Mutually Assured Destruction† (or the Joy of Sync)

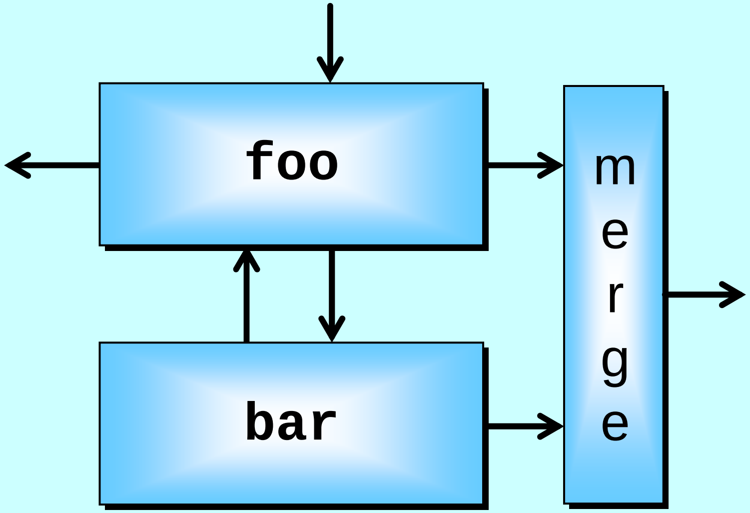
Peter Welch (**phw@kent.ac.uk**),
Jan Bækgaard Pedersen (**matt.pedersen@unlv.edu**)
Frederick R.M. Barnes (**frmb@kent.ac.uk**)

CPA 2013 Fringe, Napier University, 25 August, 2013

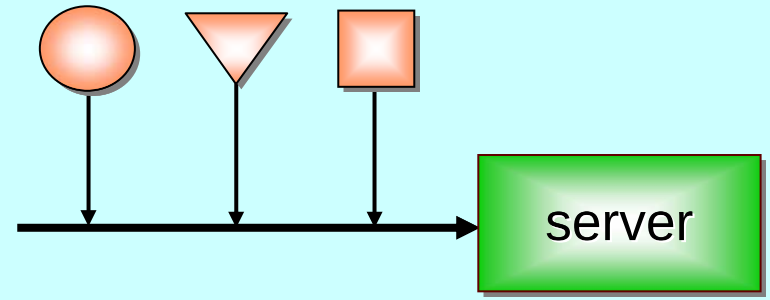
† plus non-blocking barriers and performance …

The Joy of Sync

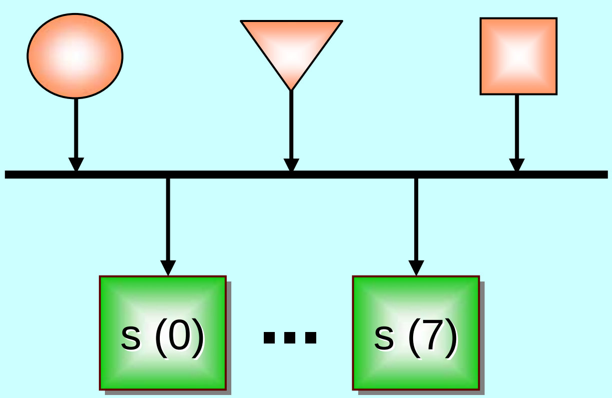
- Process oriented design ...
- Synchronous communications …
- Synchronous barriers …
- Mutually assured destruction …
- Non-blocking barriers …
- Performance …



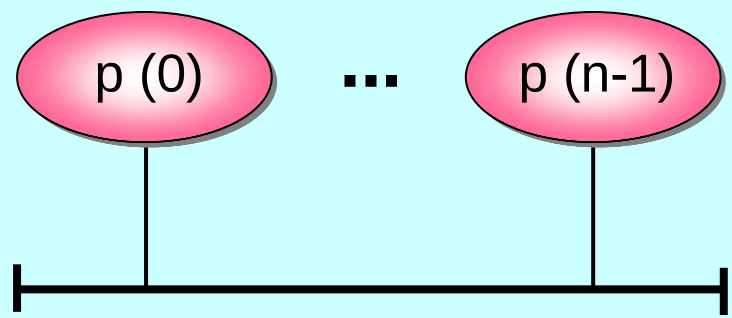
(a) a network of three processes, connected by four internal (hidden) and three external channels.



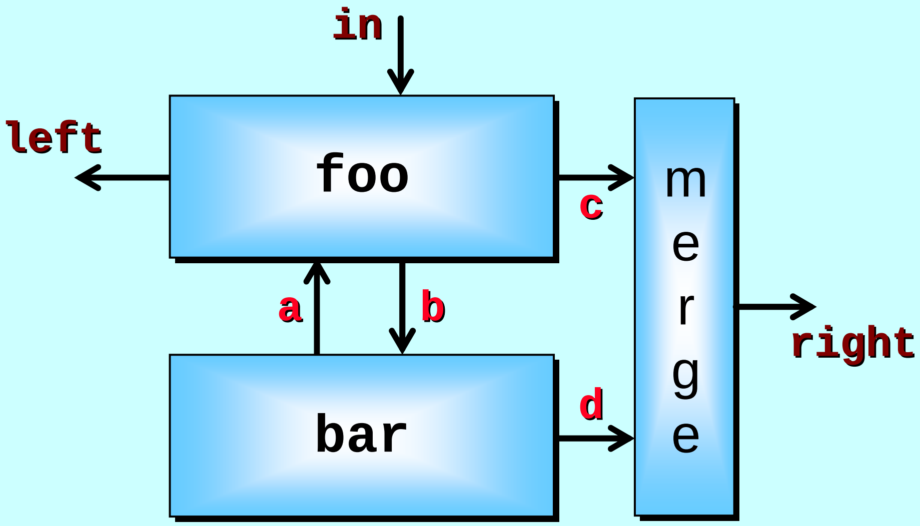
(b) three processes sharing the writing end of a channel to a server process.



(c) three processes sharing the writing end of a channel to a bank of servers sharing the reading end.

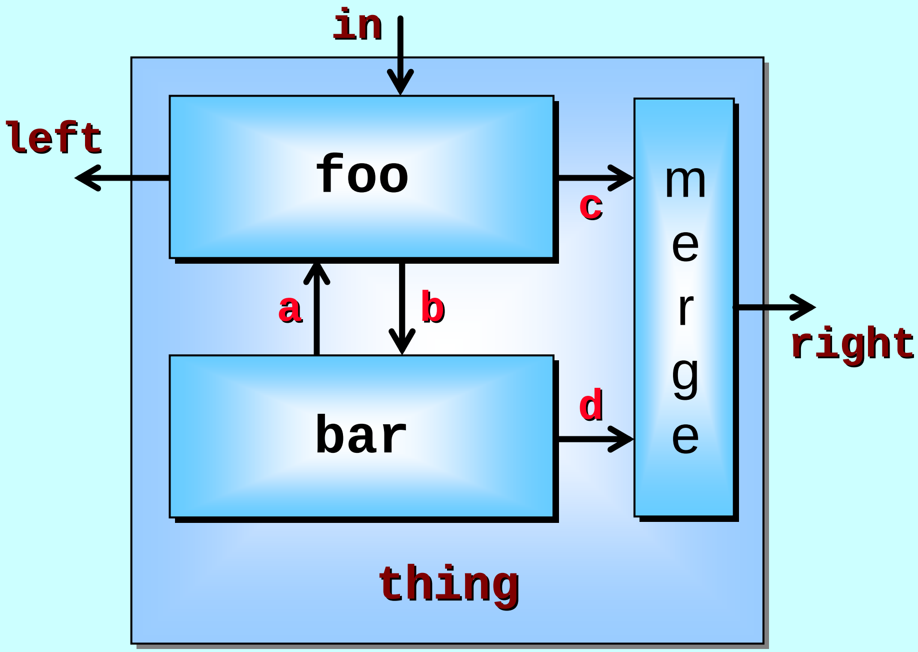


(d) n processes enrolled on a shared barrier (any process synchronising must wait for all to synchronise).

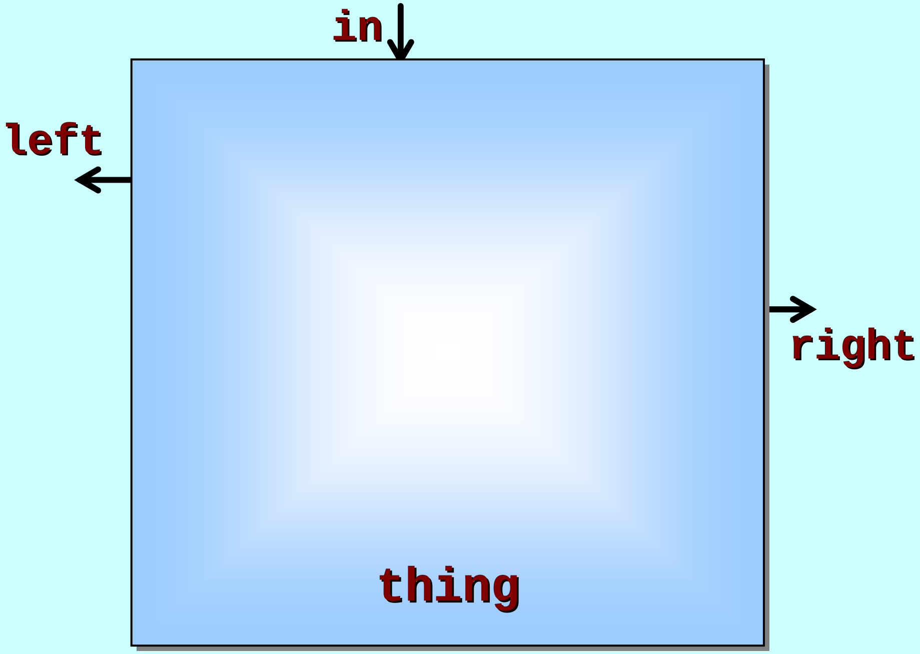


(a) a network of three processes, connected by four internal (hidden) and three external channels.

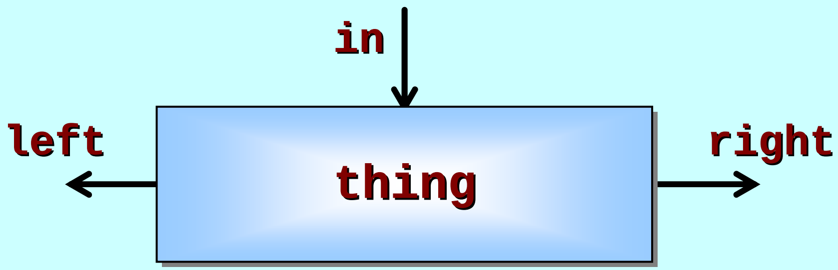
```
CHAN BYTE a, b, c, d:
PAR
  foo (in?, left!, a?, b!, c!)
  bar (a!, b?, d!)
  merge (c?, d?, right!)
```




```
PROC thing (CHAN INT in?, left!, right!)
  CHAN BYTE a, b, c, d:
  PAR
    foo (in?, left!, a?, b!, c!)
    bar (a!, b?, d!)
    merge (c?, d?, right!)
:
```



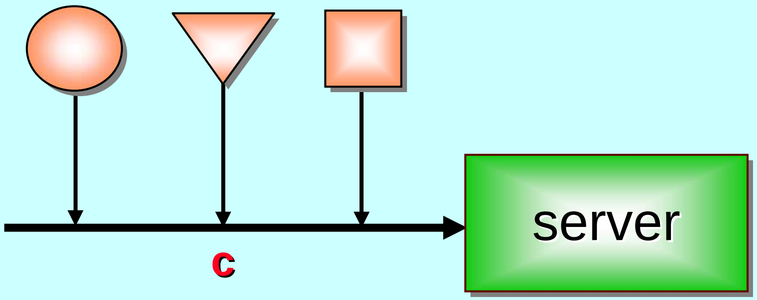

```
PROC thing (CHAN INT in?, left!, right!)
  CHAN BYTE a, b, c, d:
  PAR
    foo (in?, left!, a?, b!, c!)
    bar (a!, b?, d!)
    merge (c?, d?, right!)
:
```




```
PROC thing (CHAN INT in?, left!, right!)
```

Like **foo**, **bar** and **merge** previously, **thing** is a process that can be used as a component in another network.

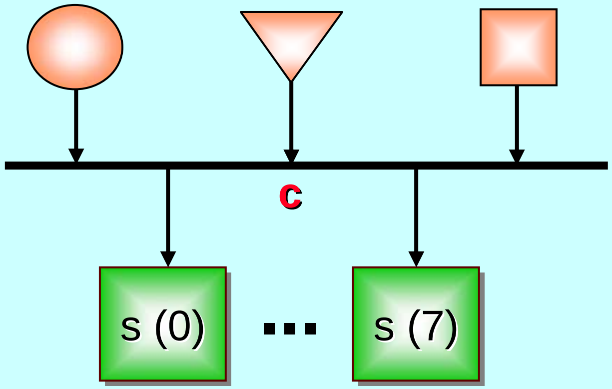
Concurrent systems have structure – networks within networks.
We must be able to express this! And we can … ☺ ☺ ☺



(b) three processes sharing the writing end of a channel to a server process.

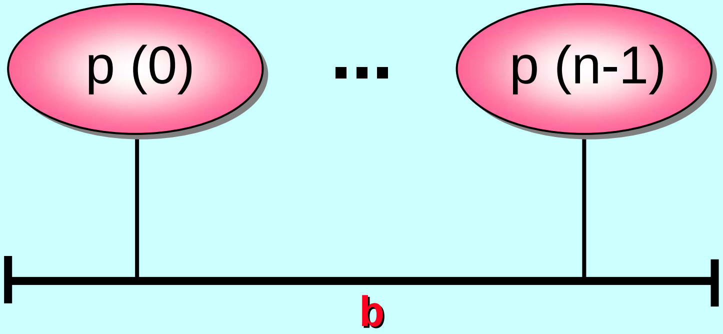
```
SHARED ! CHAN SOME.SERVICE c:
PAR
  circle (c!)
  triangle (c!)
  square (c!)
  server (c?)
```

```
SHARED CHAN ANOTHER.SERVICE c:
PAR
  PAR
    circle (c!)
    triangle (c!)
    square (c!)
  PAR i = 0 FOR 8
    s (i, c?)
```



(c) three processes sharing the writing end of a channel
to a bank of servers sharing the reading end.

```
BARRIER b:
PAR i = 0 FOR n ENROLL b
  p (i, b)
```



(d) n processes enrolled on a shared barrier (any process synchronising must wait for all to synchronise).

The Joy of Sync

Process oriented design ...

Synchronous communications …

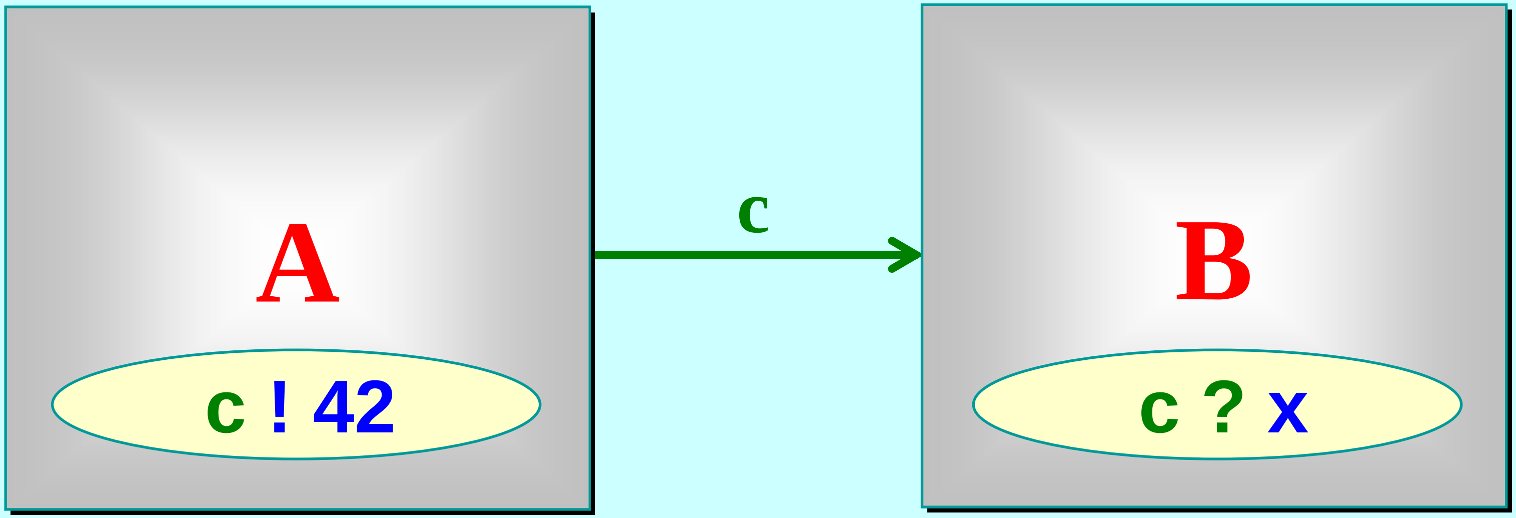
Synchronous barriers …

Mutually assured destruction …

Non-blocking barriers …

Performance …

Synchronised Communication

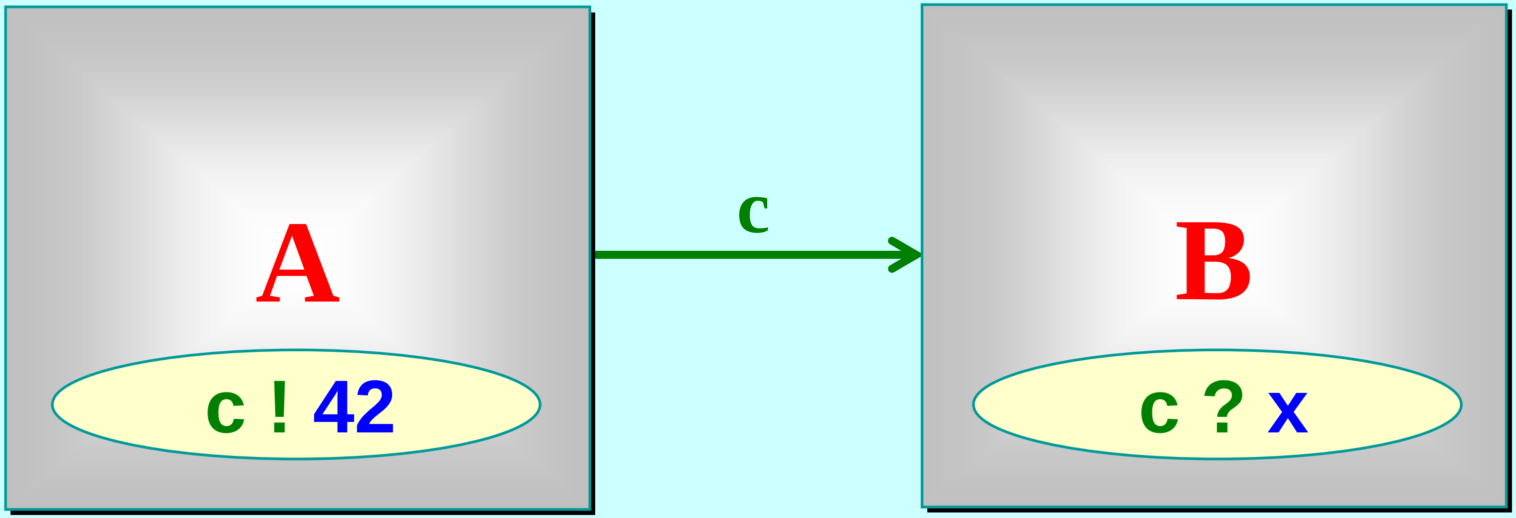


A may ***write*** on ***c*** at any time, but has to wait for a ***read***.

B may ***read*** from ***c*** at any time, but has to wait for a ***write***.

(A (c) || B (c)) \ {c}

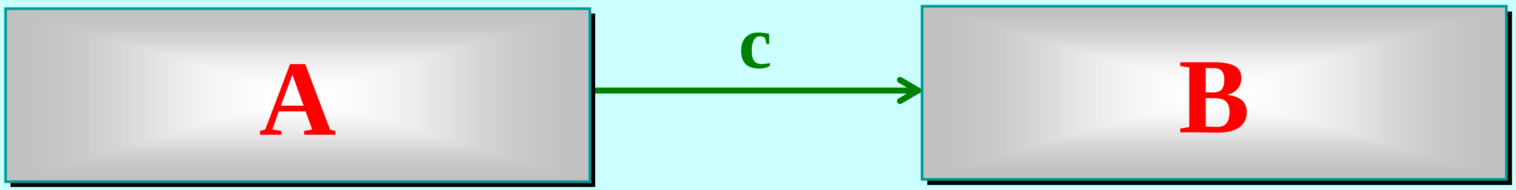
Synchronised Communication



Only when both ***A*** and ***B*** are ready can the communication proceed over the channel ***c***.

(A (c) || B (c)) \ {c}

Synchronised Communication



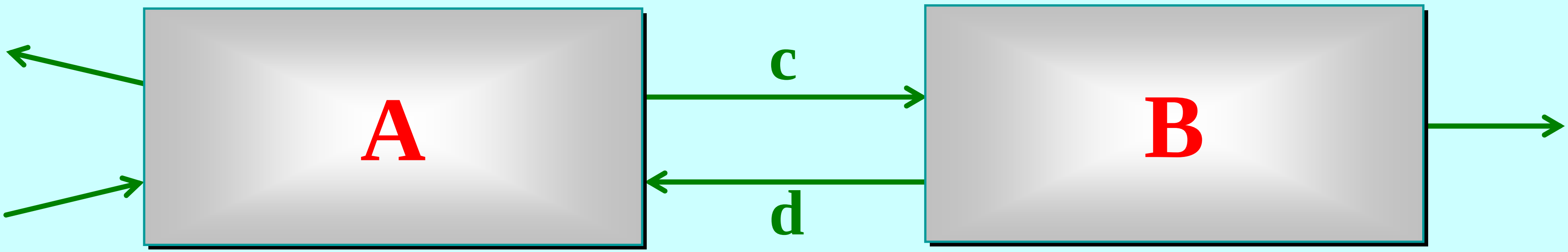
- **Benefit**
  - Once the writer has written, it *knows* the reader has read

OK: plenty of other processes to run and ultra-fast context switch (comparable to a procedure call)

- **Careful**
  - Writer blocks if reader is not ready
  - Lots of deadlock possibilities

OK: work with (a small set of) synchronisation patterns for which we have proven safety theorems

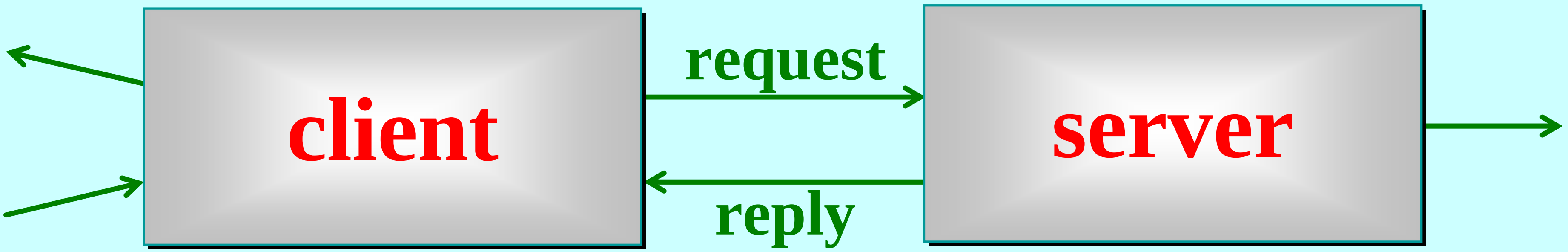
Simple Deadlock Example



If there is no discipline on when *A* and *B* communicate, then *A* may commit to output on *c*, followed by *B* on *d* … or vice-versa. Either way, neither are listening and both are stuck. Same happens if both commit to input.

☹ ☹ ☹ ☹ ☹

Client-Server Pattern



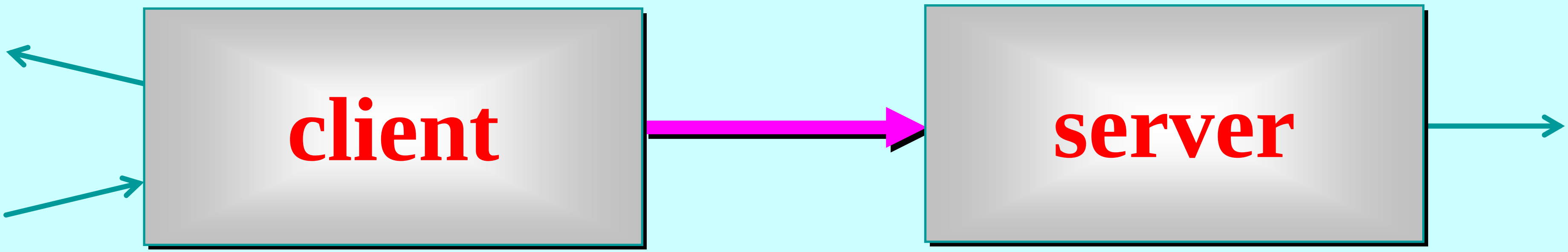
client : makes a *request* any time, then commits to taking *reply*.

server : always accepts a *request* (within some bounded time), then always makes a *reply* (within some bounded time). It may make requests itself, as a *client* to other *servers*.

No deadlock is now possible from this client-server relationship.

☺ ☺ ☺ ☺ ☺

Client-Server Pattern



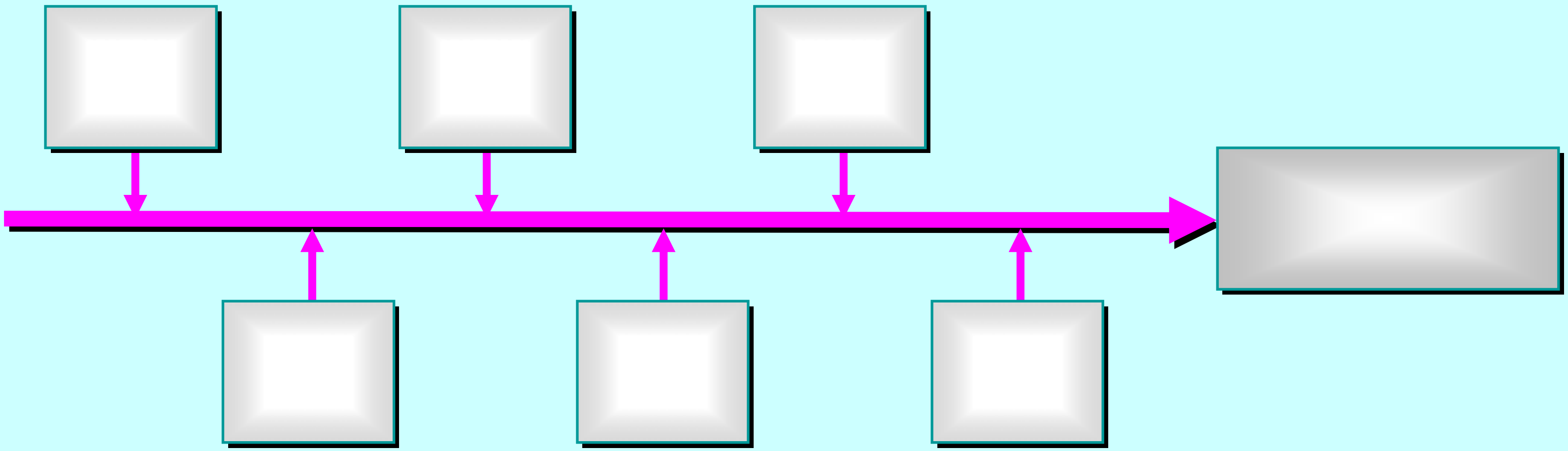
client : makes a *request* any time, then commits to taking *reply*.

server : always accepts a *request* (within some bounded time), then always makes a *reply* (within some bounded time). It may make requests itself, as a *client* to other *servers*.

Symbology: this represents a client-server relation. It points *to* the server and allows a *2-way* conversation (initiated by the client)

Client-Server Pattern

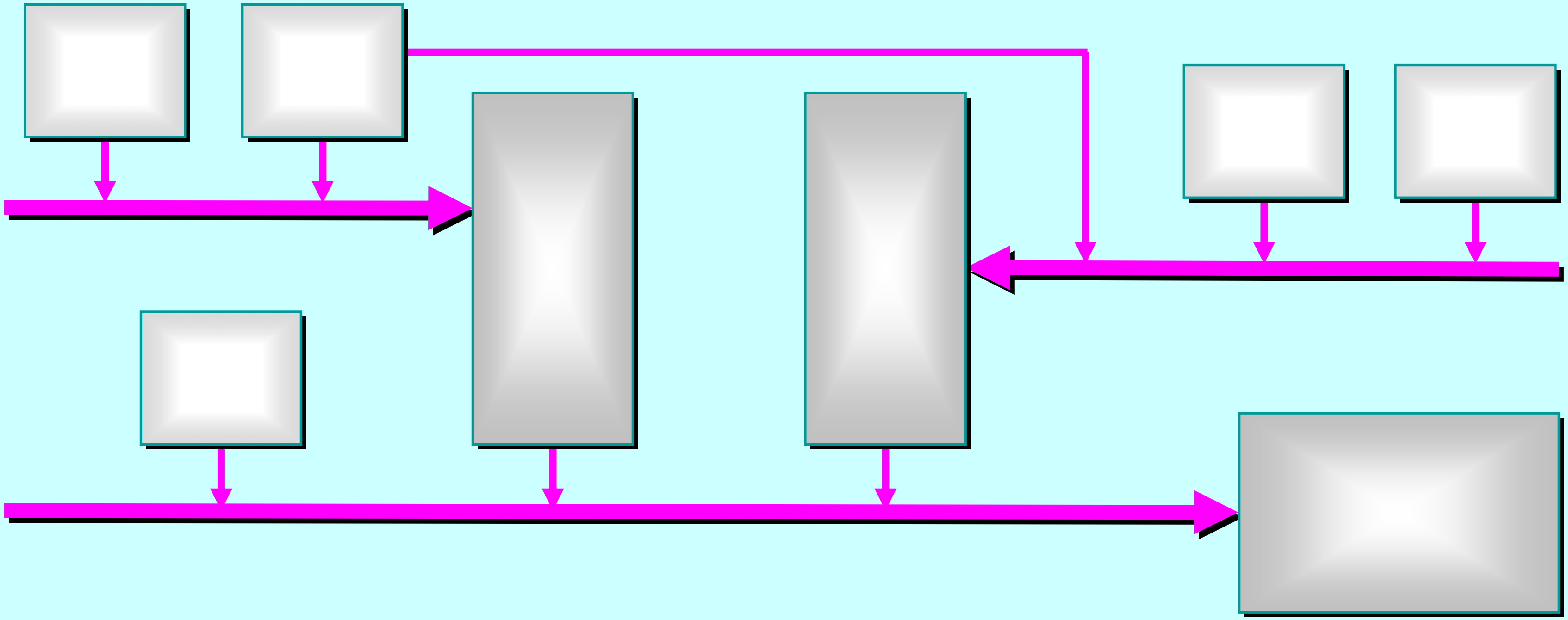
A *server* may have many *clients* …



Only one *client* at a time converses with the *server* . They form an orderly queue. Still no deadlock possible – and no *client* starvation. No polling on the queue, so no livelock either.

Client-Server Theorem

A ***client-server*** system that has no cycles in its ***client-server*** relations is deadlock, livelock and starvation free.



The Joy of Sync

Process oriented design ...

Synchronous communications …

Synchronous barriers …

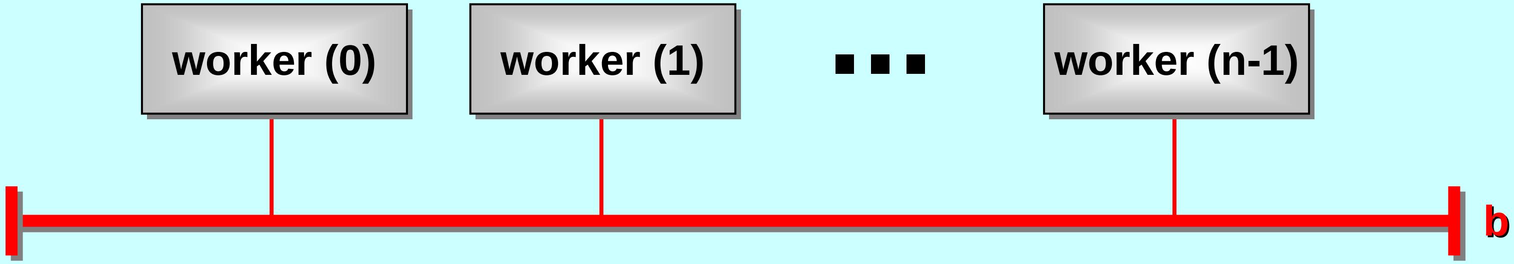
Mutually assured destruction …

Non-blocking barriers …

Performance …

Barriers

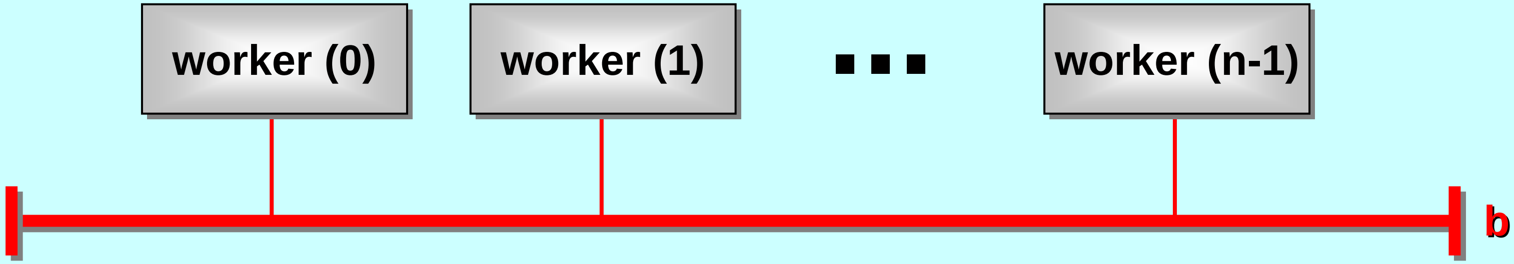
The occam-π **BARRIER** type corresponds to a multiway CSP ***event***, though some higher level design patterns (such as *resignation*) have been built in.



Basic CSP semantics apply. When a process *synchronises* on a barrier, it blocks until all other processes *enrolled* on the barrier have also *synchronised*. Once the barrier has completed (i.e. all *enrolled* processes have *synchronised*), all blocked processes are rescheduled for execution.

Barriers

The occam-π **BARRIER** type corresponds to a multiway CSP *event*, though some higher level design patterns (such as *resignation*) have been built in.



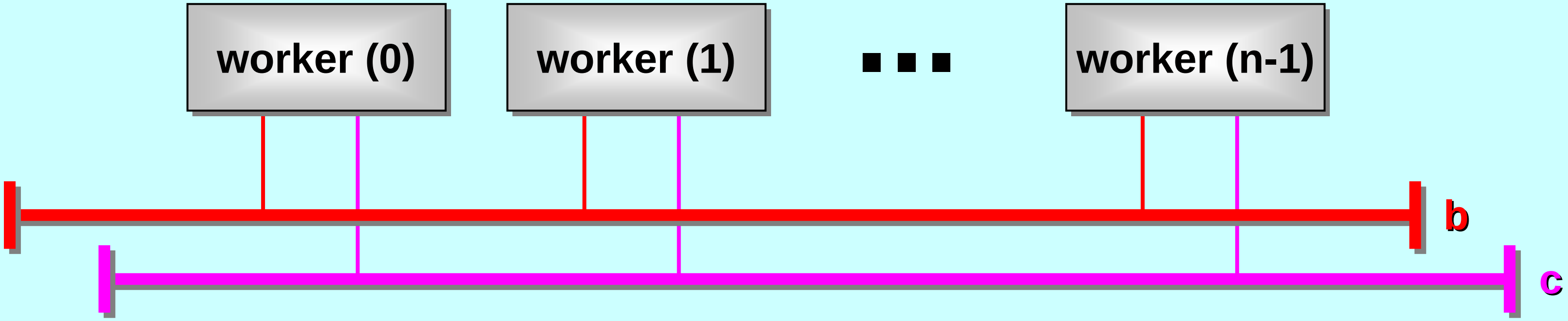

```
BARRIER b:
PAR i = 0 FOR n ENROLL b
  worker (i, b)
```

A **PAR** construct must *explicitly* **ENROLL** its components on barriers

The number of processes enrolled on an in-scope barrier is unchanged by a *non-enrolling* **PAR** – but *only one* of its components may reference it.

Barriers

Processes may synchronise on more than one barrier:




```
BARRIER b, c:
PAR i = 0 FOR n ENROLL b, c
  worker (i, b, c)
```

To synchronise on a barrier:

`SYNC b` or `SYNC c`

Barriers

Barriers are commonly used to synchronise multiple ***phases*** of computation between a set of processes. Within each phase, other synchronisations (channel / barrier) may take place:

```
PROC worker (VAL INT id, BARRIER b, c)
  ...  local declarations / initialisation
  WHILE running
    SEQ
      SYNC b
      ...  observe neighbourhood phase
      SYNC c
      ...  change neighbourhood phase
:
```

All workers do this together – all see the same thing …

All workers do this together – may need to negotiate …

Barriers

But it's safer programming for each phase to be synchronised by its own barrier …

Of course, only ***one*** barrier is actually needed to synchronise the phases in this example:

```
PROC worker (VAL INT id, BARRIER a)
  ...  local declarations / initialisation
  WHILE running
    SEQ
      SYNC a
      ...  observe neighbourhood phase
      SYNC a
      ...  change neighbourhood phase
:
```

All workers do this together – all see the same thing …

All workers do this together – may need to negotiate …

Barriers – Safety

occam-π **BARRIER** *synchronisation* is *safe* in the sense that *enrollment* and *resignation* are automatically managed. A process may *synchronise* on a **BARRIER** if and only if it is *enrolled*.

Try to break this rule … your program won't compile. There are zero memory and run-time costs to enforce it. ☺

The Joy of Sync

Process oriented design ...

Synchronous communications …

Synchronous barriers …

Mutually assured destruction …

Non-blocking barriers …

Performance …

Mutually Assured Destruction

Two processes are given, at the same time, their own task to complete; we are satisfied with the completion of either one of them; whichever process finishes first interrupts the other and reports its completion; the one that is interrupted abandons its task and reports that fact.

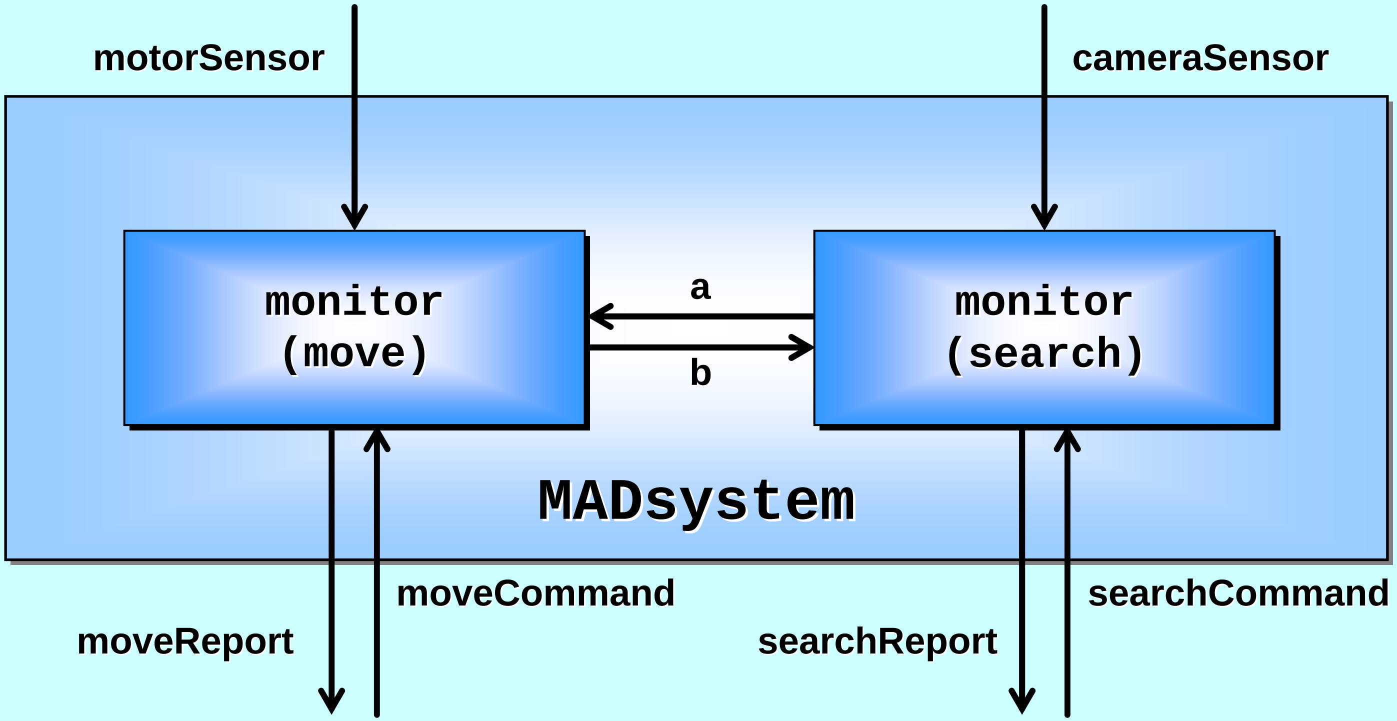
Such requirements are common in control systems, robotics, e-commerce, model-checking, …

e.g.

– Drive rover vehicle forwards *target* meters.

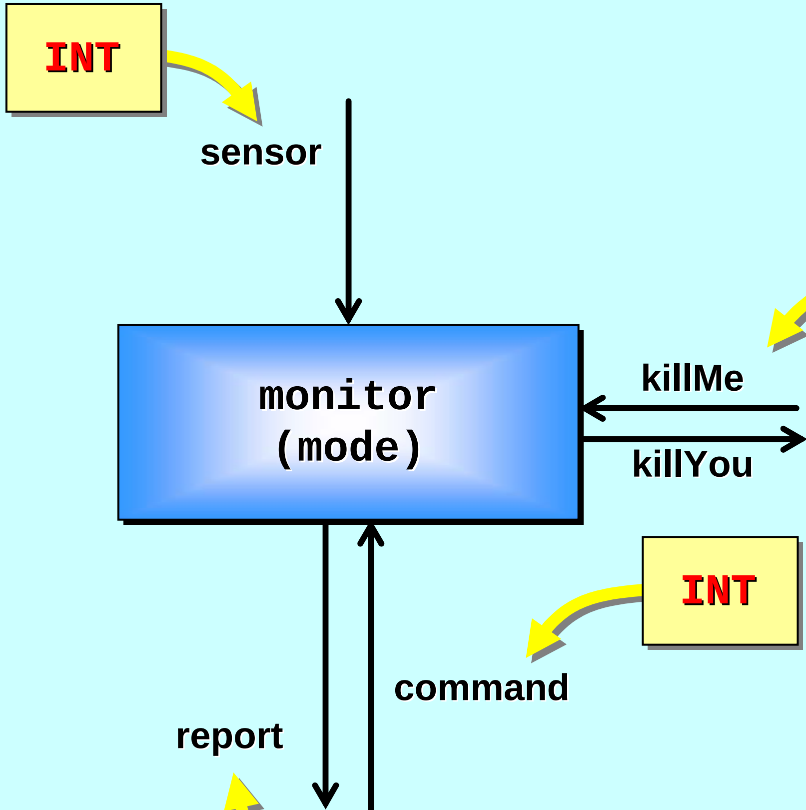
– Look out for *Martians*.

– Stop and report when either is achieved.



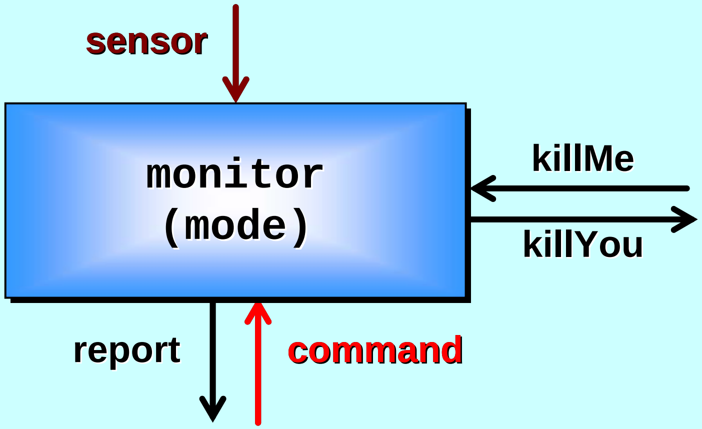
e.g.

– Drive rover vehicle forwards *target* meters.
– Look out for *Martians*.
– Stop and report when either is achieved.

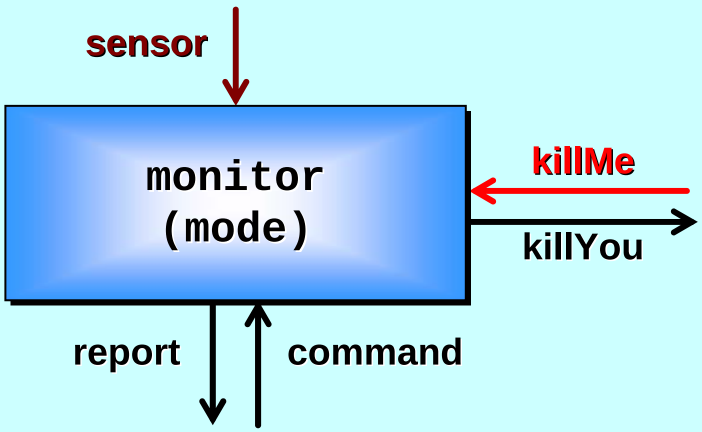



```
PROTOCOL KILL
  CASE
    kill
:
```

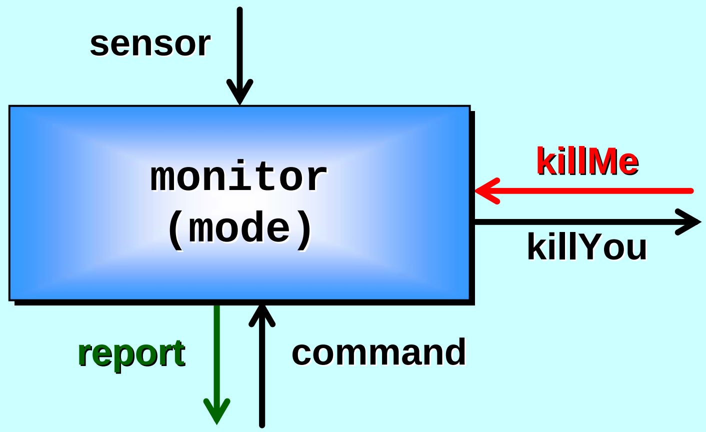
```
PROTOCOL REPORT
  CASE
    me              -- task completed
    she             -- task abandoned
:
```



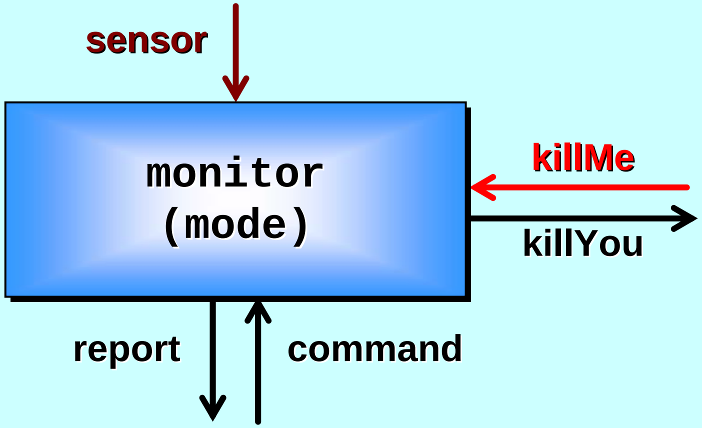

```
PROC monitor (VAL INT mode, CHAN INT command?, sensor?,
              CHAN REPORT report!,
              CHAN KILL killYou!, killMe?)
  WHILE TRUE
    PRI ALT
      INT target:
      command ? target                     -- service requested
        service (mode, target, sensor?, report!,
                 killYou!, killMe?)
      INT x:
      sensor ? x                           -- accept and discard
        SKIP
:
```



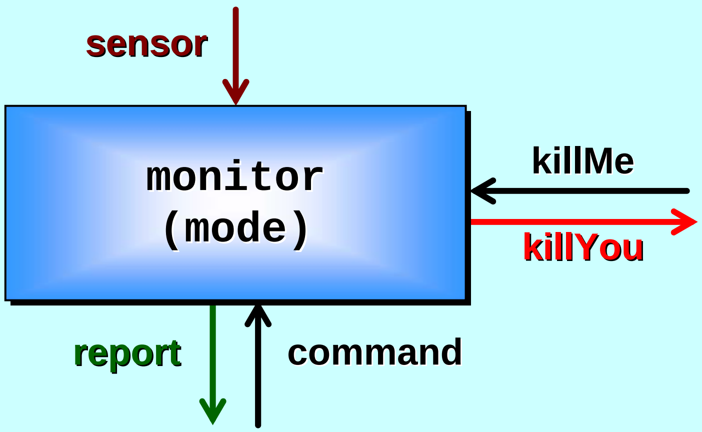

```
PROC service (VAL INT mode, target, CHAN INT sensor?,
              CHAN REPORT report!,
              CHAN KILL killYou!, killMe?)
  ...  local state and initialisation
  INITIAL BOOL running IS TRUE:
  WHILE running
    PRI ALT
      killMe ? kill
        ...  report 'she' and exit loop
      INT x:
      sensor ? x
        ...  process x
:
```



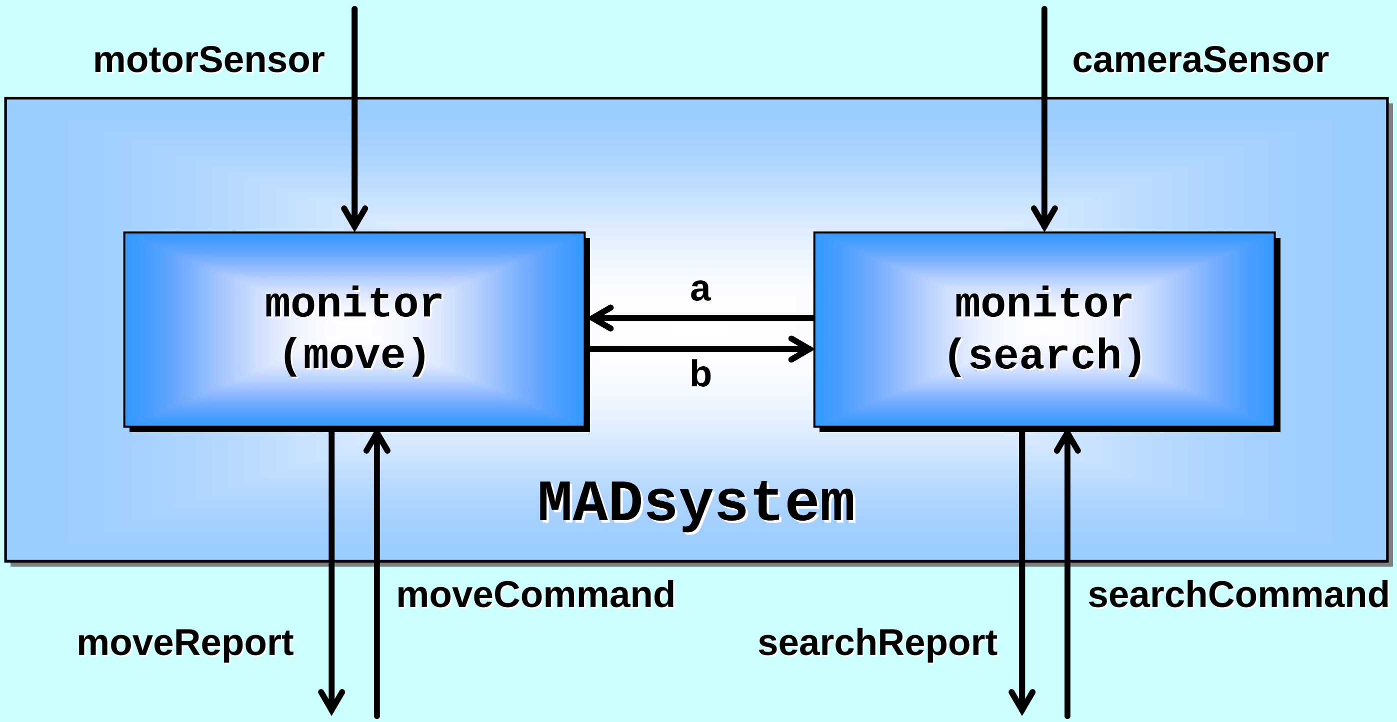

```
killMe ? kill
  SEQ
    report ! she
    running := FALSE
```




```
PROC service (VAL INT mode, target, CHAN INT sensor?,
              CHAN REPORT report!,
              CHAN KILL killYou!, killMe?)
  ...  local state and initialisation
  INITIAL BOOL running IS TRUE:
  WHILE running
    PRI ALT
      killMe ? kill
        ...  report 'she' and exit loop
      INT x:
      sensor ? x
        ...  process x
:
```



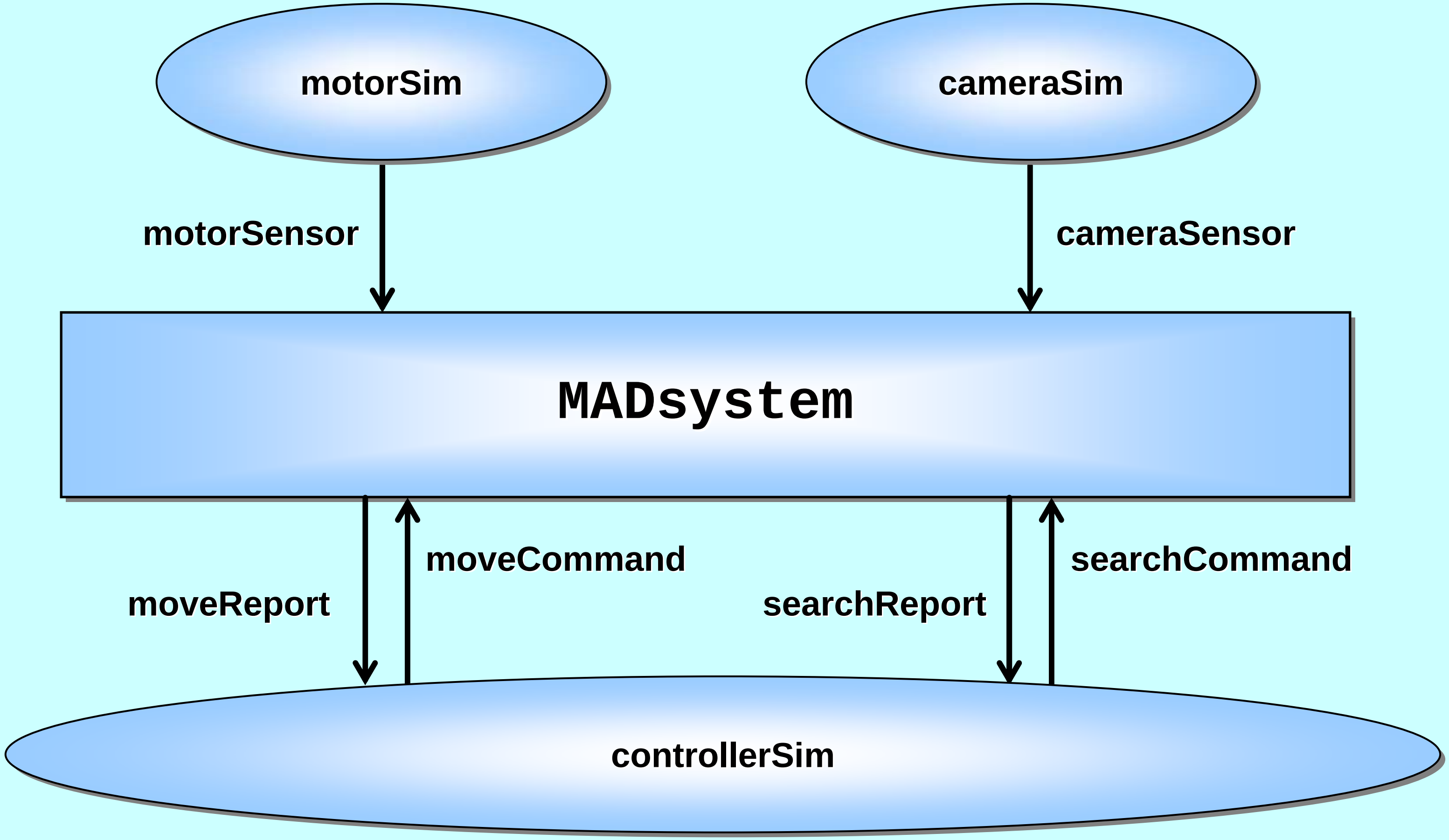

```
  INT x:
  sensor ? x
    SEQ
      ...  update local state with x (depends on mode)
      IF
        ...  task complete
          SEQ
            killYou ! kill
            report ! me
            running := FALSE
        TRUE
          SKIP
:
```




```
PROC MADsystem (CHAN INT moveCommand?, searchCommand?,
                CHAN INT motorSensor?, cameraSensor?,
                CHAN REPORT moveReport!, searchReport!)
  CHAN KILL a, b:
  PAR
    monitor (move, moveCommand?, motorSensor?,
             moveReport!, b!, a?)
    monitor (search, searchCommand?, cameraSensor?,
             searchReport!, a!, b?)
:
```

Soak Testing

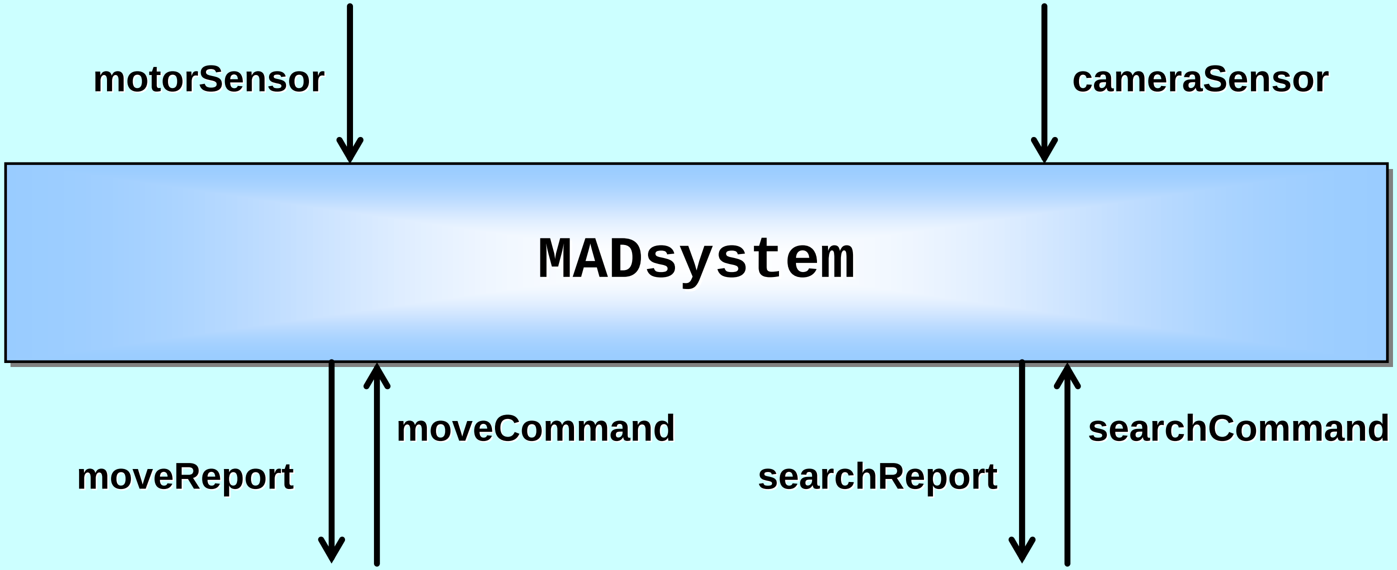
average sensor data interval = 10 ms (randomised)
average sensor inputs per service = 100 (randomised)



Ran for 30 days (approx. 2.5m trials): P A S S E D

In Service

average sensor data interval = 10 ms (varying)
average sensor inputs per service = 100 (varying)

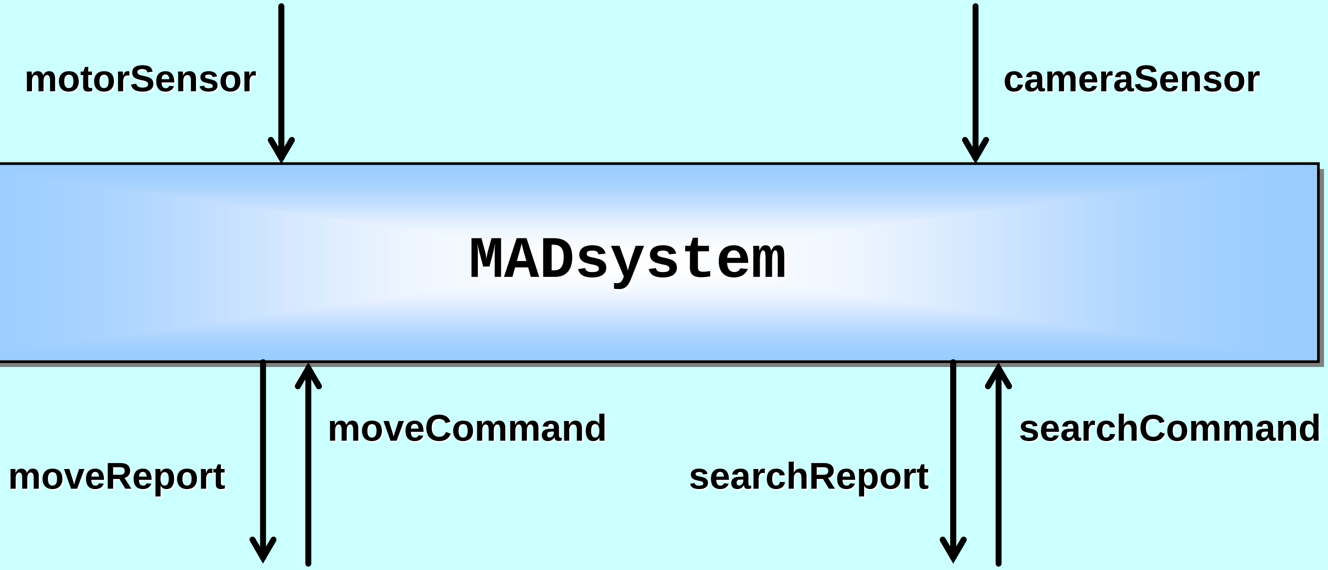


Ran for 2 years (approx. 64m trials): D E A D L O C K E D ☹ ☹ ☹

Should have asked for a model check …

```
VERIFY DEADLOCK.FREE MADsystem
```

✗



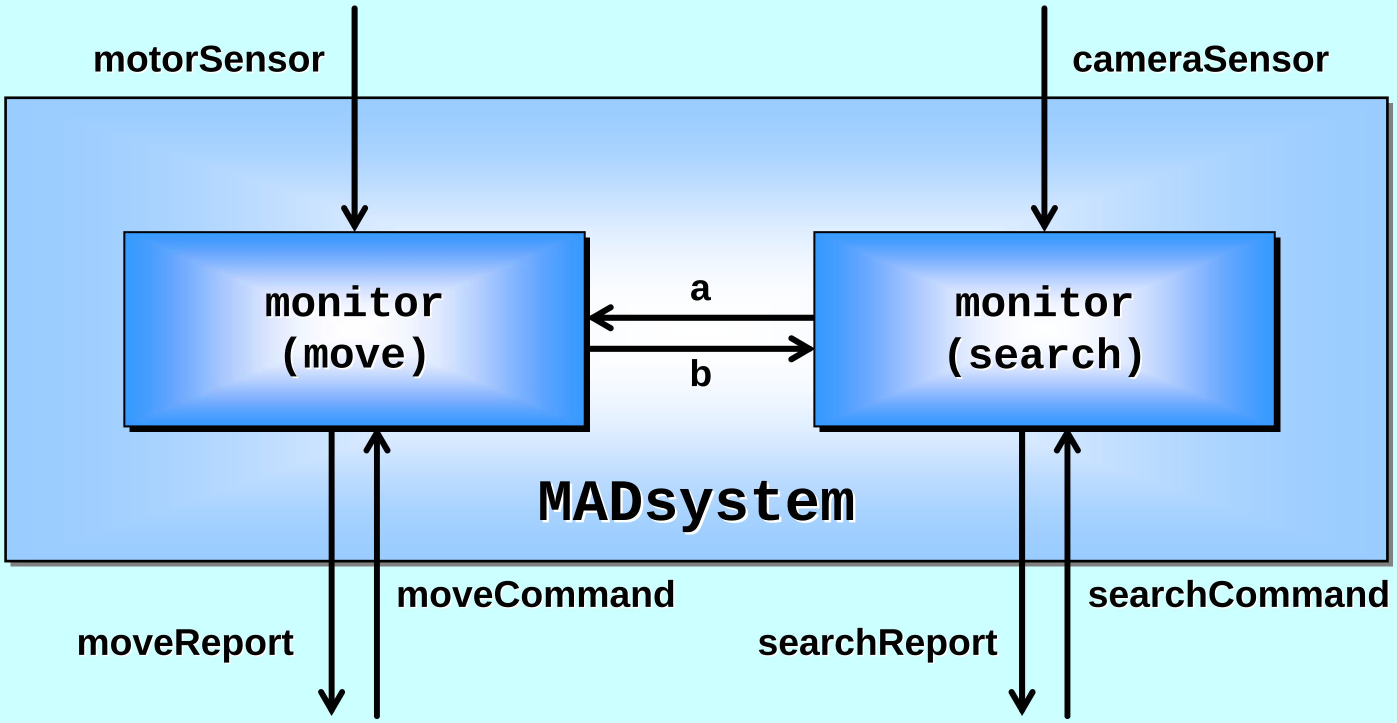
A trace leading to deadlock is provided:

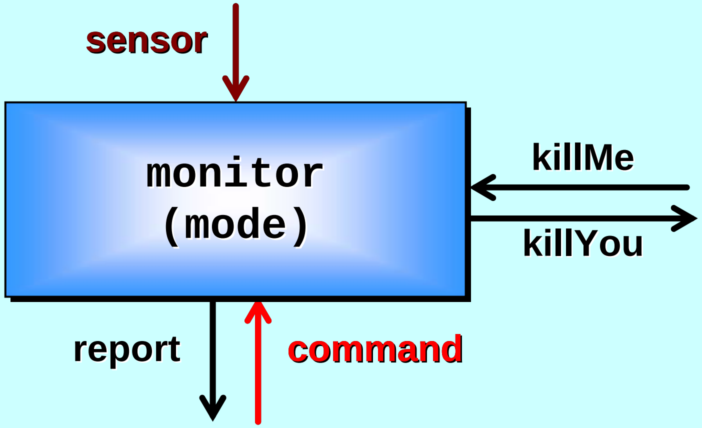
```
<moveCommand, motorSensor, searchCommand, cameraSensor>
```

Should have asked for a model check …

A trace leading to deadlock is provided:

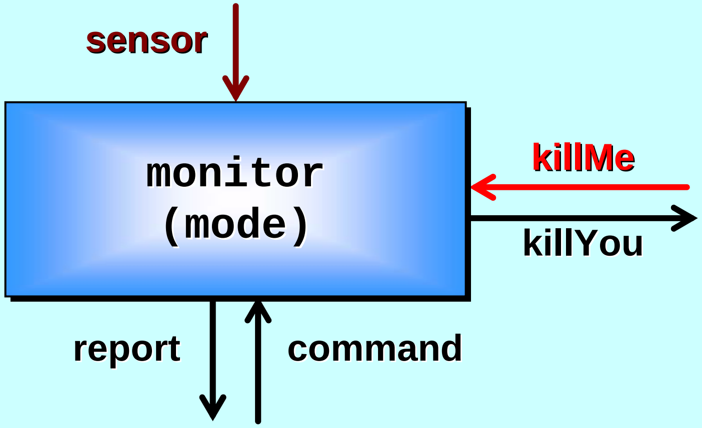
```
<moveCommand, motorSensor, searchCommand, cameraSensor>
```





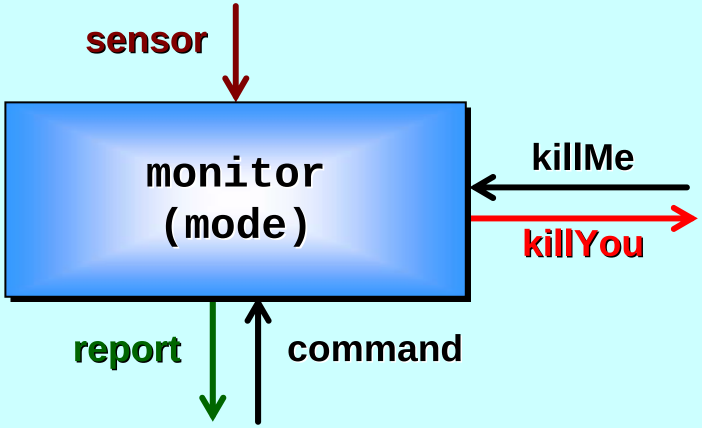
<moveCommand, motorSensor,
searchCommand, cameraSensor>

```
PROC monitor (VAL INT mode, CHAN INT command?, sensor?,
              CHAN REPORT report!,
              CHAN KILL killYou!, killMe?)
  WHILE TRUE
    PRI ALT
      INT target:
      command ? target                      -- service requested
        service (mode, target, sensor?, report!,
                 killYou!, killMe?)
      INT x:
      sensor ? x                            -- accept and discard
        SKIP
:
```



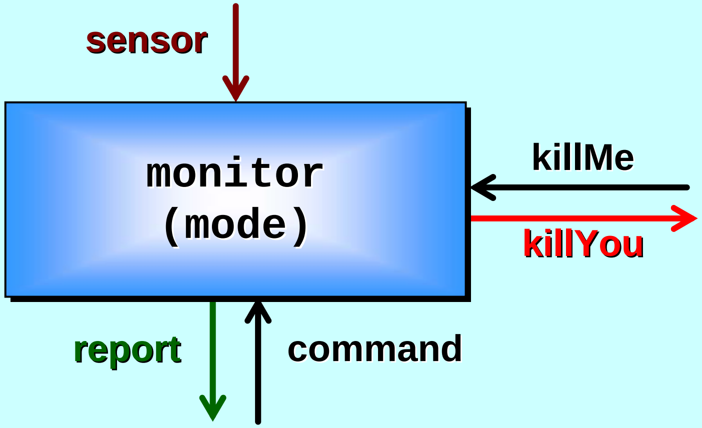
<moveCommand, motorSensor, searchCommand, cameraSensor>

```
PROC service (VAL INT mode, target, CHAN INT sensor?,
              CHAN REPORT report!,
              CHAN KILL killYou!, killMe?)
  ...  local state and initialisation
  INITIAL BOOL running IS TRUE:
  WHILE running
    PRI ALT
      killMe ? kill
        ...  report 'she' and exit loop
      INT x:
      sensor ? x
        ...  process x
:
```



<moveCommand, motorSensor, searchCommand, cameraSensor>

```
  INT x:
  sensor ? x
    SEQ
      ...  update local state with x (depends on mode)
      IF
        ...  task complete
          SEQ
            killYou ! kill
            report ! me
            running := FALSE
        TRUE
          SKIP
:
```



If the kill windows of the two monitors overlap, both will try to kill the other – resulting in deadlock.

Kill Window

```
INT x:
sensor ? x
  SEQ
    ...  update local state with x (depends on mode)
    IF
      ...  task complete
        SEQ
          killYou ! kill
          report ! me
          running := FALSE
      TRUE
        SKIP
:
```

average sensor data interval = 10 ms (randomised)
average sensor inputs per service = 100 (randomised)
➔
average service time = 1 second
kill window = 100 nanoseconds (approx.)
➔
chance of kill window overlap (deadlock) = 1/10,000,000
➔
time before 50% chance of deadlock = 90 days (approx.)

Kill Window

```
INT x:
sensor ? x
  SEQ
    ...  update local state with x (depends on mode)
    IF
      ...  task complete
        SEQ
          killYou ! kill
          report ! me
          running := FALSE
      TRUE
        SKIP
:
```

chance of kill window overlap (deadlock) = 1/10,000,000

This assumes each monitor runs on its own dedicated core …

CCSP multicore scheduler dynamically bacthes processes to cores …

If monitors are in the same batch, they will not deadlock …

chance of kill window overlap (deadlock) = 1/100,000,000 (approx.)

➔

time before 50% chance of deadlock = 2 years (approx.)

Kill Window

```
INT x:
sensor ? x
  SEQ
    ...  update local state with x (depends on mode)
    IF
      ...  task complete
        SEQ
          killYou ! kill
          report ! me
          running := FALSE
      TRUE
        SKIP
:
```

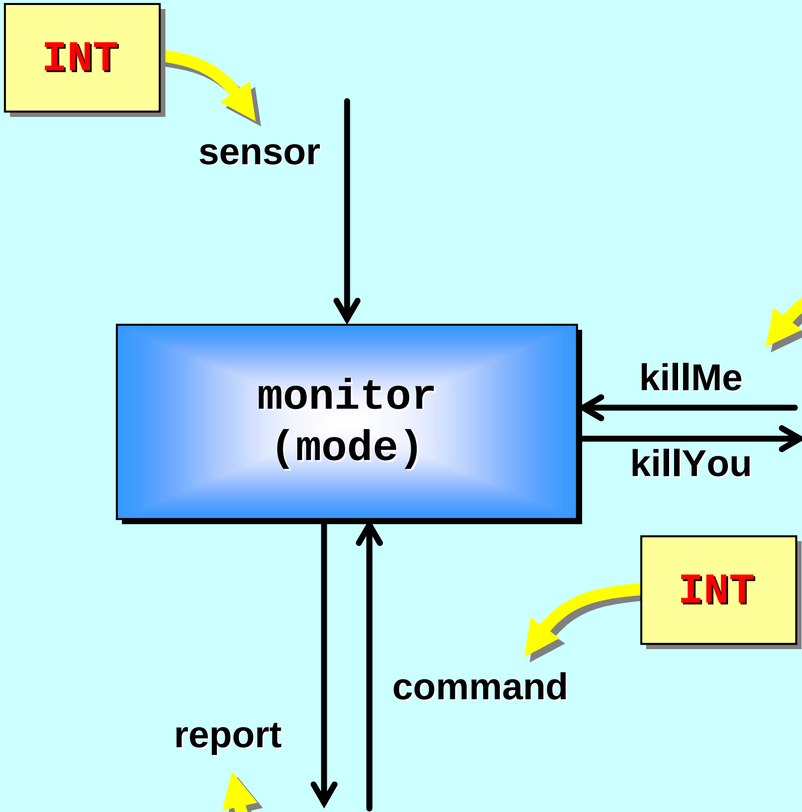
Mutually Assured Destruction (revised implementation)

Communication between the monitors is mostly a *one-way* kill (from killer to killed). Deadlock happens when both turn killer – *two-way* communications.

Idea: make communication between the monitors *always two-way* – either a **`kill`** in both directions (should both tasks complete around the same time) or a **`kill`** in one direction followed by an **`ack`** in the other (which will be most of the time).

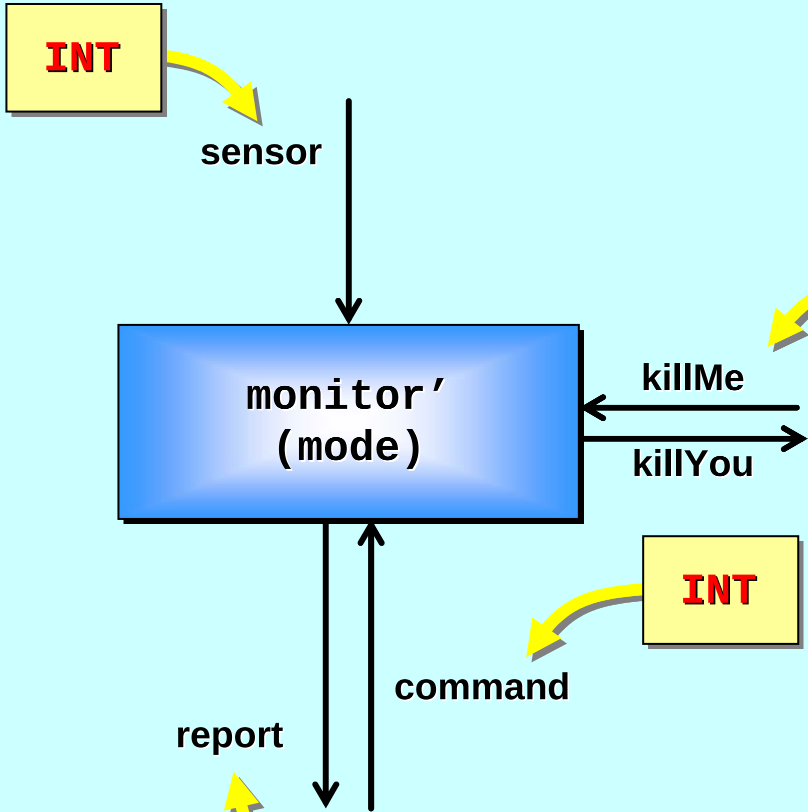
Claim: this eliminates all deadlock (at the cost of an extra **`ack`**).

Previously …



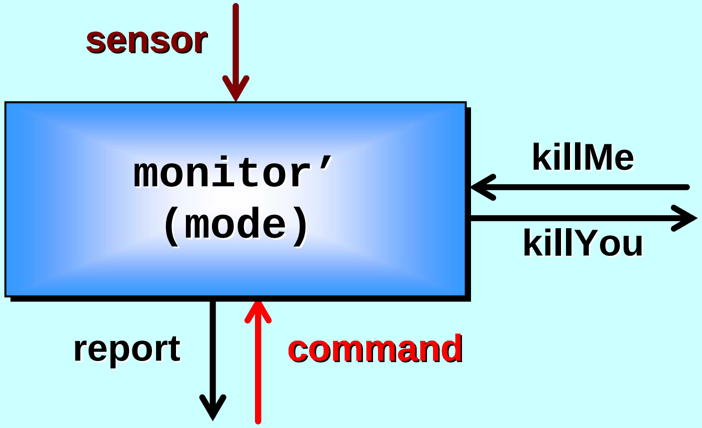

```
PROTOCOL KILL
  CASE
    kill
:
```

```
PROTOCOL REPORT
  CASE
    me              -- task completed
    she             -- task abandoned
:
```



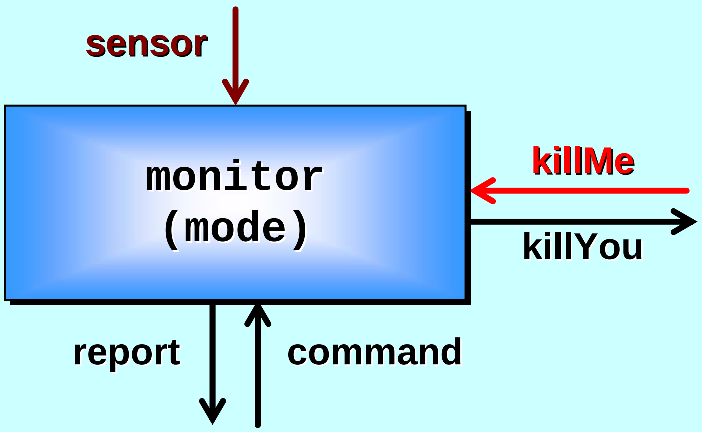

```
PROTOCOL KILL'
  CASE
    kill
    ack
:
```

```
PROTOCOL REPORT'
  CASE
    me            -- task completed
    she           -- task abandoned
    both          -- both completed
:
```

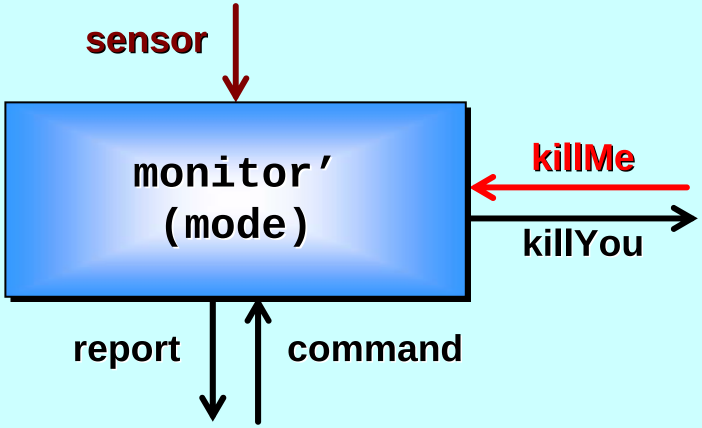



```
PROC monitor' (VAL INT mode, CHAN INT command?, sensor?,
               CHAN REPORT' report!,
               CHAN KILL' killYou!, killMe?)
  WHILE TRUE
    PRI ALT
      INT target:
      command ? target                      -- service requested
        service' (mode, target, sensor?, report!,
                  killYou!, killMe?)
      INT x:
      sensor ? x                            -- accept and discard
        SKIP
:
```

Previously …

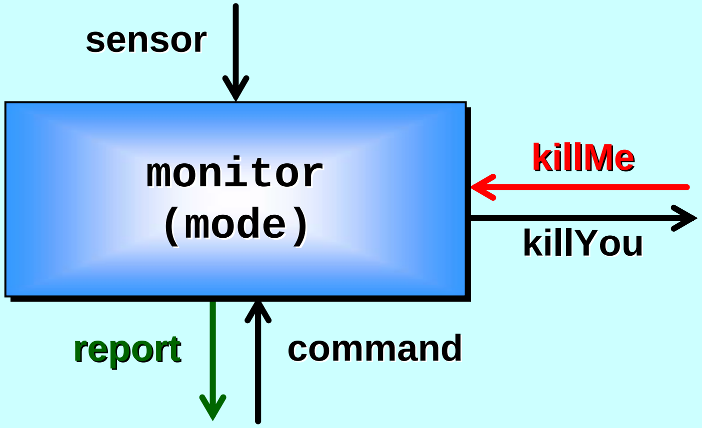



```
PROC service (VAL INT mode, target, CHAN INT sensor?,
              CHAN REPORT report!,
              CHAN KILL killYou!, killMe?)
  ...  local state and initialisation
  INITIAL BOOL running IS TRUE:
  WHILE running
    PRI ALT
      killMe ? kill
        ...  report 'she' and exit loop
      INT x:
      sensor ? x
        ...  process x
:
```

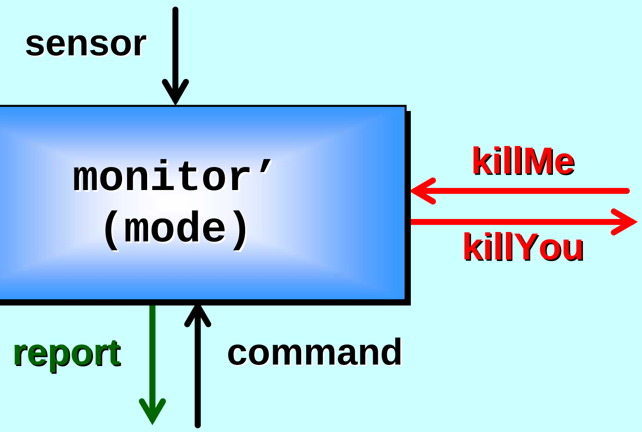



```
PROC service' (VAL INT mode, target, CHAN INT sensor?,
               CHAN REPORT' report!,
               CHAN KILL' killYou!, killMe?)
  ...  local state and initialisation
  INITIAL BOOL running IS TRUE:
  WHILE running
    PRI ALT
      killMe ? kill
        ... 'ack' the kill, report 'she' and exit loop
      INT x:
      sensor ? x
        ...  process x
:
```

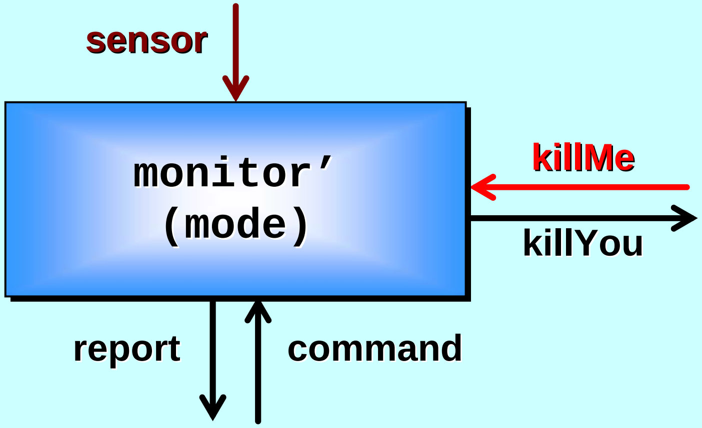
Previously …




```
killMe ? kill
  SEQ
    report ! she
    running := FALSE
```

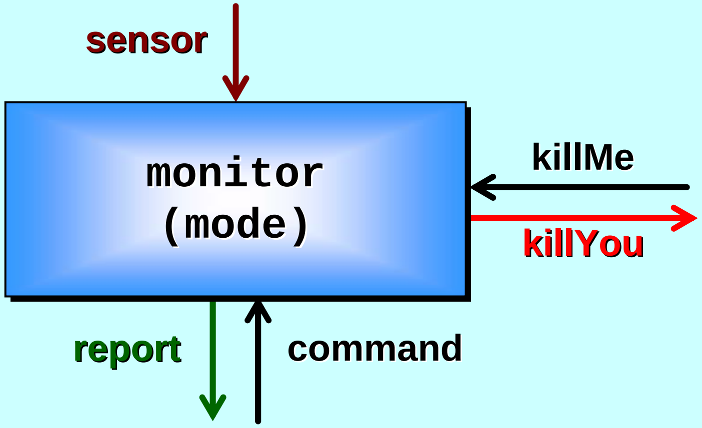



```
killMe ? kill
  SEQ
    killYou ! ack
    report ! she
    running := FALSE
```

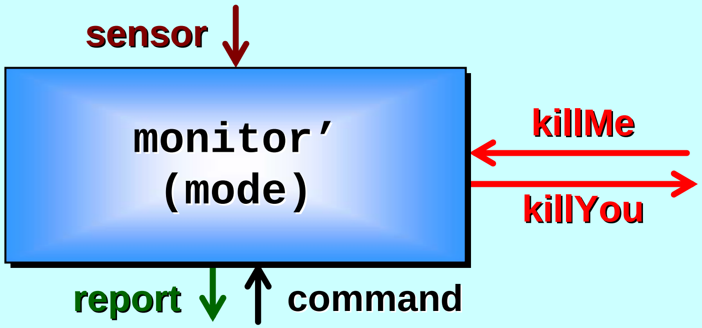



```
PROC service' (VAL INT mode, target, CHAN INT sensor?,
               CHAN REPORT' report!,
               CHAN KILL' killYou!, killMe?)
  ...  local state and initialisation
  INITIAL BOOL running IS TRUE:
  WHILE running
    PRI ALT
      killMe ? kill
        ... 'ack' the kill, report 'she' and exit loop
      INT x:
      sensor ? x
        ...  process x
:
```

Previously …

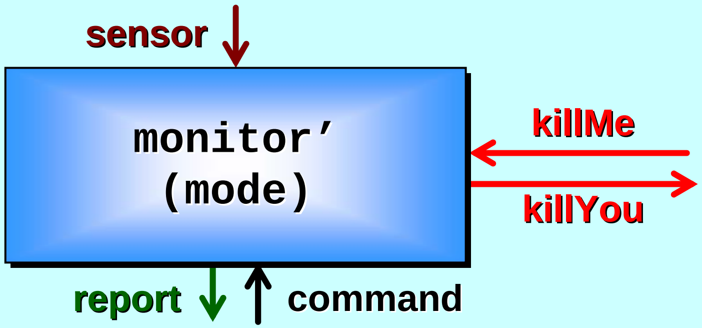



```
  INT x:
  sensor ? x
    SEQ
      ...  update local state with x (depends on mode)
      IF
        ...  task complete
          SEQ
            killYou ! kill
            report ! me
            running := FALSE
        TRUE
          SKIP
:
```



Each process knows what happened in the other – potentially a very useful side benefit.

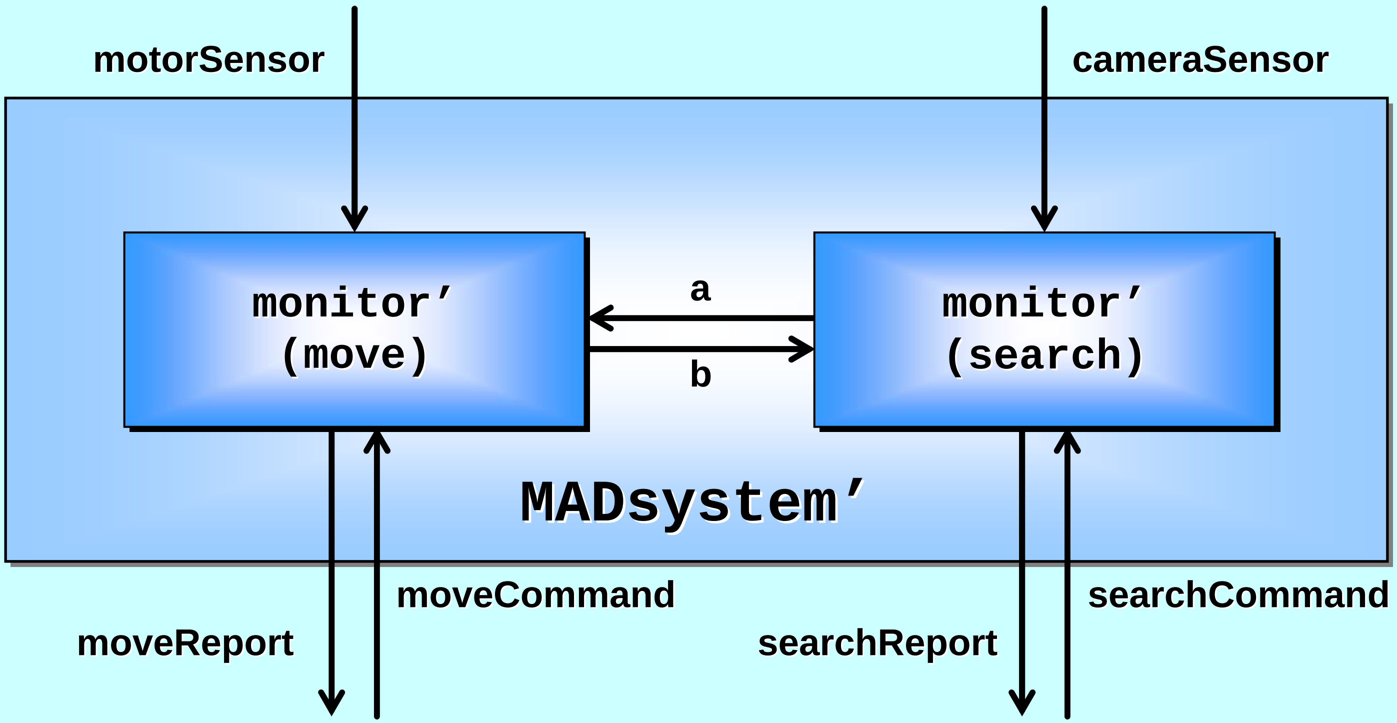
```
  INT x:
  sensor ? x
    SEQ
      ...  update local state with x (depends on mode)
      IF
        ...  task complete
          SEQ
            PAR
              killYou ! kill        -- send and
              killMe ? CASE         -- receive in parallel
                ack
                  report ! me
                kill
                  report ! both
            running := FALSE
        TRUE
          SKIP
:
```



Key state information could easily be piggy-backed on the **kill** and **ack** signals – i.e. each process would know what the other found.

```
    INT x:
    sensor ? x
      SEQ
        ...  update local state with x (depends on mode)
        IF
          ...  task complete
            SEQ
              PAR
                killYou ! kill        -- send and
                killMe ? CASE         -- receive in parallel
                  ack
                    report ! me
                  kill
                    report ! both
              running := FALSE
          TRUE
            SKIP
  :
```

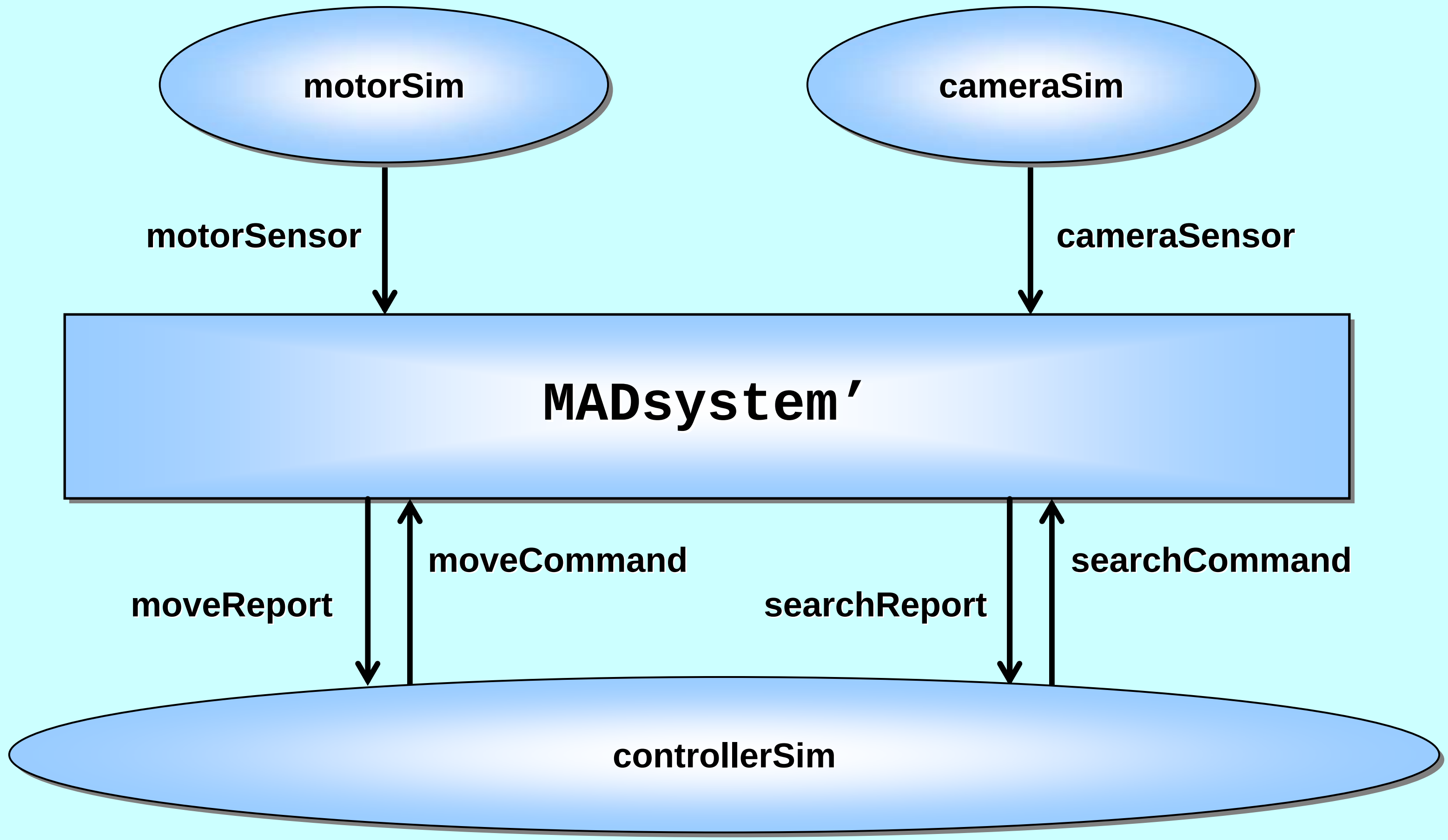
Better ask for a model check …



VERIFY DEADLOCK.FREE MADsystem’

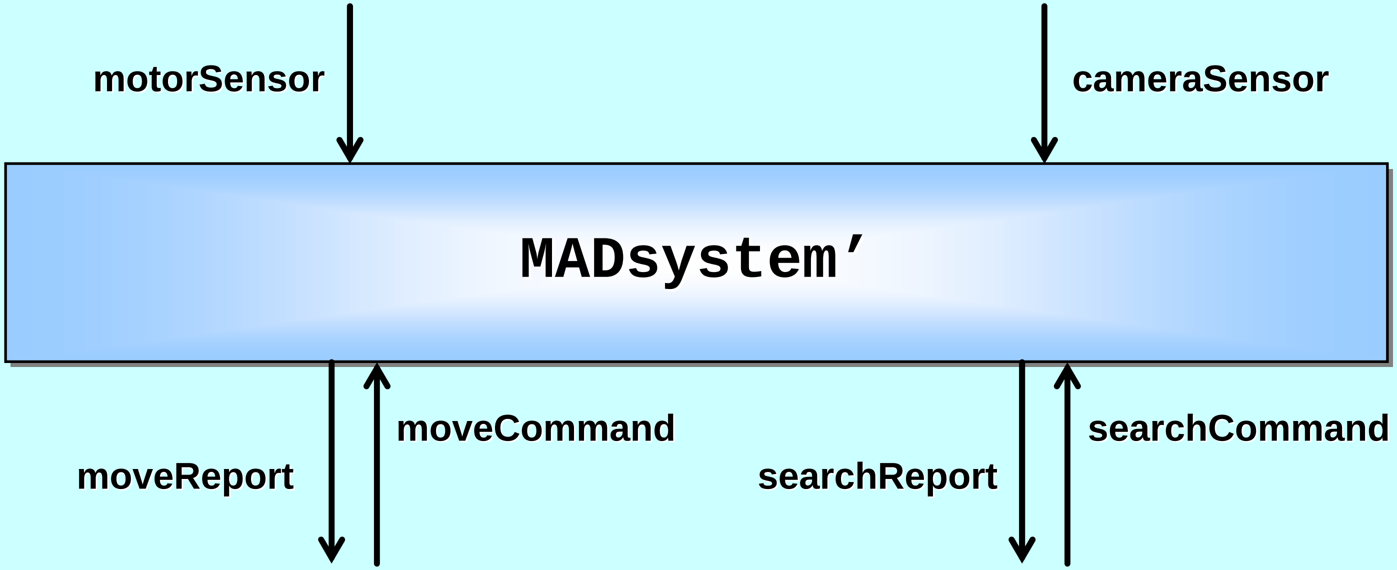
✓ ☺ ☺ ☺

Soak Testing



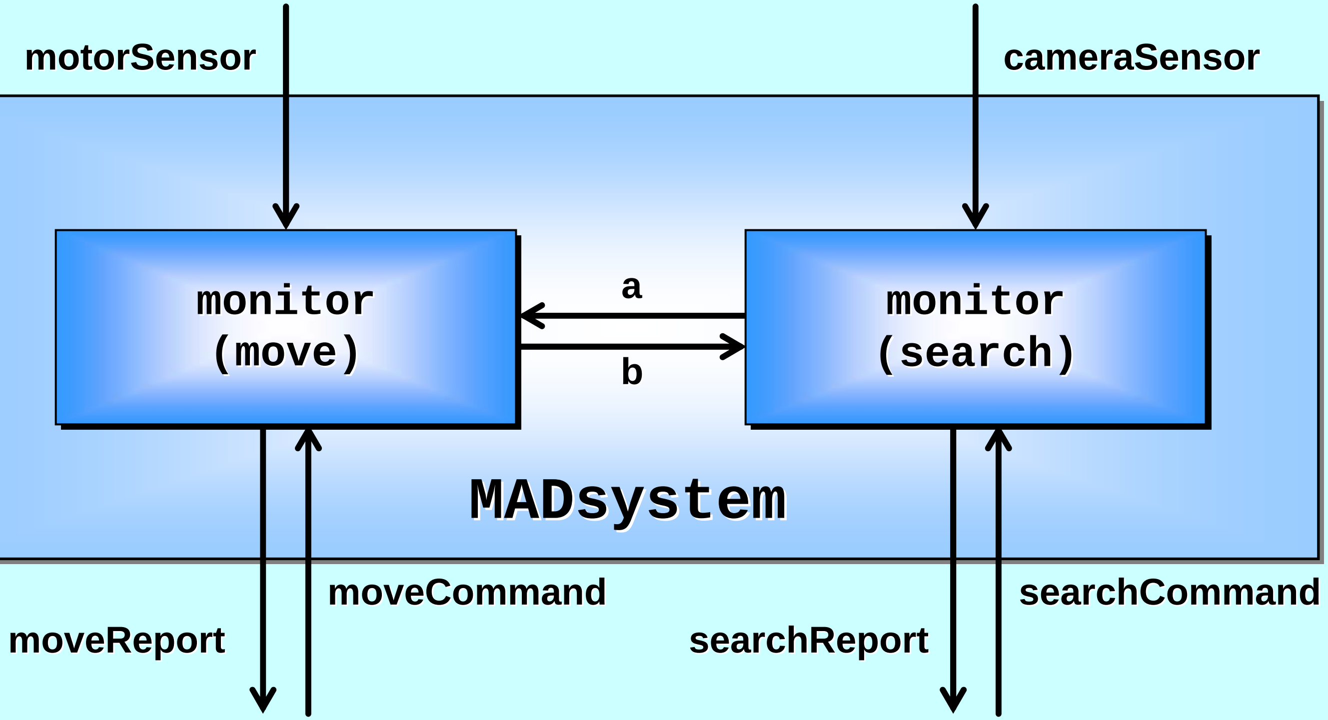
Only for confidence boosting – it will not deadlock (assuming compiler, run-time kernel, microprocessor are OK)

In Service



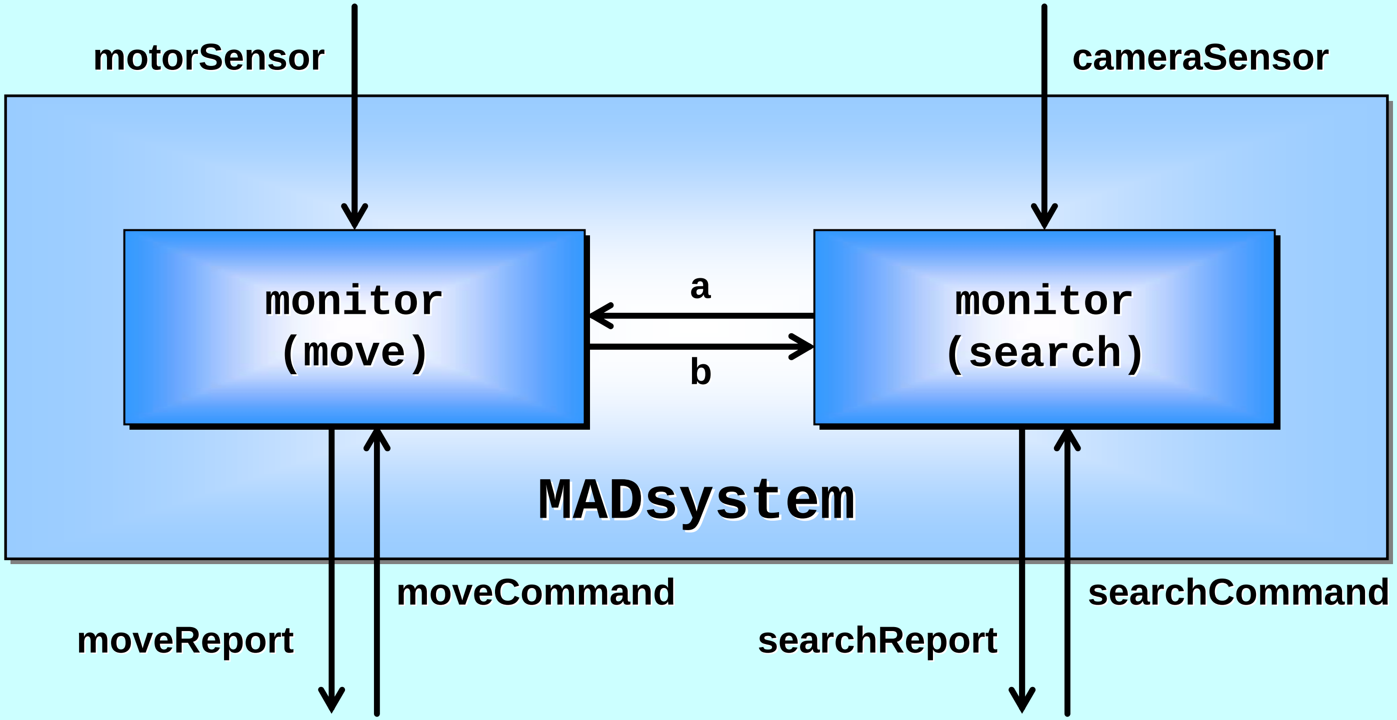
This will not deadlock
(assuming compiler, run-time kernel, microprocessor are OK)

Mutually Assured Destruction (asynchronous channels?)



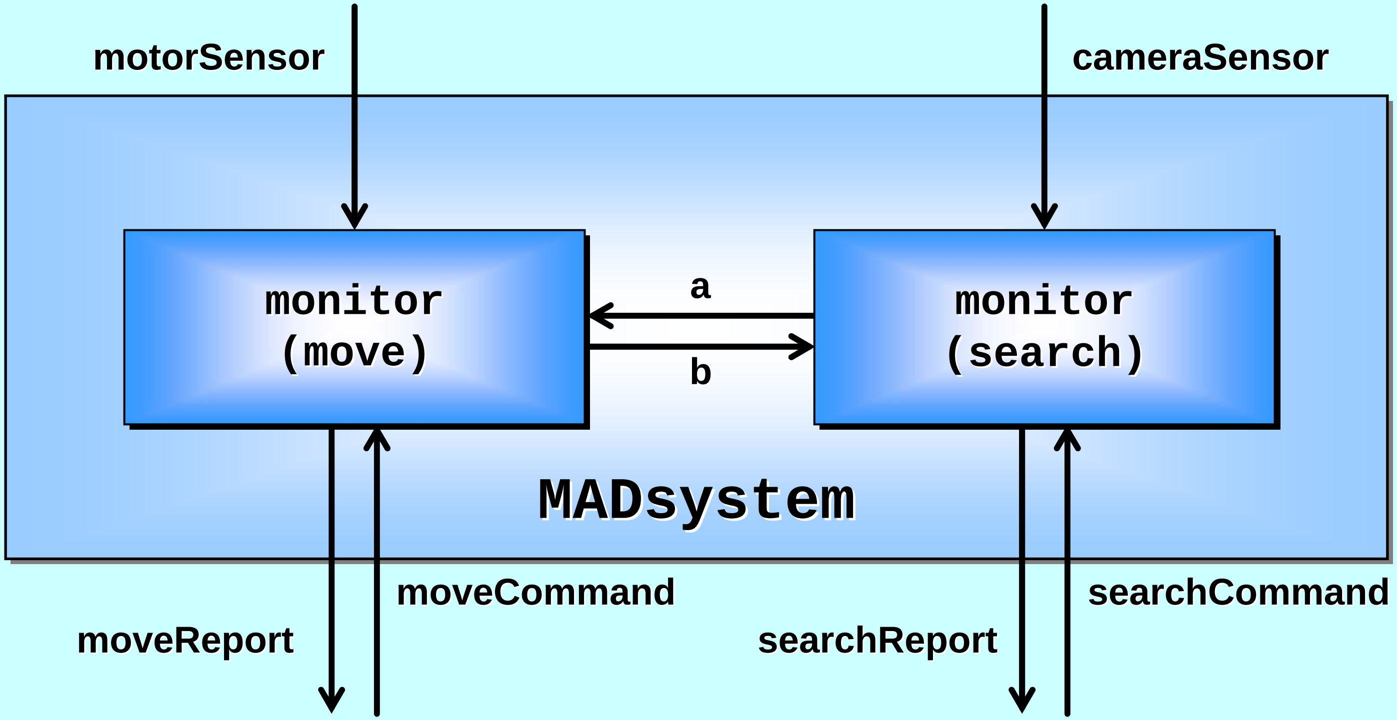
If the channels were finitely buffered (with capacity greater than zero), the deadlock found with synchronous (i.e. zero-buffered) channels would not happen – both monitors would complete their kills, reports and service routines.

Mutually Assured Destruction (asynchronous channels?)



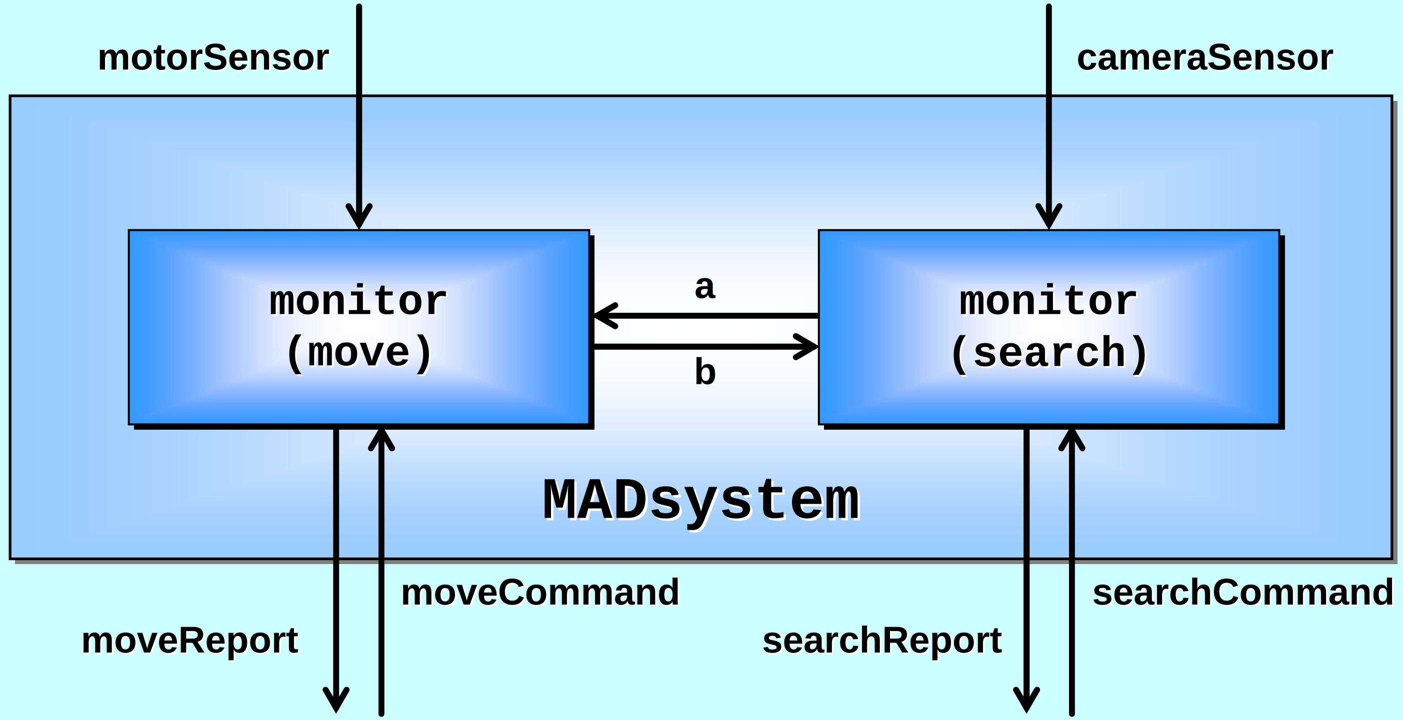
If the channels were finitely buffered, deadlock is still possible – but less likely (exponentially) with increasing buffer size. Infinitely expandable buffer capacity would be needed to eliminate deadlock from the basic algorithm. For practical purposes, I would feel safe with a capacity of 3.

Mutually Assured Destruction (asynchronous channels?)



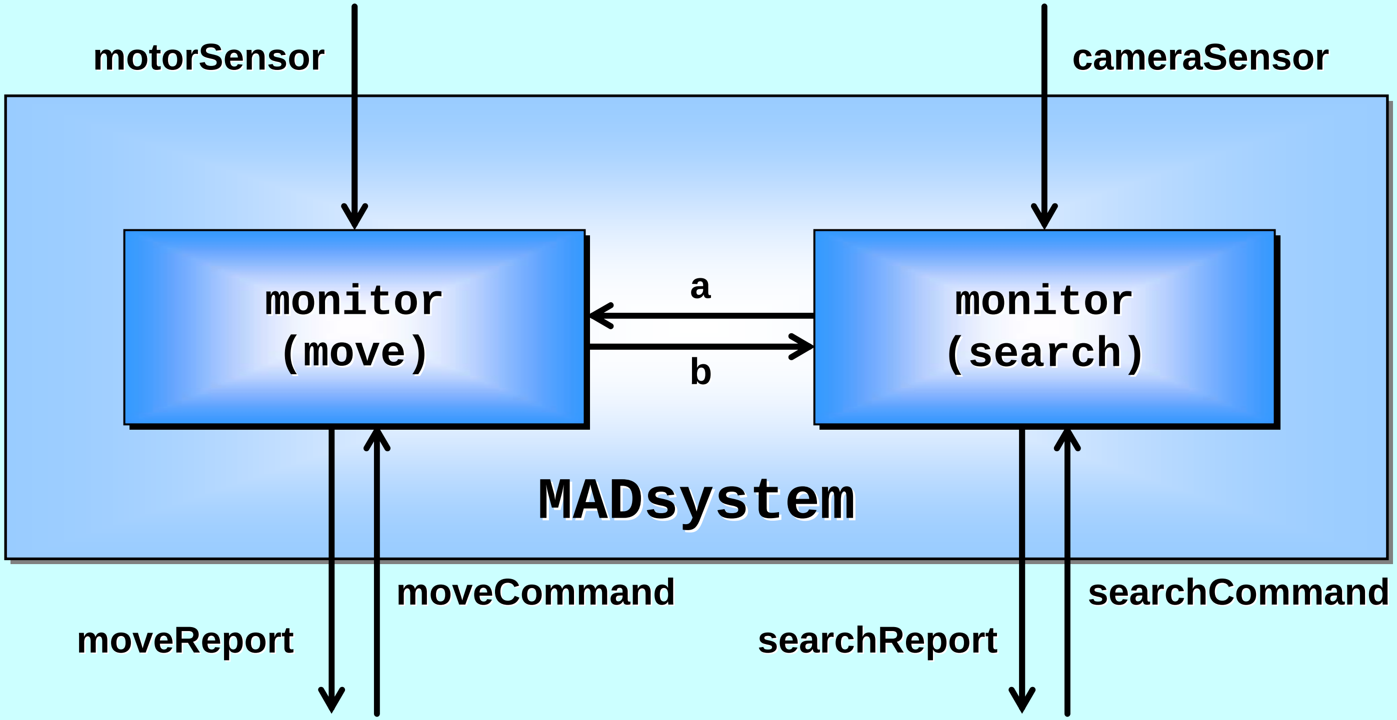
However, there is a nasty problem. If both monitors send a **`kill`**, neither is taken and they remain lurking in the buffered channels. Some time in the next service cycle, both will strike and the services will be erroneously aborted.

Mutually Assured Destruction (asynchronous channels?)



This could be overcome by counting cycles and *sequence numbering* the **`kill`** signals: just ignore any **`kill`** with a number less than the current count. This adds complexity and run-time overhead.

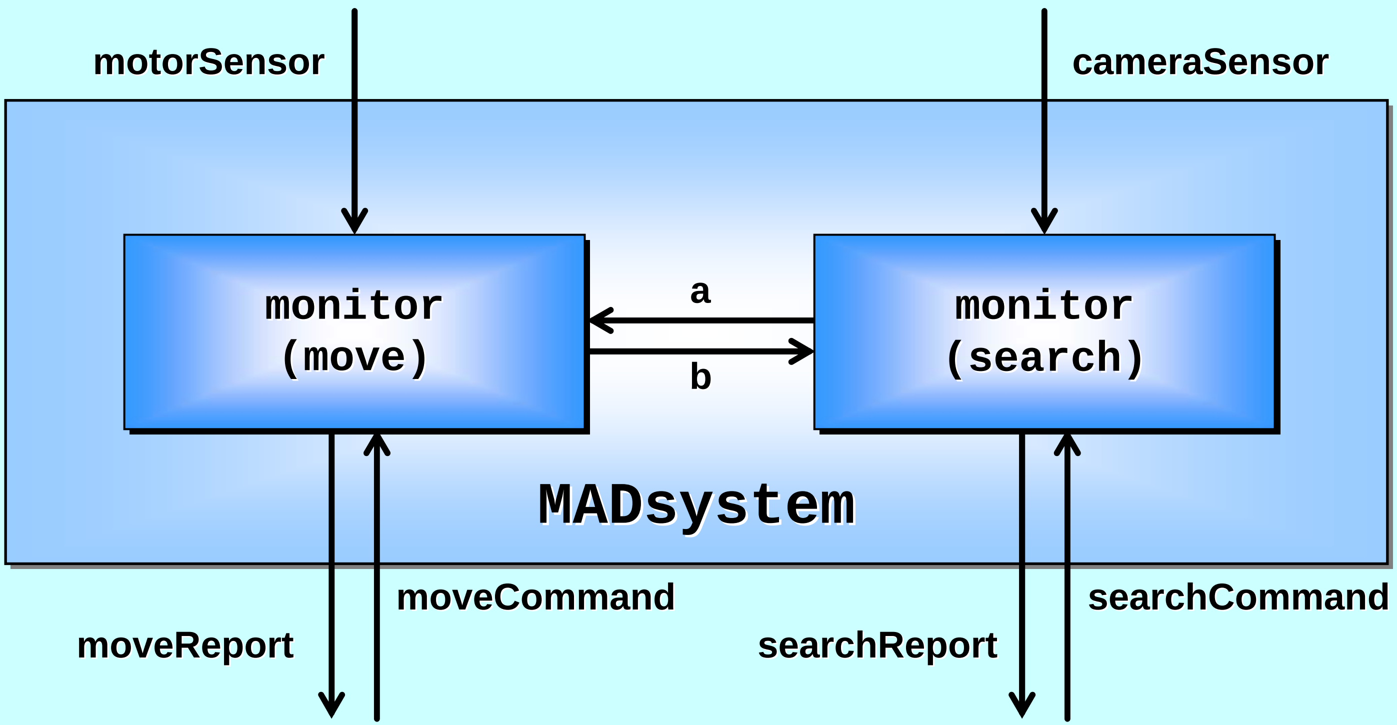
Mutually Assured Destruction (asynchronous channels?)



This could be overcome by counting cycles and *sequence numbering* the **`kill`** signals: just ignore any **`kill`** with a number less than the current count.

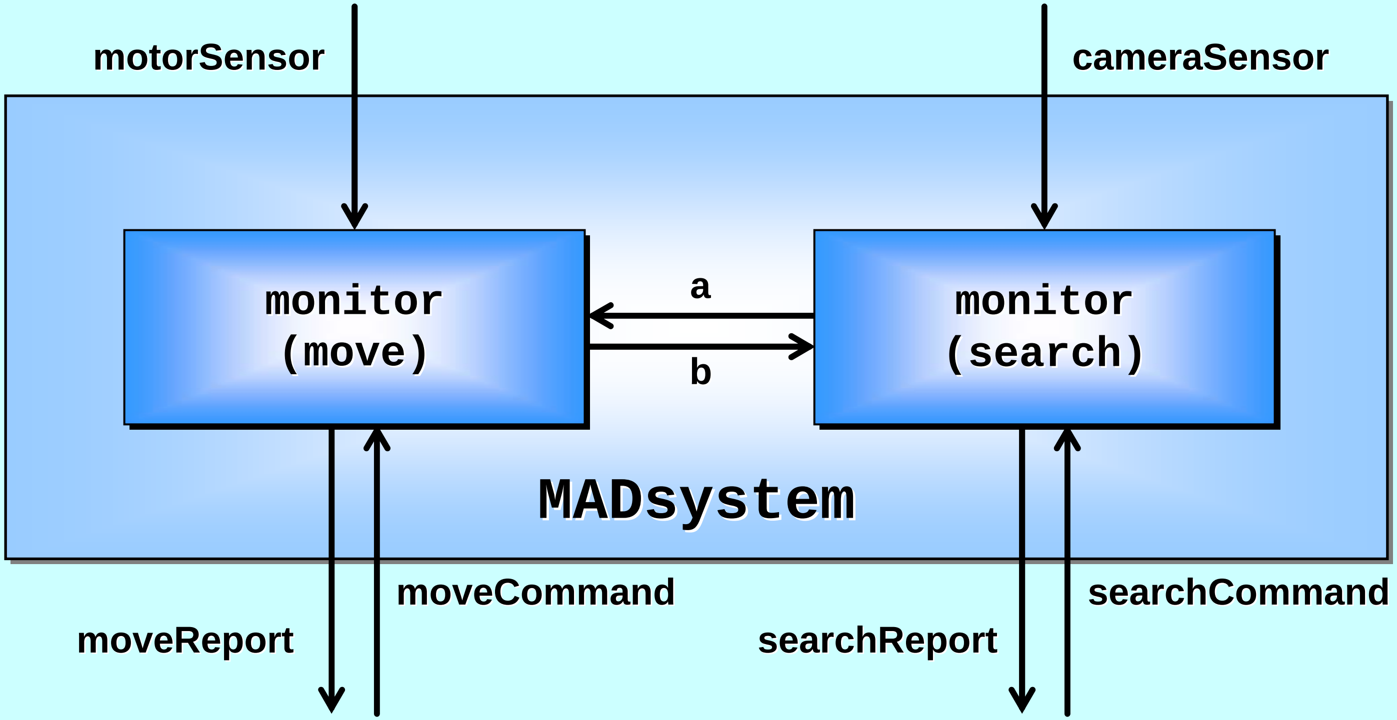
Further, this only works if the processes engaged in MAD are in lock-step (which they are in this scenario, but not in general).

Mutually Assured Destruction (asynchronous channels?)



Alternatively, the mess could be sorted out by the **Controller** process. When/if it gets two **me** reports from the monitors, it tells each monitor (as part of its next command) to read and discard an incoming **kill**. Again, this adds complexity – we shouldn't have a mess to clean up!

Mutually Assured Destruction (asynchronous channels?)



Alternatively, the mess could be sorted out by the **`Controller`** process. When/if it gets two **`me`** reports from the monitors, it tells each monitor (as part of its next command) to read and discard an incoming **`kill`**. Further, this assumes a **`Controller`**, which processes engaged in MAD may not have.

The Joy of Sync

Process oriented design ...

Synchronous communications …

Synchronous barriers …

Mutually assured destruction …

Non-blocking barriers …

Performance …

Non-Blocking Barriers

Recently (2012) introduced to MPI, *non-blocking barrier synchronisation* seems, at first glance, a contradiction of terms … the whole point of a *barrier* is to *block* until all parties are there!

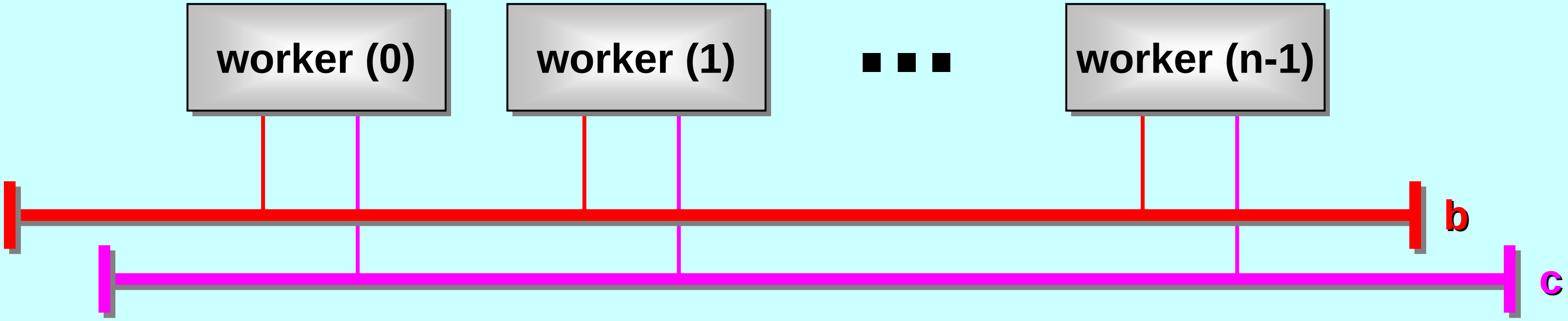
When we have completed our work before a *barrier*, we normally synchronise on it – thereby notifying that we are there and waiting for the others.

Recall …

Repeat .

Barriers

Processes may synchronise on more than one barrier:




```
BARRIER b, c:
PAR i = 0 FOR n ENROLL b, c
  worker (i, b, c)
```

To synchronise on a barrier:

`SYNC b` or `SYNC c`

Repeat .

Barriers

Barriers are commonly used to synchronise multiple ***phases*** of computation between a set of processes. Within each phase, other synchronisations (channel / barrier) may take place:

```
PROC worker (VAL INT id, BARRIER b, c)
  ...  local declarations / initialisation
  WHILE running
    SEQ
      SYNC b
      ...  observe neighbourhood phase
      SYNC c
      ...  change neighbourhood phase
:
```

All workers do this together – all see the same thing …

All workers do this together – may need to negotiate …

Non-Blocking Barriers

Recently (2012) introduced to MPI, *non-blocking barrier synchronisation* seems, at first glance, a contradiction of terms … the whole point of a *barrier* is to *block* until all parties are there!

When we have completed our work before a *barrier*, we normally synchronise on it – thereby notifying that we are there and waiting for the others.

However, if there is something we can be getting on with that does not disturb our fellow *synchronisers,* (e.g. preparatory work for the phase following the *barrier*), it would be good to be able to do so. Only when we need stuff that depends on the other *synchronisers,* should we have to wait for them.

Blocking Barrier Sync (MPI)

```
...   phase 0 computation
MPI_Barrier (b, ...);                    // wait for everyone ...
...   preparatory work for next phase
...   phase 1 computation
```

Blocking Barrier Sync (occam-π)

```
SEQ
  ...   phase 0 computation
  SYNC b                                 -- wait for everyone ...
  ...   preparatory work for next phase
  ...   phase 1 computation
```

Non-Blocking Barrier Sync (MPI)

```
...  phase 0 computation
MPI_IBarrier (b, ...);              // hey, I'm done ...
...  preparatory work for next phase
MPI_WBarrier (b, ...);              // I'm waiting now ...
...  phase 1 computation
```

Non-Blocking Barrier Sync (o-π)

```
SEQ
  ...  phase 0 computation
  PAR
    SYNC b                          -- hey, I'm done ...
    ...  preparatory work for next phase
  ...  phase 1 computation
```

Non-Blocking Barrier Sync (o-π)

```
SEQ
  ...  phase 0 computation
  PAR
    SYNC b                                    -- hey, I'm done ...
    ...  preparatory work for next phase
  ...  phase 1 computation
```

The **SYNC** registers that we have arrived at the barrier and lets all move forward when the rest arrive. In parallel with the above, we get on with our ***preparatory work***. ☺

When our ***preparatory work*** is complete, if all the others have reached the barrier, the **SYNC** will have completed – so the **PAR** completes and we immediately move on to **phase 1**. And we have not delayed the others. ☺

When our ***preparatory work*** is complete, if not all the others have reached the barrier, the **SYNC** will not have completed. We wait for the others at the **SYNC** before moving on to **phase 1** – as we must! ☺

Non-Blocking Barrier Sync (o-π)

```
SEQ
  ...  phase 0 computation
  PAR
    SYNC b                                -- hey, I'm done ...
    ...  preparatory work for next phase
  ...  phase 1 computation
```

The **SYNC** registers that we have arrived at the barrier and lets all move forward when the rest arrive. In parallel with the above, we get on our ***preparatory work***. ☺

When our ***preparatory work*** is co… …e others have reached the barrier, the **SYNC** will h… …so the **PAR** completes and we immediately mo… …1. And we have not delayed the others. ☺

Wh… …eparatory work*** is complete, if not all the others have reac… …ed the barrier, the **SYNC** will not have completed. We wait for the others at the **SYNC** before moving on to **phase 1** – as we must! ☺

Nothing new in occam-π is needed for this.

The Joy of Sync

Process oriented design ...

Synchronous communications …

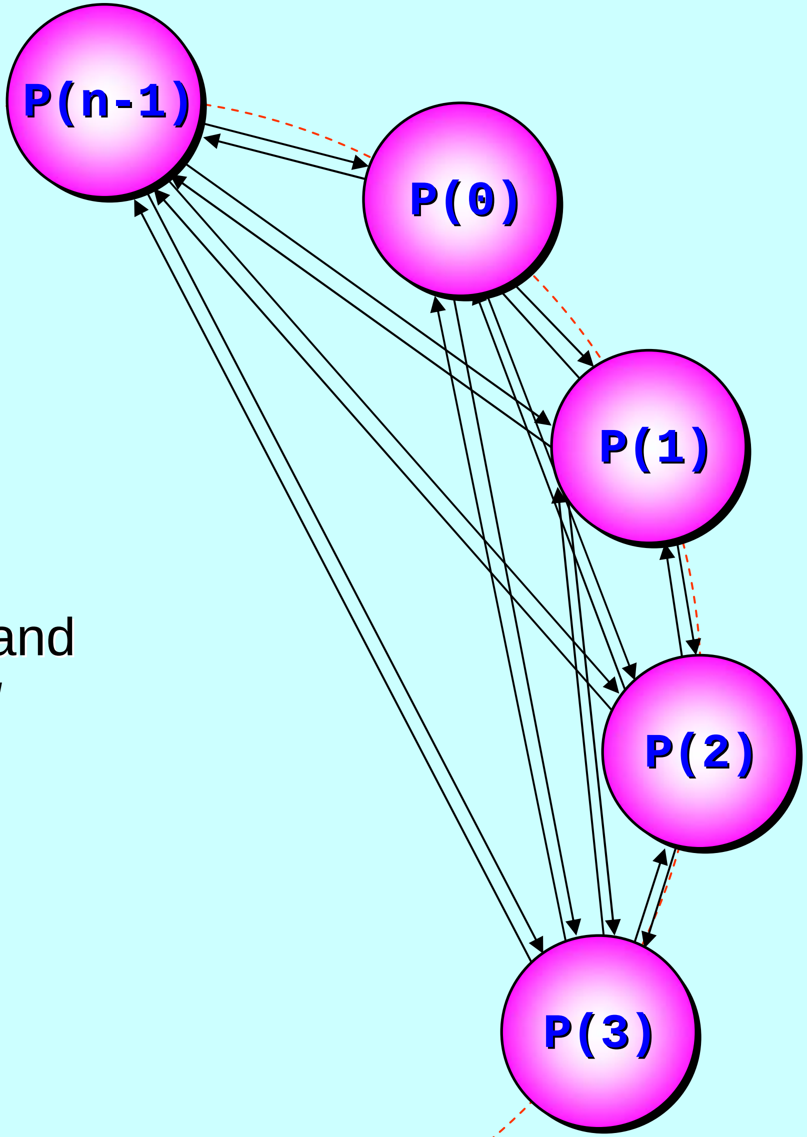
Synchronous barriers …

Mutually assured destruction …

Non-blocking barriers …

Performance …

Performance



Take a ring of n processes …

Each process connects to all …

In parallel, each process sends and receives m messages (e.g. its *id number*) to all, including itself …

That's mn^2 messages …

How long per message?

Performance

* A. Bate: "Scalable Performance for Scala Message-Passing Concurrency", CPA 2013, pp. 113-132, Open Channel Publishing.

Andrew's "Say Hello to Everyone" Benchmark (v1) *

$2n^2$ processes

mn^2 messages

```
[n][n]CHAN INT c:

PAR i = 0 FOR n
  PAR
    PAR j = 0 FOR n          -- for each j in parallel,
      SEQ k = 0 FOR m          -- send m messages (i to j)
        c[i][j] ! i
    PAR j = 0 FOR n          -- for each j in parallel,
      SEQ k = 0 FOR m          -- receive m messages (j to i)
        INT x:
        SEQ
          c[j][i] ? x
          ASSERT (x = j)    -- sanity check
```

Performance

* A. Bate: "Scalable Performance for Scala Message-Passing Concurrency", CPA 2013, pp. 113-132, Open Channel Publishing.

Andrew's "Say Hello to Everyone" Benchmark (v2) *

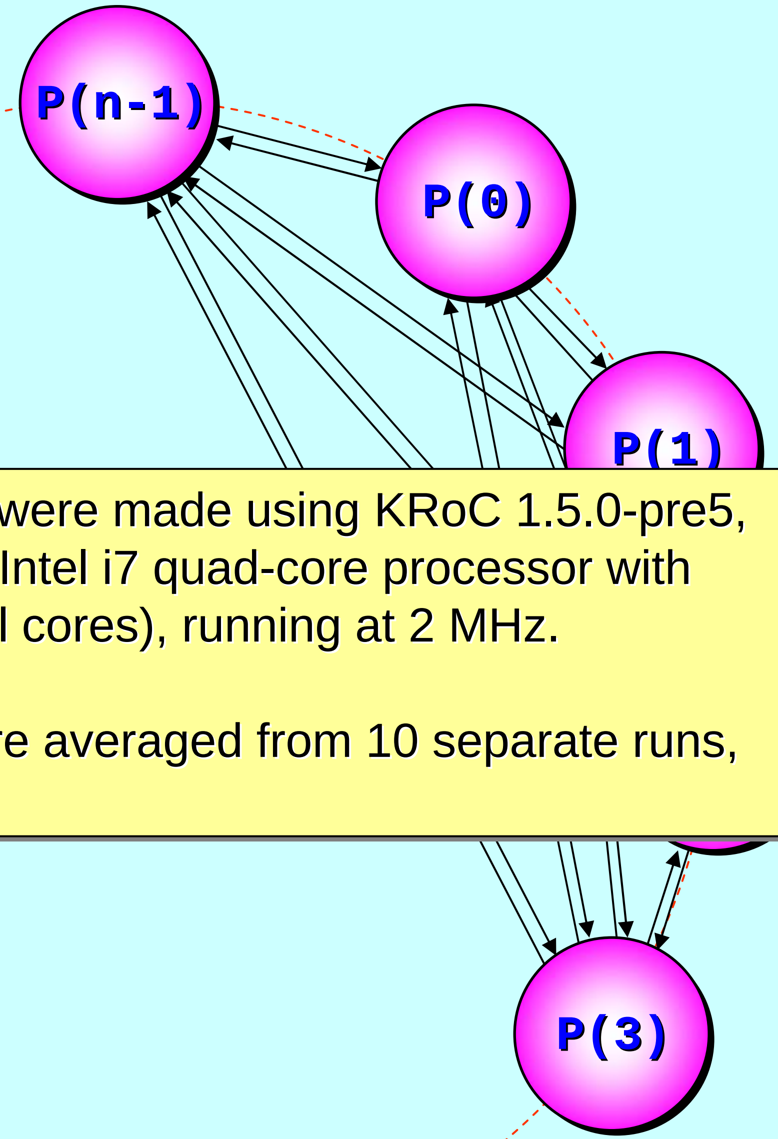
2n processes

mn^2 messages

```
[n][n]CHAN INT c:

PAR i = 0 FOR n
  PAR
    SEQ j = 0 FOR n            -- for each j in series,
      SEQ k = 0 FOR m            -- send m messages (i to j)
        c[i][j] ! i
    SEQ j = 0 FOR n            -- for each j in series,
      SEQ k = 0 FOR m            -- receive m messages (j to i)
        INT x:
        SEQ
          c[j][i] ? x
          ASSERT (x = j)         -- sanity check
```

Performance



Take a ring of N processes …

Each p

In para
and re
its id)

The following observations were made using KRoC 1.5.0-pre5, Ubuntu 11.04 (natty) on an Intel i7 quad-core processor with hyperthreading (i.e. 8 virtual cores), running at 2 MHz.

The benchmark timings were averaged from 10 separate runs, with negligible variance.

That's mn^2 messages …

How long per message?

Performance

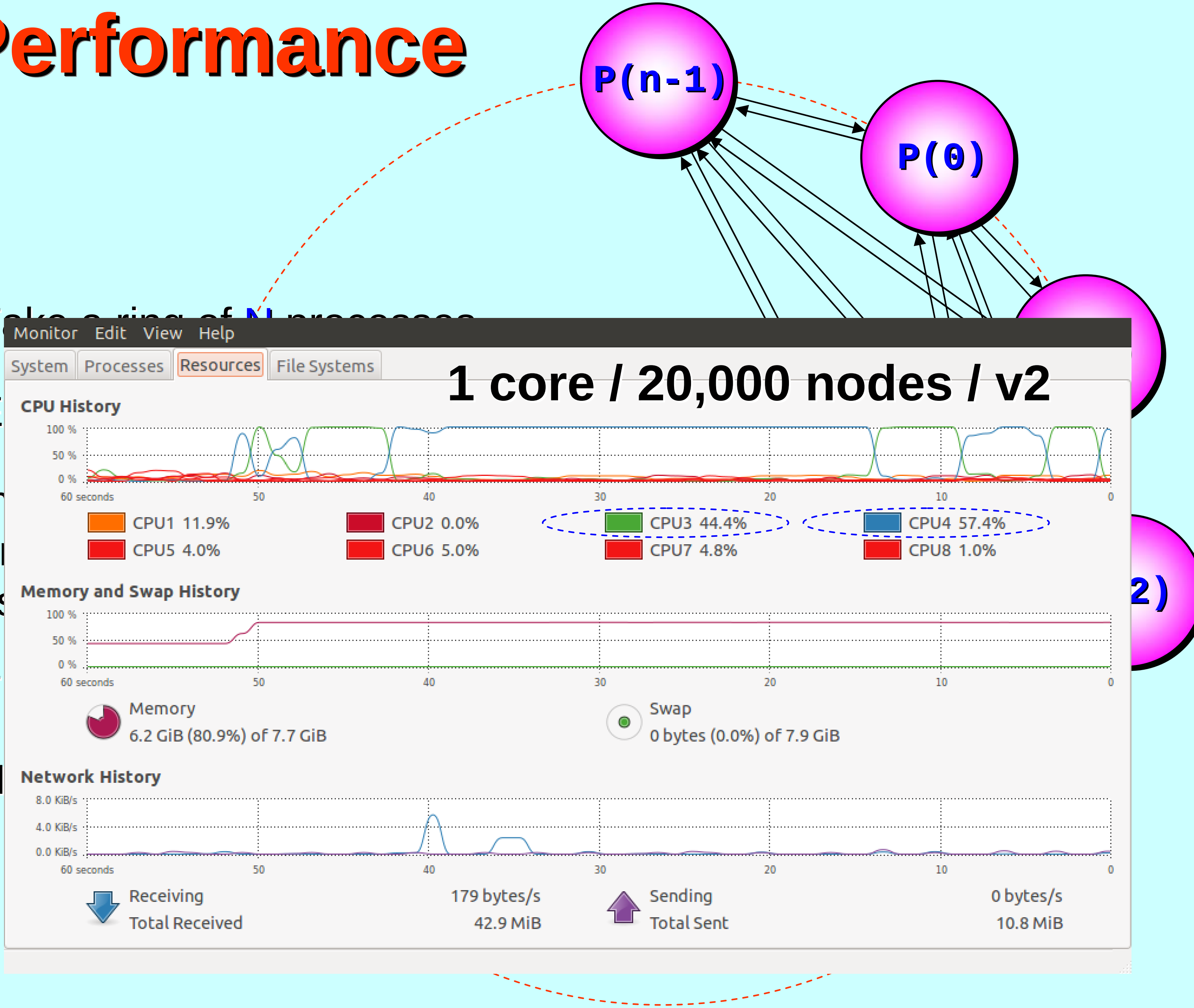
T

E

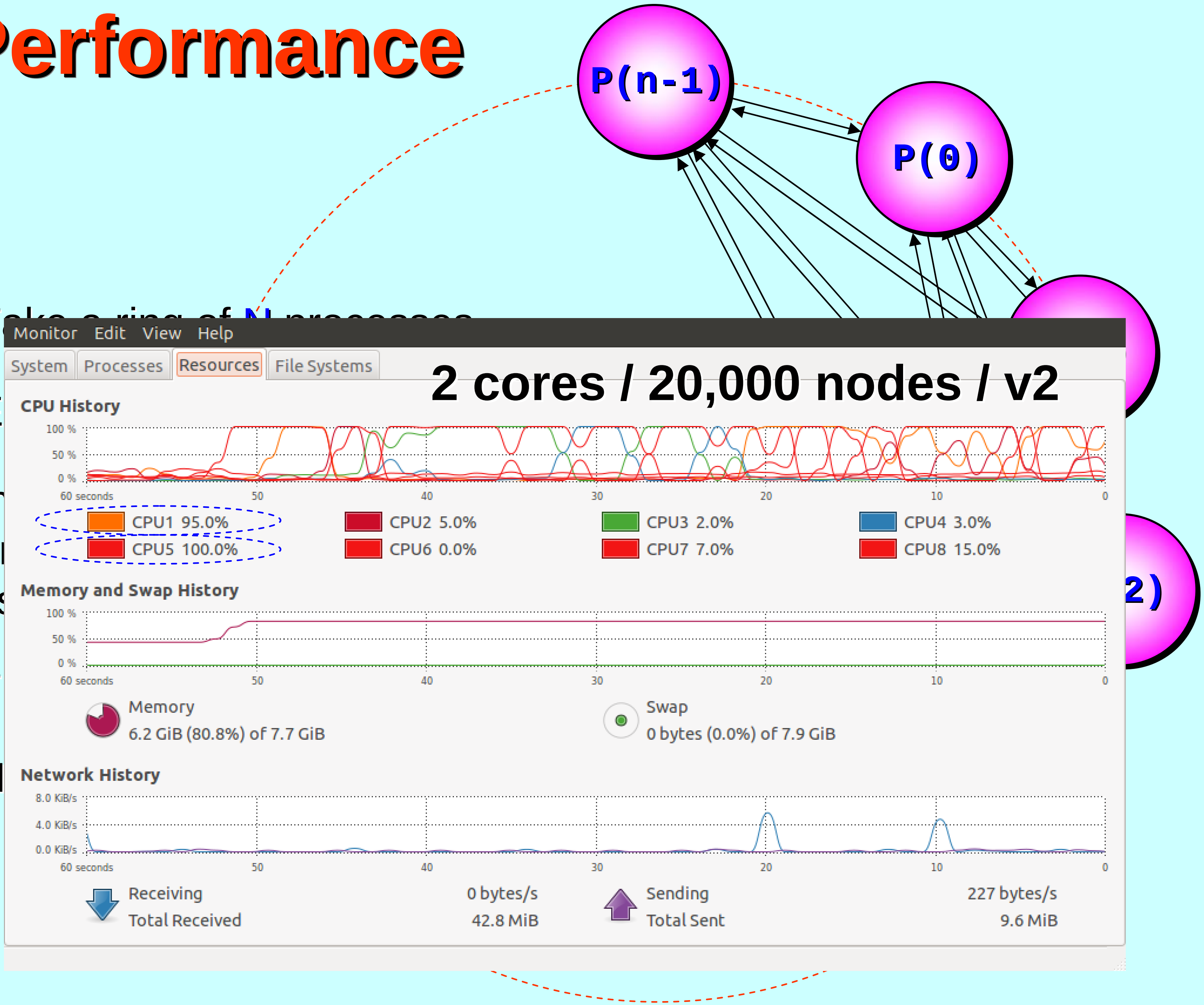
In
a
its

T

H

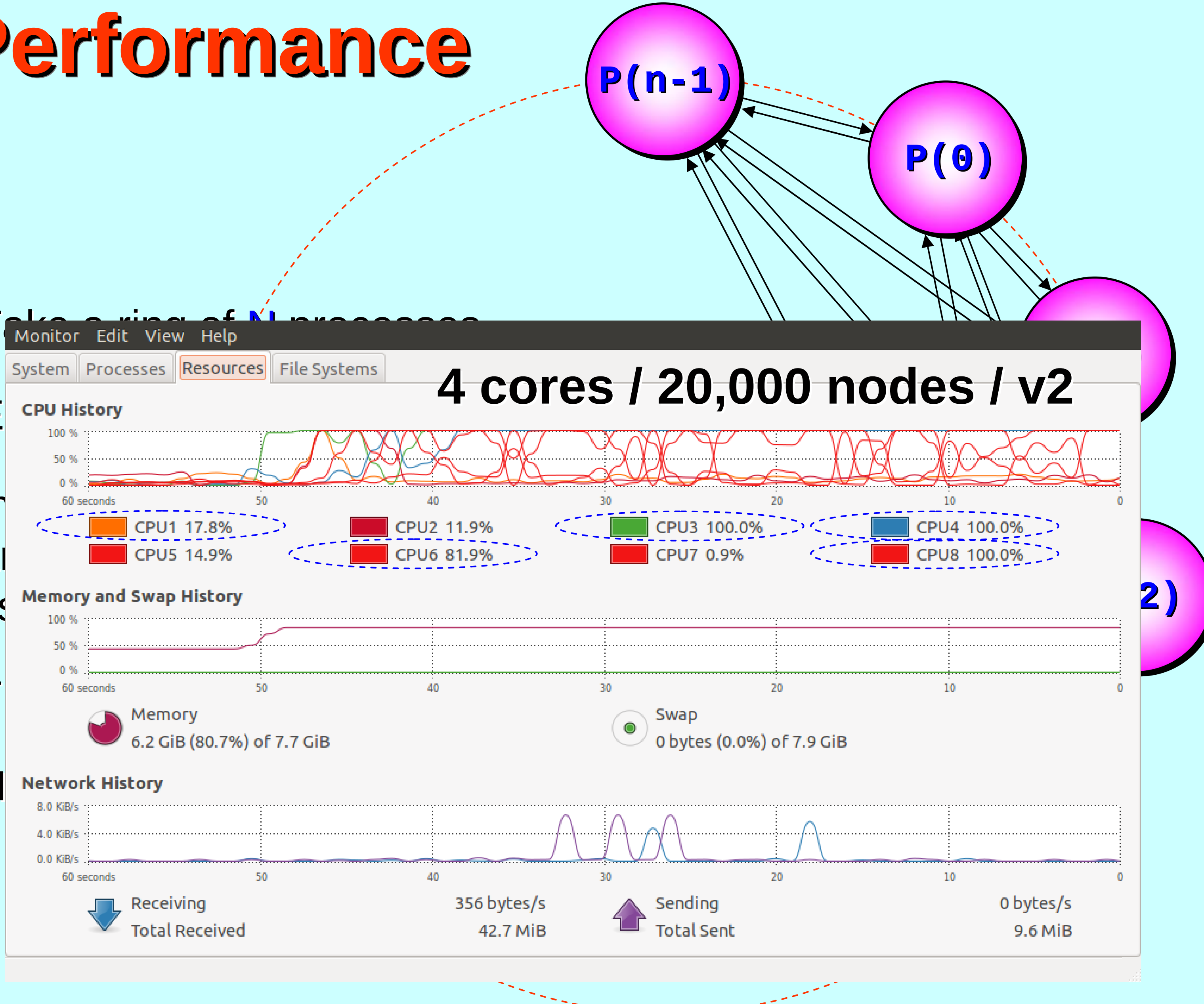


Performance

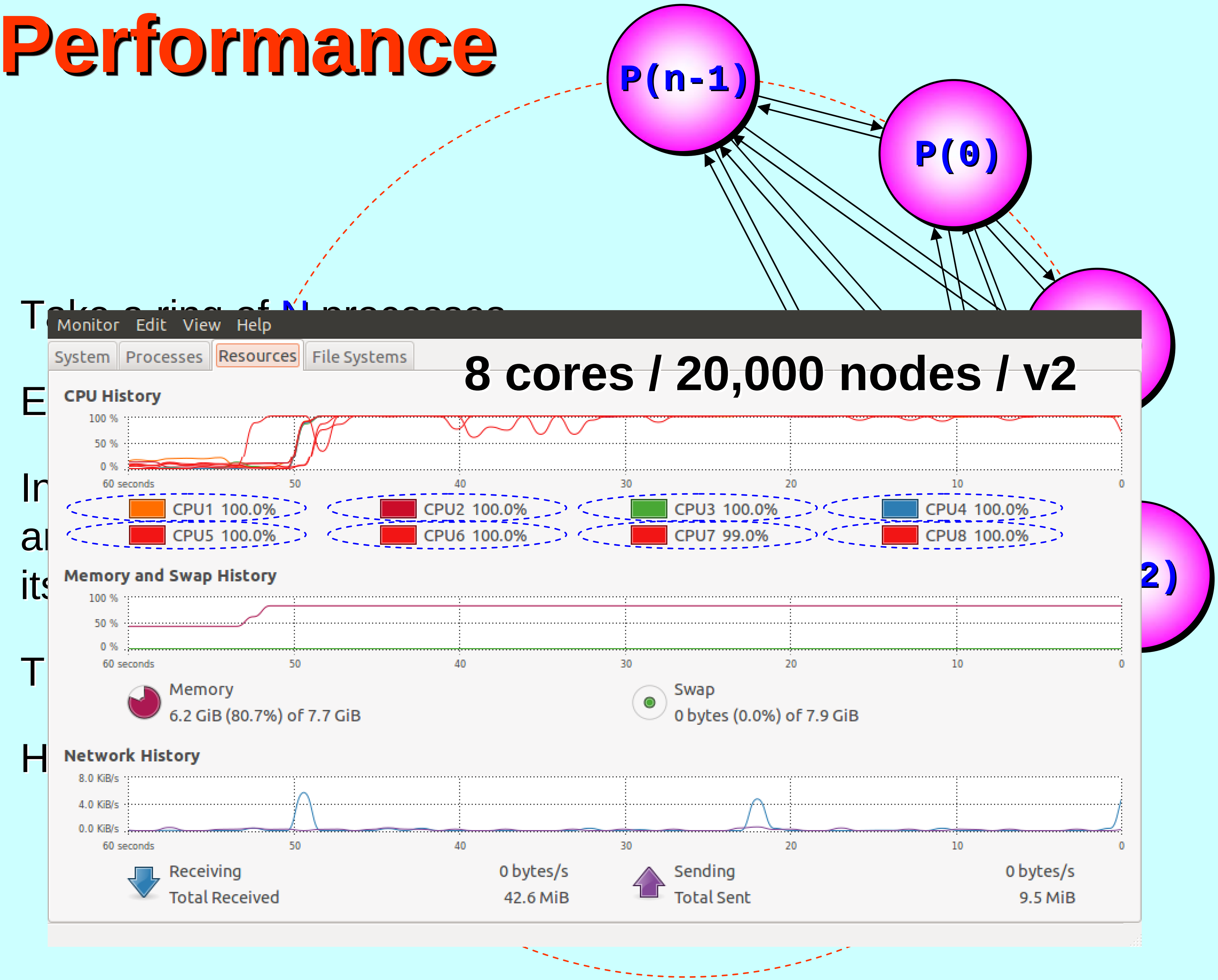


Performance

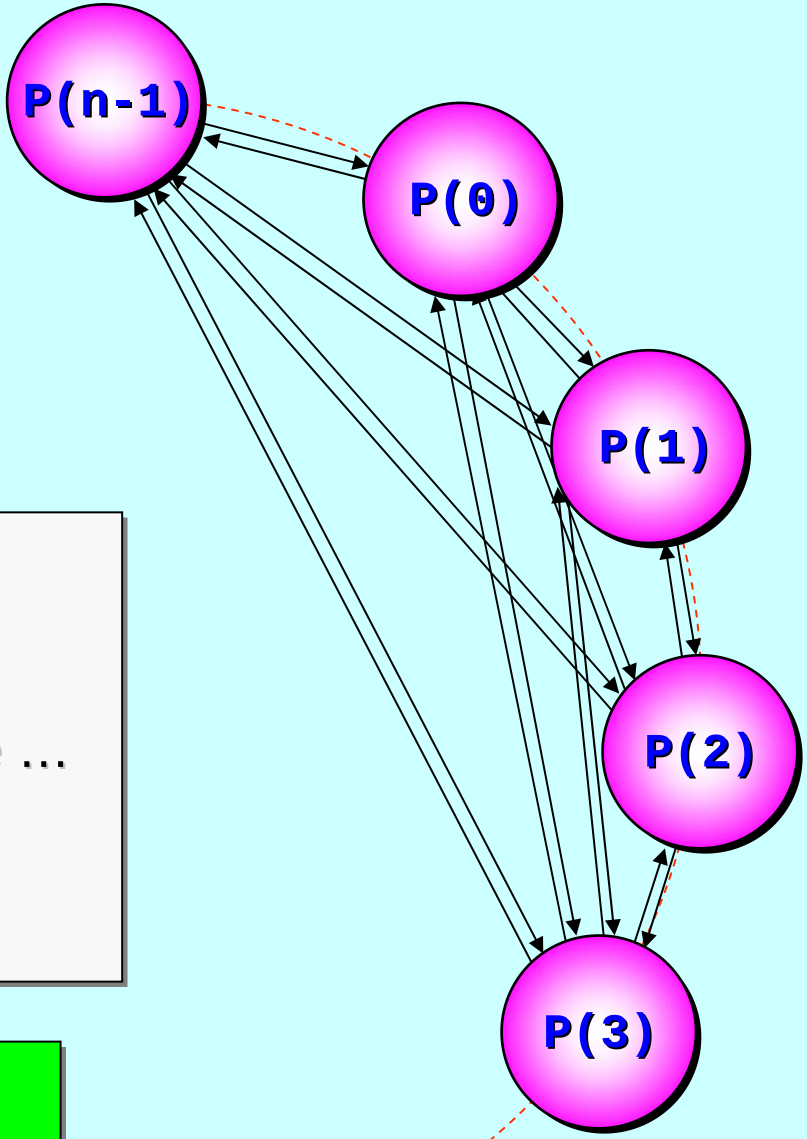
T
E
In
ar
its
T
H



Performance



Performance



So … how long per message?

6,000 nodes …

(V2) 12,000 processes …

30 messages node-to-node …

1.08 billion messages …

8 cores …

7 nanoseconds

Performance

So … how long per message?

10,000 nodes …

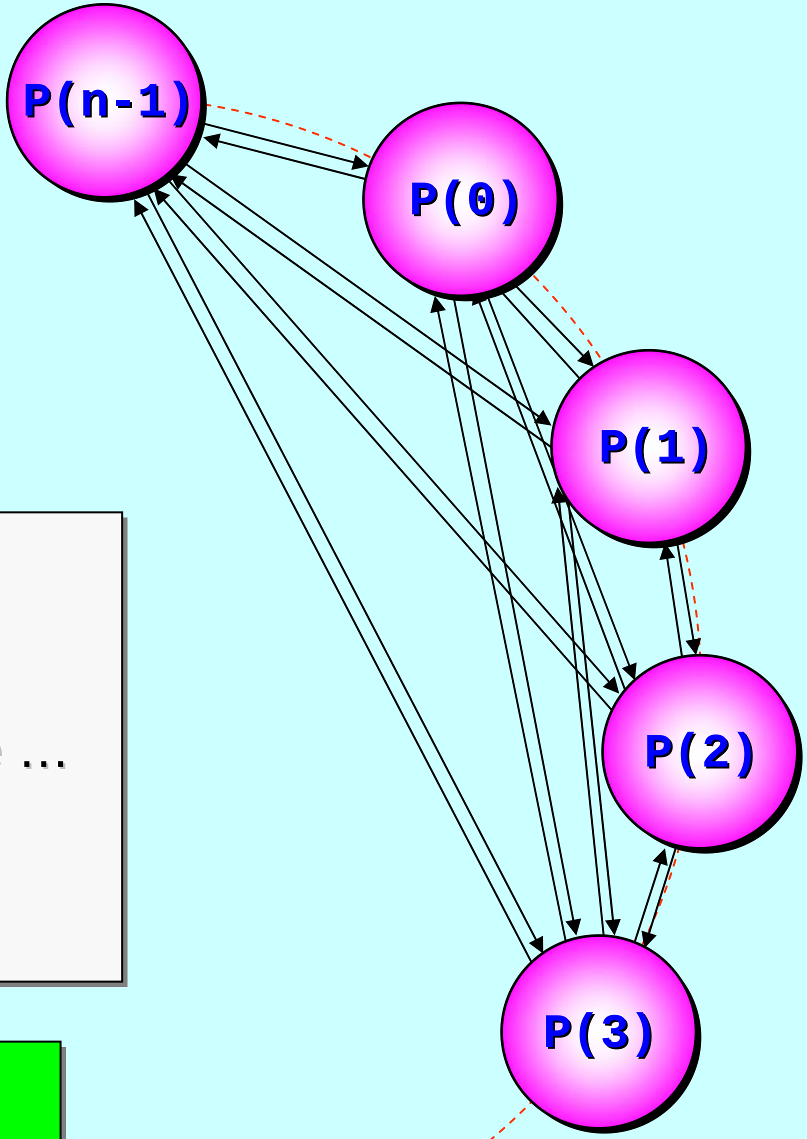
(V2) 20,000 processes …

30 messages node-to-node …

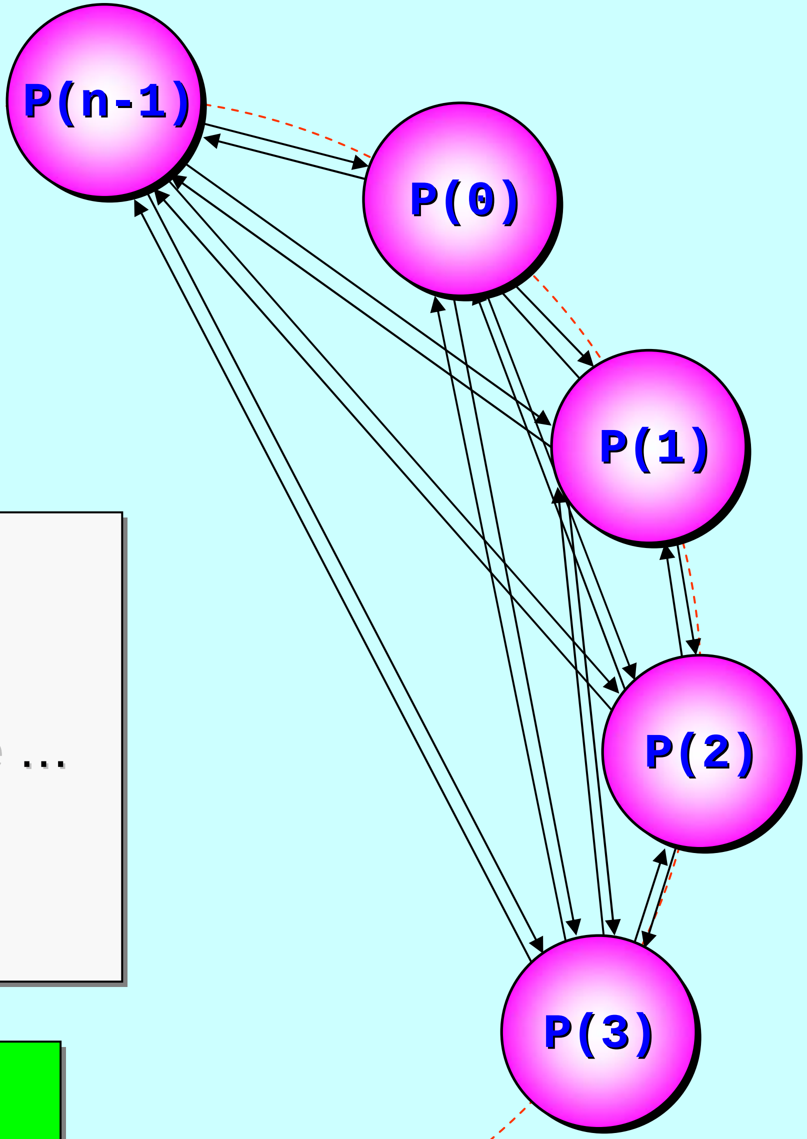
3 billion messages …

8 cores …

7 nanoseconds



Performance



So … how long per message?

20,000 nodes …

(V2) 40,000 processes …

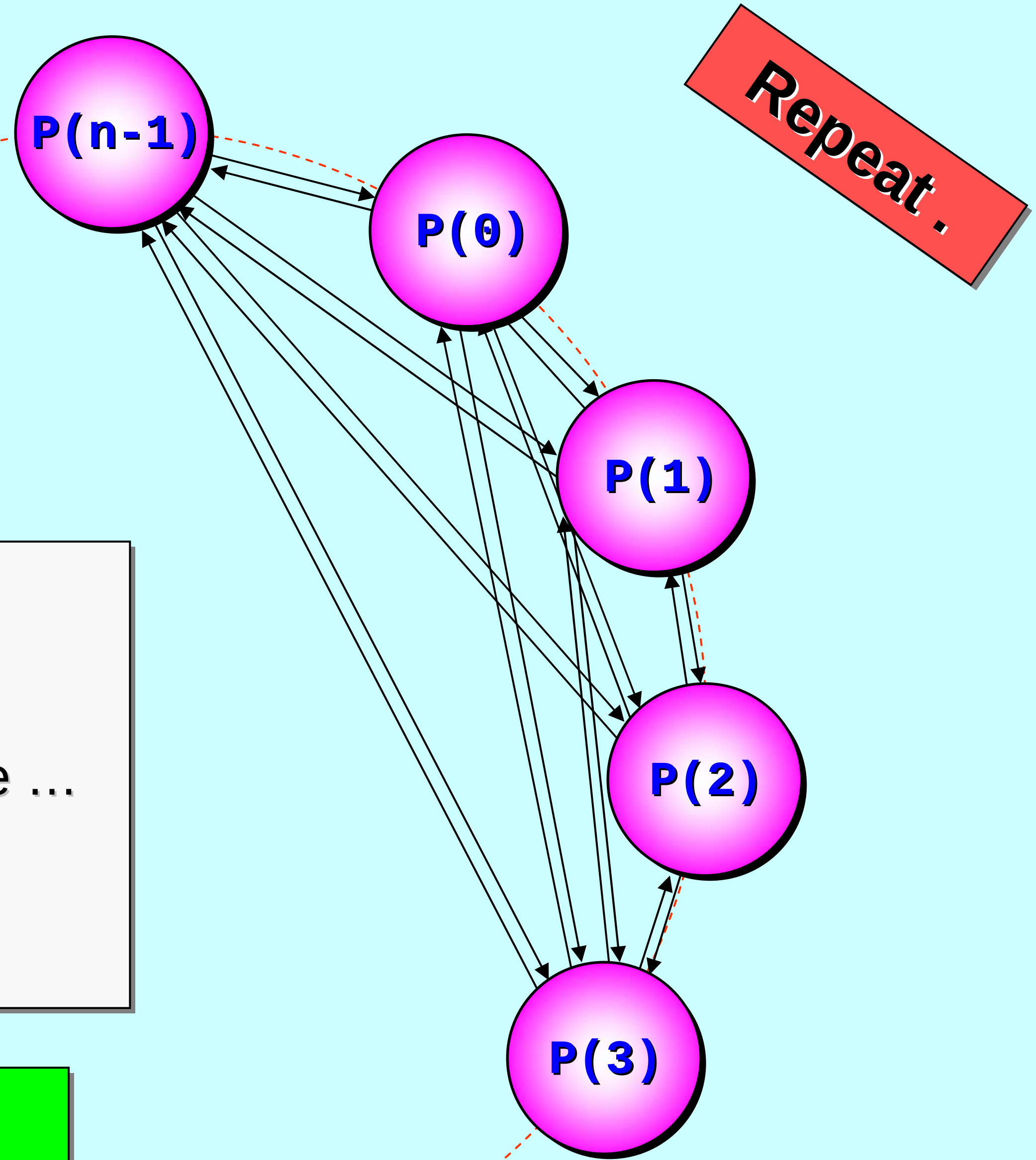
30 messages node-to-node …

12 billion messages …

8 cores …

8 **nanoseconds**

Performance



So … how long per message?

6,000 nodes …

(V2) 12,000 processes …

30 messages node-to-node …

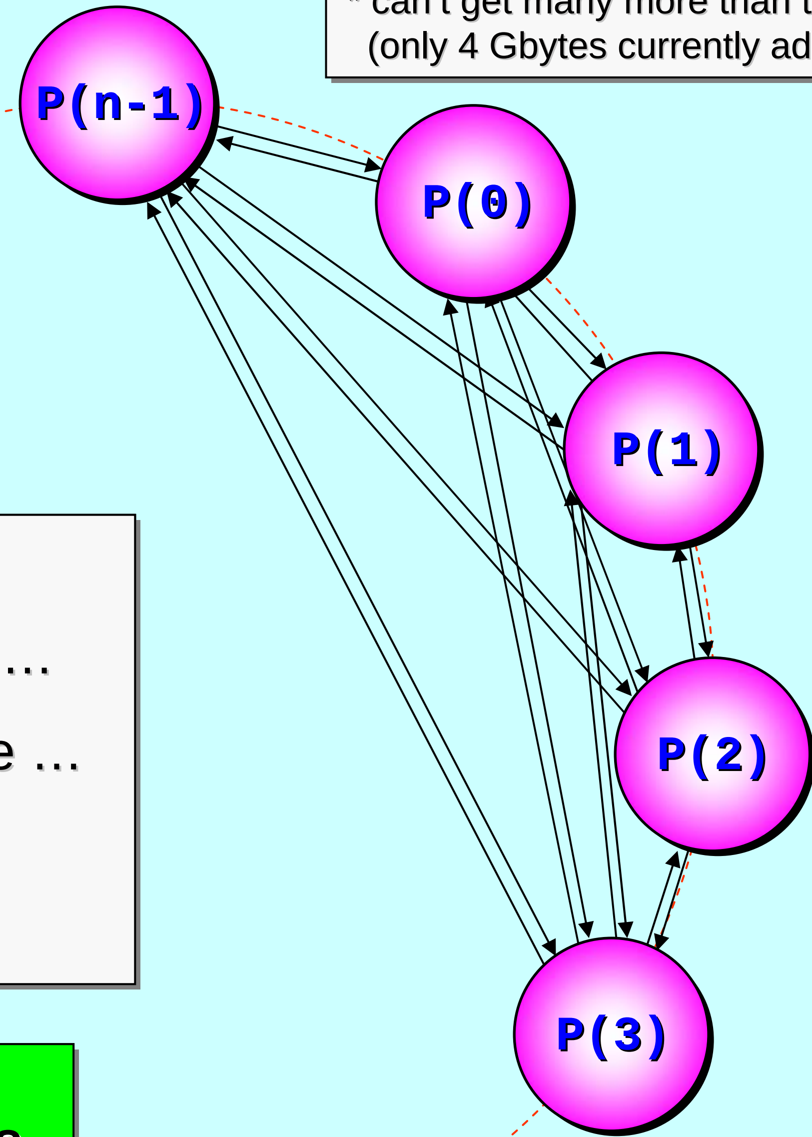
1.08 billion messages …

8 cores …

7 nanoseconds

Performance

* can't get many more than this
(only 4 Gbytes currently addressable).



So … how long per message?

6,000 nodes …

(V1) 72 million processes* …

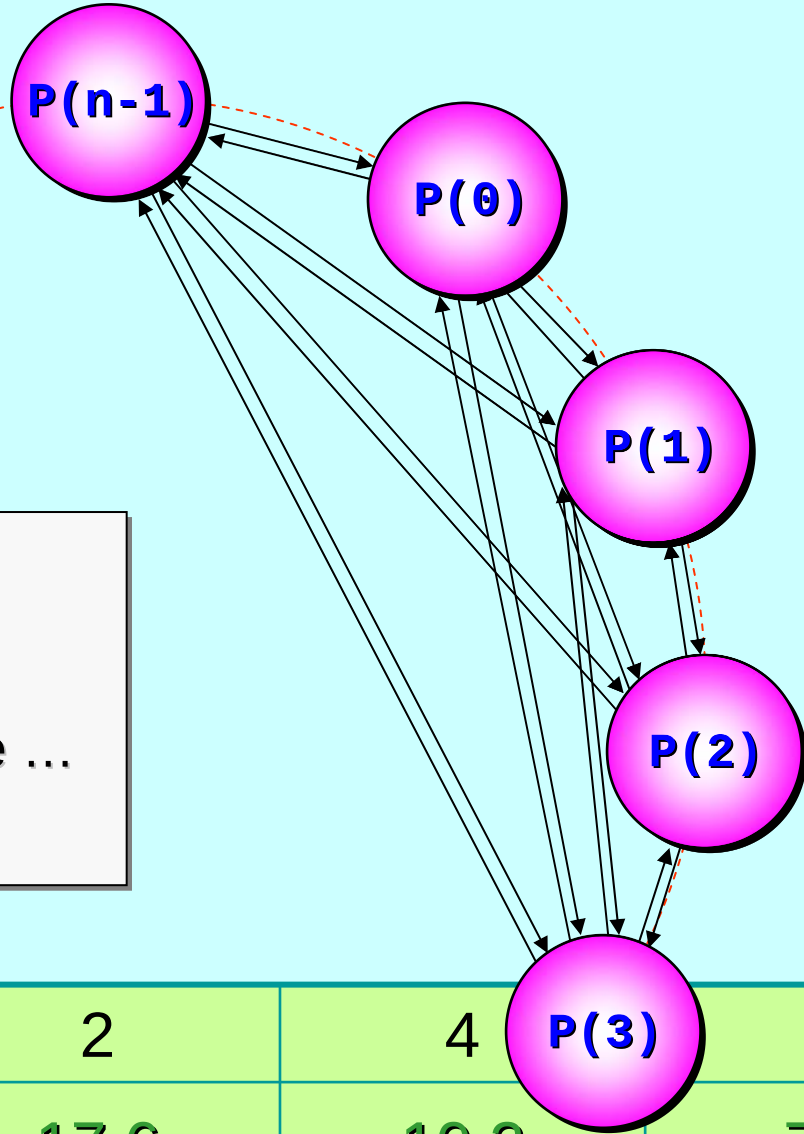
30 messages node-to-node …

1.08 billion messages …

8 cores …

14 nanoseconds

Performance



So … how long per message?

6,000 nodes …

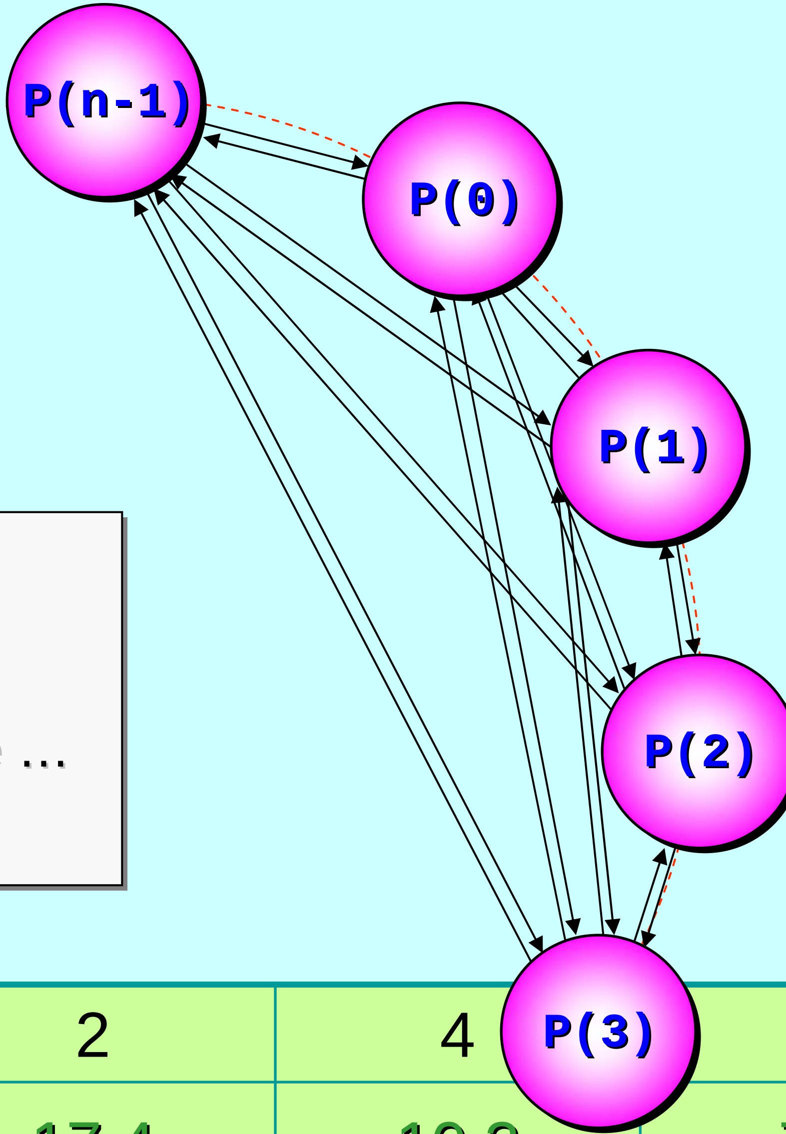
(V2) 12,000 processes …

30 messages node-to-node …

1.08 billion messages …

# cores	1	2	4	8
per message (nanoseconds)	34.5	17.6	10.2	7.0
speed up	1.0	2.0	3.4	4.9

Performance



So … how long per message?

10,000 nodes …

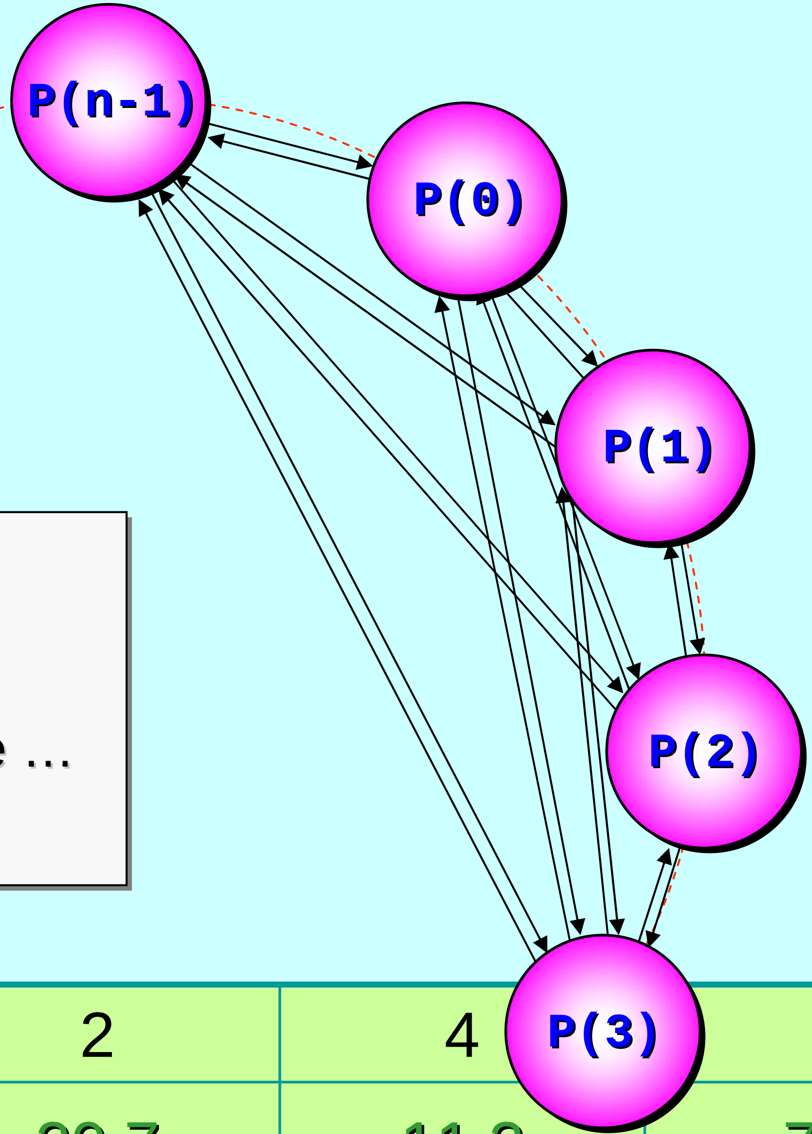
(V2) 20,000 processes …

30 messages node-to-node …

3 billion messages …

# cores	1	2	4	8
per message (nanoseconds)	37.5	17.4	10.3	7.2
speed up	1.0	2.2	3.6	5.2

Performance



So … how long per message?

20,000 nodes …

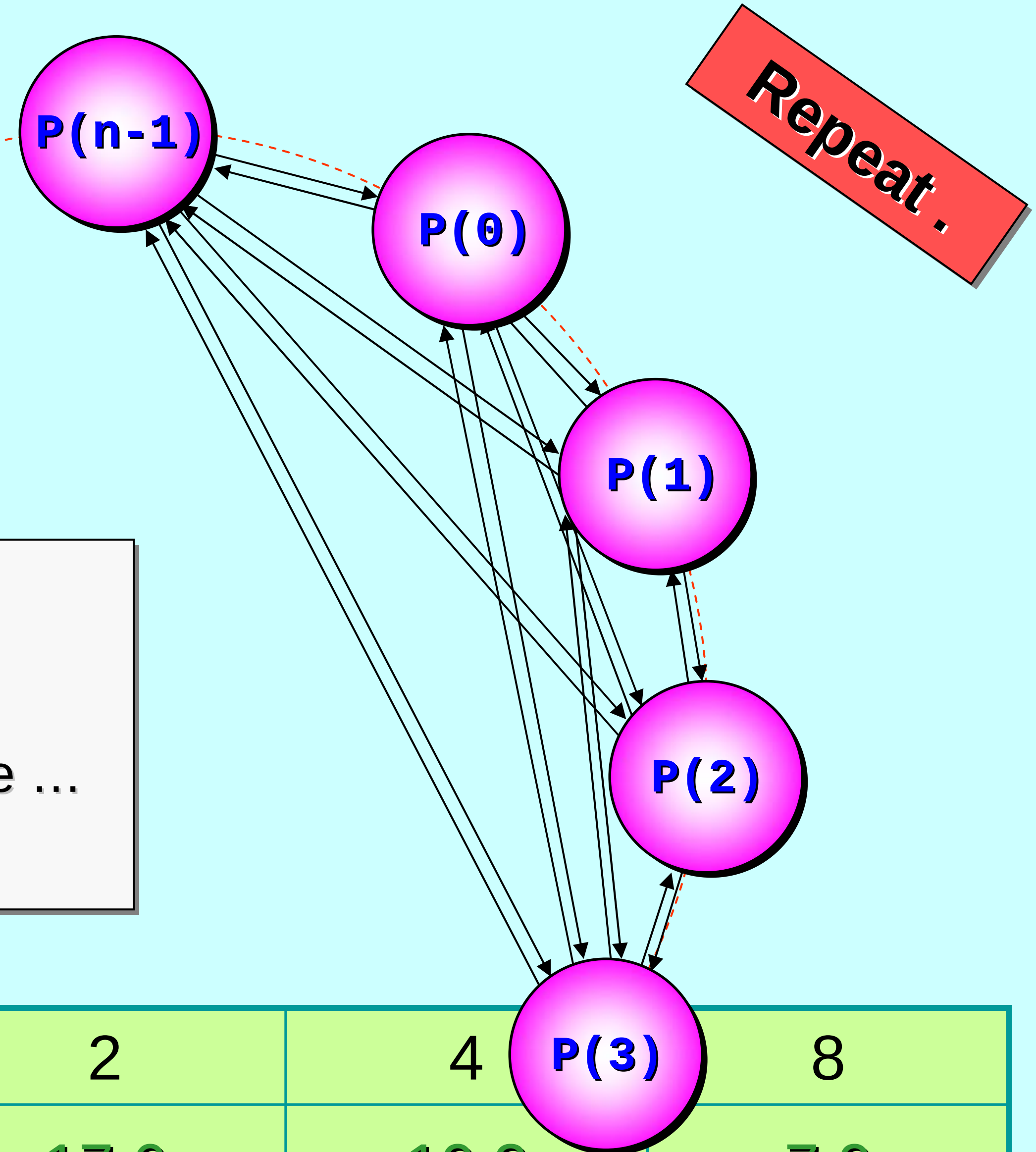
(V2) 40,000 processes …

30 messages node-to-node …

12 billion messages …

# cores	1	2	4	8
per message (nanoseconds)	61.8	20.7	11.3	7.7
speed up	1.0	3.0	5.5	8.0

Performance



So … how long per message?

6,000 nodes …

(V2) 12,000 processes …

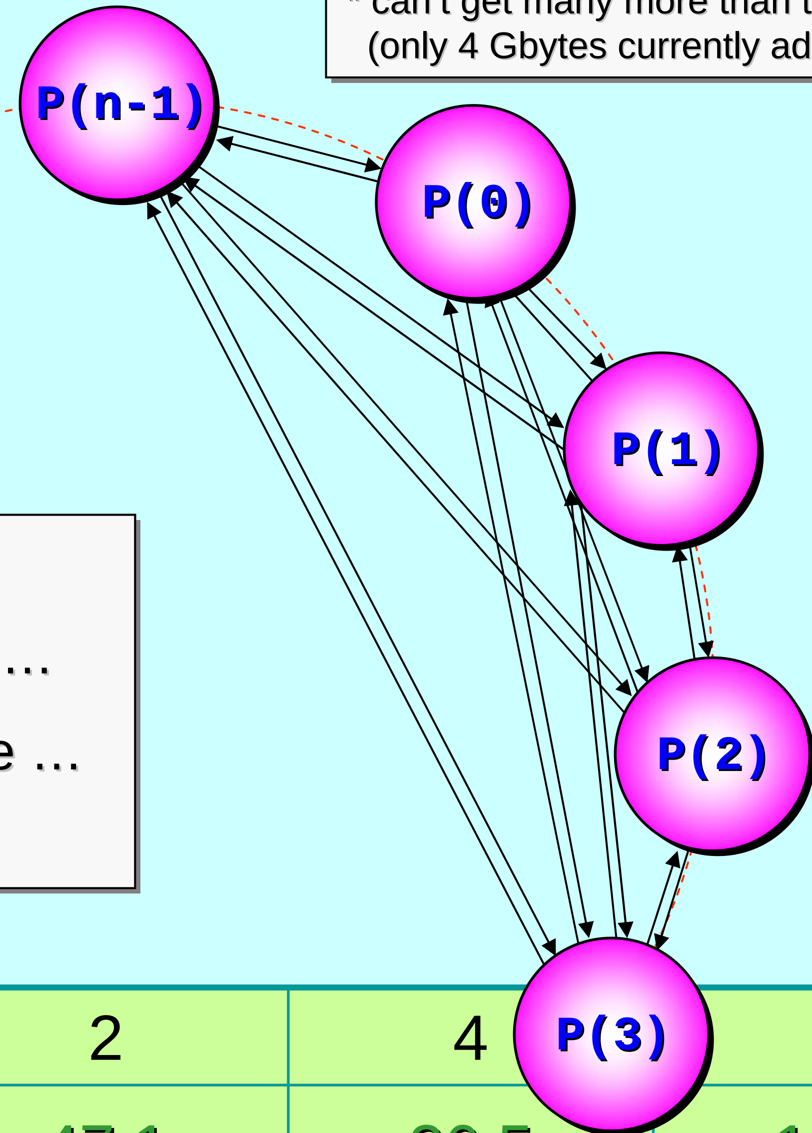
30 messages node-to-node …

1.08 billion messages …

# cores	1	2	4	8
per message (nanoseconds)	34.5	17.6	10.2	7.0
speed up	1.0	2.0	3.4	4.9

Performance

* can't get many more than this
(only 4 Gbytes currently addressable).



So … how long per message?

6,000 nodes …

(V1) 72 million processes* …

30 messages node-to-node …

1.08 billion messages …

# cores	1	2	4	8
per message (nanoseconds)	120.0	47.1	26.5	14.0
speed up	1.0	2.5	4.5	8.6

The Joy of Sync

Process oriented design ...

Synchronous communications …

Synchronous barriers …

Mutually assured destruction …

Non-blocking barriers …

Performance …

Any questions?