

Fred BARNES
frmb2@kent.ac.uk / fred@frmb.org

Peter WELCH
P.H.Welch@kent.ac.uk

Computing Laboratory, University of Kent
Canterbury, Kent, CT2 7NF, UK

Abstract. KRoC.net is an extension to KRoC supporting the distribution of occam channels over networks, including the internet. Starting in 2001, the development of KRoC.net has gone through a number of stages, each one making the system more flexible, transparent and dynamic. It now enables the occam programmer to set up and close network channels dynamically. Configuration has been simplified. All occam PROTOCOLs can now be sent over network channels, without need for conversion. Many of the new dynamic features in occam have been used to improve KRoC.net. Many of the concepts in KRoC.net are similar to those in the JCSP Network Edition (JCSP.net), KRoC.net's counterpart in the JCSP world. This paper will give an overview over KRoC.net, its usage, its design and implementation, and its future. It will also provide some benchmarks and discuss how the new occam features are being used in the latest KRoC.net version.

1 Introduction and Motivation

Originally, occam[1] [1] was targeted at multi-processor (transputer) platforms for some specific embedded system application. That platform was finite and fixed at *configuration time*. There have been projects to target multi-processor platforms other than transputers. An example is Vella's work on porting occam to Networks of Workstations (NoWs) [2, 3] which uses an interface to communicate to the network, called ocnet. This interface is a layer between the occam kernel and the operating system and the networking hardware. Like the original networks of transputers, this system was static and the layout of the network had to be configured in advance. Another example is MESH [4] which is completely hardware dependent. MESH is a messaging and scheduling system that enables accessing network hardware (e.g. ethernet cards) and provides user-level scheduling in Linux. The original version of CCSP [5], on which the current occam kernel is based, could access the messaging capabilities of MESH. But again, this was hardware dependent and static.

We want our new [6, 7, 8] occam also to target multi-processor platforms, but we want to be abe to use it for an open-ended network of machines, connected by network channels, whose size and topology are constructed dynamically at *runtime* and may be changing. One important aim is to enable occam applications to use the infrastructure of the largest network

[1] occam is a trademark of ST Microelectronics.

available — the internet. The internet plays an increasingly important role in today's society, and we want occam programs to be able to utilise this global network by simple and (in terms of CSP) safe channel communication. This requires an approach that is completely different to the transputer-style configuration and other static solutions.

This is where KRoC.net comes into play. KRoC.net is an extension to KRoC [9] which provides a framework enabling the occam programmer to set up channels between processes that are running on different machines. KRoC.net has been improved significantly since its first versions [10, 11]. In this process, many of the new features in KRoC were utilised, in particular the new Generic Protocol Converters described in section 3.

KRoC.net has now become a very dynamic tool which enables the programmer to set up network channels at runtime just when they are needed. The configuration of KRoC.net has been simplified, and a new URL-based channel setup mechanism has been introduced, allowing greater flexibility in the creation of network channels. KRoC.net's network channels have reached a high level of transparency now. They can now be plugged into any old occam process without the need to change its code. All occam `PROTOCOL`s are now supported for network channels, including components that are `MOBILE` [6].

KRoC.net is the KRoC/occam counterpart of the *JCSP Network Edition* (JCSP.net) [12, 13], the network extension of JCSP (*CSP for Java*) [14]. Many of the concepts in KRoC.net and JCSP.net are similar. Development of JCSP.net has recently been taken over by Quickstone Technologies. Some of the general mechanisms in KRoC.net are similar to the *Virtual Channel Processor* (VCP) of the T9000 transputer [15].

Sections 2 through 4 discuss the infrastructure and implementation of KRoC.net, along with the new paradigm of network channel-types and the new Generic Protocol Converters. Section 5 describes in detail how network channel-types are being set up by the occam programmer, and will give an outlook to future adaptions in the occam language which will make this setup process easier for the programmer. Section 6 describes the configuration of KRoC.net. The performance of KRoC.net and some benchmarks are discussed in section 7. Section 8 summarises the paper and gives some ideas about future work on the KRoC.net project.

2 Network Channel-types

The original idea behind KRoC.net was to distribute normal occam channels over networks, i.e. implement their behaviour, so that from the point of view of the processes which are connected by the channels, their behaviour would be identical, no matter whether the channels were local or networked.

In classical occam, there was no such notion as a channel-end, the paradigm used was always the whole channel. For a distributed system, however, it is natural for each participating processor to set up its end of a network channel separately — so we need this concept of channel-ends. Another newly introduced dynamic feature of occam greatly extends and empowers KRoC.net's model of distributed communication. We refer to *channel-types* [7, 8], which are bundles of channels, dynamically created and mobile. These also have well-defined ends, called *client-end* and *server-end* for convenience.

2.1 'Classical' Local Channel-types

To remind people about how channel-types work, here is a short example. A channel-type would be declared like this:

```
CHAN TYPE THING
  MOBILE RECORD
    CHAN INT req?:                          -- request channel
    CHAN MOBILE []BYTE reply!:              -- reply channel
:
```

THING is a channel-type which contains two channels, one request channel and one reply channel. The direction specifiers ('?' and '!') are from the point of view of the server-end of the channel-type, i.e. it would read from req and write to reply.

To create an instance of a channel-type, we have to declare two variables, one for the server-end and one for the client-end, and allocate them as in pairs:

```
THING? thing.svr:                           -- declare server-end
THING! thing.cli:                           -- declare client-end
SEQ
  thing.svr, thing.cli := MOBILE THING      -- allocation
  ...  use them
```

The server-end of a channel-type is marked by a '?', the client-end by a '!'. To use the channel-type, one would typically pass the ends of the channel-type to different processes who would then communicate over the channels inside the channel-type:

```
PROC server(THING? thing.svr)
  WHILE TRUE
    INT size:
    MOBILE []BYTE buffer:
    SEQ
      thing.svr[req] ? size                 -- get size
      buffer := MOBILE [size]BYTE           -- allocate buffer
      ...  fill buffer with data
      thing.svr[reply] ! buffer             -- send buffer back
:

PROC client(THING! thing.cli)
  WHILE TRUE
    INT size:
    MOBILE []BYTE buffer:
    SEQ
      ...  set size
      thing.cli[req] ! size                 -- send size wanted
      thing.cli[reply] ? buffer             -- get buffer
      ...  use buffer
:

THING? thing.svr:
THING! thing.cli:
SEQ
  thing.svr, thing.cli := MOBILE THING
  PAR
    server(thing.svr)                       -- pass server-end to server
    client(thing.cli)                       -- pass client-end to client
```

Alternatively, channel-type ends can be sent over channels:

```
PROC generator(CHAN THING? svr.out!, CHAN THING! cli.out!)
  THING? thing.svr:
  THING! thing.cli:
  SEQ
    thing.svr, thing.cli := MOBILE THING
    svr.out ! thing.svr                    -- send server-end
    cli.out ! thing.cli                    -- send client-end
:

PROC server(CHAN THING? svr.in?)
  THING? thing.svr:
  SEQ
    svr.in ? thing.svr                     -- get server-end
    ...  use thing.svr
:

PROC client(CHAN THING! cli.in?)
  THING! thing.cli:
  SEQ
    cli.in ? thing.cli                     -- get client-end
    ...  use thing.cli
:

CHAN THING? svr.chan:
CHAN THING! cli.chan:
PAR
  generator(svr.chan!, cli.chan!)
  server(svr.chan?)
  client(cli.chan?)
```

A general remark to the terms *"client-end"* and *"server-end"*: for channel-types, these are just names in order to be able to distinguish between the two ends of the channel-type. They may, of course, be used in a 'genuine' client/server relationship, but this is not a must. Channel-types may well be used for peer-to-peer communication, in which case "client" and "server" would not mean much more than, for instance, "left-hand side" and "right-hand side".

2.2 The New Paradigm of Network Channel-types

KRoC.net implements the behaviour of channel-types in a networking context. This means that internally, KRoC.net only deals with network channel-types (NCTs) that contain network channels. Using this construct, we are able to implement networked versions of both channel-types *and* classical occam channels (which are simply treated as channel-types containing just one single channel). The occam programmer creates a client-end of a channel-type on one machine, and a server-end of the same channel-type on another machine, and then connects them with KRoC.net's network infrastructure. For the application level processes dealing with the client- and the server-ends of that NCT, their behaviours are the same as if is was a local channel-type. For the occam programmer, it is also possible to connect ordinary occam channels via KRoC.net, but internally, KRoC.net treats those as NCTs containing one single network channel. Details about how NCTs are set up are described in section 5.

JCSP.net's channels are automatically *any-to-one*, which is actually a little odd and will be tidied up in a future release. KRoC.net's channels and channel-types have to be explicitly declared to be `SHARED` at the writing- (or client-) ends if we want that property. As with JCSP.net, KRoC net does not support networked sharing of the reading- (or server-) ends of channels/channel-types — although we are working on it!

The new occam channel-types offer a more flexible way of setting up two-way communication across a network than JCSP.net's basic channels or `Connection` channels. This becomes especially useful for those with `SHARED` client-ends. The networked "client" must `CLAIM` its end before it can use it — just like its non-networked version. For the duration of the `CLAIM`, other clients are locked out and a two-way conversation to the server can be completed without interference.

Currently, both ends of networked channel-types must be `CLAIM`ed — even if neither is `SHARED`. This is for technical reasons associated with secure shut-down or movement. That does not apply, of course, to the use of non-networked channel-type ends. This is one area of non-transparency in KRoC.net that we are still considering how best to resolve. There is an argument to enforce `CLAIM`ing on these channel-types regardless of whether they are `SHARED` (or networked), which would remove the anomaly. That argument concerns building in an extra level of checked security to enforce correct patterns of use of the channel components making up the type — but we leave that to a later paper.

3 New occam Features Used in KRoC.net

Our aim is to make network channels *transparent* — i.e. make their behaviour indistinguishable from that of normal channels. Two important requirements have to be met in order to achieve this aim: Firstly, network channels have to retain CSP channel semantics; and secondly, they must be able to carry any `PROTOCOL` — from plain data types to user-defined variant protocols.

3.1 The Extended Rendezvous

The first requirement can be met by utilising the extended rendezvous [6], that enables channel synchronisation to be 'extended' across multiple communicating processes. It operates in such a way that if a simple 'tap' process is inserted between two previously directly connected processes, those two processes will be unable to identify its presence in the network (or lack of). For example, an `INT` 'tap' process is simply:

```
PROC tap(CHAN INT in?, report!, out!)
  WHILE TRUE
    INT i:
    in ?? i                     -- extended input
      out ! i                   -- extended process
      report ! i                -- following process (assume always taken)
:
```

The extended process is executed whilst the outputting process (connected to `in`) remains blocked. This can be any process — or processes — provided that they do not attempt to communicate on `in` (which would immediately deadlock). The 'following' process is optional. If present, it is executed *after* the communication on `in` has completed (and the outputting process resumed). This 'following' process is provided mainly for use in `ALT`s and variant ('`CASE`') inputs, where expressing the same behaviour without it can be tricky.

Note that neither of the processes connected to `in?` and `out!` need any modification. The only changes are in the network setup code (the addition of an extra channel and the `tap` process). Processes that extend communication, such as `tap`, can be connected together in a pipeline, extending the communication through multiple processes.

For KRoC.net, the extended rendezvous provides the mechanism that allows communication through multiple processes (the networking infrastructure), without affecting the

end-to-end synchronisation of the two communicating processes. Between two remote processes, each communication is extended whilst the networking infrastructure performs the communication.

Specifically, the OCP (Output Control Process, see section 4) performs an extended input from the local application (via DECODE.CHANNEL, explained in the following section), communicating the data and waiting for the acknowledgement inside the extended process. The acknowledgement will only be sent by the remote ICP (Input Control Process) when it has communicated successfully with the application.

3.2 Generic Protocol Converters

The second requirement, that of being able to communicate any channel PROTOCOL, is handled by Generic Protocol Converters (GPCs), two compiler built-in PROCs named "DECODE.CHANNEL" and "ENCODE.CHANNEL". Essentially, these provide a pair of processes that extend communication, of any PROTOCOL, between two processes either side, but use a well-known protocol for the channel between themselves.

The PROTOCOL used for communication between DECODE.CHANNEL and ENCODE.CHANNEL is a sequential protocol carrying two INTs. This protocol must be defined by the application. For example:

```
PROTOCOL LINK IS INT; INT:
```

The information carried by this protocol is pointer and size pairs. When the application outputs into a channel connected to DECODE.CHANNEL, the extended-process within communicates the address and size (of the application data) to its output channel. The address communicated is only valid during an extended process whose input acquired it, or where extra synchronisations are put in place to prevent the communication completing.

The reverse of this operation is implemented by ENCODE.CHANNEL. It receives two INTs on its input channel (using an extended input), which it then communicates to the process connected on its output channel — using the same application PROTOCOL as was used for DECODE.CHANNEL's input channel. The two built-in PROCs do not have fixed parameter types and are prototyped by the (illegal) occam:

```
PROC DECODE.CHANNEL(CHAN * in?, CHAN ** term?, CHAN *** out!)
PROC ENCODE.CHANNEL(CHAN *** in?, CHAN ** term?, CHAN * out!)
```

The PROTOCOL represented by *** is the link protocol — the earlier 'LINK' for example. The term channels may be typed as either INT or BOOL and are used to shut-down the GPCs (which otherwise run indefinitely). The PROTOCOL represented by * is the application protocol. These two PROCs do not check that * is the same on both sides. This is done by the KRoC.net infrastructure which ensures that the PROTOCOL.HASH value on both sides is the same. "PROTOCOL.HASH" is a simple compiler built-in that evaluates to a unique[2] constant for the given type, protocol or name (variables, abbreviations, PROCs and FUNCTIONs), that is passed as a parameter.

[2]The hashing algorithm used for PROTOCOL.HASH generates sufficiently unique values, but there exists a slim probability of the same value being generated for two completely different types/protocols.

3.2.1 Pointer Handling

When the synchronisation is extended correctly, ENCODE.CHANNEL will receive and encode a valid address. However, this is only true for same/shared memory systems. Within the KRoC.net framework, where DECODE.CHANNEL and ENCODE.CHANNEL are expected to be on physically different systems, the data output from DECODE.CHANNEL must be *copied*.

Within a networked system, the data (address/size pair) received by ENCODE.CHANNEL will be for a local memory block — allocated by the networking infrastructure. Thus, unless such data is actually *moved* into the application (by mobile communication), ENCODE.CHANNEL must ensure that the memory block is freed after the communication.

The (dynamic) memory blocks consumed by ENCODE.CHANNEL are expected to be dynamic MOBILE arrays (of BYTEs). To convert between a dynamic mobile array and address/size pairs, two additional compiler built-in PROCs are provided: "DETACH.DYNMOB" and "ATTACH.DYNMOB". These have the simple PROC signatures:

```
PROC DETACH.DYNMOB(MOBILE []BYTE var, RESULT INT addr, size)
PROC ATTACH.DYNMOB(INT addr, size, RESULT MOBILE []BYTE var)
```

These PROCs work in such a way that detaching a dynamic mobile leaves that mobile *undefined*, and attaching a dynamic mobile does the same for the INT variables. Furthermore, ATTACH.DYNMOB sets the address (in addr) to zero after attaching the mobile. These ensure that a mobile variable cannot be accessed after it has been 'detached', and that any address used to 'attach' a mobile is left undefined (and invalid).

Given the operation of ENCODE.CHANNEL, that either releases or moves the dynamic mobiles communicated into it, connecting DECODE.CHANNEL and ENCODE.CHANNEL directly is not possible. The address/size pair communicated by DECODE.CHANNEL is that of the occam data item being communicated, which is not necessarily a dynamic mobile array.

To network an encode/decode pair locally requires the use of an intermediary "link" process. Such a process also serves as a good example of using the dynamic mobile attach/detach operations:

```
PROC link(CHAN LINK in?, out!)
  WHILE TRUE
    INT addr, size:
    in ?? addr; size                      -- extended input
      MOBILE []BYTE tmp, new:
      SEQ
        ATTACH.DYNMOB(addr, size, tmp)    -- access the data as a
                                          -- dynamic mobile array
        new := CLONE tmp                  -- copy into a new array
        DETACH.DYNMOB(tmp, addr, size)    -- detach false dynamic array
        DETACH.DYNMOB(new, addr, size)    -- detach new dynamic array
        out ! addr; size                  -- communicate
:
```

When received by ENCODE.CHANNEL, the dynamic mobile created by the CLONE here is either freed, if the output PROTOCOL is *not* a dynamic mobile, or *moved* if it is.

3.2.2 Mobile Protocol Conversion

The handling of ordinary data types, such as those allocated in workspace or vectorspace, is trivial — communication always *copies*. Thus, when ENCODE.CHANNEL receives an address/size pair (using an extended input), it simply engages in communication directly with

those parameters (typically the OUT Transputer instruction). Once output, it frees the memory block using MRELEASE, returning it to the free-lists for re-use.

PROTOCOLs that are implemented by the compiler as a sequential series of communications remain in that way, with the exception of some counted-array protocols. This applies to standard sequential protocol communication and variant ('CASE') communication. Counted-array protocols whose 'count' type is *not* INT64 are sent as a single communication — since the size is included. The current implementation of the GPCs only supports INT-sized counts, so any INT64 counted arrays must have their count sent separately (even there is no possibility of an application communicating more elements than can be held within an INT).

The implementation of DECODE.CHANNEL and ENCODE.CHANNEL is more complex for mobile types and mobile communication, since the application *moves* data. There are three basic forms of MOBILE types: static mobiles; dynamic mobile arrays; and mobile channel-types (that are also dynamic). Mobile channel-types are not currently supported by the GPCs, discussed in section 3.2.3.

Communication of static mobiles is done using pointer-swapping. When the application outputs into DECODE.CHANNEL, and synchronises with the extended input within, the address is extracted leaving the original pointer intact. The implementation of ENCODE.CHANNEL uses a 'temporary' static mobile, into which data is copied when the address and size are received. This is then communicated with the application using the standard mobile output instruction (MOUT).

Dynamic mobile arrays are treated slightly differently. The communication of a dynamic mobile array (using either MOUT64 or MOUTN) is performed as a single operation — the moving of dimension counts and a pointer. After a process has outputted a dynamic mobile array, the local dimension count is set to zero (such that any attempts to access it after the output result in range-check errors).

The implementation of DECODE.CHANNEL for dynamic mobile arrays outputs either one or two address/size pairs. If there is any more than one unknown dimension, the dimension counts are output first (organised sequentially in the originating process's workspace), followed by the data, both as an address/size pair. For dynamic mobiles with only one dimension (which will be the majority in most cases), only the address/size pair for the data is sent. The single dimension count can be calculated from the size of the communicated data.

Dynamic mobile handling for ENCODE.CHANNEL is particularly simple — if there is more than one unknown dimension, the first address/size pair received is a dynamic mobile that holds the dimension counts for the target mobile. ENCODE.CHANNEL uses a local (dynamic mobile) temporary into which these dimensions are copied, before freeing the communicated memory block. For mobiles with *only* one unknown dimension, the temporary is still used, but has its count field initialised from the size of the communicated data.

The next input performed by ENCODE.CHANNEL is the address/size of the data for the dynamic mobile, that is itself a dynamic mobile. If dimension counts have already been set, the communicated address is simply placed in the temporary mobile's pointer slot and the size checked against the expected size (calculated by multiplying the dimension counts with the base-type size). If the dimension counts are unset, the address is copied into the temporary mobile's pointer slot (as before), and the dimension count is calculated by dividing the size of the data by the base-type size.

This temporary is then communicated into the application, with the pointer specifically not returned to the free-lists.

3.2.3 Limitations

Mobile channel-type protocol conversion is not currently supported, but support for this is planned for the future (see section 8). Semantically, communication of channel-type ends

is the *stretching* of those channel-types over the network. However, some non-trivial logic is required inside the KRoC.net infrastructure to support this. Many of these issues have already been investigated and implemented by Muller and May for the Icarus language [16]. Our approach will follow a similar path.

Support for the proposed mobile process types [17] will likely be similar in nature — requiring interaction with the KRoC.net infrastructure in order to set up links for any contained mobile channel-type ends, and to ensure that the receiving node has the code for these processes.

4 Basic Infrastructure

We want to be able to design a system in the same way regardless of the physical distribution of its processes and channels. The underlying model — channels and channel-types — ought to be the same both locally and networked. Figure 1 shows four processes running on the same machine[3]. Processes A and B are connected by a channel (with A having the channel's writing-end and B having the reading-end). Processes C and D are connected by a channel-type (with C having the client-end and D having the server-end of the channel-type). Figure 2 shows the same system of occam processes running on two different machines. The channel between A and B would now be a network channel, and the channel-type between C and D would now be an NCT containing network channels.

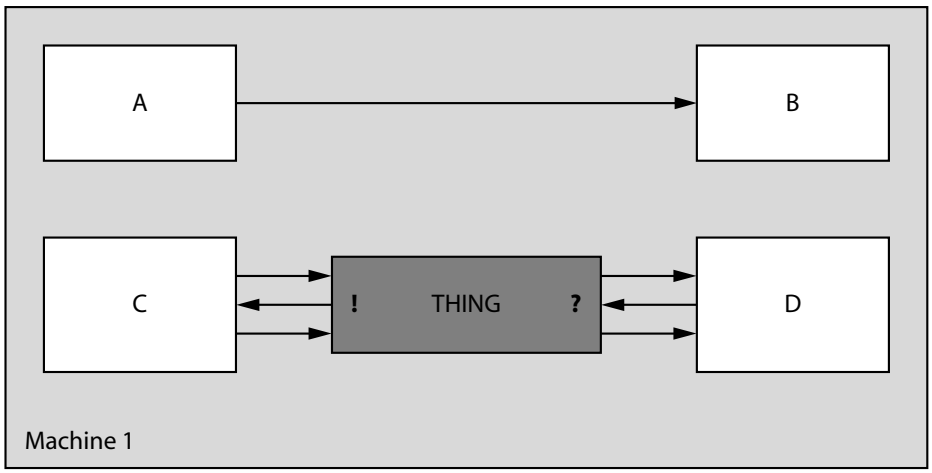

Figure 1: Local Channels and Channel-types

From the point of view of the user level processes (A, B, C and D), there is no semantical difference in the behaviour of the overall system between the two figures. Internally, however, in Figure 2, the user level processes on both machines are running in parallel with an instance of the *KRoC.net manager*, a process provided by the KRoC.net framework. The network channel and the NCT depicted in figure 2 are provided by the KRoC.net manager and multiplexed over a network link between the two machines.

The KRoC.net manager has two main tasks: setting up network channels and NCTs, and (once that is done) administrating them and managing the communication over them. The setup process is described in detail in section 5.

[3]In this paper, we define 'machine' as an occam program (i.e. the whole OS level process), that might run an instance of the KRoC.net manager described in this section. I.e. it is possible to run two occam programs on the same physical computer and to connect them with KRoC.net's infrastructure. These programs would be referred to as separate machines.

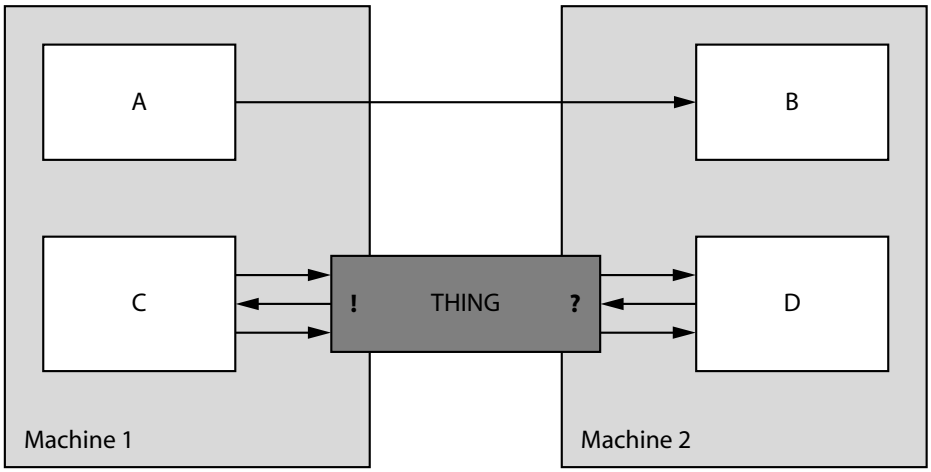

Figure 2: Network Channels and Network Channel-types

4.1 Communication Over Network Channels

As for the communication over a network channel, it makes no difference whether this network channel is a stand-alone one (cf. the one between processes A and B in Figure 2)[4] or inside an NCT (cf. the three network channels inside the NCT between processes C and D in Figure 2).

Figure 3 shows the data flow of the network channel between processes A and B in detail[5]. `DECODE.CHANNEL` and `ENCODE.CHANNEL` are the Generic Protocol Converters described in section 3. Apart from their main purpose — giving us transparency by enabling the occam programmer to set up network channels of any given `PROTOCOL` — the second purpose of the GPCs is to prevent us from unnecessary copying.

Every time process A sends something to `DECODE.CHANNEL`, `DECODE.CHANNEL` outputs the address/size pair of the data item it just received. The KRoC.net manager will then deal with the address and the size only, without copying any data around. This reduces overheads especially for larger data items. The `??` implies that `DECODE.CHANNEL` uses the extended rendezvous (see section 3). This means that process A will be blocked not only until `DECODE.CHANNEL` has read the data item from it, but until `DECODE.CHANNEL` explicitly releases it.

The KRoC.net manager maintains a network link between itself and any other machine to which a network channel has been established. All network communication between the two machines is multiplexed over this link, which saves network resources. Communication over a network link is handled by a pair of Tx/Rx processes on both of the machines that are connected by that link. The Tx process transmits network packets over the link, the Rx process receives network packets from the link.

Every writing-end of a network channel handled by the KRoC.net manager has a unique Output Control Number (OCN), every reading-end has a unique Input Control Number (ICN). These two numbers unambiguously identify each network channel.

Each writing-end of a network channel is handled by an Output Control Process (OCP). The OCP reads the address/size pair from `DECODE.CHANNEL`, also using the extended rendezvous. Each OCP knows over which link its network channel is multiplexed, and it also knows the ICN of the reading-end on the remote machine. The OCP sends the address/size

[4] which KRoC.net internally treats as an NCT containing a single network channel

[5] Channels used for setting up and administrating network channels have been omitted in order to simplify the figure.

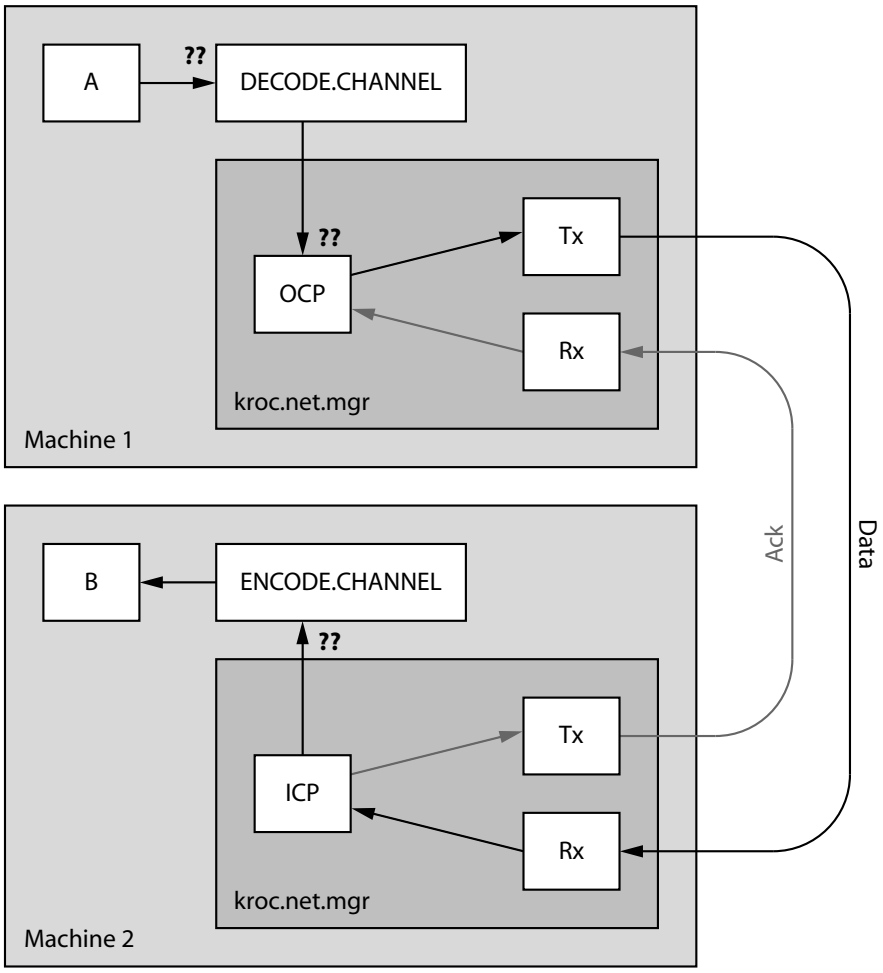

Figure 3: Network Channel Communication

pair, together with the destination ICN, to responsible Tx process. The Tx process then sends the destination ICN and the size as well as the data packet over the network link, where it is received by the responsible Rx process.

Every reading-end of a network channel is handled by an Input Control Process (ICP). When the Rx process receives a data packet, it stores it in a dynamic mobile byte array, detaches it (see section 3) and sends the address/size pair of it to the ICP. The ICP contains a buffer where the address/size pair is stored until it is requested (i.e. until the reading-end of the network channel decides to read). The buffer is necessary in order to prevent the Rx process from blocking, because the Rx process receives the data for *all* network channels that are multiplexed over this link.

The ICP sends the address/size pair to `ENCODE.CHANNEL`, which will take it using the extended rendezvous. `ENCODE.CHANNEL` will encode the address and the size into the appropriate data item and release the ICP as soon as process B has read the data item.

The ICP then sends an acknowledgement, containing the OCN of the writing-end, to the Tx process, who sends it out to the network. The remote Rx process receives it and passes it on to the OCP. As soon as the OCP has received the acknowledgement, it releases `DECODE.CHANNEL` who on his part releases process A.

This proceeding preserves the channel synchronisation semantics of CSP [18]. For process A, the whole communication procedure is an atomic action. It simply sends something to a channel, and as soon as it is released from this communication, it knows for sure that process B has received this ‘something’.

4.2 Handling Network Channels-types

Setting up an NCT means to set up its ends and to connect them. Specifically, 'connecting them' means to make the location of the server-end known to the client-end. The setup procedure is described in detail in section 5. Once the two ends are connected, they can be administrated. As mentioned in section 2.2, the administration of an NCT end and the communication over the network channels inside the NCT are two disjoint states, i.e. only one of them can happen at any one time. Administrating an NCT end may involve a number of operations, including claiming/releasing it, shutting it down, and (in future versions of KRoC.net) moving it.

4.2.1 The Need For a Clear Protocol

As mentioned before, claiming/releasing is *mandatory* for both ends of an NCT, no matter whether the end is shared or not. Claiming/releasing switches between administration and communication state. Claiming a server-end means telling the KRoC.net manager that the server-end is ready to accept the claim from a client-end. Claiming a client-end means that the client-end is making a claim request to the server-end and waits until the server-end accepts this request. Thus, although the semantical meaning of a claim — 'grabbing' the channel-ends inside a channel-type for (exclusive) communication — is the same for both claiming the end of an NCT and `CLAIM`ing the end of a local channel-type, both the reason for the claim and what happens practically are very different.

A local `CLAIM` is necessary for `SHARED` channel-type ends. The claim accesses a semaphore attached to the channel-type end. This is no problem, as everything is located in the same shared memory. In the networking context, there is no 'central' resource like a semaphore, therefore a claim requires a decentralised agreement of both ends. Shared client-ends of an NCT would be located on different machines, thus they have to contact the server-end directly if they want to communicate. This means that the server-end needs to be aware of the fact that a remote client is trying to make a claim. To be so, it must be able to distinguish cleanly between when it is communicating with a client, and when it is waiting for another client to make a claim.

Without this clean distinction, there is the danger of serious race hazards that cannot happen with local channel-types. Consider the following example: The server-end is trying to send something to a shared client-end, but the client-end has already been released. Locally, this would not be a problem. The server-end would simply be blocked until another process claims the client-end and receives the data sent by the server-end. For NCTs, this is not possible. A typical raze hazard situation could arise: The server-end could send the data to the old client-end, whose ICP eventually tries to output it to the application. The application would not take it, but instead tell the KRoC.net manager to release the claim. Once we are at this stage, we are bound to have a deadlock. As there are no output guards in occam, there would be no way to stop the ICP from trying to output to the application.

In a decentralised system such as KRoC.net, an unambiguous communication protocol is therefore vital. This is the reason for the clear distinction between communication and administration. Both sides, the 'server-end side' and the 'client-end side', have to conform to an agreed pattern of communication between themselves. Claim and release become synchronisation points between the ends of an NCT. Both sides have to claim their NCT end, communicate according to their agreed protocol, and release it afterwards. Failure to do so will result in problems such as deadlock. This is, however, acceptable, as for purely local occam programs, similarly misprogrammed communication will result in deadlock, too.

The occam programmer, therefore, must be relied upon to ensure correct patterns of communication — both on local and on networked systems. [19] proposes ways to *make*

processes conformant, i.e. to define the usage of the channels inside a channel-type. The communication pattern of the channel-type would have to be defined by the programmer/designer and would be enforced by the compiler. This is not yet implemented, however, and it is an issue not directly related to KRoC.net, as it would apply to channel-types in general, i.e. both local and networked.

Currently, NCT server-ends cannot be shared, and client-ends are implicitly shared, as currently it makes no difference for the implementation of KRoC.net whether there is one or more client-end connected to the server-end — they have to make a claim before communication anyway.[6]

An NCT server-end may still be `SHARED` locally, of course. Also client-ends of NCTs may be `SHARED` locally. In fact, it is strongly encouraged *not* to have more than one NCT client-end on the same machine connected to the same server-end, but to use the same NCT client-end and share it locally. In order to avoid confusion, a good design rule is to declare each client-end that is supposed to be network-shared *always* locally `SHARED` as well.[7]

4.2.2 Administrating Ends of Network Channel-types

Figure 4 shows the data flow involved in claiming the client-end of the NCT between processes C and D of Figure 2 in detail. We assume that the server-end has just been claimed, i.e. the application connected to the server-end has just told the KRoC.net manager that it is ready for communication.

Every client-end of an NCT is handled by a Client Control Process (CCP), every server-end of an NCT is handled by a Server Control Process (SCP). These are connected to the application by a channel-type called `NET.NCT.MGR`. `NET.NCT.MGR` contains a `req`uest and a `reply` channel. In order to claim an NCT end, the application sends a claim request down the `req` channel, and waits for a reply.

If a client is claimed, the CCP sends the claim request to the remote SCP. This is done via Tx and Rx, in the same way as the OCP/ICP handle the communication: server-ends and client-end are identified by a unique Server/Client Control Number (SCN/CCN) which is sent along with each communication over Tx/Rx in order to identify the destination SCP/CCP.

When the SCP receives the claim, it informs all the OCPs/ICPs belonging to the network channels in the NCT about it and tells them over which link the remote client-end can be accessed, as well as the respective remote ICNs/OCNs. Then it sends an acknowledgement back to the remote CCP, who on his part will inform its own OCPs/ICPs about the new connection.

If a claim arrives while the server-end is already engaged in a communication with another client, the claim request is put into a queue where it waits until the server is released from this communication. Then the server takes the next claim from the queue and proceeds as described above.

5 Setting up Network Channel-types

The goal of KRoC.net is *transparency* (i.e. application level processes to be blind as to whether their external channels are networked or local) and *simple*/*flexible*/*dynamic* setup. Transparency is nearly achieved and will eventually be 100 per cent. It is also flexible and

[6] Later, when KRoC.net will be further integrated into the occam language (see section 5), we *will* distinguish between shared and non-shared ends of NCTs! This is required especially for the differing behaviour of shared/non-shared ends when they are moved.

[7] Later, when the setup of NCTs will be further integrated into occam, the code generated by the compiler will automatically enforce this rule.

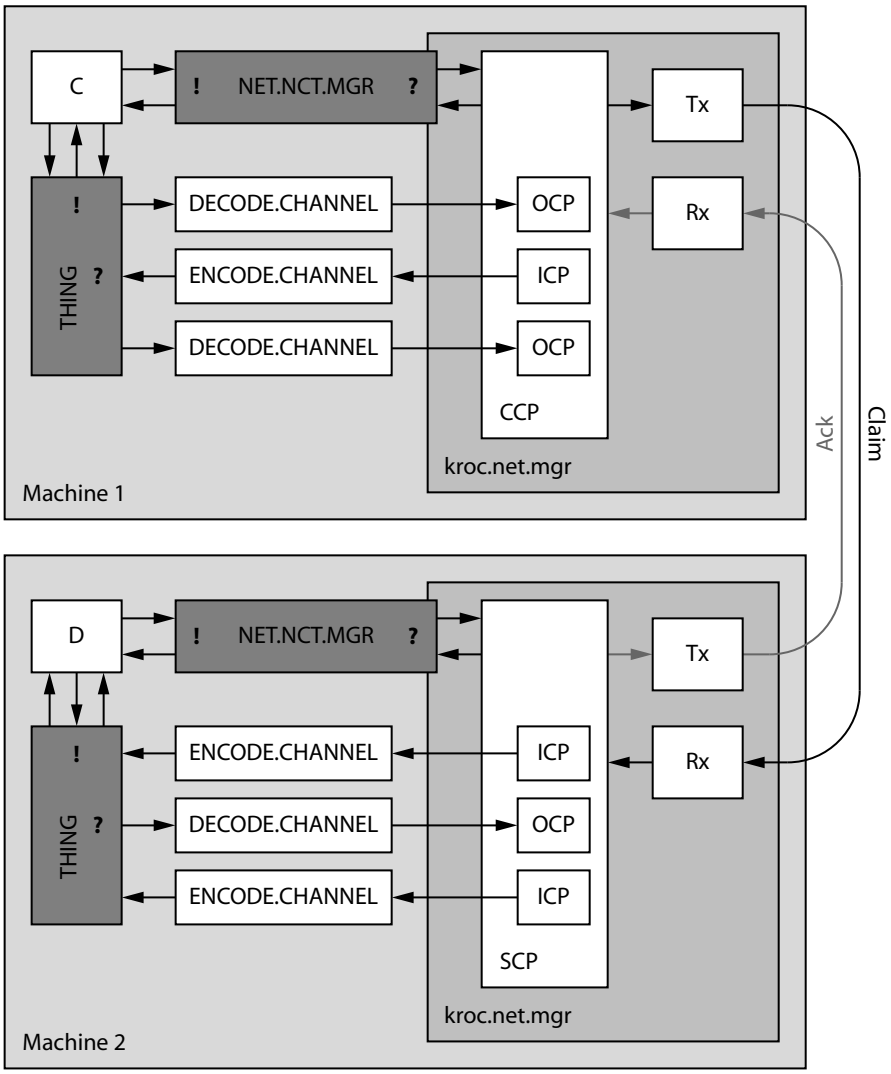

Figure 4: Claiming a Client-end

dynamic. But our current setup procedures are tedious, although straightforward. Future language enhancement will provide direct support for networked channel declaration that will make setup simple (and ensure that it is done correctly).

This section describes the current, rather tortuous but *currently* still necessary, setup mechanisms. The example used is for a networked channel-type — the `THING` defined in Section 2.1.

5.1 The KRoC.net Manager Process

The KRoC.net manager process[8] has the following header:

```
PROC kroc.net.manager.tcp(NET.MGR? net.mgr.svr)
```

`NET.MGR` is a channel-type containing two channels, a `req`est and a `reply` channel. These are used to set up NCTs. An occam program using KRoC.net would typically create an instance of `NET.MGR` with a `SHARED` client-end and a non-shared server-end. The server-end would be passed to the KRoC.net manager, the client-end would be passed to the processes running in

[8]In this section, we refer to the KRoC.net manager for TCP networks.

parallel with the KRoC.net manager who want to set up NCTs with KRoC.net. Here is the start of the code for setting up the *server-end* of a networked channel-type (THING):

```
NET.MGR? net.mgr.svr:          -- used to communicate
SHARED NET.MGR! net.mgr.cli:   -- with the kroc.net.mgr
THING? thing.svr:              -- application level channel-type
THING! thing.cli:              -- (currently needed) supporting infrastructure
SEQ
  net.mgr.svr, net.mgr.cli := MOBILE NET.MGR
  thing.svr, thing.cli := MOBILE THING
  PAR
    kroc.net.mgr.tcp(net.mgr.svr)
    SEQ
      ...  use net.mgr.cli
```

The thing.svr end will be plugged into the application level server process. The component channel-ends of thing.cli will be plugged into respective DECODE.CHANNEL or ENCODE.CHANNEL processes, whose other ends are connected to KRoC.net infrastructure channels manufactured *on-the-fly* by the KRoC.net manager and supplied to us in the following conversation.

5.2 Communications to the KRoC.net Manager

Setting up an NCT means to set up either the client-end or the server-end of it, and to connect them to the remote server-end/client-end via KRoC.net. This is done by sending a request to the KRoC.net manager:

```
net.mgr.cli[req] ! setup.server.end; ANY2ONE; <url>; <prot-hash>; <end-types>
```

Note: if we were setting up the *client-end* of an NCT, the tag in the above communication would be "setup.client.end" rather than "setup.server.end".

ANY2ONE tells the KRoC.net manager that the client-end is shared and the server-end is not. Currently, this is a dummy tag, but in later versions it will have to match for both ends of the NCT, and it will influence the behaviour when a (shared/non-shared) end is set up or moved.

<prot-hash> is the PROTOCOL.HASH value (see section 3.2) of the channel-type we want to connect (i.e. THING in our case). Both the server- and the client-end have to pass the PROTOCOL.HASH value to the KRoC.net manager who can then ensure that matching channel-types are connected.

<end-types> is a counted array protocol over which the application tells the KRoC.net manager about the end-types (i.e. reading- or writing-end) of the channels in the channel-type. Again, they have to match. (For instance if we have a reading-end/writing-end/reading-end sequence in the server-end, the client-end must have a writing-end/reading-end/writing-end sequence.) These values are necessary for the KRoC.net manager to know whether to use OCPs or ICPs.

<url> is a string (a dynamic mobile byte array) that tells the KRoC.net manager about the location of the server-end of the NCT. We created a new URL-based mechanism for setting up NCTs. For this, we have defined a new URL called "nct:". The general definition of an nct: URL is as follows[9]:

[9] <...> are variable names, [...] means optional, | is a choice, (...) is for grouping, everything else is literal. Spaces are not allowed (as usual for URLs).

```
nct: ( <server-name> [ @ ( cns: <cns-name> |
                           cns. <net-type> : <location> ) ] |
       ( <server-name> | $ <scn> )
                       @direct [ . <net.type> : <location> ]
     )
```

<server-name> is a name given to the server-end. This can be any string which does not contain "@" or "$". It could be a simple name like "fred", but also a structured name like "my-application/fred" or so. This is basically up to the programmer.

5.3 The Channel Name Server

The second part of the URL communicated to the KRoC.net manager says where to register (or look up) the <server-name>. The default place for this would be the Channel Name Server (CNS). This is a central server where server-ends can be registered and where client-ends can ask for the location of a server-end according to its name. The CNS can hold a huge number of names, even of completely different distributed applications at the same time. In the latter case, structured names would therefore be sensible.

If the "@" suffix of the URL is omitted, the default CNS will be used to register and ask about the server-end. The location of the default CNS is stored in a special configuration file, described in section 6. Alternatively, the application might want to use a different CNS whose location is stored in another configuration file (cf. section 6). To do so, it would use

```
@cns:<cns-name>
```

whereby <cns-name> is the name of a non-default CNS. The third way of using a CNS is connecting directly to it using

```
@cns.<net-type>:<location>
```

<net-type> is the type of the network; it must be the same as the network type of the KRoC.net manager. <location> is the location of the CNS specific to the network type. For TCP networks, it would be "<ip>:<port>". <ip> would be the IP address or the host name (preferably including the domain) of the CNS, <port> would be the port number. A valid URL would be for instance:

```
nct:fred@cns.tcp:gaia.kent.ac.uk:4400
```

The port can be omitted, in which case the default CNS port of 4400 will be used.

If the requested name cannot be found in the CNS, it will return an error. We will soon add the possibility to specify a certain number of retries or a timeout before an error is returned.

Alternatively to using a CNS, an application may also connect a NCT directly. The server-end would be registered using

```
@direct
```

after the name. The client-end would then ask about the server-end directly at the remote machine, using

```
@direct.<net-type>:<location>
```

The variables mean the same as above for connecting directly to a CNS. If the network type is TCP, and the port is omitted, a default of 4500 will be assumed.

5.4 Replies Back From the KRoC.net Manager

When an NCT end is set up, the KRoC.net manager will establish a CCP/SCP, along with the OCPs and ICPs for all the network channels inside the NCT. If anything goes wrong, the KRoC.net manager returns an error, otherwise it returns the client-end of an `NET.NCT.MGR` channel-type (cf. Figure 4). The server-end of this channel-type is held by the CCP/SCP handling the client-end/the server-end of the NCT just created. As mentioned in section 4.2.2, `NET.NCT.MGR` is used to administrate the NCT end.

The first thing the CCP/SCP returns over `NET.NCT.MGR`'s `reply` channel are client-ends of channel-types of type `NET.OUT` or `NET.IN` for each of the network channels inside the new NCT. The server-ends of these `NET.OUT`/`NET.IN` channel-types are held by the OCPs/ICPs handling the ends of the network channels in the NCT. The occam programmer would then plug each of the `NET.OUT`/`NET.IN` client-ends into a `DECODE.CHANNEL` or `ENCODE.CHANNEL` process, on whose other end the appropriate channels to the application process would be plugged in. Both `NET.OUT` and `NET.IN` contain two channels, one for the address/size pair, and one termination channel for the GPC.

5.5 Unregistered (Anonymous) NCTs

Another way of setting up an NCT is to do so anonymously. For this, there is a special request:

```
net.mgr.cli[req] ! setup.server.end.anon; ONE2ONE; <prot-hash>; <end-types>
```

Unlike the previously mentioned requests, this one does not contain a URL. The server-end is created anonymously, and upon request, it would return an 'anonymous URL' which a client-end could then use to connect to. An anonymous URL has the form

```
nct:$<scn>@direct.<net-type>:<location>
```

where `<scn>` would be the SCN of the server-end. The `@direct<...>` suffix is the same as mentioned above.

This returned string contains all the KRoC.net information normally held in the CNS and supplied to processes requesting `client-end` connections to this NCT. The server process, having not registered its `server-end` anywhere, may now pass that anonymous URL along existing channels (e.g. to a new client with whom it has established a connection over a published NCT). The receiver may use this string to set up its private `client-end`, without going via the CNS.

Note: There is no "`setup.client.end.anon`" tag for the request to the KRoC.net manager, as it is obvious from the URL that we want to set up the client-end anonymously. Instead, the normal `setup.client.end` tag is used. The KRoC.net manager will construct the necessary KRoC.net infrastructure to set up that client-end — just as though it was obtained normally from the CNS. In this way, a client may use a CNS-published NCT only to establish a private connection to a server, who would then `FORK` off a special server process to deal with that client. Otherwise, a server can deal with only one client at a time over its published channel bundle — and we may want to deal with millions concurrently!

In a later version of KRoC.net, anonymous NCTs will be blended into the existing occam semantics for moving ends of channel-types. Instead of declaring an anonymous NCT server-end and passing its URL to the client side who then connects to it anonymously, the programmer would simply allocate an instance of a channel-type in the normal way (i.e. both ends of it) and send one end over a network channel to another machine. This is the same mechanism as it works for local channel-types already. With this new level of transparency, anonymous NCTs will no longer be necessary.

5.6 Modular Design of the KRoC.net Manager

Currently, the only network type supported is TCP (i.e. the network links are TCP/IP socket connections). KRoC.net, however, is easily extensible because of the modular design of the KRoC.net manager. Figure 5 shows the component processes of the TCP KRoC.net manager. For each NCT end that the KRoC.net manager handles, there is a SCP/CCP, along with the OCPs/ICPs for the network channel-ends inside that NCT end.

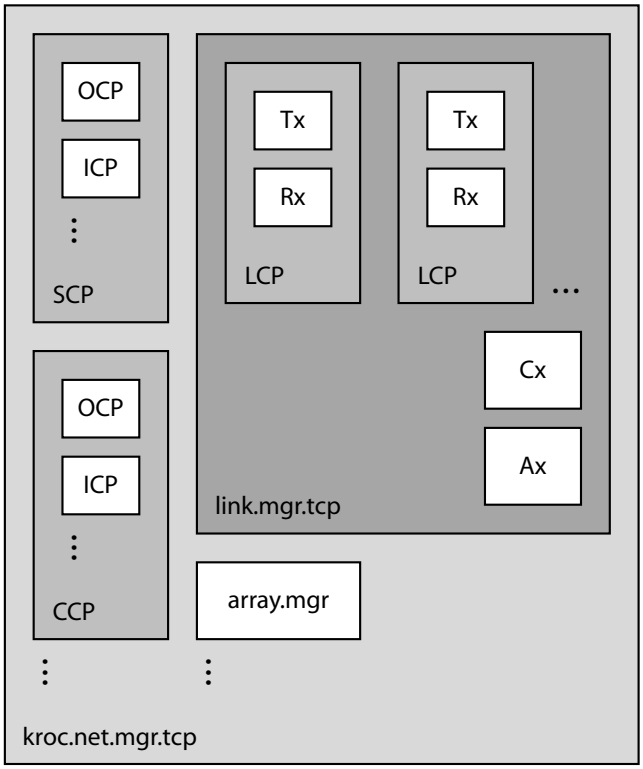

Figure 5: Components of the KRoC.net Manager

The back-end, which deals with the network communication, is inside the link manager, which forms a part of the KRoC.net manager. For each network link, the link manager contains a Link Control Process (LCP), each of which is identified by a unique Link Control Number (LCN). Figure 5 shows the link manager for TCP networks. Each LCP contains a pair of Tx/Rx processes. Additionally, the link manager contains a Cx and an Ax process, dealing with connecting to remote sockets and accepting incoming socket connections.

The TCP version of the link manager uses Barnes' occam Socket Library [20] which enables TCP socket communication under Linux/x86. The Socket Library uses the mechanism to make blocking system calls without blocking the occam kernel, described in [21].

In order to create a KRoC.net manager for a network of a type other than TCP, we would only have to replace the link manager with one for the new network type. Every other code could remain unchanged.

Inside the KRoC.net manager, the LCPs, OCPs/ICPs and SCPs/CCPs are accessed via their respective Control Numbers. For each of them, there is a channel-type called `TO.LCP`, `TO.OCP`, etc., each of which contains just one `req` channel. The server-end (holding the reading-end of `req`) is held by the respective Control Process, the client-end is inside an array at the index that corresponds with the respective Control Number. In this way, all Control Processes can be accessed by their Control Number.

The arrays are managed by special array manager processes who keep a free-list of the

empty elements. When an array is full, its size will be doubled. For this, each array is protected by a CREW lock [22]. Accessing the array to communicate over one of the TO.xCPs requires setting a read lock on the respective array. When the array is doubled, the responsible array manager will set a write lock, create the new array and copy all elements of the old one into it.

The KRoC.net manager is completely modular in its design, i.e. processes are only FORKed off when they are needed. This applies to the LCPs as well as to the 'front-end' Control Processes. Processes that are not needed anymore are terminated. The procedure follows the 'poisoning' approach described in [23].

When the application makes a request to shut down an NCT end, the SCP/CCP sends a shut-down signal to the OCPs/ICPs it has created. These on their part will terminate the GPCs. The link manager keeps a count about the number of NCTs multiplexed over each link. If this number reaches 0, the LCP will be sent a shut-down signal and take care of closing the link and shutting down the Tx/Rx processes. When the KRoC.net manager gets the final shut-down request, it passes it on to all its component processes. Only when everything has cleanly terminated, it will shut-down itself. The basic rule is graceful termination from 'inside out'.

5.7 Proposals for the Further Integration of KRoC.net Into occam

The setup process described in sections 5.1 through 5.5 is relatively easy to understand, but involves quite a number of steps. Although the communication over network channels is nearly transparent now, the setup process is not yet. In order to change this, we will extend the KRoC compiler to hide this setup and administration code behind an extended language syntax for declaring networked channels and channel-types. For this, we have proposed some ideas. These are not 'fixed' yet, however, and subject to discussion.

The idea of forcing a CLAIM for all use of channel-types would enable removal of the explicit network claims that an application process currently has to add to code using channel-types. A run-time test would also need to be generated for these CLAIMs to determine whether the associated channel-type was networked or local. The extra test would only cost a few nanoseconds and be insignificant even for CLAIMs on local channel-types.

For setting up an NCT end, there may be a new keyword called "NET". Contrary to normal channel-type ends, NCT ends would not be allocated in pairs (as the other end will be on another machine!). Here is an example that declares and dynamically constructs the server-end of an *any-to-one* NCT:

```
THING? net.thing.svr:
INT result:
...
SEQ
  net.thing.svr, result := MOBILE NET ANY2ONE TCP THING? "nct:fred"
```

Here is an example for setting up one of many possible networked client-ends for the same NCT.

```
SHARED THING! net.thing.cli:
INT result, timeout:
...
SEQ
  timeout := 5 * seconds
  net.thing.cli, result := MOBILE NET ANY2ONE TCP THING! "nct:fred" timeout
```

TCP is the network type. This is needed to ensure that the right KRoC.net manager will be used. "nct:fred" would be the URL, and result would either be an OK or an error message. timeout specifies how long (or possibly how often) to retry if the required name is not stored in the CNS yet.

This is all that will be needed. All necessary DECODE.CHANNELs/ENCODE.CHANNELs will be created automatically and wired up correctly. All the user needs to do is plug the constructed network channel-type end into the relevant server or client process (after, of course, checking the result).

When a CLAIM is made on an NCT end, KRoC would generate the necessary code to make a network claim with the KRoC.net manager, and also to make a network release when the CLAIM has been finished. Apart from that, if the NCT end is shared, the CLAIM triggers the usual action (i.e. setting the semaphore etc.).

Simple *one-to-one* network channel-ends could be declared like this:

```
INT result:
NET ONE2ONE TCP CHAN INT c? "nct:sue" result:
```

on one machine, and on the other:

```
INT result:
NET ONE2ONE TCP CHAN INT c! "nct:sue" result timeout:
```

KRoC would then create the necessary code as if c? and c! were the ends of an NCT containing a single network channel-end (the server-end containing the reading-end, and the client-end containing the writing-end). As this channel would not be shared, KRoC could do the network claim for both ends immediately after declaration, and the release before the occam program terminates or as soon as c is getting out of scope.

A simple network channel with a SHARED writing-end would be declared:

```
INT result:
SHARED NET ANY2ONE TCP CHAN INT c! "nct:sue" result timeout:
```

on one machine, and its reading-end on the other:

```
INT result:
NET ANY2ONE TCP CHAN INT c? "nct:sue" result:
```

6 Configuration

The only settings which need to be configured are the location (i.e. IP address and port number in the TCP/IP world) of both the own machine[10] (remember that 'machine' refers to an occam program running an instance of the KRoC.net manager) and the CNS. In previous versions, this information was compiled directly into the occam programs. This was OK for experimental use but rather inflexible for real life applications, therefore we have simplified KRoC.net's configuration and made it more flexible.

The locations of the CNS and the own machine are kept in special configuration files. KRoC.net is looking for these files in the program directory and in the user's home directory (in this order).

[10]The location of the own machine is used as a unique identifier for it within the KRoC.net system. This is the location that gets registered with the CNS when the machine registers an NCT server-end.

The 'own' file should be called "`.kroc.net`". Its content should be:

```
[<network-type>]
<location-info>
```

For TCP, this would be:

```
[tcp]
ip=<ip>
port=<port>
```

with `<ip>` being the IP address and `<port>` being the port number. If the port number is omitted, a default of 4500 will be assumed. If the IP address is missing, the standard outgoing IP address will be used.

The CNS uses a similar file, called "`.self.cns`". In this file, however, only the port number is needed, as the CNS does not use its IP address as part of an identifier. If the file is missing, the default CNS port of 4400 will be used.

Finally, an application using KRoC.net needs a file called "`.cns`", where the location of the default CNS is stored. Here, we can use the IP address or the host name (preferably including the domain) of the CNS as a value for `<ip>`. The port can be omitted, in which case the default port number would be 4400.

Information about a CNS other than the default CNS can be stored in a file called "`.cns.<cns-name>`", where `<cns-name>` is the name of the CNS used in the "`@cns:<cns-name>`" part of the `nct:` URL.

7 Performance of KRoC.net

We have run a number of benchmarks to measure the performance of KRoC.net. The benchmarking system consists of a sender on one machine and a receiver on another machine. The sender resides on gaia, a 1GHz Pentium III PC. The receiver is on catch22, a 2.4GHz Pentium 4 PC. Both PCs are connected to a local 100MBit/s ethernet.

7.1 Bandwidth Measurement

For measuring the bandwidth, the sender sends a sequence of equal-sized packets to the receiver over a network channel provided by KRoC.net. This is repeated for different packet sizes between 1 Byte and 64 KBytes (doubling the size each time).

Figure 6 shows the bandwidth (in Bytes/s) for the different packet sizes. For comparison, we have also included the results for sending the packets over raw sockets (without acknowledgement), as well as for sending them over raw sockets where the receiver acknowledges each packet.

Especially the comparison between the 'KRoC.net' and the 'Sockets with ack' results shows the small overhead of the KRoC.net framework. With 1 Byte sized packets, the bandwidth for KRoC.net is about half the bandwidth for 'Sockets with ack'. This is due to the fact that in the 'Sockets with ack' version, the receiver knows which packet size to expect, which is not the case for KRoC.net. Therefore, KRoC.net always needs to send the size first, and then the data[11]. With increasing packet sizes, the impact of the second communication diminishes, however.

[11] We are currently experimenting with a `WRITEV` implementation that sends size and data at once; this is not implemented yet, however.

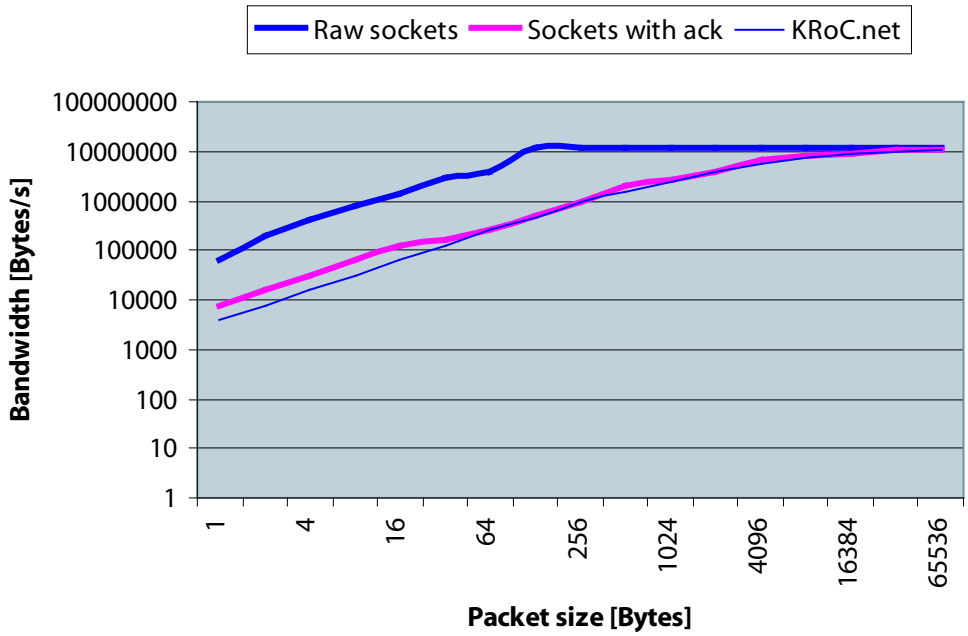

Figure 6: Bandwidth Benchmarks

It is also, perhaps, a little unfair for the 'Sockets with ack' version to know about packet sizes — for flexibility, the sizes would also need to be transmitted. Tests with a modified version of 'Sockets with ack' that also sends the size have produced practically the same results as KRoC.net, which shows the low overheads of KRoC.net.

We have also measured the overheads introduced by KRoC.net with a modified version of the benchmark program. Imagine Machine 1 of Figure 3, but instead of starting network communication when receiving data from the OCP, the Tx process sends the header of the network message over an ordinary occam channel to the Rx process. When the Rx process receives the header, it treats it like an acknowledgement from the network, i.e. it sends an acknowledgement to the OCP, who then releases the extended rendezvous as usual.

On gaia, the ping-pong time for sending a 1 Byte packet (i.e. the time between the sender sends it and receives the acknowledgement) is 1.9μs on the sending machine. This is completely negligible, as the error range for the benchmarks is about 10% (i.e. about 20μs for 1 Byte sized packets). Tests with the Distributed occam Protocol (DoP) [11], a predecessor of KRoC.net from 2001, have shown no measurable difference in the benchmark figures.

All this shows the big impact of socket communication, which involves making blocking system calls to the operating system and heavyweight OS-level IPC. At some point, the functionality of KRoC.net will be introduced into RMoX [24], the Raw Metal occam operating system. This will give us a much better performance, as all the socket communication could then be performed from within occam, i.e. without involving heavyweight OS-level context switches.

7.2 Load Measurement

Another benchmark is measuring the load of KRoC.net against other occam processes running in parallel with it. The setup of this benchmark is based on a similar one in [25]. It is shown in Figure 7.

The benchmark is a modification of the previous one. On catch22, the receiver is running

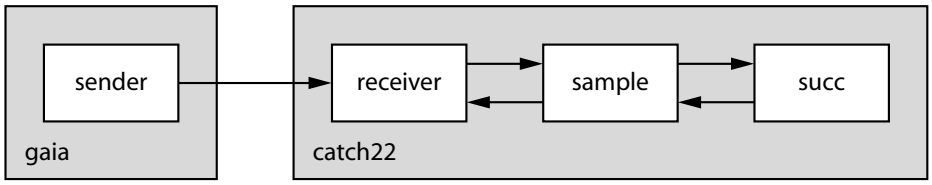

Figure 7: Load Measurement Setup

in parallel with two other processes, called `sample` and `succ`. These two processes are running at the lowest (KRoC occam) process priority [7]. They are always active and running indefinitely — but only, of course, when neither the `receiver` nor any KRoC.net infrastructure processes are active. `sample` is passing a number to `succ` which `succ` increases and sends back. The receiver can interrupt `sample` at any time, whereupon `sample` will return the current count.

The receiver measures the increase of the count against the time passed. First it does so before it starts to receive, normalising this value to 100% background processing capability. Then it measures and compares this value against the ones for the different packet sizes. The result is shown in Figure 8.

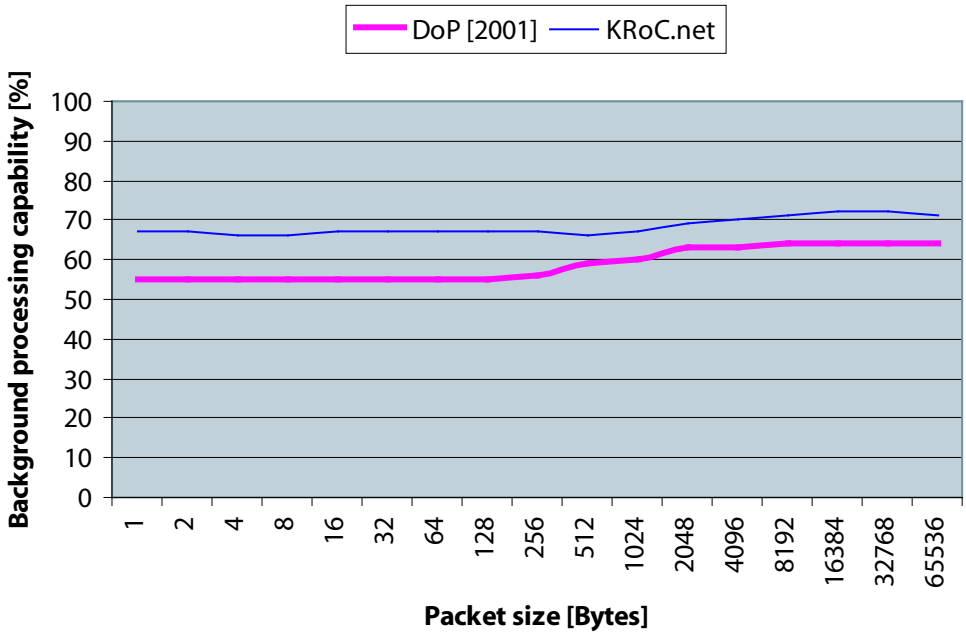

Figure 8: Load Benchmarks

This shows that KRoC.net has a relatively low impact on processes running in the background. It is even slightly better than its predecessor DoP, which is most likely due to the dynamic setup of the network infrastructure — processes are only are started when they are needed — as well as the fact that there is no copying of data involved in the communication process, as KRoC.net only deals with address/size pairs.

8 Conclusions and Future Work

This paper has presented the basic ideas behind the KRoC.net project, its infrastructure, usage and configuration. KRoC.net is now a dynamic tool that enables the occam programmer to

set up network channels and NCTs easily and safely. New occam features in KRoC have been utilised to reach a high level of transparency in the use of KRoC.net. These will be extended to achieve total transparency and a much simpler and automated dynamic setup.

Error handling at the moment only concerns the setup procedure, but not the actual communication. We will add appropriate error messages for the case that something with the communication goes wrong.

We will examine ways to reduce network latency for certain types of communication. This can be achieved by reducing the number of acknowledgement packets sent over the network. There are three concrete approaches for this. Firstly, we will introduce buffered channels, which only require an acknowledgement after the buffer on the receiving side is full. Secondly, we will introduce ping/pong style NCTs (similar to the 'Connections' in JCSP.net). These will follow a strict 'request/reply' scheme, whereby each data packet also acts as an implicit acknowledgement for the previous one.

Additionally, we will improve the network latency for PROTOCOLs that involve more than one communication (e.g. sequential protocols or multi-dimensional dynamic mobile arrays). For this, we will introduce a third field in the link protocol of the GPCs that tells the KRoC.net infrastructure whether subsequent communications are yet to come for this PROTOCOL. Network acknowledgements would only be sent after the last communication of each PROTOCOL.

Another field that our research will look into, is to find ways of sharing server-ends of NCTs, and to implement the possibility to move ends of NCTs around networks, just as ends of local channel-types can be moved around on a local machine.

References

[1] Inmos Limited. occam 2.1 Reference Manual. Technical report, Inmos Limited, May 1995. Available at: http://www.wotug.org/occam/.

[2] Kevin Vella. *Seamless Parallel Computing on Heterogeneous Networks of Multiprocessor Workstations*. PhD thesis, The University of Kent at Canterbury, Canterbury, Kent. CT2 7NF, December 1998.

[3] K. Vella and P.H. Welch. CSP/occam on Shared Memory Multiprocessor Workstations. In B.M. Cook, editor, *Architectures, Languages and Techniques for Concurrent Systems*, volume 57 of *Concurrent Systems Engineering Series*, pages 87–119. WoTUG, IOS Press, the Netherlands, April 1999. ISBN: 90-5199-480-X.

[4] M. Boosten, R.W. Dobinson, and P.D.V. van der Stok. MESH: MEssaging and ScHeduling for Fine-Grain Parallel Processing on Commodity Platforms. In *Proceedings of the 1999 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'1999)*. CSREA press, June 1999. ISBN: 1-892512-15-7.

[5] J. Moores. CCSP – a Portable CSP-based Run-time System Supporting C and occam. In B.M. Cook, editor, *Architectures, Languages and Techniques for Concurrent Systems*, volume 57 of *Concurrent Systems Engineering series*, pages 147–168, Amsterdam, the Netherlands, April 1999. WoTUG, IOS Press. ISBN: 90-5199-480-X.

[6] F.R.M. Barnes and P.H. Welch. Mobile Data, Dynamic Allocation and Zero Aliasing: an occam Experiment. In Alan Chalmers, Majid Mirmehdi, and Henk Muller, editors, *Communicating Process Architectures 2001*, volume 59 of *Concurrent Systems Engineering*, pages 243–264, Amsterdam, The Netherlands, September 2001. WoTUG, IOS Press. ISBN: 1-58603-202-X.

[7] F.R.M. Barnes and P.H. Welch. Prioritised Dynamic Communicating Processes: Part I. In James Pascoe, Peter Welch, Roger Loader, and Vaidy Sunderam, editors, *Communicating Process Architectures 2002*, WoTUG-25, Concurrent Systems Engineering, pages 331–361, IOS Press, Amsterdam, The Netherlands, September 2002. ISBN: 1-58603-268-2.

[8] F.R.M. Barnes and P.H. Welch. Prioritised Dynamic Communicating Processes: Part II. In James Pascoe, Peter Welch, Roger Loader, and Vaidy Sunderam, editors, *Communicating Process Architectures 2002*,

WoTUG-25, Concurrent Systems Engineering, pages 363–380, IOS Press, Amsterdam, The Netherlands, September 2002. ISBN: 1-58603-268-2.

[9] P.H. Welch and D.C. Wood. The Kent Retargetable occam Compiler. In Brian O'Neill, editor, *Parallel Processing Developments, Proceedings of WoTUG 19*, volume 47 of *Concurrent Systems Engineering*, pages 143–166. World occam and Transputer User Group, IOS Press, Netherlands, March 1996. ISBN: 90-5199-261-0.

[10] I.N. Goodacre. occam NetChans, 2001. Project report.

[11] M. Schweigler. The Distributed occam Protocol - A New Layer On Top Of TCP/IP To Serve occam Channels Over The Internet. Master's thesis, Computing Laboratory, University of Kent at Canterbury, September 2001. MSc Dissertation.

[12] P.H. Welch, J.R. Aldous, and J. Foster. CSP networking for java (JCSP.net). In P.M.A. Sloot, C.J.K. Tan, J.J. Dongarra, and A.G. Hoekstra, editors, *Computational Science - ICCS 2002*, volume 2330 of *Lecture Notes in Computer Science*, pages 695–708. Springer-Verlag, April 2002. ISBN: 3-540-43593-X. See also: `http://www.cs.ukc.ac.uk/pubs/2002/1382`.

[13] P.H. Welch and B. Vinter. Cluster Computing and JCSP Networking. In James Pascoe, Peter Welch, Roger Loader, and Vaidy Sunderam, editors, *Communicating Process Architectures 2002*, WoTUG-25, Concurrent Systems Engineering, pages 213–232, IOS Press, Amsterdam, The Netherlands, September 2002. ISBN: 1-58603-268-2.

[14] P.H. Welch. CSP for Java (JCSP), 1999. Available at: `http://www.cs.ukc.ac.uk/projects/ofa/jcsp/`.

[15] M.D. May, P.W. Thompson, and P.H. Welch. *Networks, Routers and Transputers*, volume 32 of *Transputer and occam Engineering Series*. IOS Press, 1993.

[16] Henk L. Muller and David May. A simple protocol to communicate channels over channels. In *EURO-PAR '98 Parallel Processing, LNCS 1470*, pages 591–600, Southampton, UK, September 1998. Springer Verlag.

[17] F.R.M. Barnes and P.H. Welch. Prioritised dynamic communicating and mobile processes. *IEE Proceedings – Software*, 150(2), April 2003.

[18] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985. ISBN: 0-13-153271-5.

[19] Frederick R.M. Barnes. *Dynamics and Pragmatics for High Performance Concurrency*. PhD thesis, University of Kent, June 2003.

[20] Fred Barnes. *Socket, File and Process Libraries for occam*. Computing Laboratory, University of Kent at Canterbury, June 2000. Available at: `http://www.cs.ukc.ac.uk/people/rpg/frmb2/documents/`.

[21] F.R.M. Barnes. Blocking System Calls in KRoC/Linux. In P.H. Welch and A.W.P. Bakkers, editors, *Communicating Process Architectures*, volume 58 of *Concurrent Systems Engineering*, pages 155–178, Amsterdam, the Netherlands, September 2000. WoTUG, IOS Press. ISBN: 1-58603-077-9.

[22] Peter H. Welch and David C. Wood. Higher Levels of Process Synchronisation. In A. Bakkers, editor, *Parallel Programming and Java, Proceedings of WoTUG 20*, volume 50 of *Concurrent Systems Engineering*, pages 104–129, Amsterdam, The Netherlands, April 1997. World occam and Transputer User Group (WoTUG), IOS Press. ISBN: 90-5199-336-6.

[23] P.H. Welch. Graceful Termination – Graceful Resetting. In *Applying Transputer-Based Parallel Machines, Proceedings of OUG 10*, pages 310–317, Enschede, Netherlands, April 1989. Occam User Group, IOS Press, Netherlands. ISBN 90 5199 007 3.

[24] F.R.M. Barnes, C.L. Jacobsen, and B. Vinter. RMoX: a Raw Metal occam Experiment. In *Communicating Process Architectures 2003*, WoTUG-26, Concurrent Systems Engineering, IOS Press, Amsterdam, The Netherlands, September 2003.

[25] Peter H. Welch and Michael D. Poole. occam for Multi-Processor DEC Alphas. In A. Bakkers, editor, *Parallel Programming and Java, Proceedings of WoTUG 20*, volume 50 of *Concurrent Systems Engineering*, pages 152–174, Amsterdam, The Netherlands, April 1997. World occam and Transputer User Group (WoTUG), IOS Press. ISBN: 90-5199-336-6.