scalaz-stream

Reactive in Reverse


WARNING!
BUYER
BEWARE

Pull vs Push

- Push streams
  - Data assertively *pushed* into your flow
  - Naturally runs in parallel

Pull vs Push

- Push streams
    - Data assertively *pushed* into your flow
    - Naturally runs in parallel
- Pull streams
    - "Turn the crank" from the end and request data
    - Backpressure by definition

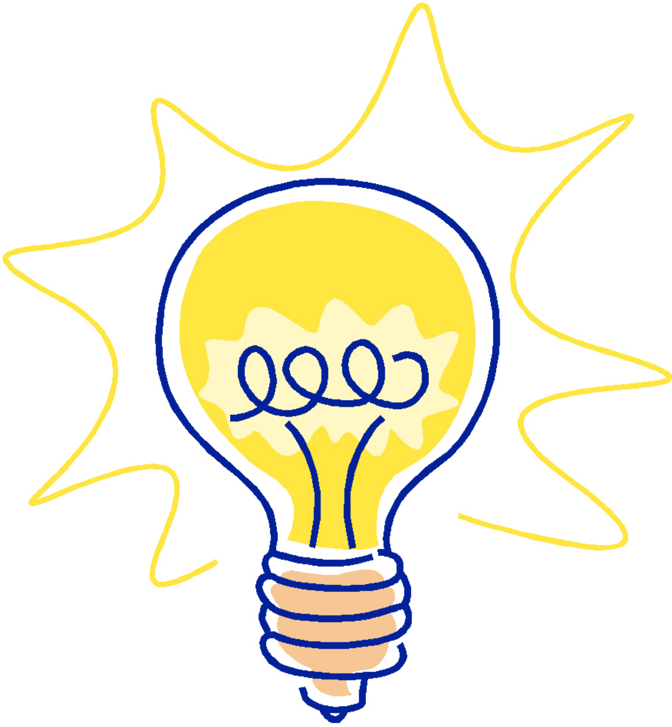
Pull vs Push

- Push streams
    - Backpressure is something you need to design
    - More intuitive control flow (imperatively)

Pull vs Push

- Push streams
    - Backpressure is something you need to design
    - More intuitive control flow (imperatively)
- Pull streams
    - Concurrency doesn't exist
    - More declarative control, which can be weird

Concepts



- `Task[A]`
  - Like `Future`, but more controlled
- `Process[Task, A]`
  - A strict sequence of *actions*

Concepts: `Task`

- Fully lazy

Concepts: `Task`

- Fully lazy
  - Creating a `Future` executes *immediately*

Concepts: `Task`

- Fully lazy
  - Creating a `Future` executes *immediately*
  - No more memory leaks!

Concepts: `Task`

- Fully lazy
  - Creating a `Future` executes *immediately*
  - No more memory leaks!
- Easy to move tasks between thread pools

Concepts: `Task`

- Fully lazy
  - Creating a `Future` executes *immediately*
  - No more memory leaks!
- Easy to move tasks between thread pools
- Better thread utilization

Concepts: `Task`

- Fully lazy
  - Creating a `Future` executes *immediately*
  - No more memory leaks!
- Easy to move tasks between thread pools
- Better thread utilization
- Explicit parallelism

```scala
def fib(n: Int): Task[Int] = n match {
  case 0 | 1 => Task now 1
  case n => {
    for {
      x <- fib(n - 1)
      y <- fib(n - 2)
    } yield x + y
  }
}

fib(42).run
```

```scala
def fib(n: Int): Task[Int] = n match {
  case 0 | 1 => Task now 1
  case n => {
    val ND = Nondeterminism[Task]

    for {
      pair <- ND.both(fib(n - 1), fib(n - 2))
      (x, y) = pair
    } yield x + y
  }
}

fib(42).run
```

```
def shiftPool[A](task: Task[A]): Task[A] =
  Task({ task })(MyThreadPool).join
```

```scala
def shiftPool[A](task: Task[A]): Task[A] =
  Task.fork(task)(MyThreadPool)
```

```scala
def futureToTask[A](f: Future[A]): Task[A] = {
  Task async { cb =>
    f onComplete {
      case Success(v) => cb(\/.right(v))
      case Failure(e) => cb(\/.left(v))
    }
  }
}
```

```scala
def futureToTask[A](f: Future[A]): Task[A] = {
  Task async { cb =>
    f onComplete {
      case Success(v) => cb(\/.right(v))
      case Failure(e) => cb(\/.left(v))
    }
  }
}
```

Concepts: `Process`

- An ordered sequence of *actions*
- Ask for an action…then the next…then the next
  - If you can't keep up, you ask less frequently
- Easy to merge (just ask for data from either "side")
- Explicit parallelism

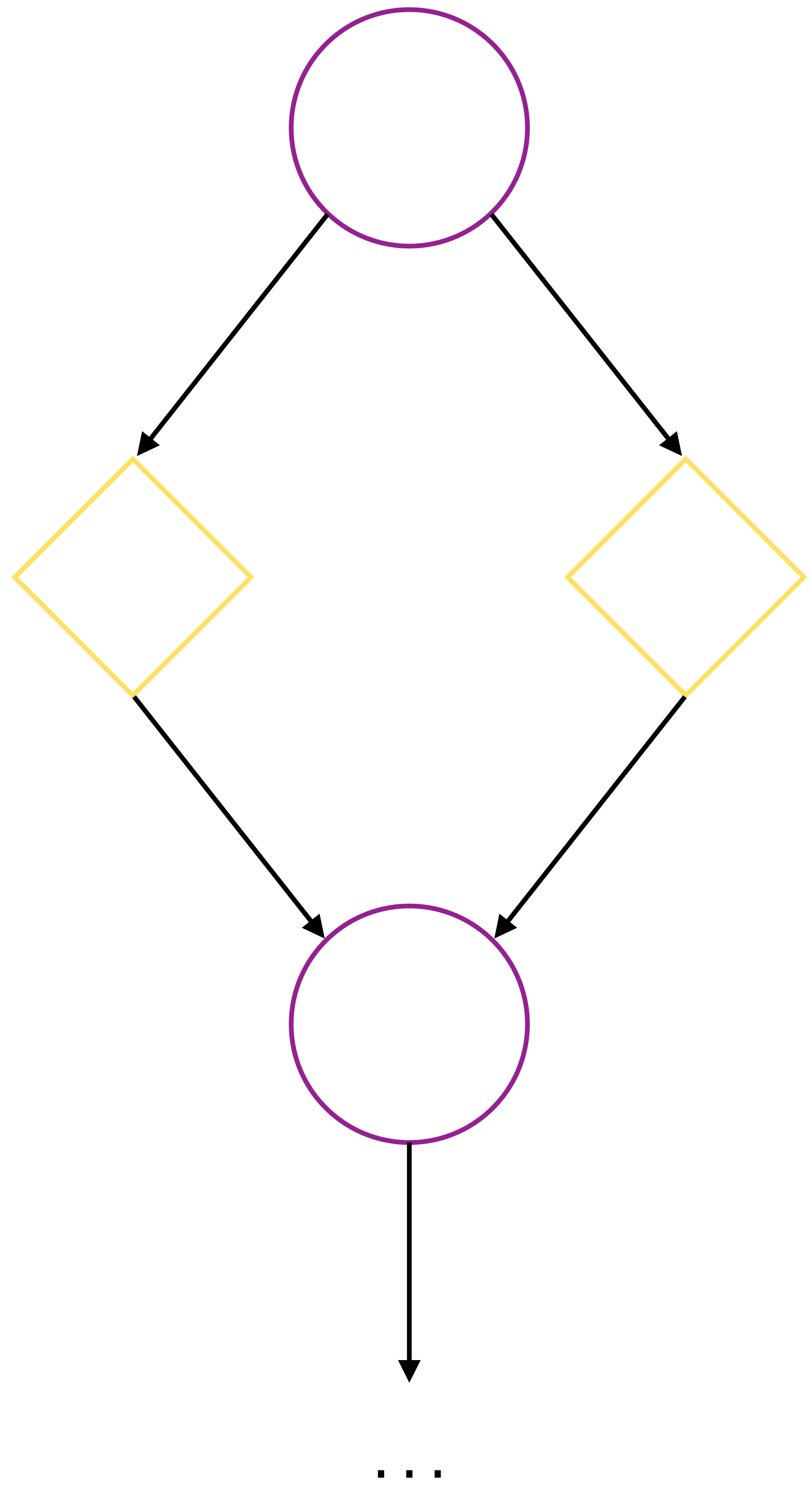
```scala
def fetchUrl(num: Int): Task[String] = {
  val fetch: Task[Task[String]] = Task delay {
    val svc = url(s"http://api.stuff.com/record/$num")
    Task fork futureToTask(Http(svc OK as.String))
  }

  fetch.join
}
```

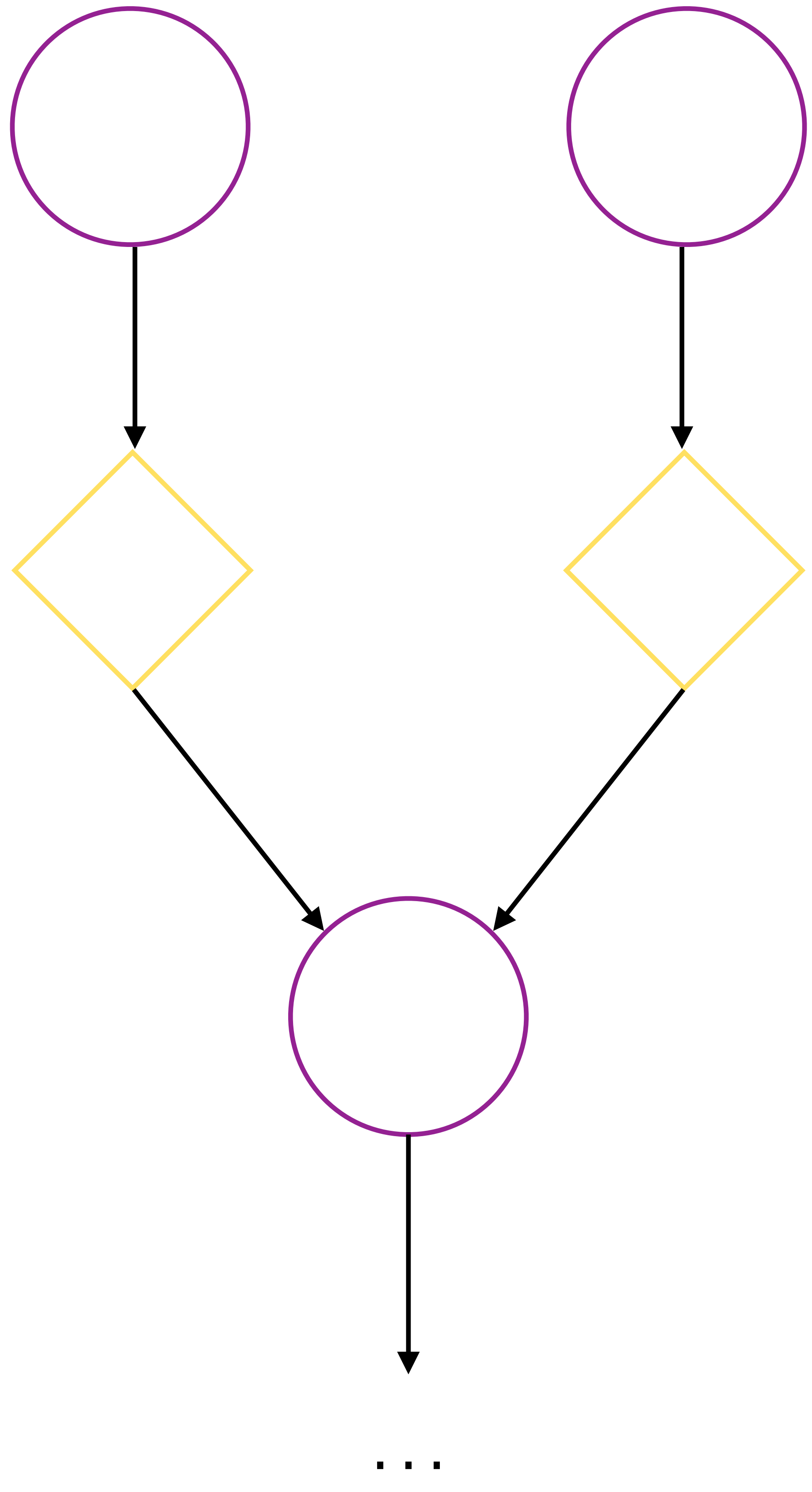
```scala
val nums: Process[Task, Int] = Process.range(0, 10)
val adjusted = nums map { _ * 2 } filter { _ < 10 }

val pages = adjusted flatMap { num =>
  Process.eval(fetchUrl(num))
}

val found = pages find { _ contains "Waldo!" }

val stuff: Task[Unit] = found to io.stdOutLines run

stuff.run
```

```scala
val nums1: Process[Task, Int] = Process.range(0, 10)
val nums2: Process[Task, Int] = Process.range(11, 20)

val nums: Process[Task, Int] = nums1 interleave nums2

...
```

```scala
val i = new AtomicInteger
val read = Task delay {
  i.getAndIncrement()
}

val src = Process.eval(read).repeat

val left = src map { i => s"left: $i" }
val right = src map { i => s"right: $i" }

left interleave right to io.stdOutLines
```

```
left: 0
right: 1
left: 2
right: 3
left: 4
right: 5
left: 6
right: 7
left: 8
right: 9
left: 10
right: 11
left: 12
right: 13
...
```





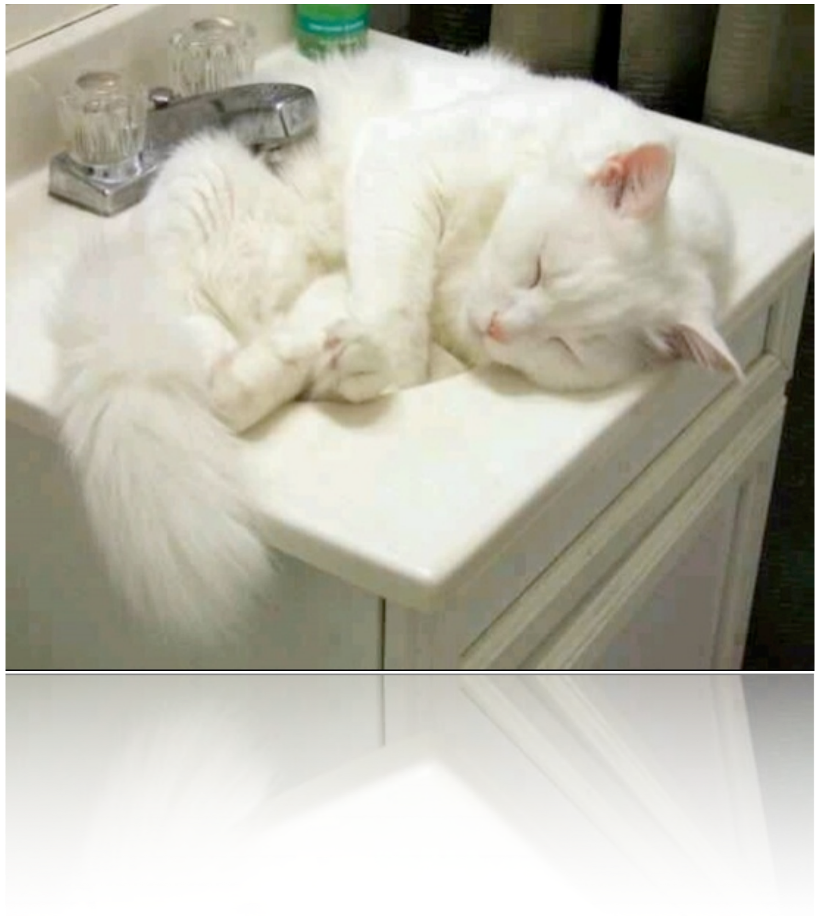
```
// bounded queues are for wimps...
```

```scala
// bounded queues are for wimps...
val queue = new ArrayBlockingQueue[Message](10)

// looks like I'm a wimp

val read: Task[Message] = Task delay { queue.take() }

val src: Process[Task, Message] =
  Process.eval(read).repeat

...
```

```scala
val queue = async.blockingQueue[Message](10)

val src: Process[Task, Message] = queue.dequeue
...
```

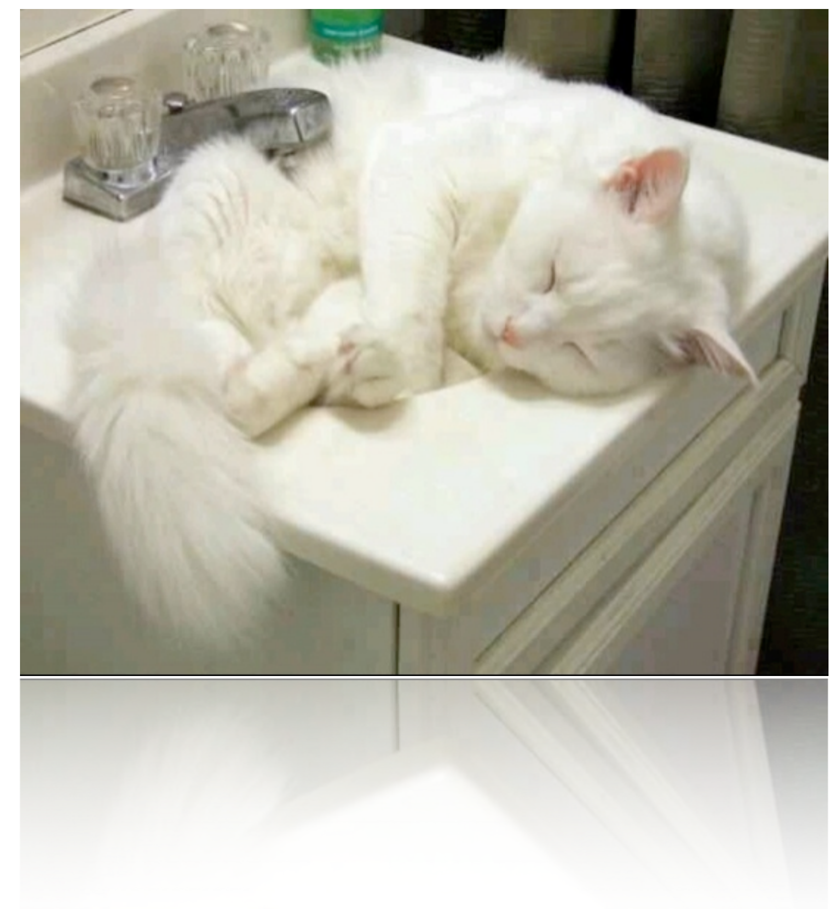
Sinks

- Data has to go somewhere



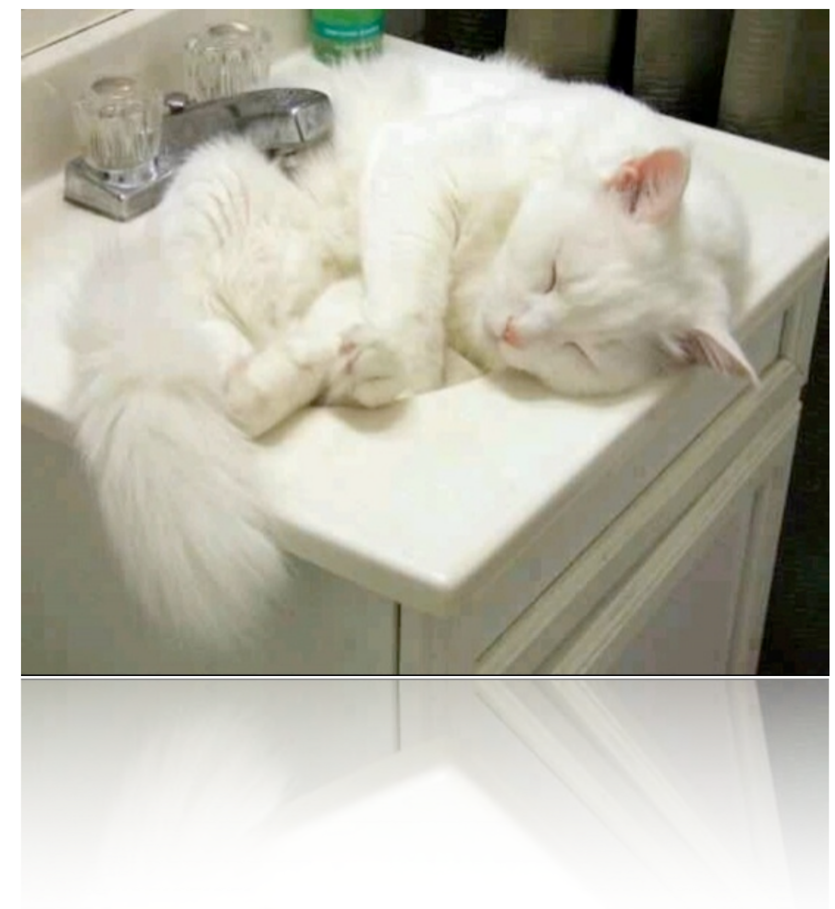
Sinks

- Data has to go somewhere
  - Writing out to a channel



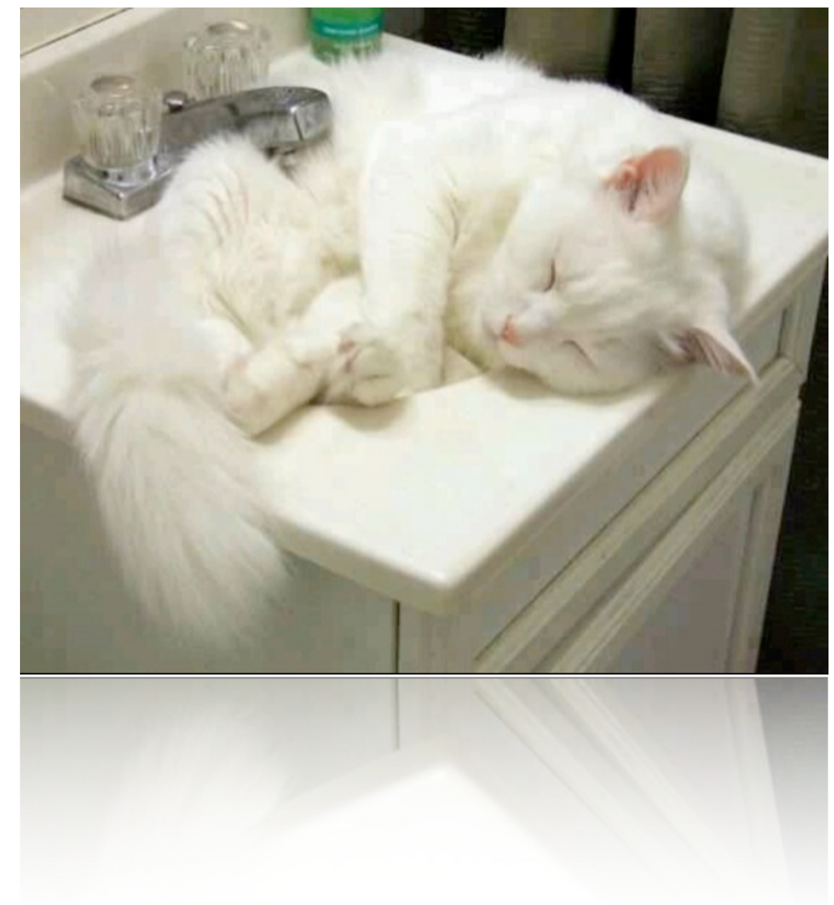
Sinks

- Data has to go somewhere
  - Writing out to a channel
  - Writing to disk



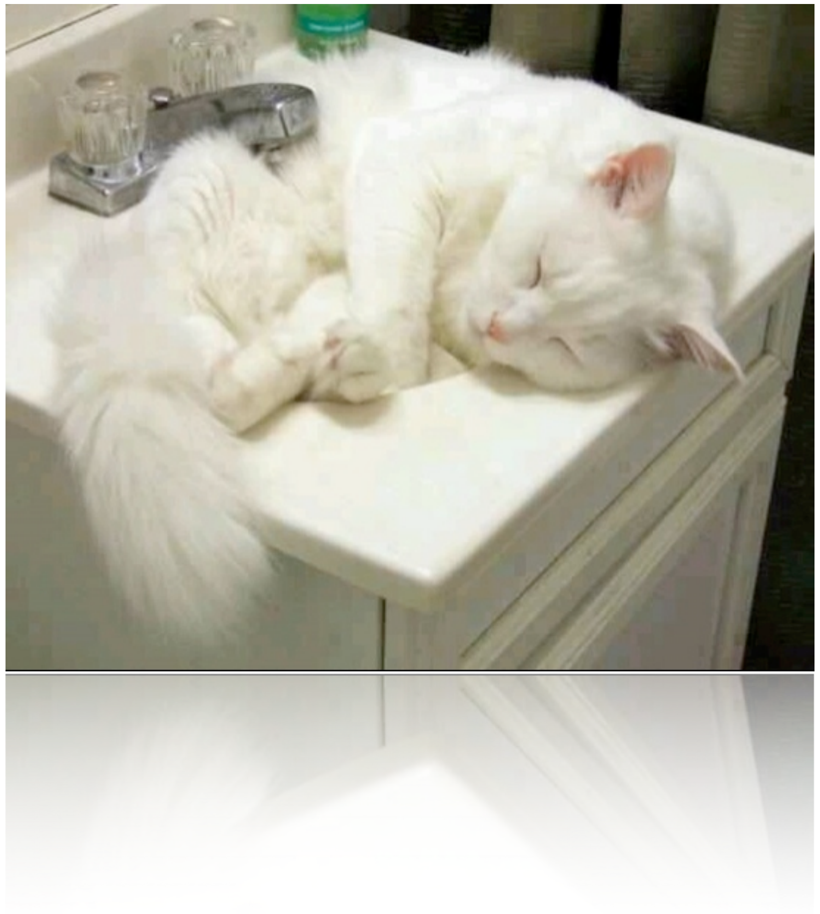
Sinks

- Data has to go somewhere
  - Writing out to a channel
  - Writing to disk
  - …or all of the above



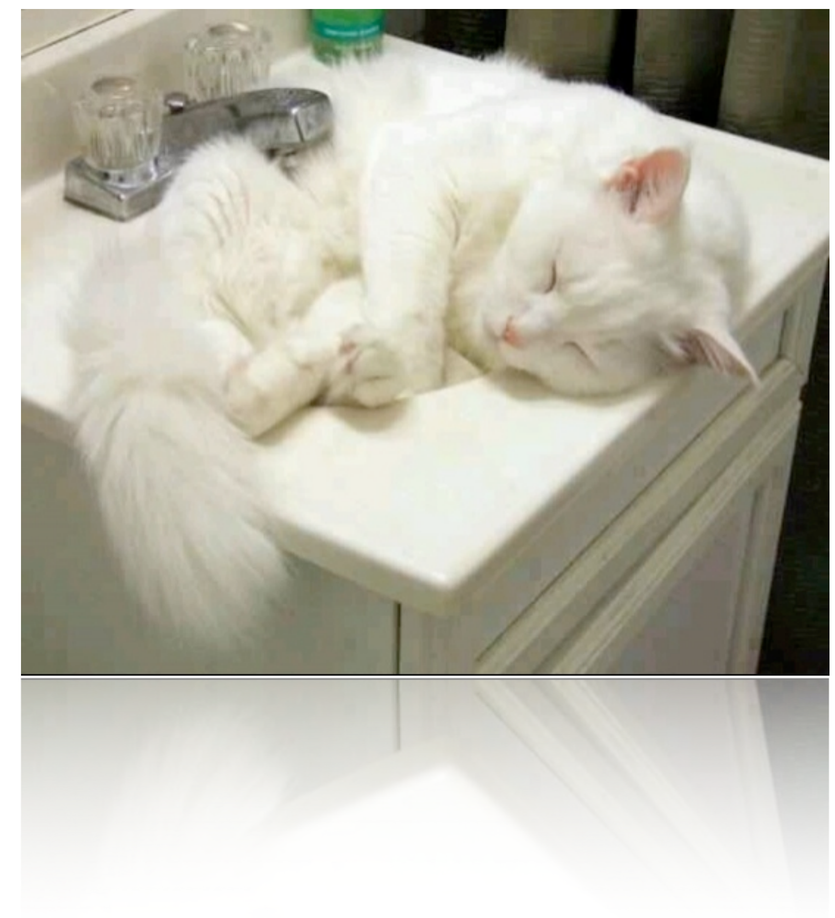
Sinks

- Data has to go somewhere
    - Writing out to a channel
    - Writing to disk
    - …or all of the above
- What is a sink anyway?



Sinks

- Data has to go somewhere
  - Writing out to a channel
  - Writing to disk
  - …or all of the above
- What is a sink anyway?
  - A stream of functions!



```scala
type Sink[F[_], A] = Process[F, A => F[Unit]]
```

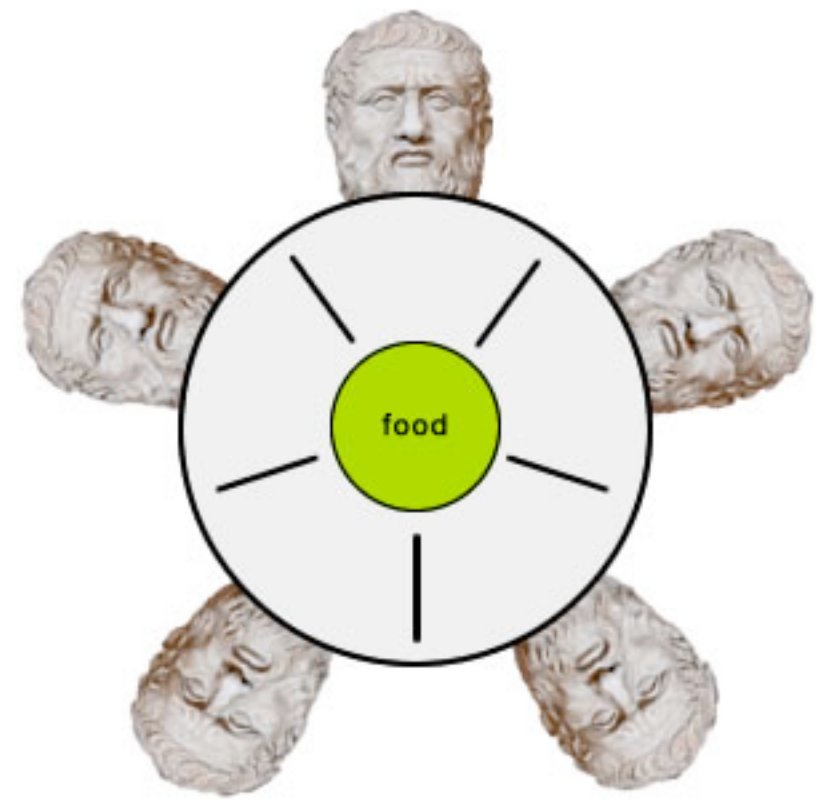
```scala
def write(str: String): Task[Unit] =
  Task delay { println(str) }

val sink: Sink[Task, String] = Process.constant(write _)
val src = Process.range(0, 10) map { _.toString }

val results = src zip sink flatMap {
  case (str, f) => Process eval f(str)
}

val universe: Task[Unit] = results.run
```

```scala
val stdOut: Sink[Task, String] = ...
val channel: Sink[Task, String] = ...

val src = Process.range(0, 10) map { _.toString }

val results = src zip stdOut zip channel flatMap {
  case ((str, f1), f2) => {
    for {
      _ <- Process eval f1(str)
      _ <- Process eval f2(str)
    } yield ()
  }
}

val universe: Task[Unit] = results.run
```

```scala
val stdOut: Sink[Task, String] = ...
val channel: Sink[Task, String] = ...

val src = Process.range(0, 10) map { _.toString }

val results = src observe stdOut to channel

val universe: Task[Unit] = results.run
```

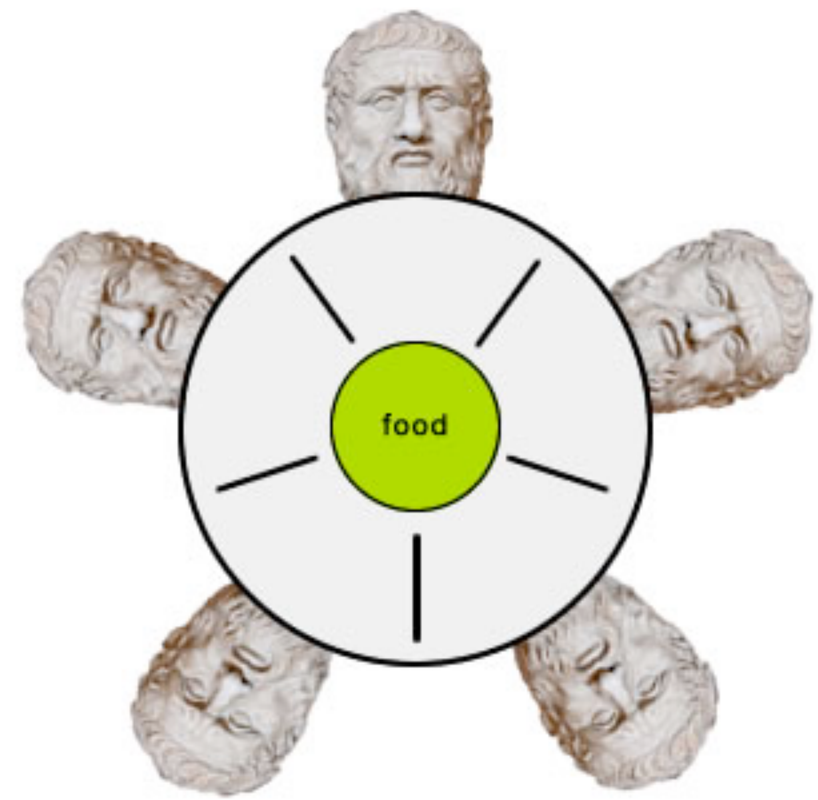
```scala
def debug[A](stream: Process[Task, A]): Process[Task, A] =
  stream map { a => s"debug: $a" } observe io.stdOutLines
```

Concurrency



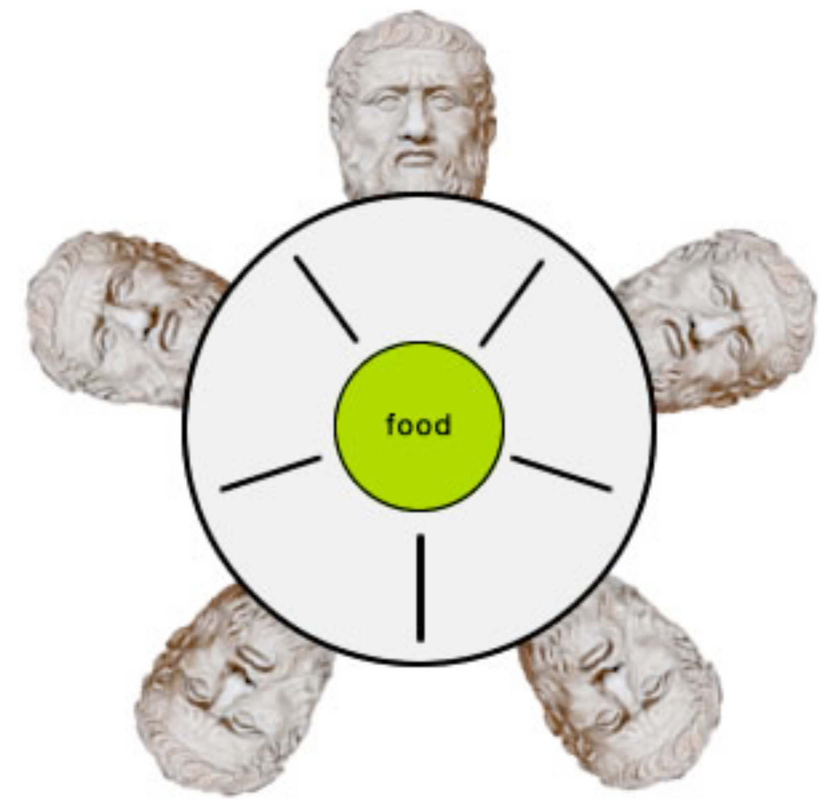
Concurrency

- Always explicit!



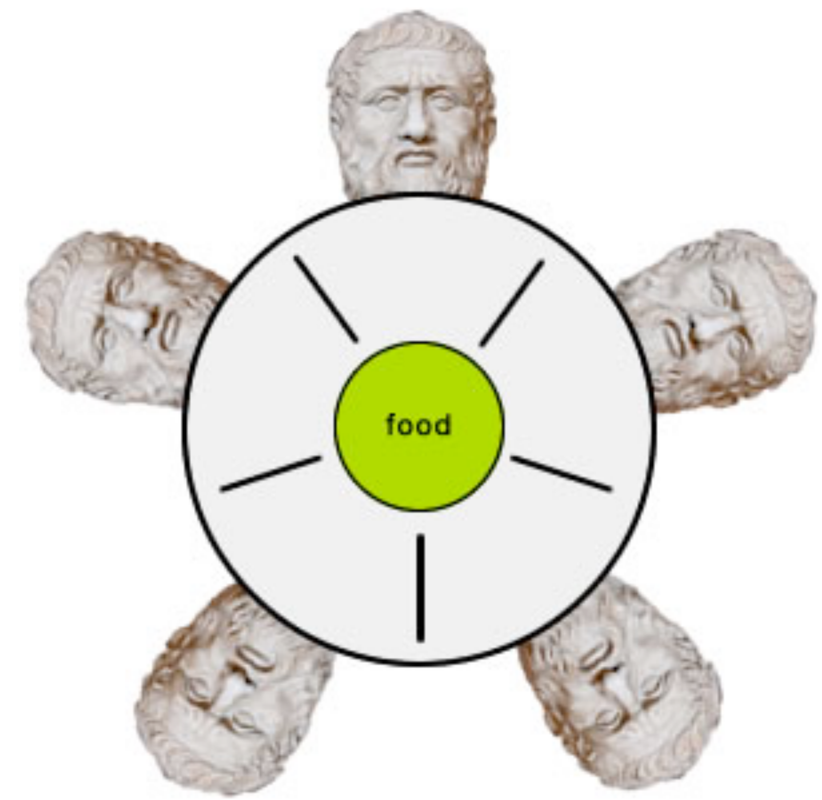
Concurrency

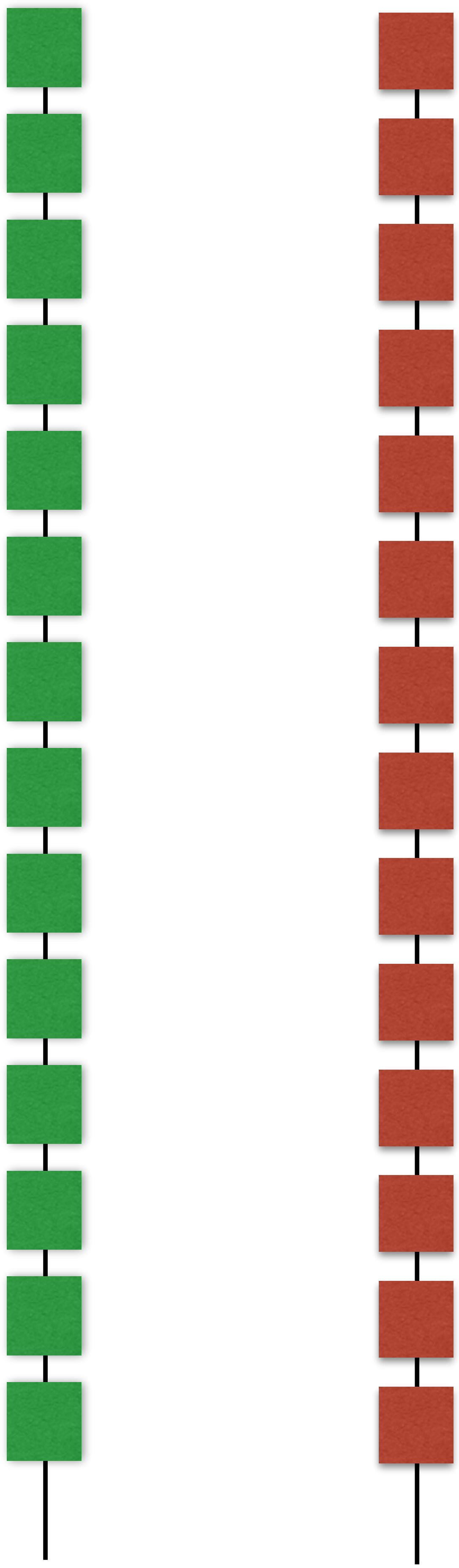
- Always explicit!
- Two forms of parallelism
  - Racing two streams into one
  - Turning a stream "sideways"

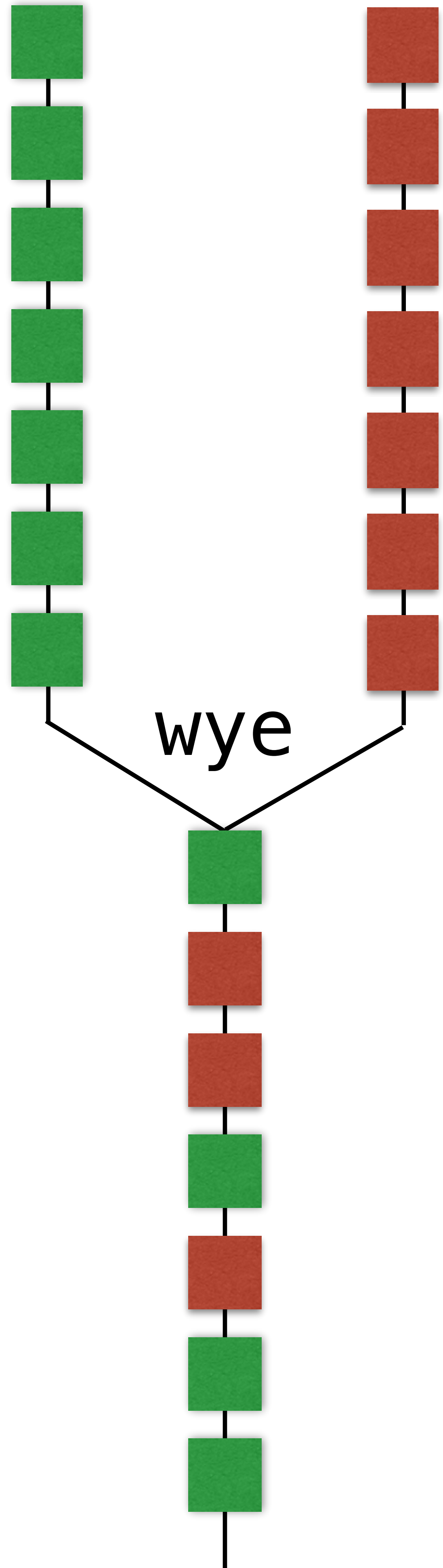


Concurrency

- Always explicit!
- Two forms of parallelism
  - Racing two streams into one
  - Turning a stream "sideways"
- Almost everything implemented on top of `wye`






wye

```scala
val left: Process[Task, Message] = ...
val right: Process[Task, Message] = ...

val merged: Process[Task, Message] =
  left.wye(right)(wye.merge)
```

```scala
val left: Process[Task, Message] = ...
val right: Process[Task, Message] = ...

val merged: Process[Task, Message] =
  left merge right    // should be "race"
```

```scala
val left: Process[Task, Message] = ...
val right: Process[Task, Line] = ...

// oh NOES! teh symbols cometh!
val merged: Process[Task, Message \/ Line] =
  left either right
```

Useful `wyes`

- `wye.merge`

Useful `wyes`

- `wye.merge`
- `wye.either`

Useful `wyes`

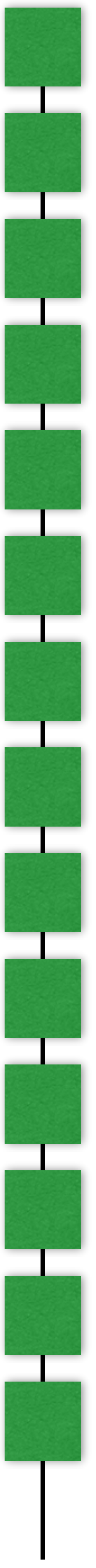
- `wye.merge`
- `wye.either`
- `wye.interrupt`

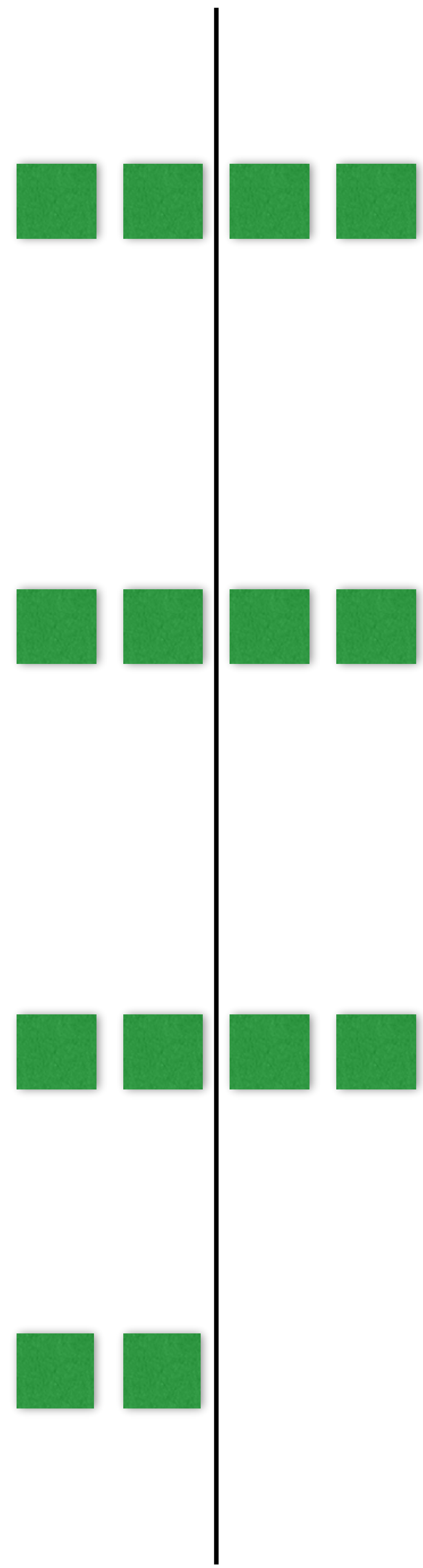
Useful `wyes`

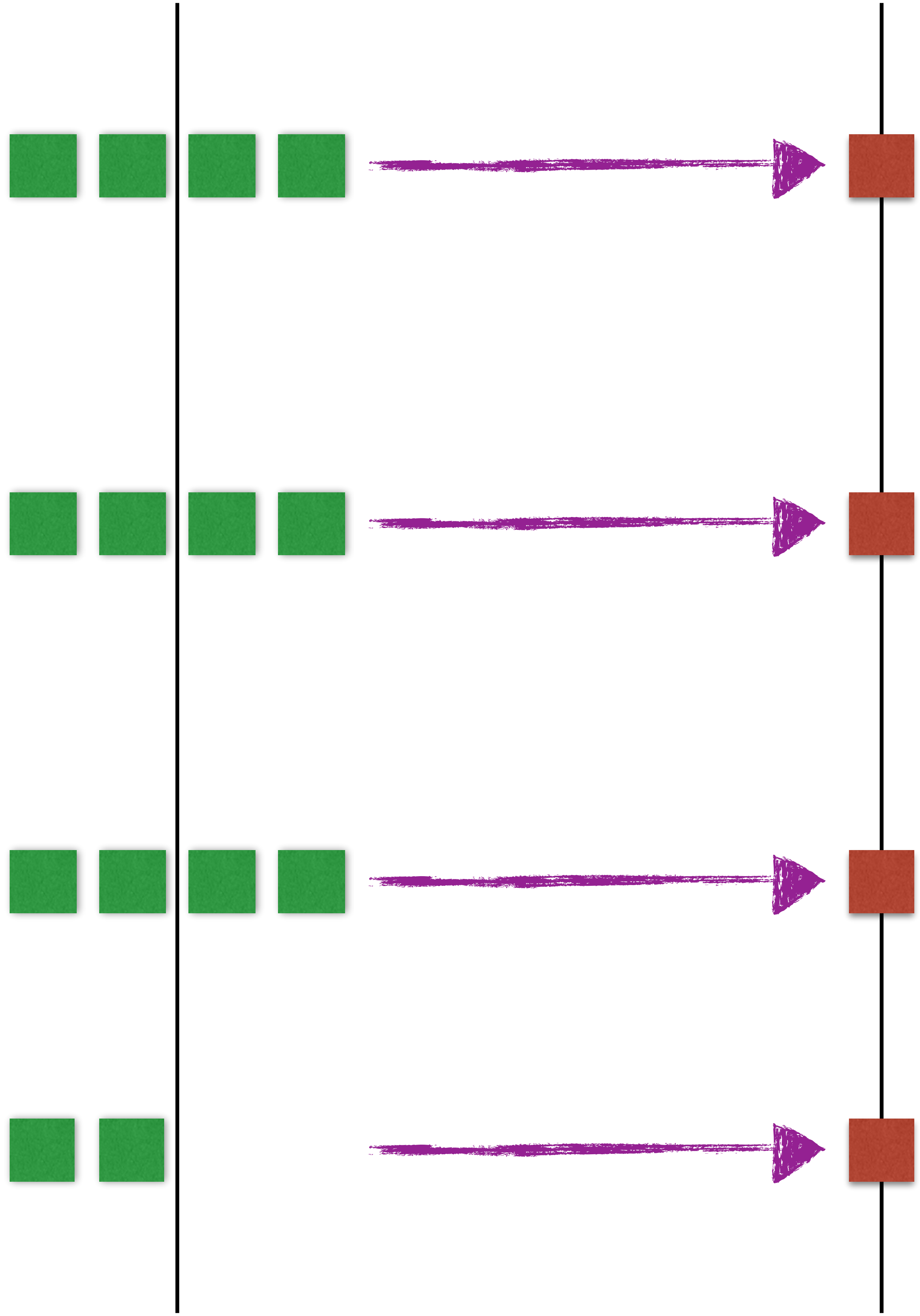
- `wye.merge`
- `wye.either`
- `wye.interrupt`
- `wye.drainL` / `wye.drainR`

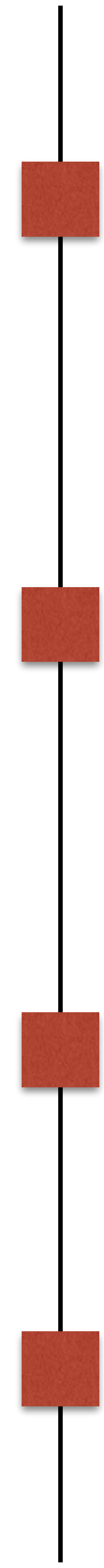
Useful `wyes`

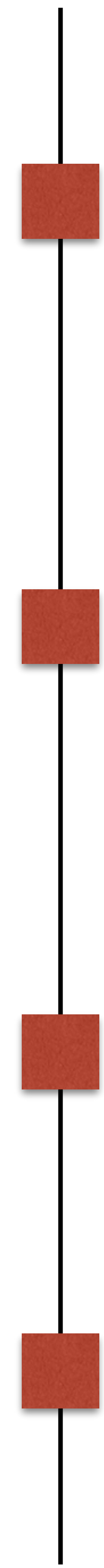
- `wye.merge`
- `wye.either`
- `wye.interrupt`
- `wye.drainL` / `wye.drainR`
    - Doesn't work!

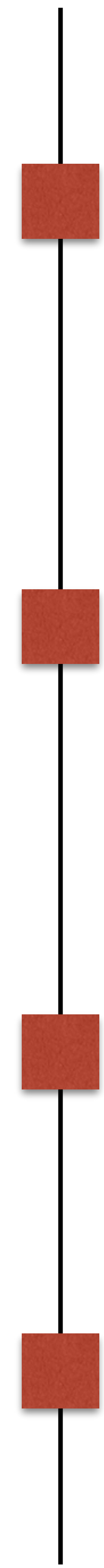


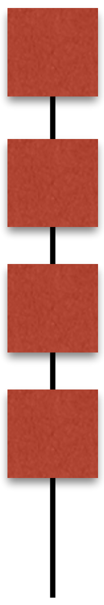












```scala
val nums: Process[Task, Int] = Process.range(0, 10)
val adjusted = nums map { _ * 2 } filter { _ < 10 }

val pages = adjusted flatMap { num =>
  Process.eval(fetchUrl(num))
}
```

```scala
val nums: Process[Task, Int] = Process.range(0, 10)
val adjusted = nums map { _ * 2 } filter { _ < 10 }

val pages: Process[Task, Task[String]] =
  adjusted map { num =>
    fetchUrl(num)
  }

val parallel: Process[Task, String] =
  pages.gather(4)
```

`gather(`n`)`

- Grabs chunks of n and parallelizes

`gather(`n`)`

- Grabs chunks of n and parallelizes
- Last chunk of stream may be truncated

`gather(`n`)`

- Grabs chunks of n and parallelizes
- Last chunk of stream may be truncated
- Great for finite streams!

`gather(`n`)`

- Grabs chunks of n and parallelizes
- Last chunk of stream may be truncated
- Great for finite streams!
- Causes **DEADLOCK** on infinite streams

`gather(`n`)`

- Grabs chunks of n and parallelizes
- Last chunk of stream may be truncated
- Great for finite streams!
- Causes **DEADLOCK** on infinite streams
  - Don't use if you source from a queue!

```scala
val nums: Process[Task, Int] = Process.range(0, 10)
val adjusted = nums map { _ * 2 } filter { _ < 10 }

val pages: Process[Task, Process[Task, String]] =
  adjusted map { num =>
    Process.eval(fetchUrl(num))
  }

val parallel: Process[Task, String] =
  merge.mergeN(pages)
```

`merge.mergeN`

- A little weirder to use…

merge.mergeN

- A little weirder to use…
  - `Process` of `Process`

`merge.mergeN`

- A little weirder to use…
  - `Process` of `Process`
- Uses a variable bounded queue

`merge.mergeN`

- A little weirder to use…
  - `Process` of `Process`
- Uses a variable bounded queue
- Races all input streams

`merge.mergeN`

- A little weirder to use…
  - `Process` of `Process`
- Uses a variable bounded queue
- Races all input streams
  - Up to n at a time

`merge.mergeN`

- A little weirder to use…
  - `Process` of `Process`
- Uses a variable bounded queue
- Races all input streams
  - Up to n at a time
- Almost always what you really want

Chat Server

- Uses scalaz-netty project

Chat Server

- Uses scalaz-netty project
    - Currently closed-source, but OSS soon™!

Chat Server

- Uses scalaz-netty project
  - Currently closed-source, but OSS soon™!
  - Would also work with scalaz-nio

Chat Server

- Uses scalaz-netty project
  - Currently closed-source, but OSS soon™!
  - Would also work with scalaz-nio
- Uses scodec

Chat Server

- Uses scalaz-netty project
  - Currently closed-source, but OSS soon™!
  - Would also work with scalaz-nio
- Uses scodec
  - Use this. Use it. It's amazing.

Chat Server

- Uses scalaz-netty project
  - Currently closed-source, but OSS soon™!
  - Would also work with scalaz-nio
- Uses scodec
  - Use this. Use it. It's amazing.
- Demonstrates the power of `Process` abstraction

Server

- Accept connections asynchronously
  - …and in parallel!

Server

- Accept connections asynchronously
  - …and in parallel!
- Pipe inbound data to a relay queue

Server

- Accept connections asynchronously
    - …and in parallel!
- Pipe inbound data to a relay queue
- Pipe relay queue into the outbound channel

Server

- Accept connections asynchronously
  - …and in parallel!
- Pipe inbound data to a relay queue
- Pipe relay queue into the outbound channel
- Continue until client closes connection

```scala
val address: InetSocketAddress = ???

val relay = async.topic[BitVector]

val handlers = Netty server address map { client =>
  for {
    Exchange(src, sink) <- client

    in = src to relay.publish
    out = relay.subscribe to sink

    _ <- in merge out
  } yield ()
}

val server: Task[Unit] = merge.mergeN(handlers).run
```

Client

- Establish connection

Client

- Establish connection
- Pipe standard input to the server (as UTF-8)

Client

- Establish connection
- Pipe standard input to the server (as UTF-8)
- Pipe server response to standard output

Client

- Establish connection
- Pipe standard input to the server (as UTF-8)
- Pipe server response to standard output
- Continue until user fail-sauce Ctrl-C kills us

```scala
implicit val codec: Codec[String] = utf8

def transcode(ex: Exchange[BitVector, BitVector]) = {
  val decoder = decode.many[String]
  val encoder = encode.many[String]

  val Exchange(src, sink) = ex

  val src2 = src flatMap decoder.decode
  val sink2 = sink pipeIn encoder.encoder

  Exchange(src2, sink2)
}
```

```scala
val clientP = for {
  rawData <- Netty connect address
  Exchange(src, sink) = transcode(rawData)

  in = src to io.stdOutLines
  out = io.stdInLines to sink

  _ <- in merge out
} yield ()

val client: Task[Unit] = clientP.run
```

Notes

- Resources are managed and *cannot* leak

Notes

- Resources are managed and *cannot* leak
- Logic is pure and encapsulated from networking

Notes

- Resources are managed and *cannot* leak
- Logic is pure and encapsulated from networking
- Backpressure "just works" (sort of)

Notes

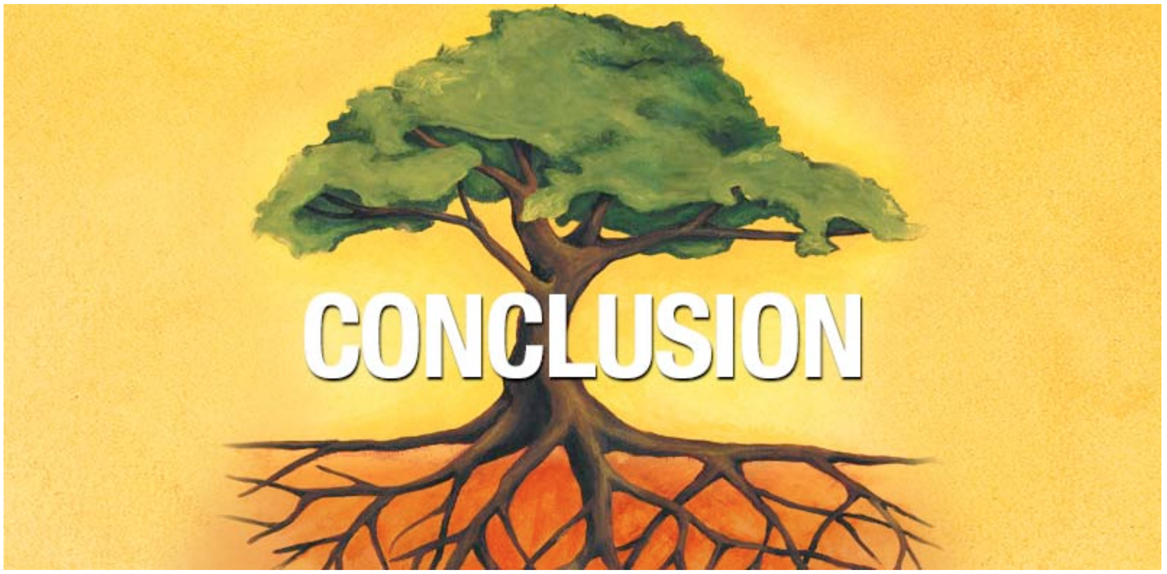
- Resources are managed and *cannot* leak
- Logic is pure and encapsulated from networking
- Backpressure "just works" (sort of)
    - Our `Topic` is unbounded, because I'm lazy

Notes

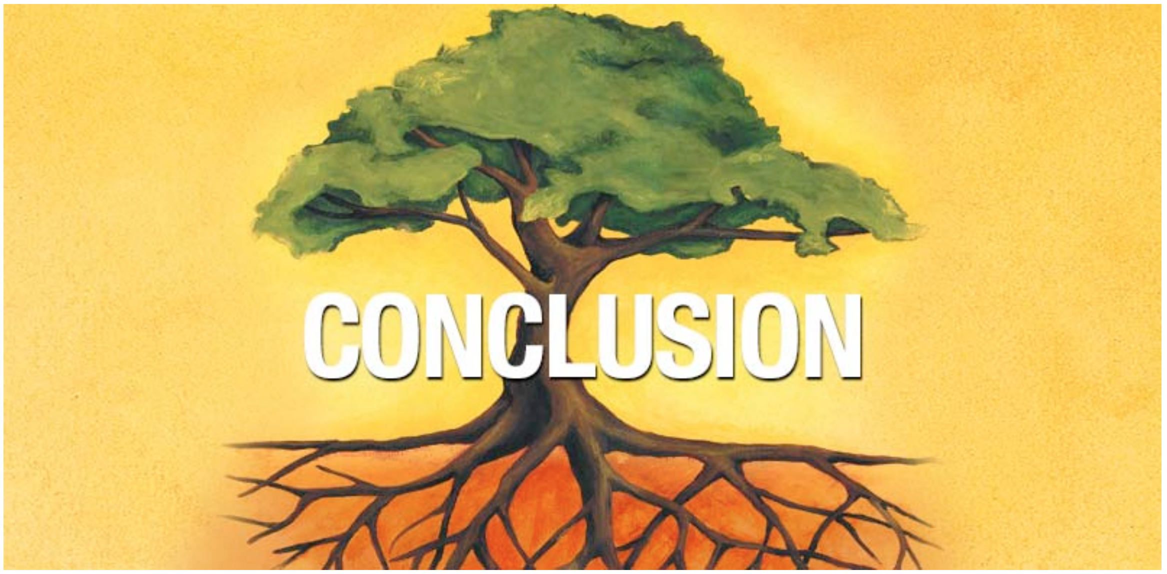
- Resources are managed and *cannot* leak
- Logic is pure and encapsulated from networking
- Backpressure "just works" (sort of)
  - Our `Topic` is unbounded, because I'm lazy
- Handshaking would be almost trivial

Notes

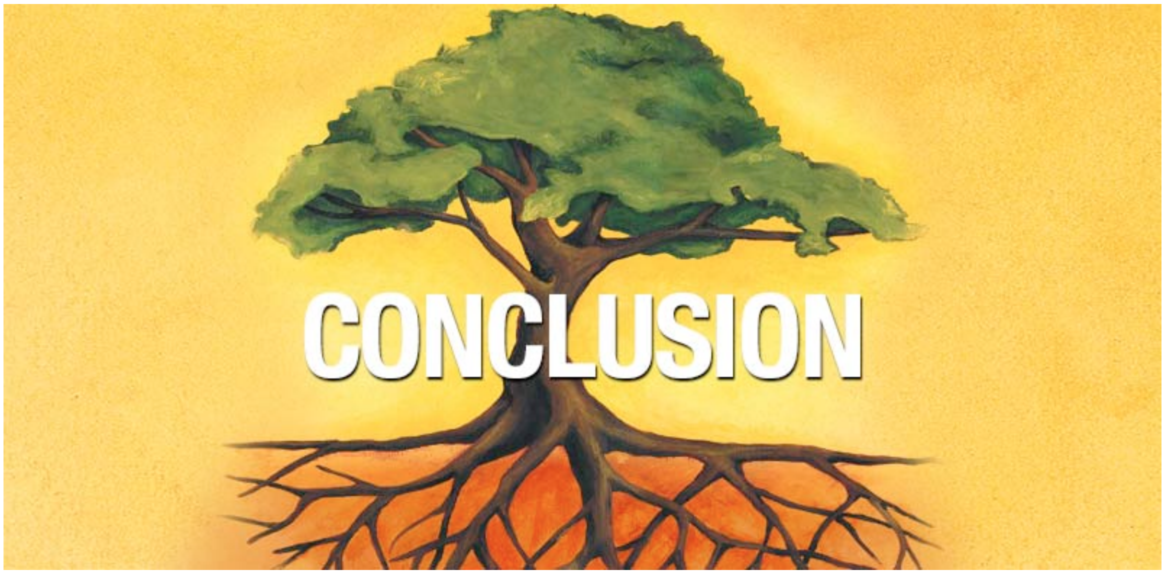
- Resources are managed and *cannot* leak
- Logic is pure and encapsulated from networking
- Backpressure "just works" (sort of)
    - Our `Topic` is unbounded, because I'm lazy
- Handshaking would be almost trivial
- Client and server logic looks *almost* the same!



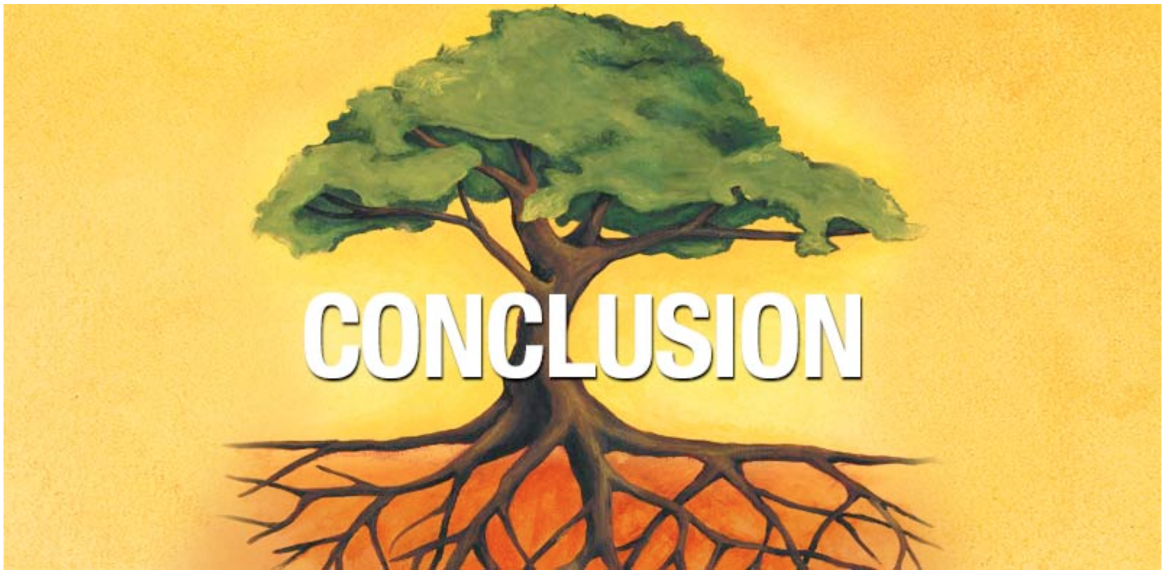
- A different take on "reactive"



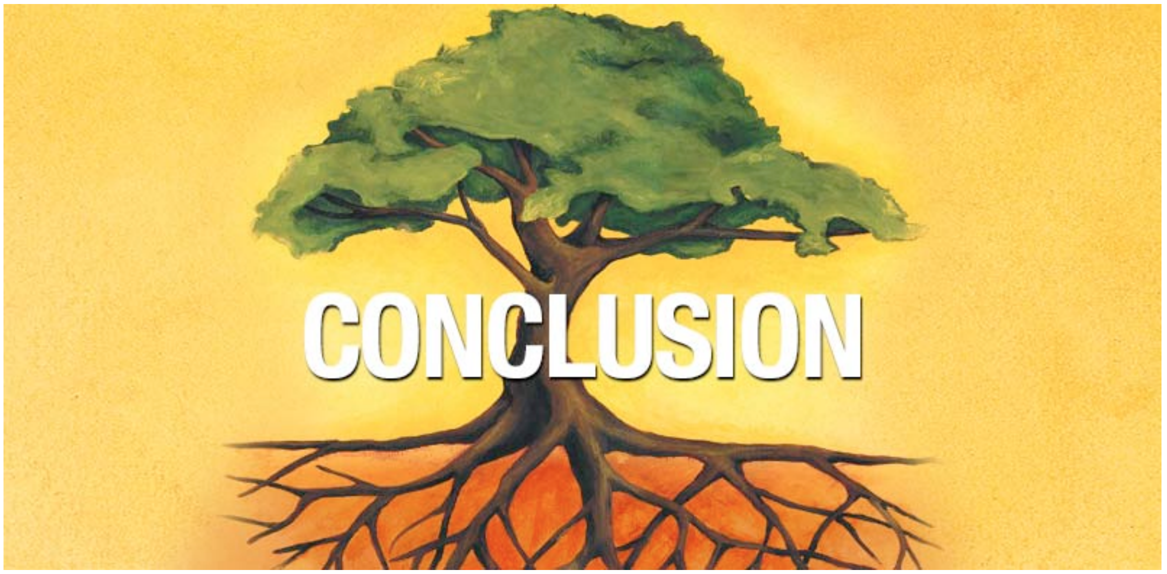
- A different take on "reactive"
- Purity helps us understand complex logic!



- A different take on "reactive"
- Purity helps us understand complex logic!
  - No more puzzling about state or resource leaks



- A different take on "reactive"
- Purity helps us understand complex logic!
    - No more puzzling about state or resource leaks
- Simple and easy combinators scale well



- A different take on "reactive"
- Purity helps us understand complex logic!
  - No more puzzling about state or resource leaks
- Simple and easy combinators scale well
- You know almost everything you need


Questions?