Event-driven Programming in Clojure

Zach Tellman
Runa
@ztellman

synchronous is simple

```
def handle_request(request)
  result = query_database(request)
  return generate_response(result)
```

- easy to read and reason about
- errors can be handled at any scope

asynchronous is complex

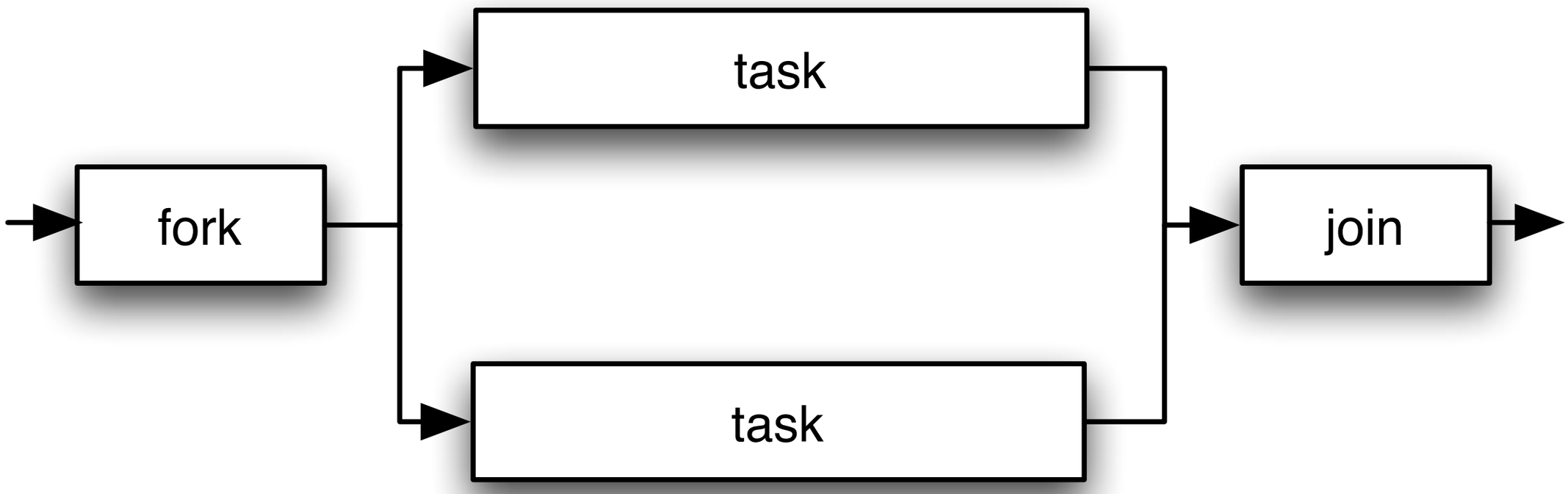
```
def handle_request(request)
  query_database(request,
    error => {
      handle_error(error)
    },
    result => {
      response = generate_response(result)
      send_response(response)
    })
```

- difficult to read and reason about
- errors must be handled locally

the importance of being concurrent

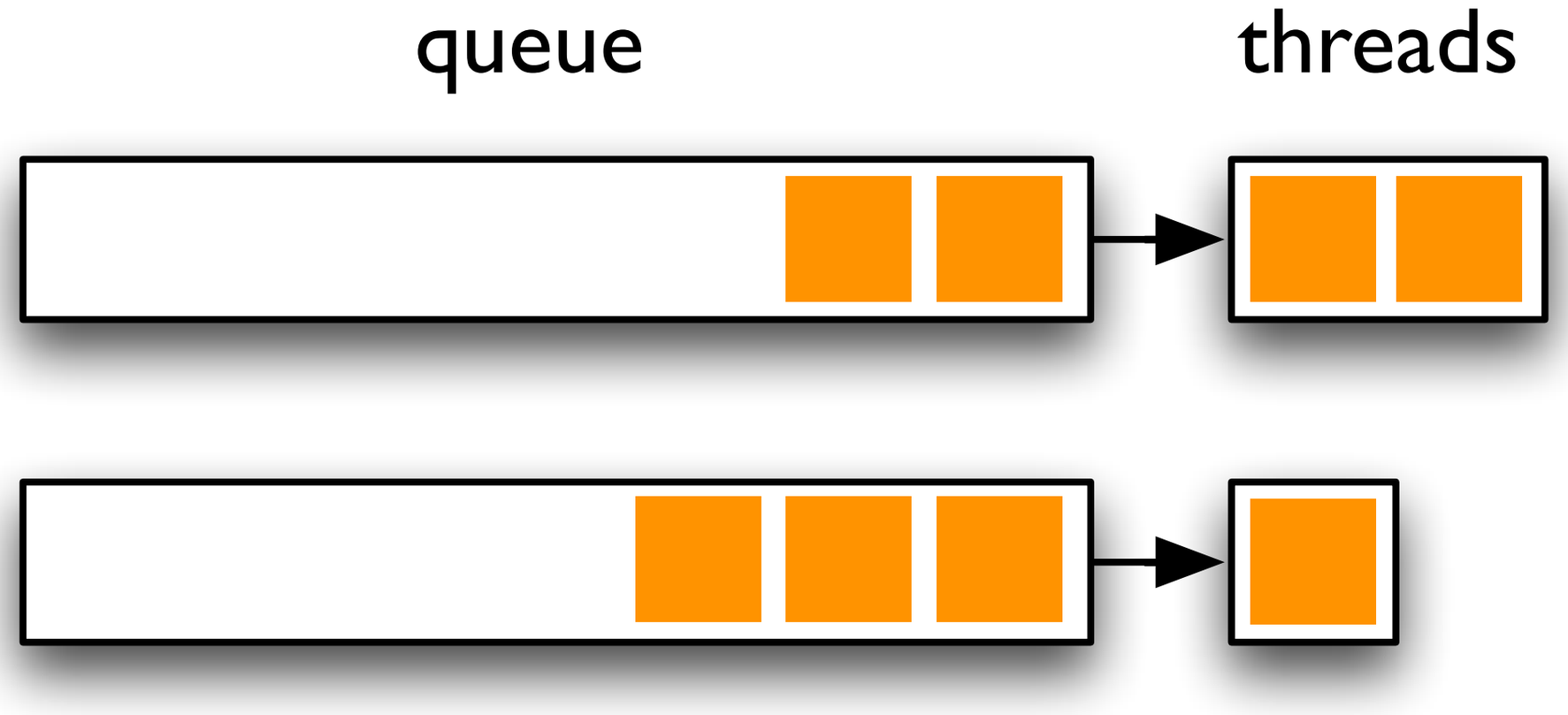
- we often spend more time **waiting** on data than we do **computing** our response
- we can respond more quickly if we wait for multiple things at once

synchronous concurrency

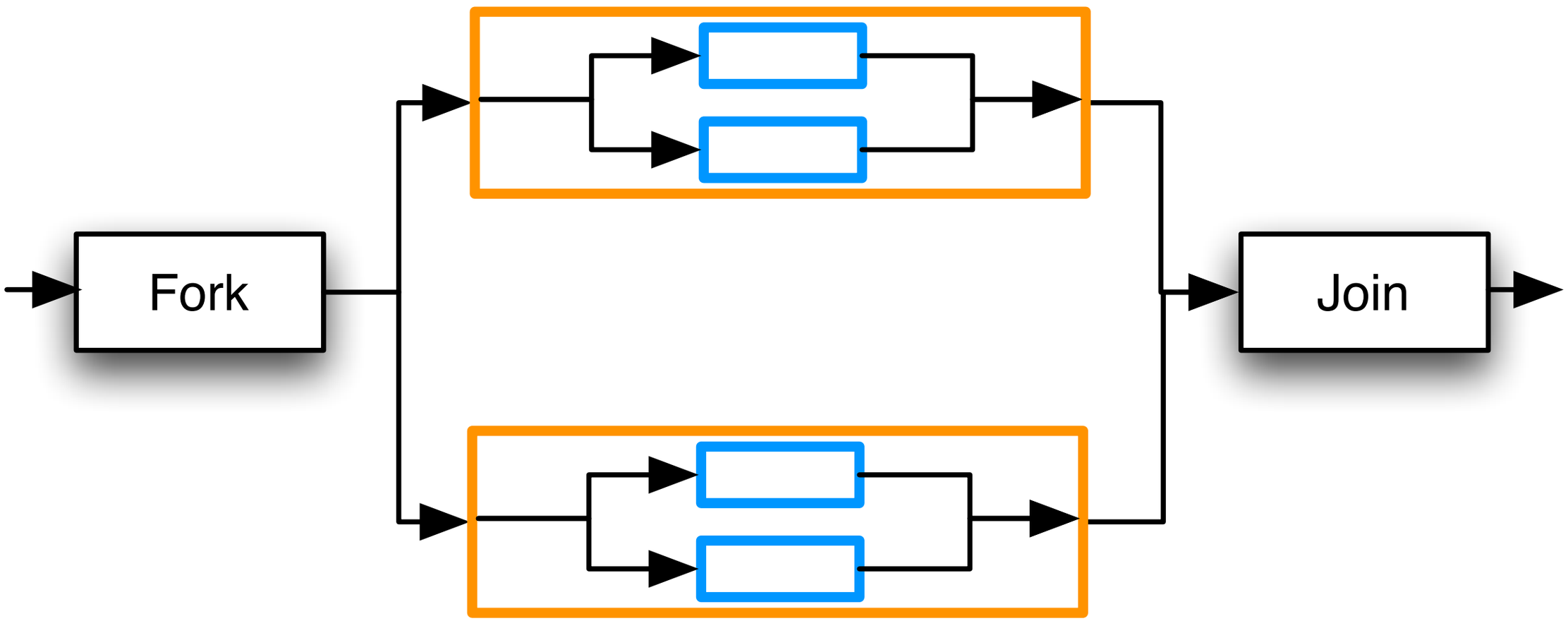


- still pretty simple
- requires one thread per task, plus the original thread

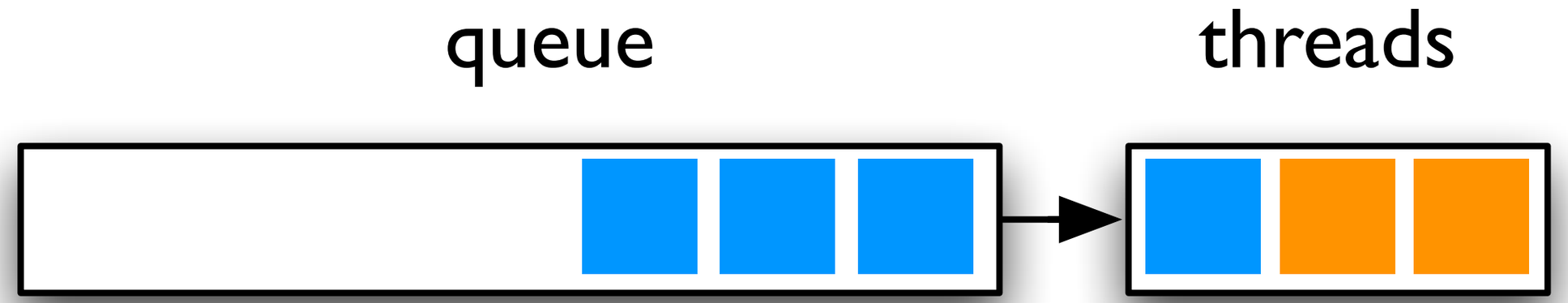
but threads don't grow on trees



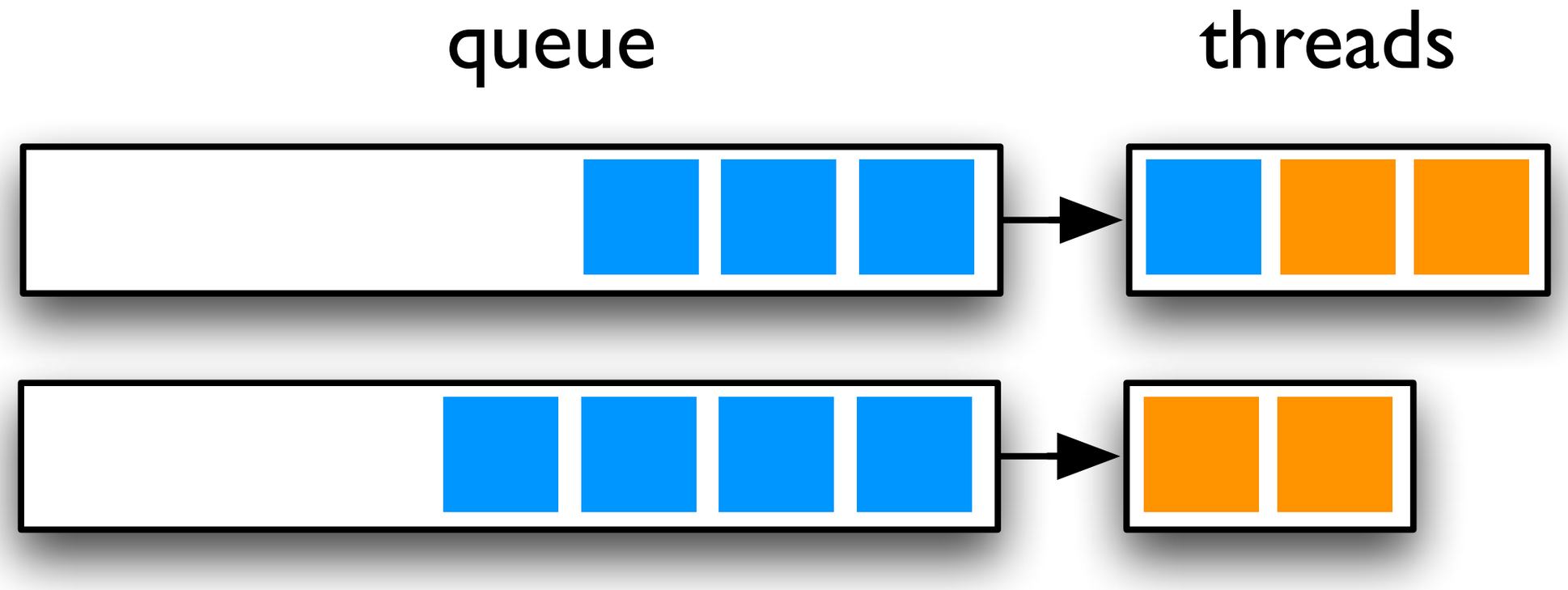
nested concurrency



two tasks, one pool



two tasks, one pool



deadlock!

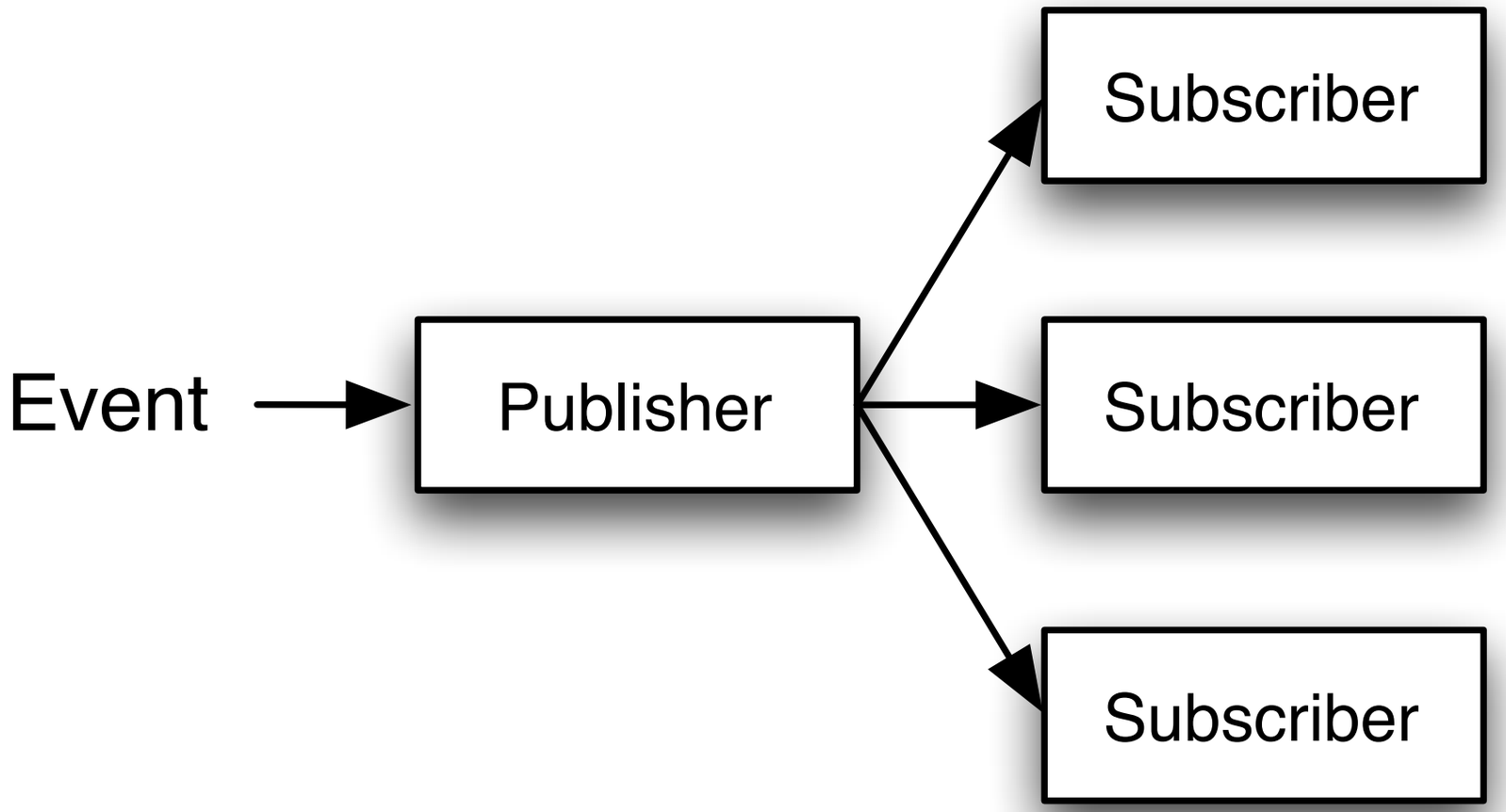
asynchronous exposes an inherent complexity

- everything is concurrent by default
- a less leaky abstraction (for highly coupled concurrent problems)

two great tastes

- these are not mutually exclusive approaches
- interop should be easy
- we want to simplify the asynchronous approach as much as possible

publisher/subscriber



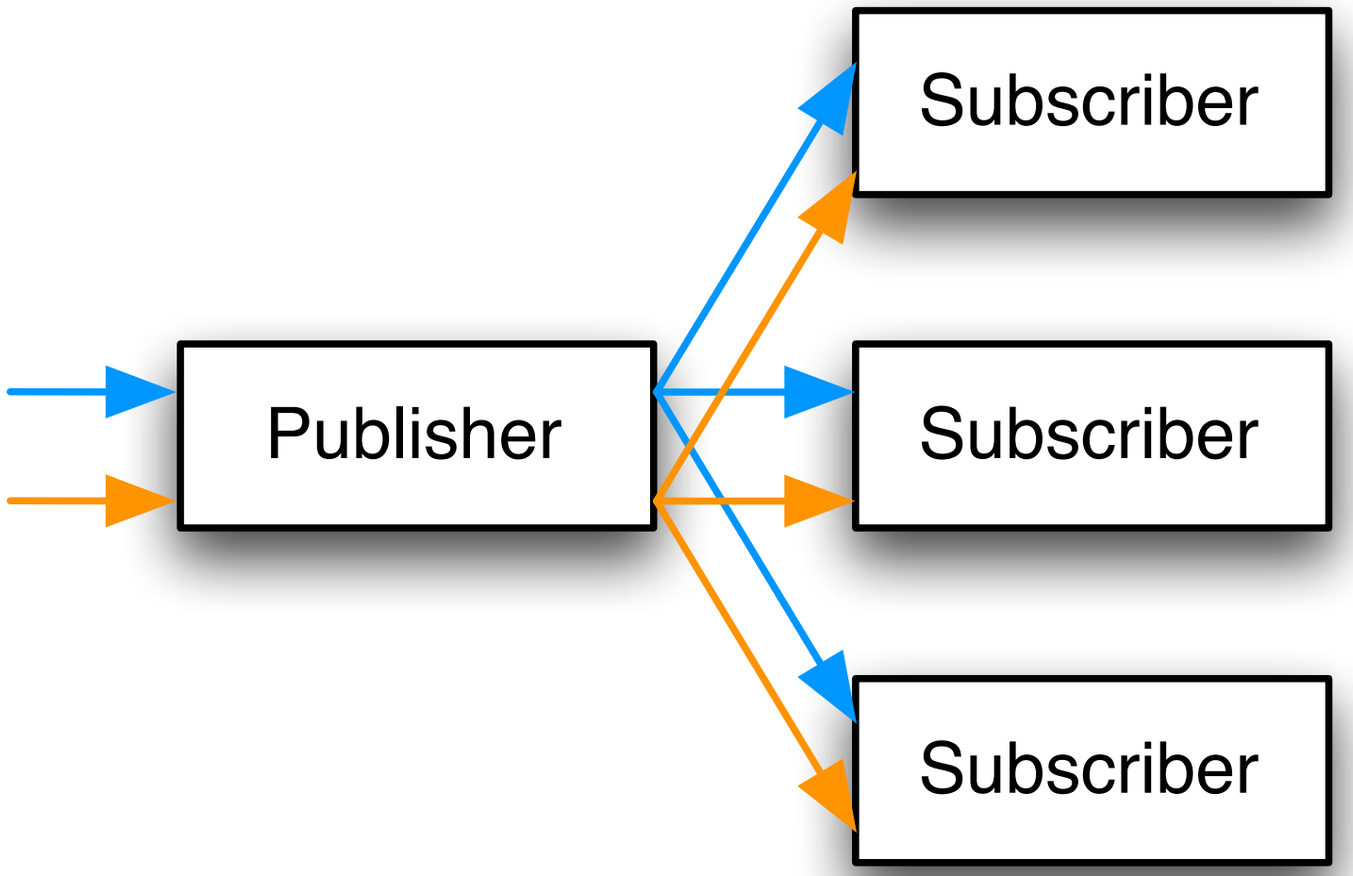
a naive implementation

```
subscribers = [ ]

def subscribe(callback)
  subscribers.add(callback)

def publish(event)
  foreach s in subscribers
    s(event)
```

stop the presses



The same callback can be executed on different threads with different data

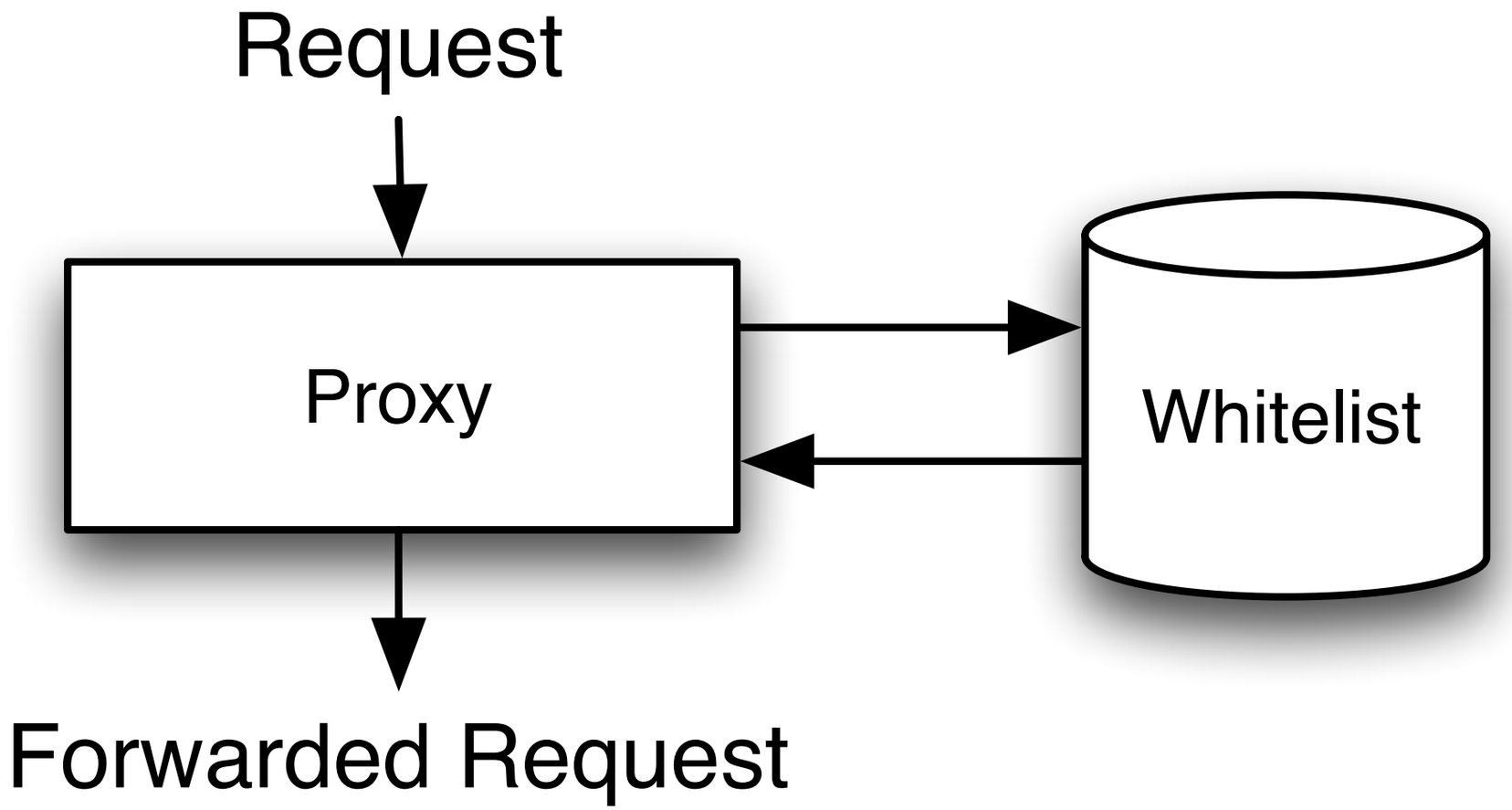
ordered vs. unordered

- mouse clicks and keystrokes need to be handled one at a time and in-order
- logging we can be less fussy about

a software koan

If an event has no subscribers, does it have any effect?

a simple example



a simple example

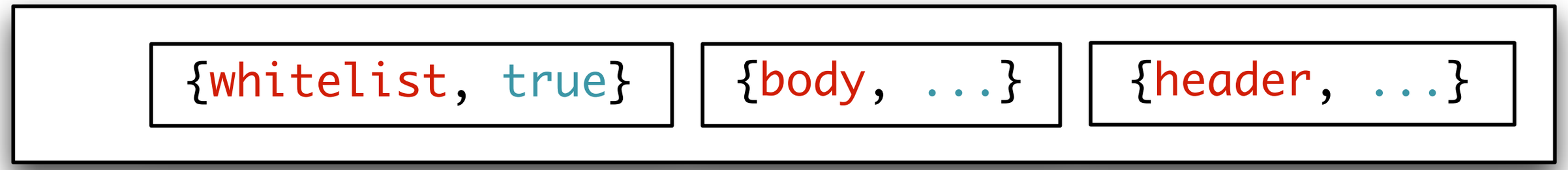
```
def handle_request(request)
  if whitelisted?(request.url)
    forward_request(request)
  else
    forbid_request(request)
```

the node.js approach

- receive headers
- send whitelist request, subscribe to response
- if it's whitelisted, forward the headers and subscribe to the body
- but what if the body arrived first?

the erlang approach

mailbox




```
receive {header, Header}

receive {whitelist, Response}

receive {body, Body}
```

push vs. pull

- the pub/sub model is **push**
- the erlang mailbox model is **pull**
- just because we get an event doesn't mean we know what to do with it

the story so far

- event streams can be **ordered** or **unordered**
- events can be published using **push** or consumed using **pull**

events in clojure

we need to consider:

- immutable data structures
- software transactional memory
- thread-agnosticism

immutability

all that, and a bag of chips

STM

- publishing is a side-effect
- side-effects inside transactions can be hazardous

threads

- Clojure is thread-agnostic
- what happens if an event is published before another thread subscribes?

a wish list

- support for push and pull event consumption
- efficient mechanisms for ordered and unordered event streams
- plays nicely with transactions

channels

- represents a **stream of messages**
- messages are consumed by callbacks
- if there are no callbacks, messages wait in the queue
- queue is transactional

unordered messages

```
(receive-all channel callback)
```

- callback is invoked by thread that enqueues the message
- bypasses the queue entirely

ordered messages

```
(receive-in-order channel callback)

(receive channel callback)
```

pull messages from the queue one at a time

closing channels

```
(close channel)
(on-closed channel callback)
(on-drained channel callback)
```

- when a channel is **closed**, no further messages can be enqueued
- when a closed channel is empty, it is **drained**

channel errors

```
(error! channel exception)
```

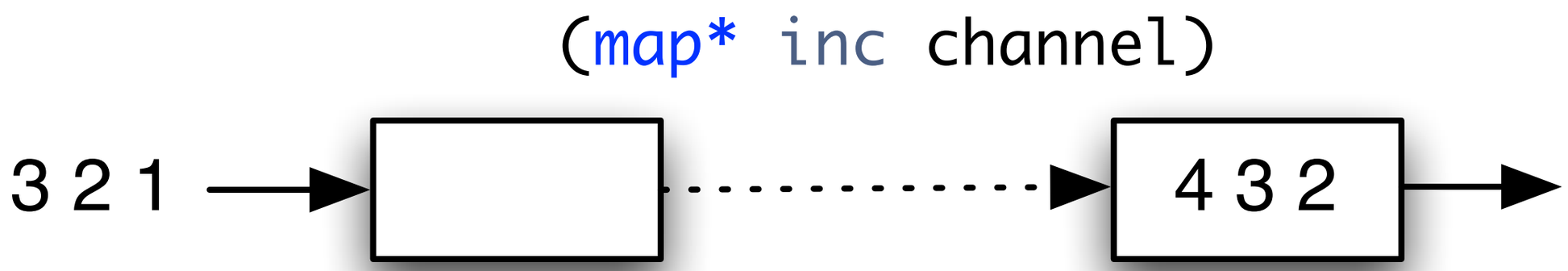
```
(on-error channel callback)
```

errors also close the channel

channel operators

- familiar operators such as `map*`, `filter*`, `reduce*`, `take*`, and `partition*`
- less familiar operators such as `sample-every*`, `partition-every*`, and `siphon`

channel operators

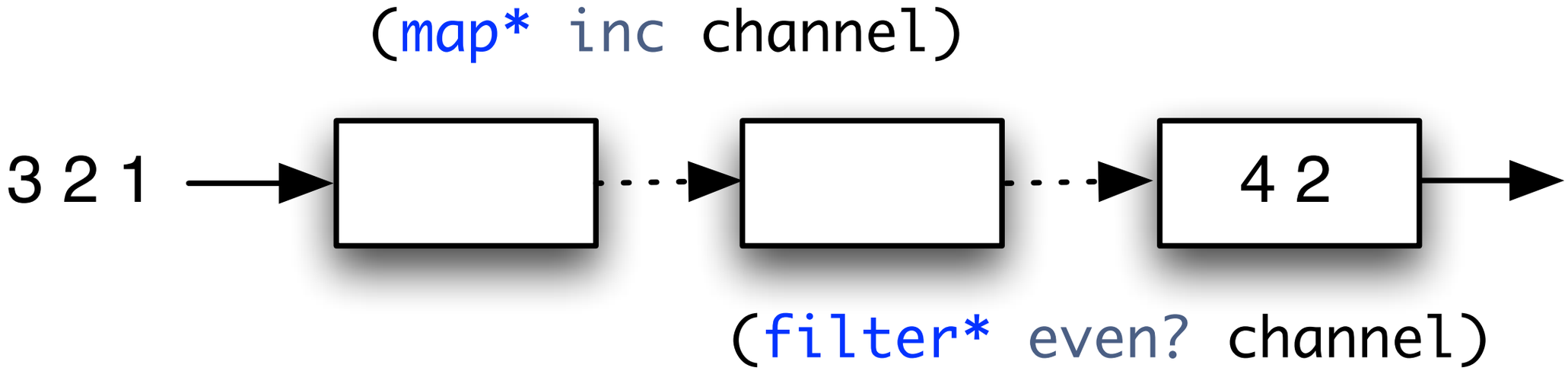


to create a derivative channel, you must consume messages from the source channel

channel operators

```
(->> channel (map* inc) (filter* even?))
```

channel operators



but what happens when we close the right-most channel?

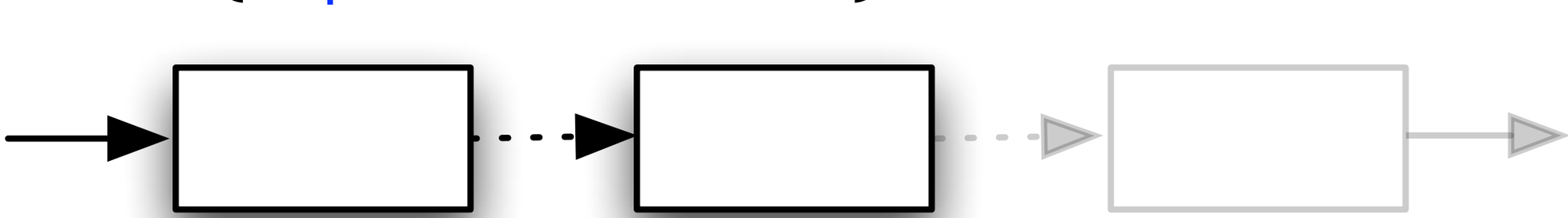
backpropagation

```
(map* inc channel)
```

```
(filter* even? channel)
```

backpropagation

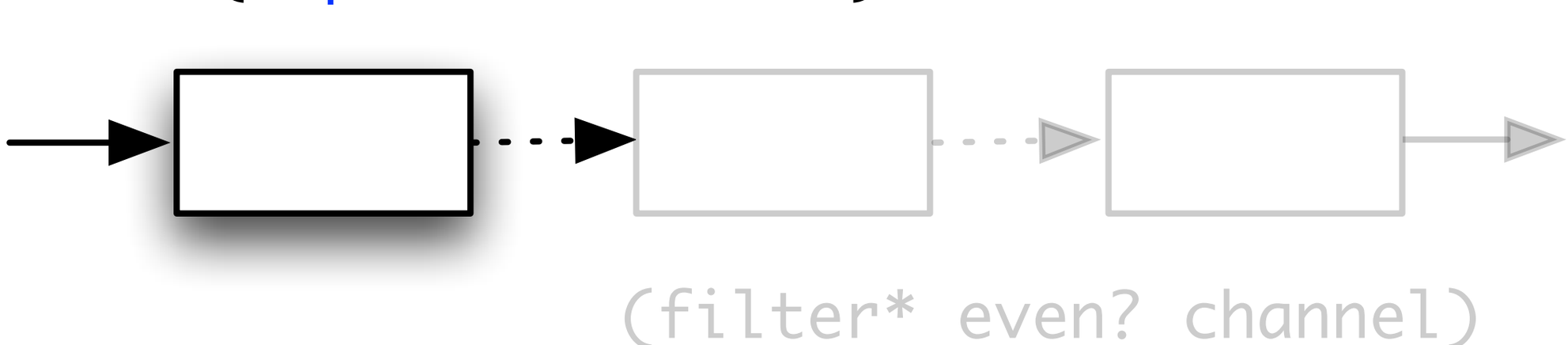
```
(map* inc channel)
```



```
(filter* even? channel)
```

backpropagation

```
(map* inc channel)
```



```
(filter* even? channel)
```

backpropagation

```
(map* inc channel)
```

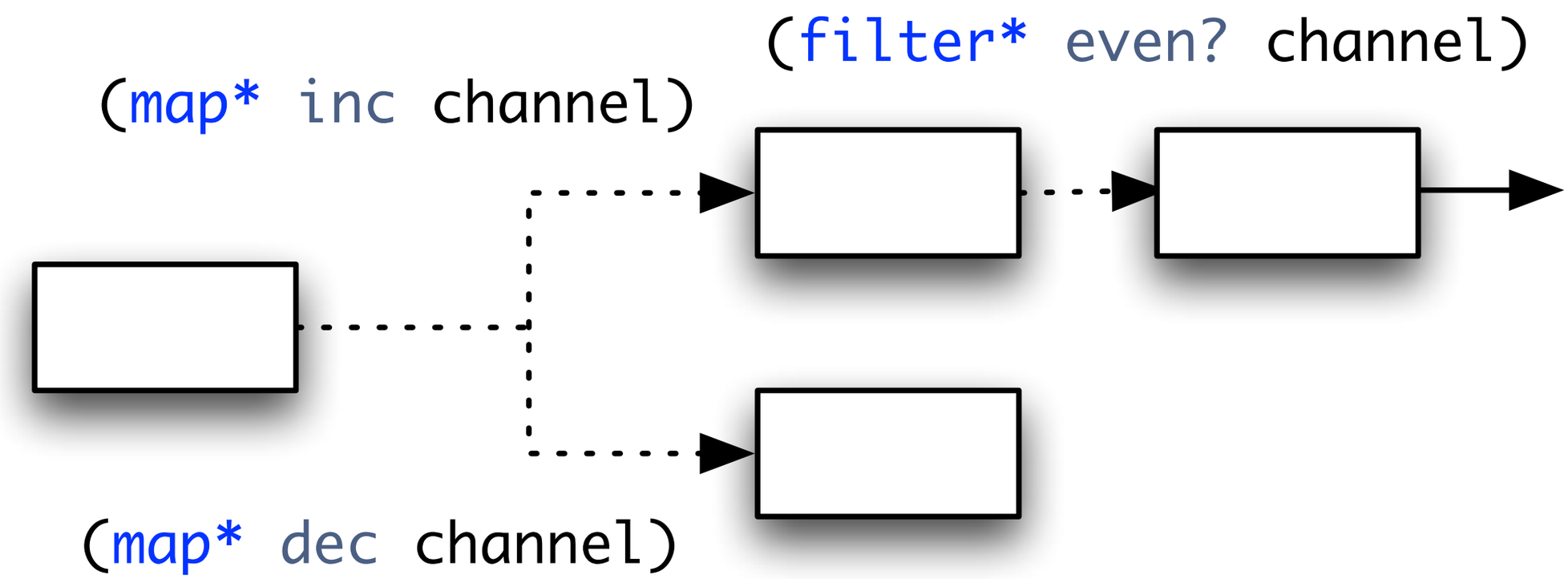
```
(filter* even? channel)
```

backpropagation

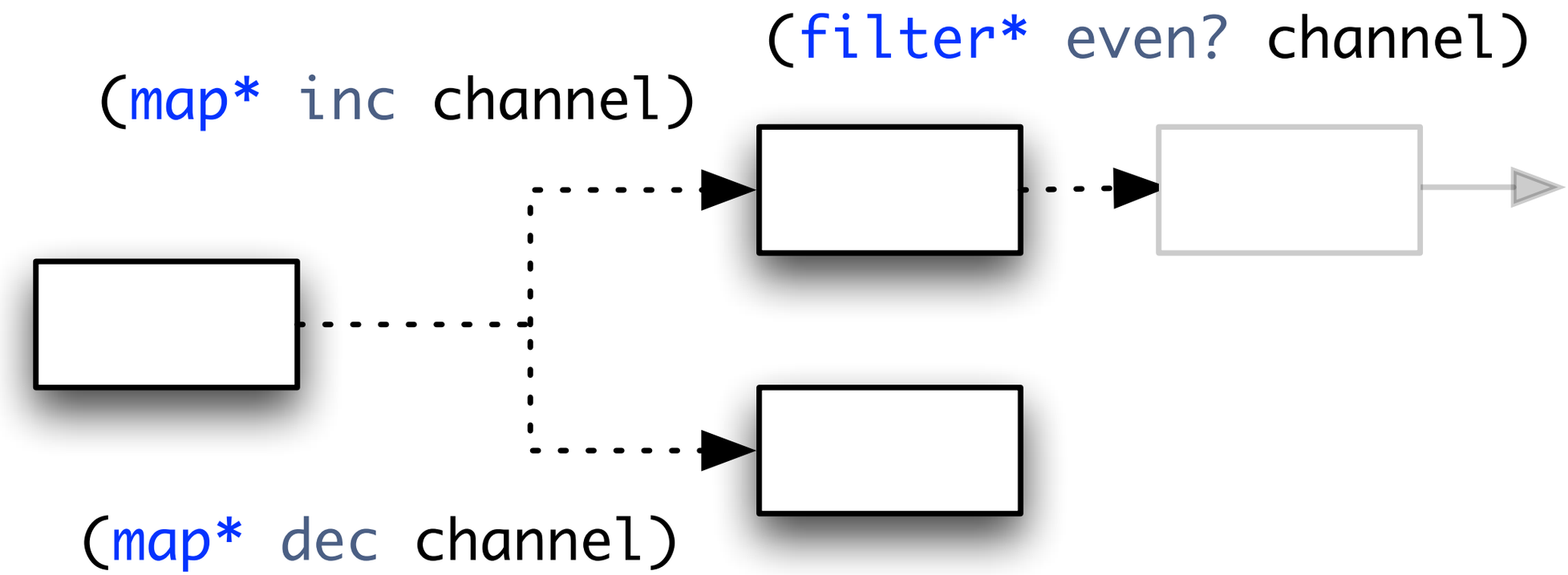
```
(map* inc channel)
```

```
(filter* even? channel)
```

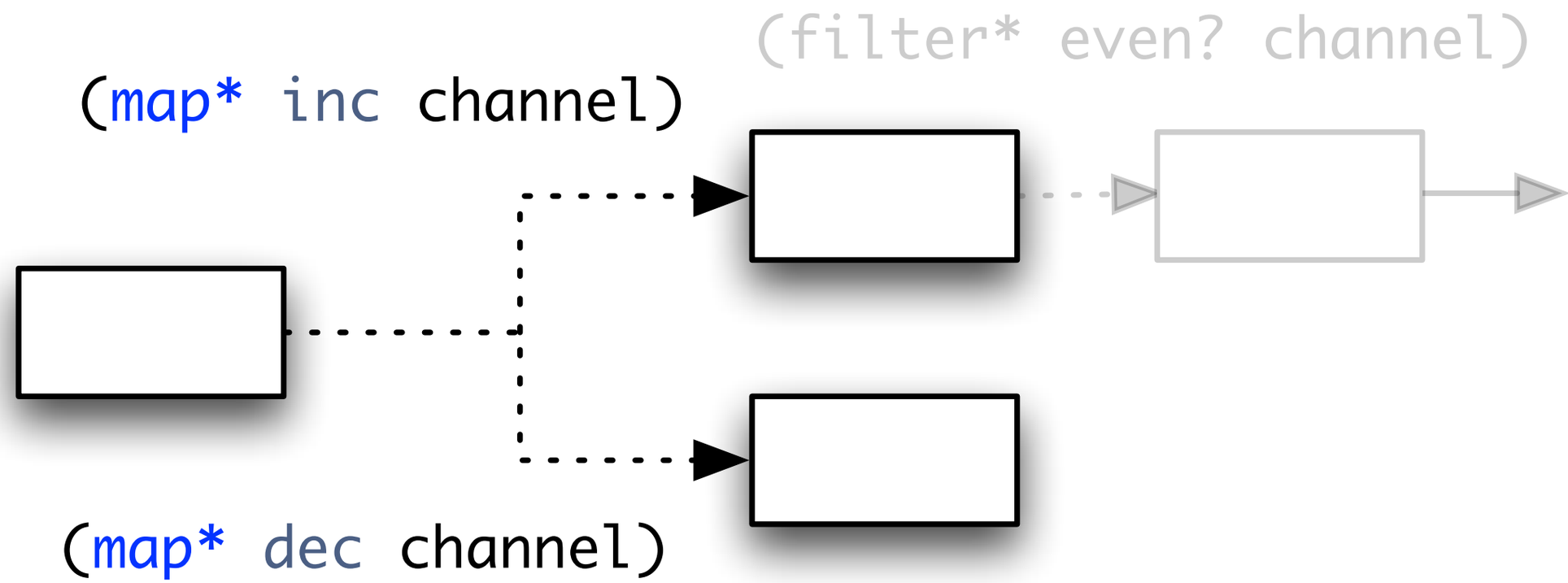
backpropagation



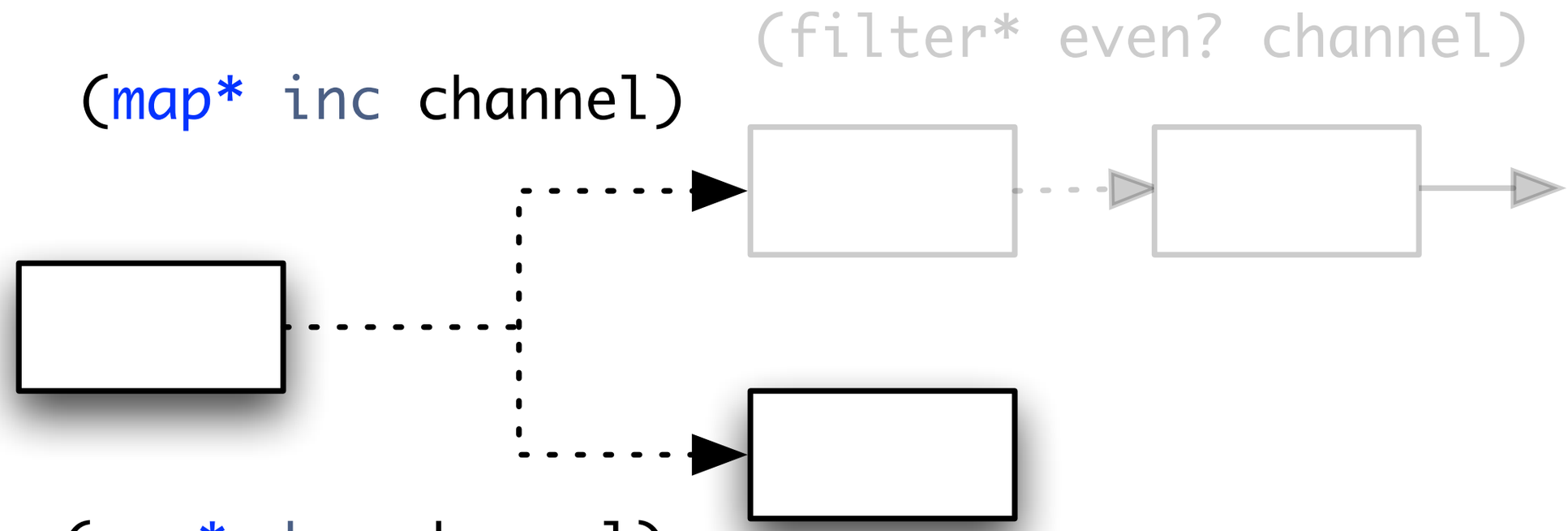
backpropagation



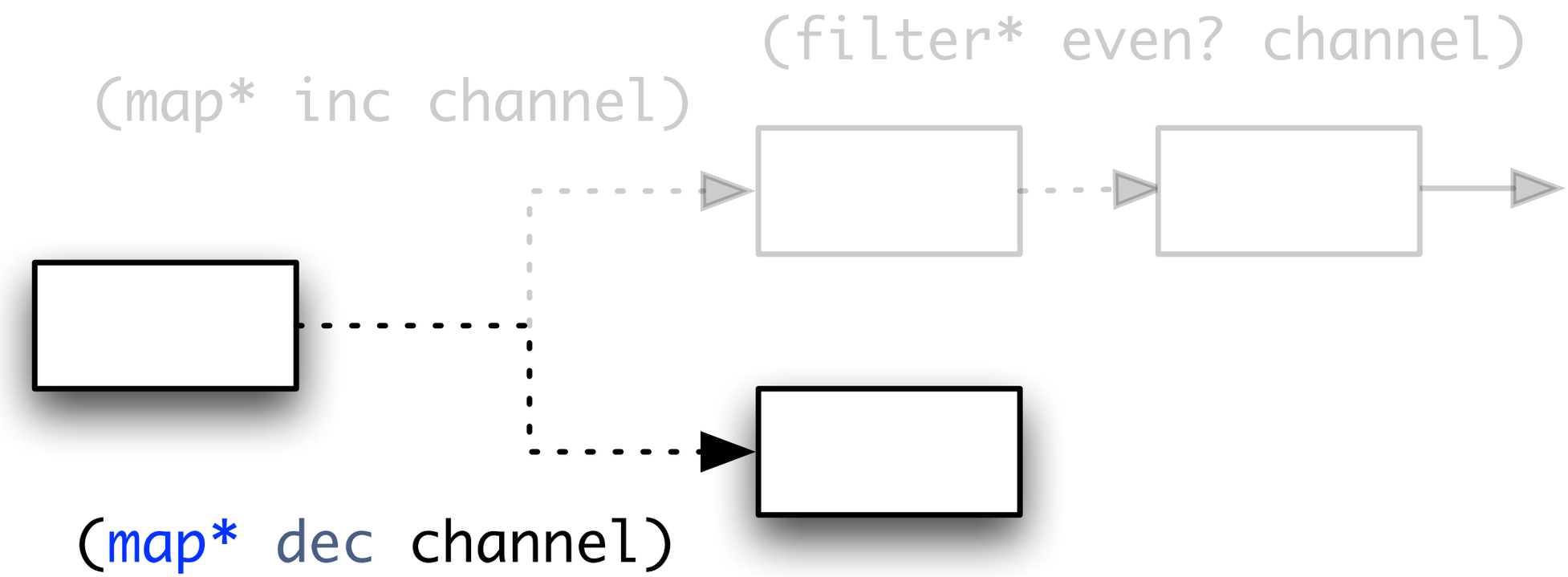
backpropagation



backpropagation



backpropagation



channels and seqs

- in channels, messages exist once they're enqueued
- in lazy-seqs, elements exist once they're consumed

channels and seqs

- in seqs, the same operator will always return the same result (`map` is idempotent)
- in channels, the same operator will not always return the same result (`map*` is not idempotent)

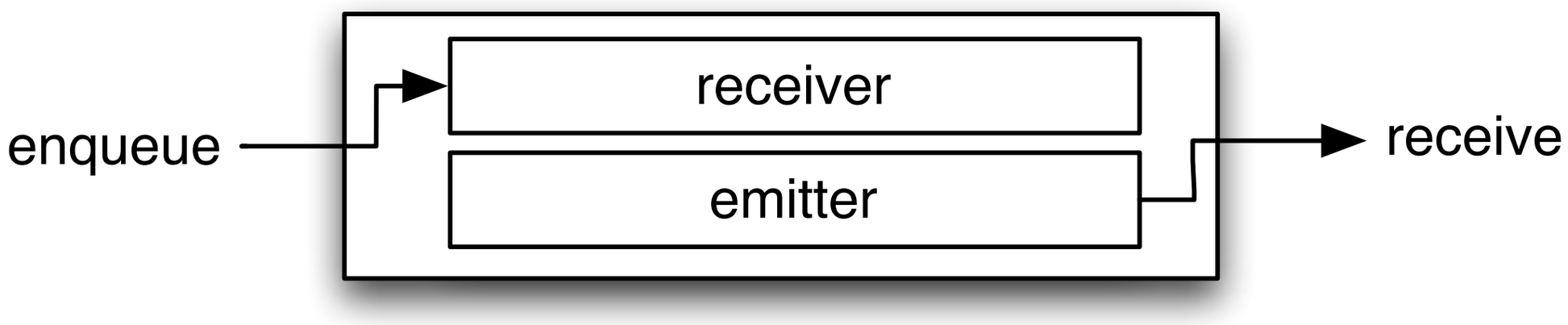
forking channels

```
(fork channel)
```

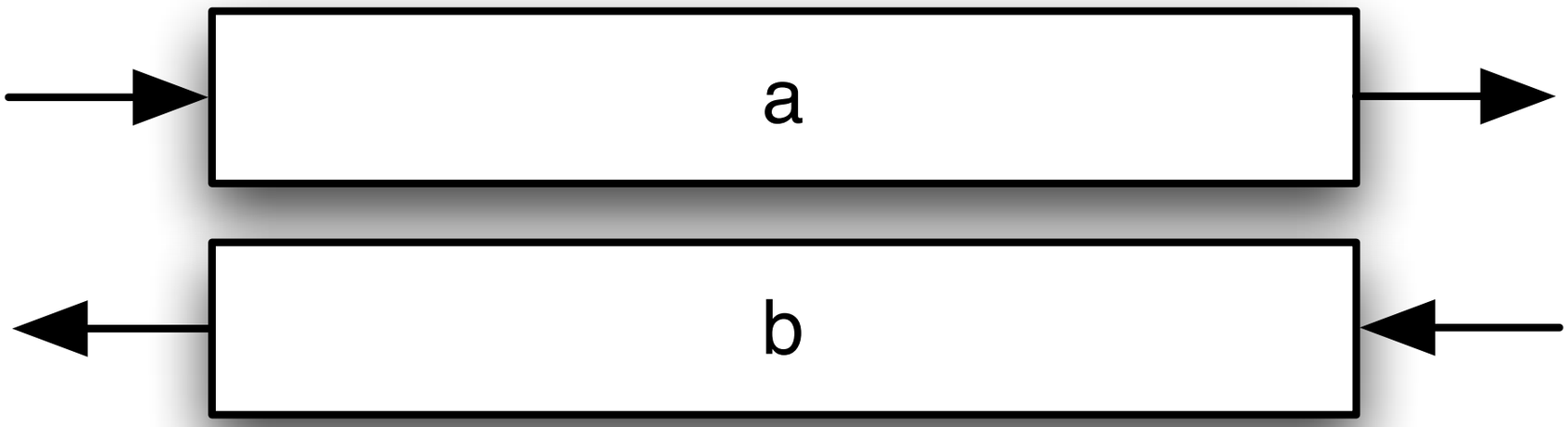
- returns a duplicate channel that can be independently consumed
- closing the original channel will close the forked channel, but not vise-versa

splicing channels

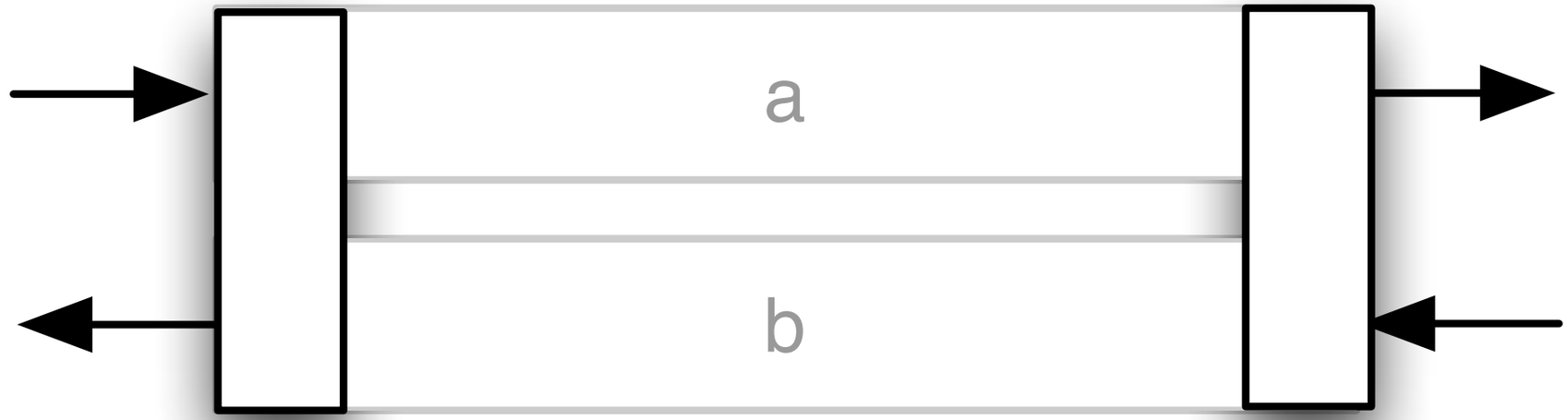
```
(splice emitter receiver)
```



splicing channels



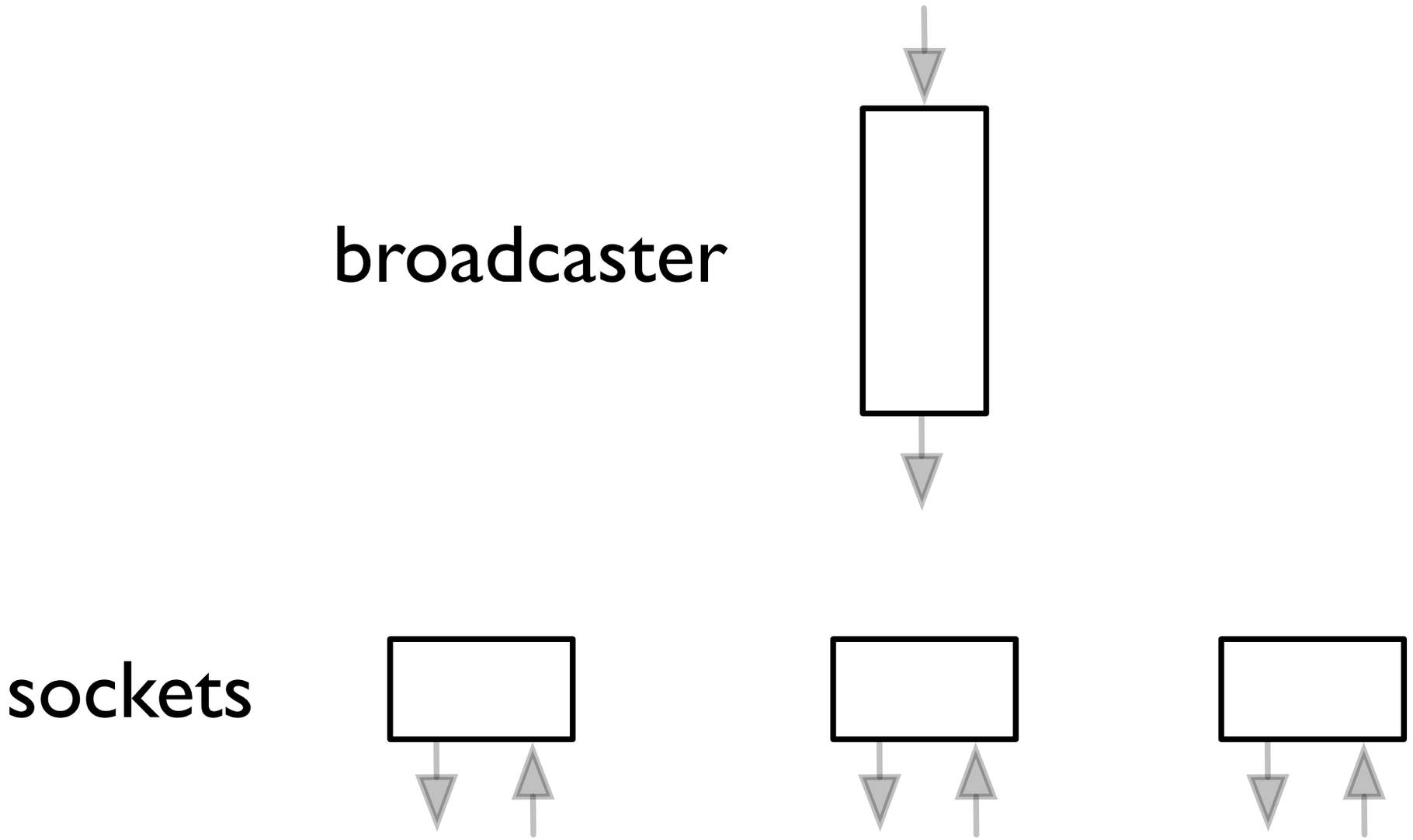
splicing channels




```
(splice a b)
```

```
(splice b a)
```

a simple chat server

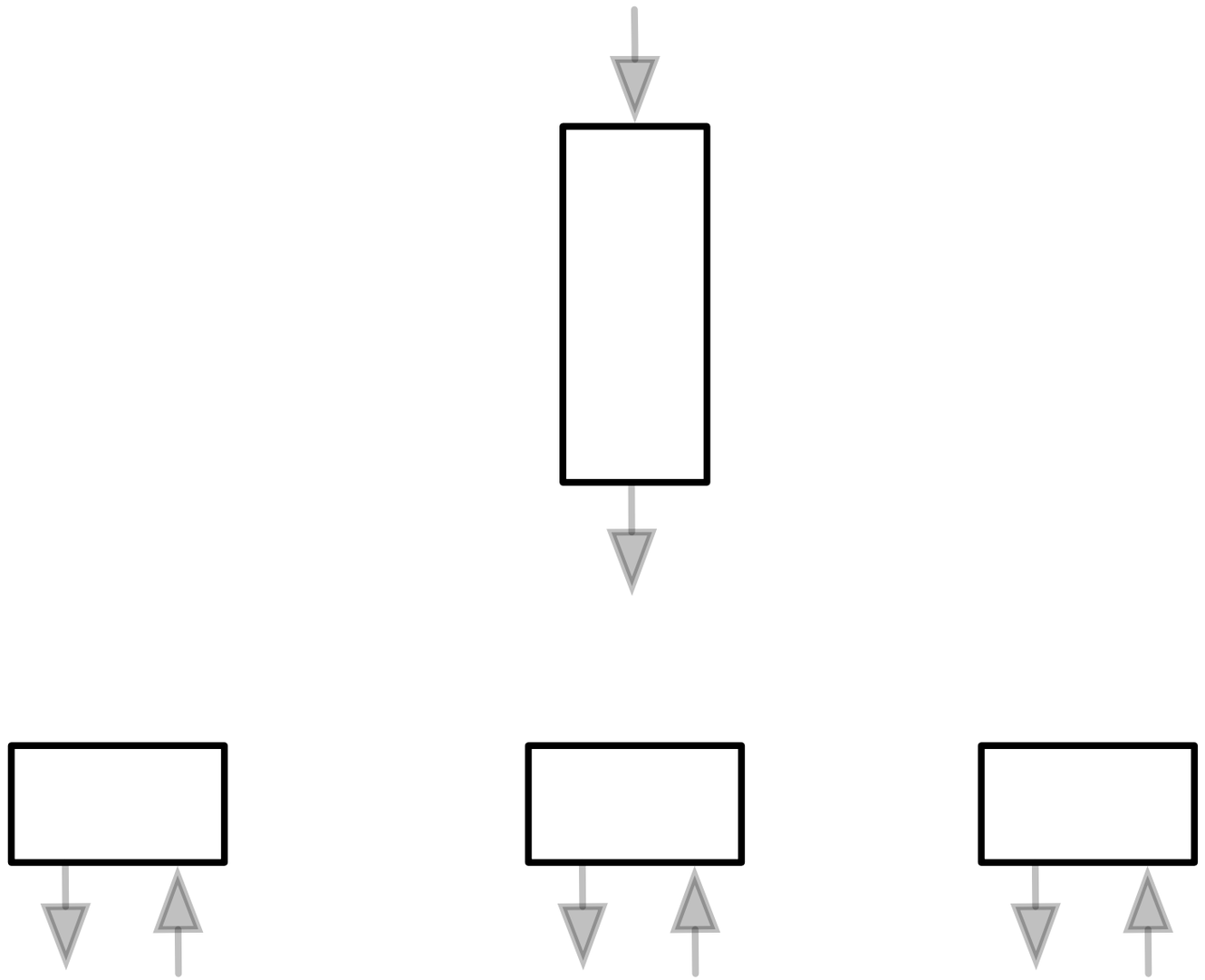


a simple chat server

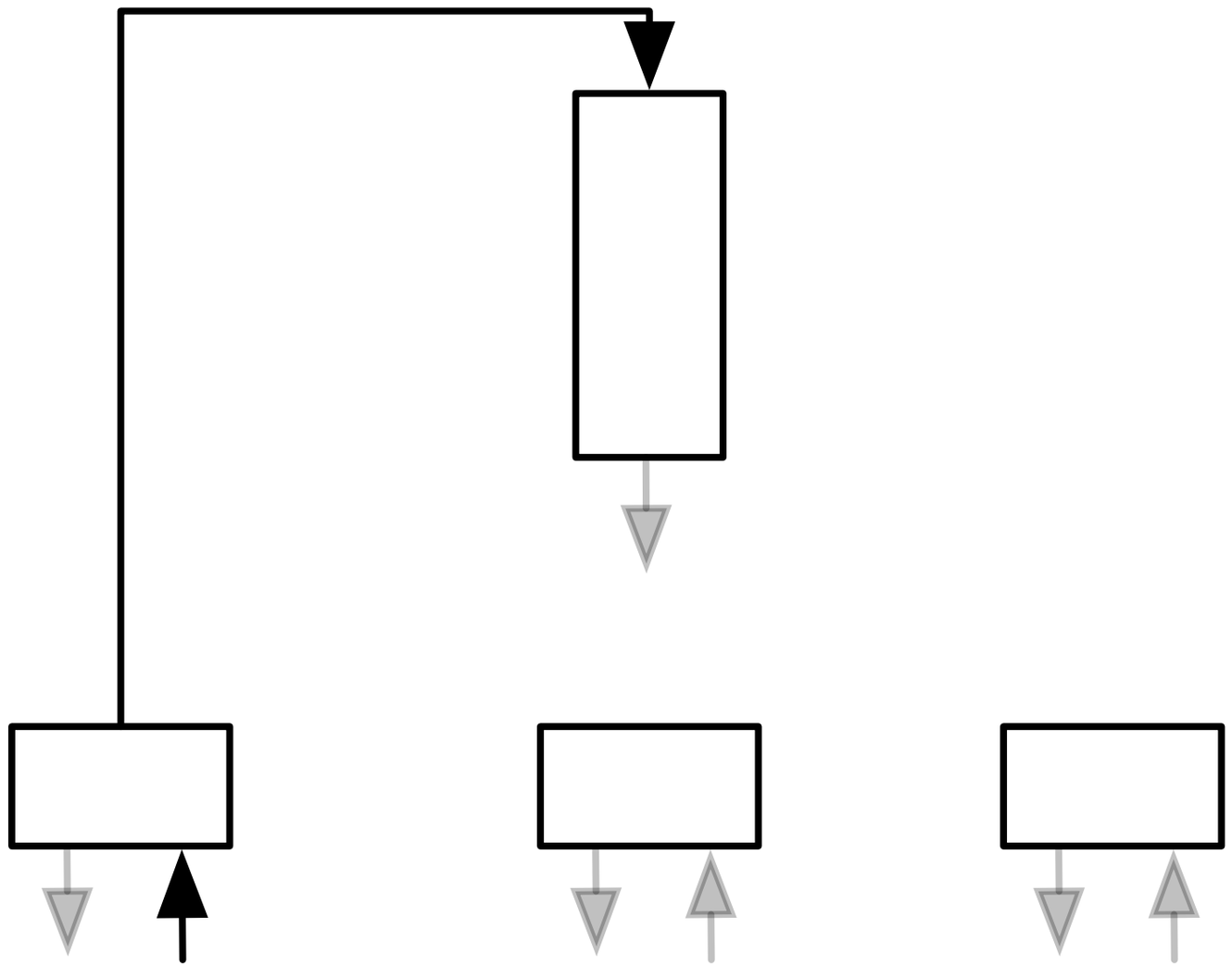
```
(siphon broadcaster socket)

(siphon socket broadcaster)
```

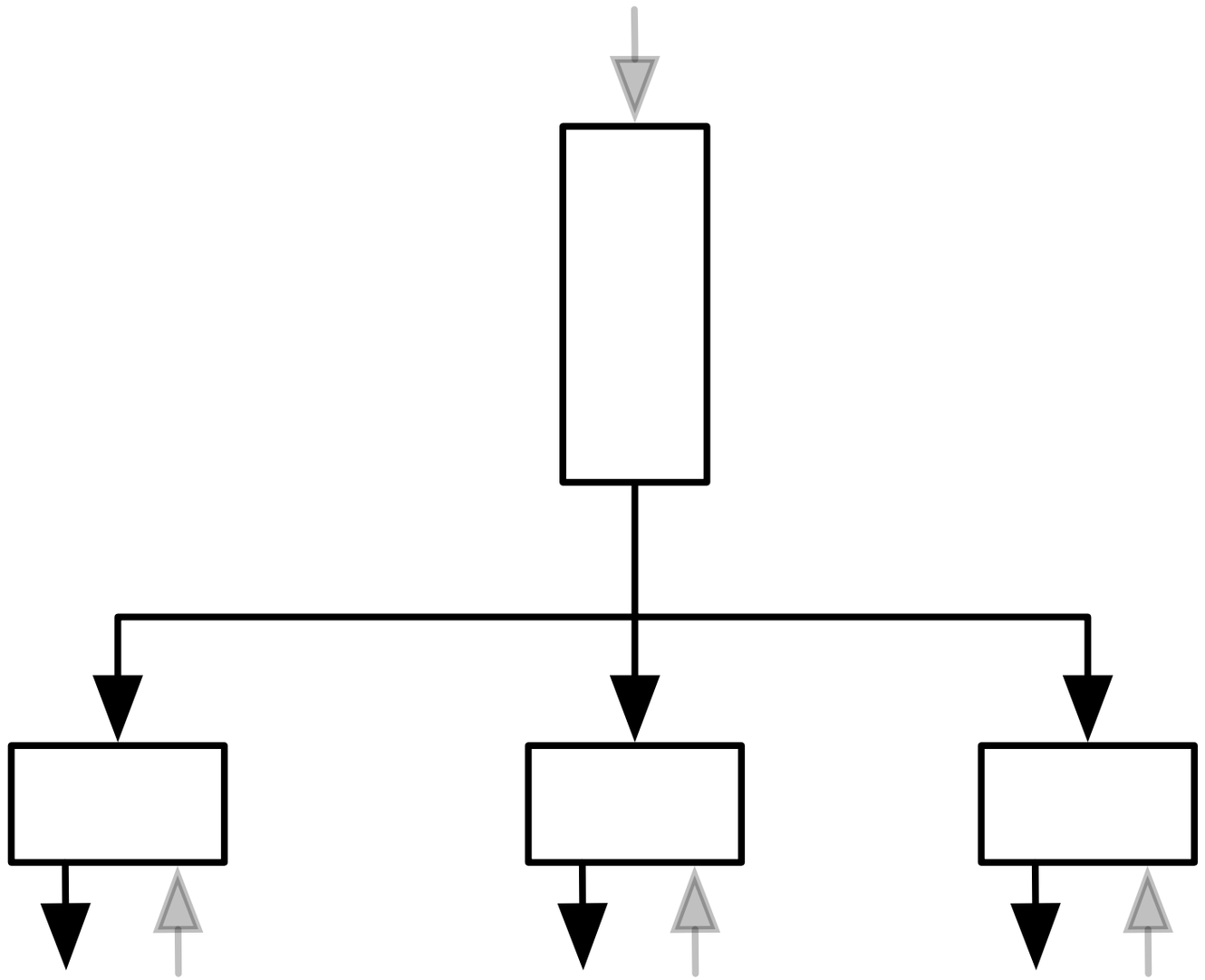
a simple chat server



a simple chat server



a simple chat server



result channels

- represents a single outcome: a **result** or an **error**
- only a single value can be enqueued, but that value can be received multiple times
- this is our bridge to the synchronous world

synchronous results

```
(wait-for-result result-channel timeout)

@result-channel
```

if the result-channel emits an error, synchronous accessors will throw an exception

asynchronous contamination

returns a sum:

```
(+ 2 @(request-a-number))
```

asynchronous contamination

returns a sum:

```
(+ 2 @(request-a-number))
```

returns an eventual sum:

```
(async (+ 2 (request-a-number))
```

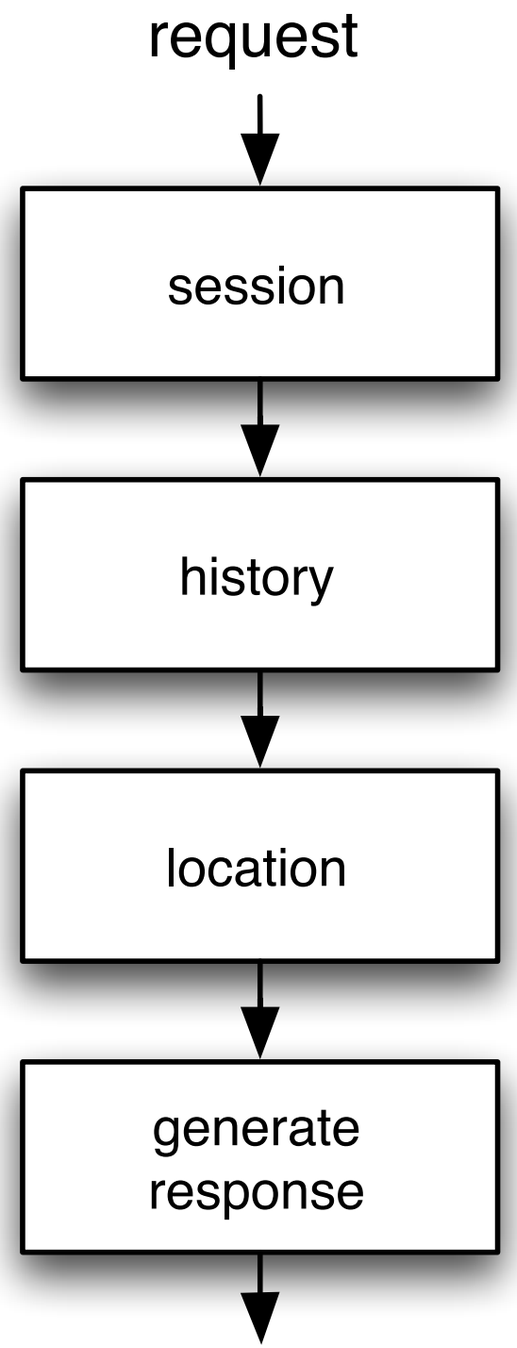
a request workflow

- receive a request
- fetch the **session** data
- if the session doesn't have the **user history**, fetch it
- if the session doesn't have the **user location**, fetch it
- generate a response based on the request, session, history, and location

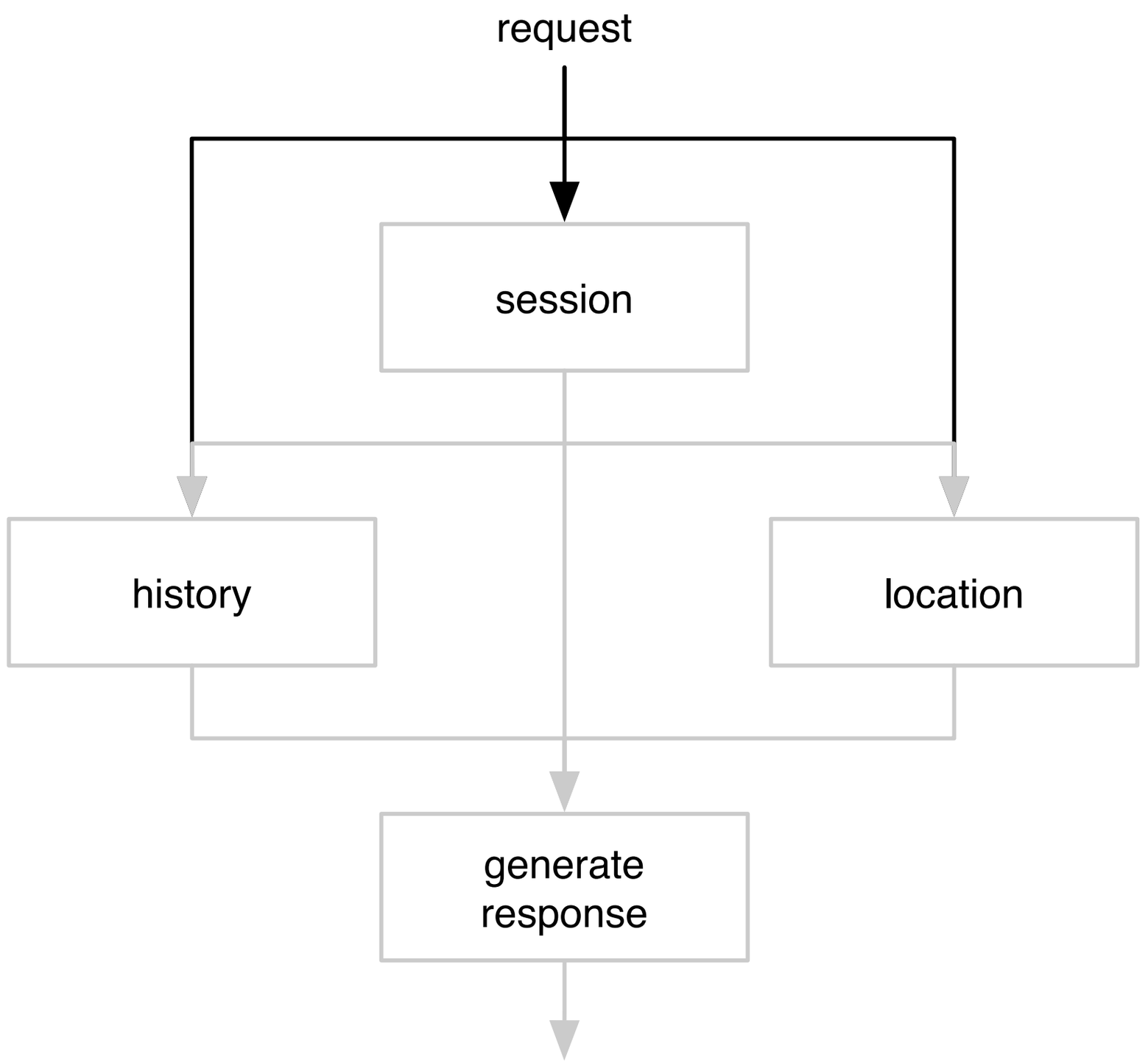
a request workflow

```
(defn handler [request]
  (let [session (get-session request)
        history (or (:history session)
                    (get-history request))
        location (or (:location session)
                     (get-location request))]
    (generate-response
      request
      session
      history
      location))
```

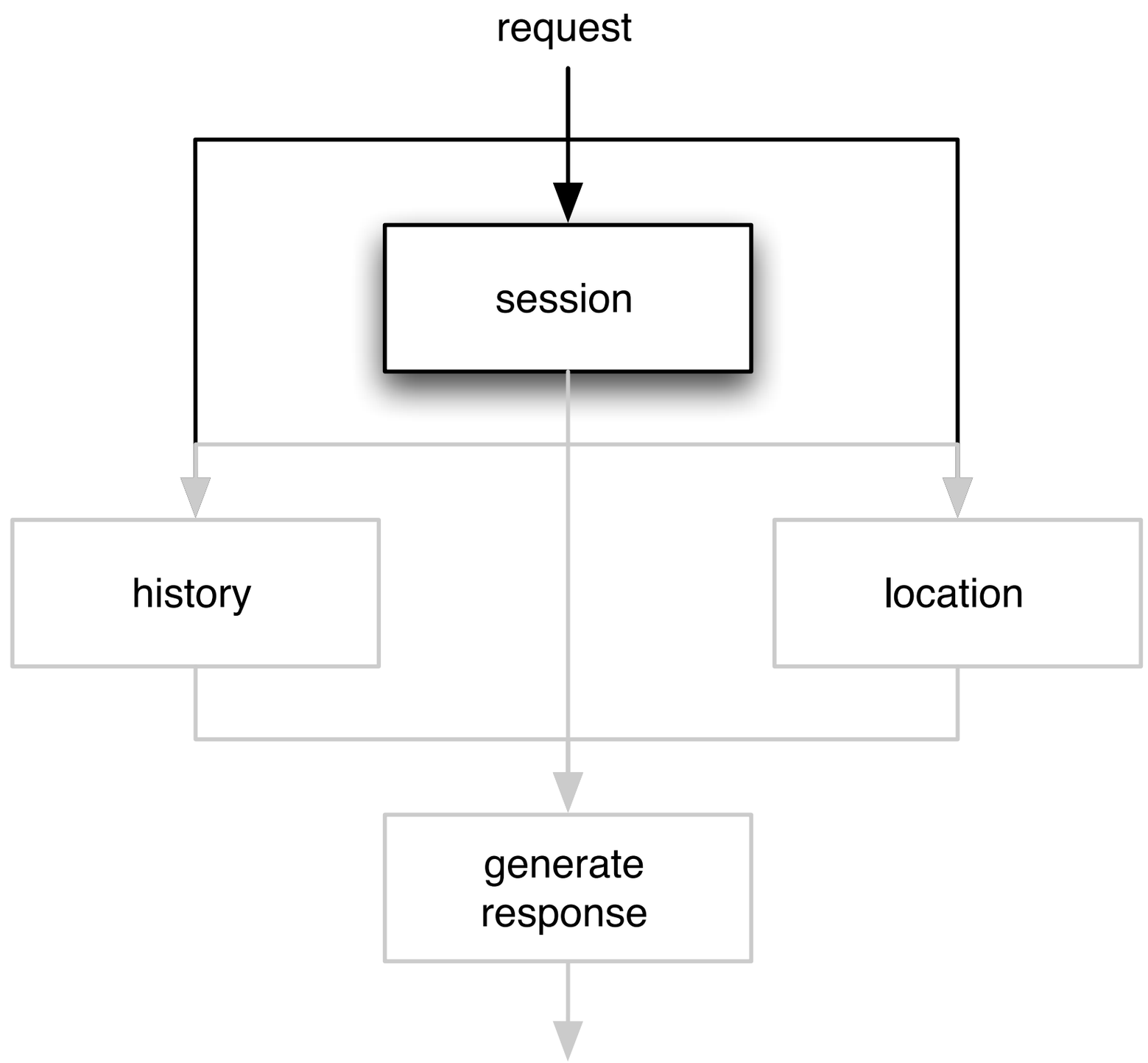
flow of execution



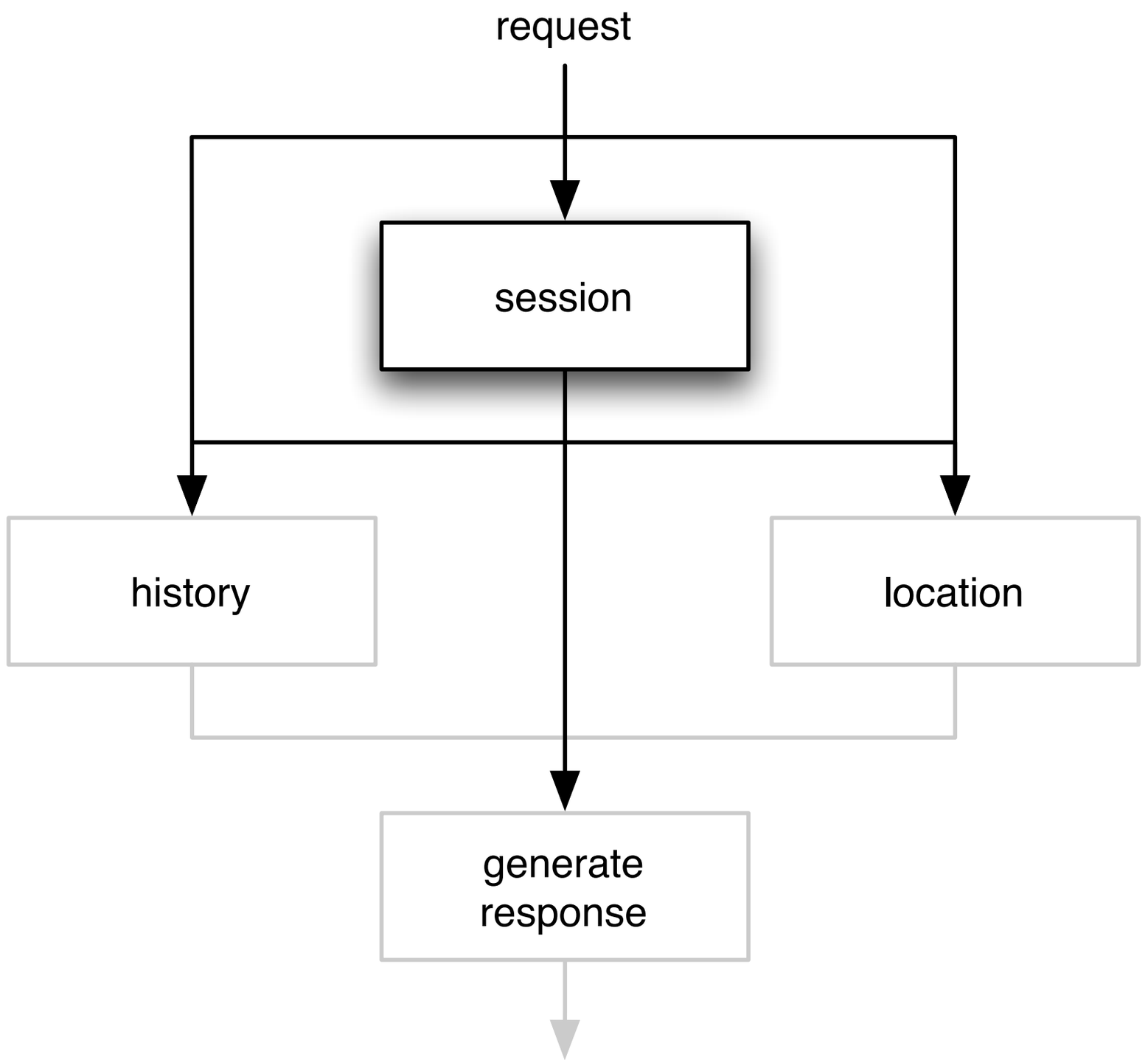
flow of data



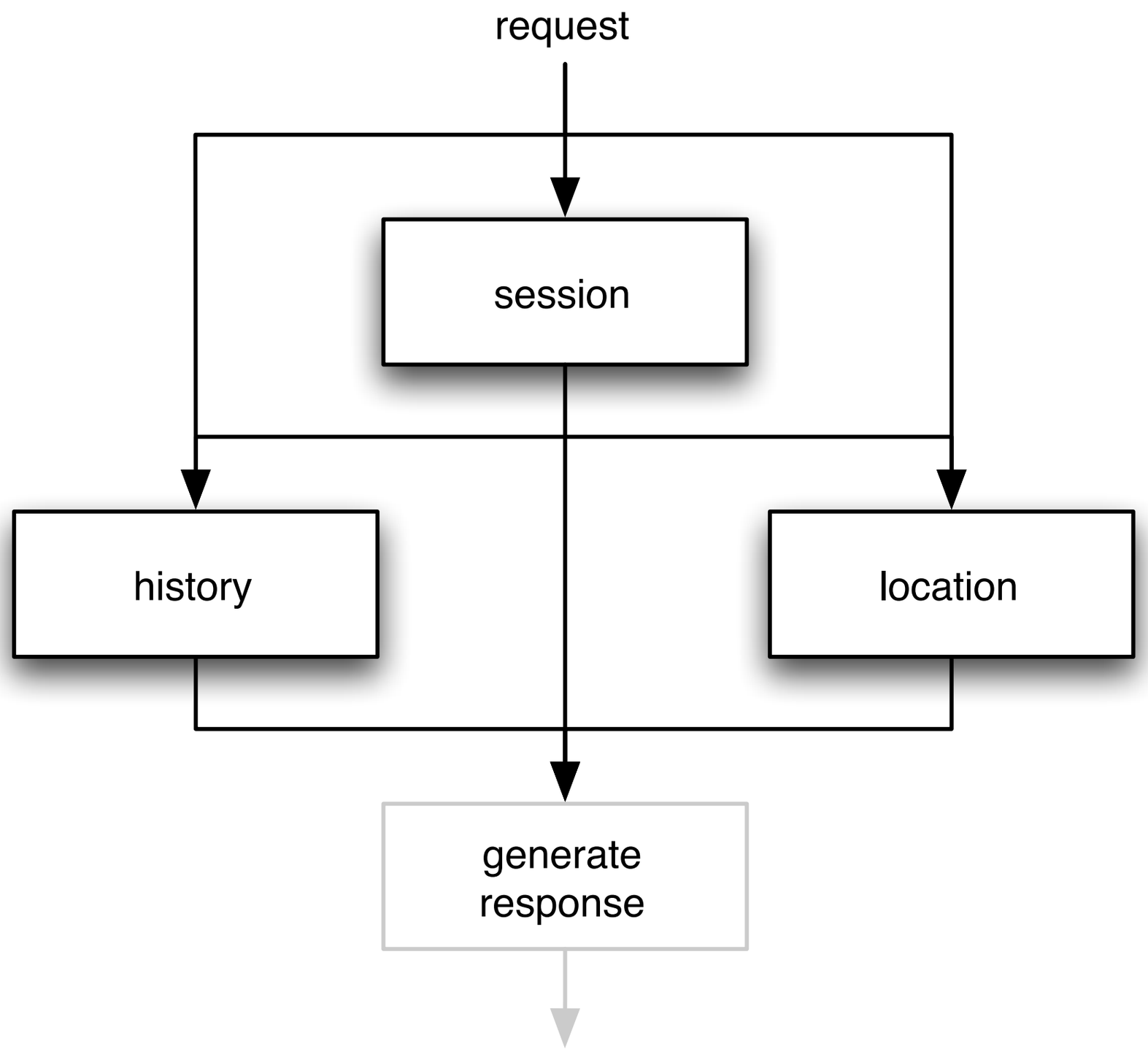
flow of data



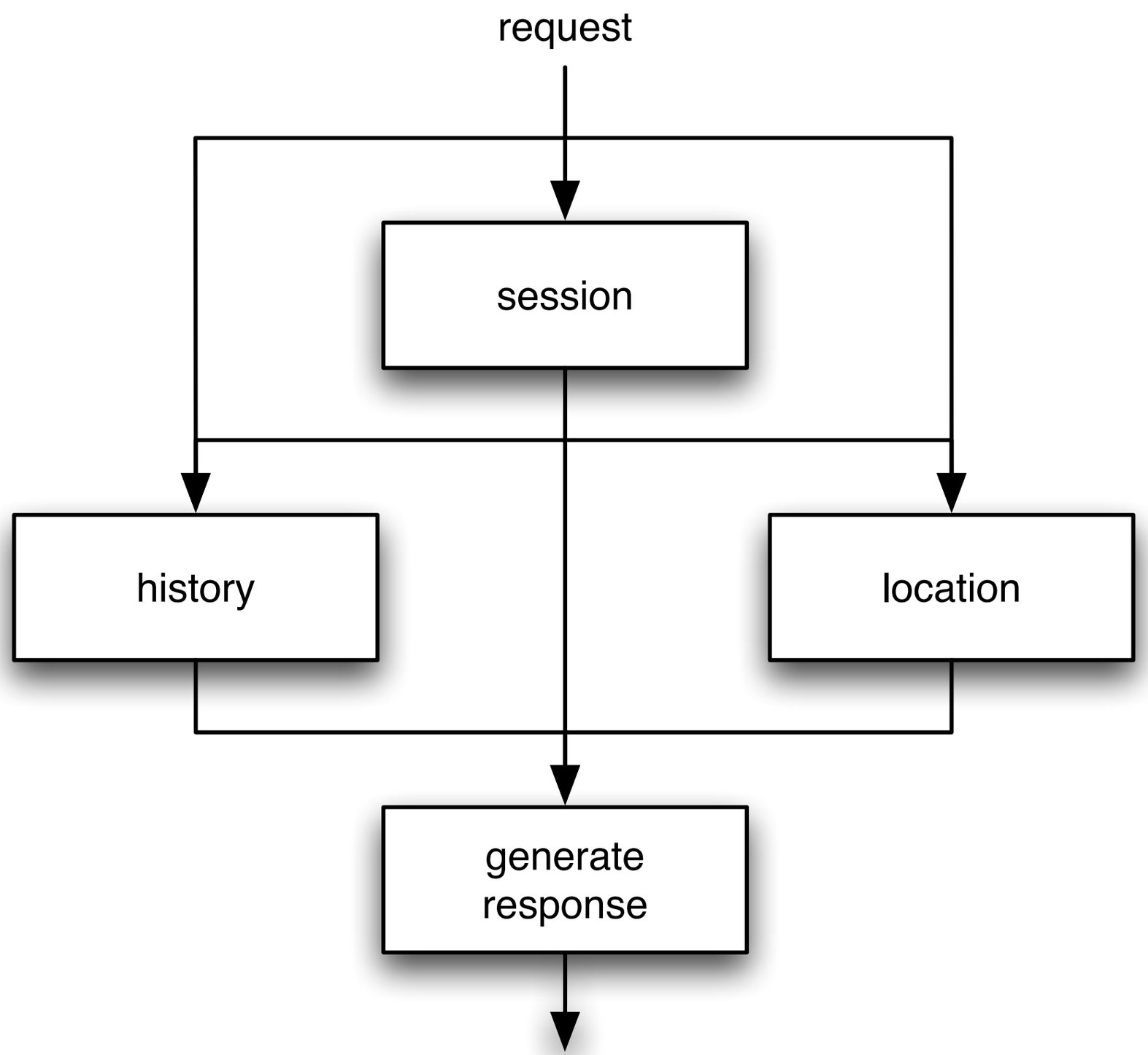
flow of data



flow of data



flow of data



flow of data

```
(defn handler [request]
  (let [session (get-session request)
        history (or (:history session)
                    (get-history request))
        location (or (:location session)
                     (get-location request))]
    (generate-response
      request
      session
      history
      location))
```

the code is a perfect expression of the dataflow!

the async macro

- expressions immediately return a result-channel representing the eventual result
- execution of expressions are deferred until all parameters are realized
- if a parameter emits an error, the expression emits the same error

the async macro

```
(defn handler [request]
  (async
    (let [session (get-session request)
          history (or (:history session)
                      (get-user-history request))
          location (or (:location session)
                       (get-user-location request))]
      (generate-response
        request
        session
        history
        location))))
```

the async macro

```
(defn handler [request]
  (async
    (let [session (get-session request)
          history (or (:history session)
                      (get-user-history request))
          location (or (:location session)
                       (get-user-location request))]
      (generate-response
        request
        session
        history
        location))))
```

the async macro

```
(defn handler [request]
  (async
    (let [session (get-session request)
          history (or (:history session)
                      (get-user-history request))
          location (or (:location session)
                       (get-user-location request))]
      (generate-response
        request
        session
        history
        location))))
```

the async macro

```
(defn handler [request]
  (async
    (let [session (get-session request)
          history (or (:history session)
                      (get-user-history request))
          location (or (:location session)
                       (get-user-location request))]
      (generate-response
        request
        session
        history
        location))))
```

the async macro

```
(defn handler [request]
  (async
    (let [session (get-session request)
          history (or (:history session)
                      (get-user-history request))
          location (or (:location session)
                       (get-user-location request))]
      (generate-response
        request
        session
        history
        location))))
```

the async macro

```
(defn handler [request]
  (async
    (let [session (get-session request)
          history (or (:history session)
                      (get-user-history request))
          location (or (:location session)
                       (get-user-location request))]
      (generate-response
        request
        session
        history
        location)))
```

the async macro

```
(defn handler [request]
  (async
    (let [session (get-session request)
          history (or (:history session)
                      (get-user-history request))
          location (or (:location session)
                       (get-user-location request))]
      (generate-response
        request
        session
        history
        location))))
```

the async macro

```
(defn handler [request]
  (async
    (let [session (get-session request)
          history (or (:history session)
                      (get-user-history request))
          location (or (:location session)
                       (get-user-location request))]
      (generate-response
        request
        session
        history
        location))))
```

async try/catch

```
(async
  (try
    (handler request)
    (catch Exception e
      (handle-error e))))
```

error handling doesn't have to be local!

task

```
(async
  (= 3 (task (calculate-pi))))
```

- returns result-channel, executes body on thread-pool
- is just an **annotation**, not a structural change
- because async expressions will return when they can't continue, deadlock is impossible

async side-effects

```
(async
  (do
    (query-moon-base)
    (query-database)))
```

- the moon base's response isn't part of our dataflow
- if there's an error, how do we know?

force

```
(async
  (do
    (force (query-moon-base))
    (query-database)))
```

- makes an expression a pre-requisite for all subsequent expressions
- an error will cause subsequent expressions to also error out

however, async is opaque

- no way to examine partially complete flows
- difficult to debug
- complicated flows are still complicated, under the covers

future work

- improved debugging for async workflows
- improved instrumentation for distributed systems
- ClojureScript!

the projects

- all functionality discussed is contained in **lamina** (http://github.com/ztellman/lamina)
- it is used to model network communication in **aleph** (http://github.com/ztellman/aleph)

questions?