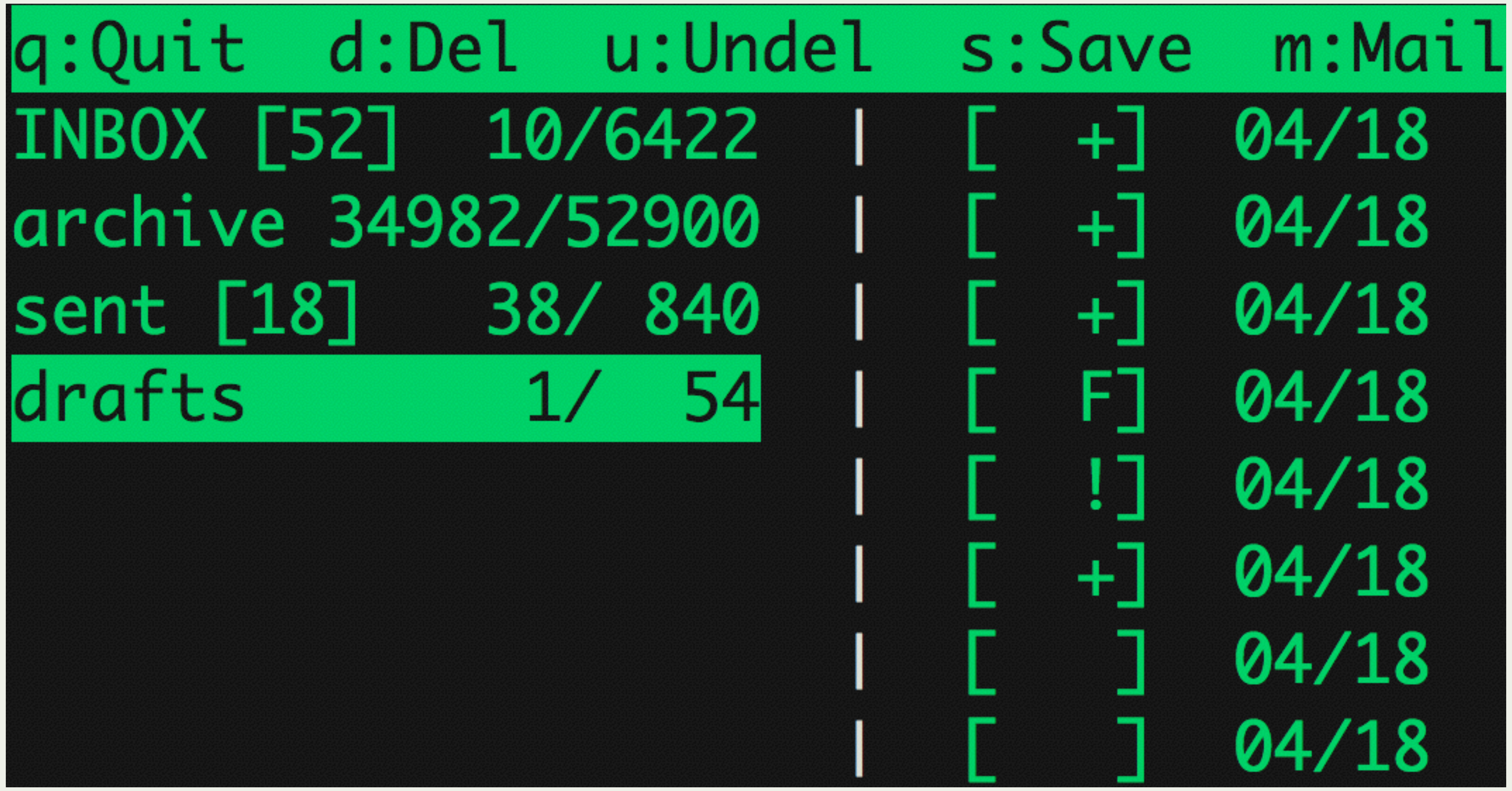
The Art of Incremental Stream Processing

@markhibberd

I have a problem



```
q:Quit  d:Del  u:Undel  s:Save  m:Mail
INBOX [52]  10/6422  |  [  +]  04/18
archive 34982/52900  |  [  +]  04/18
sent [18]    38/ 840  |  [  +]  04/18
drafts         1/  54  |  [  F]  04/18
                       |  [  !]  04/18
                       |  [  +]  04/18
                       |  [   ]  04/18
                       |  [   ]  04/18
```

~150k Emails

~4GB Emails

100s of new email / day

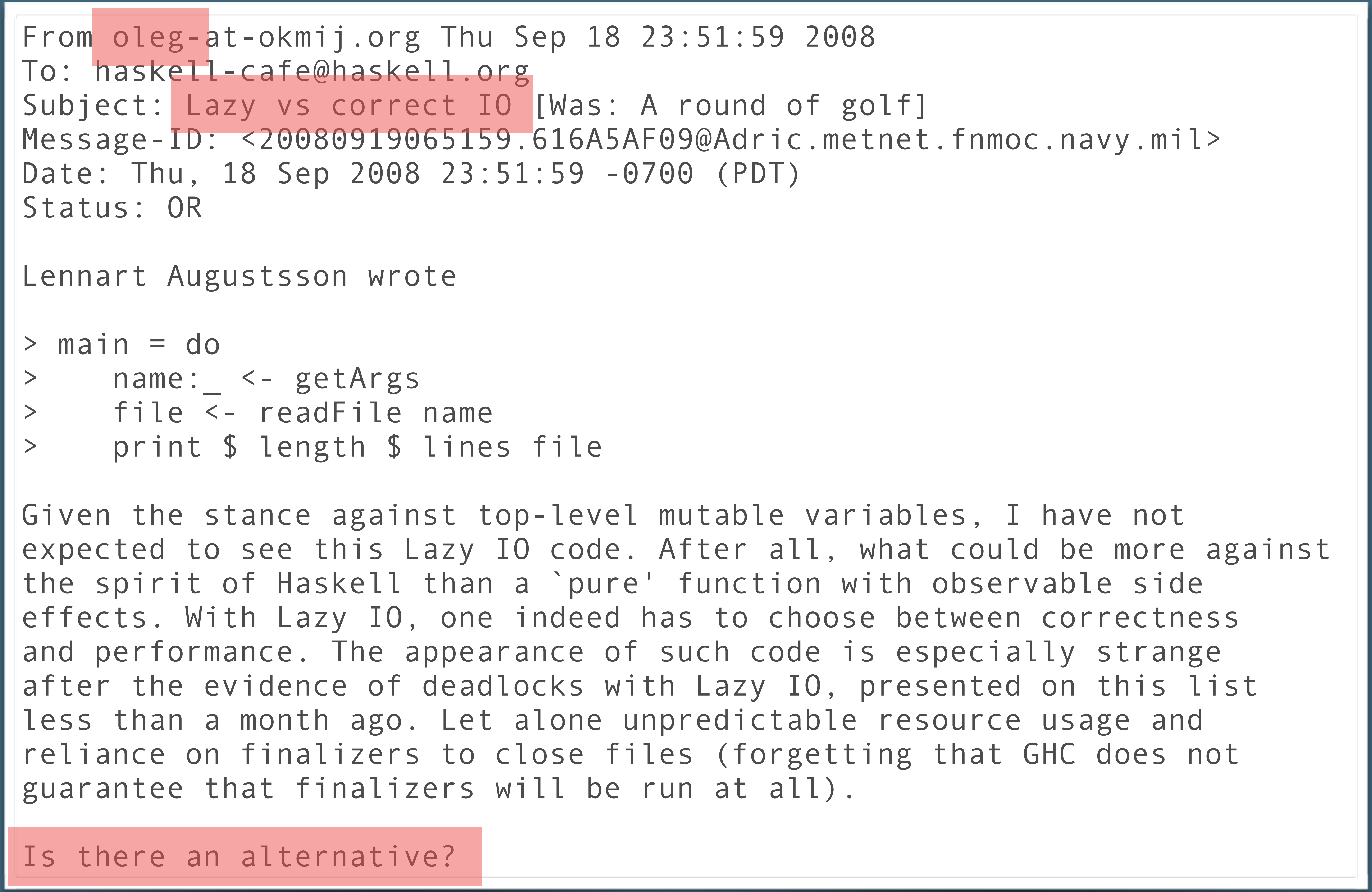
~5 I want to Read

~2 I want to Reply To

Messages delivered where and when I need them

Ability to locate important messages from days past

```
From oleg-at-okmij.org Thu Sep 18 23:51:59 2008
To: haskell-cafe@haskell.org
Subject: Lazy vs correct IO [Was: A round of golf]
Message-ID: <20080919065159.616A5AF09@Adric.metnet.fnmoc.navy.mil>
Date: Thu, 18 Sep 2008 23:51:59 -0700 (PDT)
Status: OR

Lennart Augustsson wrote

> main = do
>    name:_ <- getArgs
>    file <- readFile name
>    print $ length $ lines file

Given the stance against top-level mutable variables, I have not
expected to see this Lazy IO code. After all, what could be more against
the spirit of Haskell than a `pure' function with observable side
effects. With Lazy IO, one indeed has to choose between correctness
and performance. The appearance of such code is especially strange
after the evidence of deadlocks with Lazy IO, presented on this list
less than a month ago. Let alone unpredictable resource usage and
reliance on finalizers to close files (forgetting that GHC does not
guarantee that finalizers will be run at all).

Is there an alternative?
```



```
From oleg-at-okmij.org Thu Sep 18 23:51:59 2008
To: haskell-cafe@haskell.org
Subject: Lazy vs correct IO [Was: A round of golf]
Message-ID: <20080919065159.616A5AF09@Adric.metnet.fnmoc.navy.mil>
Date: Thu, 18 Sep 2008 23:51:59 -0700 (PDT)
Status: OR

Lennart Augustsson wrote

> main = do
>    name:_ <- getArgs
>    file <- readFile name
>    print $ length $ lines file

Given the stance against top-level mutable variables, I have not
expected to see this Lazy IO code. After all, what could be more against
the spirit of Haskell than a `pure' function with observable side
effects. With Lazy IO, one indeed has to choose between correctness
and performance. The appearance of such code is especially strange
after the evidence of deadlocks with Lazy IO, presented on this list
less than a month ago. Let alone unpredictable resource usage and
reliance on finalizers to close files (forgetting that GHC does not
guarantee that finalizers will be run at all).

Is there an alternative?
```

Zeitgeist

```
find "$MAILDIR" -type f       | \
  xargs -n 1 stat -f "%m|%N" | \
  sort -n                     | \
  cut -d '|' -f 2             | \
  xargs grep -l "$QUERY"      | \
  head -5                     | \
  xargs less
```

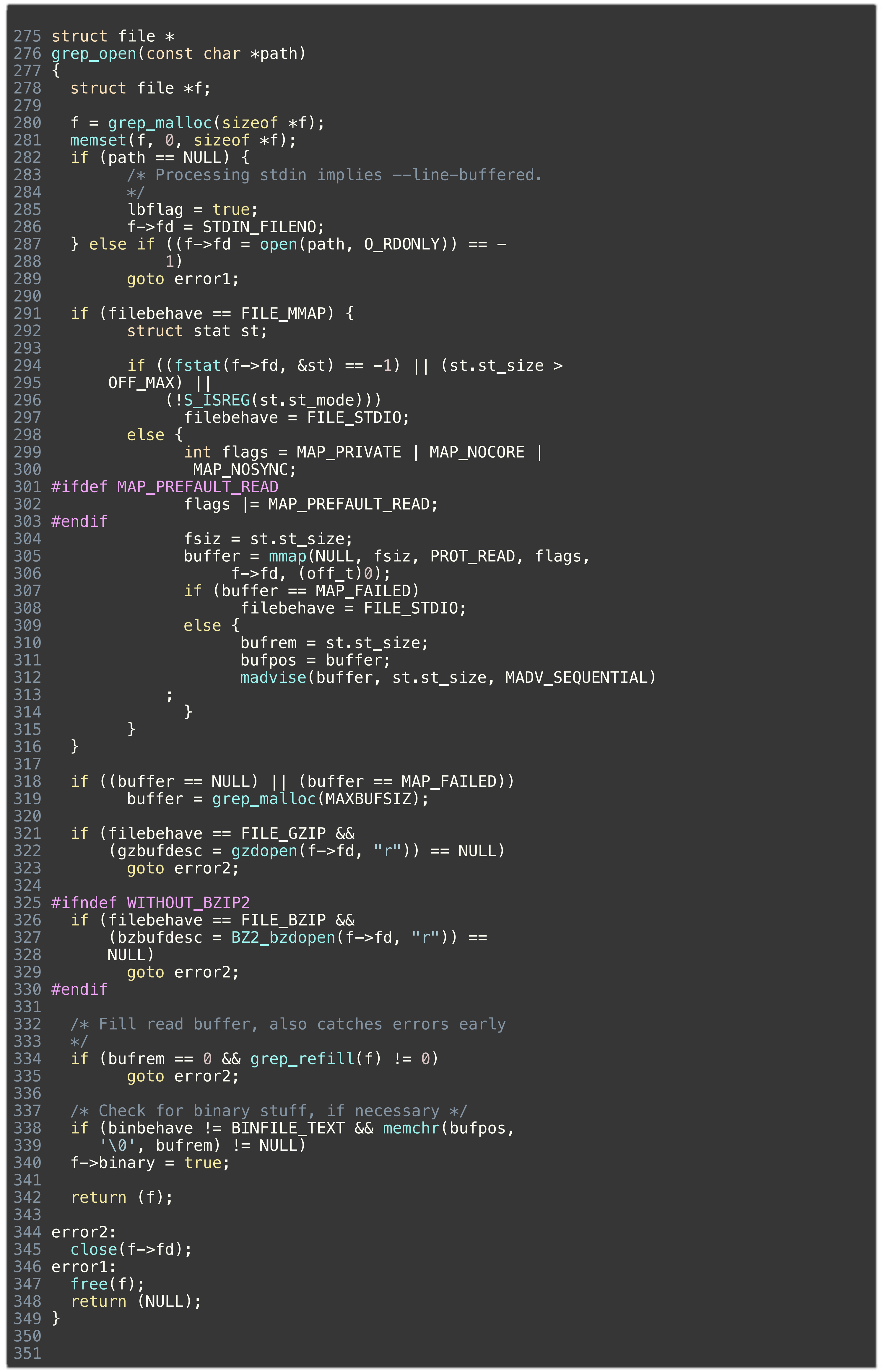
```haskell
data Email =
  Email { date :: Int,  content :: String }
  deriving (Show, Eq)

search :: String -> [Email] -> [Email]
search term =
  take 5
  . filter (isInfixOf term . content)
  . sortBy (compare `on` date)
```

Reality Calling

“--mmap

Use mmap(2) instead of read(2) to read input, which can result in better performance under some circumstances but can cause undefined behaviour. ”

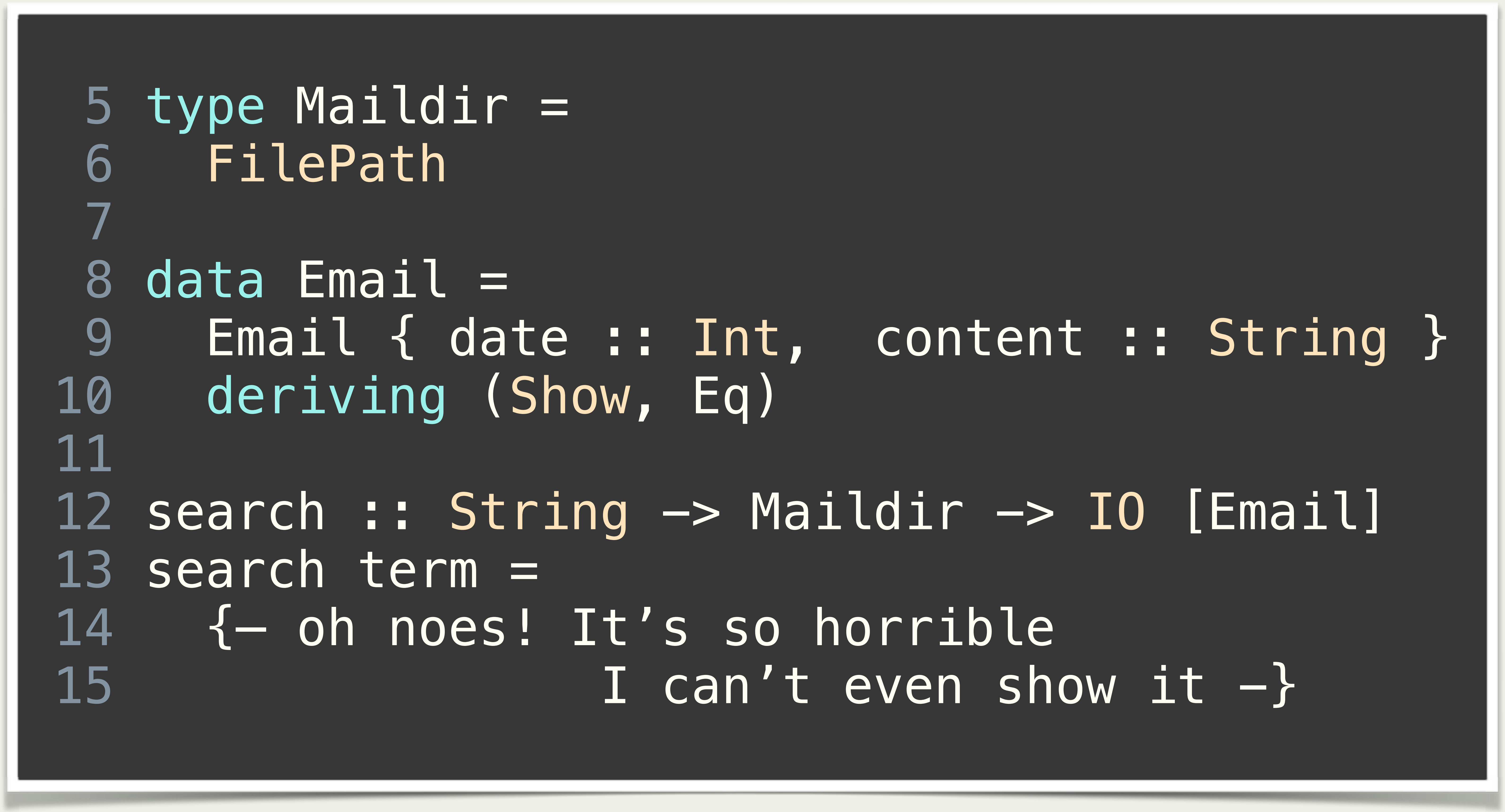
— $(man grep)



```c
struct file *
grep_open(const char *path)
{
  struct file *f;

  f = grep_malloc(sizeof *f);
  memset(f, 0, sizeof *f);
  if (path == NULL) {
        /* Processing stdin implies --line-buffered.
        */
        lbflag = true;
        f->fd = STDIN_FILENO;
  } else if ((f->fd = open(path, O_RDONLY)) == -
            1)
        goto error1;

  if (filebehave == FILE_MMAP) {
        struct stat st;

        if ((fstat(f->fd, &st) == -1) || (st.st_size >
      OFF_MAX) ||
            (!S_ISREG(st.st_mode)))
              filebehave = FILE_STDIO;
        else {
              int flags = MAP_PRIVATE | MAP_NOCORE |
               MAP_NOSYNC;
#ifdef MAP_PREFAULT_READ
              flags |= MAP_PREFAULT_READ;
#endif
              fsiz = st.st_size;
              buffer = mmap(NULL, fsiz, PROT_READ, flags,
                   f->fd, (off_t)0);
              if (buffer == MAP_FAILED)
                    filebehave = FILE_STDIO;
              else {
                    bufrem = st.st_size;
                    bufpos = buffer;
                    madvise(buffer, st.st_size, MADV_SEQUENTIAL)
            ;
              }
        }
  }

  if ((buffer == NULL) || (buffer == MAP_FAILED))
        buffer = grep_malloc(MAXBUFSIZ);

  if (filebehave == FILE_GZIP &&
      (gzbufdesc = gzdopen(f->fd, "r")) == NULL)
        goto error2;

#ifndef WITHOUT_BZIP2
  if (filebehave == FILE_BZIP &&
      (bzbufdesc = BZ2_bzdopen(f->fd, "r")) ==
      NULL)
        goto error2;
#endif

  /* Fill read buffer, also catches errors early
  */
  if (bufrem == 0 && grep_refill(f) != 0)
        goto error2;

  /* Check for binary stuff, if necessary */
  if (binbehave != BINFILE_TEXT && memchr(bufpos,
     '\0', bufrem) != NULL)
  f->binary = true;

  return (f);

error2:
  close(f->fd);
error1:
  free(f);
  return (NULL);
}
```

“With Lazy IO, one indeed has to choose between correctness and performance.”

— Oleg Kiselyov



```haskell
type Maildir =
  FilePath

data Email =
  Email { date :: Int,  content :: String }
  deriving (Show, Eq)

search :: String -> Maildir -> IO [Email]
search term =
  {- oh noes! It’s so horrible
                 I can’t even show it -}
```

Is there an alternative?

Intuition 1: A Language

I Need To Produce Values

```
type In i
```

I Need To Consume Values

```
type In i

type Out o
```

I Need To Transform Values

```haskell
type In i

type Out o

data Pipeline i o
```

I May Have Effects

```haskell
type In i m

type Out o m

data Pipeline i o m
```

I May **Compute** A Value

```
type In i m a

type Out o m a

data Pipeline i o m a
```

A (Simple) Interface

```haskell
type In i m a = Pipeline i () m a

type Out o m a = Pipeline Void o m a

data Pipeline i o m a
```

```haskell
type In i m a = Pipeline i () m a

type Out o m a = Pipeline Void o m a

data Pipeline i o m a
  = Done a
  | Yield o (Pipeline i o a)
  | Await (i -> Pipeline i o a)
```

Intuition 2: Pipelines

Poll
Prepare
Classify
Deliver

Poll
Prepare
Classify
Deliver

Mail
Poll
Prepare
Classify
Deliver

Mail
Poll
Prepare
Classify
Deliver

Mail
Poll
Prepare
Classify
Deliver
Features

Poll
Prepare
Classify
Deliver
Features

Scores
Poll
Prepare
Classify
Deliver
Features

Scores
Poll
Prepare
Classify
Deliver

:: In Mail
Poll
:: Pipeline Mail Features
Prepare
:: Pipeline Features Scores
Classify
:: Out Scores
Deliver

Getting Real

```haskell
data Proxy i' i o' o m a

type Producer i m a = Proxy X () () o m a
type Consumer o m a = Proxy () i () X m a
type Pipe i o m a   = Proxy () i () o m a
```

```haskell
type In i m a

type Out o m a

type Pipeline i o m a
```

Pipes

```haskell
data Proxy i' i o' o m a

type Producer i m a = Proxy X () () o m a
type Consumer o m a = Proxy () i () X m a
type Pipe i o m a   = Proxy () i () o m a
```

Explicit Input and Output

at Each Component

Pipes

```haskell
data Proxy i' i o' o m a

type Producer i m a = Proxy X () () o m a
type Consumer o m a = Proxy () i () X m a
type Pipe i o m a   = Proxy () i () o m a
```

Effects On Producers,

Consumers And Pipes

Pipes

```haskell
data Proxy i' i o' o m a

type Producer i m a = Proxy X () () o m a
type Consumer o m a = Proxy () i () X m a
type Pipe i o m a   = Proxy () i () o m a
```

Can Terminate With A

Value Anywhere In Pipeline

Pipes

```haskell
data Pipe l i o u m r

newtype ConduitM i o m r =
  ConduitM { unConduitM :: Pipe i i o () m r }

type Source m o     = ConduitM () o    m ()
type Sink i m a     = ConduitM i  Void m a
type Conduit i m o = ConduitM i  o    m ()
```

```haskell
type In i m a

type Out o m a

type Pipeline i o m a
```

Conduit

```haskell
data Pipe l i o u m r

newtype ConduitM i o m r =
  ConduitM { unConduitM :: Pipe i i o () m r }

type Source m o     = ConduitM () o    m ()
type Sink i m a     = ConduitM i  Void m a
type Conduit i m o = ConduitM i  o    m ()
```

Explicit Input and Output

at Each Component

Conduit

```haskell
data Pipe l i o u m r

newtype ConduitM i o m r =
  ConduitM { unConduitM :: Pipe i i o () m r }

type Source m o     = ConduitM () o    m ()
type Sink i m a     = ConduitM i  Void m a
type Conduit i m o = ConduitM i  o    m ()
```

Effects On Sources,

Sinks And Conduits

Conduit

```haskell
data Pipe l i o u m r

newtype ConduitM i o m r =
  ConduitM { unConduitM :: Pipe i i o () m r }

type Source m o     = ConduitM () o    m ()
type Sink i m a     = ConduitM i  Void m a
type Conduit i m o = ConduitM i  o    m ()
```

Can Only Terminate With A

Value On a Sink

Conduit

```scala
sealed abstract class Process[F[_],O]

type Process0[O]          = Process[Env[_,_]#Is, O]
type Process1[I, O]       = Process[Env[I,_]#Is, O]
type Sink[F[_], O]        = Process[F, O => F[Unit]]
```

```haskell
type In i m a

type Out o m a

type Pipeline i o m a
```

Scalaz Streams

```haskell
data Process m o

type Process0 o     = forall a. Process (Is a) o
type Process1 i o   = Process (Is i) o
type Sink m o       = Process m (o -> m ())
```

```haskell
type In i m a

type Out o m a

type Pipeline i o m a
```

Scalaz Streams

```haskell
data Process m o

type Process0 o     = forall a. Process (Is a) o
type Process1 i o   = Process (Is i) o
type Sink m o       = Process m (o -> m ())
```

Model Request And Production

Rather Than Input and Output

Scalaz Streams

```haskell
data Process m o

type Process0 o    = forall a. Process (Is a) o
type Process1 i o  = Process (Is i) o
type Sink m o      = Process m (o -> m ())
```

Effects Are Returned As

Values, Transducers are Pure

Scalaz Streams

```haskell
data Process m o

type Process0 o     = forall a. Process (Is a) o
type Process1 i o   = Process (Is i) o
type Sink m o       = Process m (o -> m ())

runFoldMap :: (Monad m, Monoid b) =>
                Process m o -> (o -> m b) -> m b
```

Computation of Values

Modelled Externally

Scalaz Streams

:: Producer Mail
Poll
:: Pipe Mail Features
Prepare
:: Pipe Features Score
Classify
:: Consumer Score
Deliver

Pipes

:: Source Mail
:: Conduit Mail Features
:: Conduit Features Score
:: Sink Score
Poll
Prepare
Classify
Deliver

Conduit

:: Process m Mail
Poll
:: Process1 Mail Features
Prepare
:: Process1 Features Score
Classify
:: Sink m Score
Deliver

Scalaz Stream

Horizontal Composition

Mail Delivery
Poll
Prepare
Classify
Deliver

Mail Delivery
Poll
Prepare
Classify
Deliver

Poll
Scan
Throttle
Read

:: In Event
Poll
:: Pipeline Mail Features
Prepare
:: Pipeline Features Score
Classify
:: Out Event
Deliver

Poll
Prepare
Classify
Deliver

```
:: In Event
:: Pipeline Mail Features
:: Pipeline Features Score
:: Out Event
```

```
(>|) :: Pipeline i o m a -> Pipeline o o’ m a -> Pipeline i o’ m a
```

:: Pipeline () Void
:: In Event
:: Pipeline Mail Features
:: Pipeline Features Score
:: Out Event
Poll
>|
Prepare
>|
Classify
>|
Deliver

```
(>|) :: Pipeline i o m a -> Pipeline o o' m a -> Pipeline i o' m a
```

```
eval :: Pipeline () Void m a -> m a
```

:: In Event
:: Pipeline Mail Features
:: Pipeline Features Score
:: Out Event
Poll
>|
Prepare
>|
Classify
>|
Deliver

```
(>|) :: Pipeline i o m a -> Pipeline o o’ m a -> Pipeline i o’ m a
```

:: Effect
:: Producer Event
:: Pipe Mail Features
:: Pipe Features Score
:: Consumer Score
Poll
>->
Prepare
>->
Classify
>->
Deliver

Pipes

```
runEffect :: Effect m a -> m a
```

```
:: Producer Event
:: Pipe Mail Features
:: Pipe Features Score
:: Consumer Score
```

```
>->
```

Poll
>->
Prepare
>->
Classify
>->
Deliver

Pipes’

:: Source ()
:: Source Event
:: Conduit Mail Features
:: Conduit Features Score
:: Sink Score
Poll
=$=
Prepare
=$=
Classify
=$=
Deliver

Conduit

:: Source ()
:: Source Event
:: Conduit Mail Features
:: Conduit Features Score
:: Sink Score
Poll
$=
Prepare
=$=
Classify
=$
Deliver

Conduit’

:: m ()
:: Source Event
:: Conduit Mail Features
:: Conduit Features Score
:: Sink Score
Poll
$=
Prepare
=$=
Classify
$$
Deliver

Conduit’’

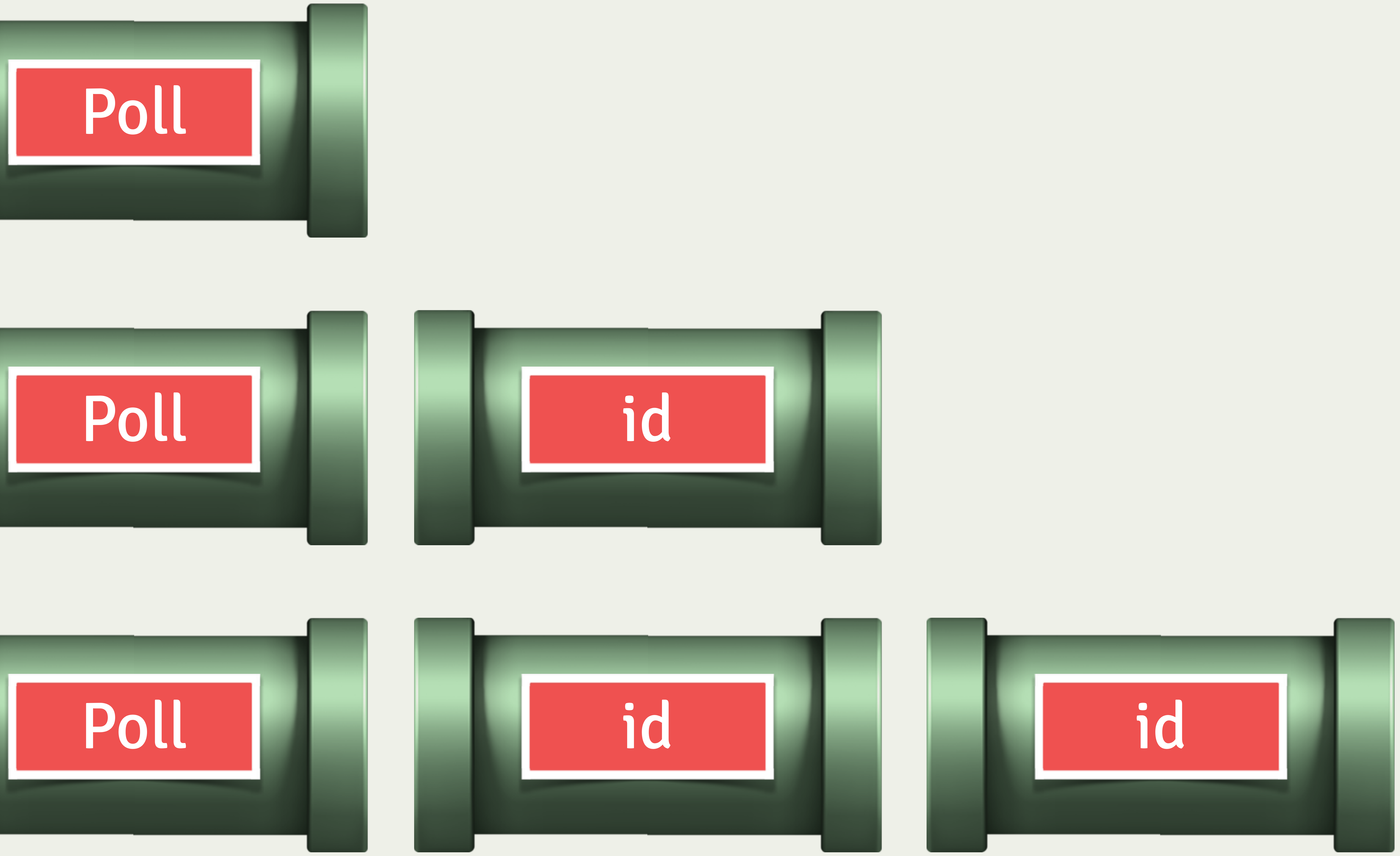
:: Process m ()
: Process m Event
:: Process1 Mail Features
:: Process1 Features Score
:: Sink m Score
Poll
|>
Prepare
|>
Classify
to
Deliver

Scalaz Stream

run :: Process m a -> m ()
: Process m Event
:: Process1 Mail Features
:: Process1 Features Score
:: Sink m Score
Poll
|>
Prepare
|>
Classify
to
Deliver

Scalaz Stream’

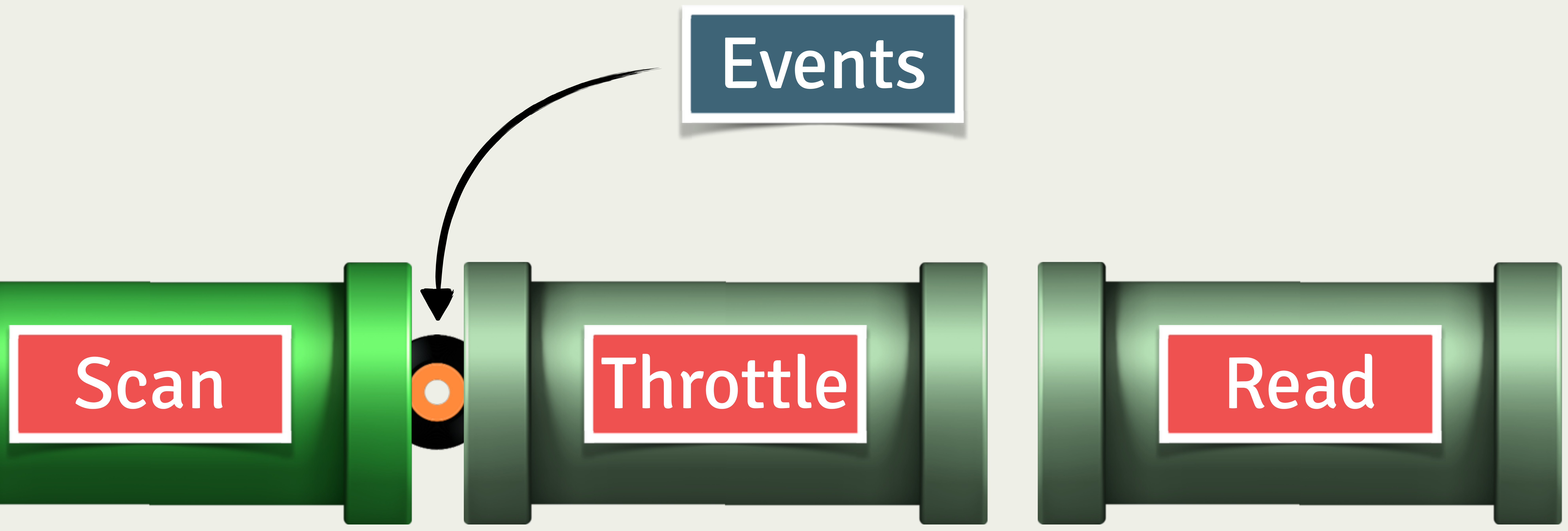
Is Composition About Combinators or Laws?

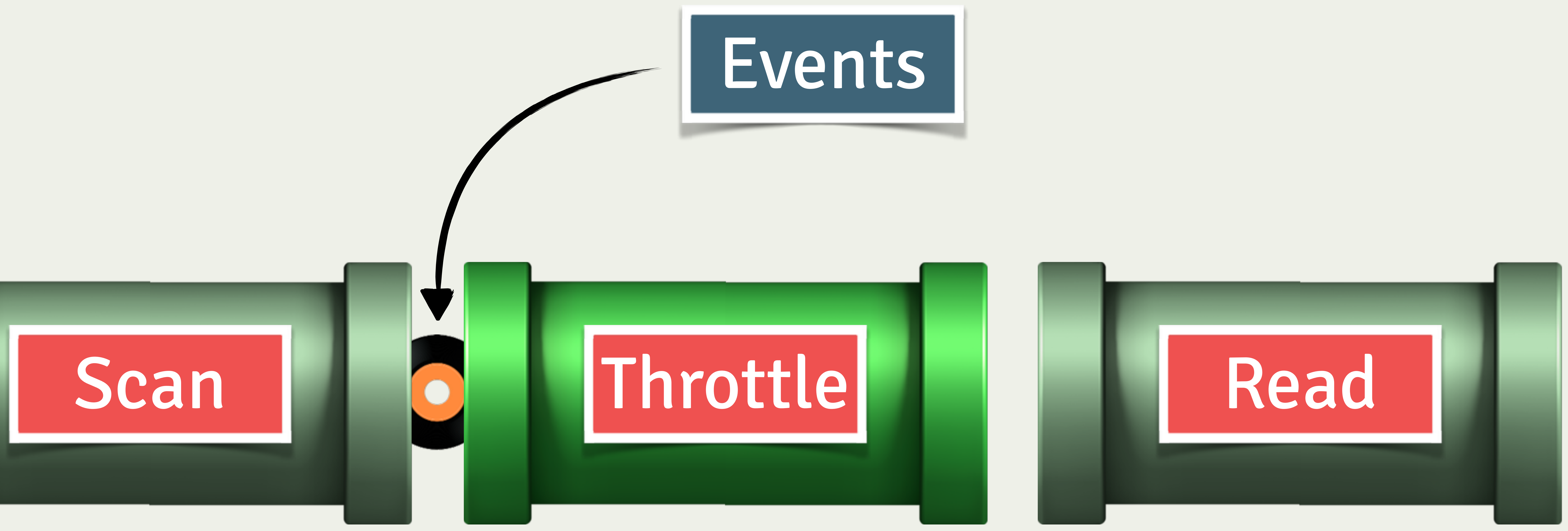

Poll
Poll
id
Poll
id
id

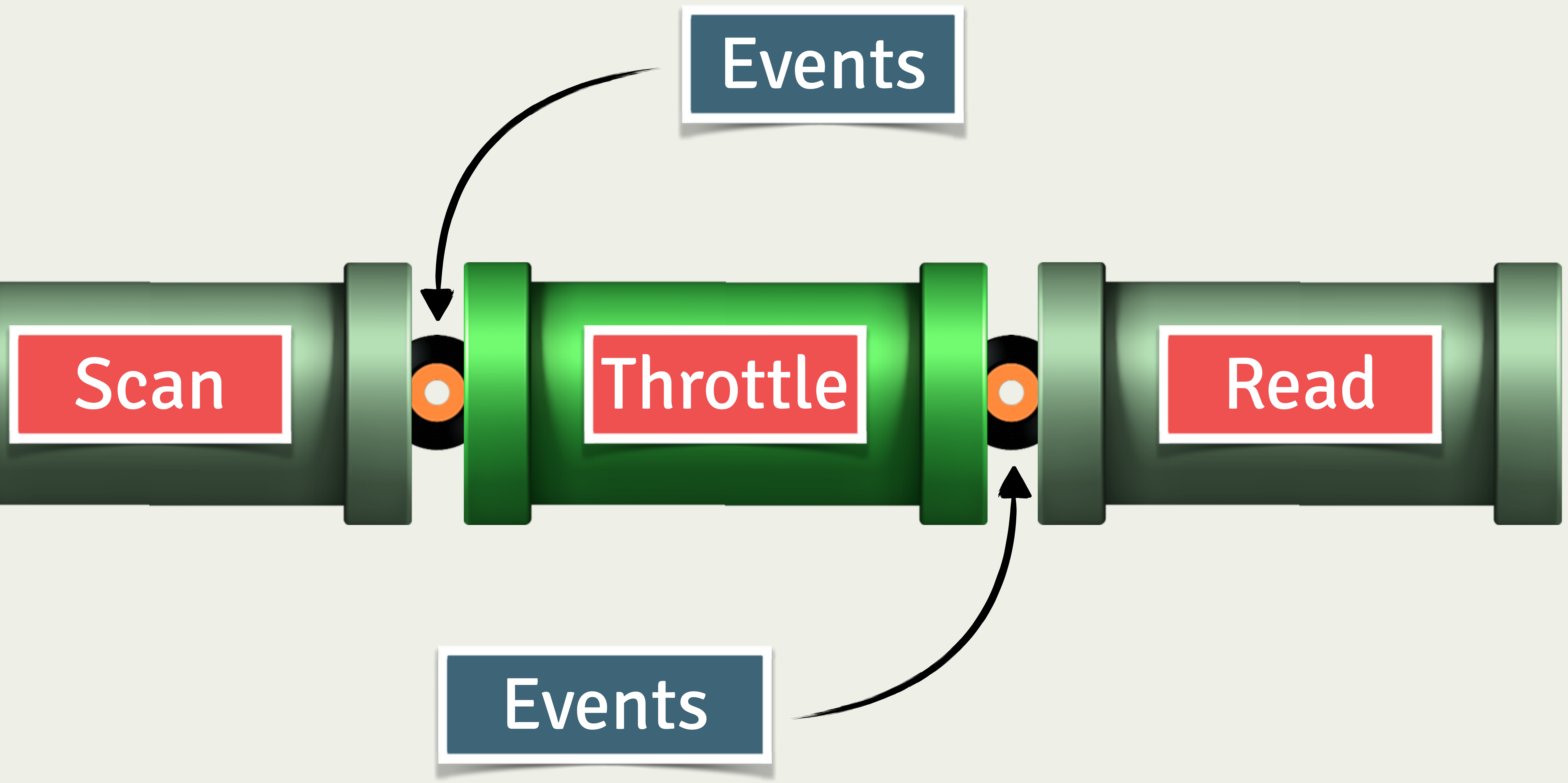
Vertical Composition

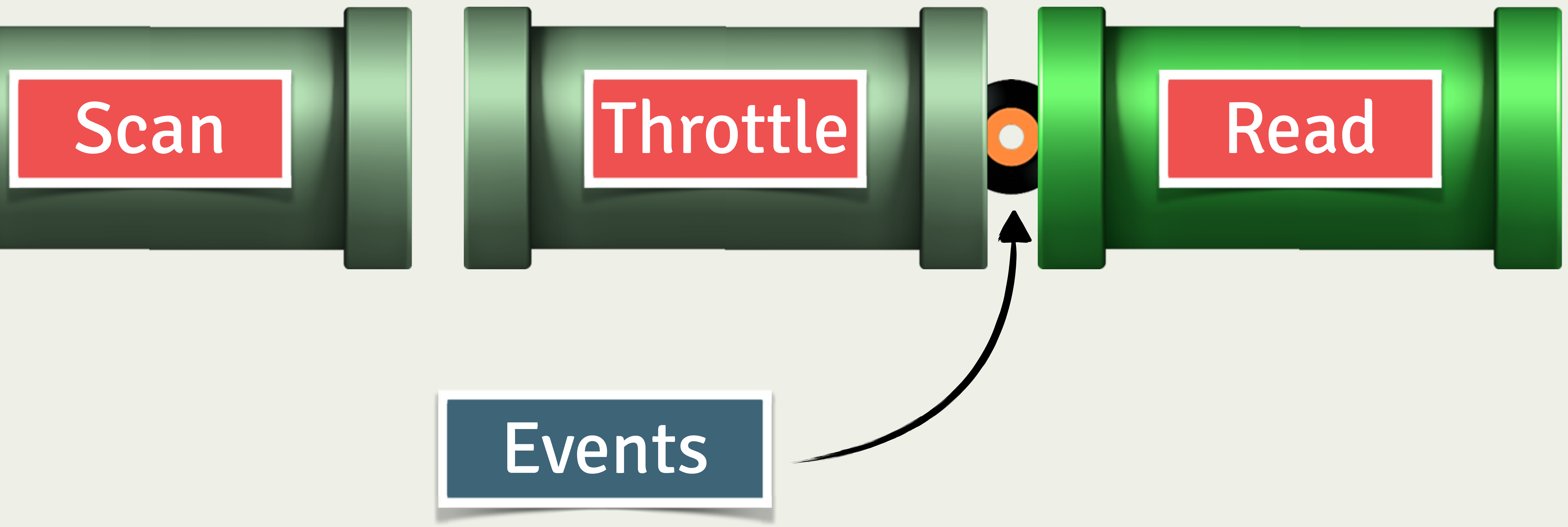
Mail Delivery
Poll
Prepare
Classify
Deliver

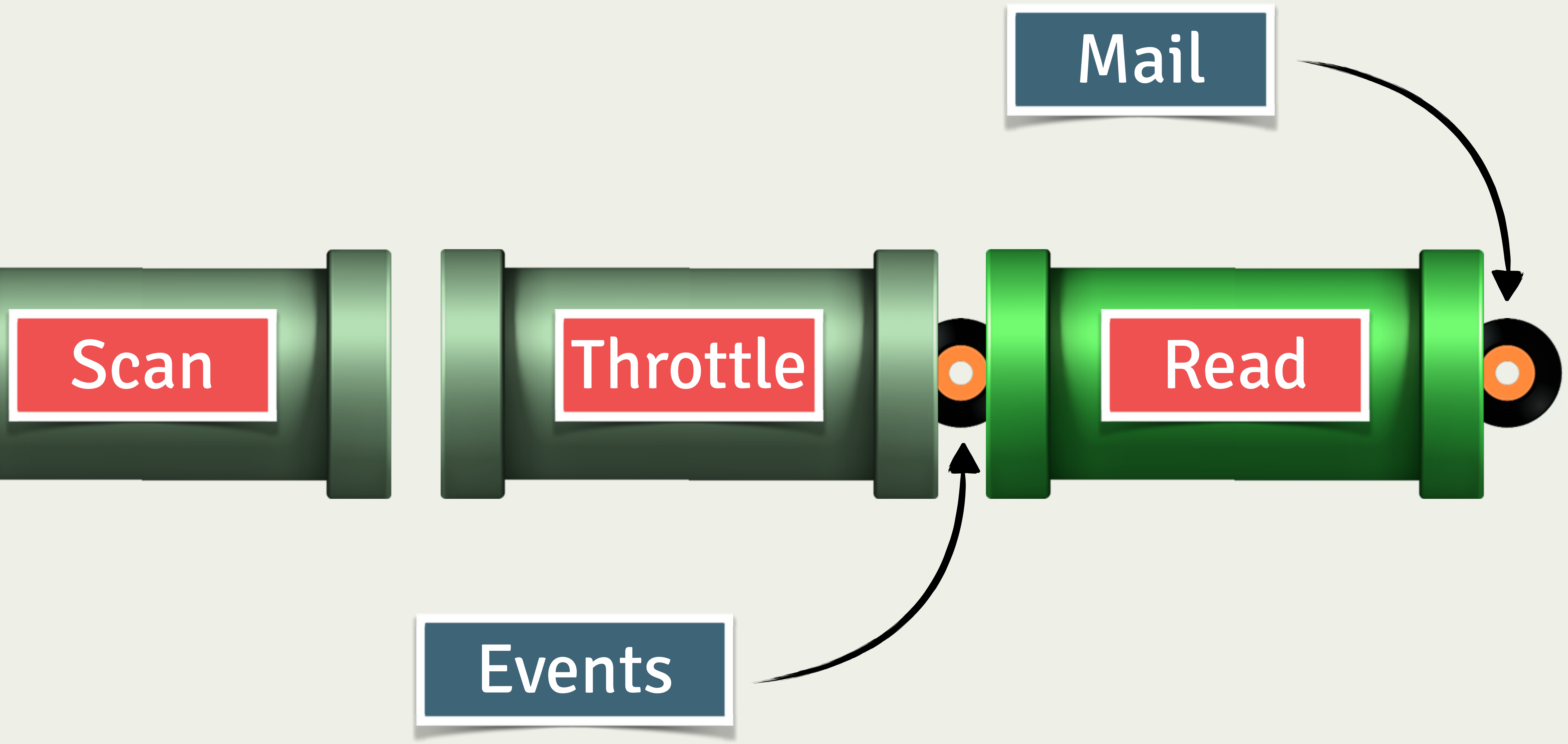
Poll
Scan
Throttle
Read

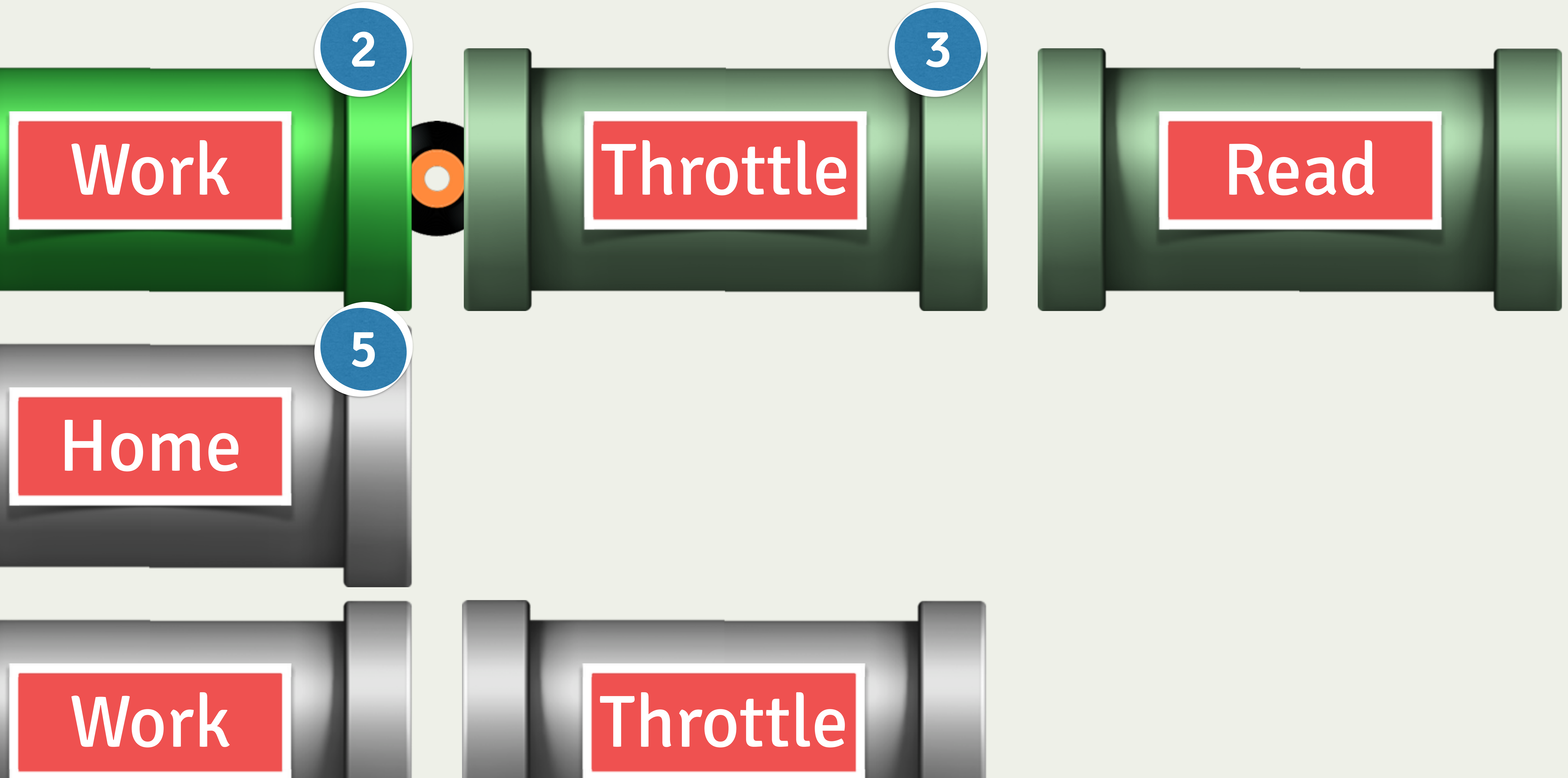

Events
Scan
Throttle
Read

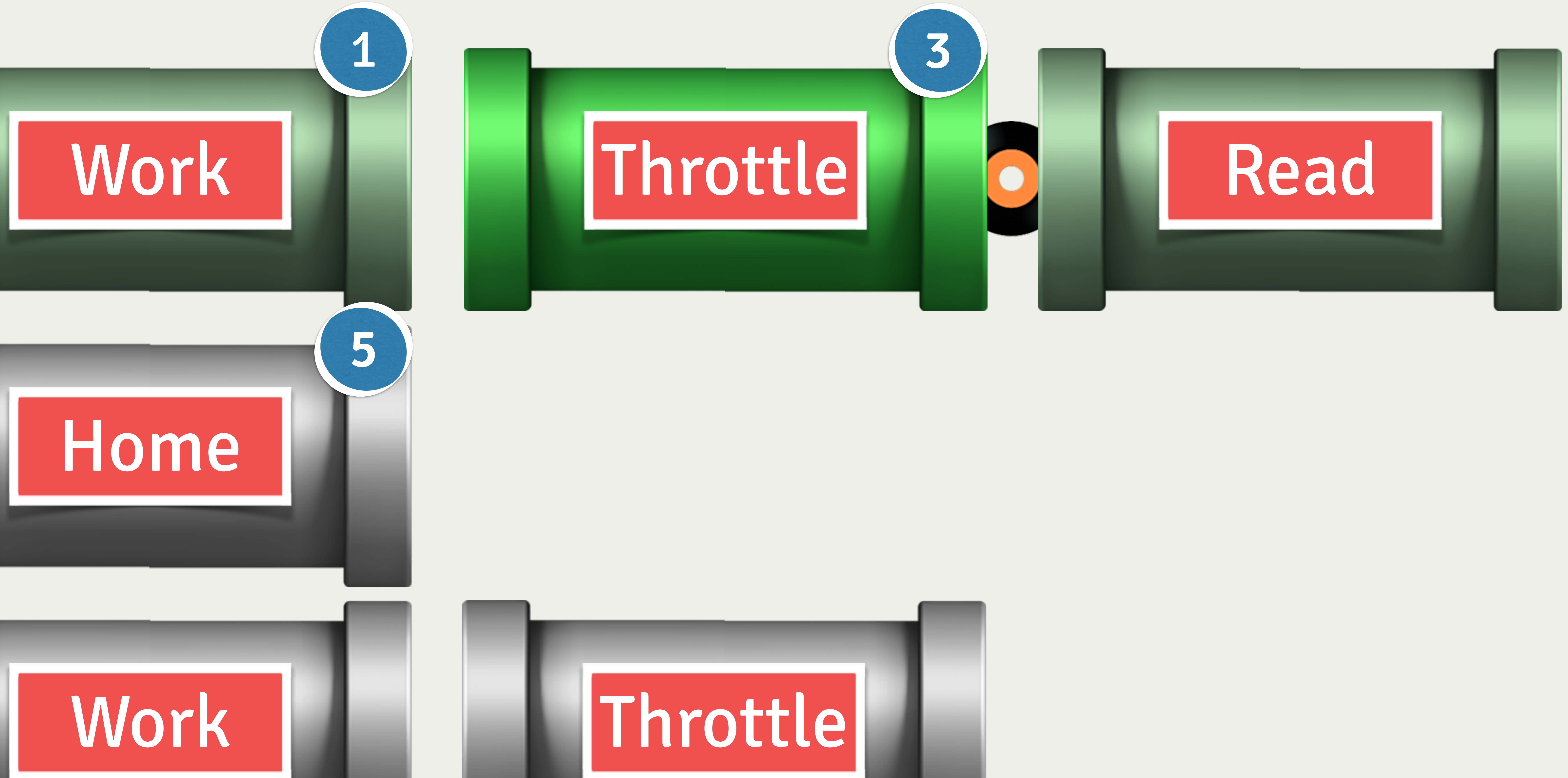

Events
Scan
Throttle
Read

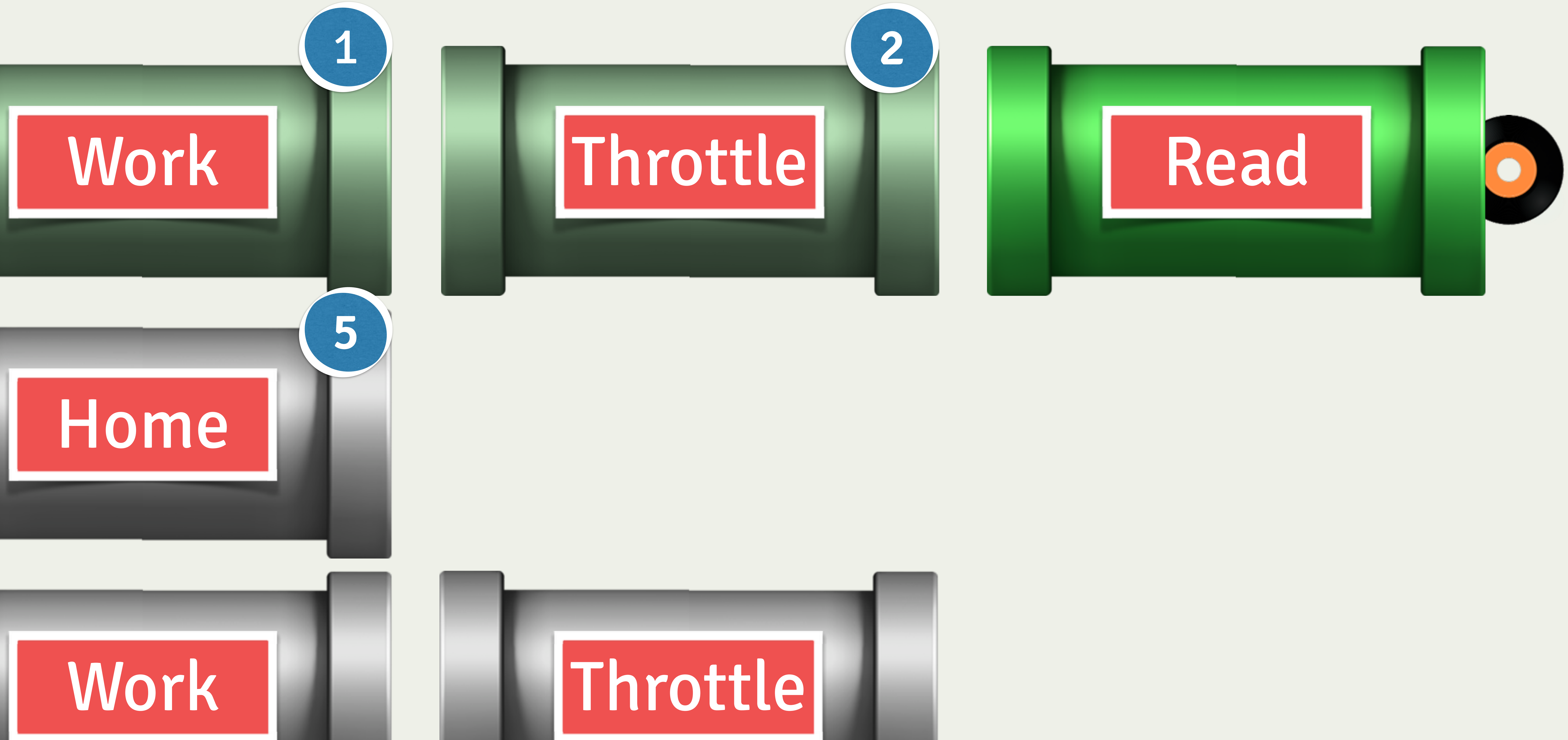

Events
Scan
Throttle
Read
Events

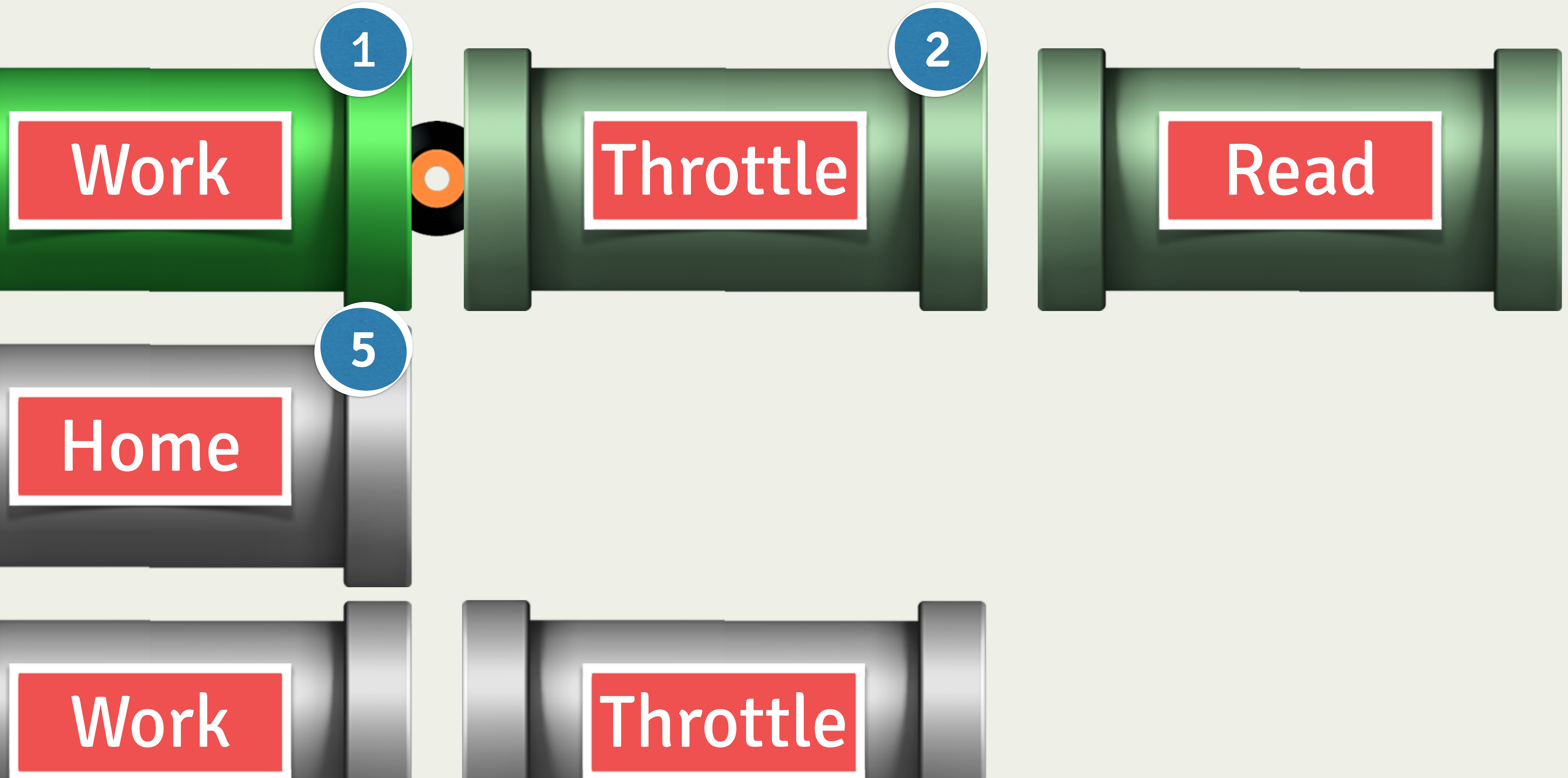

Scan
Throttle
Read
Events

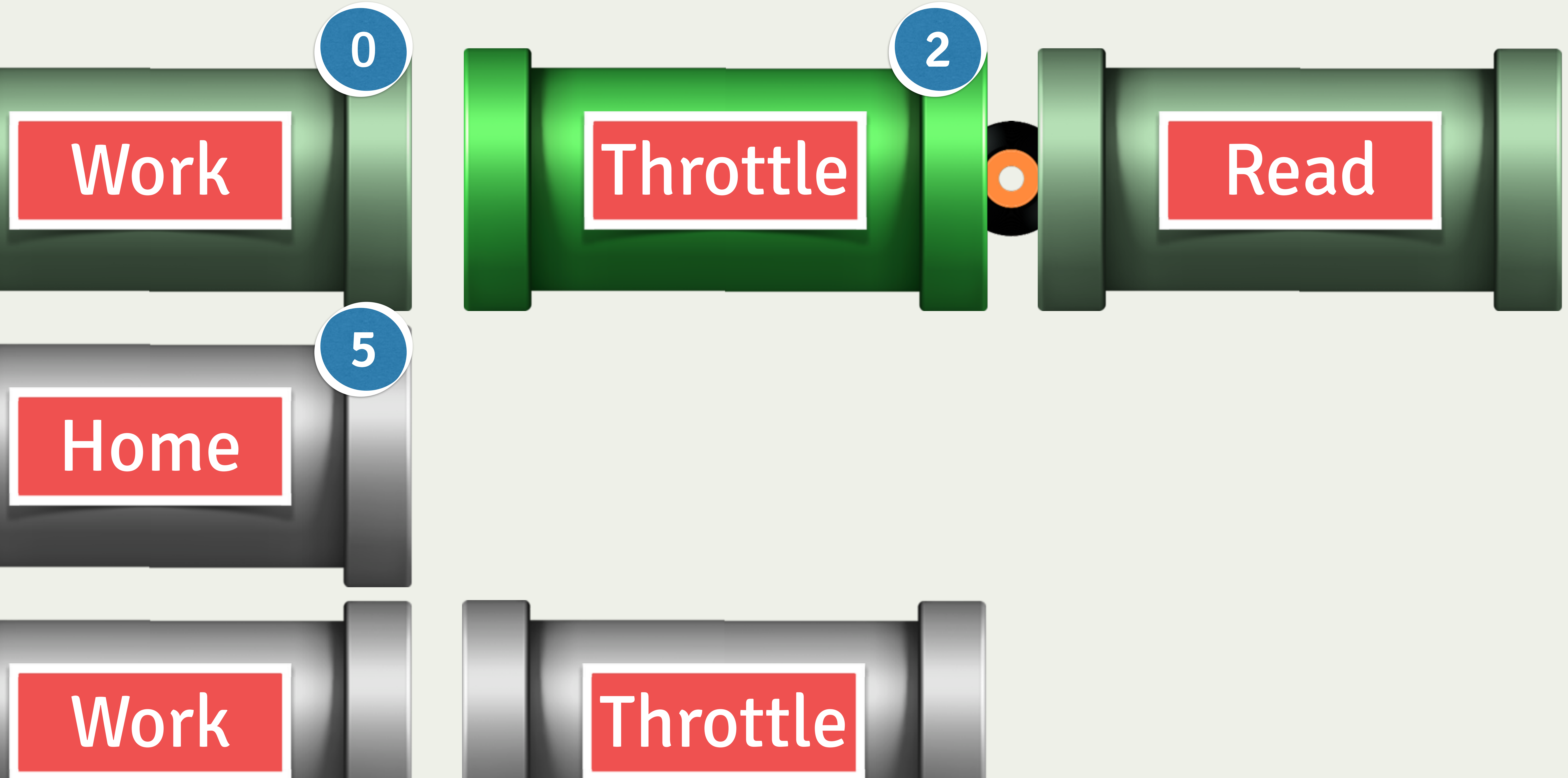

Mail
Scan
Throttle
Read
Events

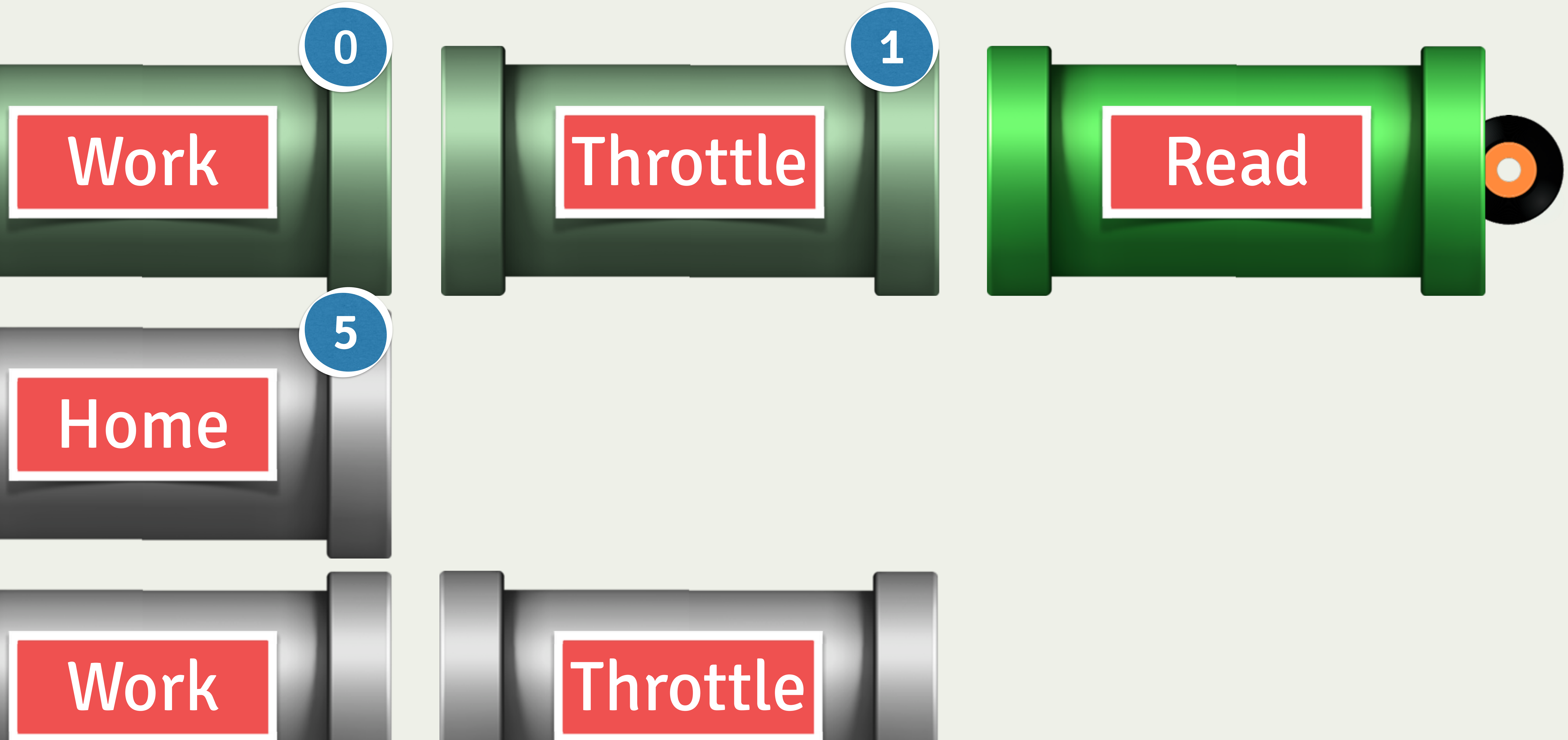

2
3
5
Work
Throttle
Read
Home
Work
Throttle

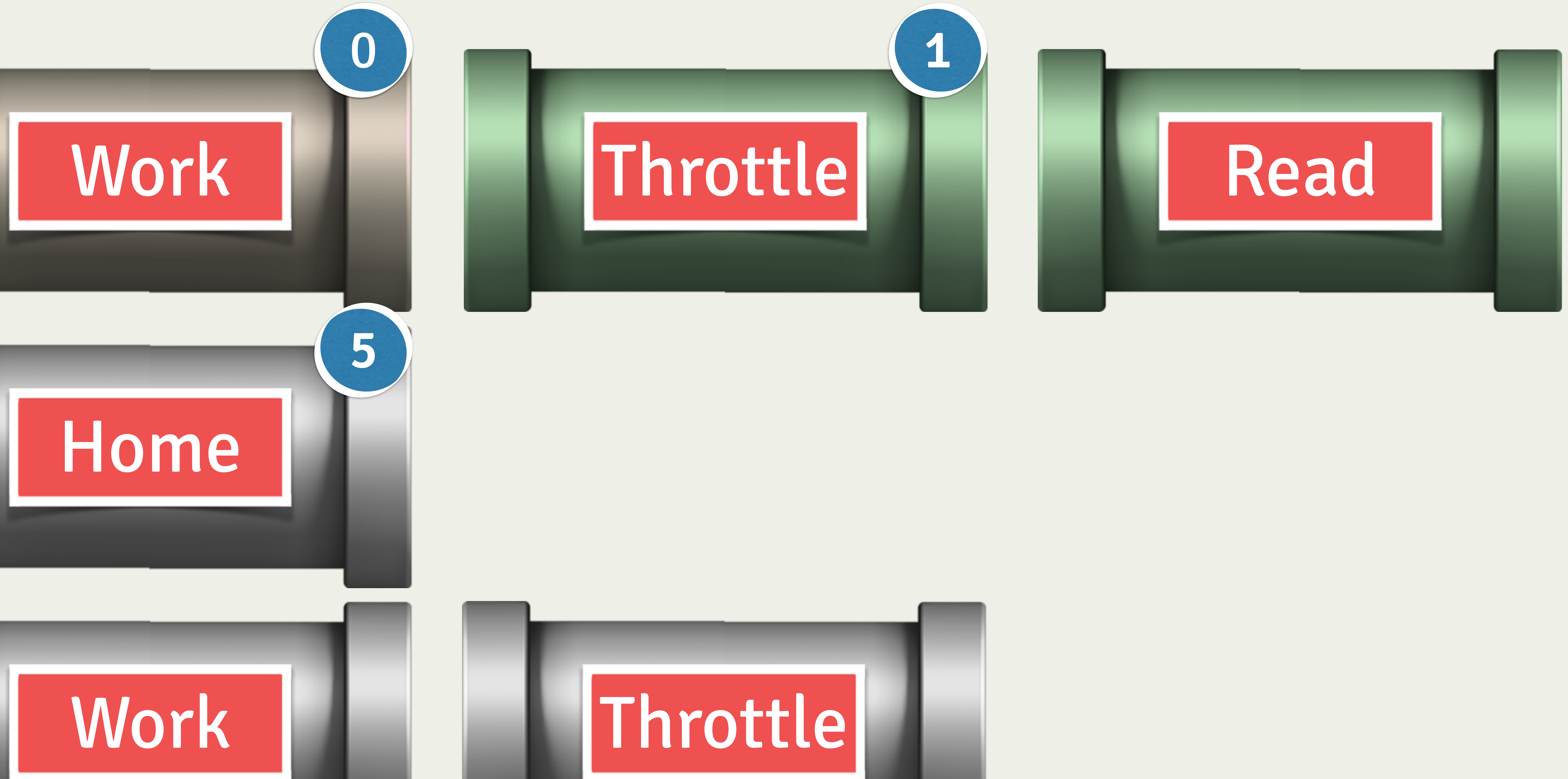

1
Work
Throttle
3
Read
5
Home
Work
Throttle

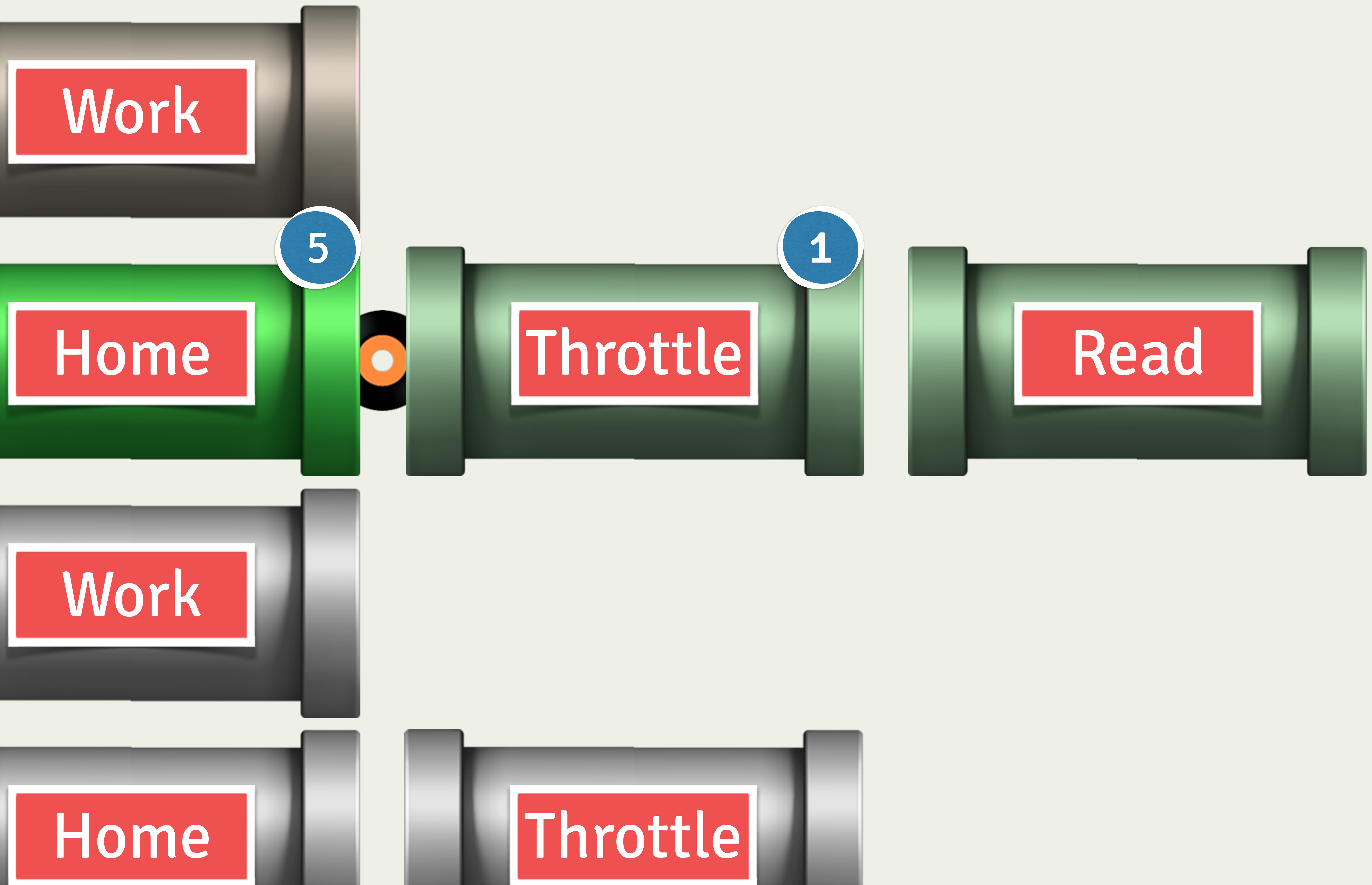

1
Work
2
Throttle
Read
5
Home
Work
Throttle

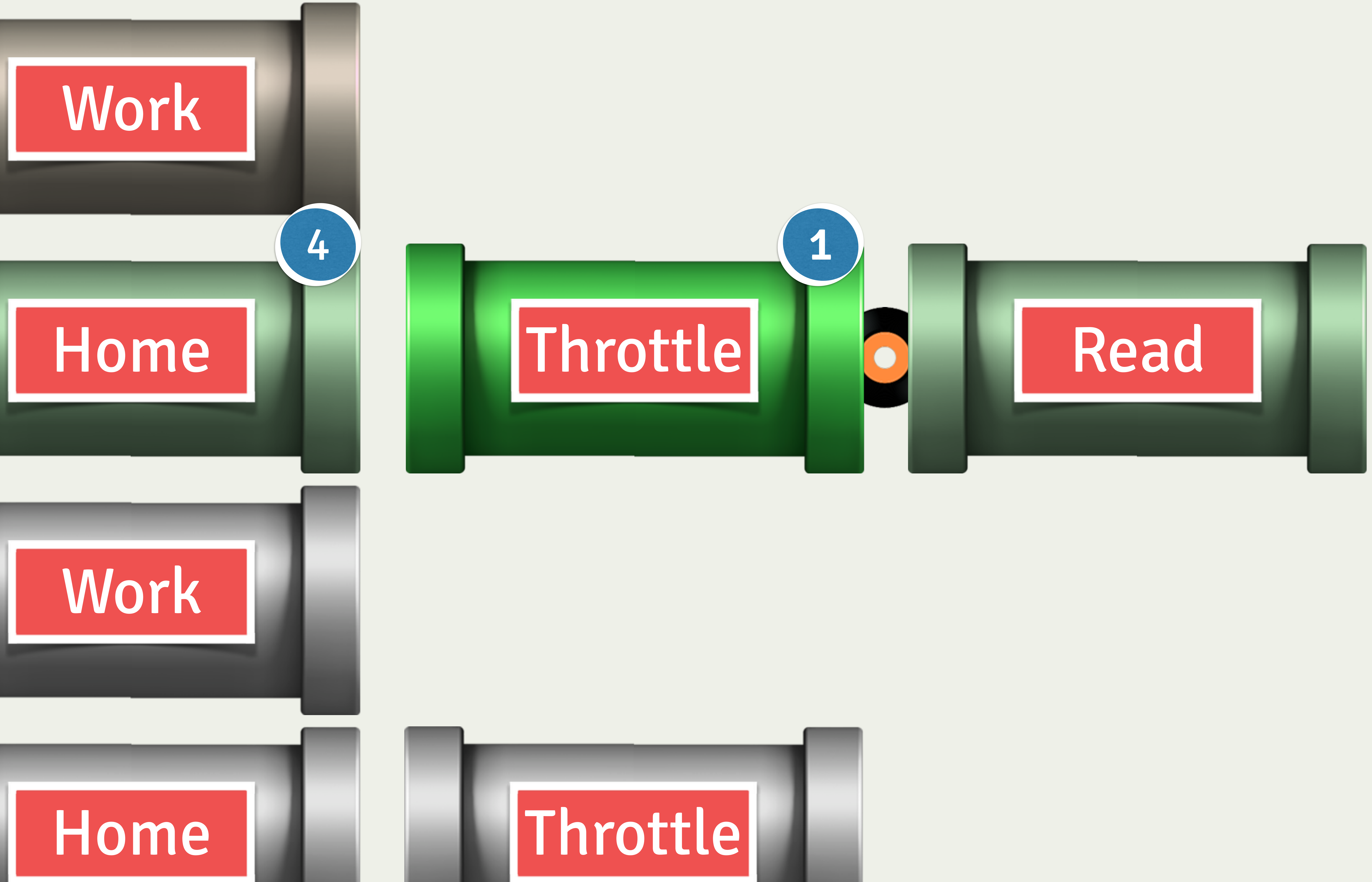

1
Work
2
Throttle
Read
5
Home
Work
Throttle

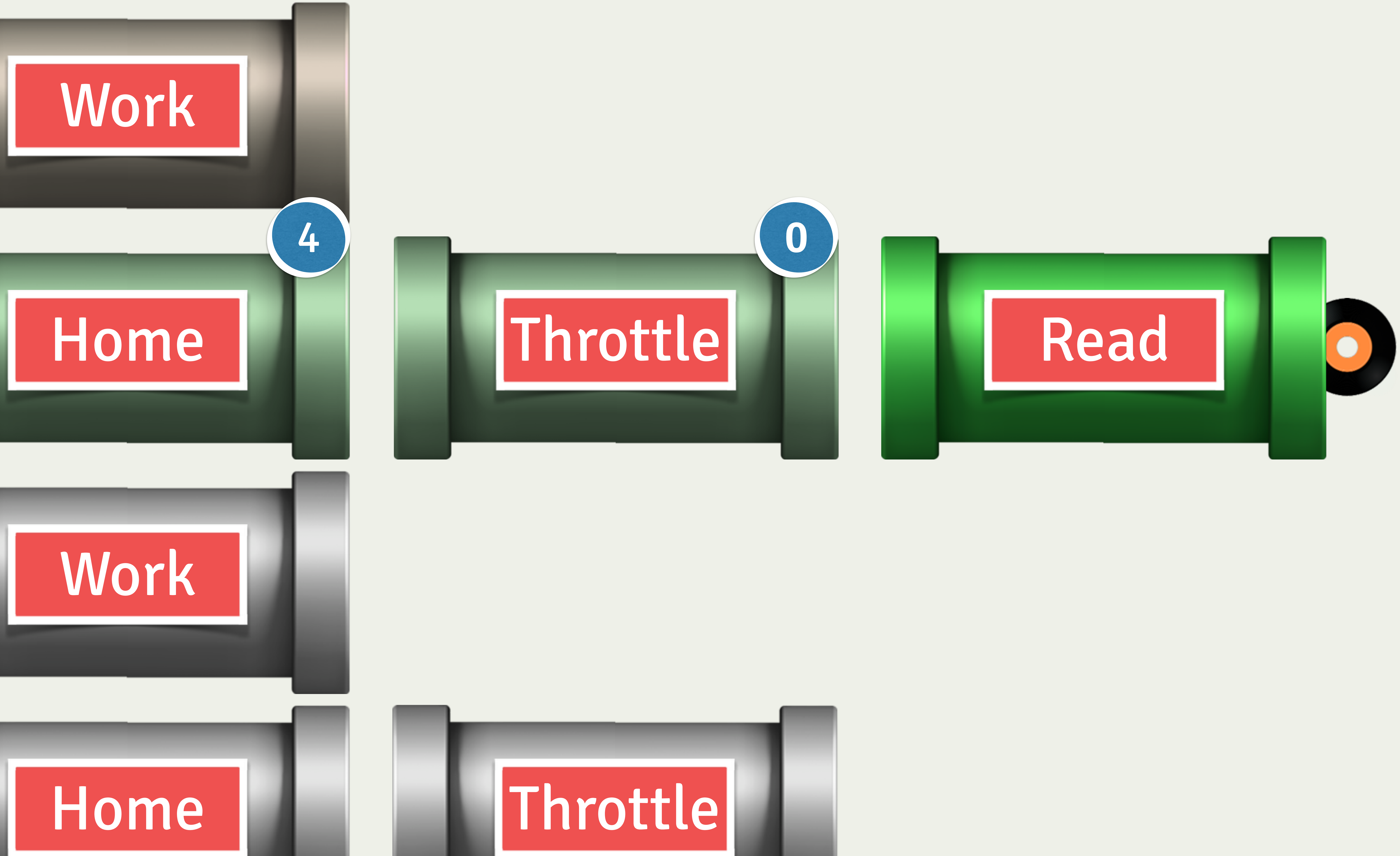

0
Work
Throttle
2
Read
5
Home
Work
Throttle

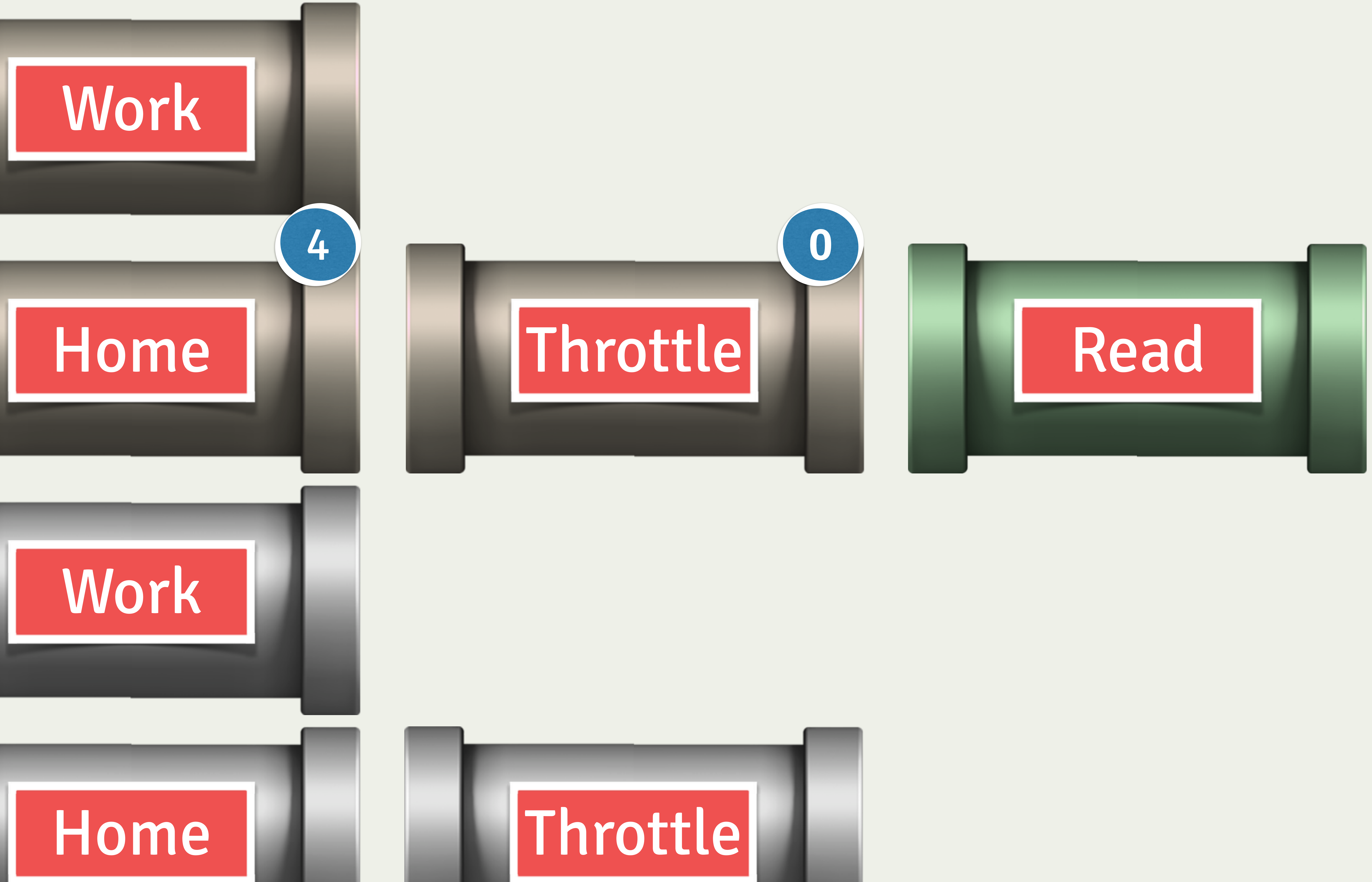

0
Work
1
Throttle
Read
5
Home
Work
Throttle

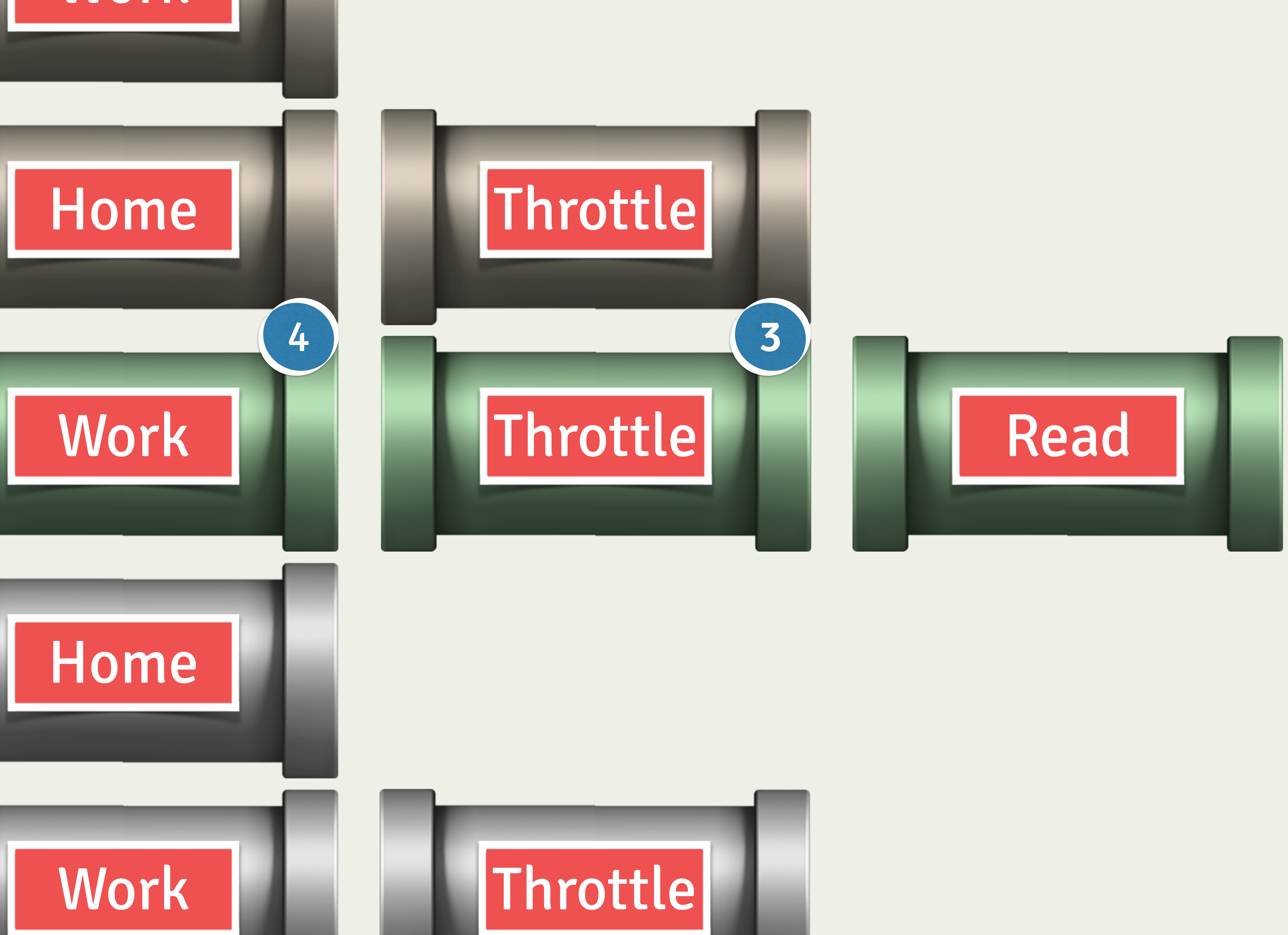

0
Work
Throttle
1
Read
5
Home
Work
Throttle

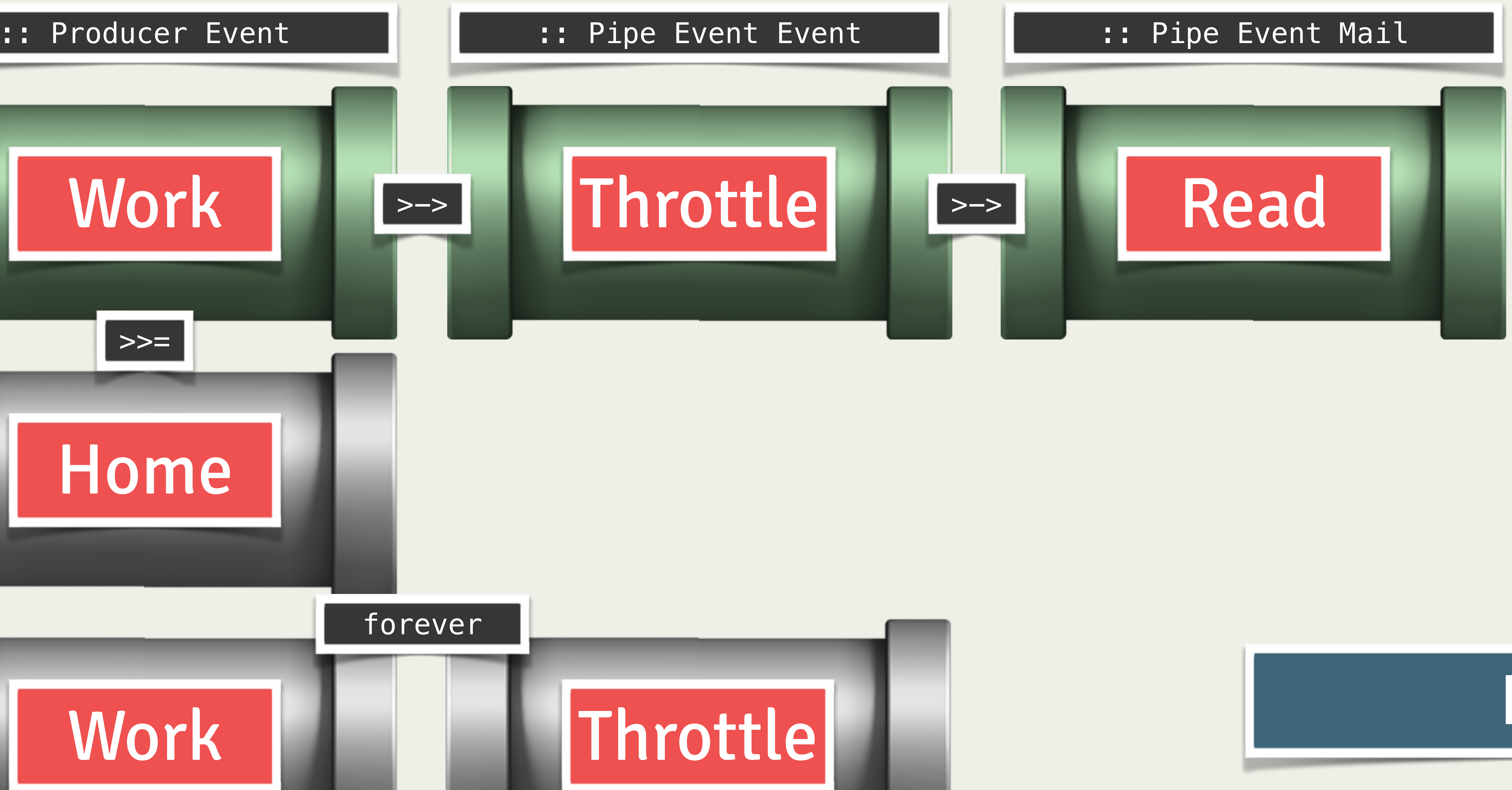

Work
5
1
Home
Throttle
Read
Work
Home
Throttle


Work
4
Home
Throttle
1
Read
Work
Home
Throttle

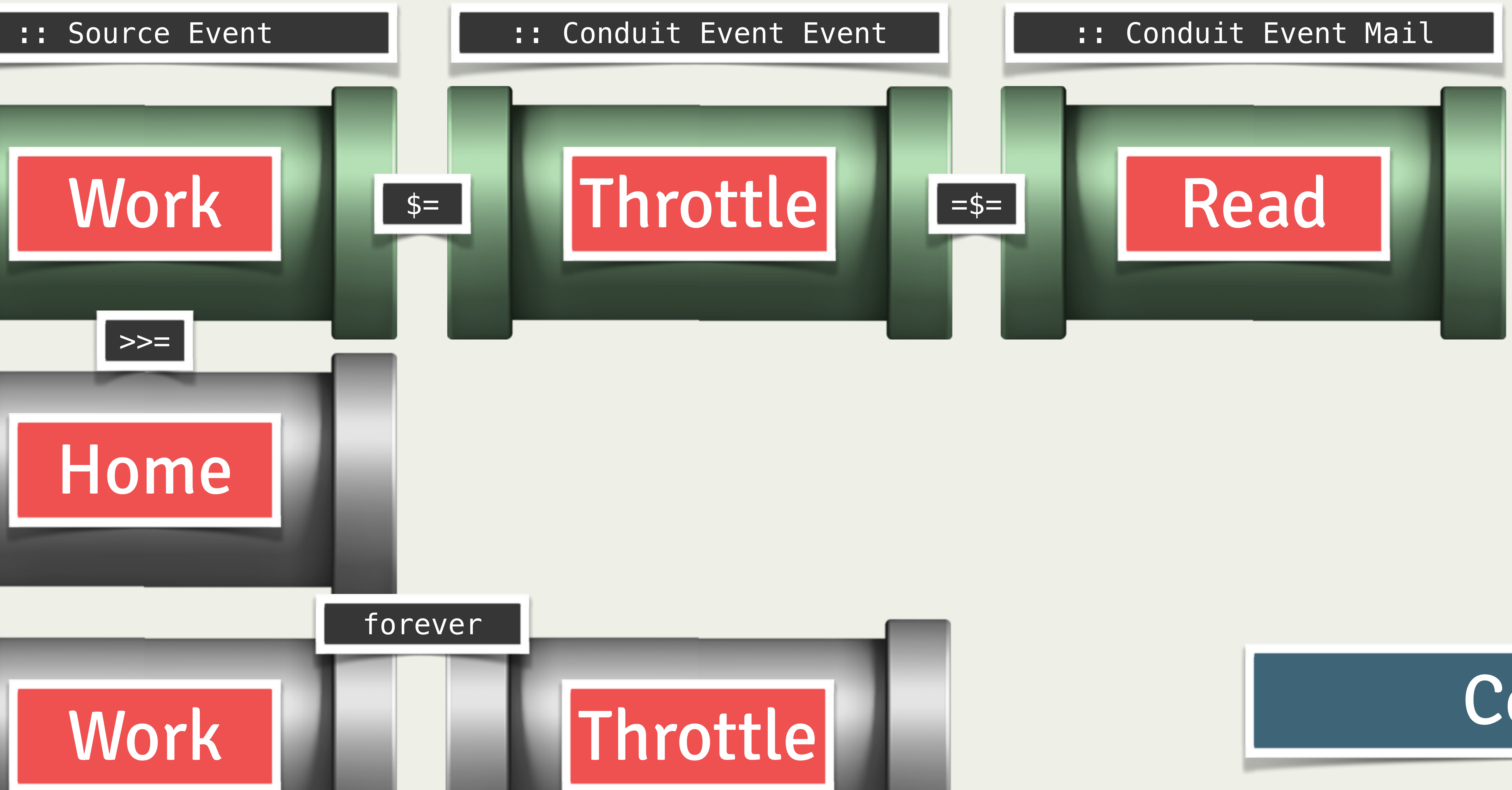

Work
4
Home
Throttle
0
Read
Work
Home
Throttle


Work
4
Home
Throttle
0
Read
Work
Home
Throttle

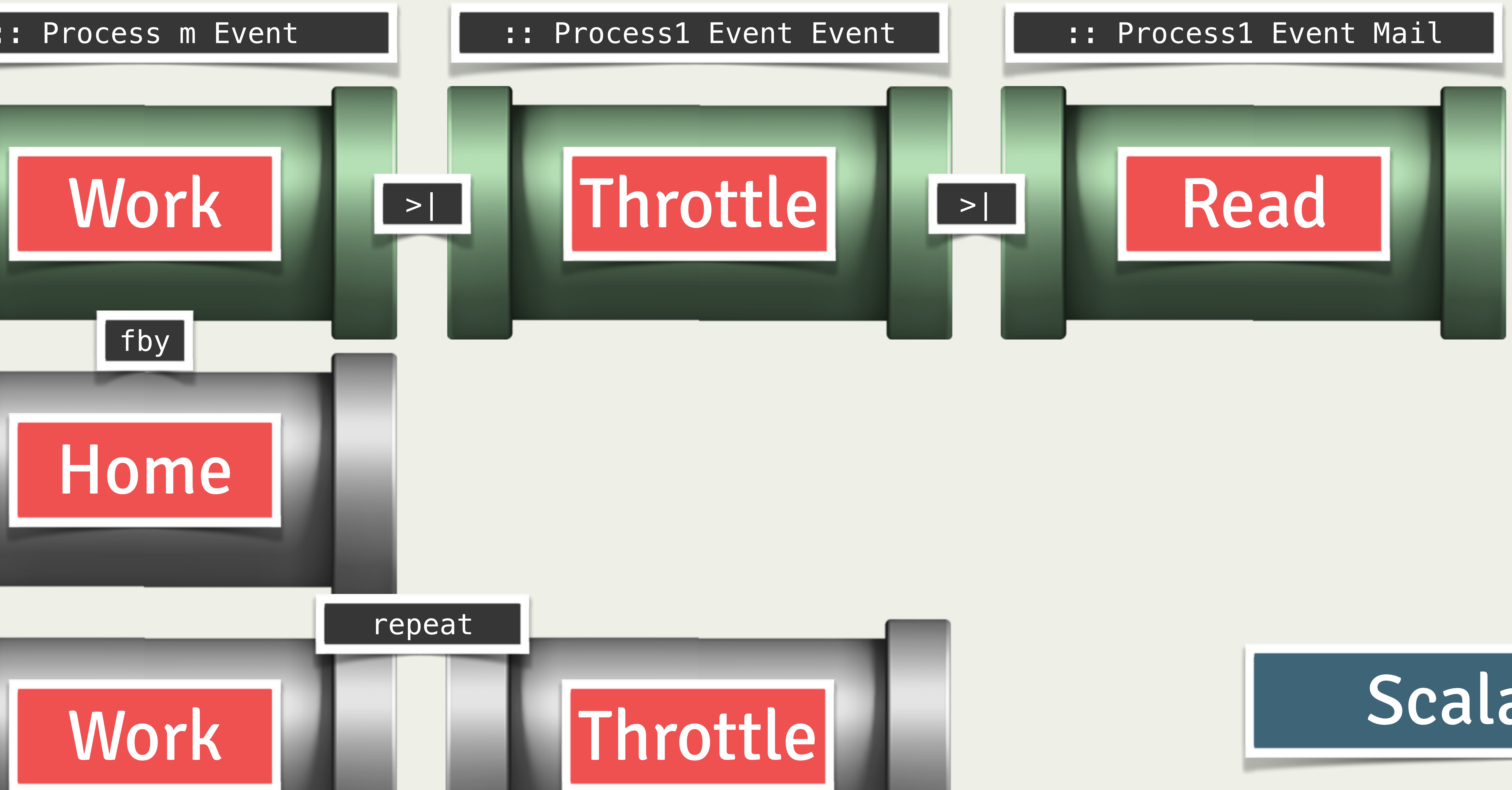

Home
Throttle
4
3
Work
Throttle
Read
Home
Work
Throttle


:: Producer Event
:: Pipe Event Event
:: Pipe Event Mail
Work
>->
Throttle
>->
Read
>>=
Home
forever
Work
Throttle

Pipes


:: Source Event
:: Conduit Event Event
:: Conduit Event Mail
Work
$=
Throttle
=$=
Read
>>=
Home
forever
Work
Throttle

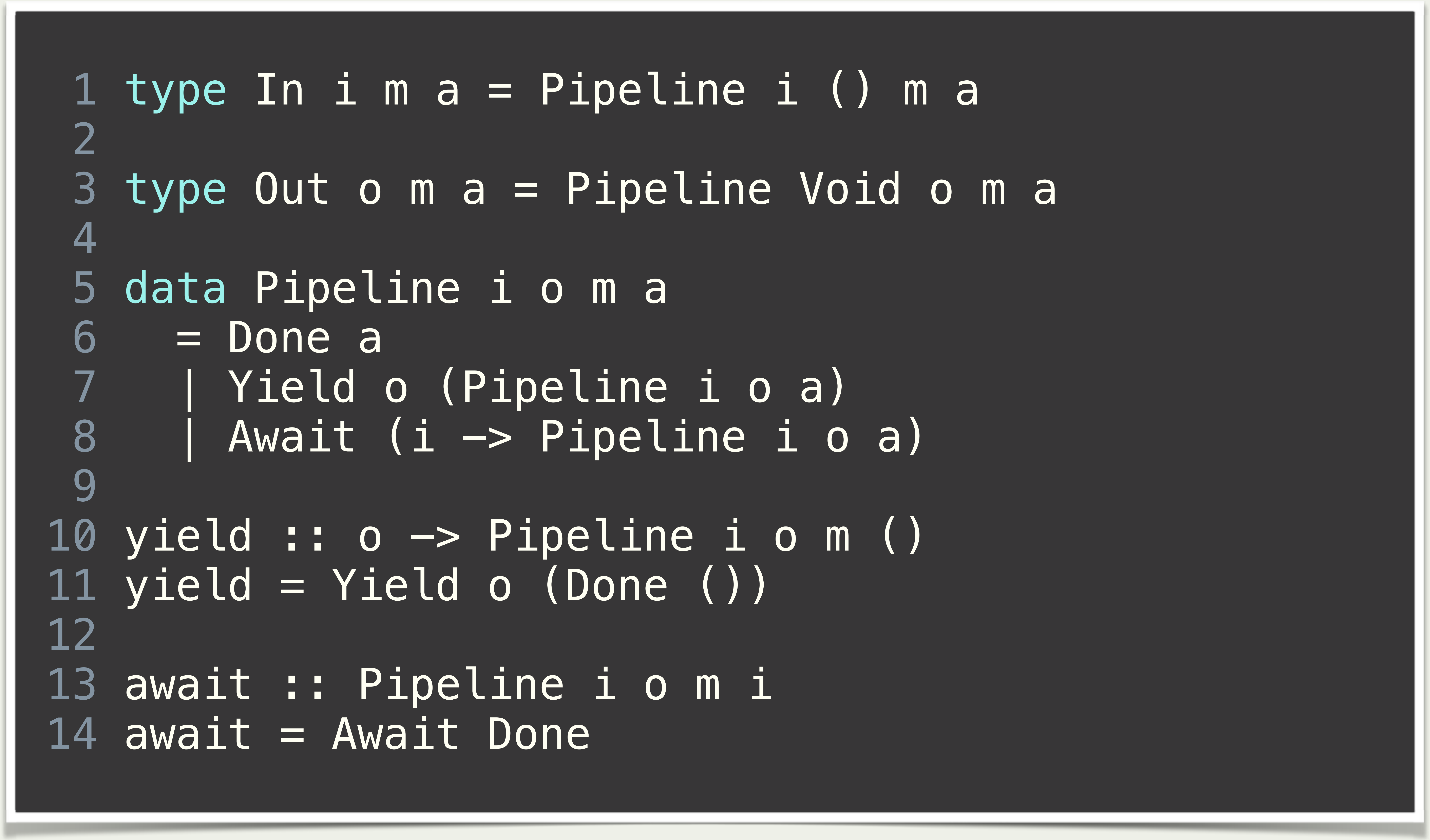
Conduit



Scalaz Stream

Intuition 3: Parsers

```haskell
type In i m a = Pipeline i () m a

type Out o m a = Pipeline Void o m a

data Pipeline i o m a
  = Done a
  | Yield o (Pipeline i o a)
  | Await (i -> Pipeline i o a)

```



```haskell
type In i m a = Pipeline i () m a

type Out o m a = Pipeline Void o m a

data Pipeline i o m a
  = Done a
  | Yield o (Pipeline i o a)
  | Await (i -> Pipeline i o a)

yield :: o -> Pipeline i o m ()
yield = Yield o (Done ())

await :: Pipeline i o m i
await = Await Done
```

```haskell
one :: Pipeline i i m ()
one = do
  i <- await
  yield i

cat :: Pipeline i i m ()
cat = forever one

pairs :: Pipeline i (i, i) m ()
pairs = forever $ do
  i1 <- await
  i2 <- await
  yield (i1, i2)
```

```haskell
counter :: Monad m => Pipeline i (Int, i) m ()
counter = flip evalStateT 0 . forever $ do
  i <- lift await
  n <- get
  lift . yield $ (n, i)

filter :: (i -> Bool) -> Pipeline i i m a
filter f = forever $ do
  i <- await
  when (f i) $ yield i
```

```
yield :: o -> Pipe i o m ()

await :: Pipe i o m i
```

Pipes

```
yield :: o -> ConduitM i o m r

await :: ConduitM i o m (Maybe i)

awaitForever :: (\i -> ConduitM i o m a)
             -> ConduitM i o m ()
```

Conduit

```
emit :: o -> Process f o

await1 :: Process1 i i
```

Scalaz Stream

Subtlety Fights Back

Internal vs External Management of Resources

Layered Streams

Constant Memory Streaming

How much does elegance **cost?**

to be continued...

@markhibberd