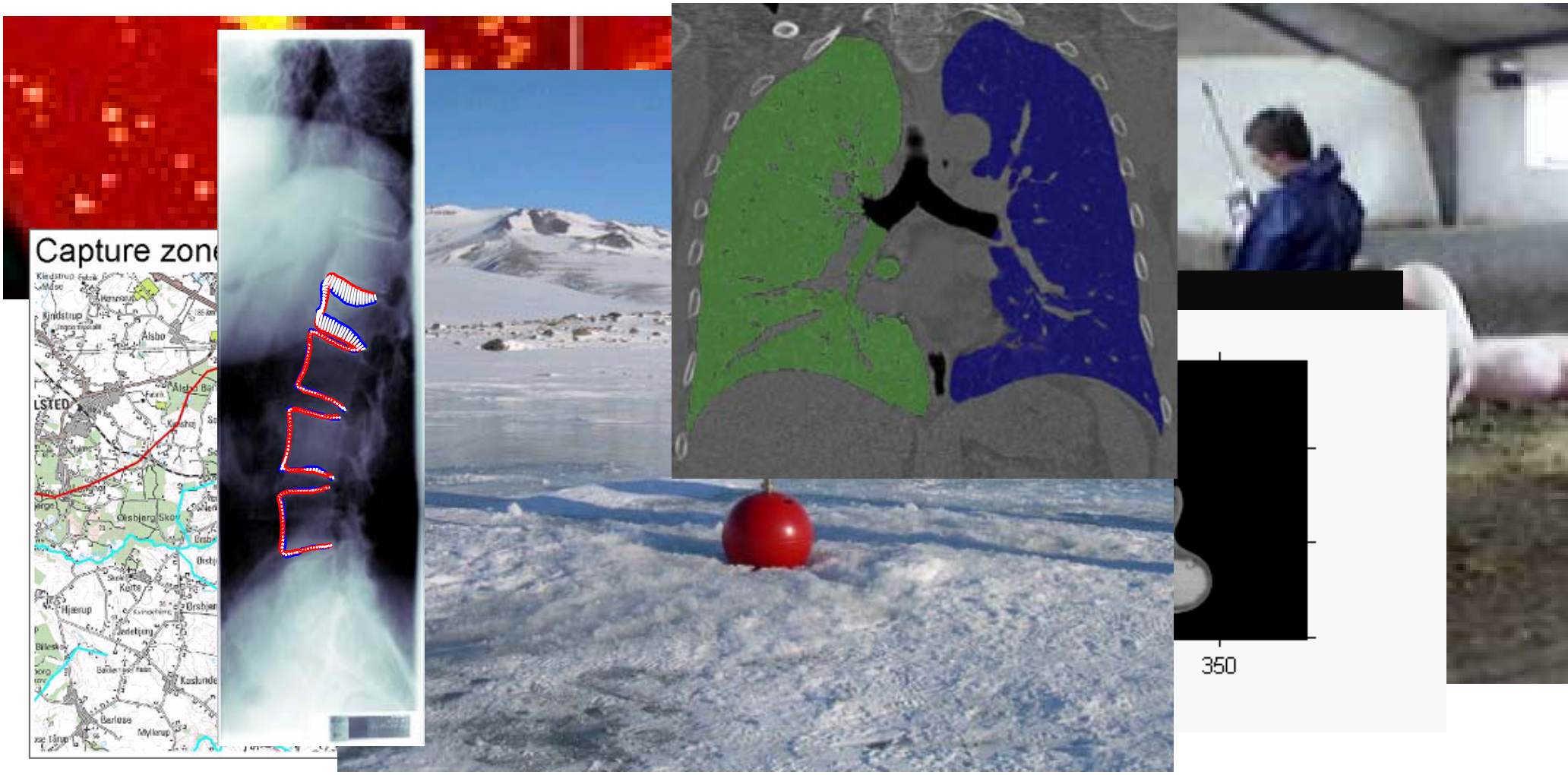
PyCSP Revisited

Brian Vinter
John Markus Bjørndalen
Rune Møllegaard Friborg

Target domain



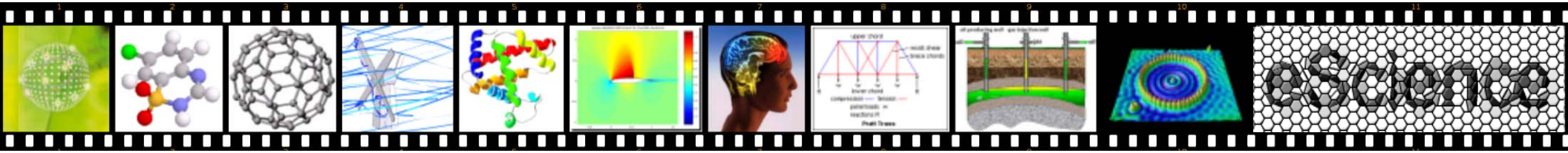
History of PyCSP

Started at CPA 2006

Presented at CPA 2007

- Based on Python (OS) threads

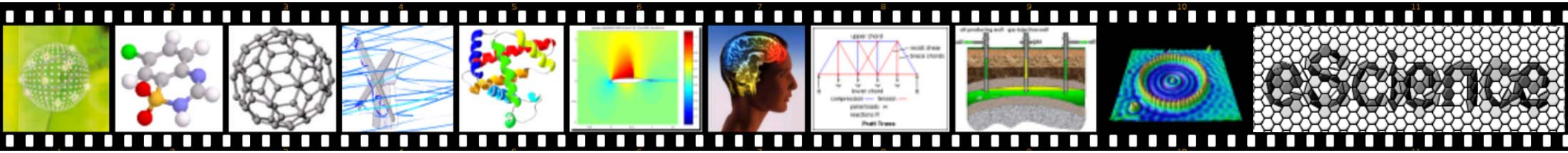
A GUI and multiple minor additions in 2008



Reality check

Live or die for PyCSP?

- \- The exercise was done
- \- GIL reduces all applications to serialized execution
- \- OS limits reduces the number of threads significantly
- \+ Is is very popular amongst our own students
- \+ Python is growing in popularity amongst “scientists as programmers”



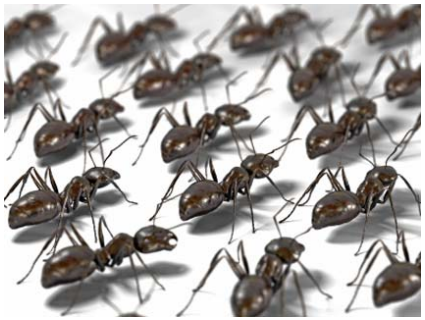
A look at the users and applications

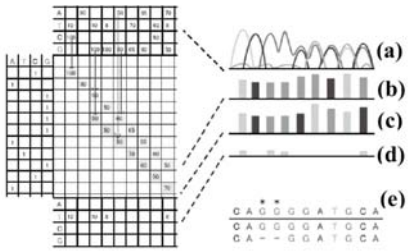
Mostly CS students

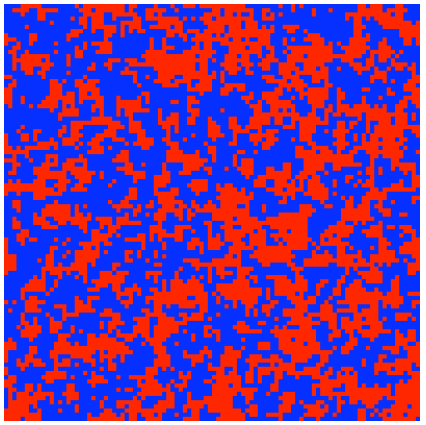
- But a sizable number of “science” students also

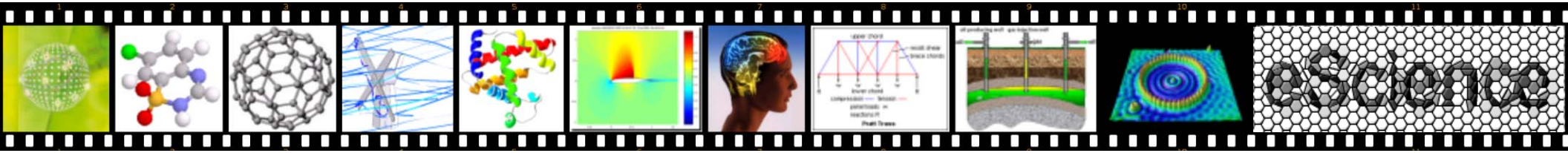
Predominantly scientific applications are build using PyCSP

- This is what the class the introduce PyCSP focus on
- It is also where the need for parallelism is highest









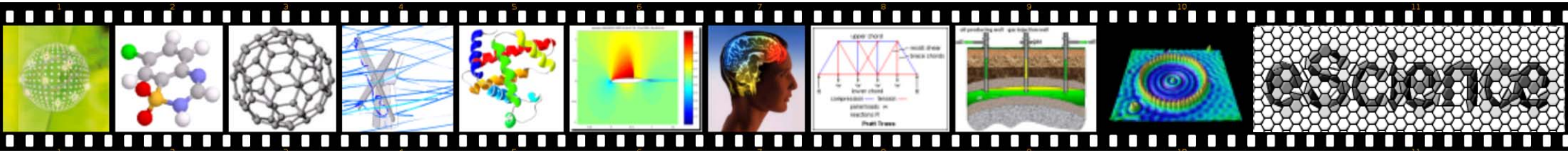
The verdict is “live”

We chose to let PyCSP live

- Which means invest more effort in the package

After reviewing many (~200!) student reports and comments we decided to revise PyCSP on four points:

- There should be only one channel type, any-to-any, and it must support external choice
- The channels should support both input and output guards for external choice
- PyCSP should provide a mechanism for joining and leaving a channel with support for automatic poisoning of a network
- The expressive power in Python should be used to make PyCSP look more like occam where possible

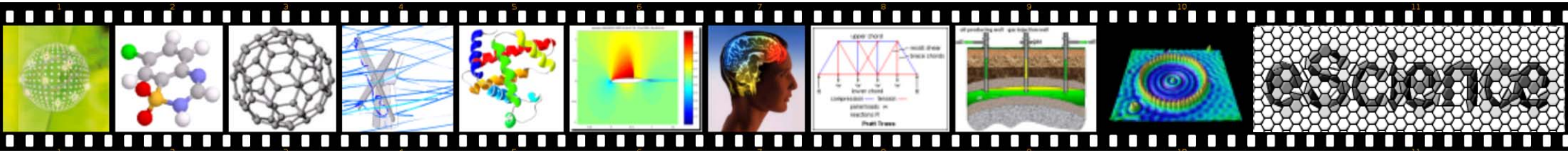


Processes

At first glance processes have not changed since the first version

However the Parallel construct now supports a combination of scalars and lists

```
@process
def hello_world (msg ):
    print " Hello world , this is my message " + msg

Parallel (
    source (),
    [ worker () for i in range (10)] ,
    sink ()
)
```

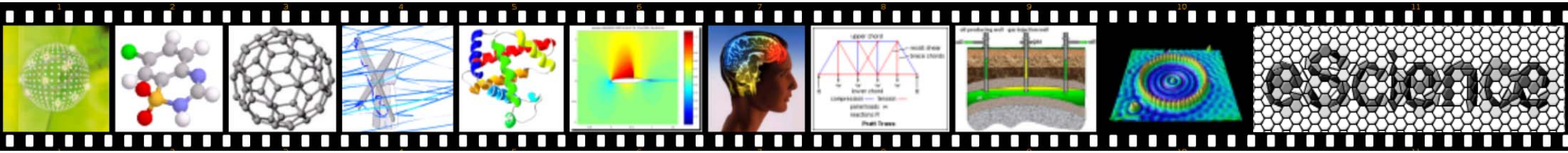


Processes

At first glance processes have not changed since the first version

However the Parallel construct now supports a combination of scalars and lists

```
@process
def hello_world (msg ):
    print " Hello world , this is my message " + msg

Parallel (
    source (),
    [ worker() for i in range(10)] ,  10*worker()
    sink ()
)
```



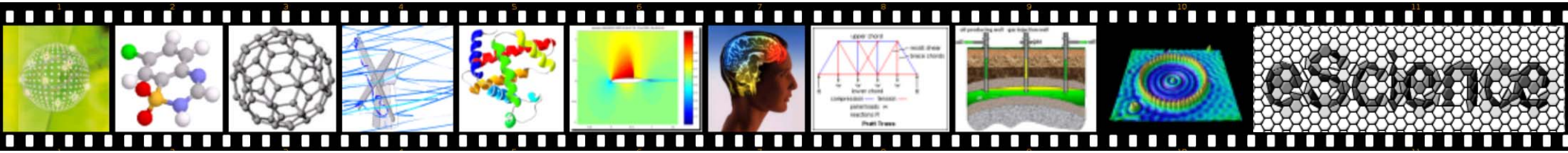
Channels

In programming and in engineering the use of different channels makes sense

- In science they become a nuisance

Any process that has a given channel in its context may ask for a channel-end from that channel

- Input or output end



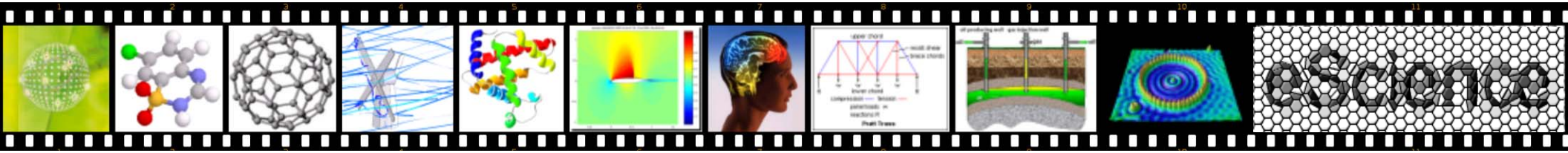
Channels

Channels are easily defined

- `my_channel = Channel ()`

Channel ends are obtained by requesting an input or output end

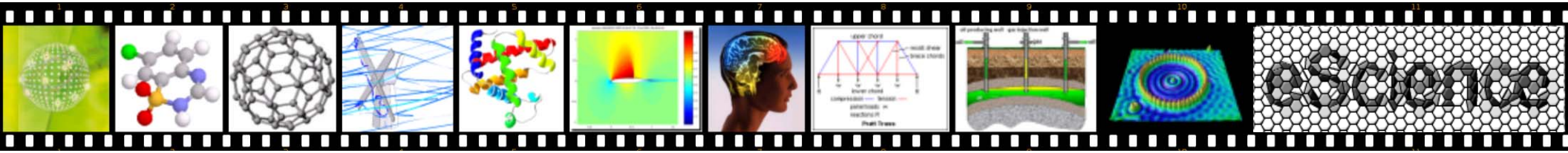
- `my_reader = my_channel.reader()`
  - `my_reader = +my_channel`
- `my_writer = my_channel.writer()`
  - `my_writer = -my_channel`



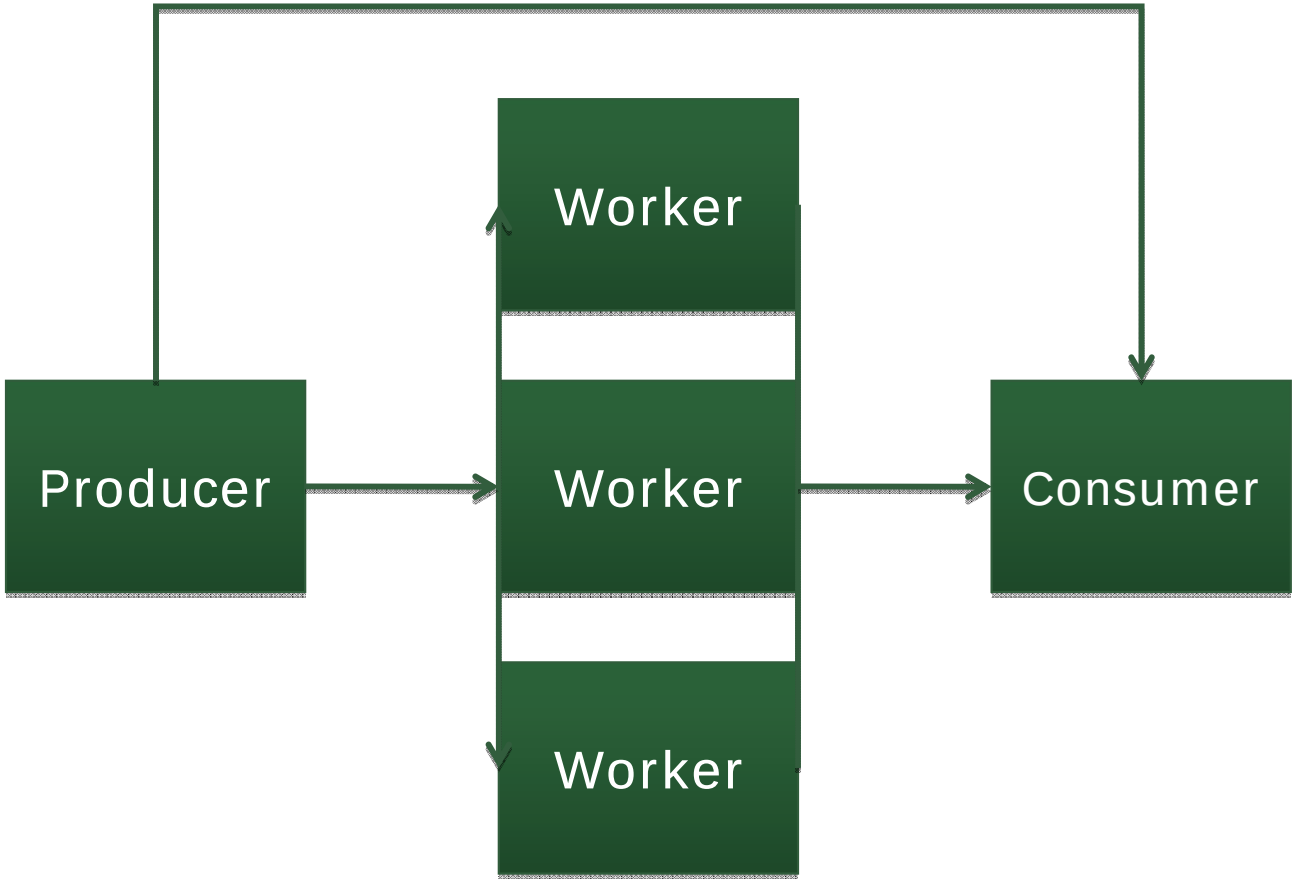
Controlled shutdown of network

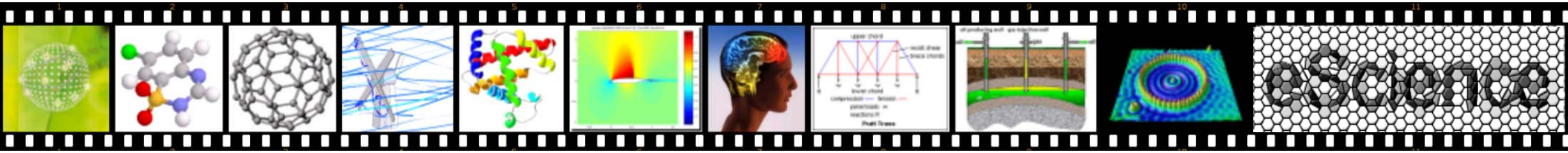
Channel poisoning was a huge step forward for CSP libraries

But controlling the shutdown to avoid race conditions is still important

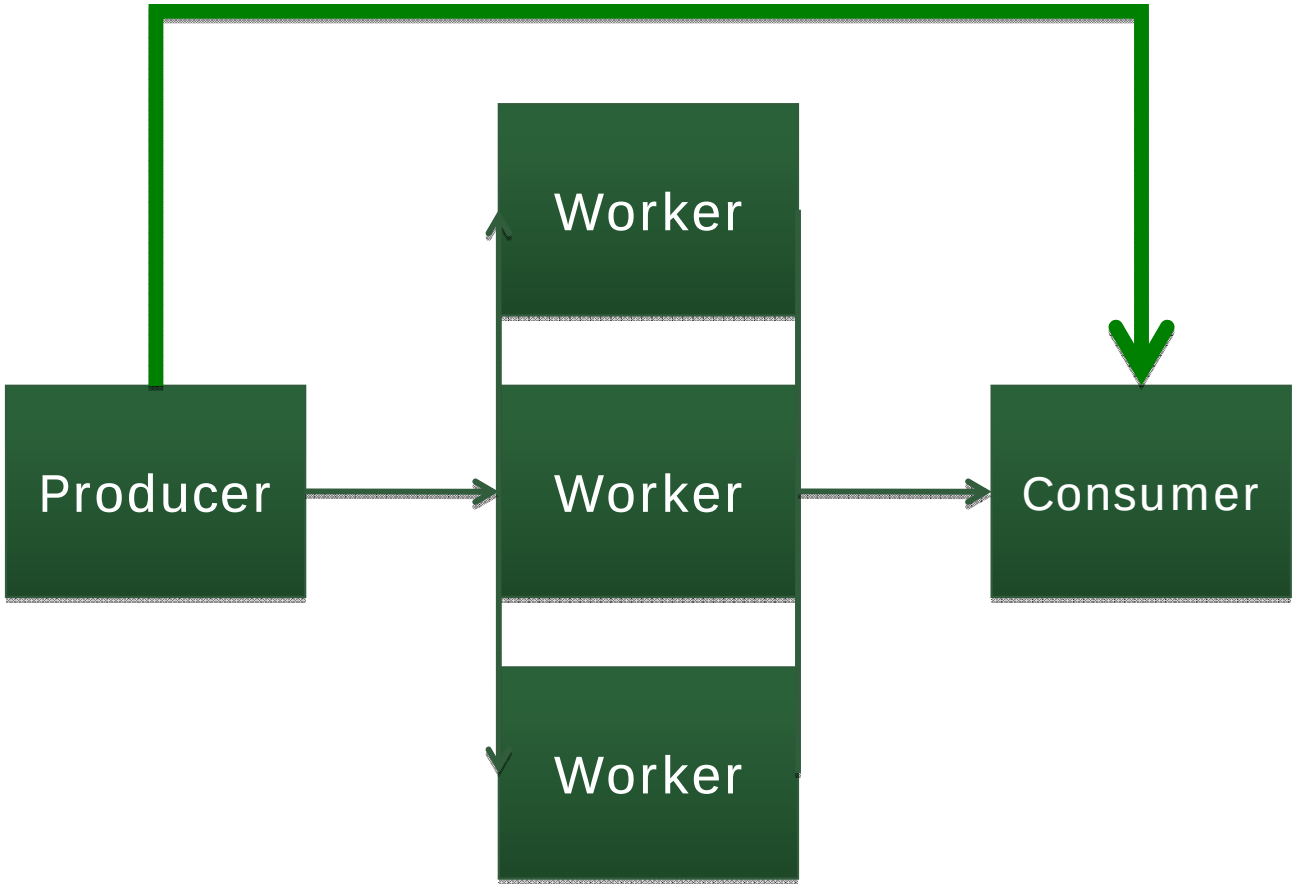


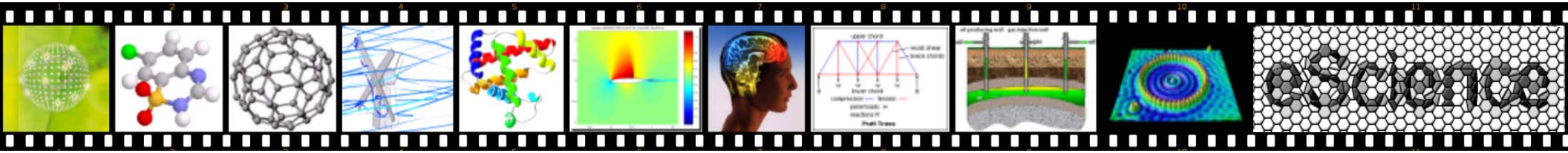
Poisoning



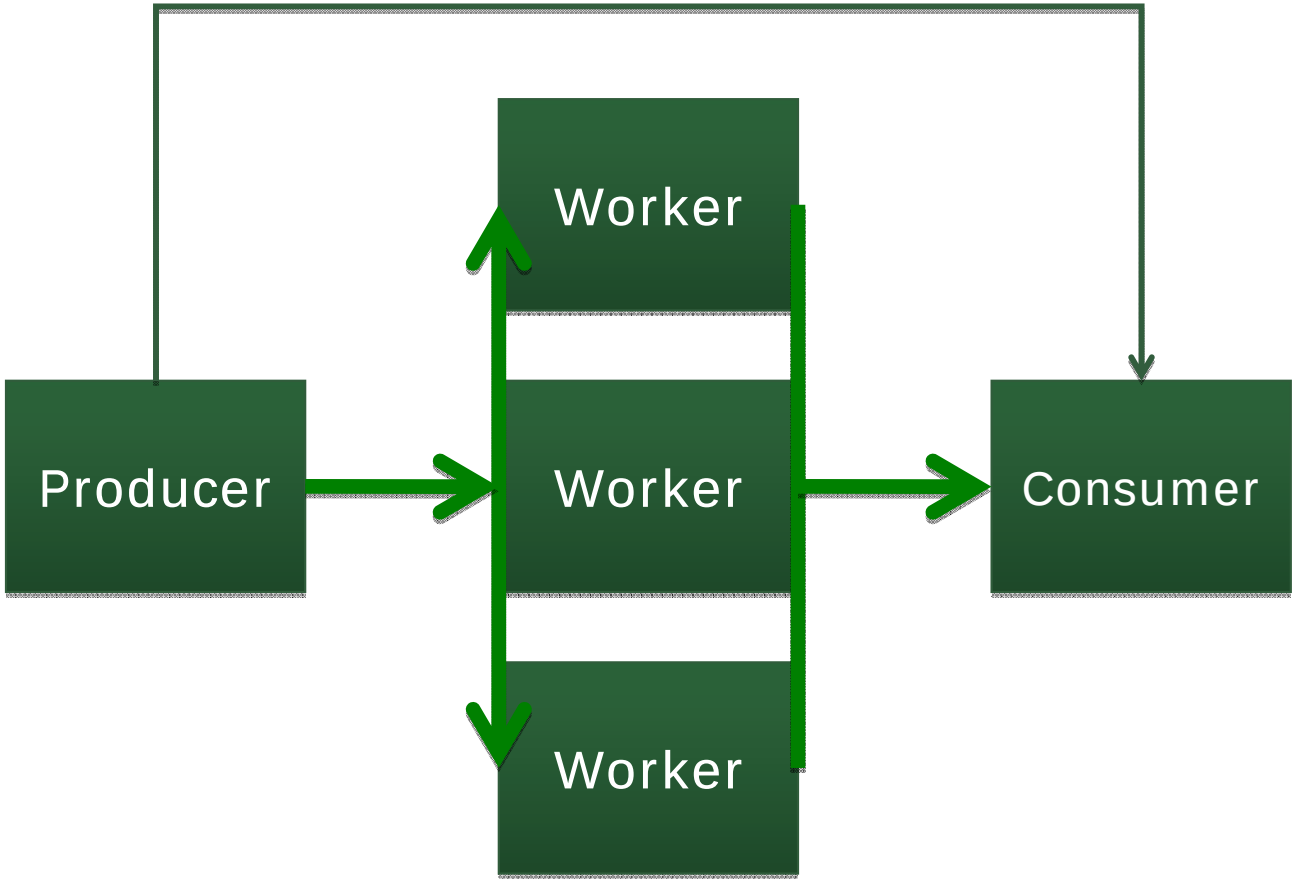


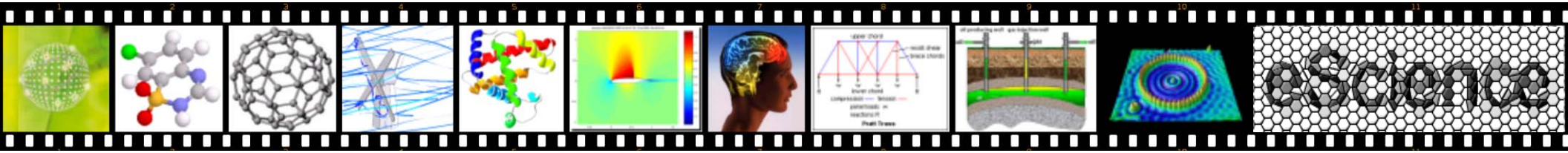
Poisoning



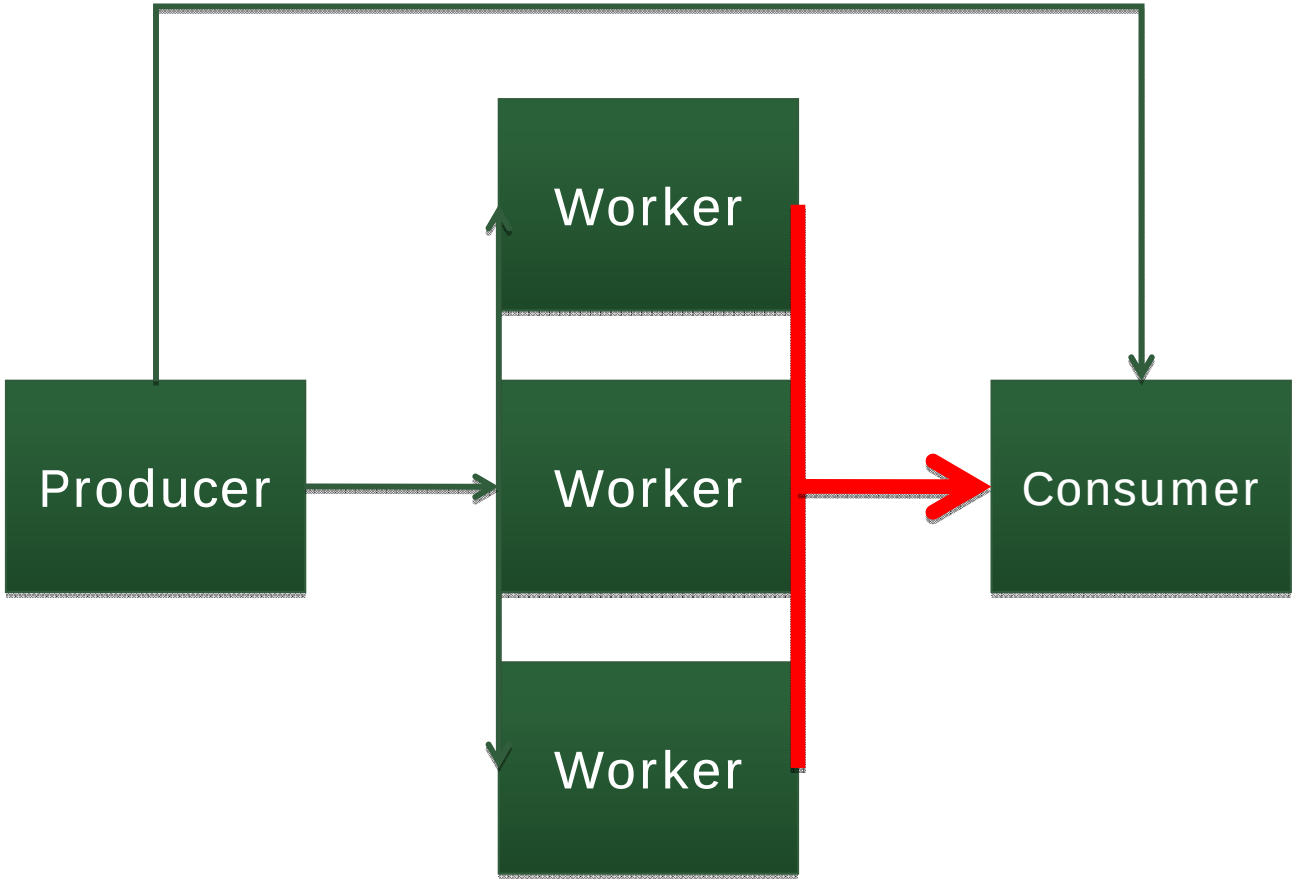


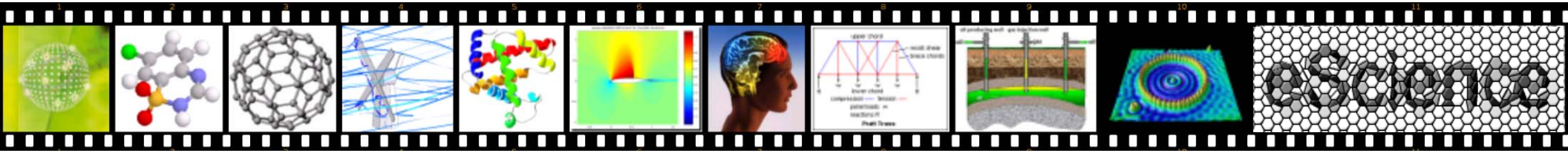
Poisoning



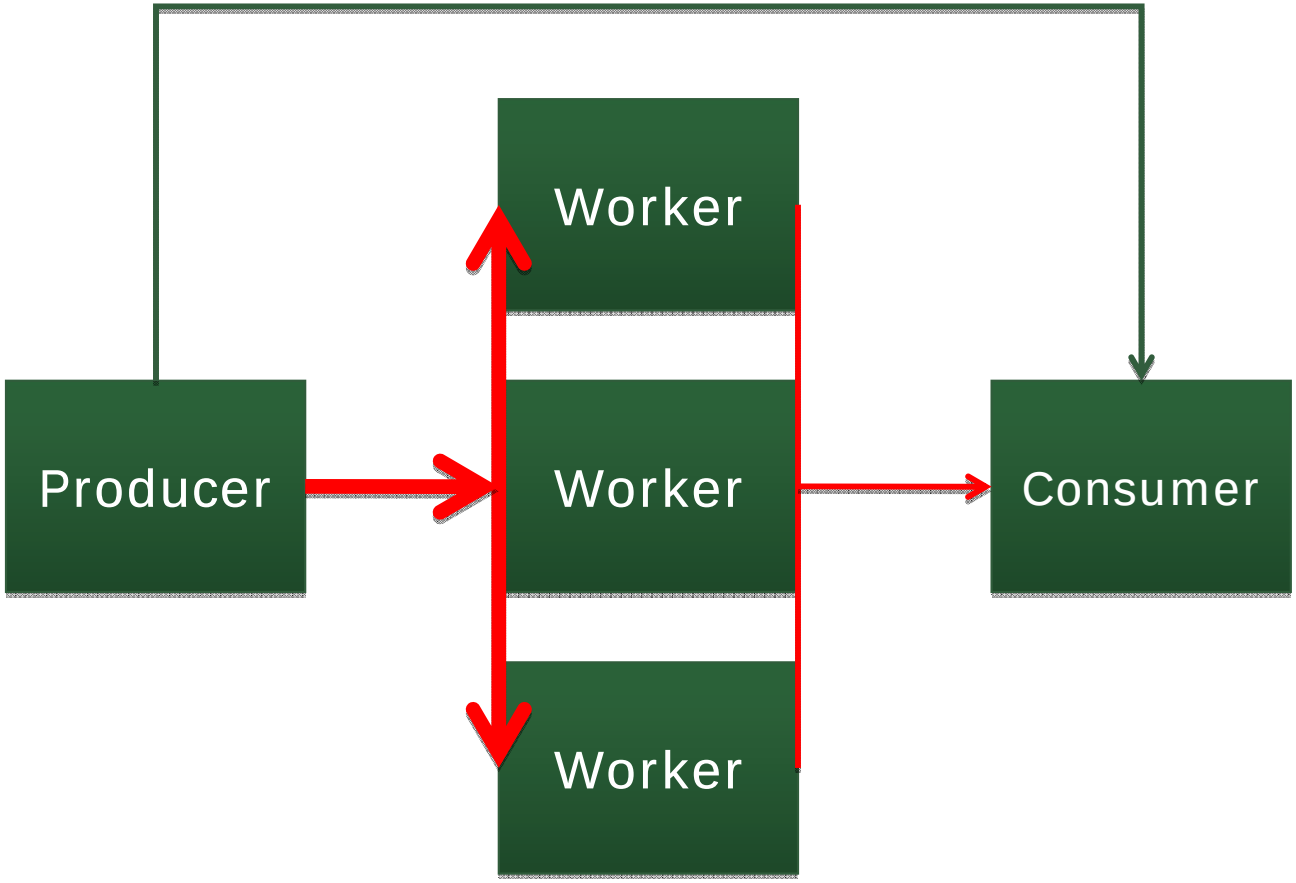


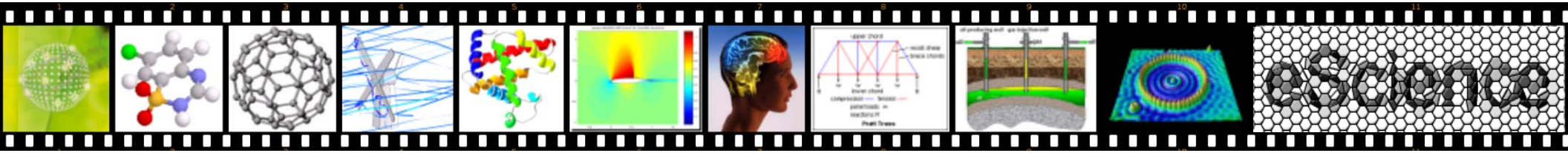
Poisoning





Poisoning





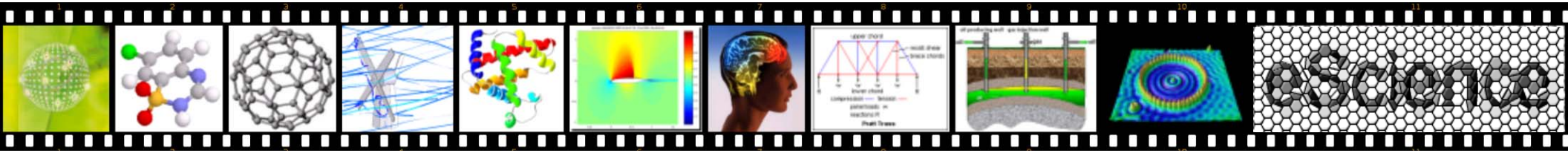
Controlled shutdown

Rather than poisoning channels PyCSP also support reference counting

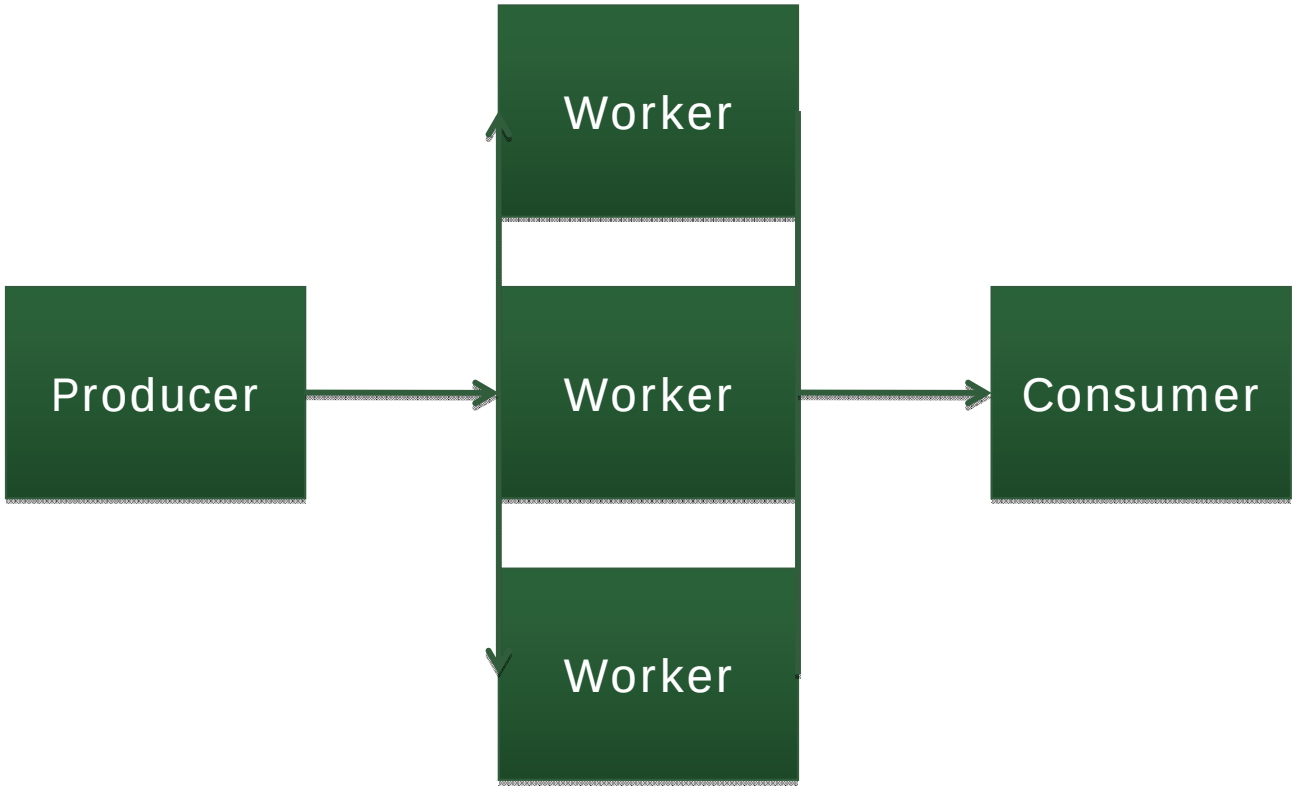
When a channel end is created the count on that direction is increased

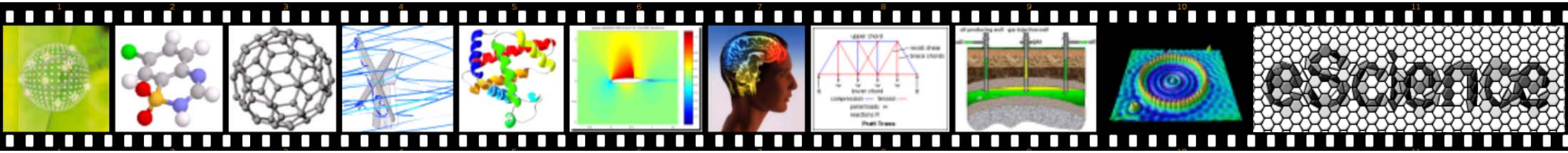
A process can, where it would otherwise do a poison issue a retire

When the reference count on a channel-end reaches zero the whole channel enters a retired state

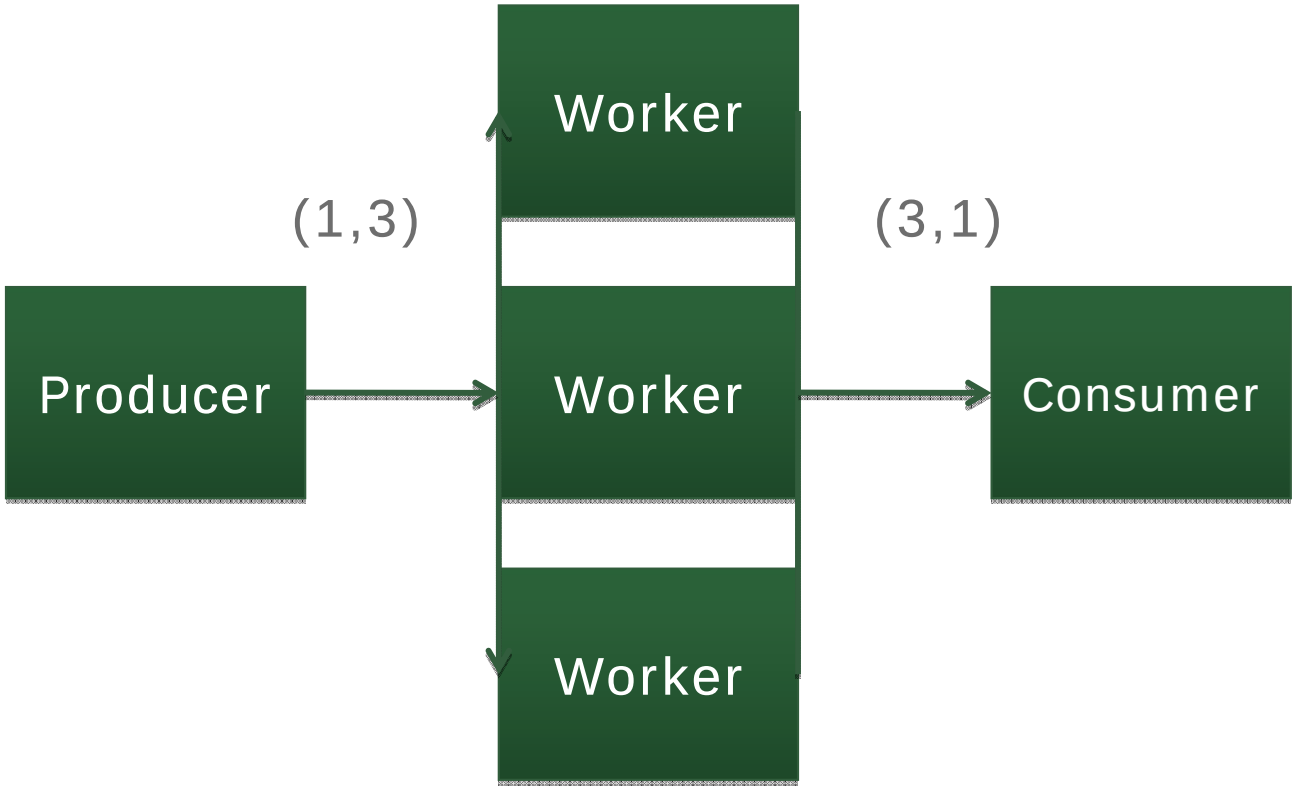


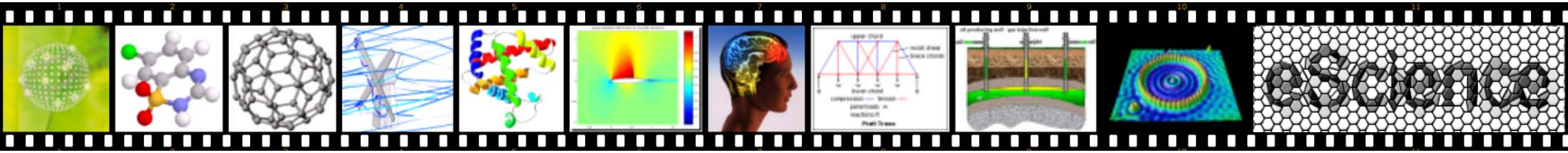
Controlled shutdown



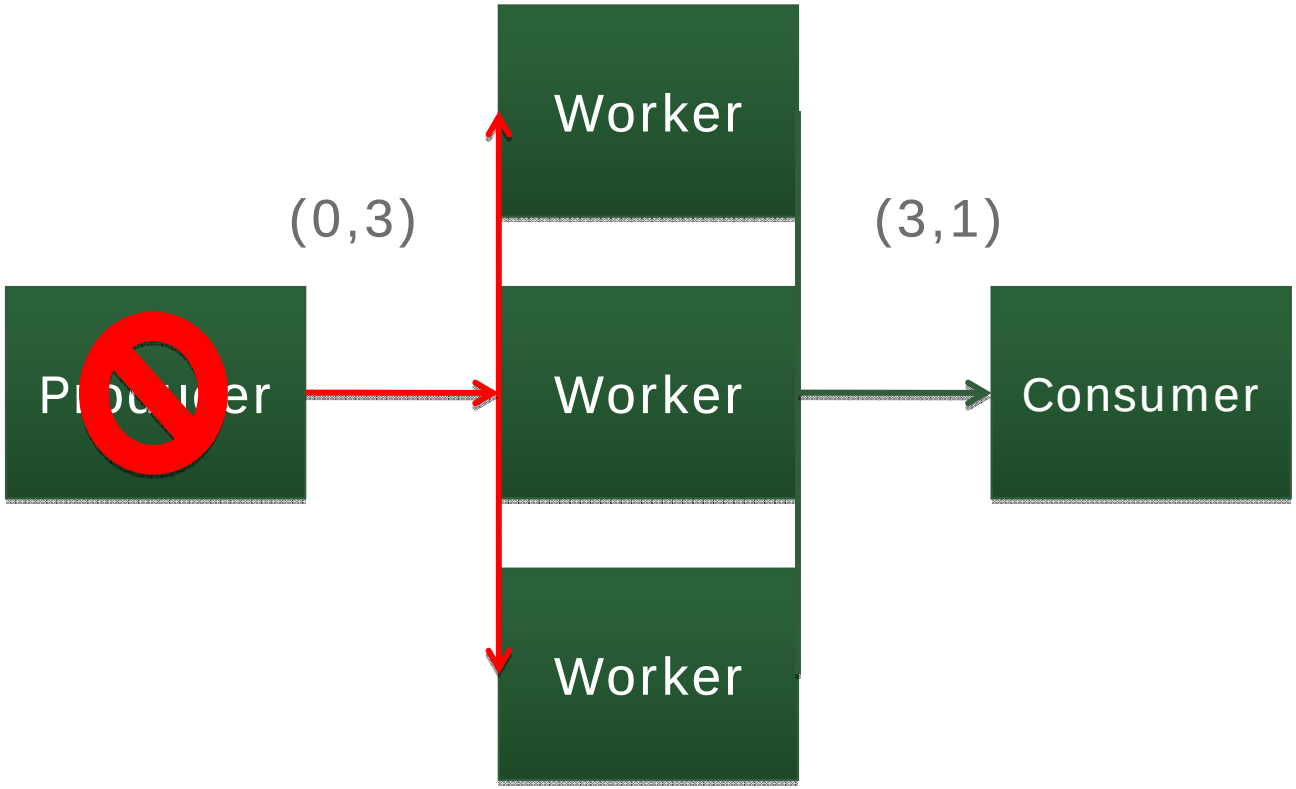


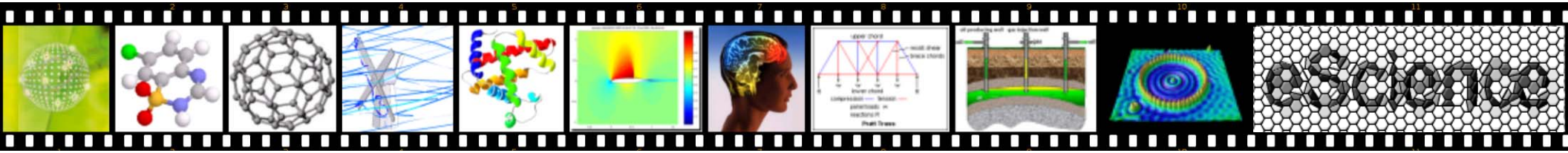
Controlled shutdown



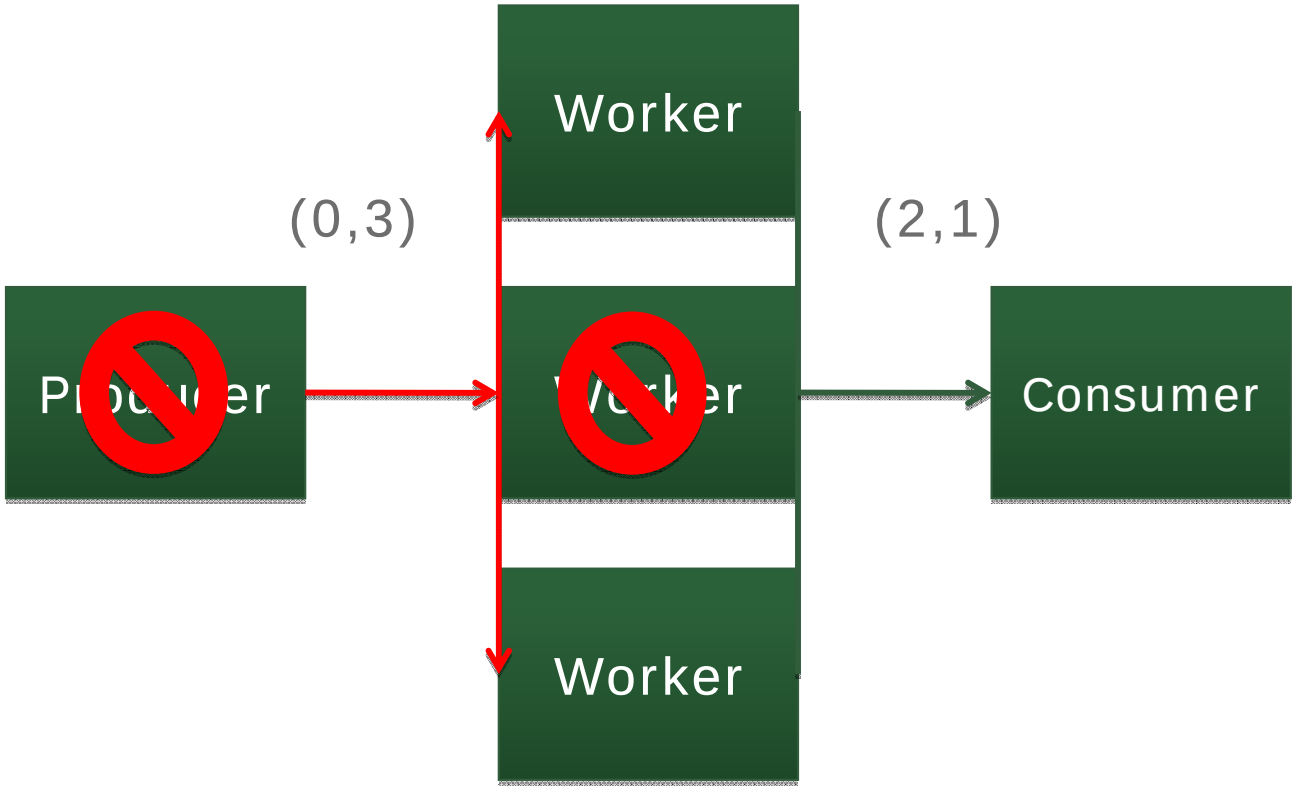


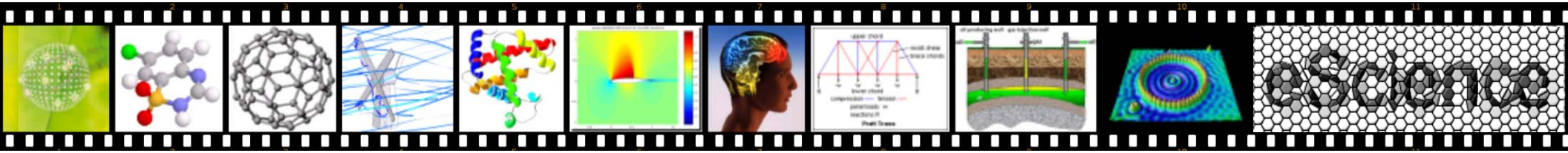
Controlled shutdown



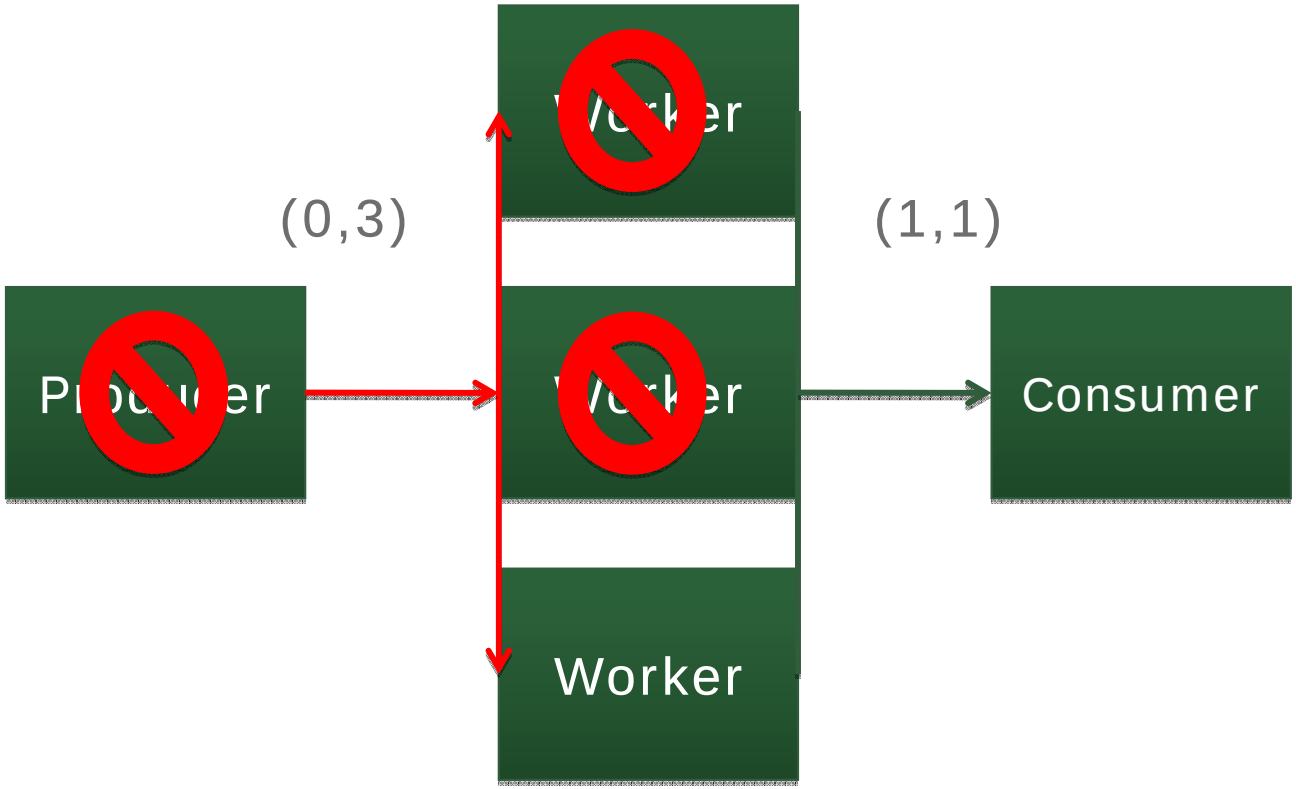


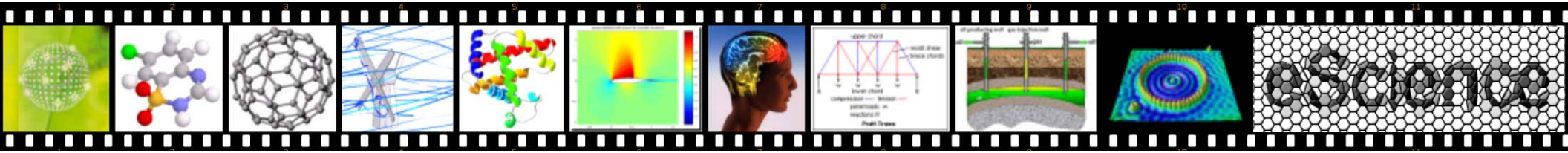
Controlled shutdown



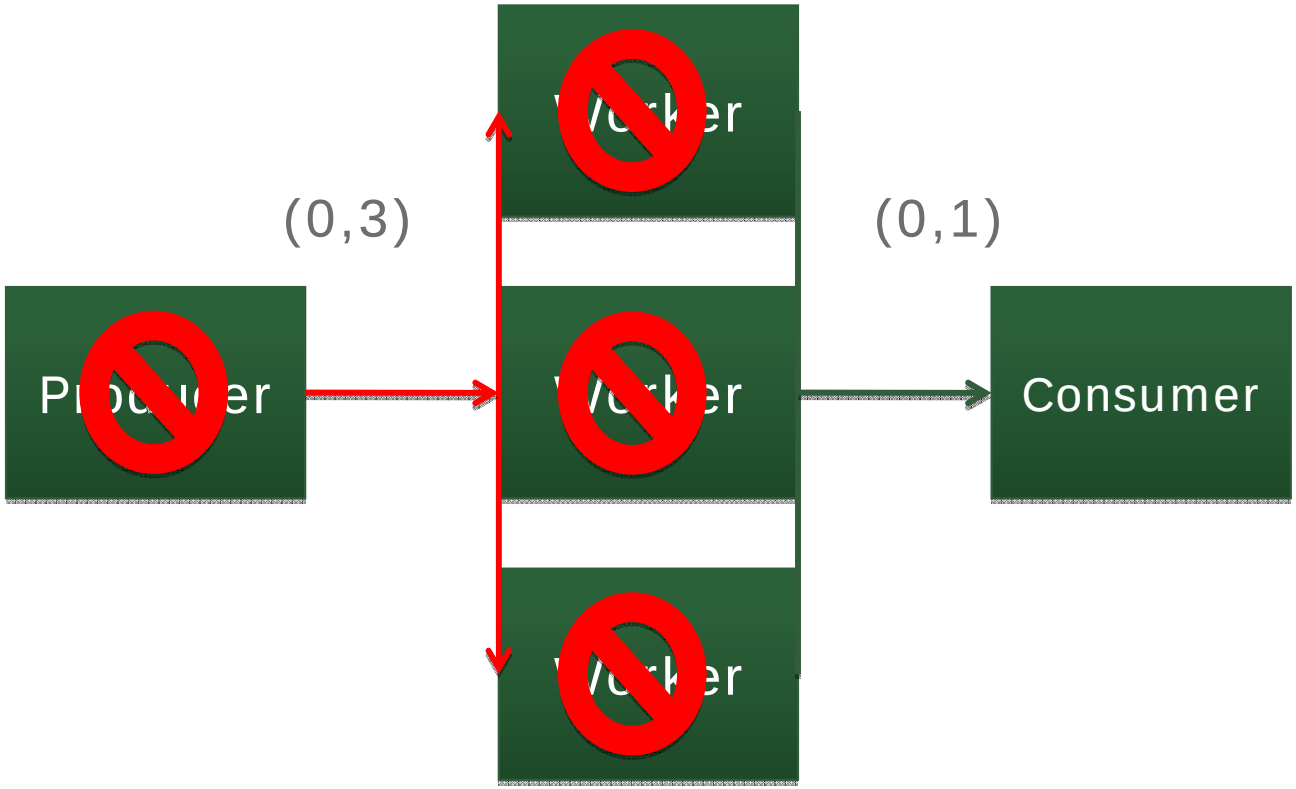


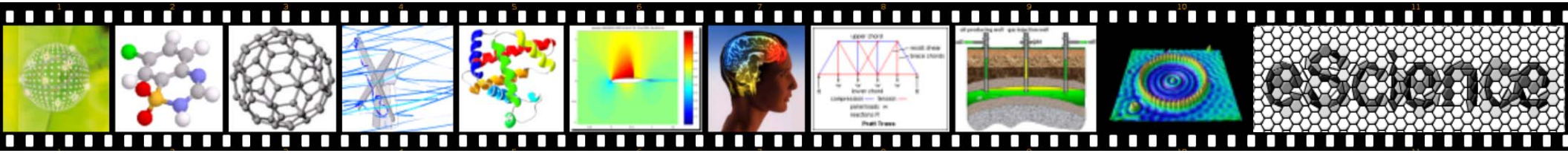
Controlled shutdown



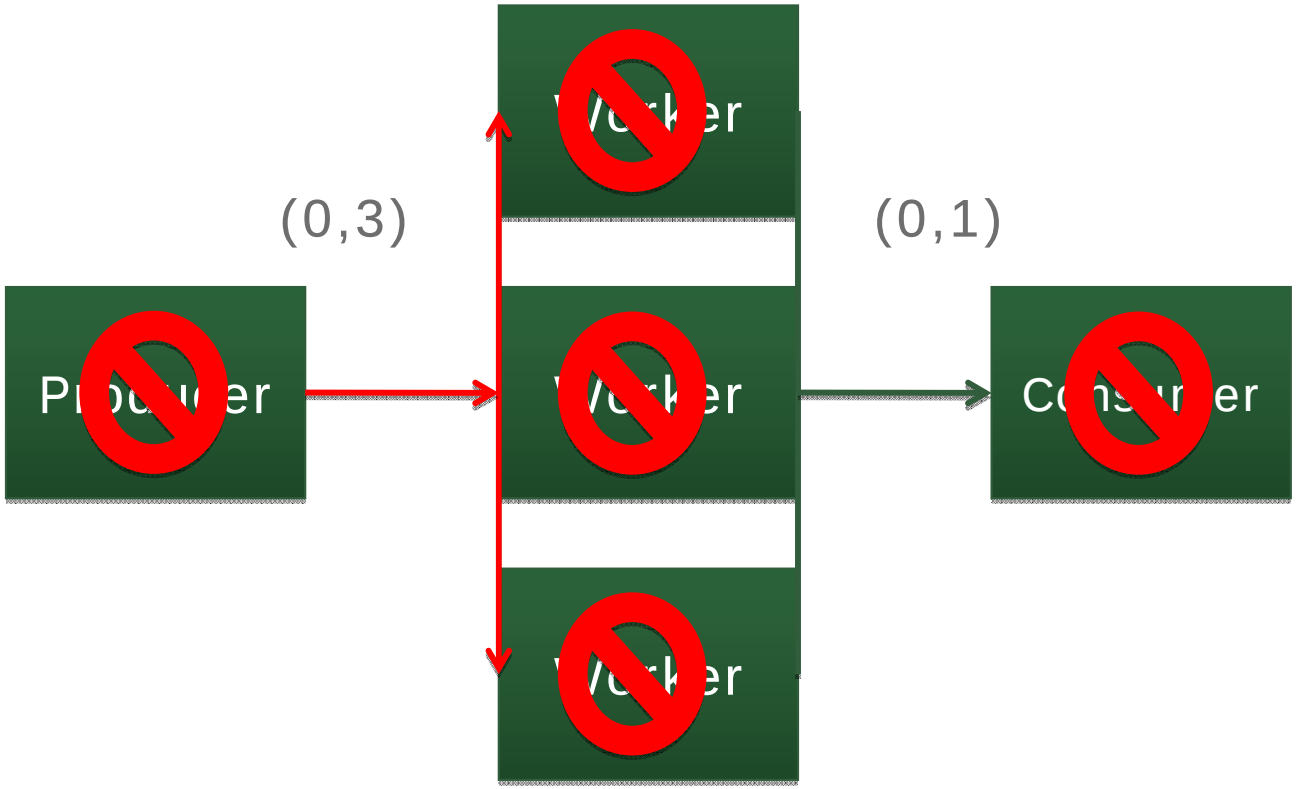


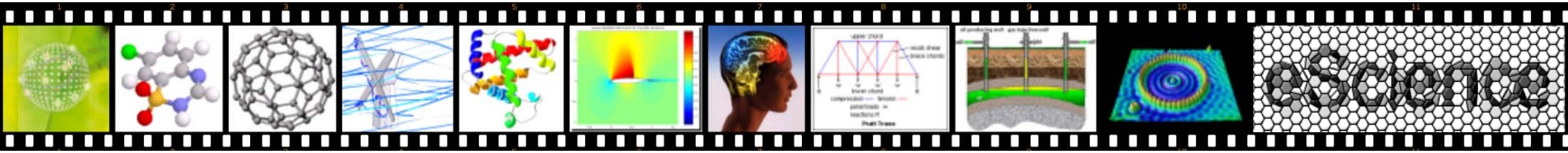
Controlled shutdown





Controlled shutdown





Choice

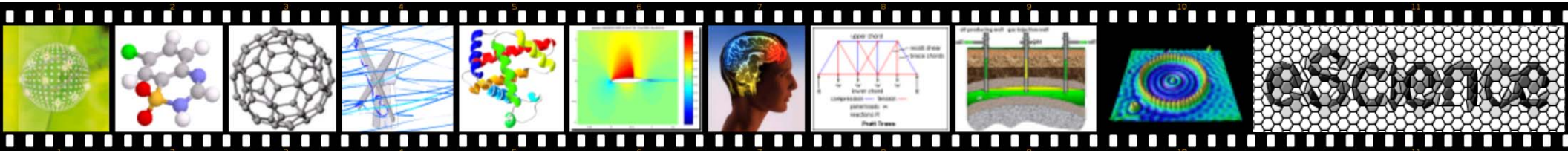
Choices are now selected and executed in one step

- More like Occam less like select()

The execution part is either a (small) direct statement or a function

- Declared with @choice

Both input and output guards are supported



Choice

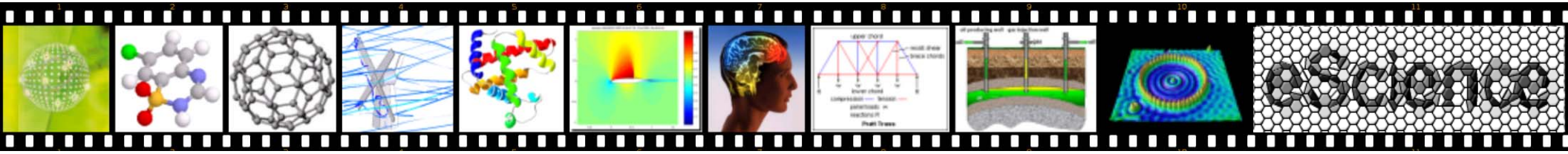
Input guards are

- <channel> : <guard>

Output guards are

- (<channel>, <value>) : <guard>

```
@choice
def print_result():
    print __channel_input

Alternation([
                {in : print_result()},
                {(out , value) : “value += 1”}
]).execute()
```



Prioritization

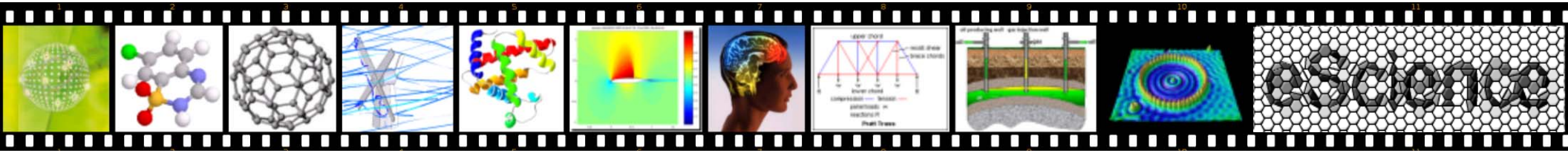
Alternation support mixing prioritized and unprioritized guards

An alternation is a list of dictionaries

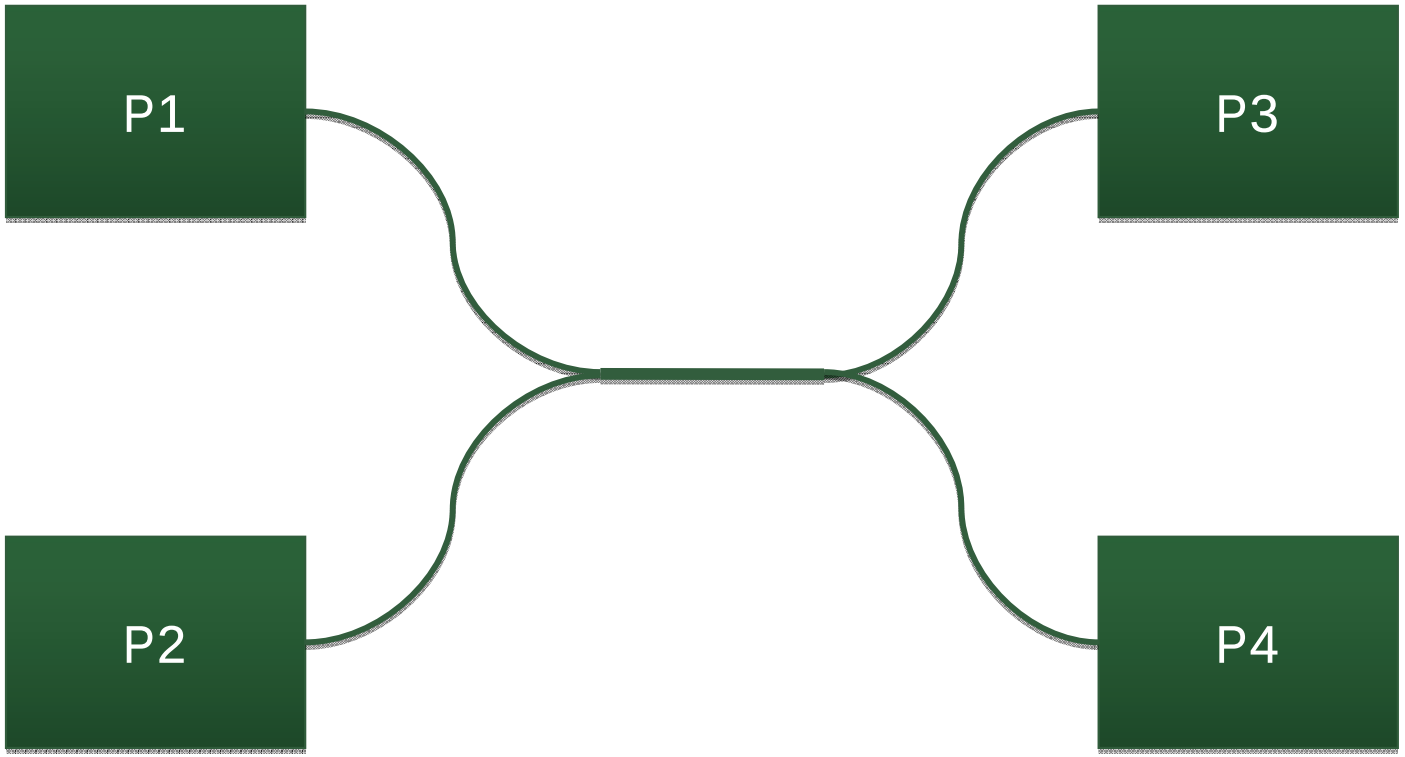
- List order define priority
- Within a dictionary the elements are peer

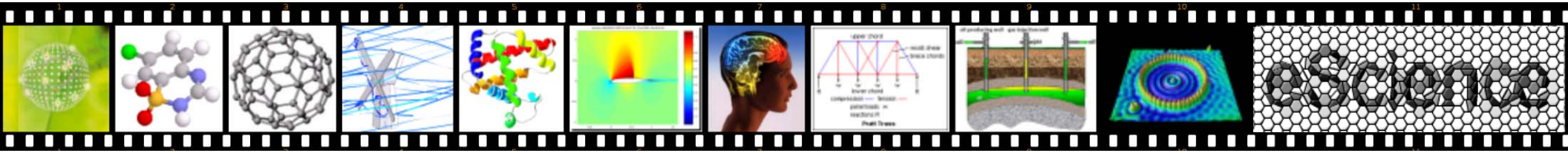
```
#ALT
@choice
def print_result():
    print __channel_input

Alternation([{
                in : print_result(),
               (out , value) : “value += 1”
}]).execute()
```

```
#PRIALT
@choice
def print_result():
    print __channel_input

Alternation([
                {in : print_result()},
                {(out , value) : “value += 1”}
]).execute()
```



Everything is “Any2Any”





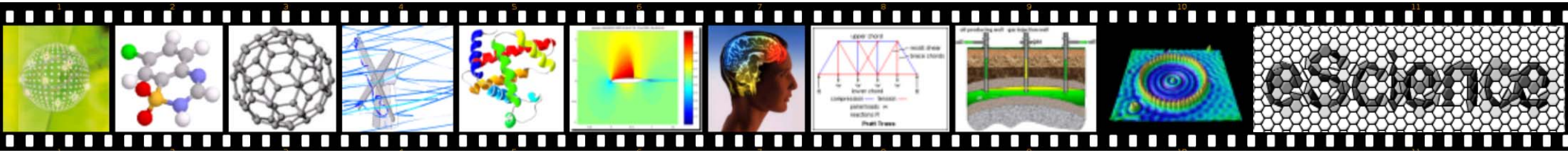
Challenge

When we combine input and output guards and multi-ended channels we have a well established challenge

- A given guard may by matched by several other guards

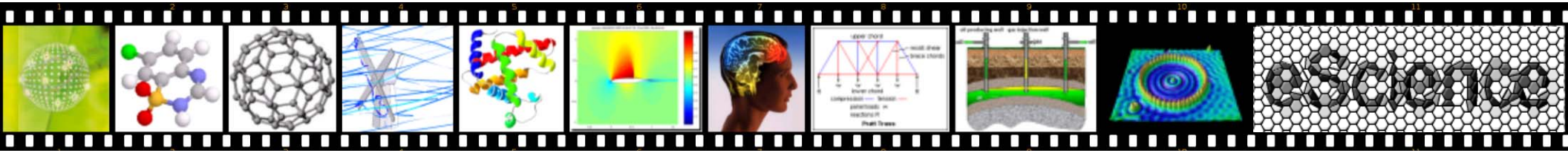
How do we ensure that a match is performed

- Atomically
- Without deadlock
- Without livelock

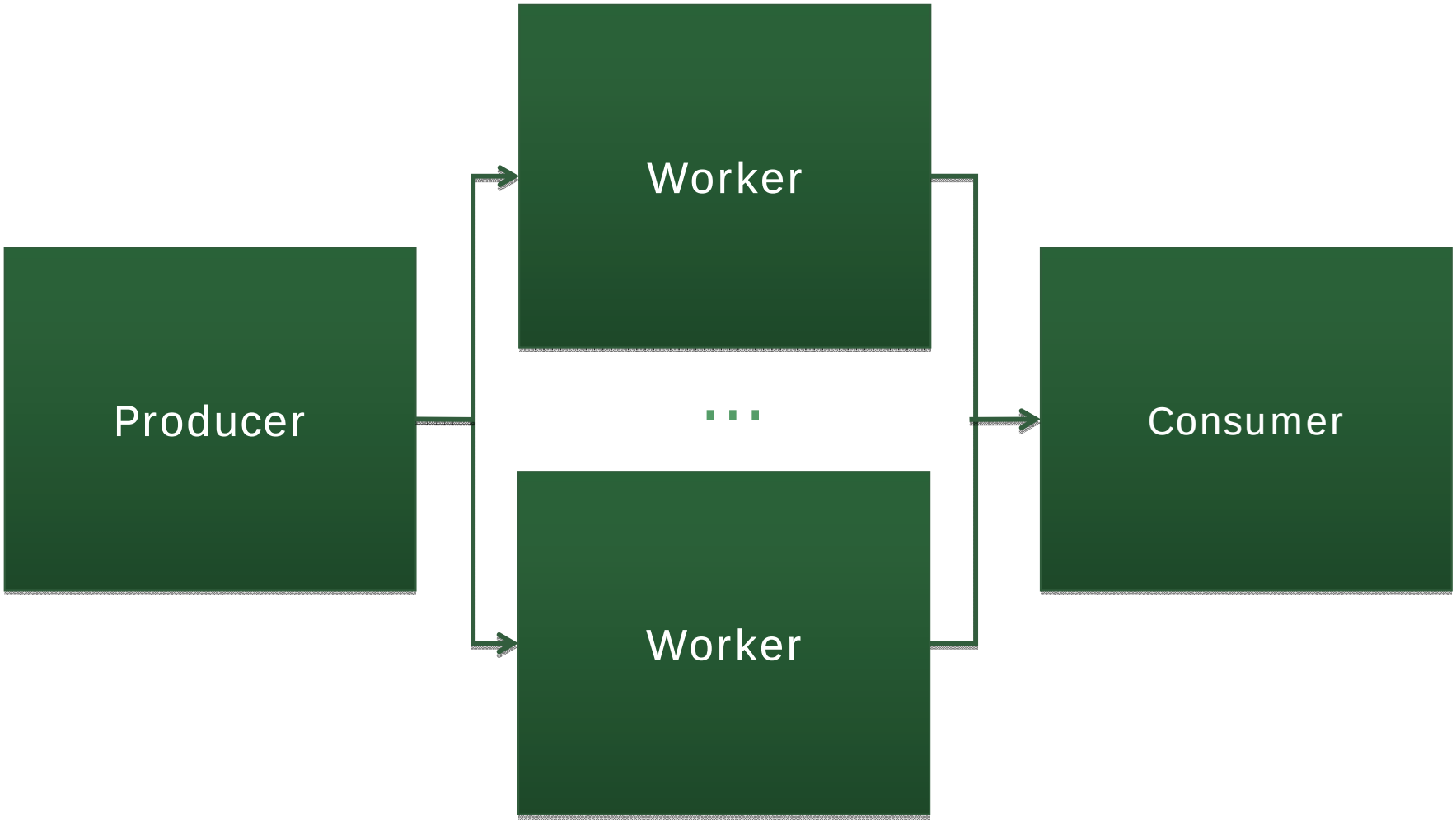


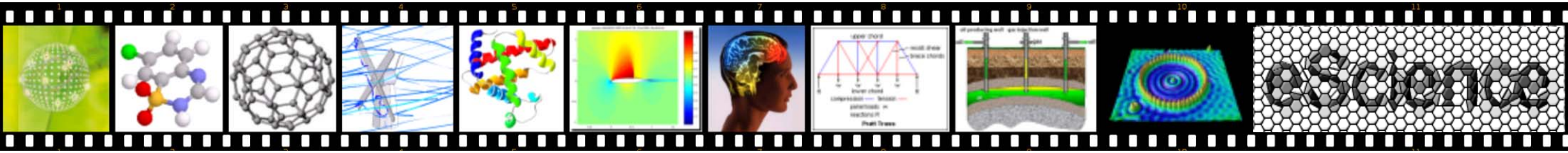
Simplified matching algorithm

```
handle = new_request_handle ()
guard_channel.registered_handle.add(handle)
for guard in choice :
    if handle match registered_handle in guard.channel :
        perform communication
        make_active (handle , registered_handle)
waitfor active (handle)
guard_channel.registered_handle.remove(handle)
```



Monte Carlo Pi





Monte Carlo Pi

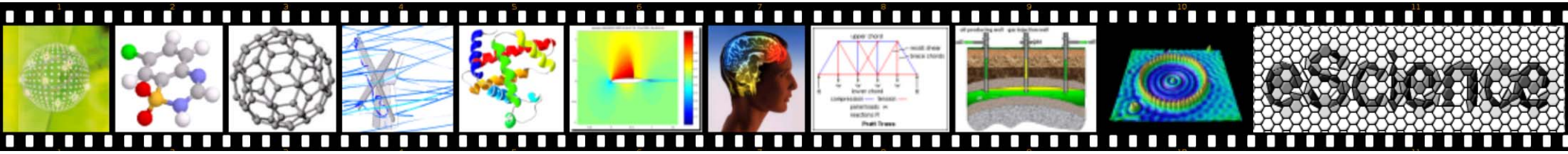
```
@process
def producer ( job_out , bagsize , bags
):
   for i in range ( bags ):
      job_out ( bagsize )
   retire ( job_out )
```

```
@process
def worker (job_in , result_out ):
    while True :
    cnt = job_in () #Get task
    sum = reduce ( lambda x,y:
x+( random ()**2+ random
()**2 <1.0) ,range (cnt ))
    result_out (( cnt ,sum )) #
Forward result
```

...

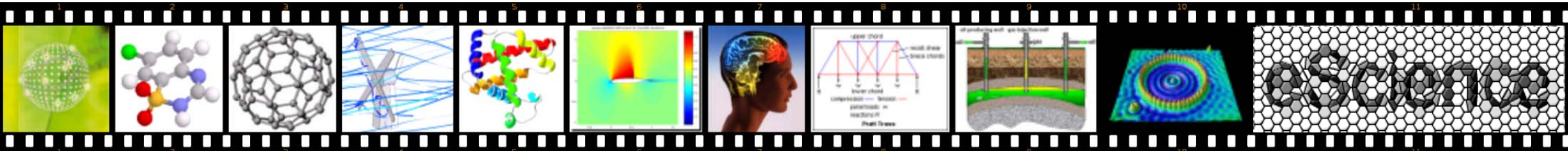
```
@process
def worker (job_in , result_out ):
    while True :
    cnt = job_in () #Get task
    sum = reduce ( lambda x,y:
x+( random ()**2+ random
()**2 <1.0) ,range (cnt ))
    result_out (( cnt ,sum )) #
Forward result
```

```
@process
def consumer ( result_in ):
    cnt , sum =0 ,0
    try:
    while True :
    c,s= result_in () #Get result
    cnt , sum = cnt +c, sum +s
    except ChannelRetireException :
    print 4.0* sum/cnt
```



An example...

```
from pycsp import *
from random import random
@process
def producer ( job_out , bagsize , bags ):
    for i in range ( bags ): job_out ( bagsize )
    retire ( job_out )
@process
def worker (job_in , result_out ):
    while True :
    cnt = job_in () #Get task
    sum = reduce ( lambda x,y: x+( random ()**2+ random ()**2 <1.0) ,range (cnt ))
    result_out (( cnt ,sum )) # Forward result
@process
def consumer ( result_in ):
    cnt , sum =0 ,0
    try:
    while True :
    c,s= result_in () #Get result
    cnt , sum = cnt +c, sum +s
    except ChannelRetireException :
    print 4.0* sum/cnt #We are done - print result

jobs = Channel ()
results = Channel ()
Parallel ( producer ( jobs.writer () ,1000 , 10000) ,
[ worker ( jobs.reader (), results.writer ()) for i in range (10)] ,
consumer ( results.reader ()))
```



Conclusions

PyCSP is alive
- Target is scientists not programmers

Only one channel-type
- With multi ended channels

External choice is supported by these channels
- And adds an output guard

Guards are now handled atomically

Graceful shutdown is introduced through channel reference counting