Through the Concurrency Gateway

Peter Welch
University of Kent at Canterbury
Computing Laboratory
p.h.welch@kent.ac.uk

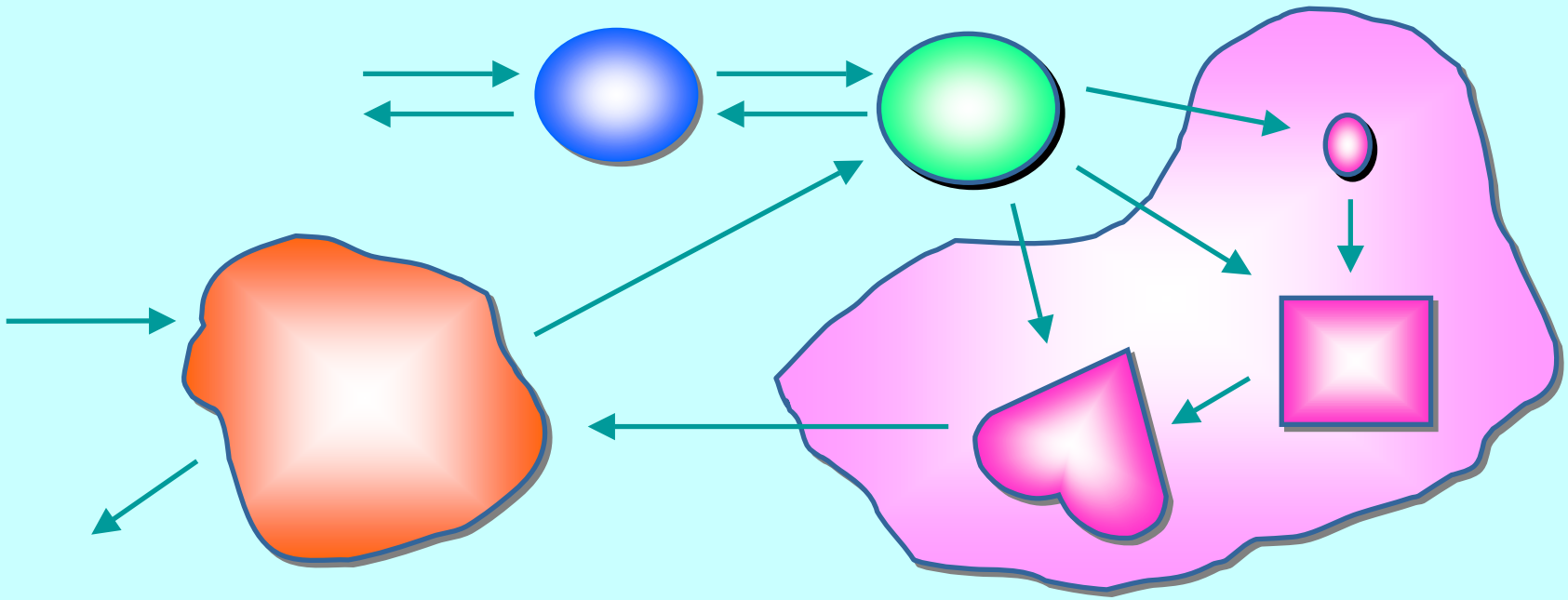
Grand Challenges in Computing Research
Journeys in Non-Classical Computation (GC6)
Newcastle (30th. March, 2004)

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 in need** *(now and real soon)*
  - Air traffic control.
  - Web services and mobile agents.
  - Biological system modelling and ***In Silico*** experiments.
  - Nannite modelling, safety and control.
  - …

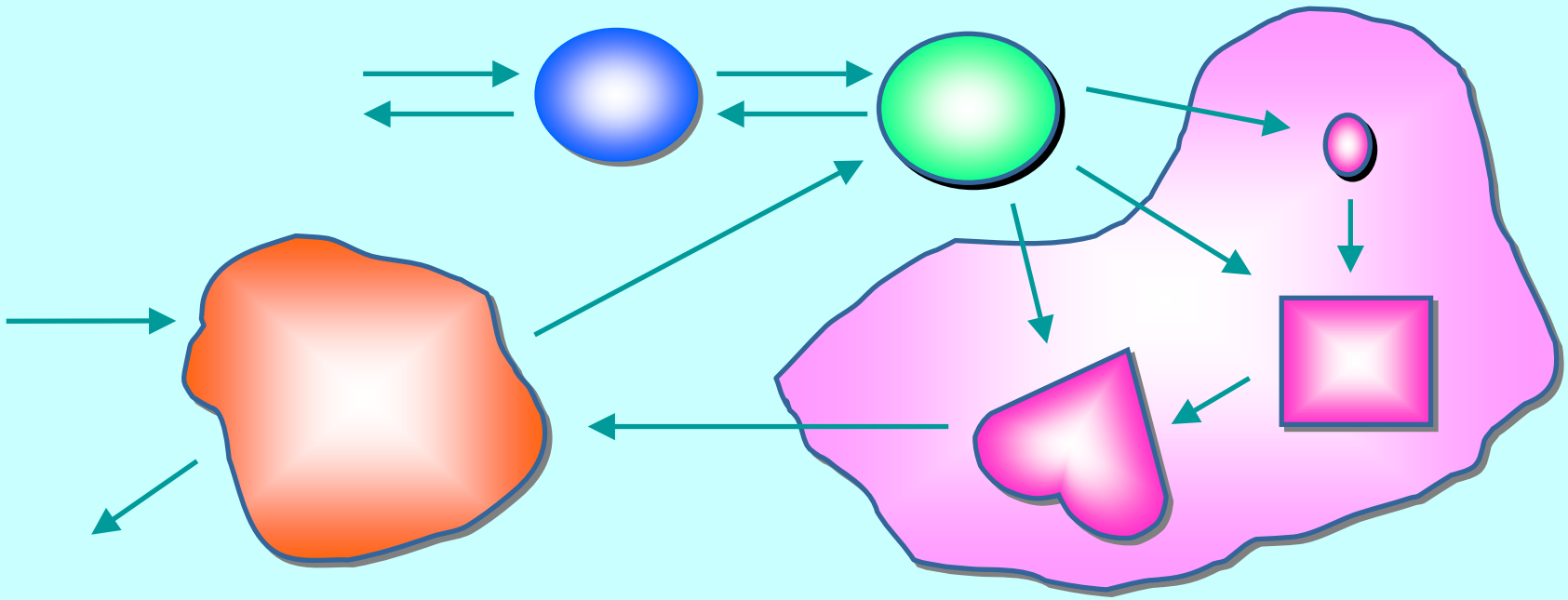
Nature has very large numbers of independent agents, interacting with each other in regular and chaotic patterns, at all levels of scale:

… nannite … human … astronomic ...



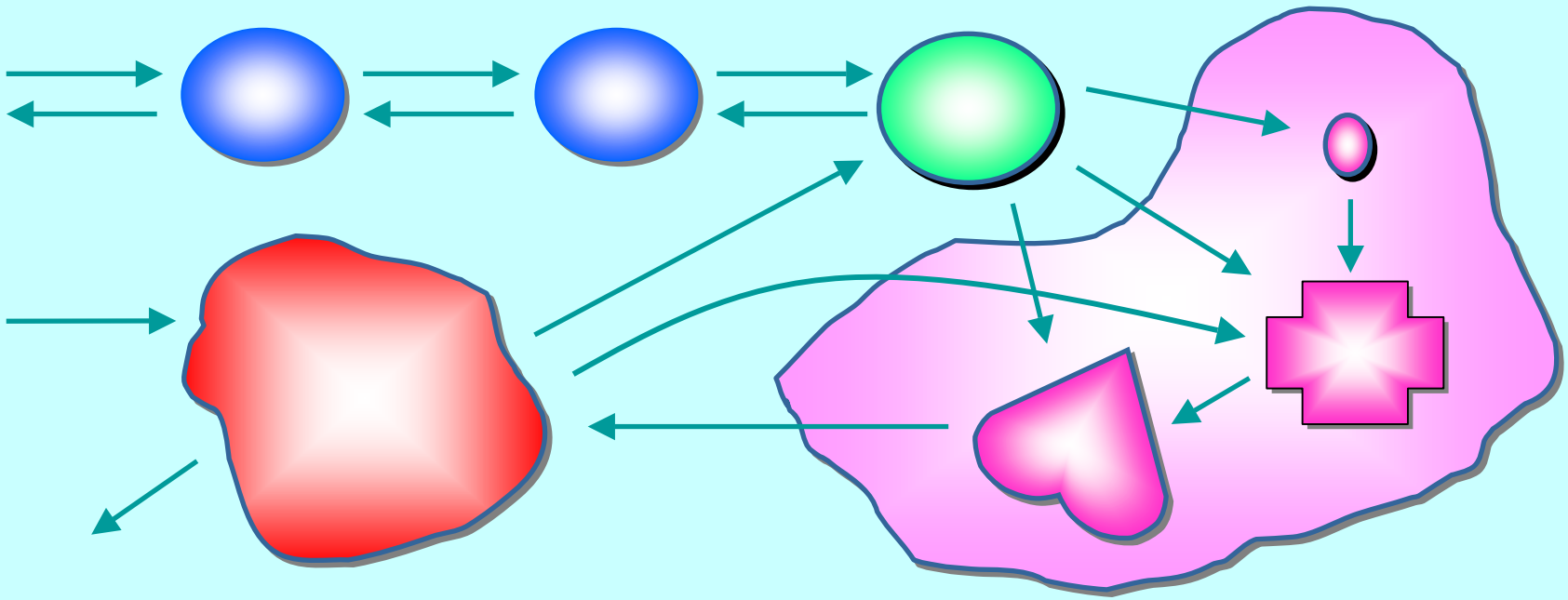
The networks are dynamic: growing, decaying and mutating internal topology (in response to environmental pressure and self-motivation):

… nannite … human … astronomic ...

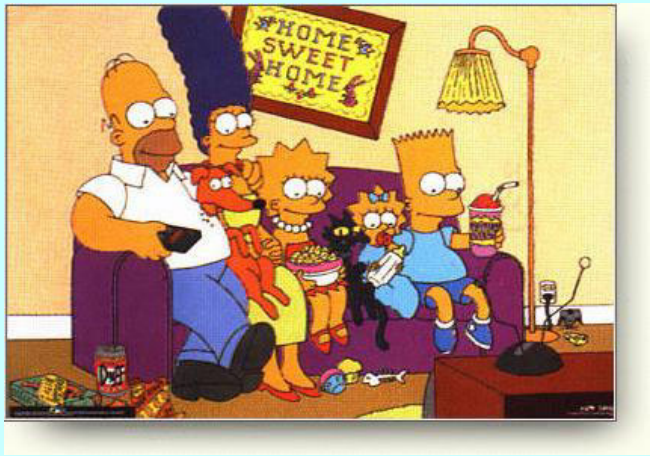


The networks are dynamic: growing, decaying and mutating internal topology (in response to environmental pressure and self-motivation):

… nannite … human … astronomic ...



Components: the key to complexity?

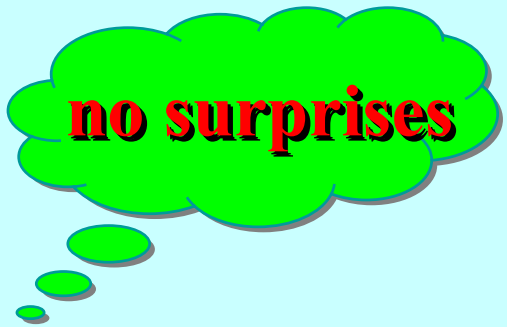


Components must be ***composeable*** …

… and they must compose ***simply***!

Components: the key to complexity?

- If we understand A and B separately, we must be able to deduce ***simply*** their combined behaviour.



- Semantics [A + B] = Semantics [A] + Semantics [B]

- A and B must be ***composeable*** …

Components: the key to complexity?

- ***Complex*** systems are ***composed*** from ***simpler*** components …
- … which are ***composed*** from ***simpler*** components …
- … which are ***composed*** from ***simpler*** components …
- … *etc.*
- … which are ***composed*** from ***simple*** components.

Components: the key to complexity?

- Composition rules must be simple and yield no surprises.
- Whatever it is they encapsulate, ***components*** must have ***interfaces*** that are ***clean***, ***complete*** and ***explicit***.
- Hardware systems are forced (by physics/geometry) to be built like this.
- Software systems have no such constraints. We think we can do better than nature … and get into trouble.

Concurrency in the Real World

Computer systems - to be of use in this world - need to model that part of the world for which it is to be used.

If that modeling can reflect the natural concurrency in the system … it should be ***simpler***.

Yet concurrency is thought to be an ***advanced*** topic, ***harder*** than serial computing (which therefore needs to be mastered first).

Threads-n-Locks - CONCERNS

<java.sun.com/products/jfc/tsc/articles/threads/threads1.html>

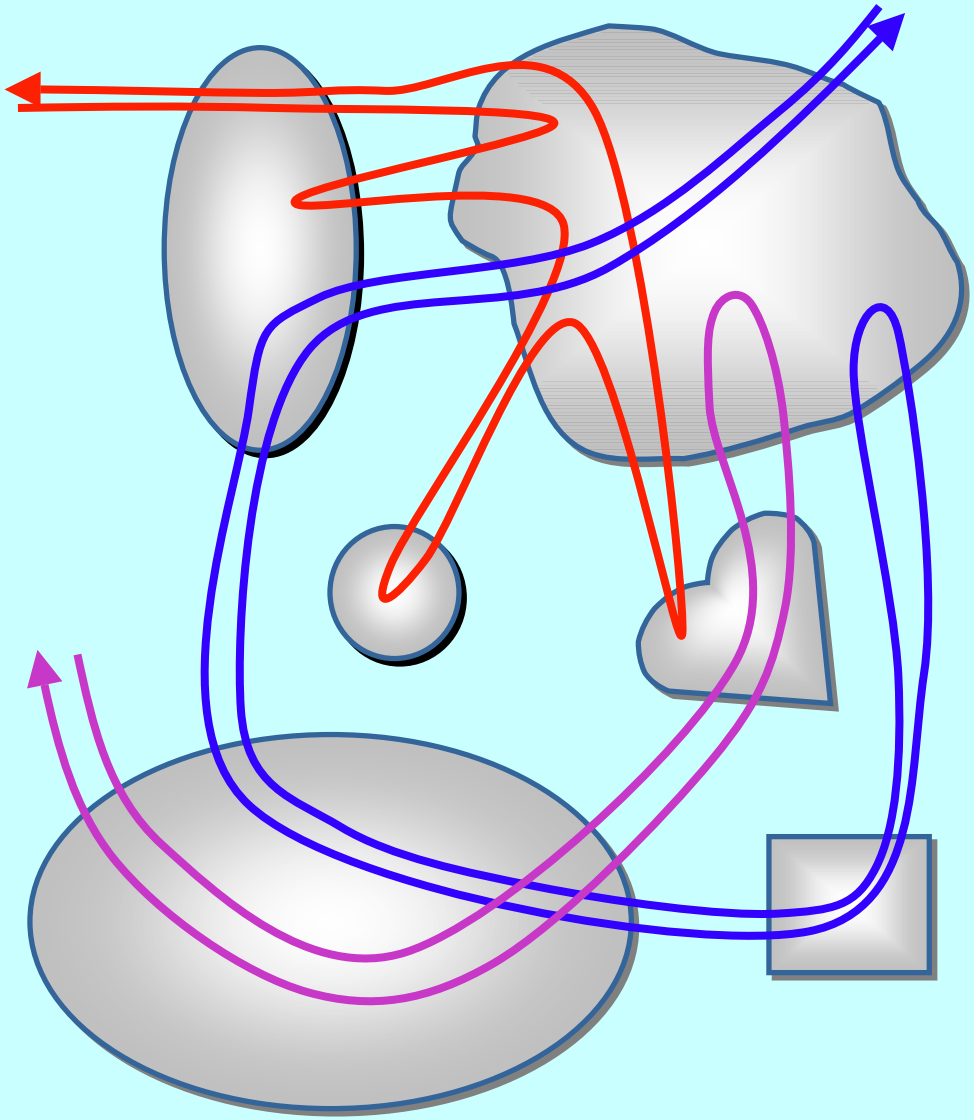
- ***"If you can get away with it, avoid using threads. Threads can be difficult to use, and they make programs harder to debug."***

- ***"Component developers do not have to have an in-depth understanding of threads programming: toolkits in which all components must fully support multithreaded access, can be difficult to extend, particularly for developers who are not expert at threads programming."***

Threads-n-Locks Considered Harmful

Each single thread of control snakes around objects in the system, bringing them to life *transiently* as their methods are executed.

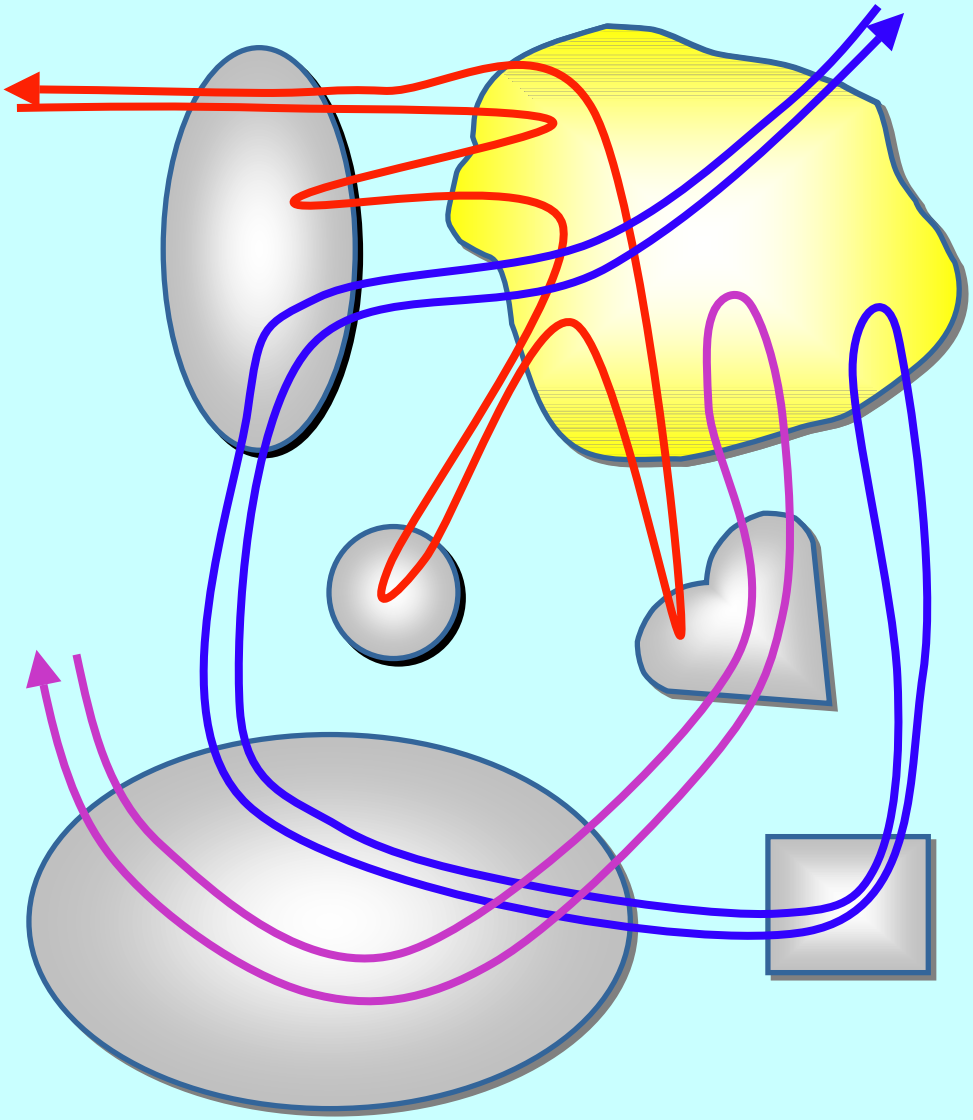
Threads cut across object boundaries leaving spaghetti-like trails, *paying no regard to the underlying structure*.



Threads-n-Locks Considered Harmful

Each object is at the mercy of ***any*** thread that sees it. Nothing can be done to ***prevent*** method invocation … even if the object is not in a fit state to service it. ***The object is not in control of its life***.

Big problems occur when multiple threads hit the same object.



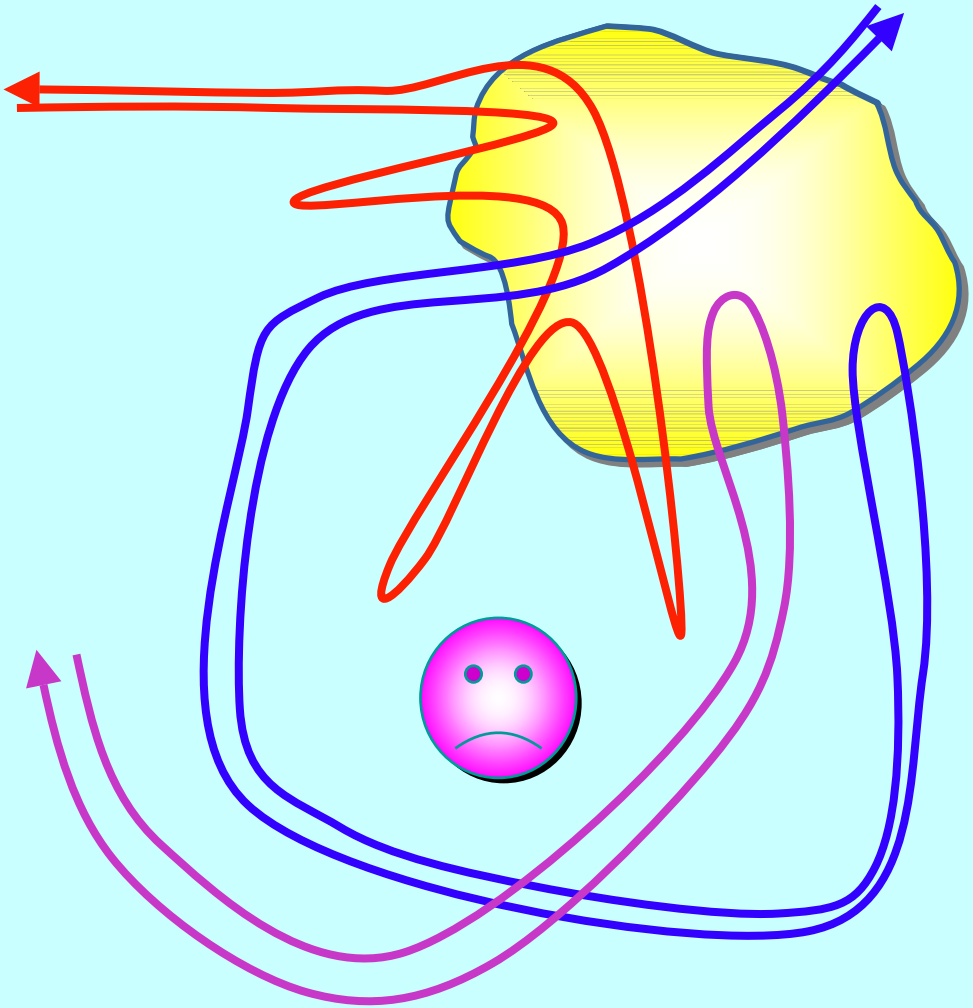
Threads-n-Locks Considered Harmful

Errors in claiming/releasing locks is probably the main reason our systems fail …

Too much locking and we get deadlock …

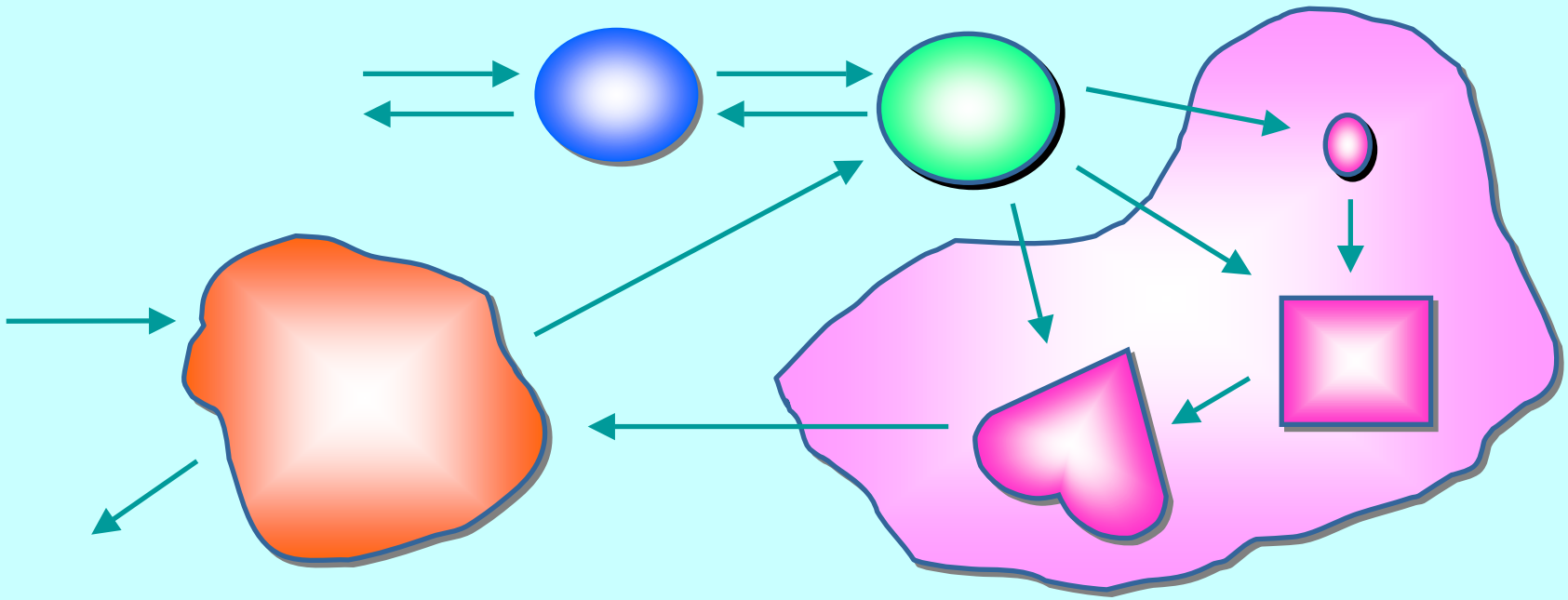
Too little locking and race hazards slowly corrupt …

Sorting this out requires controlling all possible interleavings … which is exponential in the number of threads …



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

… nannite … human … astronomic ...



Threads-n-Locks - CONCERNS

<java.sun.com/products/jfc/tsc/articles/threads/threads1.html>

- ***"It is our basic belief that extreme caution is warranted when designing and building multi-threaded applications … use of threads can be very deceptive … in almost all cases they make debugging, testing, and maintenance vastly more difficult and sometimes impossible. Neither the training, experience, or actual practices of most programmers, nor the tools we have to help us, are designed to cope with the non-determinism … this is particularly true in Java … we urge you to think twice about using threads in cases where they are not absolutely necessary …"***

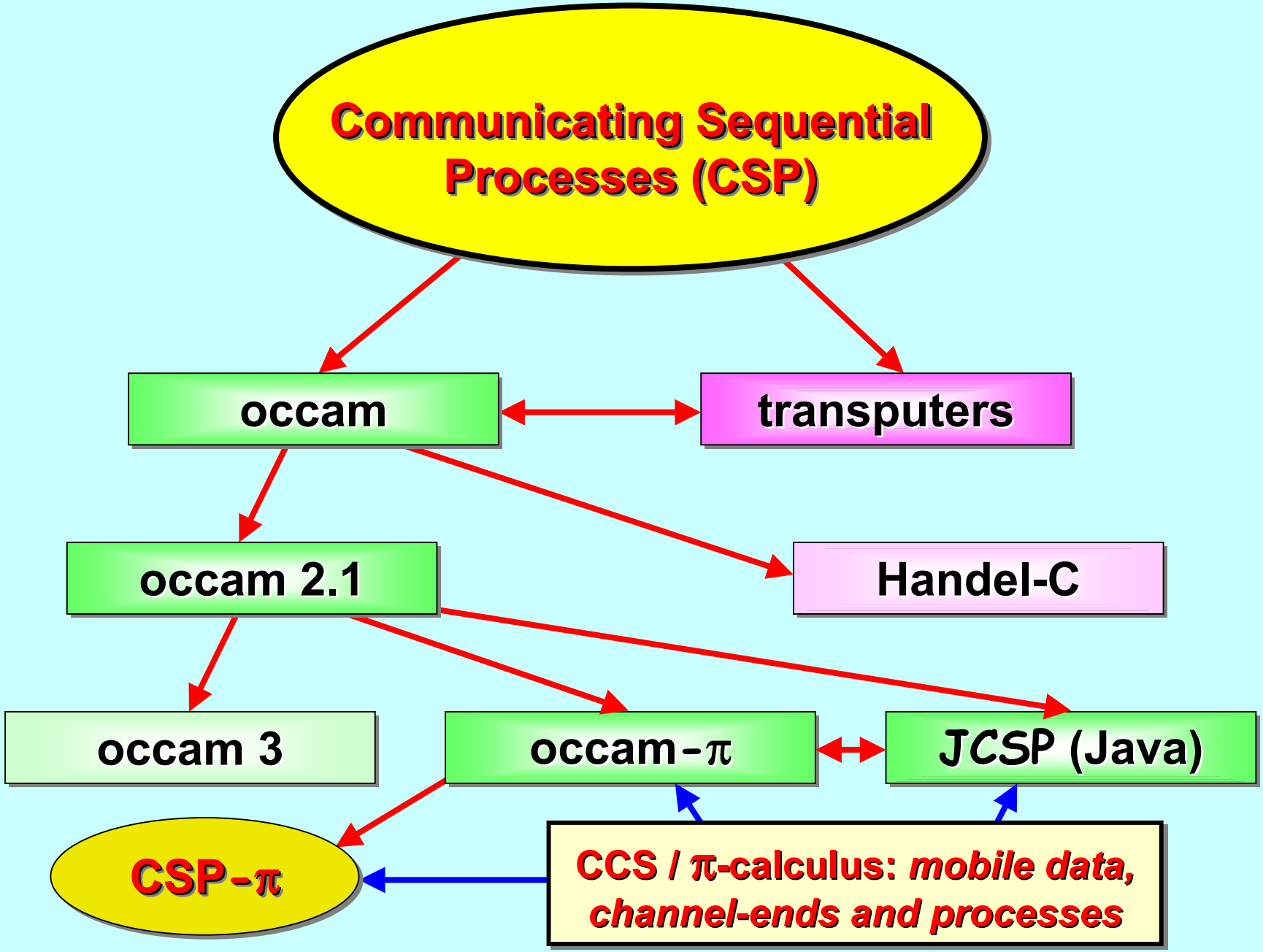
This tradition is WRONG

… which has (radical) implications on how we should educate people for computer science …

… and on how we apply what we have learnt …

What we want from *Concurrency*

- A powerful tool for ***simplifying*** the description of systems.
- ***Performance*** that spins out from the above, but is ***not*** the primary focus.
- A model of concurrency that is ***mathematically clean***, yields no engineering surprises and scales well with system complexity.



occam-π

- Processes, channels, (**PAR**) networks
- (**ALT**) choice between multiple events
- Mobile data types
- Mobile channel types
- Mobile process types
- Performance

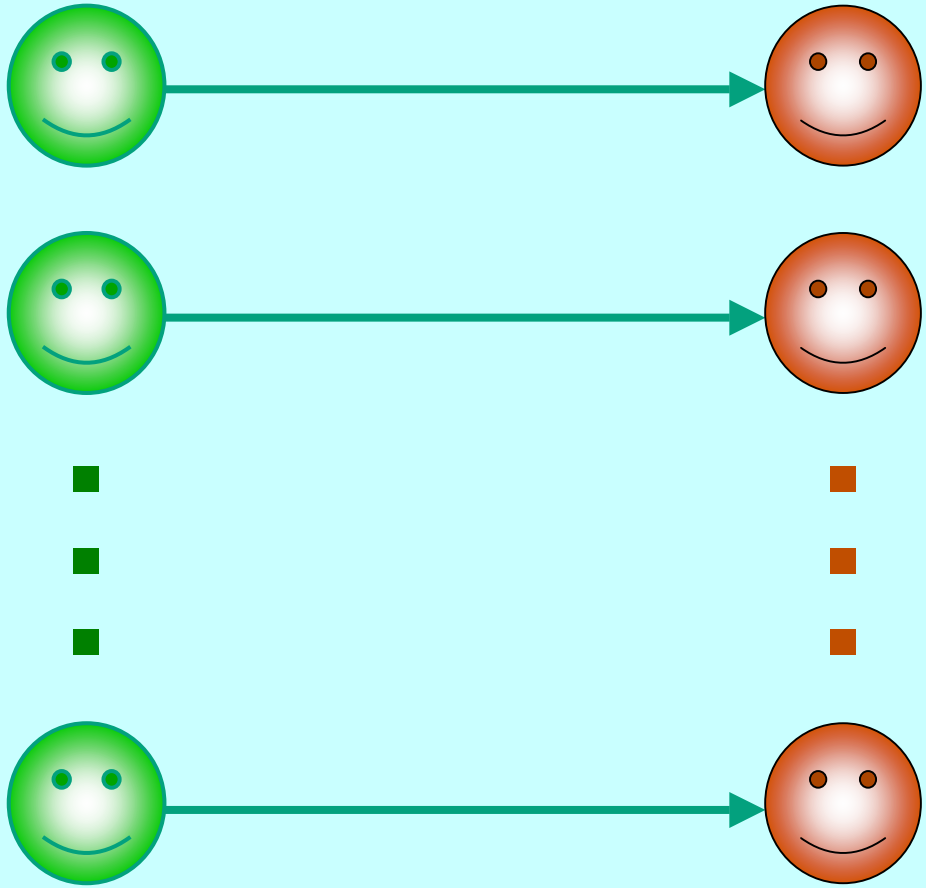
Aspirations and Principles

- **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, location awareness, 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.

Process Performance (occam-π)

- **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



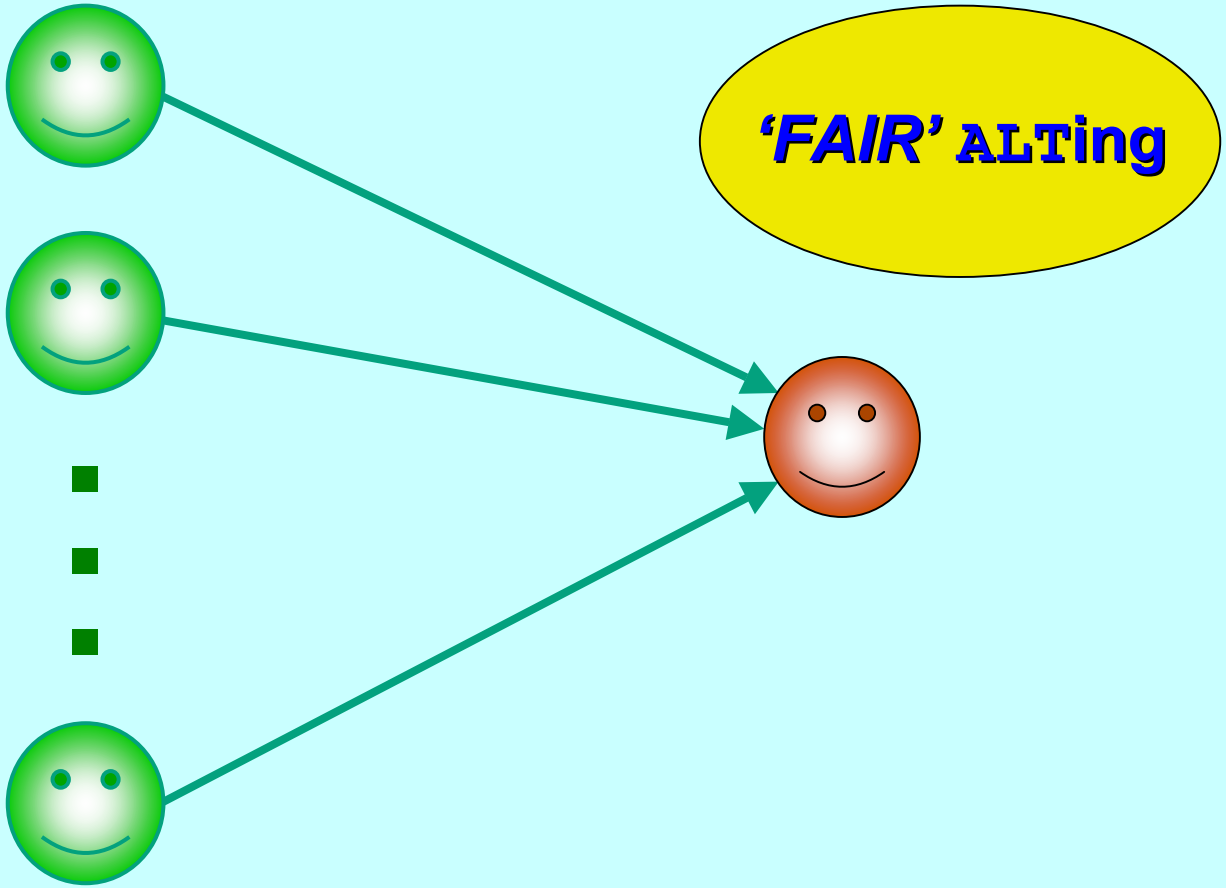
p process pairs, **m** messages (INT) per pair
– where (**p*****m**) = 128,000,000.

Process Performance

- ***Micro*-benchmarks (*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:**

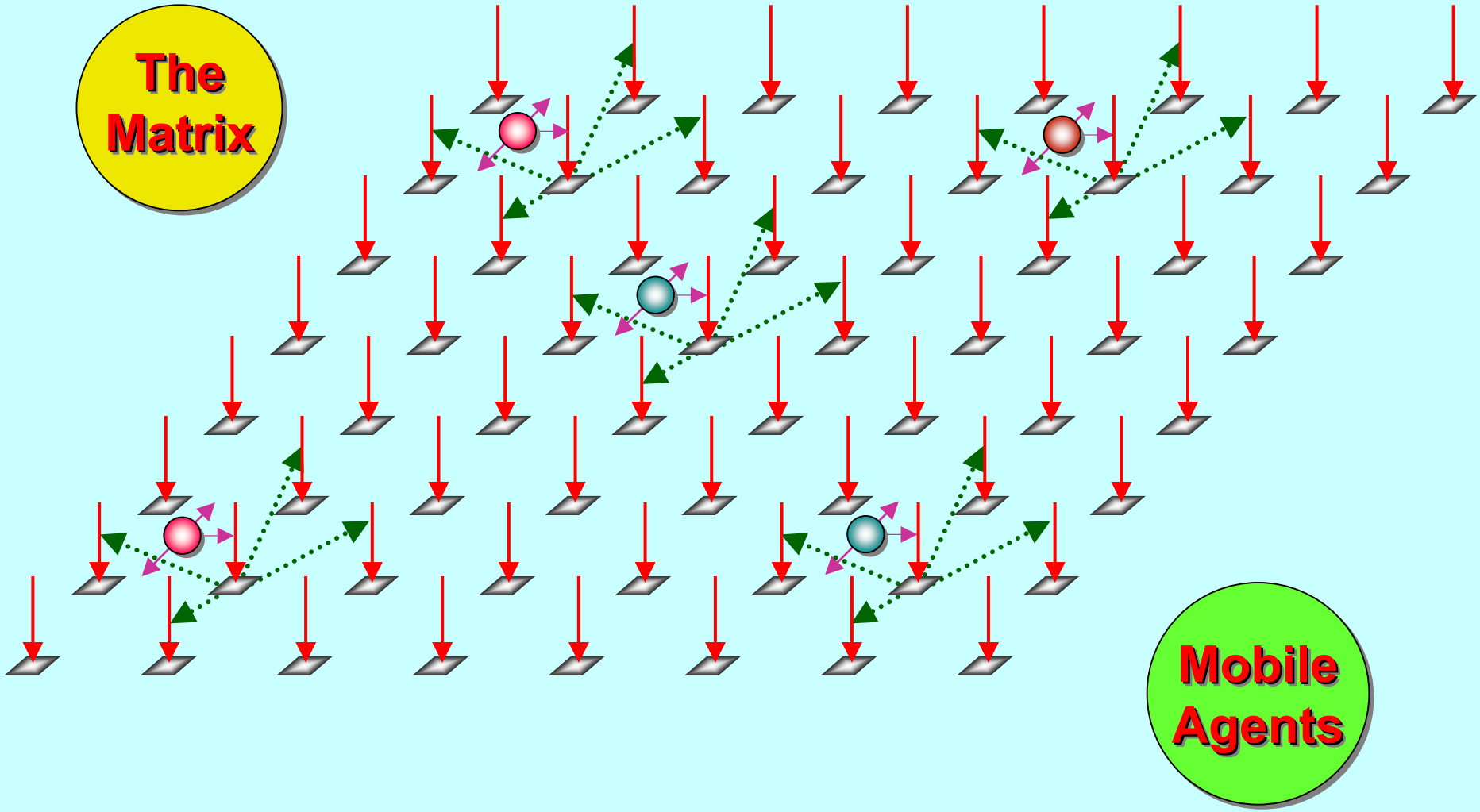
`'fair' ALT` communication	fixed overhead	cost per guard
'stressed' (events always being offered)	(80 + 32) ns	14 ns
'unstressed' (no events on offer - initially)	2000 ns*	63 ns

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

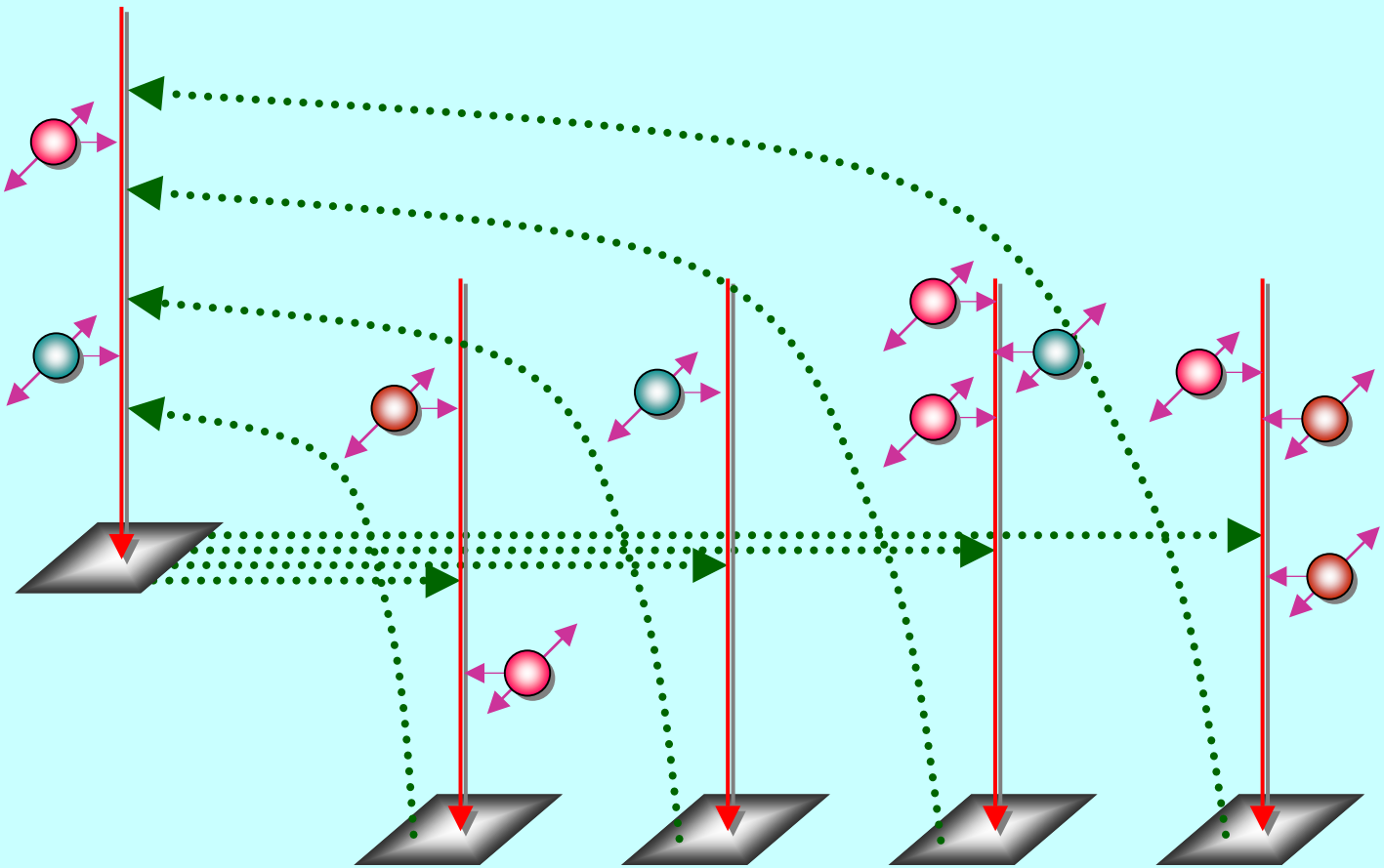
Modelling Bio-Mechanisms

- **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-π (and JCSP)**
  - Robust and lightweight – good theoretical support.
  - *O(1,000,000)* processes with useful behaviour in useful time.
  - Enough to make a start … *Moore's Law really needed!*

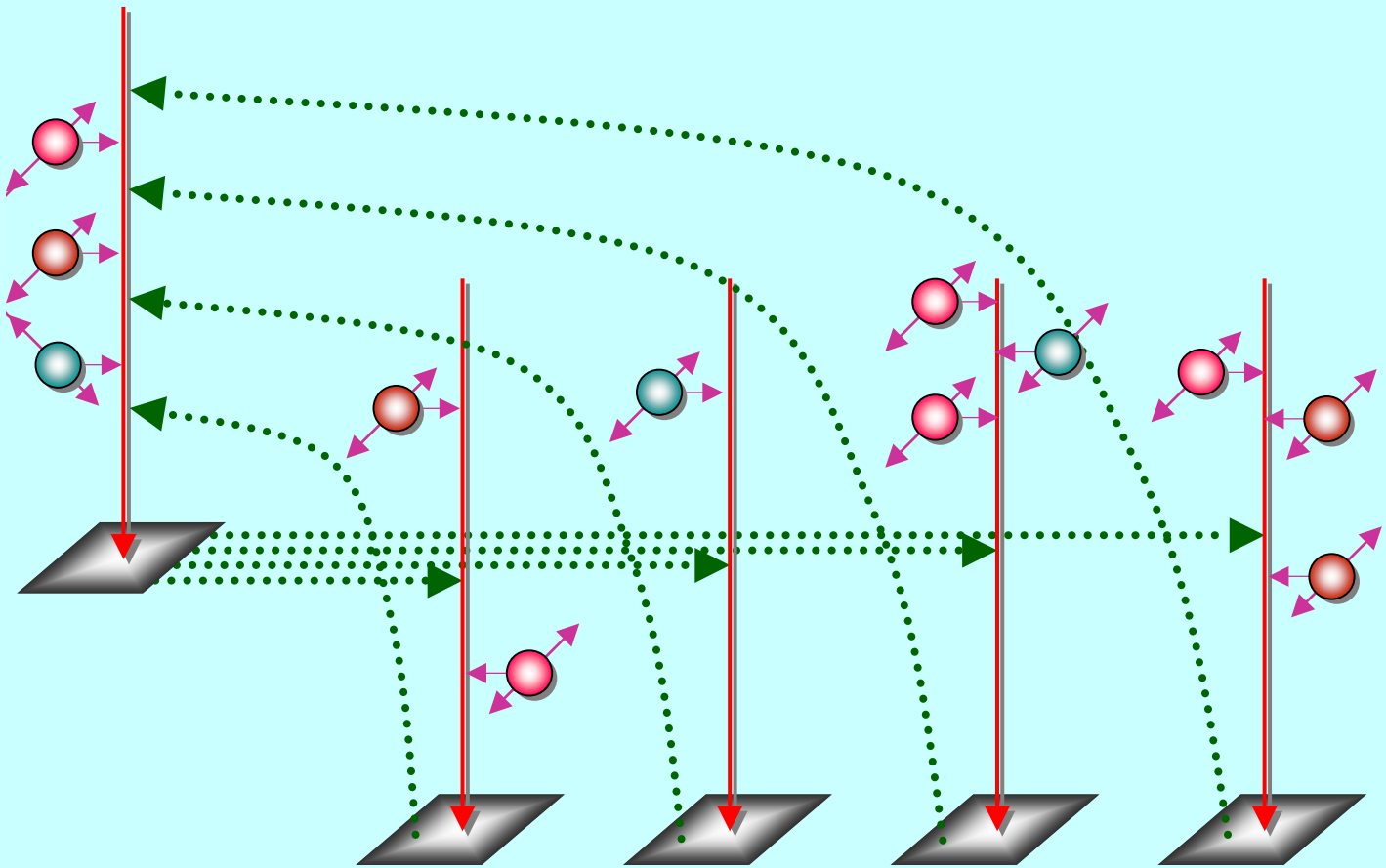
Location (Neighbourhood) Awareness



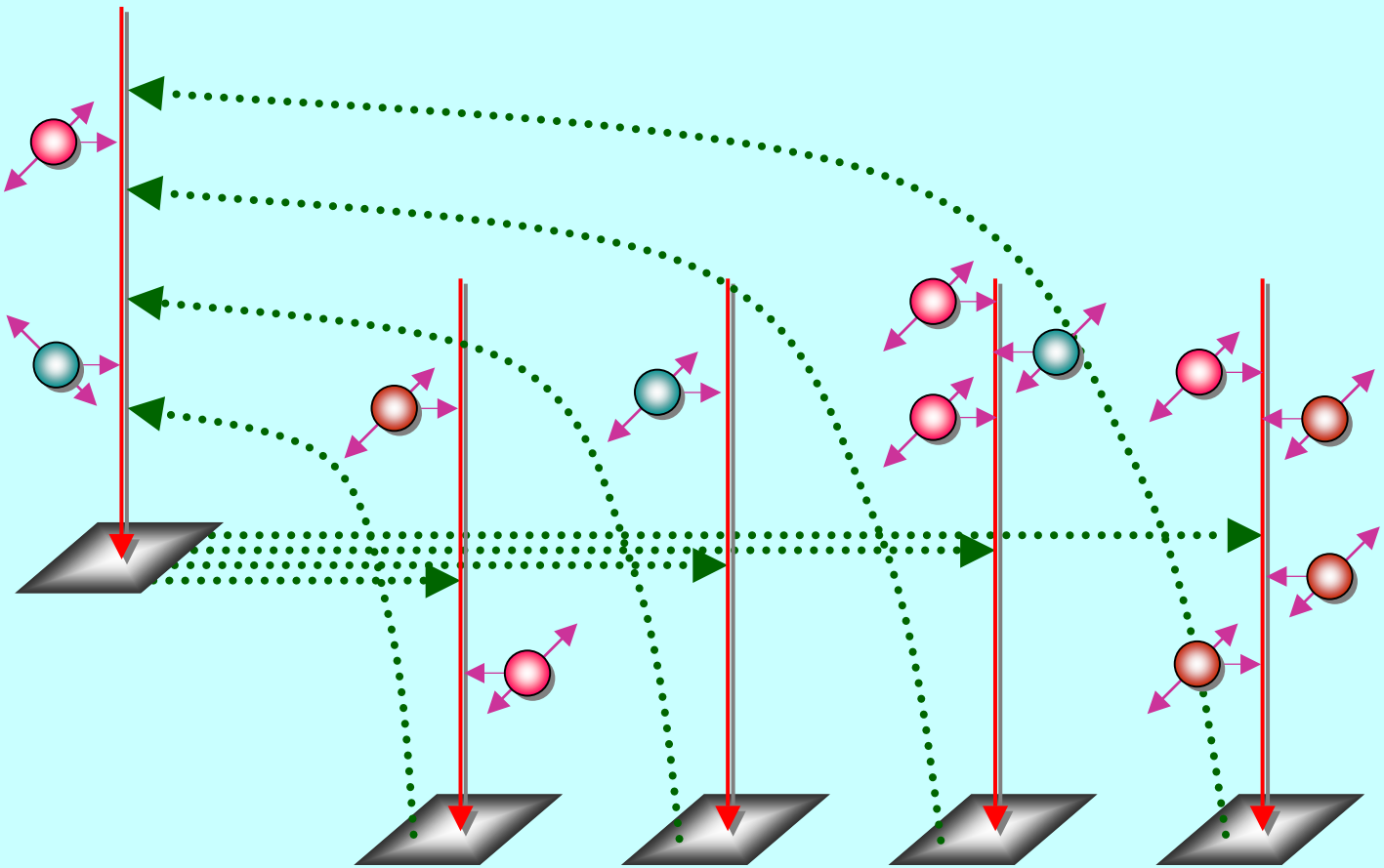
Location (Neighbourhood) Awareness



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 *neighbouring* 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.

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?
  - *Simple operational semantics:* low overhead implementation? *Yes.*
  - *Process algebra:* combine the best of CSP and the *π-calculus*? *Yes.*
  - *Refinement:* for manageable system verification … can we keep?
  - *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?

Summary – 1/4

- **CSP** has a ***compositional*** semantics.
- ***Concurrency-for-All now*** ➔ ***simplify*** design:
  - data encapsulation within processes does not break down (unlike the case for objects);
  - channel interfaces impose clean decoupling between processes (unlike method interfaces between objects).
- **JCSP** enables direct Java implementation of **CSP-π** design. **KRoC** gives us the real (**occam-π**) thing.

Summary – 2/4

- **The right stuff (occam-π)**
  - Nature builds robust, complex and successful systems by allowing independent organisms control of their own lives and letting them interact. *Central points of control do not remain viable for long*.
  - Computer (software) engineers should take the hint! Concurrency should be a *natural way* to design any computer system (or component) above a minimal level of complexity.
  - It should *simplify* and *hasten* the construction, commissioning and maintenance of systems; it should not introduce the hazards that are evident in current practice; *and it should be employed as a matter of routine*.
  - *Natural* mechanisms should map into *simple* engineering mechanisms *with low cost and high benefit*.
  - To do this requires a paradigm shift in the way we approach concurrency ... *to something much simpler*.
  - Failure to do this will result in failure to meet the *'Grand Challenges'* that the 21st. Century is stacking up for us.

Summary – 3/4

- **occam-π**
  - All dynamic extensions (bar mobile *processes*) implemented in **KRoC** 1.3.3 (*pre-22*).
  - Mobile processes *proposed* with denotational semantics (**CSP-π**) in draft (Jim Woodcock, Xinbei Tang) – implementation 'not hard'.
  - Hierarchical networks, dynamic topologies, structural integrity, 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-π**), Circus (**CSP** + **Z**).

Summary – 4/4

- **We Aim to Have Fun …** 
  - Make a start on the GC7 journey …
  - Beat the complexity / scalability rap …
  - Necessary to start now …

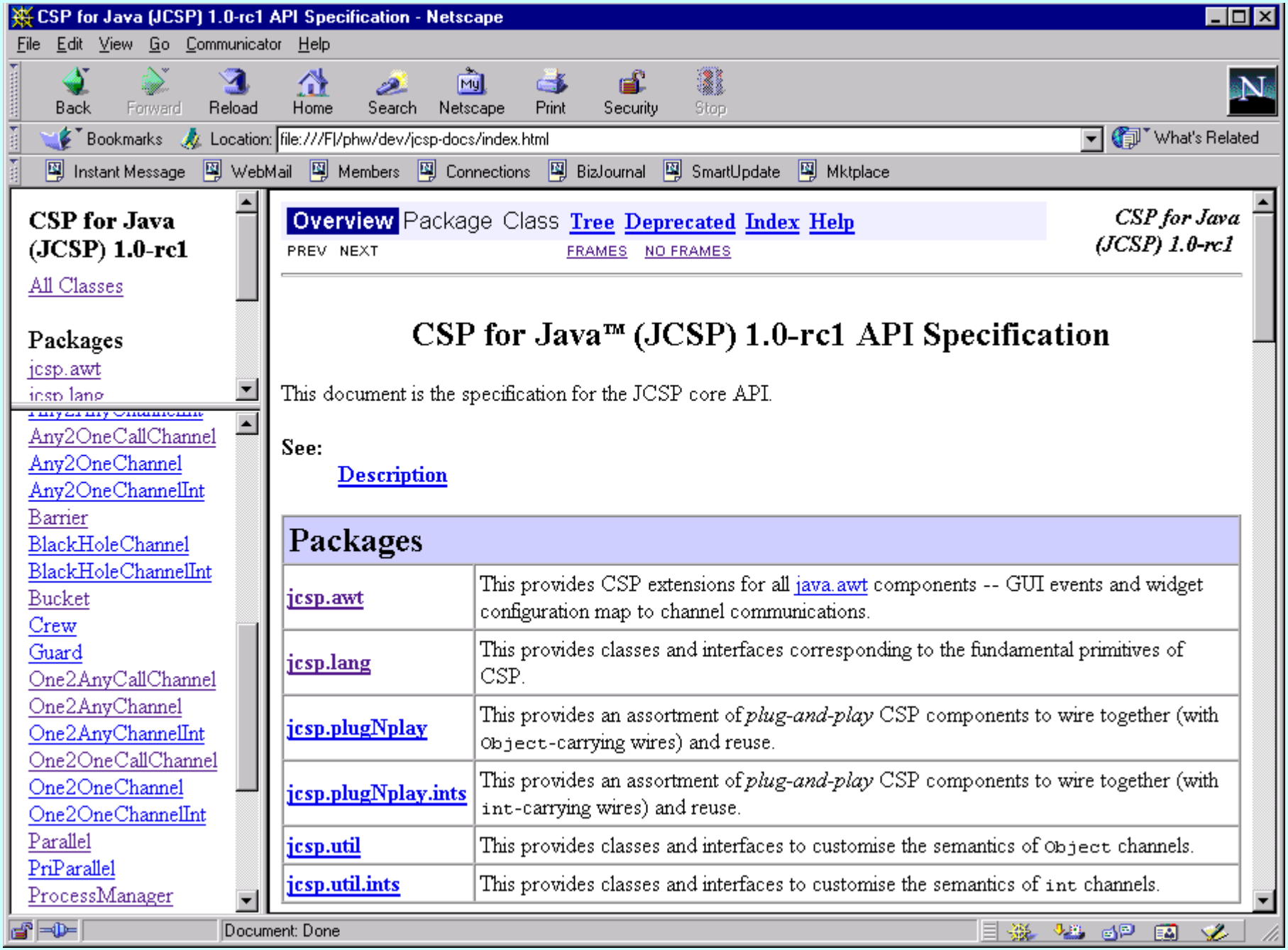
- ***Google – I'm feeling Lucky …***
  - **KRoC + ofa** -- occam-π (official)
  - **KRoC + linux** -- occam-π (latest)
  - **JCSP** -- CSP-π for Java
  - **Quickstone** -- *JCSP* Networking Edition (Java / J#)
  - **Grand Challenges + UK** -- In-vivo ⇔ In-silico
  - **CPA 2004 + Conference** -- *'Communicating Process Architectures'* conference
  - **WoTUG** -- Lots of good people ...
- **Mailing lists …**
  - **occam-com@kent.ac.uk**
  - **java-threads@kent.ac.uk**

Putting CSP into practice …

http://www.cs.ukc.ac.uk/projects/ofa/kroc/

Putting CSP into practice …

http://www.cs.ukc.ac.uk/projects/ofa/jcsp/


CSP for Java (JCSP) 1.0-rc1 API Specification - Netscape
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print Security Stop
Bookmarks Location: file:///F|/phw/dev/jcsp-docs/index.html What's Related
Instant Message WebMail Members Connections BizJournal SmartUpdate Mktplace
CSP for Java (JCSP) 1.0-rc1
All Classes
Packages
jcsp.awt
jcsp.lang
Any2OneCallChannel
Any2OneChannel
Any2OneChannelInt
Barrier
BlackHoleChannel
BlackHoleChannelInt
Bucket
Crew
Guard
One2AnyCallChannel
One2AnyChannel
One2AnyChannelInt
One2OneCallChannel
One2OneChannel
One2OneChannelInt
Parallel
PriParallel
ProcessManager
Overview Package Class Tree Deprecated Index Help
PREV NEXT FRAMES NO FRAMES
CSP for Java (JCSP) 1.0-rc1
CSP for Java™ (JCSP) 1.0-rc1 API Specification
This document is the specification for the JCSP core API.
See:
Description
Packages
jcsp.awt
This provides CSP extensions for all java.awt components -- GUI events and widget configuration map to channel communications.
jcsp.lang
This provides classes and interfaces corresponding to the fundamental primitives of CSP.
jcsp.plugNplay
This provides an assortment of plug-and-play CSP components to wire together (with Object-carrying wires) and reuse.
jcsp.plugNplay.ints
This provides an assortment of plug-and-play CSP components to wire together (with int-carrying wires) and reuse.
jcsp.util
This provides classes and interfaces to customise the semantics of Object channels.
jcsp.util.ints
This provides classes and interfaces to customise the semantics of int channels.
Document: Done