CSP Networking for Java (JCSP.net)

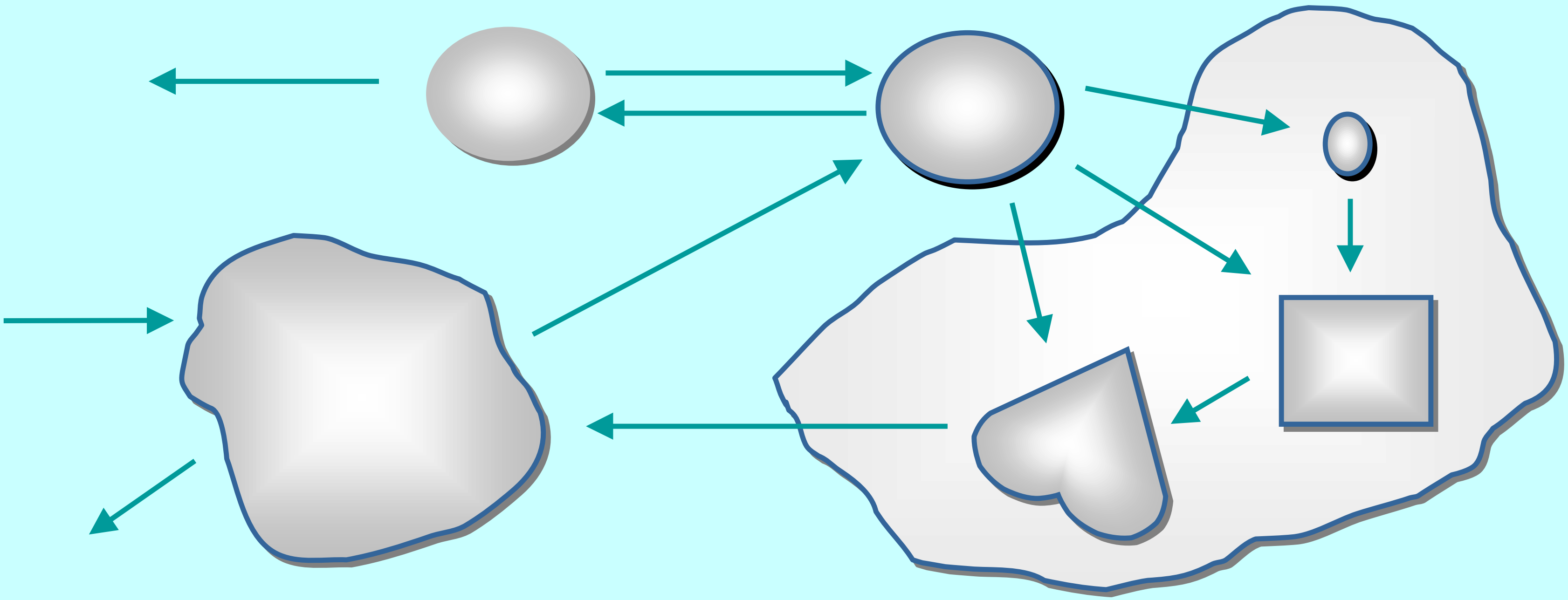
Jo Aldous (jra@dial.pipex.com)
Jon Foster (jon@jon-foster.co.uk)
Peter Welch (phw@ukc.ac.uk)

Computing Laboratory
University of Kent at Canterbury

ICCS 2002 (Global and Collaborative Computing, 22nd. April, 2002)

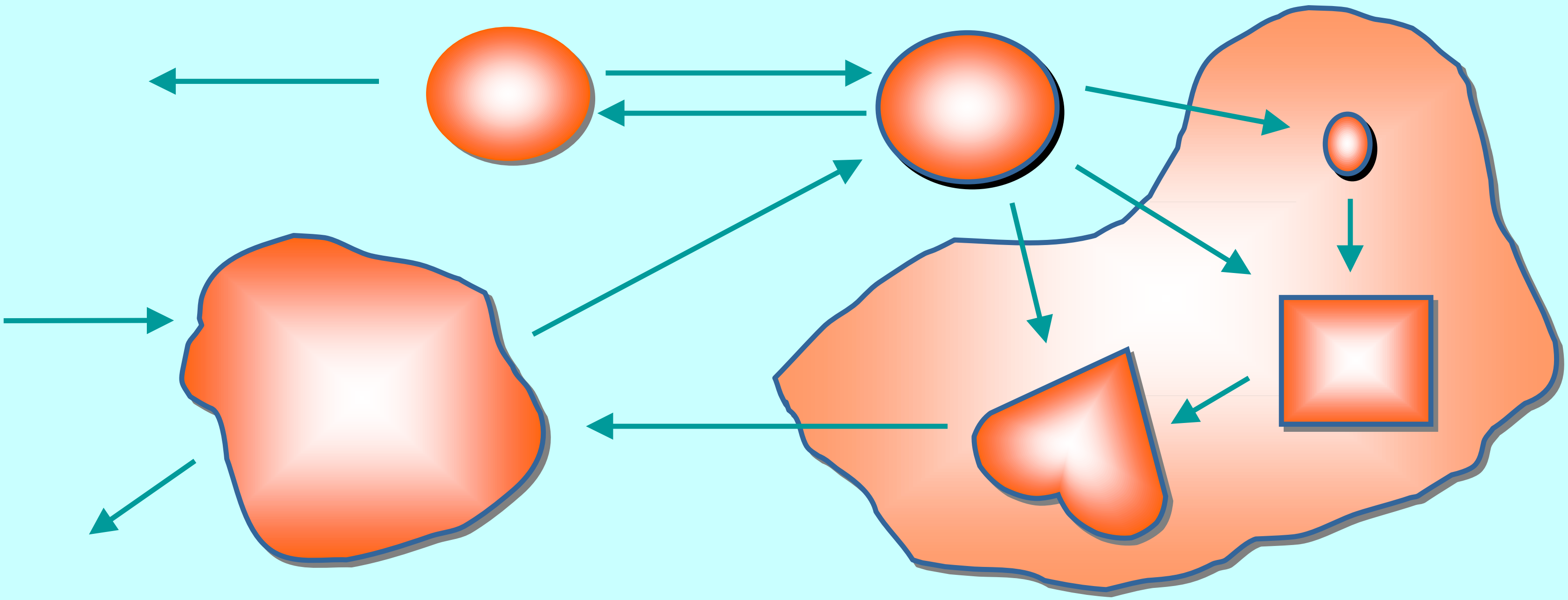
Nature has very large numbers of independent agents, interacting with each other in regular and chaotic patterns, at all levels of scale:

… nuclear … human … astronomic …



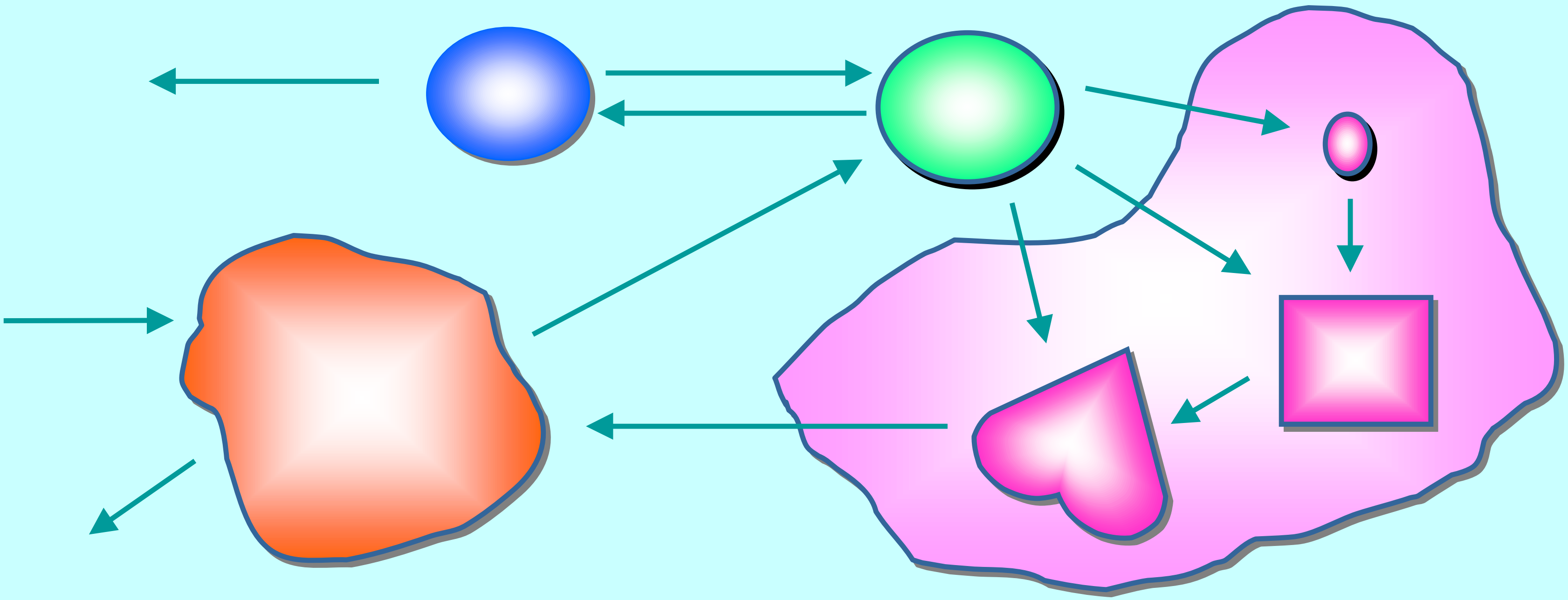
JCSP enables the dynamic construction of layered networks of communicating and synchronising processes (*CSP/occam*):

… natural design within a single JVM



JCSP.net enables the dynamic construction of layered networks of communicating and synchronising processes (*CSP/occam*):

… with the processes distributed over many JVMs



This Presentation

- **Introduction to JCSP**
  - **What is it?**
  - **A few details (with examples)**
- **JCSP.net**
  - ***Virtual* Channels**
  - ***Links* and *Channel Name Server***
  - ***Connections* (2-way extended transactions)**
  - ***Anonymous* Network Channels and Connections**
  - ***Process Farms* and *Chains* (including *Rings*)**
  - ***User-Defined Brokers* (and *Scaleable Parallel Servers*)**
  - ***Remote Process Launching***
  - ***Mobile Processes* (*Agents*) / *Channel Migration***
- **Summary**

JCSP – What is it?

- **JCSP** provides the Java programmer with a process model based upon **occam** and **CSP**:
  - ***Layered networks of encapsulated processes;***
  - ***Processes communicate using channels:***
    - One-to-One / Any-to-One / One-to-Any / Any-to-Any
    - optional buffering (finite / overwriting / infinite)
    - Call Channels / Connections (2-way transactions)
    - Barriers / Buckets / CREW locks
- The current library offers this only within a single JVM (which may, of course, be multi-processor).

JCSP – a few details

- **JCSP** provides and implements an API for Java giving interfaces and classes corresponding to the *fundamental* operators and processes of **CSP** (as well as some *higher-level* mechanisms built on top of those **CSP** primitives).
- A ***process*** is an object of a class implementing:

```
interface CSProcess {
  public void run();
}
```

- The behaviour of the process is determined by the body of its `run()` method.

Copyright P.H.Welch

JCSP – a few details

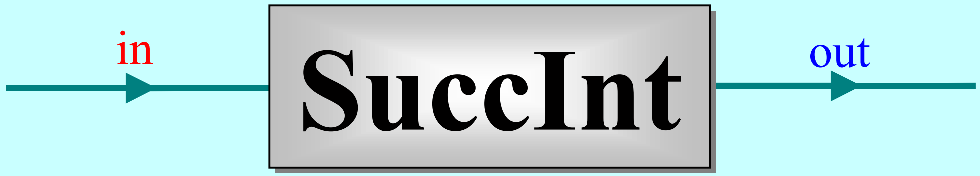
- Channels are accessed via two interfaces:

```
interface ChannelInput {
  public Object read ();
}

interface ChannelOutput {
  public void write (Object obj);
}
```

- The `Parallel` class provides the CSP parallel operator.
- The `Alternative` class provides occam-like ***ALT**ing* (which is a mix of CSP *external / internal choice*).
- `CSTimer` provides *timeout* guards for `Alternative`s.

JCSP Process Structure

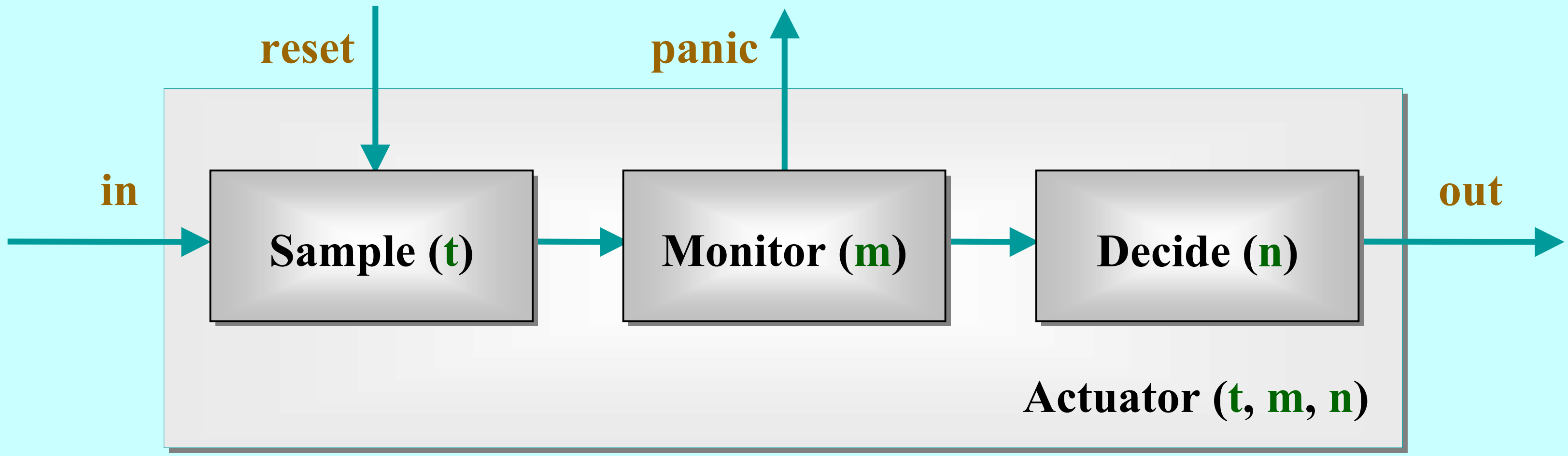
```
class Example implements CSProcess {

   ...  private shared synchronisation objects
        (channels etc.)
   ...  private state information

   ...  public constructors
   ...  public accessors(gets)/mutators(sets)
        (only to be used when not running)

   ...  private support methods (part of a run)
   ...  public void run() (process starts here)

}
```



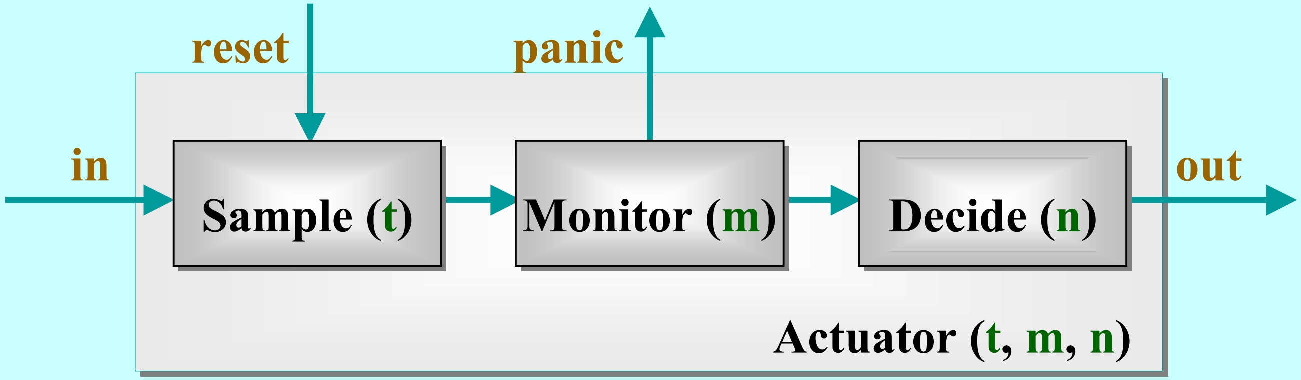
This is a simple process that adds one to each integer flowing through it.

```
class SuccInt implements CSProcess {

  private final ChannelInputInt in;
  private final ChannelOutputInt out;

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

  public void run () {
    while (true) {
      int n = in.read ();
      out.write (n + 1);
    }
  }

}
```

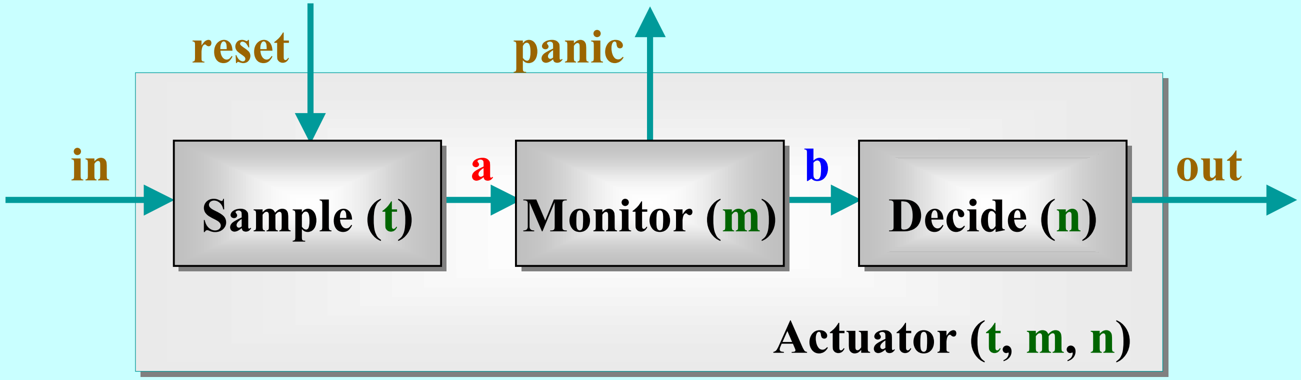
Final Stage Actuator



- **`Sample(t)`:** *every* `t` time units, output the *latest* input (or `null` *if none*); the value of `t` may be reset;
- **`Monitor(m)`:** copy input to output counting `nulls` - if `m` `nulls` occur *in a row*, send panic message and terminate;
- **`Decide(n)`:** copy non-`null` input to output and *remember* last `n` outputs - convert `nulls` to a *best guess* depending on those last `n` outputs.

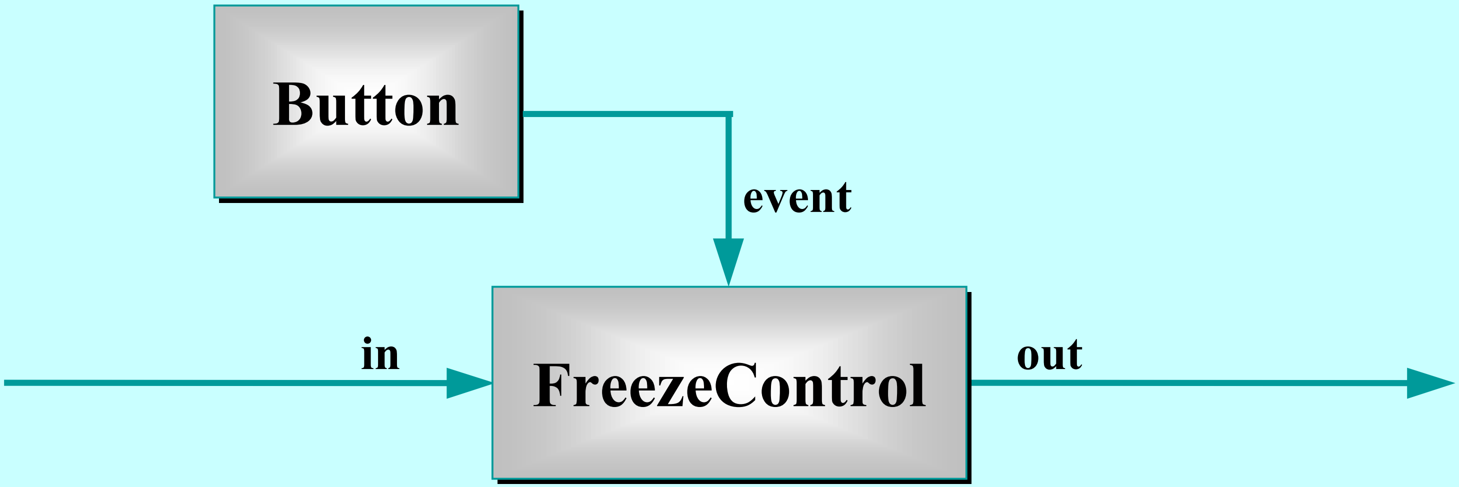



```
class Actuator implements CSProcess {

  ...  private state (t, m and n)

  ...  private interface channels
       (in, reset, panic and out)

  ...  public constructor
       (assign parameters t, m, n, in, reset,
        panic and out to the above fields)

  ...  public void run ()

}
```




```
public void run ()

  final One2OneChannel a = new One2OneChannel ();
  final One2OneChannel b = new One2OneChannel ();

  new Parallel (
    new CSProcess[] {
      new Sample (t, in, reset, a),
      new Monitor (m, a, panic, b),
      new Decide (n, b, out)
    }
  ).run ();

}
```

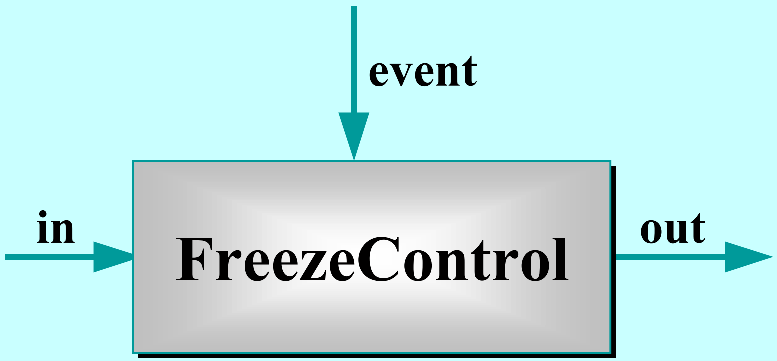
ALTing Between Events



- **`Button`** is a (GUI widget) process that outputs a *ping* whenever it's clicked.
- **`FreezeControl`** controls a data-stream flowing from its **`in`** to **`out`** channels. Clicking the **`Button`** freezes the data-stream - clicking again resumes it.

Copyright P.H.Welch

ALTing Between Events

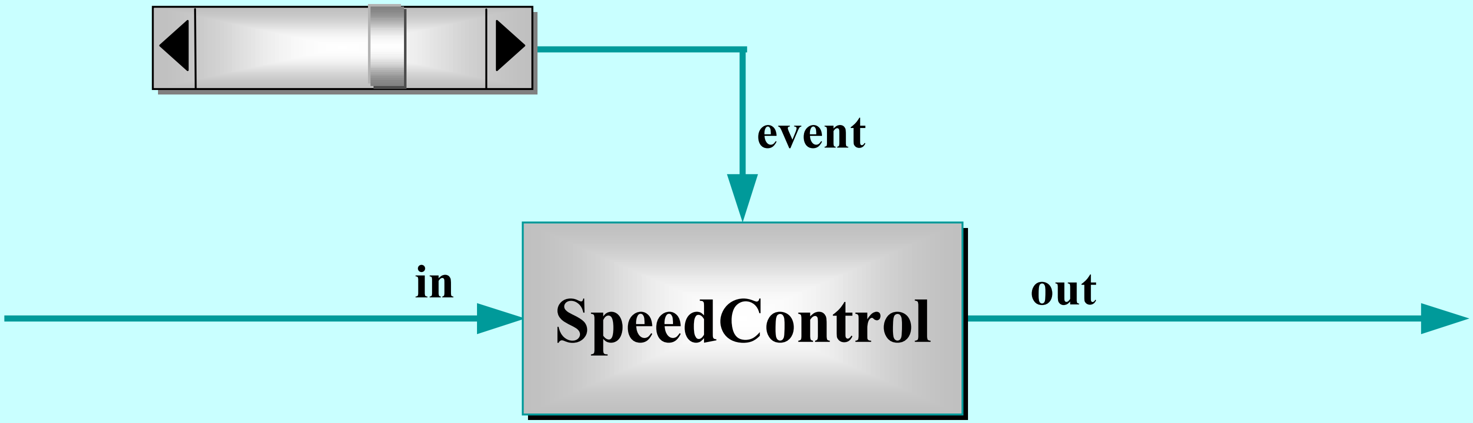



```
final Alternative alt =
  new Alternative (
    new Guard[] {event, in};
  );
final int EVENT = 0, IN = 1;
```

```
while (true) {
  switch (alt.priSelect ()) {
    case EVENT:
      event.read ();
      event.read ();
    break;
    case IN:
      out.write (in.read ());
    break;
  }
}
```

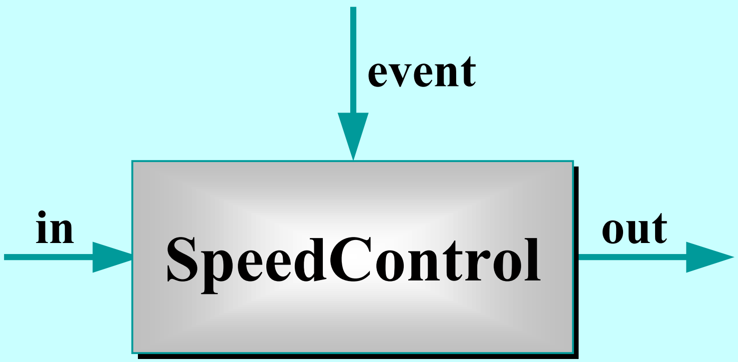
No ***SPIN***

ALTing Between Events



- The *slider* (GUI widget) process outputs an integer (**`0..100`**) whenever its *slider-key* is moved.
- **`SpeedControl`** controls the speed of a data-stream flowing from its **`in`** to **`out`** channels. Moving the *slider-key* changes that speed - from frozen (**`0`**) to some defined maximum (**`100`**).

ALTing Between Events



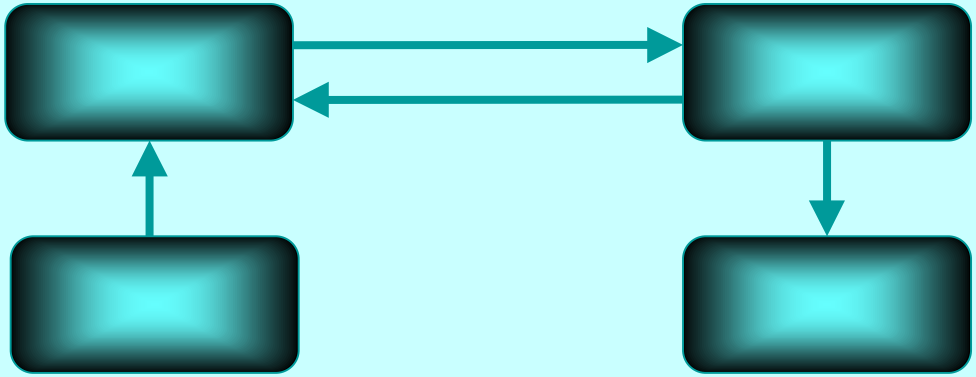

```
final CSTimer tim =
  new CSTimer ();
final Alternative alt =
  new Alternative (
    new Guard[] {event, tim};
  );
final int EVENT = 0, TIM = 1;
```

```
while (true) {
  switch (alt.priSelect ()) {
    case EVENT:
      int position = event.read ();
      while (position == 0) {
        position = event.read ();
      }
      speed = (position*maxSpd)/maxPos
      interval = 1000/speed;   // ms
      timeout = tim.read ();
      // fall through
    case TIM:
      timeout += interval;
      tim.setAlarm (timeout);
      out.write (in.read ());
    break;
  }
}
```

No ***SPIN***

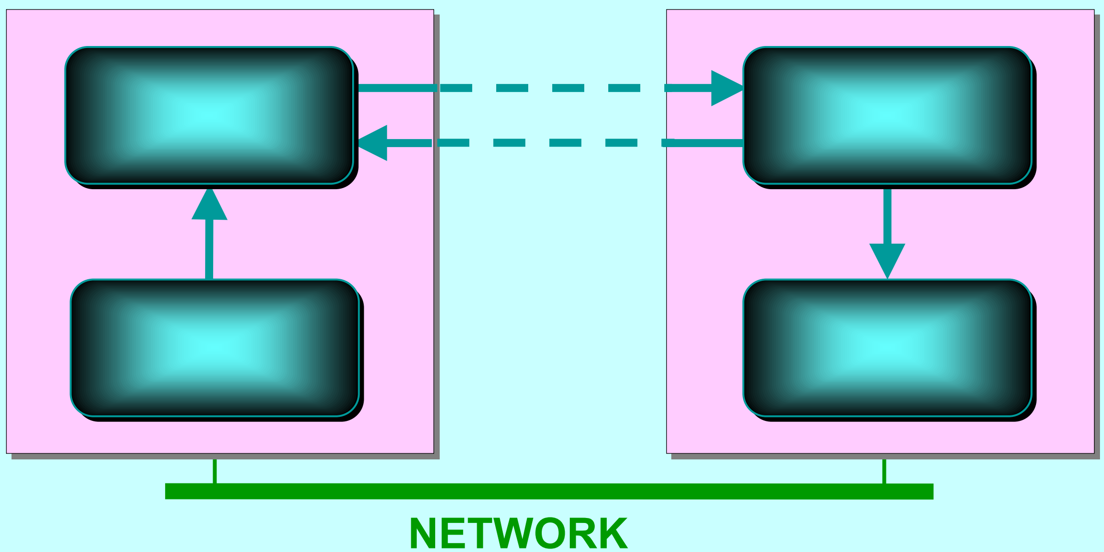
Distributed JCSP

- Want to use the same model for concurrent processes *whether or not they are on the same machine*:



Distributed JCSP

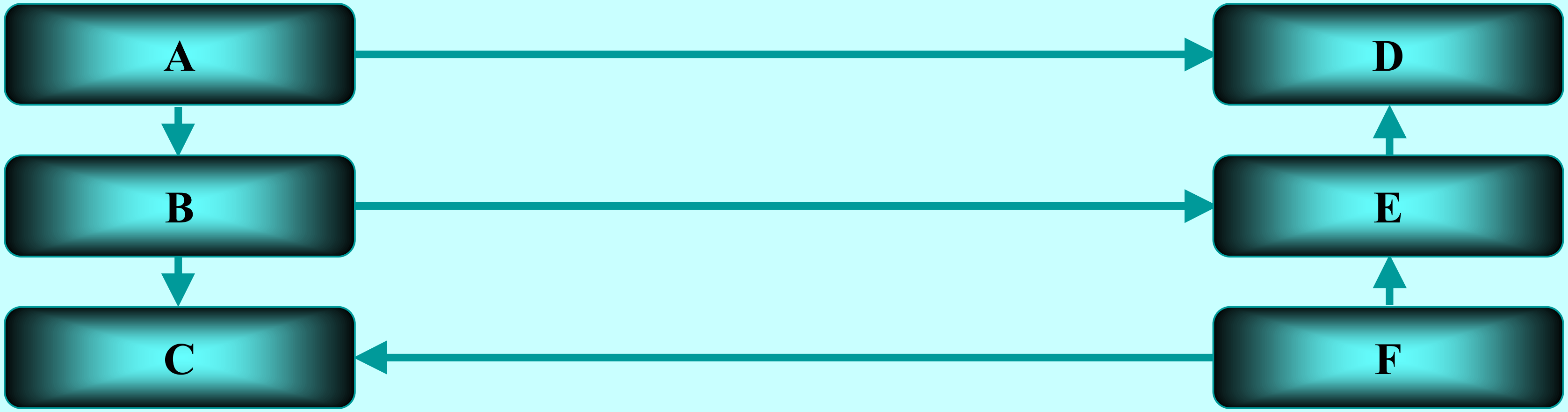
- Want to use the same model for concurrent processes *whether or not they are on the same machine*:



- Processes on different processing *nodes* communicate via *virtual channels*.

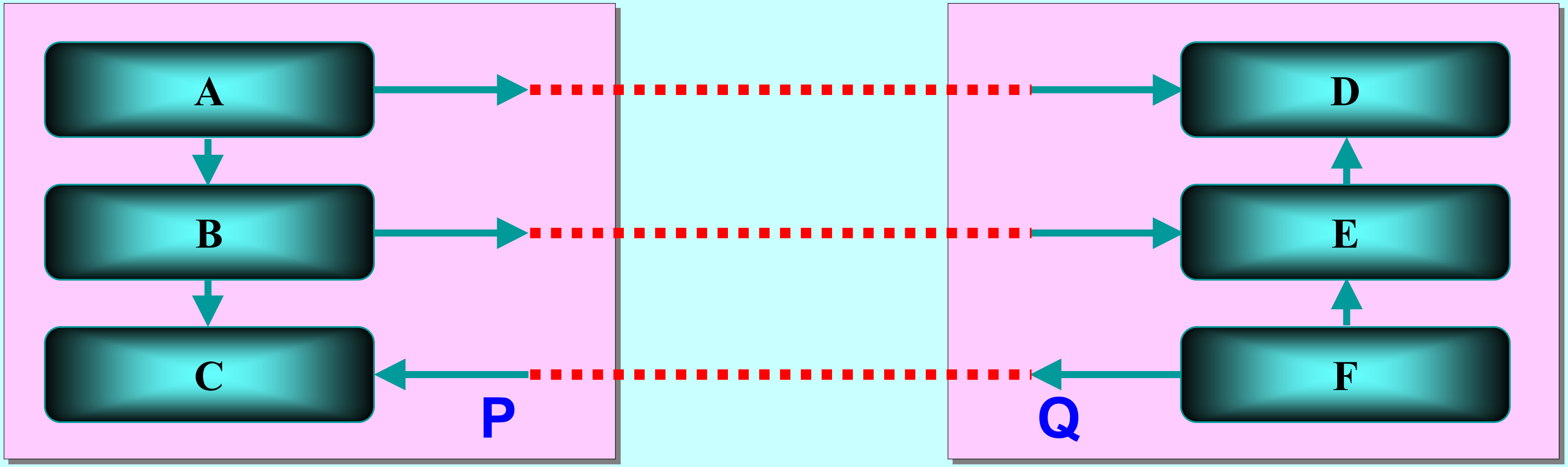
Logical Network

- Suppose a system contains processes **`A`**, **`B`**, **`C`**, **`D`**, **`E`** and **`F`**, communicating as shown below.
- There may be other processes and communication channels (but they are not relevant here).
- Suppose we want to distribute these processes over two processors …



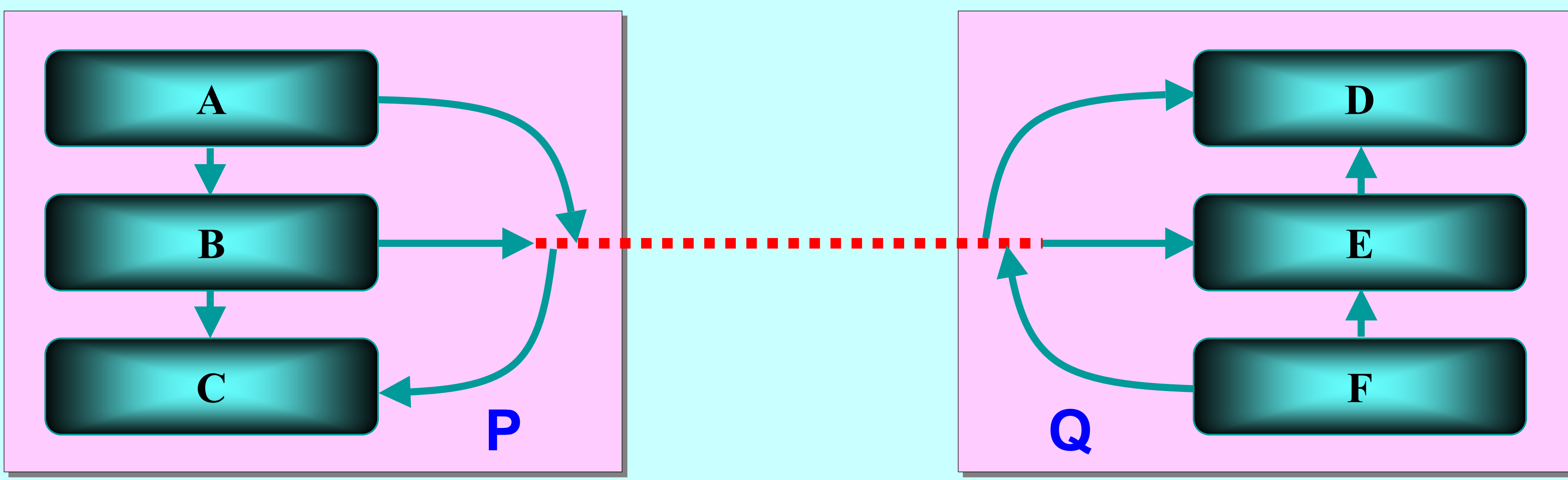
Physical Network

- Suppose we want to distribute these processes over two processors (**P** and **Q**, say) …
- We could set up separate network links …
- Or, since links may be a scarce resource, we could multiplex over a shared link …



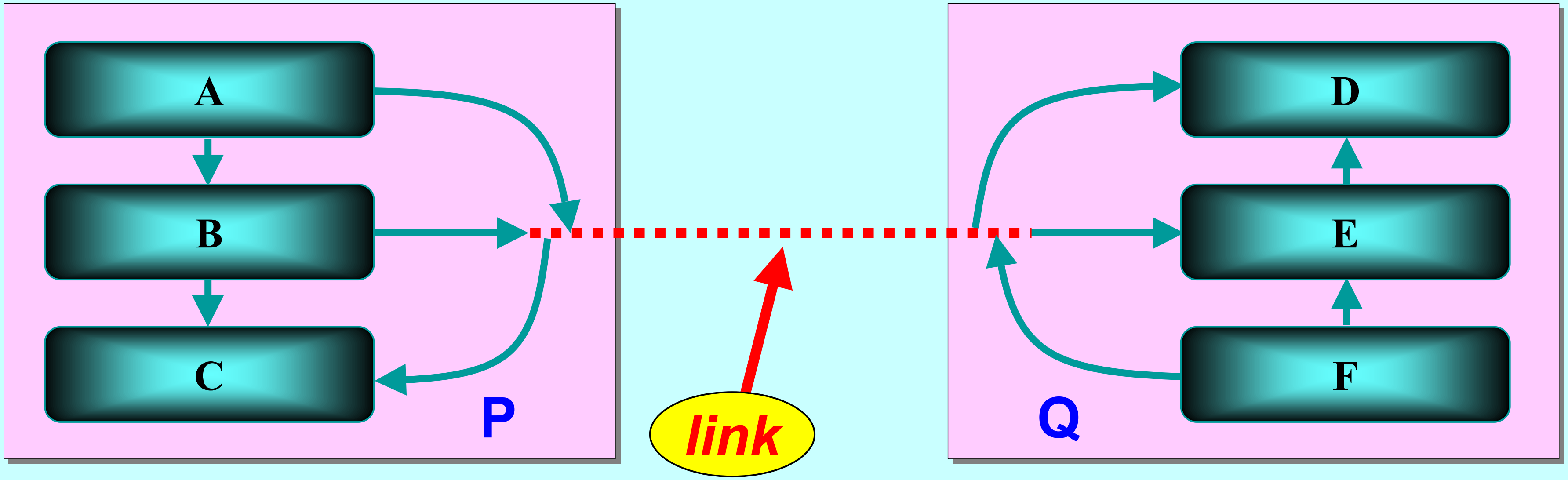
Physical Network

- Suppose we want to distribute these processes over two processors (**P** and **Q**, say) …
- We could set up separate network links …
- Or, since links may be a scarce resource, we could multiplex over a shared link …



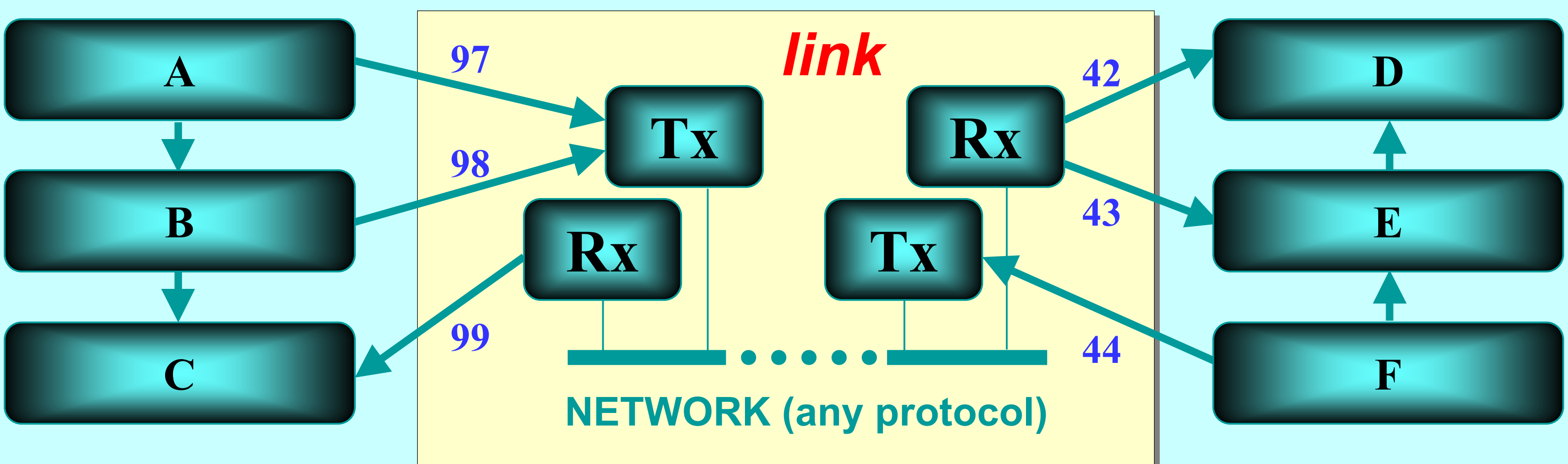
JCSP Links

- A connection between two processing nodes (*JVMs* in the context of **JCSP**) is called a ***link***.
- Multiple channels between two nodes may use the same link – data is multiplexed in both directions.
- Links can ride on any network infrastructure (*TCP/IP*, *Firewire*, *1355*, …).



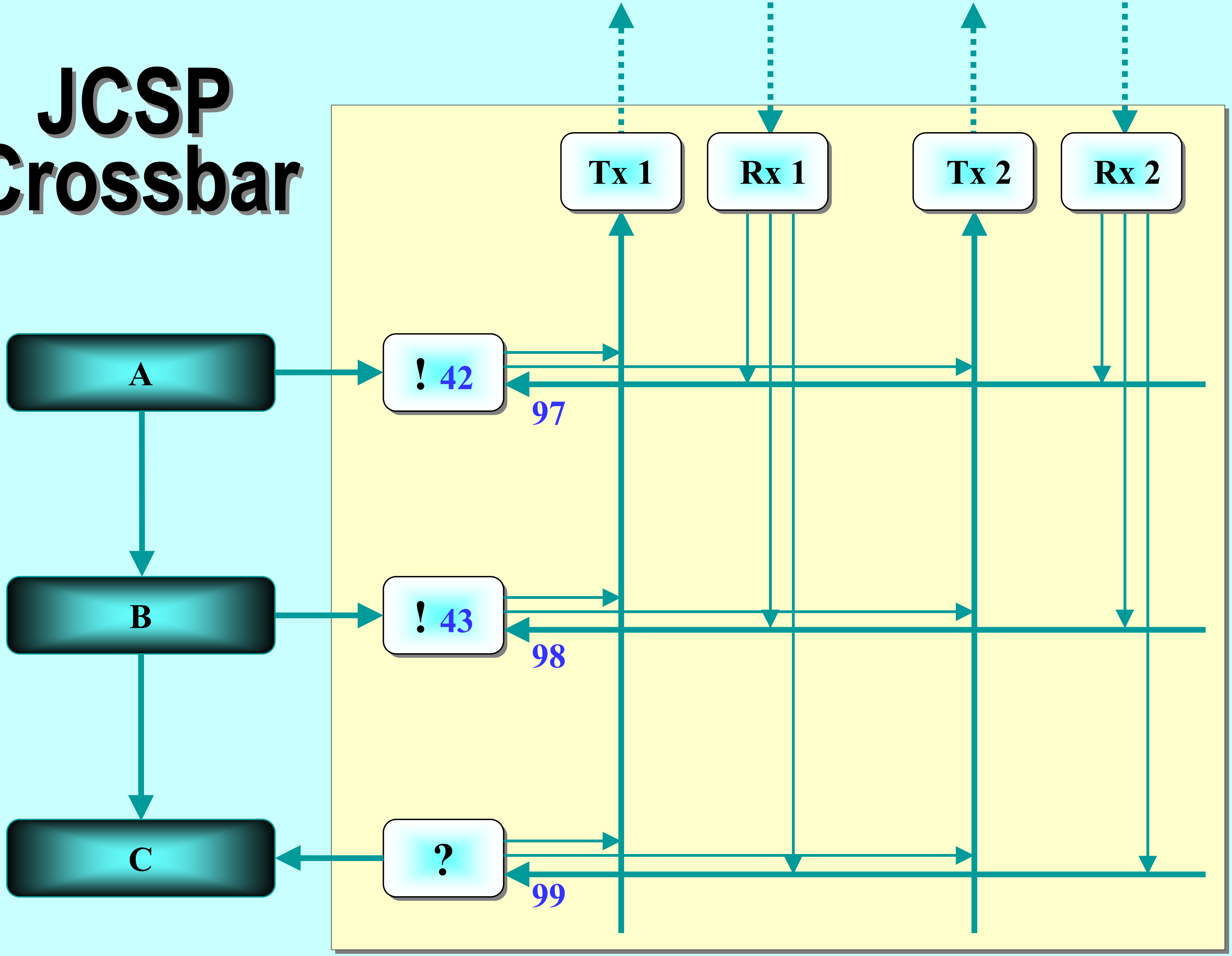
JCSP Links

- Each end of a (e.g. *TCP/IP*) network channel has a network address (e.g. <*IP-address, port-number*>) and **JCSP** ***virtual-channel-number*** (see below).
- **JCSP** uses the channel-numbers to multiplex and de-multiplex data and acknowledgements.
- The ***JCSP.net*** programmer sees none of this.



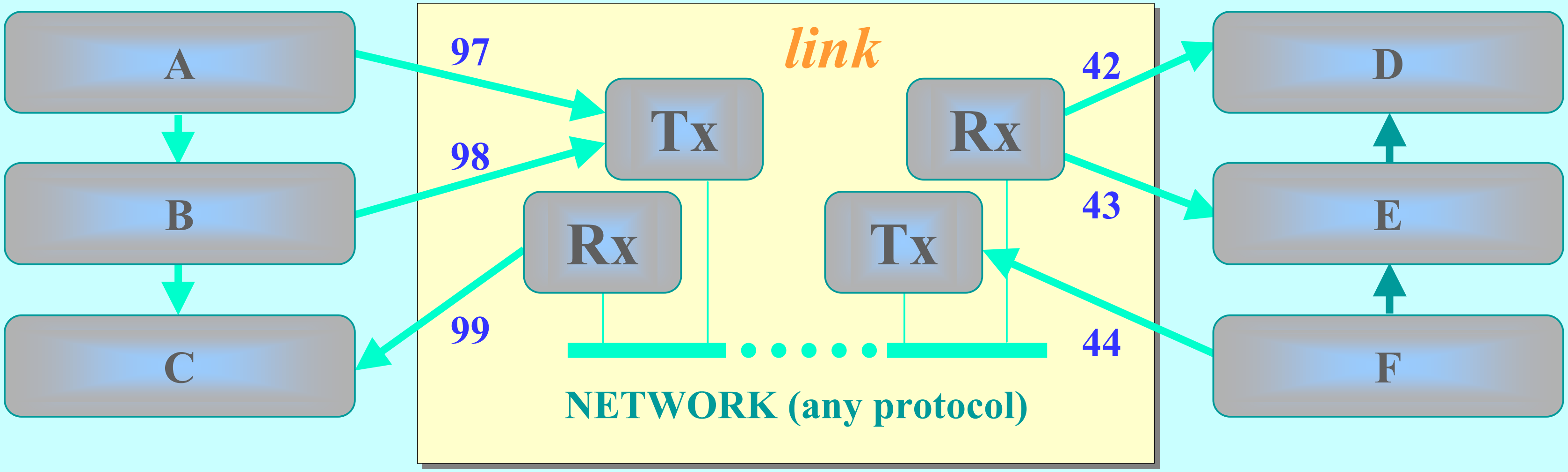
Copyright P.H.Welch

JCSP Crossbar



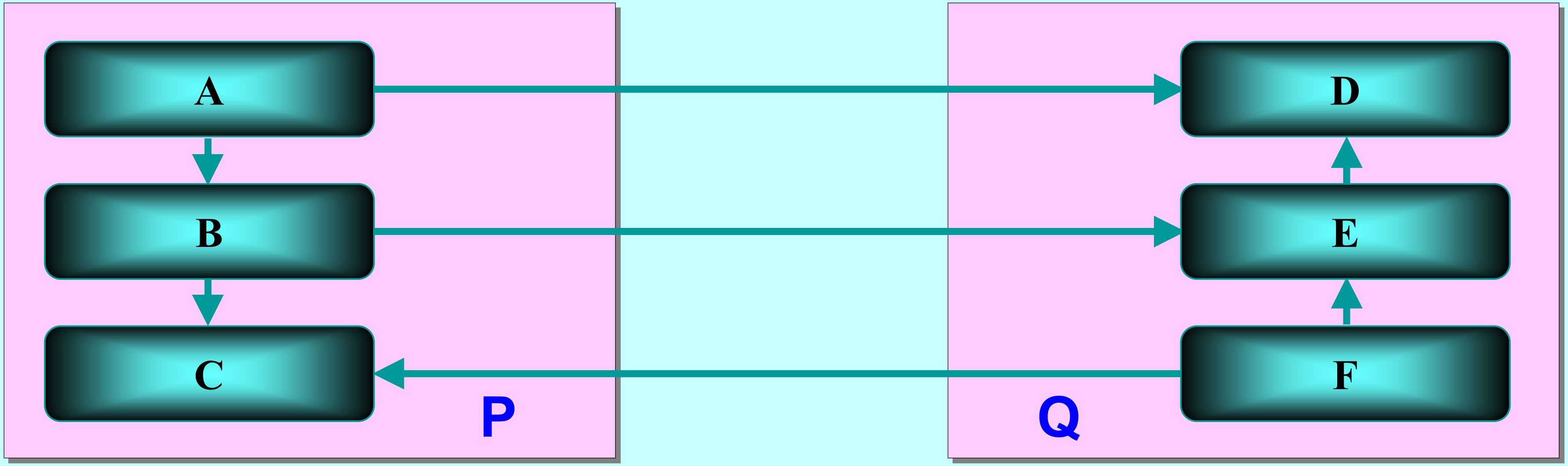
JCSP Links

- Each end of a (e.g. *TCP/IP*) network channel has a network address (e.g. <*IP-address, port-number*>) and JCSP *virtual-channel-number* (see below).
- JCSP uses the channel-numbers to multiplex and de-multiplex data and acknowledgements.
- **The *JCSP.net* programmer sees none of this.**



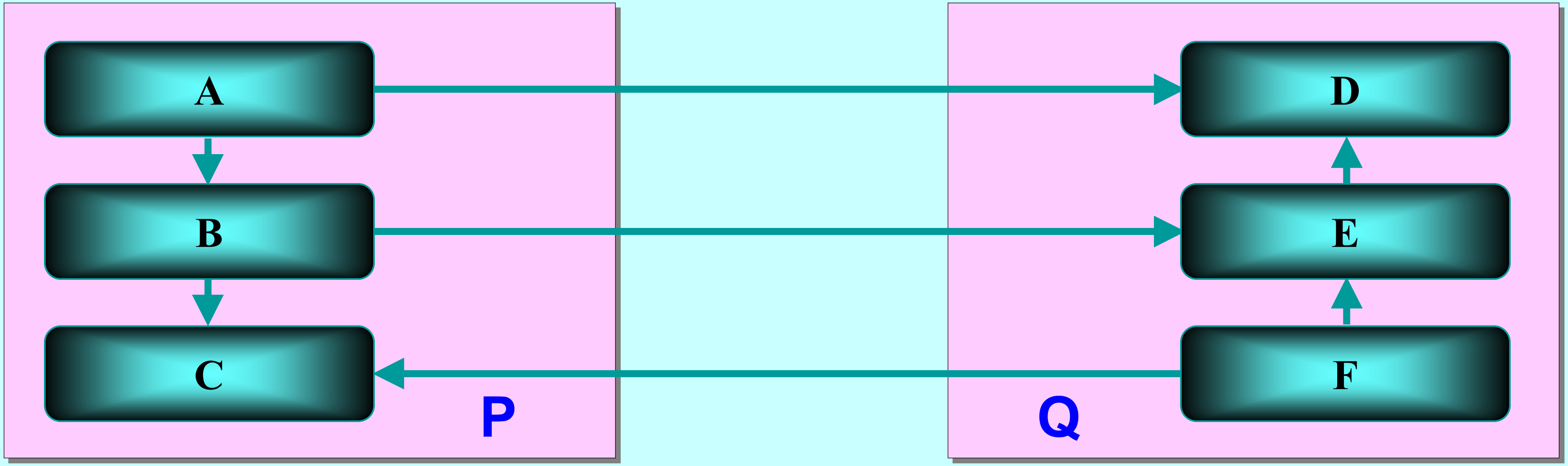
JCSP Networks

- **The *JCSP.net* programmer just sees this.**
- Channel synchronisation semantics for network channels are *exactly the same* as for internal ones.
- Buffered network channels can be *streamed* - i.e. network acks can be saved through *windowing*.



JCSP Networks

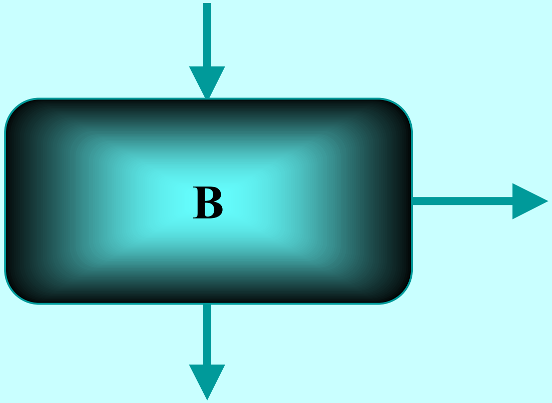
- However, there is one important semantic difference between a network channel and a local channel.
- Over *local* channels, objects are passed by *reference* (which leads to *race hazards* if careless).
- Over *network* channels, objects are passed by *copying* (currently, using Java *serialization*).



JCSP Networks

- That semantic difference will not impact correctly designed **JCSP** systems (i.e. those free from *race hazards*).
- With that *caveat*, **JCSP** processes *are blind* as to whether they are connected to local or network channels.

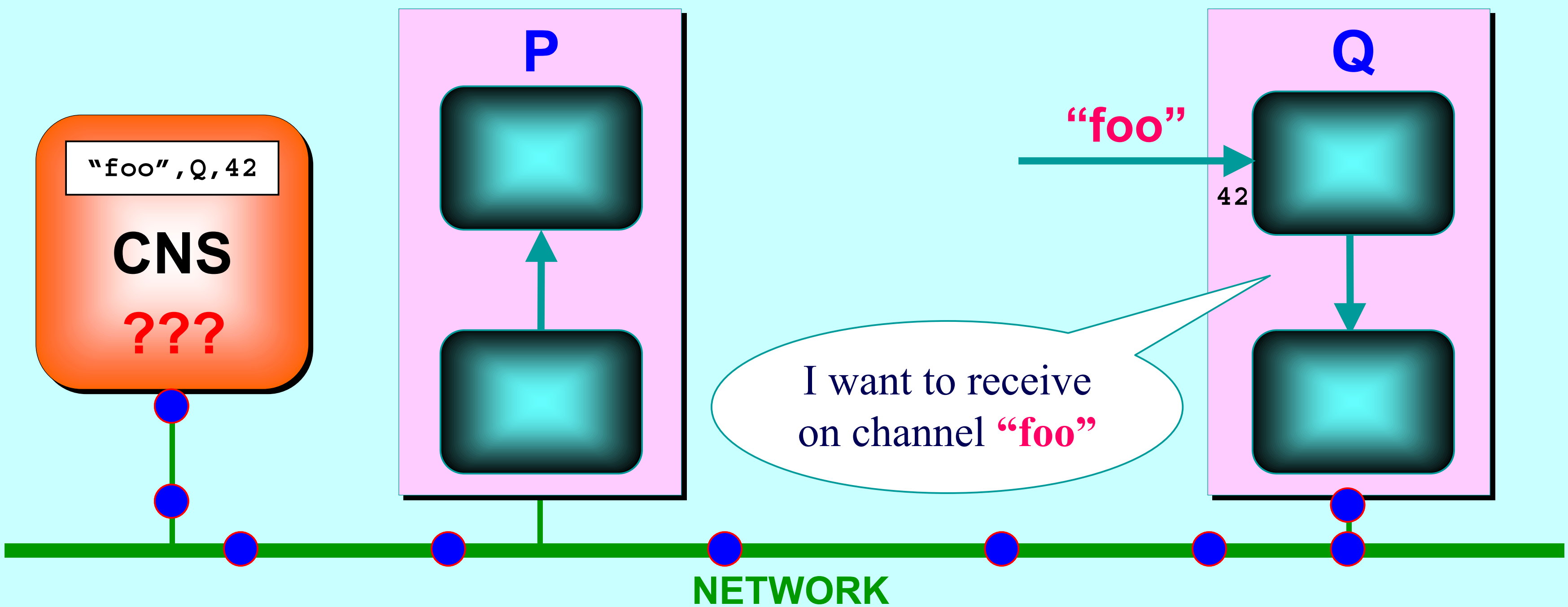
Process `B` still sees its external channels as `ChannelInput` / `ChannelOutput`



- One other *caveat* - currently, only `Serializable` objects are copied over network channels - sorry!

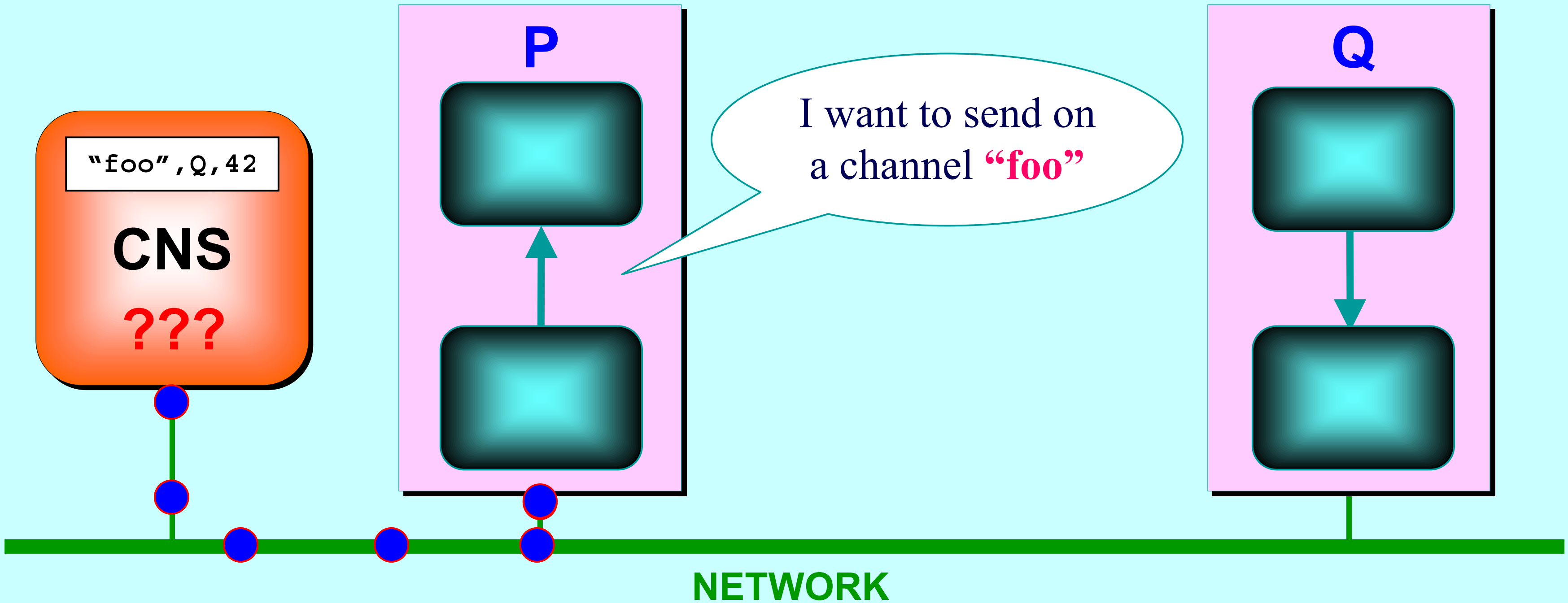
Establishing Network Channels

- Network channels may be connected by the **JCSP** ***Channel Name Server*** (CNS).
- Channel *read ends* register names with the CNS.



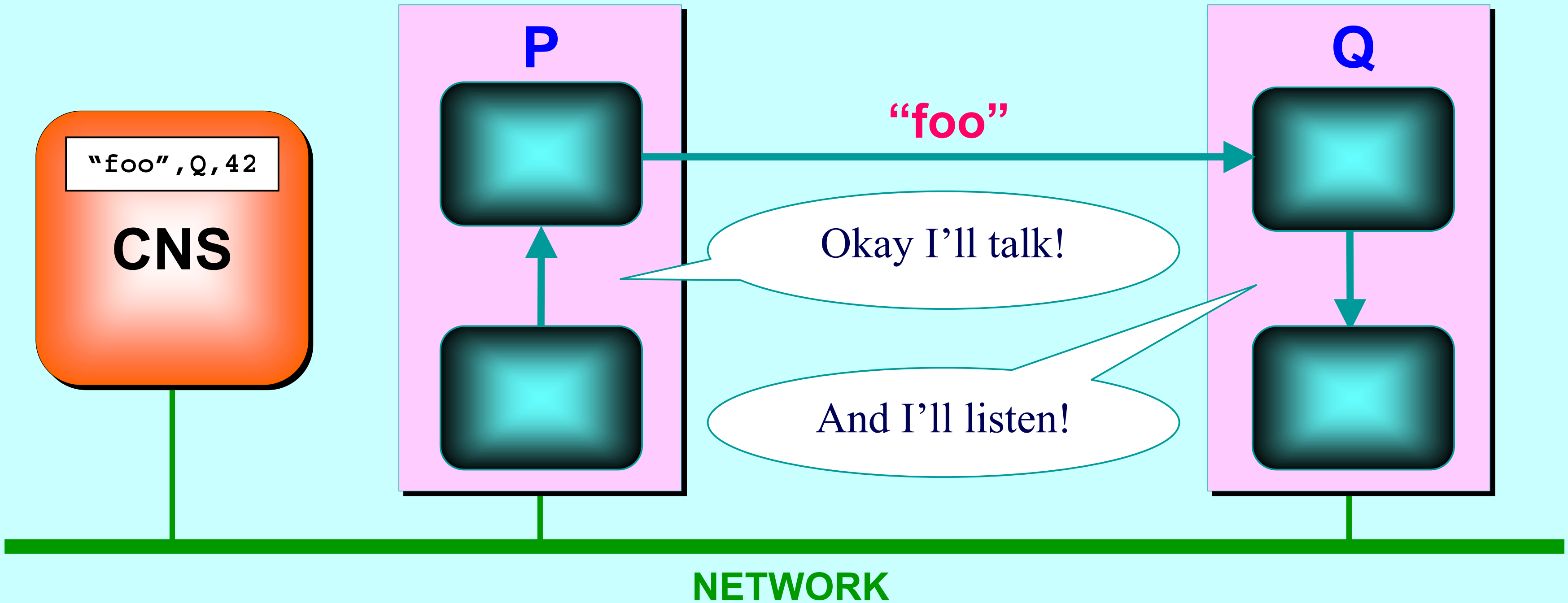
Establishing Network Channels

- Network channels may be connected by the JCSP ***Channel Name Server*** (CNS).
- Channel *read ends* register names with the CNS.
- Channel *write ends* ask CNS about names.



Establishing Network Channels

- Network channels may be connected by the JCSP ***Channel Name Server*** (CNS).
- Channel *read ends* register names with the CNS.
- Channel *write ends* ask CNS about names.



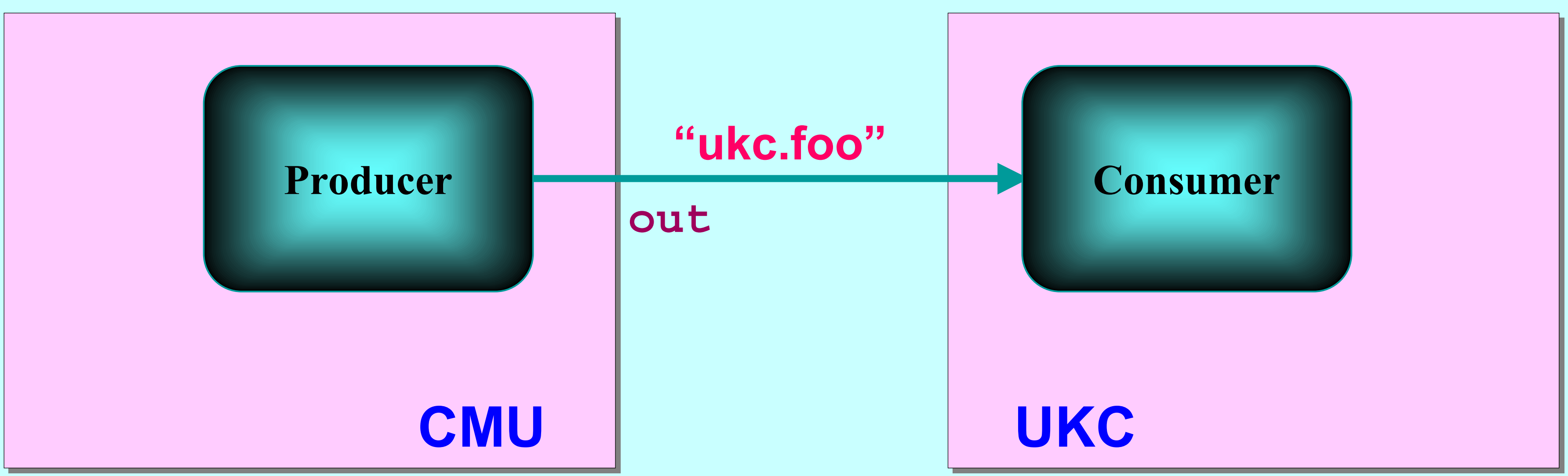
Using Distributed JCSP

- On each machine, do this once:

```
Node.getInstance().init();  // use default CNS
```

- On the **CMU** machine:

```
One2NetChannel out = new One2NetChannel ("ukc.foo");
new Producer (out);
```



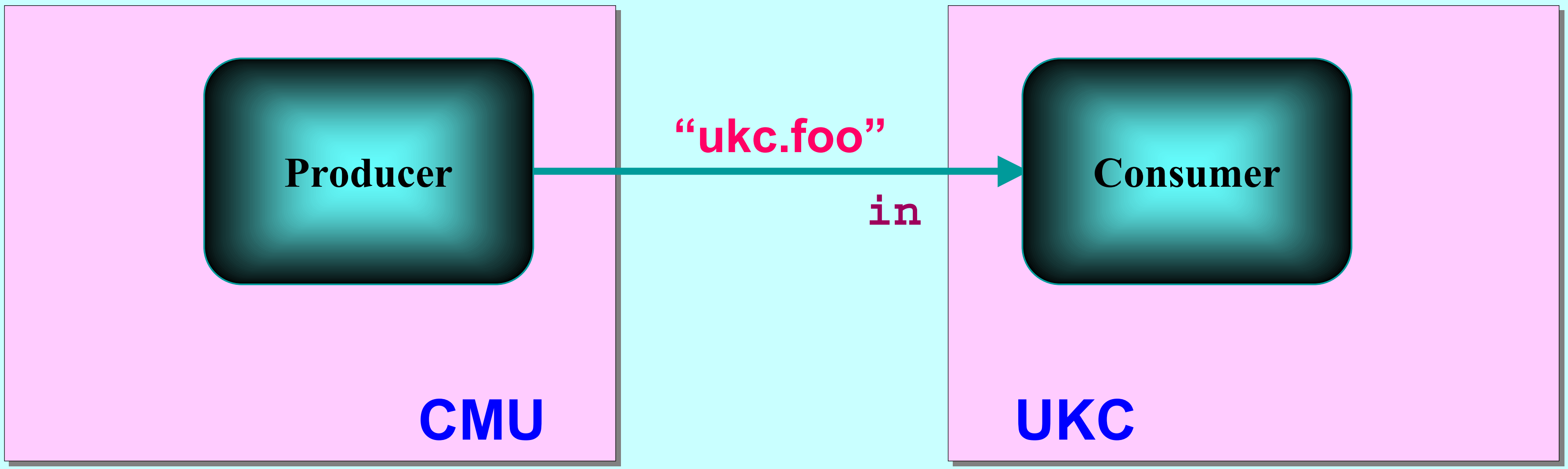
Using Distributed JCSP

- On each machine, do this once:

```
Node.getInstance().init();  // use default CNS
```

- On the **UKC** machine:

```
Net2OneChannel in = new Net2OneChannel ("ukc.foo");
new Consumer (in);
```



Using Distributed JCSP

```
One2NetChannel out = new One2NetChannel ("ukc.foo");
```

Named network *output* channel construction blocks until the name is registered by a reader

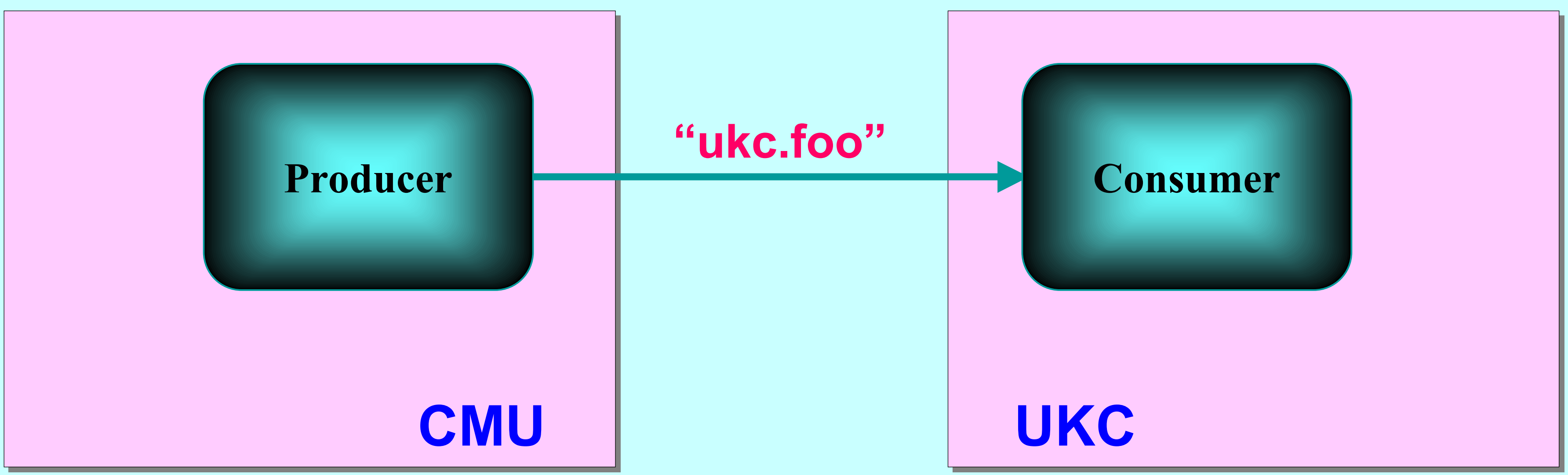
```
Net2OneChannel in = new Net2OneChannel ("ukc.foo");
```

Named network *input* channel construction registers the name with the CNS (will fail if already registered)

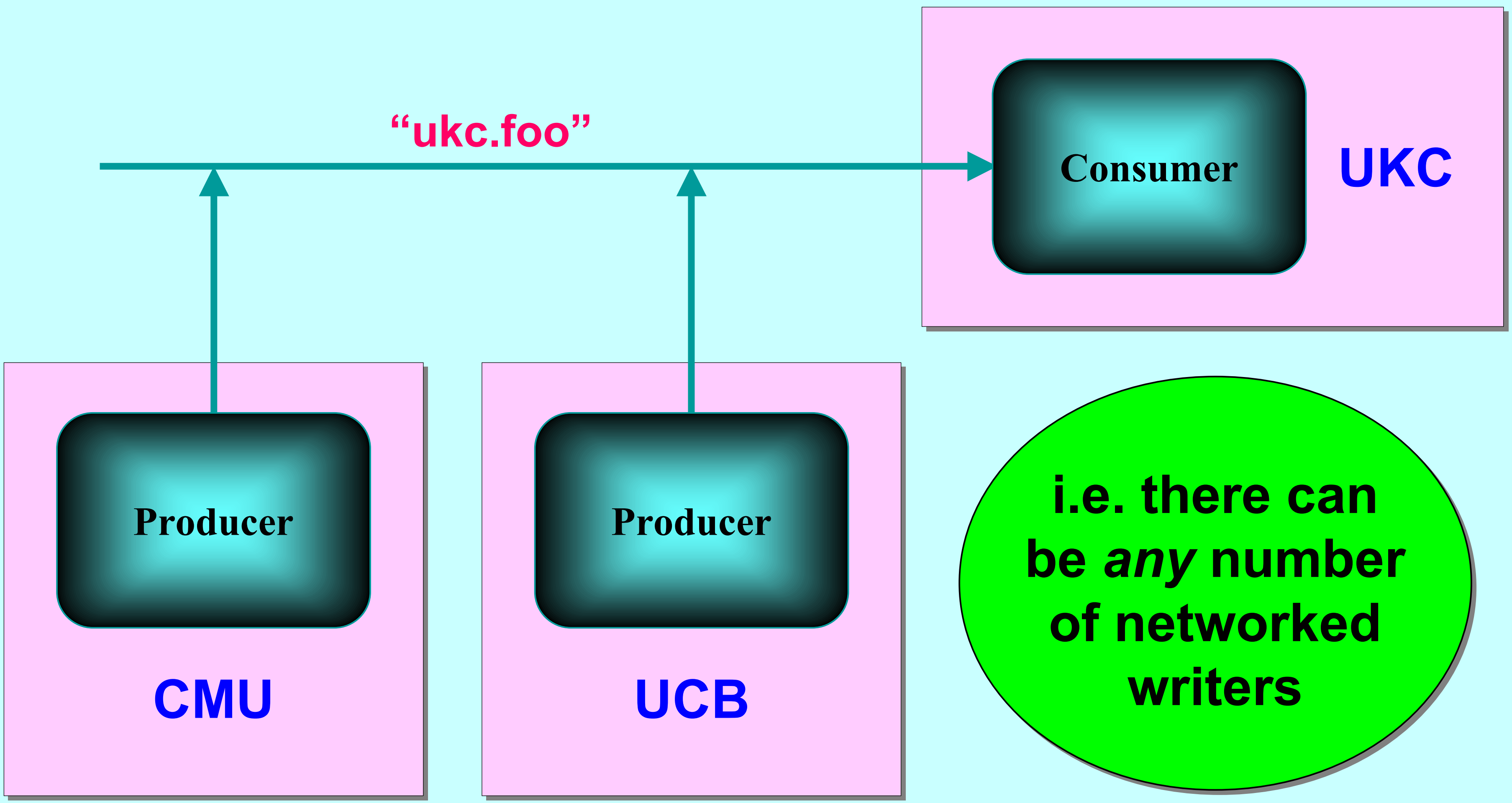
Copyright P.H.Welch

Using Distributed JCSP

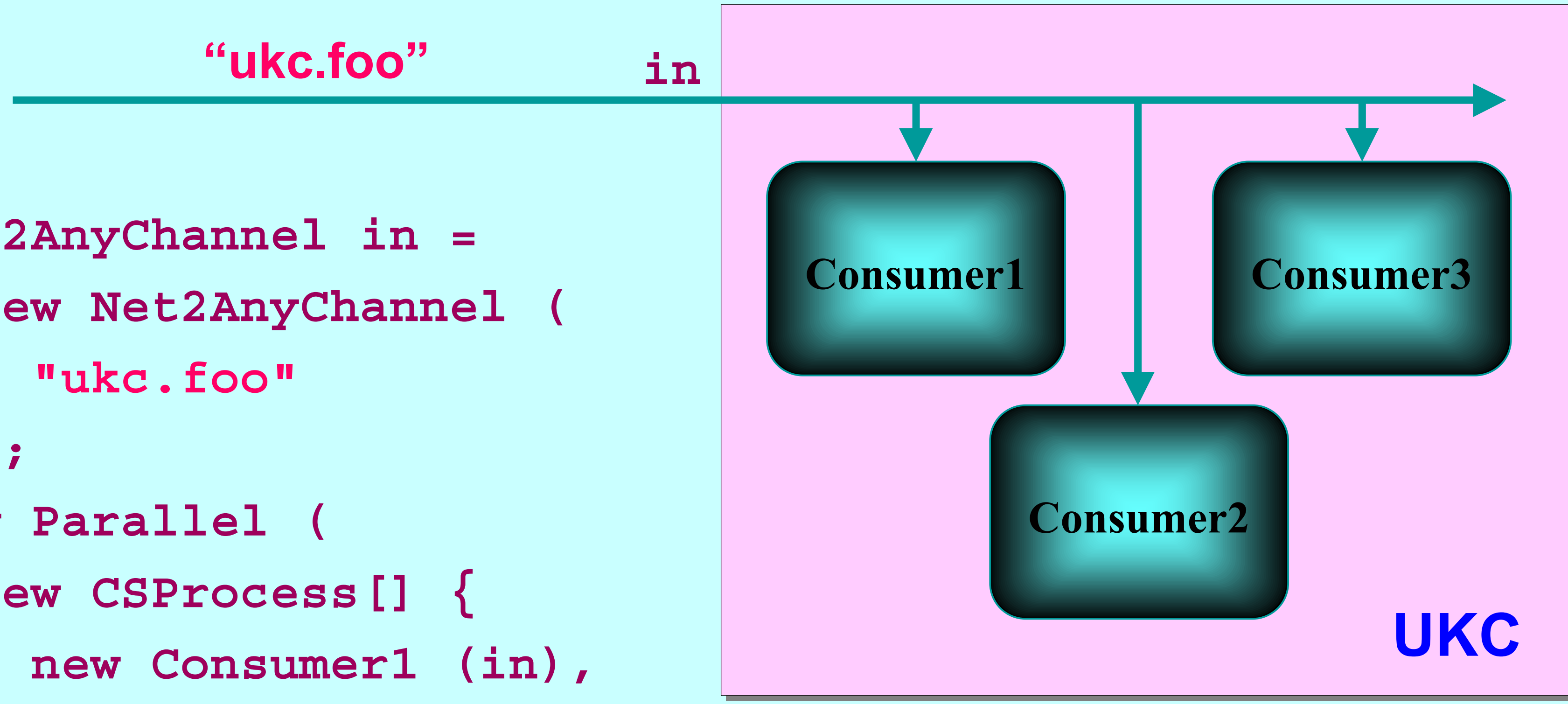
- User processes just have to agree on (or find out) *names* for the channels they will use to communicate.
- User processes do not have to know where each other is located (e.g. *IP-address* / *port-number* / *virtual-channel-number*).



Network Channels are *Any-1*



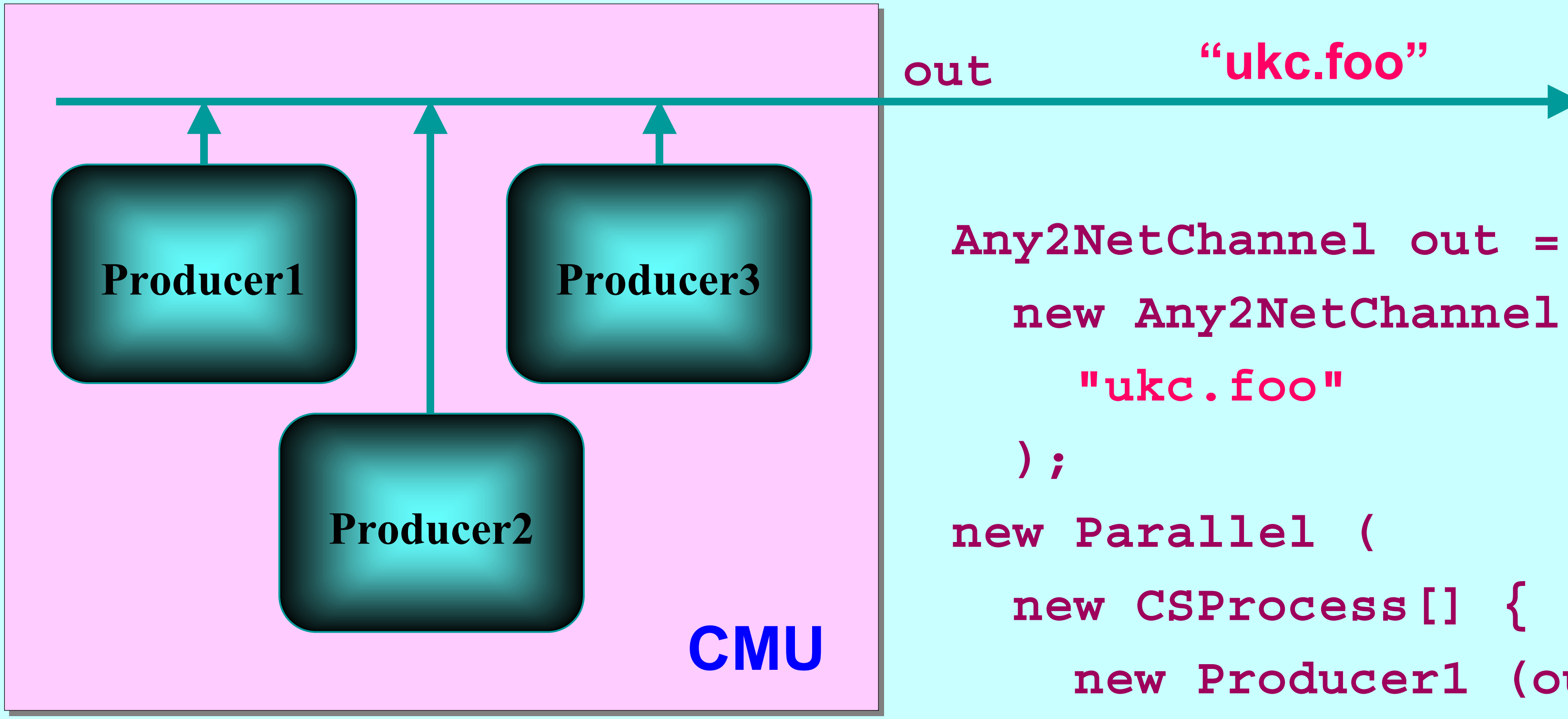
Net-Any Channels




```
Net2AnyChannel in =
  new Net2AnyChannel (
    "ukc.foo"
  );
new Parallel (
  new CSProcess[] {
    new Consumer1 (in),
    new Consumer2 (in),
    new Consumer2 (in)
  }
).run ();
```

i.e. within a node, there can be *any* number of readers

Any-Net Channels




```
Any2NetChannel out =
  new Any2NetChannel (
    "ukc.foo"
  );
new Parallel (
  new CSProcess[] {
    new Producer1 (out),
    new Producer2 (out),
    new Producer3 (out)
  }
).run ();
```

i.e. within a node, there can be *any* number of writers

Connections (*two-way channels*)

- On the **CMU** machine:

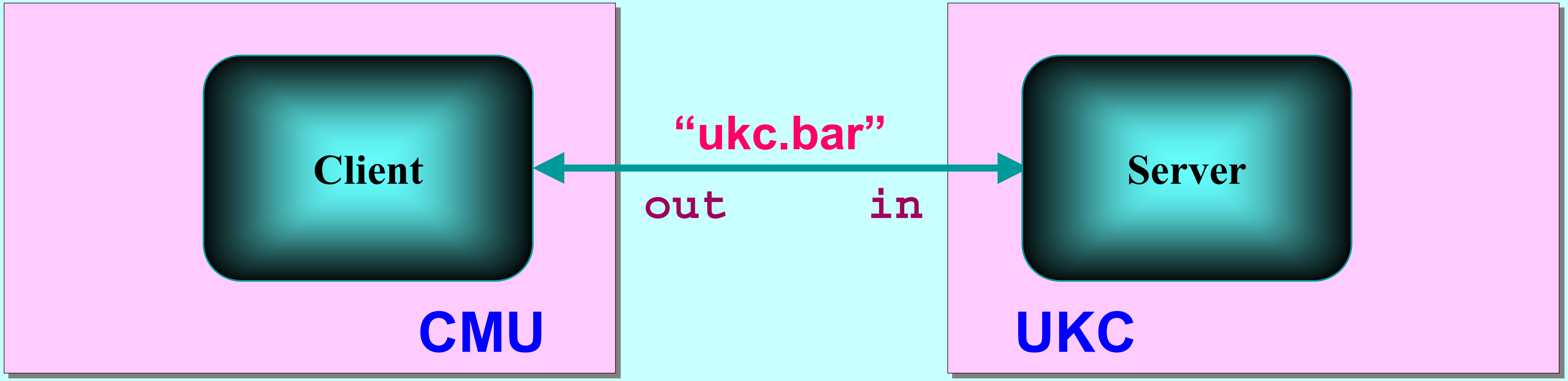
find

```
One2NetConnection out = new One2NetConnection ("ukc.bar");
new Client (out);
```

- On the **UKC** machine:

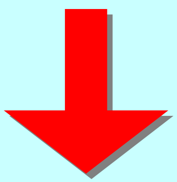
register

```
Net2OneConnection in = new Net2OneConnection ("ukc.bar");
new Server (in);
```

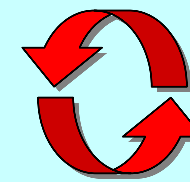


Connections (*two-way channels*)

- Connection channels have *client* and *server* interfaces (rather than *writer* and *reader* ones):

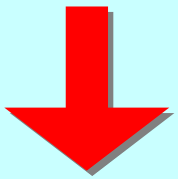


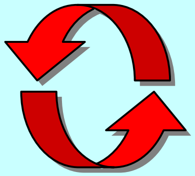
```
interface ConnectionClient {
  public void request (Object o);   // write
  public Object reply ();           // read
  public boolean stillOPen ();      // check?
}
```



Connections (*two-way channels*)

- Connection channels have *client* and *server* interfaces (rather than *writer* and *reader* ones):



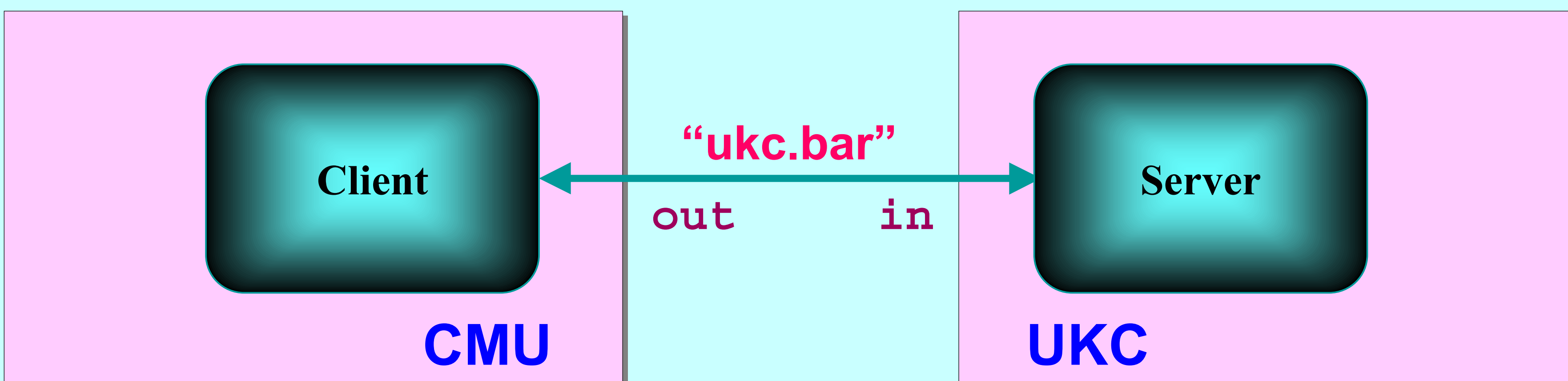


```
interface ConnectionServer {
  public Object request ();            // read
  public void reply (Object o);        // write & close
  public void reply (                  // write &
    Object o, boolean keepOpen         //   maybe close
  );
}
```

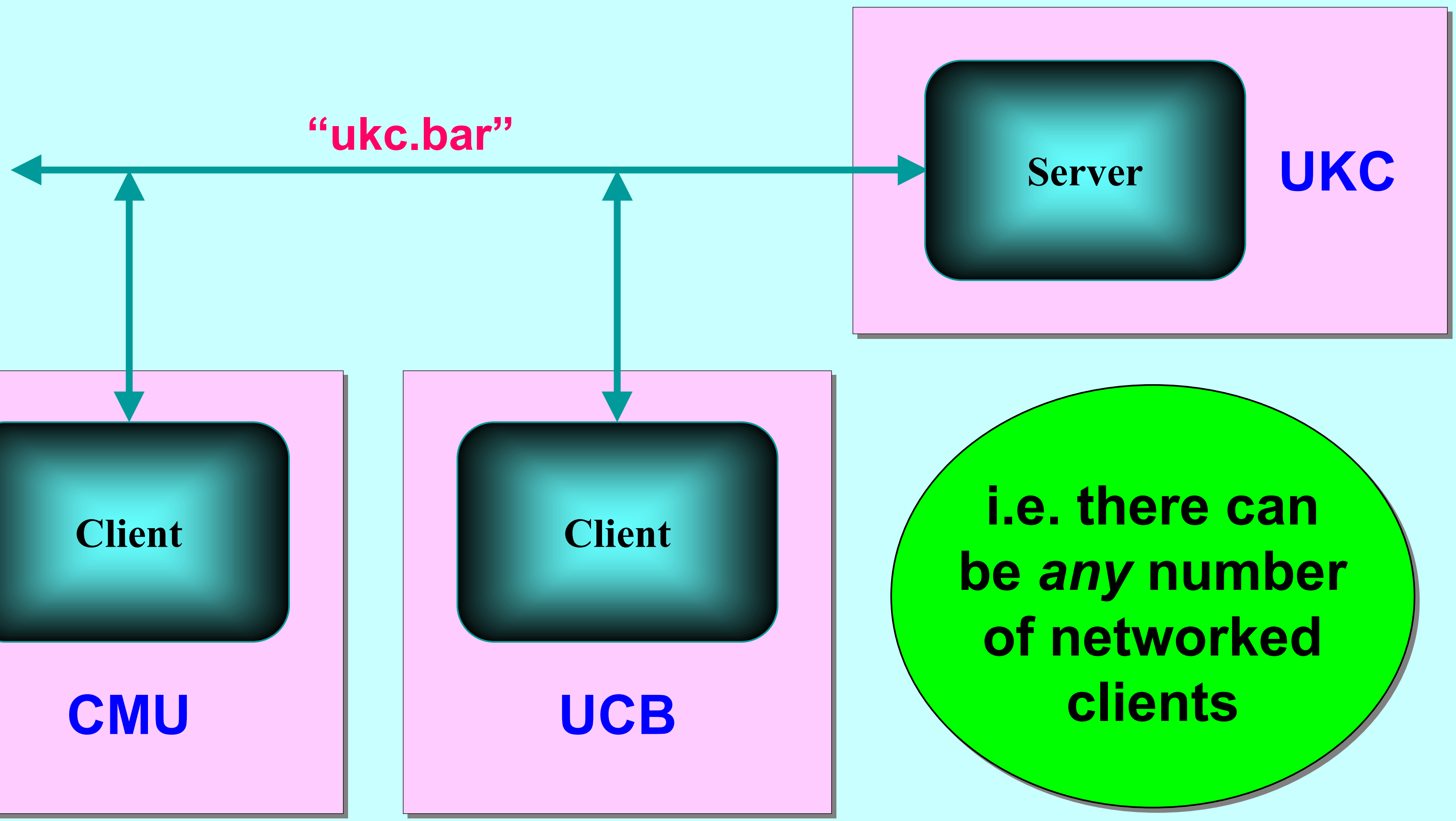
Connections (*extended rendezvous*)

```
out.request (data);
answer = out.reply ();
...  work out followUp
out.request (followUp);
...  more ping/pong
answer = out.reply ();
```

```
data = in.request ();
...  work out answer
in.reply (answer, true);
followUp = in.request ();
...  more ping/pong
in.reply (answer);
```



Network Connections are *Any-1*



Connections (*two-way channels*)

- Connections allow *extended* two-way client–server communication (from *any* number of clients).
- Without them, two-way network communications would be tedious to set up. The server would have to construct two named (***input***) channels: one for the *opening* messages and the other for *follow-ups*; the clients would have to create individual named (***input***) channels for replies. The server would have to find all its client reply channels (***outputs***).
- With them, only one name is needed. The server constructs a (***server***) connection and each client constructs a (***client***) connection - with same name.

Connections (*extended rendezvous*)

- Connections allow *extended* two-way client–server communication (from *any* number of clients).
- A connection is not ***open*** until the first **`reply`** has been received (to the first **`request`**).
- Once a connection is ***opened***, only the client that opened it can interact with the server until the connection is ***closed***.
- Following an **`request`**, a client must commit to a **`reply`** (i.e. no intervening synchronisations) .
- A client may have several servers ***open*** at the same time - but only if they are opened in a sequence honoured by all clients … else ***deadlock*** will occur!

Connections (*extended rendezvous*)

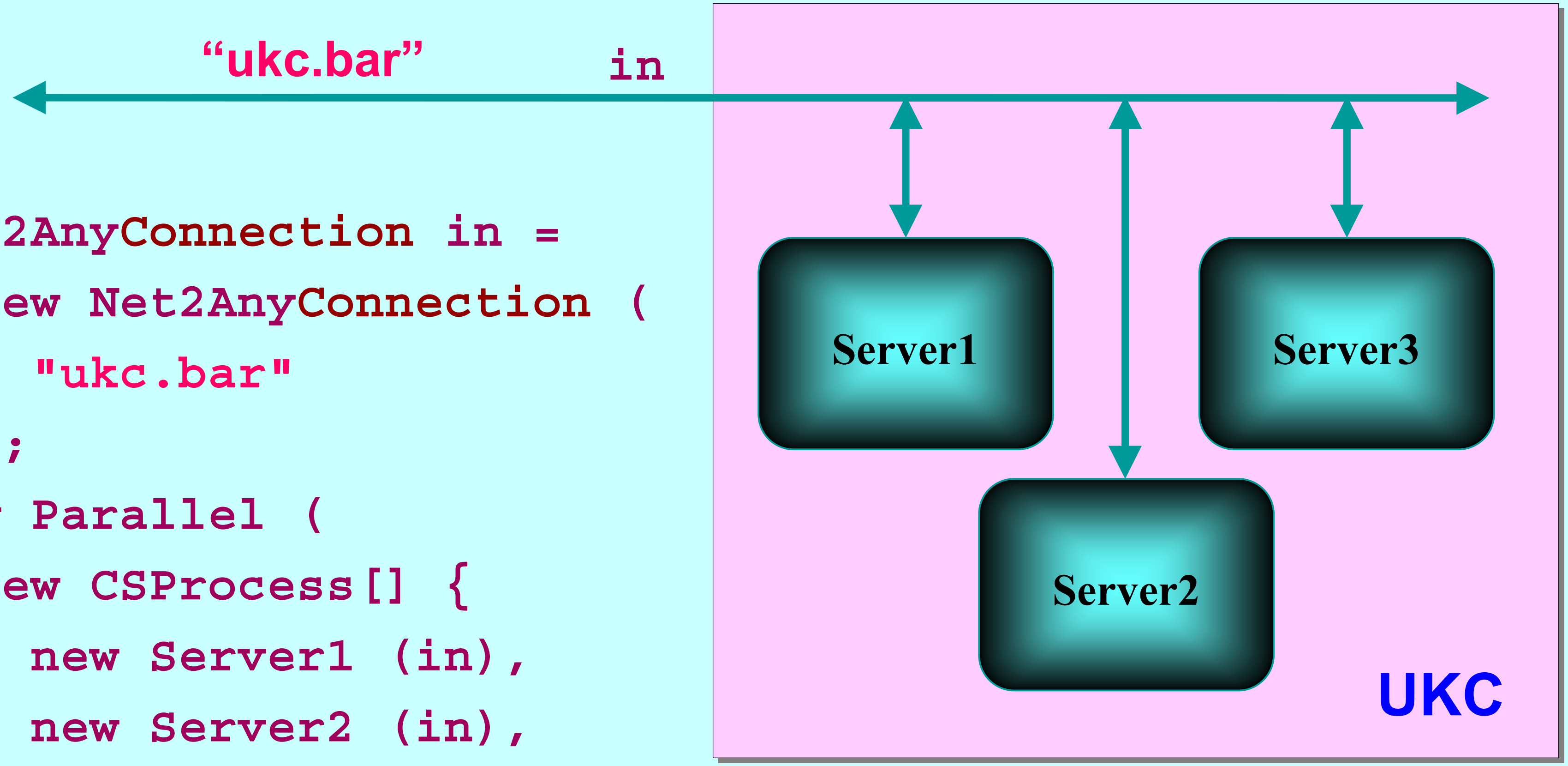
- Connections allow *extended* two-way client–server communication (from *any* number of clients).
- The connection ***protocol***:

`request (reply+ request)* reply`

is self-synchronising across the network - no extra acknowledgements are needed.

- For completeness, **JCSP** provides connection channels for local networks (**`One2OneConnection`**, **`Any2OneConnection`**, etc.).

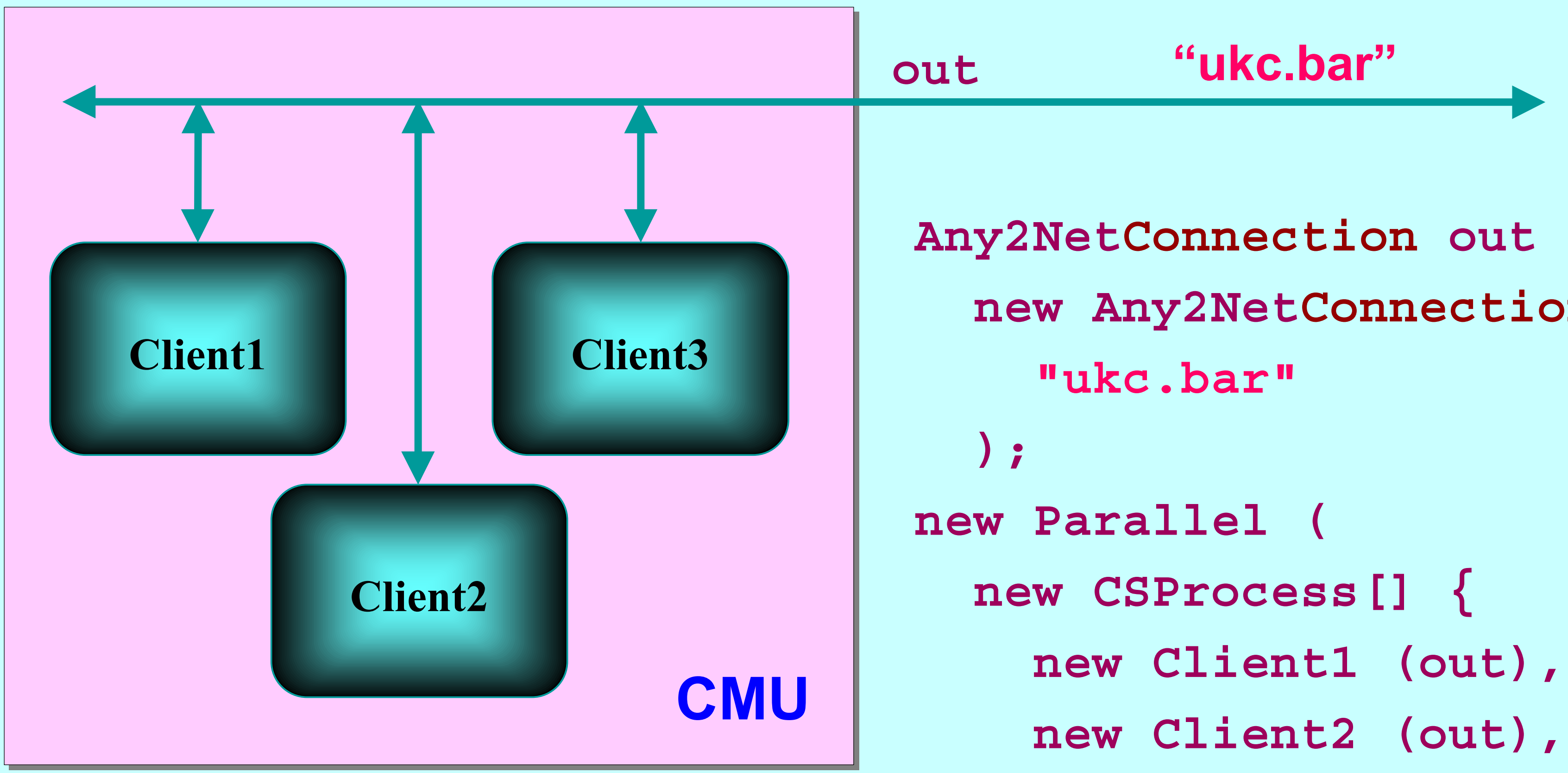
Net-Any Connections




```
Net2AnyConnection in =
  new Net2AnyConnection (
    "ukc.bar"
  );
new Parallel (
  new CSProcess[] {
    new Server1 (in),
    new Server2 (in),
    new Server2 (in)
  }
).run ();
```

i.e. within a node, there can be *any* number of servers

Any-Net Connections

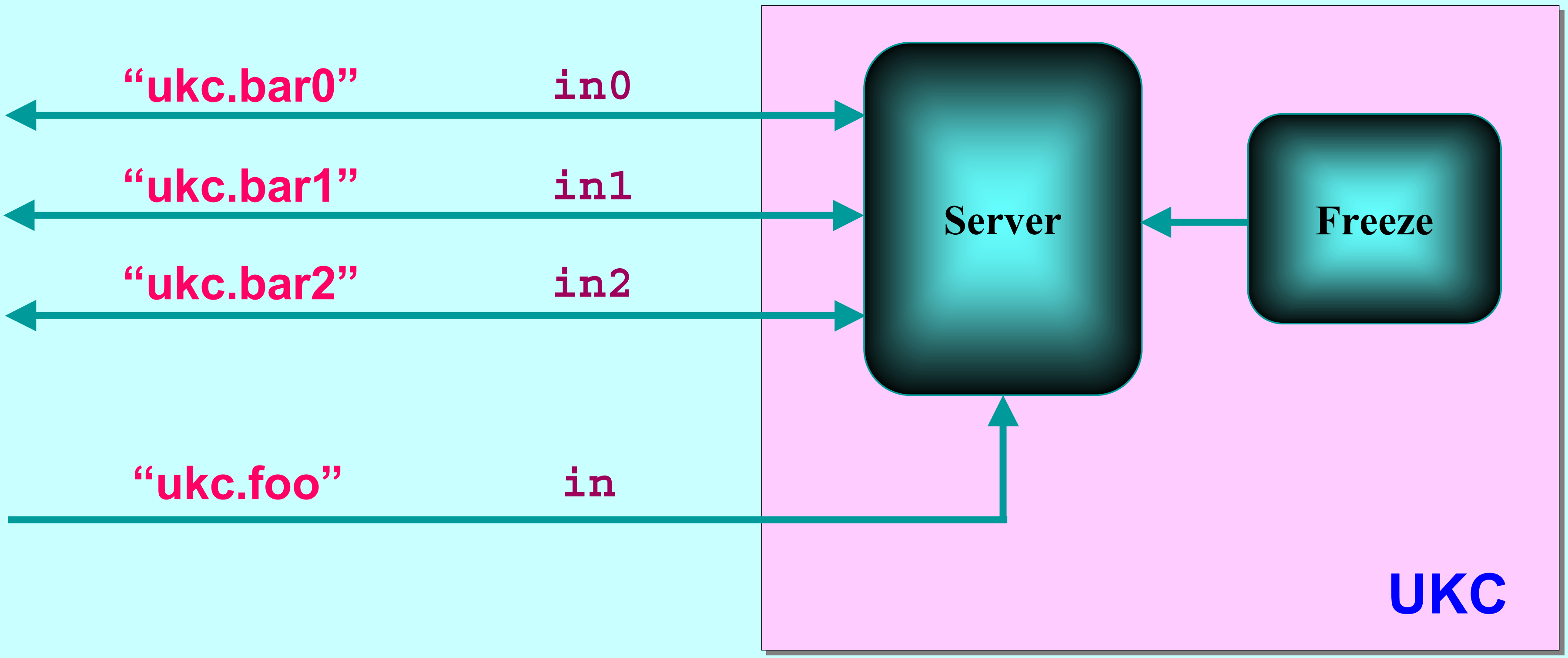



```
Any2NetConnection out =
  new Any2NetConnection (
    "ukc.bar"
  );
new Parallel (
  new CSProcess[] {
    new Client1 (out),
    new Client2 (out),
    new Client3 (out)
  }
).run ();
```

i.e. within a node, there can be *any* number of clients

Copyright P.H.Welch

Net-One Connections are *ALTable*



- The `Server` process can ***ALT*** over its 3 *networked* server connections, its *networked* input channel and its *local* input channel.

Anonymous Channels

- Network channels *may be* connected by the **JCSP** *Channel Name Server* (CNS) …
- *… but they don't have to be!*

- A network channel can be created *(always by the inputter)* without registering a name with the CNS:

```
Net2OneChannel in = new Net2OneChannel ();  // no name!
```

- Remote processes cannot, of course, find it for themselves … *but you can tell your friends …*

Anonymous Channels

- Location information (*<IP-address, port-number, virtual-channel-number>*) is held within the constructed network channel. This is the data registered with the CNS - if we had given it a name.

- Extract that information:

```
Net2OneChannel in = new Net2OneChannel ();  // no name!
NetChannelLocation inLocation = in.getLocation ();
```

- The information can be distributed using existing (network) channels to those you trust:

```
toMyFriend.write (inLocation);
// remember your friend may distribute it further ...
```

Anonymous Channels

- Your friend inputs the location information *(of your unregistered channel)* via an existing channel:

```
NetChannelLocation outLocation =
  (NetChannelLocation) fromMyFriend.read ();
```

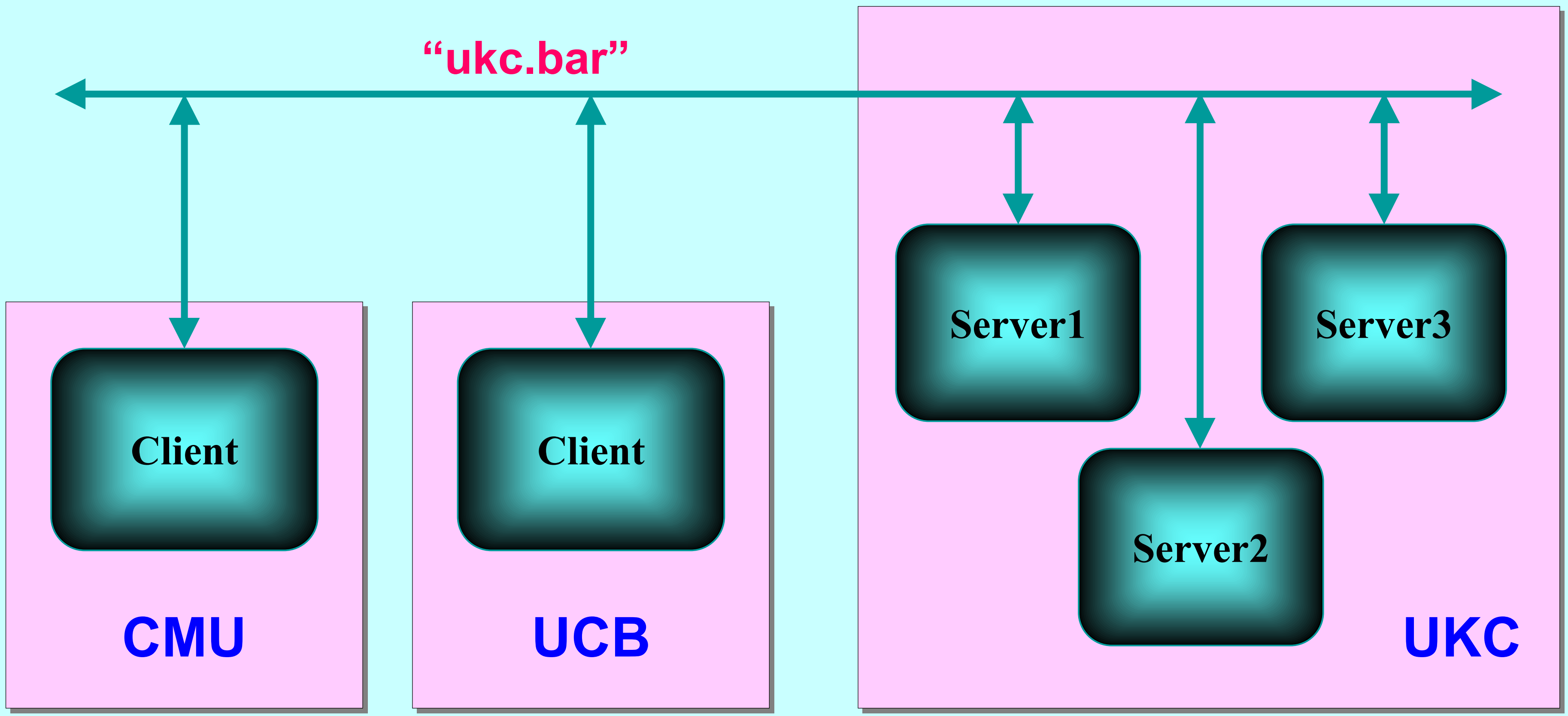
- And can then construct her end of the channel:

```
One2NetChannel out = new One2NetChannel (outLocation);
```

- The `One2NetChannel` constructor has been given the information it would have got from the CNS (had it been given a registered name to resolve).
- You and your friends can now communicate over the unregistered channel.

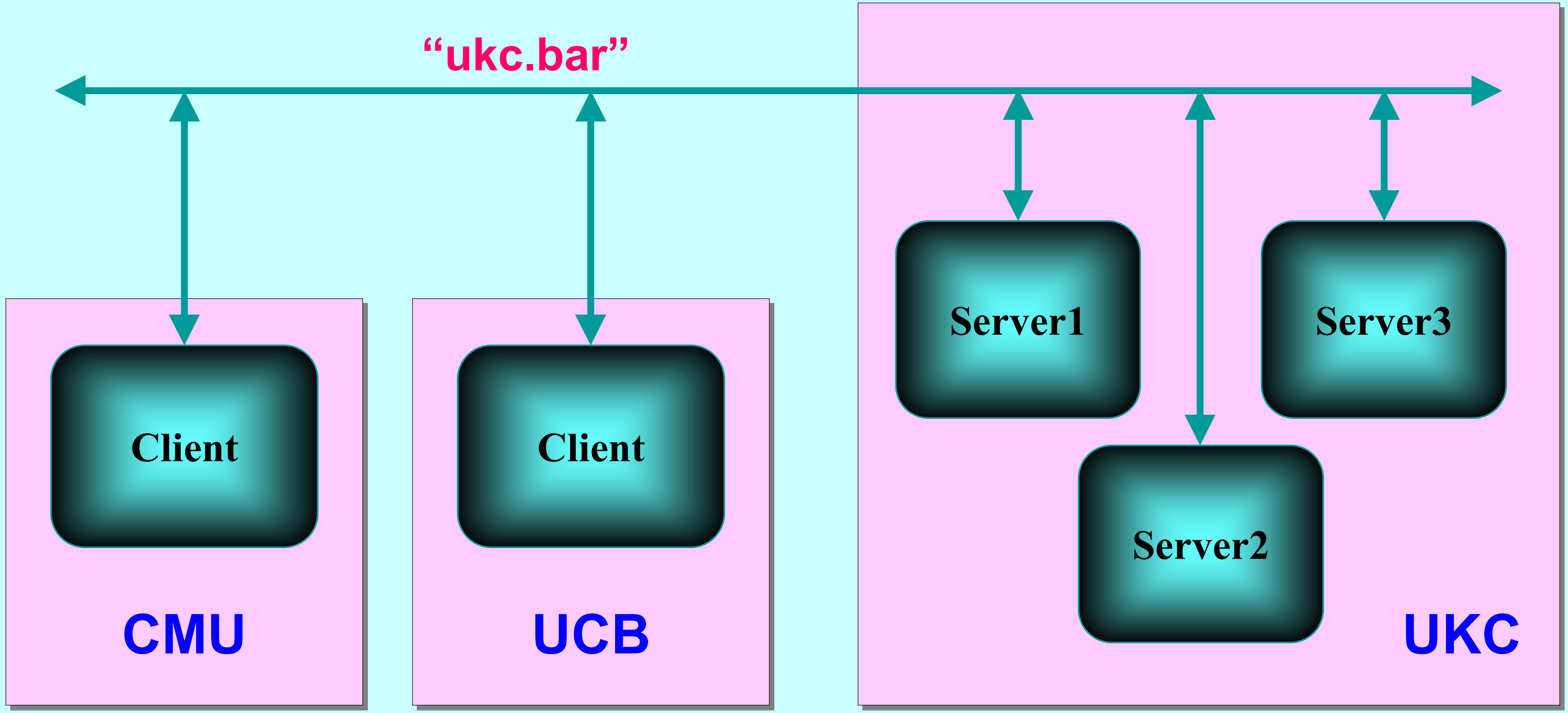
Anonymous Connections

- These work in exactly the same way as anonymous channels … and are possibly more useful …



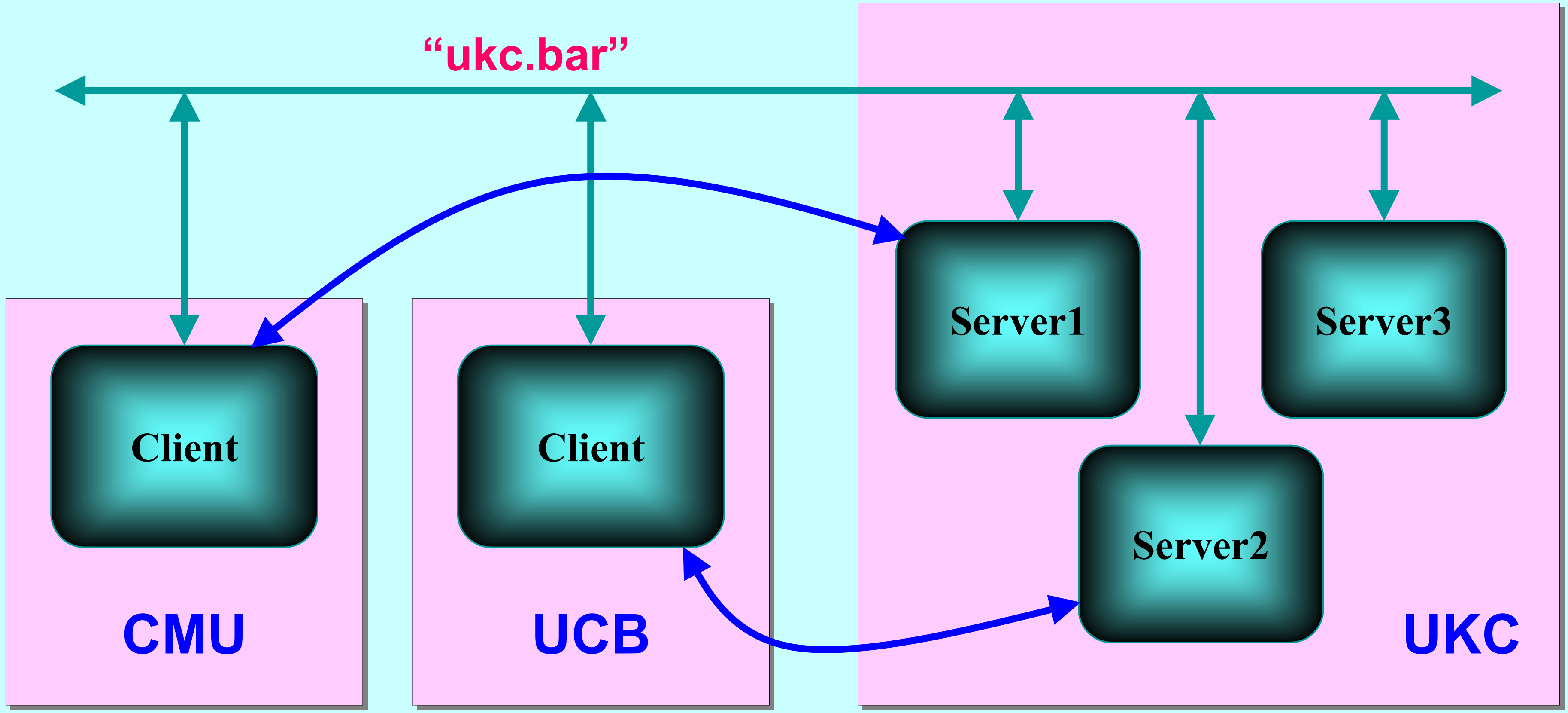
Anonymous Connections

- Right now, *only one client and one server* can be doing business at a time over the shared connection.



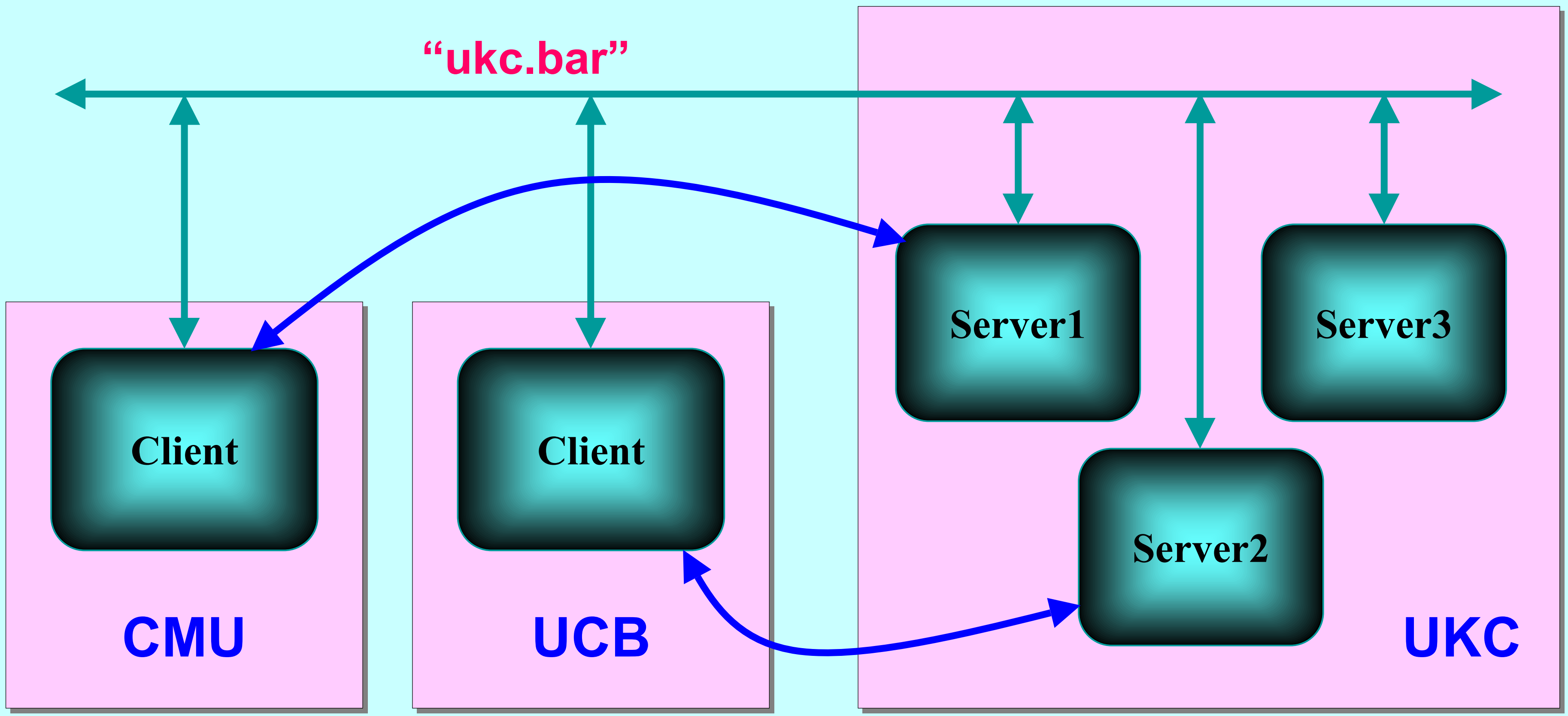
Anonymous Connections

- But that business could be: "gimme a connection" (*client*) & "OK - here's a private one" (*server*) …



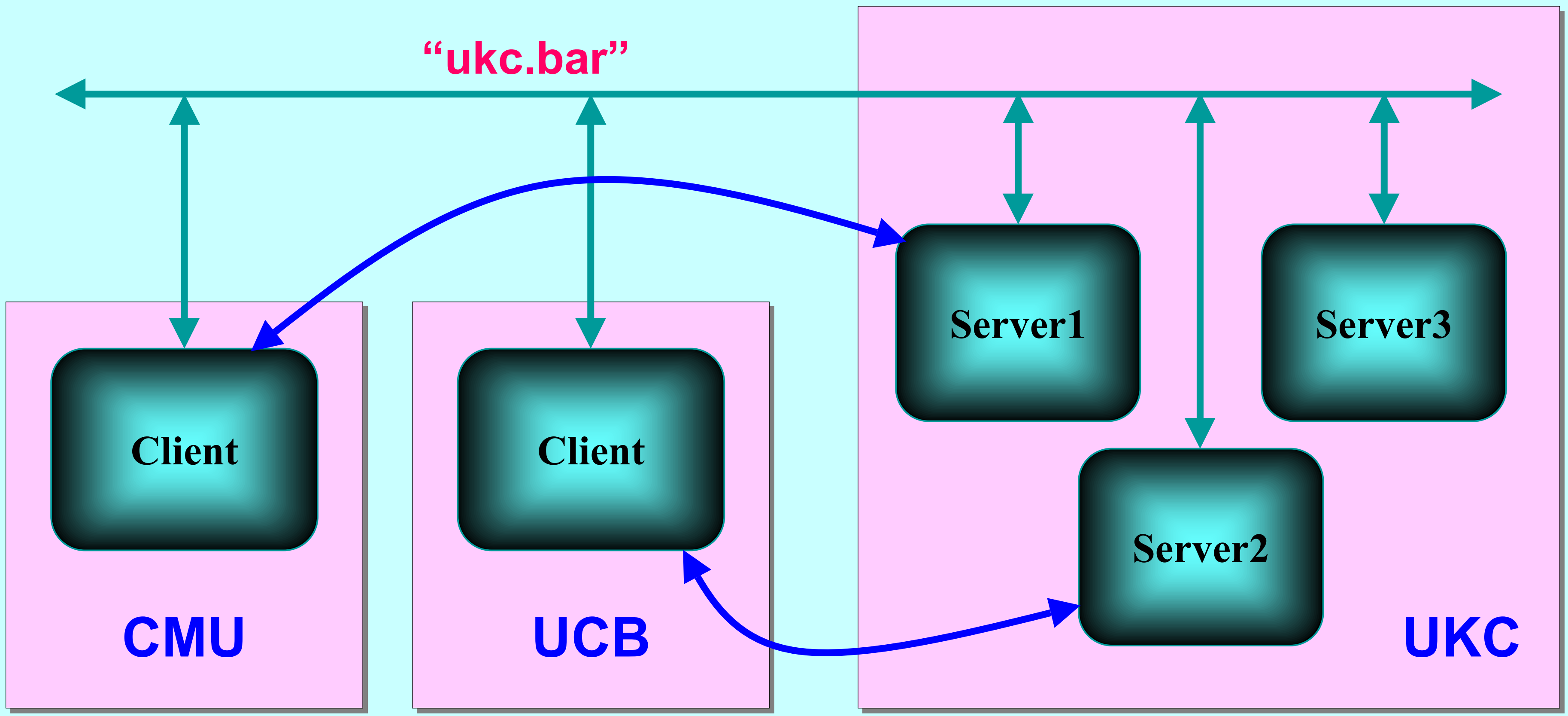
Anonymous Connections

- So, the registered connection is only used to let a *client* and *server* find each other …



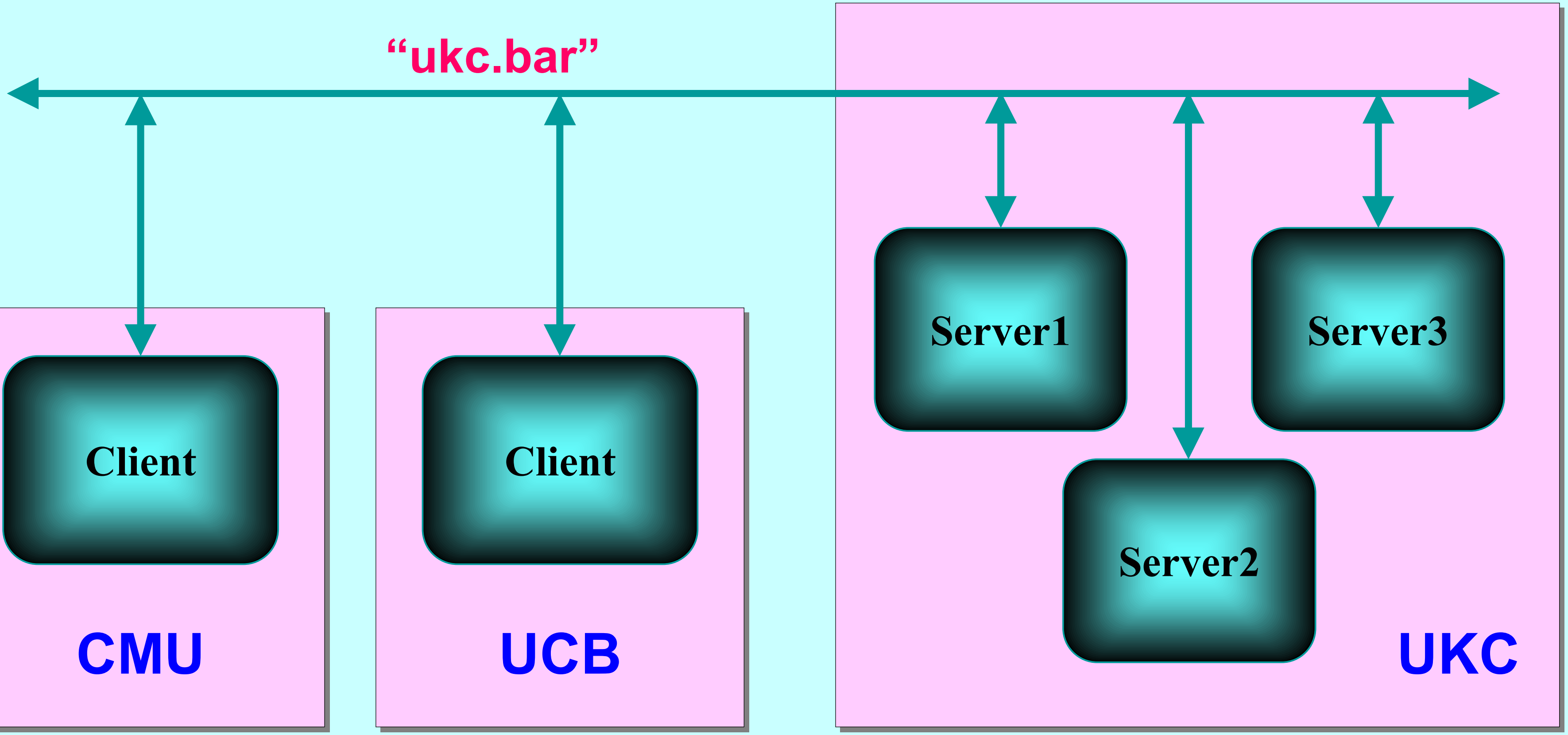
Anonymous Connections

- The real *client-server* work is now conducted over dedicated (unregistered) connections - *in parallel*.

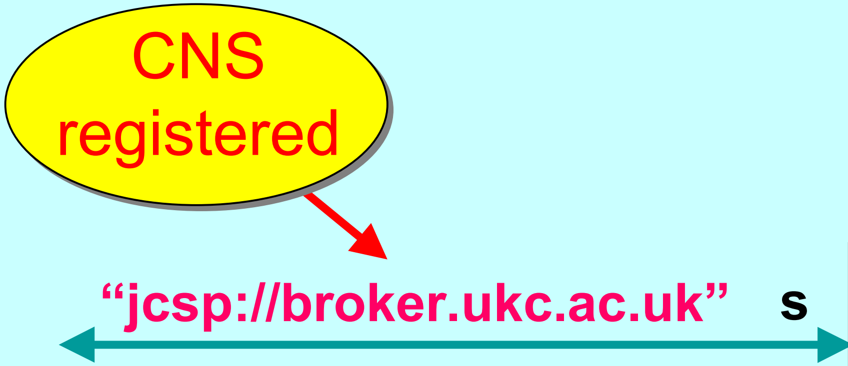


Anonymous Connections

- After the *client-server* transaction has finished the server deletes the special connections.



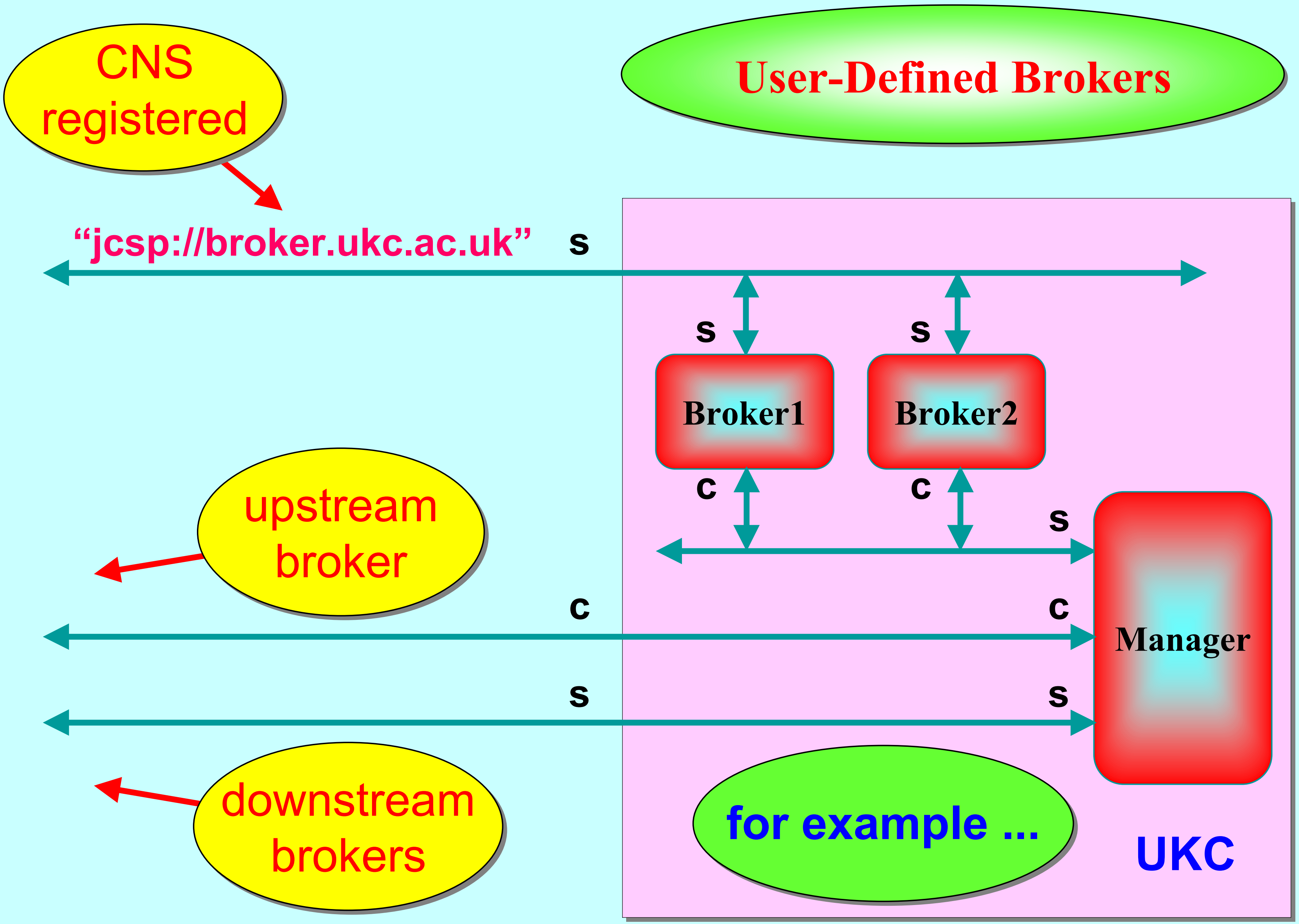
User-Defined Brokers



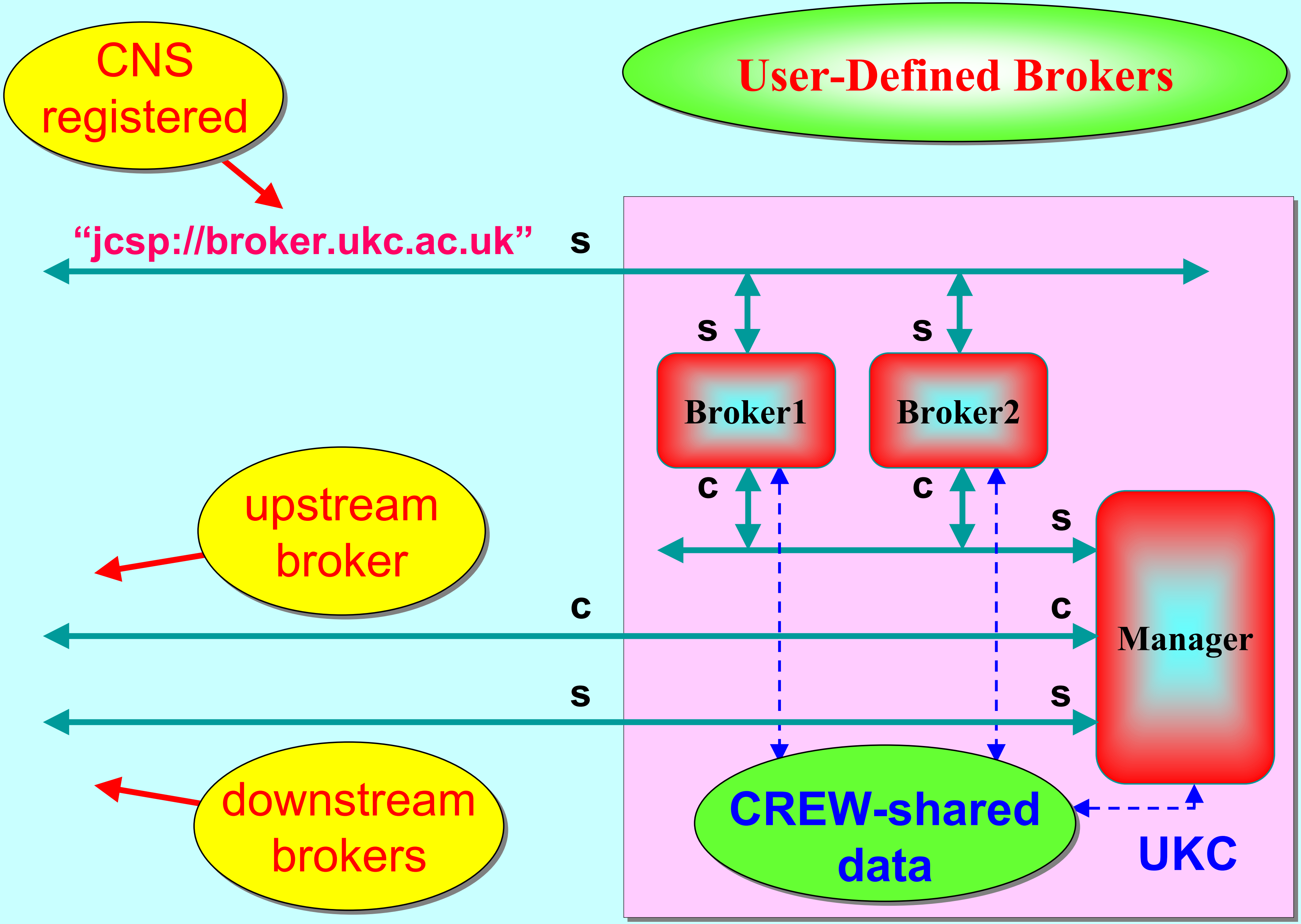
roll-your-own broker

If you want a matching service more sophisticated than the given CNS, simply build what you want as the server for your ***CNS registered*** connection. Anyone finding that can use your new broker.

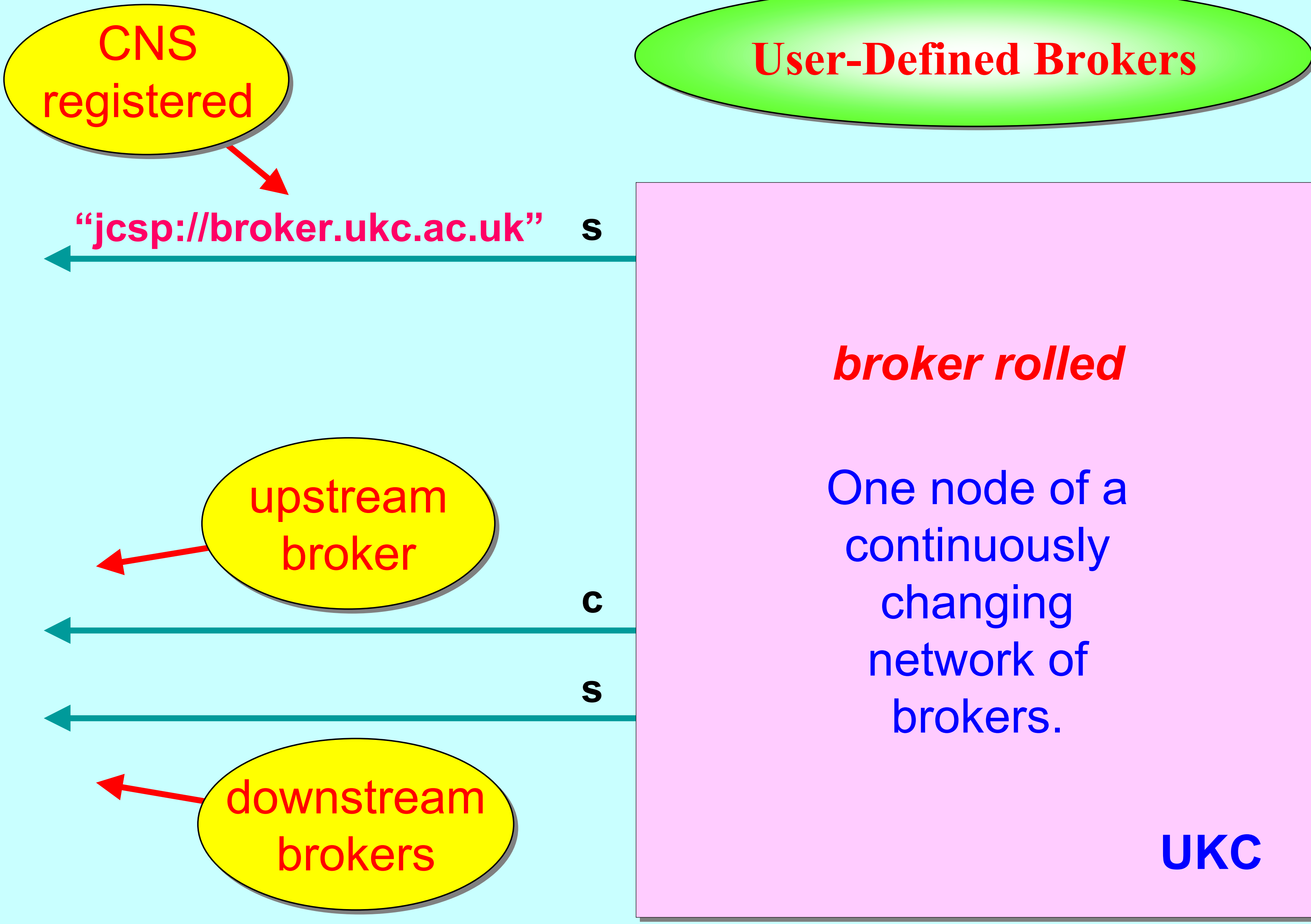
Copyright P.H.Welch



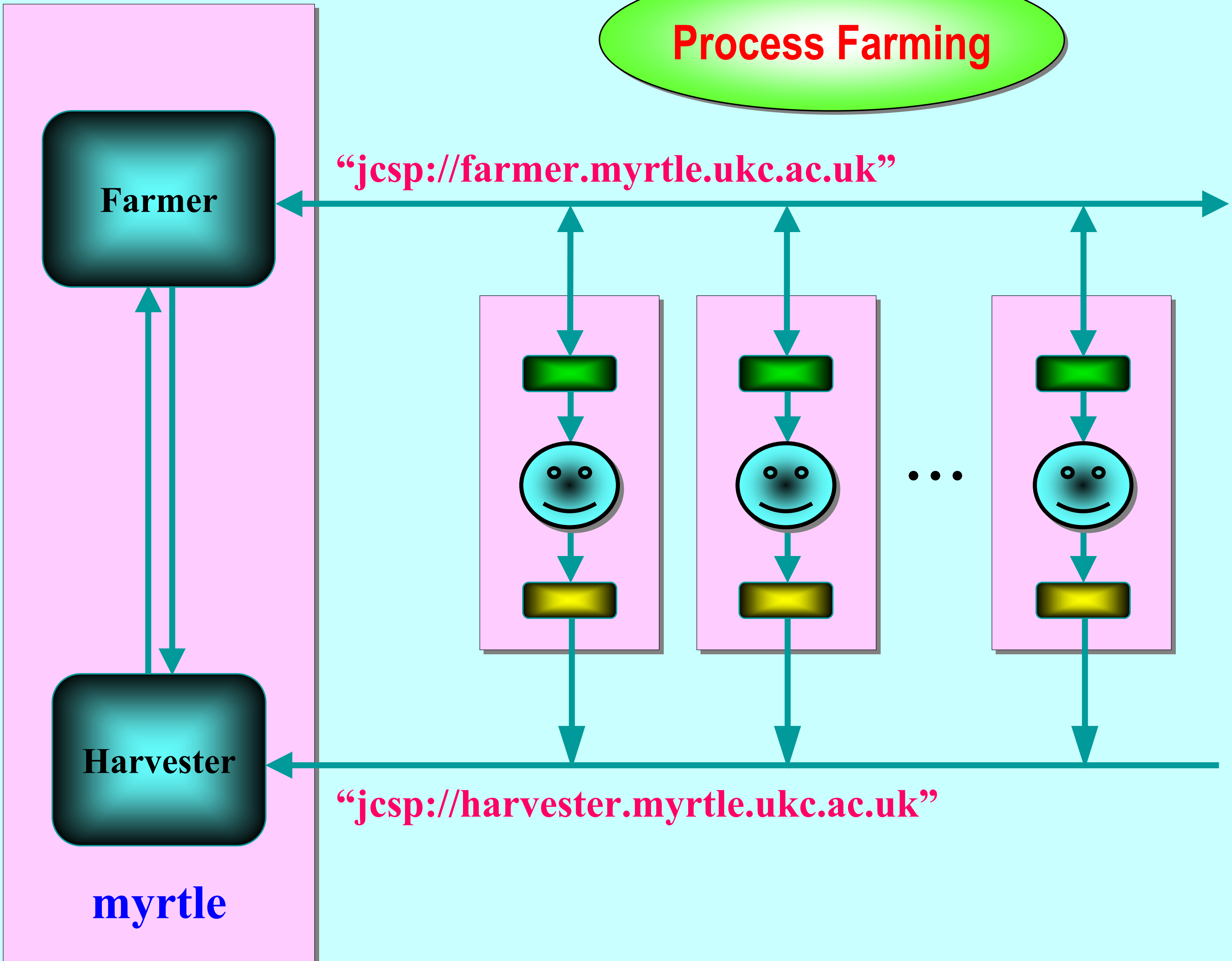
CNS registered
User-Defined Brokers
“jcsp://broker.ukc.ac.uk”
s
s
s
Broker1
Broker2
c
c
s
upstream broker
c
c
Manager
s
s
downstream brokers
for example ...
UKC


CNS
registered
User-Defined Brokers
"jcsp://broker.ukc.ac.uk"
s
s
s
Broker1
Broker2
c
c
s
upstream
broker
c
c
Manager
s
s
downstream
brokers
CREW-shared
data
UKC

User-Defined Brokers

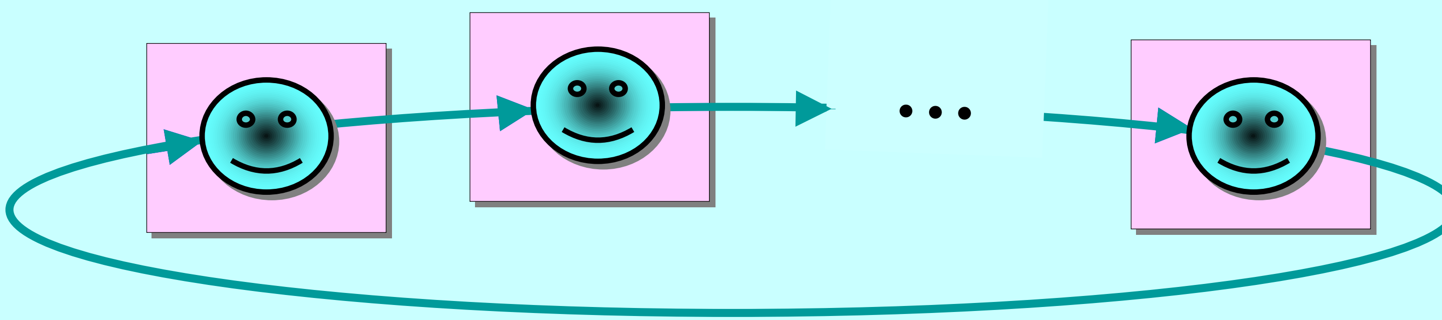


Process Farming



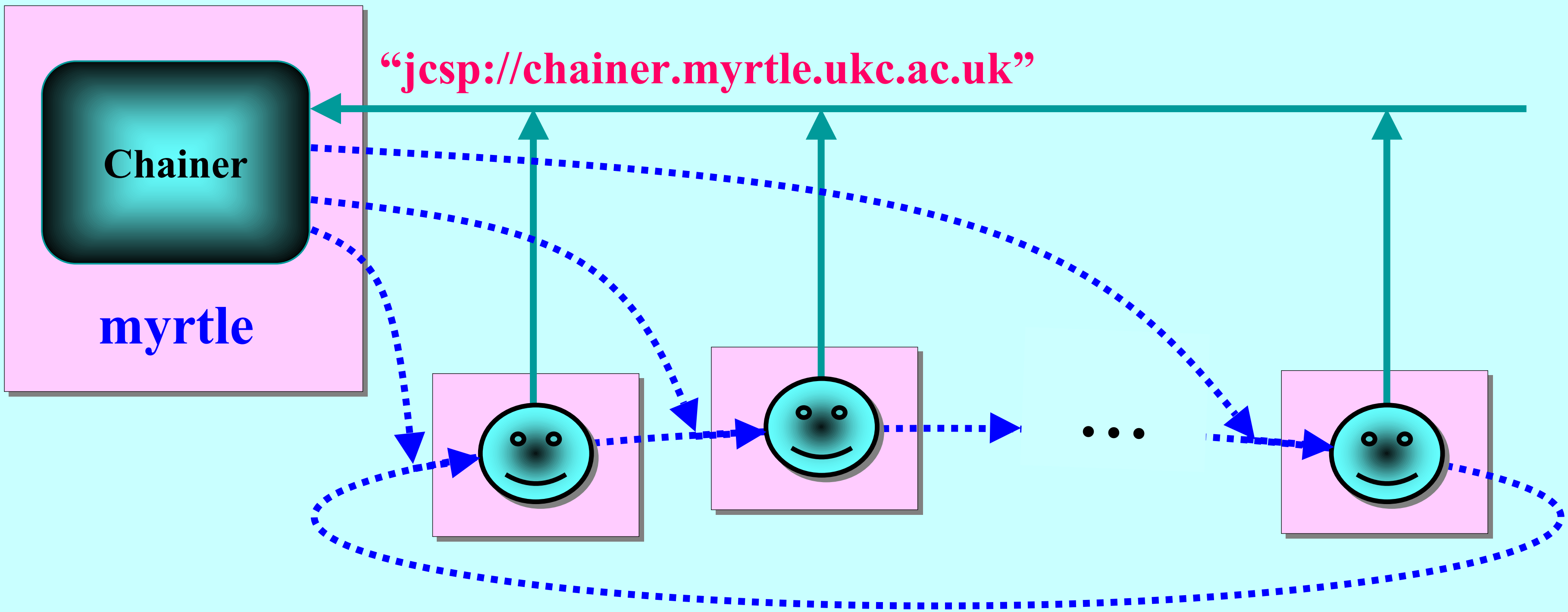
Process Chaining

```
final int MY_ID = ... ;

final int N_NODES = ... ;

final int NEXT_ID = ((MY_ID + 1) % N_NODES;

Net2OneChannel in = new Net2OneChannel ("node-" + MY_ID);

Net2OneChannel out = new Net2OneChannel ("node-" + NEXT_ID);

new WorkProcess (MY_ID, N_NODES, in, out).run ();
```



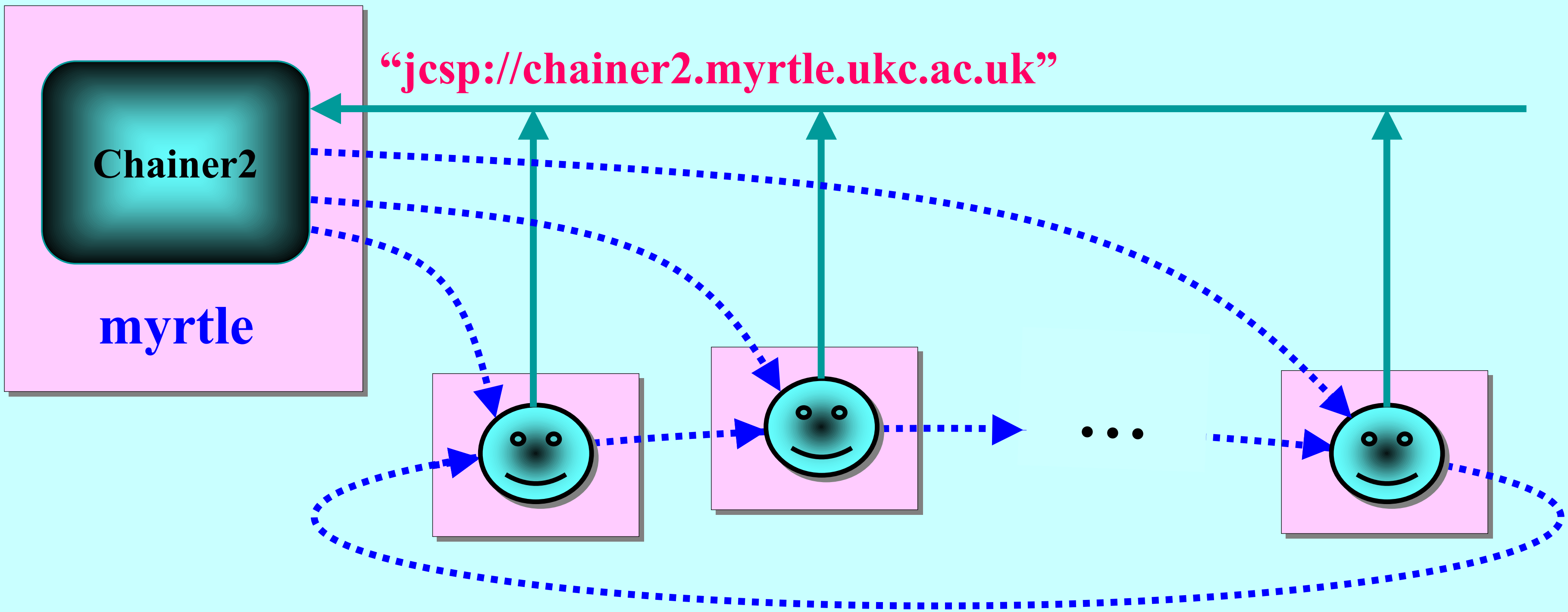
OK – so long as each worker knows the length of the chain and its place in it.

Process Chaining



A volunteer worker won't know this! But it can make its own network input channel *anonymously* and send its location to someone who does …

Process Chaining



It's slightly easier if each node makes *two* network input channels – so that its *control* line is different from its *data* line from the chain …

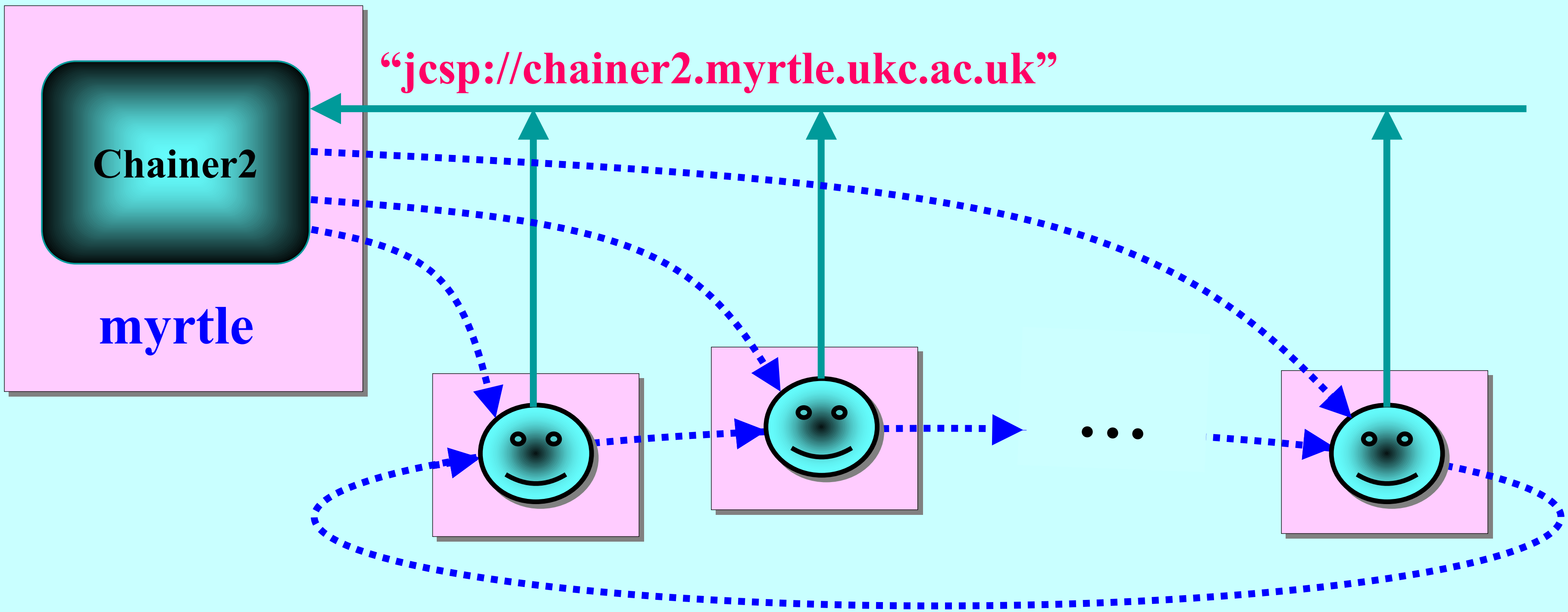
Ring worker code

```
One2NetChannel toChainer =
  = new One2NetChannel ("jcsp://chainer.myrtle.ukc.ac.uk");

Net2OneChannel in = new Net2OneChannel ();
NetChannelLocation inLocation = in.getLocation ();
toChainer.write (inLocation);

NetChannelLocation outLocation = (NetChannelLocation) in.read ();
One2NetChannel out = new One2NetChannel (outLocation);

int[] info = (int[]) in.read ();               // wait for ring sync
final int MY_ID = info[0];                     // (optional)
final int N_NODES = info[1];                   // (optional)

info[0]++;
if (info[0] < info[1]) out.write (info);       // pass on ring sync

new WorkProcess (MY_ID, N_NODES, in, out).run ();
```

Chainer (ringer) code

```
final int N_NODES = ... ;

Net2OneChannel fromWorkers =
  = new Net2OneChannel ("jcsp://chainer.myrtle.ukc.ac.uk");

NetChannelLocation lastL =
  (NetChannelLocation) fromWorkers (read);
One2NetChannel lastC = new One2NetChannel (lastL);

for (int nWorkers = 1; nWorkers < N_NODES; nWorkers++) {
  NetChannelLocation nextL =
    (NetChannelLocation) fromWorkers (read);
  One2NetChannel nextC = new One2NetChannel (nextL);
  nextC.write (lastL);
  lastL = nextL;
}
lastC.write (lastL);                          // completes the network ring
lastC.write (new int[] {0, N_NODES});  // final ring synchronisation
```

Process Chaining



It's slightly easier if each node makes *two* network input channels – so that its *control* line is different from its *data* line from the chain …

Example Applications

Process Farming

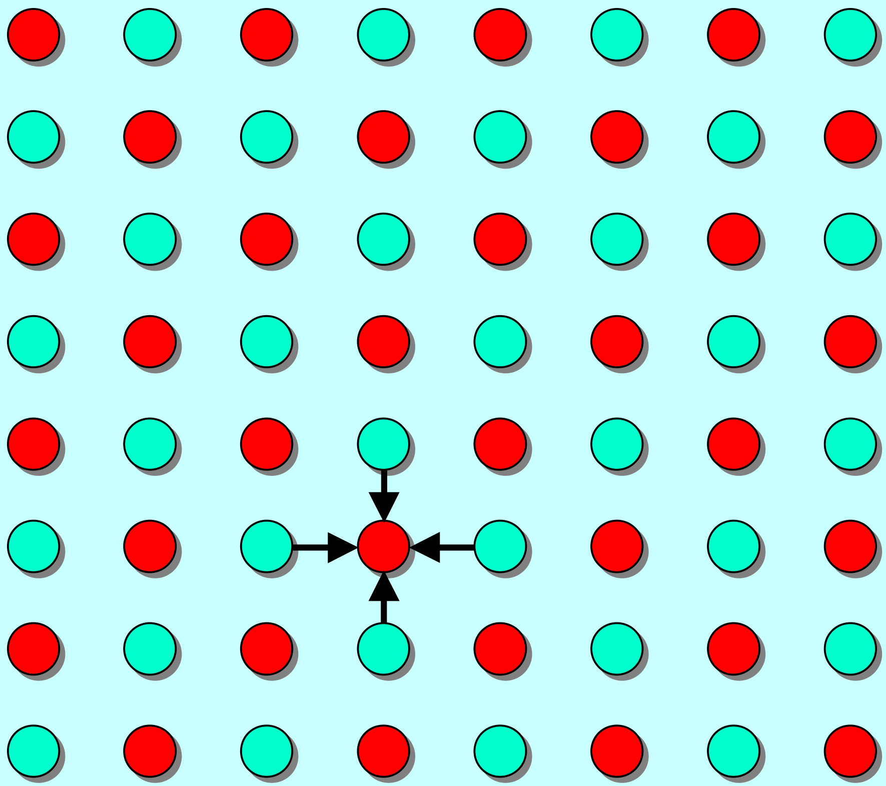
All '*embarassingly parallel*' ones, ray tracing, Mandelbrot, travelling salesman (needs dynamic control though), …

Process Chaining

All space-division system modelling, n-body simulations, SORs, cellular automata, … (some need *bi-directional* chains/rings)

Copyright P.H.Welch

SOR – red/black checker pointing

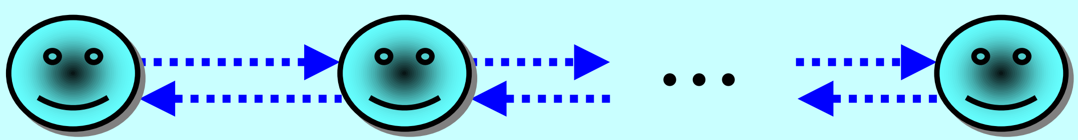


where = black

and = red

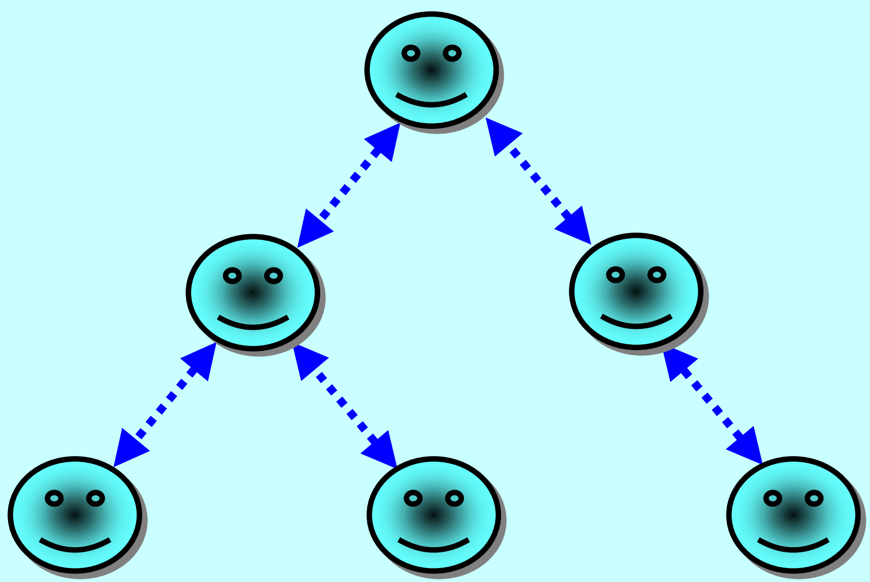
SOR – red/black checker pointing

This needs a *two-way* chain to exchange information on boundary regions being looked after by each worker …



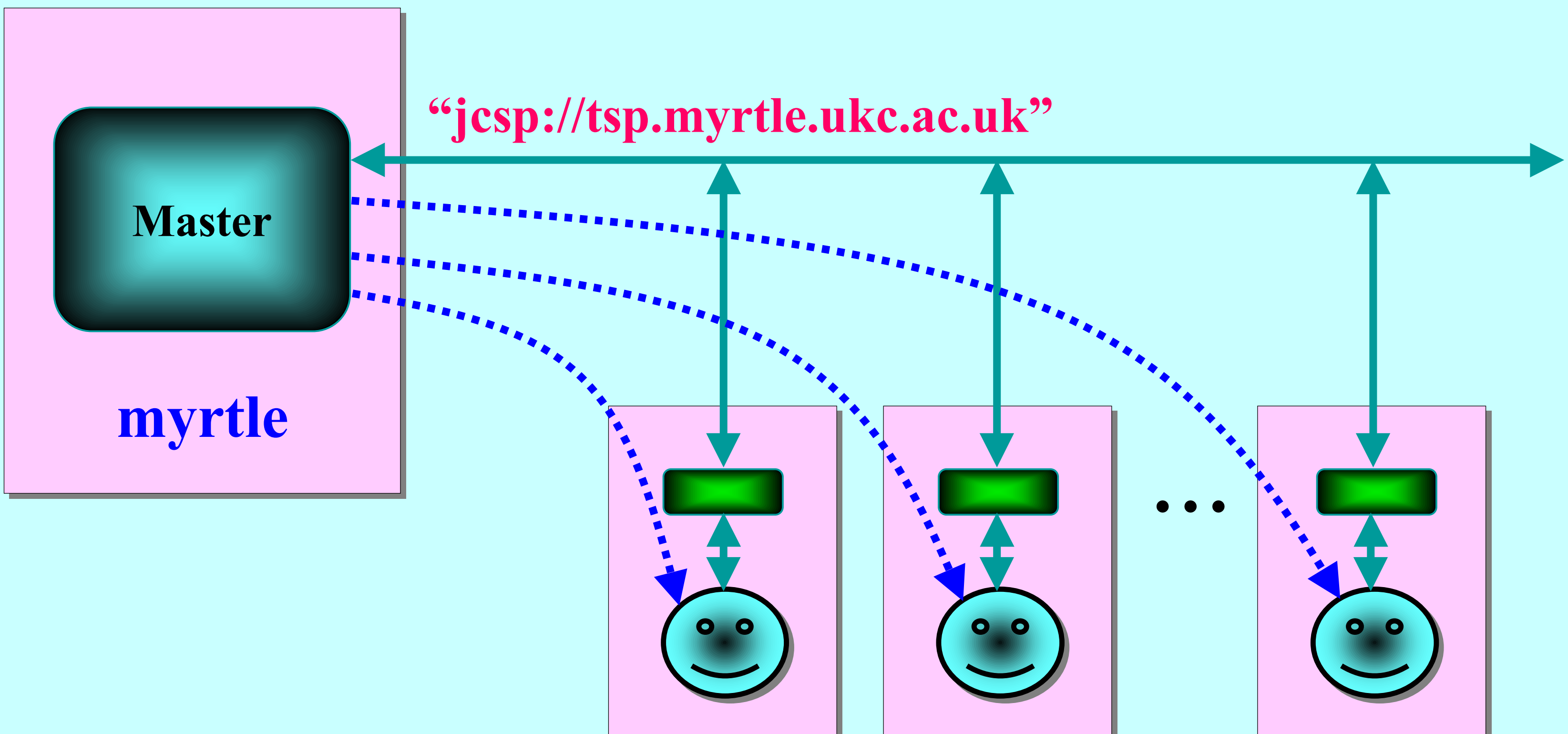
SOR – red/black checker pointing

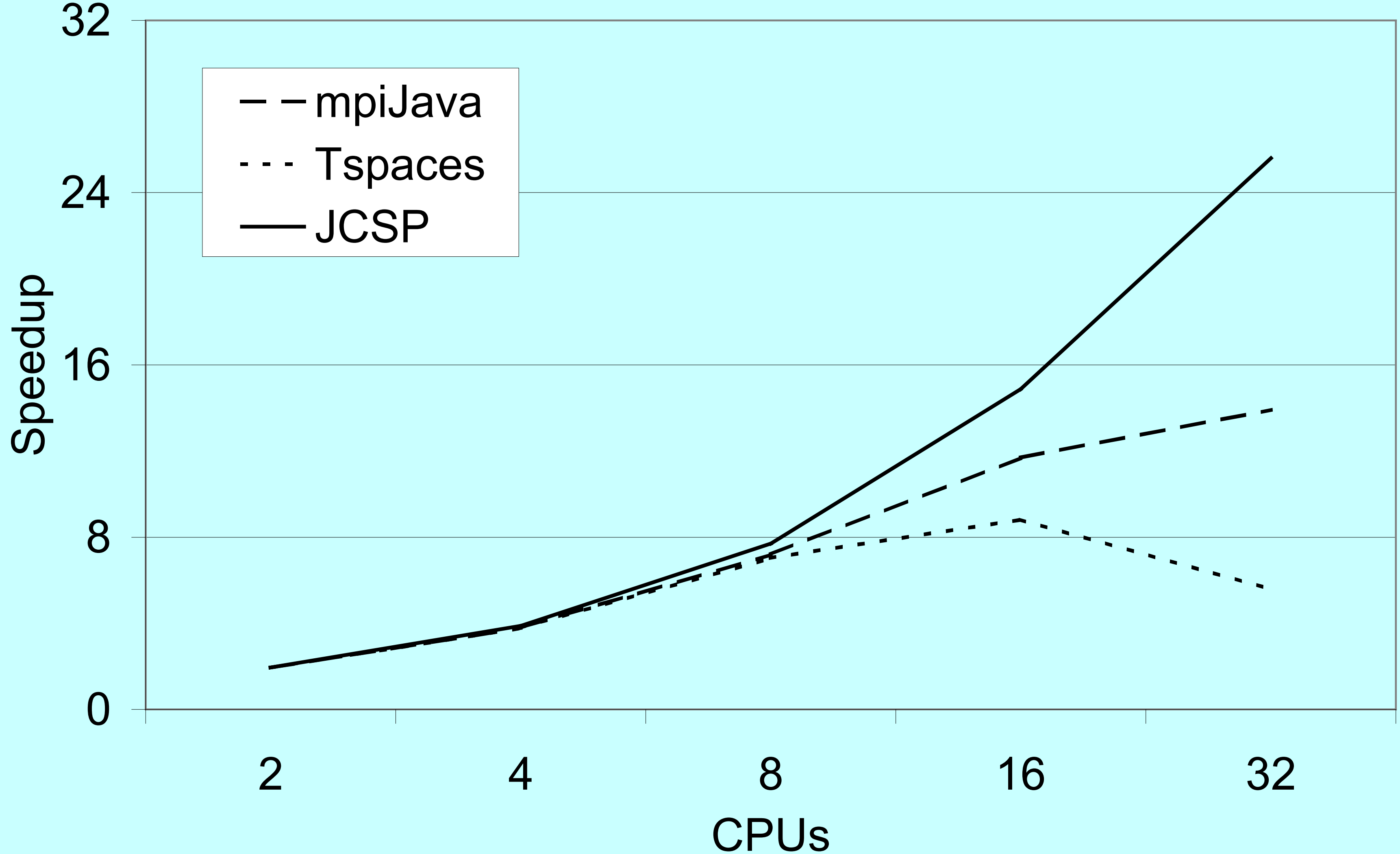
Also, a global *sum-of-changes* (found by each node) has to be computed each cycle to resolve halting criteria. This is speeded-up by connecting the nodes into a tree (so that *adds* and *communications* can take place in parallel).



Travelling Salesman Problem

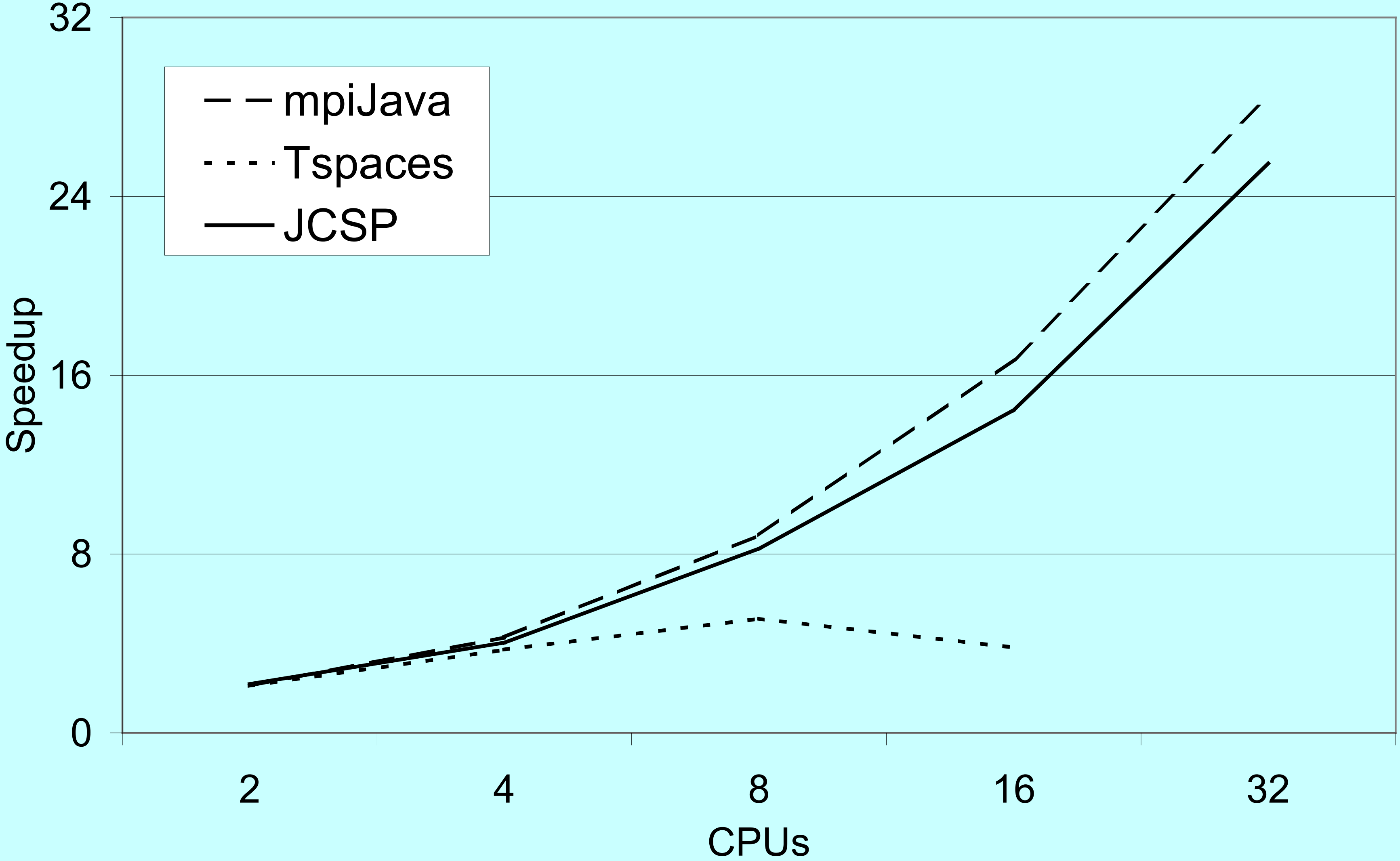
Basically a process farm … but when better lower bounds arrive, they must be communicated to all concerned workers.



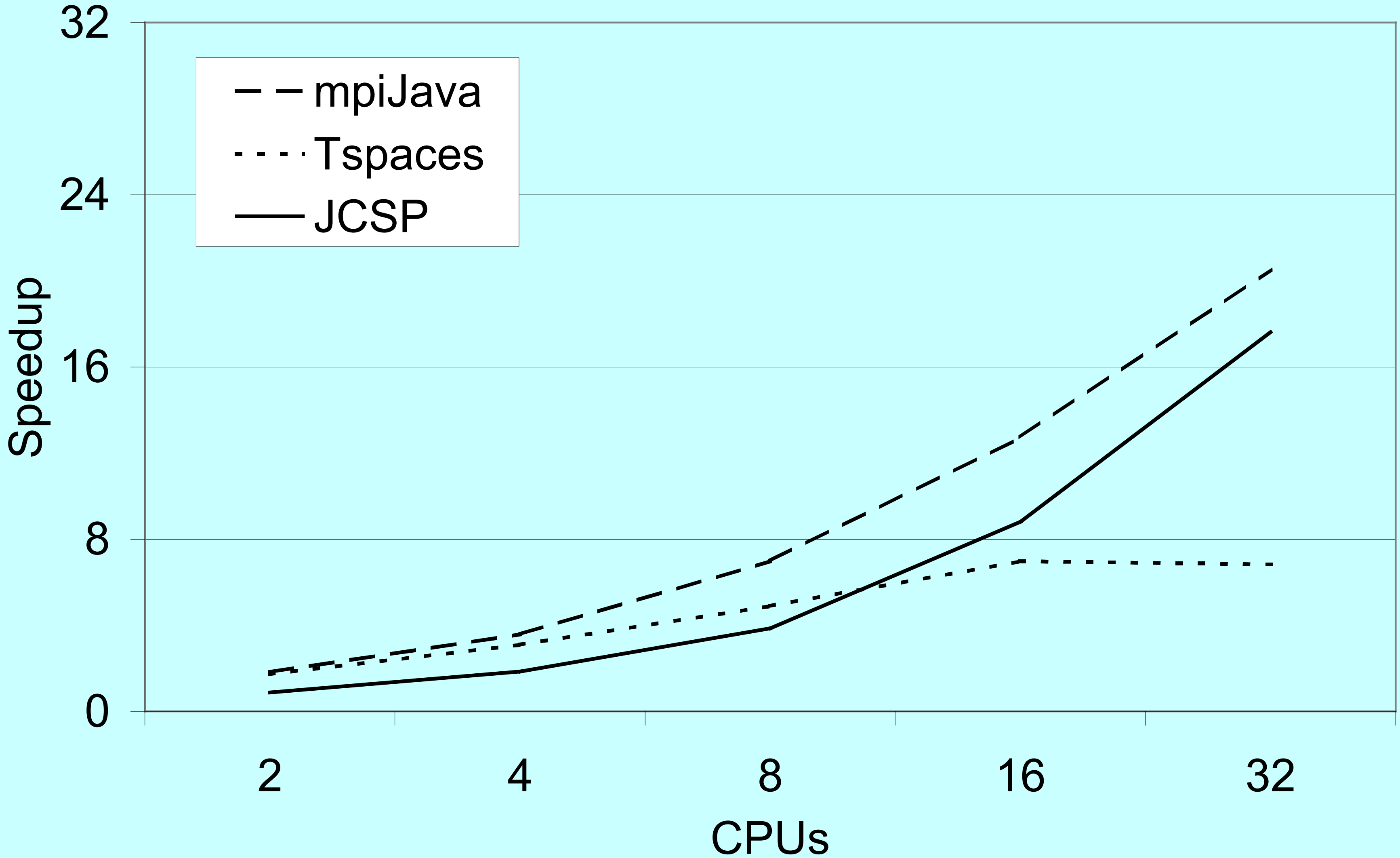


The n-Body benchmark (n = 10000)

Copyright P.H.Welch

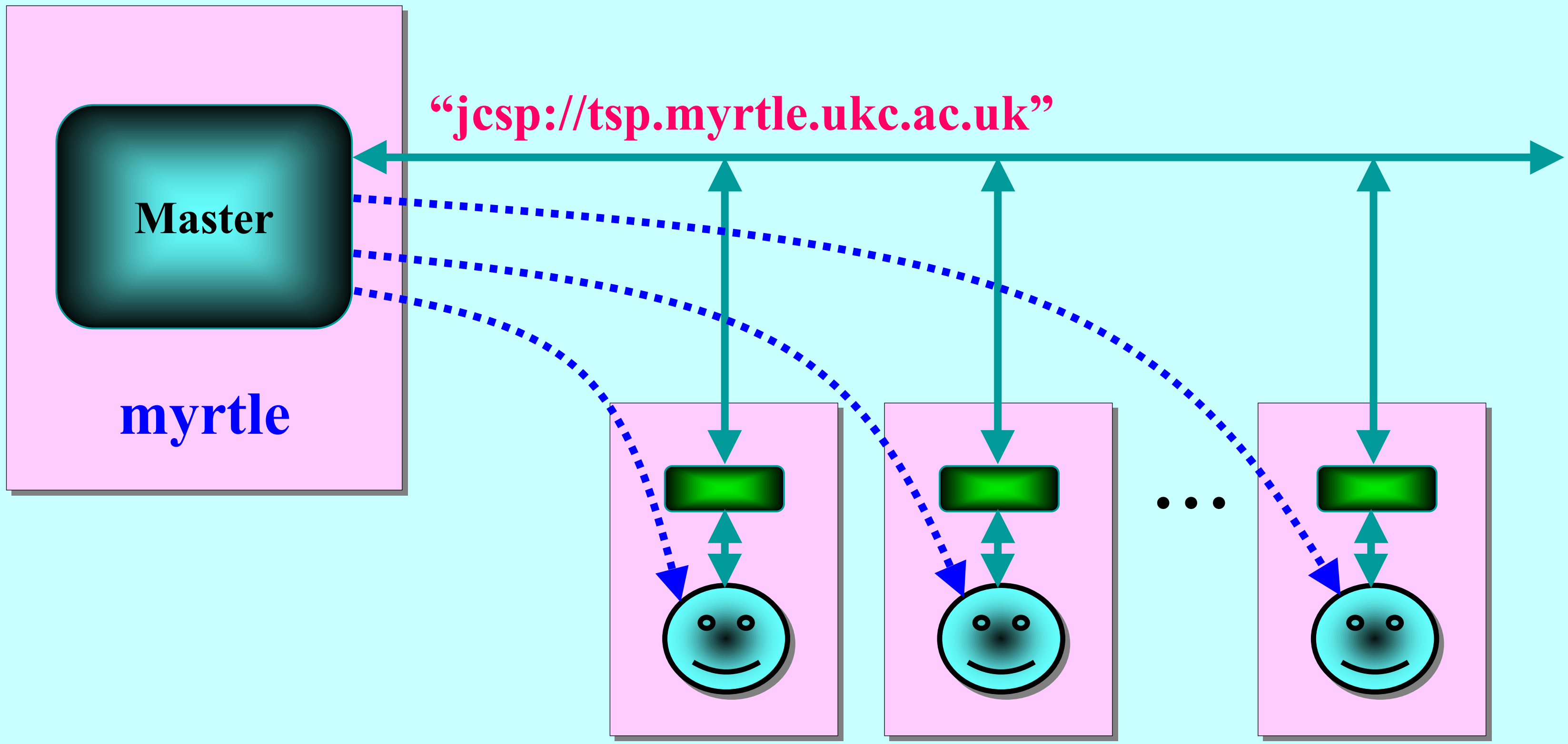


The SOR benchmark (7000 x 7000 matrix)



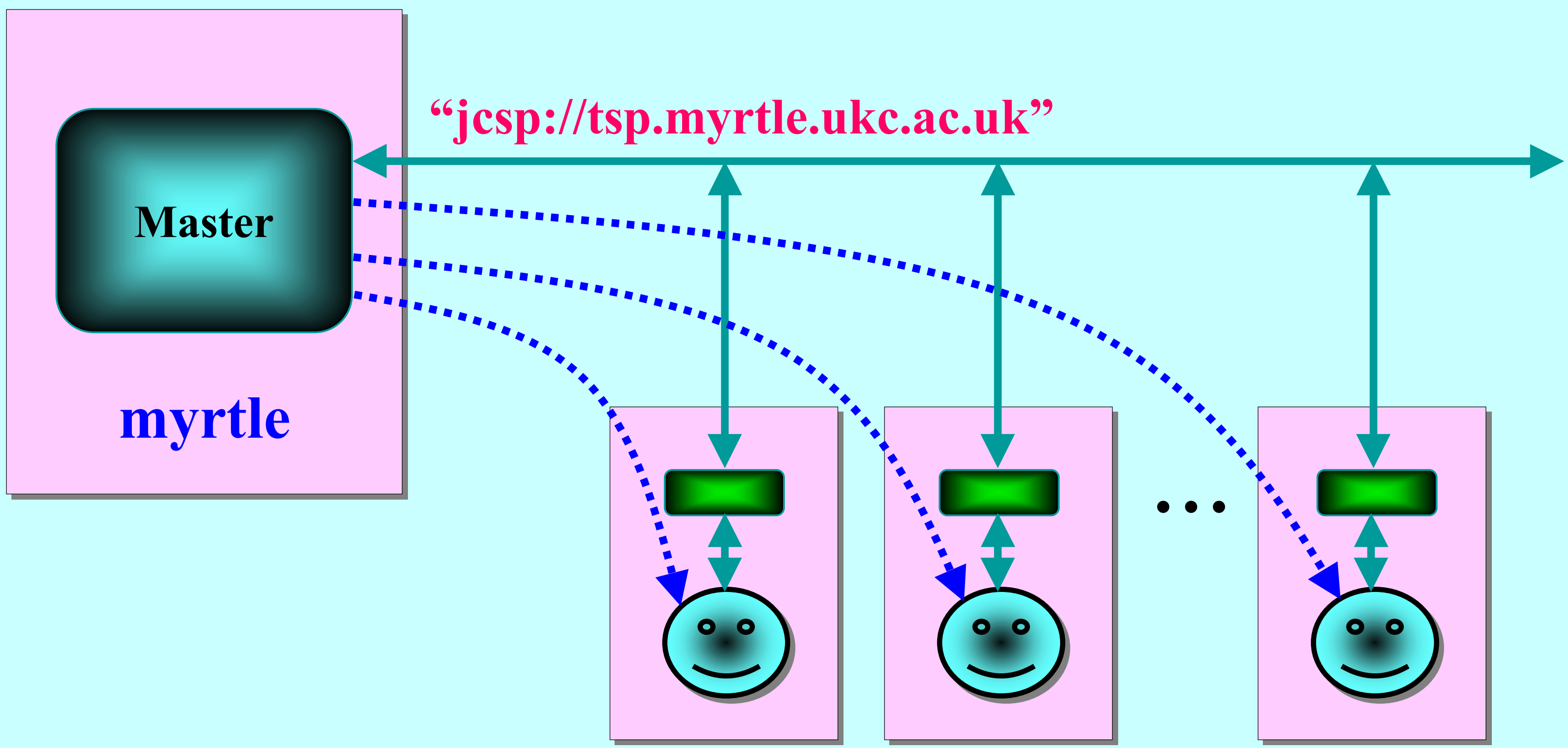
The Travelling Salesman Problem (15 cities)

Travelling Salesman Problem



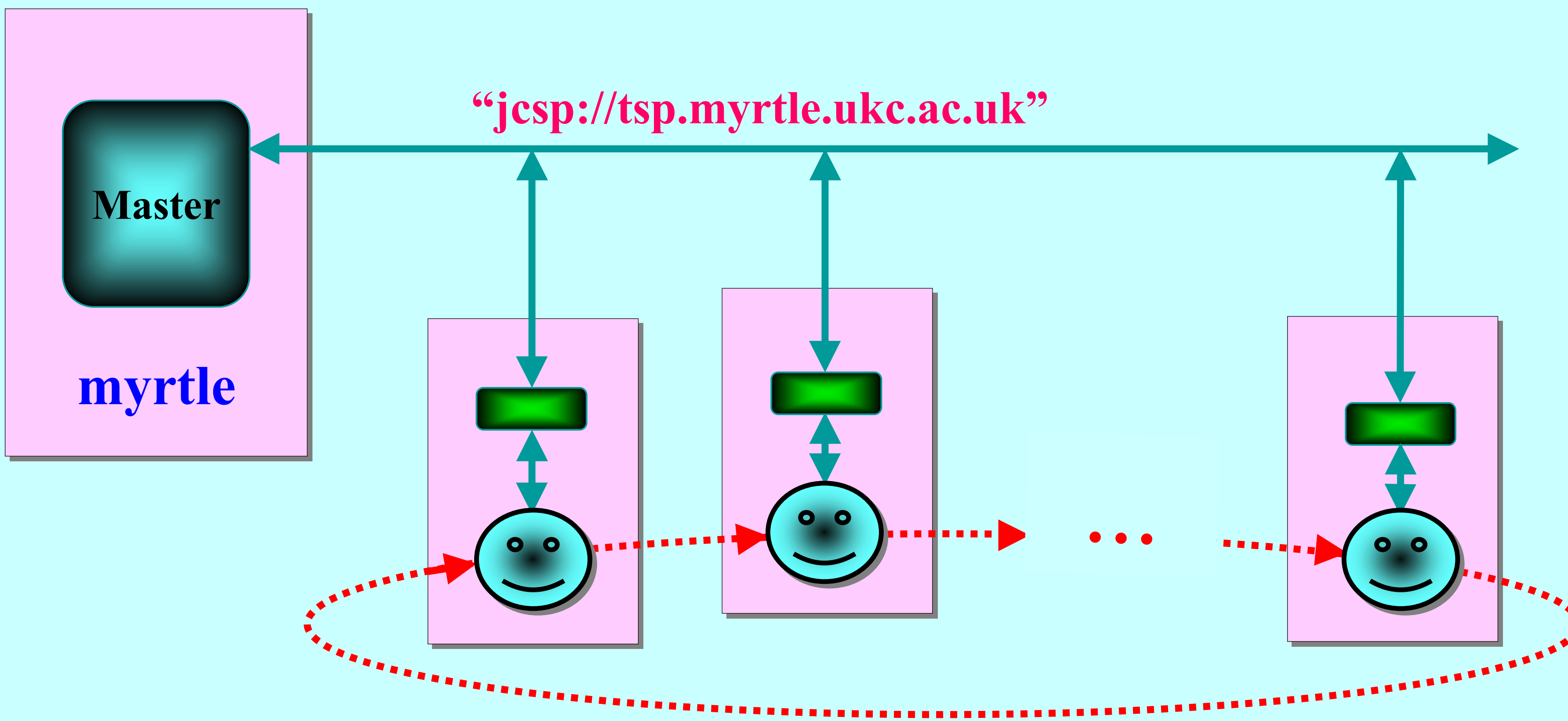
Currently, workers report newly discovered shorter paths back to the master (who maintains the global shortest). If master receives a better one, it broadcasts back to workers.

Travelling Salesman Problem



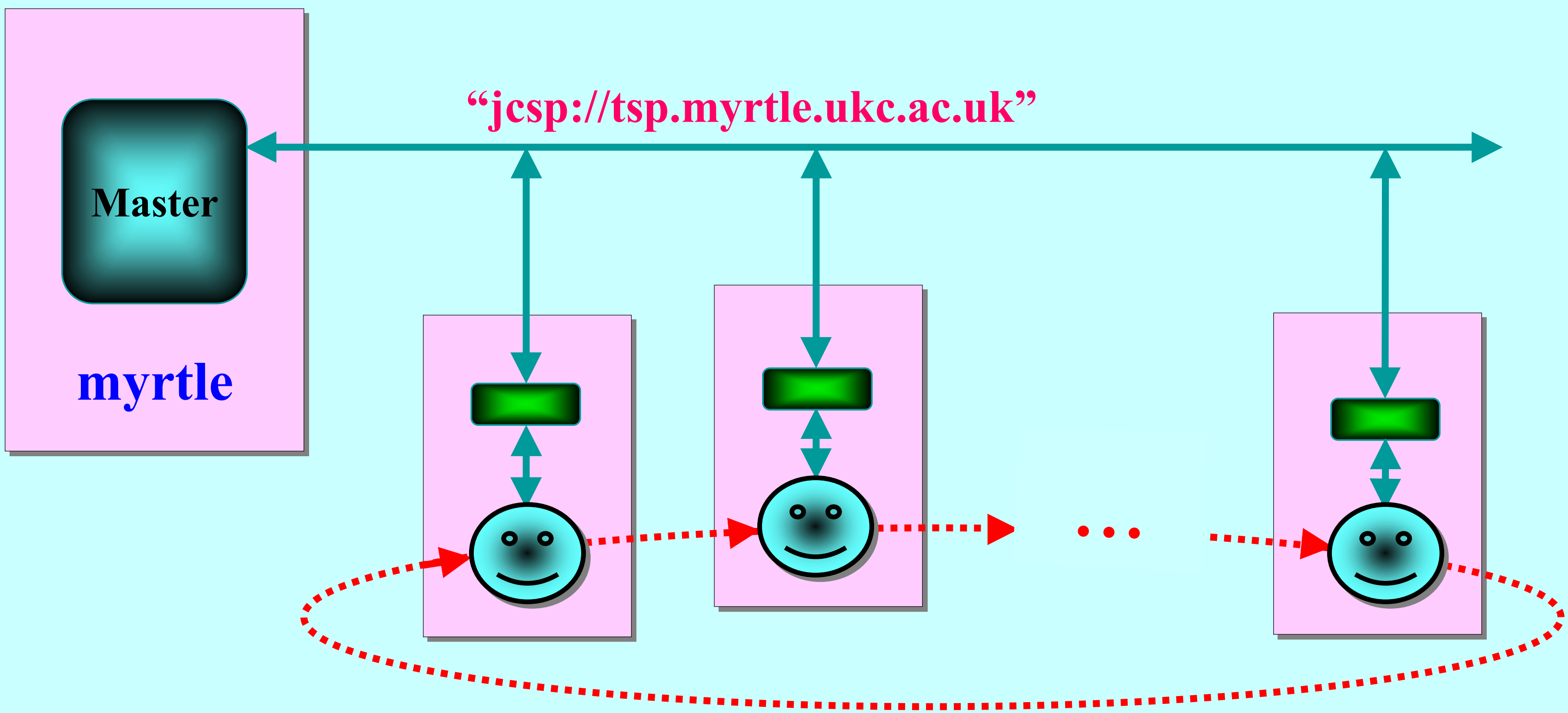
Massive swamping of network links in the early stages.
Also, generation of garbage *currently* provokes garbage collector – and clobbers cache on dual-processor nodes.

Travelling Salesman Problem



Eliminate the broadcasting – control against swamping – stop generating garbage … ☺ ☺ ☺

Travelling Salesman Problem

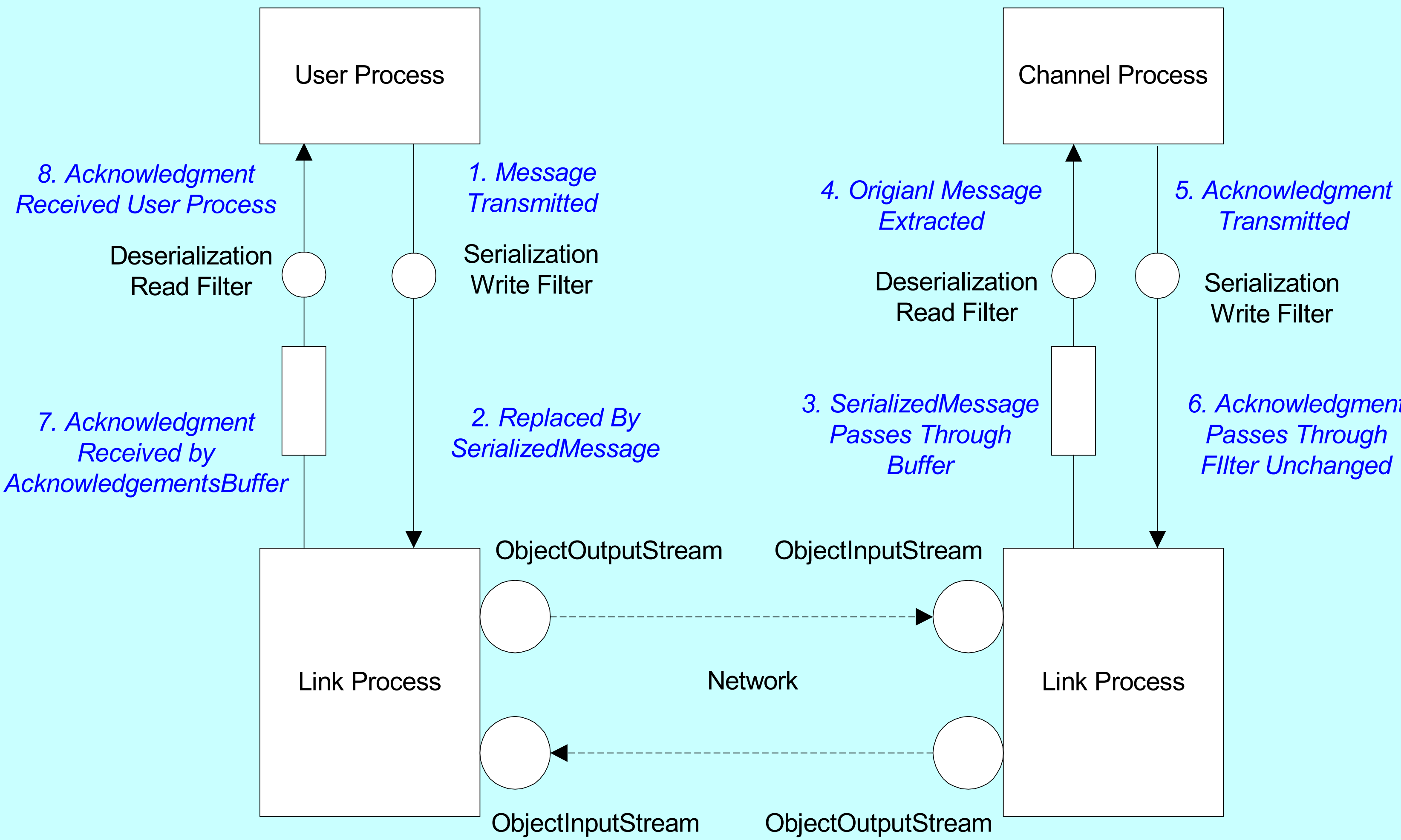


Global minimum maintained in ring (made with one-place overwriting channel buffers) … ***easy!!!***

Networked Class Loading

- By default, objects sent across a networked channel (or connection) use Java *serialization*.
- This means the receiving JVM is expected to be able to load (or already have loaded) the class files needed for its received objects.
- However, JCSP networked channels/connections can be set to communicate those class files *automatically* (if the receiver can't find them locally).
- Machine nodes cache those class files locally in case they themselves need to forward them.

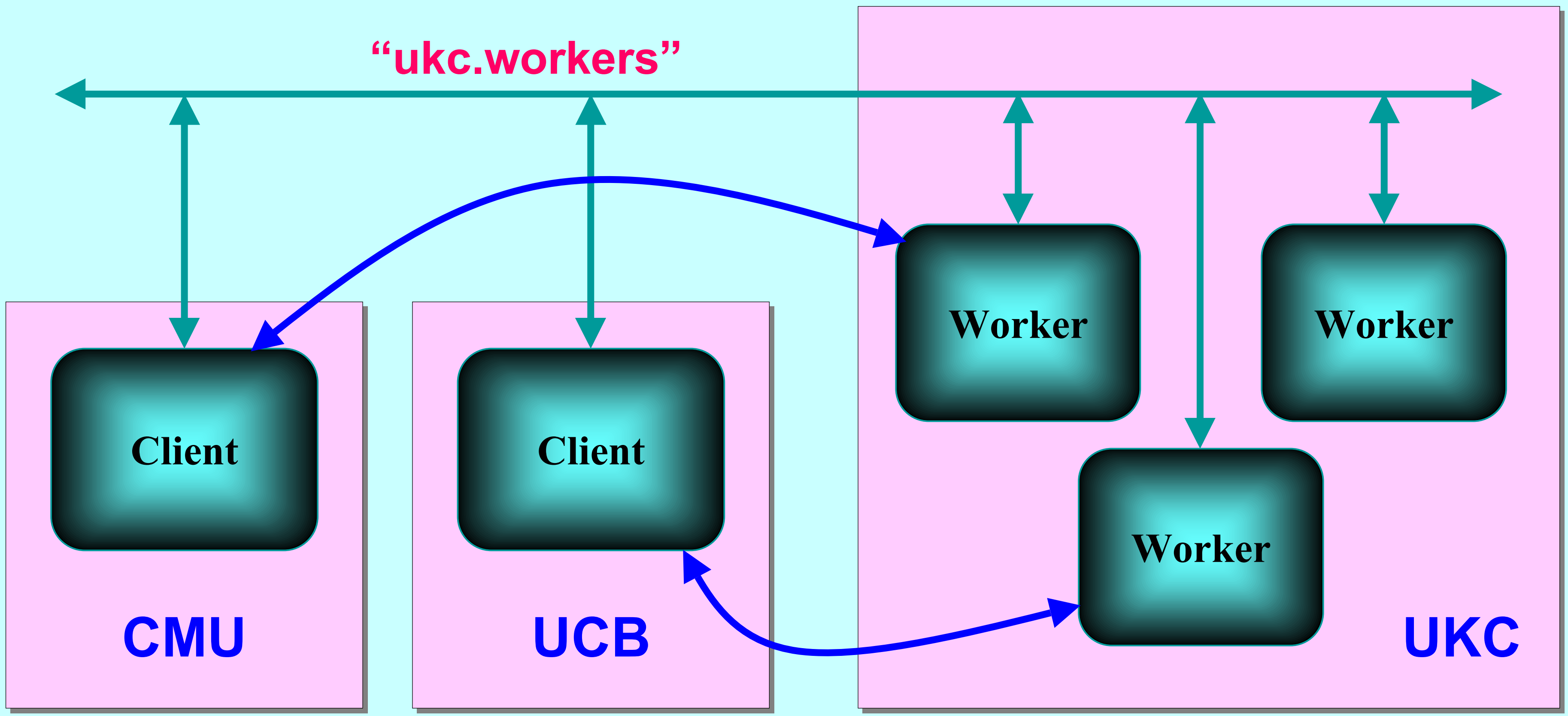
Networked Class Loading



Copyright P.H.Welch

Remote Process Launching

- Example: UKC offers a simple *worker farm* …
- *Clients* grab available *workers* …



Remote Process Launching

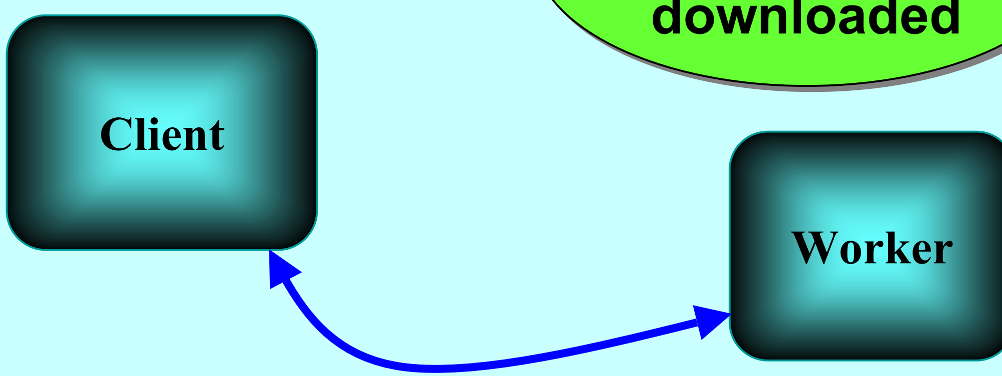
```
Work work = new Work ();  // CSProcess
out.request (work);
work = (Work) out.reply ();
```

client code

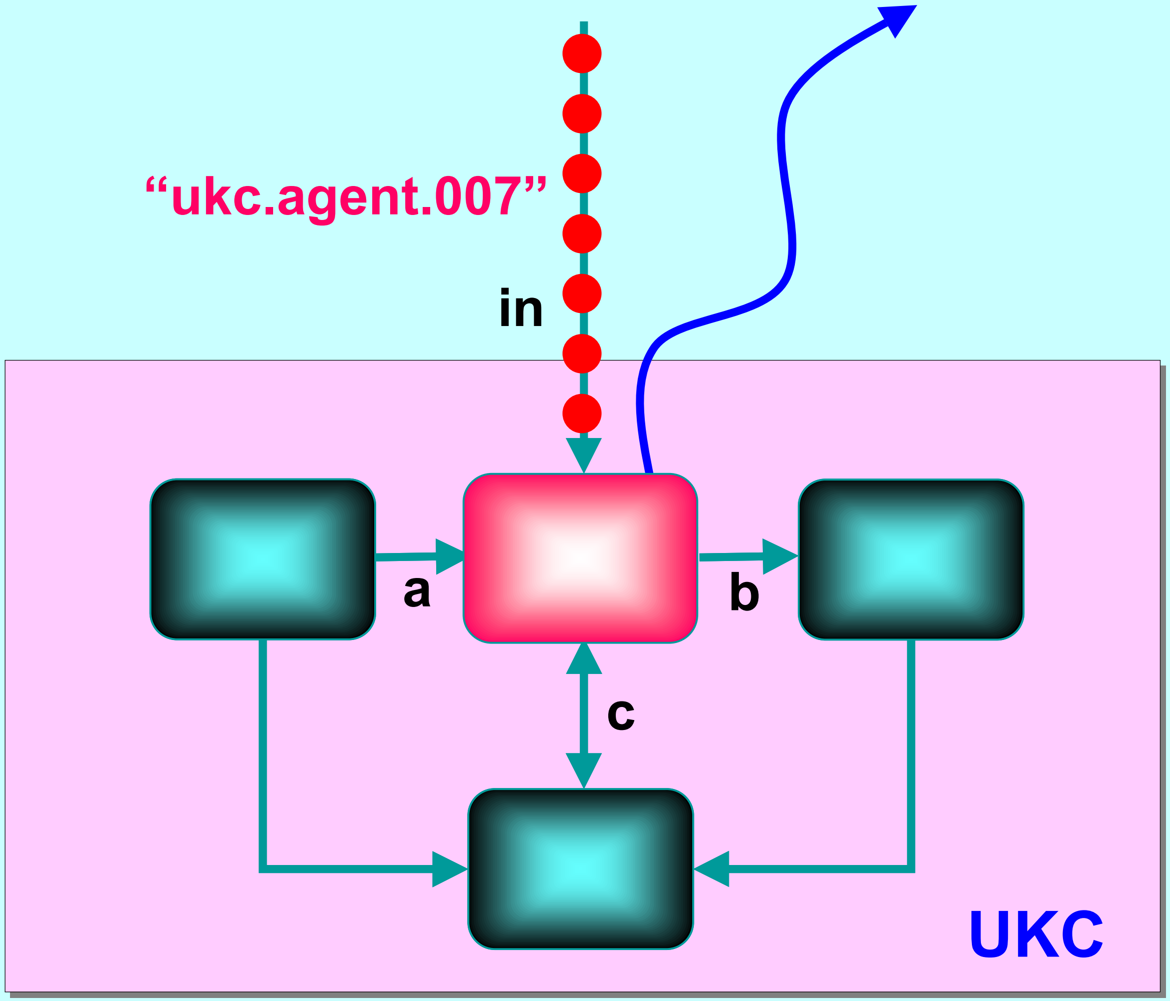
worker code

```
CSProcess work = (CSProcess) in.request ();
work.run ();
in.reply (work);
```

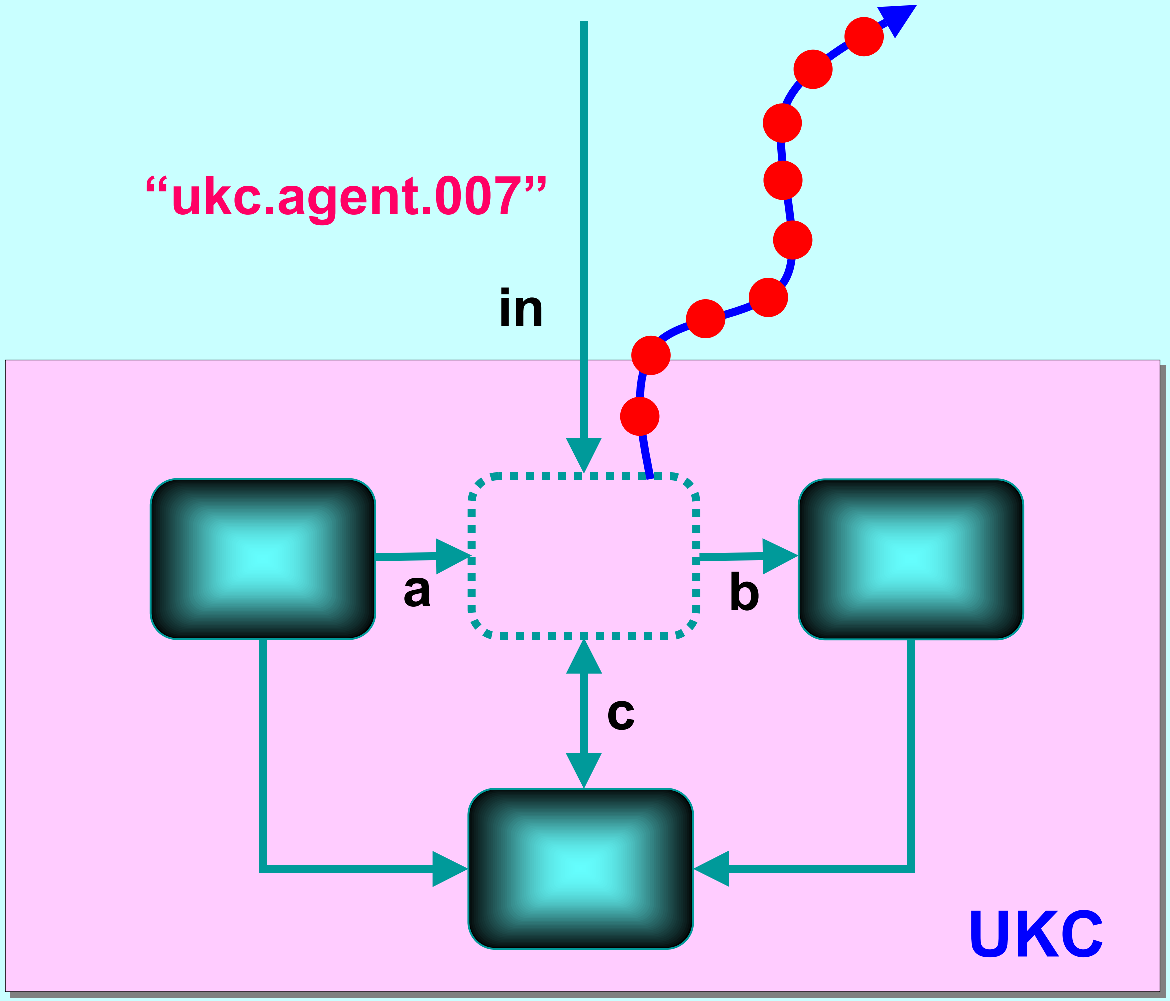
`Work` class file automatically downloaded



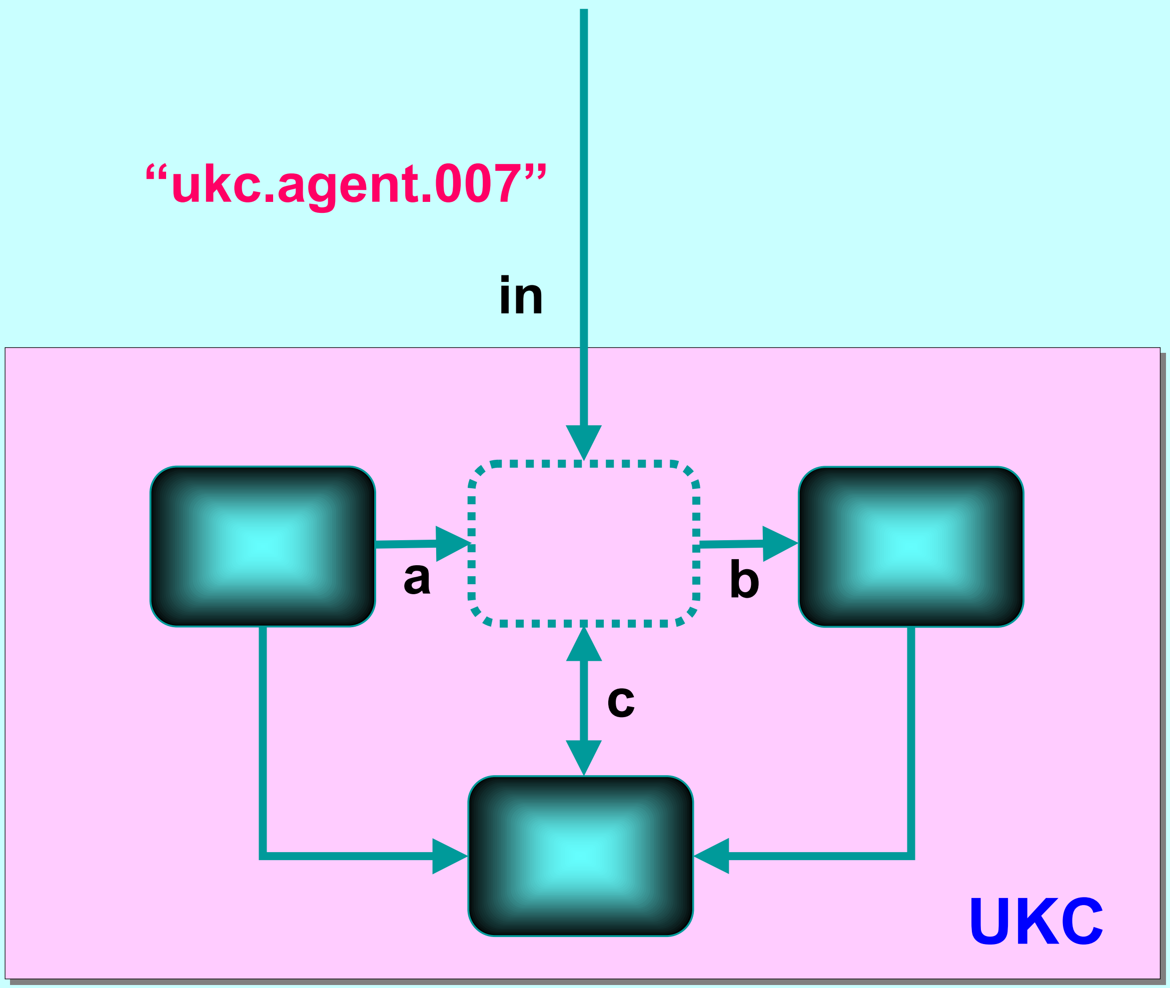
Mobile Processes (Agents)



Mobile Processes (Agents)

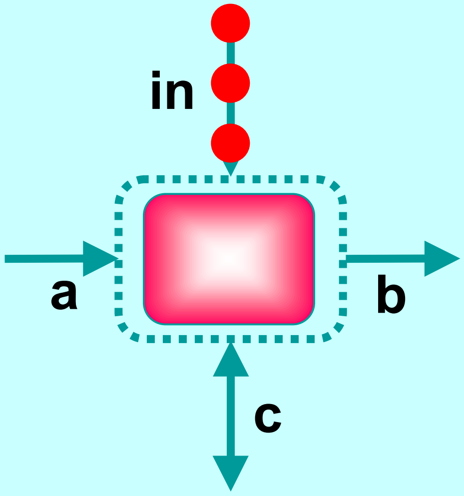


Mobile Processes (Agents)



Mobile Processes (Agents)

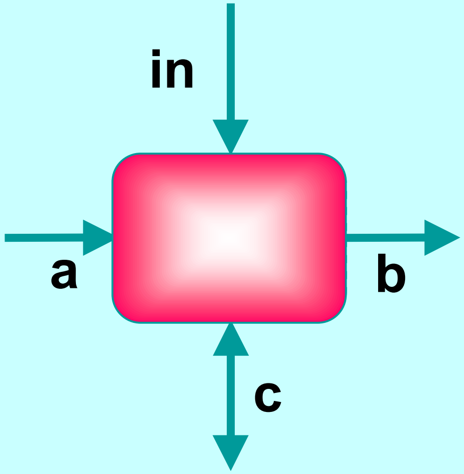
```
while (running) {
  Bond james = (Bond) in.read ();
  james.plugin (a, b, c);
  james.run ();
  NetChannelLocation escapeRoute =
    james.getNextLocation ();
  One2NetChannel escape =
    new One2NetChannel (escapeRoute);
  running = james.getNuke ();
  escape.write (james);
  escape.disconnect ();
}
```



local 007
controller

Mobile Processes (Agents)

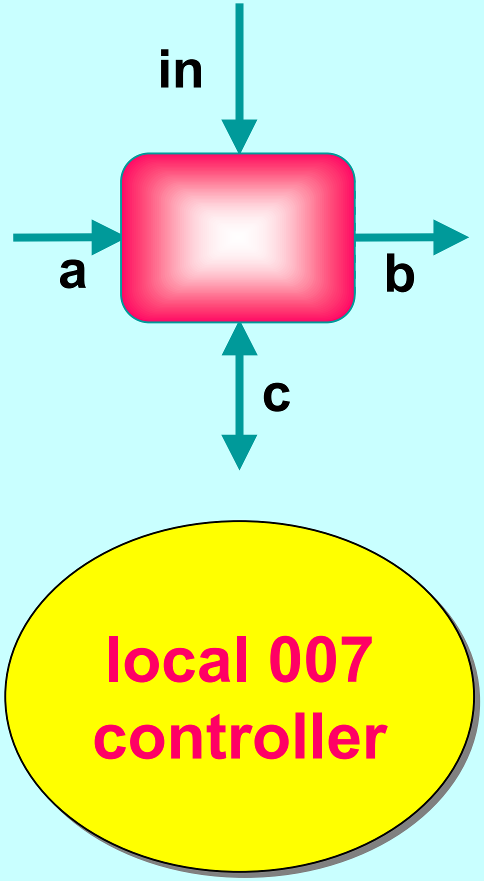
```
while (running) {
  Bond james = (Bond) in.read ();
  james.plugin (a, b, c);
  james.run ();
  NetChannelLocation escapeRoute =
    james.getNextLocation ();
  One2NetChannel escape =
    new One2NetChannel (escapeRoute);
  running = james.getNuke ();
  escape.write (james);
  escape.disconnect ();
}
```



local 007 controller

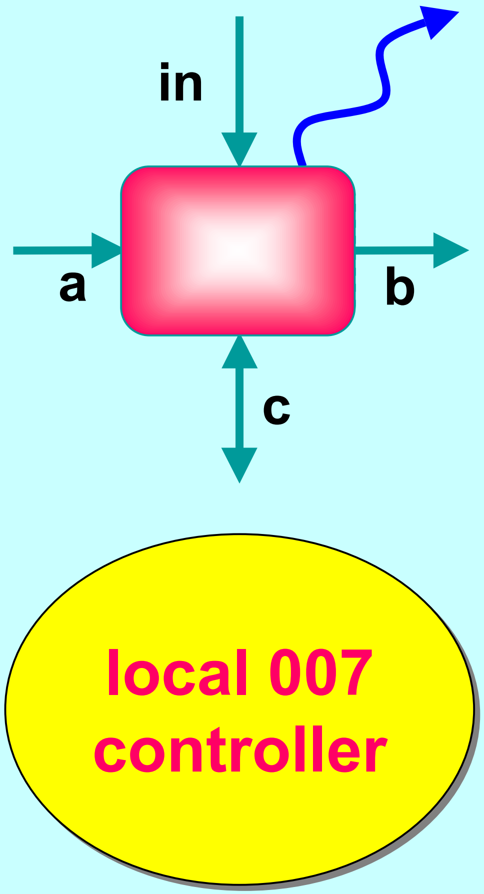
Mobile Processes (Agents)

```
while (running) {
  Bond james = (Bond) in.read ();
  james.plugin (a, b, c);
  james.run ();
  NetChannelLocation escapeRoute =
    james.getNextLocation ();
  One2NetChannel escape =
    new One2NetChannel (escapeRoute);
  running = james.getNuke ();
  escape.write (james);
  escape.disconnect ();
}
```



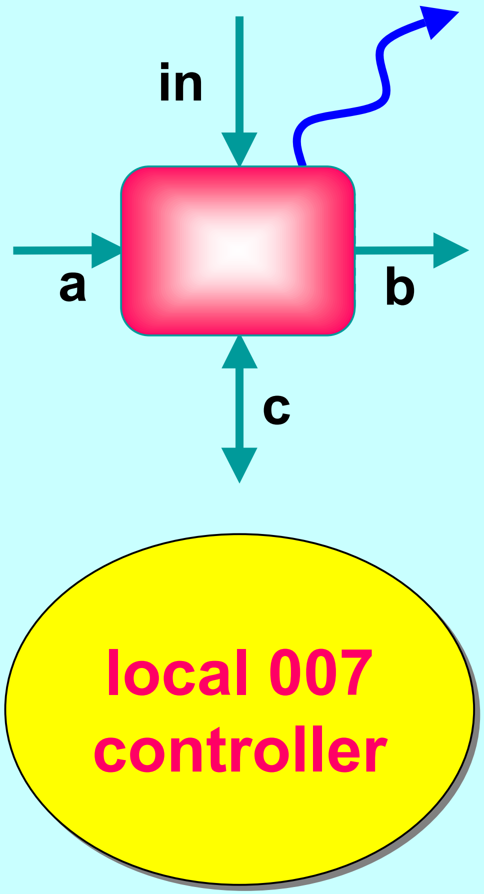
Mobile Processes (Agents)

```
while (running) {
  Bond james = (Bond) in.read ();
  james.plugin (a, b, c);
  james.run ();
  NetChannelLocation escapeRoute =
    james.getNextLocation ();
  One2NetChannel escape =
    new One2NetChannel (escapeRoute);
  running = james.getNuke ();
  escape.write (james);
  escape.disconnect ();
}
```



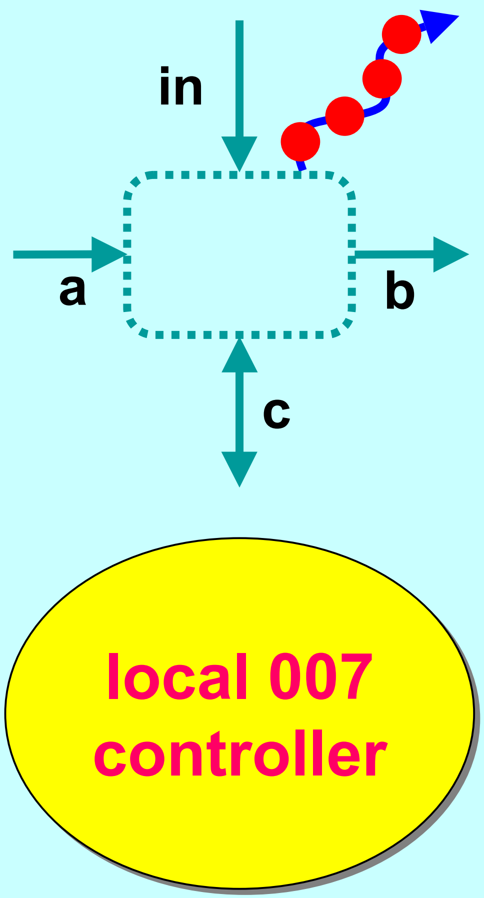
Mobile Processes (Agents)

```
while (running) {
  Bond james = (Bond) in.read ();
  james.plugin (a, b, c);
  james.run ();
  NetChannelLocation escapeRoute =
    james.getNextLocation ();
  One2NetChannel escape =
    new One2NetChannel (escapeRoute);
  running = james.getNuke ();
  escape.write (james);
  escape.disconnect ();
}
```



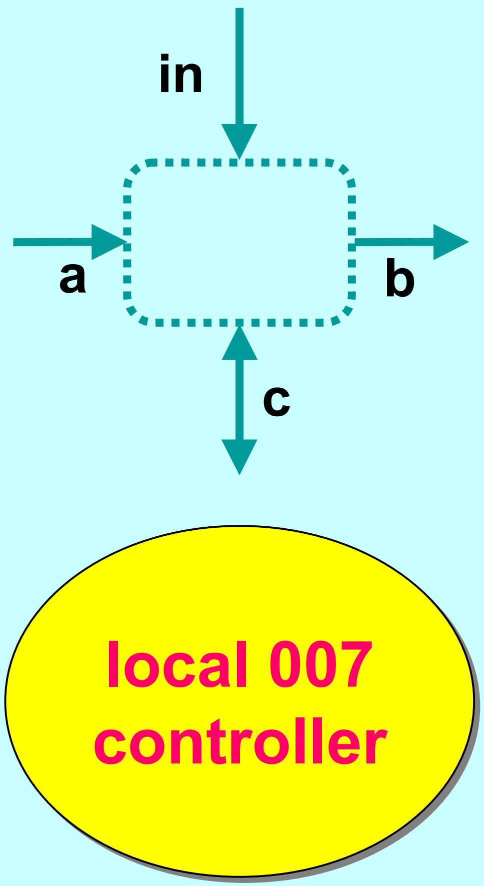
Mobile Processes (Agents)

```
while (running) {
  Bond james = (Bond) in.read ();
  james.plugin (a, b, c);
  james.run ();
  NetChannelLocation escapeRoute =
    james.getNextLocation ();
  One2NetChannel escape =
    new One2NetChannel (escapeRoute);
  running = james.getNuke ();
  escape.write (james);
  escape.disconnect ();
}
```



Mobile Processes (Agents)

```
while (running) {
  Bond james = (Bond) in.read ();
  james.plugin (a, b, c);
  james.run ();
  NetChannelLocation escapeRoute =
    james.getNextLocation ();
  One2NetChannel escape =
    new One2NetChannel (escapeRoute);
  running = james.getNuke ();
  escape.write (james);
  escape.disconnect ();
}
```



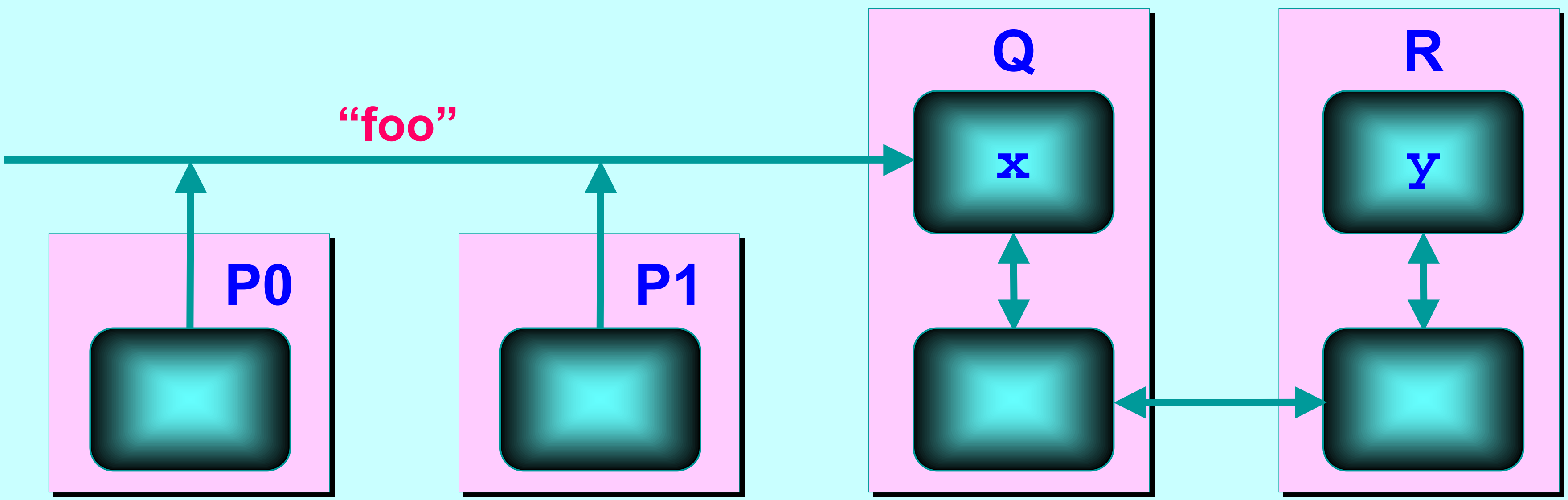
Copyright P.H.Welch

Mobile Network Channels

- Channel *ends* may be moved around a network.
- This is potentially dangerous as we are changing network topology, which may introduce deadlock ***- considerable care must be taken***.
- There is nothing special to do to migrate channel *write-ends*. Network channels are naturally *any-one*. All that is needed is to communicate the *CNS channel name* (or `NetChannelLocation`) to the new writer process.
- Migrating channel *read-ends* securely requires a special protocol …

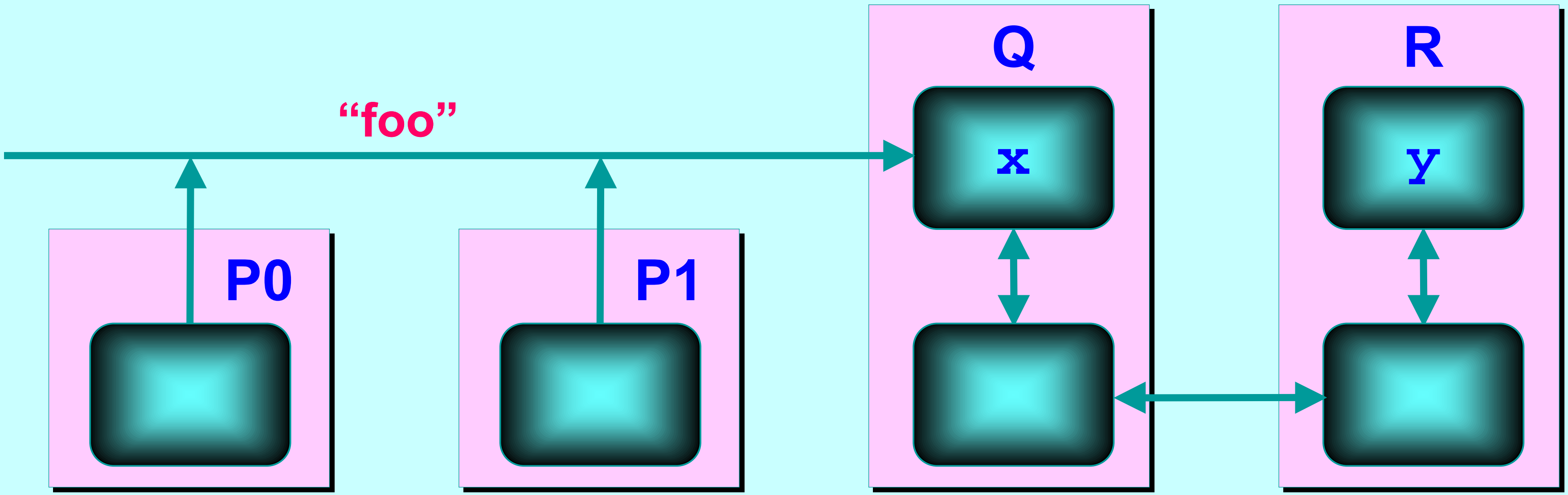
Mobile Network Channels

- Consider a process, `x`, on node **Q**, currently servicing the *CNS-registered* channel **“foo”**.
- It wants to pass on this responsibility to a (willing) process, `y`, in node **R**, with whom it is in contact.



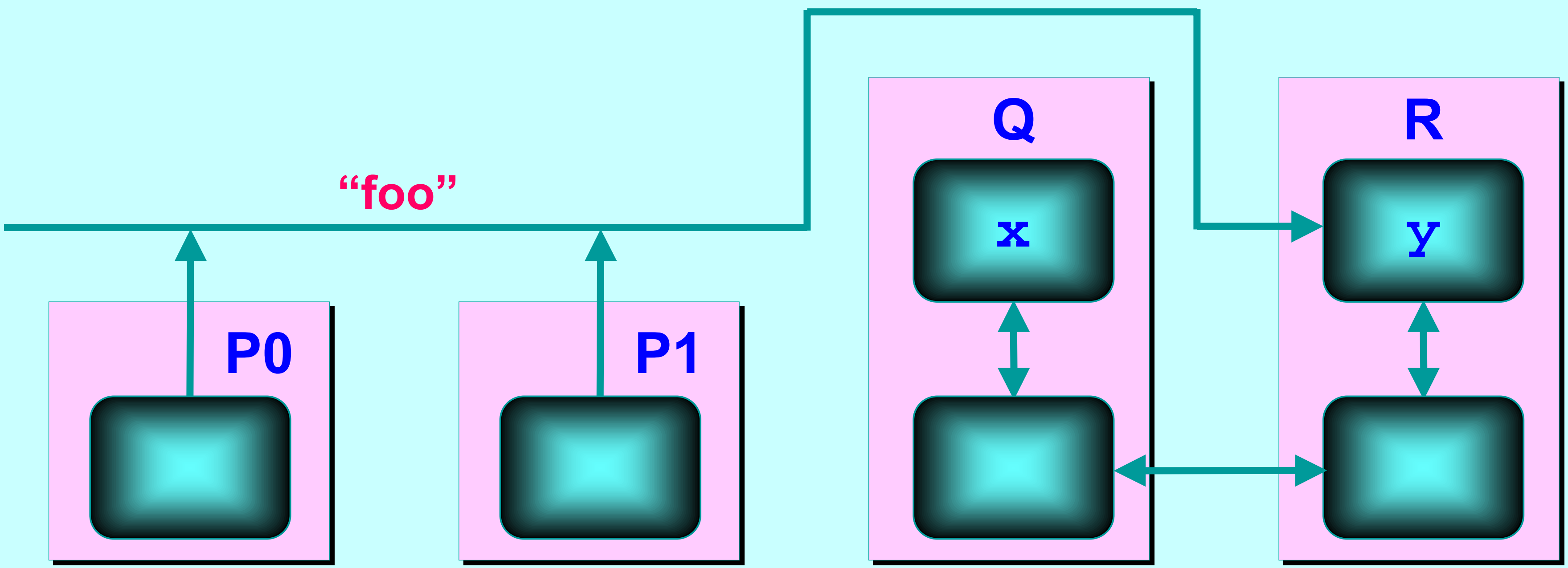
Mobile Network Channels

- Processes writing to **“foo”** are to be unaware of this channel migration.



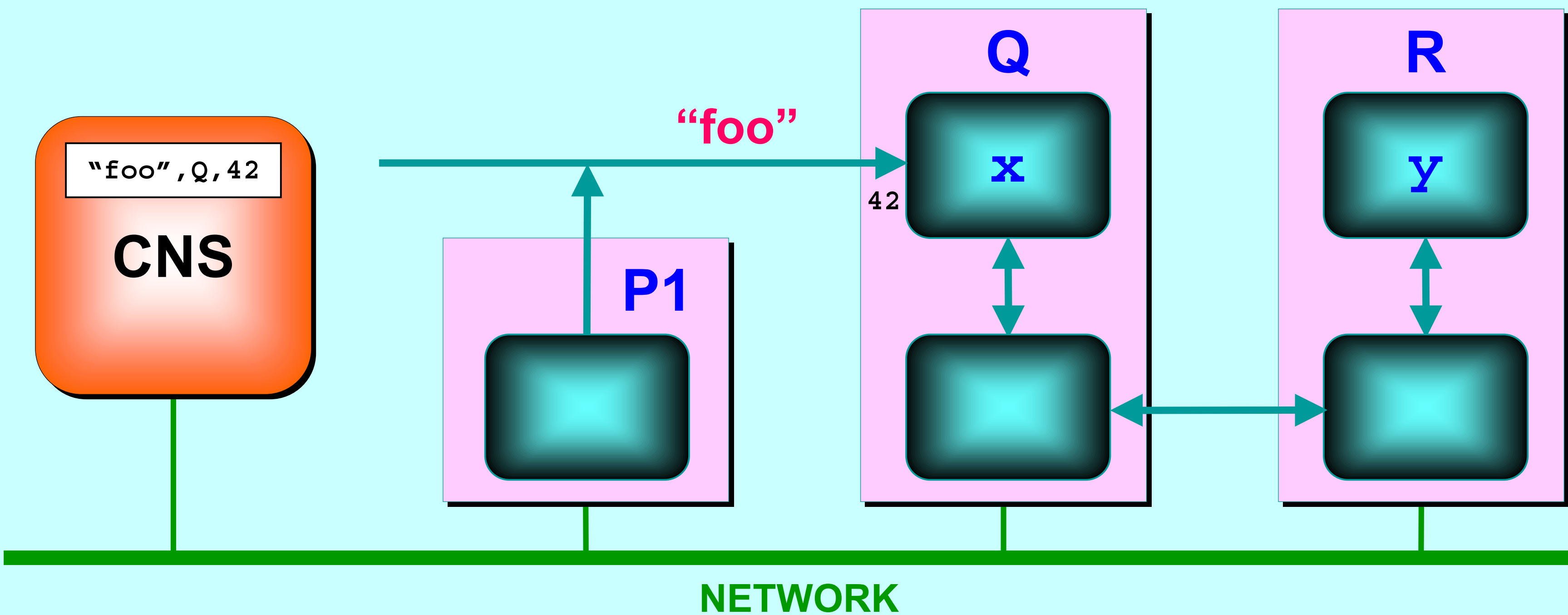
Mobile Network Channels

- Processes writing to **"foo"** are to be unaware of this channel migration.



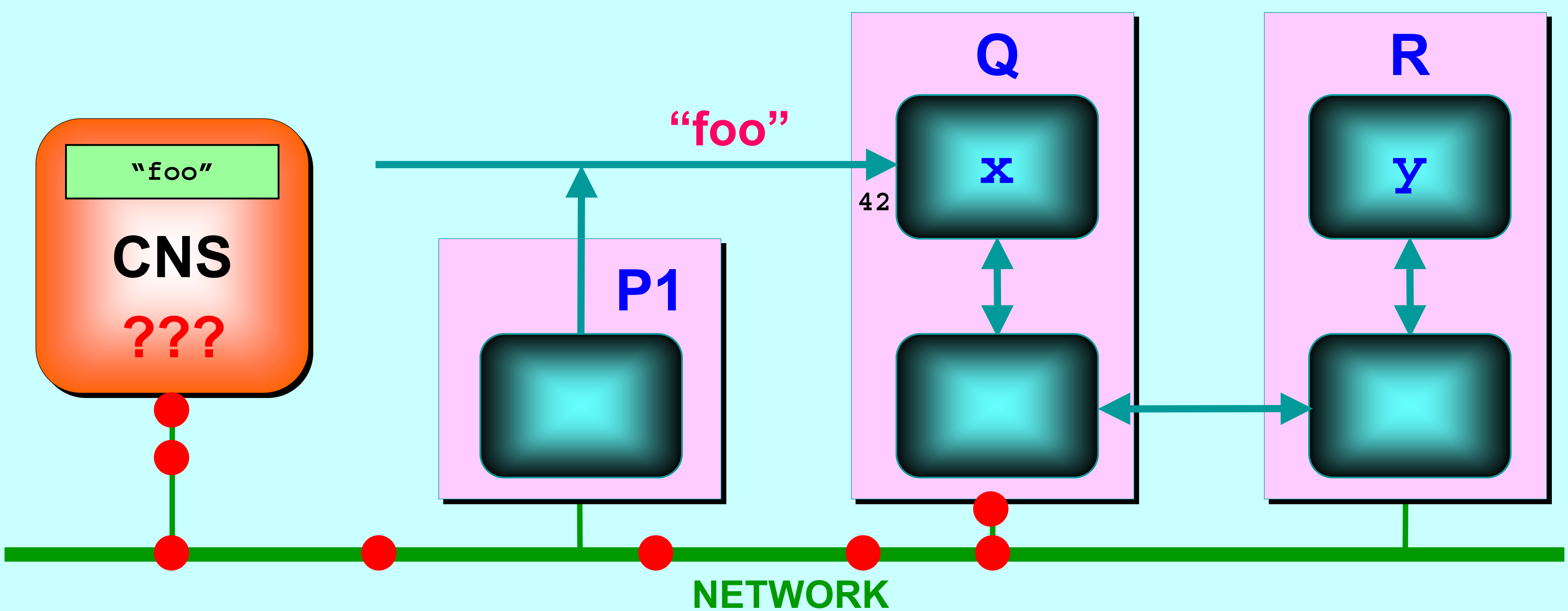
Mobile Network Channels

- Let's get back to the initial state (**"foo"** being serviced by `x` on node **Q**).
- Let's show the network … and the CNS …



Mobile Network Channels

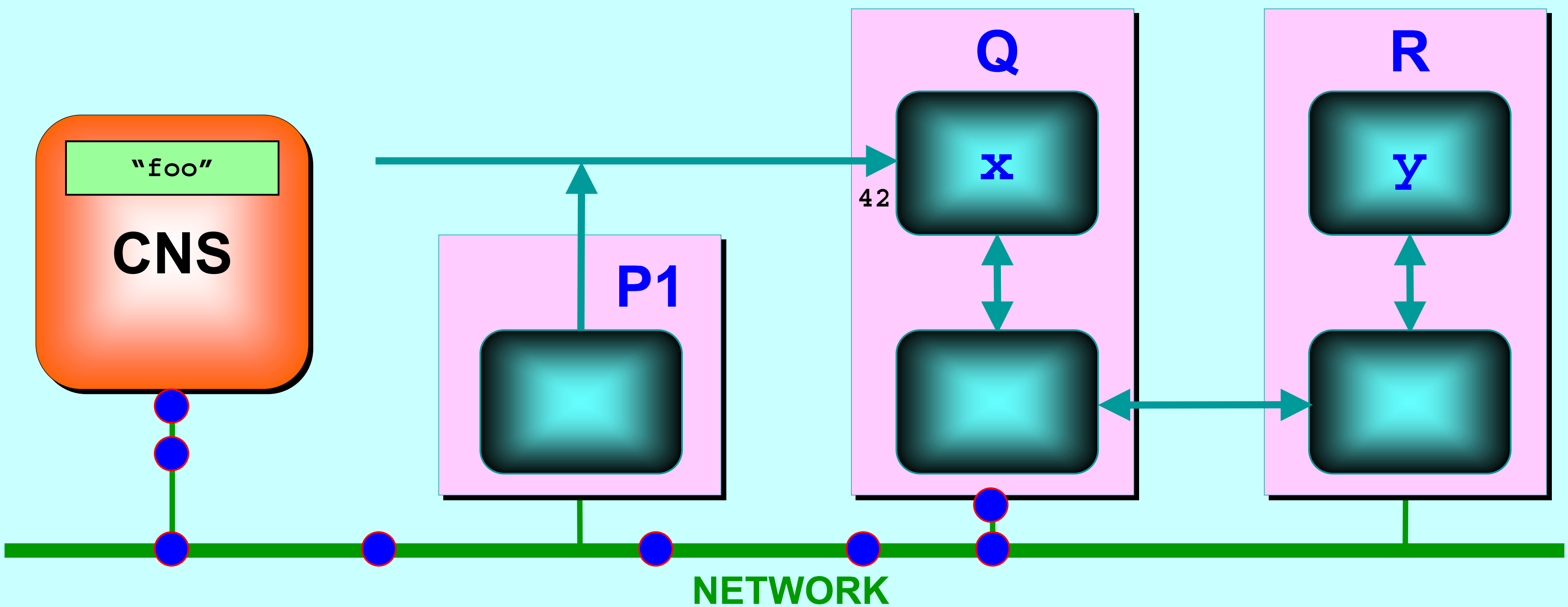
- First, process **x** *freezes* the name **"foo"** on the CNS …



Copyright P.H.Welch

Mobile Network Channels

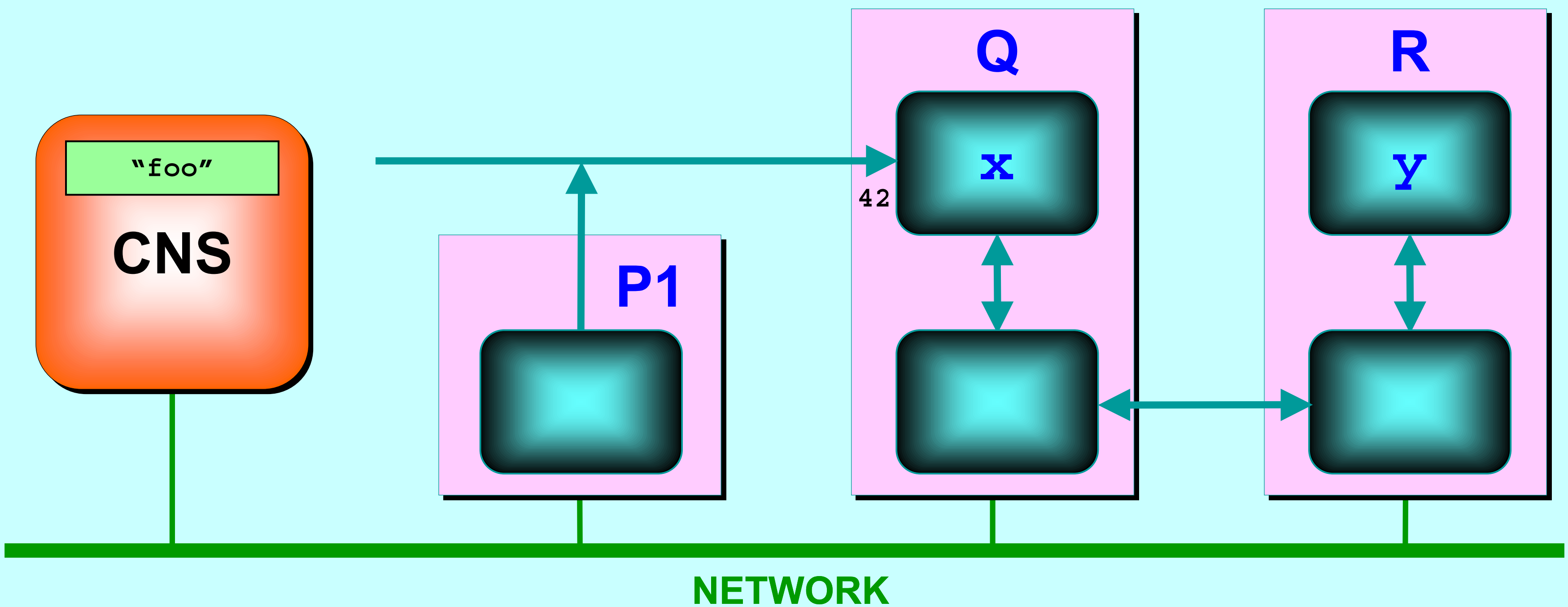
- First, process `x` *freezes* the name **"foo"** on the CNS …
- The CNS returns an *unfreeze key* to process `x` …



Copyright P.H.Welch

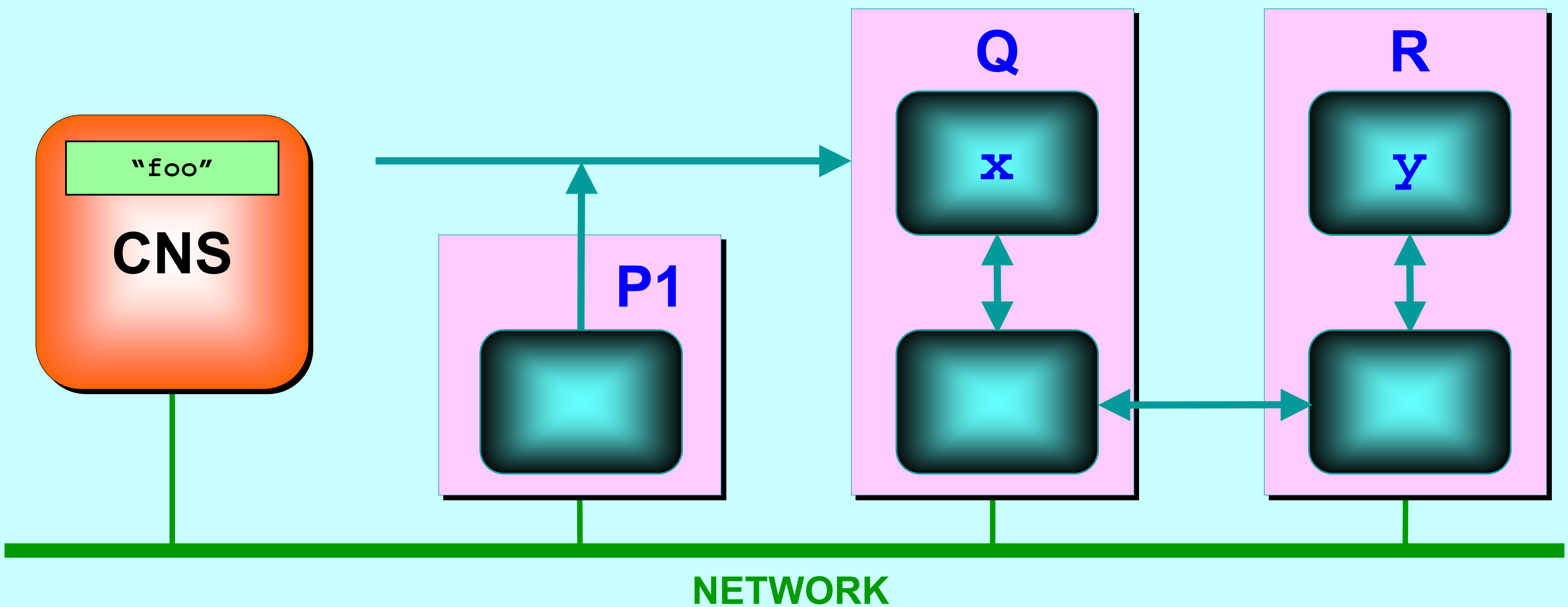
Mobile Network Channels

- The CNS no longer resolves **"foo"** for new writers and also disallows new registrations of the name.
- The network channel is deleted from processor **Q**.



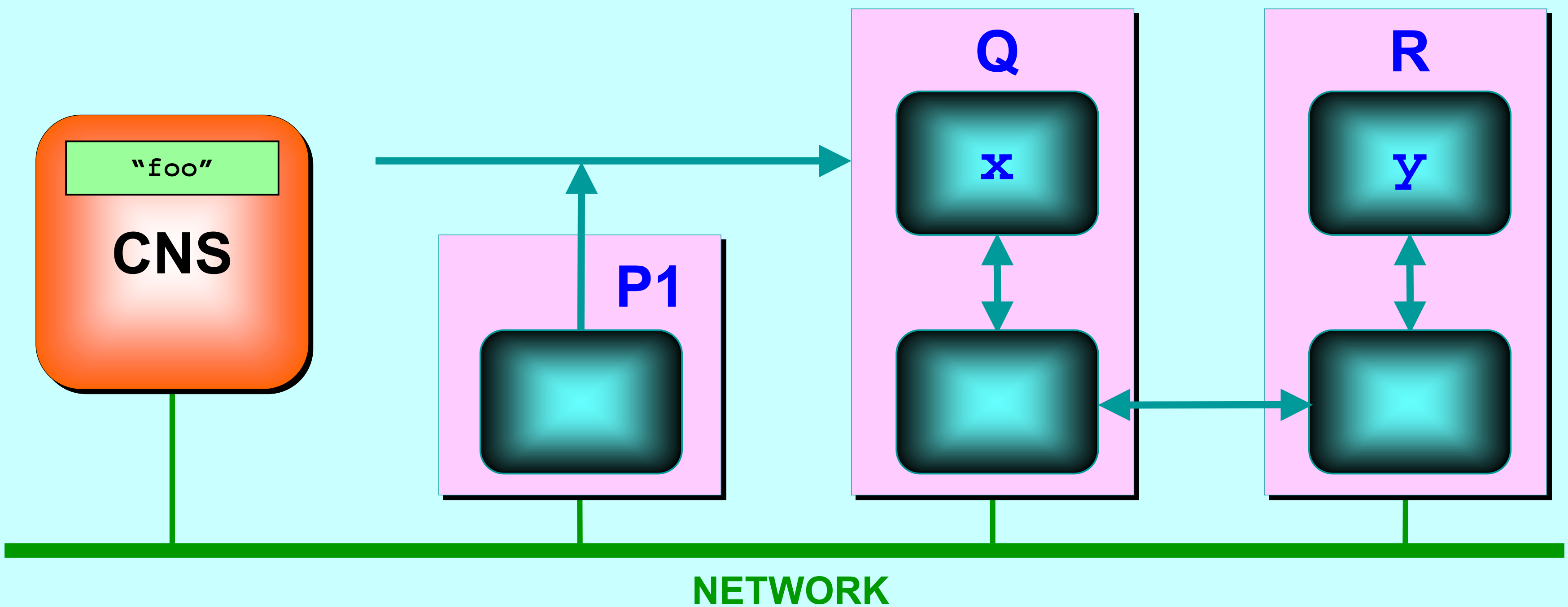
Mobile Network Channels

- The network channel is deleted from processor **Q**.
- Any *pending* and *future* messages for that channel (`42`) on **Q** are bounced (`NetChannelIndexException`).



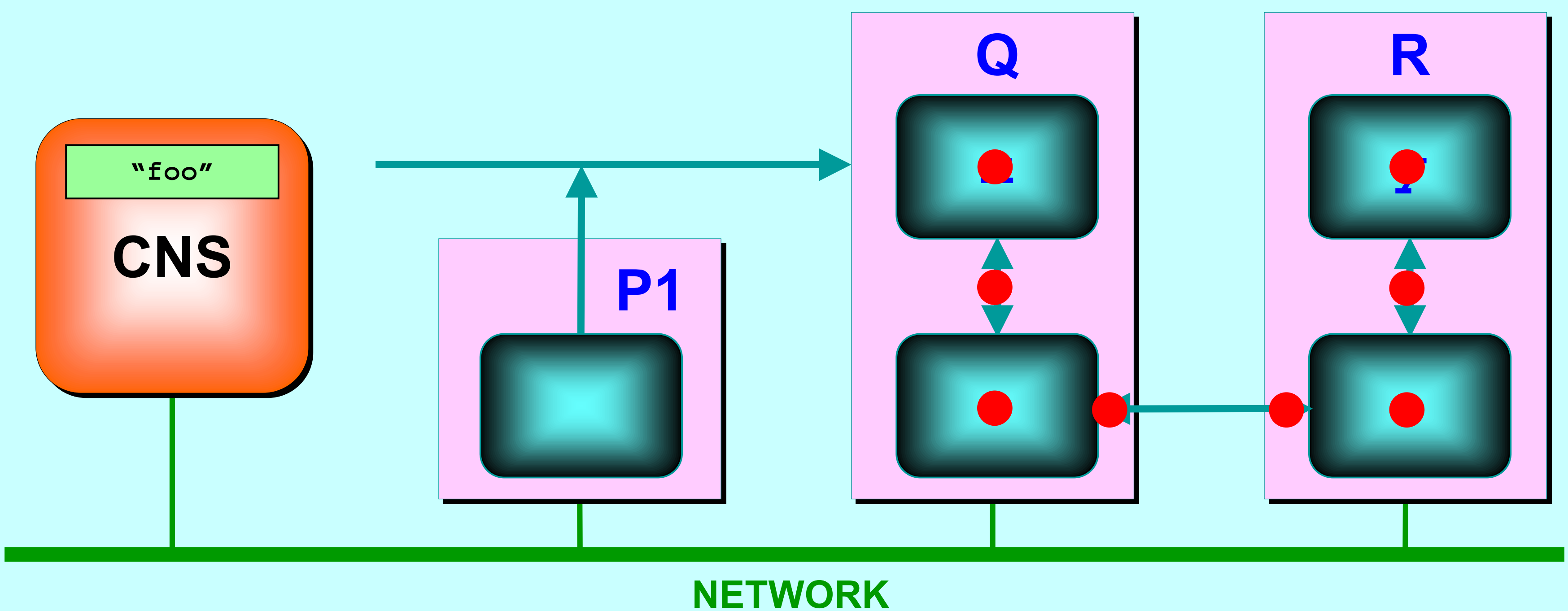
Mobile Network Channels

- The `write()` method at **P1** handles that bounce by appeal to the CNS for the new location of **"foo"**.
- This will not succeed until …



Mobile Network Channels

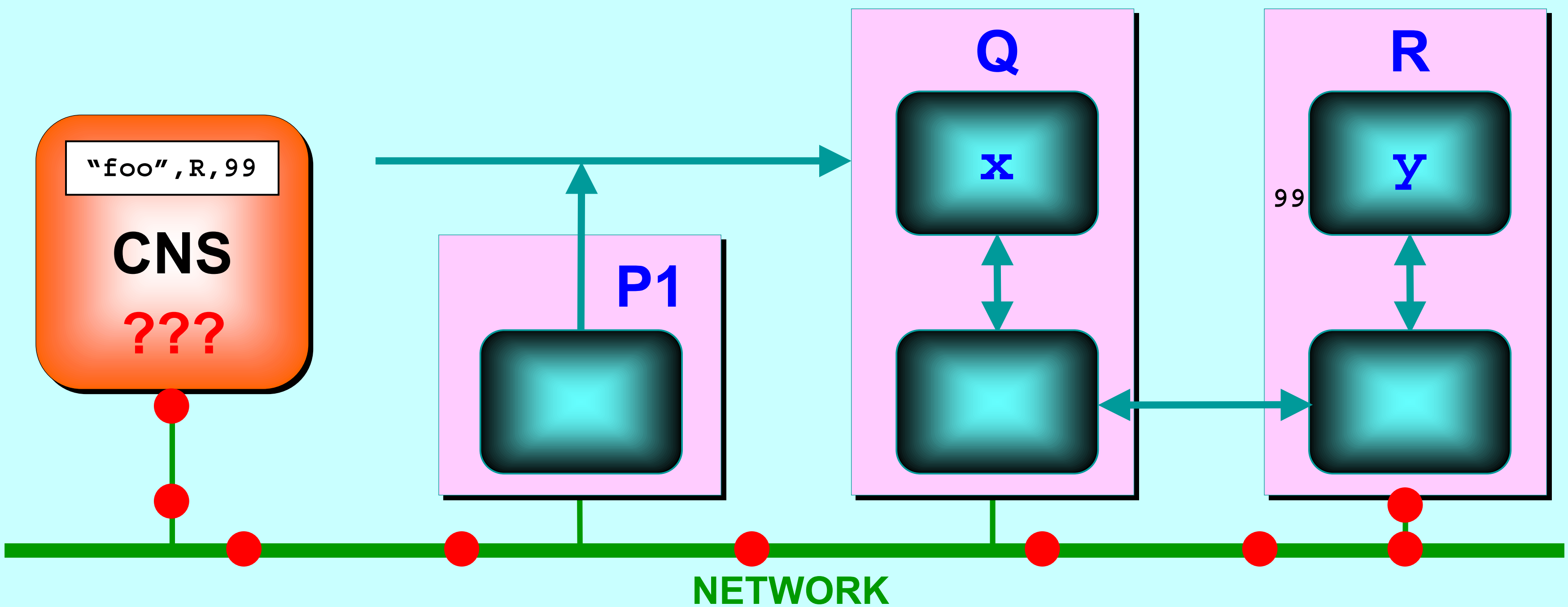
- … process **x** (on node **Q**) passes on the channel name (**"foo"**) and *CNS unfreeze key* …



Copyright P.H.Welch

Mobile Network Channels

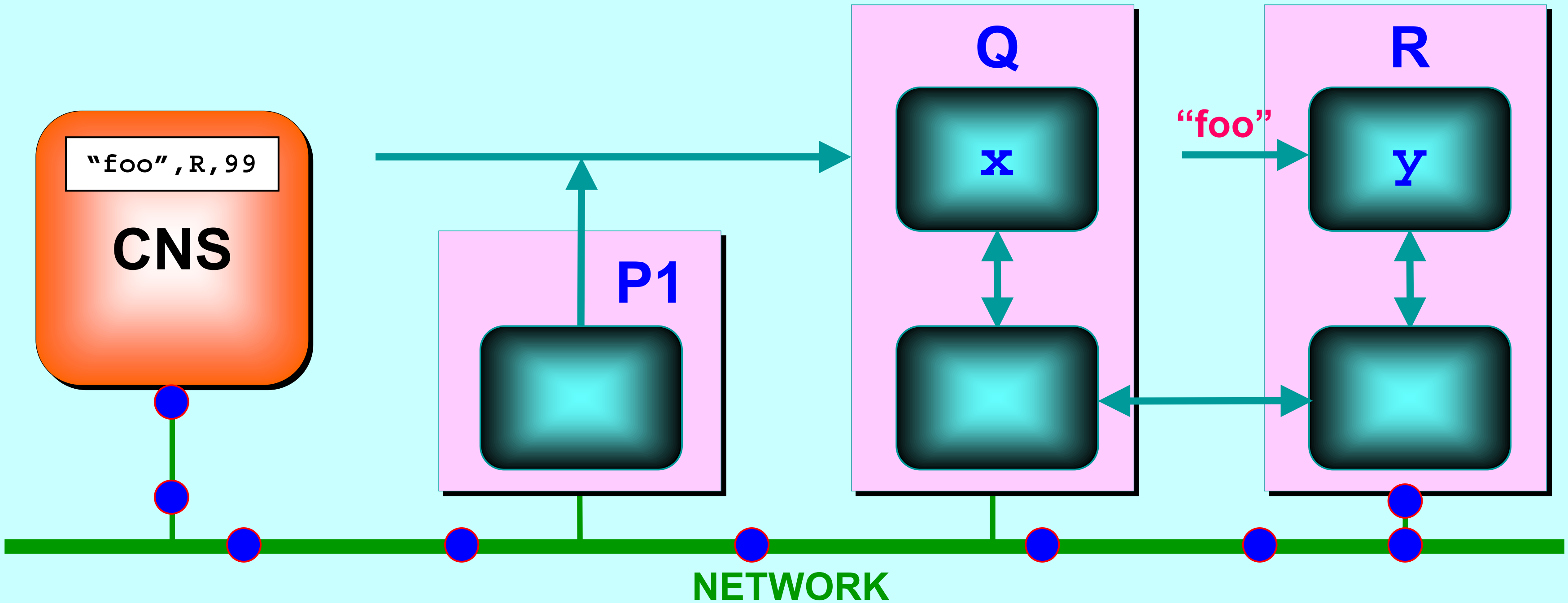
- … process **x** (on node **Q**) passes on the channel name (**"foo"**) and *CNS unfreeze key* …
- … and the receiver (process **y** on **R**) unlocks the name **"foo"** (using the *key*) and re-registers it.



Copyright P.H.Welch

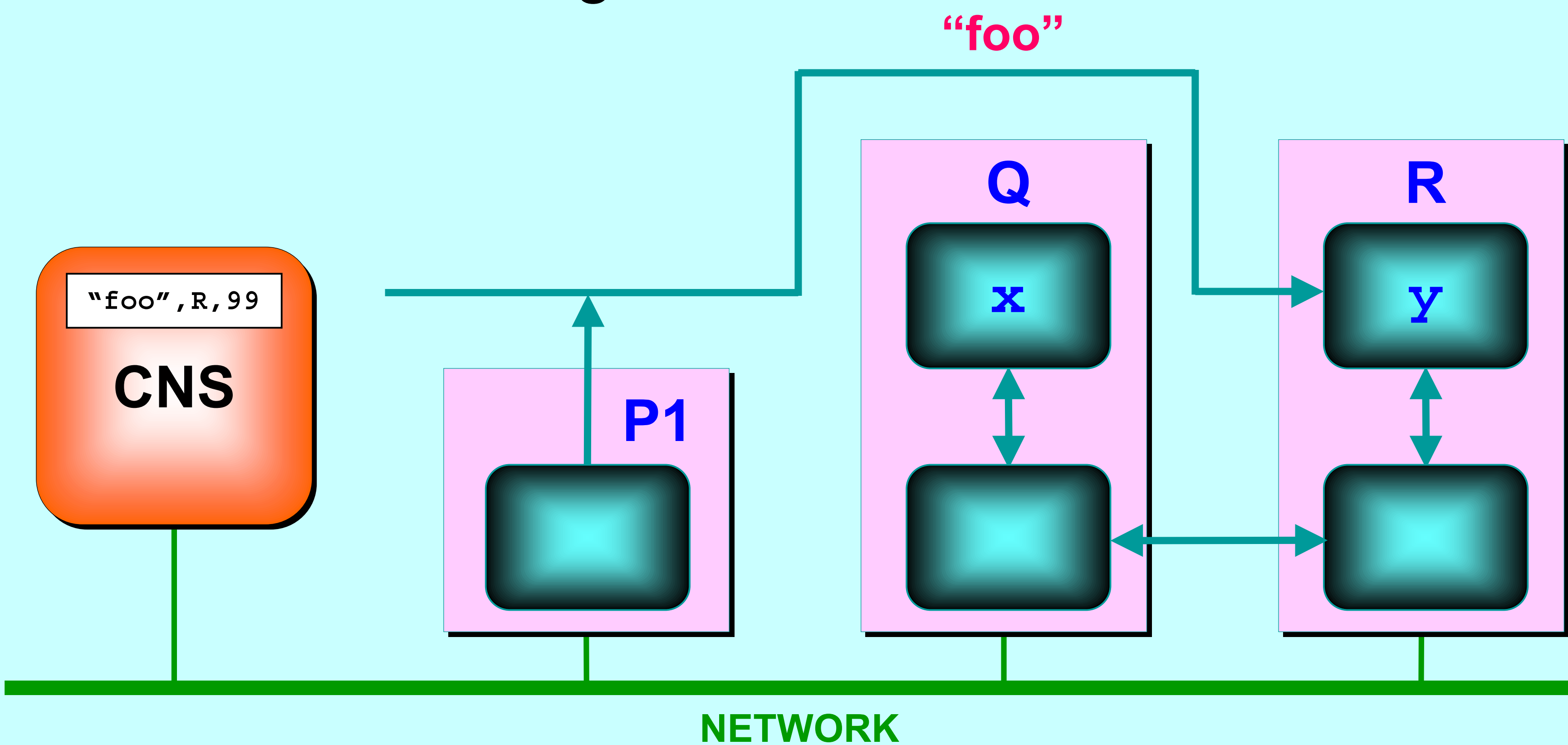
Mobile Network Channels

- … and the receiver (process `y` on **R**) unlocks the name **"foo"** (using the *key*) and re-registers it.
- The **`write()`** method at **P1** now hears back from the CNS the new location of **"foo"** …



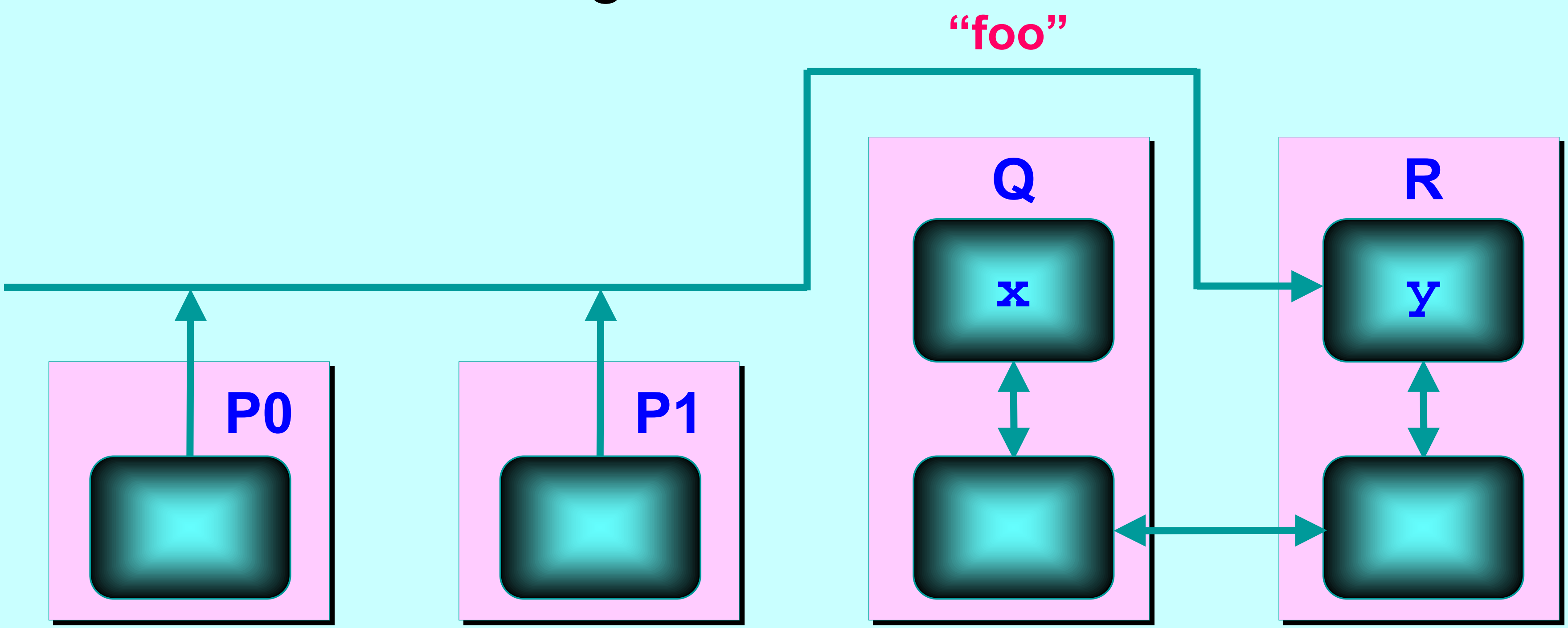
Mobile Network Channels

- … and resends the message that was bounced.
- The writing process(es) at **P1** (and elsewhere) are unaware of the migration.



Mobile Network Channels

- … and resends the message that was bounced.
- The writing process(es) at **P1** (and elsewhere) are unaware of the migration.



Mobile Network Connections

- Connection *ends* may be moved around a network.
- This is potentially dangerous as we are changing network topology, which may introduce deadlock ***- considerable care must be taken***.
- There is nothing special to do to migrate connection *client-ends*. Network connections are naturally *any-one*. All that is needed is to communicate the *CNS connection name* (or `NetConnectionLocation`) to the new writer process.
- Migrating *server-ends* safely requires a special protocol … **the same as for channel *write-ends*.**

Summary

- ***JCSP.net*** enables ***virtual channel communication*** between processes on separate machines (JVMs).
- Application ***channels/connections*** between machines are set up (and taken down) dynamically.
- Channels/connections are multiplexed over ***links***.
- Links can be developed for ***any network protocol*** and plugged into the ***JCSP.net*** infrastructure.
- No central management – ***peer-to-peer*** connections (bootstrapped off a basic *Channel Name Server*).
- Brokers for ***user-definable matching services*** are easy to set up as ordinary application servers.

Summary

- Processes can ***migrate*** between processors (with classes loaded dynamically as necessary) – hence *mobile agents, worker farms, grid computation …*
- ***JCSP.net*** provides ***exactly the same*** (**CSP/occam**) concurrency model for networked systems as ***JCSP*** provides within each physical node of that system.
- Network logic is ***independent of physical distribution*** (or even whether it is distributed).
- Major emphasis on ***simplicity*** – both in setting up application networks and in reasoning about them.
- ***Lot's of fun*** to be had – but still some work to do.

Acknowledgements

The ***application*** slides (**71- 80** inclusive) report joint work with Brian Vinter of the Department of Mathematics and Computer Science, University of Southern Denmark, Odense, Denmark - **`<vinter@imada.sdu.dk>`**

These results are reported in: *"Cluster Computing and JCSP Networking"*, B.Vinter and P.H.Welch, in *'Communicating Process Architectures 2002'*, vol. 60 Concurrent Systems Engineering, pp. 203-222, IOS Press, Amsterdam, September 2002.

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

wotug.ukc.ac.uk/

Copyright P.H.Welch

Stop Press

JCSP Networking Edition

JCSP.net

www.quickstone.com