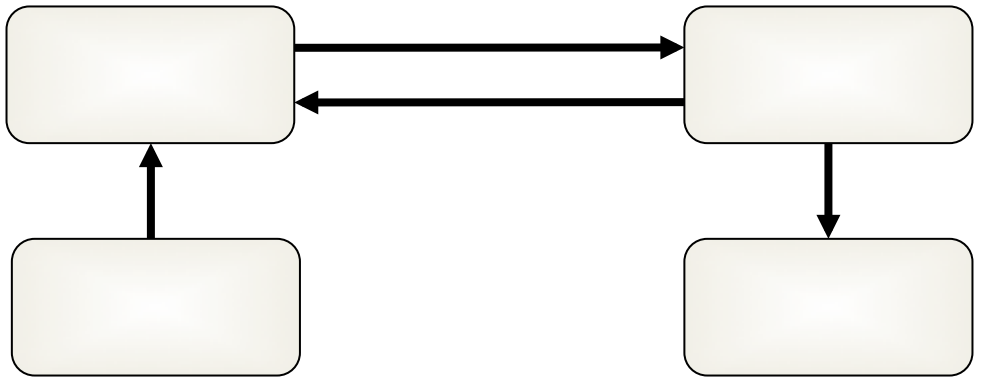
Distribution and Mobility

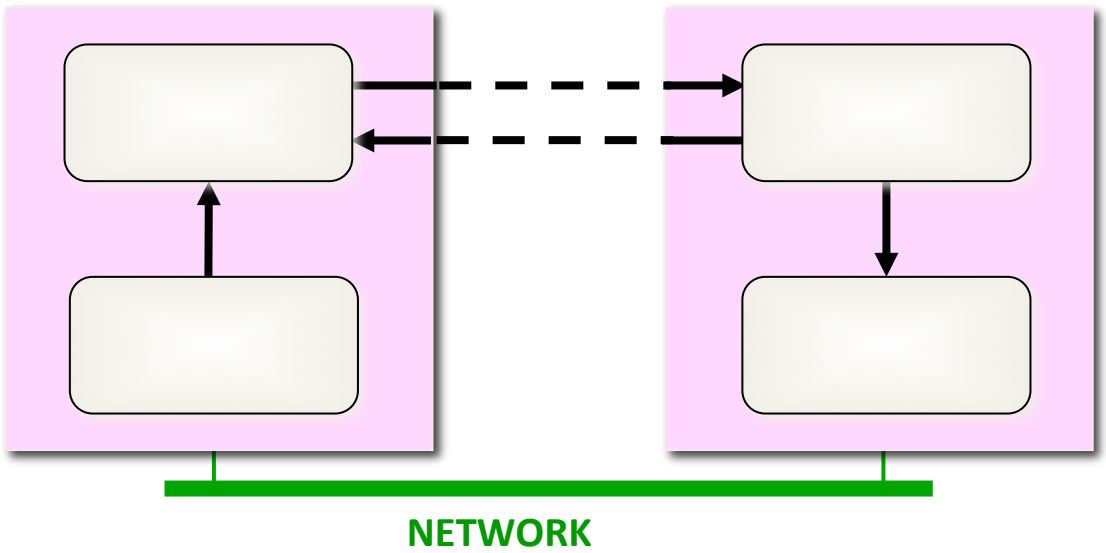
Distributed JCSP

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



Distributed JCSP

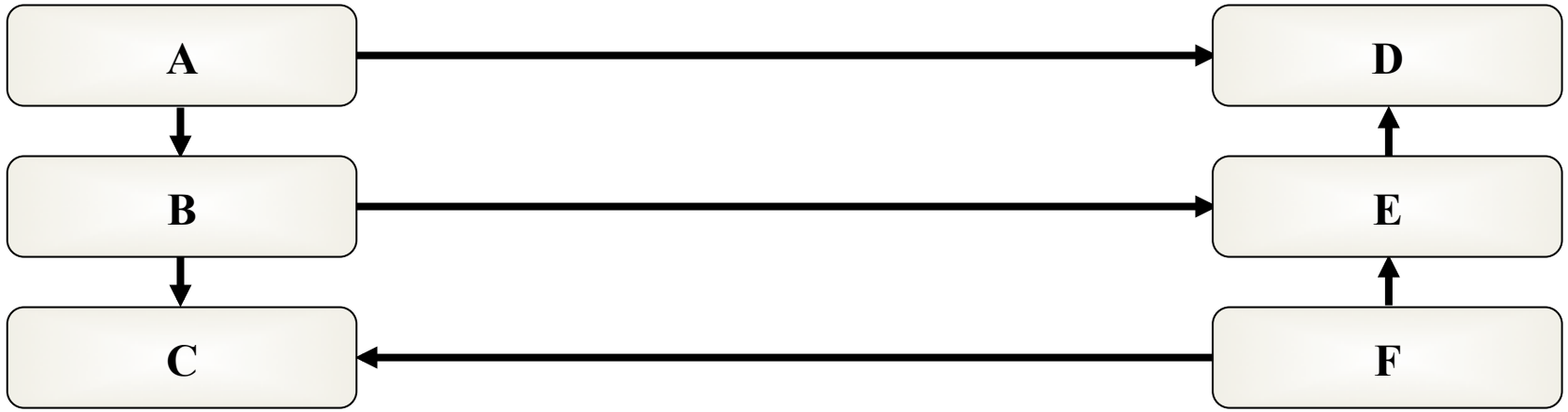
- Want to use the same model for concurrent processes *whether or not they are on the same machine*:



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

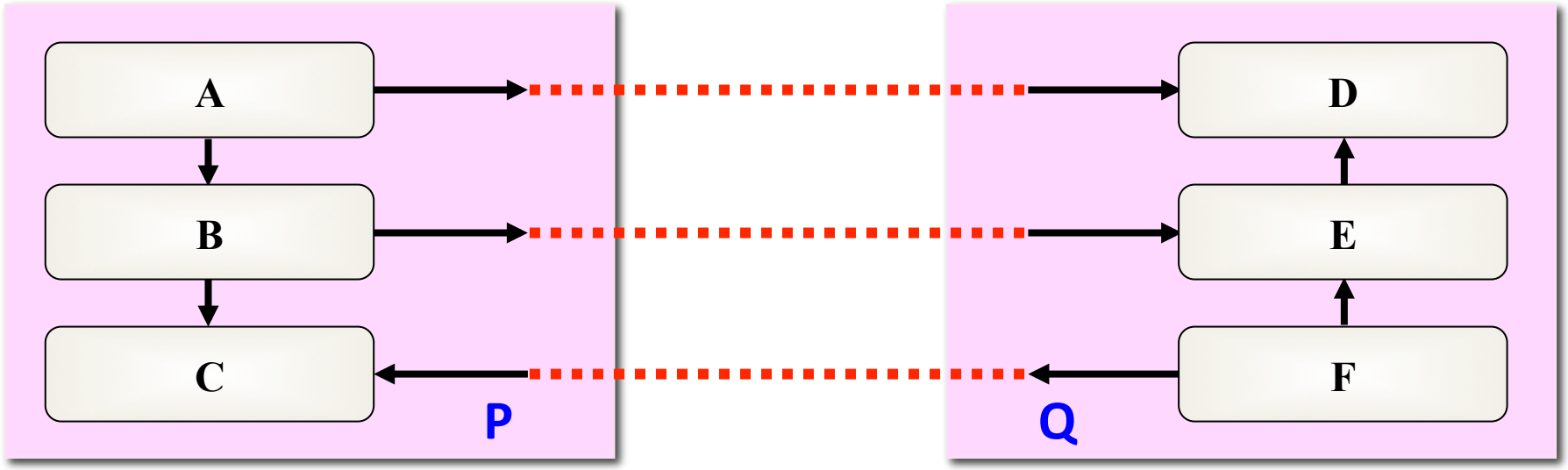
Logical Network

- 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 …



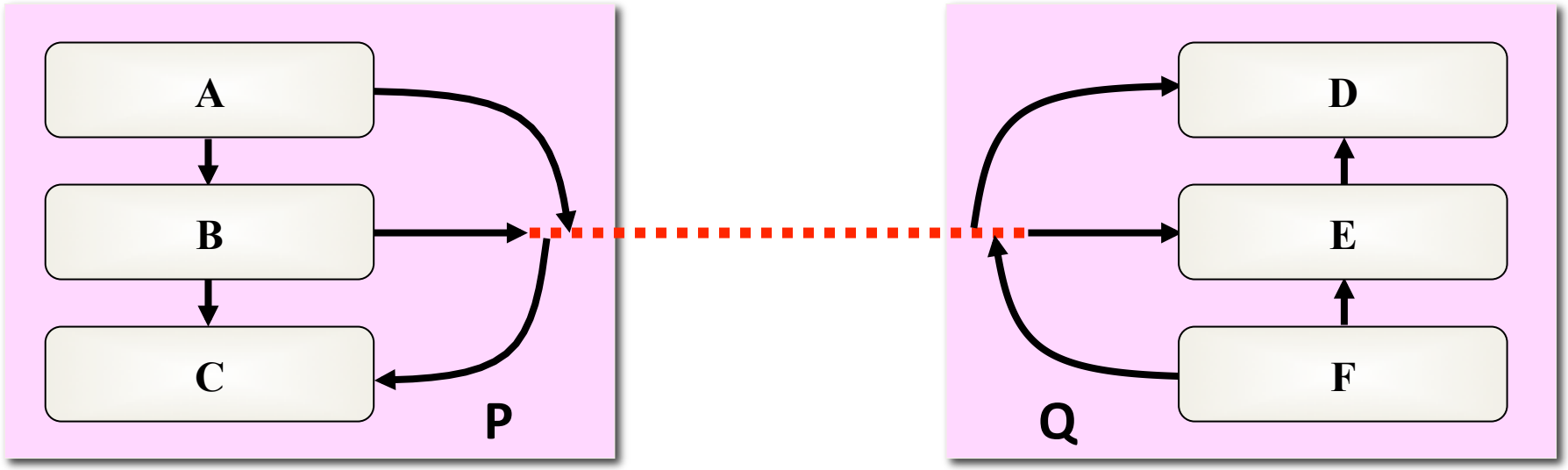
Physical Network

- 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 …



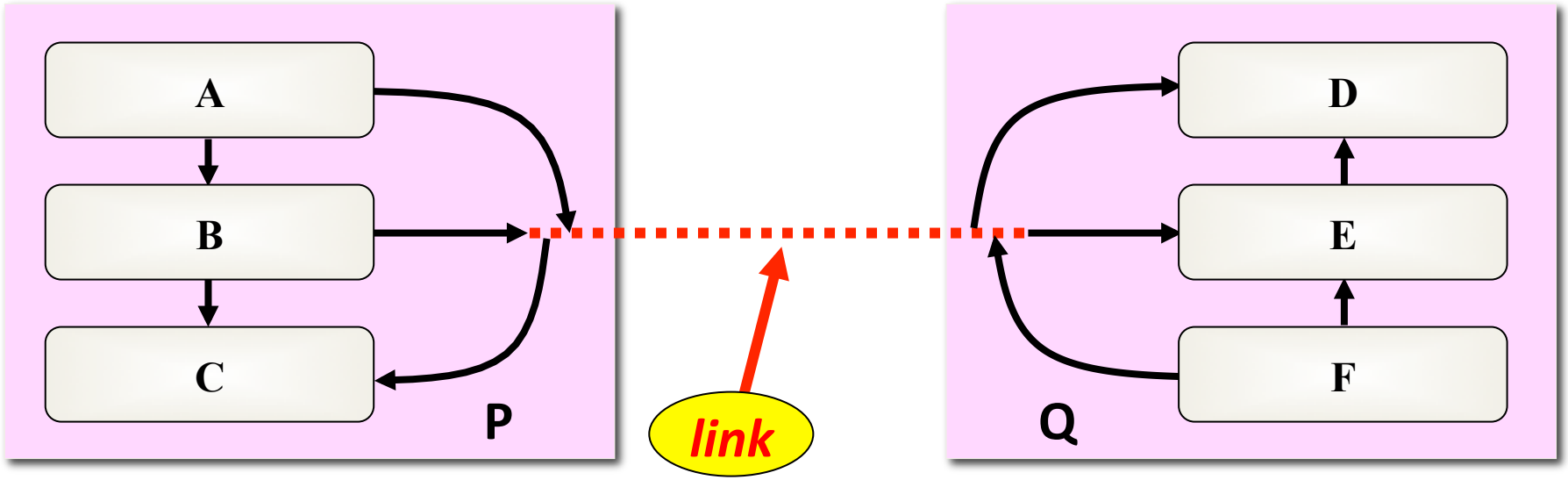
Physical Network

- 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 …



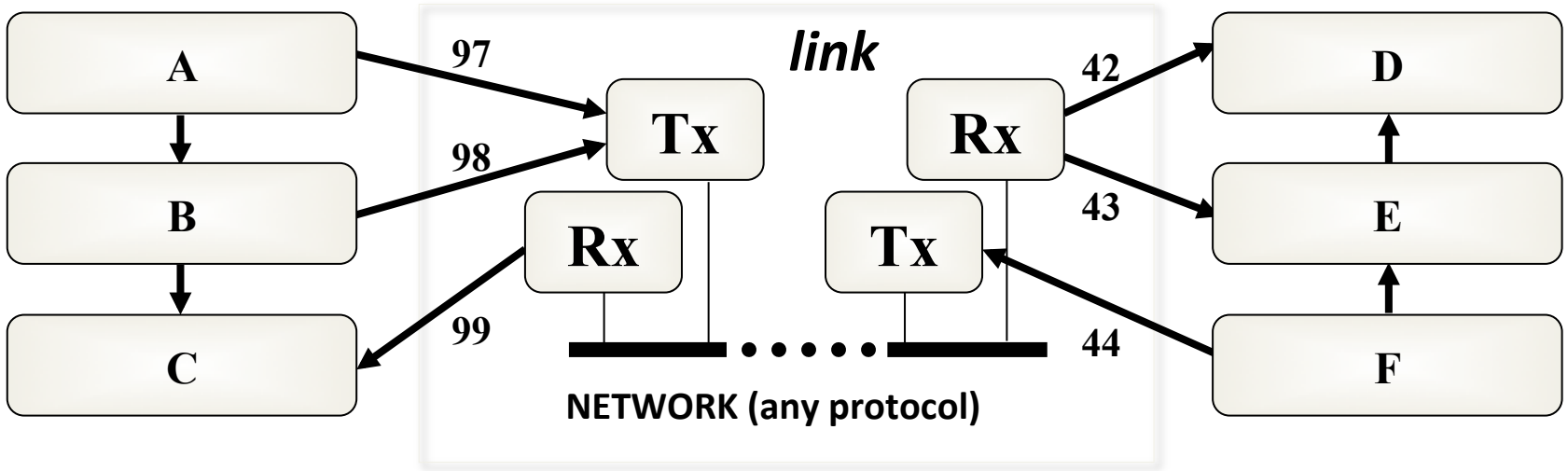
JCSP Links

- A connection between two processing nodes (*JVM*s 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*, …).

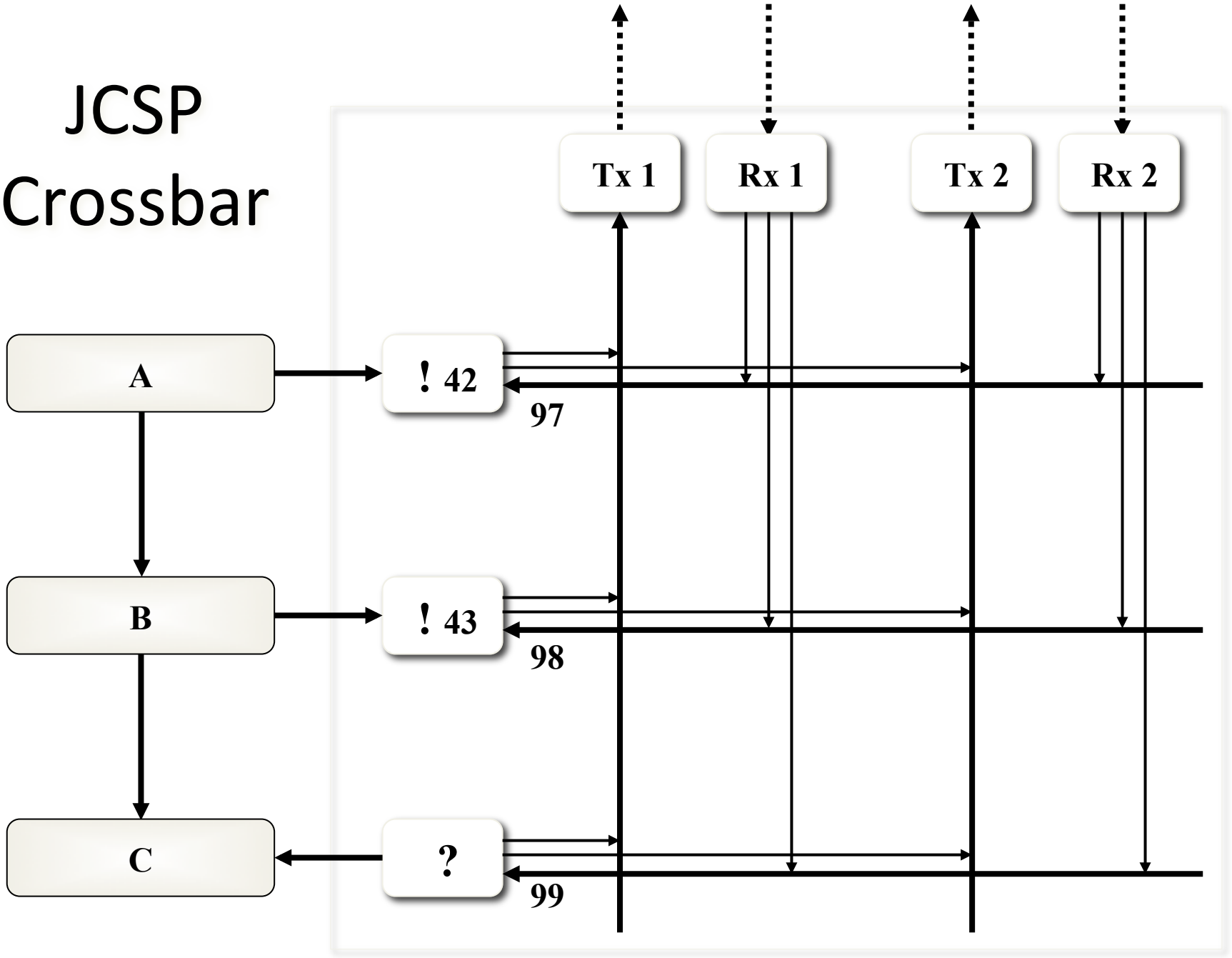


JCSP Links

- 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.

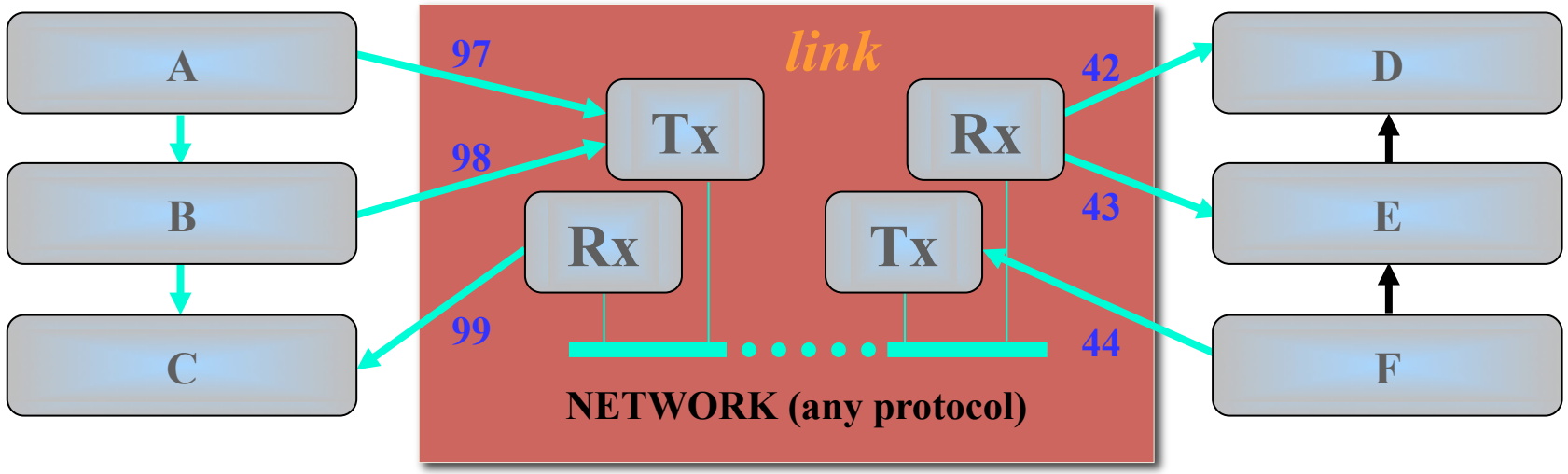


JCSP Crossbar



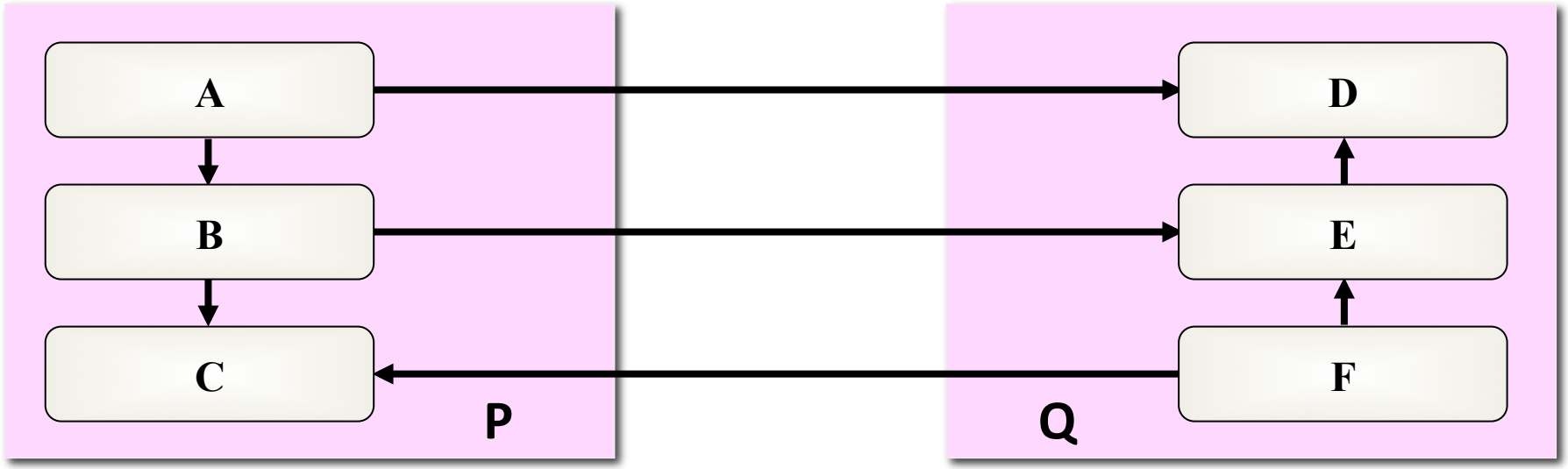
JCSP Links

- 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.**



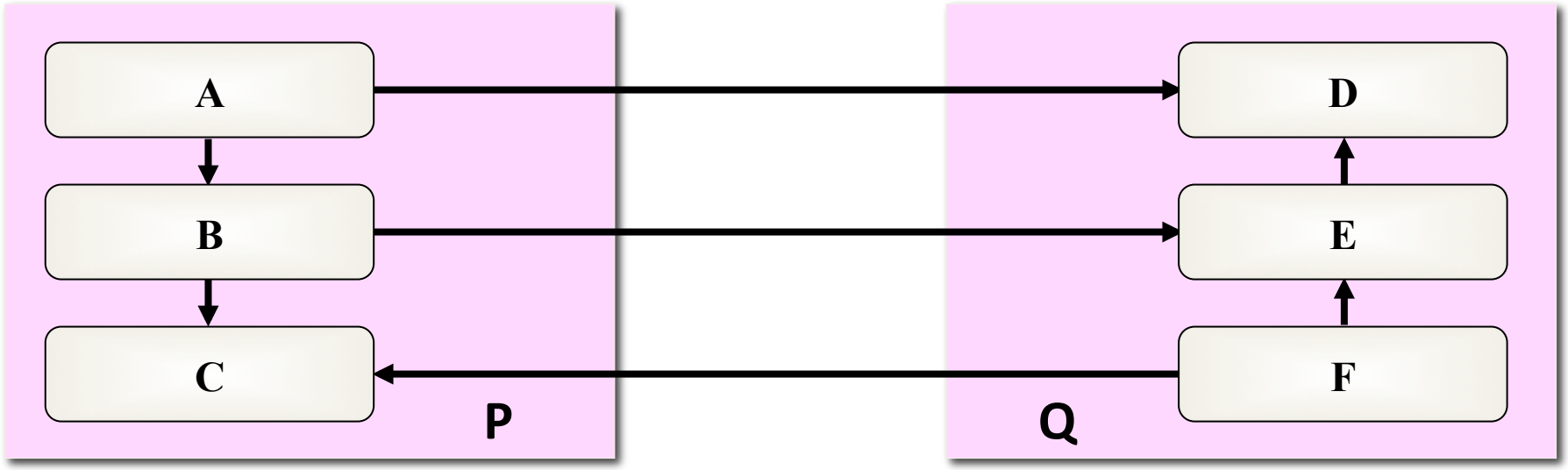
JCSP Networks

- **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

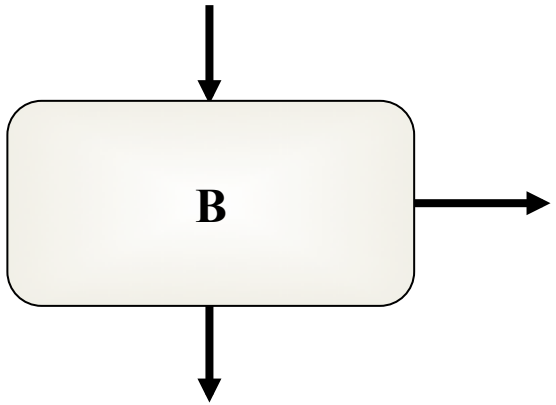
- However, there is one important semantic difference between a network channel and a local channel.
- Over *local* channels, objects are passed by *reference* (which leads to *race hazards* if careless).
- Over *network* channels, objects are passed by *copying* (currently, using Java *serialization*).



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.

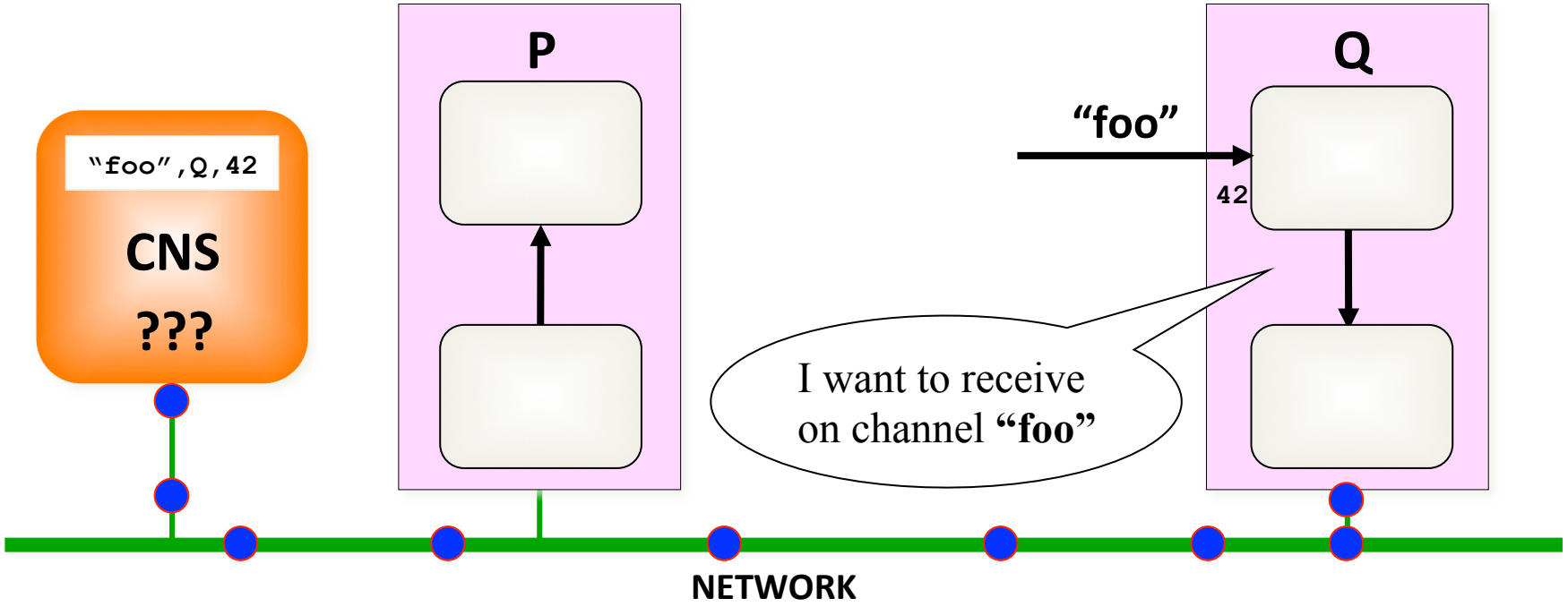
Process **B** still sees its external channels as **`ChannelInput`** / **`ChannelOutput`**



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

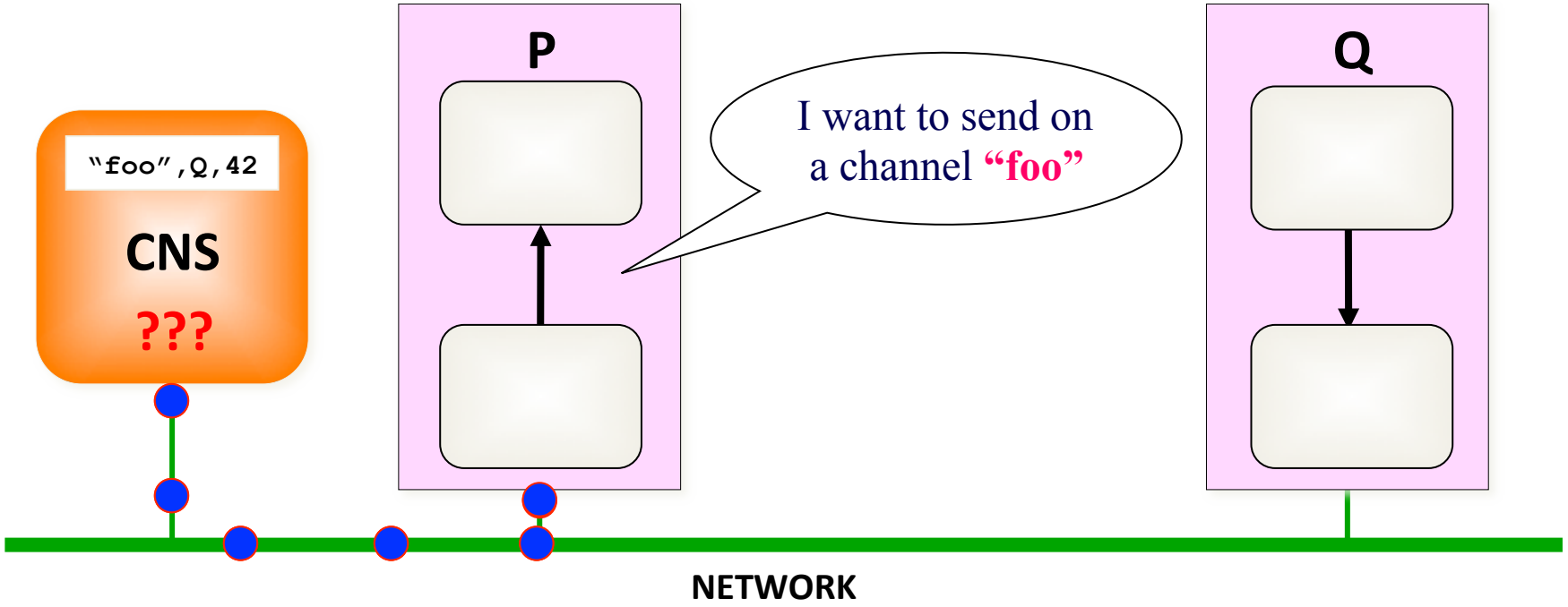
Establishing Network Channels

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



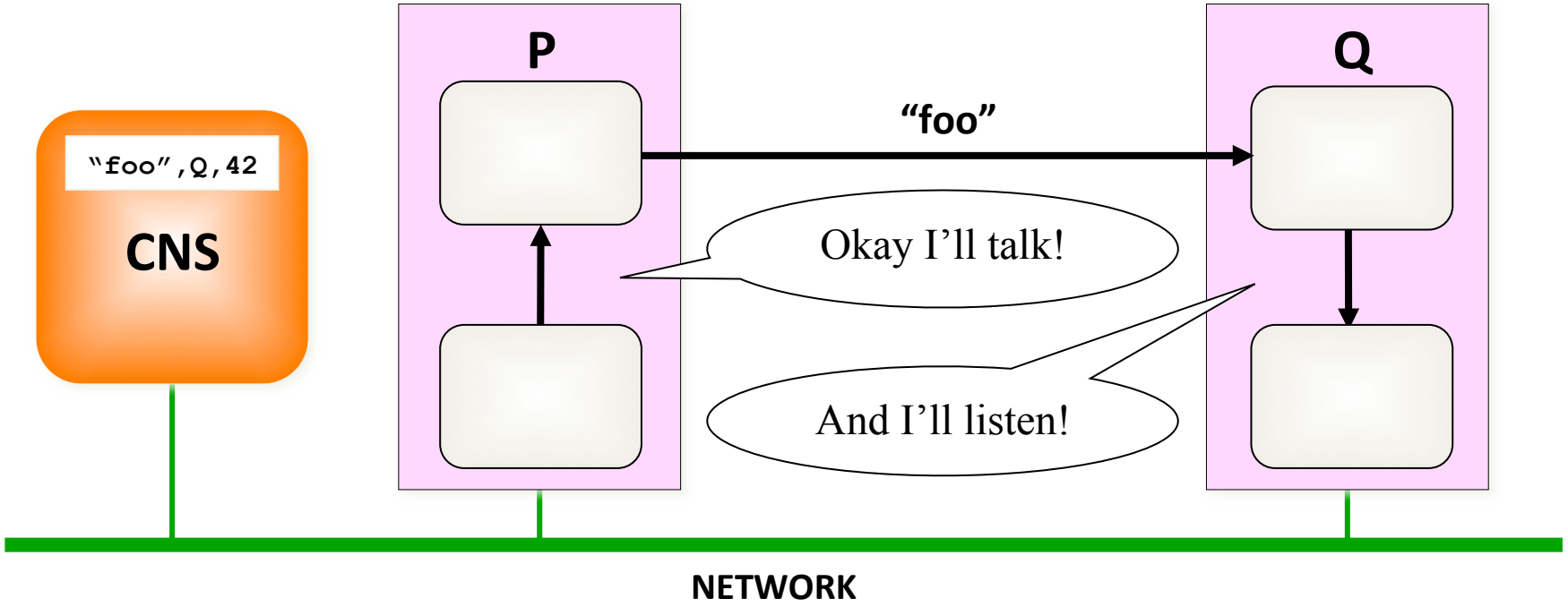
Establishing Network Channels

- 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.



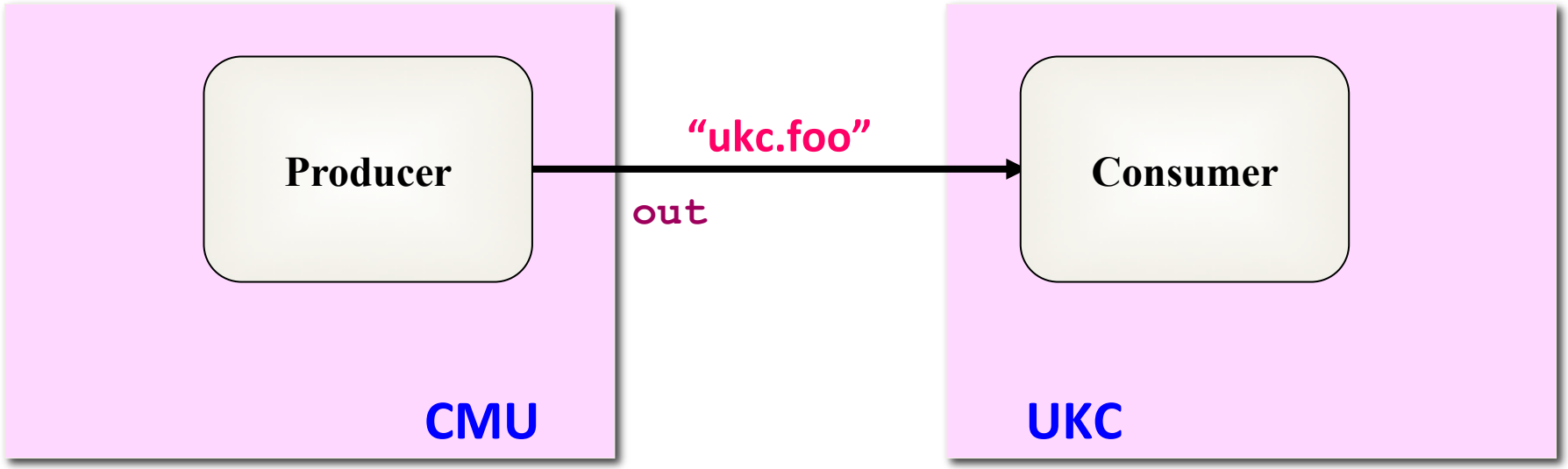
Using Distributed JCSP

- 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);
```



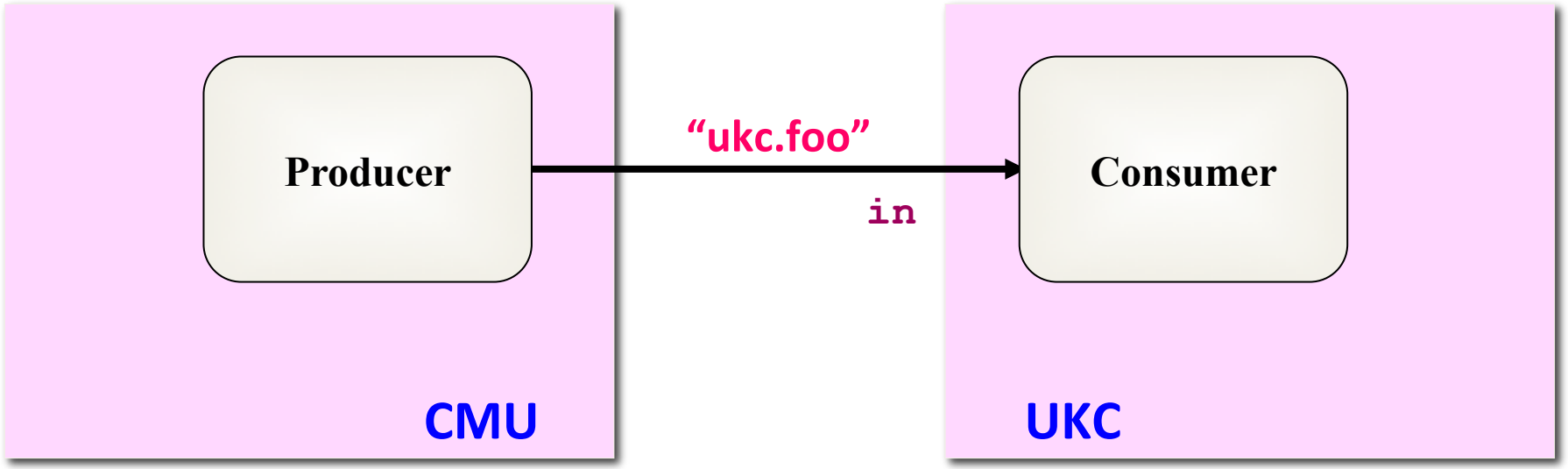
Using Distributed JCSP

- 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");
```

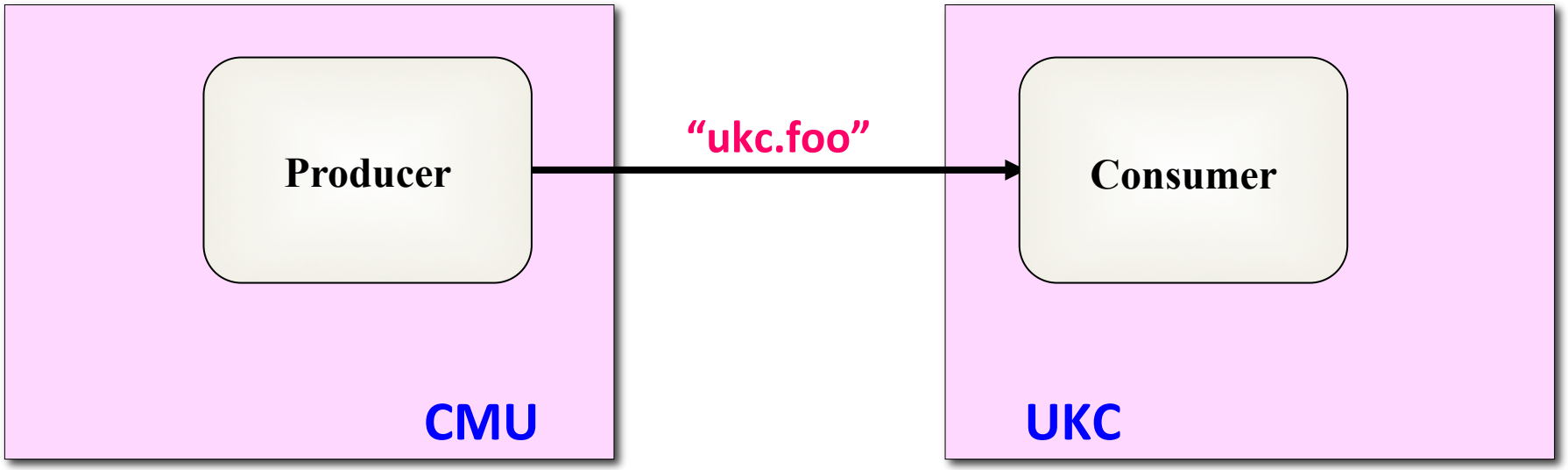
Named network *output* channel construction blocks until the name is registered by a reader

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

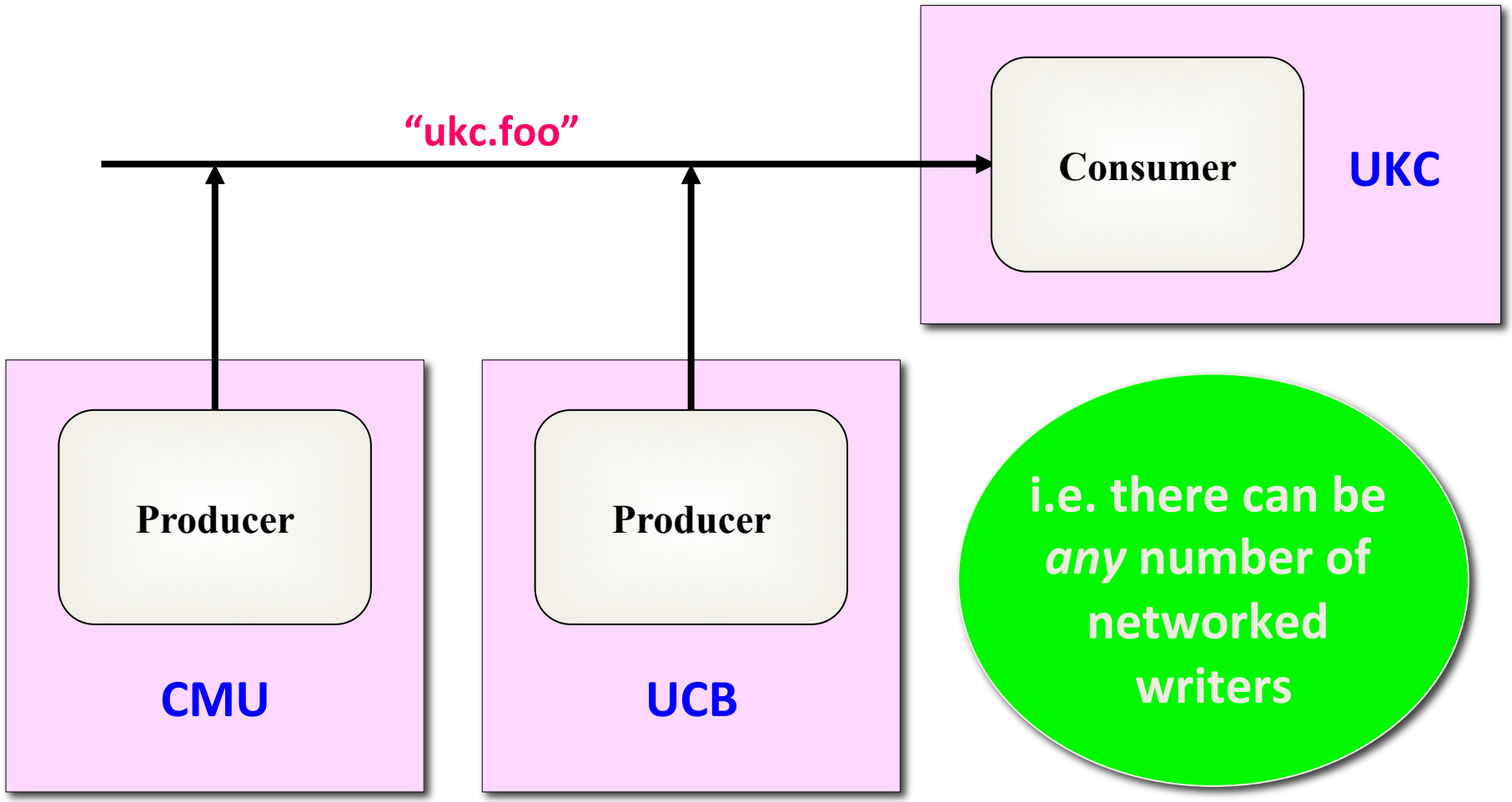
Named network *input* channel construction registers the name with the CNS (will fail if already registered)

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*).

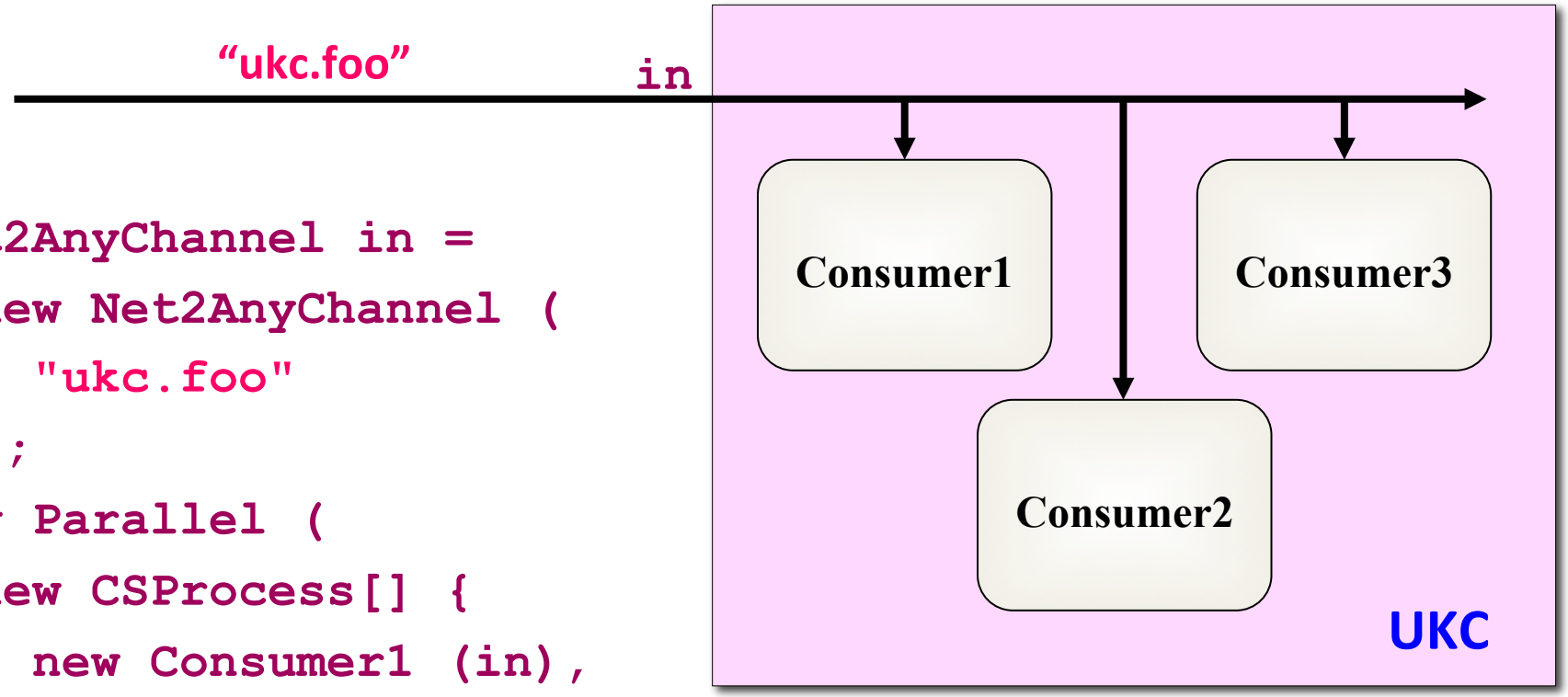


Network Channels are *Any-1*



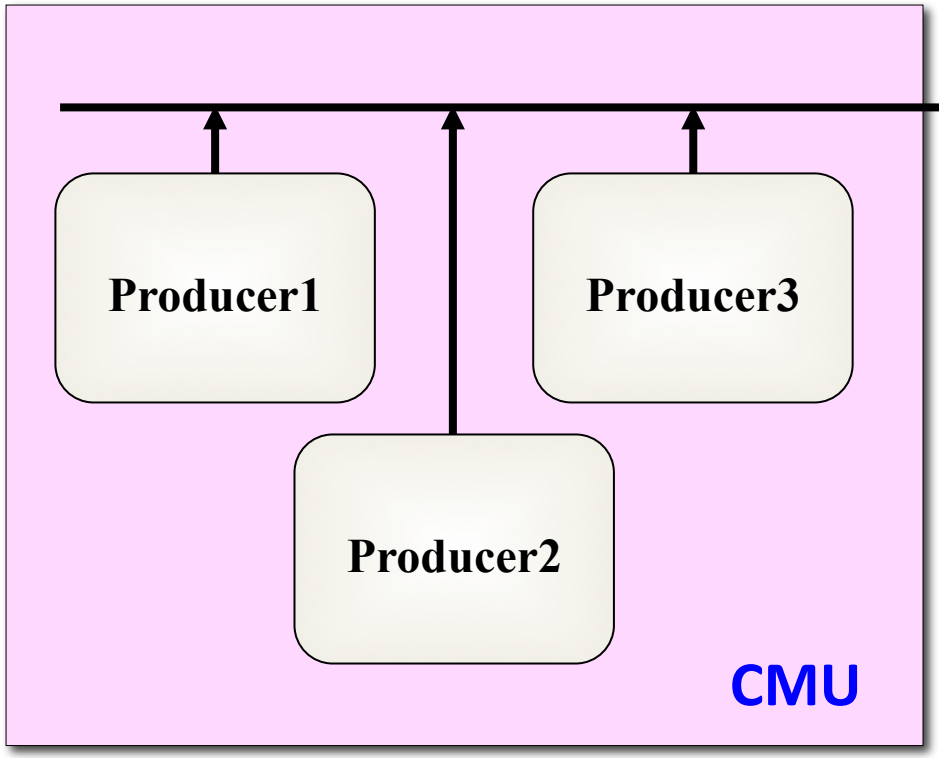
Net-Any Channels

```
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:

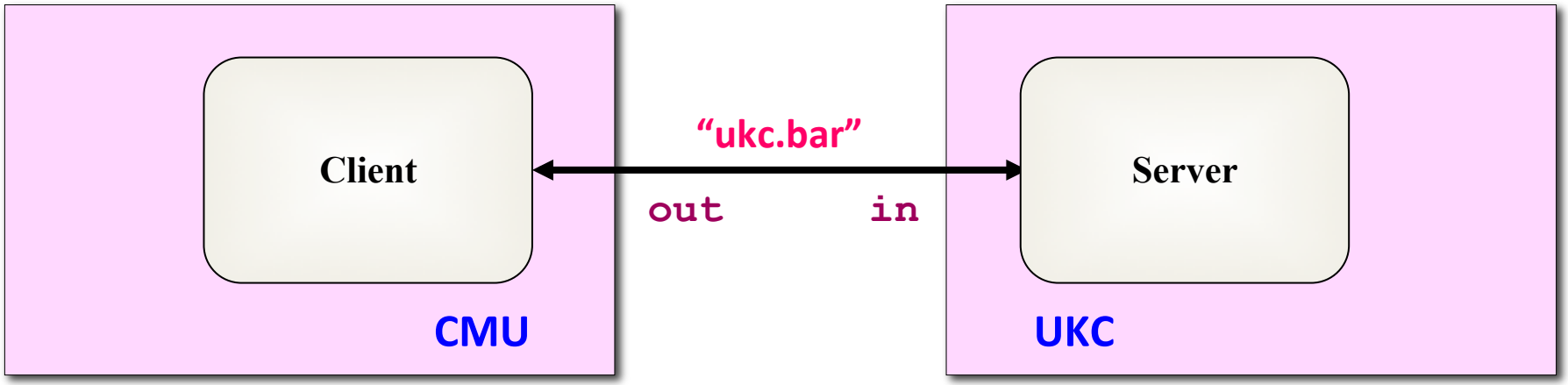
find

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

- On the **UKC** machine:

register

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



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 …*

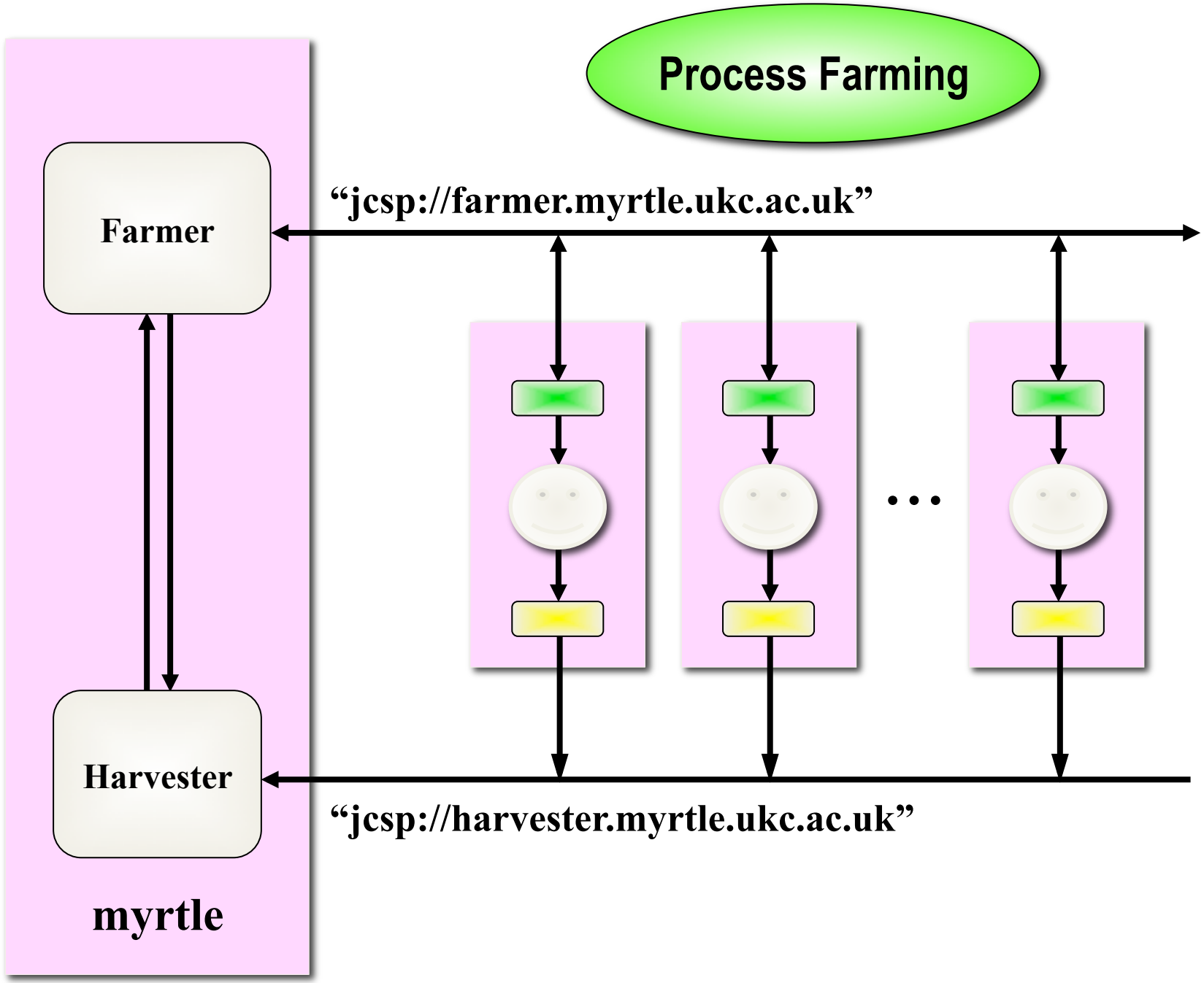
Anonymous Channels

- 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 ();
```

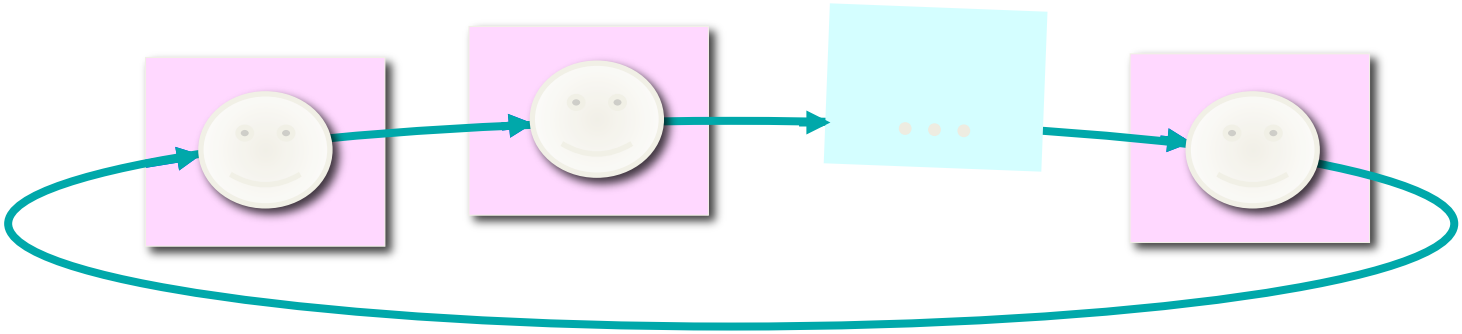
- The information can be distributed using existing (network) channels to those you trust:

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


Process Farming
Farmer
"jcsp://farmer.myrtle.ukc.ac.uk"
Harvester
"jcsp://harvester.myrtle.ukc.ac.uk"
myrtle

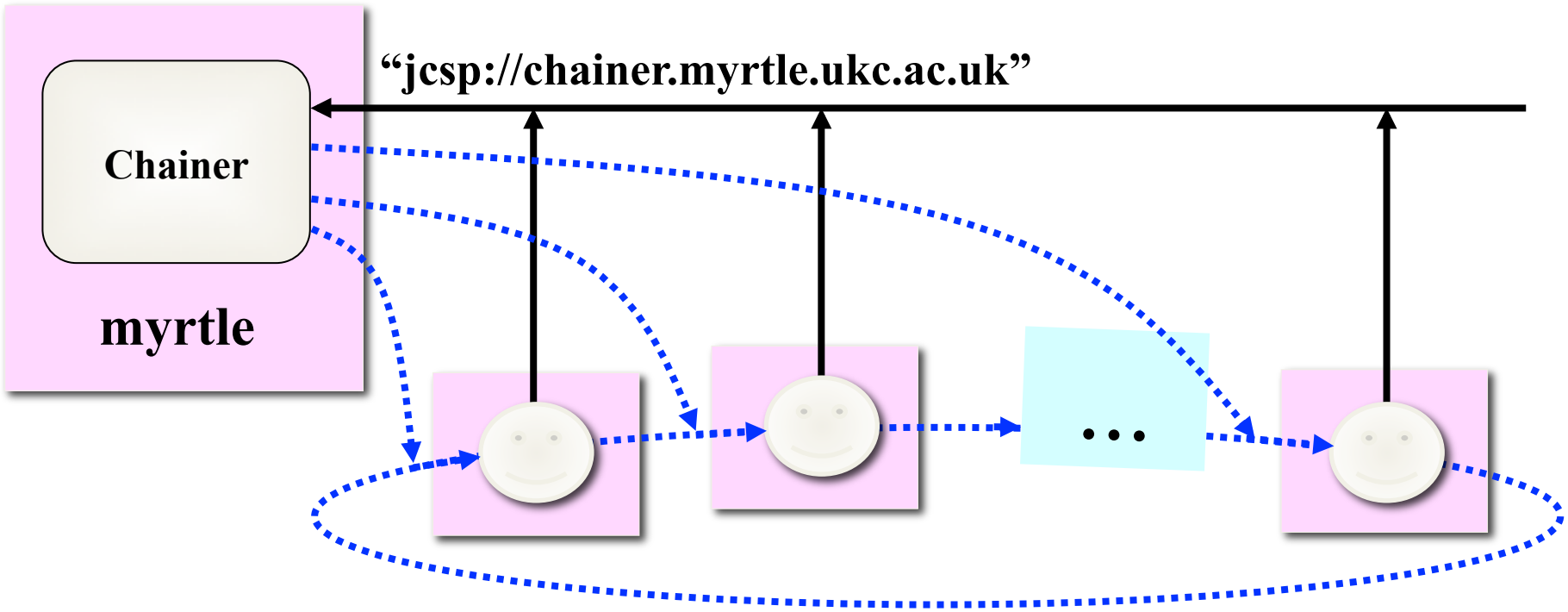
Process Chaining

```
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 One2NetChannel ("node-" + NEXT_ID);

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



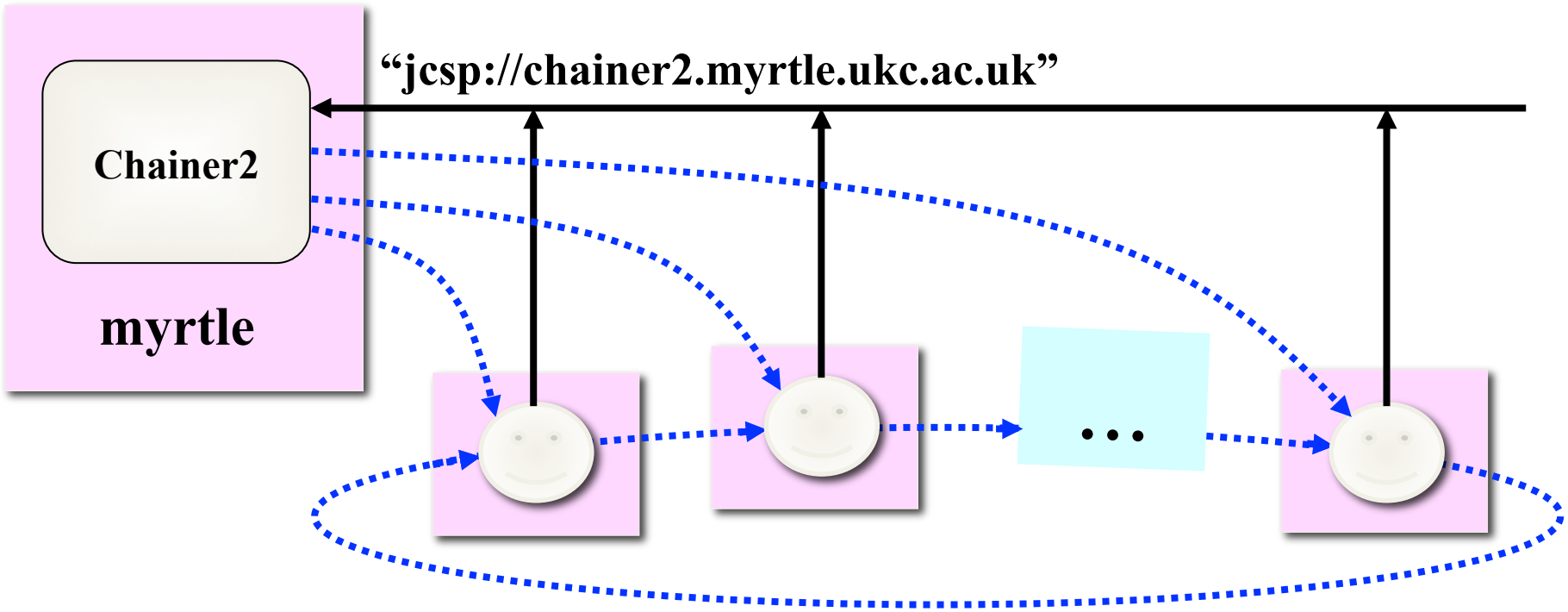
OK – so long as each worker knows the length of the chain and its place in it.

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 …

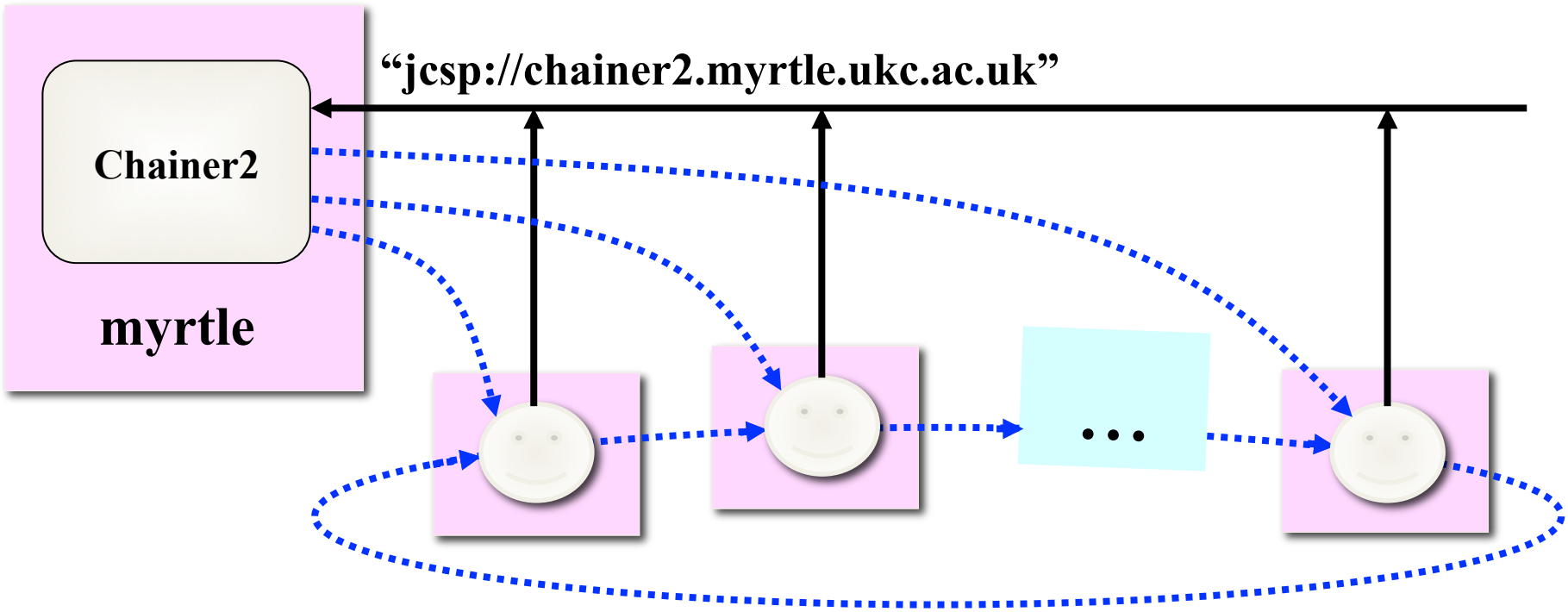
Ring worker code

```
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 …

PyCSP

- Master
  - sys.path.insert(0, "..")
  - Configuration().set(PYCSP_PORT, <portnr>)
- Others
  - addr = (<host>, <portnr>)
- @Multiprocess(<host>,<portnr>)

Example Applications

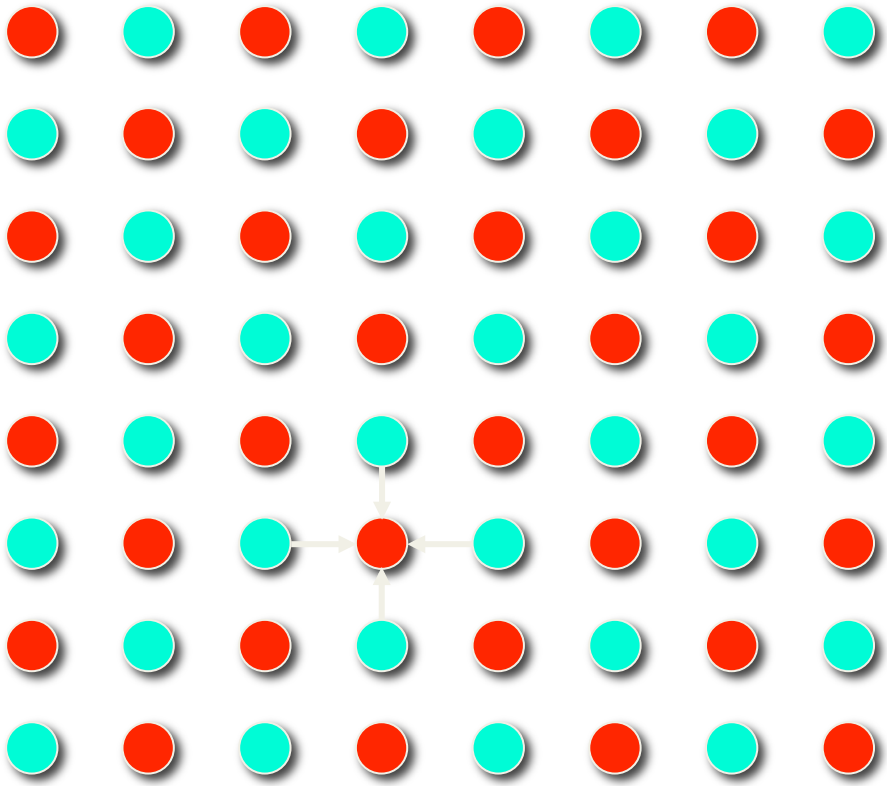
Process Farming

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)

SOR – red/black checker pointing

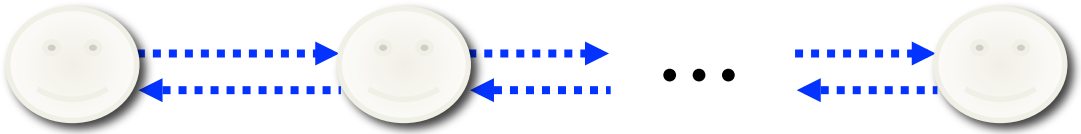


where = black

and = red

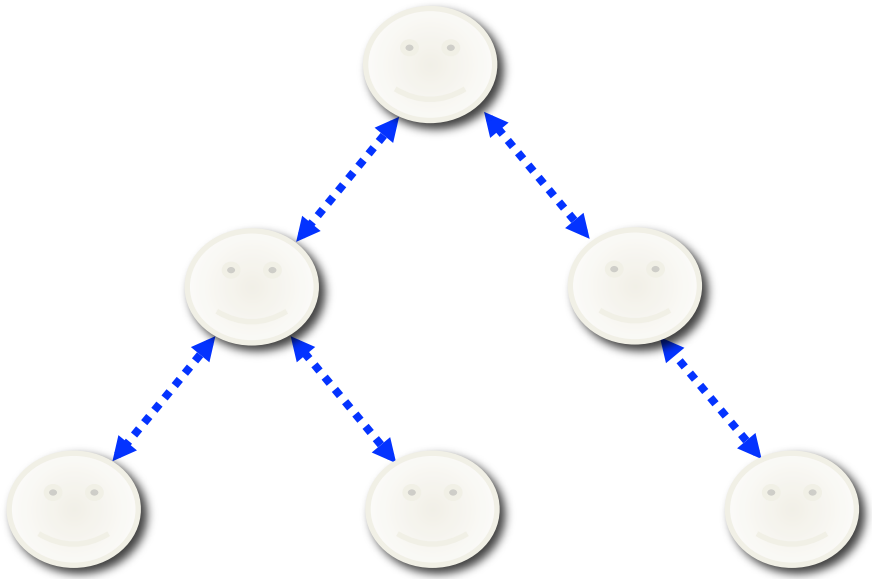
SOR – red/black checker pointing

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



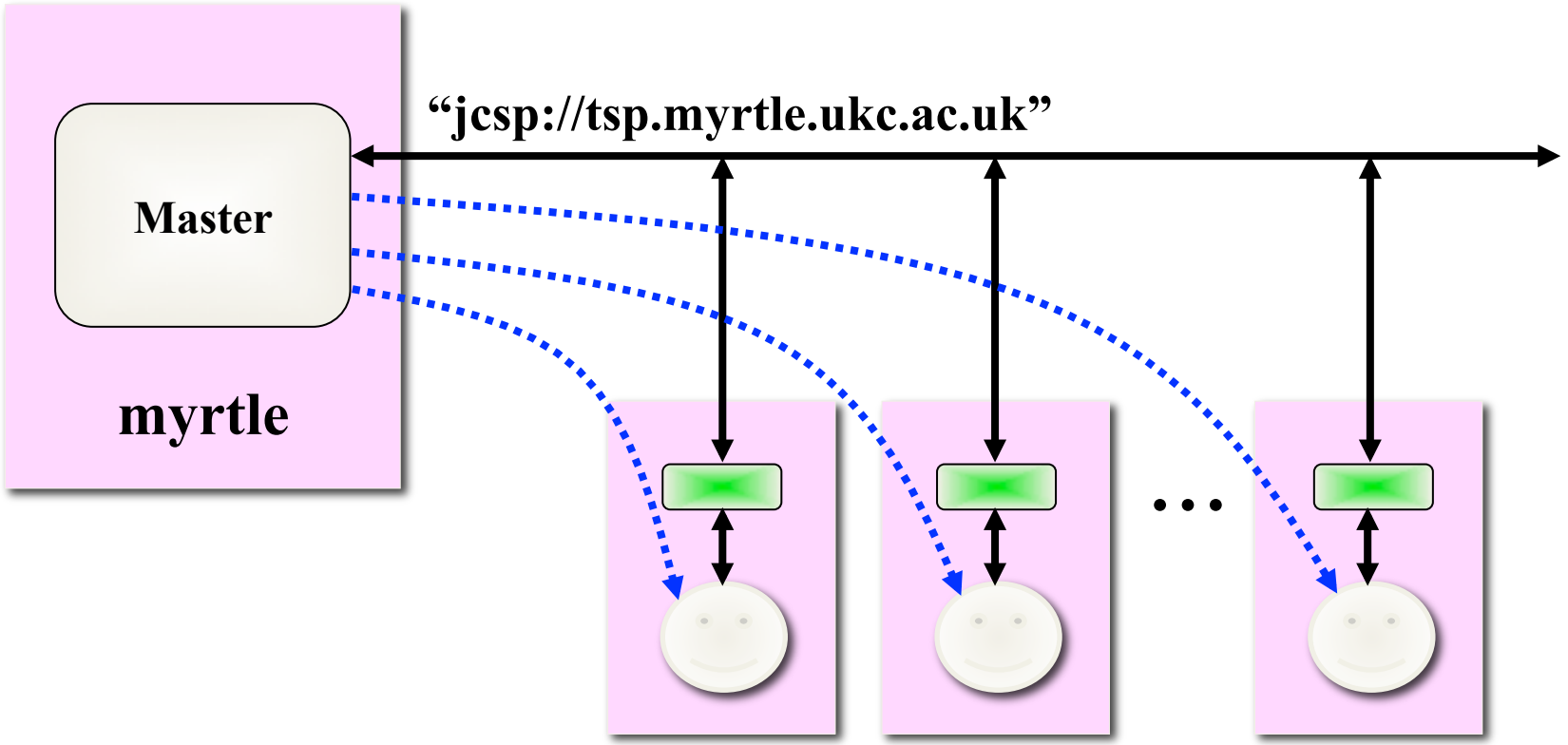
SOR – red/black checker pointing

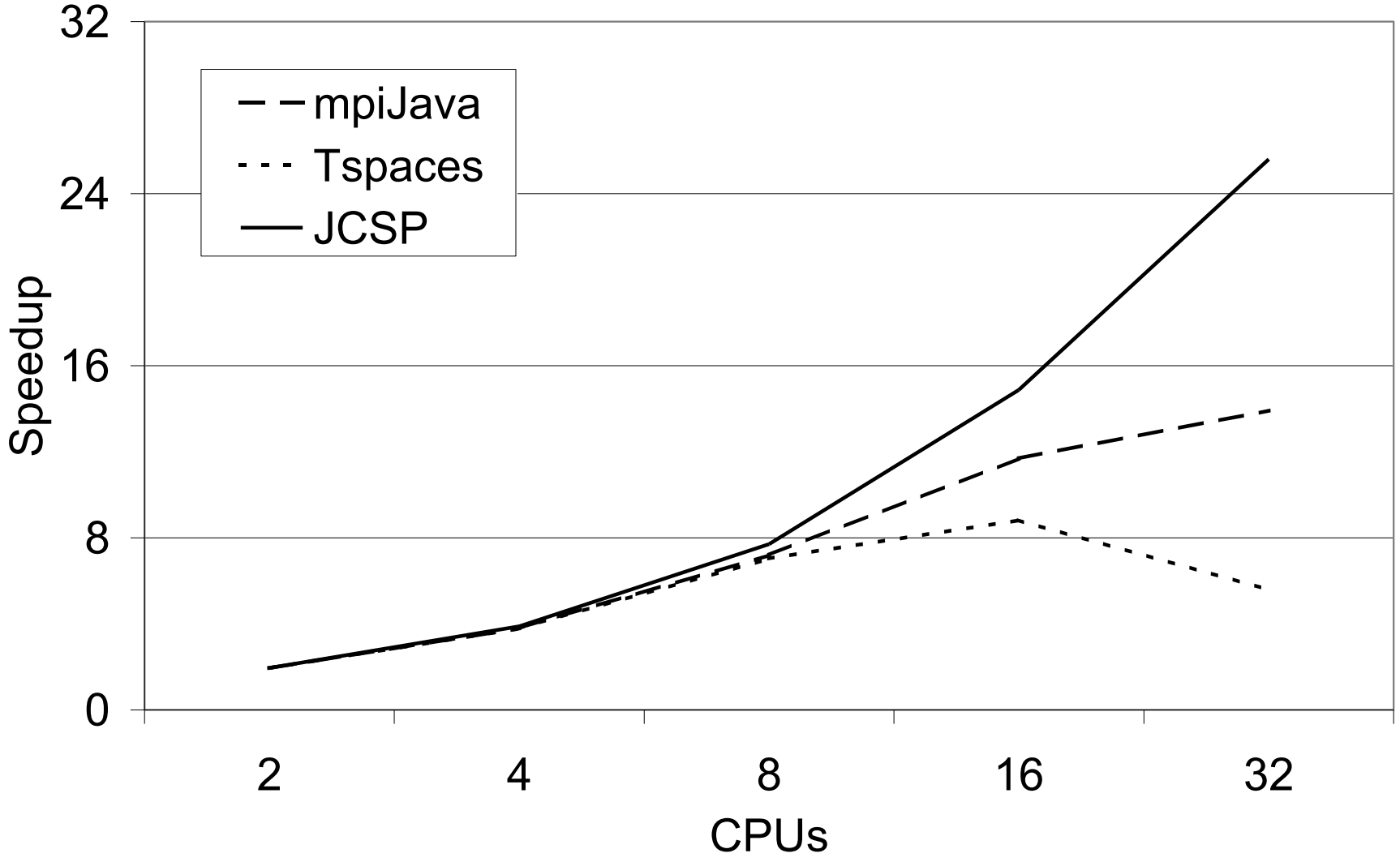
Also, a global *sum-of-changes* (found by each node) has to be computed each cycle to resolve halting criteria. This is speeded-up by connecting the nodes into a tree (so that *adds* and *communications* can take place in parallel).



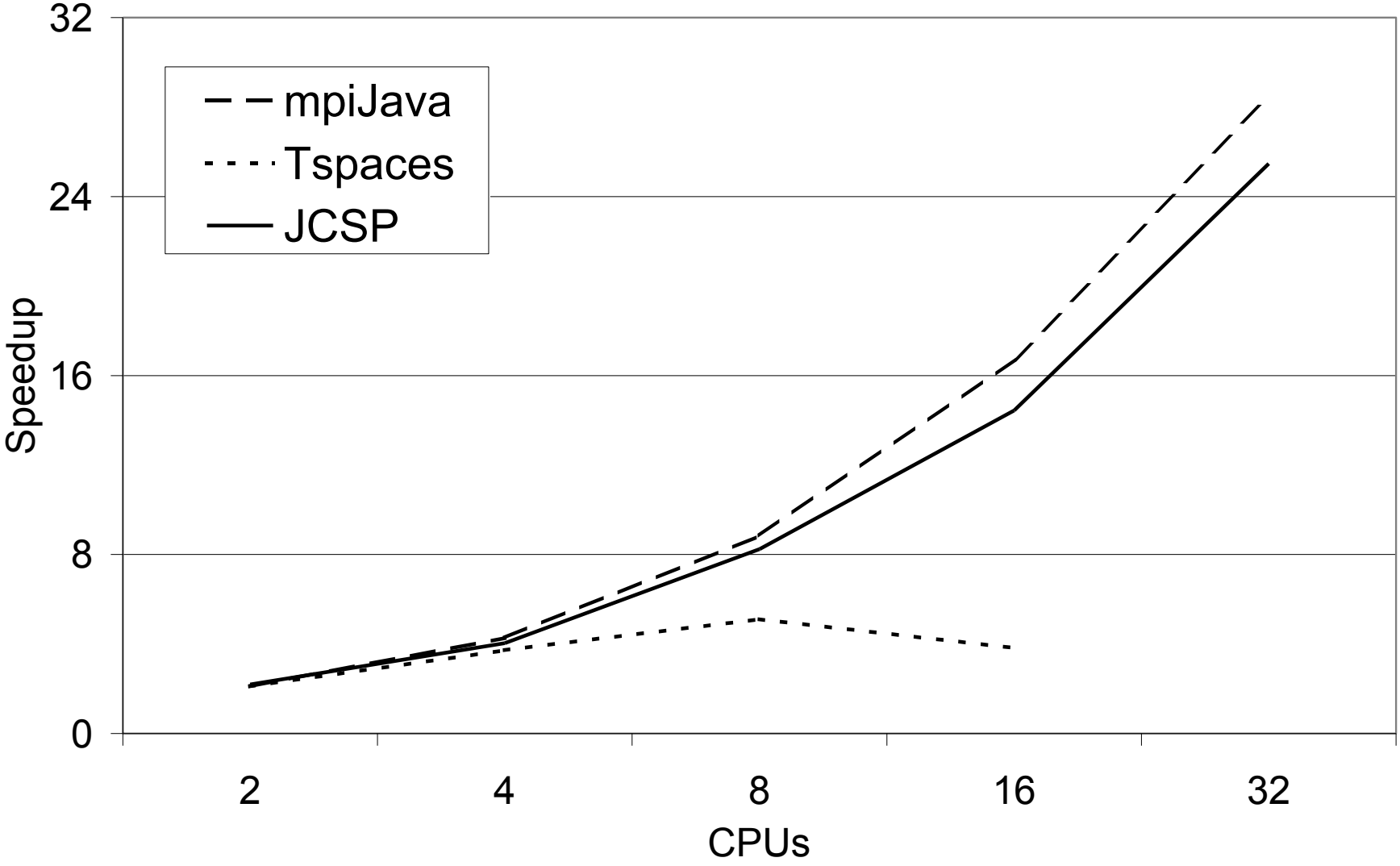
Travelling Salesman Problem

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

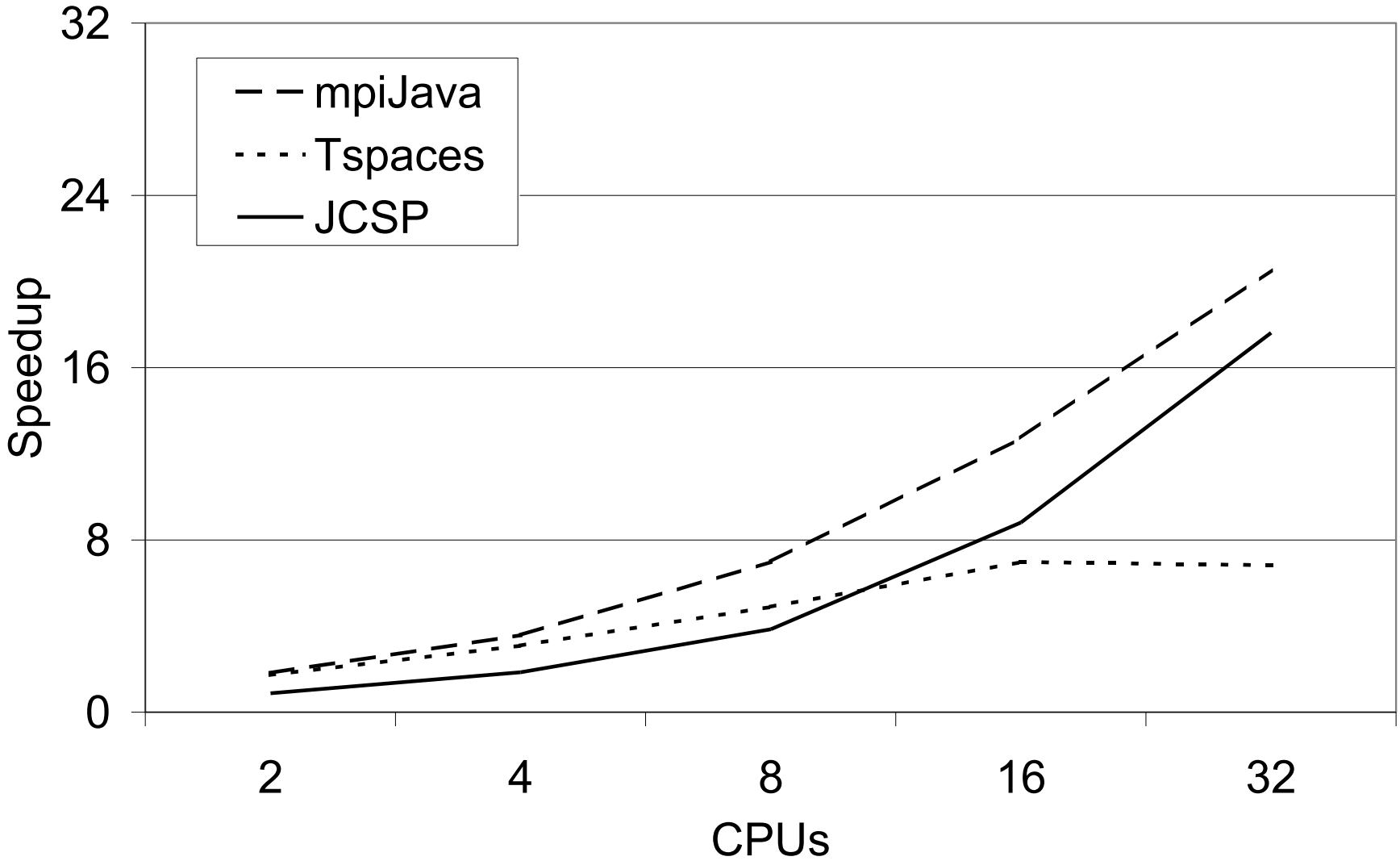




The n-Body benchmark (n = 10000)

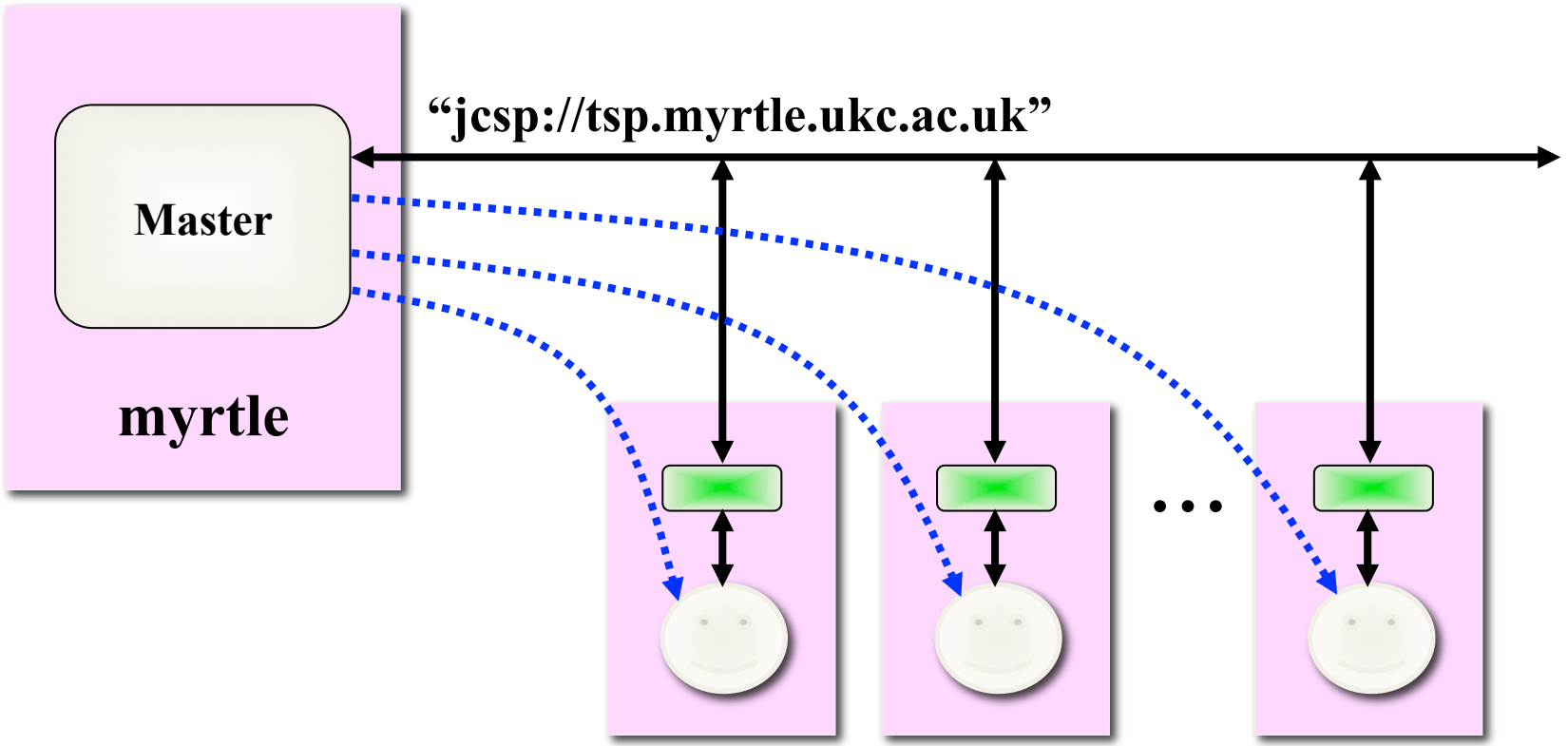


The SOR benchmark (7000 x 7000 matrix)



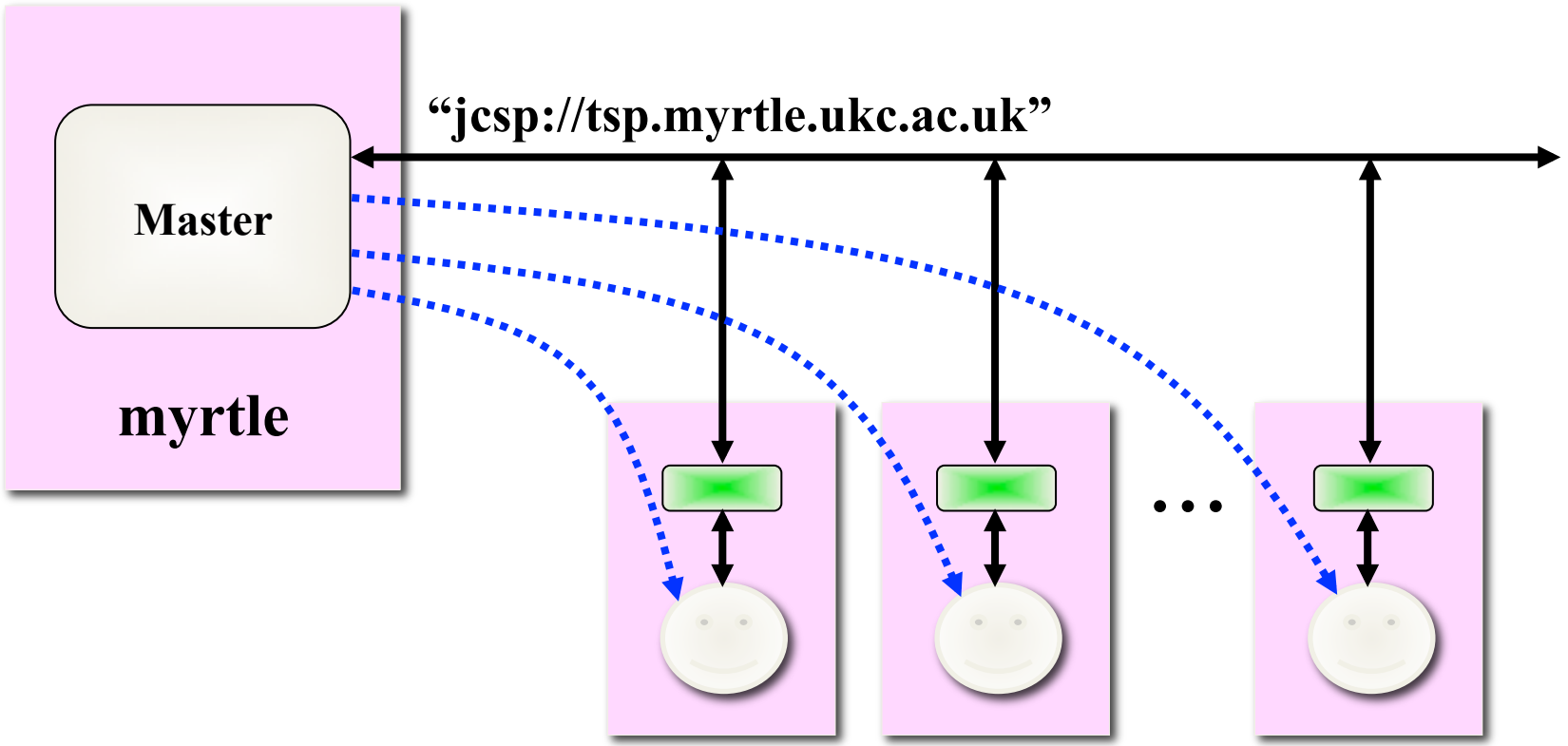
The Travelling Salesman Problem (15 cities)

Travelling Salesman Problem



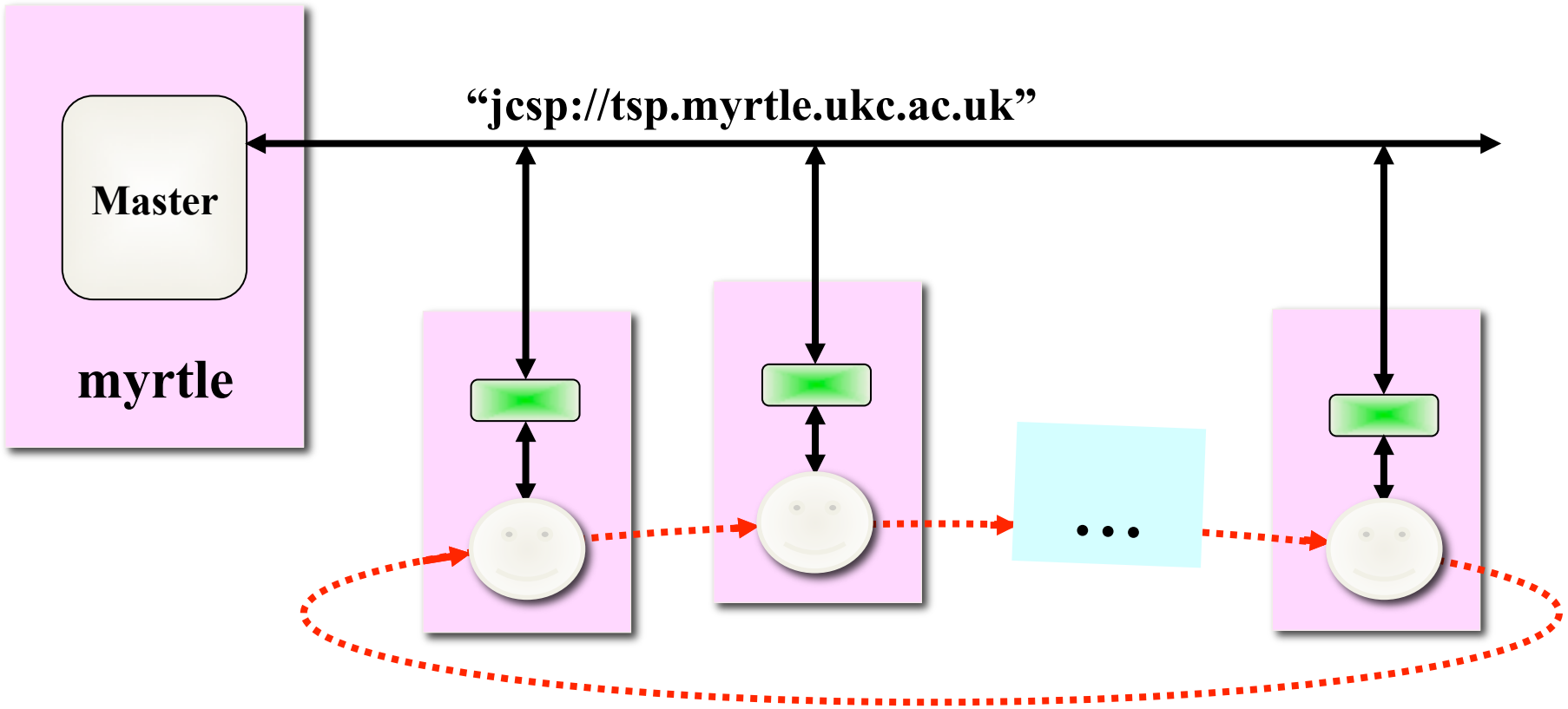
Currently, workers report newly discovered shorter paths back to the master (who maintains the global shortest). If master receives a better one, it broadcasts back to workers.

Travelling Salesman Problem



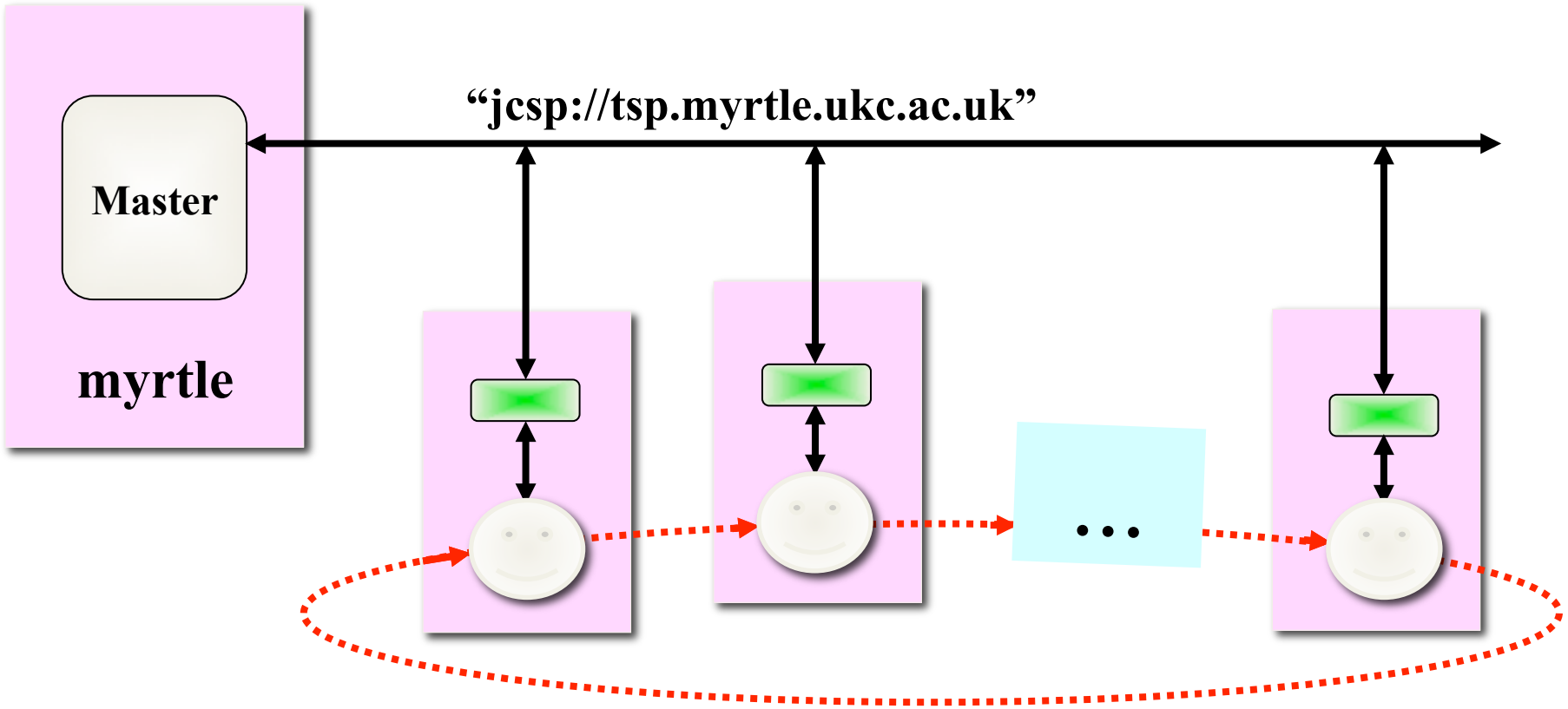
Massive swamping of network links in the early stages.
Also, generation of garbage *currently* provokes garbage collector – and clobbers cache on dual-processor nodes.

Travelling Salesman Problem



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

Travelling Salesman Problem

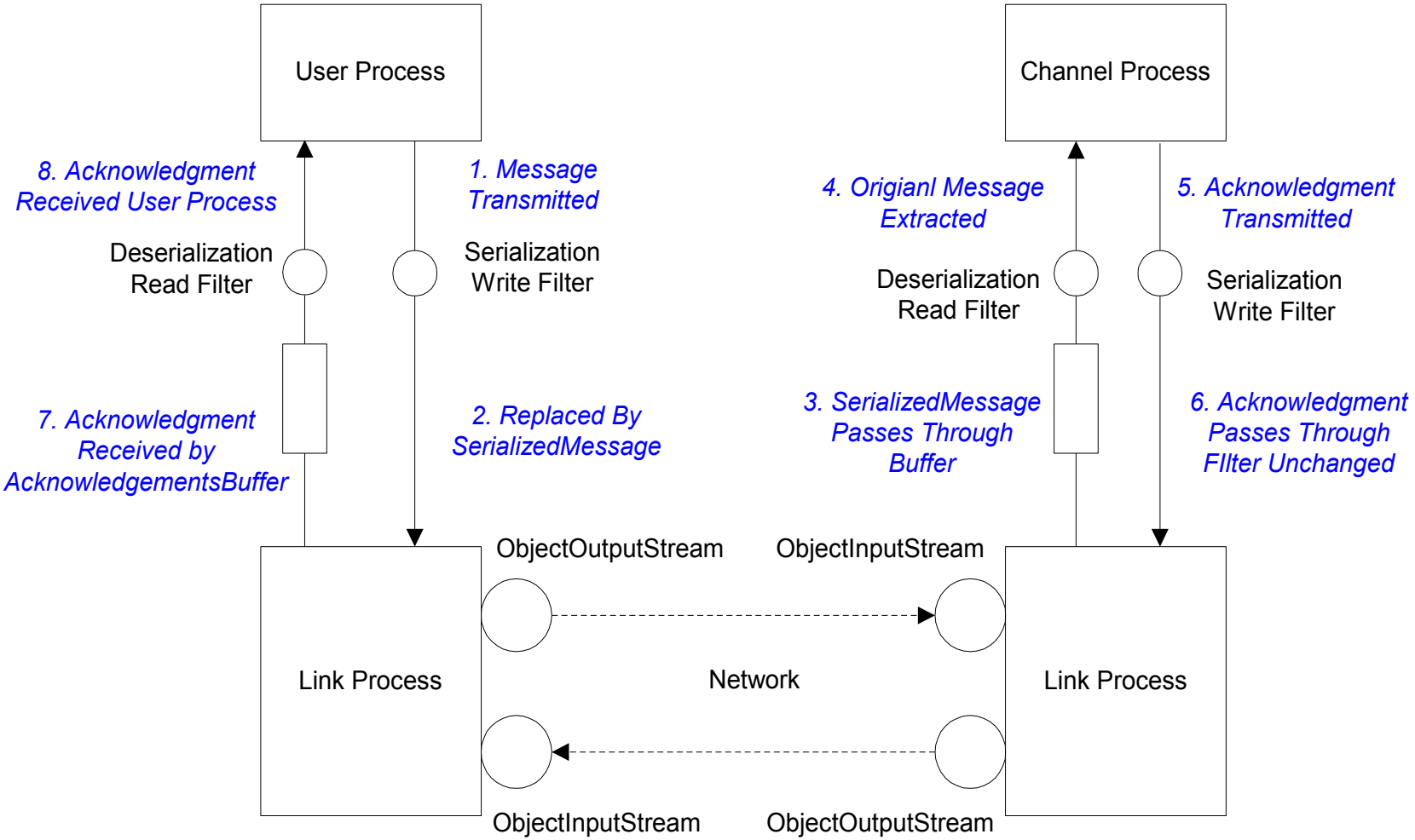


Global minimum maintained in ring (made with one-place overwriting channel buffers) … ***easy!!!***

Networked Class Loading

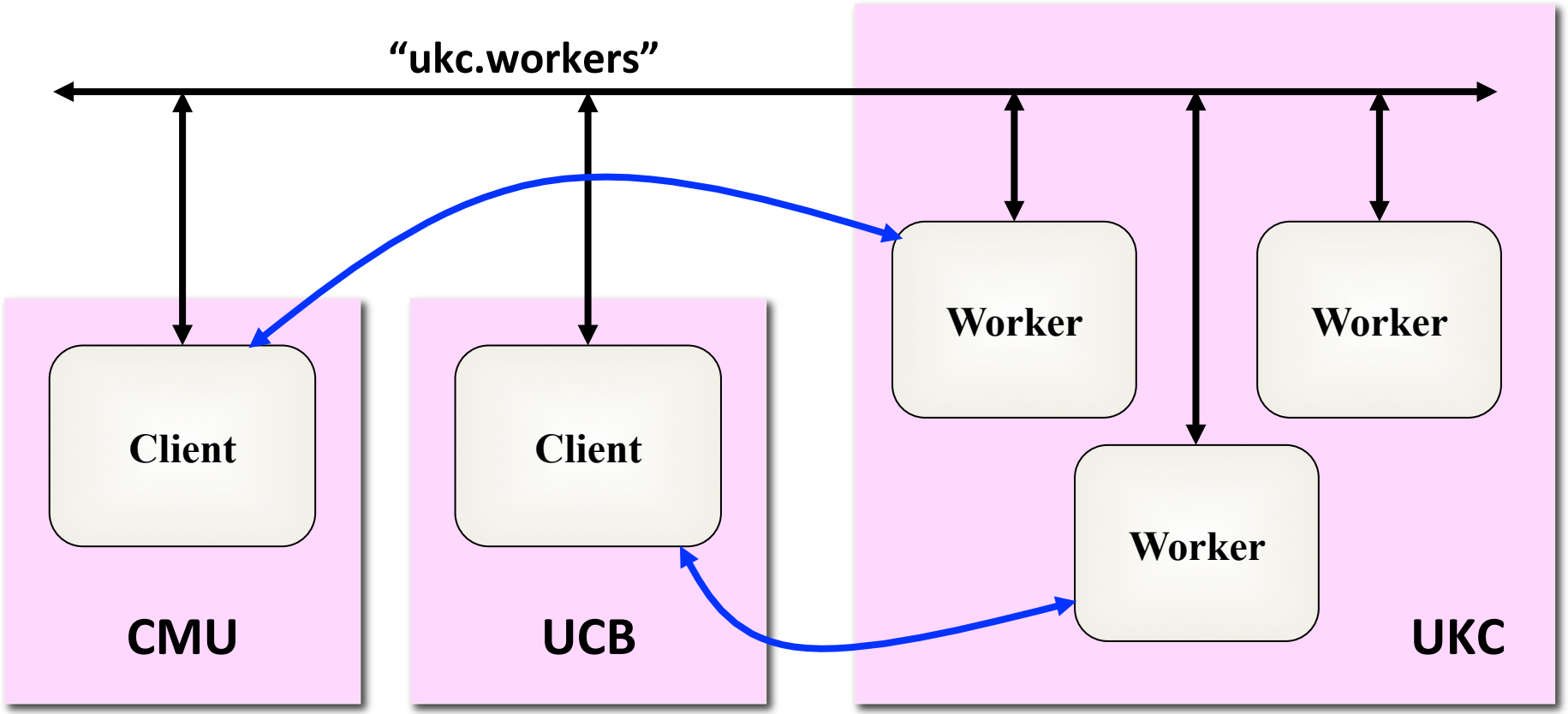
- By default, objects sent across a networked channel (or connection) use Java *serialization*.
- This means the receiving JVM is expected to be able to load (or already have loaded) the class files needed for its received objects.
- However, JCSP networked channels/connections can be set to communicate 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



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 ();
```

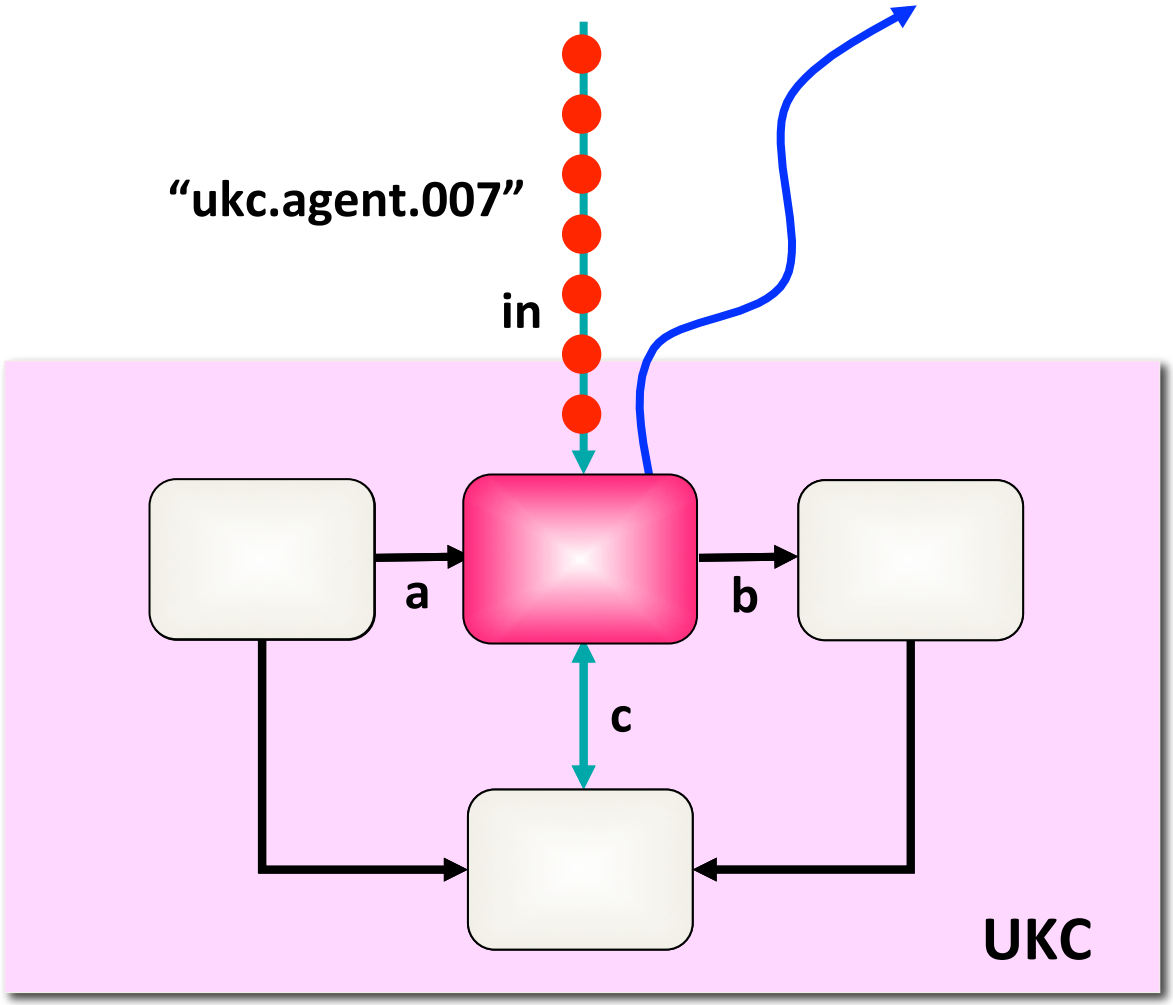
client code

worker code

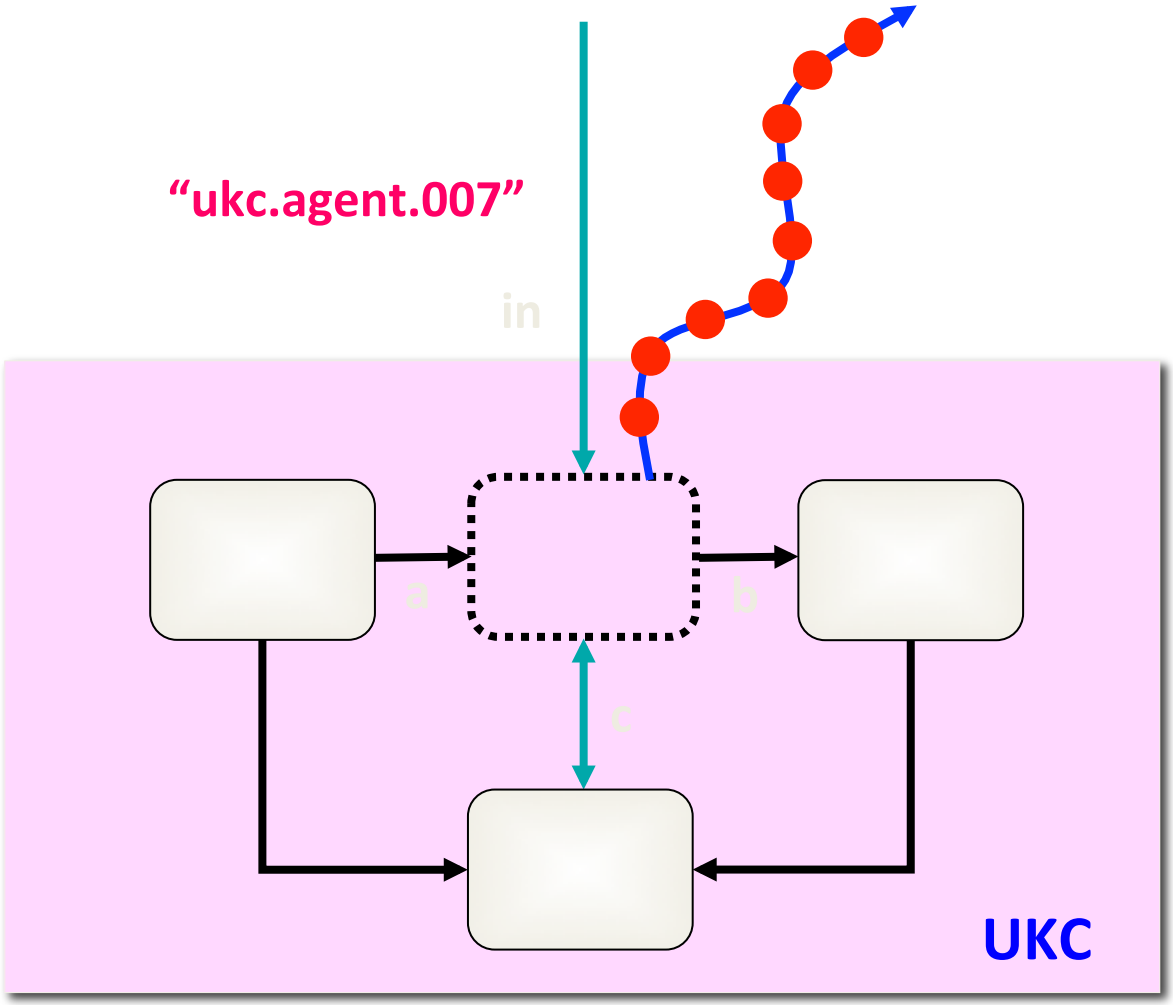
```
CSProcess work = (CSProcess) in.request ();
work.run ();
in.reply (work);
```

Work class file automatically downloaded

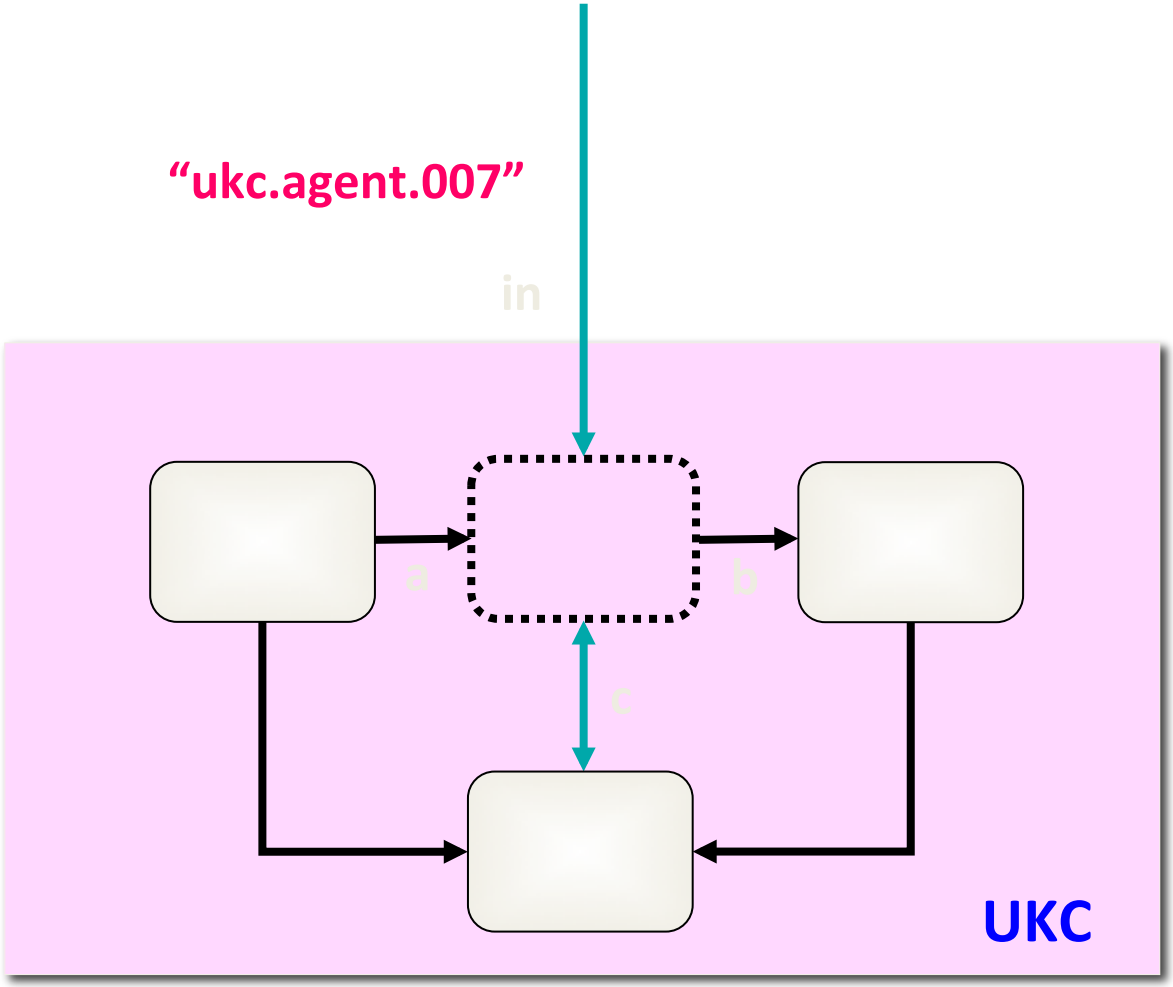
Mobile Processes (Agents)



Mobile Processes (Agents)

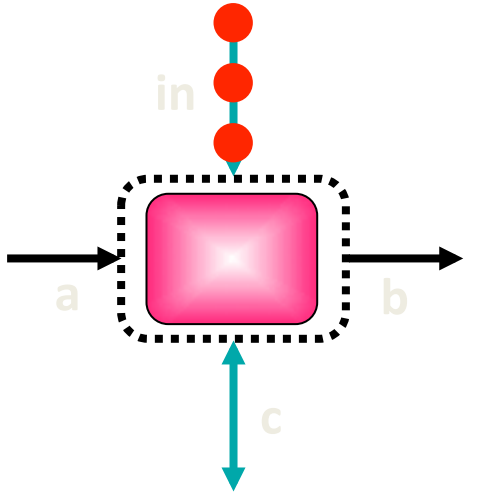


Mobile Processes (Agents)



Mobile Processes (Agents)

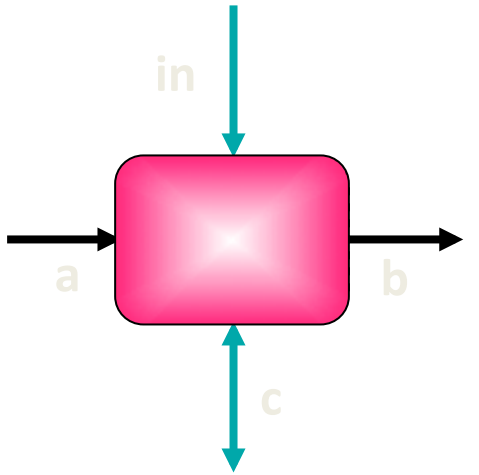
```
while (running) {
  Bond james = (Bond) in.read ();
  james.plugin (a, b, c);
  james.run ();
  NetChannelLocation escapeRoute =
    james.getNextLocation ();
  One2NetChannel escape =
    new One2NetChannel (escapeRoute);
  running = james.getNuke ();
  escape.write (james);
  escape.disconnect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



local 007 controller

Mobile Processes (Agents)

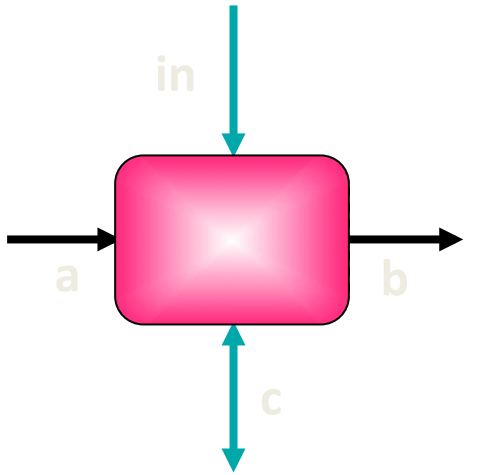
```
while (running) {
  Bond james = (Bond) in.read ();
  james.plugin (a, b, c);
  james.run ();
  NetChannelLocation escapeRoute =
    james.getNextLocation ();
  One2NetChannel escape =
    new One2NetChannel (escapeRoute);
  running = james.getNuke ();
  escape.write (james);
  escape.disconnect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



local 007 controller

Mobile Processes (Agents)

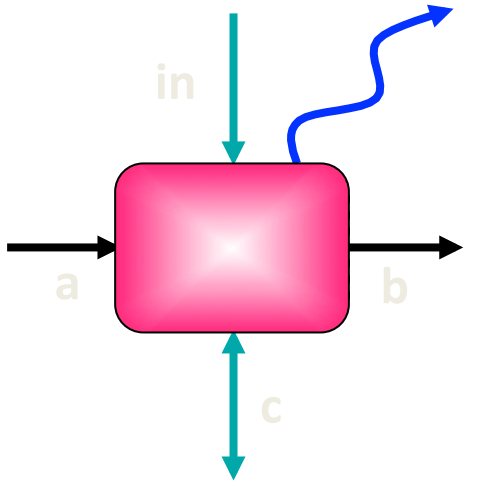
```
while (running) {
  Bond james = (Bond) in.read ();
  james.plugin (a, b, c);
  james.run ();
  NetChannelLocation escapeRoute =
    james.getNextLocation ();
  One2NetChannel escape =
    new One2NetChannel (escapeRoute);
  running = james.getNuke ();
  escape.write (james);
  escape.disconnect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



local 007 controller

Mobile Processes (Agents)

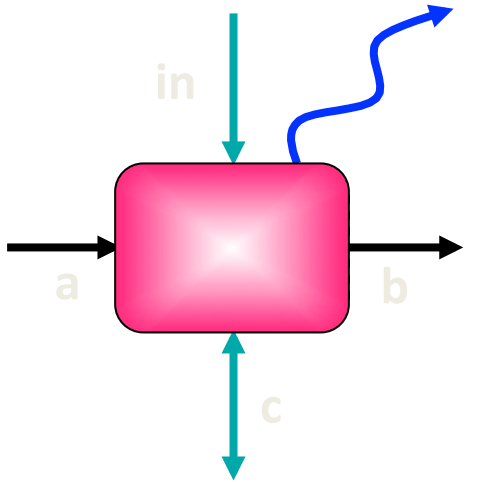
```
while (running) {
  Bond james = (Bond) in.read ();
  james.plugin (a, b, c);
  james.run ();
  NetChannelLocation escapeRoute =
    james.getNextLocation ();
  One2NetChannel escape =
    new One2NetChannel (escapeRoute);
  running = james.getNuke ();
  escape.write (james);
  escape.disconnect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



local 007 controller

Mobile Processes (Agents)

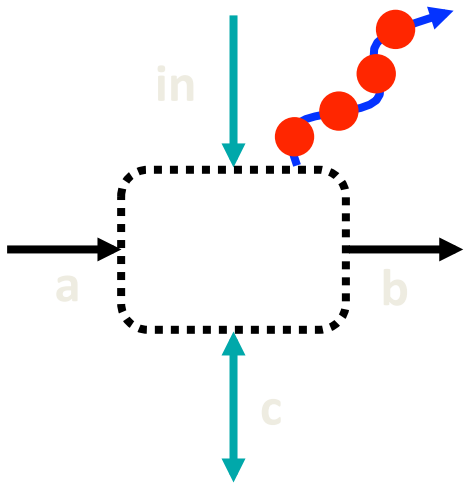
```
while (running) {
  Bond james = (Bond) in.read ();
  james.plugin (a, b, c);
  james.run ();
  NetChannelLocation escapeRoute =
    james.getNextLocation ();
  One2NetChannel escape =
    new One2NetChannel (escapeRoute);
  running = james.getNuke ();
  escape.write (james);
  escape.disconnect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



local 007 controller

Mobile Processes (Agents)

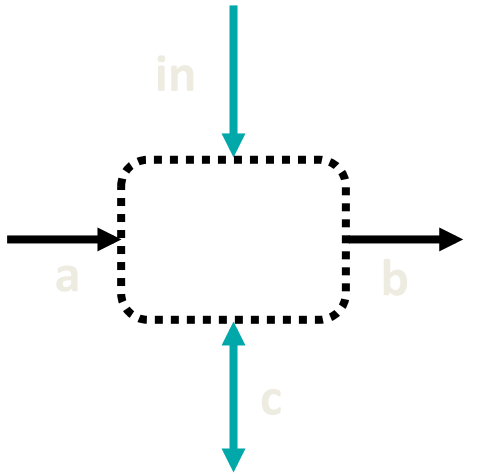
```
while (running) {
  Bond james = (Bond) in.read ();
  james.plugin (a, b, c);
  james.run ();
  NetChannelLocation escapeRoute =
    james.getNextLocation ();
  One2NetChannel escape =
    new One2NetChannel (escapeRoute);
  running = james.getNuke ();
  escape.write (james);
  escape.disconnect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



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 ();
}
```



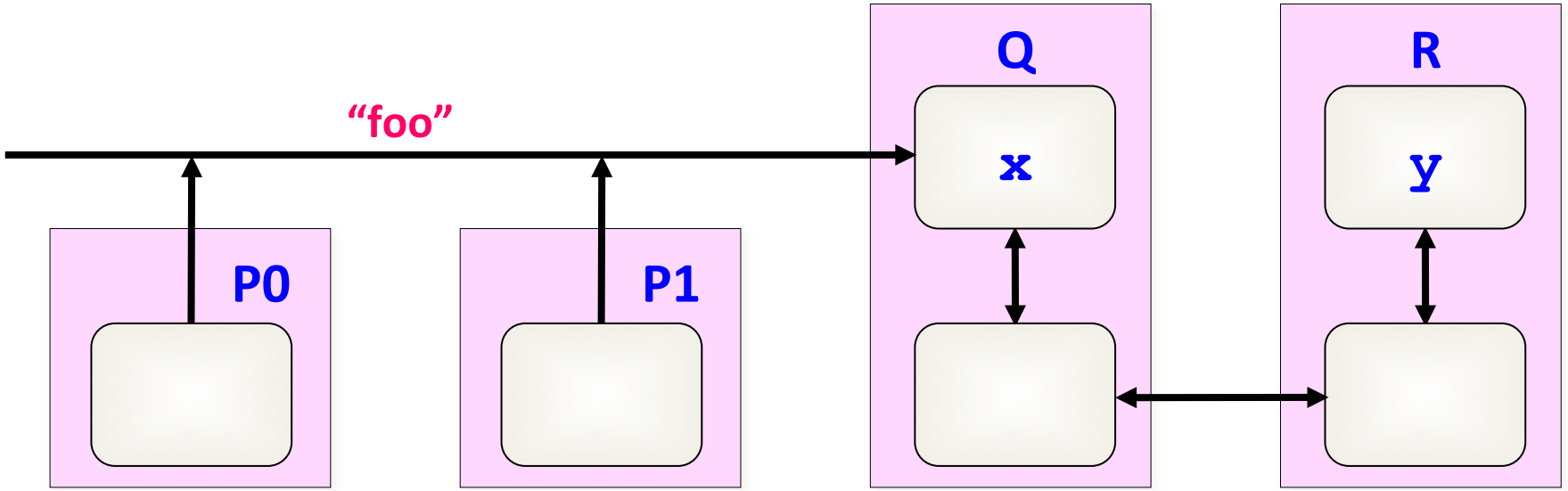
local 007 controller

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 …

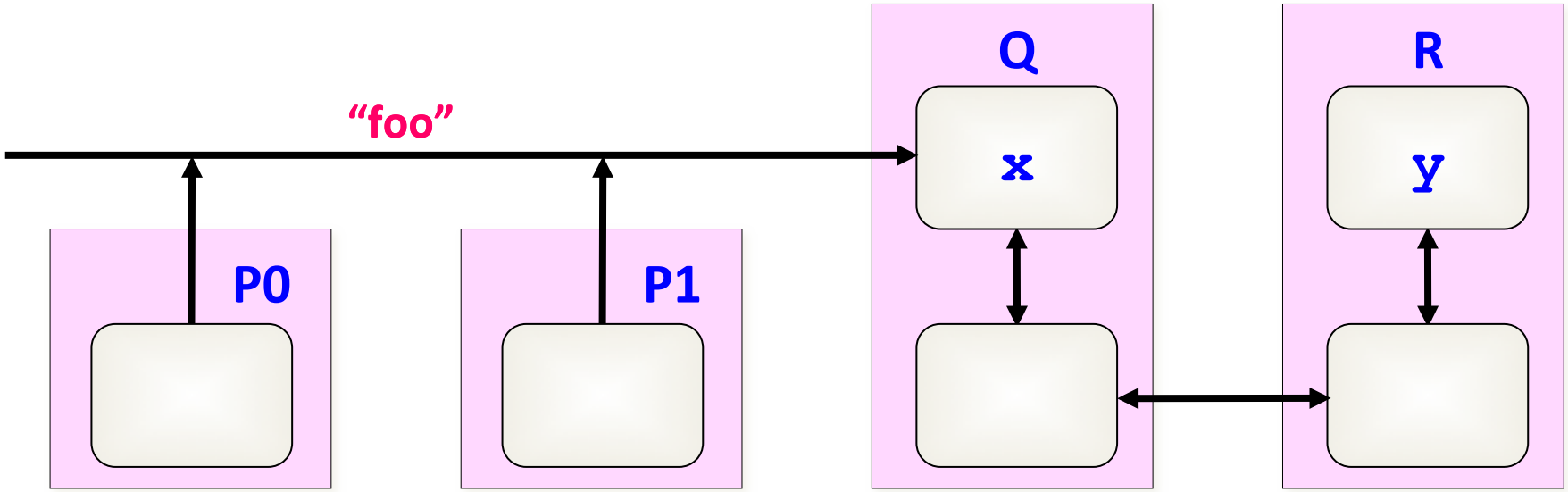
Mobile Network Channels

- 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.



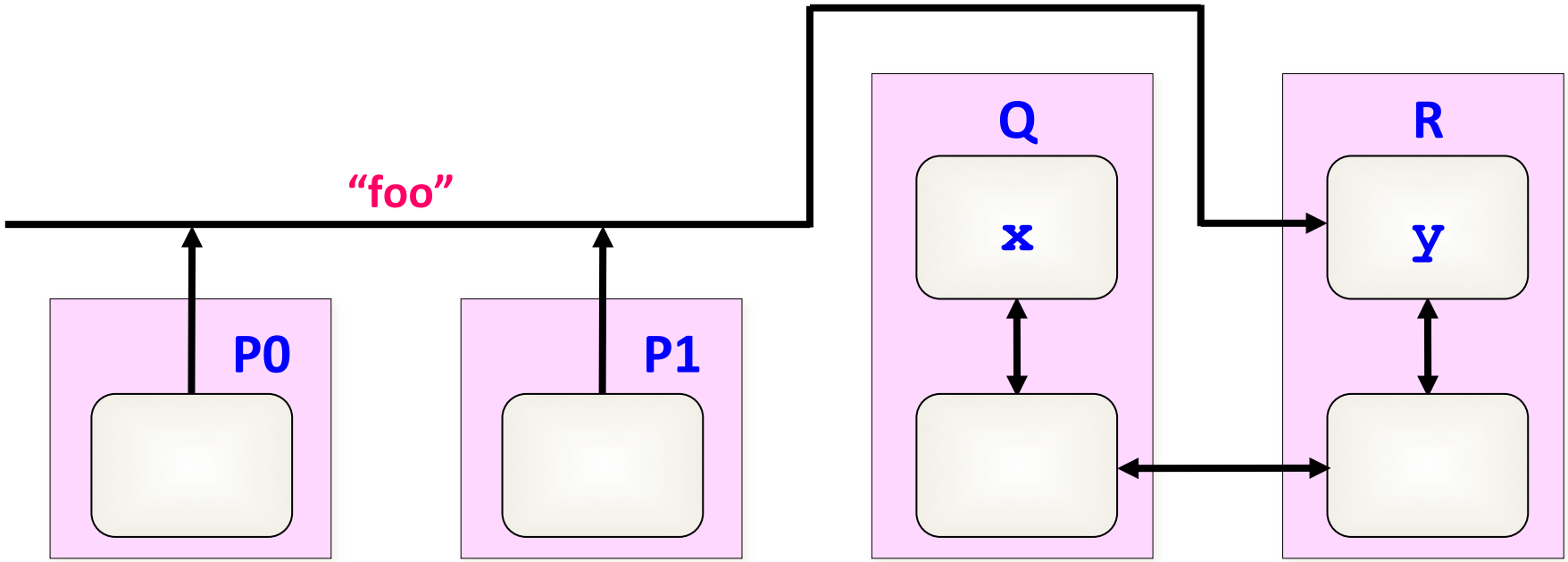
Mobile Network Channels

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



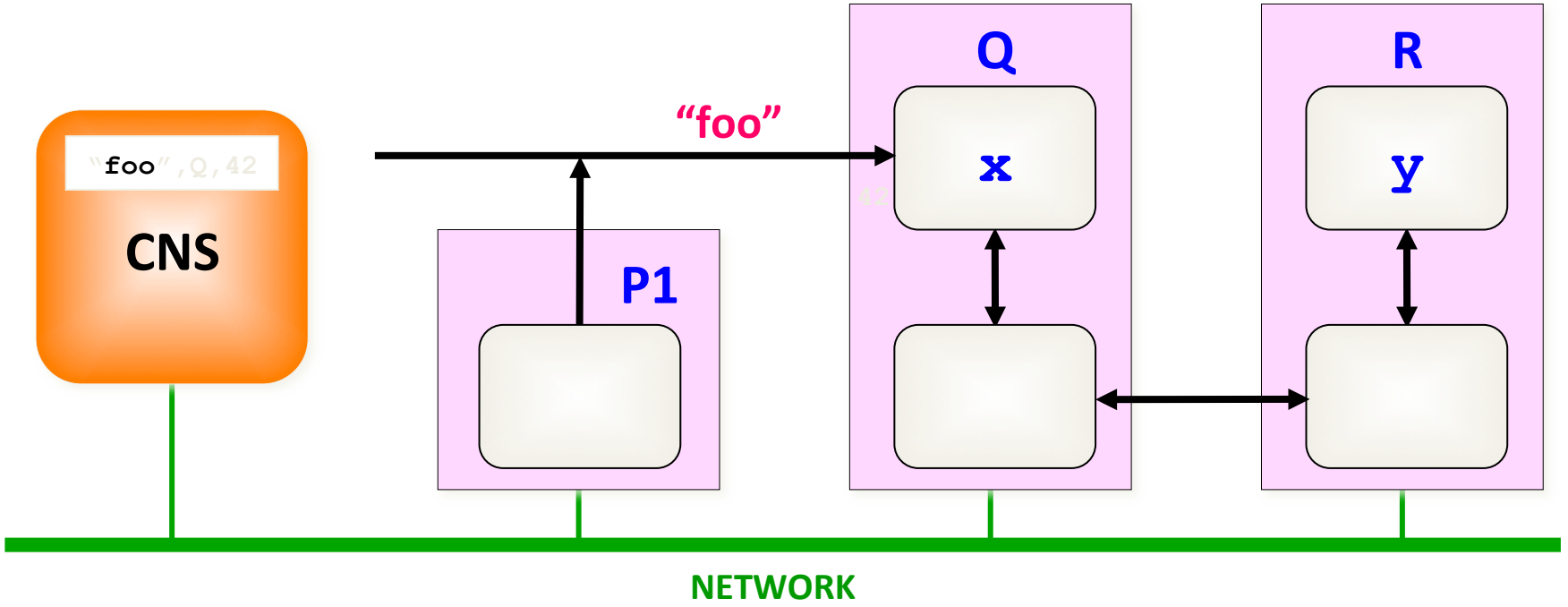
Mobile Network Channels

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



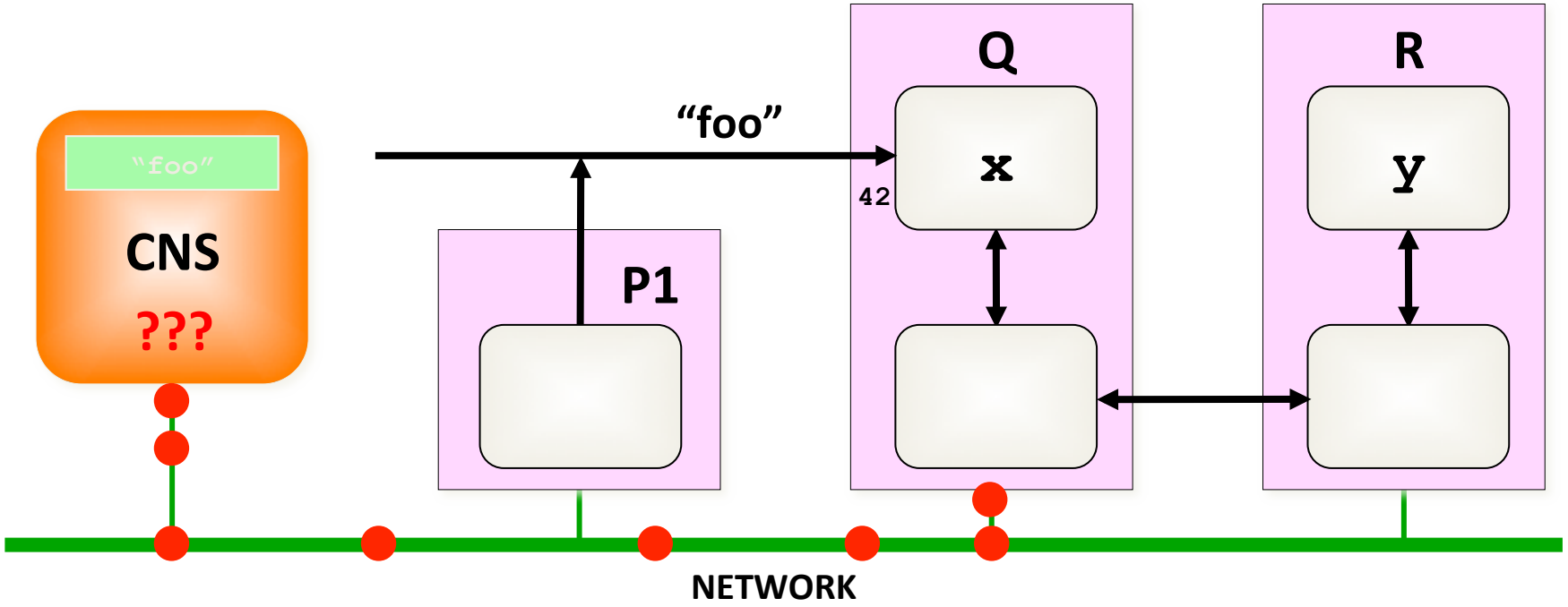
Mobile Network Channels

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



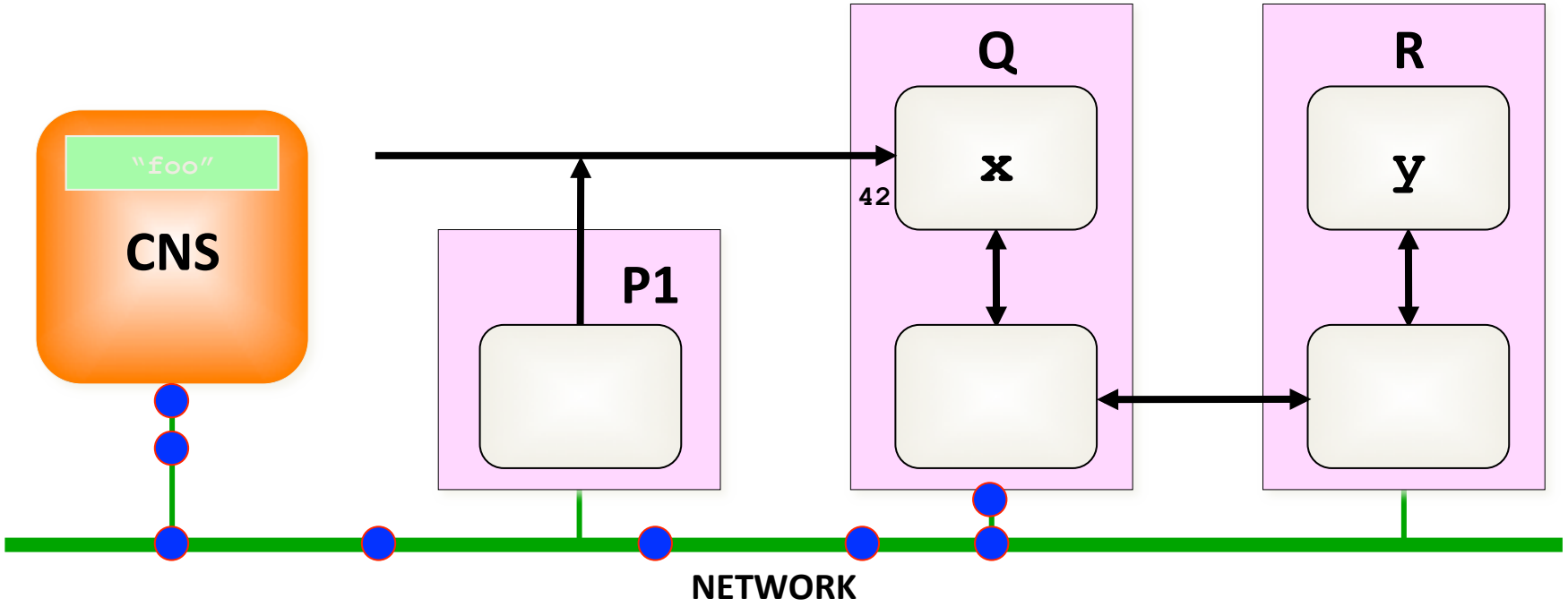
Mobile Network Channels

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



Mobile Network Channels

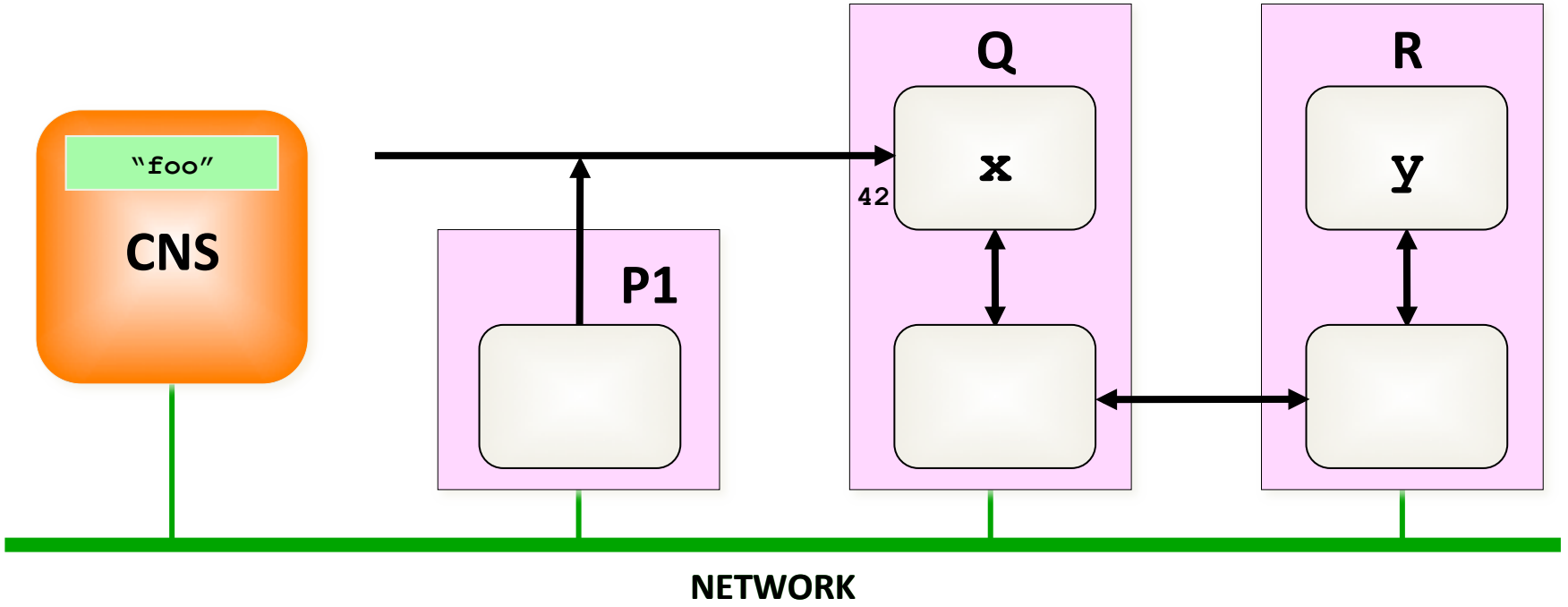
- First, process **x** *freezes* the name **"foo"** on the CNS …
- The CNS returns an *unfreeze key* to process **x** …



Mobile Network Channels

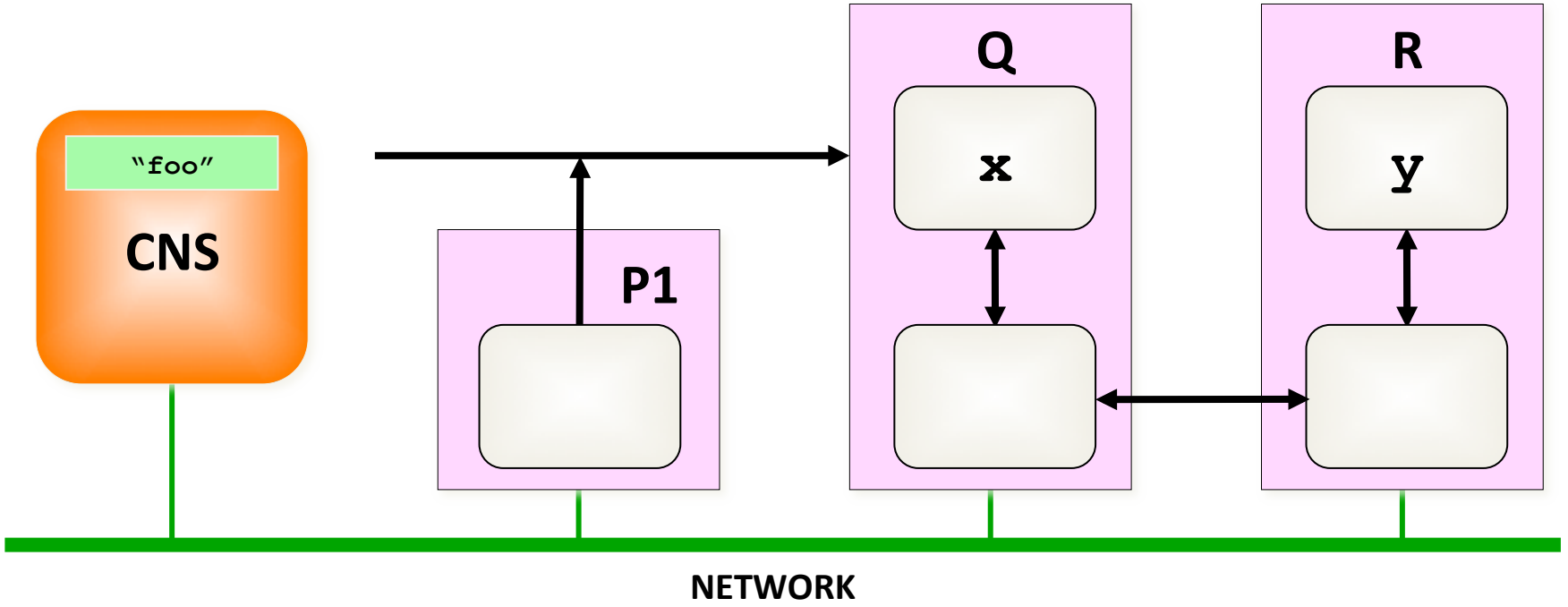
- The CNS no longer resolves **"foo"** for new writers and also disallows new registrations of the name.

- The network channel is deleted from processor **Q**.



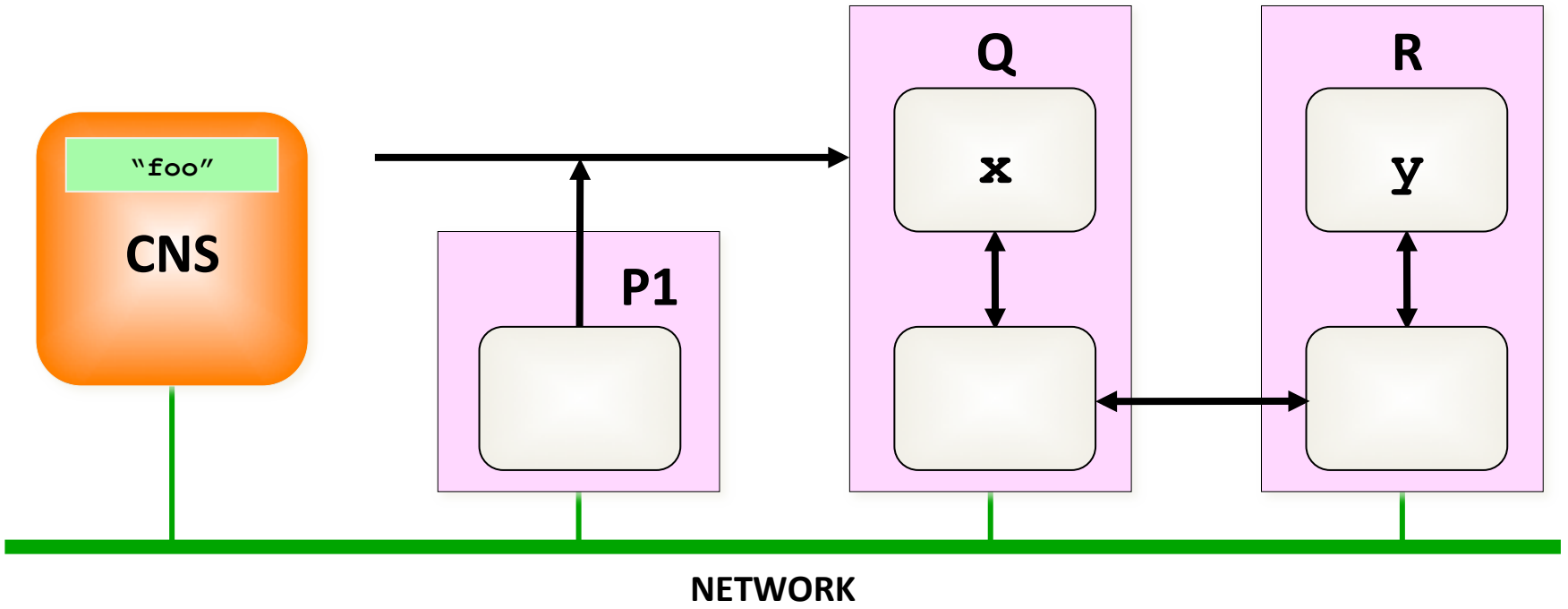
Mobile Network Channels

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



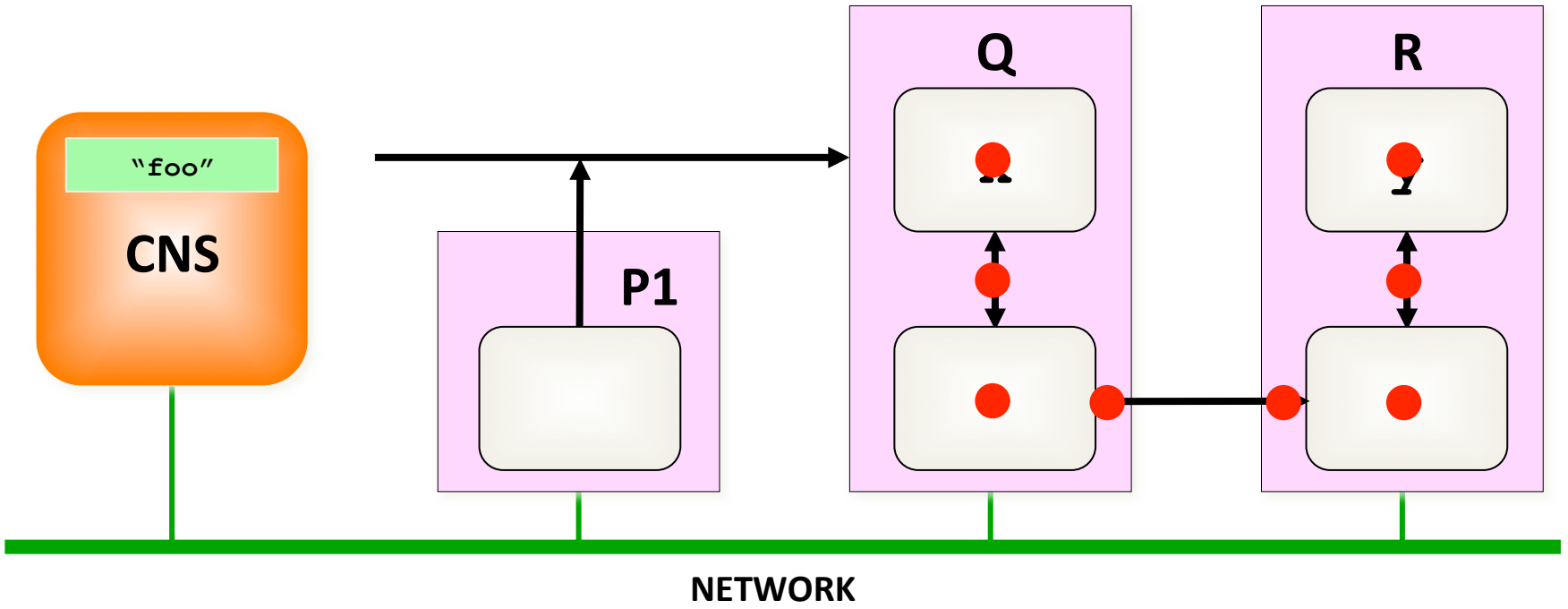
Mobile Network Channels

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



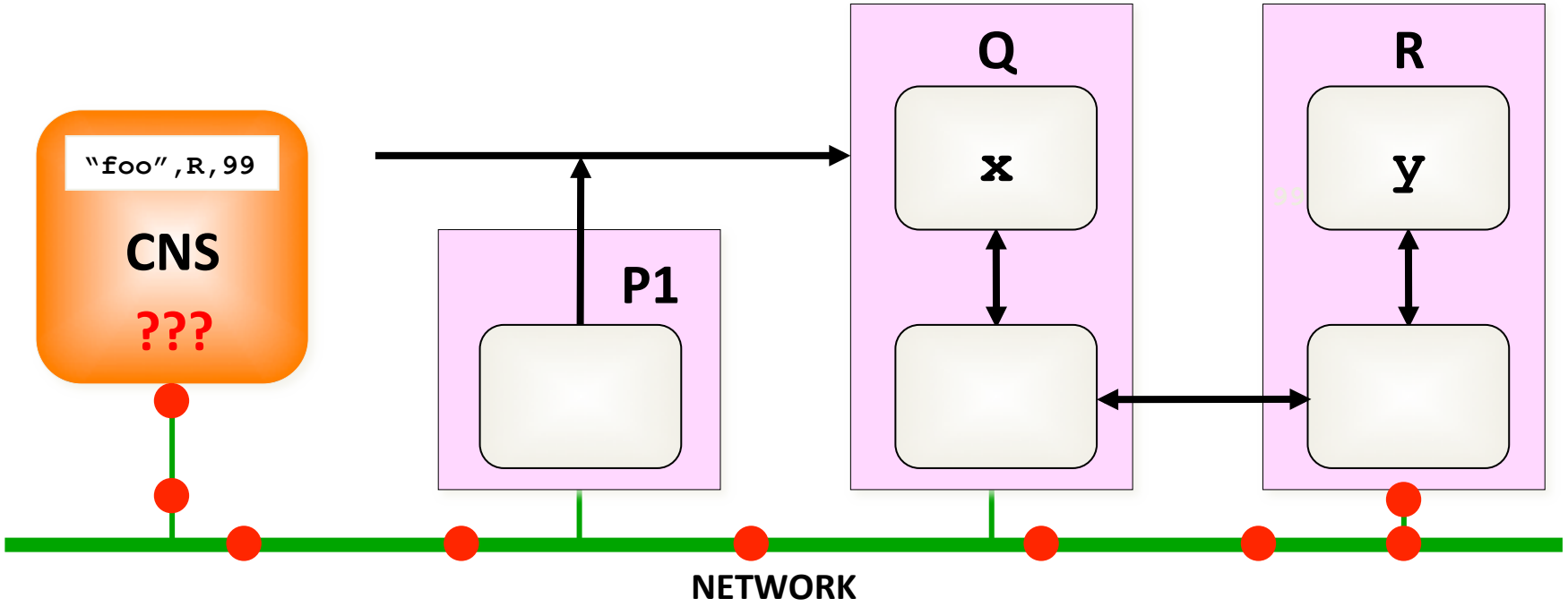
Mobile Network Channels

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



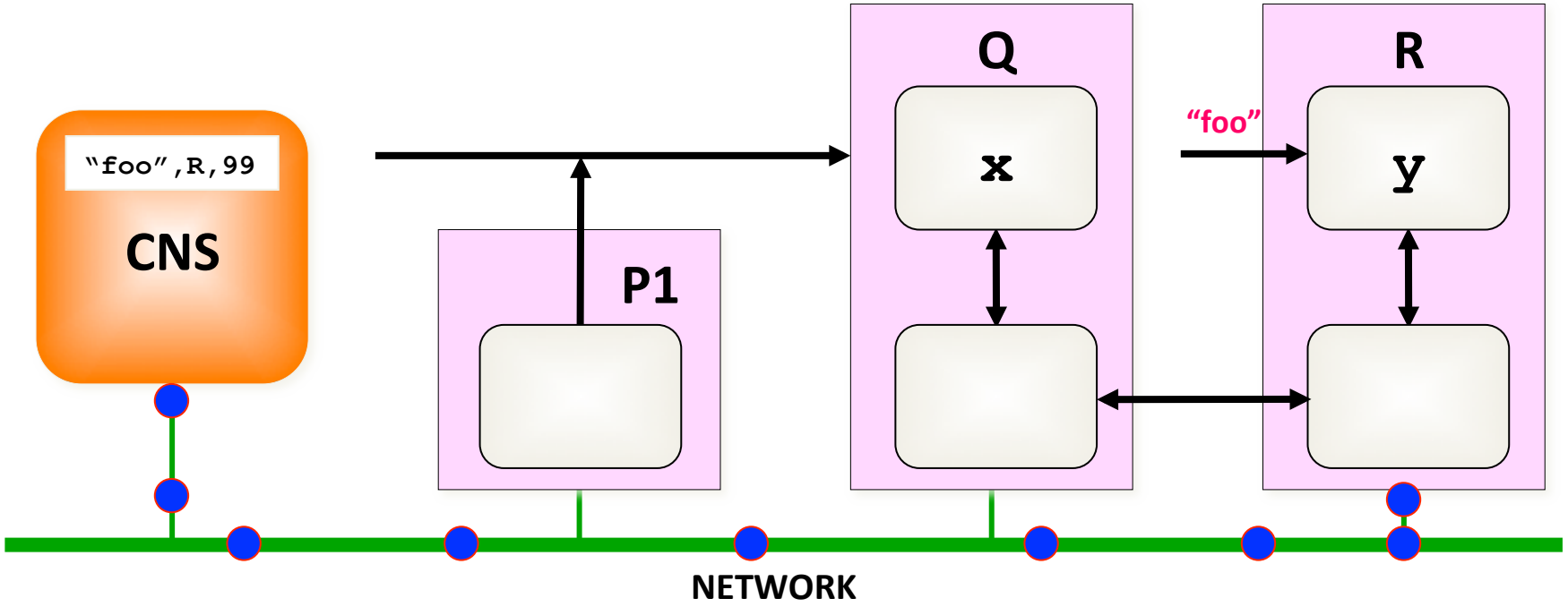
Mobile Network Channels

- … 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.



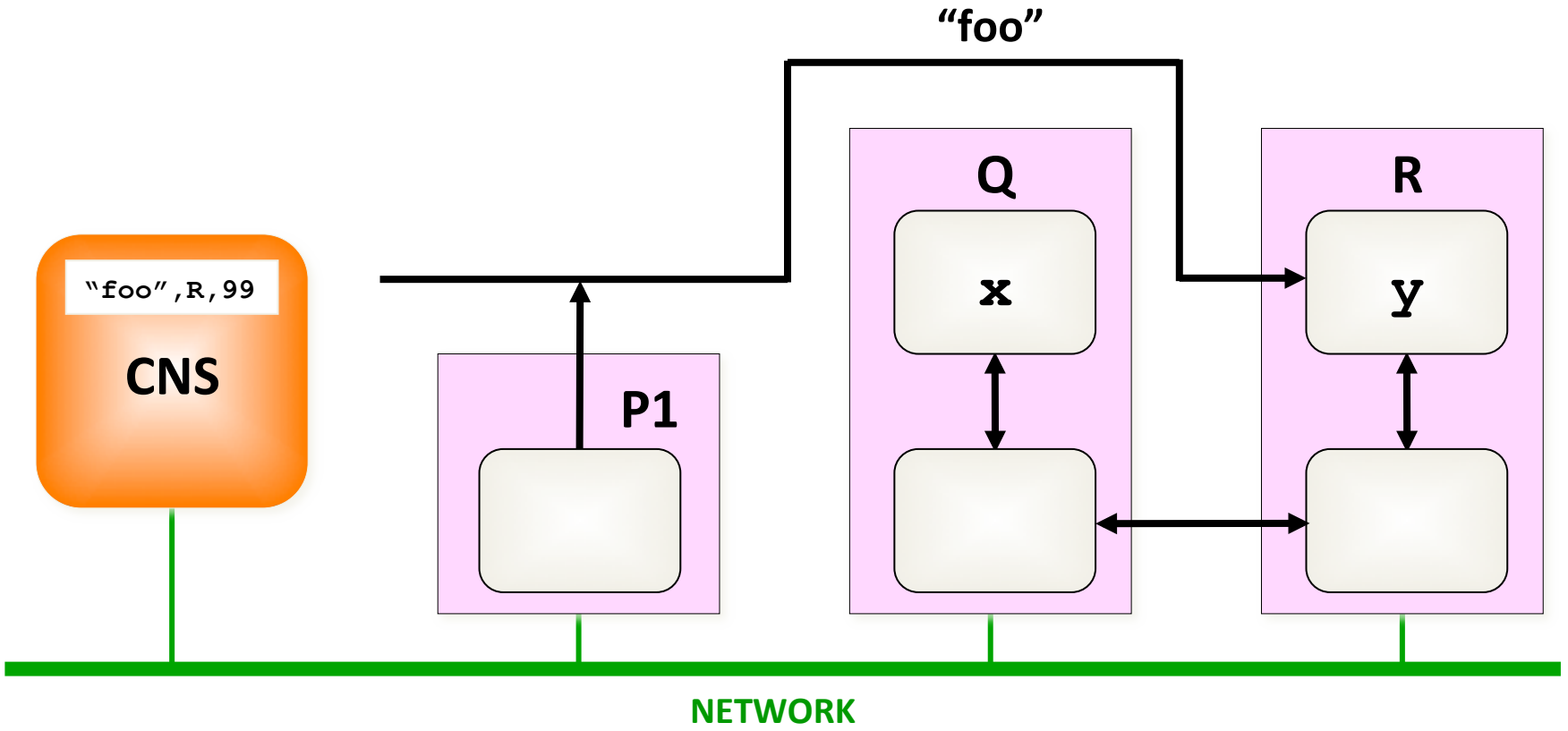
Mobile Network Channels

- … 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" …



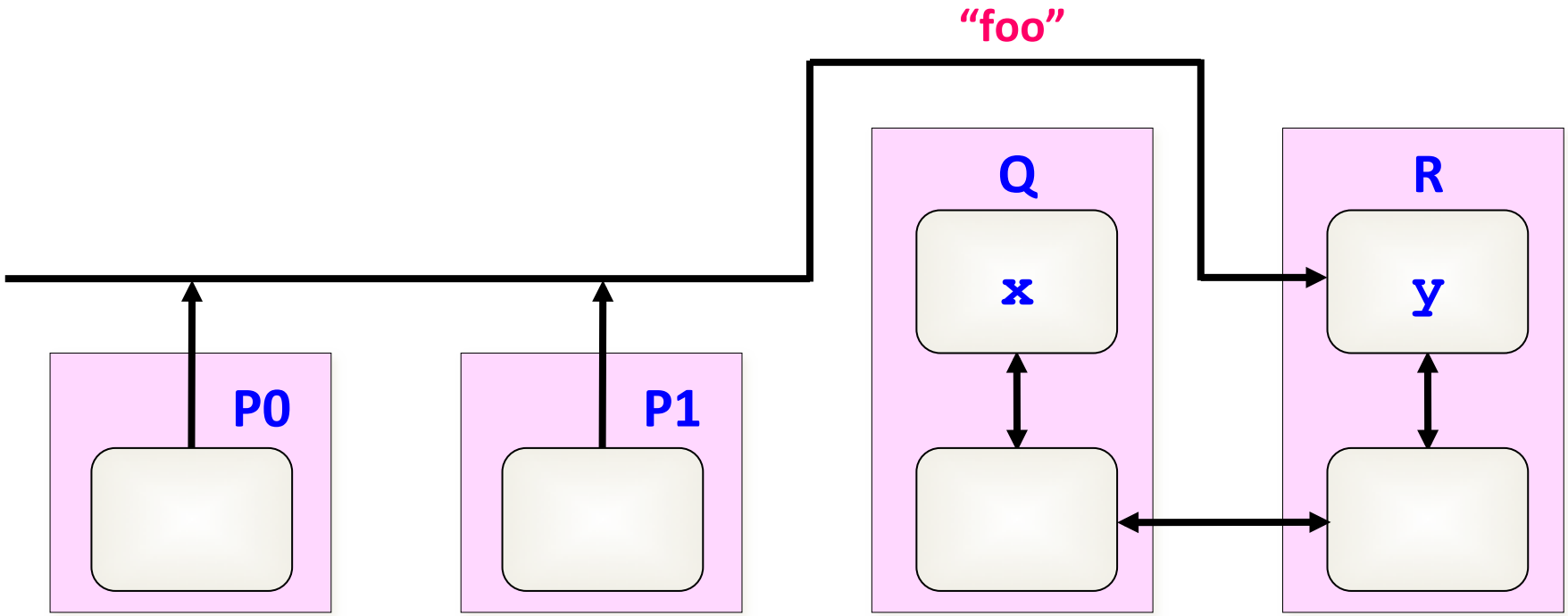
Mobile Network Channels

- … 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

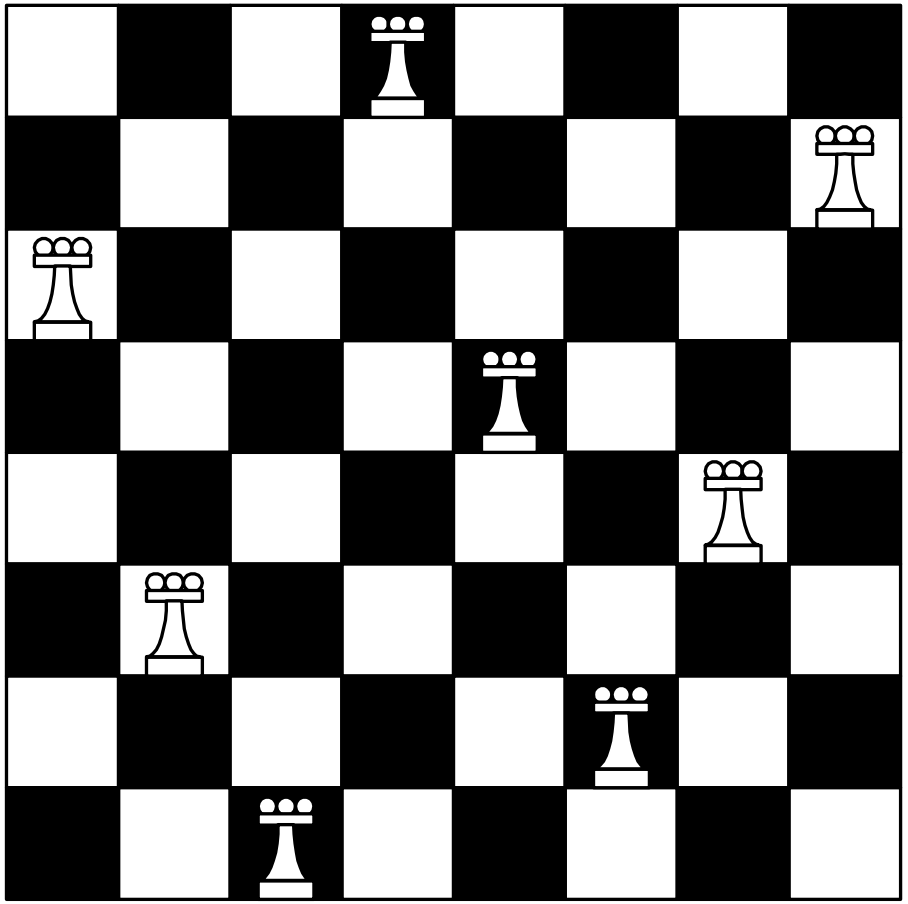
- Processes can ***migrate*** between processors (with classes loaded dynamically as necessary) – hence *mobile agents, worker farms, grid computation …*
- ***JCSP.net*** provides ***exactly the same*** (**CSP/occam**) concurrency model for networked systems as ***JCSP*** provides within each physical 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.

Examples

Bag-of-tasks programming

- When a problem can easily be split into independent sub-tasks
- Concurrent versions are very easy
  - But you can run into load-imbalance if your task size is too big

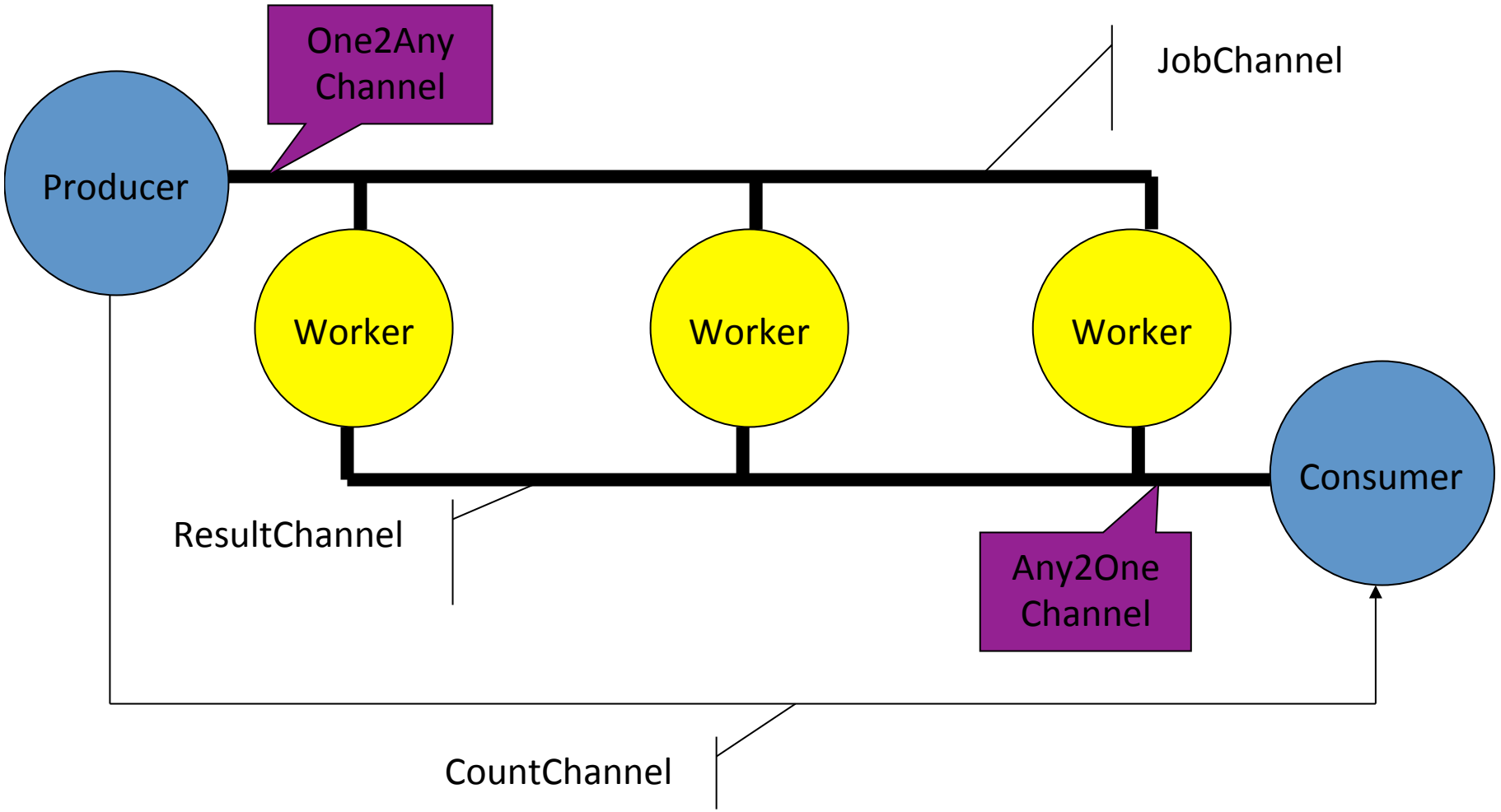
N-Queens Problem



Approach

- Generate a set of subtrees
- Put each subtree in a bag
  - Or simply make a list – it does not take long
- Have workers take tasks from the bag
  - And place results in another bag
- Sum the results
  - Or just let the bag do the summing

N-Queens example



Implementation

- **Master**

```
pool=[]
for i in range(2):
  for j in range(n):
    if(board(i,j)):
      pool.append([i,j])
CountChannel.Write(len(pool))
for x in pool:
  JobChannel.write(x)
```

Implementation

- **Worker**

```
while(true):
  coor=JobChannel.read()
  valid=checkboard(i,j)
  ResultChannel.write(valid)
```

Implementation

- **Consumer**

```
n = CountChannel.read()
sum = 0
for i in range(n):
  sum = sum + ResultChannel.read()
print sum
```

Two concurrent web servers

- Two basic designs for a concurrent web server
  - High throughput web server
  - High performance web server

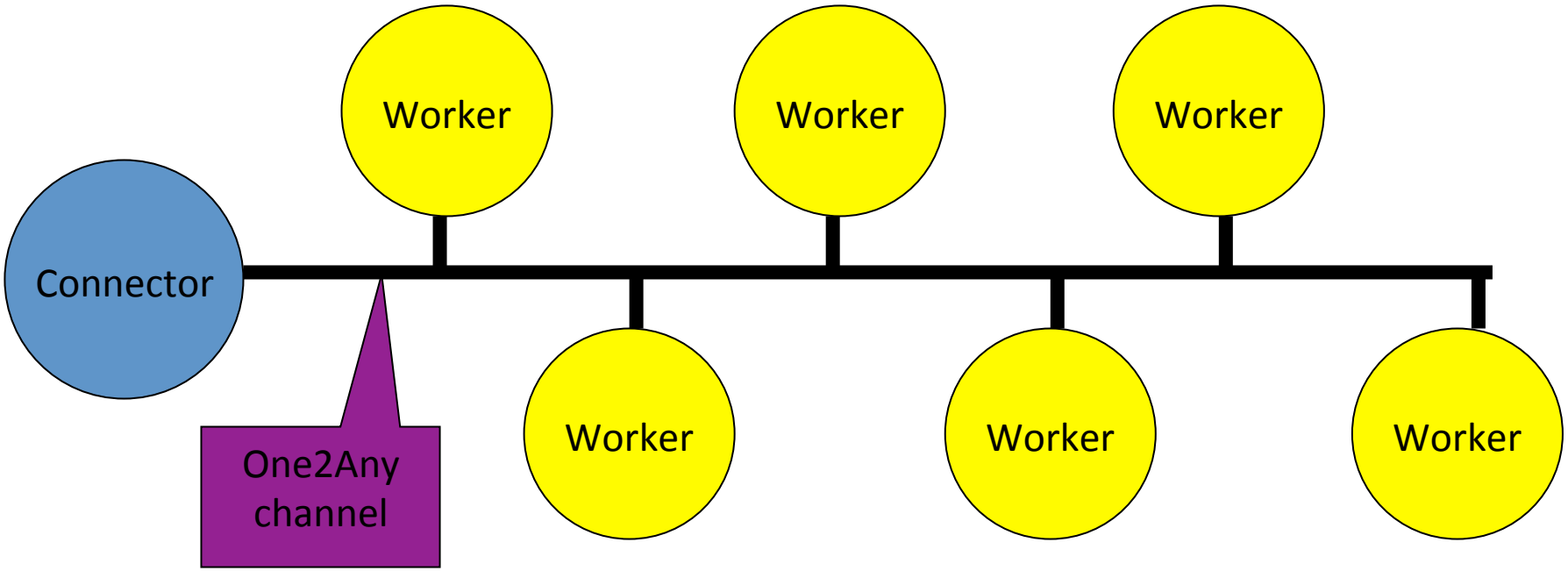
High throughput web server

- We desire a web server that can handle many concurrent sessions
- Each request is an operation in its own right
  - So each request can be implemented as a process in it's own right
  - This is the same approach that is used in common web-servers

High throughput web server

- Two basic approaches to load-balancing
  - Use a semaphore to protect the listen/accept calls
  - Have one thread do all listen/accept calls and the pass the new socket to a worker thread
- The latter is far preferable

High throughput webserver



Psudo Code

Master:

```
x = new socket()
bind(x, 80)
while(true):
  listen(x,1)
  y=accept(x)
  chan ! y
```

Worker:

```
while(true):
  chan ? y
  service(y)
```

High performance web server

- We desire a web server that can service a single webpage very quickly
- http 1.1 has support for ’keep-alive’ and pipelined requests
  - So that when parsing a html page a client can send multiple requests towards the same page
    - Multiple images i.e.

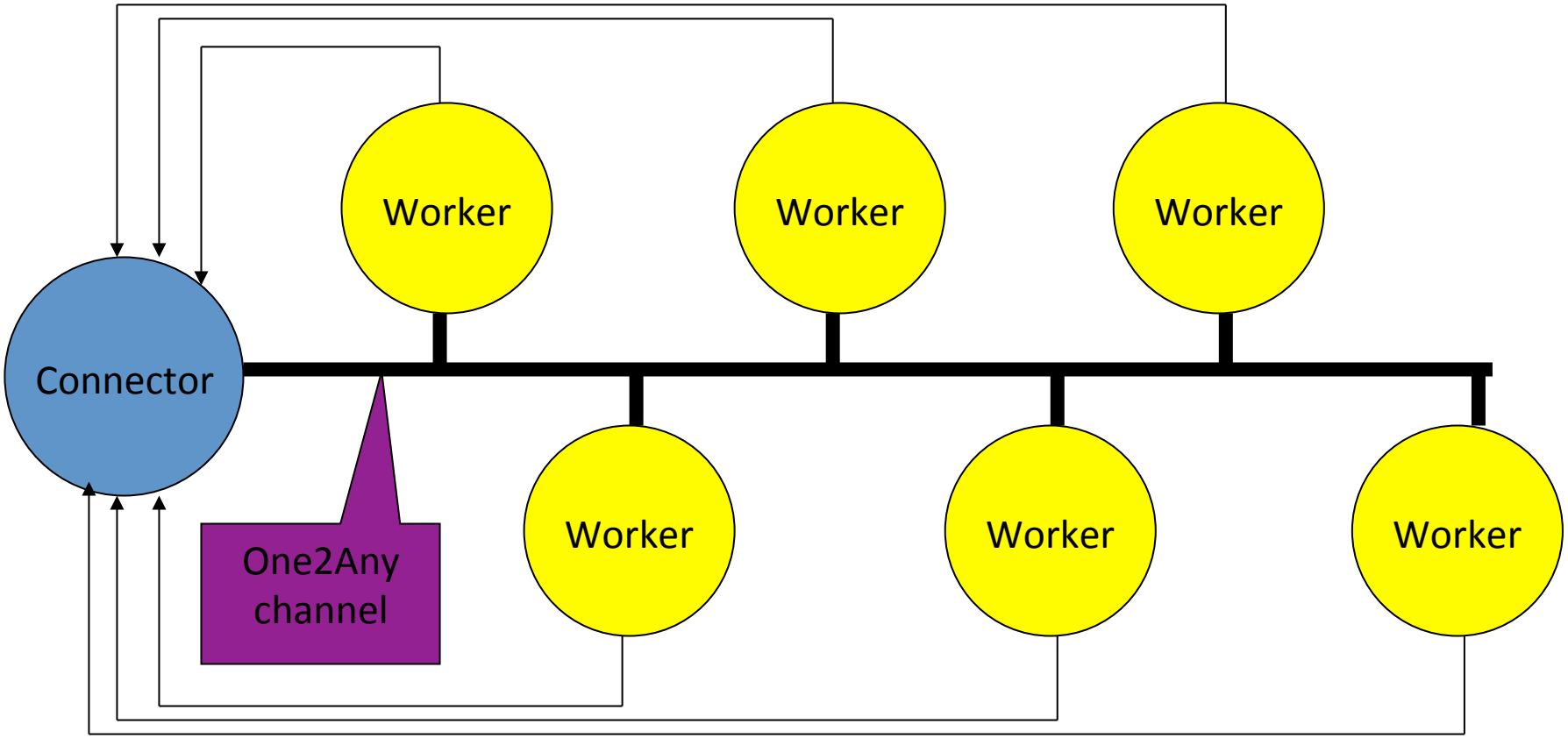
Approach

- Similar to the high throughput
  - But instead of passing sockets to the workers we pass requests
  - But the workers cannot write the answer directly to the socket any longer
- Thus the replies will have to be sent through the master

Approach

- We can use the same basic approach to distribute the requests
- But then we need each process to pass its results to the master who can then send it back

High performance web server



Psudo Code

Master:

```
x = new socket()
bind(x, 80)
while(true):
  listen(x,1)
  y=accept(x)
  PRIALT:
    y      ? req; chan ! req
    w[0] ? ans; y ! ans
    …
    w[n] ? ans; y ! ans
```

Worker:

```
while(true):
  chan?y
  chan ! service(y)
```