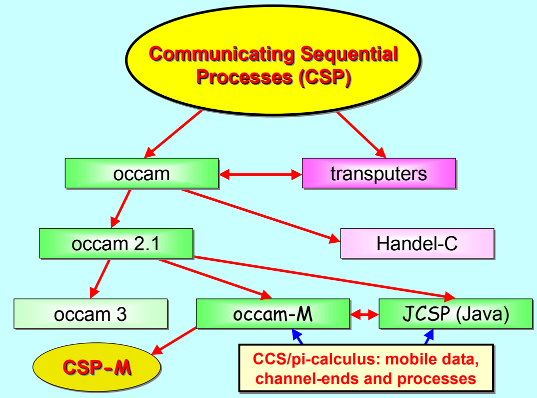
Communicating Processes, Safety and Dynamics: the New occam

Peter Welch and Fred Barnes
Computing Laboratory
University of Kent at Canterbury
{phw, frmb}@kent.ac.uk

IFIP WG 2.4, Dagstuhl, Germany (14th. November, 2002)



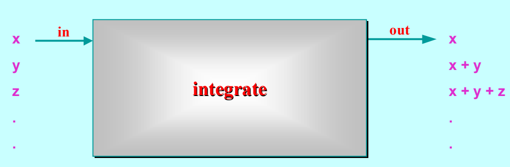
Dynamic occam

- **Introduction to Dynamic occam**
  - Motivation and Principles
- **Details**
  - Channel *Ends* and Direction Specifiers
  - Mobile Channel Structures (and `SHARED` Channels)
  - Dynamic Process Creation (`FORK`)
  - Extended Rendezvous
  - Process Priorities (32 levels now supported)
  - Extensions (parallel recursion, nested `PROTOCOL` definitions, …)
- **Examples**
  - Dynamic Process Farms
  - Intercepting Channel Communications
  - Networked Channels
  - **RMoX** and **occWeb**
- **Summary**

Motivation and Principles

- **Motivation**
  - Classical **occam** ←→ embedded systems; hence pre-allocated memory (i.e. compile-time defined concurrency limits, array sizes and no recursion). ***It's long been time to move on!***
  - Remove static constraints (***but retain as a voluntary option for use in hardware design and some embedded systems***).
  - Move towards general-purpose capability (***because occam is too good to keep to ourselves*** ☺).
- **Principles for changes/extensions**
  - they must be useful and easy to use;
  - they must be semantically sound and policed against misuse;
  - they must have very light implementation (***nano-memory and warp speed***);
  - they must be aligned with the core language (***no semantic, safety or performance disturbance***).

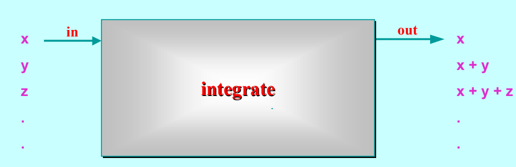
Channel *Ends* and Direction Specifiers




```
PROC integrate (CHAN INT in?, out!)
```

An **occam** process may only use a channel parameter *one-way* (either for input or for output). That direction is specified (? or !), along with the structure of the messages carried – in this case, simple `INT`s. The compiler checks that channel useage within the body of the `PROC` conforms to its declared direction.

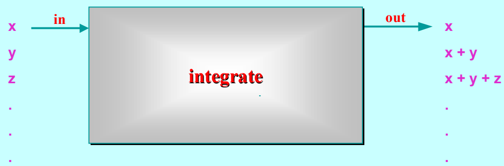
Channel *Ends* and Direction Specifiers




```
PROC integrate (CHAN INT in?, out!)
  INITIAL INT total IS 0:
  WHILE TRUE
    INT x:
    SEQ
      in ? x
      total := total + x
      out ! total
:
```

serial implementation

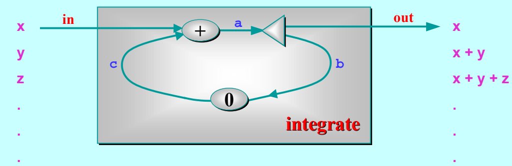
Channel *Ends* and Direction Specifiers




```
PROC integrate (CHAN INT in?, out!)
```

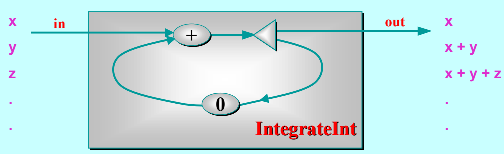
parallel implementation

Channel *Ends* and Direction Specifiers



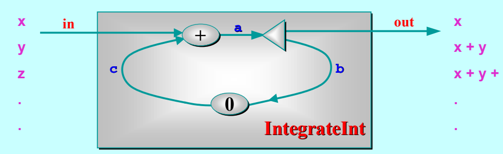

```
PROC integrate (CHAN INT in?, out!)
  CHAN INT a, b, c:
  PAR
    plus (in?, c?, a!)
    delta (a?, out!, b!)
    prefix (0, b?, c!)
:
```

parallel implementation



Java (JCSP)

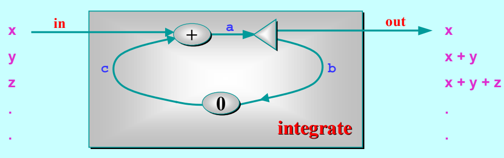
```
class IntegrateInt implements CSProcess {

    private final ChannelInputInt in;
    private final ChannelOutputInt out;

  public IntegrateInt (ChannelInputInt in,
                       ChannelOutputInt out) {
    this.in = in;
    this.out = out;
  }

  ...  public void run ()

}
```

Java (JCSP)

```
public void run () {

  One2OneChannelInt a = Channel.createOne2OneInt ();
  One2OneChannelInt b = Channel.createOne2OneInt ();
  One2OneChannelInt c = Channel.createOne2OneInt ();

  new Parallel (
    new CSProcess[] {
      new PlusInt (in, c.in(), a.out()),
      new Delta2Int (a.in(), out, b.out()),
      new PrefixInt (0, b.in(), c.out())
    }
  ).run ();

}
```


Channel *Ends* and Direction Specifiers




```
PROC integrate (CHAN INT in?, out!)
  CHAN INT a, b, c:
  PAR
    plus (in?, c?, a!)
    delta (a?, out!, b!)
    prefix (0, b?, c!)
:
```


Mobile Channel Structures


```
CHAN TYPE BUF.MGR
  MOBILE RECORD
    CHAN INT req?:              -- requested buffer size
    CHAN MOBILE []BYTE buf!:    -- delivered array
    CHAN MOBILE []BYTE ret?:    -- returned array
:
```

Channel types declare a *bundle* of channels that will always be kept together. They are similar to the idea proposed for **occam3**, except that the *ends* of our bundles are mobile …

Mobile Channel Structures


```
CHAN TYPE BUF.MGR
  MOBILE RECORD
    CHAN INT req?:              -- requested buffer size
    CHAN MOBILE []BYTE buf!:    -- delivered array
    CHAN MOBILE []BYTE ret?:    -- returned array
:
```

… and we also specify the *directions* of the component channels …

Mobile Channel Structures


```
CHAN TYPE BUF.MGR
  MOBILE RECORD
    CHAN INT req?:              -- requested buffer size
    CHAN MOBILE []BYTE buf!:    -- delivered array
    CHAN MOBILE []BYTE ret?:    -- returned array
:
```

… [channel *bundles*, like *atomic* channels, have two ends which we call, arbitrarily, the "**?**" (or "**server**") end and the "**!**" (or "**client**") end] …

Mobile Channel Structures


```
CHAN TYPE BUF.MGR
  MOBILE RECORD
    CHAN INT req?:              -- requested buffer size
    CHAN MOBILE []BYTE buf!:    -- delivered array
    CHAN MOBILE []BYTE ret?:    -- returned array
:
```

… the formal declaration indicates these directions from the viewpoint of the "**?**" end.

Mobile Channel Structures

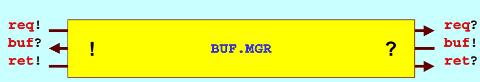
For these *mobile* channel types, variables are declared only for their *ends*. Those ends are going to be *independently* mobile – not the channel as a whole.

```
BUF.MGR! buf.cli:     -- "client"-end variable
BUF.MGR? buf.svr:     -- "server"-end variable
```

They are allocated in pairs *dynamically*:

```
buf.cli, buf.svr := MOBILE BUF.MGR
```


Mobile Channel Structures




```
buf.cli, buf.svr := MOBILE BUF.MGR
```

Those variables need to be given to separate parallel processes before it makes sense to use them – e.g:

```
MOBILE []BYTE b:
SEQ
  buf.cli[req] ! 42
  buf.cli[buf] ? b
  ...  use b
  buf.cli[ret] ! b
```

PAR

```
MOBILE []BYTE b:
INT s:
SEQ
  buf.svr[req] ? s
  b := MOBILE [s]BYTE
  buf.svr[buf] ! b
  buf.svr[ret] ? b
```


Mobile Channel Structures

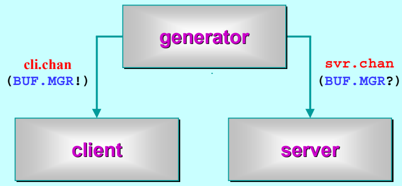

```
buf.cli, buf.svr := MOBILE BUF.MGR
```

However, it's more flexible (and fun) to take advantage of their *mobility*.

Mobile channel-end variables may be assigned to each other and sent down channels – strong typing rules apply, of course.

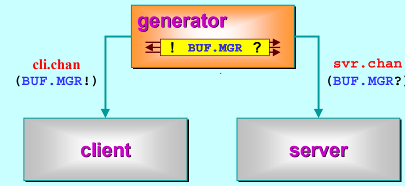
Recall, also, the basic rules of mobile assignment and communication: *once assigned or communicated* ***from****, the mobile variable becomes* ***undefined***. It may not be used again *until re-allocated, assigned or communicated* ***to***.

Mobile Channel Structures



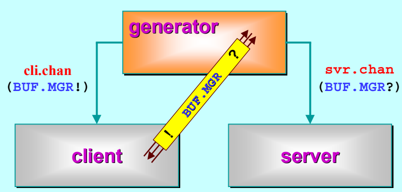

```
CHAN BUF.MGR! cli.chan:
CHAN BUF.MGR? svr.chan:
PAR
  generator (cli.chan! svr.chan!)
  client (cli.chan?)
  server (svr.chan?)
```


Mobile Channel Structures



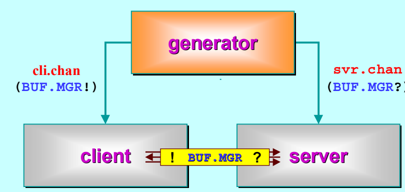

```
BUF.MGR! buf.cli:
BUF.MGR? buf.svr:
SEQ
  buf.cli, buf.svr := MOBILE BUF.MGR
```


Mobile Channel Structures



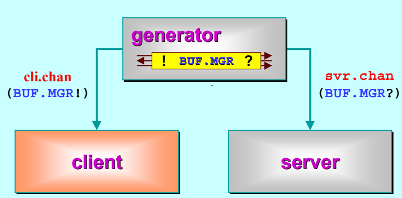

```
BUF.MGR! buf.cli:
BUF.MGR? buf.svr:
SEQ
  buf.cli, buf.svr := MOBILE BUF.MGR
  cli.chan ! buf.cli
```


Mobile Channel Structures



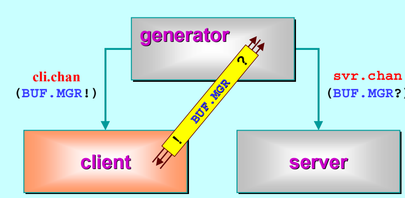

```
BUF.MGR! buf.cli:
BUF.MGR? buf.svr:
SEQ
  buf.cli, buf.svr := MOBILE BUF.MGR
  cli.chan ! buf.cli
  svr.chan ! buf.svr
  -- buf.cli and buf.svr are now undefined
```


Mobile Channel Structures



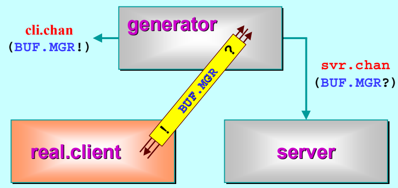

```
PROC client (CHAN BUF.MGR! cli.chan?)
  BUF.MGR! cv:
  SEQ
```


Mobile Channel Structures



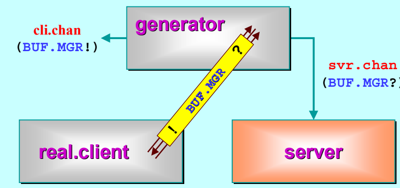

```
PROC client (CHAN BUF.MGR! cli.chan?)
  BUF.MGR! cv:
  SEQ
    cli.chan ? cv
```

Mobile Channel Structures



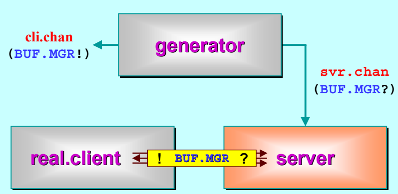

```
PROC client (CHAN BUF.MGR! cli.chan?)
  BUF.MGR! cv:
  SEQ
    cli.chan ? cv
    real.client (cv)
:
```


Mobile Channel Structures



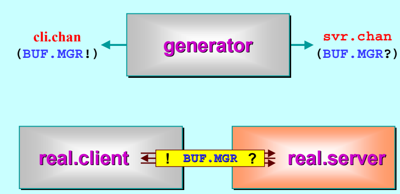

```
PROC server (CHAN BUF.MGR? svr.chan?)
  BUF.MGR? sv:
  SEQ
```


Mobile Channel Structures



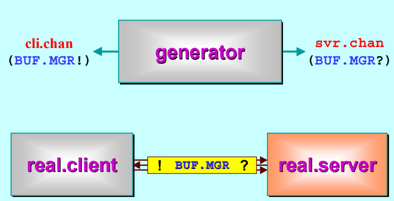

```
PROC server (CHAN BUF.MGR? svr.chan?)
  BUF.MGR? sv:
  SEQ
    svr.chan ? sv
```


Mobile Channel Structures



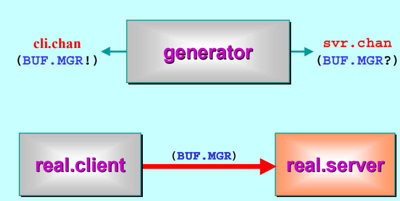

```
PROC server (CHAN BUF.MGR? svr.chan?)
  BUF.MGR? sv:
  SEQ
    svr.chan ? sv
    real.server (sv)
:
```


Mobile Channel Structures



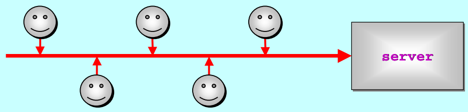

```
PROC real.client (BUF.MGR! call)
  ...
:

PROC real.server (BUF.MGR? serve)
  ...
:
```


Mobile Channel Structures




```
PROC real.client (BUF.MGR! call)
  ...
:

PROC real.server (BUF.MGR? serve)
  ...
:
```

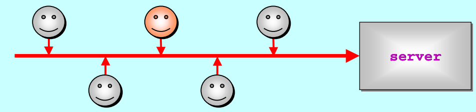
Shared Channel-Ends




```
SHARED BUF.MGR! s.buf.cli:    -- "client"-end variable
BUF.MGR? buf.svr:             -- "server"-end variable
SEQ
  s.buf.cli, buf.svr := MOBILE BUF.MGR
  PAR
    PAR i = 0 FOR n.clients
      client.2 (s.buf.cli)
    server (buf.svr)
```

`n.clients` may be computed at run-time

Shared Channel-Ends

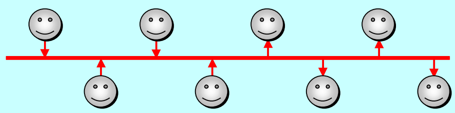



```
PROC client.2 (SHARED BUF.MGR! s.buf.cli)
  ...
  CLAIM s.buf.cli
    MOBILE []BYTE b:
    SEQ
      s.buf.cli[req] ! 42
      s.buf.cli[buf] ? b
      ...  use b
      s.buf.cli[ret] ! b
  ...
:
```

`s.buf.cli` may not be used outside of a `CLAIM` block

Only `s.buf.cli` channels may be used within its `CLAIM` block and no nested `CLAIMS`

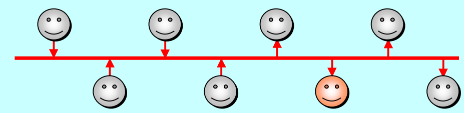
Both Ends Shared



```
SHARED BUF.MGR! s.buf.cli:    -- "client"-end variable
SHARED BUF.MGR? s.buf.svr:    -- "server"-end variable
SEQ
  s.buf.cli, s.buf.svr := MOBILE BUF.MGR
  PAR
    PAR i = 0 FOR n.clients
      client.2 (s.buf.cli)
    PAR i = 0 FOR n.servers
      server.2 (s.buf.svr)
```

`n.clients/servers` may be computed at run-time

Both Ends Shared

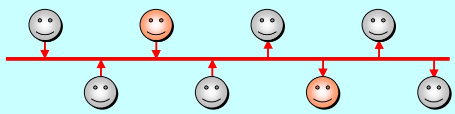


```
PROC server.2 (SHARED BUF.MGR? s.buf.svr)
  ...
  CLAIM s.buf.svr
    MOBILE []BYTE b:
    INT s:
    SEQ
      s.buf.svr[req] ? s
      b := MOBILE [s]BYTE
      s.buf.svr[buf] ! b
      s.buf.svr[ret] ? b
  ...
```

`s.buf.svr` may not be used outside of a `CLAIM` block

Other channels and nested *client* `CLAIMS` may be used within a *server* `CLAIM` block

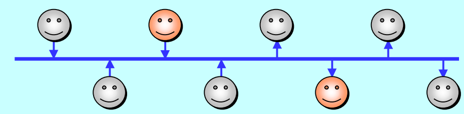
Both Ends Shared



PROBLEM: once a *client* and *server* process have made their claims, they can do business across the shared channel bundle. Whilst this is happening, all other *client* and *server* processes are locked out from the communication resource.

SOLUTION: use the shared channel structure just to enable *clients* and *servers* to find each other and pass between them a private channel structure. Then, let go of the shared channel and transact business over the private links.

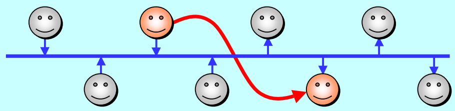
Both Ends Shared



```
CHAN TYPE CARRY.BUF.MGR
  MOBILE RECORD
    CHAN BUF.MGR? svr?:
:
```

Set up a similar network, but with the shared channel type being `CARRY.BUF.MGR` (rather than `BUF.MGR`).

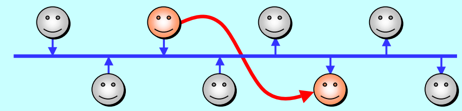
Both Ends Shared



```
CHAN TYPE CARRY.BUF.MGR
  MOBILE RECORD
    CHAN BUF.MGR? svr?:
:
```

A *client* process makes both ends of a non-shared BUF.MGR channel and *claims* the shared channel. When successful, it sends the *server-end* of its BUF.MGR down the shared channel. This blocks until a *server* process *claims* its end of the shared channel and inputs that *server-end*.

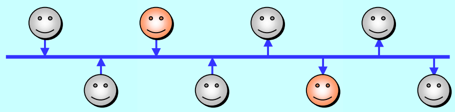
Both Ends Shared



```
CHAN TYPE CARRY.BUF.MGR
  MOBILE RECORD
    CHAN BUF.MGR? svr?:
:
```

Note that the *client* process, having output the *server* of its (unshared) BUF.MGR channel, no longer has that *server-end* and cannot use it or send it anywhere else. Only that *client* has the *client-end* and only the receiving *server* has the *server-end*.

Both Ends Shared



```
CHAN TYPE CARRY.BUF.MGR
  MOBILE RECORD
    CHAN BUF.MGR? svr?:
:
```

Once that *client* and *server* finish their business, the *server* should return the *server-end* of the BUF.MGR channel back to the *client*, who may then reuse it to send to someone else. With a slightly modified definition of BUF.MGR, its *server-end* may be sent back down itself to the *client*. ☺

Dynamic Process Creation

The PAR construct creates processes dynamically, but the creating process has to wait for them all to terminate before it can do anything else.

This is not always what we want! Many processes need to be able to *fork* off new processes (whose memory will need to be allocated at run-time) and carry on concurrently with them. Examples include web servers and operating systems.

But we are not operating a *free-for-all* heap in our new **occam** – strict aliasing control is maintained even for dynamically allocated structures. So, we must take care about memory referenced by long-lived *forked* processes.

Dynamic Process Creation

```
SEQ
  ...
  FORKING
    SEQ
      ...
      WHILE test
        SEQ
          ...
          ...
          FORK P (n, answer, in, out)
          ...
      ...
  ...
```

Can only FORK processes within a FORKING block

All FORKed processes must terminate before a FORKING block can terminate

Dynamic Process Creation

```
SEQ
  ...
  FORKING
    SEQ
      ...
      WHILE test
        SEQ
          ...
          ...
          FORK P (n, answer, in, out)
          ...
      ...
  ...
```

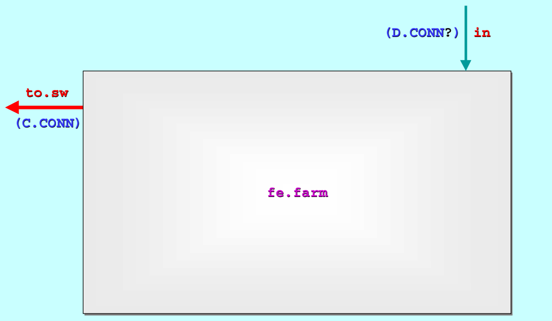
VAL data are ***copied*** into a FORKed process

MOBILE data and channel-ends are ***moved*** into a FORKed process

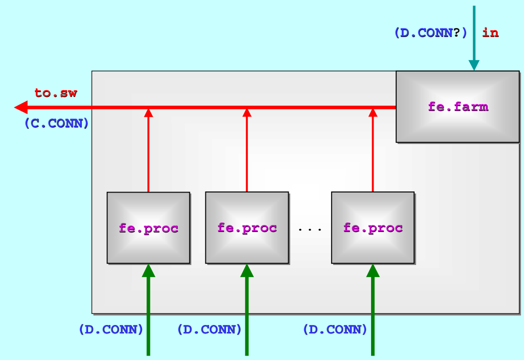
Reference data must be SHARED and declared ***global*** to the FORKING block

Otherwise, it may have ceased to exist before the FORKed process terminates

Dynamic Process Creation



Dynamic Process Creation



Dynamic Process Creation

```
PROC fe.farm (CHAN D.CONN? in?, SHARED C.CONN! to.sw)
  D.CONN? local:
  FORKING
    INITIAL INT c IS 0:
    WHILE TRUE
      SEQ
        in ? local
        FORK fe.proc (c, local, to.sw)
        c := c + 1
        ...
:
```

Outline of the front-end process farm handling incoming connections to the dynamic version of the **occam** web server.

```
PROC fe.proc (VAL INT n, D.CONN? in, SHARED C.CONN! to.sw)
  ...
:
```


Dynamic Process Farms (RMoX)

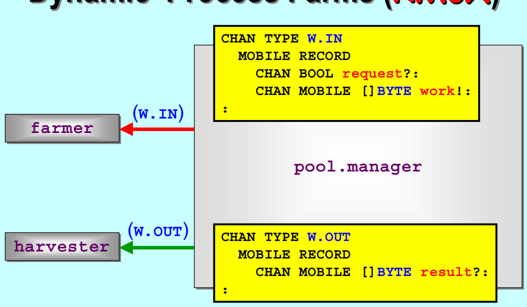
A `pool.manager` is responsible for a pool of `worker`s who queue up to request work packets from a `farmer`.

The `pool.manager` must ensure that at least `min.idle` `worker`s are always waiting to request new packets.

Each `worker` must keep the `pool.manager` informed as to whether it is *working* or *idle*. The `pool.manager` maintains a count of how many `worker`s are *idle* and `FORK`s off new ones as the need arises.

Of course, this means the number of `worker`s can never decrease – it can only ever keep growing. Limiting the number of *idle* `worker`s to `max.idle` is left as an exercise.

Dynamic Process Farms (RMoX)



Dynamic Process Farms (RMoX)

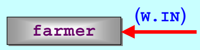
```
VAL INT min.idle IS ... :

SHARED W.IN! in.cli:
W.IN? in.svr:

SHARED W.OUT! out.cli:
W.OUT? out.svr:

SEQ

  in.cli, in.svr := MOBILE W.IN
  out.cli, out.svr := MOBILE W.OUT

  PAR
    farmer (in.svr)
    pool.manager (min.idle, in.cli, out.cli)
    harvester (out.svr)
```

create ***any-1*** channels

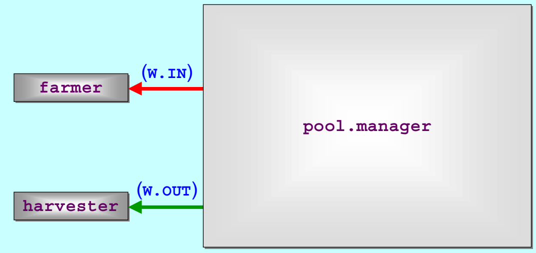
create network

Dynamic Process Farms (RMoX)

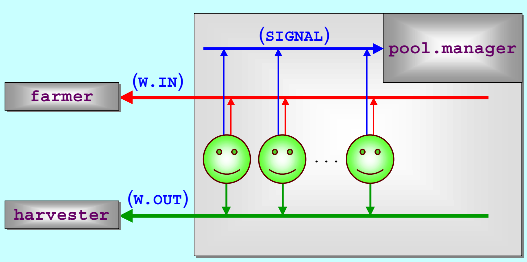
```
PROC farmer (W.IN? workers)
  WHILE TRUE
    MOBILE []BYTE packet:
    SEQ
      ...  manufacture work packet
      BOOL any:
      workers[request] ? any
      workers[work] ! packet
:

PROC harvester (W.OUT? workers)
  WHILE TRUE
    MOBILE []BYTE packet:
    SEQ
      workers[result] ? packet
      ...  consume result packet
:
```



Dynamic Process Farms (RMoX)



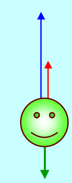
Dynamic Process Farms (RMoX)



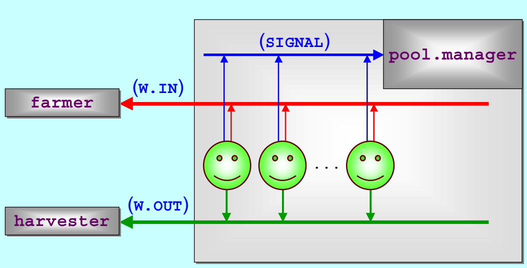

```
CHAN TYPE SIGNAL
  MOBILE RECORD
    CHAN INT count?:  -- working (-1) or idle (+1)
:
```


Dynamic Process Farms (RMoX)

```
PROC worker (SHARED W.IN! in,
             SHARED W.OUT! out,
             SHARED SIGNAL! signal)
  WHILE TRUE
    MOBILE []BYTE packet:
    SEQ
      CLAIM in
        SEQ
          in[request] ! TRUE
          in[work] ? packet
      CLAIM signal
        signal[count] ! -1     -- say we are working
      ...  do the work
      CLAIM out
        out[result] ! packet  -- hopefully, a modified one
      CLAIM signal
        signal[count] ! +1     -- say we are idle
:
```



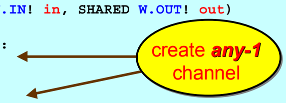
Dynamic Process Farms (RMoX)




```
CHAN TYPE SIGNAL
  MOBILE RECORD
    CHAN INT count?:  -- working (-1) or idle (+1)
:
```


Dynamic Process Farms (RMoX)

```
PROC pool.manager (VAL INT min.idle,
                   SHARED W.IN! in, SHARED W.OUT! out)

  SHARED SIGNAL! signal.cli:
  SIGNAL? signal.svr:

  SEQ
    signal.cli, signal.svr := MOBILE SIGNAL

    FORKING
      INITIAL INT n.idle IS 0:
      WHILE TRUE
        SEQ
          ...  (n.idle < min.idle) ==> FORK new workers
          INT n:
          SEQ
            signal.svr[count] ? n     -- working/idle (-1/+1)
            n.idle := n.idle + n
:
```



Dynamic Process Farms (RMoX)

```
{{{  (n.idle < min.idle) ==> FORK new workers
VAL INT needed IS min.idle - n.idle:
IF
  needed > 0
    SEQ
      SEQ i = 0 FOR needed
        FORK worker (in, out, signal.cli)
      n.idle := min.idle
  TRUE
    SKIP
}}}
```


Dynamic Process Farms (RMoX)

The dynamic management of process farms is one of the common design idioms used to support:

RMoX ("Raw Metal occam ix")

- an experimental operating system for general and real-time embedded applications, built exclusively on this extended **CSP** model and programmed (almost and eventually) entirely in **occam**.

Extended Rendezvous

This is a *convenience* – and it's free!

```
SEQ
  ...
  ...
  in ?? x
    ...  rendezvous block
  ...
  ...
```

wait for input ***but do not*** reschedule outputting process!

reschedule outputting process ***only after*** the rendezvous block has terminated

The outputting process is unaware of the **extended** nature of the rendezvous

Extended Rendezvous

They can be used as **ALT** guards:

```
ALT
  a ? x
    ...  react
  in ?? x
    ...  rendezvous block
    ...  react (optional and outside the rendezvous)
  tim ? AFTER timeout
    ...  react
```

guards

Extended Rendezvous

Here is an informal *operational* semantics:

```
c ! 42          --c-->   c ?? x
                           ...  rendezvous block
```

=

```
BOOL any:       --c-->   SEQ
SEQ                        c ? x
  c ! 42                   ...  rendezvous block
  c.ack ? any   <--c.ack-- c.ack ! TRUE
```

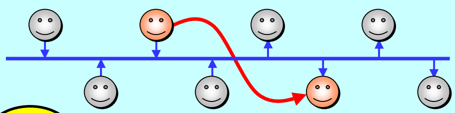

Extended Rendezvous

Not that it's implemented that way!

```
c ! 42          --c-->   c ?? x
                           ...  rendezvous block
```

- ***No additional overheads*** for normal channel communication.
- Implementation is very lightweight *(approx. 30 cycles)*:
  - ***no change*** in outputting process code;
  - new ***occam Virtual Machine*** (**oVM**) instructions for "`??`".
- Solves a long-standing semantic anomaly of unhandled tags in variant protocols:
  - `((d ! apple) || (d ? CASE banana)) = STOP`

Extended Rendezvous Taps

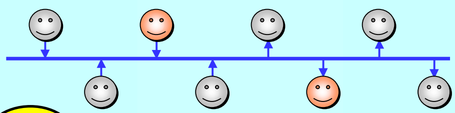


recall

```
CHAN TYPE CARRY.BUF.MGR
  MOBILE RECORD
    CHAN BUF.MGR? svr?:
:
```

A *client* process makes both ends of a non-shared `BUF.MGR` channel and *claims* the shared channel. When successful, it sends the *server-end* of its `BUF.MGR` down the shared channel. This blocks until a *server* process *claims* its end of the shared channel and inputs that *server-end*.

Extended Rendezvous Taps

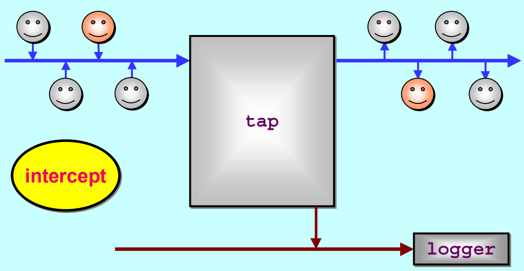


recall

```
CHAN TYPE CARRY.BUF.MGR
  MOBILE RECORD
    CHAN BUF.MGR? svr?:
:
```

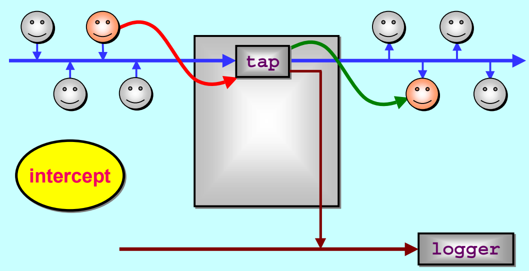
Once that *client* and *server* finish their business, the *server* should return the *server-end* of the `BUF.MGR` channel back to the *client*, who may then reuse it to send to someone else. With a slightly modified definition of `BUF.MGR`, its *server-end* may be sent down itself back to the *client*. ☺

Extended Rendezvous Taps



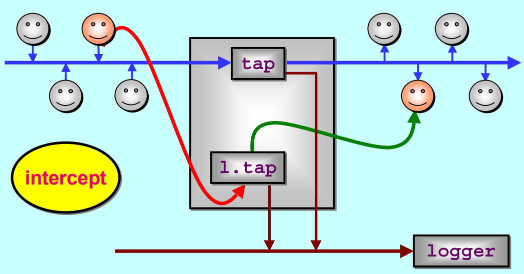
Note: *client* and *server* processes are unchanged.

Extended Rendezvous Taps



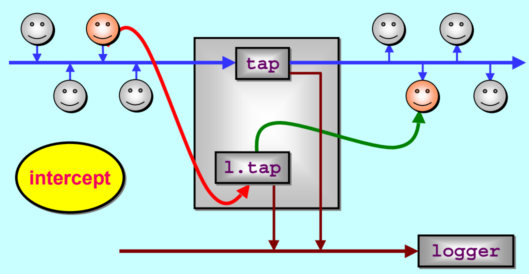
Intercept the sent `BUF.MGR?` and forward our own.

Extended Rendezvous Taps



`FORK l.tap` process and plug in loose ends.

Extended Rendezvous Taps



client and *server* processes cannot detect the taps.

Extended Rendezvous Taps

```
PROC tap (CARRY.BUF.MGR? in, out, SHARED LOG! log)
  FORKING
    WHILE TRUE
      BUF.MGR? client.svr, tap.svr
      BUF.MGR! tap.cli
      SEQ
        tap.cli, tap.svr := MOBILE BUF.MGR
        in[svr] ?? client.svr
          out[svr] ! tap.svr
        FORK l.tap (client.svr, tap.cli, log)
:

PROC l.tap (BUF.MGR? in, BUF.MGR! out, SHARED LOG! log)
  PAR
    ...  tap the req channel
    ...  tap the buf channel
    ...  tap the ret channel
:
```


Extended Rendezvous Taps

```
PROC l.tap (BUF.MGR? in, BUF.MGR! out, SHARED LOG! log)
  PAR
    ...  tap the req channel
    ...  tap the buf channel
    ...  tap the ret channel
:
```


Extended Rendezvous Taps

```
PROC l.tap (BUF.MGR? in, BUF.MGR! out, SHARED LOG! log)
  PAR
    {{{  tap the req channel
    WHILE TRUE
      BOOL b:
      in[req] ?? b
        out[req] ! b
        CLAIM log
          log[report] ! request; b
    }}}
    ...  tap the buf channel
    ...  tap the ret channel
:
```


Extended Rendezvous Taps

```
PROC l.tap (BUF.MGR? in, BUF.MGR! out, SHARED LOG! log)
  PAR
    ...  tap the req channel
    ...  tap the buf channel
    ...  tap the ret channel
:
```


Extended Rendezvous Taps

```
PROC l.tap (BUF.MGR? in, BUF.MGR! out, SHARED LOG! log)
  PAR
    ...  tap the req channel
    {{{  tap the buf channel
    WHILE TRUE
      MOBILE []BYTE b:
      out[buf] ?? b
        in[buf] ! b
        CLAIM log
          log[report] ! supplied; SIZE b
    }}}
    ...  tap the ret channel
:
```

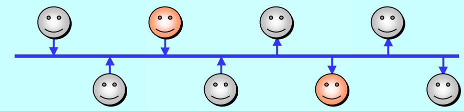

Extended Rendezvous Taps

```
PROC l.tap (BUF.MGR? in, BUF.MGR! out, SHARED LOG! log)
  PAR
    ...  tap the req channel
    ...  tap the buf channel
    ...  tap the ret channel
:
```

Extended Rendezvous Taps

```
PROC l.tap (BUF.MGR? in, BUF.MGR! out, SHARED LOG! log)
  PAR
    ...  tap the req channel
    ...  tap the buf channel
    {{{  tap the ret channel
    WHILE TRUE
      MOBILE []BYTE b:
      in[ret] ?? b
        out[ret] ! CLONE b
        CLAIM log
          log[report] ! returned; b
    }}}
:
```

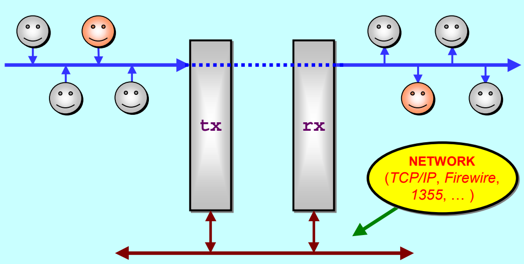

Networked Channel Structures



```
CHAN TYPE CARRY.BUF.MGR
  MOBILE RECORD
    CHAN BUF.MGR? svr?:
:
```

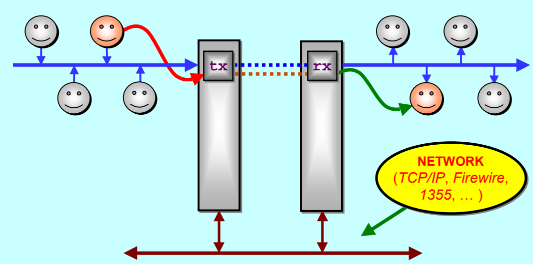
Back to the original design … but this time, we want to stretch the shared (`CARRY.BUF.MGR`) channel over some communication network without changing the semantics of the system.

Networked Channel Structures



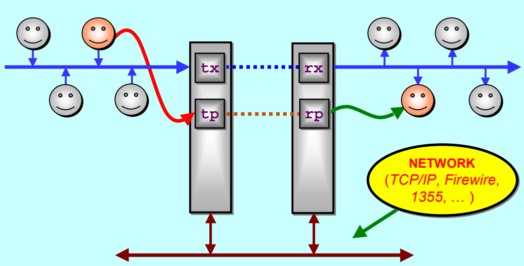
Note: *client* and *server* processes are unchanged …

Networked Channel Structures



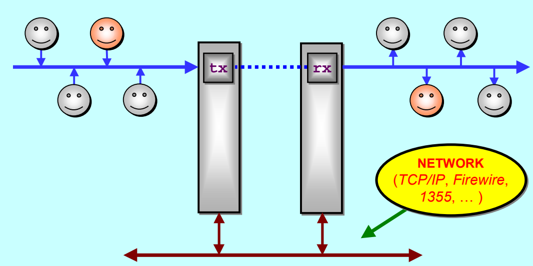
Note: *client* and *server* processes are unchanged …

Networked Channel Structures



… and still detect no change in system semantics.

Networked Channel Structures



… and still detect no change in system semantics.

Networked Channel Structures

To set this up, the **KRoC** programmer (designer) only constructs the ***named network channel structure*** – the processes supporting the network are automatically forked and have no impact on system semantics.

Process Priority

- **Currently, support for 32 levels of priority (0 = *highest*)**
- **Priorities are dynamic (not using `PRI PAR`)**
  - but a process may only change ***its own*** priority;
  - which enables ***very low unit time*** overheads.
- **Currently, priorities set by library routines:**

```
PROC SETPRI (VAL INT p.absolute)
PROC RELPRI (VAL INT p.relative)
PROC INCPRI (VAL INT p.up)
PROC DECPRI (VAL INT p.down)
```

- **A process may discover its own priority:**

```
INT FUNCTION GETPRI ()
```

- `GETPRI` **does not damage the *referential tranparency* of occam expressions.**

Process Priority

- **Pre-emption by a (newly ready) higher priority process takes place only at the next scheduling point:**
  - blocked synchronisation (e.g. on a channel);
  - waiting for a timeout;
  - loop-end.
- ***"Immediate"*** **pre-emption is possible – but with higher overheads …**
- ***Micro*-benchmarks (*800 MHz. Pentium III*) show:**
  - channel communication: 52 ns (no priorities) → 75 ns (priorites);
  - process (startup + shutdown): 28 ns (without) → 67 ns (priorites);
  - change priority (up /\ down): 63 ns;
  - independent of number of processes and priorities used.

Additional occam Extensions

- **`STEP` size in replicators**
- **Fixing the transputer `PRI ALT` bug**
  - Reversing the `ALT` disable sequence (as done by ***JCSP***)
- **(`PRI`) `ALT`, `SKIP` guards and pre-conditions**
- **Run-time computed `PAR` replicators**
- **Parallel Recursion**
- **`RESULT` Parameters and Abbreviations**
- **Nested `PROTOCOL` Definitions**
- **In-line Array Constructors**
- **Anonymous Channel Types**
  - e.g: `SHARED CHAN BYTE screen!`

Summary

- **Everything available in KRoC 1.3.3 ☺ ☺ ☺**
  - GPL (and some L-GPL) open source
  - `http://www.cs.ukc.ac.uk/projects/ofa/kroc/`
- **occam is now directly applicable to a wide range of industrial/commercial practice:**
  - embedded systems, safety-critical, real-time (of course) …
  - operating systems (**RMoX**), web servers (**occWeb**) …
  - web farms, e-commerce, Internet and parallel computing …
- **Working on:**
  - **KRoC** Network Edition (Mario Schweigler)
  - mobile processes (that carry state)
  - graphics/GUIs (again!)
- **Can someone come up with a really good name?!!**

URLs

CSP www.comlab.ox.ac.uk/archive/csp.html

JCSP www.cs.ukc.ac.uk/projects/ofa/jcsp/

CTJ www.rt.el.utwente.nl/javapp/

KRoC www.cs.ukc.ac.uk/projects/ofa/kroc/

java-threads@ukc.ac.uk

www.cs.ukc.ac.uk/projects/ofa/java-threads/

WoTUG

www/wotug.org/

Stop Press

JCSP Networking Edition
KRoC Commercial Support

JCSP.net

KRoC

www.quickstone.com

Stop Press

To get the *dynamic* capabilities presented in this talk, you need KRoC 1.3.3 or later.

The current (Linux/x86) on the KRoC website (www.cs.ukc.ac.uk/projects/ofa/kroc/) is 1.3.2. Pre-releases of 1.3.3 are available from the occam webserver pages (wotug.ukc.ac.uk/ocweb/), which links off the KRoC site.

Raw Metal occam iX: (RMoX)

Peter Welch and Fred Barnes
Computing Laboratory
University of Kent at Canterbury
{frmb2, phw}@ukc.ac.uk

Next Time ???

Stop Press

A boot image of the RMoX demonstrator is available from the occam webserver pages (wotug.ukc.ac.uk/ocweb/), which links off the KRoC site.

To switch between the demo applications, use the *Function* keys, F1 through F6.