Integrating and Extending JCSP

Peter Welch, Neil Brown (University of Kent)
James Moores
Kevin Chalmers (Napier University)
Bernhard Sputh (University of Aberdeen)

CPA 2007, University of Surrey (10th. July, 2007)

Talk roadmap …

- History …
- Explicit channel "ends" …
- Alting barriers …
- Output guards …
- Extended rendezvous …
- Poison …
- Future (broadcast channels, generics, networking) ...

1996: *Java Threads Workshop (⇨ JCSP, CTJ …)*

1997: *JCSP 0.5 (early API and logic …)*

1999: *JCSP 0.94 (call chans, barriers, crew, tutorials …)*

2001: *JCSP 1.0-rc4 (major refactoring and documentation …)*

2004: *Quickstone JCSP Network Edition (channel "ends" , dynamic networking …)*

2006: *JCSP 1.0-rc7 (AltingBarriers, "spurious wakeup" protection …)*

2007: *JCSP 1.1 (output guards, extended rendezvous, poison …)*

Talk roadmap …

History …

Explicit channel "ends" …

Alting barriers …

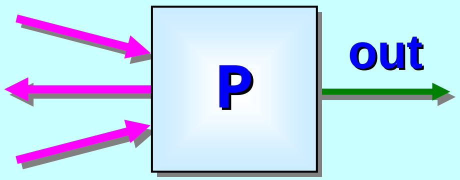
Output guards …

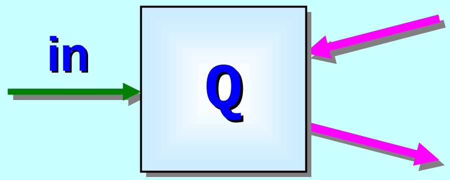
Extended rendezvous …

Poison …

Future (broadcast channels, generics, networking) ...

Channel *"Ends"* in occam-π



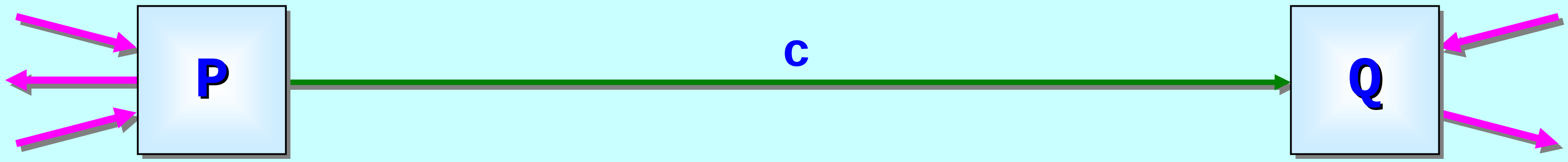


Each process gets its own "ends" of its external channels

```
PROC P (CHAN STUFF out!, ...)
  ...  local state
  SEQ
    ...  initialise state
    WHILE running
      SEQ
        ...  do stuff
        out ! value
        ...  more stuff
:
```

```
PROC Q (CHAN STUFF in?, ...)
  ...  local state
  SEQ
    ...  initialise state
    WHILE running
      SEQ
        ...  do stuff
        in ? x
        ...  more stuff
:
```

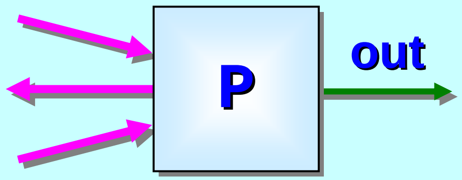
Channel *"Ends"* in occam-π



Each process gets its own "ends" of its external channels

```
CHAN STUFF c:
...  other channels
PAR
  P (c!, ...)
  Q (c?, ...)
  ...  other processes
```

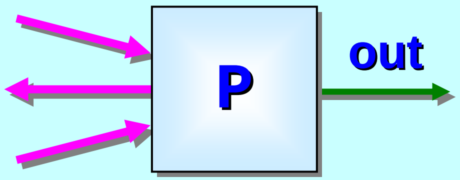
Channel *"Ends"* in JCSP




```
class P implements CSProcess {

  private final ChannelOutput out;
  ...  other channels and local state

  public P (ChannelOutput out, ...) {
    this.out = out;
    ...
  }

  public void run () {...}

}
```

Each process gets its own "ends" of its external channels

Channel *"Ends"* in JCSP




```
class P implements CSProcess {

  ...  external channels and local state

  public P (ChannelOutput out, ...) {...}


  public void run () {
    ...  initialise local state
    while (running) {
      ...  do stuff
      out.write (value);
      ...  more stuff
  }

}
```

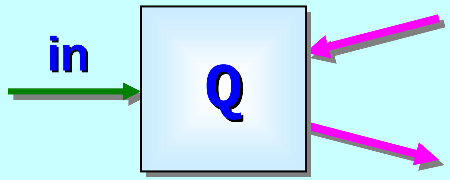
Each process gets its own "ends" of its external channels

Channel *"Ends"* in JCSP

*Each process gets its own **"ends"** of its external channels*

```
class Q implements CSProcess {

  private final ChannelInput in;
  ...   other channels and local state

  public Q (ChannelInput in, ...) {
    this.in = in;
    ...
  }

  public void run () {...}

:
```

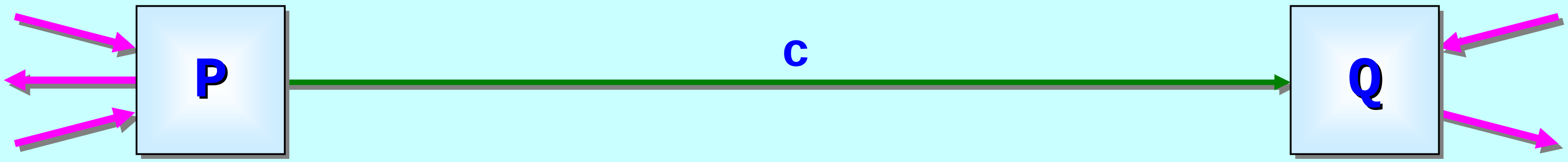
Channel *"Ends"* in JCSP



*Each process gets its own **"ends"** of its external channels*

```
class Q implements CSProcess {

  ...  external channels and local state

  public Q (ChannelInput in, ...) {...}


  public void run () {
    ...  initialise local state
    while (running) {
      ...  do stuff
      x = (Stuff) in.read ();
      ...  more stuff
  }

}
```

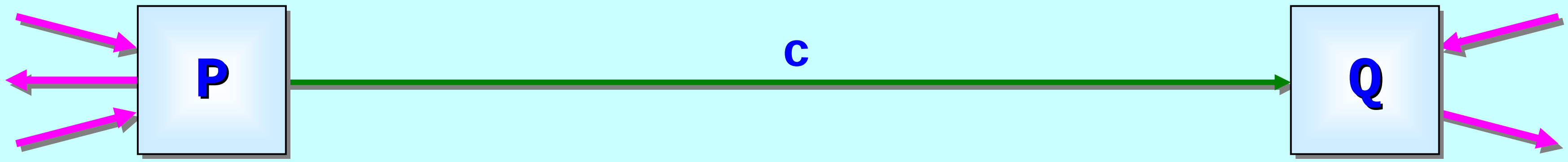
Channel *"Ends"* in JCSP 1.0-rc7



Each process gets "all" of its external channels

```
final One2OneChannel c = new One2OneChannel ();
...  other channels

new Parallel (
  new CSProcess[] {
    new P (c, ...),
    new Q (c, ...),
    ...  other processes
  }
).run ();
```

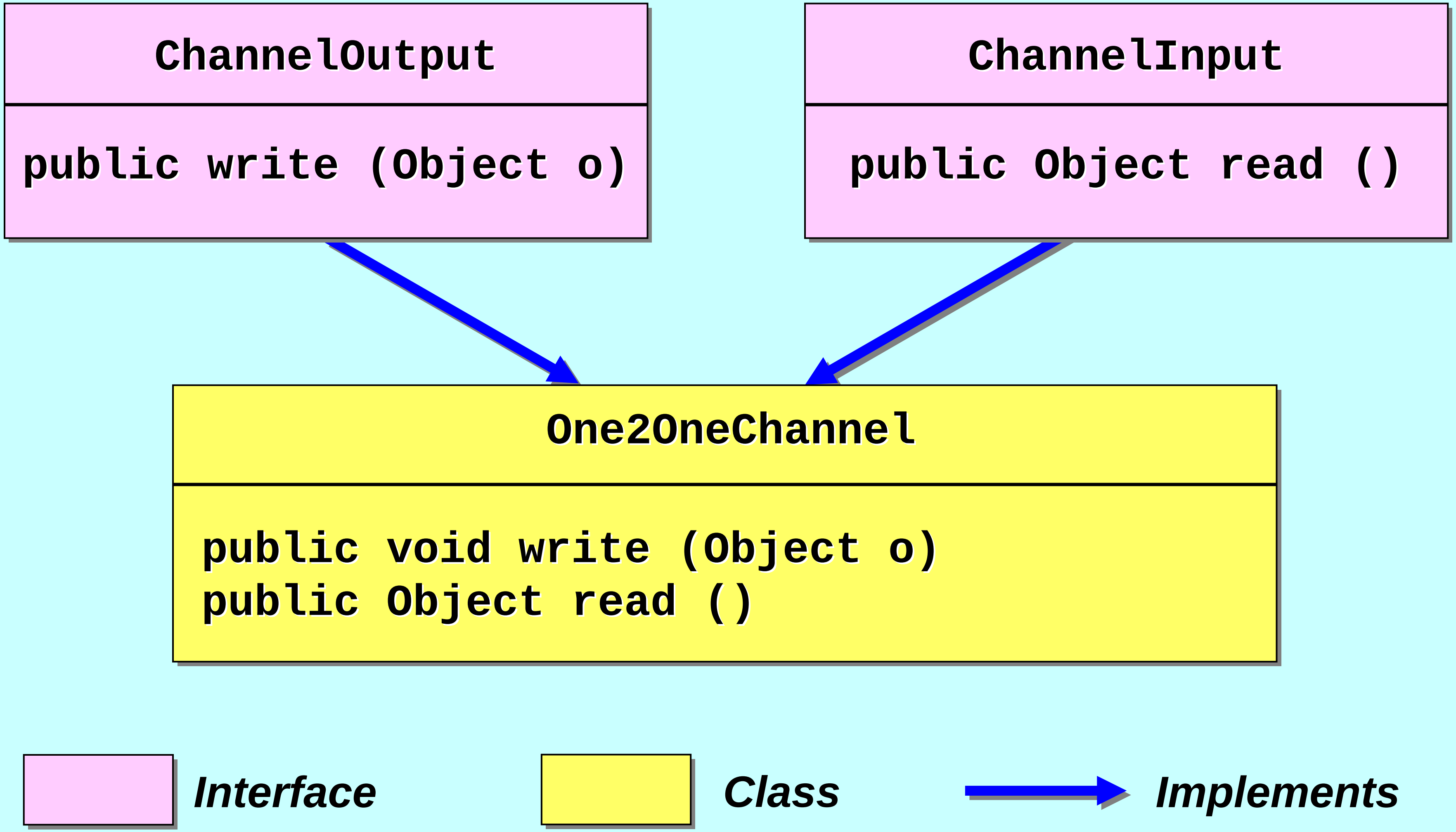
Channel *"Ends"* in JCSP 1.1



Each process gets its own "ends" of its external channels

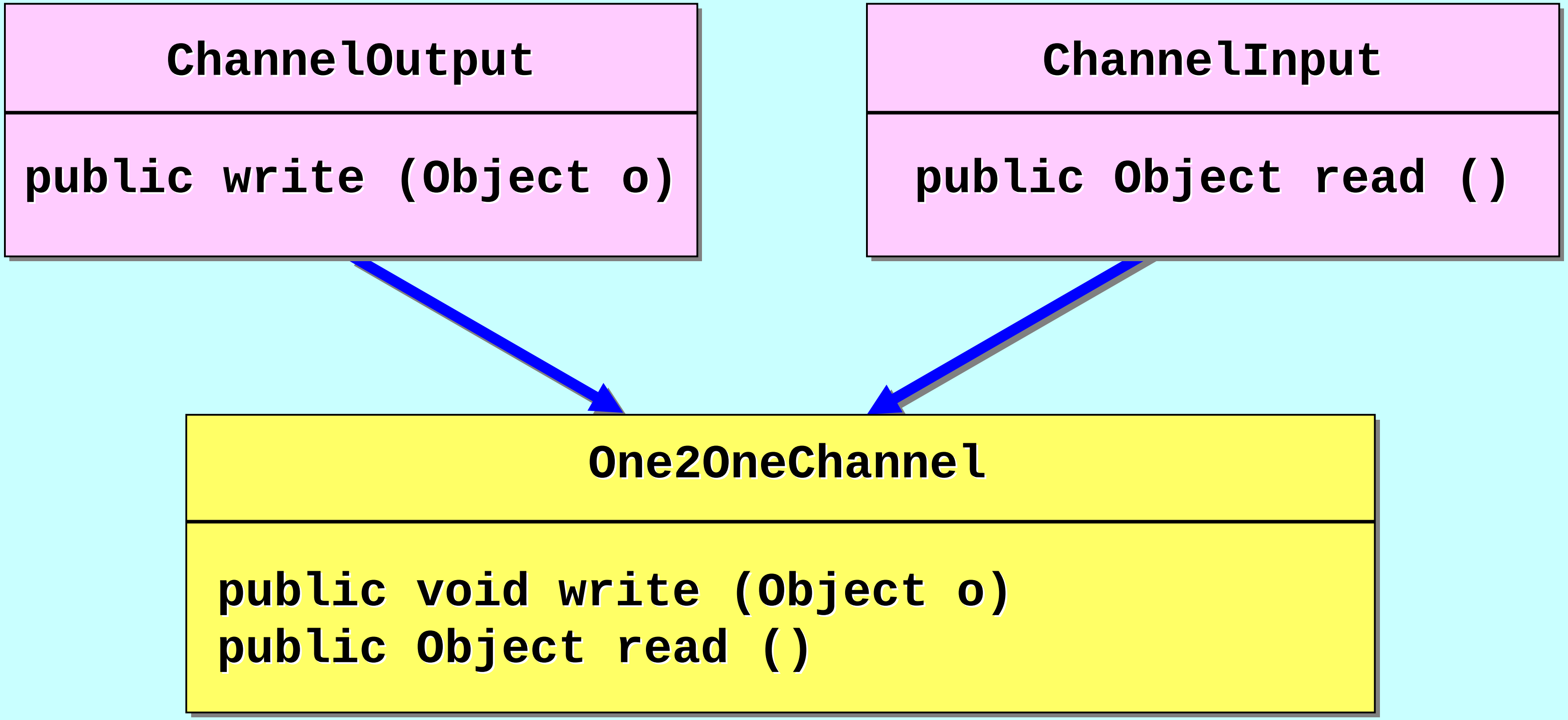
```
final One2OneChannel c = Channel.one2one ();
...  other channels

new Parallel (
  new CSProcess[] {
    new P (c.out (), ...),
    new Q (c.in (), ...),
    ...  other processes
  }
).run ();
```

Class Hierarchy* in JCSP 1.0-rc7



*** Ignoring Alting***

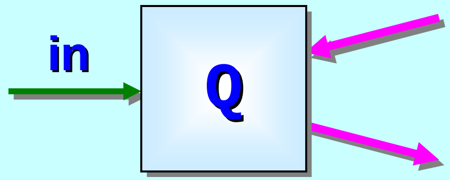
Class Hierarchy in JCSP 1.0-rc7



DANGER: any process, having been given a `ChannelInput`*, can cast it into a* `ChannelOutput` *and write to it! And vice-versa.*

Class Hierarchy in JCSP 1.0-rc7

```
class Q implements CSProcess {

  ...  external channels and local state

  public Q (ChannelInput in, ...) {...}


  public void run () {
    ...  initialise local state
    while (running) {
      ...  do stuff
      ((ChannelOutput) in).write (value);
      ...  more stuff
  }

}
```



☹ ☹ ☹ ☹ ☹ ☹

DANGER: any process, having been given a `ChannelInput`, *can cast it into a* `ChannelOutput` *and write to it! And vice-versa.*

Class Hierarchy* in JCSP 1.1

ChannelOutput
`public write (Object o)`

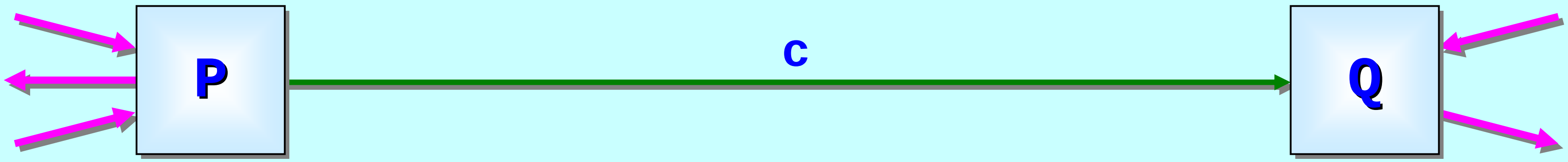
ChannelInput
`public Object read ()`

One2OneChannel
`public ChannelOutput out ()` `public ChannelInput in ()`

NO DANGER: users see only Java interfaces. The classes behind them are invisible, unrelated by class hierarchy and cannot be cast into each other. Processes must be given correct channel "ends".

**** Ignoring Alting***

Channel *"Ends"* in JCSP 1.1



Each process gets its own "ends" of its external channels

```
final One2OneChannel c = Channel.one2one ();
...  other channels

new Parallel (
  new CSProcess[] {
    new P (c.out (), ...),
    new Q (c.in (), ...),
    ...  other processes
  }
).run ();
```

channel manufacture

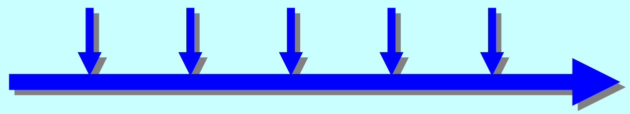
Channel Manufacture

All channels are made using **static** methods of the **Channel** class.

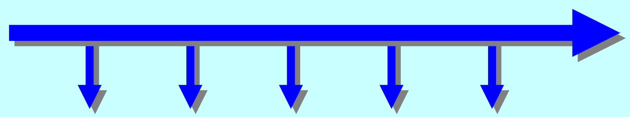
Decide whether the "ends" are to be shared:

Channel.one2one ()

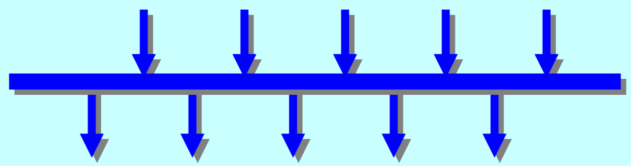
Channel.any2one ()



Channel.one2any ()



Channel.any2any ()



Channel Manufacture

All channels are made using **`static`** methods of the **`Channel`** class.

Decide whether the channels are to be buffered and, if so, how:

```
Channel.one2one (new Buffer (42))

Channel.any2one (new OverWriteOldestBuffer (8))

Channel.one2any (new OverFlowingBuffer (100))

Channel.any2any (new InfiniteBuffer ())
```

Channel Manufacture

All channels are made using **`static`** methods of the **`Channel`** class.

Decide whether the channels are poisonable and, if so, their immunity:

```
Channel.one2one (10)

Channel.any2one (5)

Channel.one2any (1000)

Channel.any2any (0)
```

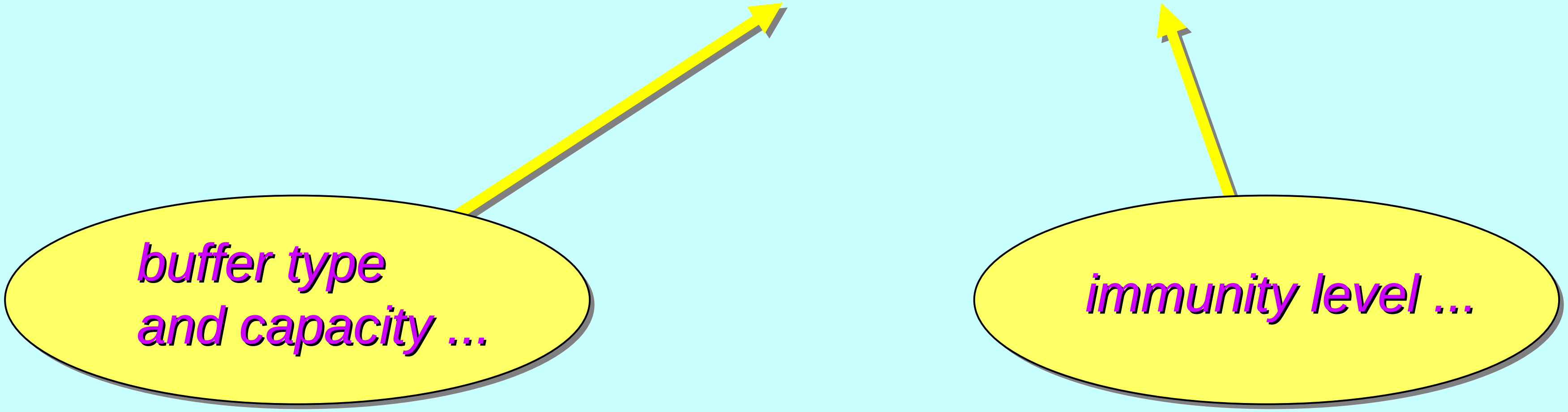
Immunity Level: the channel is immune to poisons up to this strength ...

Channel Manufacture

All channels are made using **`static`** methods of the **`Channel`** class.

The channels may be buffered and poisonable:

```
Channel.one2one (new Buffer (42), 10)
```

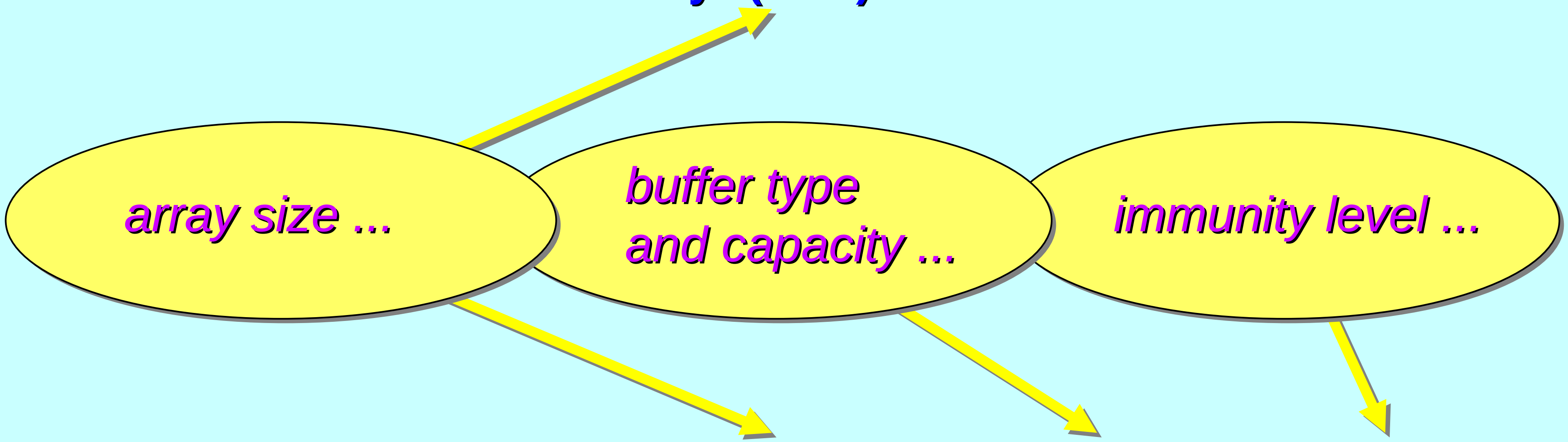


Channel Manufacture

All channels are made using **`static`** methods of the **`Channel`** class.

Arrays of channels – all kinds – may be built in one go:

```
Channel.one2oneArray (100)
```




```
Channel.any2oneArray (200, new Buffer (42), 10)
```

Channel Manufacture

All channels are made using **`static`** methods of the **`Channel`** class.

Channels may be specialised to carry **`int`**s:

```
Channel.one2oneInt ()

Channel.any2oneIntArray (200, new Buffer (42), 10)
```

In future, channels will be specialised using Java generics ...

Channel Summary

The JCSP process view and use of its external channels:

Unchanged – sees **`ChannelInput`**, **`AltingChannelInput`**, **`ChannelOutput`**, **`ChannelInputInt`**, *etc.*

Increased safety – cannot violate "endianness" …

A process does not (usually) care about the kind of channel – whether it is shared, buffered, poisonable, …*

* If a process needs to share an external channel-end between many sub-processes, it must be given one that is shareable – i.e. an **Any** end. JCSP 1.1 does cater for this.

Channel Summary

The JCSP network view of channels:

Changed – the correct channel "ends" must be extracted from channels and plugged into the processes using them ...

Increased safety – cannot violate "endianness" ...

A wide range of channel kinds (fully synchronised, buffered, poisonable, typed) are built from the **`Channel`** *class...*

JCSP processes work only with *interfaces* both for channels (whatever their kind) and for channel-ends. We think this will prove safer than providing *classes*.

Talk roadmap …

History …

Explicit channel "ends" …

Alting barriers …

Output guards …

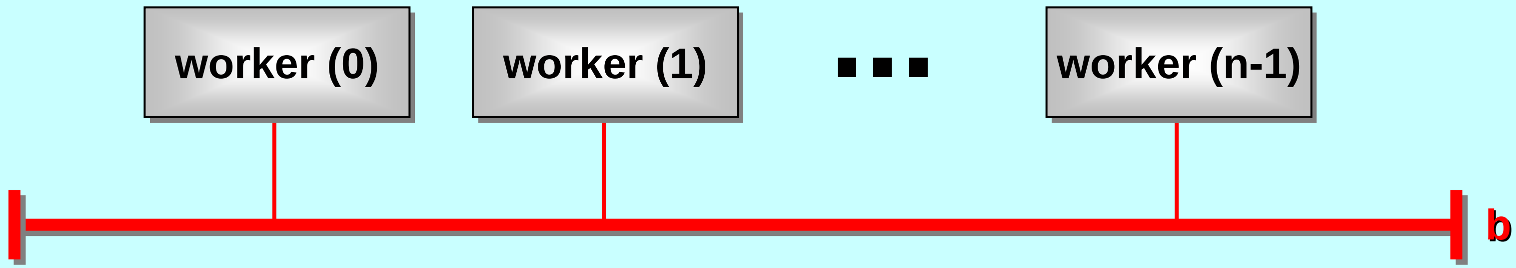
Extended rendezvous …

Poison …

Future (broadcast channels, generics, networking) ...

Barrier Synchronisation

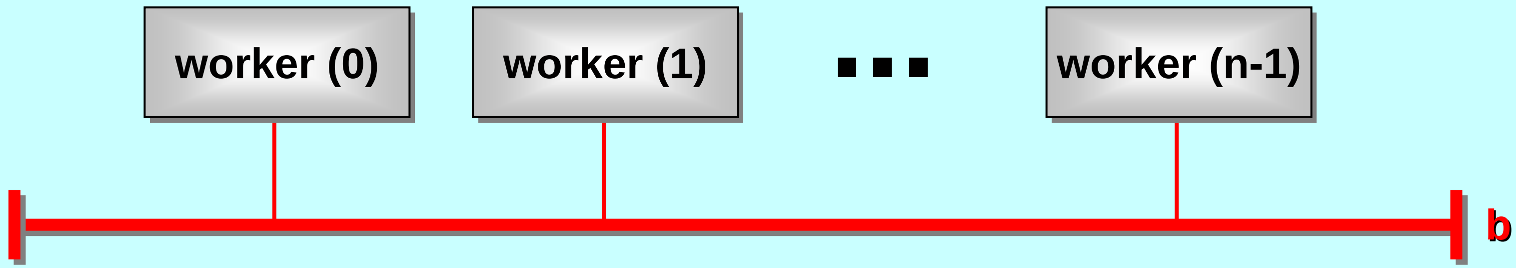
The existing *JCSP* **`Barrier`** type corresponds to a multiway CSP ***event***, though some higher level design patterns (such as *resignation*) have been built in.



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.

Barrier Synchronisation

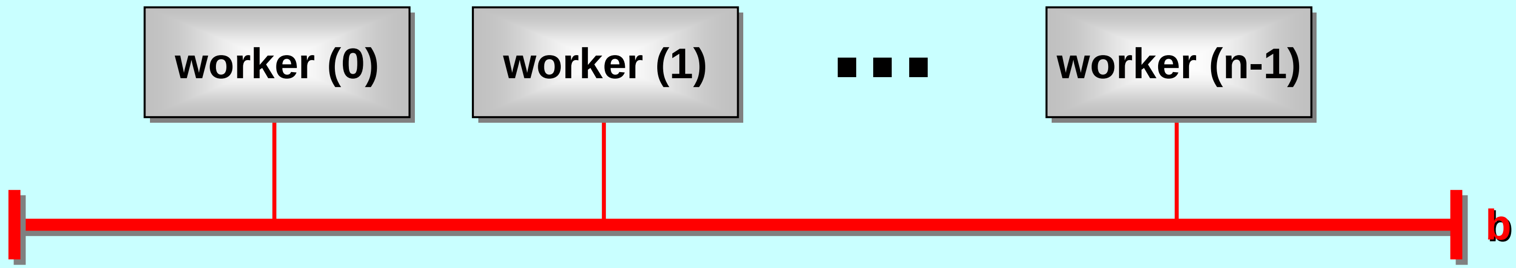
The existing *JCSP* **`Barrier`** type corresponds to a multiway CSP ***event***, though some higher level design patterns (such as *resignation*) have been built in.



However, once a process offers to *synchronise* on a **`Barrier`**, it is *committed*. In particular, it cannot offer this as part of an **`Alternative`** – so that it could timeout or choose another synchronisation (e.g. a channel communication or a different barrier) that *was ready to complete*! This is allowed by CSP.

Barrier Synchronisation

The existing *JCSP* **`Barrier`** type corresponds to a multiway CSP ***event***, though some higher level design patterns (such as *resignation*) have been built in.

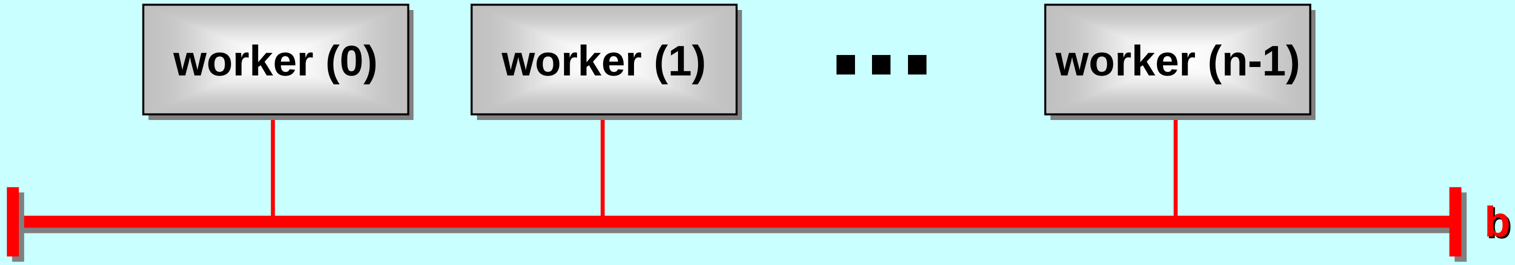


Disallowing more than one party in a synchronisation from withdrawing an offer to synchronise ... has been a constraint applied to all practical CSP implementations to date.

The *JCSP* **`AltingBarrier`** overcomes this constraint – at least within a single JVM. It uses the fast *'Oracle'* mechanism for choice over multiway synchronisations (presented last year).

Alting Barriers – the User View

An **`AltingBarrier`** is represented by a family of *front-ends*. Each process must use *its own* front-end (in the same way as a process must use a channel via one or other channel-end).




```
final AltingBarrier[] b = AltingBarrier.create (n);

final Worker[] workers = new Worker[n];
for (int i = 0; i < n; i++) {
  workers[i] = new Worker (i, b[i]);
}

new Parallel (workers).run ();
```

Alting Barriers – the User View

To offer to synchronise on an **`AltingBarrier`**, a process simply includes its *front-end* in a **`Guard`** array associated with an **`Alternative`** and invokes a **`select()`** method.

That is all !!

Its index will be returned *if-and-only-if* all processes currently enrolled on the **`AltingBarrier`** have made the same offer (using their *front-ends*). Either *all* these processes select their *front-end's* index – or *none* do.

Alting Barriers – the User View

Two shortcuts:

If a process is able to *commit* to synchronise on an **`AltingBarrier`**, it may **`sync()`** on its *front-end* (rather than set up an **`Alternative`** with one **`Guard`**).

A further shortcut (over an **`Alternative`**) is provided to *poll-with-timeout* its *front-end* for completion of the **`AltingBarrier`**.

Alting Barriers – the User View

Dynamics:

Further *front-ends* to an **`AltingBarrier`** may be made from an existing one (through **`expand()`** and **`contract()`** methods).

As for the earlier *(committed-only)* **`Barrier`** class, processes may temporarily **`resign()`** from an **`AltingBarrier`** and, later, re-**`enrol()`**.

A process may communicate a *(non-resigned)* **`AltingBarrier`** *front-end* to another process, which must **`mark()`** it before use. Only one process at a time may use a *front-end*. This is checked!

Alting Barriers – the User View

Priorities:

The **`priSelect()`** method prioritises the guards *locally* for the process making the offers.

Suppose process **`A`** offers alting barrier **`x`** with higher priority than alting barrier **`y`** ... and process **`B`** offers **`y`** with higher priority than **`x`**. It would be impossible to resolve the choice in favour of either **`x`** or **`y`** in any way that satisfied the conflicting requirements of **`A`** and **`B`**.

Alting Barriers – the User View

Priorities:

However, **`priSelect()`** is allowed for choices including barrier guards.

It *honours* the respective priorities defined between non-barrier guards.

It *honours* the respective priorities defined between a barrier guard and non-barrier guards (enabling, for example, priority response to *timeouts* or *channel interrupts* over ever-offered barriers).

Relative priorities between barrier guards are *inoperative*.

Alting Barriers – the User View

Misuse:

The implementation guards against misuse, throwing an `AltingBarrierError` when riled:

Different threads trying to use the same front-end ...

Attempt to enrol whilst enrolled ...

Attempt to use as a guard whilst resigned ...

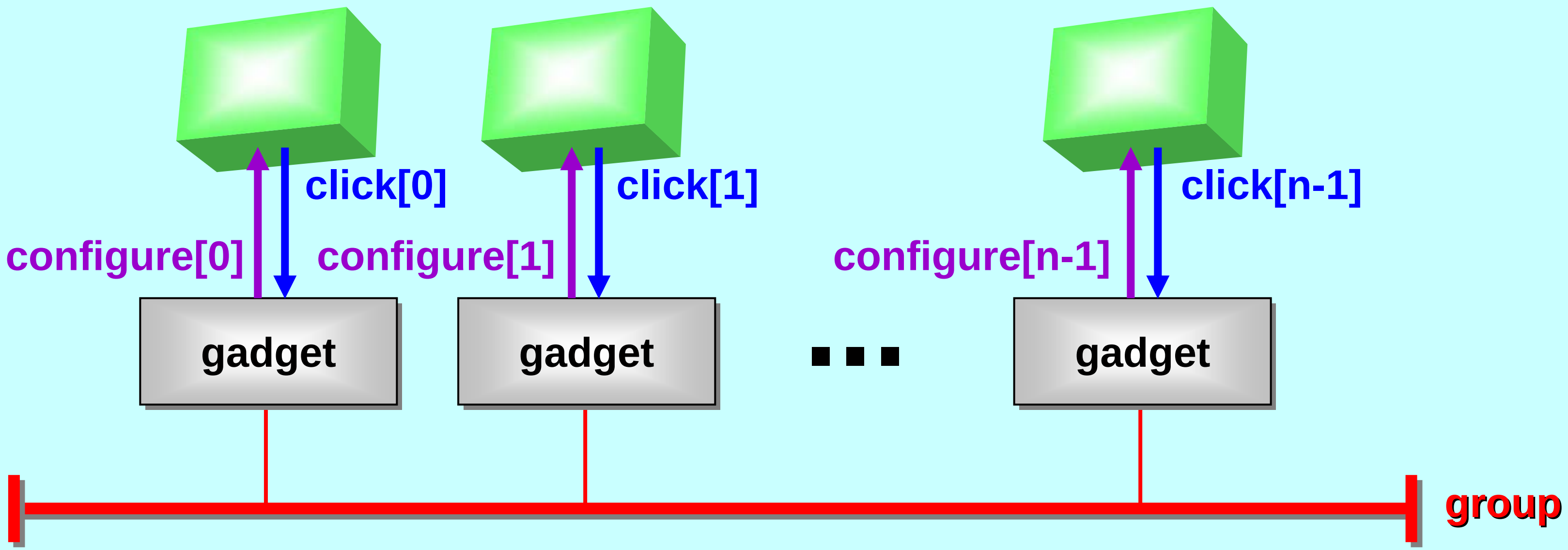
Attempt to resign, sync, expand, contract or mark whilst resigned ...

Alting Barriers – Example

An array of *gadgets* control and react to an array of *display buttons*.

Each gadget may *configure its button with colour and text* and *receives click signals if the button is pressed*.

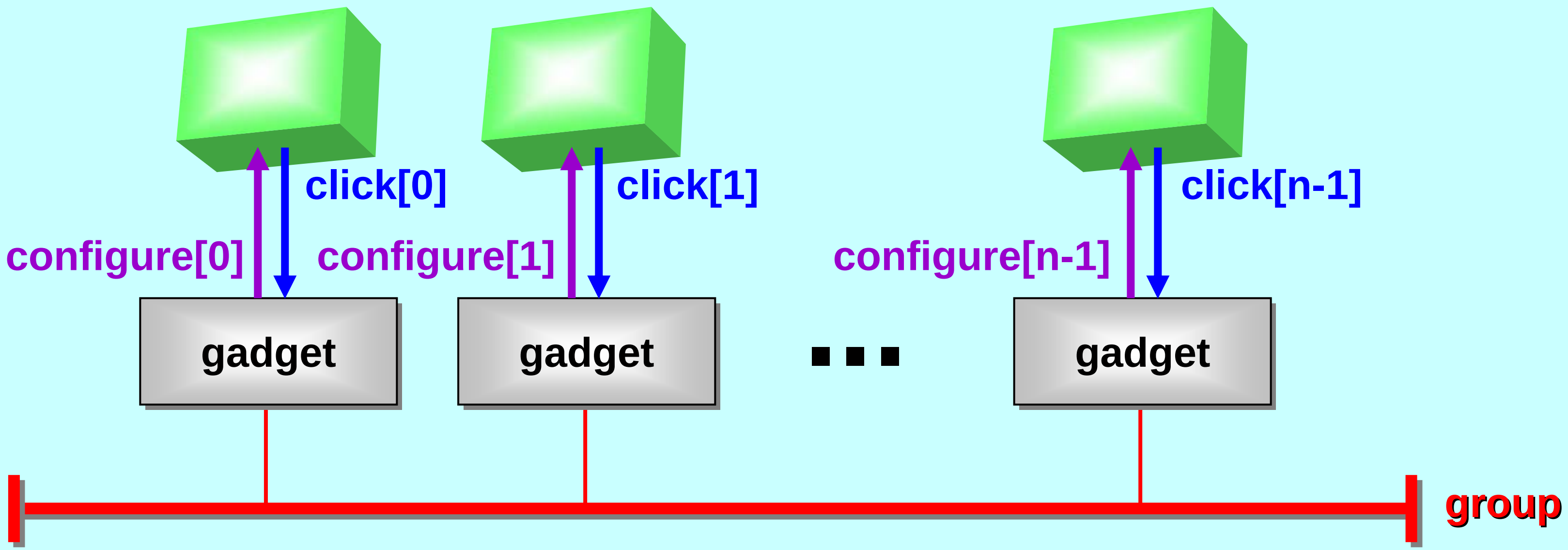
The *gadgets* coordinate *"group actions"* with an **`AltingBarrier`**.



Alting Barriers – Example

Each *gadget* maintains an individual count. Each *gadget* has two modes of operation, switched *at any time* by a *click* event.

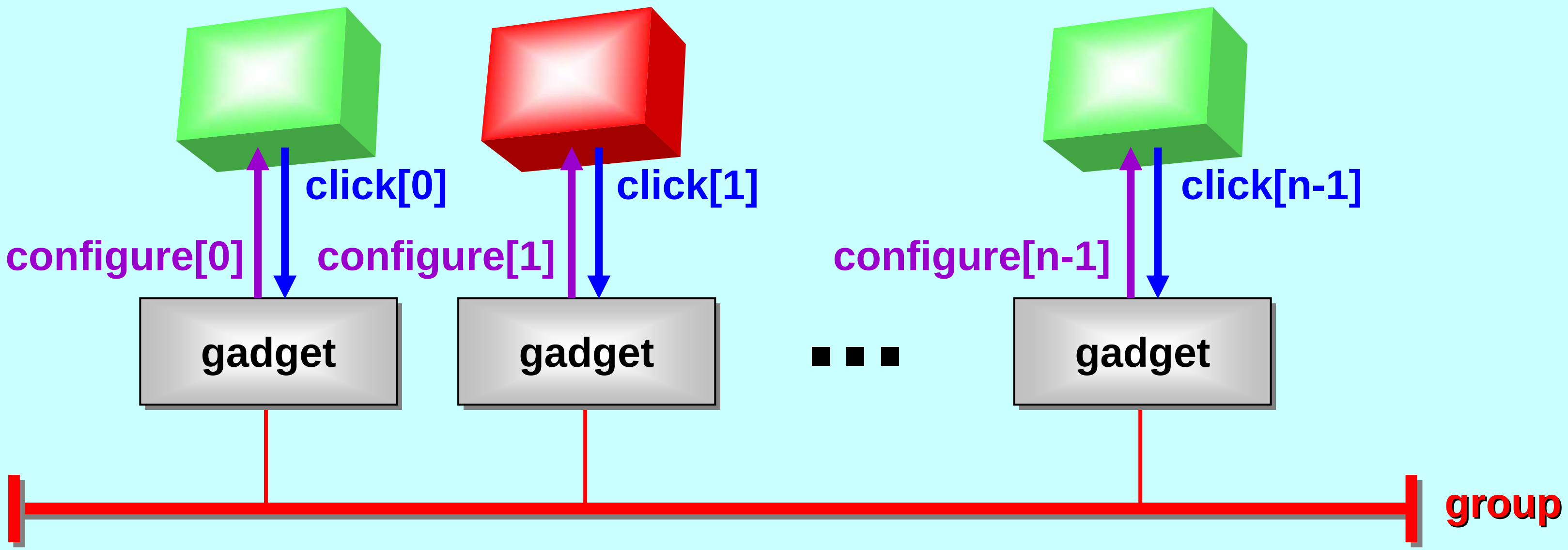
In *individual* mode, a gadget sets its button *green* and *increments* its count as fast as possible, displaying the value as text upon its button.



Alting Barriers – Example

Each *gadget* maintains an individual count. Each *gadget* has two modes of operation, switched *at any time* by a *click* event.

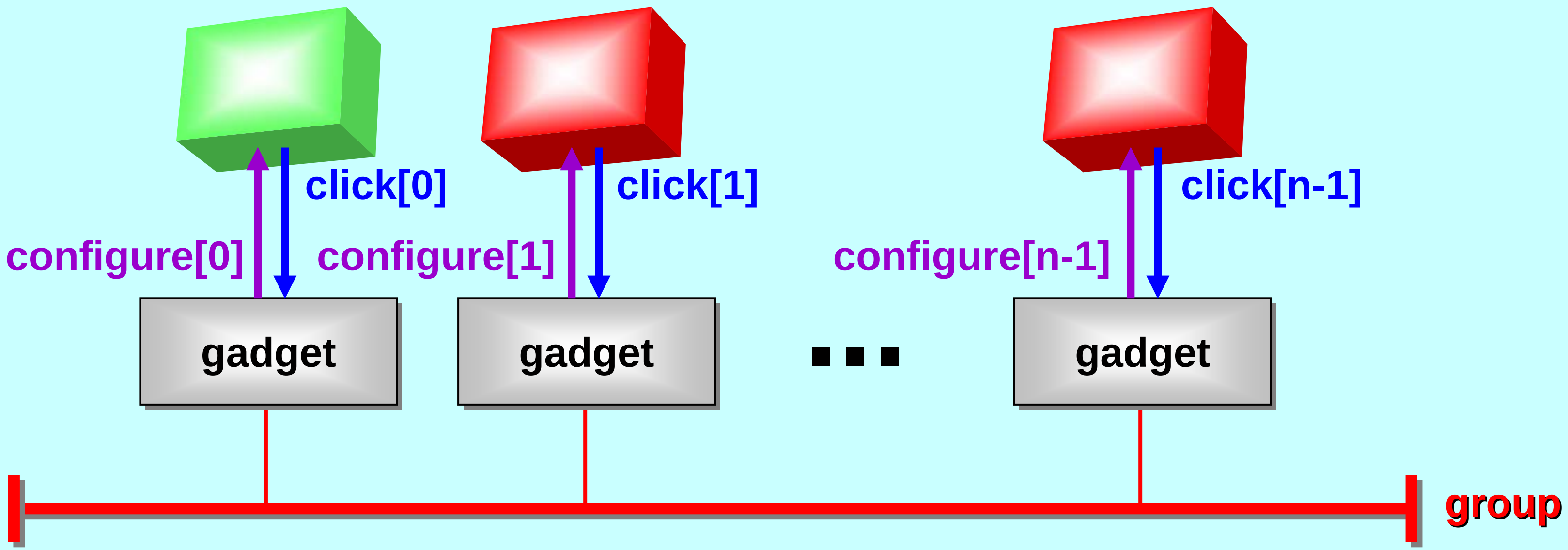
In *group* mode, a gadget sets its button *red* and waits for all other gadgets to get into *group* mode. Whilst waiting, a *click* on its *button* would return it to *individual* mode.



Alting Barriers – Example

Each *gadget* maintains an individual count. Each *gadget* has two modes of operation, switched *at any time* by a *click* event.

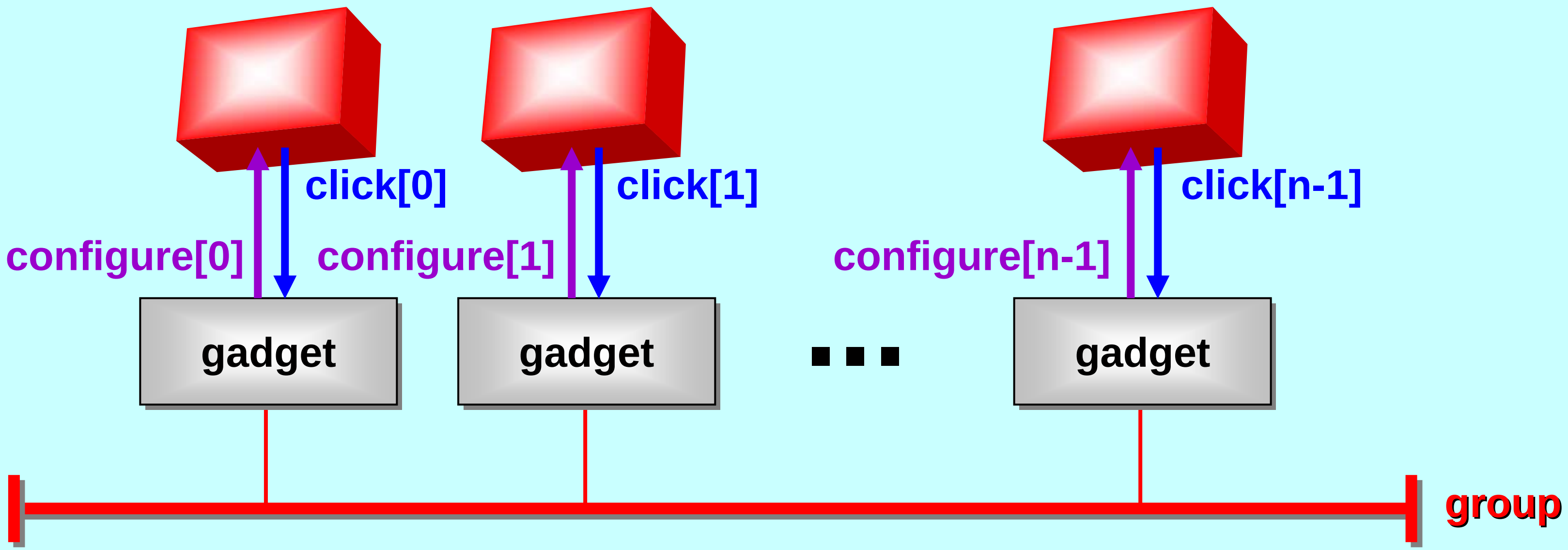
In *group* mode, a gadget sets its button *red* and waits for all other gadgets to get into *group* mode. Whilst waiting, a *click* on its *button* would return it to *individual* mode.



Alting Barriers – Example

Each *gadget* maintains an individual count. Each *gadget* has two modes of operation, switched *at any time* by a *click* event.

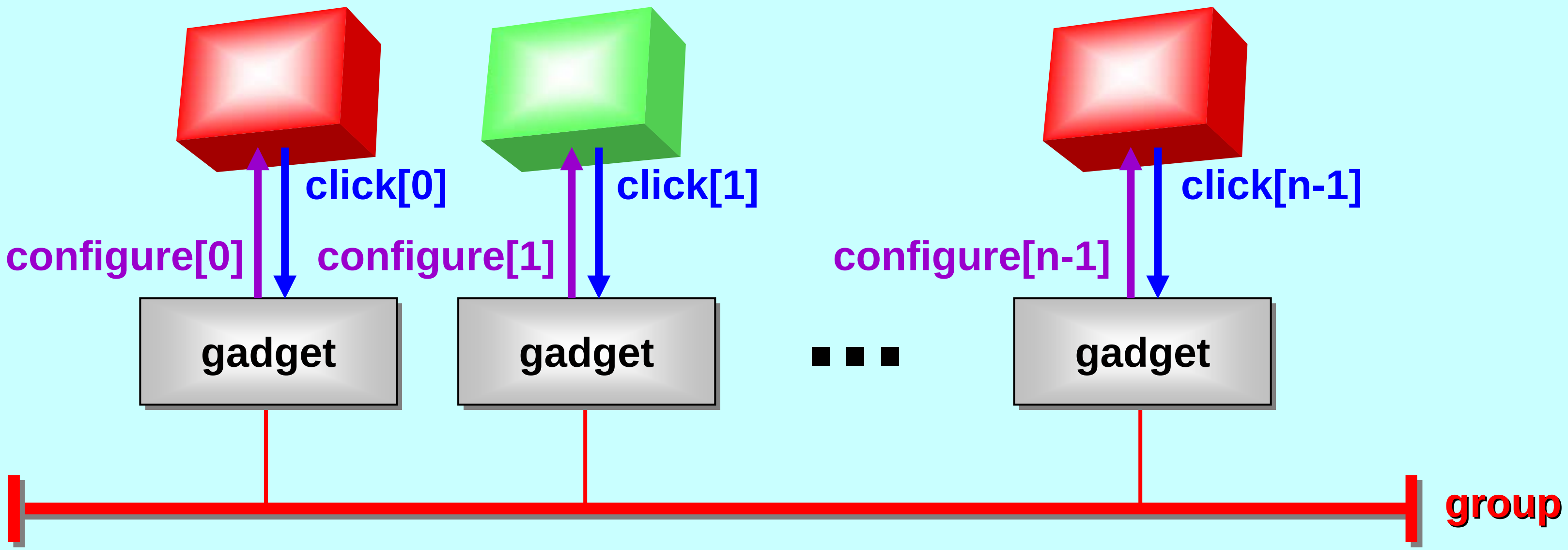
Whilst all are in *group* mode, each *gadget decrements* its count in synchrony with all *gadgets* and as fast as possible, displaying the value as text upon its *button*.



Alting Barriers – Example

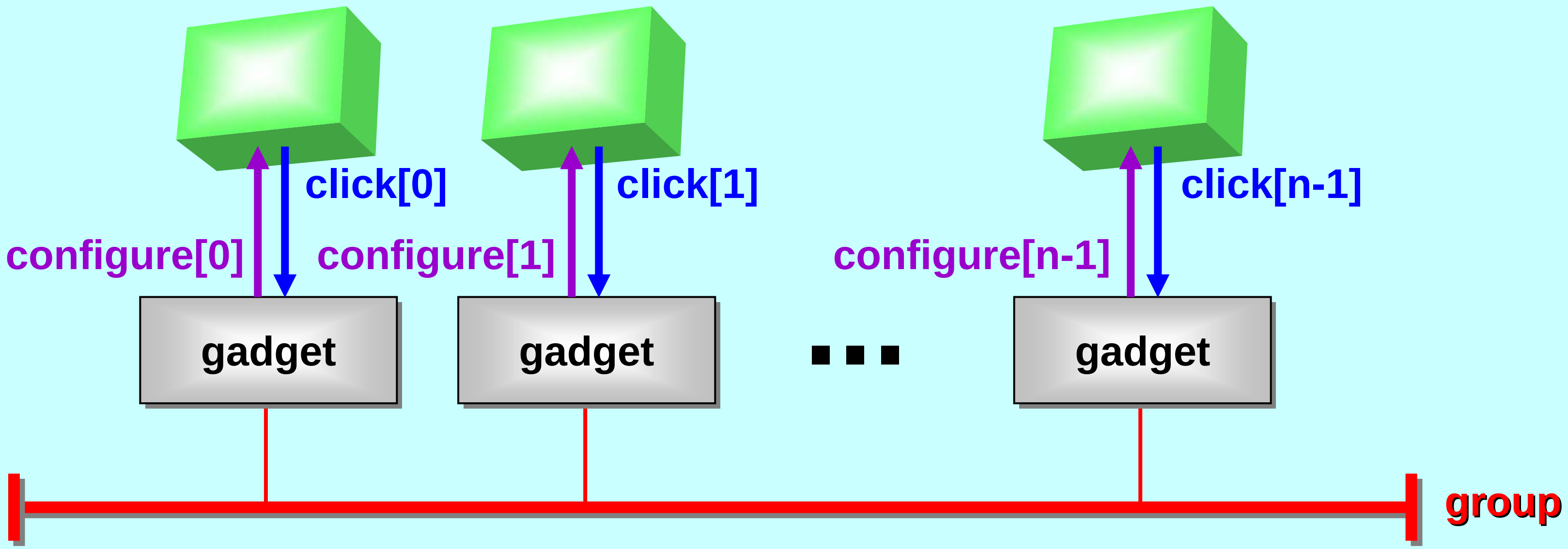
Each *gadget* maintains an individual count. Each *gadget* has two modes of operation, switched *at any time* by a *click* event.

If any *gadget clicks* back to *individual* mode, the *group* work ceases.



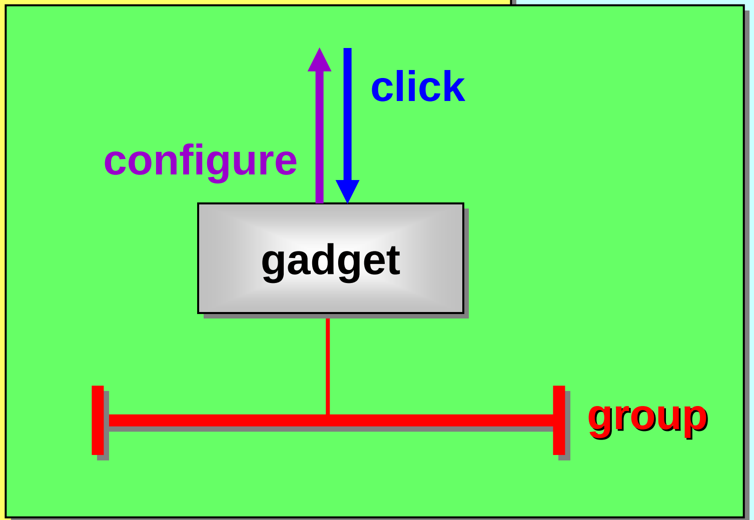
Play game …

Alting Barriers – Example



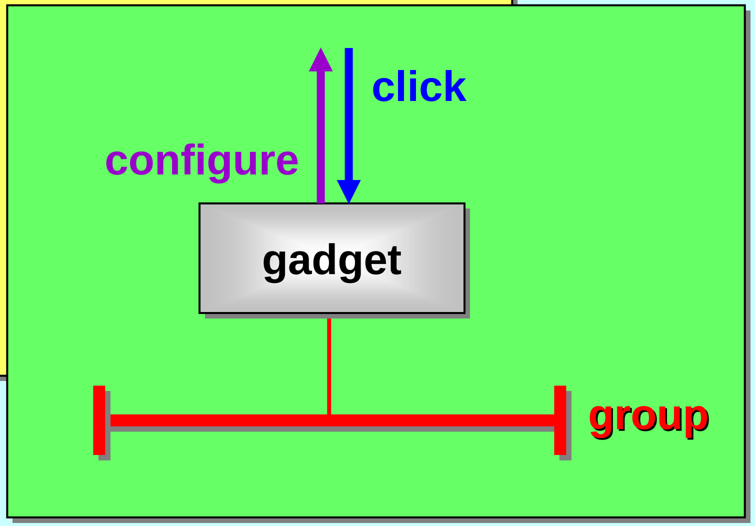
Alting Barriers – Example

```
public class Gadget implements CSProcess {

  private final AltingChannelInput click;
  private final AltingBarrier group;
  private final ChannelOutput configure;

  public Gadget (
    AltingChannelInput click, AltingBarrier group,
    ChannelOutput configure
  ) {
    this.click = click;
    this.group = group;
    this.configure = configure;
  }

  ...  public void run ()

}
```



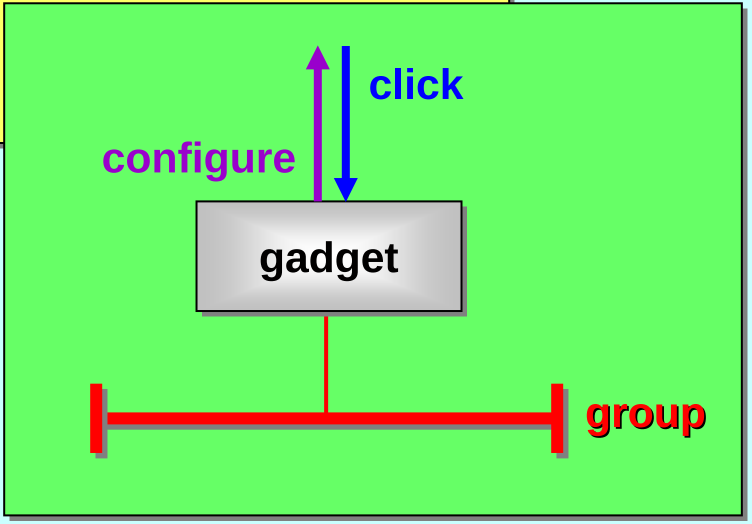
Alting Barriers – Example

```
public void run () {

  final Alternative clickGroup =
    new Alternative (new Guard[] {click, group});

  final int CLICK = 0, GROUP = 1;

  int count = 0;

  while (true) {
    ...   individual mode
    ...   group mode
  )

}
```



Alting Barriers – Example

```
{{{  individual mode

  configure.write (Color.green);

  while (!click.pending ()) {
    count++;
    configure.write (String.valueOf (count));
  }
  click.read ();

}}}
```



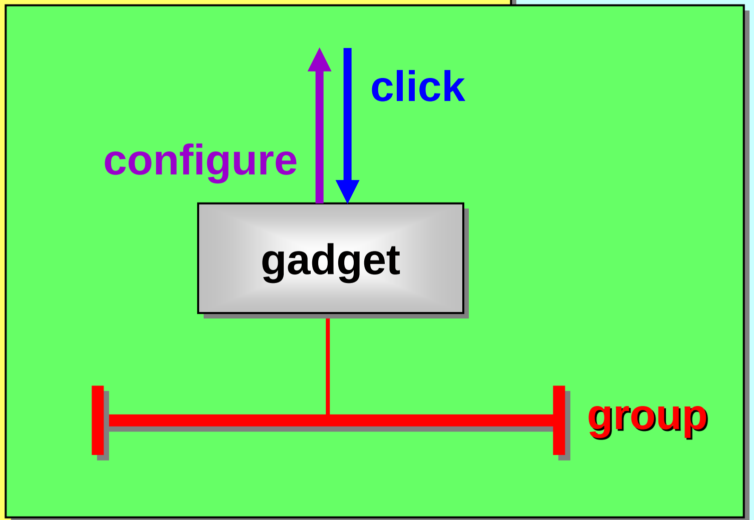
Alting Barriers – Example

```
{{{  group mode

  configure.write (Color.red);

  boolean group = true;
  while (group) {
    switch (clickGroup.priSelect ()) {
      case CLICK:
        click.read ();
        group = false;
      break;
      case GROUP:
        count--;
        configure.write (
          String.valueOf (count)
        );
      break;
    }
  }

}}}
```

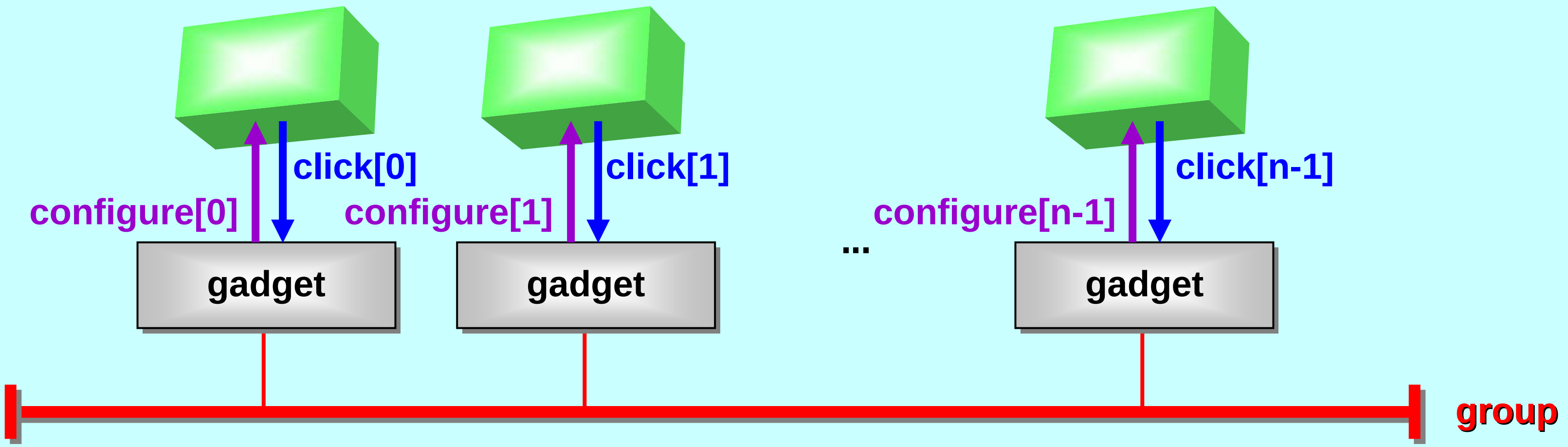
offer to work with the group

drop out of group

group work

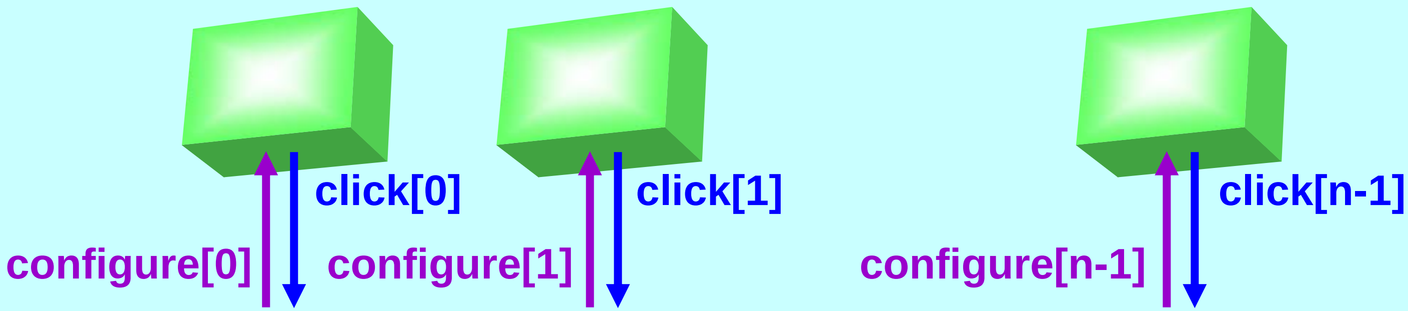


Alting Barriers – Example




```
final int n = 8;

...  make the buttons (and its configure and click channels)

...  make the AltingBarrier (front-ends)

...  make the gadgets

new Parallel (
  new CSProcess[] {
    buttons, new Parallel (gadgets)
  }
).run ();
```

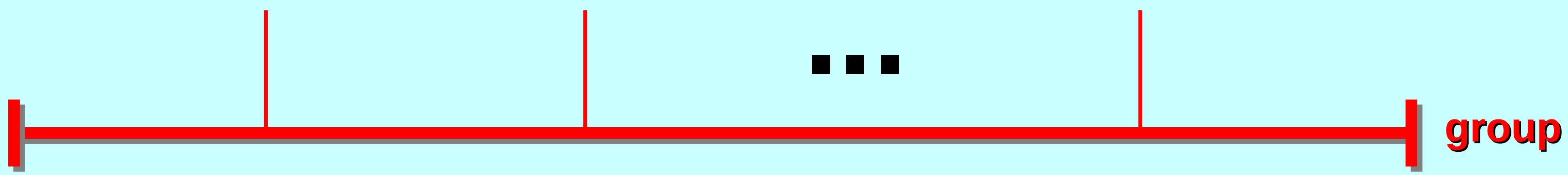
Alting Barriers – Example




```
{{{  make the buttons (and its configure and click channels)
final One2OneChannel[] event = Channel.one2oneArray (n);
final One2OneChannel[] configure = Channel.one2oneArray (n);
final boolean horizontal = true;
final FramedButtonArray buttons =
  new FramedButtonArray (
    "AltingBarrier: GadgetDemo", n, 120, n*100,
    horizontal, configure, event
  );
}}}
```

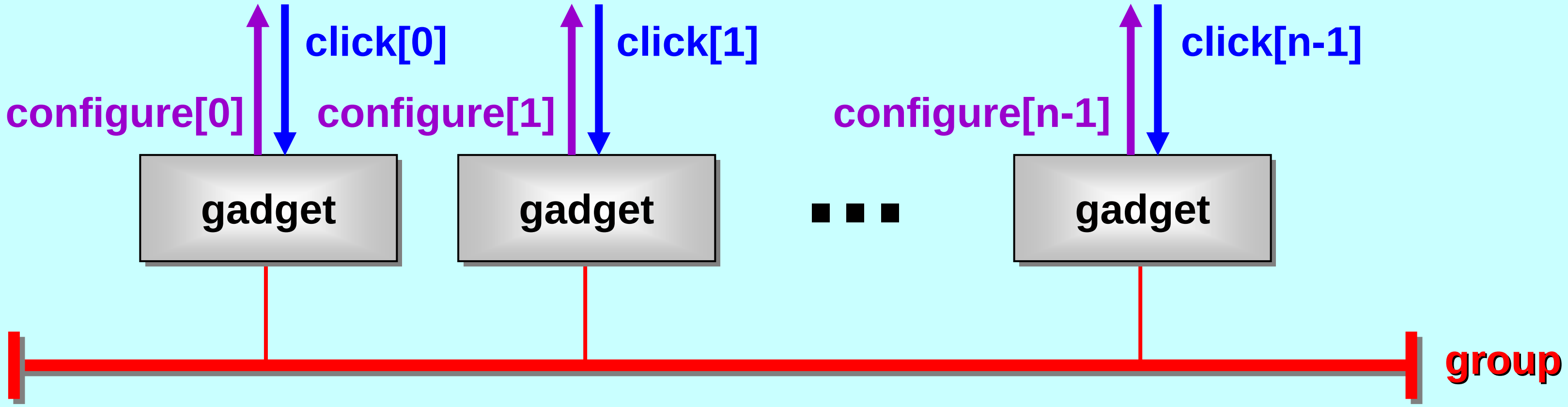
jcsp.plugNplay

Alting Barriers – Example



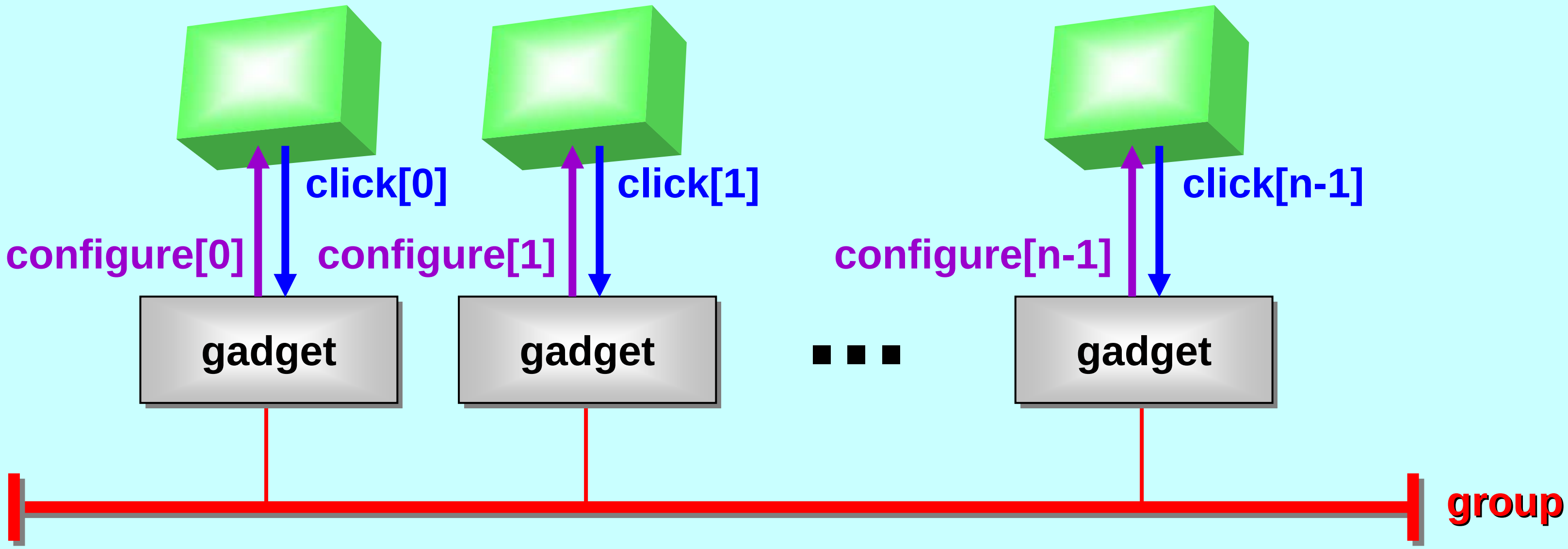

```
{{{  make the AltingBarrier (front-ends)

final AltingBarrier[] group = AltingBarrier.create (n);

}}}
```

Alting Barriers – Example



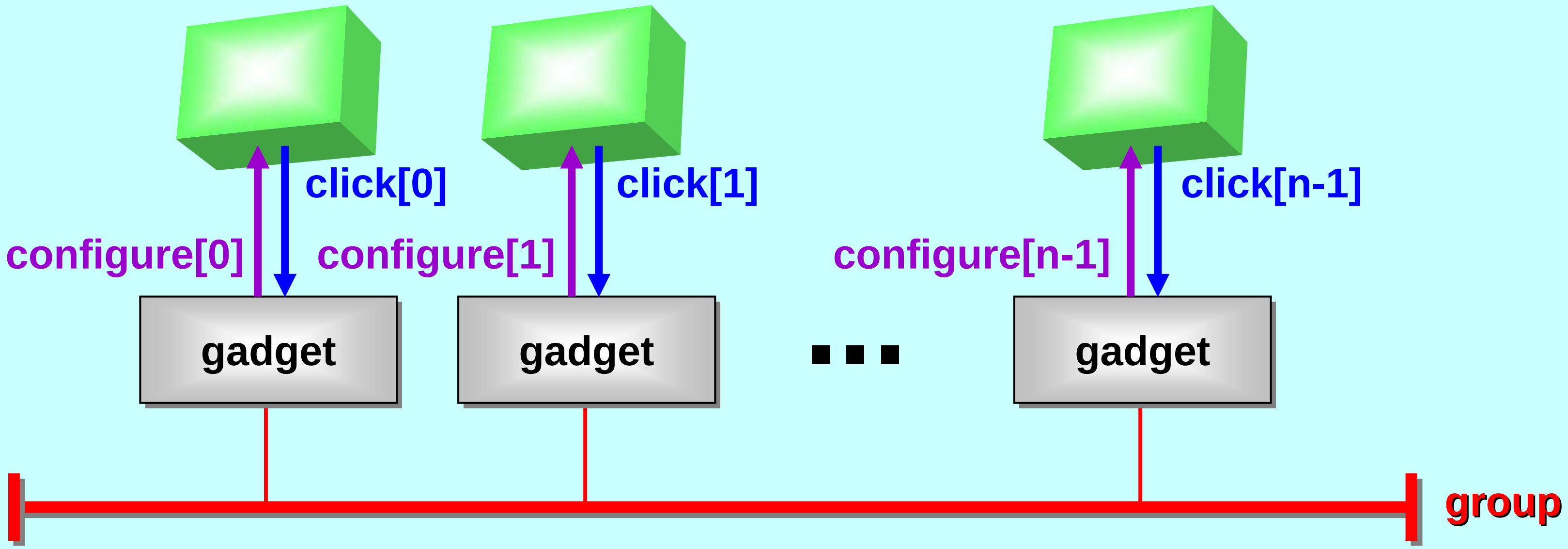

```
{{{  make the gadgets

final Gadget[] gadgets = new Gadget[nUnits];

for (int i = 0; i < gadgets.length; i++) {
  gadgets[i] = new Gadget (event[i], group[i], configure[i]);
}

}}}
```

Alting Barriers – Example




```
{{{  run everything

new Parallel (
  new CSProcess[] {
    buttons, new Parallel (gadgets)
  }
).run ();

}}}
```

Alting Barriers – Example



This example has only a single alting barrier. The *JCSP* documentation provides many more examples – including systems with intersecting sets of processes offering multiple multiway barrier synchronisations (one for each set to which they belong), together with timeouts and ordinary channel communications. *There are also some games … ☺ ☺ ☺.*

Alting Barriers – Implementation

The fast *Oracle* for choice over multiway synchronisations is a server database holding information for each barrier and for each process enrolled on a barrier. Its decisions have time complexity linearly dependent on the number of barriers offered *– it does not use a two-phase commit protocol*.

A process atomically offers the *Oracle* a set of barriers with which it is prepared to engage and blocks until the *Oracle* tells it which one has been breached.

The *Oracle* simply keeps counts of, and records, all the offer sets as they arrive. If a count for a particular barrier becomes complete (i.e. all enrolled processes have made an offer), it informs the lucky waiting processes and atomically withdraws all their other offers *– before considering any new offers*.

Alting Barriers – Implementation

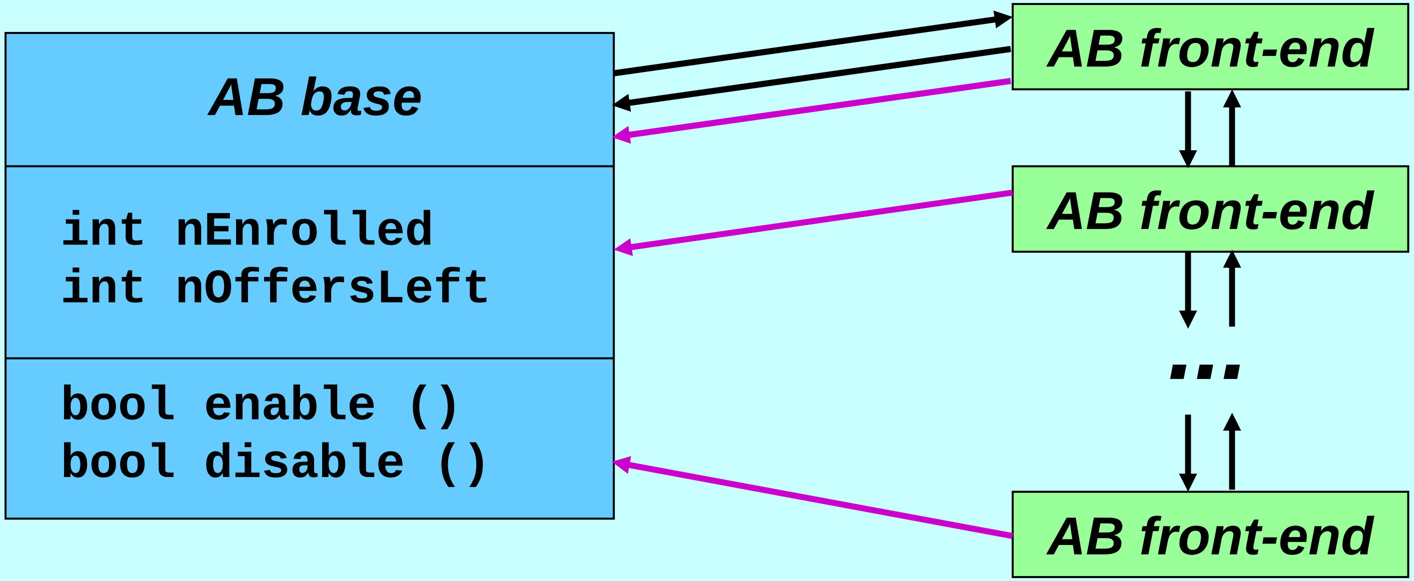
For *JCSP*, the *Oracle* mechanism needs adapting to allow processes to make offers to synchronise that include all varieties of **`Guard`** – not just **`AltingBarrier`**s.

The logic of the single *Oracle* process is also distributed to work with the usual enable/disable sequences implementing the select methods invoked on **`Alternative`**. These sequences already record all the offers that have been made – so we just need to maintain countdowns for each **`AltingBarrier`**.

The techniques used here for *JCSP* carry over to a similar notion of *alting barriers* for an extended **occam-π**.

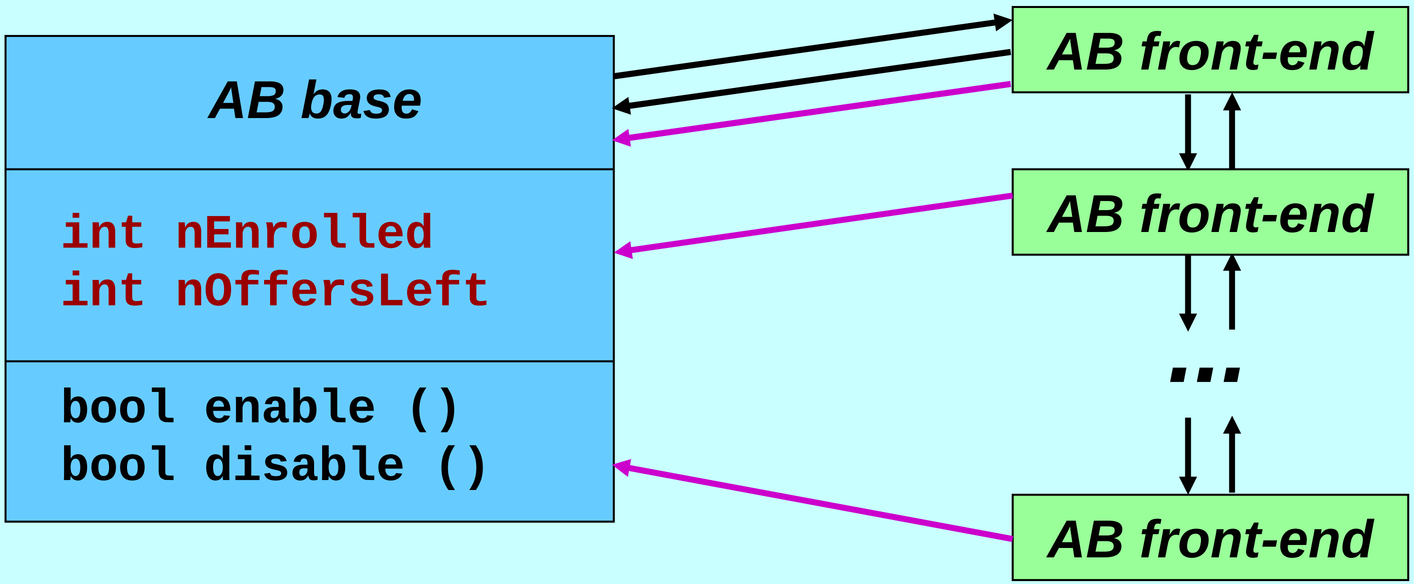
Alting Barriers – Implementation

The **`AltingBarrier.create(n)`** method first constructs a hidden base object *– the actual alting barrier –* before constructing and returning the array of **`AltingBarrier`** front-ends. These front-ends reference the base and are chained together. The base object is not shown to JCSP users and holds the first link to the chain of front-ends.



Alting Barriers – Implementation

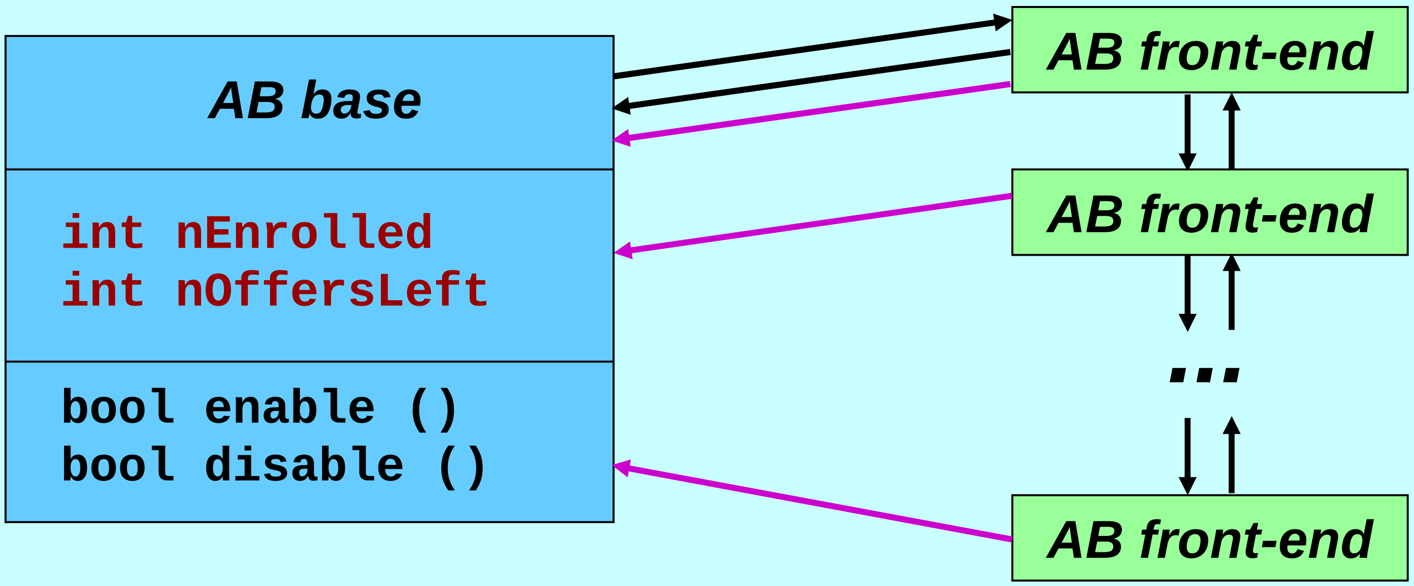
The `AltingBarrier` front-ends delegate their `enable()` and `disable()` to the base. The base `enable()` decrements its `nOffersLeft` count and, if zero, resets it to `nEnrolled` and returns `true`. The `disable()` returns `true` if `nOffersLeft` equals `nEnrolled` – otherwise, it increments `nOffersLeft` and returns `false`.



Alting Barriers – Implementation

For the *Oracle* logic to work, each full offer set from a process to all its guards must be handled *automically*.

A global lock, therefore, must be obtained and held throughout any *enable* sequence involving an **`AltingBarrier`**.



Alting Barriers – Implementation

For the *Oracle* logic to work, each full offer set from a process to all its guards must be handled *automically*.

A global lock, therefore, must be obtained and held throughout any *enable* sequence involving an **`AltingBarrier`**.

If the *enables* all fail, the lock must be released before the alting process blocks.

If a barrier *enable* succeeds, the barrier is complete and selected – ignoring any higher priority guards that may become *enabled* later. The lock must continue to be held throughout the consequent *disable* sequence *and* throughout the *disable* sequences of all the other processes that are enrolled on this barrier (triggered by the successful *enable*). *This lock needs to be a counting semaphore.*

Disable sequences (triggered by the successful *non-barrier enable*) do not need to acquire this lock – even if an **`AltingBarrier`** guard is in the list.

Alting Barriers – Implementation

Take care ...

The logic required for a correct implementation of CSP external choice is never easy ...

The JCSP version just for *channel input* synchronisation required *formalising and model checking* before we got it right.

Our implementation has not (yet) been observed to break under stress testing, but we shall not feel comfortable until this has been repeated for these multiway events. Full LGPL source codes are available from the JCSP website.

Talk roadmap ...

History …

Explicit channel "ends" …

Alting barriers …

Output guards …

Extended rendezvous …

Poison …

Future (broadcast channels, generics, networking) ...

Output Guards

Channel *output guards* were not supported by CSP languages or libraries for the same reason that general *multiway sync guards* were not supported – they enable more than one party to a synchronisation to withdraw, which spoils implementation via simple handshake.

However:

`One2OneChannel + AltingBarrier`
`= SymmetricOne2OneChannel`

A **`SymmetricOne2OneChannel`** is the same as an ordinary **`One2OneChannel`** – except that both its *input* and *output ends* may be offered as *guards* in an ALT.

Output Guards

`One2OneChannel` + `AltingBarrier`
= `SymmetricOne2OneChannel`

A **`SymmetricOne2OneChannel`** consists of a **`One2OneChannel`** and an *alting barrier* with two *front-ends* (**`AltingBarrier`**s) – one for the *input-end* of the channel and one for the *output-end*.

Offering the *input-end* of the channel simply means offering to synchronise on the *input-end* **`AltingBarrier`**. If selected, the **`read()`** operation is then delegated to the **`One2OneChannel`**.

Offering the *output-end* of the channel simply means offering to synchronise on the *output-end* **`AltingBarrier`**. If selected, the **`write()`** operation is then delegated to the **`One2OneChannel`**.

Output Guards

`One2OneChannel` + `AltingBarrier`
= `SymmetricOne2OneChannel`

A **`SymmetricOne2OneChannel`** consists of a **`One2OneChannel`** and an *alting barrier* with two *front-ends* (**`AltingBarrier`**s) – one for the *input-end* of the channel and one for the *output-end*.

A *non-alting (i.e. committed)* **`read()`** or **`write()`** operation must still be prefixed by a *(committed)* synchronisation on the *alting barrier* – because neither side knows whether the other party is actually committed!

This is a direct application of ideas and theorems proven in Alistair McEwan's thesis (and presented at CPA 2005).

Talk roadmap …

History …

Explicit channel “ends” …

Alting barriers …

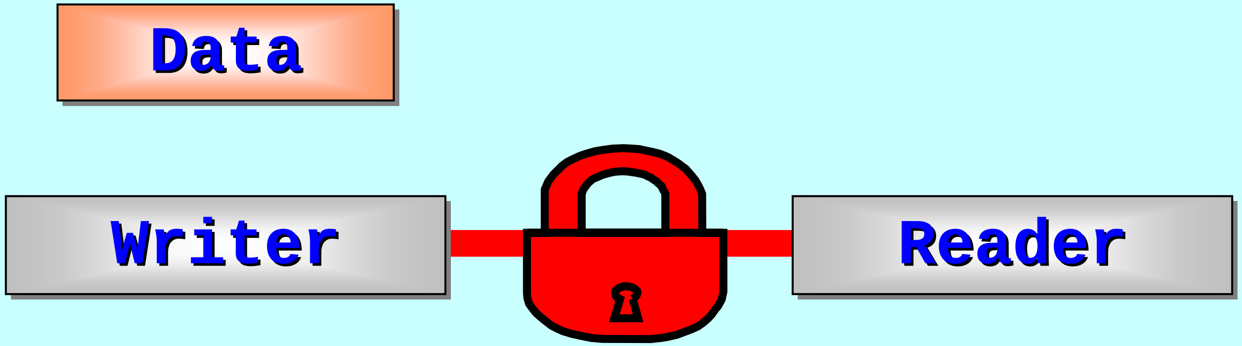
Output guards …

Extended rendezvous …

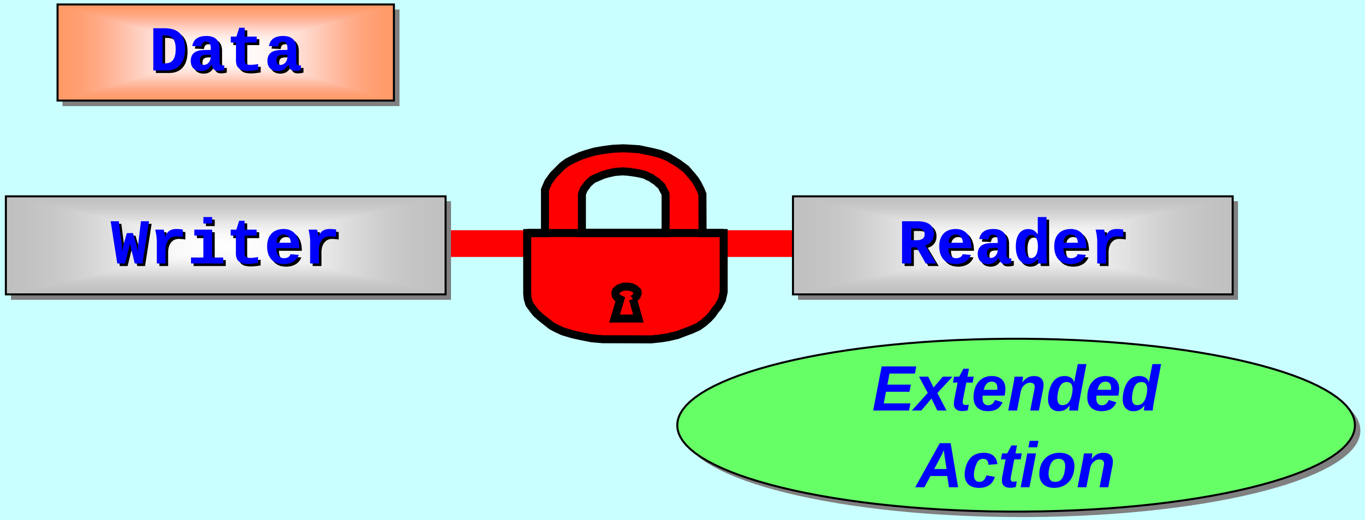
Poison …

Future (broadcast channels, generics, networking) ...

Standard Communication



Extended Rendezvous



Extended Rendezvous API

`ChannelInput` has two new methods:

```
Object startRead();
void endRead();
```

For example:

```
Object x = c0.startRead();
System.out.println(x);
c1.write(x);
c0.endRead();
```

Extended Action

Buffered Extended Rendezvous

- Extended rendezvous is now allowed on buffered channels.

- FIFO

  `startRead()` only "peeks" on FIFO buffers
  `endRead()` then removes

- Overwriting

  `startRead()` gets and removes
  `endRead()` does nothing

Talk roadmap …

History …

Explicit channel "ends" …

Alting barriers …

Output guards …

Extended rendezvous …

Poison …

Future (broadcast channels, generics, networking) ...

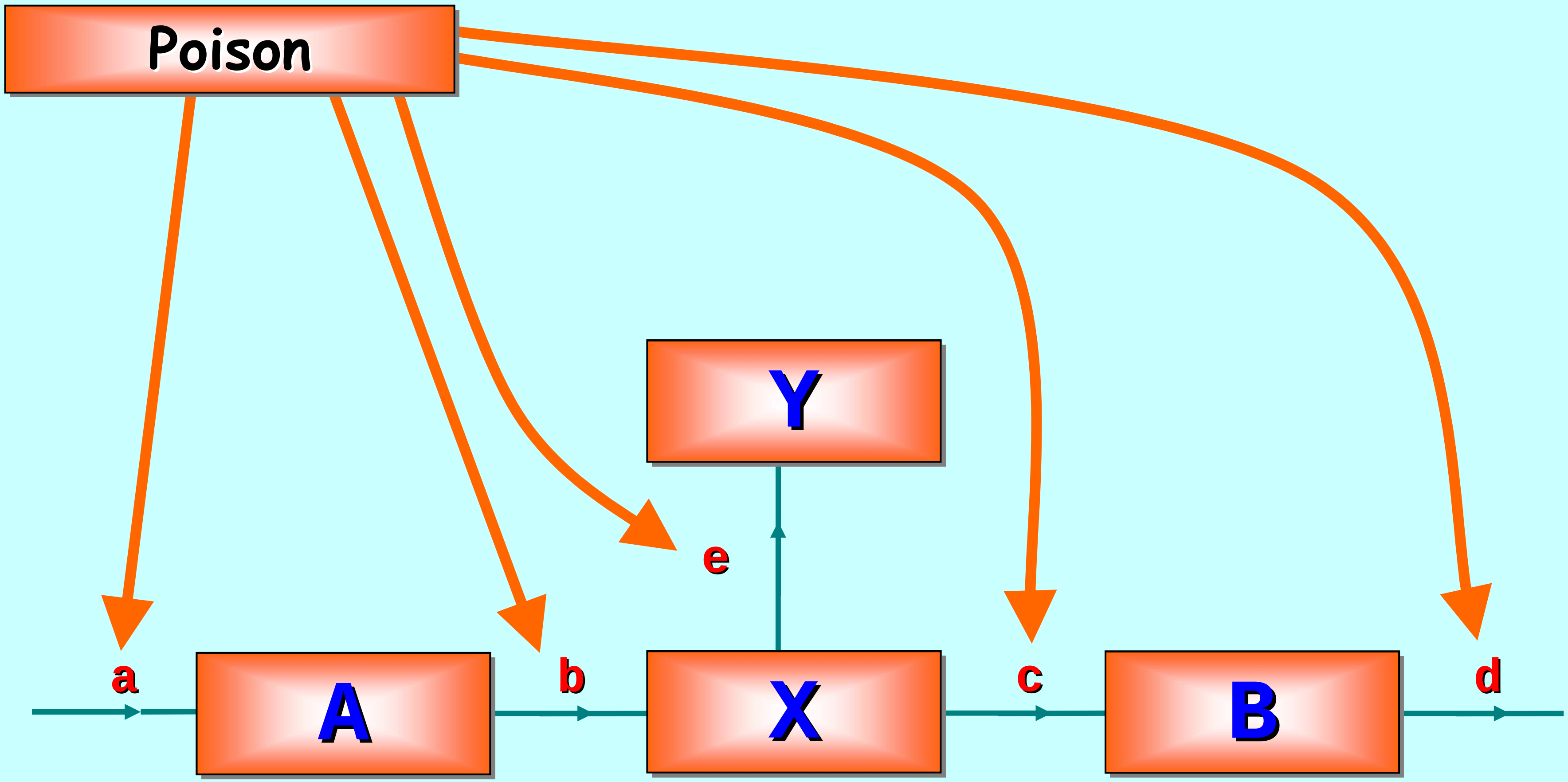
Poison

- Used for terminating process networks.

- Poison renders a channel unusable …
  - No antidote

- Attempting to use a poisoned channel throws a *poison exception* in the using process …
  - Normal action on catching a poison exception:
    - Poison all channel-ends
    - Terminate

Poison Propagation

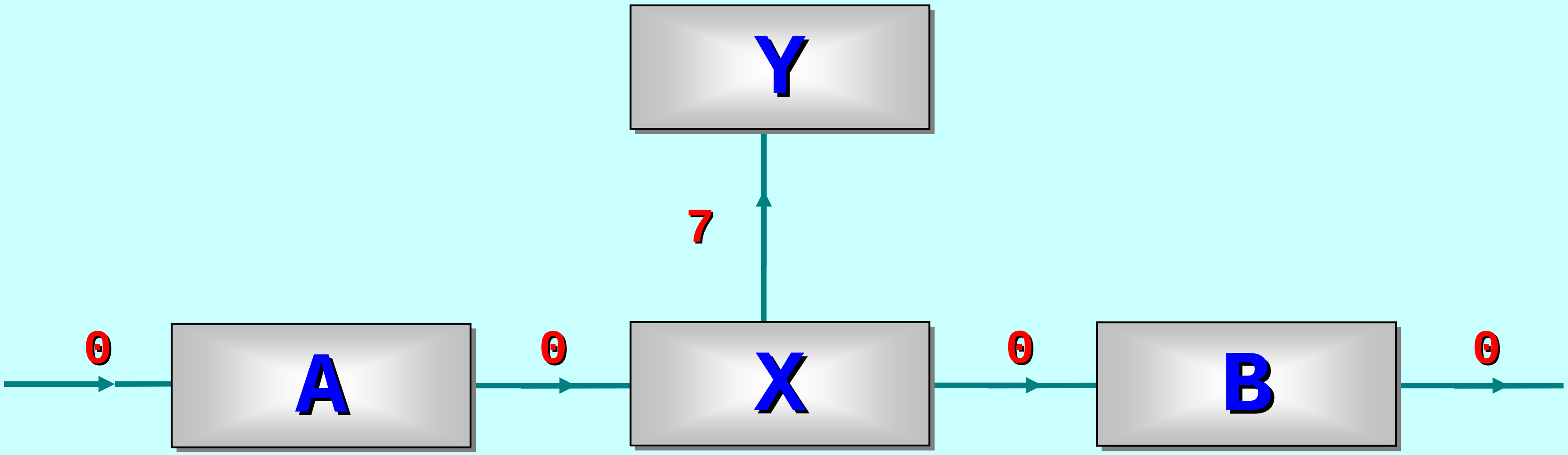


Poison

- *JCSP* introduces poison *strength* and channel *immunity*
  - Each channel-end has a level of *immunity*:
    - It only succumbs to poison stronger than its immunity
    - Used to contain network poisoning within sub-regions
  - Poison strength propagates throughout network:
    - Normally, a process poisons with the strength of the poison in the channel it tried to use.
    - This can result in *non-deterministic* behaviour if two (or more) wave fronts of poison are spreading at the same time.
    - Propagation may depend on the strength of the poison wave front that hits a process first.

Poison Non-determinism

Here's a happy system …

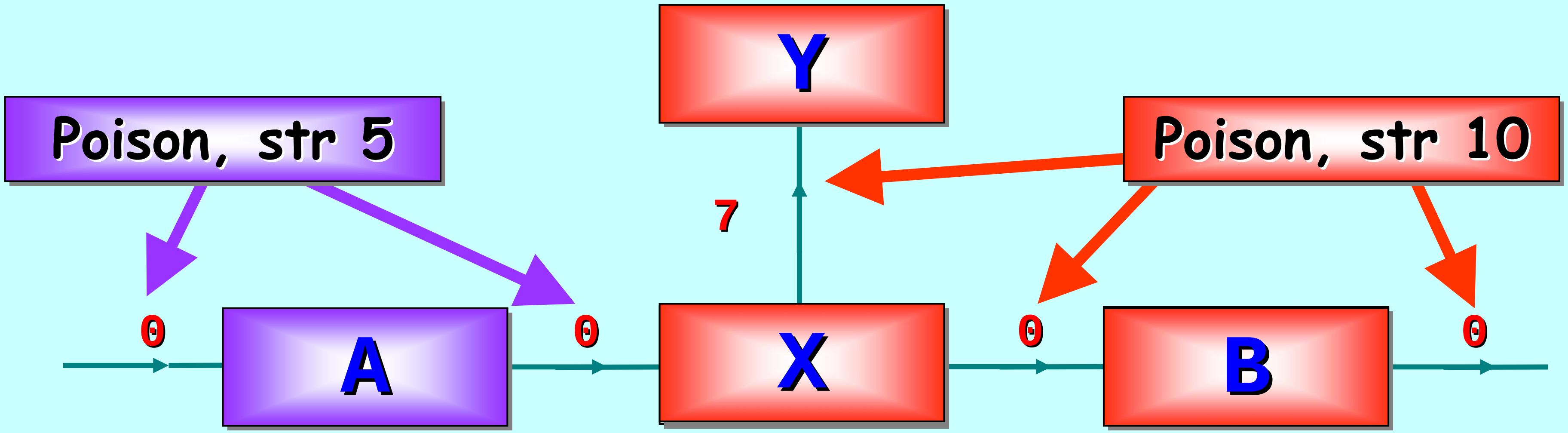


The channels are labelled with their immunity levels …

Schedule 1: all are poisoned

Poison (strength 10) hits B, then X, then Y … all terminate.

Then, poison (strength 5) hits A … but no further (X is dead).

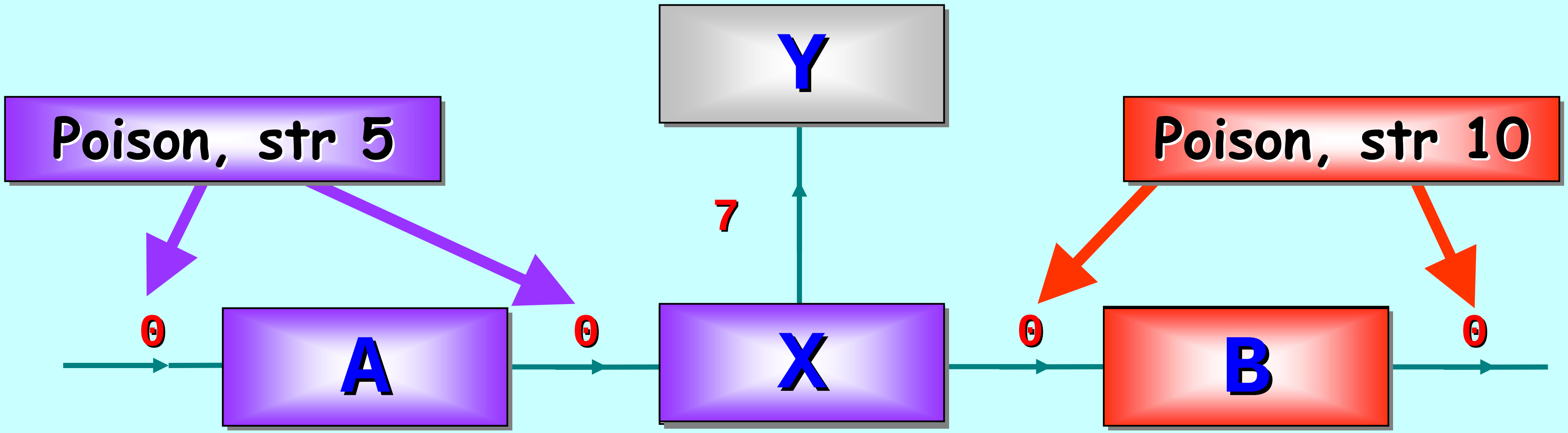


The channels are labelled with their immunity levels …

Schedule 2: one survives

Poison (strength 5) hits A, then X … but can't reach Y.

Then, poison (strength 10) hits B … but no further (X is dead).



The channels are labelled with their immunity levels …

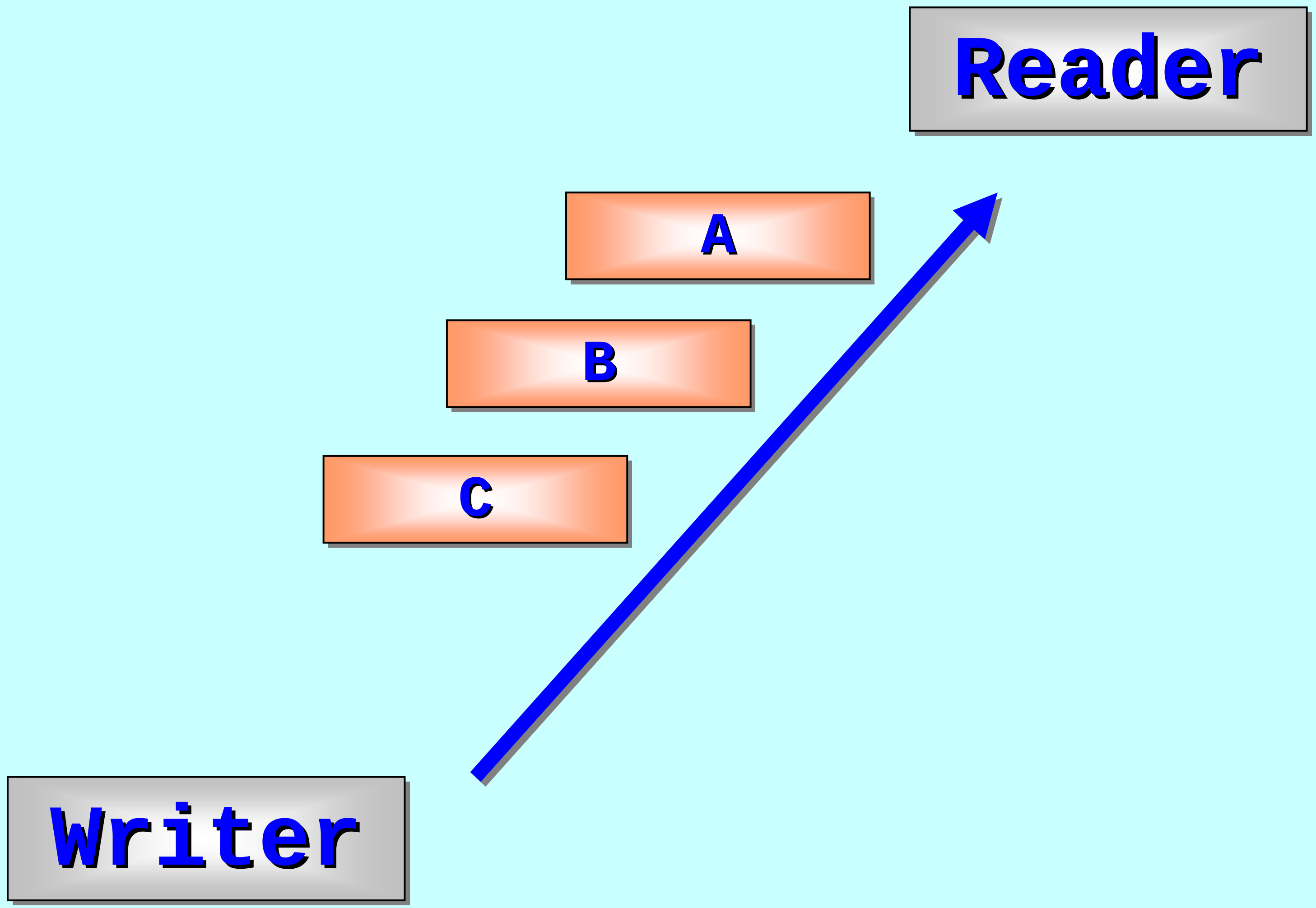
Poison API

- Channel-ends have a new method:
  - `void poison (int strength)`

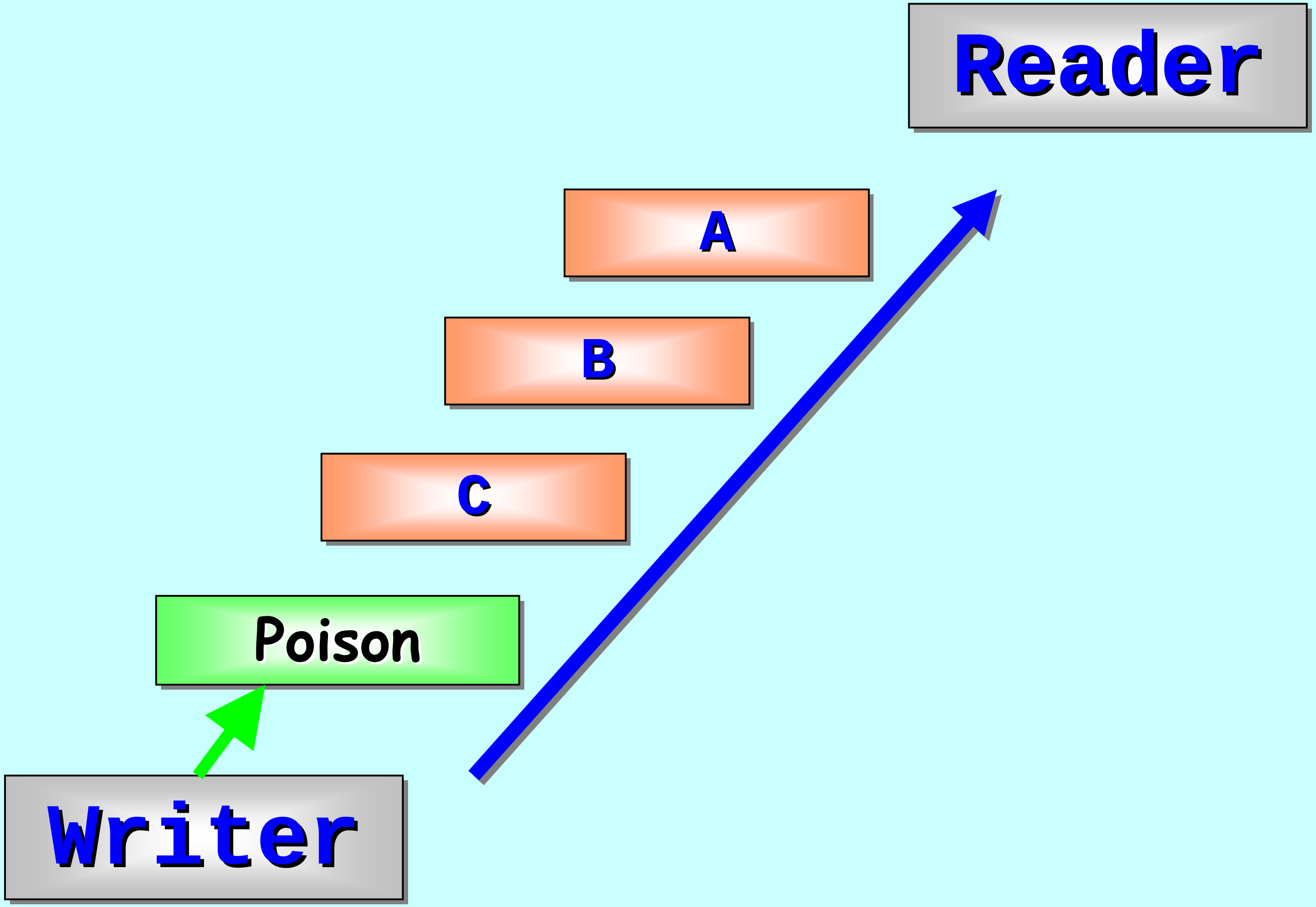
- All other channel methods may now throw a `PoisonException`
  - only if poisoned channels are used (not mandatory)
  - `PoisonException` has a `getStrength()` method

- Implementation uses Sputh's algorithm.

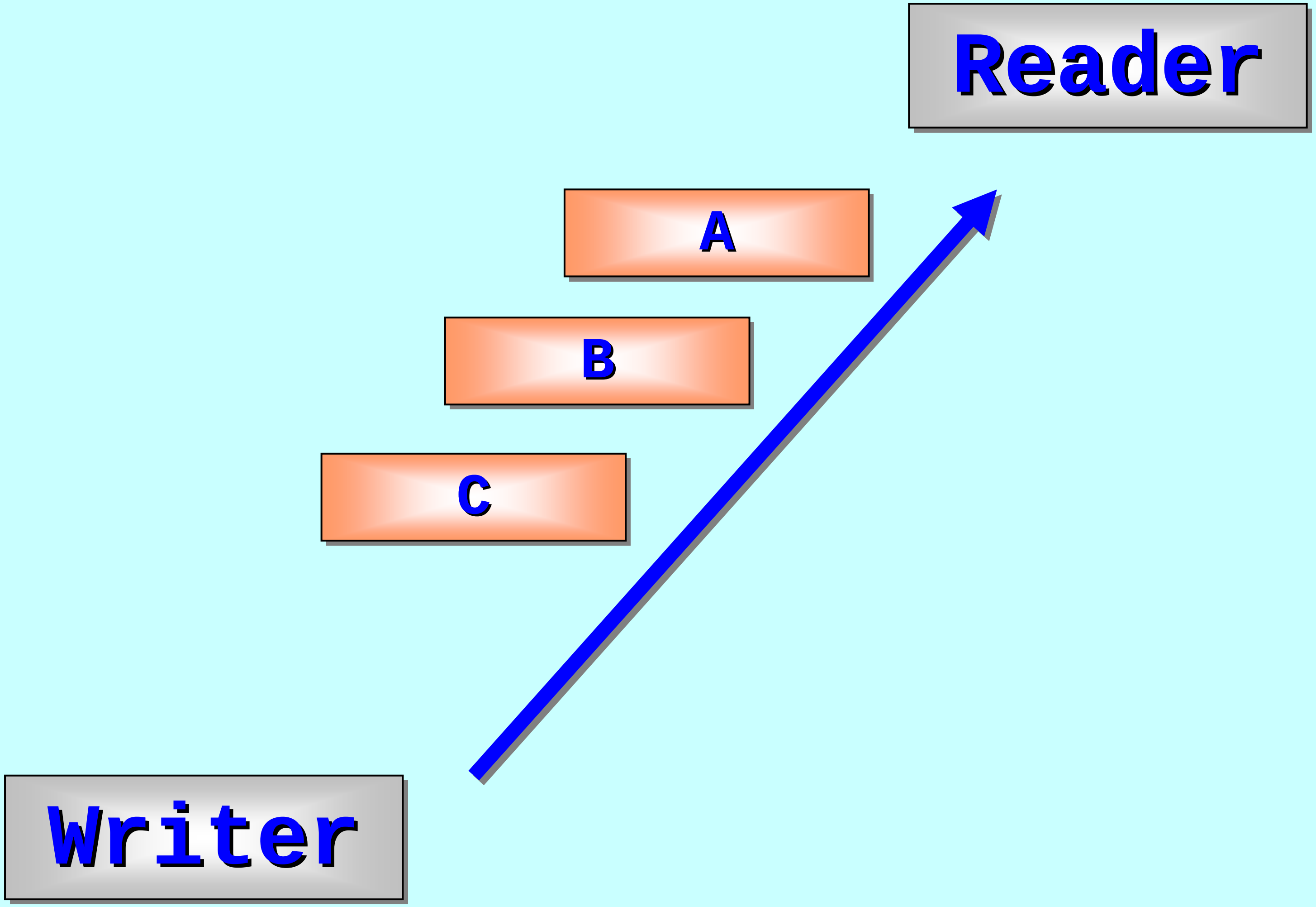
Poison, Buffered Channels



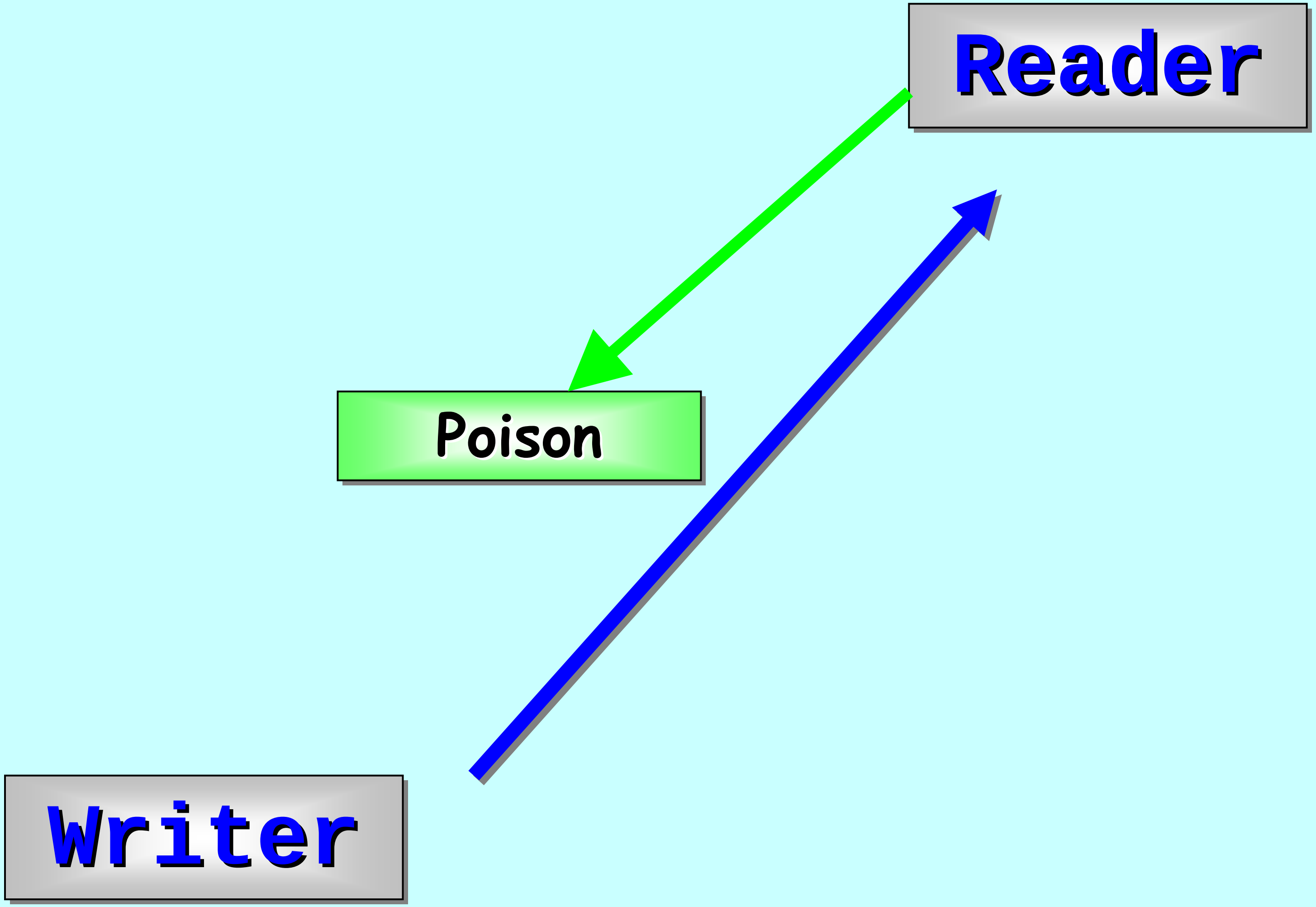
Poison, Buffered Channels



Poison, Buffered Channels



Poison, Buffered Channels



Talk roadmap …

History …

Explicit channel "ends" …

Alting barriers …

Output guards …

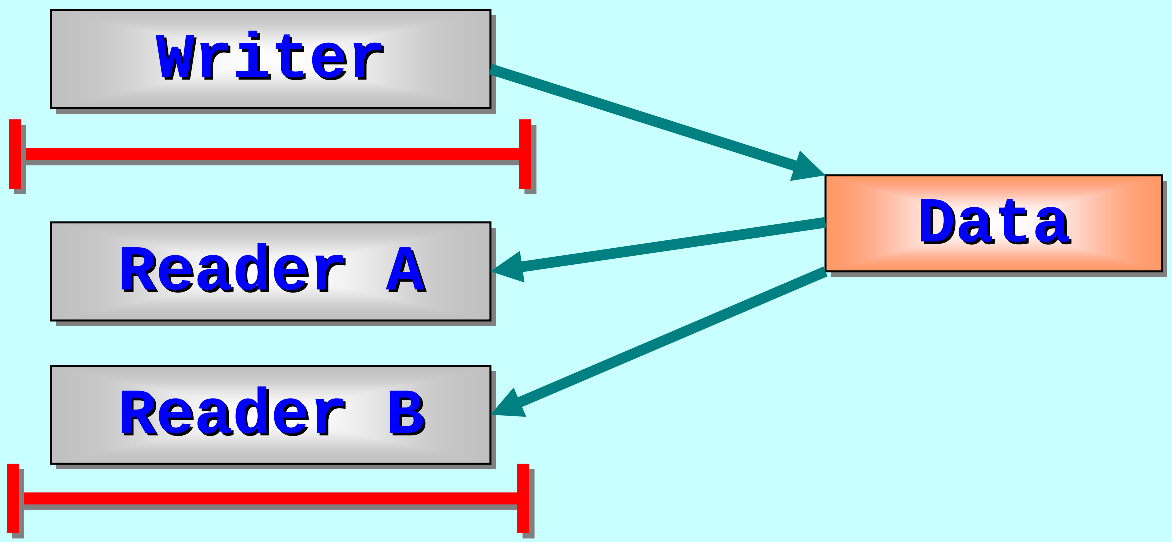
Extended rendezvous …

Poison …

Future (broadcast channels, generics, networking) ...

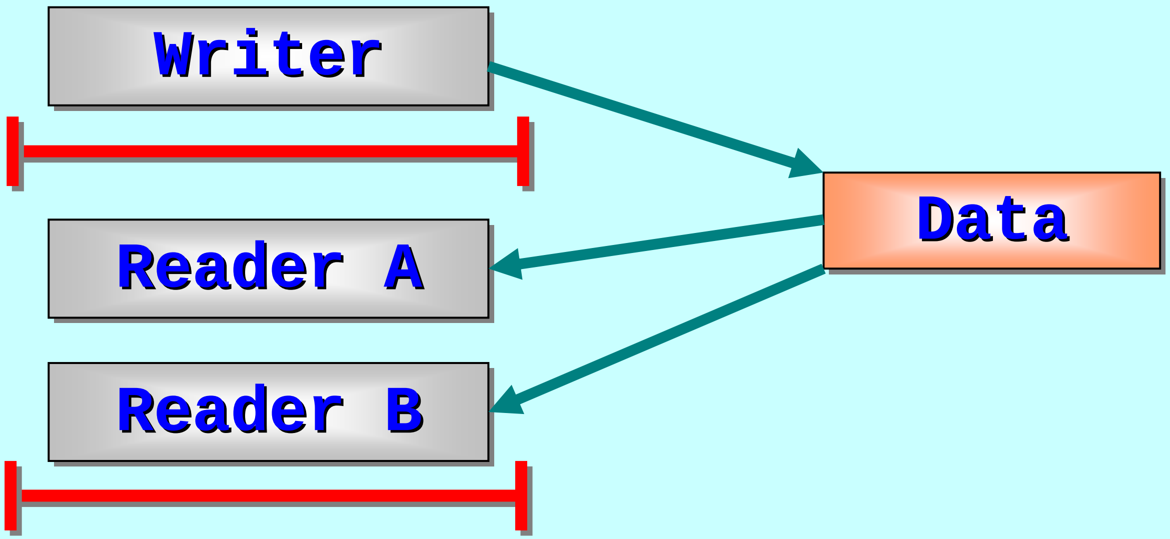
Broadcast Channels

- *One-to-**many*** channels
- Implemented with a write phase, then read phase:
  - enforced by barrier synchronisation.



Broadcast Channels

- *ALTing* should be possible (via ALTing Barriers)
- Poison needs more work
  - Need to make barriers poisonable



Java 1.5

- Channels could use *generics*
  - like C++CSP's templated channels

- New java.util.concurrent package
  - has channel-like objects
    - but no ALTing!
  - has a barrier object
    - but no dynamic enrollment / resignation
    - or ALTing
  - has very low level atomic operations (e.g. CAS)
    - consider re-implementing JCSP sync primitives using these
    - may win some performance

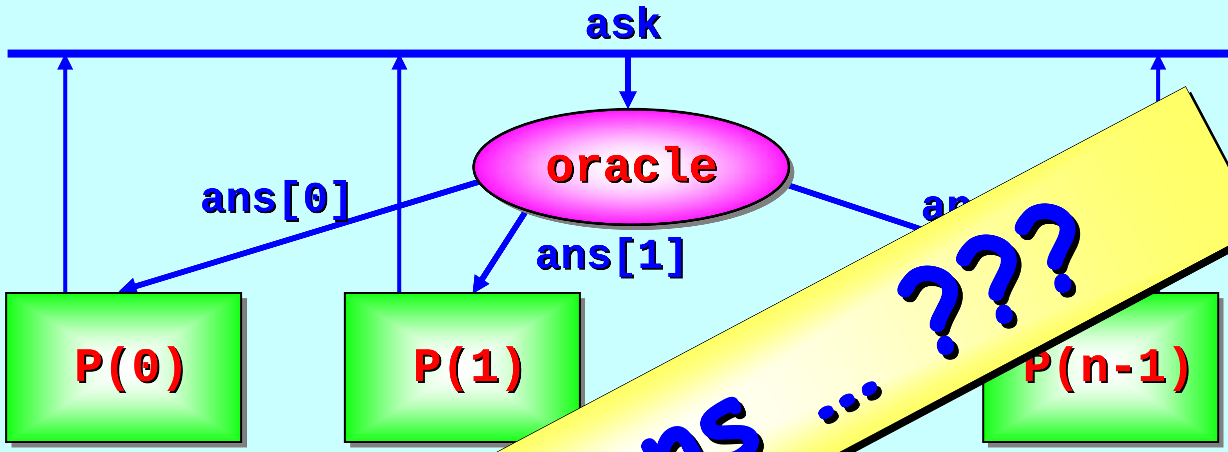
Networking

- Networked barriers
  - currently only supported within a single JVM
- Networked *ALTing* barriers
  - distribute the *Oracle* structures?
    - implies network traffic for each enable/disable ☹
  - use correct two-phase commit protocol
    - may imply as much network traffic as above ☹
    - plus cancelation overheads ☹
  - combine local *Oracle* logic with the two-phase commit
    - fast local synchronisation with secure global synchronisation
    - imposing network traffic only when local syncs complete
    - *tricky !!!*

Summary

- Class re-organisation (internal), channel-ends, new API (for channel creation)
- ALTing barriers
  - Symmetric channels (output guards)
- Extended Rendezvous
- Poison
- Network integrated and extended (JCSP 1.1) released
- See paper for attribution and thanks (lots!)

Resolution Oracle: occam-π



Any questions … ???

```
PROTOCOL ORACLE.AS                     FER:

PROTOCOL O             S INT; OFFER:

PROC O        (MOBILE []ENROLLED enrolled,
               CHAN ORACLE.ASK ask?,
               []CHAN ORACLE.ANS ans!)
  ...
:
```

who is asking

the chosen event

returned offer

setup care needed