Communicating Process Architectures 2004
Ian East, Jeremy Martin, Peter Welch, David Duce, and Mark Green (Eds.)
IOS Press, 2004

Adding Mobility to Networked Channel-Types

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

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

Abstract. This paper reports the specification of a sound concept for the *mobility* of network-channel-types in KRoC.net. The syntax and semantics of KRoC.net have also been modified in order to integrate it more seamlessly into the occam-π language. These new features are currently in the process of being implemented. Recent developments in occam-π and KRoC (such as live/dead channel-type-ends and mobile processes) are described, together with their impact on KRoC.net. This paper gives an overview of the recent developments in KRoC.net, and presents its proposed final semantics, as well as the proposed interface between the KRoC.net infrastructure and the KRoC compiler.

1 Introduction and Motivation

Distributed applications are increasingly important in today's world. Infrastructures such as the Grid [1, 2, 3] are specifically designed for the distribution of large computational tasks onto decentralised resources. Many systems supporting the development of distributed applications are built on the paradigm of remote process or method calls. This applies to systems such as CORBA [4], or the Globus Toolkit [5, 6], which is built on Grid technology to provide a basic infrastructure for metacomputing. Other architectures are built on distributed shared memory or tuple spaces, for instance Linda [7] and Java PastSet [8]. Another common approach for developing distributed applications is the Message Passing Interface [9], implemented for instance in the popular LAM/MPI library [10].

KRoC.net is the networking extension of KRoC, the Kent Retargetable occam Compiler [11]. KRoC has been developed at the University of Kent. The programming language it compiles is occam-π, the new dynamic version of the classical occam[1] [12]. Originally targeted at transputer platforms, it was specifically designed for the efficient execution of fine-grained, highly concurrent programs.

The dynamic features of occam-π [13, 14] offer a new way of distributed application development. A well-designed parallel programming language like occam-π can naturally capture the highly parallel 'real world'. This is particularly so if it enables the programmer to use the same concurrency mechanisms for local and distributed applications. Networked services in KRoC.net are represented by *network-channel-types (NCTs)*. These are networked versions of occam-π's mobile channel-types. Thus, distributed occam-π applications can be designed and programmed in the same way as local ones.

KRoC.net has gone through a number of development cycles [15, 16, 17]. Reported here is the planned support for the mobility of NCTs. Supporting this feature, NCTs will be fully transparent in their use; the programmer can communicate over them and move them around just like their local equivalents. We are also incorporating various recent developments in occam-π (such as live/dead channel-type-ends and mobile processes) into KRoC.net.

[1] occam is a trademark of ST Microelectronics.

The syntax and semantics of KRoC.net have been modified in order to integrate it more seamlessly into the occam-π language. We have now specified a final semantics for KRoC.net and the interface between the KRoC.net infrastructure and the compiler. Supporting mobility, network-channel-types will be fully exchangeable with their local equivalents so far as the processes using and moving them are concerned (i.e. the networking will be transparent). The final step in the development of KRoC.net will be the completion of the implementation of the KRoC.net infrastructure and its full integration with the KRoC compiler, so that KRoC.net can be part of a future KRoC release.

Section 2 discusses some new features in occam-π that have influenced the design of KRoC.net. Section 3 explains KRoC.net's basic architecture. Section 4 introduces the new concept of network-handles. Sections 5 and 6 discuss KRoC.net's new features in detail. Section 7 summarises the paper and identifies areas for future work.

2 Related Extensions

We have proposed, and partly already implemented, various new features for occam-π. This section gives a summary of those of the new features that have had an impact on the design of KRoC.net in one way or another.

2.1 `GATE`, `HOLE` *and Dead Channel-Type-Ends*

These have been proposed by Welch [18] (and may be added to KRoC in the near future) to tackle a drawback arising from the new dynamic features in occam-π. The classical static occam fixed the design of its process networks (and the channels between the processes) at compile time. It was always obvious over which 'interface' a process would communicate with its environment. With the recently introduced mobile channel-types [13], we have unfortunately also introduced the possibility of hidden communication routes that are not declared in the interface (i.e. the header) of the process.

Previously, the ways in which a process interacted with its environment (e.g. through channels and barriers) could be statically and explicitly listed in the process header. Introducing mobile channel-types means that the set of possible interactions for any process can grow at runtime, so that interactions can take place that were not declared by its interface. This raises specification and security issues that are similar to those found in common OO languages (where aliasing is endemic and the opportunities for object interaction exceed those declared by their public interfaces [19, 20]).

Within our research group at Kent, the following rules have been proposed, for which we now invite feedback from the community. They ensure that there are no hidden interactions between a process and its environment — a property which we call 'structural integrity' — despite the mobility of channel-types:

Definition:

(a) Channel-type-end parameters *may* be qualified as being `GATE` or `HOLE`. `GATE` and `HOLE` parameters are *live*.

(b) All other channel-type-ends are *dead* (i.e. locally declared channel-type-end-variables and parameters not qualified as `GATE` or `HOLE`).[2]

[2]This property of channel-type-ends is *static* — each variable is either *always* `GATE` or *always* `HOLE` or *always* dead.

Usage:

(c) A process may not communicate over a dead channel-type-end.

Assignment/Communication:

(d) GATE channel-type-end parameters have VAL semantics — they may not be changed inside the process in whose header they are declared.

(e) GATE, HOLE and dead channel-type-ends may be freely assigned/communicated to each other as long as this does not break Rule (d).[3]

Parameter-Passing:

(f) Arguments for GATE parameters may only be live variables — unless the process is being FORKed. If the process is being FORKed, both live and dead arguments are allowed, as long as this does not break Rule (d).[4]

(g) Inside the scope of a CLAIM, a claimed SHARED channel-type-end may be passed as an argument *only* to a non-shared GATE parameter of a process that is not being FORKed.[5]

(h) HOLE parameters are initially undefined when a process starts. Arguments for HOLE parameters may be outer HOLE parameters of matching type which must be currently undefined, or the keyword HOLE. The latter may only be supplied to FORKed processes. HOLE parameters have no return value (i.e. for the calling process they are still undefined when the callee process terminates).

(i) Arguments for dead parameters may be dead or HOLE variables — unless the process is being FORKed. If the process is being FORKed, GATE variables are also allowed as arguments, as long as this does not break Rule (d).

The aim of these rules is that processes only interact with their environment through formally declared live parameters. In the case of HOLE parameters, what they are bound to may change dynamically, but only by explicit action of the processes themselves (by internally assigning or communicating a newly acquired channel-type-end to one of its HOLE parameters). But the external shape of a process does not change — we have structural integrity. There are no undeclared routes into or out from the process.

Additional issues arise from FORKing. It is proposed to restrict forking so that a process cannot fork off another process without the calling process being aware of it. This could be implemented by introducing a 'FORKS' keyword after which a process would declare all possible processes that it (or any subsequently called processes) might fork off (similarly to exceptions and the 'throws' keyword in Java). Since this might be a rather heavy burden for the programmer, we will probably go for a lighter approach. This could be a marker by which a process could be marked as a 'FORKING PROC'. Only FORKING PROCs would then be allowed to fork off other processes or subsequently call other FORKING PROCs.

[3]It would, for instance, be possible to assign a SHARED GATE parameter to a dead variable, but it would not be possible to assign a non-shared GATE parameter to another variable because this would leave the GATE parameter undefined, which is not allowed. Nothing can be assigned/communicated to a GATE.

[4]Note that the semantics of passing arguments to parameters of FORKed processes is anyway that of communication. So, this clause conforms with Rule (e).

[5]This forces conformity to the existing rule that inside a CLAIM, a live parameter may only be used for communication; it technically becomes non-shared and its value frozen.

For KRoC.net, the new live/dead property of channel-type-ends is important when it comes to moving NCT-ends. An NCT-end may be allocated on one node, and then passed on to several different nodes before it is actually used for communication. We will be able to increase the performance of KRoC.net by only setting up the network infrastructure and the Generic Protocol Converters (GPCs) [16] for an NCT if an NCT-end is assigned to a live variable. The network infrastructure and the GPCs will not be set up as long as it is assigned to a dead variable, which may be passed on to several different nodes before it ends up in a live variable that actually *is* used for communication.

2.2 Mobile Processes

occam-π's new mobile processes are planned to be supported by KRoC.net as well. That is, it will be possible to move mobile processes to remote nodes via networked channels. The semantics of mobile processes has changed quite significantly compared to the first proposals. Details about these changes and the first steps of their implementation can be found in [14].

Particularly important for KRoC.net is the following:

> The header of a `MOBILE PROC`ess may only contain 'synchronisation objects' like channels, barriers, etc., as well as `GATE` or `HOLE` channel-type-ends. It may *not* contain data items or dead channel-type-ends.

This provides a clean interface when it comes to moving mobile processes to remote nodes. When a mobile process, including its workspace, is moved to a remote node, also channel-type-ends stored in this workspace must be moved to the new destination. This would be done in exactly the same way and using the same mechanics as if they were moved directly over a networked channel carrying channel-type-ends (i.e. channel-types may be stretched between nodes). After moving both the mobile process and the channel-type-ends it contains to the new location,[6] all pointers in the process' workspace to channel-type-ends would be updated — the compiler knows where they are in the new location's mobilespace.

3 Architecture

We introduce here some terms used in the rest of this paper:

- A *node* is an occam-π program which is using KRoC.net, i.e. whose main process has declared a network-handle (see Section 4).
- A *network-channel-type (NCT)* is a channel-type that connects several nodes, i.e. whose ends are on more than one node. A *network-channel* is the networked version of a 'classical' occam channel.
- A group of nodes forms a logical *application*. In the non-networked world, node and application would be congruent. In the networked world, an application is made up of several nodes. Nodes can only communicate over NCTs that belong to the same application as they do, and accordingly, each NCT can only connect nodes of the same application.

[6] Note that this only applies to dead and `HOLE` variables. `GATE` parameters are re-assigned anyway when the mobile process is invoked the next time, therefore it would be pointless to move them to the new location.

- A *Channel Name Server (CNS)* is an external server that administrates applications, nodes and NCTs. Each application has a name that is unique within the CNS by which it is administrated. Within the application, each node and each NCT has a unique name or automatically assigned ID. Each node has to register with a CNS before it can do any network communication. NCTs are either allocated under an application-unique name *explicitly* via the CNS, or *implicitly* by moving ends of locally allocated channel-types to remote nodes.

- The *network-type* is the type of a network infrastructure used by KRoC.net. Every network-type has its own *KRoC.net Manager* (the process managing KRoC.net's network connections). If a node declares a network-handle of a particular network-type, an instance of the respective KRoC.net Manager will be started. Details can be found in Section 4. Currently, TCP/IP is the only supported network-type. However, adding support for others will be relatively easy, as the front-end of the KRoC.net Manager (which the compiler interfaces) is the same for all network-types; just the back-end (the 'network driver') needs to be exchanged.

In summary, KRoC.net allows many network-types, each of which may be served by many CNSes. Each CNS may administrate many different applications. Every application may consist of many nodes which may be connected by many NCTs.

4 Network-Handles

KRoC.net will be released as a library bundled with KRoC. In order to be able to allocate ends of network-channels and NCTs (see Section 5) inside a process, that process must declare a *network-handle* as a `GATE` parameter in its header. A network-handle is the client-end of the following channel-type declared in the KRoC.net library:

```
CHAN TYPE NET.HANDLE
  MOBILE RECORD
    CHAN NET.HANDLE.REQ req?
    CHAN NET.HANDLE.REPLY reply!
:
```

The KRoC.net Manager holds the server-end, the user-level program holds the client-end of the '`NET.HANDLE`' channel-type. Each communication with the KRoC.net Manager over a network-handle is a sequence of a '`req`'uest and a '`reply`'. However, these communications are not meant to be done in the user-level code; they are wrapped by special processes.

For all allocations of NCTs, the network-handle is required. This ensures structural integrity, since the calling process will be aware of the fact that the callee might allocate an NCT-end.

4.1 Typing Network-Handles

If the process that declares the network-handle is the *main* process of an occam-π program, the compiler forks off an instance of the KRoC.net Manager to run in parallel with the main process.

For each network-type, there is a separate KRoC.net Manager. In order to know which one to use, the network-handle in the main process must be typed. The compiler can then use the correct KRoC.net Manager depending on the type of the network-handle. Currently, the only supported network-type is TCP/IP, but as the KRoC.net Manager is modular in its

design, other types could easily be supported by exchanging the back-end of the KROC.net Manager.

So, the following code:

```
PROC main (CHAN BYTE keyb, scr, err, GATE NET.HANDLE$TCPIP! net)
  ...  do stuff using 'net'
:
```

would have the semantics of:

```
FORKING PROC main (CHAN BYTE keyb, scr, err, HOLE NET.HANDLE! net)
  NET.HANDLE? net.handle.svr:
  SEQ
    net.handle.svr, net := MOBILE NET.HANDLE
    FORK kroc.net.mgr.tcpip (net.handle.svr)
    ...  do stuff using 'net'
:
```

The '$TCPIP' refers to the network-type. The compiler would in this case use 'kroc.net.manager.tcpip' as the KROC.net Manager. A network-handle of another network-type could be declared accordingly:

```
PROC main (CHAN BYTE keyb, scr, err, GATE NET.HANDLE$OTHERTYPE! net)
  ...  do stuff using 'net'
:
```

would have the semantics of:

```
FORKING PROC main (CHAN BYTE keyb, scr, err, HOLE NET.HANDLE! net)
  NET.HANDLE? net.handle.svr:
  SEQ
    net.handle.svr, net := MOBILE NET.HANDLE
    FORK kroc.net.mgr.othertype (net.handle.svr)
    ...  do stuff using 'net'
:
```

When a *non-main* process declares a network-handle, it *may* be typed, but it does not have to be. If it is typed, the calling process may only pass a network-handle of the same type to the callee. If a network-handle is not typed, the calling process may pass any network-handle to this parameter, either typed or untyped. So, if a non-main process wants to allocate NCTs, it would declare something like:

```
PROC my.proc (<...>, GATE NET.HANDLE! net, <...>)
  ...  do stuff using 'net'
:
```

where the 'NET.HANDLE' parameter (untyped in this example) could be at any position in the parameter list. In a non-main process, the declaration of a network-handle would *not* cause the forking of the KROC.net Manager. It is just a parameter to which the calling process can pass its own 'NET.HANDLE' variable as an argument.

4.2 Shared Network-Handles

Network-handles may be SHARED. The normal rules for sharing channel-type-ends apply. If the main process declares a SHARED network-handle parameter:

```
PROC main (CHAN BYTE keyb, scr, err, SHARED GATE NET.HANDLE$TCPIP! net)
  ...  do stuff using 'net'
:
```

this would have the semantics of:

```
FORKING PROC main (CHAN BYTE keyb, scr, err, SHARED HOLE NET.HANDLE! net)
  NET.HANDLE? net.handle.svr:
  SEQ
    net.handle.svr, net := MOBILE NET.HANDLE
    FORK kroc.net.mgr.tcpip (net.handle.svr)
    ...  do stuff using 'net'
:
```

A SHARED network-handle may be passed to many parallel processes who may then use it to call KRoC.net's special processes (e.g. the allocation processes described in Section 5.4). To do this, the network-handle must be CLAIMed first (cf. Rule (g) in Section 2.1). So, a non-shared network-handle would be used in this way:

```
PROC my.proc (<...>, GATE NET.HANDLE! net, <...>)
  ...  declarations
  SEQ
    ...  do stuff
    <alloc-proc> (net, <...>)
    ...  do more stuff
:
```

whereas a SHARED network-handle would be used in the following way:

```
PROC my.proc (<...>, SHARED GATE NET.HANDLE! net, <...>)
  ...  declarations
  SEQ
    ...  do stuff
    CLAIM net
      <alloc-proc> (net, <...>)
    ...  do more stuff
:
```

4.3 Restrictions

The programmer should use the 'NET.HANDLE' channel-type only in certain ways; this may be enforced by the compiler later. To comply with the rules, user-level code may *only* declare (possibly SHARED) GATE client-ends of that channel-type (which may be network-typed) in the headers of processes. It is not allowed to declare 'NET.HANDLE' channel-type-end-variables that don't meet these criteria. A user-level process may not declare channels carrying 'NET.HANDLE' channel-type-end-variables (which would not make much sense anyway, since only GATE client-end-variables are allowed — into which communication is prohibited anyway, cf. Rule (d) in Section 2.1).

User-level code may not communicate over network-handles directly. The only thing for which network-handles may be used is passing them as arguments to parameters — either to user-level processes (who on their part can do the same) or to KRoC.net's special processes (who then *will* use the network-handle to communicate with the KRoC.net Manager).

5 Network-Channels and Network-Channel-Types

Network-channels and NCTs are the 'backbone' of KRoC.net. KRoC.net uses occam-π's existing paradigm of channels and channel-types as an abstraction on which networked services are built. Channel communication, embedded in the semantics of the CSP calculus [21, 22], is a powerful paradigm for concurrent applications. Therefore KRoC.net uses the same paradigm for distributed applications — which are concurrent by nature. The occam-π programmer will therefore be able to use network-channels and NCTs in the same way as their local equivalents. This transparency is the key requirement in KRoC.net, outweighing all other requirements.

There are two ways to allocate an NCT: explicitly via the CNS, under a name that is unique within the application to which the nodes belong who allocate the NCT's ends; or implicitly, by allocating a channel-type locally, and then sending one of its ends to a remote node. In the latter case, the NCT will be assigned an application-unique ID as soon as it becomes networked. In both cases, no matter how the NCT-ends were originally allocated, their usage (communication, claiming of shared ends, moving of ends) has the usual syntax and semantics. This will all be transparent to the programmer.

5.1 Joining An Application

The first thing which a node must do in order to use KRoC.net is join an application on a CNS. Each node belongs to exactly one application. To join an application, the following process must be called:

```
PROC net.join.app (GATE NET.HANDLE! net, VAL []BYTE cns, app.name, node.name,
                   RESULT INT result, RESULT MOBILE []BYTE full.node.name)
```

- '`net`' is the network-handle.
- '`cns`' is the name or location of the CNS that administrates the application that the node wants to join. If the string is empty, the default CNS is used. Otherwise, if the string does not contain a '`$`', it will be regarded as the name of a non-default CNS. If the string has the form '`<net-type>$<location>`', it will be used to find the CNS directly (without looking at a configuration file).
- '`app.name`' is the name of the application that the node wants to join. This can be any name, containing any characters.
- '`node.name`' is the name of the node. This can be any name that does not contain '`$`'.
- '`result`' is the return value of the process. It is either an OK or an error message (e.g. if the CNS is not available).
- '`full.node.name`' is a return value that contains a unique identifier of the node. With this mechanism, it is possible for identical nodes (with the same name) to join the same application. This could be several 'client nodes' who want to communicate with the same 'server node', or similar architectures. If several nodes with the same name try to join the same application, the CNS will add a unique number to their name, so that they can be distinguished. So, if three nodes named '`darwin`' try to join the same application, they will be named '`darwin`', '`darwin$1`' and '`darwin$2`' by the CNS.

The '`net.join.app()`' process is implemented by the following sequence of communications over the network-handle:

```
net[req] ! join.app; <cns>; <app-name>; <node-name>
net[reply] ? join.app; <result>; <full-node-name>
```

- '<cns>', '<app-name>' and '<node-name>' are 'MOBILE []BYTE' arrays. The parameters of the 'net.join.app()' process are used for these.
- '<result>' is an 'INT'; '<full-node-name>' is a 'MOBILE []BYTE' array. They are used as the return values of the 'net.join.app()' process.

5.2 Extended Memory Structure For Channel-Types

The memory block of a channel-type, which is located in the dynamic mobilespace of the node and to which the channel-type-end-variables point, is subsequently called *channel-type-block (CTB)*. A traditional (local) channel-type is made up of exactly one CTB. An NCT is made up of several CTBs, namely one CTB on each node where there are end-variables of that NCT. The CTBs of an NCT are connected by the KRoC.net infrastructure.

In order to accommodate the needs of NCTs, KRoC needs to extend the memory structure of CTBs. Currently, a CTB contains the channel-words, the reference-count, and the client- resp. server-semaphores if SHARED. This needs to be extended by the following:

1. A *state-flag* that has one of the following states: *local*, *networked* or *localised*.
2. A *live-reference-count* which counts the number of channel-type-ends held by a GATE or HOLE variable.
3. A *connected-flag* that has one of the following states: *disconnected* or *connected*.
4. A *state-semaphore* that protects the state- and the connected-flag.
5. A pointer to the *NCT-handle* (an 'NCT.HANDLE!' client-end-variable).
6. A pointer to the *client-handle* (a 'CLI.HANDLE!' client-end-variable).
7. A pointer to the *server-handle* (a 'SVR.HANDLE!' client-end-variable).

The possible states of a CTB, as well as the newly introduced handles, are described in the following sections.

5.3 States of a Channel-Type-Block

A CTB can be in several states. The simplest state is the *local* state. In this state, the CTB is in no way connected to the KRoC.net Manager. Communication and claiming of shared ends is done in the traditional way. A CTB can only be in local state if it has been allocated traditionally:

```
THING! thing.cli:
THING? thing.svr:
thing.cli, thing.svr := MOBILE THING
```

In order to use the CTB as part of an NCT, it needs to be in *networked* state. A CTB can enter networked state in the following ways:

- The CTB has been allocated by allocating its end-variable(s) as part of an NCT explicitly via the CNS. Details can be found in Section 5.4. In this case, the CTB starts its life in networked state (it will never be in local state at all, but may become localised — see below).

- The CTB has originally been allocated traditionally, i.e. it was in local state at the beginning. Then an end-variable of the CTB is moved to a remote node over a networked channel. As soon as that happens, the state of the CTB changes to networked.

- The CTB starts its life when an end-variable of an NCT is received by our node (provided that no end-variables of that NCT were present on our node before). Like in the first case, the CTB starts its life in networked state and will never be in local state. Details about the movement of channel-type-ends over networked channels can be found in Section 6.

The last state is *localised*. If a CTB is in localised state, this means that it has been in networked state before, but currently *all* end-variables of the NCT are on our node (i.e. the CTB is currently the only one that the NCT is made up of). If this is the case, the claiming and communication will be like in local state. However, previously set-up network infrastructure and and GPCs will remain in place (hence the distinction between 'local' and 'localised'). A CTB can become localised in the following ways:

- The NCT to which the CTB belongs is *one2one* (i.e. it has a non-shared client-end and a non-shared server-end). Our CTB was referred to by one of the ends. Either the other end was not yet allocated and is now allocated on our node as well; or the other end was on a remote node and now moves (maybe back) to our node.

- The NCT is either *any2one* or *one2any* (i.e. it has one `SHARED` and one non-shared end) and has been allocated implicitly (not via the CNS). Only the non-shared end has ever been on a remote node. Now the non-shared end moves back to our node.

Note that in any other case (*any2any* NCTs, or explicitly allocated any2one or one2any NCTs, or implicitly allocated any2one or one2any NCTs whose `SHARED` end was on a remote node before), the CTB cannot enter localised state, because our node can never be sure that no end of this NCT is on a remote node. The reason is that new variables of a `SHARED` channel-type-end can be created on a remote node at any time (by assigning/communicating existing variables into new ones; or, in the case of explicitly allocated NCTs, by allocating new ones via the CNS). Although it would theoretically be possible to devise a protocol to keep track of these things, in practice this may be very complex and far outweigh the advantages of having the localised state in the first place.

The connected-flag can be in two states: *disconnected* or *connected*. It is used to determine whether the network infrastructure and the GPCs have already been set up. If the CTB is in local state, the connected-flag is always 'disconnected'. As soon as the CTB enters networked state, the run-time system will check the live-reference-count. If it is greater than 0, it will set the connected-flag to 'connected' and set up the network infrastructure and the GPCs. If the live-reference-count was 0 when the CTB entered networked state, as soon it increases the connected-flag will be set to 'connected' and the network infrastructure and the GPCs be set up.

Once the connected-flag is 'connected', it will stay 'connected' as long as the CTB exists. Even if the live-reference-count gets back to 0, the connected-flag will never change back to 'disconnected'. This also means that once the network infrastructure and the GPCs have been set up, they will remain in place until the CTB has been shut down.

If the CTB's state is 'localised', the connected-flag will not be changed even when the live-reference-count changes. It will remain 'connected' or 'disconnected', whichever it was before the CTB entered localised state. When the CTB goes back from localised to networked state and the connected-flag is 'disconnected', the behaviour is the same as if the CTB changes from local to networked state (checking the live-reference-count etc.) If the connected-flag is 'connected' already when the CTB goes from localised to networked state, the connection needs to be confirmed with the KRoC.net Manager. The reason for this, as well as details on connecting CTBs and confirming the connection, can be found in Section 5.6.

The state-semaphore protects the state- and the connected-flag. It must be claimed before changing one of the flags, and during actions that depend on the state of the CTB. Details will be given later in the paper when these actions are discussed.

5.4 Explicit Allocation of NCT-Ends

Explicit allocation of an NCT-end is made by invoking a process such as:

```
PROC net.alloc.one2one.client (GATE NET.HANDLE! net, <CT>! the.cli,
                               VAL []BYTE nct.name, RESULT INT result)
```

This allocates the client-end of a one2one NCT (an NCT with a non-shared client-end and a non-shared server-end). Similar processes exist for the allocation of server-ends, and for any2one, one2any and any2any NCTs.

The first parameter is the network-handle. If a process does not have one, it cannot allocate an NCT-end. This further guarantees our notion of structural integrity — a process cannot establish external connections unless explicitly given that capability. The network-handle is that capability. The other parameters are the channel-type-end being networked, the name for the NCT and a result code. Implementation requires external communication with the CNS, which may fail. In this case the allocation process would return an error.

The full set of allocation processes and details of their parameters and implementation is given in Appendix A.

5.5 Implementation of Network-Channels

Network-channels — i.e. networked single (classical occam) channels — will be implemented as 'anonymous' NCTs that contain exactly one channel. This approach is similar to the implementation of the anonymous SHARED channels discussed in [13].

The following declaration:

```
NET CHAN INT iw!:               -- These network-channel-ends
NET CHAN BOOL br?:              -- must be allocated before
SHARED NET CHAN BOOL sbw!:      -- we can communicate over them!
SHARED NET CHAN INT sir?:
...  allocate iw!, br?, sbw!, sir?
...  use iw!, br?, sbw!, sir?
```

would have the semantics of:

```
CHAN TYPE $anon.INT             -- compiler-generated type
  MOBILE RECORD
    CHAN INT x?:                -- server-end holds reading-end
:
```

```
CHAN TYPE $anon.BOOL            -- compiler-generated type
  MOBILE RECORD
    CHAN BOOL x?:               -- server-end holds reading-end
:

$anon.INT! iw$cli:
$anon.BOOL? br$svr:
SHARED $anon.BOOL! sbw$cli:
SHARED $anon.INT? sir$svr:
...  allocate iw$cli, br$svr, sbw$cli, sir$svr
...  use iw$cli, br$svr, sbw$cli, sir$svr
...  resp. iw$cli[x], br$svr[x], sbw$cli[x], sir$svr[x]
```

where the server-end of the compiler-generated channel-type by definition holds the reading-end of the channel. Before a process can communicate over a network-channel-end, that end needs to be allocated. This is done by allocation processes similar to those for NCT-ends. Details can be found in Appendix A.

Occurrences of network-channel-ends in user-level code are replaced by the compiler by the appropriate generated variables. In parameters, network-channel-ends are replaced by the generated channel-type-end (e.g. '`br?`' is replaced by '`br$svr`'). When used for communication, they are replaced by the actual channel-field (e.g. '`iw ! 5`' is replaced by '`iw$cli[x] ! 5`').

5.6 Connecting a Channel-Type-Block

As pointed out in Section 5.3, the connected-flag stores whether the network infrastructure and the GPCs for the CTB have already been set up. As soon as the live-reference-count becomes greater than 0 (which may be instantly after the allocation if the CTB is allocated into a live variable), the CTB must be connected. This is done (automatically by the run-time system) by first claiming the state-semaphore and updating the connected-flag, and then carrying out the following sequence of communications over the NCT-handle stored in the CTB:

```
nct.handle[req] ! connect; <chan-descs>; <cli-claimed>; <svr-claimed>
nct.handle[reply] ? connect; <cli-handle>; <svr-handle>;
                             <write-handles>; <read-handles>
```

- '`<chan-descs>`' is a '`MOBILE []CHAN.DESC`' array whose size equals the number of channels in the CTB. '`CHAN.DESC`' is the following record structure[7]:

```
DATA TYPE CHAN.DESC
  RECORD
    BOOL write.read:
:
```

  - '`write.read`' specifies whether the channel is a writing-end or a reading-end from the point of view of the server-end of the CTB. It is either '`NET.WRITER`' or '`NET.READER`'.

[7] We implement the channel descriptor as a record, even though it contains only one element at present, in order to be able to easily extend it if necessary (e.g. to support the buffered channels planned for occam-π in the future).

- '<cli-claimed>' and '<svr-claimed>' are 'BOOL's. They specify whether the client- resp. server-end of the CTB is claimed at the moment when it is connected — which can only be the case if the CTB was in local or localised state and has just entered networked state (cf. Section 5.3).

- '<cli-handle>' and '<svr-handle>' are channel-type-ends of type 'CLI.HANDLE!' resp. 'SVR.HANDLE!', which will be stored in the CTB. The server-ends of the client- and the server-handle are held by the KRoC.net Manager.

- '<write-handles>' and '<read-handles>' are dynamic MOBILE arrays whose size equals the number of channels in the CTB. They contain channel-type-ends of type 'WRITE.HANDLE!' resp. 'READ.HANDLE!', who on their part contain the necessary channels to be plugged into the GPCs. (For details on how the GPCs work, refer to [17].) The server-ends of the write- and read-handles are held by the KRoC.net Manager.

The runtime system will then FORK off a DECODE.CHANNEL and an ENCODE.CHANNEL process for each channel in the CTB, and connect them to the reading- resp. writing-ends of the channel-ends in the CTB on the one hand, and to the writing- resp. reading-handles, that are connected to the KRoC.net Manager, on the other hand. When all this is done, the state-semaphore can be released.

5.7 Confirming the Connection of a CTB

If the CTB is in localised state and is already connected, changing back to networked state requires to 'confirm the connection' of the CTB (again, this is done automatically by the runtime system). The reason is that when the CTB is in networked state and connected, the KRoC.net Manager needs to know whether the client- resp. server-end of the CTB is claimed. Confirming the connection is done immediately after changing the state to networked (while the state-semaphore is still claimed), and implemented by the following communication sequence:

```
nct.handle[req] ! confirm.connect; <cli-claimed>; <svr-claimed>
nct.handle[reply] ? confirm.connect
```

5.8 Shutting Down a Channel-Type-Block

If the reference-count of a CTB becomes 0 (which means that all its end-variables have gone out of scope or been overwritten), the runtime system will send a shut-down request to the NCT-handle:

```
nct.handle[req] ! shut.down
nct.handle[reply] ? shut.down
```

The KRoC.net Manager will shut down the GPCs connected to the CTB and then send the reply. When the reply is received, the CTB can be deallocated.

Similarly, the entire KRoC.net Manager as such will be shut down by sending a shut-down signal to the network-handle at the very end of the main process.

6 Mobility of NCT-Ends

6.1 Claiming and Releasing NCT-Ends

In order to communicate over NCT-ends, they must be claimed from the KRoC.net Manager. This is done (automatically) by sending a request to the KRoC.net Manager over the client- resp. server-handle. For `SHARED` ends, this request will be generated at the beginning of a `CLAIM` block, right after the CTB's client- resp. server-semaphore has been successfully claimed. At the end of the `CLAIM` block, just before releasing the client- resp. server-semaphore, a release request is sent to the KRoC.net Manager in order to release the end.

Non-shared ends are claimed when the CTB gets connected. They remain claimed as long as they stay on the same node. When a non-shared end is sent to another node (see below), the `DECODE.CHANNEL` process is responsible for releasing the end before it leaves the old node. When the end is received by the destination node, the `ENCODE.CHANNEL` process immediately claims it again as soon as it arrives, provided that a connected CTB for that NCT already exists on this node; otherwise the end will be claimed, as pointed out above, when its new CTB gets connected.

The claim and release requests are (automatically) implemented as follows:

```
cli.or.svr.handle[req] ! claim
cli.or.svr.handle[reply] ? claim

cli.or.svr.handle[req] ! release
cli.or.svr.handle[reply] ? release
```

When the KRoC.net Manager receives a claim or a release request, it forwards it to the *administration node* of the NCT. Initially, this is the first node that allocates an end of that NCT. The CNS stores the administration node of each NCT. Should that node be shut down, the administration of the NCT will dynamically be moved to another node that holds a CTB that is part of the NCT.

The administration node holds FIFO queues of client-ends and server-ends that have made a claim request. When a previously claimed end is released, the next end in the queue is selected. When this happens, an acknowledgement is sent to the KRoC.net Manager of the node on which the end is located. This acknowledgement contains the current location of the opposite end of the NCT. The KRoC.net Manager then returns a reply over the client- resp. server-handle.

The network link between the nodes of the claimed client-end and the claimed server-end is established dynamically on demand.

6.2 Moving NCT-Ends

The movement of NCT-ends was inspired by the mobile channels in Muller and May's Icarus language [23]. However, implementing mobility for KRoC.net's NCT-ends is more complex because it needs to take into account the special properties of channel-types (e.g. that they are bundles of channels, that the ends may be shared, etc.), as well as KRoC.net's general architecture with the CNS etc.

Moving a channel-type-end is a three phase approach, automated within the KRoC.net infrastructure. Firstly, the `DECODE.CHANNEL` process reads the channel-type-end from the user-level application. If the CTB is local or localised, it claims the state-semaphore and changes the state to networked. If the state of the CTB was local, an implicit allocation is done via the network-handle, similar to the mechanism for explicit allocation described in Appendix A:

```
net[req] ! alloc.implicit.sending; <x2x>; <c/s>; <ctb-pointer>
net[reply] ? alloc.implicit.sending; <nct-handle>
```

where '<c/s>' specifies whether a client-end or a server-end is sent away. The other fields have the same meaning as discussed in Appendix A. The KRoC.net Manager will contact the CNS, where the new NCT will be registered under an application-unique ID. This ID is a '$' followed by a number. As with explicitly allocated NCT-ends, the live-reference-count is checked and the CTB is connected/the connection confirmed if necessary when networked state is entered.

In the second stage, DECODE.CHANNEL sends a release request to the KRoC.net Manager if the end is non-shared and the CTB was already networked and connected. Then DECODE.CHANNEL passes the '<ctb-pointer>' to the KRoC.net Manager, together with a flag which indicates that this pointer refers to a channel-type-end rather than a data item. The KRoC.net Manager matches the '<ctb-pointer>' with the name or ID of the NCT to which the CTB belongs. Then it sends a special 'data packet' to the remote node, containing the name of the NCT. On the receiving node, the KRoC.net Manager finds out whether an NCT-end of that name is already located on the receiving node. If yes, it sets '<ctb-pointer>' to the pointer of the matching CTB — otherwise it sets '<ctb-pointer>' to 0. It sets '<is-localised>' to the correct value (i.e. TRUE in the cases mentioned in Section 5.3).

In the last stage, the KRoC.net Manager passes '<nct-name>', '<ctb-pointer>' and '<is-localised>' to ENCODE.CHANNEL. ENCODE.CHANNEL checks whether '<ctb-pointer>' is 0, in which case it allocates a new CTB and sets its initial state to 'networked'. Then it sends an 'implicit allocation' request via the network-handle:

```
net[req] ! alloc.implicit.receiving; <x2x>; <c/s>; <nct-name>; <ctb-pointer>
net[reply] ? alloc.implicit.receiving; <nct-handle>
```

where '<c/s>' specifies whether a client-end or a server-end has been received. As with explicitly allocated NCT-ends, the live-reference-count is checked and the CTB is connected if necessary.

If the '<ctb-pointer>' received from the KRoC.net Manager was not 0, '<is-localised>' is checked. If it is TRUE, the state-semaphore of the (already existing) CTB would be claimed, the state be set to 'localised', and then the state-semaphore be released. If the (already existing) CTB does not become localised, and if it was already connected and the received NCT-end is non-shared, ENCODE.CHANNEL sends a claim request for the received end to the KRoC.net Manager.

Once this is all done, ENCODE.CHANNEL communicates a channel-type-end pointing to '<ctb-pointer>' to the user-level application.

7 Conclusions and Future Work

This paper has presented the specification of those of KRoC.net's features that were still missing in order for it to be fully transparent. The most important development was the specification of KRoC.net's mobility semantics. Other improvements have been made in order to make the use of KRoC.net as simple and intuitive as possible for the programmer. Wherever possible, we have used existing occam-π syntax and avoided inventing new syntax that would make occam-π more complicated. An example for this is the use of allocation processes rather than the previously proposed new special keywords.

We welcome feedback on all these plans. So far, the basic infrastructure for KRoC.net's network communication has been implemented. The next task will be the implementation

of the new features discussed in this paper. This includes both the KRoC.net library, implemented in occam-π, and the compiler-related parts, such as the extension of the CTBs and the special compiler-generated processes.

Finally, when the implementation of KRoC.net will be fully completed, its performance needs to be examined by a new series of benchmarks.

Acknowledgements

The author is very grateful to Fred Barnes for his work on the KRoC compiler and his helpfulness in discussing the various issues arising from the development of KRoC.net and its integration into the KRoC environment. It has always been a great help to be able to rely on him implementing the compiler-related parts of KRoC.net.

The author would also like to thank the other members of the University of Kent's concurrency research group (Peter Welch, David Wood and Christian Jacobsen), as well as Matt Jadud and the anonymous reviewers. Their advice in reviewing this paper and their contributions to our many discussions on the project were very valuable.

References

[1] I. Foster, C. Kesselman, and S. Tuecke. What is the Grid? A Three Point Checklist. *GRIDToday*, July 2002. Available at: `http://www-fp.mcs.anl.gov/~foster/Articles/WhatIsTheGrid.pdf`.

[2] I. Foster, C. Kesselman, and S. Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International Journal of Supercomputer Applications*, 2001. Available at: `http://www.globus.org/research/papers/anatomy.pdf`.

[3] I. Foster, C. Kesselman, J.M. Nick, and S. Tuecke. The Psychology of the Grid: An Open Grid Services Architechture for Distributed Systems Integration. *Global Grid Forum*, June 2002. Available at: `http://www.globus.org/research/papers/ogsa.pdf`.

[4] Object Management Group. The Common Object Request Broker: Architecture and Specification (CORBA). Technical report, Object Management Group, December 1993. Available at: `ftp://ftp.omg.org/`.

[5] I. Foster and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *International Journal of Supercomputer Applications*, 1997. Available at: `ftp://ftp.globus.org/pub/globus/papers/globus.pdf`.

[6] IBM Corporation. Globus Toolkit 3.0 Quickstart, Redpaper. Technical report, IBM Corporation, 2003. Available at: `http://www.redbooks.ibm.com/redpapers/pdfs/redp3697.pdf`.

[7] Nicholas Carriero and David Gelernter. Linda in Context. *Communications of the ACM*, 32(4):444–459, April 1989.

[8] K.S. Pedersen and B. Vinter. Java PastSet: A Structured Distributed Shared Memory System. In James Pascoe, Peter Welch, Roger Loader, and Vaidy Sunderam, editors, *Communicating Process Architectures 2002*, WoTUG-25, Concurrent Systems Engineering, pages 97–108, IOS Press, Amsterdam, The Netherlands, September 2002. ISBN: 1-58603-268-2.

[9] MPI Forum. MPI-2: Extensions to the Message-Passing Interface. Technical report, MPI Forum, July 1997. Available at: `http://www.mpi-forum.org/docs/mpi-20.ps`.

[10] Indiana University LAM Team. LAM/MPI User's Guide. Technical report, Indiana University, May 2004. Available at: `http://www.lam-mpi.org/download/files/7.0.6-user.pdf`.

[11] 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.

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

[13] Frederick R.M. Barnes. *Dynamics and Pragmatics for High Performance Concurrency*. PhD thesis, University of Kent, UK, Canterbury, Kent, CT2 7NF, June 2003.

[14] F.R.M. Barnes and P.H. Welch. Communicating Mobile Processes. In I. East, J. Martin, P. Welch, D. Duce, and M. Green, editors, *Communicating Process Architectures 2004*, WoTUG-27, Concurrent Systems Engineering Series, ISSN 1383-7575, IOS Press, Amsterdam, The Netherlands, September 2004.

[15] I.N. Goodacre. occam NetChans. Technical report, Computing Laboratory, University of Kent at Canterbury, UK, March 2001. Project report.

[16] 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, Canterbury, UK, September 2001.

[17] M. Schweigler, F.R.M. Barnes, and P.H. Welch. Flexible, Transparent and Dynamic occam Networking With KRoC.net. In J.F. Broenink and G.H. Hilderink, editors, *Communicating Process Architectures 2003*, WoTUG-26, Concurrent Systems Engineering Series, pages 199–224, IOS Press, Amsterdam, The Netherlands, September 2003. ISBN: 1-58603-381-6.

[18] P.H. Welch. Maintaining Structural Integrity in Dynamic Systems. Technical Report UKC-CRG-11-03-2004, Computing Laboratory, University of Kent, Canterbury, UK, March 2004.

[19] T.S. Locke. Towards a Viable Alternative to OO – extending the occam/CSP programming model. In Alan Chalmers, Majid Mirmehdi, and Henk Muller, editors, *Communicating Process Architectures 2001*, volume 59 of *Concurrent Systems Engineering*, pages 329–349, Amsterdam, The Netherlands, September 2001. WoTUG, IOS Press. ISBN: 1-58603-202-X.

[20] P.H. Welch. Process Oriented Design for Java – Concurrency for All. In *PDPTA 2000*, volume 1, pages 51–57. CSREA Press, June 2000. ISBN: 1-892512-52-1.

[21] C.A.R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, August 1978.

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

[23] 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.

Appendix A: Allocation Processes

Allocation Processes for NCTs

To explicitly allocate an NCT-end, one of the following allocation processes must be called:

```
PROC net.alloc.one2one.client (GATE NET.HANDLE! net, <CT>! the.cli,
                               VAL []BYTE nct.name, RESULT INT result)
PROC net.alloc.any2one.client (GATE NET.HANDLE! net, SHARED <CT>! the.cli,
                               VAL []BYTE nct.name, RESULT INT result)
PROC net.alloc.one2any.client (GATE NET.HANDLE! net, <CT>! the.cli,
                               VAL []BYTE nct.name, RESULT INT result)
PROC net.alloc.any2any.client (GATE NET.HANDLE! net, SHARED <CT>! the.cli,
                               VAL []BYTE nct.name, RESULT INT result)

PROC net.alloc.one2one.server (GATE NET.HANDLE! net, <CT>? the.svr,
                               VAL []BYTE nct.name, RESULT INT result)
PROC net.alloc.any2one.server (GATE NET.HANDLE! net, <CT>? the.svr,
                               VAL []BYTE nct.name, RESULT INT result)
PROC net.alloc.one2any.server (GATE NET.HANDLE! net, SHARED <CT>? the.svr,
                               VAL []BYTE nct.name, RESULT INT result)
PROC net.alloc.any2any.server (GATE NET.HANDLE! net, SHARED <CT>? the.svr,
                               VAL []BYTE nct.name, RESULT INT result)
```

The process names are fairly self-explanatory. ‘one2one’ means that the NCT has a non-shared client-end and a non-shared server-end; ‘any2one’ means that the NCT has a SHARED client-end and a non-shared server-end; etc.

- The first argument is the network-handle; the second argument is a (SHARED or non-shared, depending on the ‘<...>2<...>’ part of the process name) client- or server-end of *any* possible channel-type that is in scope.
- ‘nct.name’ is the name of the NCT. This can be any name that does not start with ‘$’.
- ‘result’ is the return value of the process. It is either an OK or an error message (e.g. if the ‘x2x’ type is different from a previously allocated NCT-end of the same name, or if one is trying to allocate more than one non-shared end).

Each allocation process is implemented by the following communication sequence:

```
  net[req] ! alloc; <x2x>; <c/s>; <phash>; <nct-name>
  net[reply] ? alloc; <result>; <ctb-pointer>; <is-localised>
[ net[req] ! confirm.alloc; <ctb-pointer>
  net[reply] ? confirm.alloc; <nct-handle> ]
```

- ‘<x2x>’ is the ‘x2x’ type of the channel-type, and carries one of the following constants: ‘NET.ALLOC.ONE2ONE’, ‘NET.ALLOC.ANY2ONE’, ‘NET.ALLOC.ONE2ANY’ or ‘NET.ALLOC.ANY2ANY’.
- ‘<c/s>’ specifies whether we want to allocate a client-end or a server-end. It is either ‘NET.CLI’ or ‘NET.SVR’.
- ‘<phash>’ is an ‘INT’ carrying the protocol-hash of the channel-type. It is determined using the ‘PROTOCOL.HASH()’ function described in [17].
- ‘<nct-name>’ is a ‘MOBILE []BYTE’ array carrying the name of the NCT.

- '<result>' is an 'INT' and will be returned as the result of the allocation process. In order for the result to be an OK, the 'x2x' type and the protocol-hash must be the same as for any previously allocated NCT-ends of the same name. This is checked in the CNS, where these values are stored. Also, the client/server property must make sense; e.g. only one client-end can be allocated for a one2x channel-type etc.
- '<ctb-pointer>' is an 'INT' carrying the pointer to the CTB of the channel-type. If on our node there is already a CTB connected to a channel-type of the same name, the KROC.net Manager will return the pointer to it. If not, it will return 0.
- '<is-localised>' is a 'BOOL' returned by the KROC.net Manager. If '<ctb-pointer>' is 0, '<is-localised>' is ignored. Otherwise, if '<is-localised>' is TRUE, the state-semaphore of the (already existing) CTB would be claimed, the state be set to 'localised', and then the state-semaphore be released. The only case when this can happen is if the NCT is one2one, and one end has already been on the same node.
- If the '<result>' was an OK and the returned '<ctb-pointer>' was 0, the compiler-generated code would allocate a new CTB, set its initial state to 'networked', and send its pointer back to the KROC.net Manager.
- The KROC.net Manager would then return the NCT-handle, which would be stored in the CTB. This handle will then be used to communicate with the KROC.net Manager with regards to this particular CTB.

The allocation process returns a channel-type-end pointing to '<ctb-pointer>'.

Allocation Processes for Network-Channels

These are the allocation processes for network-channels-ends:

```
PROC net.alloc.one2one.writer (GATE NET.HANDLE! net, <NCE> the.writer!,
                               VAL []BYTE nc.name, RESULT INT result)
PROC net.alloc.any2one.writer (GATE NET.HANDLE! net, SHARED <NCE> the.writer!,
                               VAL []BYTE nc.name, RESULT INT result)
PROC net.alloc.one2any.writer (GATE NET.HANDLE! net, <NCE> the.writer!,
                               VAL []BYTE nc.name, RESULT INT result)
PROC net.alloc.any2any.writer (GATE NET.HANDLE! net, SHARED <NCE> the.writer!,
                               VAL []BYTE nc.name, RESULT INT result)

PROC net.alloc.one2one.reader (GATE NET.HANDLE! net, <NCE> the.reader?,
                               VAL []BYTE nc.name, RESULT INT result)
PROC net.alloc.any2one.reader (GATE NET.HANDLE! net, <NCE> the.reader?,
                               VAL []BYTE nc.name, RESULT INT result)
PROC net.alloc.one2any.reader (GATE NET.HANDLE! net, SHARED <NCE> the.reader?,
                               VAL []BYTE nc.name, RESULT INT result)
PROC net.alloc.any2any.reader (GATE NET.HANDLE! net, SHARED <NCE> the.reader?,
                               VAL []BYTE nc.name, RESULT INT result)
```

'<NCE>' is a 'NET CHAN' network-channel-end as described in Section 5.5. The network-handle, network-channel-name and result parameters are the same as in the allocation processes for NCT-ends. The '.writer' processes are the equivalent of the '.client' processes; the '.reader' processes are the equivalent of the '.server' processes.