Communicating Mobile Processes

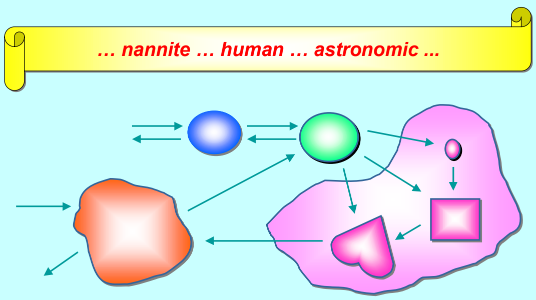
Peter Welch and Fred Barnes
Computing Laboratory
University of Kent at Canterbury
{phw, frmb}@kent.ac.uk

IFIP WG 2.4, Santa Cruz, U.S.A. (7th. August, 2003)

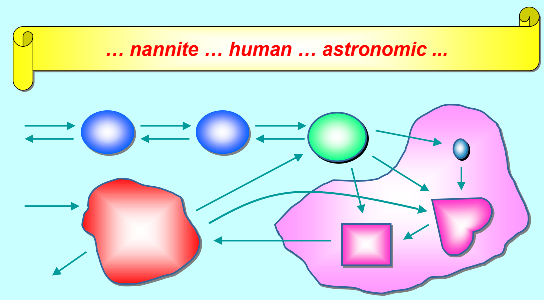
Communicating Mobile Processes

- **Introduction**
  - Motivation and Applications
  - CSP and occam-M
  - Mobility and location / neighbour awareness
  - Simplicity, dynamics, performance and safety
- **occam-M**
  - Processes, channels, (`PAR`) networks and (`ALT`) choice
  - Mobile data types - review
  - Mobile process types - new
  - Mobile channel types - review
  - Performance
- **Some applications**
  - Operating and field-programmable embedded systems (**RMoX**)
  - **In-vivo ⇔ In-silico** modelling (UK *'Grand Challenge'* 3)
- **Summary**

Nature has very large numbers of independent agents, interacting with each other in regular and chaotic patterns, at all levels of scale:

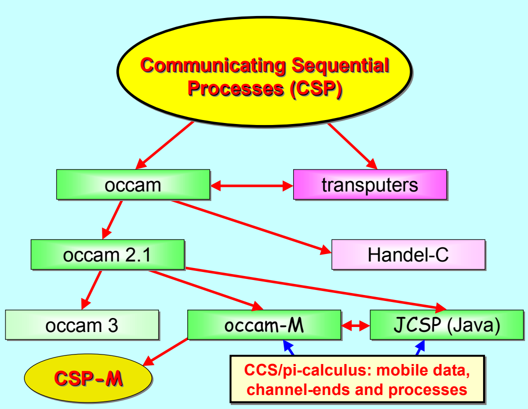


The networks are dynamic: growing, decaying and mutating internal topology (in response to environmental pressure and self-motivation):



Motivation and Applications

- **Thesis**
  - Natural systems are robust, efficient, long-lived and continuously evolving. *We should take the hint!*
  - Look on concurrency as a *core design mechanism* – not as something difficult, used only to boost performance.
- **Some applications**
  - Hardware design and modelling.
  - *Static* embedded systems and classical parallel supercomputing.
  - *Field-programmable* (or *evolving*) embedded systems and dynamic supercomputing (e.g. **SETI-at-home**).
  - Operating systems and games.
  - Biological system and *nannite* modelling.
  - eCommerce and business processes.



Mobility and Location Awareness

- **Classical communicating process applications**
  - Static network structures.
  - Static memory / silicon requirements (pre-allocated).
  - Great for hardware design and software for embedded controllers.
  - Consistent and rich underlying theory – CSP.
- **Dynamic communicating processes – some questions**
  - Mutating topologies – how to keep them safe?
  - Mobile channel-ends and processes: dual notions?
  - Intuitive operational semantics (and, hence, implementation)?
  - Process algebra theory: extend CSP or go for the *pi-calculus*?
  - Location awareness: how can mobile processes know where they are, how can they find each other and link up?
  - Programmability: at what level – individual processes or clusters?
  - Overall behaviour: planned or emergent?

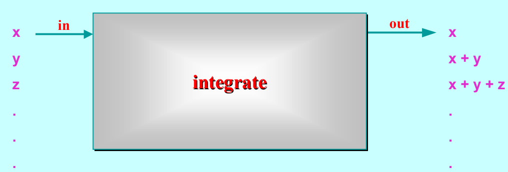
Requirements and Principles

- **Simplicity**
  - There must be a consistent (*denotational*) semantics that matches our intuitive understanding for *Communicating Mobile Processes*.
  - There must be as direct a relationship as possible between the formal theory and the implementation technologies to be used.
  - Without the above link (*e.g. using C++/posix or Java/monitors*), there will be too much uncertainty as to how well the systems we build correspond to the theoretical design.
- **Dynamics**
  - Theory and practice must be flexible enough to cope with process mobility, network growth and decay, disconnect and re-connect and resource sharing.
- **Performance**
  - Computational overheads for managing (*millions of*) evolving processes must be sufficiently low so as not to be a show-stopper.
- **Safety**
  - Massive concurrency – but no race hazards, deadlock, livelock or process starvation. The theory must be practical.

occam-M

- Processes, channels, (**PAR**) networks
- (**ALT**) choice between multiple events
- **Mobile data types - *review***
- **Mobile process types - *new***
- **Mobile channel types - *review***
- **Performance - *measured in nanoseconds***
- **Semantics - *not in this talk*** (Jim Woodcock, Xinbei Tang)

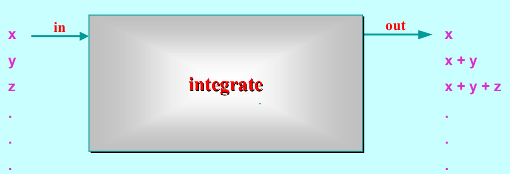
Processes and Channel-*Ends*




```
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.

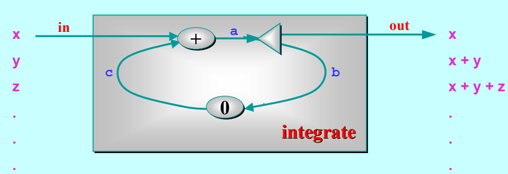
Processes and Channel-*Ends*




```
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

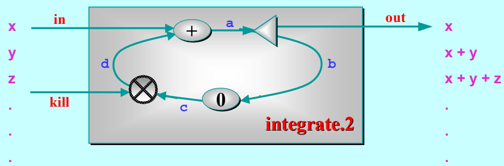
Parallel Process Networks




```
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

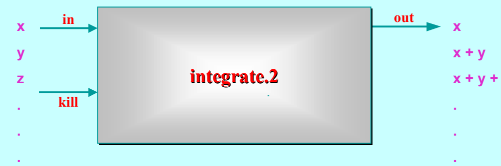
With an Added Kill Channel




```
PROC integrate.2 (CHAN INT in?, out!, kill?)
  CHAN INT a, b, c, d:
  PAR
    plus (in?, d?, a!)
    delta (a?, out!, b!)
    prefix (0, b?, c!)
    poison (kill?, c?, d!)
:
```

parallel implementation

With an Added Kill Channel




```
PROC integrate.2 (CHAN INT in?, out!, kill?)
  INITIAL INT total IS 0:
  INITIAL BOOL ok IS TRUE:
  ...  main loop
:
```

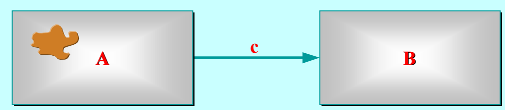
serial implementation

Choosing between Multiple Events


```
WHILE ok       -- main loop
  INT x:
  PRI ALT
    kill ? x
      ok := FALSE
    in ? x
      SEQ
        total := total + x
        out ! total
```

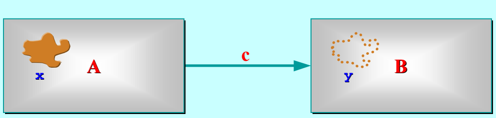
serial implementation

Copy *Data* Types



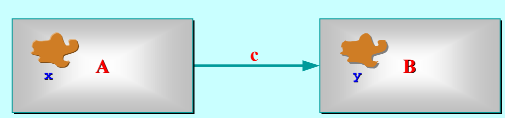

```
DATA TYPE FOO IS ... :

CHAN FOO c:
PAR
  A (c!)
  B (c?)
```


Copy *Data* Types




```
DATA TYPE FOO IS ... :

PROC A (CHAN FOO c!)          PROC B (CHAN FOO c!)
  FOO x:                        FOO y:
  SEQ                           SEQ
    ...  set up x                 ...  some stuff
    c ! x
```

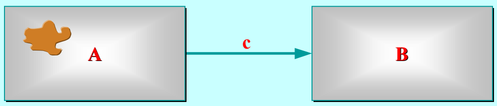

Copy *Data* Types




```
DATA TYPE FOO IS ... :

PROC A (CHAN FOO c!)          PROC B (CHAN FOO c!)
  FOO x:                        FOO y:
  SEQ                           SEQ
    ...  set up x                 ...  some stuff
    c ! x                         c ? y
    ...  more stuff               ...  more stuff
:                             :
```

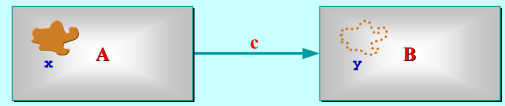
x and y reference *different* pieces of data

Mobile *Data* Types



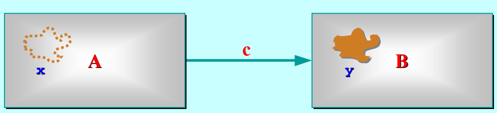

```
DATA TYPE M.FOO IS MOBILE ... :

CHAN M.FOO c:
PAR
  A (c!)
  B (c?)
```


Mobile *Data* Types




```
DATA TYPE M.FOO IS MOBILE ... :

PROC A (CHAN M.FOO c!)
  M.FOO x:
  SEQ
    ...  set up x
    c ! x

PROC B (CHAN M.FOO c!)
  M.FOO y:
  SEQ
    ...  some stuff
```


Mobile *Data* Types




```
DATA TYPE M.FOO IS MOBILE ... :

PROC A (CHAN M.FOO c!)
  M.FOO x:
  SEQ
    ...  set up x
    c ! x
    ...  more stuff
:

PROC B (CHAN M.FOO c!)
  M.FOO y:
  SEQ
    ...  some stuff
    c ? y
    ...  more stuff
:
```

The data has *moved* – x cannot be referenced

Mobile *Process* Types

Mobile processes exist in many technologies – such as *applets*, *agents* and in distributed operating systems.

occam-M offers (will offer) support for them with a formal *denotational* semantics, very high security and very low overheads.

Process mobility semantics follows naturally from that for mobile data and mobile channel-ends.

We need to introduce a concept of process *types* and *variables*.

Mobile *Process* Types

Process *type* declarations give names to **PROC** header templates. There are no restrictions on the types of parameters – they may be channels, data, timers, ports … and processes types as well.

```
PROC TYPE IN.OUT.KILL (CHAN INT in?, out!, kill?):
```

The above declares a process *type* called **IN.OUT.KILL**. Note that the earlier example, **integrate.2**, conforms to this type.

Process *types* are used in two ways: for the declaration of process *variables* and to define the *implementation interface* to a mobile process.

Mobile *Processes*

Mobile processes are entities encapsulating state and code. They may be *active* or *passive*. Initially, they are *passive*.

When *passive*, they may be *activated* or *moved*. A *moved* process remains *passive*. An *active* process cannot be *moved* or *activated* in parallel.

When an *active* mobile process *terminates*, it becomes *passive* – retaining its state. When it moves, its state moves with it. When re-*activated*, it sees its previous state.

The state of a mobile process can only be discovered by interacting with it when *active*. When passive, its state is locked – even against reading.

Mobile *Process* Example

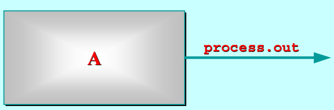
```
MOBILE PROC mobile.integrator.2

  INT total:                 -- private state

  CONSTRUCT ()               -- constructor 0
    total := 0
  :

  CONSTRUCT (VAL INT i)      -- constructor 1
    total := i
  :

  IMPLEMENTS IN.OUT.KILL (CHAN INT in?, out!, kill?)
    ...  active code body
  :

:
```

This is not an object – honest!

Mobile *Process* Example

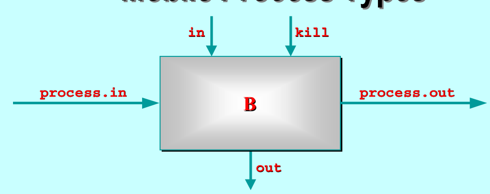
```
MOBILE PROC mobile.integrator.2

  ...  private state (total)

  ...  constructors (initialise total)

  IMPLEMENTS IN.OUT.KILL (CHAN INT in?, out!, kill?)
    INITIAL BOOL ok IS TRUE:
    WHILE ok
      INT x:
      PRI ALT
        kill ? x
          ok := FALSE
        in ? x
          SEQ
            total := total + x
            out ! total
  :
:
```


Mobile *Process* Types



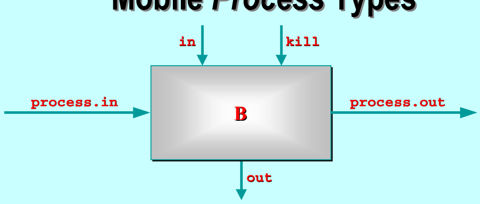

```
PROC A (CHAN IN.OUT.KILL process.out!)
  IN.OUT.KILL p:
  SEQ
    -- p is not yet defined (can't move or activate it)
    p := MOBILE mobile.integrator.2 ()
    -- p is now defined (can move and activate)
    process.out ! p
    -- p is now undefined (can't move or activate it)
:
```


Mobile *Process* Types



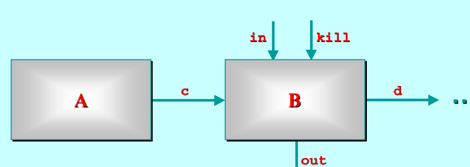

```
PROC B (CHAN IN.OUT.KILL process.in?, process.out!,
        CHAN IN in?, out!, kill?)
  IN.OUT.KILL q:
  WHILE TRUE
    SEQ  -- loop body
      ...  input a process to q
      ...  plug into local channels and activate q
      ...  when finished, send it on its way
:
```


Mobile *Process* Types




```
SEQ  -- loop body
  -- q is not yet defined (can't move or activate it)
  process.in ? q
  -- q is now defined (can move and activate)
  q (in?, out!, kill?)
  -- q is still defined (can move and activate)
  process.out ! q
  -- q is now undefined (can't move or activate it)
```


Mobile *Process* Network




```
CHAN IN.OUT.KILL c, d:
CHAN INT in, out, kill:
...  other channels
PAR
  A (c!)
  B (c?, d!, in?, out!, kill?)
  ...  other processes
```

Mobile *Processes* and *Types*

A process ***type*** may be implemented by many mobile processes – each offering different behaviours.

A ***mobile*** process may implement many process types – so it can be activated to provide different behaviours.

A process ***variable*** has a specific process type. Its value may be ***undefined*** or ***some mobile process*** implementing its type. When ***defined***, it can only be activated according to that type.

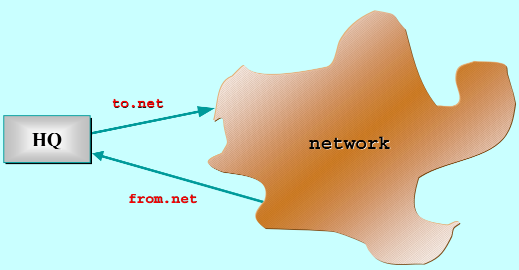
To activate one of the other behaviours offered by a mobile process, its process variable must first be ***re-typed***. This is a security issue – managed statically by the compiler with no run-time cost.

Mobile *Process* Example

```
MOBILE PROC mobile.integrator.3

  ...  private state (total)

  ...  constructors (initialises total)

  IMPLEMENTS IN.OUT.KILL (CHAN INT in?, out!, kill?)
    ...  active code body
  :

  IMPLEMENTS REFRESH (CHAN INT dump!, reset?)
    SEQ
      dump ! total
      reset ? total
  :

:
```

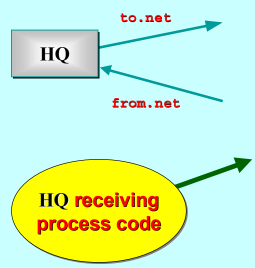
PROC TYPE IN.OUT.KILL (CHAN INT in?, out!, kill?):
PROC TYPE REFRESH (CHAN INT dump!, reset?):

Mobile *Process* Network



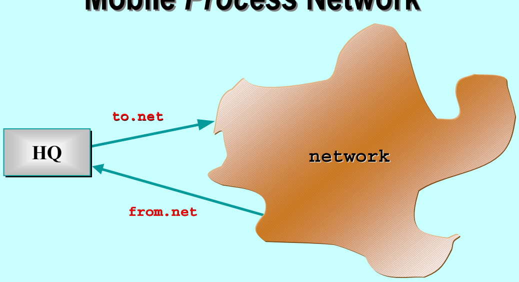

```
CHAN IN.OUT.KILL to.net, from.net:
PAR
  HQ (to.net!, from.net?)
  network (to.net?, from.net!)
```


Mobile *Process* Network




```
INT net.info:
SEQ
  ...
  WHILE TRUE
    IN.OUT.KILL q:
    SEQ
      from.net ? q
      REFRESH q RETYPES q:
      CHAN INT dump, reset:
      PAR
        q (dump!, reset?)
        INT reset.val:
        SEQ
          dump ? net.info
          ...  decide reset.val
          reset ! reset.val
```


Mobile *Process* Network



The REFRESH *process type* can be hidden within the HQ process – i.e. network processes cannot RETYPE the IN.OUT.KILL mobiles sent out from HQ.

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
:
```

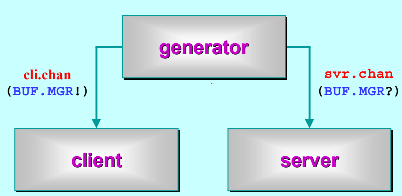
… and we also specify the *directions* of the component channels …

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
:
```

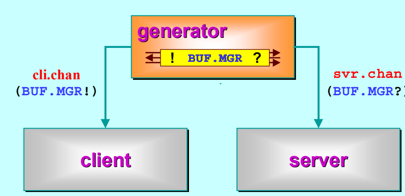
… the formal declaration indicates these directions from the viewpoint of the "**?**" end.

Mobile Channel Structures



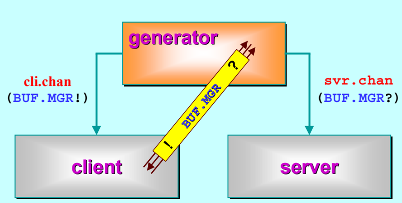

```
CHAN BUF.MGR! cli.chan:
CHAN BUF.MGR? svr.chan:
PAR
  generator (cli.chan! svr.chan!)
  client (cli.chan?)
  server (svr.chan?)
```


Mobile Channel Structures



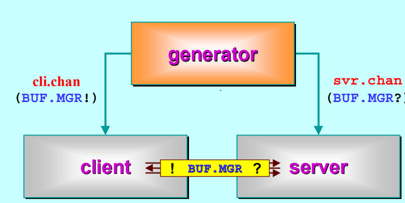

```
BUF.MGR! buf.cli:
BUF.MGR? buf.svr:
SEQ
  buf.cli, buf.svr := MOBILE BUF.MGR
```


Mobile Channel Structures



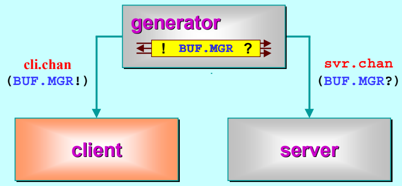

```
BUF.MGR! buf.cli:
BUF.MGR? buf.svr:
SEQ
  buf.cli, buf.svr := MOBILE BUF.MGR
  cli.chan ! buf.cli
```


Mobile Channel Structures



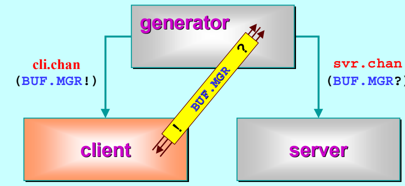

```
BUF.MGR! buf.cli:
BUF.MGR? buf.svr:
SEQ
  buf.cli, buf.svr := MOBILE BUF.MGR
  cli.chan ! buf.cli
  svr.chan ! buf.svr
  -- buf.cli and buf.svr are now undefined
```

Mobile Channel Structures



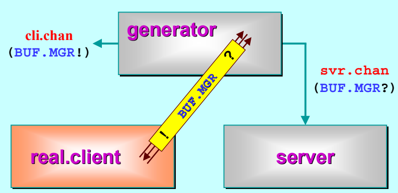

```
PROC client (CHAN BUF.MGR! cli.chan?)
  BUF.MGR! cv:
  SEQ
```


Mobile Channel Structures



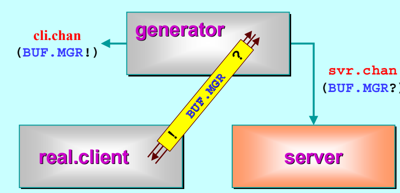

```
PROC client (CHAN BUF.MGR! cli.chan?)
  BUF.MGR! cv:
  SEQ
    cli.chan ? cv
```


Mobile Channel Structures



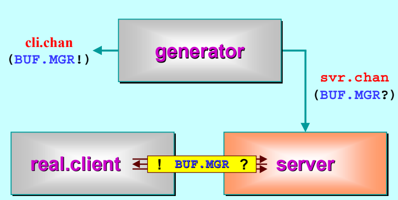

```
PROC client (CHAN BUF.MGR! cli.chan?)
  BUF.MGR! cv:
  SEQ
    cli.chan ? cv
    real.client (cv)
:
```


Mobile Channel Structures



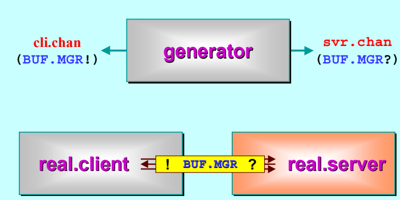

```
PROC server (CHAN BUF.MGR? svr.chan?)
  BUF.MGR? sv:
  SEQ
```


Mobile Channel Structures



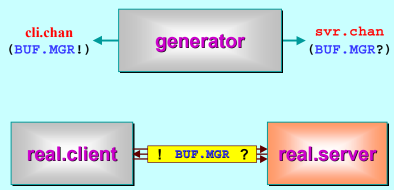

```
PROC server (CHAN BUF.MGR? svr.chan?)
  BUF.MGR? sv:
  SEQ
    svr.chan ? sv
```


Mobile Channel Structures



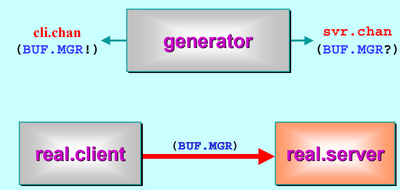

```
PROC server (CHAN BUF.MGR? svr.chan?)
  BUF.MGR? sv:
  SEQ
    svr.chan ? sv
    real.server (sv)
:
```

Mobile Channel Structures




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

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


Mobile Channel Structures

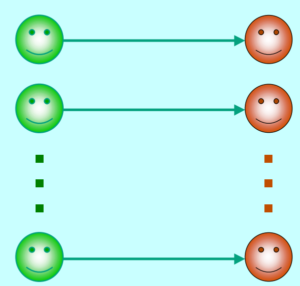



```
PROC real.client (BUF.MGR! call)
  ...
:

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


Process Performance

- **Memory overheads per parallel process:**
  - <= 32 bytes (depends on whether the process needs to wait on ***timeouts*** or perform ***choice*** (ALT) operations).
- ***Micro*-benchmarks (*800 MHz. Pentium III*) show:**
  - process (startup + shutdown): 28 ns (without) → 67 ns (priorites);
  - change priority (up /\ down): 63 ns;
  - channel communication (INT): 52 ns (no priorities) → 80 ns (priorites);
  - channel communication (***fixed-sized*** MOBILE): 120 ns (priorities, independent of size of the MOBILE) ;
  - channel communication (***dynamic-sized*** MOBILE): 180 ns (priorities, independent of size of the MOBILE) ;
  - all times independent of number of processes and priorities used – ***until cache misses kick in***.

Process Performance



p process pairs, m messages (INT) per pair – where (p*m) = 128,000,000.

Process Performance

- ***Micro*-benchmarks (*800 MHz. Pentium III*) show:**

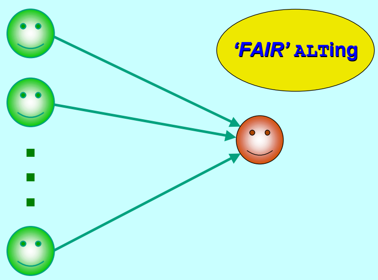
No. of pairs	CHAN INT communication
10	80 ns
100	77 ns
1,000	81 ns
10,000	455 ns
100,000	455 ns
1,000,000	494 ns

Process Performance

- ***Micro*-benchmarks (*2.4 GHz. Pentium IV*) show:**

No. of pairs	CHAN INT communication
10	97 ns
100	97 ns
1,000	112 ns
10,000	115 ns
100,000	119 ns
1,000,000	120 ns

Process Performance



128 writers (p *active*), m messages (INT) per *active* writer – where (p*m) = 128,000,000.

Process Performance

- ***Micro*-benchmarks (*800 MHz. Pentium III*) show:**

No. of *active* writers (out of 128)	'fair' ALT communication	'pri' ALT communication
128	126 ns	106 ns
64	-	107 ns
8	1124 ns	788 ns
1	1986 ns	1393 ns
0	10,000 ns	9,600 ns

Process Performance

- ***Micro*-benchmarks (*800 MHz. Pentium III*) show:**

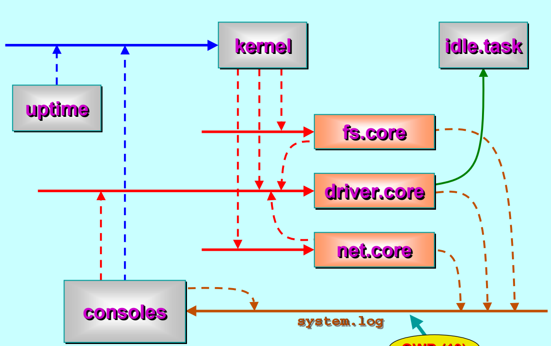
'fair' ALT communication	fixed overhead	cost per guard
'stressed' (events always being offered)	(80 + 32) ns	14 ns
'unstressed' (no events on offer - initially)	2000 ns*	63 ns

*for 128 guards (= 'stressed' cost when no guards are ready)

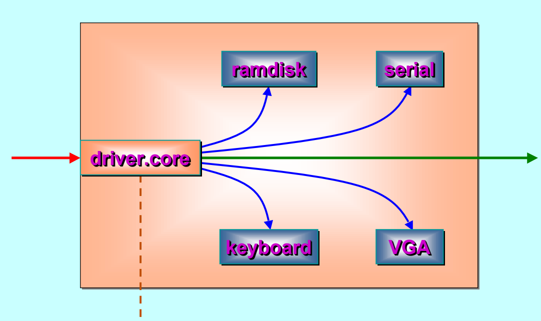
The Raw Metal occam eXperience (RMoX)

- **An operating system based on (extended) CSP**
  - Simple, fast and safe concurrency (natural *'plug-and-play'*)
  - Design confidence (mature theory of refinement)
- **Written in occam-M**
  - Good testing ground for our dynamic extensions and priorities
  - Low memory footprint and very quick
  - Compositional development
  - Interrupts mapped to channel communications
  - Millions of processes (per processor)
  - Scaleable across networks
  - *Fun !!!*
- **Applications**
  - Field-programmable embedded systems (including real-time)
  - General operating system (with support for Linux)

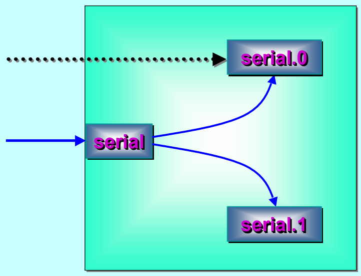
RMoX: Initial Process Network



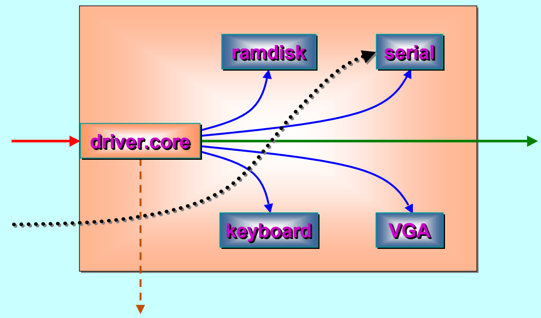
RMoX: Device Drivers



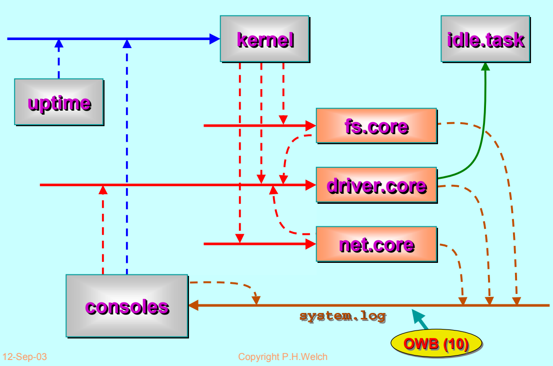
RMoX: Serial Drivers



RMoX: Device Drivers



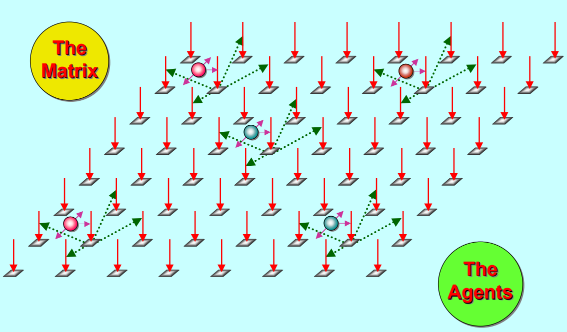
RMoX: Dynamic Process Network



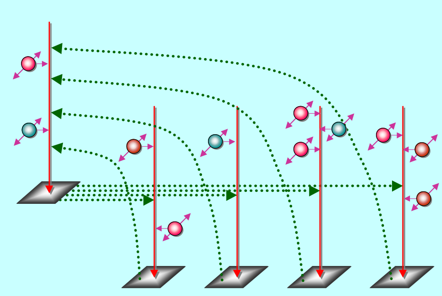
Modelling Bio-Mechanisms

- **In-vivo ←→ In-silico**
  - One of the UK *'Grand Challenge'* areas.
  - Move *life-sciences* from *description* to *modelling / prediction*.
  - Example: the Nematode worm
  - Development: from fertilised cell to adult (with virtual experiments).
  - Sensors and movement: reaction to stimuli.
  - Interaction between organisms and other pieces of environment.
- **Modelling technologies**
  - Communicating process networks – fundamentally good fit.
  - Cope with growth / decay, combine / split (evolving topologies).
  - Mobility and location / neighbour awareness.
  - Simplicity, dynamics, performance and safety.
- **occam-M (and JCSP)**
  - Robust and lightweight – good theoretical support.
  - *O(10,000,000)* processes with useful behaviour in useful time.
  - Enough to make a start …

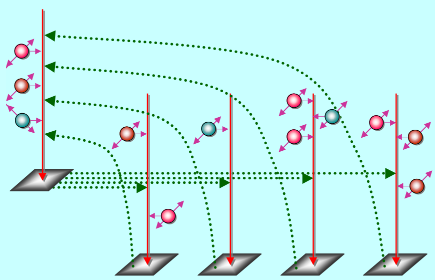
Location (Neighbourhood) Awareness



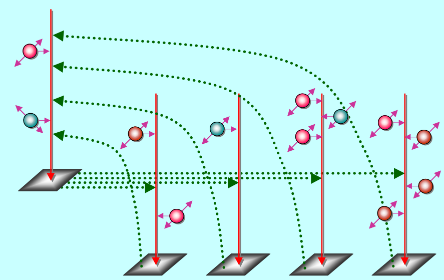
Location (Neighbourhood) Awareness



Location (Neighbourhood) Awareness



Location (Neighbourhood) Awareness



Mobility and Location Awareness

- **The Matrix**
  - A network of (mostly passive) server processes.
  - Responds to client requests from the mobile agents and, occasionally, from other server nodes.
  - Deadlock avoided (in the matrix) *either* by one-place buffered server channels *or* by pure-client slave processes (one per matrix node) that ask their server node for elements (e.g. mobile agents) and forward them to neighbouring nodes.
  - Server nodes only see neighbours, maintain registry of currently located agents (and, maybe, agents on the neighbouring nodes) and answer queries from local agents (including moving them).
- **The Agents**
  - Attached to one node of the Matrix at a time.
  - Sense presence of other agents – on local or neighbouring nodes.
  - Interact with other local agents – must use agent-specific protocol to avoid deadlock. May decide to reproduce, split or move.
  - Local (or global) *sync barriers* to maintain sense of time.

Summary – 1/2

- **occam-M**
  - All dynamic extensions (bar mobile *processes*) implemented in **KRoC** 1.3.3 (*pre-16*).
  - Mobile processes *proposed* with denotational semantics (**CSP-M**) in first draft (Jim Woodcock, Xinbei Tang) – implementation not too hard.
  - Hierarchical networks, dynamic topologies, safe sharing (of data and channels).
  - *Total alias control* by compiler : zero aliasing accidents, zero race hazards, zero nil-pointer exceptions and zero garbage collection.
  - Zero buffer overruns.
  - Most concurrency management is unit time – *O(100)* nanosecs on modern architecture.
  - Only implemented for x86 Linux and **RMoX** – other targets straightforward (but no time to do them).
  - Full open source (GPL / L-GPL).
  - Formal methods: **FDR** model checker, refinement calculus (**CSP** and **CSP-M**), Circus (**CSP** + **Z**).

Summary – 2/2

- ***We Aim to Have Fun …***
  - Interesting applications everywhere …
  - Beat the complexity / scalability rap …
  - Would anyone like to join us … ?

Any Questions?

- ***Google – I'm feeling Lucky …***
  - `KRoC + ofa` -- occam (official)
  - `occam + web server` -- occam (latest)
  - `JCSP` -- CSP for Java
  - `Quickstone` -- JCSP Networking Edition (Java / J#)
  - `Grand Challenges + UK` -- In-vivo ←→ In-silico
  - `CPA 2003 + Sept` -- *'Communicating Process* -- *Architectures'* conference
  - `WoTUG` -- Lots of good people ...
- ***Mailing lists …***
  - `occam-com@kent.ac.uk`
  - `java-threads@kent.ac.uk`