Flexible, Transparent and Dynamic occam Networking With KRoC.net

Mario Schweigler, Fred Barnes, Peter Welch

Computing Laboratory, University of Kent
Canterbury, UK

What Is KRoC.net?

- Extension to KRoC
- Framework to distribute occam channels (and channel-types) over networks
- Implemented in occam (mainly)
- Aims:
  - Transparency to the occam programmer
  - Dynamic setup of network channels
  - Flexible configuration and platform-independence

Contents

- Channel-types and network channel-types
- New occam features used in KRoC.net
- Communication over network channels
  - The KRoC.net manager
- Setting up network channel-types
- Proposals for an extended occam syntax
- Configuration
- Performance
- Conclusion and future work

Reminder: Channel-types

- Local channel-types:

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

- Allocation in pairs at runtime:

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

Reminder: Channel-types

- Usage:
  - Server-end:

```
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
:
```

Reminder: Channel-types

- Usage:
  - Client-end:

```
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
:
```

Reminder: Channel-types

- Usage:
  - Allocation of the two ends and passing as parameters:

```
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
```

Reminder: Channel-types

- Communicating channel-type ends:
  - Generator:

```
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
:
```

Reminder: Channel-types

- Communicating channel-type ends:
  - Server:

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

Reminder: Channel-types

- Communicating channel-type ends:
  - Client:

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

Reminder: Channel-types

- Communicating channel-type ends:
  - Main program:

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

Network channel-types (NCTs)

- Channel-types offer:
  - Grouping of channels to a bundle
  - Distinction between the two ends
  - Movable ends
  - Sharable ends

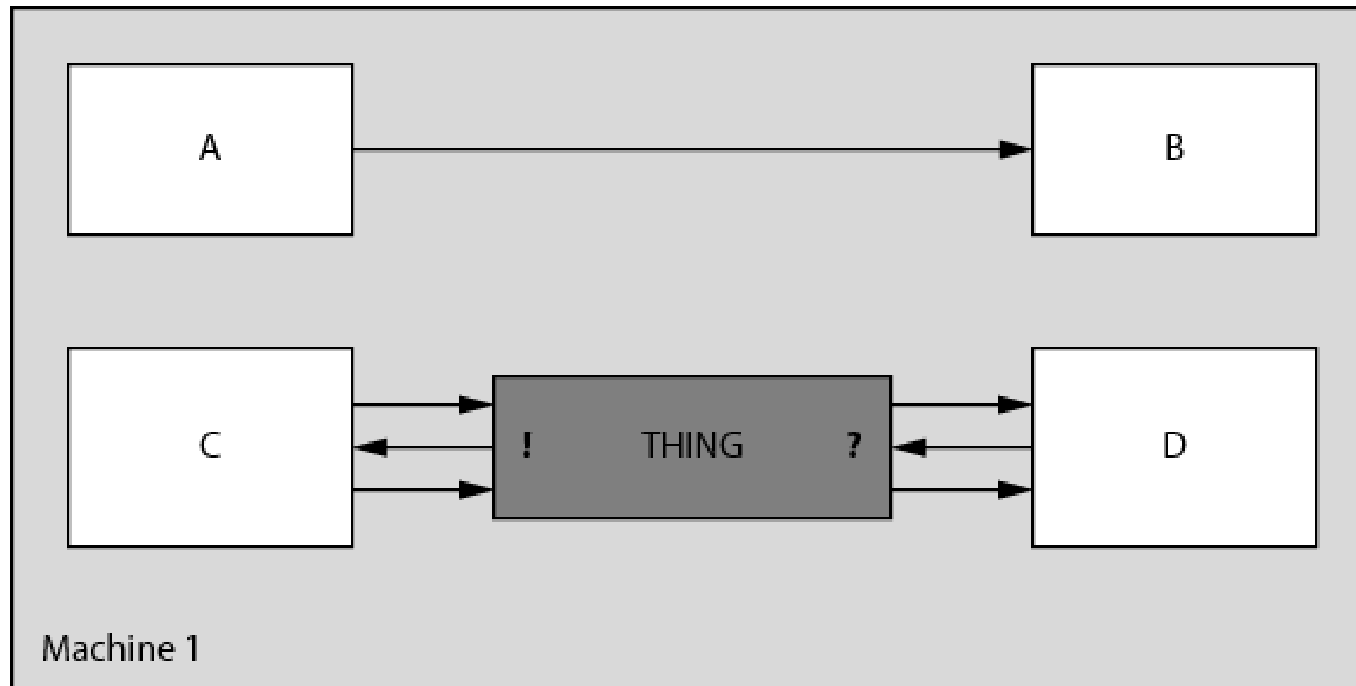
Network channel-types (NCTs)

- In distributed systems, it is natural to set up the communication ends first and then to connect them
- Channel-types: a good basis for networked CSP communication in occam
- “Simple” network channels are emulated by an NCT containing only one channel

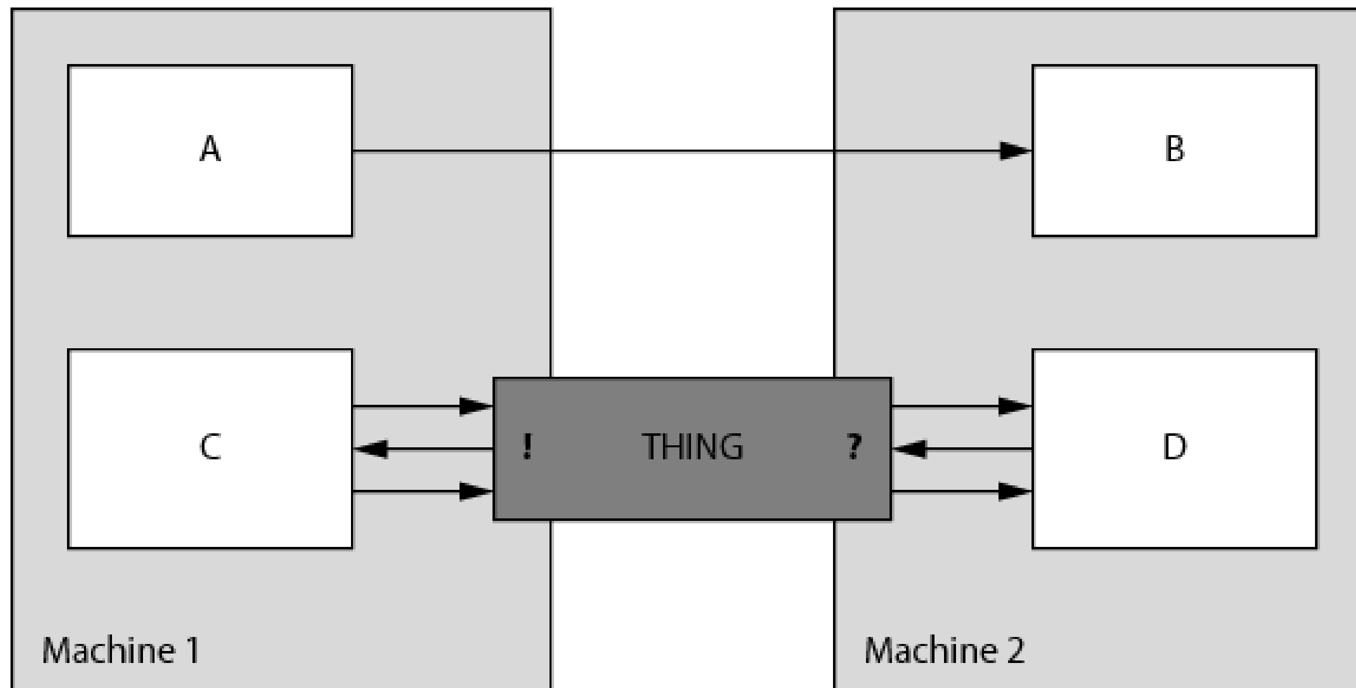
Basic Infrastructure

- We want transparency for the occam programmer
- Behaviour of channels and channel-types should be identical both locally and networked from the point of view of the processes connected via the channel/channel-type

Local Channels and Channel-types



Network Channels and Network Channel-types



New occam Features in KRoC.net

- New occam features have been used in KRoC.net:
  - Extended rendezvous
  - Generic Protocol Converters
- These features are needed for communication transparency

Extended Rendezvous

- Allows to extend channel synchronisation
- Network infrastructure can be ‘plugged’ in without losing the synchronisation between the ends

Extended Rendezvous

- Example:

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

Generic Protocol Converters

- Convert from any given occam **`PROTOCOL`** (including user-defined protocols) to a link protocol used by the network infrastructure and vice versa
- This gives us **`PROTOCOL`** transparency: channels of all protocols can now be networked

Generic Protocol Converters

- Link protocol
  - Address/size pair (just like a transputer)
  - Consists of a pointer to the data item and its size
  - Prevents unnecessary copying of data within the network infrastructure

Generic Protocol Converters

- Two compiler built-in **PROC**s:

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

- 'Asterisk' protocols:
  - **`*`** application protocol
  - **`**`** termination signal (**`INT`** or **`BOOL`**)
  - **`***`** link protocol

Generic Protocol Converters

- Decoding of **`PROTOCOL`**s :
  - **`DECODE.CHANNEL`** outputs one or multiple (for certain **`PROTOCOL`**s, e.g. sequential protocols) address/size pairs
  - Data can then be copied to the remote machine

Generic Protocol Converters

- Encoding of **`PROTOCOL`**s :
  - Network infrastructure receives data and stores it in a dynamic **`MOBILE BYTE`** array
  - **`ENCODE.CHANNEL`** converts from the address/size of the array(s) into the appropriate application protocol
  - Data is either copied or moved into the application, depending on the protocol

Communication Over Network Channels

- Identical for ‘plain’ network channels and network channels inside an NCT
- Communication handled by the KRoC.net manager
- Each end of a network channel/NCT communicates with its instance of the KRoC.net manager, who then communicates over the network with the remote KRoC.net manager

The KRoC.net Manager

- Runs on each machine with networked channels
- Runs in parallel with the application level processes
- Handles setup of NCTs and communication over network channels
- Modular design, therefore easily extensible

The KRoC.net Manager

- KRoC.net manager has a front-end and a back-end
- Front-end handles the application side of NCTs and network channels:
  - Output Control Process (OCP) or Input Control Process (ICP) for each network channel-end
  - Server Control Process (SCP) or Client Control Process (CCP) for each NCT end

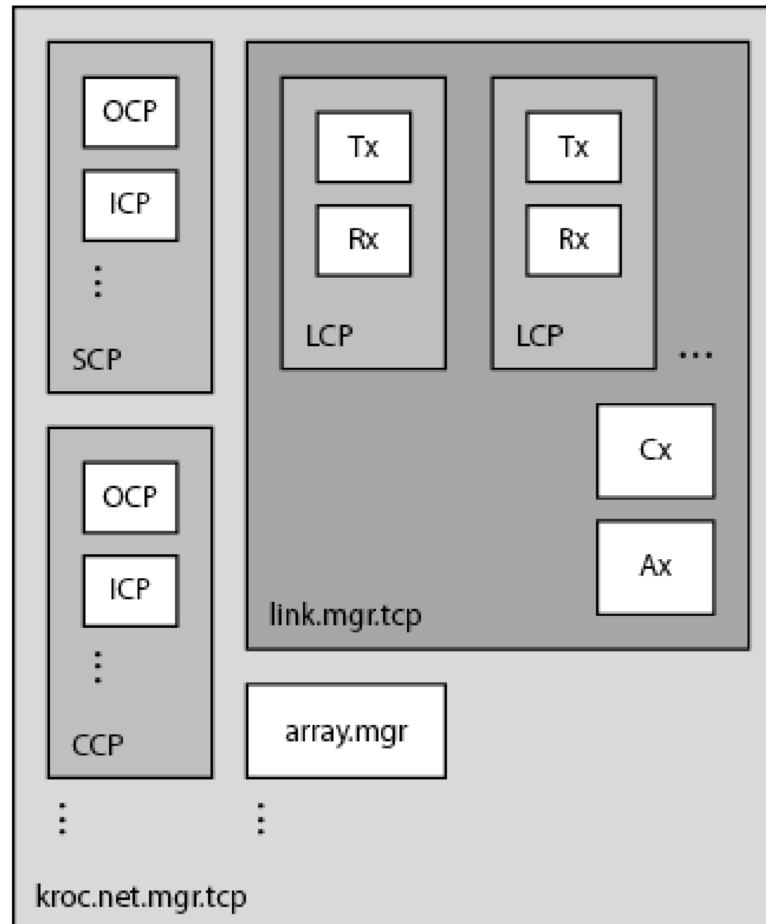
The KRoC.net Manager

- Back-end deals with network communication:
  - Separate module inside the KRoC.net manager: link manager handles network links between remote machines
  - Each link handled by a Link Control Process (LCP)
  - Link manager can be exchanged without need to change the front-end parts
  - Currently only TCP/IP networks supported, but writing a link manager for other types of networks is trivial
  - LCP for TCP contains Tx/Rx processes for each link

The KRoC.net Manager

- Multiplexing:
  - Multiple network channels between two machines are multiplexed over one network link
  - Front-end and back-end processes communicate over a ‘cross bar’, similar to JCSP.net
  - Ends of channel-types connected to the front-end and back-end components are stored in special arrays (which can be extended if need be)
  - Arrays are handled by special array manager processes

The KRoC.net Manager



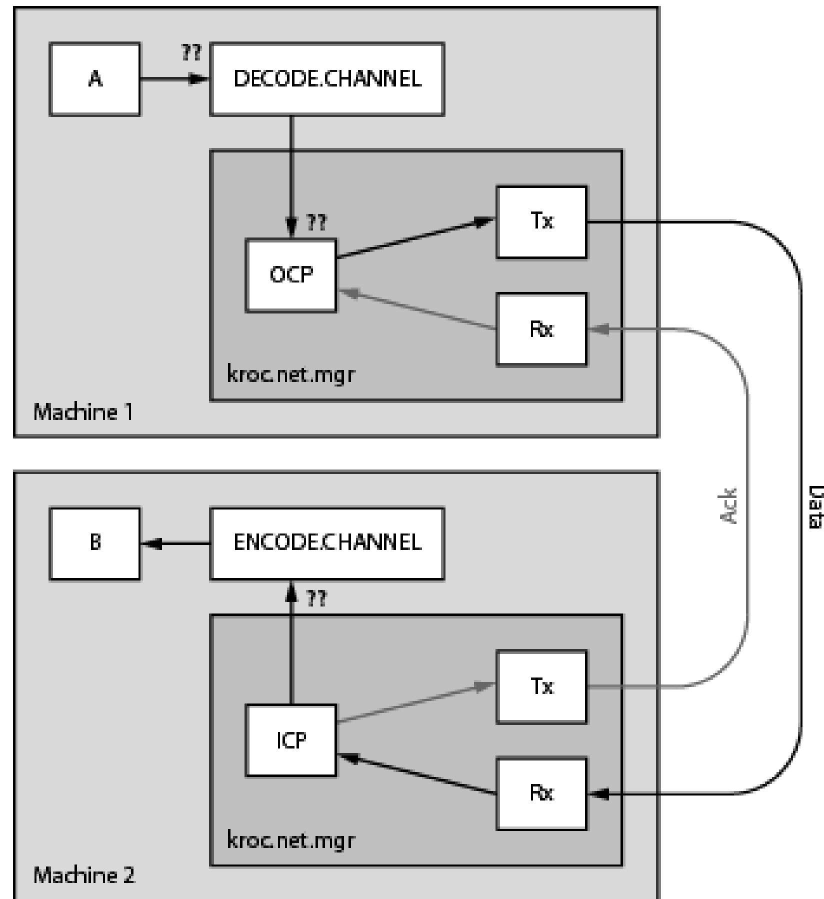
Communication Over Network Channels

- Ends of network channels are identified by a unique Output Control Number or Input Control Number
- Communication between two machines is multiplexed over the same network link, Control Numbers used as ‘local address’ to find the correct Output/Input Control Process to communicate with (via its index in the array)
- Remote Control Number sent together with the data or the acknowledgement

Communication Over Network Channels

- Generic Protocol Converters plugged between application level process and KRoC.net manager
- Extended Rendezvous used to 'extend' the communication:
  - Data is stored in the ICP until read by the application
  - Writing-end is blocked until network acknowledgement arrives
  - Preserving CSP synchronisation semantics
- Communication over network channels fully transparent to the application level process

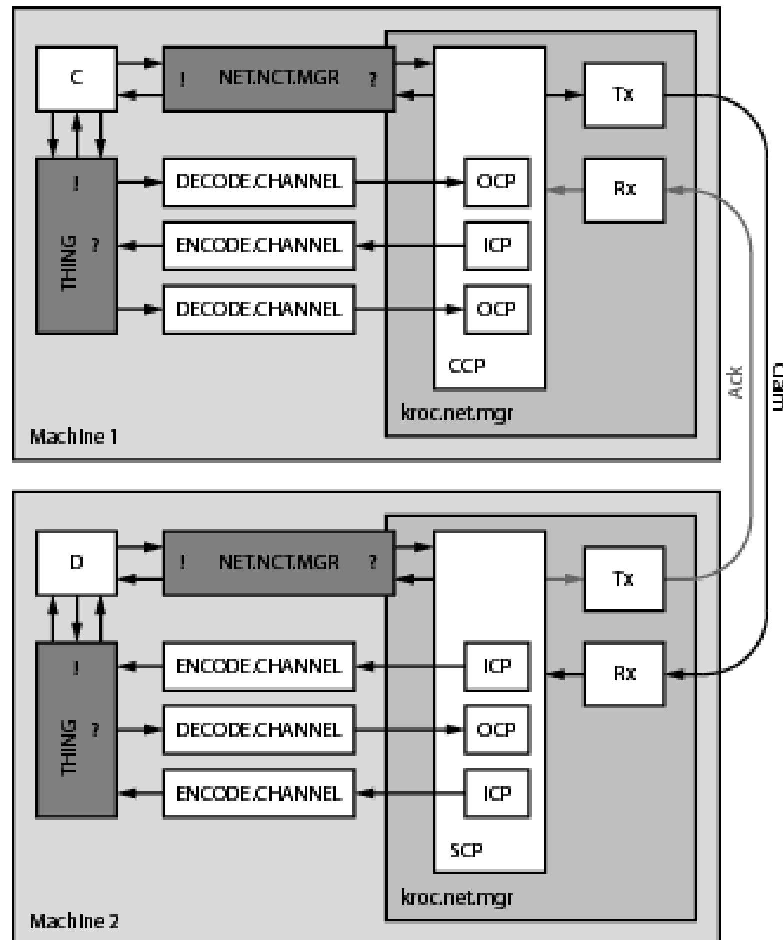
Communication Over Network Channels



Setting up NCTs

- Rather complex for the user at the moment
- Details in the paper
- All this ‘bootstrapping’ code will be hidden behind new occam language constructs later to achieve full transparency also for the setup of NCTs and the administration (e.g. claiming, moving) of their ends

Setting up NCTs



Setting up NCTs

- Setup: establishing the two ends of an NCT and connecting them
- Client-end connects to the server-side via a special URL that describes the location of the server-end:

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

Setting up NCTs

- **`<server-name>`** is a name given to the server-end
- Could be a simple name like "**`fred`**" or a structured name like "**`my-application/fred`**"
- Server-end is identified by this given name

Setting up NCTs

- Three ways of connecting an NCT:
  - Via the Channel Name Server
  - Directly
  - Anonymously

The Channel Name Server (CNS)

- Central server storing the locations of NCT ‘server’-ends
- Can be millions of entries, even of completely different applications:
  - In this case structured names are sensible
- ‘Server’-end registered with the CNS under its name
- ‘Client’-end looks up that ‘server’-end at the CNS

The Channel Name Server (CNS)

- Which CNS to use?
  - Default CNS:
    - Suffix in URL omitted

      `nct:fred`
  - Non-default CNS:
    - Suffix "**`@cns:<cns-name>`**"

      `nct:fred@cns:my-cns`
  - Directly, using to the location of the CNS:
    - Suffix "**`@cns.<net-type>:<location>`**"

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

Connecting an NCT Directly

- Name of the server-end is stored at its machine, not registered with the CNS:
  - Suffix “**`@direct`**” to set up the server-end

    `nct:fred@direct`
- Client-end ‘looks up’ the server-end, via its name, directly at its machine:
  - Suffix “**`@direct.<net-type>:<location>`**”

    `nct:fred@direct.tcp:gaia.kent.ac.uk:4500`

Connecting an NCT Anonymously

- Server-end is created anonymously, i.e. without a name
- KRoC.net manager returns an 'anonymous URL' which describes the location of the server-end:
  - URL "`nct:$<scn>@direct.<net-type>:<location>`"

    `nct:$5@direct.tcp:gaia.kent.ac.uk:4500`
- Anonymous URL is passed on to the client-end side
- Client-end can then be connected via this URL

Connecting an NCT Anonymously

- Possible use:
  - Connecting to a ‘published’ server-end
  - Server spawns off a worker process, creates an anonymous NCT and tells client its location
  - Client can then establish a ‘private’ connection to the worker process, server can deal with other clients in the meantime
- Anonymous NCTs will become obsolete when the movement of NCTs is implemented (just as it works now for local channel-types already)

Proposals for an Extended occam Syntax

- Setup process rather complex for the user at the moment
- KRoC will be extended to hide the KRoC.net setup code behind occam language constructs
- Ideas for an extended syntax: new "**`NET`**" keyword

Proposals for an Extended occam Syntax

- Dynamic construction of 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"
```

Proposals for an Extended occam Syntax

- Setting up one of many possible client-ends of 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
```

Proposals for an Extended occam Syntax

- Reading-end of a ‘simple’ one-to-one channel:

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

- The writing-end on another machine:

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

Proposals for an Extended occam Syntax

- Shared writing-end of a ‘simple’ any-to-one channel:

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

- The reading-end on another machine:

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

Configuration of KRoC.net

- Settings required:
  - Location (i.e. IP address and port number for TCP) of the own machine
  - Location of the CNS

Configuration of KRoC.net

- File "**`.kroc.net`**" for the location of the own machine:

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

- i.e. for TCP:

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

- Location is used as identifier for the machine

Configuration of KRoC.net

- File "`.self.cns`" used by the CNS itself
  - Same style, but without IP address, as the location of the CNS is not part of an identifier
- File "`.cns`" used by the the application, contains the location of the default CNS
  - For TCP: IP address or host name, and port number
- File "`.cns.<cns-name>`" may contain the location of a non-default CNS
  - "`<cns-name>`" is the name of the CNS used in the "`@cns:<cns-name>`" part of the URL

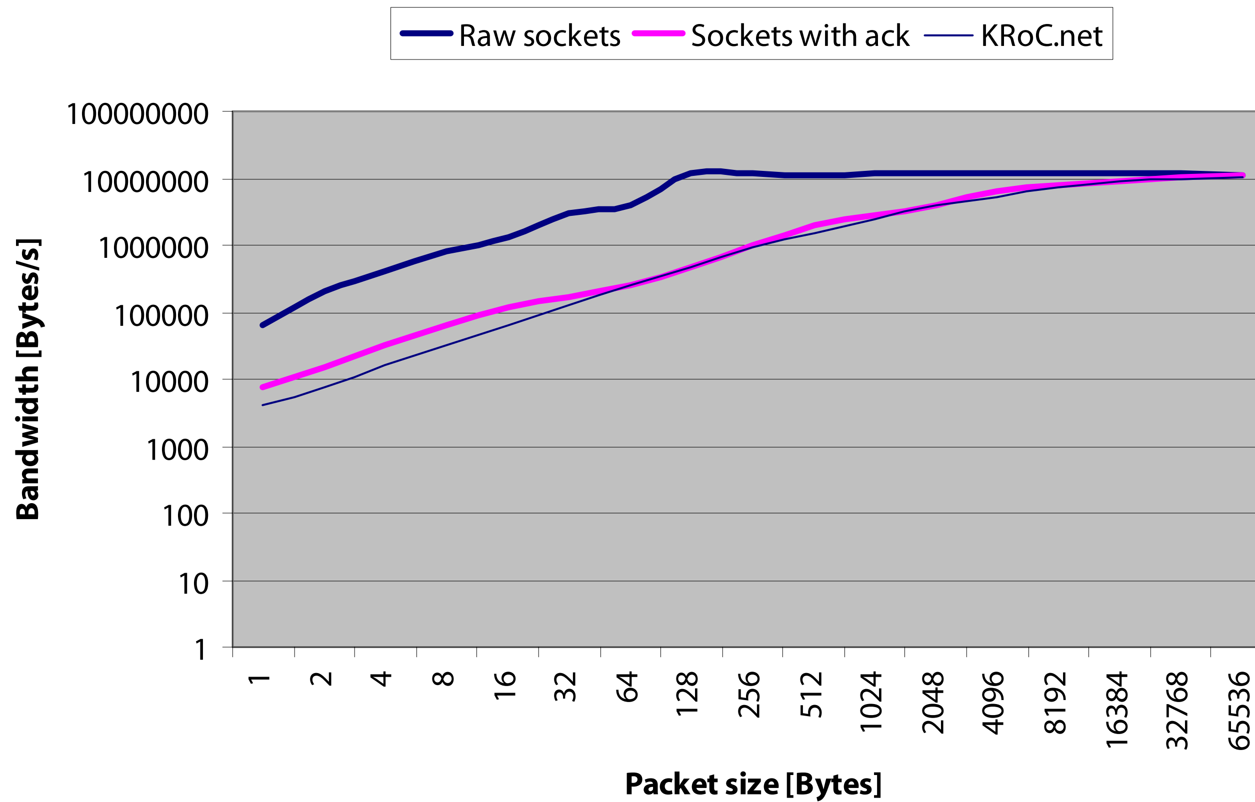
Performance of KRoC.net

- Benchmarking system:
  - Sender on **`gaia`** (1GHz Pentium III)
  - Receiver on **`catch22`** (2.4GHz Pentium 4)
  - Connected by 100MBit/s ethernet

Bandwidth Measurement

- Sender sends a sequence of equal-sized packets over a network channel to the receiver
- Repeated for sizes from 1 Byte to 64 KBytes, doubling each time
- Compared against raw sockets and sockets with acknowledgement

Bandwidth Measurement



Bandwidth Measurement

- Comparison with ‘sockets with ack’ shows small overhead of KRoC.net
- About half the bandwidth of ‘sockets with ack’ for 1 Byte packets
  - Due to separate sending of packet size and data
  - We are currently experimenting with a **`WRITEV`** implementation to avoid that
  - Comparison of KRoC.net to a modified ‘sockets with ack’ version that also uses two communications gives practically the same results

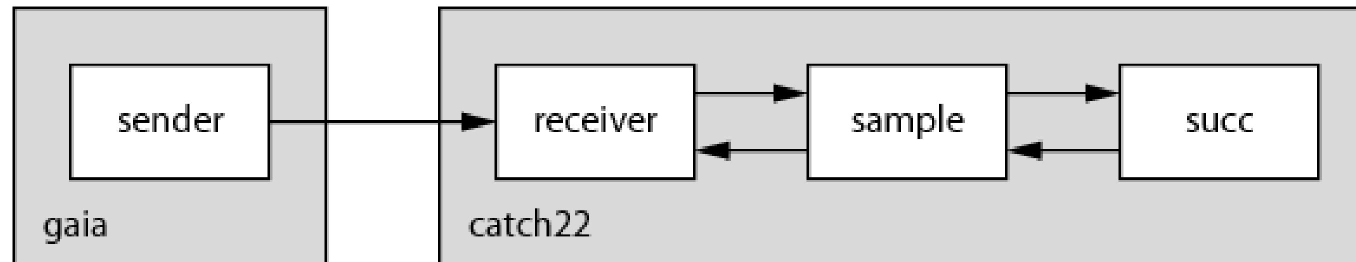
Bandwidth Measurement

- KRoC.net's ping-pong time (i.e. time between sending data and receiving acknowledgement, excluding network communication) is 1.9μs on **`gaia`**
  - Well within the error range of about 10% (about 20μs) of the benchmarks
  - Overheads negligible
- High impact of socket communication due to OS system calls and OS-level context switches
  - RMoX implementation will give better performance

Load Measurement

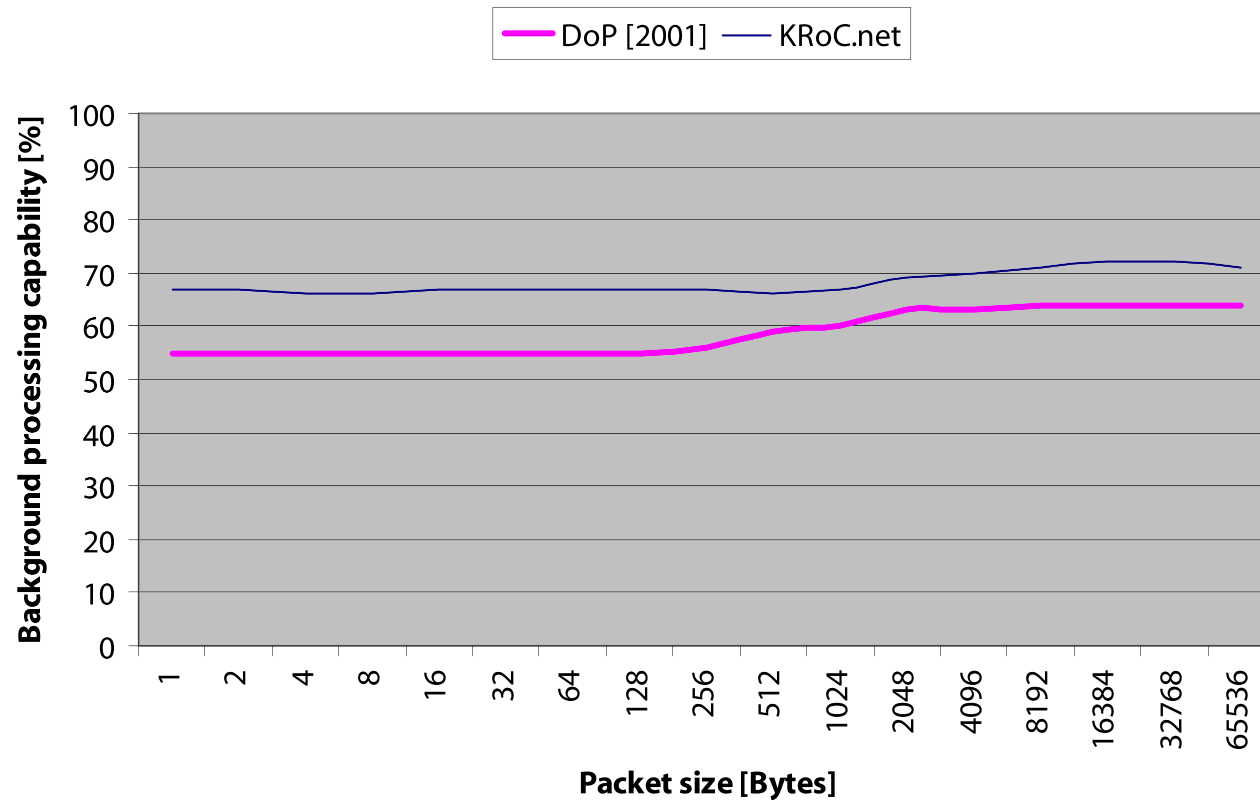
- Measuring the load of KRoC.net against other processes running in parallel
- Modified benchmark setup:
  - Sender stays the same
  - Receiver now runs in parallel with two other processes: **`sample`** and **`succ`**
    - Both processes are running infinitely at lowest occam process priority, i.e. they are always active when neither the receiver nor the KRoC.net infrastructure are active

Load Measurement



- `sample` passes a number to `succ` who increases it and sends it back
- The receiver can interrupt `sample` at any time to request the current count
- ‘Unloaded’ value is normalised to 100% background processing capability

Background Processing Capability



Load Measurement

- KRoC.net has a relatively low impact on processes running in the background
- Even slightly better that DoP, a predecessor from 2001, which is most likely due to the dynamic setup of KRoC.net:
  - KRoC.net's component processes are created 'on the fly' when they are needed
  - No copying of data involved, just address/size pairs

Conclusion

- KRoC.net has become very dynamic and flexible
- Easy and safe setup and use of network channels
- High level of transparency in terms of channel communication, but still work to do to make the setup process fully transparent

Future Work

- Achieve total transparency:
  - Automated dynamic setup, introducing new language constructs
  - Implement the moving of NCT ends
  - Hide the claiming of shared ends behind the existing occam **`CLAIM`** syntax
- Introduce proper error messages for cases when the network communication goes wrong

Future Work

- Reduce network latency
  - Introduce buffered channels
  - Introduce ping/pong style NCTs
  - Improve the Generic Protocol Converters for **`PROTOCOL`**s which involve more than one communication, so that we know how many communications are following for the same **`PROTOCOL`**
  - Using **`WRITEV`** to reduce the number of network communications where possible