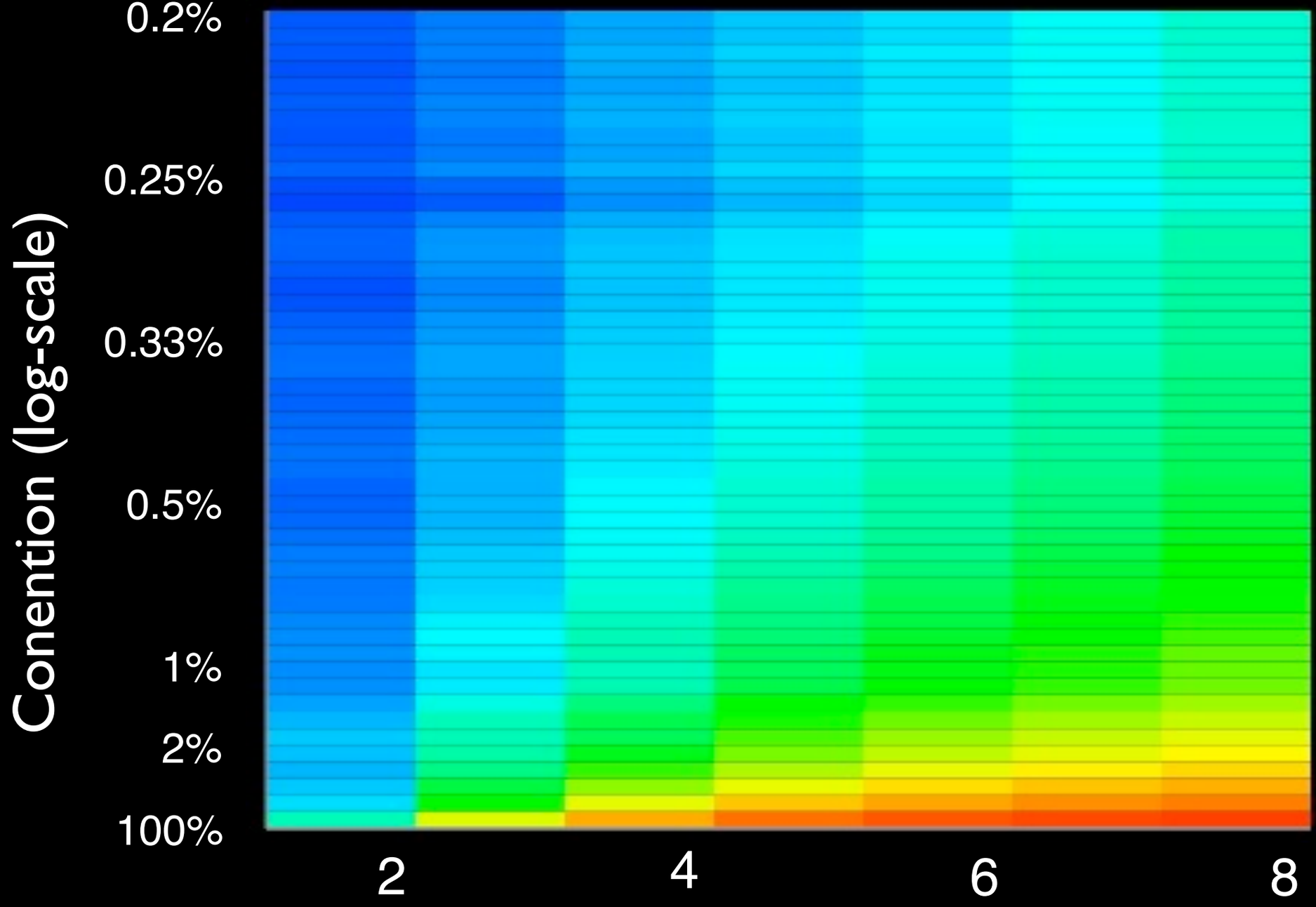
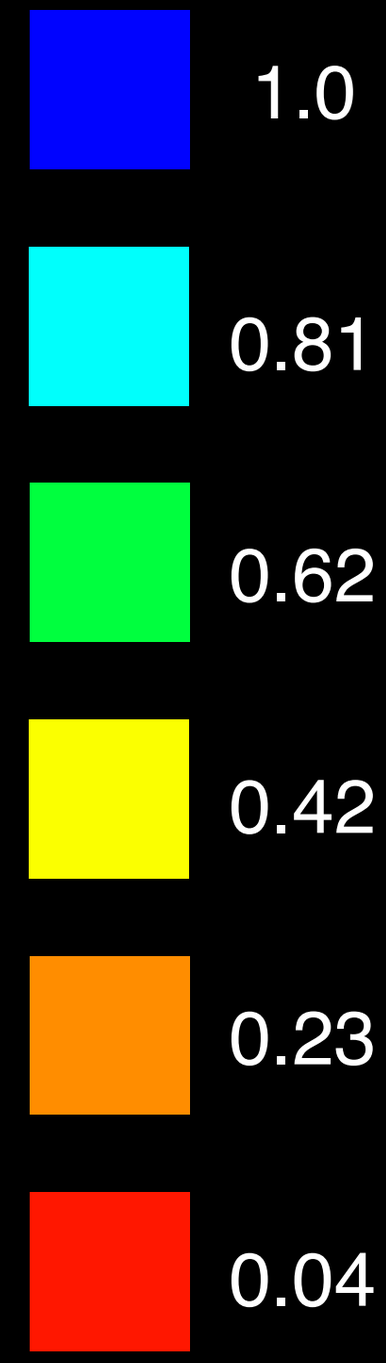
Reagents:

expressing and composing fine-grained concurrency

Aaron Turon
Northeastern University

CAS: cost versus contention





Throughput / Sequential

java.util.concurrent

Synchronization

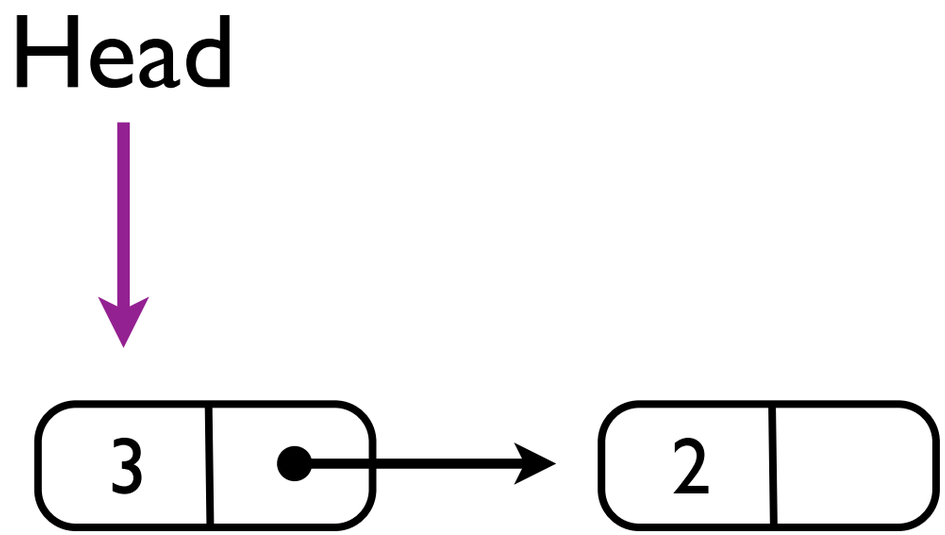
Reentrant locks
Semaphores
R/W locks
Reentrant R/W locks
Condition variables
Countdown latches
Cyclic barriers
Phasers
Exchangers

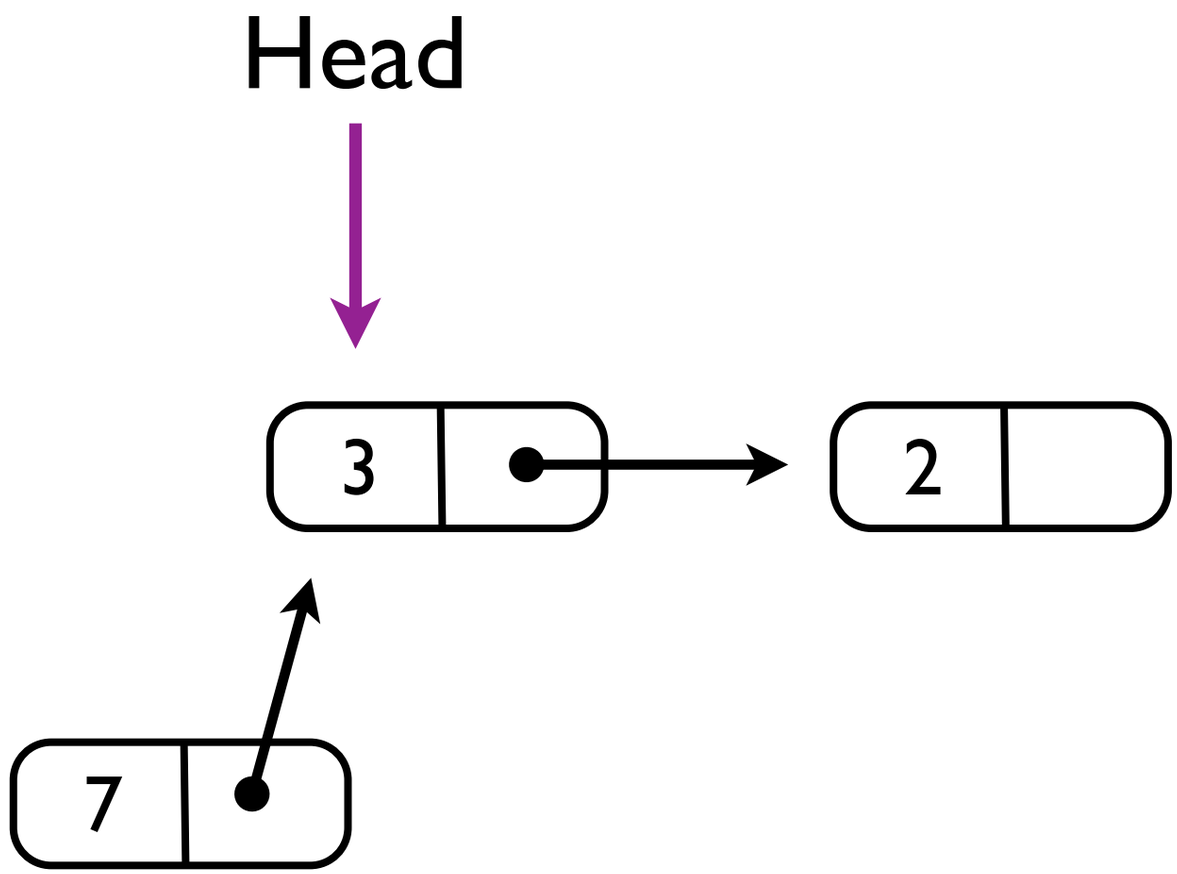
Data structures

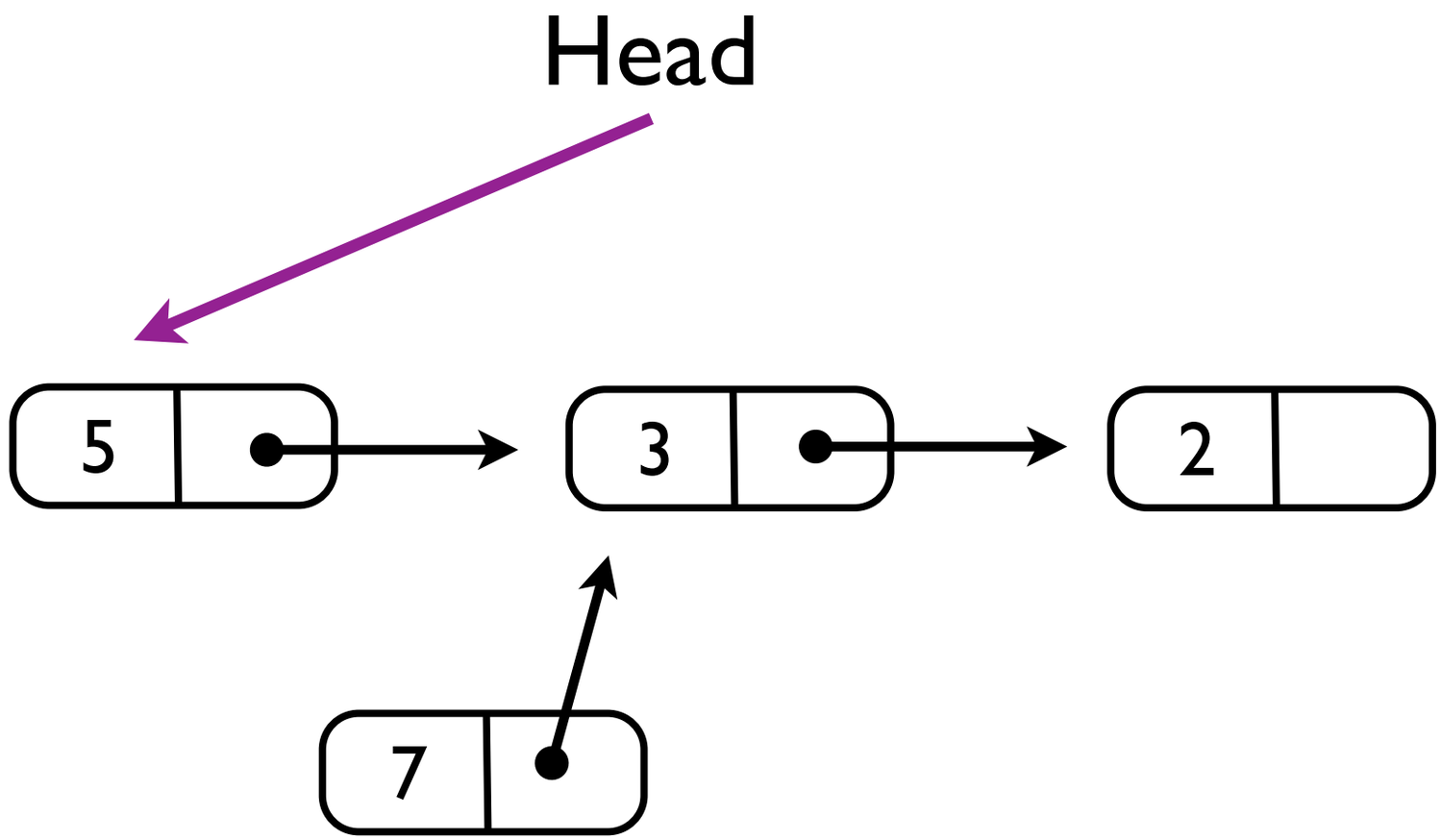
Queues
- Nonblocking
- Blocking (array & list)
- Synchronous
- Priority, nonblocking
- Priority, blocking

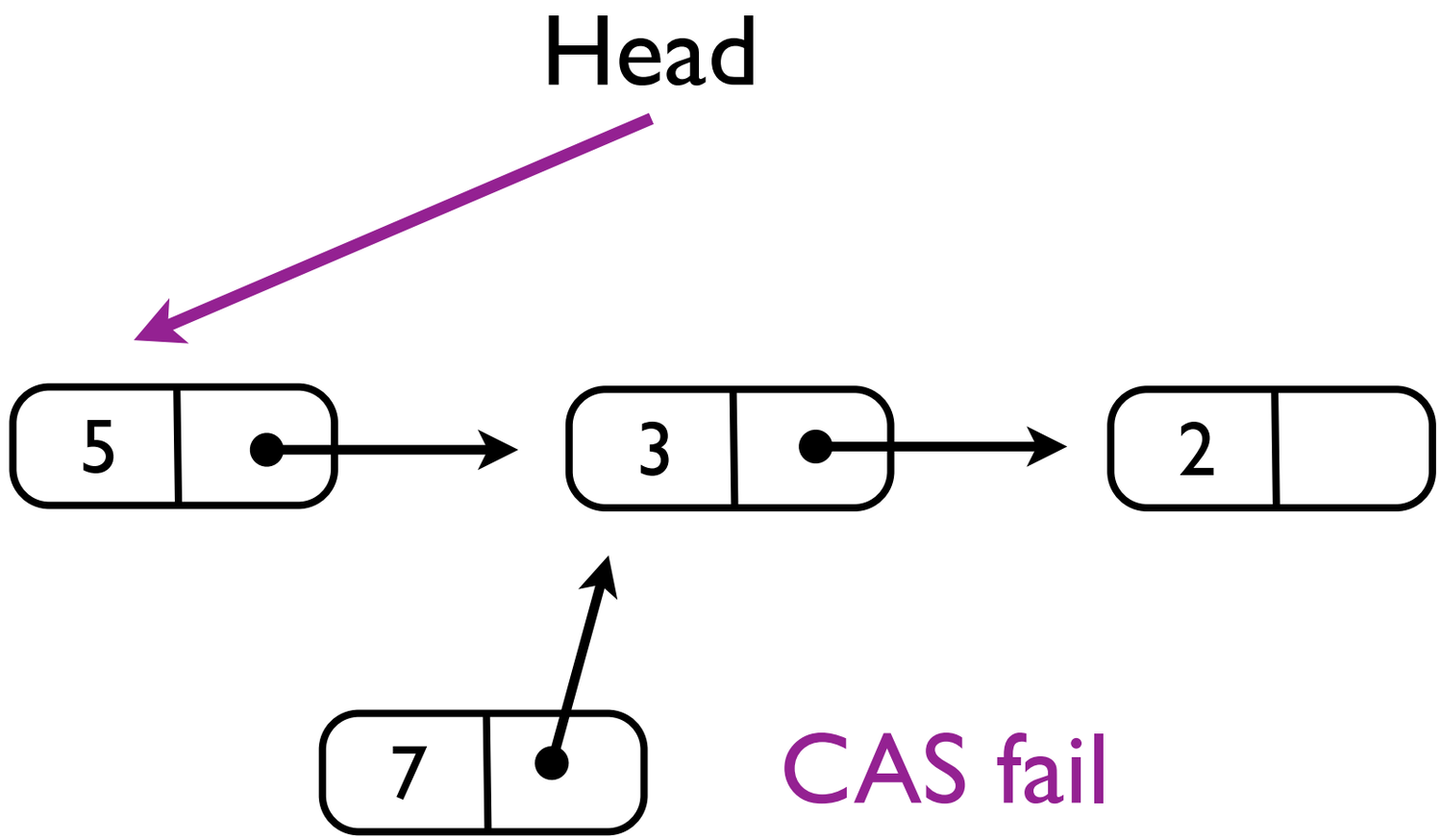
Deques
Sets
Maps (hash & skiplist)

```
class TreiberStack[A] {
  private val head =
    new AtomicRef[List[A]](Nil)

  def push(a: A) {
    val backoff = new Backoff
    while (true) {
      val cur = head.get()
      if (head.cas(cur, a :: cur)) return
      backoff.once()
    }
  }

...
```

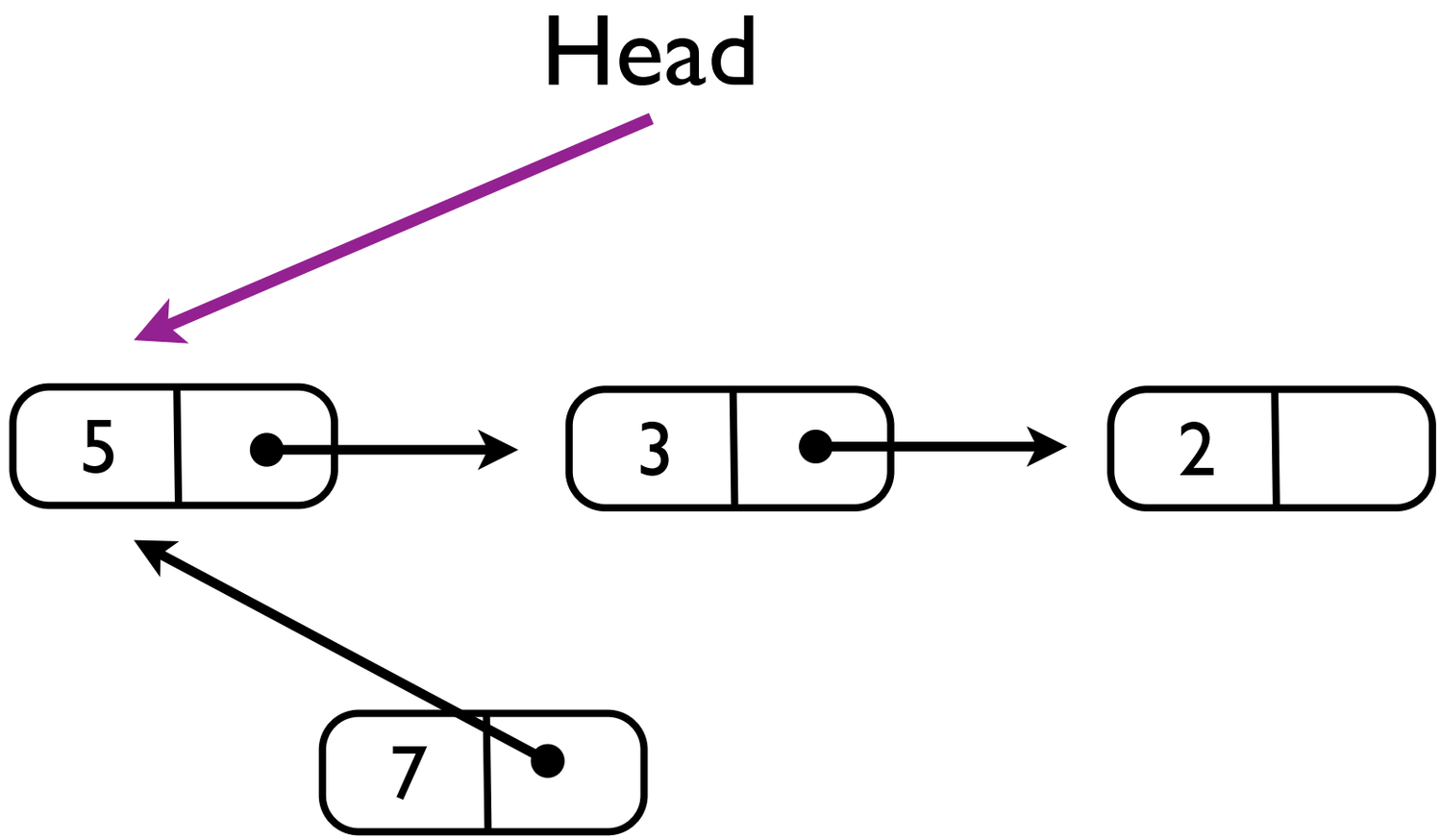

Head
3
2

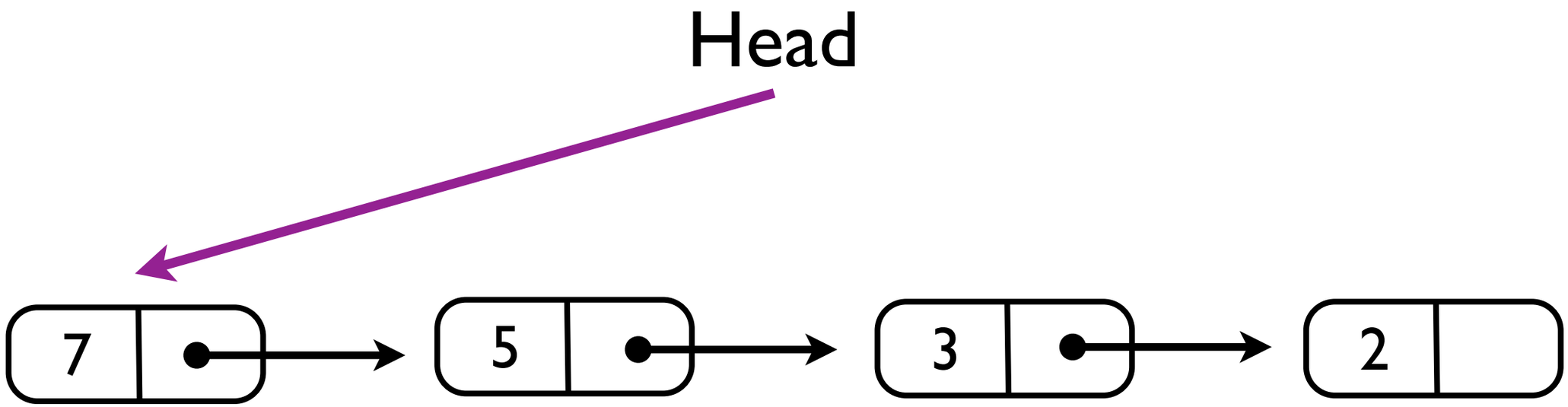

Head
3
2
7


Head
5
3
2
7



CAS fail


Head
5
3
2
7


Head
7
5
3
2

```
def tryPop(): Option[A] = {
  val backoff = new Backoff
  while (true) {
    val cur = head.get()
    cur match {
      case Nil      => return None
      case a::tail =>
        if (head.cas(cur, tail))
          return Some(a)
    }
    backoff.once()
  }
}
```

The Problem:

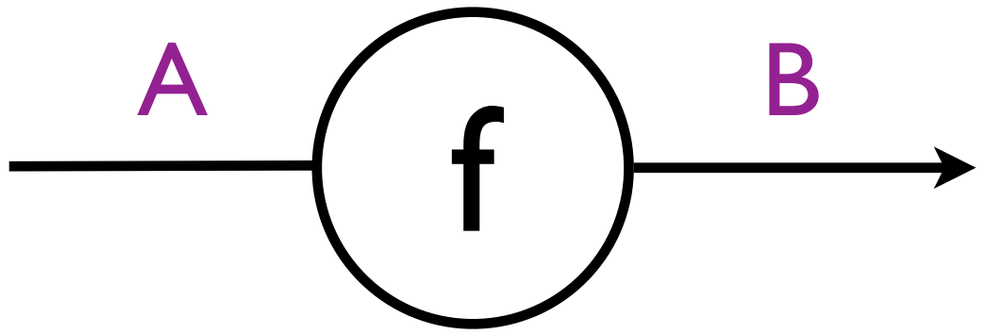
Concurrency libraries are indispensable, but hard to build and extend

The Proposal:

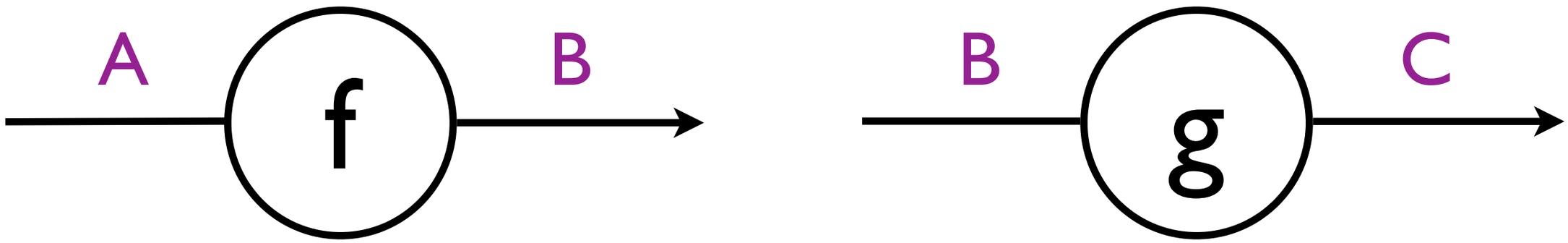
Scalable concurrent algorithms can be built and extended using abstraction and composition

Design

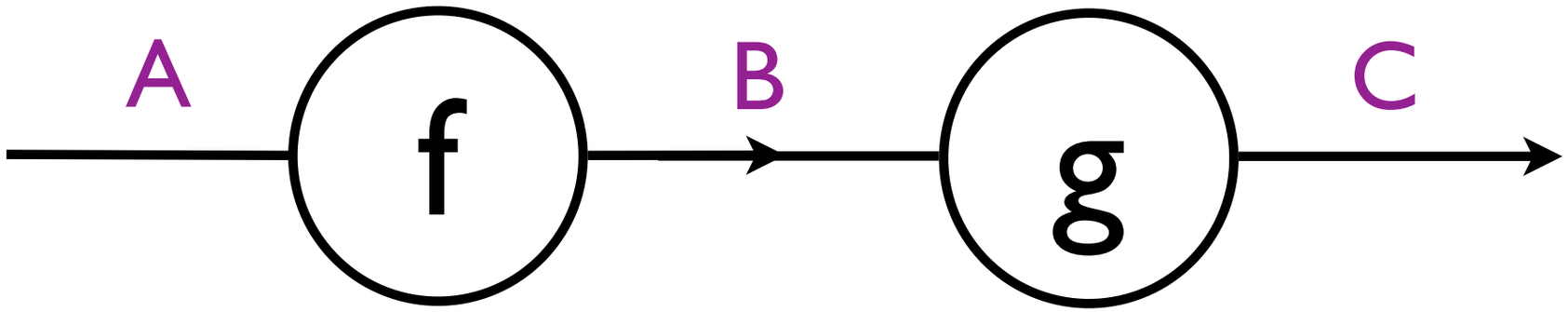
Lambda: the ultimate abstraction



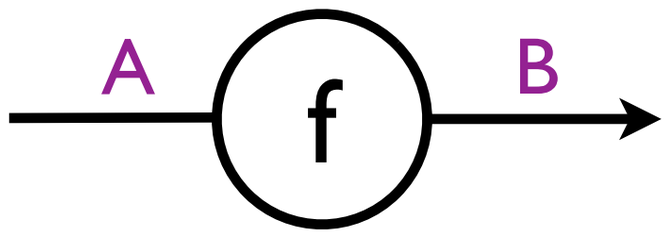
Lambda: the ultimate abstraction



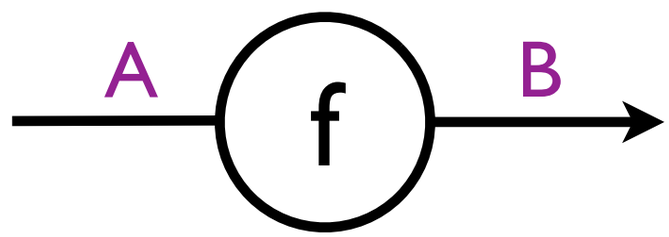
Lambda: the ultimate abstraction



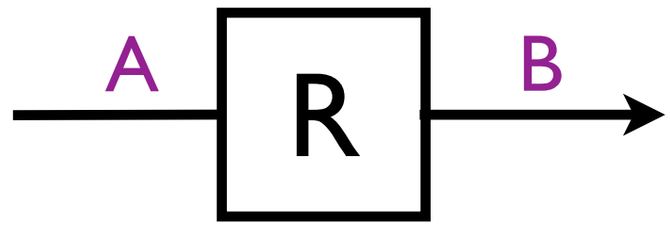
Lambda abstraction:



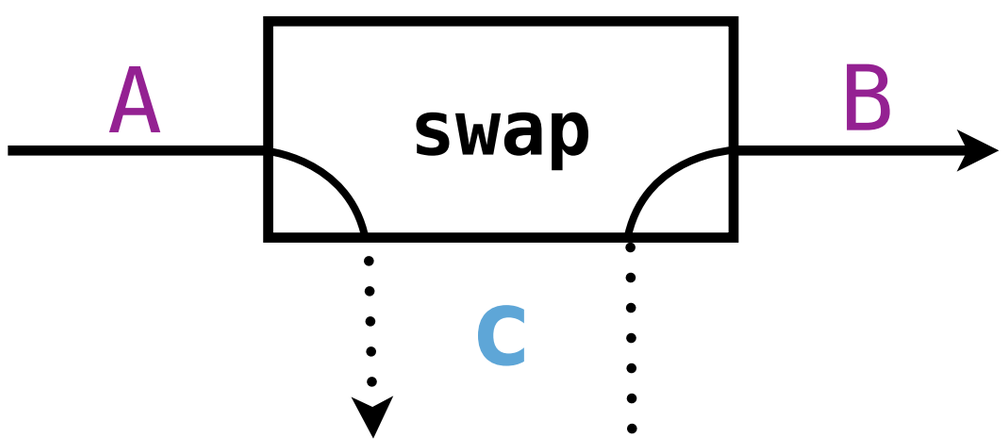
Lambda abstraction:



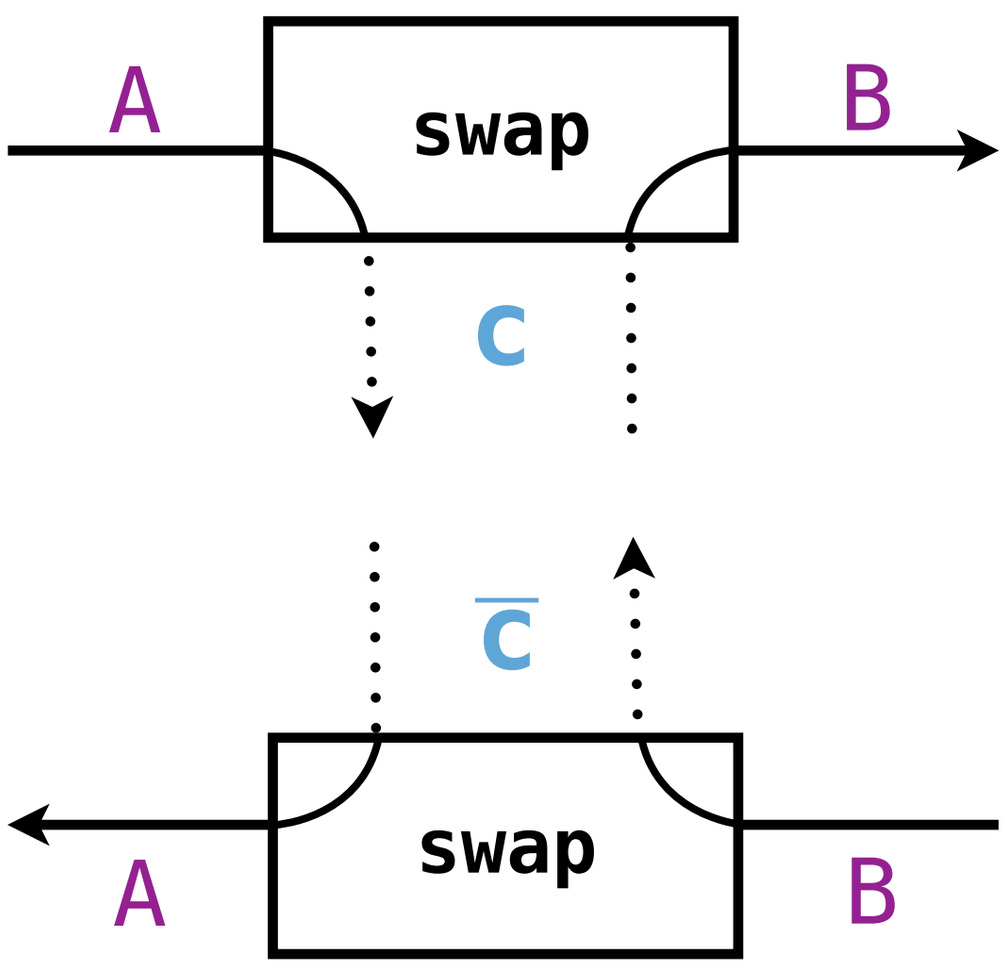
Reagent abstraction:



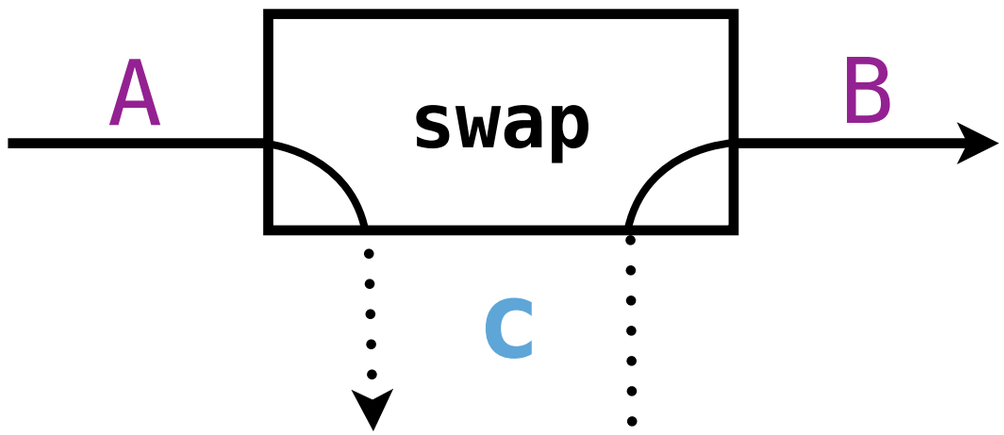
c: Chan[A,B]



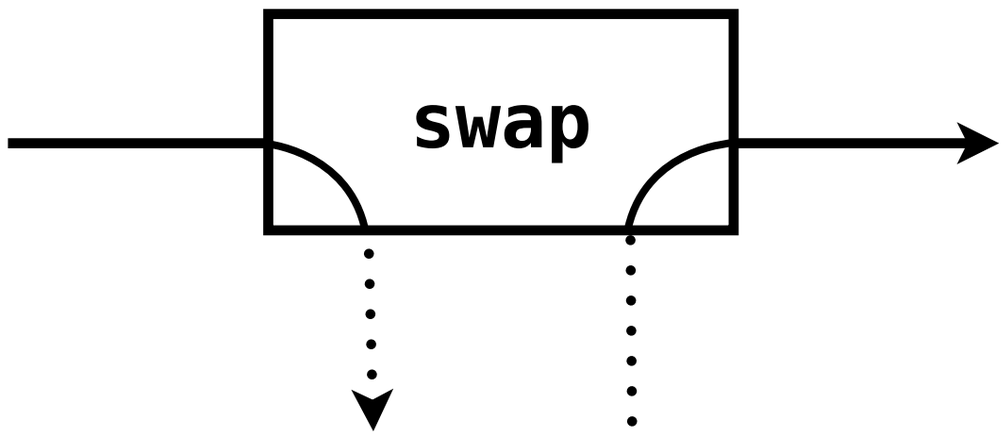
c: Chan[A,B]



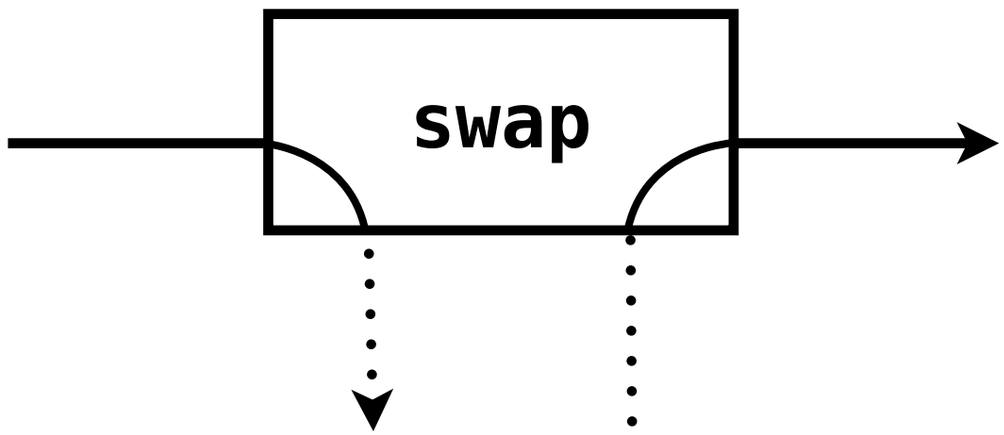
```
c: Chan[A,B]
```



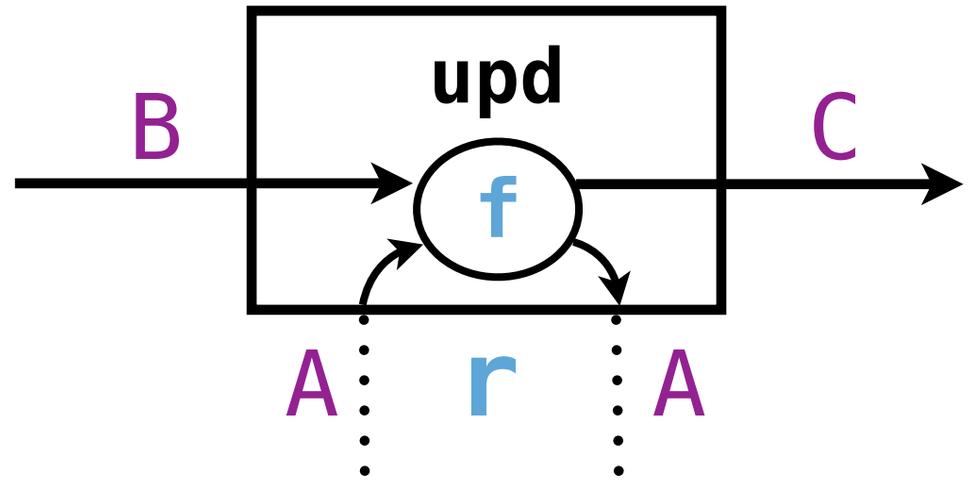
Message passing



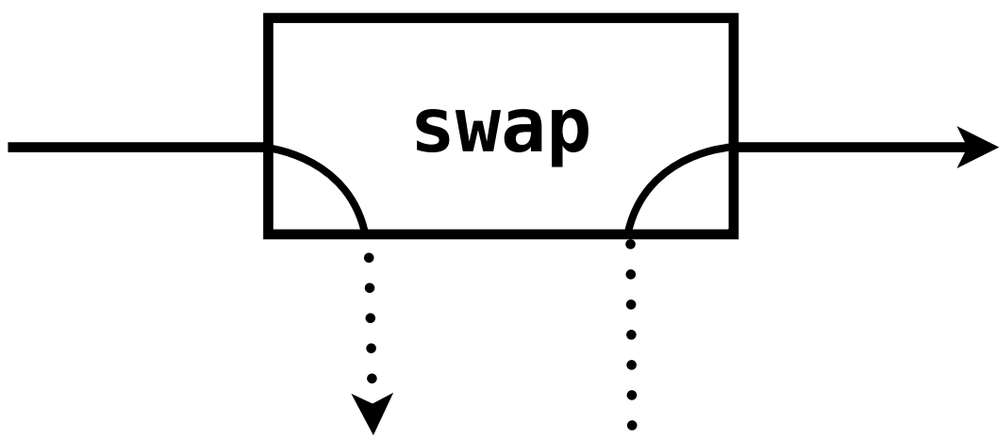
Message passing



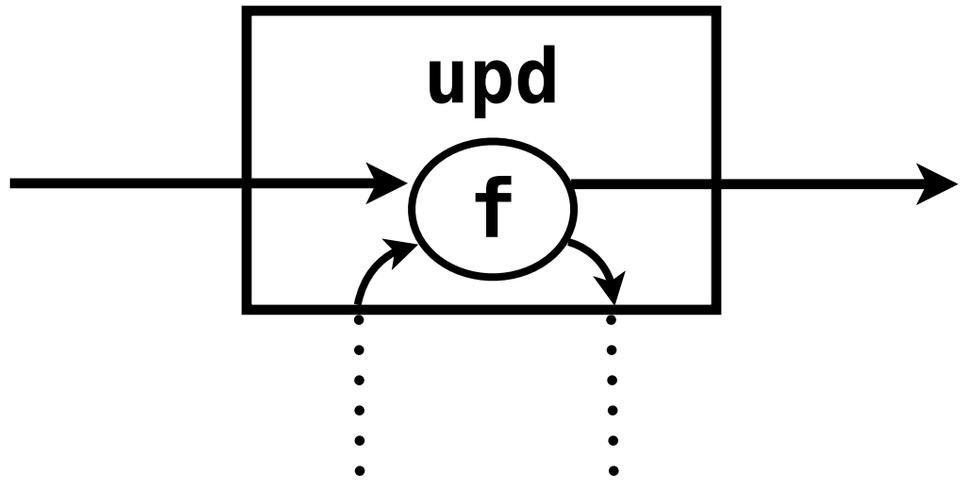

```
r: Ref[A]
f: (A,B)→(A,C)
```



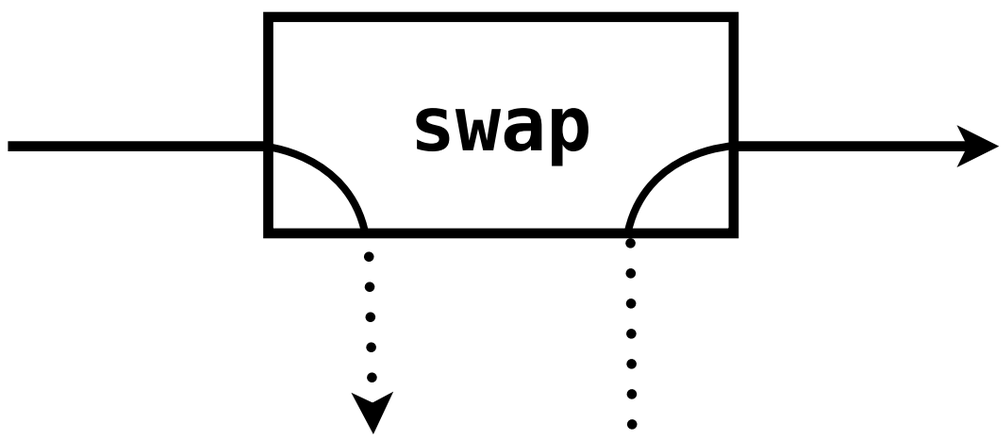
Message passing



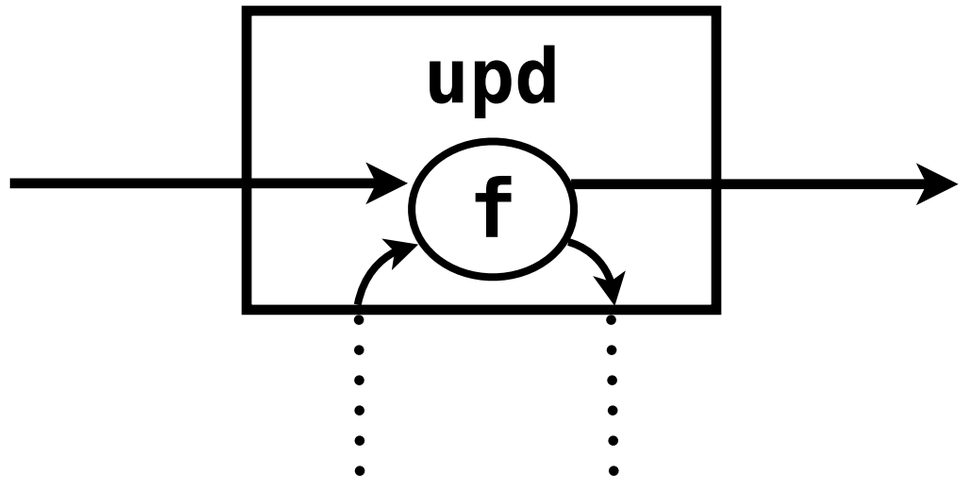
Shared state

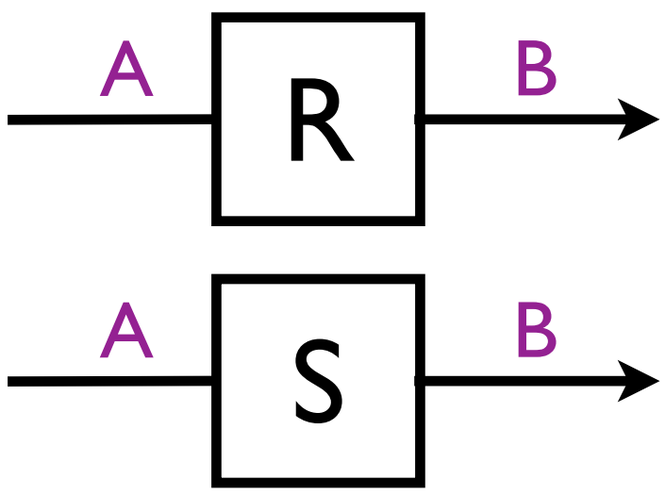


Message passing

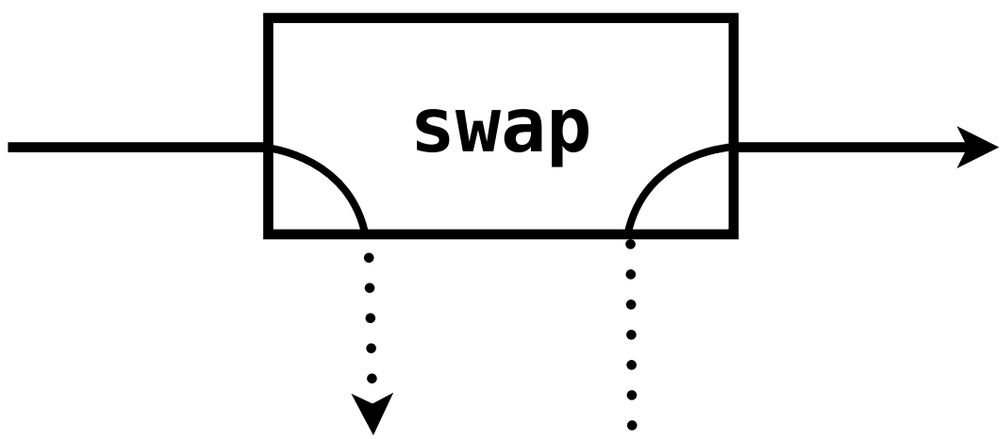


Shared state

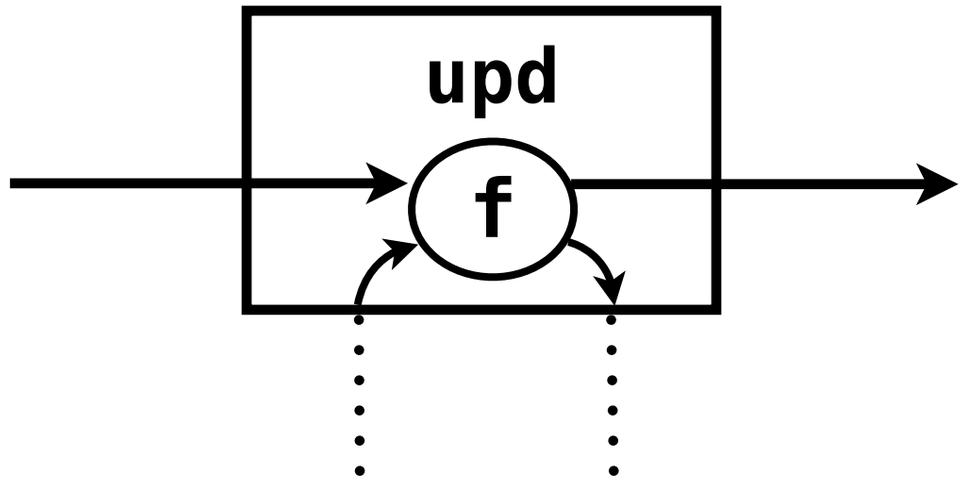


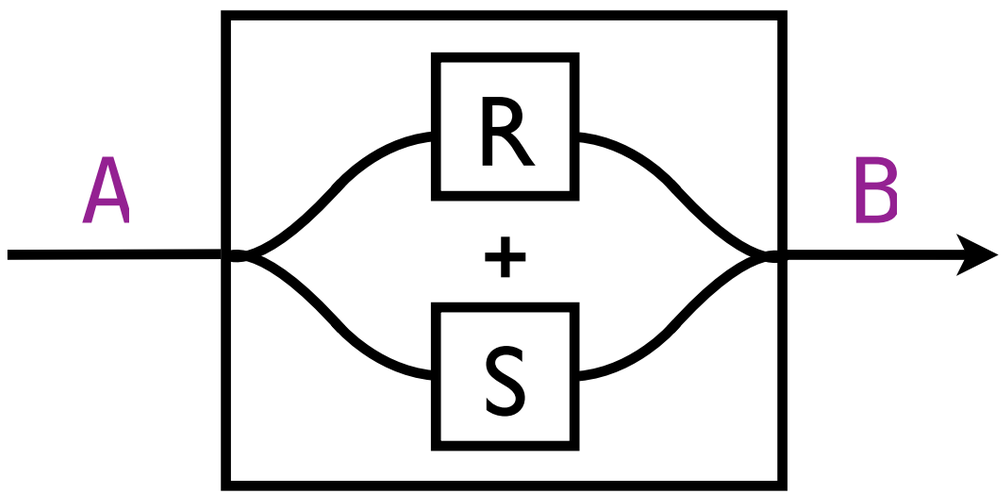


Message passing

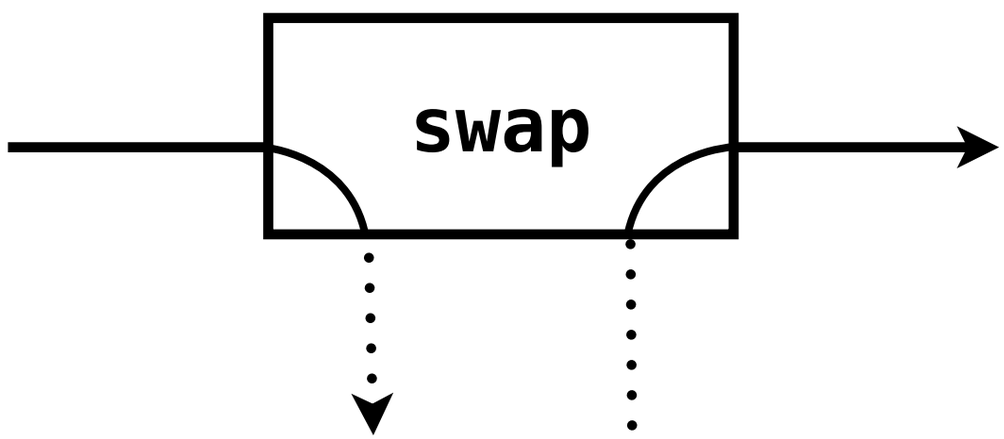


Shared state

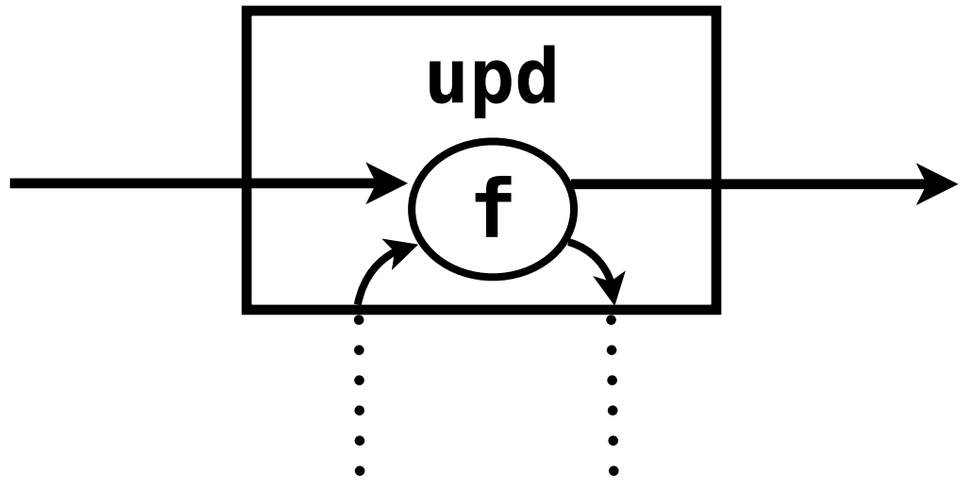




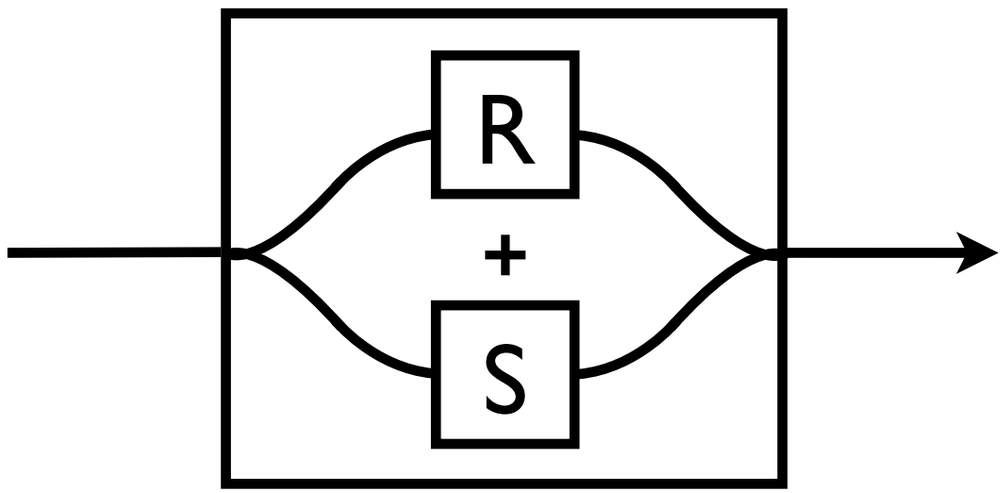
Message passing



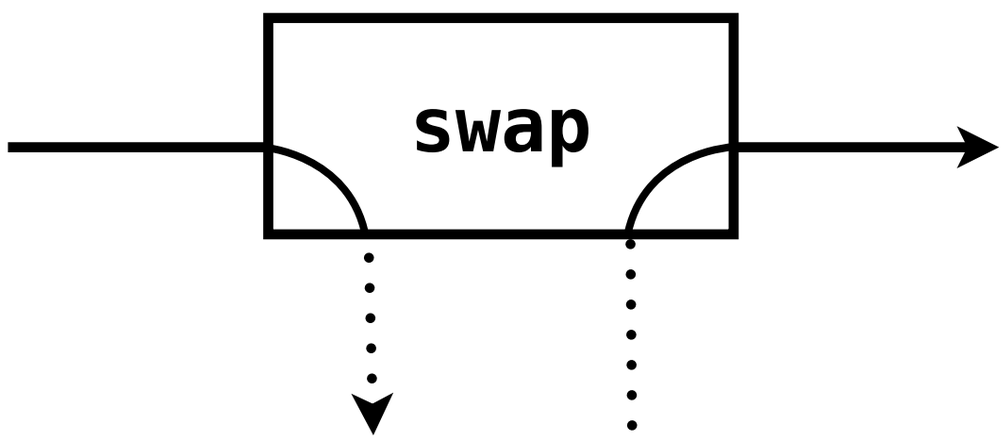
Shared state



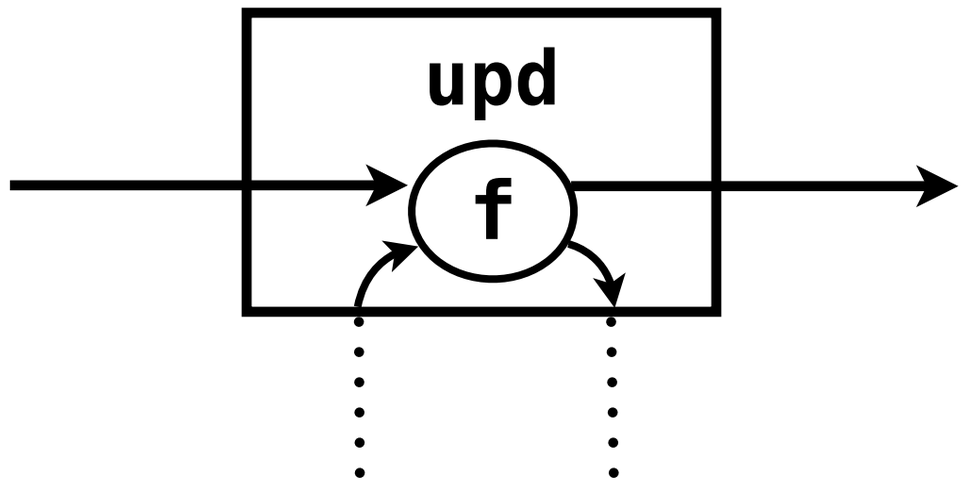
Disjunction



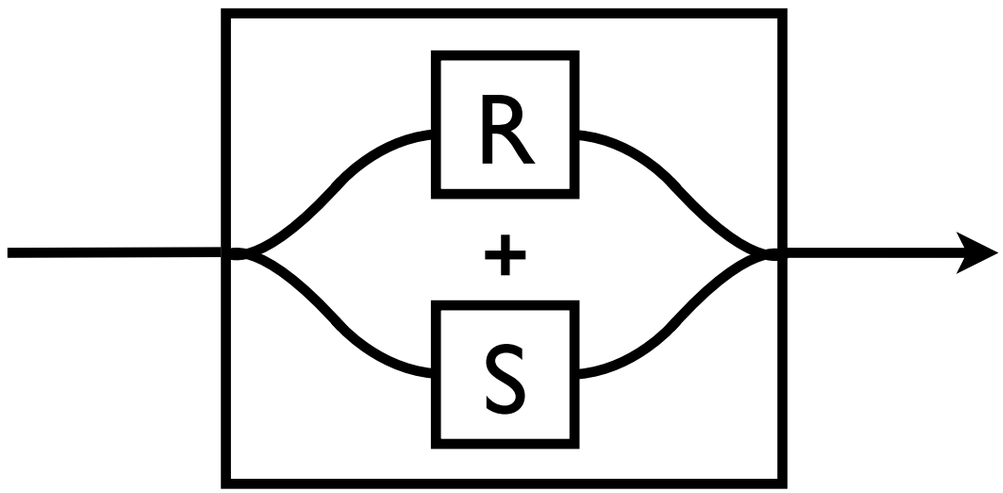
Message passing

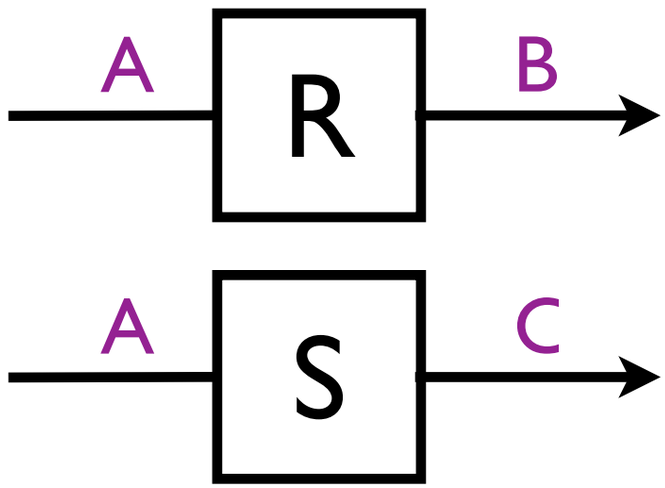


Shared state

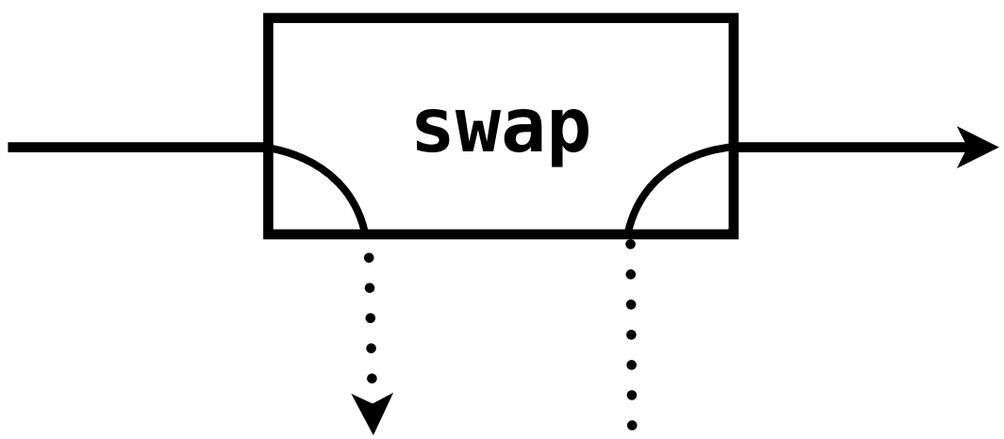


Disjunction

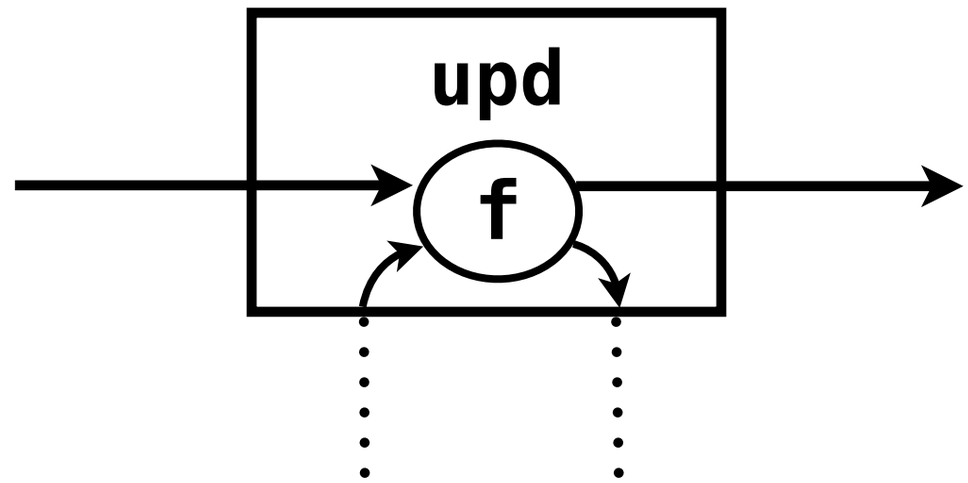




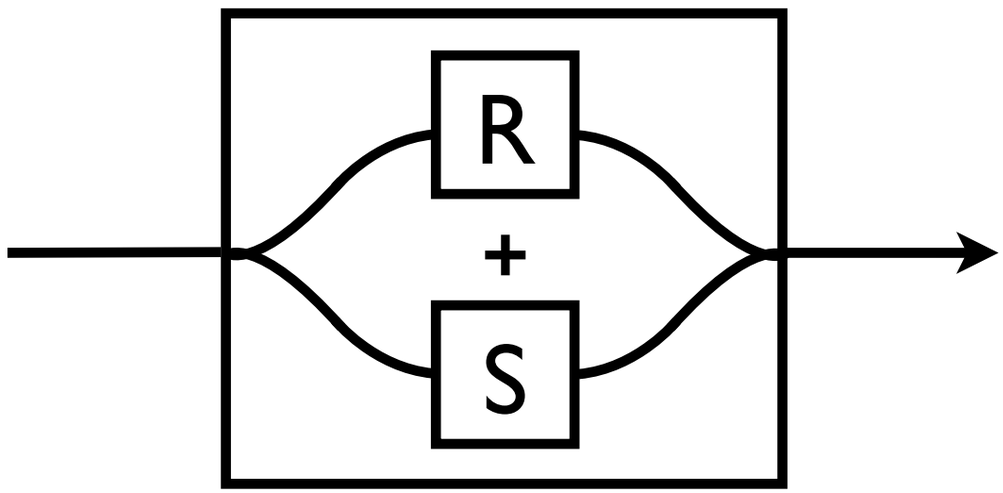
Message passing

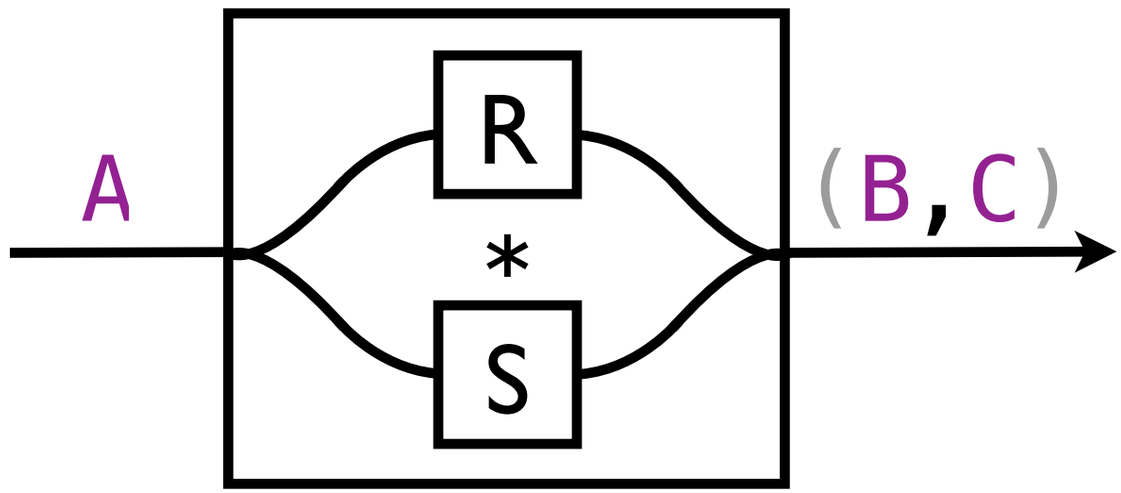


Shared state

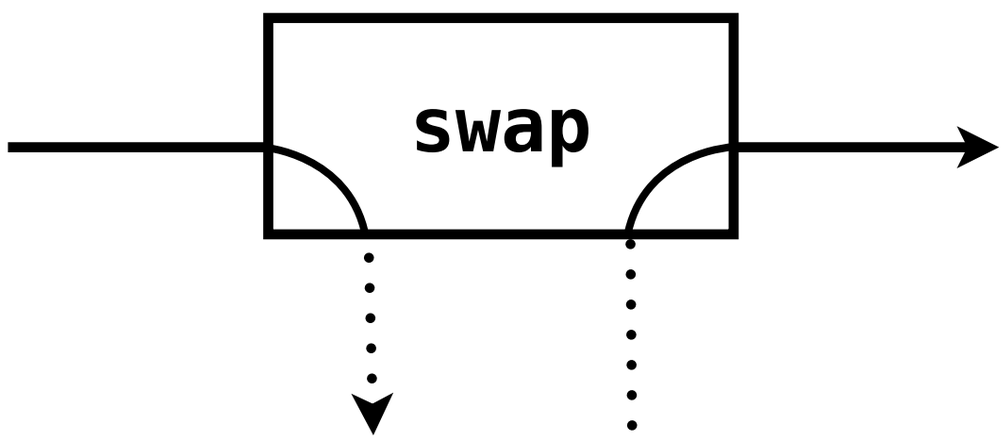


Disjunction

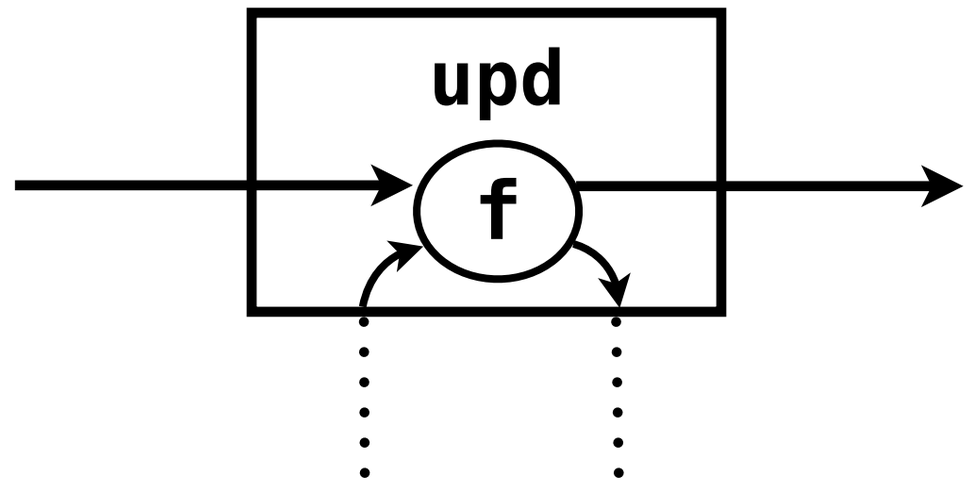




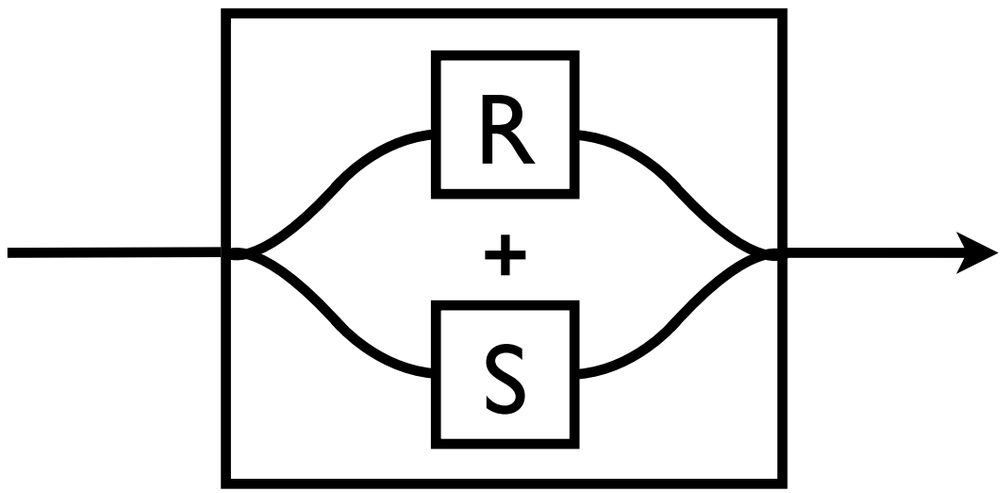
Message passing



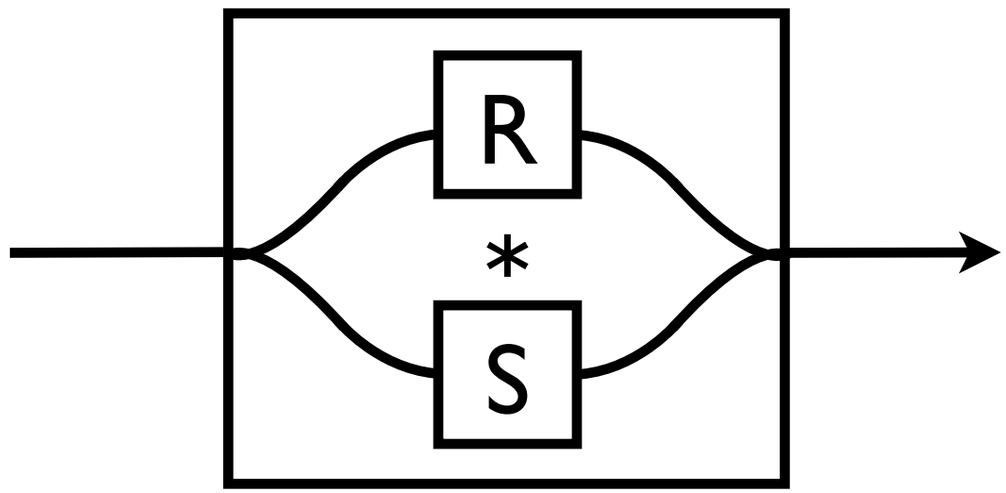
Shared state

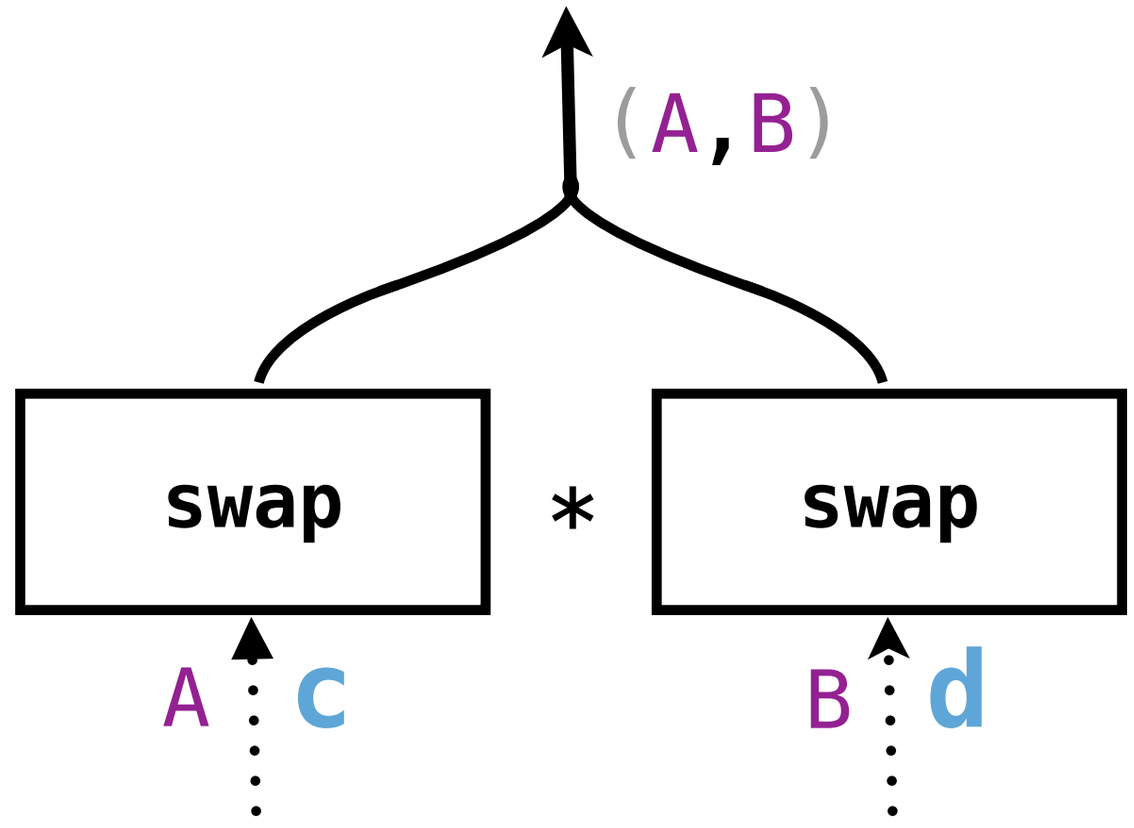


Disjunction

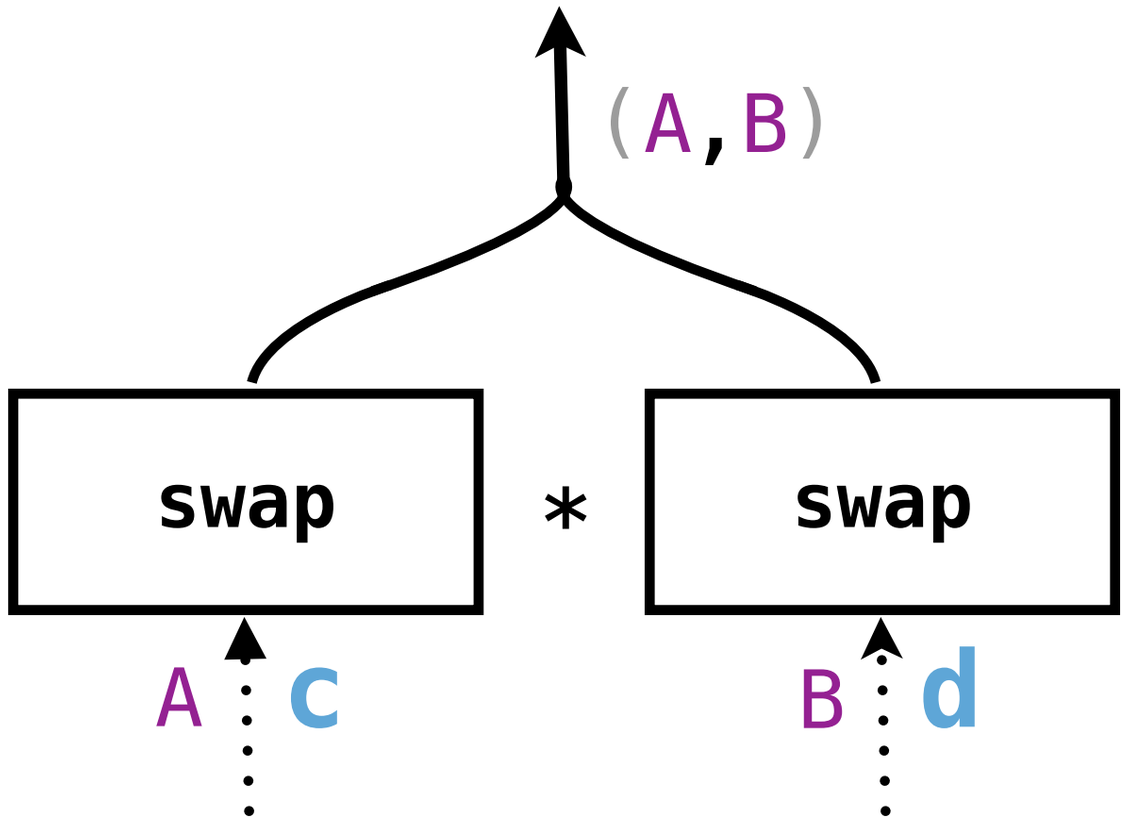


Conjunction

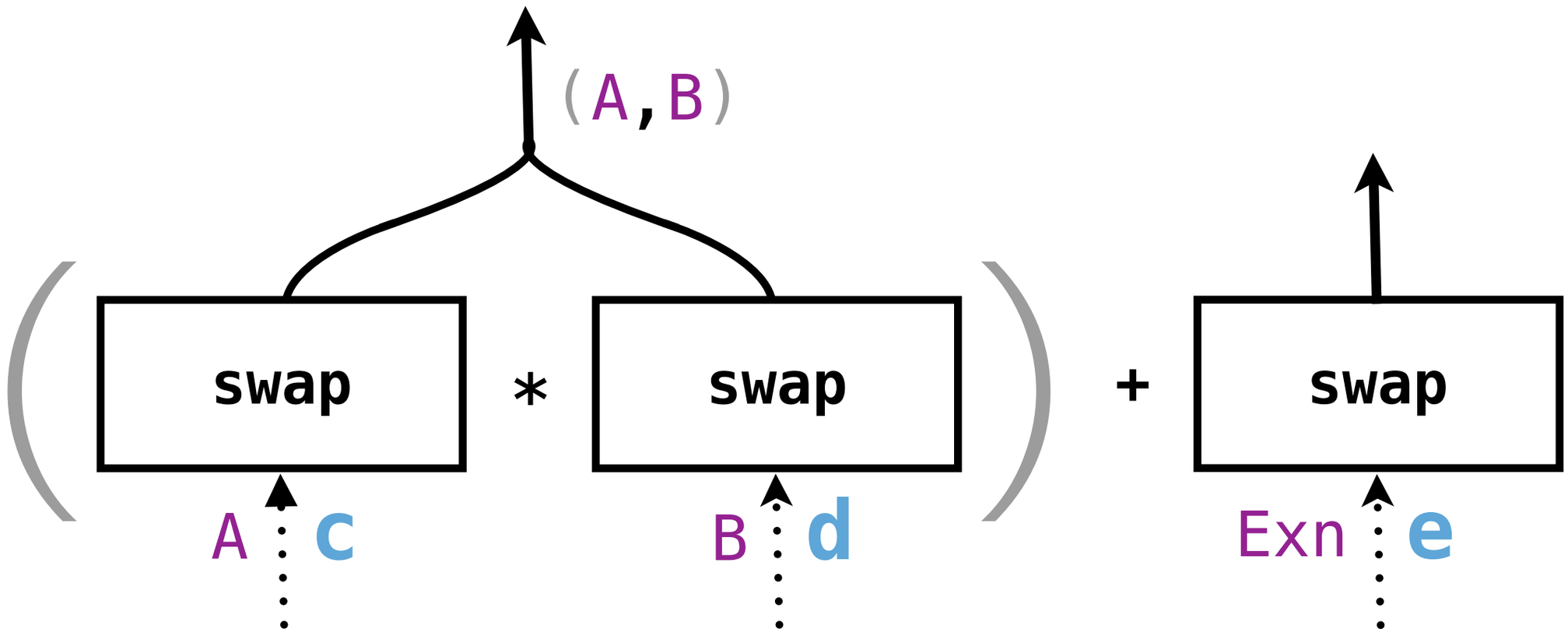



(A,B)
swap
*
swap
A
c
B
d

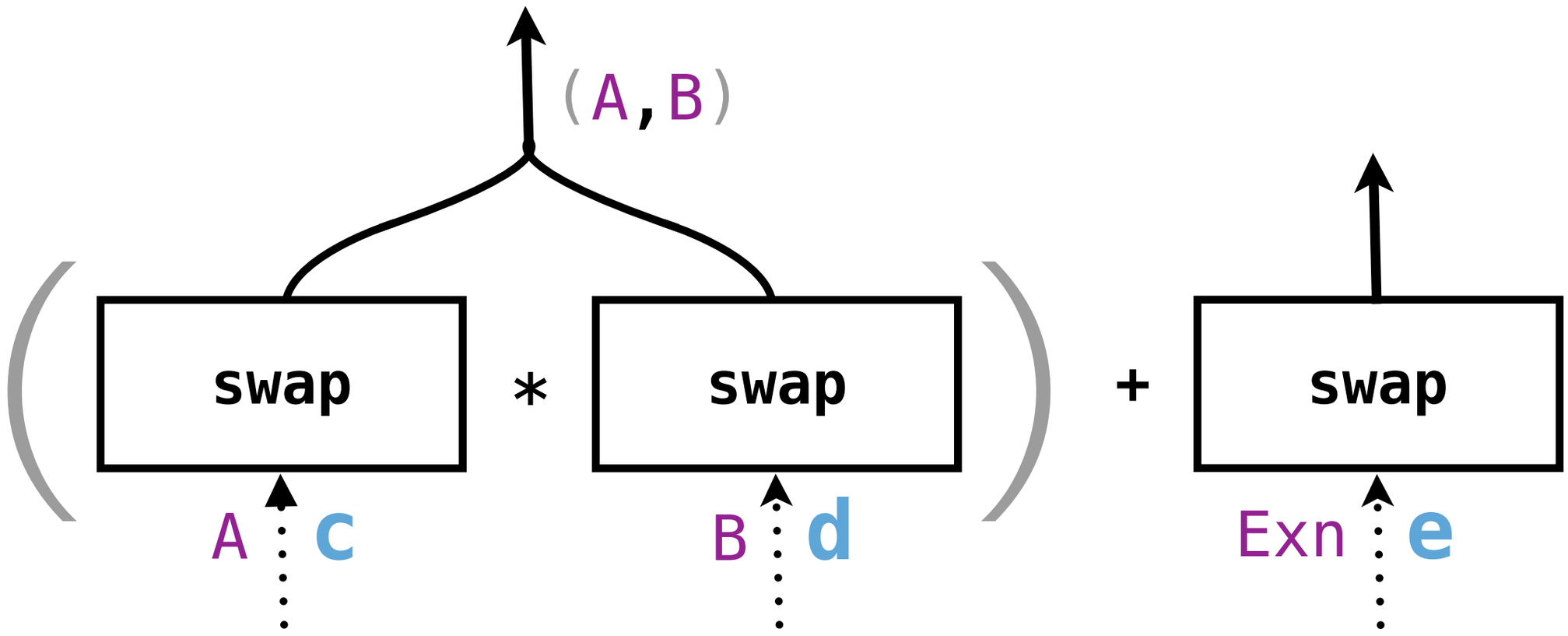
2-way join



2-way join



Abortable 2-way join



```scala
class TreiberStack [A] {
  private val head = new Ref[List[A]](Nil)
  val push    = upd(head)(cons)
  val tryPop = upd(head) {
    case (x :: xs) => (xs,  Some(x))
    case Nil       => (Nil, None)
  }
}
```

```
class TreiberStack [A] {
  private val head = new Ref[List[A]](Nil)
  val push    = upd(head)(cons)
  val tryPop = upd(head) {
    case (x :: xs) => (xs,  Some(x))
    case Nil       => (Nil, None)
  }
  val pop = upd(head) {
    case (x :: xs) => (xs,  x)
  }
}
```

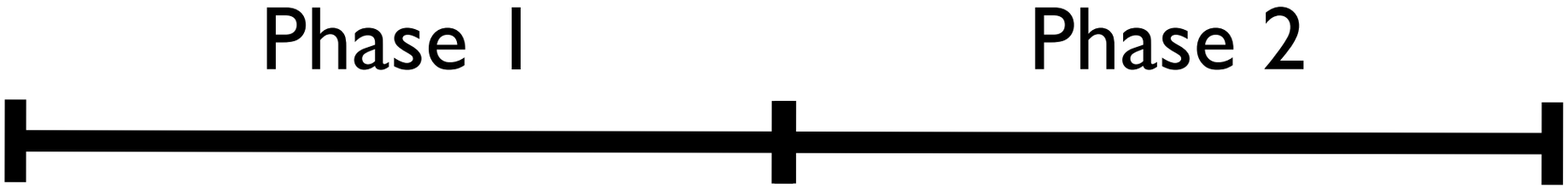
```
class TreiberStack [A] {
  private val head = new Ref[List[A]](Nil)
  val push    = upd(head)(cons)
  val tryPop = upd(head)(trySplit)
  val pop     = upd(head)(split)
}
```

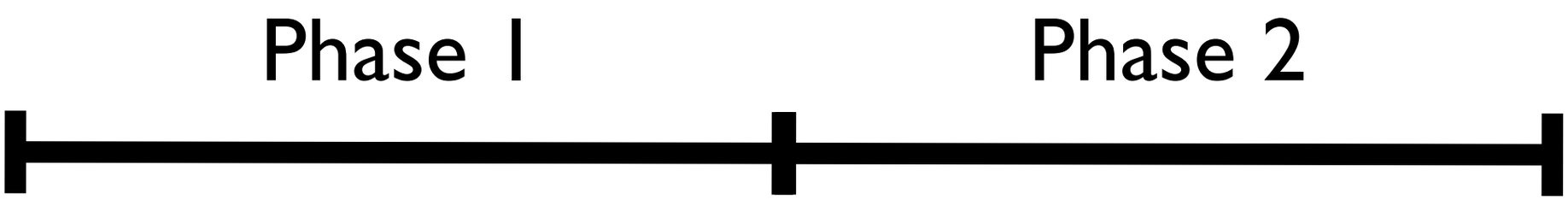
```
class TreiberStack [A] {
  private val head = new Ref[List[A]](Nil)
  val push    = upd(head)(cons)
  val tryPop = upd(head)(trySplit)
  val pop     = upd(head)(split)
}
```

```
class EliminationStack [A] {
  private val stack = new TreiberStack[A]
  private val (send, recv) = new Chan[A]
  val push = stack.push + swap(send)
  val pop  = stack.pop  + swap(recv)
}
```

stack1.pop >> stack2.push

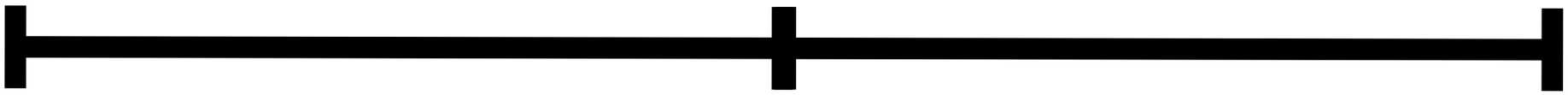
Implementation


Phase 1
Phase 2



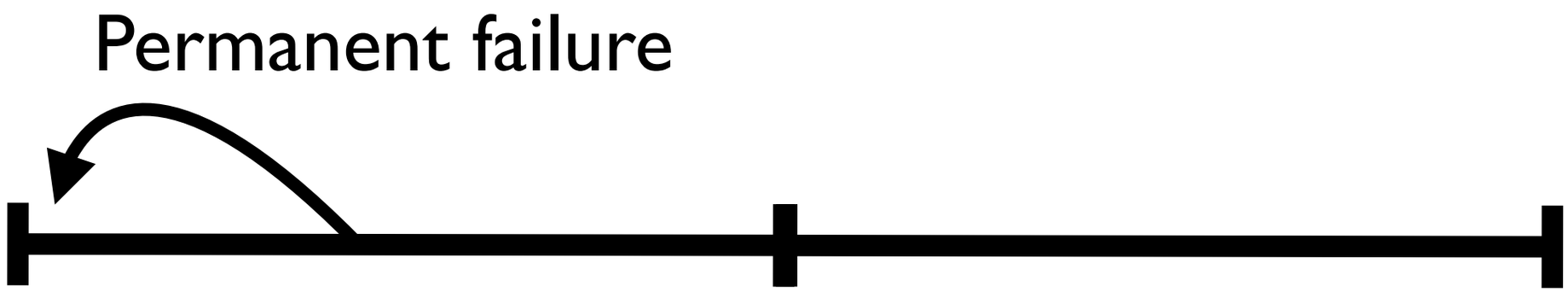
Accumulate CASes


Phase 1
Phase 2
Accumulate CASes
Attempt k-CAS



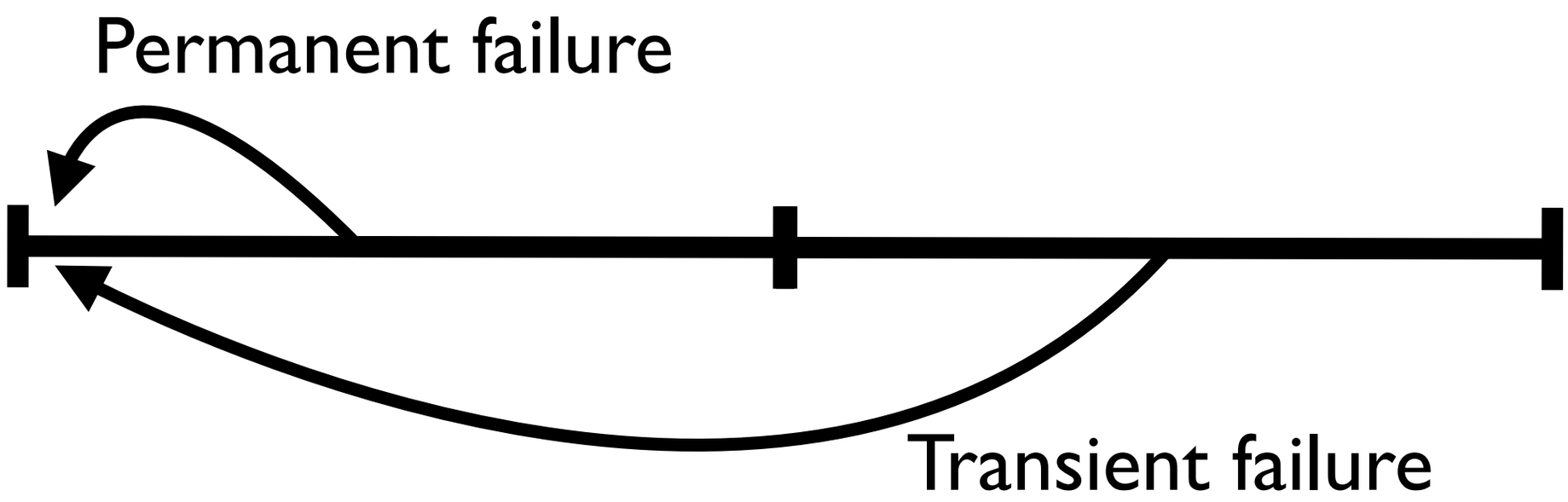
Accumulate CASes

Attempt k-CAS



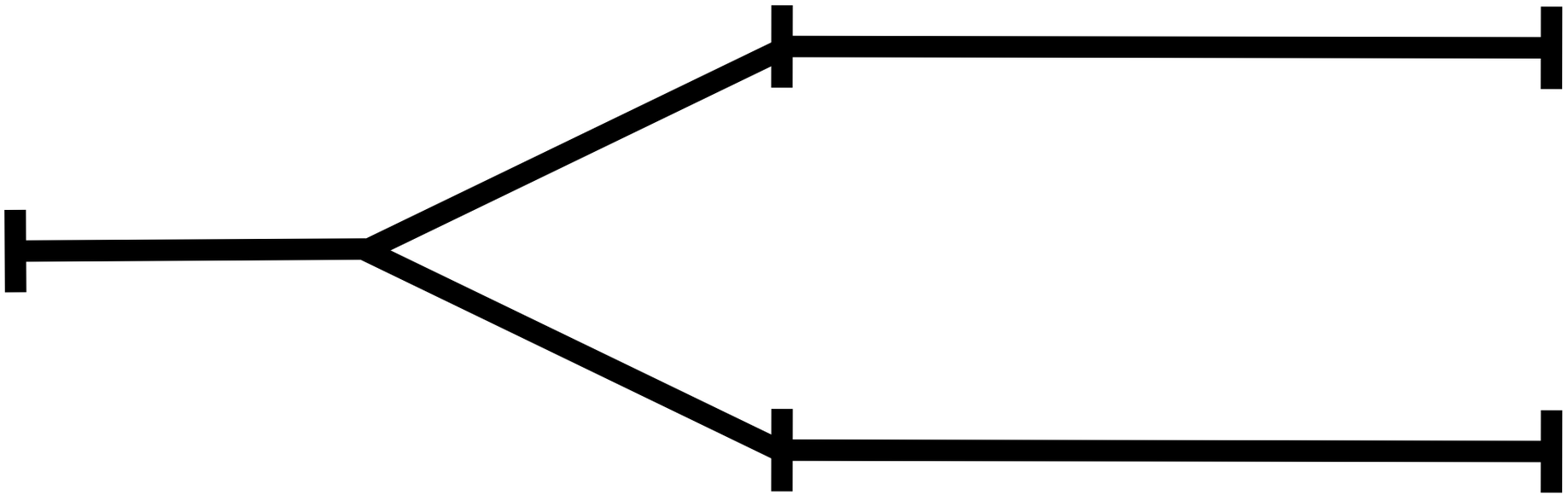
Accumulate CASes

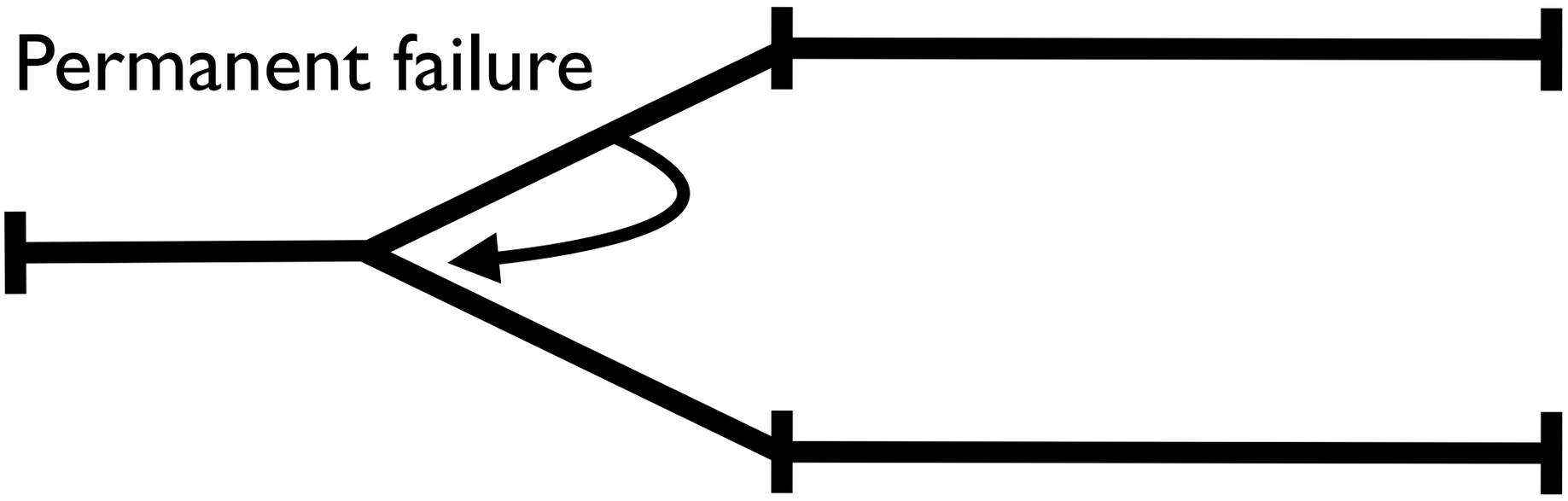
Attempt k-CAS

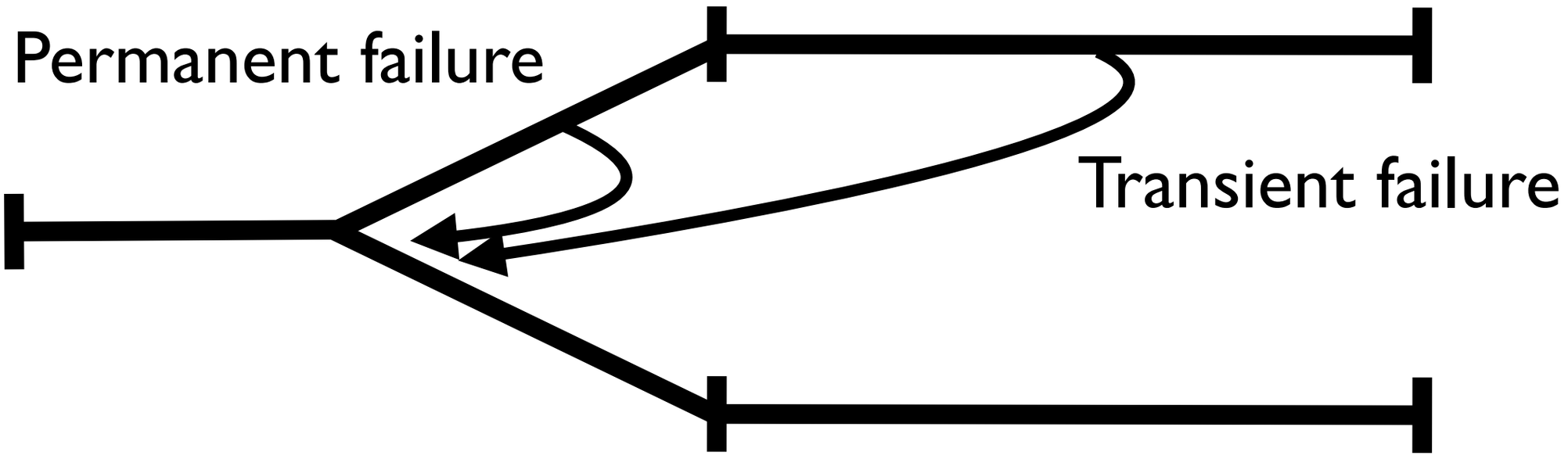


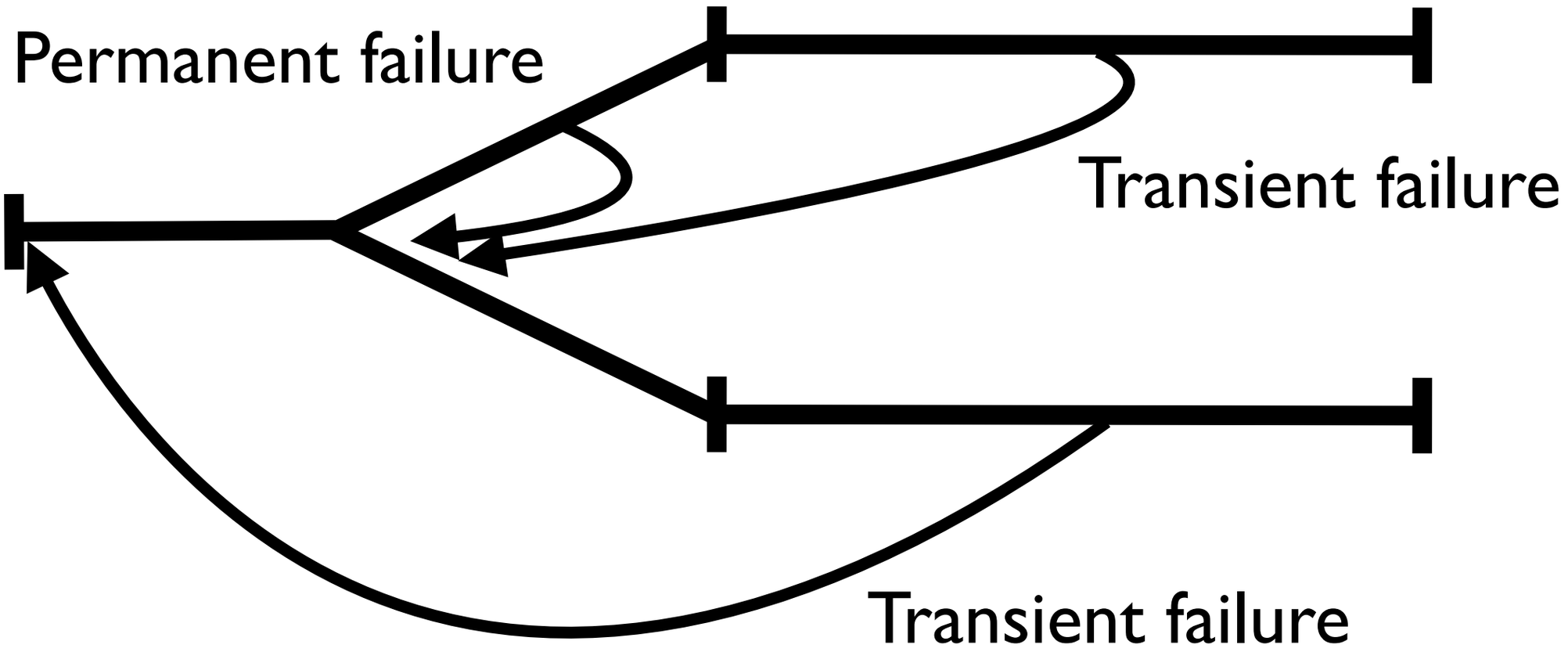
Accumulate CASes

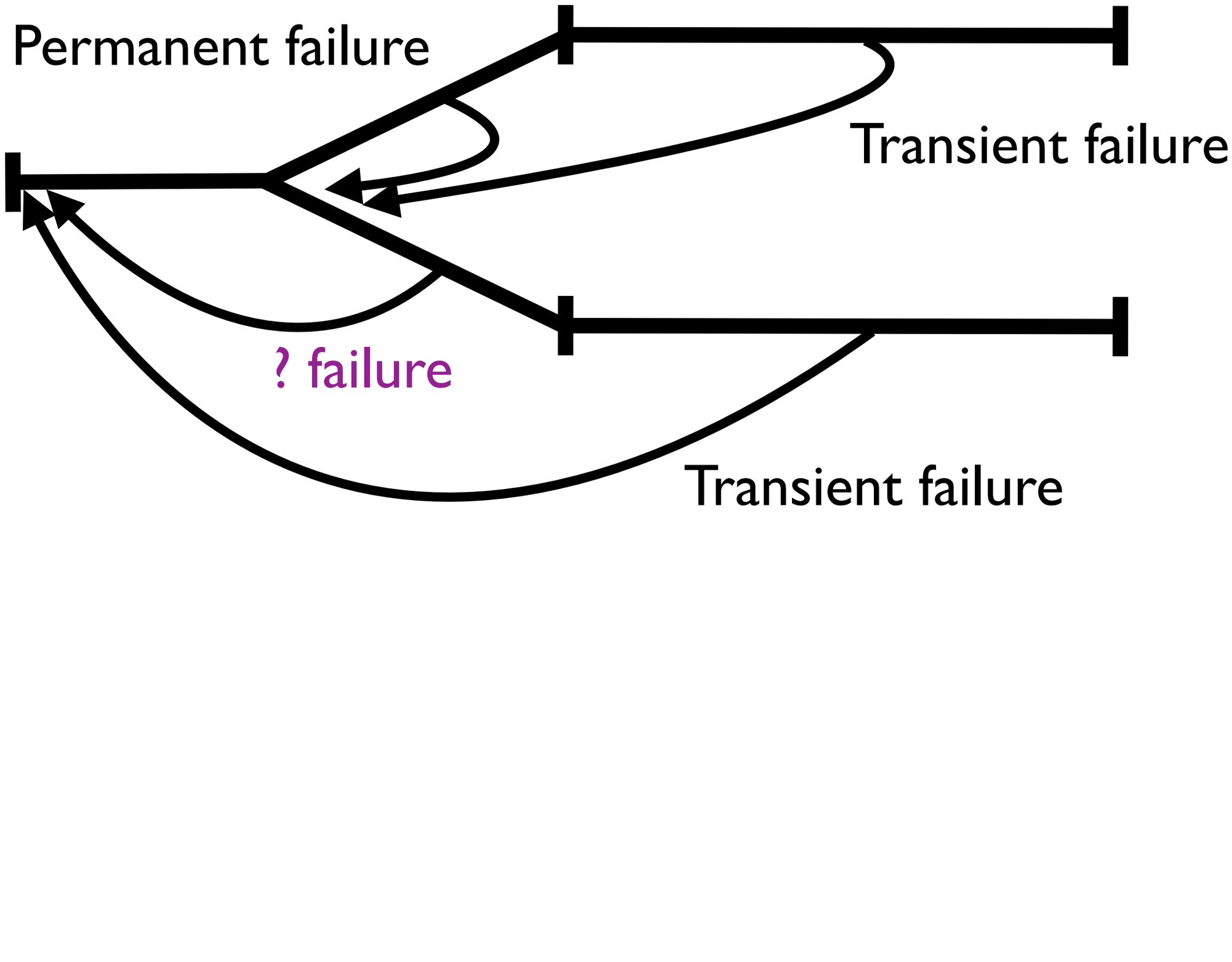
Attempt k-CAS

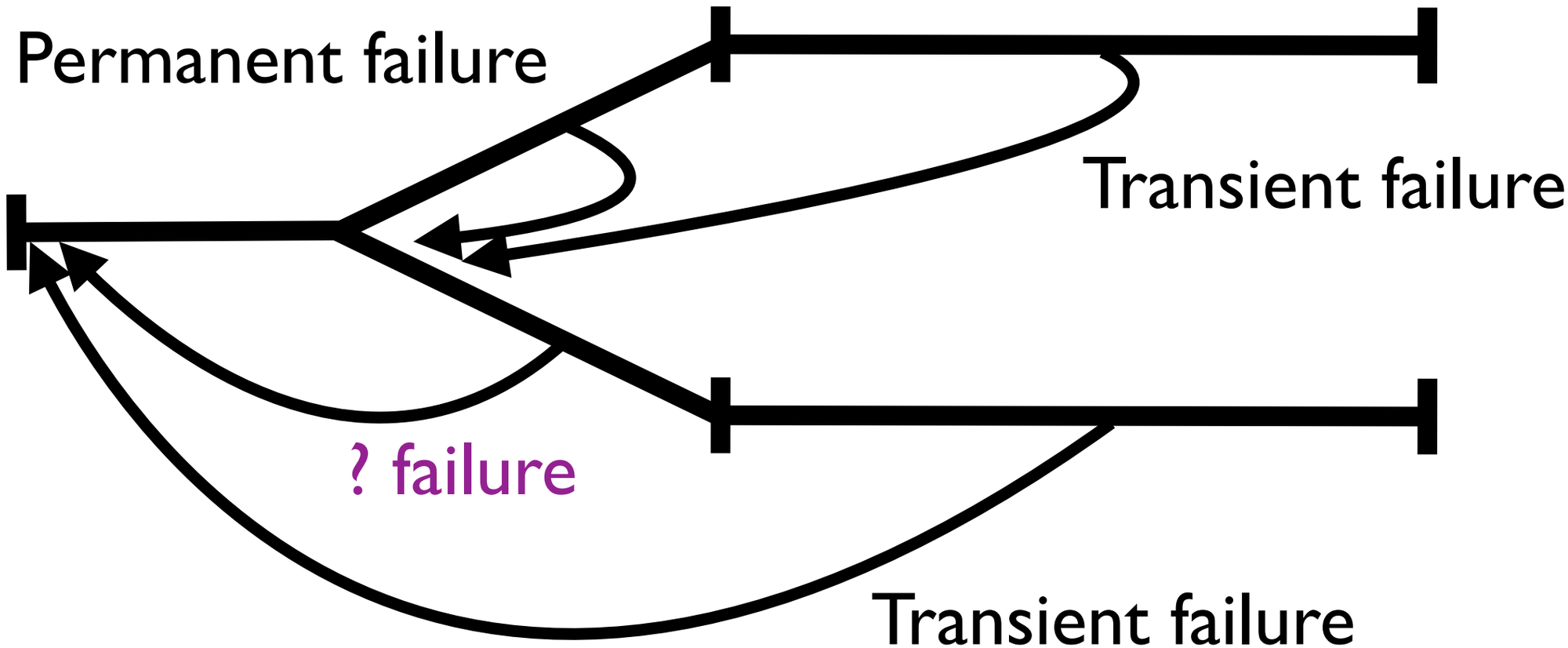



Permanent failure


Permanent failure
Transient failure


Permanent failure
Transient failure
Transient failure


Permanent failure
Transient failure
? failure
Transient failure



P & P = P

P & T = T

T & T = T

T & P = T

Is this just STM?

Is this just STM?

No:

- Single CAS collapses to single phase
- Multiple CASes to single location forbidden

So the "redo log" is write-only for phase 1

Therefore: **pay-as-you-go**

- Treiber stack is really a Treiber stack
- Pay for kCAS only for compositions

Is this just STM?

Isolation
Shared state

Interaction
Message passing

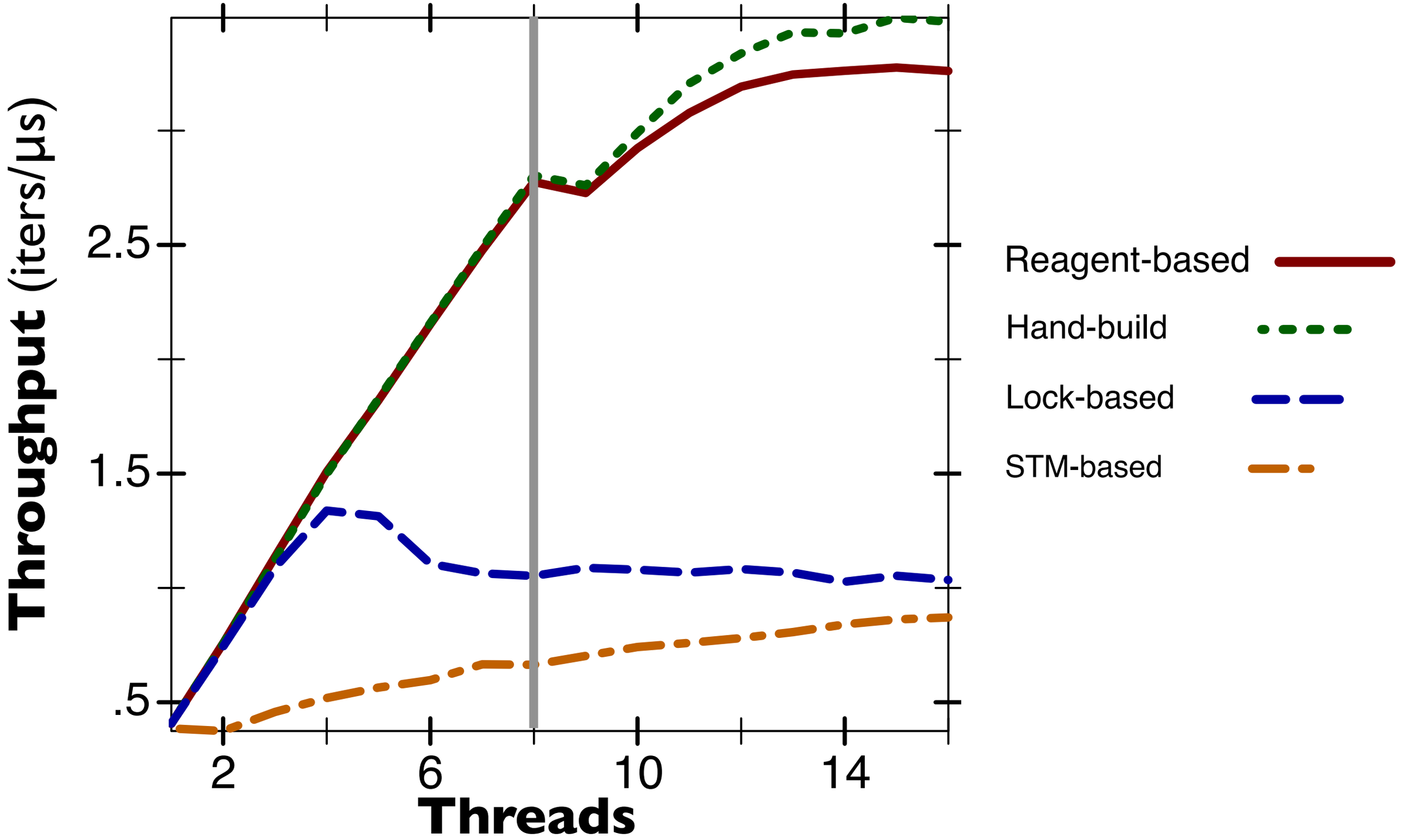
Is this just STM?

Isolation
Shared state

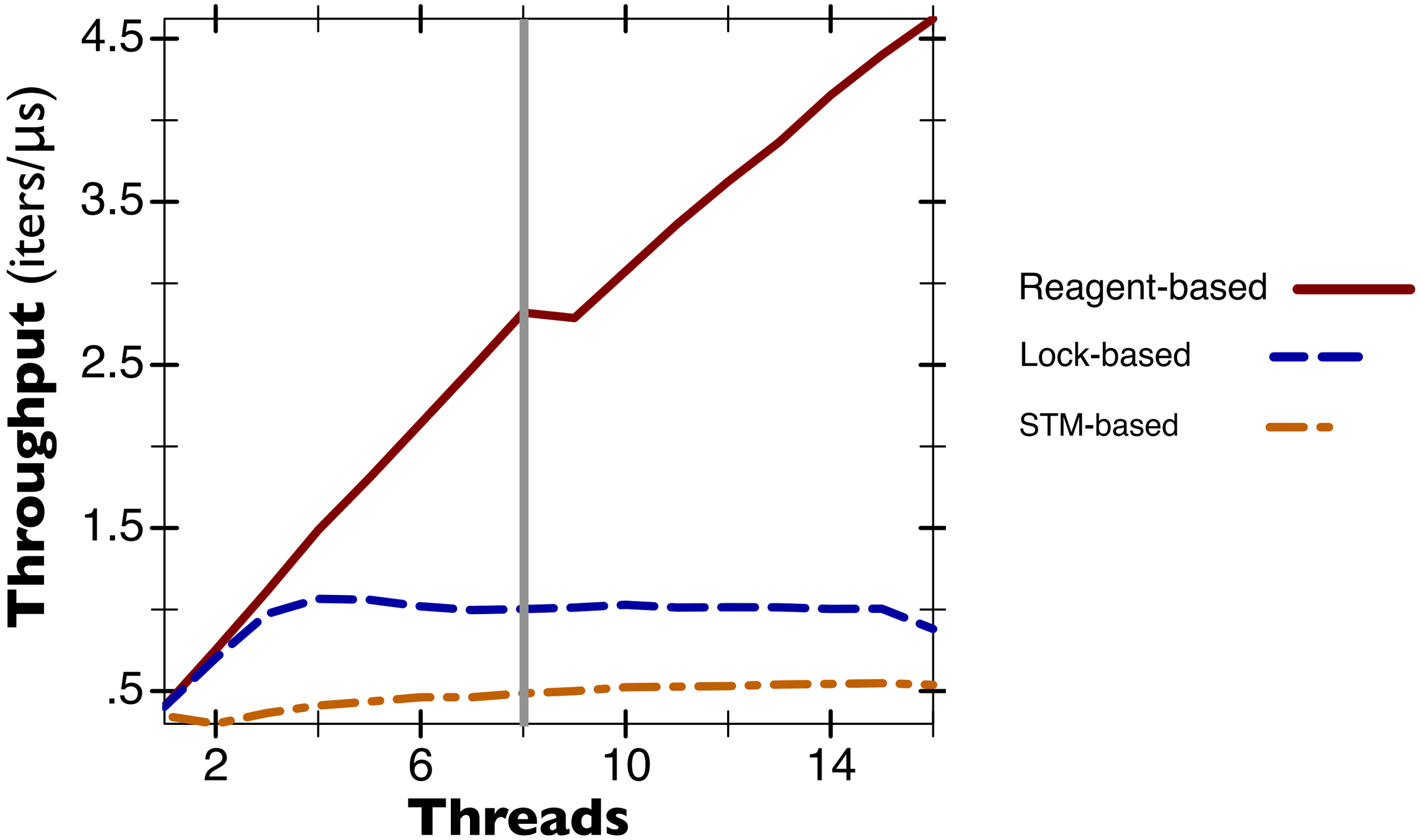
Interaction
Message passing

Using lock-free bags,
based on earlier work
with Russo [OOPSLA'11]

Treiber stack



Stack transfer



java.util.concurrent

Synchronization

Reentrant locks
Semaphores
R/W locks
Reentrant R/W locks
Condition variables
Countdown latches
Cyclic barriers
Phasers
Exchangers

Data structures

Queues
- Nonblocking
- Blocking (array & list)
- Synchronous
- Priority, nonblocking
- Priority, blocking

Deques
Sets
Maps (hash & skiplist)

CHEMISTRYSET

Synchronization

Reentrant locks
Semaphores
R/W locks
Reentrant R/W locks
Condition variables
Countdown latches
Cyclic barriers
Phasers
Exchangers

Data structures

Queues
- Nonblocking
- Blocking
- Synchronous
- Priority, nonblocking
- Priority, blocking

Deques
Sets
Maps (hash & skiplist)

The take-away:

Reagents enable scalable concurrent algorithms to be built and extended using abstraction and composition

https://github.com/aturon/ChemistrySet