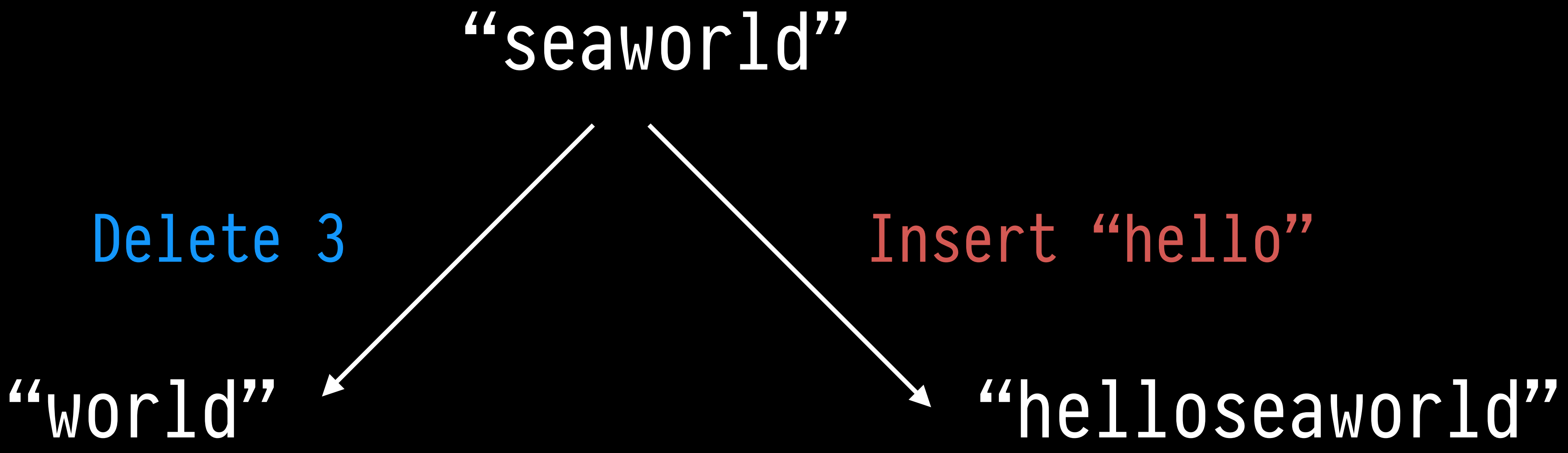
Streaming OT

or “the wonders of Gabriel Gonzalez”

OT

```
data Op = Retain Int | Delete Int | Insert String
type Delta = [Op]
      ^a diff
```

OT



“seaworld”

Delete 3

Insert “hello”

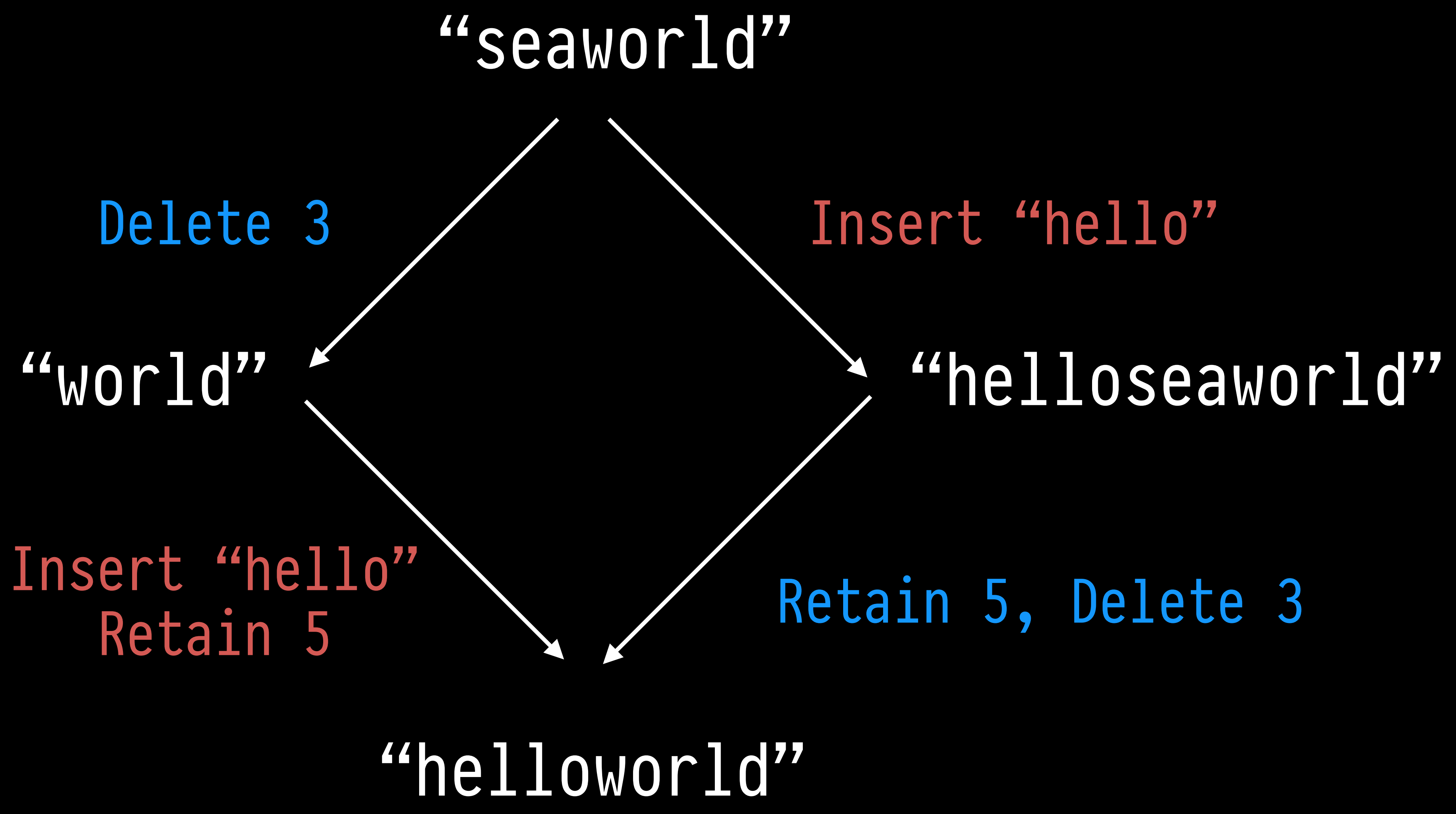
“world”

“helloseaworld”

OT

```
            incoming op    history
            v
transform (Delete 3:op1s) (Insert “hello”:op2s)
= [ Insert “hello”
  , Retain 5
  , transform (Delete 3:op1s) op2s
```

OT



“seaworld”

Delete 3

Insert “hello”

“world”

“helloseaworld”

Insert “hello”
Retain 5

Retain 5, Delete 3

“helloworld”

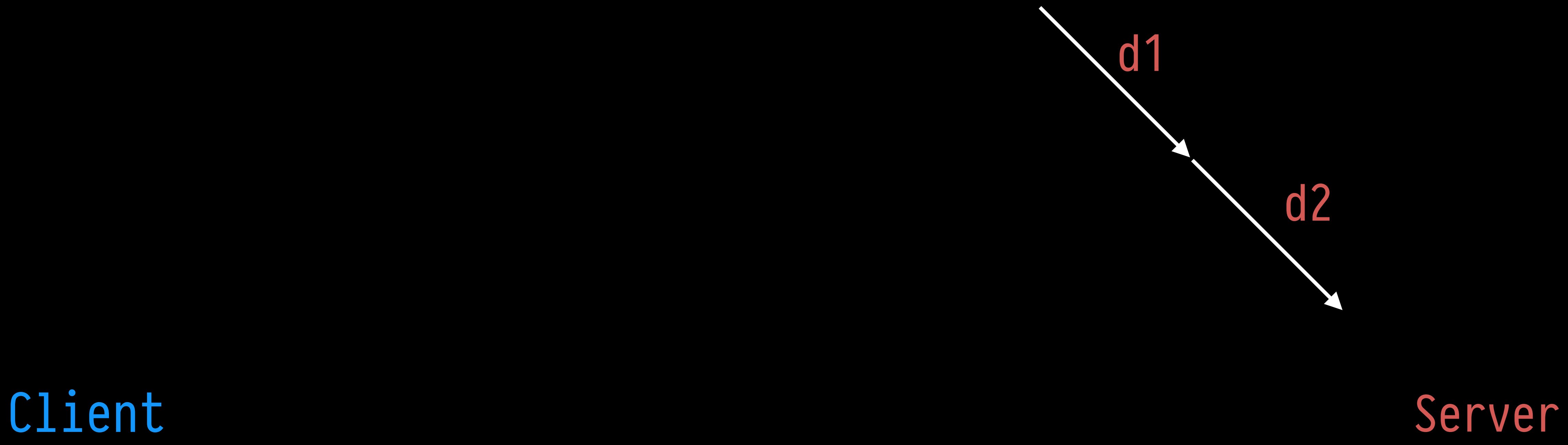
OT

$$\begin{tikzcd}
& {} \arrow[dl, "d1"'] \arrow[dr, "d2"] & \\
{} \arrow[dr, "d2'"'] & & {} \arrow[dl, "d2'"] \\
& {} &
\end{tikzcd}$$

```
d1’ = transform d1 against d2
d2’ = transform d2 against d1

     d1 <> d1’ == d2 <> d2’
```

proof: easy induction and case distinction. QED.

OT



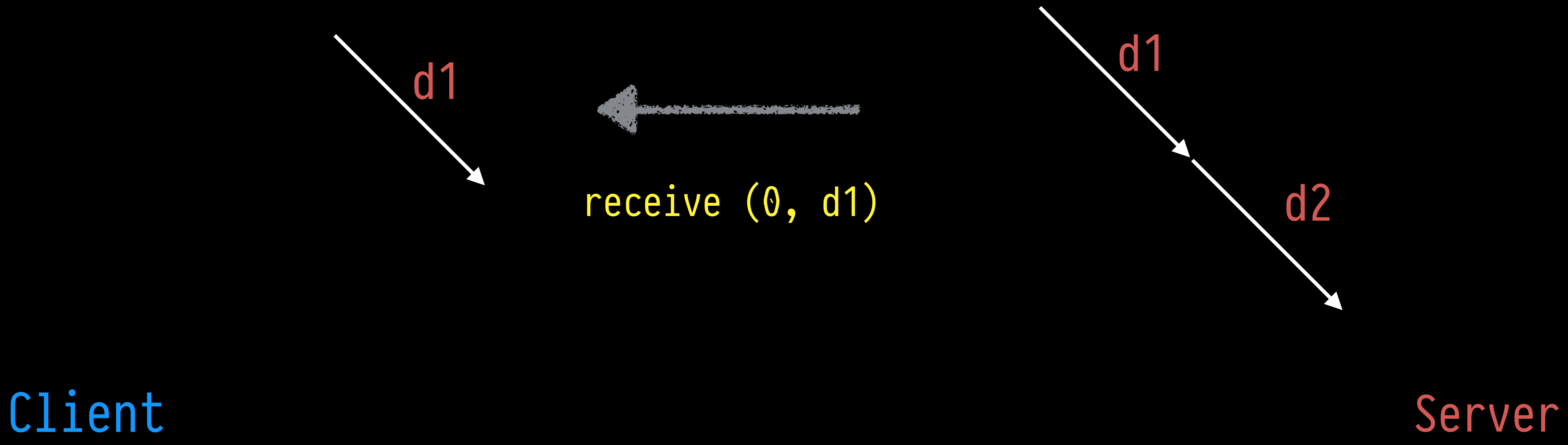
d1

d2

Client

Server

OT



d1

receive (0, d1)

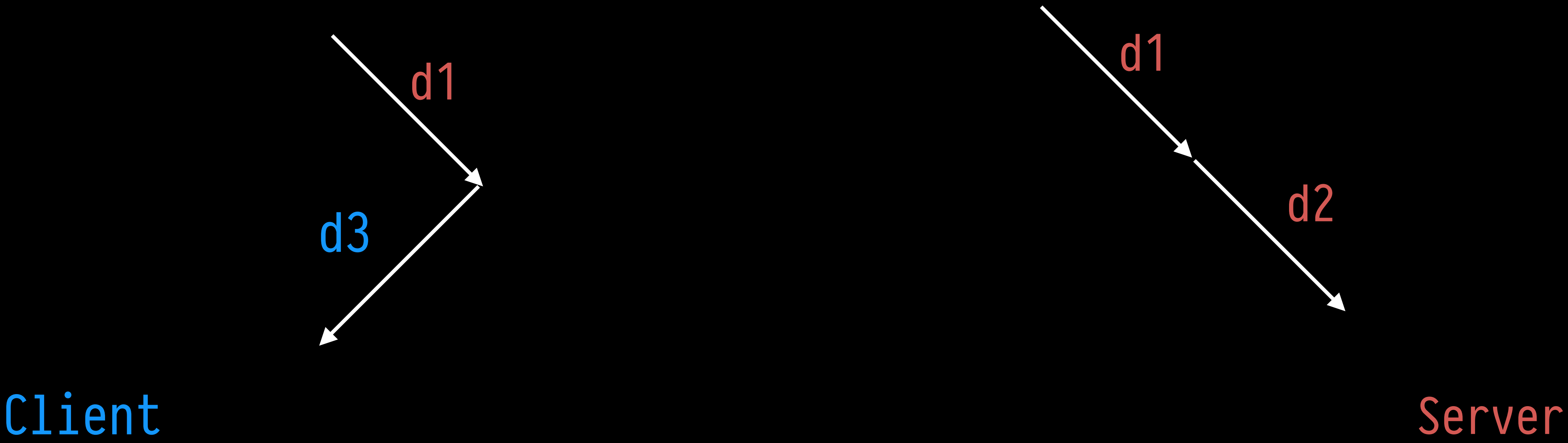
d1

d2

Client

Server

OT



d1

d3

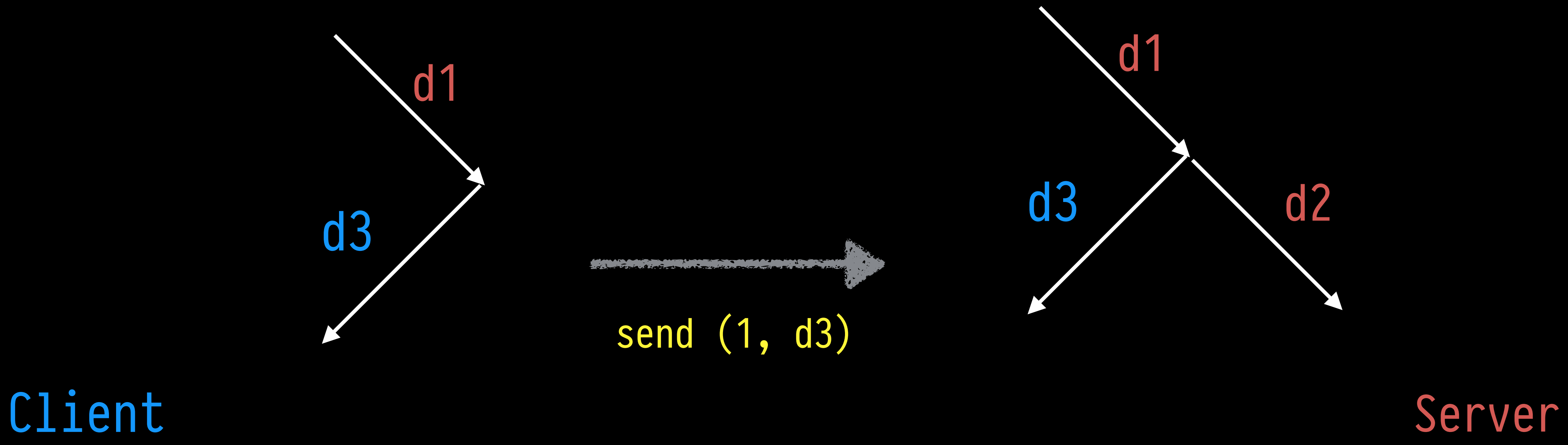
d1

d2

Client

Server

OT



d1

d3

Client

send (1, d3)

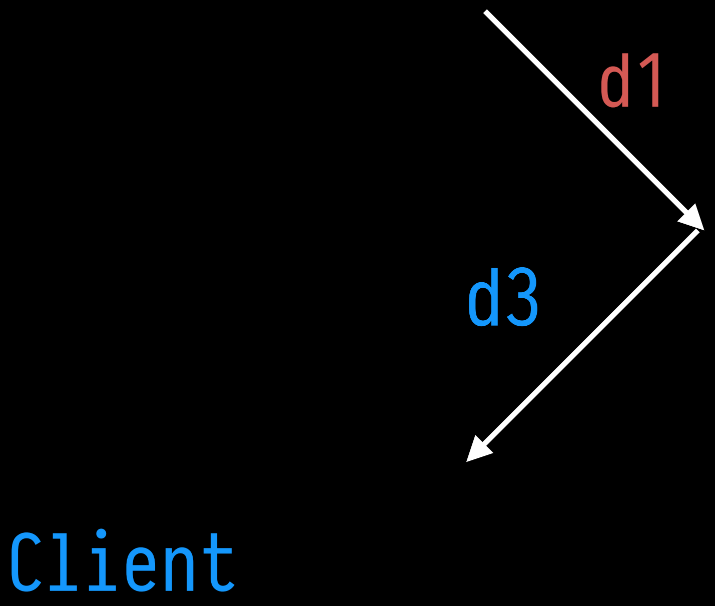
d1

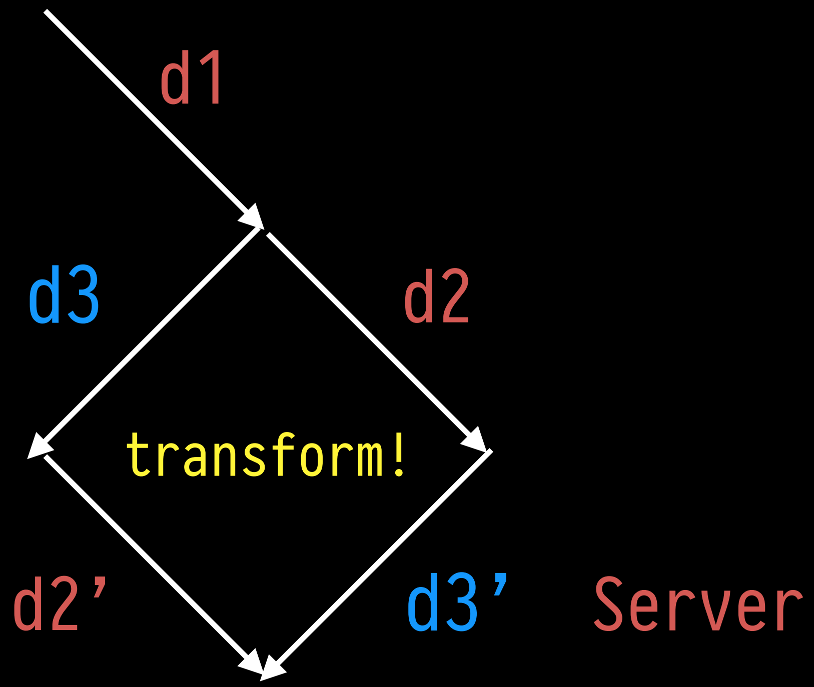
d3

d2

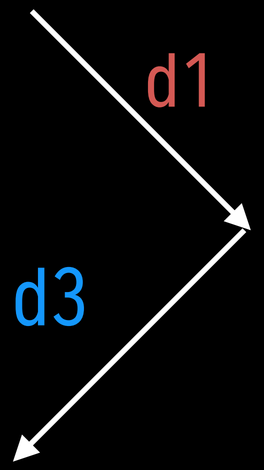
Server

OT



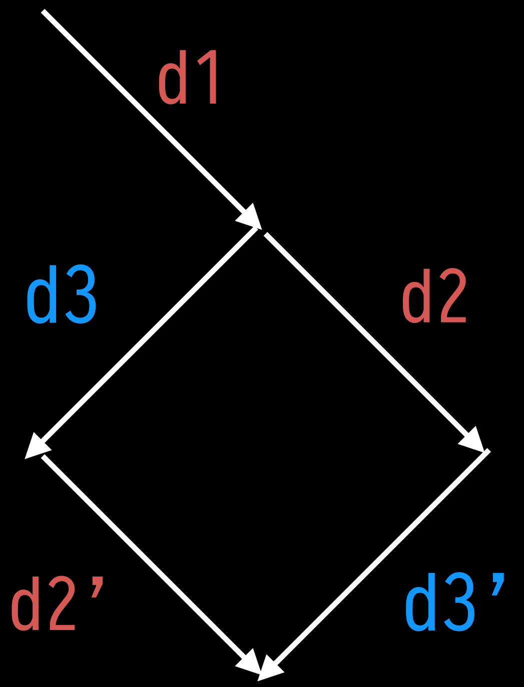


OT


d1

d3

Client


d1

d3

d2

d2’

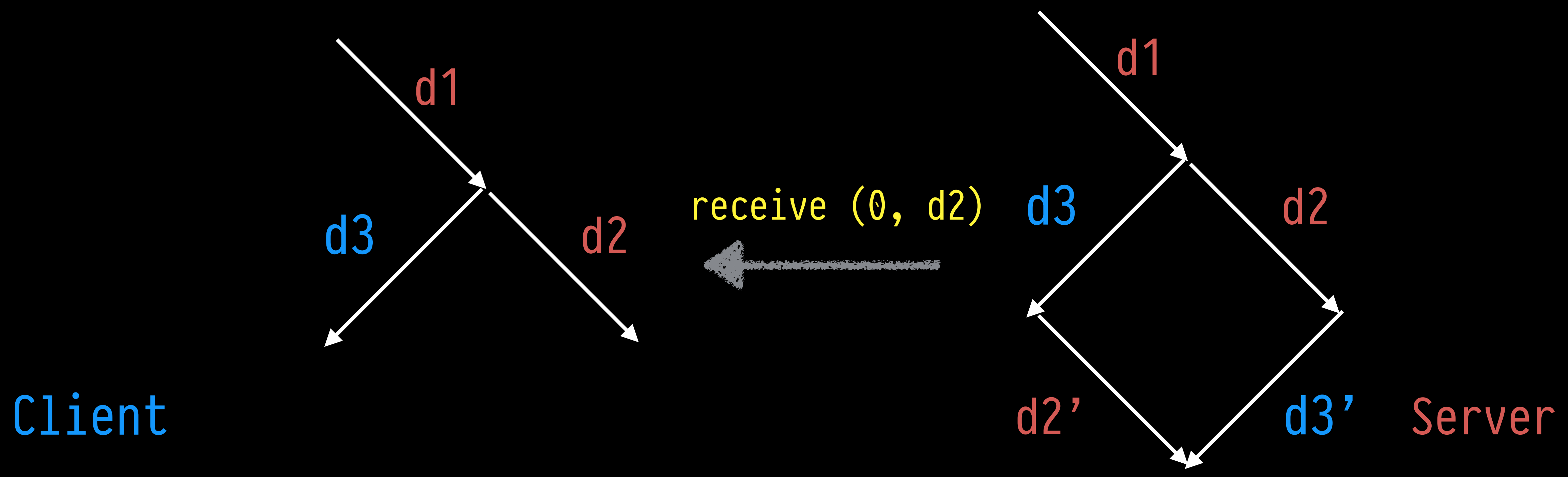
d3’

Server

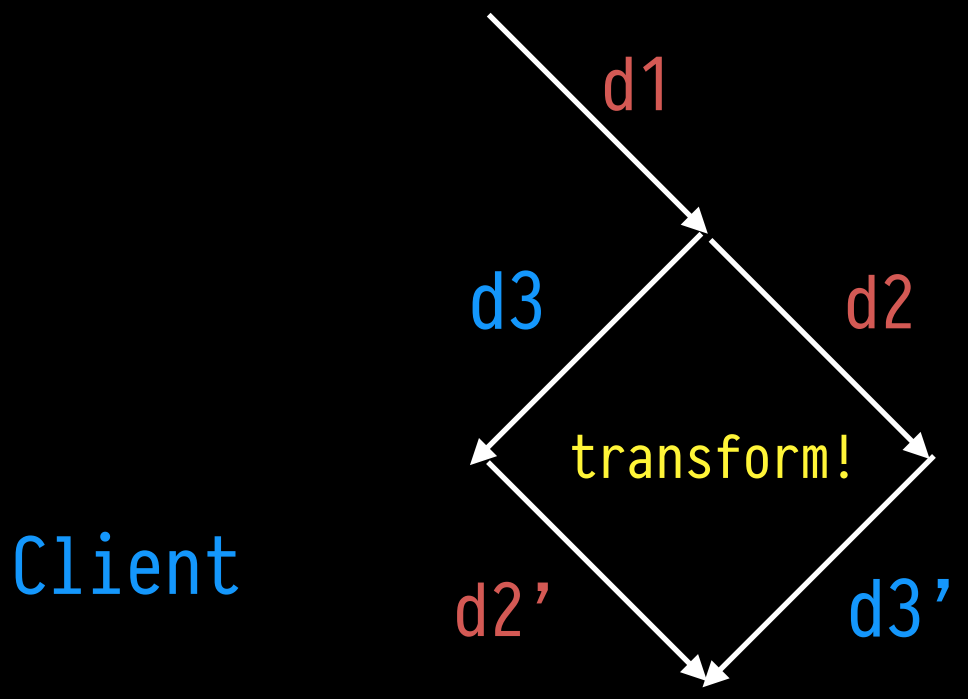
sends d3’ to other clients

new history: [d2’]

OT



OT



d1

d3

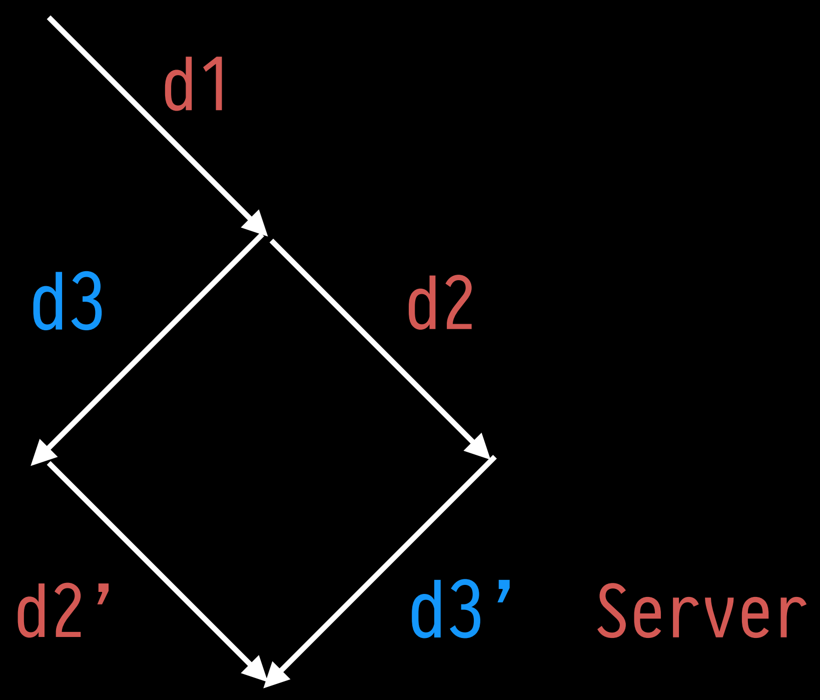
d2

transform!

Client

d2’

d3’



d1

d3

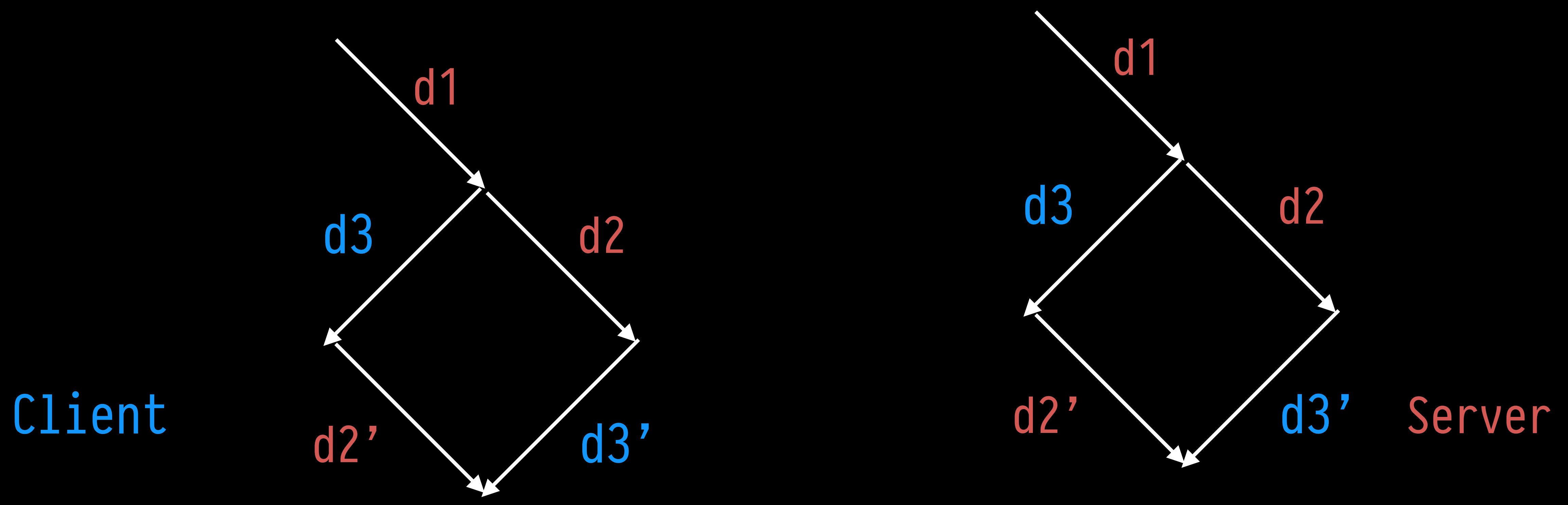
d2

d2’

d3’

Server

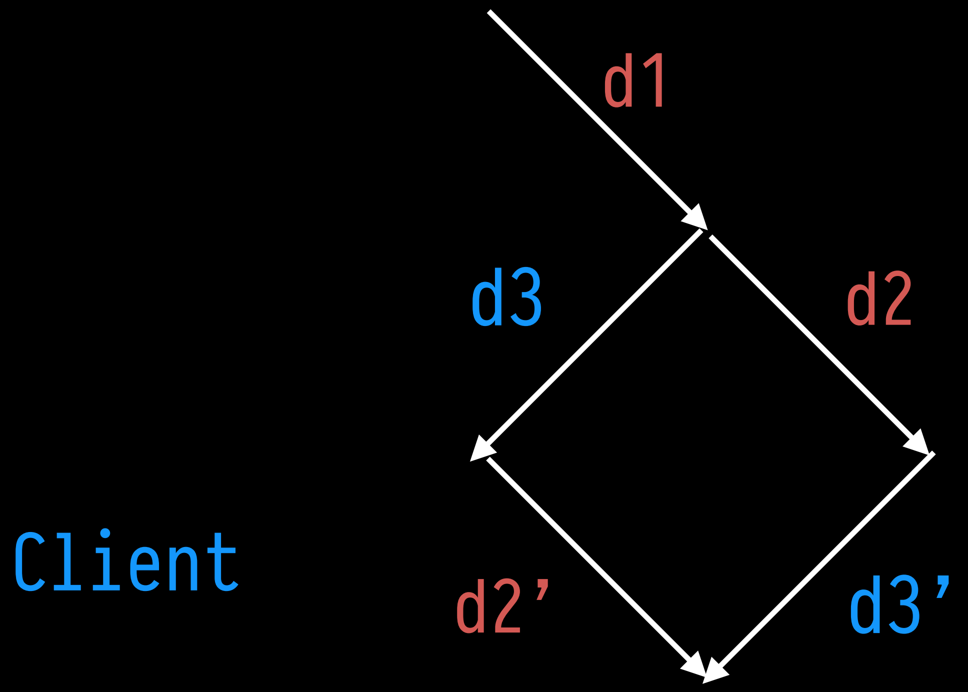
OT

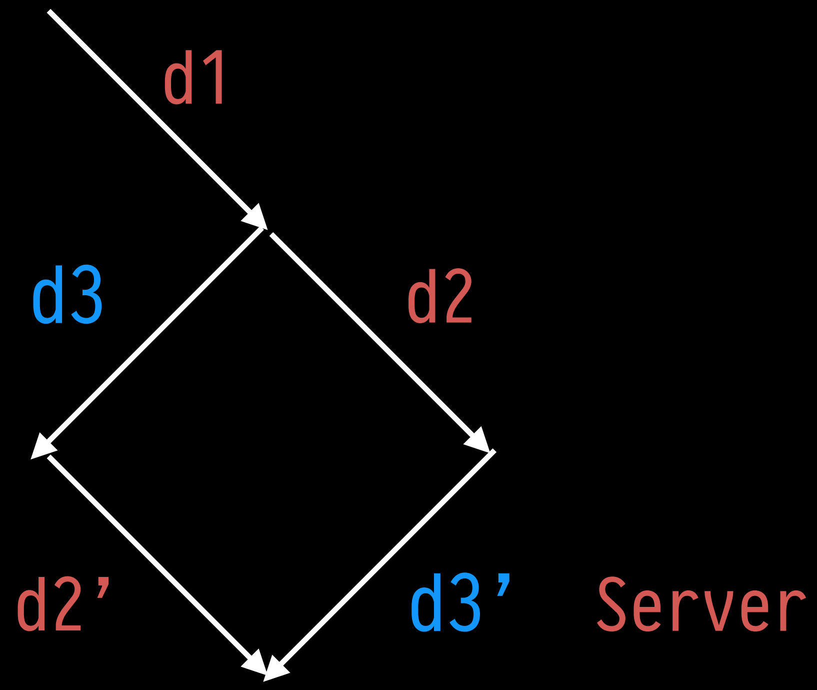


apply d2’ to document

new history: [d3’]

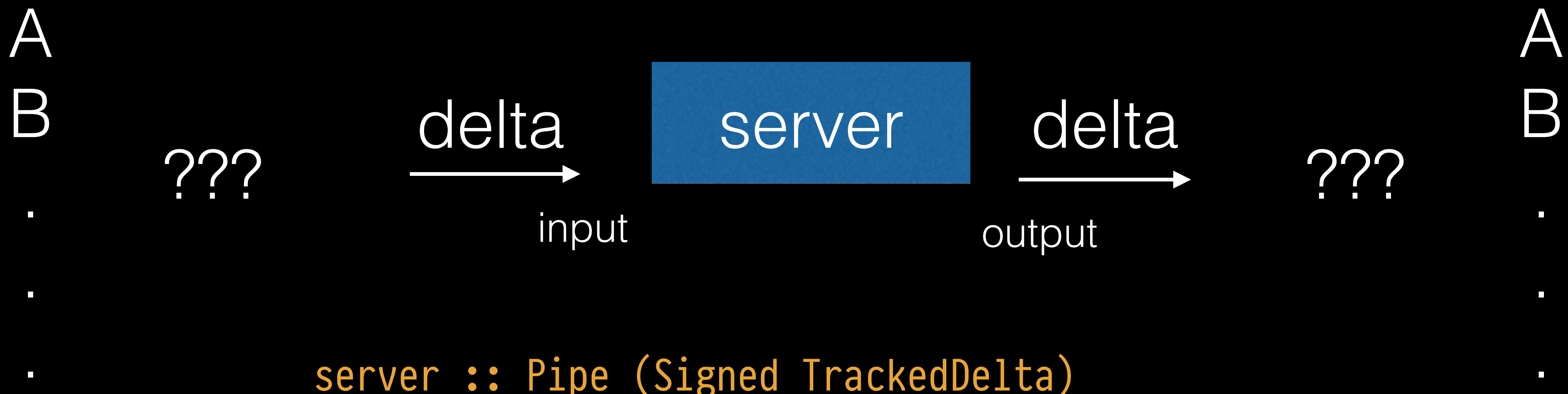
OT





d1 <> d3 <> d2’ == d1 <> d2 <> d3’
by transform property

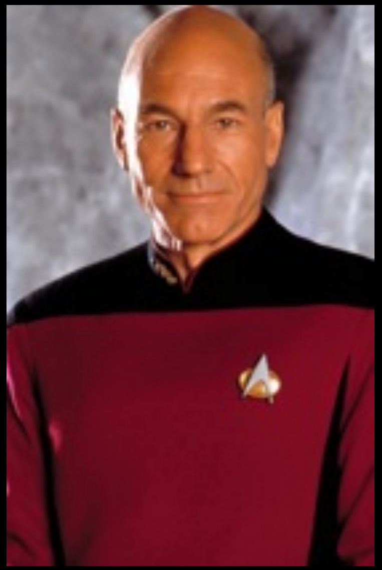
Pipes



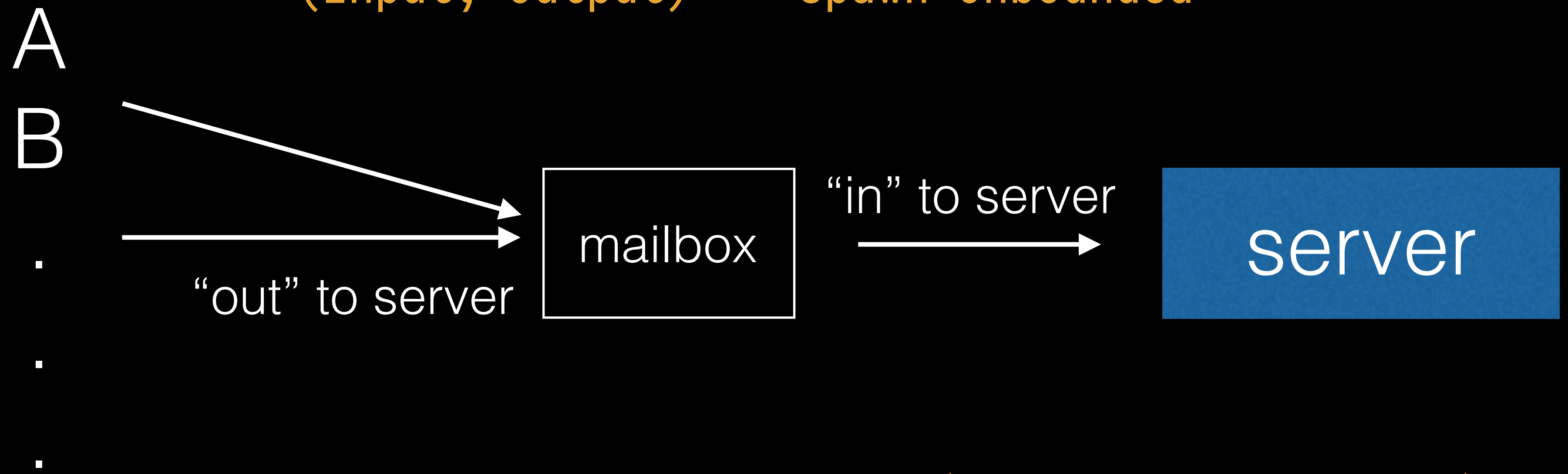
```
server :: Pipe (Signed TrackedDelta)
               (Signed Delta)
               (StateT (Map ClientID [Delta]) m)
```

it’s pure, woo!

Actors!
(more like pi-calculus)



```
spawn :: Buffer a -> IO (Output a, Input a)

(input, output) <- spawn Unbounded
```

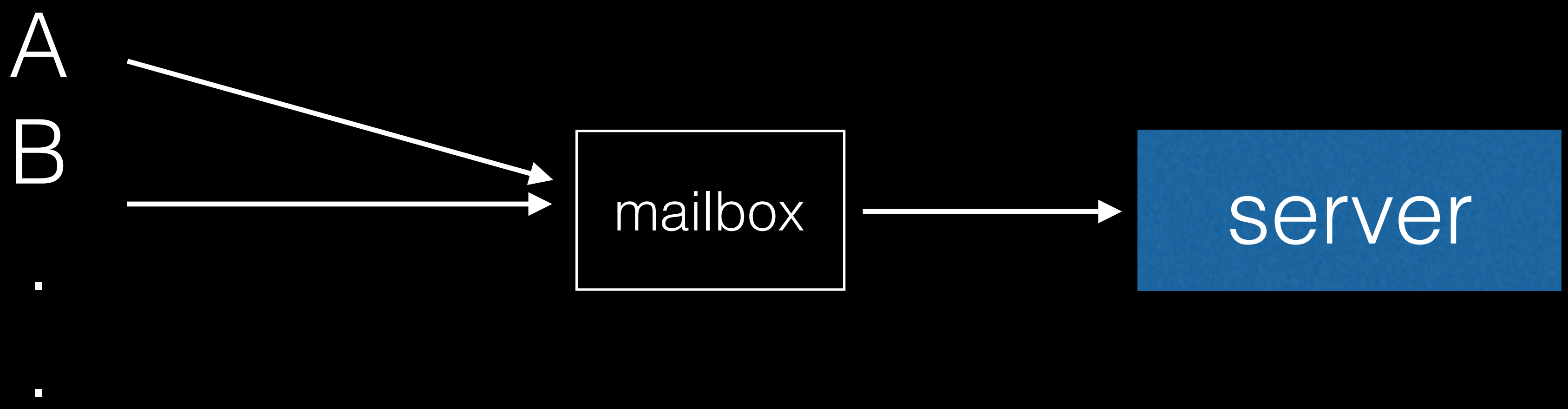


```
fromInput input :: Producer (Signed TrackedDelta)

fromInput input >-> server >-> . . .
```

streaming input, woo!

Actors!

```
spawn :: Buffer a -> IO (Output a, Input a)

(input, output) <- spawn Unbounded
```

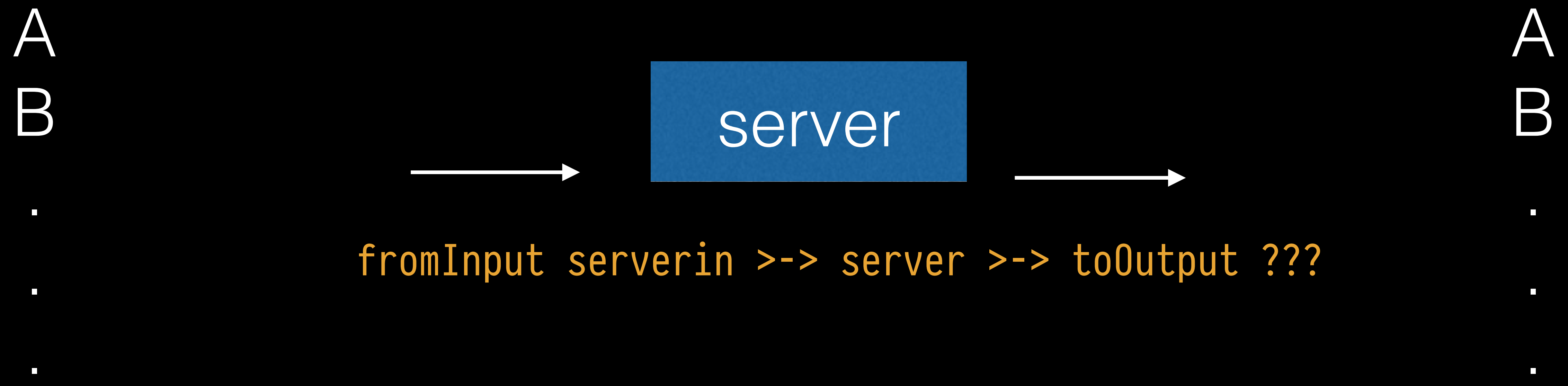


```
reader id tcp :: Producer (Signed TrackedDelta) m ()
reader id tcp =    P.fromHandle tcp
              >-> P.read
              >-> P.map (Signed id)
```

```
reader clientID handle >-> toOutput output
```

Pipes

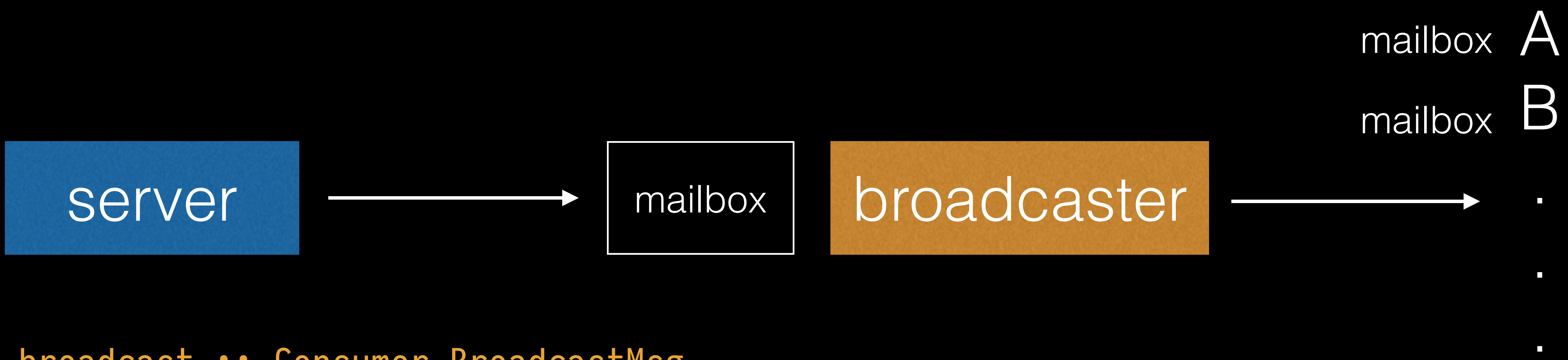
```
    fromHandle
>-> …
>-> toOutput serverout
```



```
fromInput serverin >-> server >-> toOutput ???
```

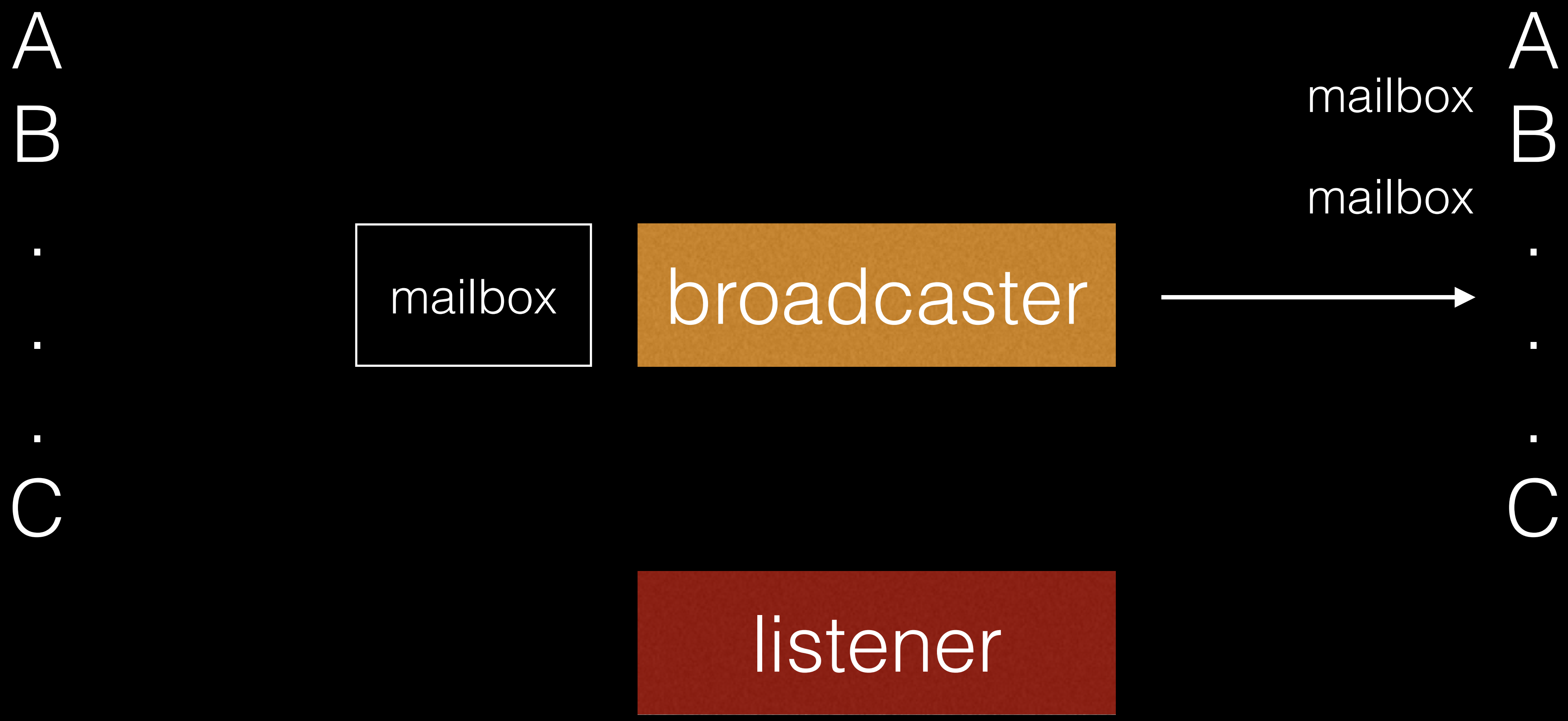
Pipes

```
writer = P.toHandle tcpHandle
```



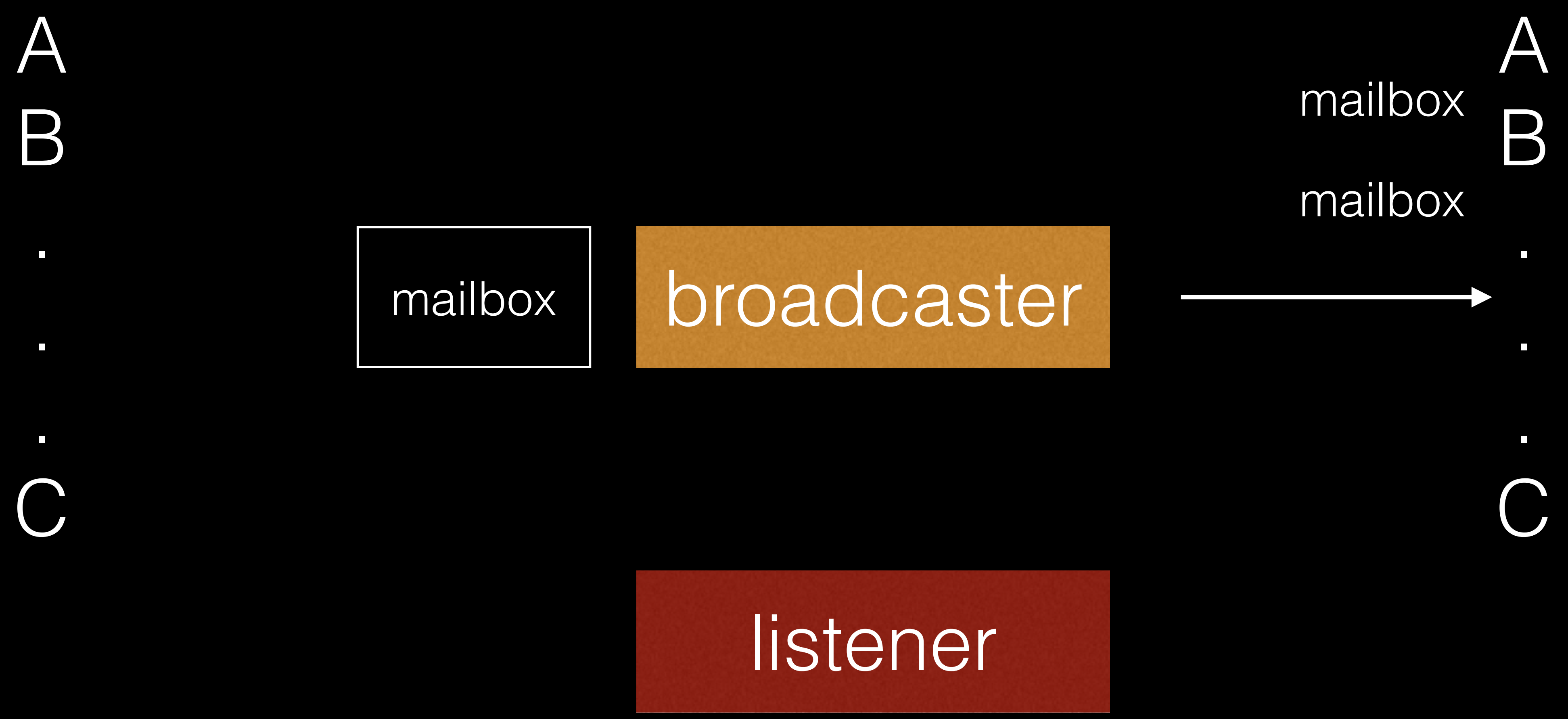
```
broadcast :: Consumer BroadcastMsg
                      (StateT (Map ClientID
                                   ( Int
                                     ^ no. deltas client reported to have seen
                                   , [Output TrackedDelta]
                                     ^ client writer’s mailbox)
                              m)
```

Pipes



listen for new clients
start a reader and writer

Pipes



A
B
.
.
.
C

mailbox

broadcaster

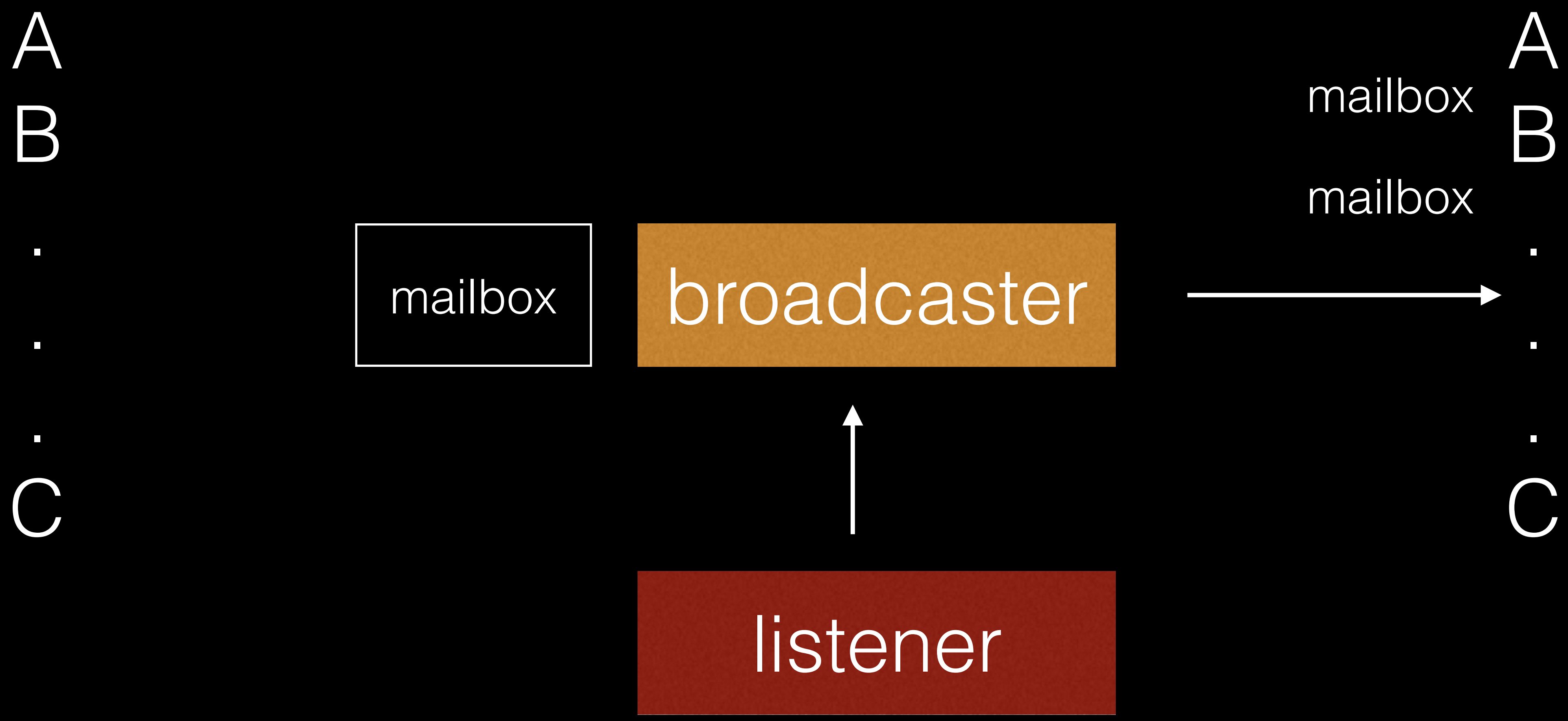
mailbox
mailbox

A
B
.
.
.
C

listener

```
(writerout, writerin, writerseal) <- spawn’ Unbounded
writerid <- forkIO $ catch (runEffect $ fromInput writerin >-> writer h)
                           (\Disconnect -> atomically $ writerseal)
forkIO $ do runEffect $ reader cid h >-> toOutput readerout
            throwTo writerid Disconnect
return $ Client cid writerout
```

Pipes



```
listener serverout handle >-> toOutput broadcastout
```

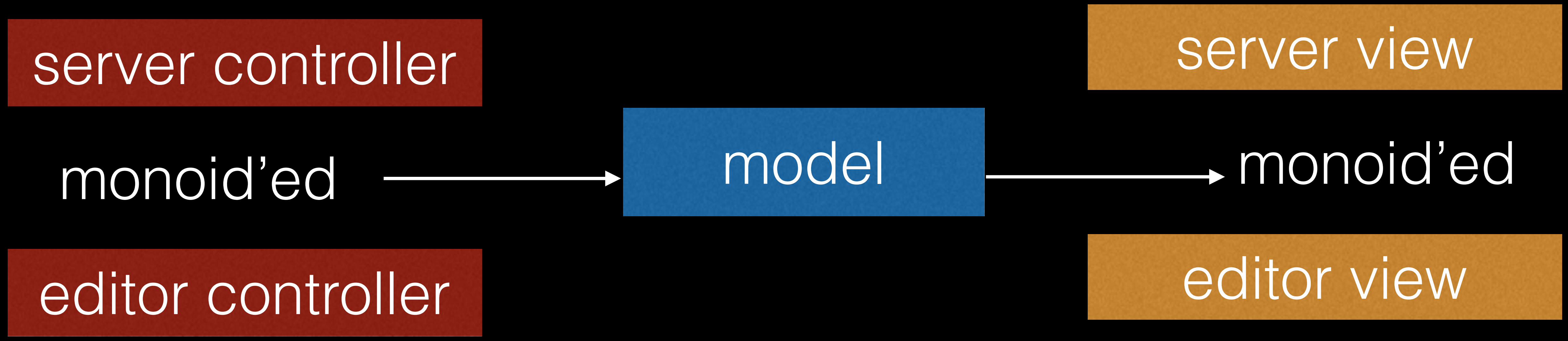
Client

model

```
Model ( Int       – no of ops to tell server to drop from history
      , [Delta])  - ops sent, not confirmed by server
      From To
```

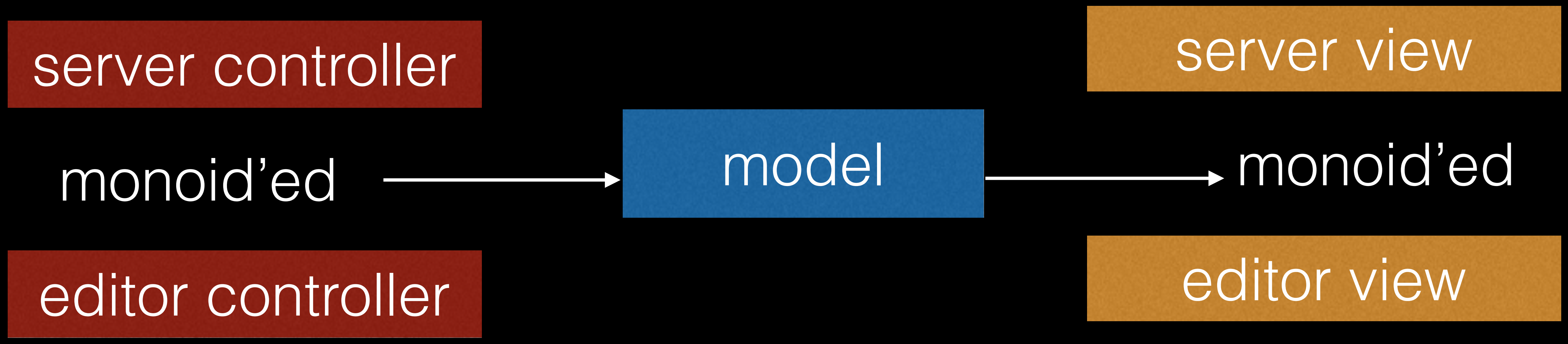
```
data From = FServer TrackedDelta    – op from server
          | FUser   Delta           - op from editor

data To   = TServer TrackedDelta
          | TUser   Delta
```

MVC (no not that one)



```
model      :: Model (Int, [Delta]) From To
controller :: Managed (Controller From)
view       :: Managed (View To)
```

MVC (no not that one)



server controller

monoid'ed

editor controller

model

server view

monoid'ed

editor view

```
controller tcp ws
=  fmap (fmap FServer) (controllerServer tcp)
<> fmap (fmap FUser)   (controllerUser   ws)

controllerServer = M.producer Single (P.fromHandle tcp >-> P.read)
^ deltas from server

controllerUser   = M.producer Single (void $ L.view P.decoded $ fromWS ws)
^ deltas from JSON from the js-based editor (yuck)
```

MVC bug

```
controller = M.producer Single $ P.stdinLn
view       = M.consumer $ P.stdoutLn
```

```
a
b
<a>
c
<b>
```

So

- pipes are strongly principled abstractions
- good ecosystem
- mvc getting there
- Gabriel is great