Onyx

Distributed Data Processing for Clojure

Michael Drogalis

Me!

- Creator of Onyx
- Years of writing commercial analytics platforms
- @MichaelDrogalis

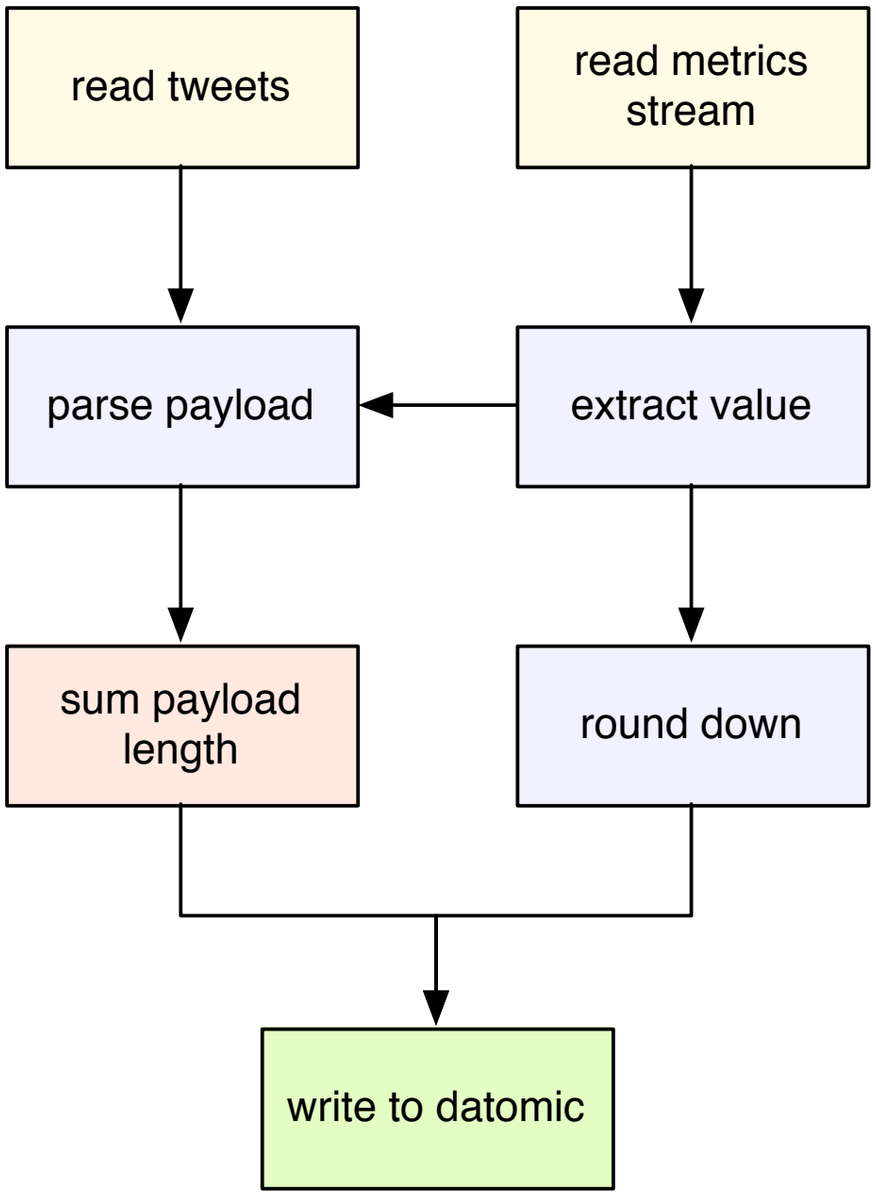
Onyx

- Cloud-scale distributed data processing platform
- Decomposes traditional computing model
- Hybrid batch & streaming API
- Masterless
- Clojure all the way down

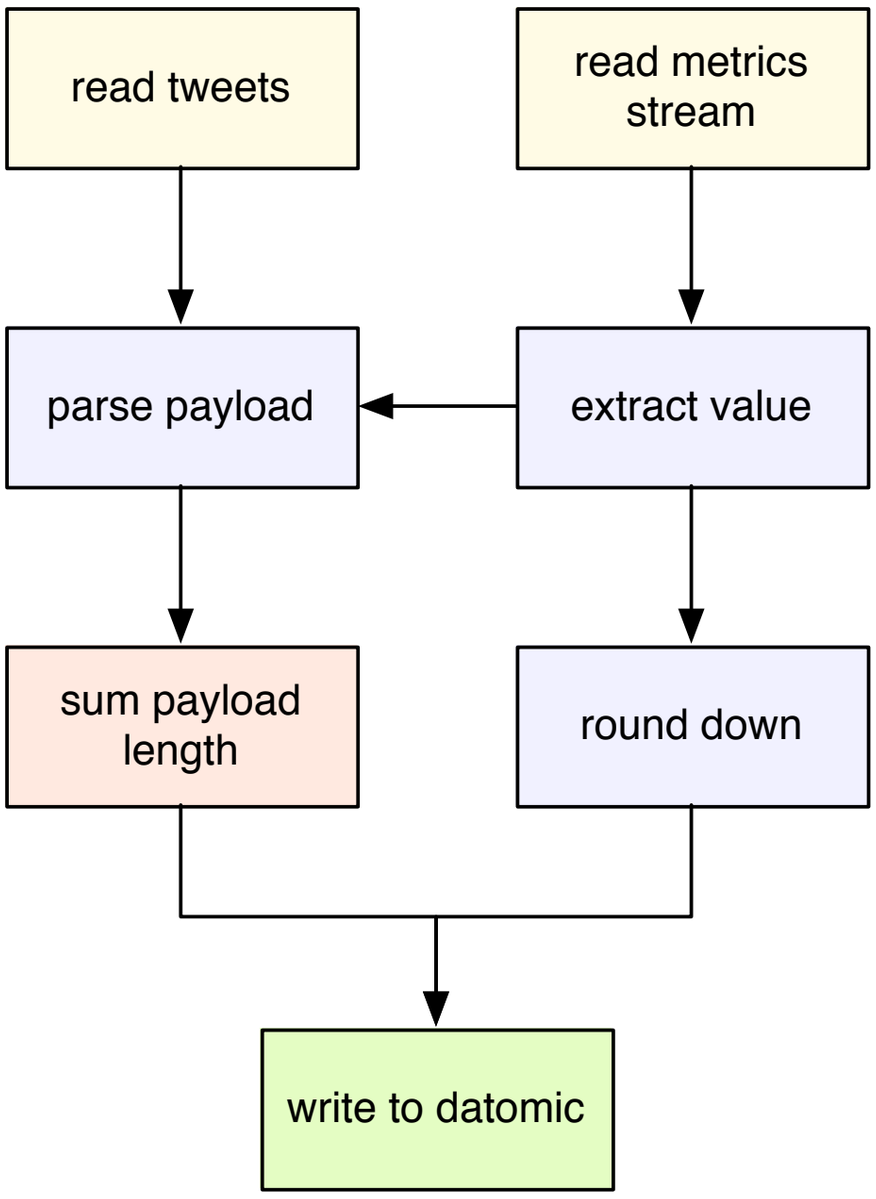
Tell me more!

- API value proposition
- Design
  - Messaging
  - Coordination

Workflows


read tweets
read metrics stream
parse payload
extract value
sum payload length
round down
write to datomic

Workflows




```
[[:read-tweets :parse-payload]
 [:read-metrics-stream :extract-value]
 [:parse-payload :sum-payload-lengths]
 [:extract-value :round-down]
 [:sum-payload-lengths :write-to-datomic]
 [:round-down :write-to-datomic]]
```

task

Catalogs

```
[[:read-tweets :parse-payload]
 [:read-metrics-stream :extract-value]
 [:parse-payload :sum-payload-lengths]
 [:extract-value :round-down]
 [:sum-payload-lengths :write-to-datomic]
 [:round-down :write-to-datomic]]
```

Catalogs

```
[{:onyx/name :read-tweets
  :onyx/ident :twitter/read-from-stream
  :onyx/type :input
  :onyx/medium :twitter
  :onyx/batch-size 20
  :onyx/max-peers 1
  :onyx/doc "Reads segments from the tweet stream"}

 {:onyx/name :parse-payload
  :onyx/fn :my.ns/parse-payload
  :onyx/type :function
  :onyx/batch-size 30
  :onyx/doc "Sums the number of characters in :tweet"}
...]
```

Catalogs

task

```
[{:onyx/name :read-tweets
  :onyx/ident :twitter/read-from-stream
  :onyx/type :input
  :onyx/medium :twitter
  :onyx/batch-size 20
  :onyx/max-peers 1
  :onyx/doc "Reads segments from the tweet stream"}

 {:onyx/name :parse-payload
  :onyx/fn :my.ns/parse-payload
  :onyx/type :function
  :onyx/batch-size 30
  :onyx/doc "Sums the number of characters in :tweet"}
...]
```

task

Catalogs

```
[{:onyx/name :read-tweets
  :onyx/ident :twitter/read-from-stream
  :onyx/type :input
  :onyx/medium :twitter
  :onyx/batch-size 20
  :onyx/max-peers 1
  :onyx/doc "Reads segments from the tweet stream"}

 {:onyx/name :parse-payload
  :onyx/fn :my.ns/parse-payload
  :onyx/type :function
  :onyx/batch-size 30
  :onyx/doc "Sums the number of characters in :tweet"}
...]
```

Clojure function

Functions

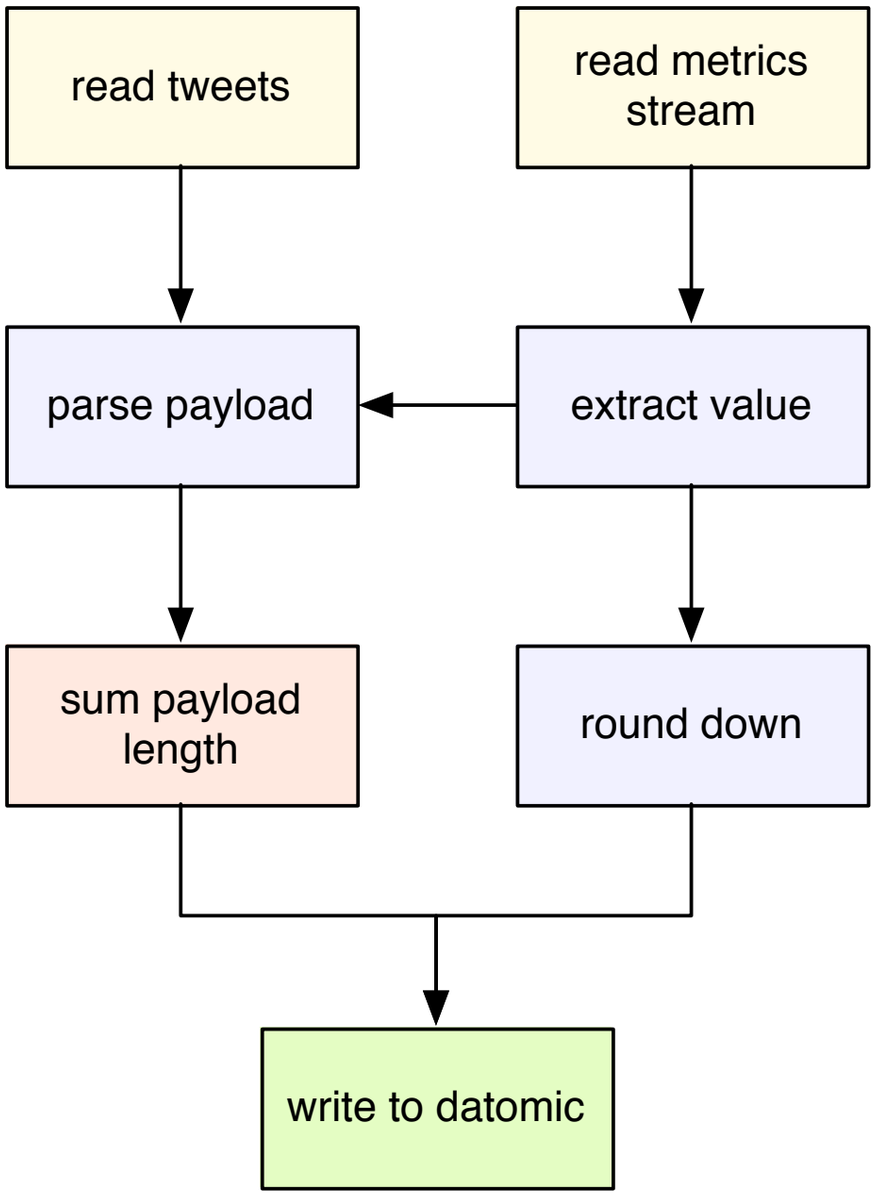
```
(ns my.ns)

(defn parse-payload [{:keys [payload] :as segment}]
  {:payload (parse-json payload)})
```

Take and return maps (segments)

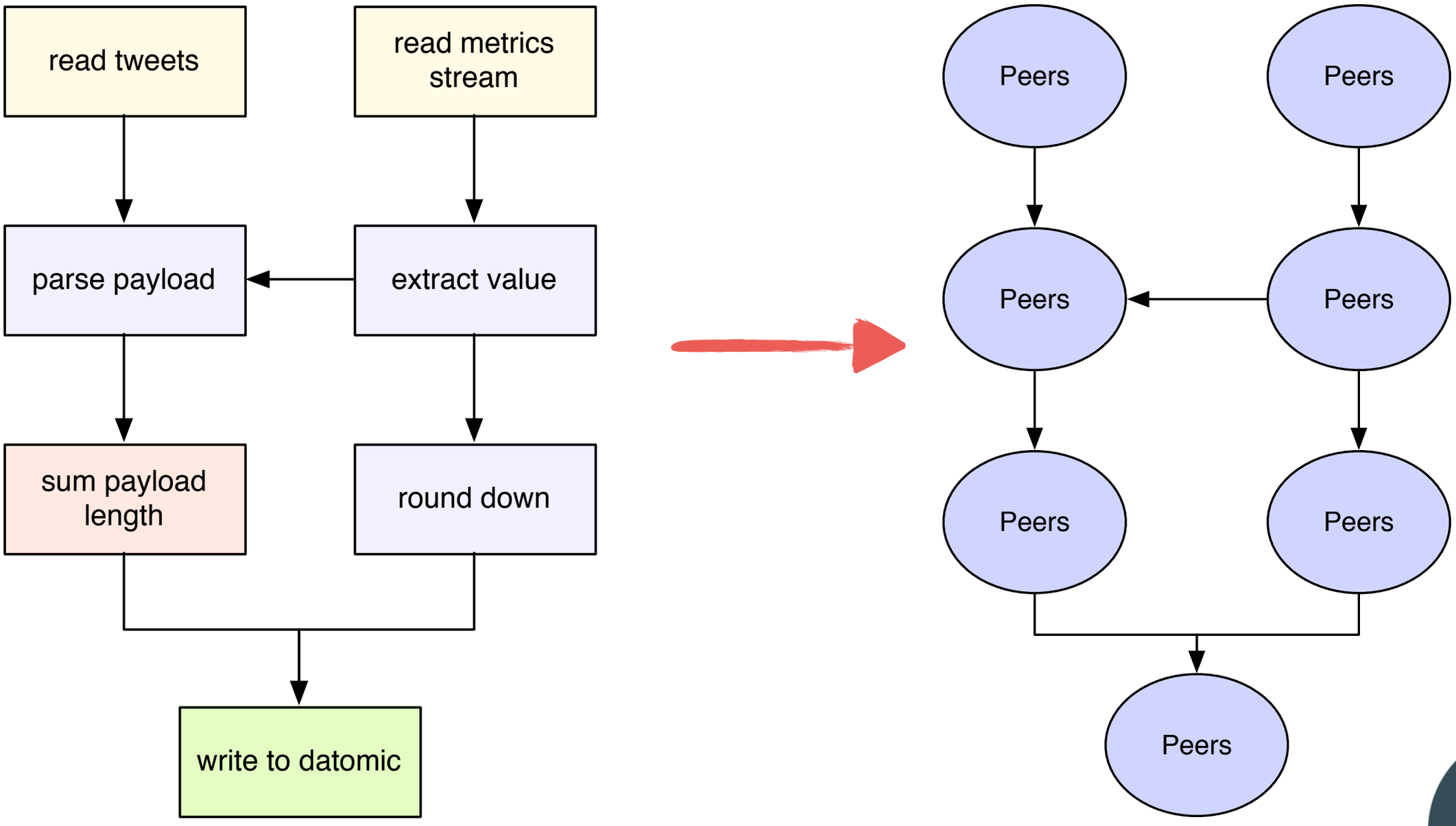
Messaging Design

- Peer-to-peer, asynchronous, guaranteed segment processing
- Messaging algorithm mostly adopted from Storm
  - Created by Nathan Marz
- Segment state tracked by 20 byte value in memory on acking daemon

Messaging Design



Messaging Design



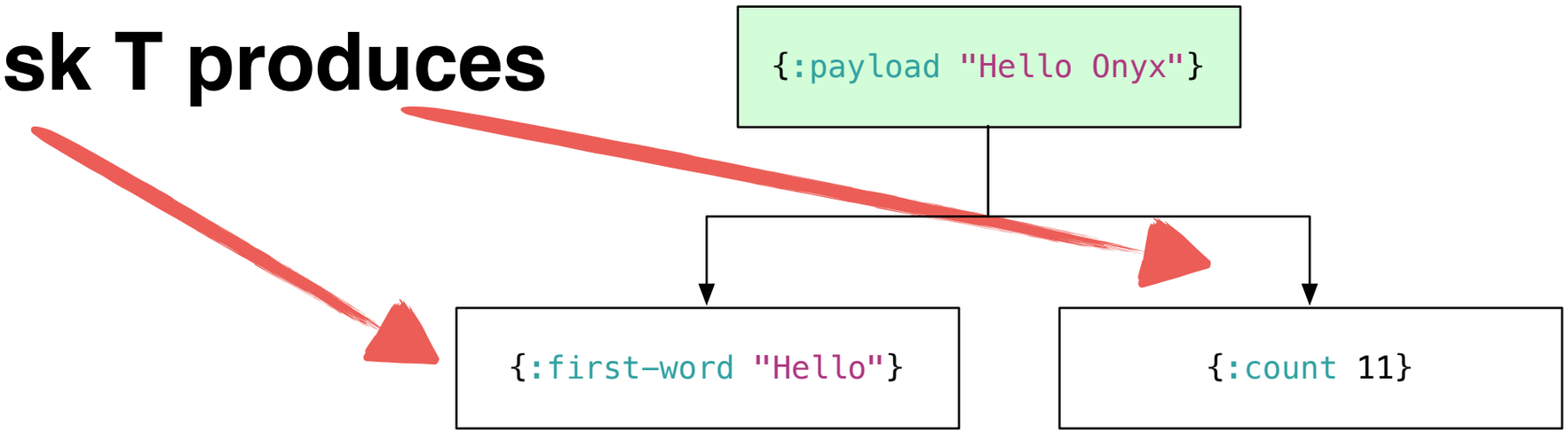
Segment Trees

root segment

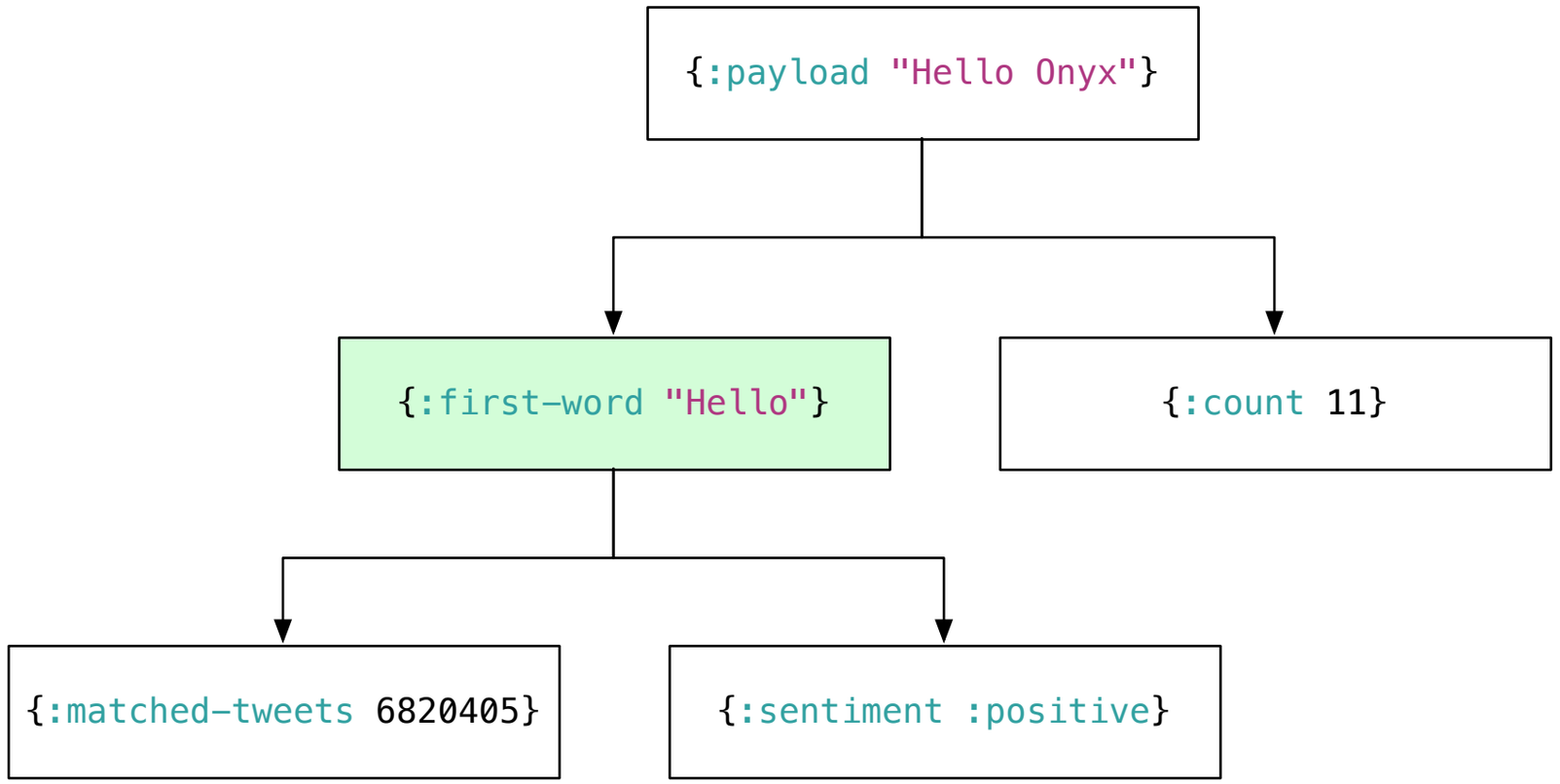
```
{:payload "Hello Onyx"}
```

Segment Trees

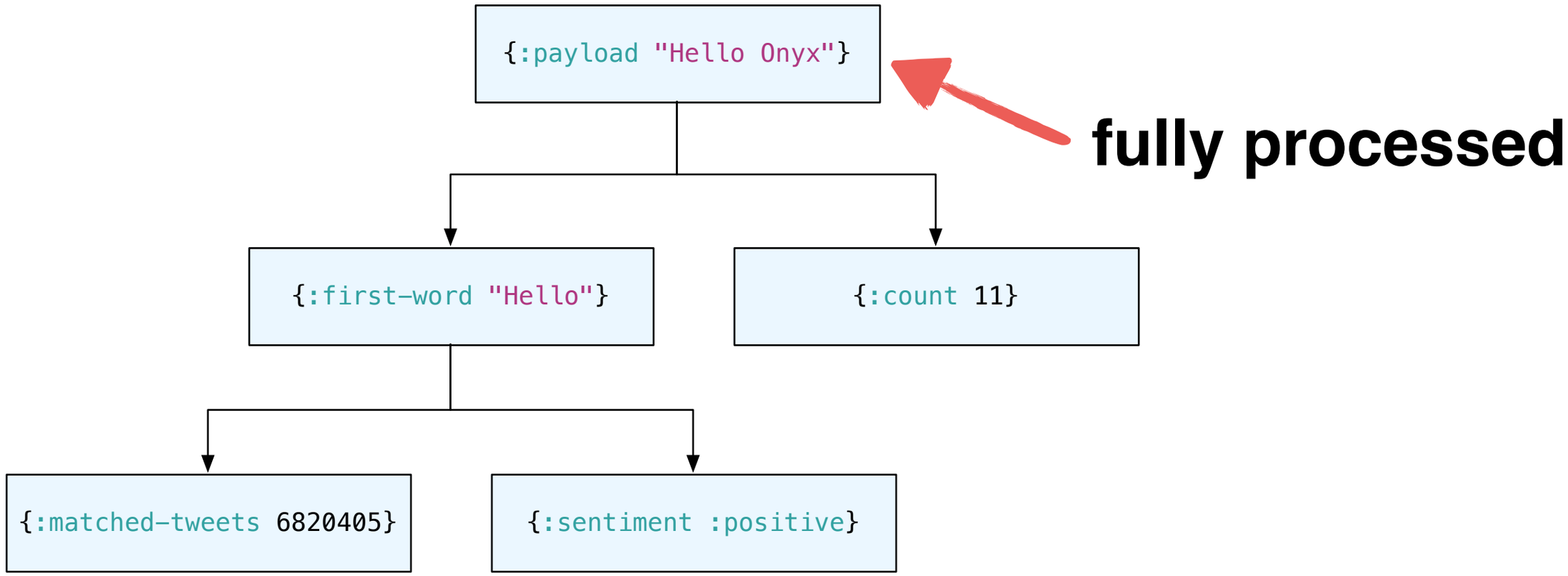
Task T produces



Segment Trees



Segment Trees



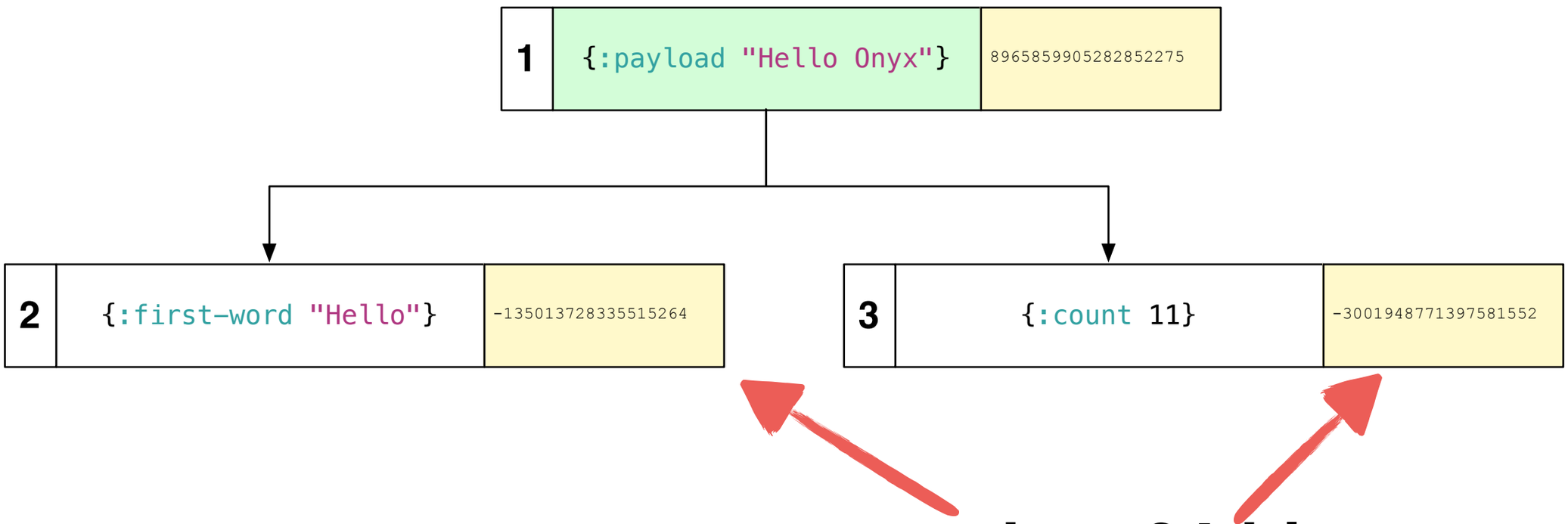
How do we Guarantee it?

1	{:payload "Hello Onyx"}	8965859905282852275

random 64-bit sequence

Tracked state [1]:
8965859905282852275

How do we Guarantee it?



random 64-bit sequences

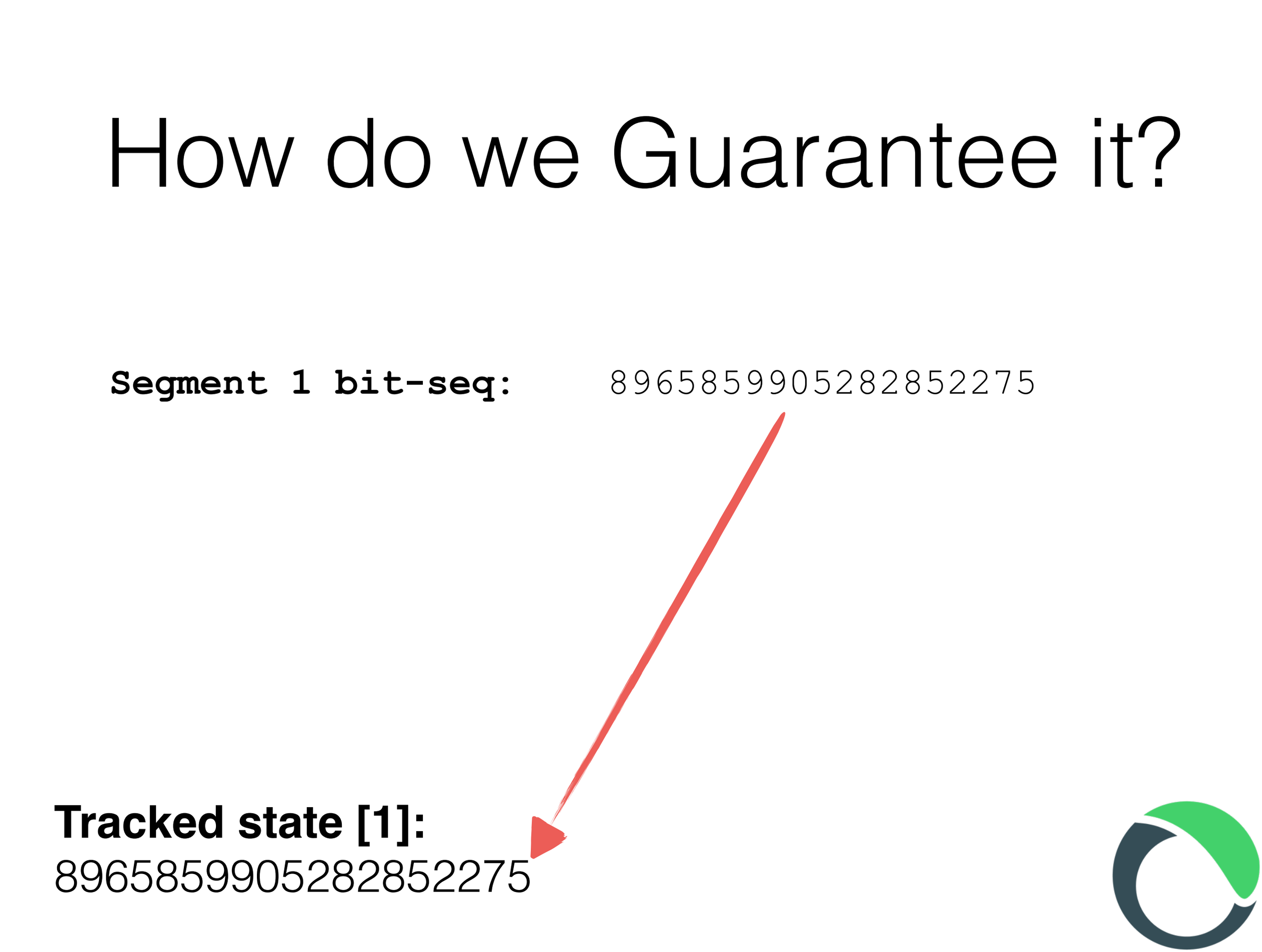
Tracked state [2]:
29157231680028642 72

updated state

How do we Guarantee it?

Segment 1 bit-seq: 8965859905282852275

Tracked state [1]:
8965859905282852275



How do we Guarantee it?

Segment 1 bit-seq: 8965859905282852275

Segment 1 ack: 8965859905282852275

Segment 2 bit-seq: -1350137283355152 64

Segment 3 bit-seq: -3001948771397581552

```
(bit-xor 8965859905282852275 8965859905282852275 -1350137283355152 64 -3001948771397581552)
```

previous state

Tracked state [2]:
2915723168002864272

How do we Guarantee it?

Segment 1 bit-seq: 8965859905282852275

Segment 1 ack: 8965859905282852275

Segment 2 bit-seq: -1350137283355152 64

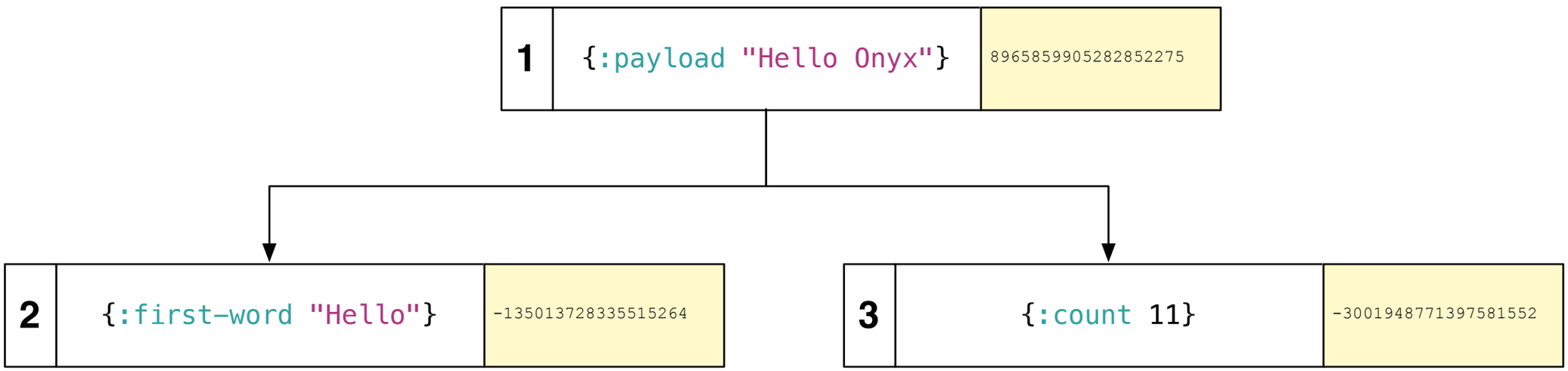
Segment 3 bit-seq: -3001948771397581552

(bit-xor ~~8965859905282852275~~ ~~8965859905282852275~~ -1350137283355152 64 -3001948771397581552)

cancels out

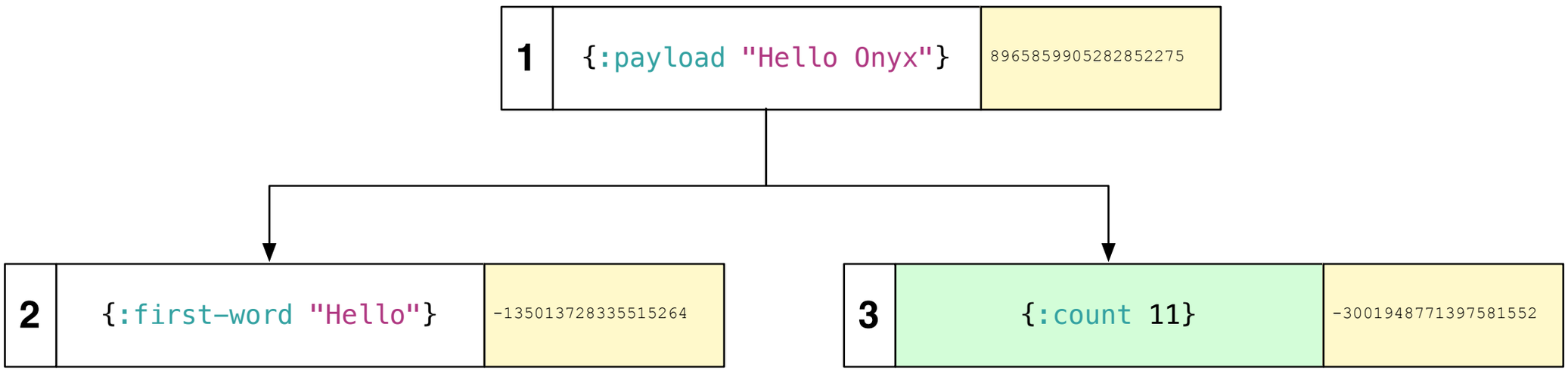
Tracked state [2]:
2915723168002864272

How do we Guarantee it?



Tracked state [2]:
29157231680028642 72

How do we Guarantee it?



Tracked state [3]:
-135013728335515264

How do we Guarantee it?

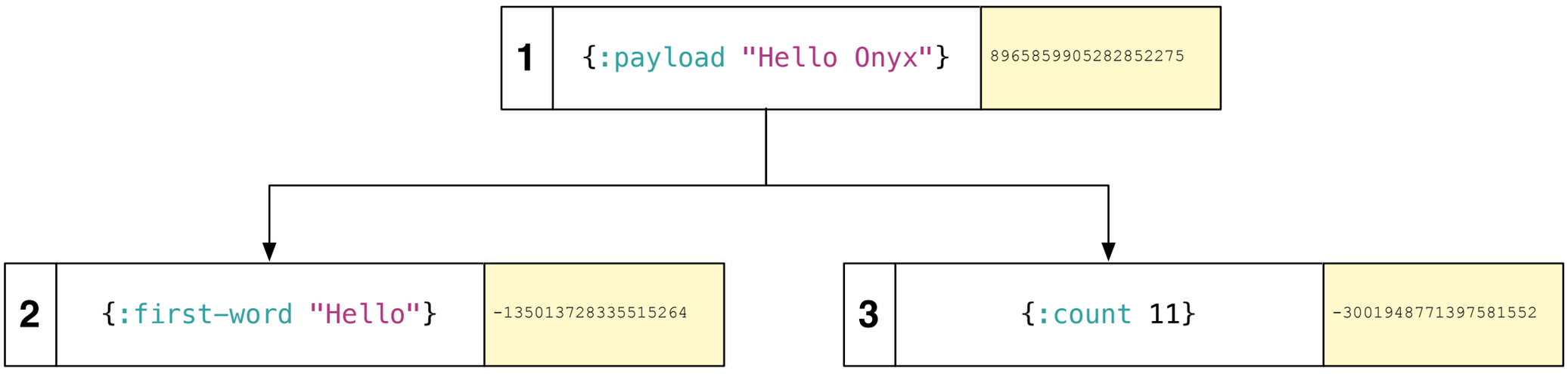
Segment 1 ack: -3001948771397581552

previous state

```
(bit-xor 2915723168002864272 -3001948771397581552)
```

Tracked state [3]:
-13501372833551526 4

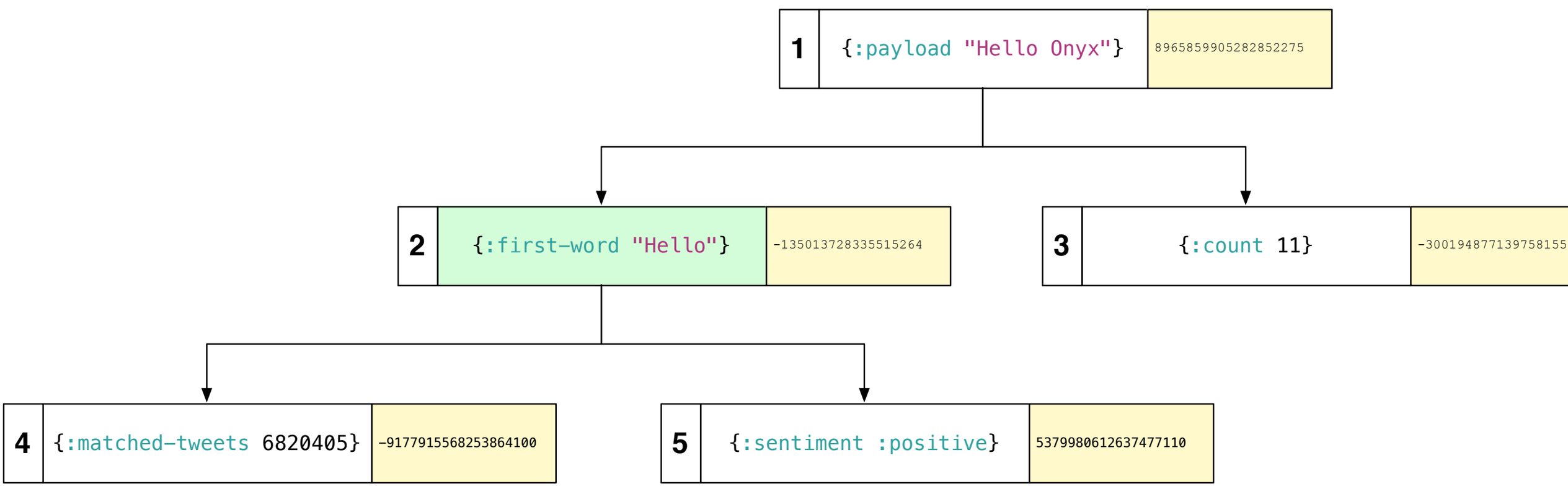
How do we Guarantee it?



Equal values!

Tracked state [3]:
-135013728335515264

How do we Guarantee it?



random 64-bit sequences

Tracked state [4]:
-3888583717264965718

How do we Guarantee it?

Segment 2 ack: -1350137283355152 64

Segment 4 bit-seq: -9177915568253864100

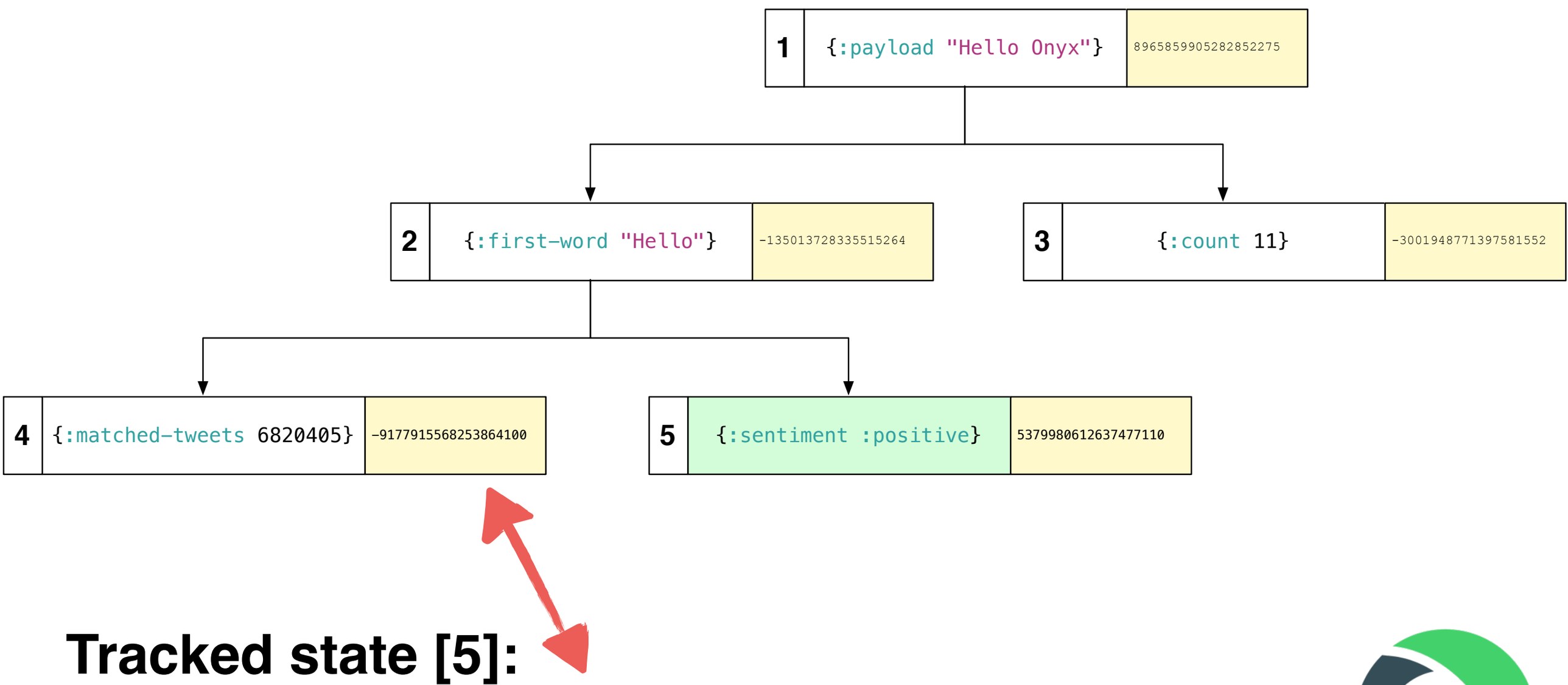
Segment 5 bit-seq: 5379980612637477110

```
(bit-xor -1350137283355152 64
         -1350137283355152 64
         -9177915568253864100
         5379980612637477110)
```

previous state

Tracked state [4]:
-3888583717264965718

How do we Guarantee it?



Tracked state [5]:
-9177915568253864100

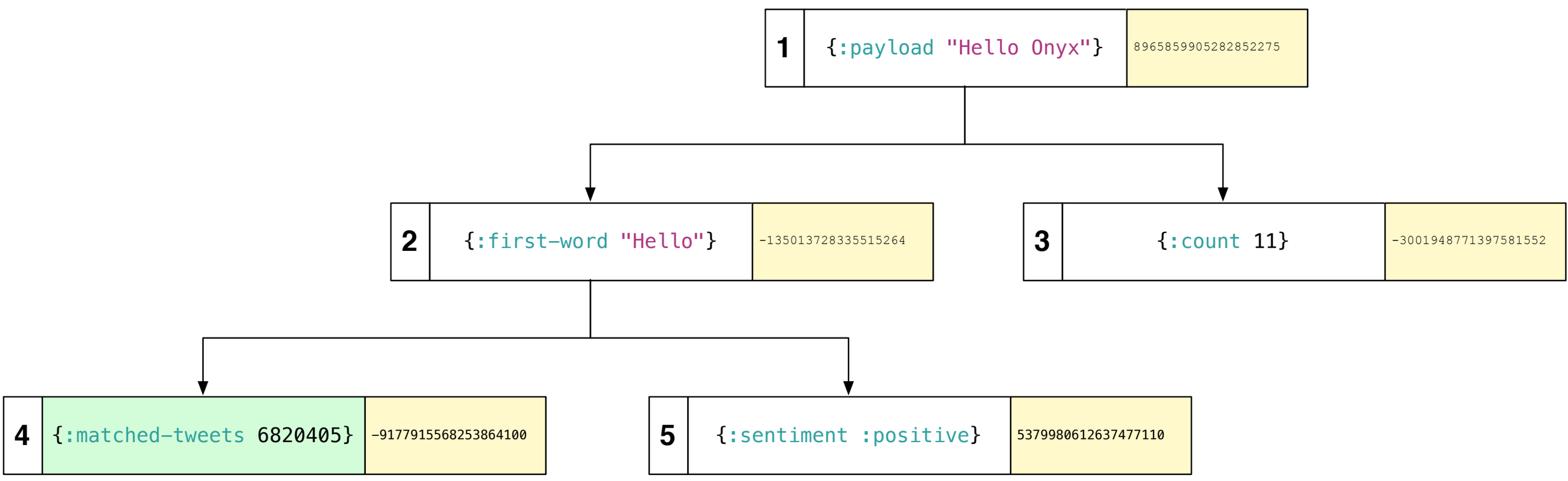
How do we Guarantee it?

Segment 5 ack: 5379980612637477110

```
(bit-xor -3888583717264965718 5379980612637477110)
```

Tracked state [5]:
-9177915568253864100

How do we Guarantee it?



Tracked state [6]:
0

How do we Guarantee it?

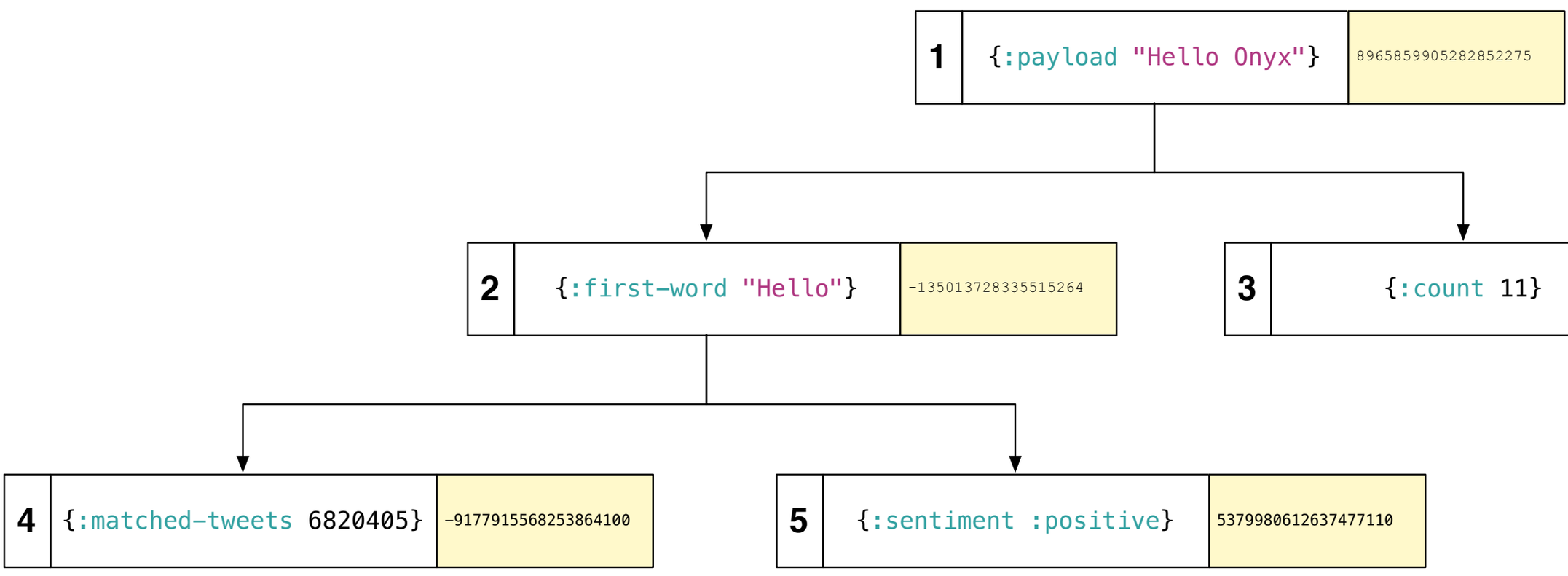
Segment 4 ack: -9177915568253864100

```
(bit-xor -9177915568253864100 -9177915568253864100)
```

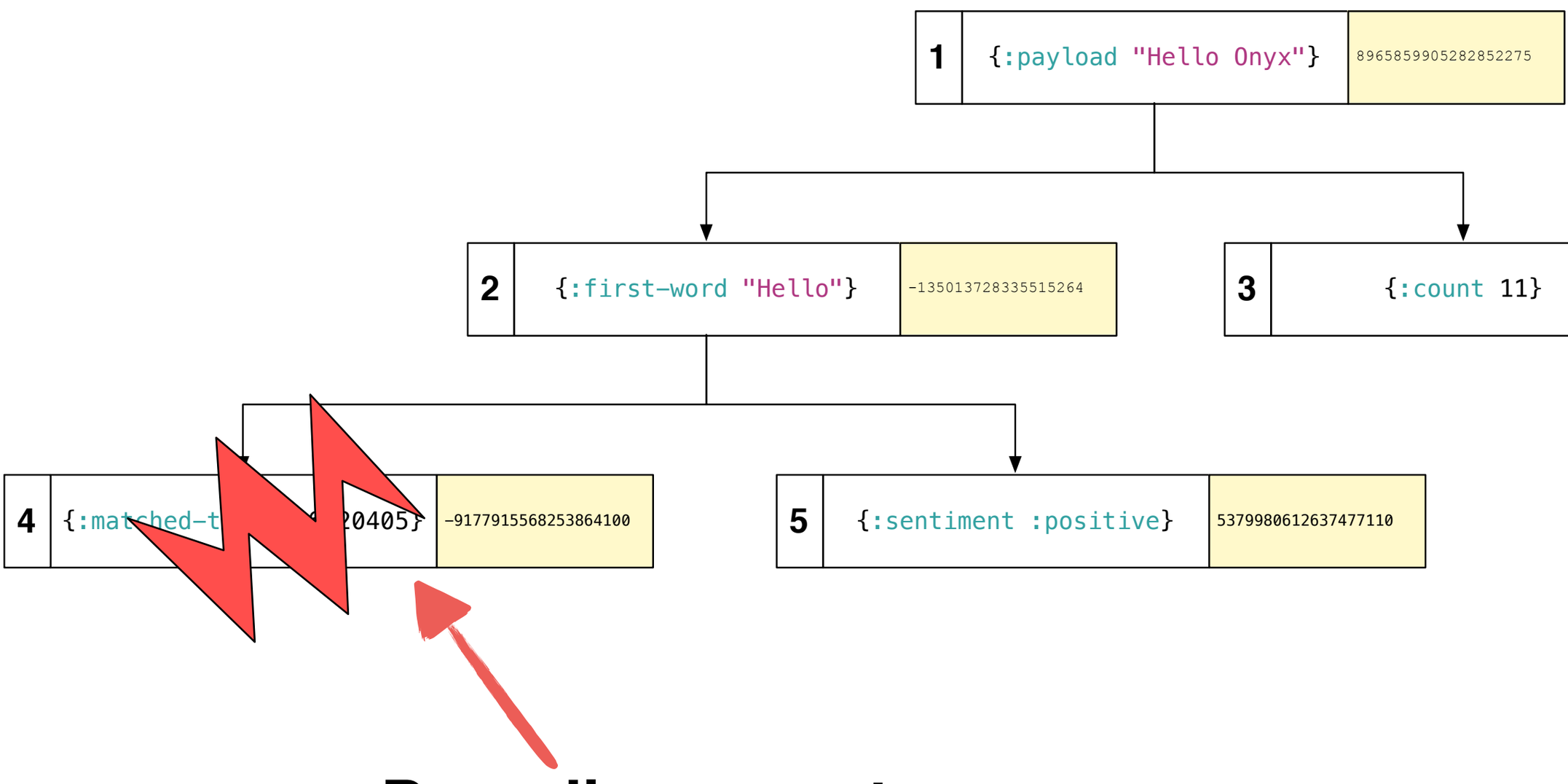
Tracked state [6]:
0

Fully processed tree!

Timeout Detection

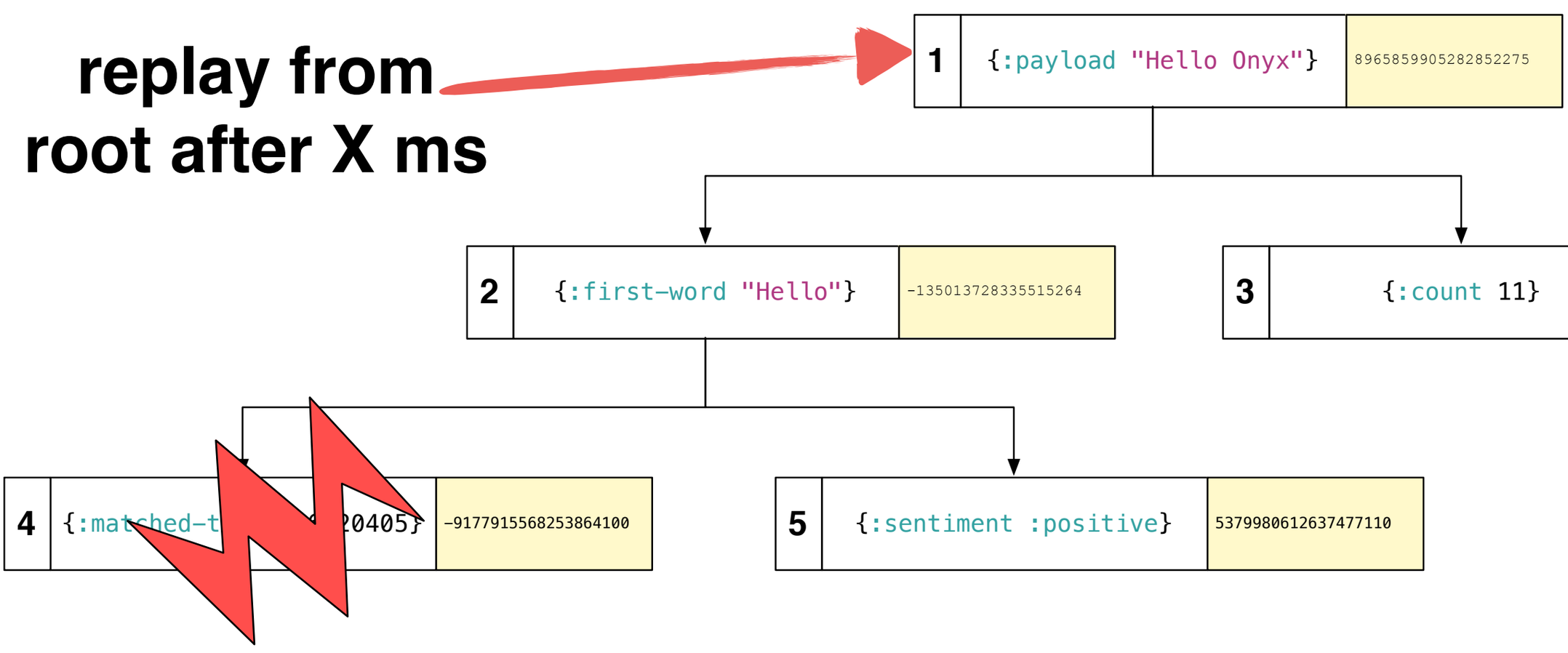


Timeout Detection



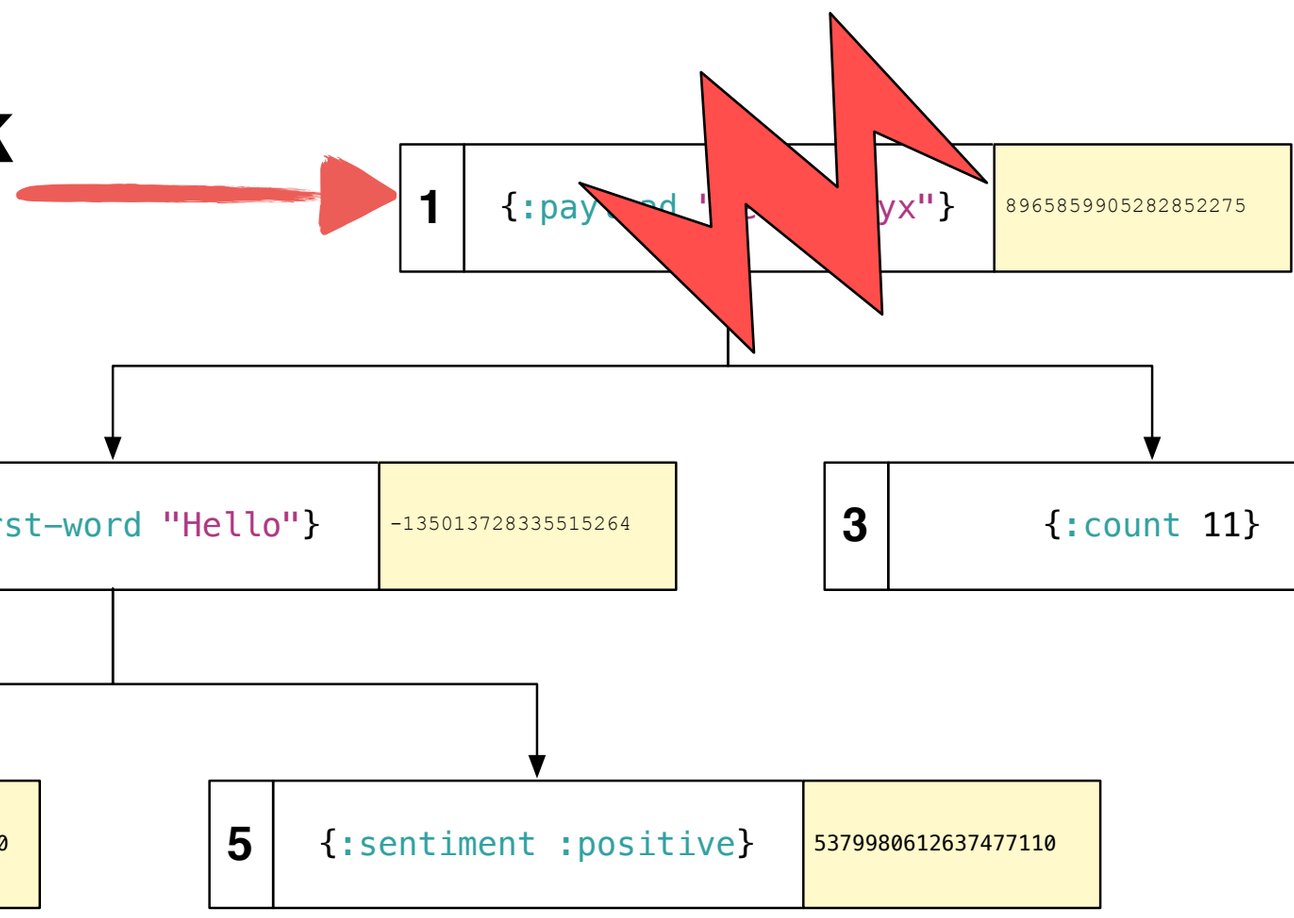
Timeout Detection

replay from root after X ms



Root Failure

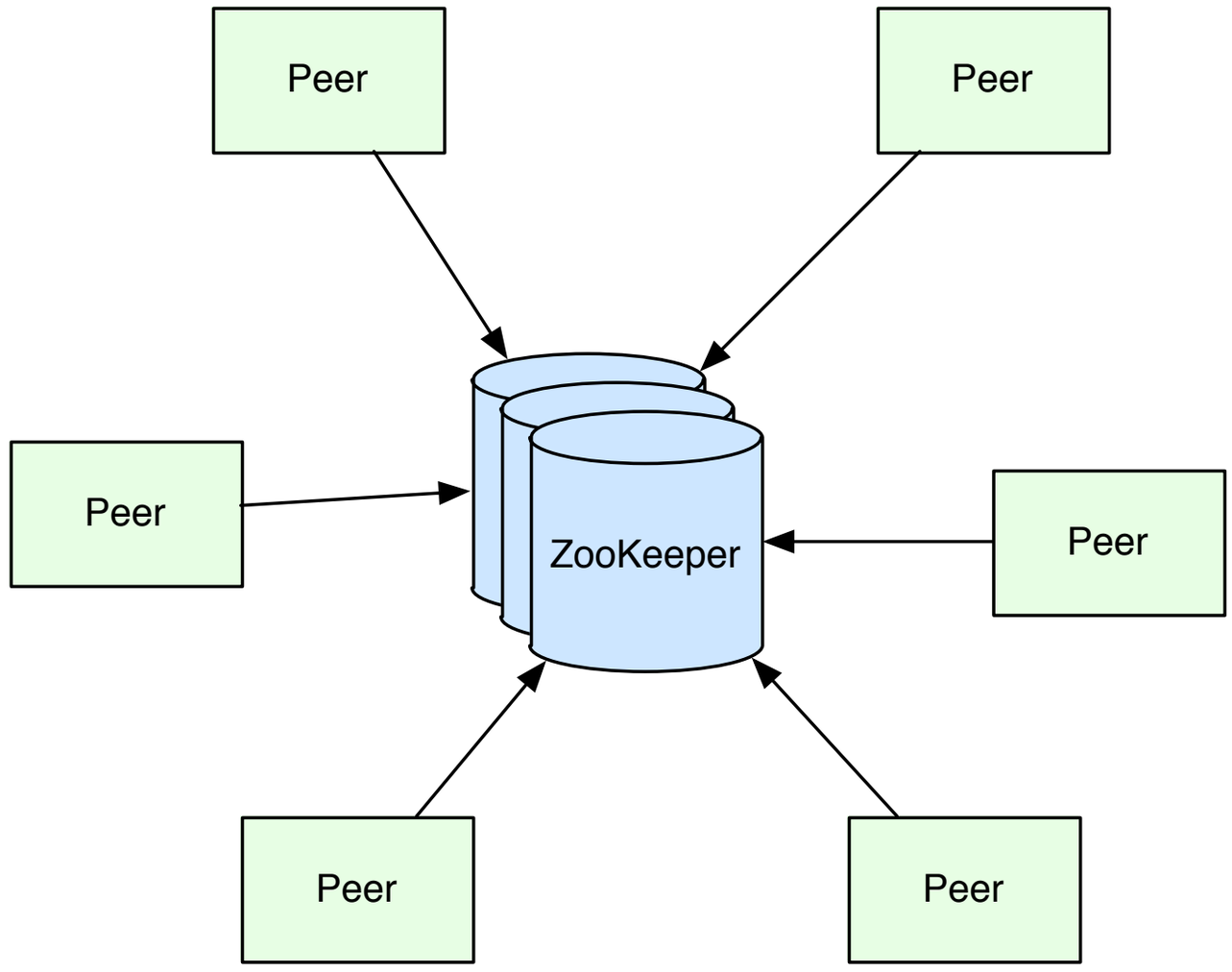
Message back to storage



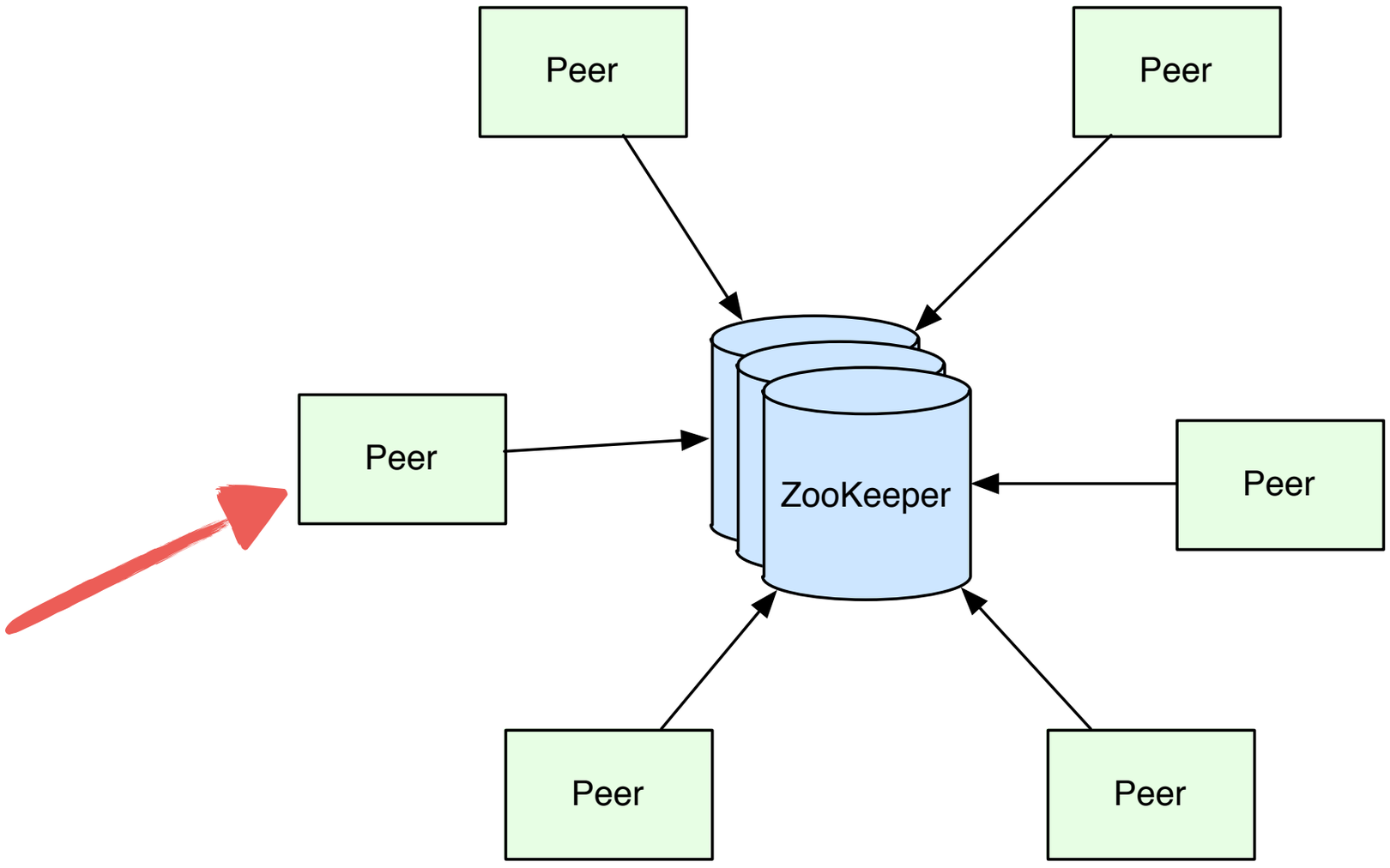
Coordination Design

- Masterless, log based design
- Largely novel, my own work
- Leverages ZooKeeper primitives

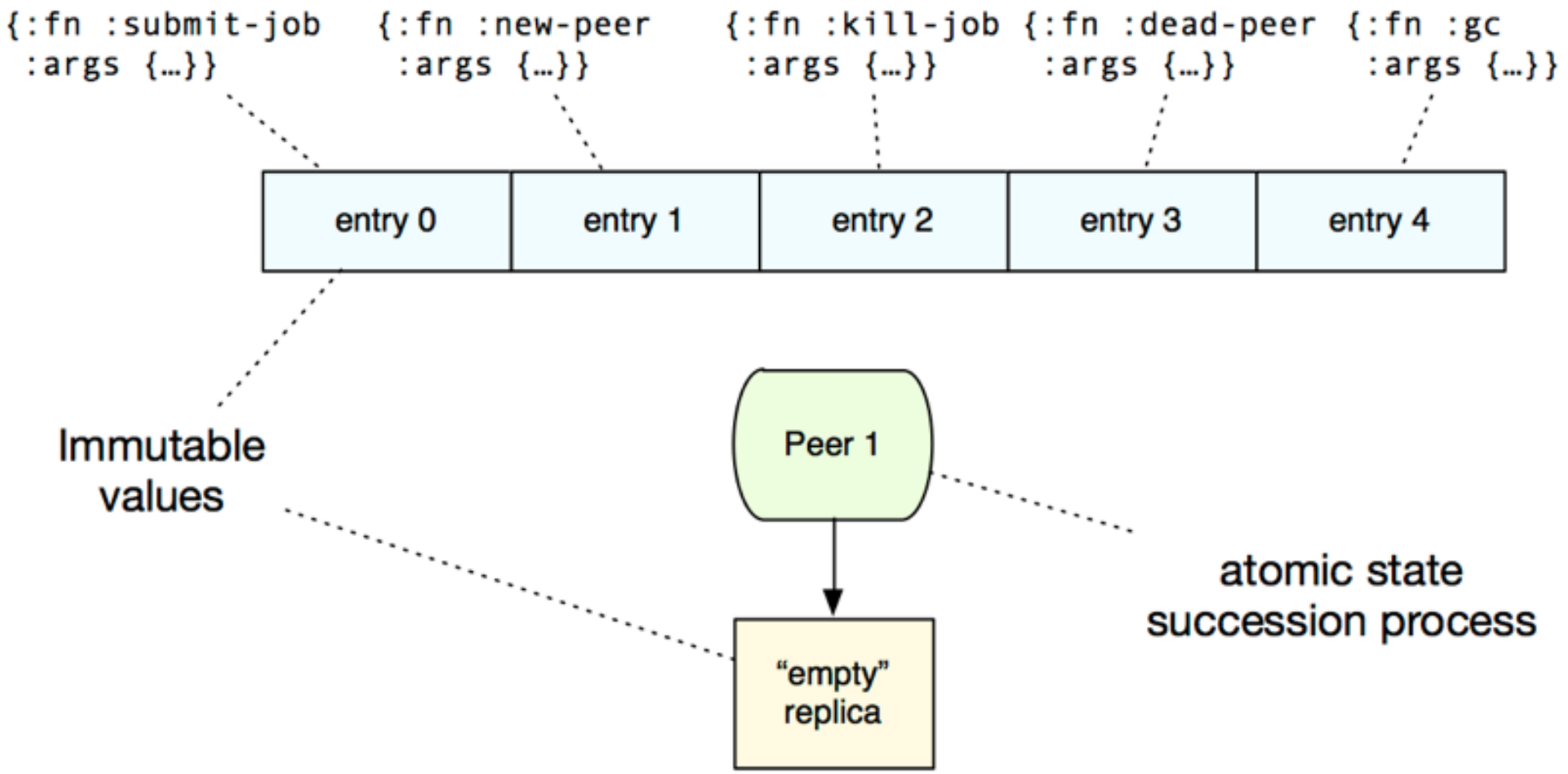
Architecture



Architecture

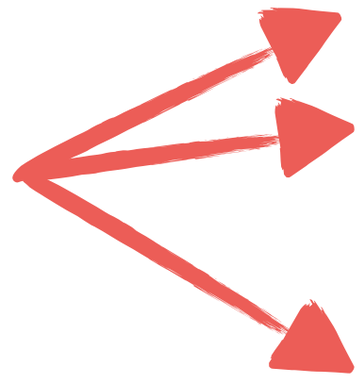


Log Design


{:fn :submit-job
:args {…}}
{:fn :new-peer
:args {…}}
{:fn :kill-job
:args {…}}
{:fn :dead-peer
:args {…}}
{:fn :gc
:args {…}}
entry 0
entry 1
entry 2
entry 3
entry 4
Immutable values
Peer 1
"empty" replica
atomic state succession process

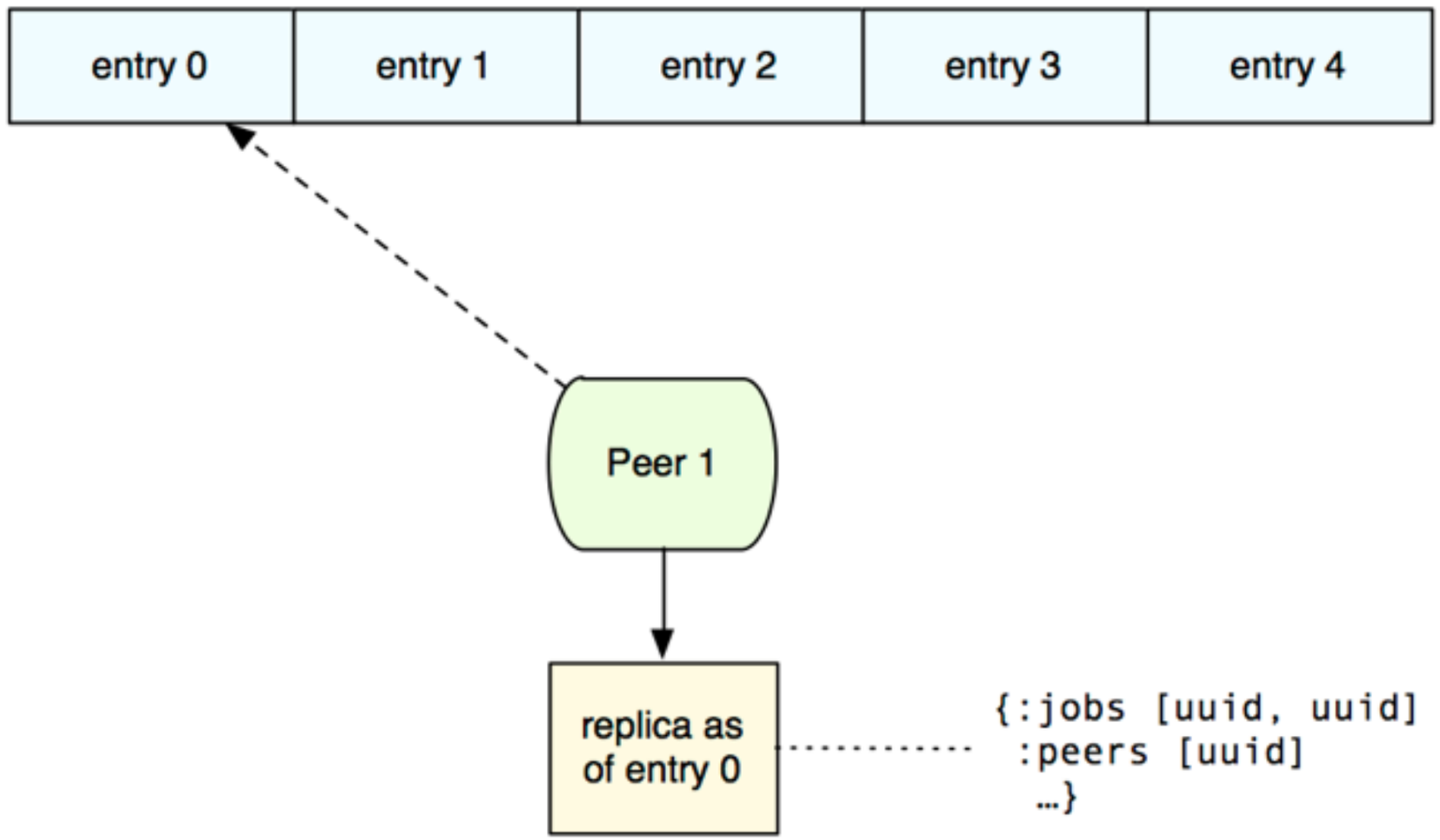
Replica Interface

pure!

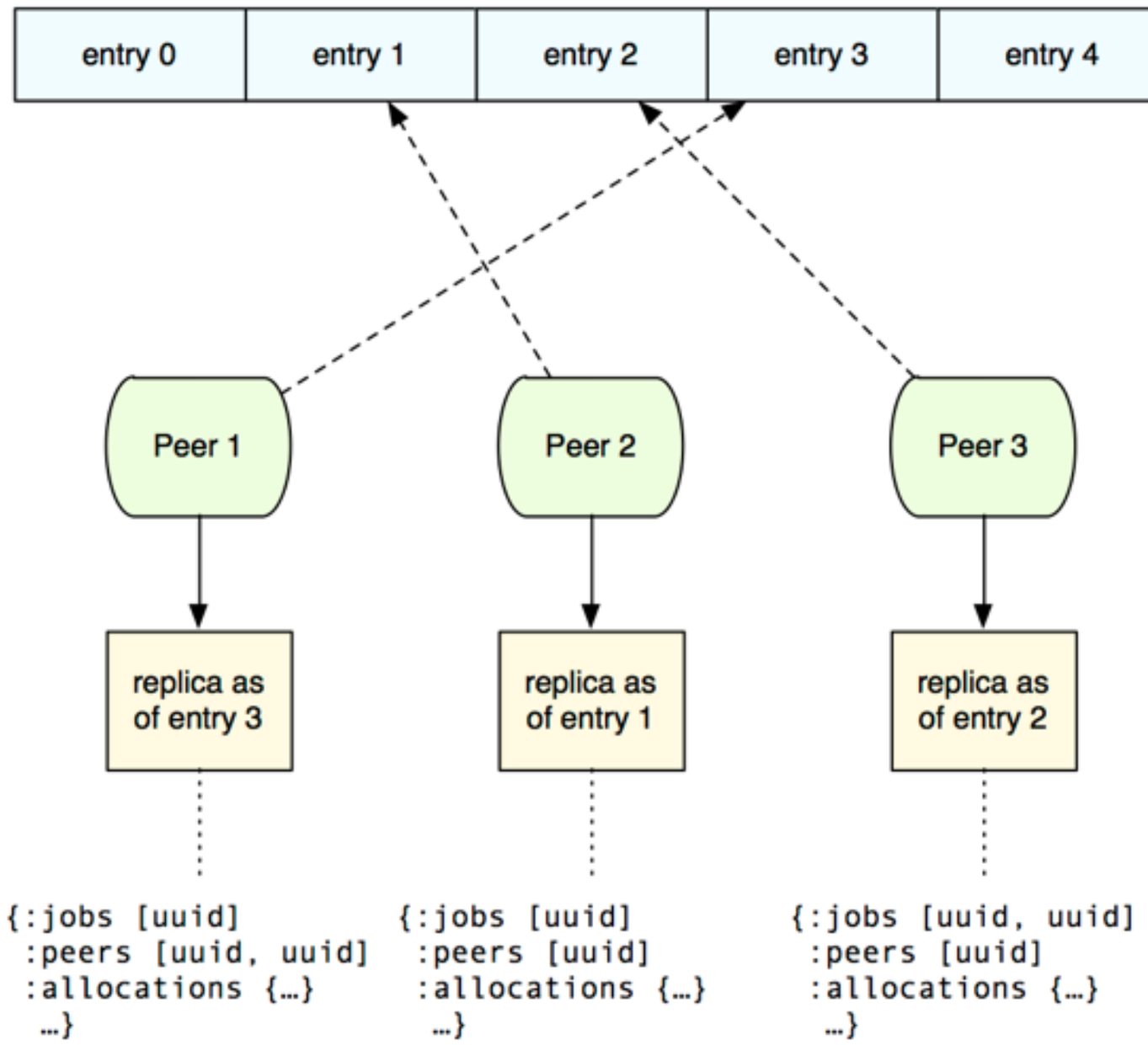


```
(defmulti apply-log-entry (λ [...] ...))

(defmulti replica-diff (λ [...] ...))

(defmulti fire-side-effects! (λ [...] ...))

(defmulti reactions (λ [...] ...))
```

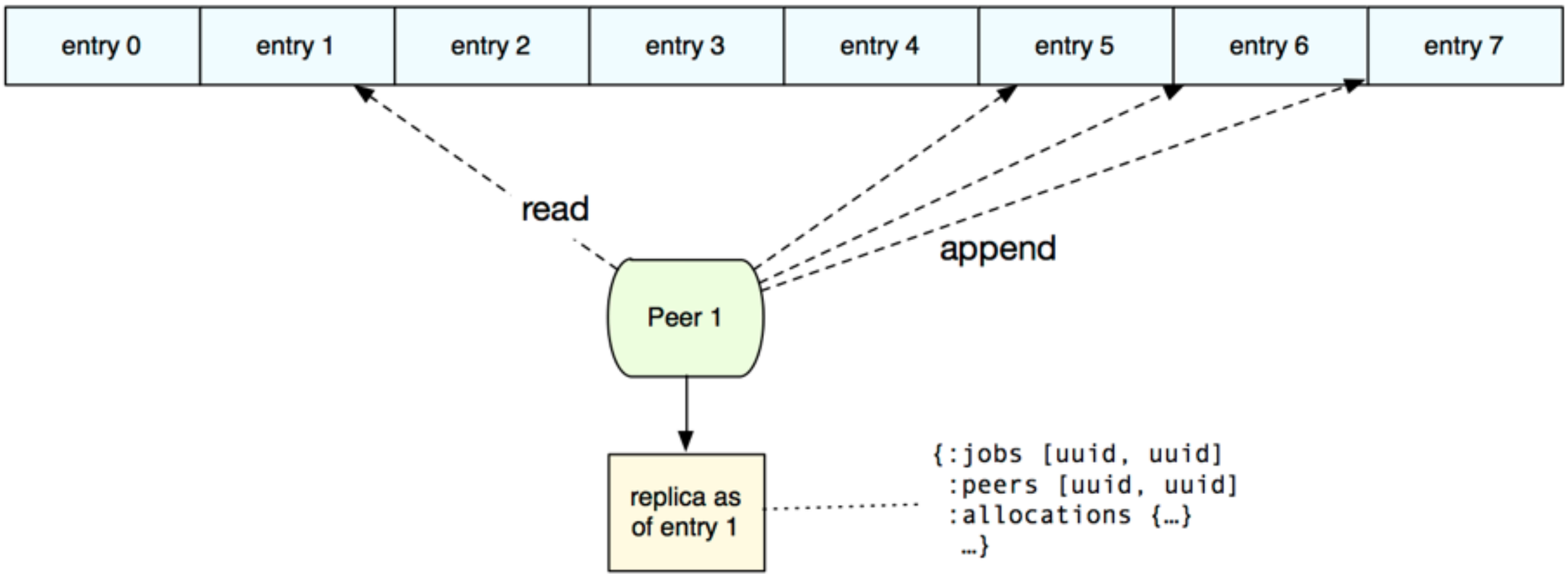
Log Entry Application



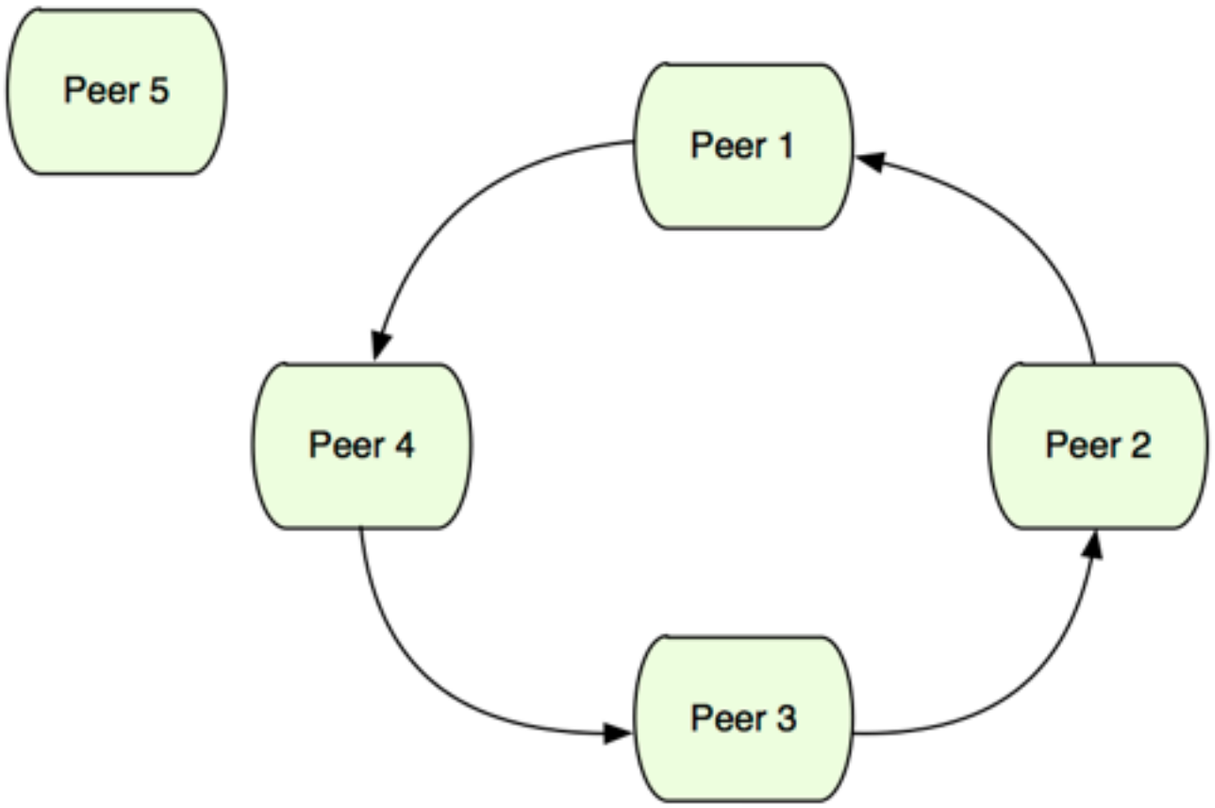
Independent Timelines



Reactive Protocol



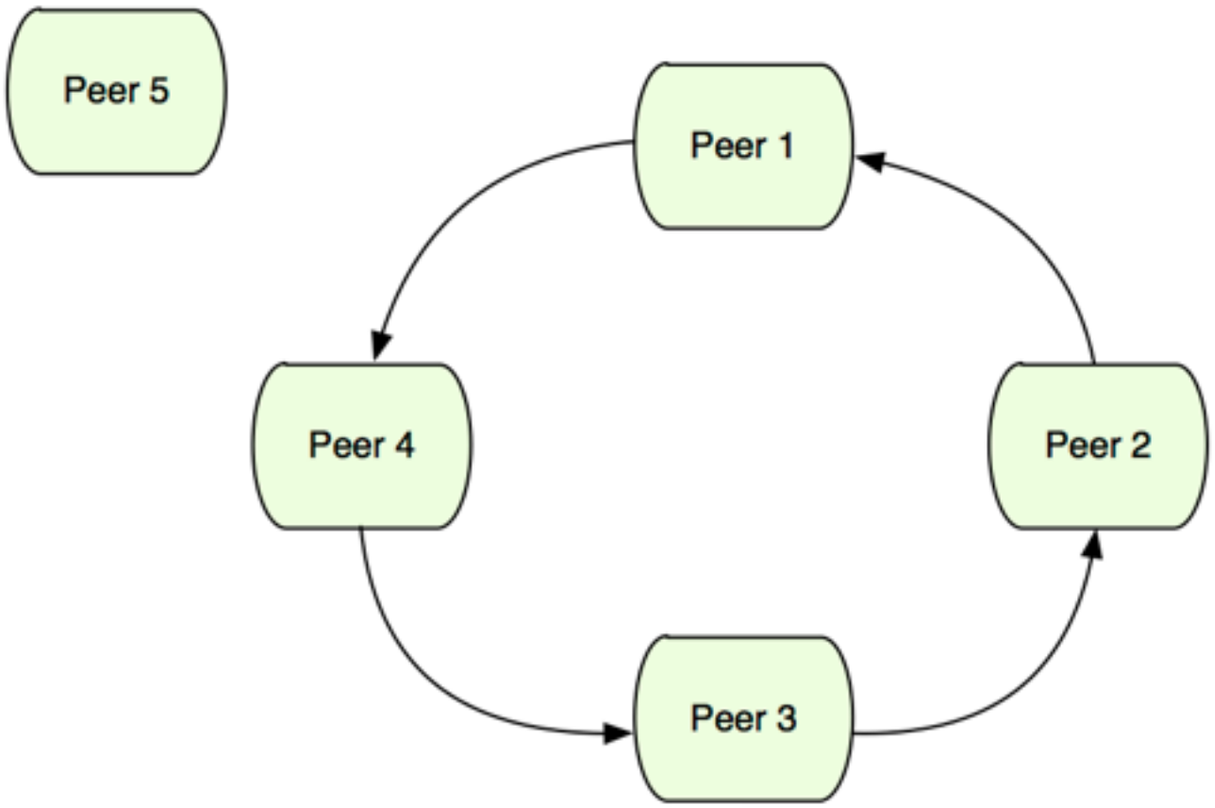
Detecting New Peers



t = 45

```
{:pairs {:peer-1 :peer-4
         :peer-4 :peer-3
         :peer-3 :peer-2
         :peer-2 :peer-1}}
```

Detecting New Peers

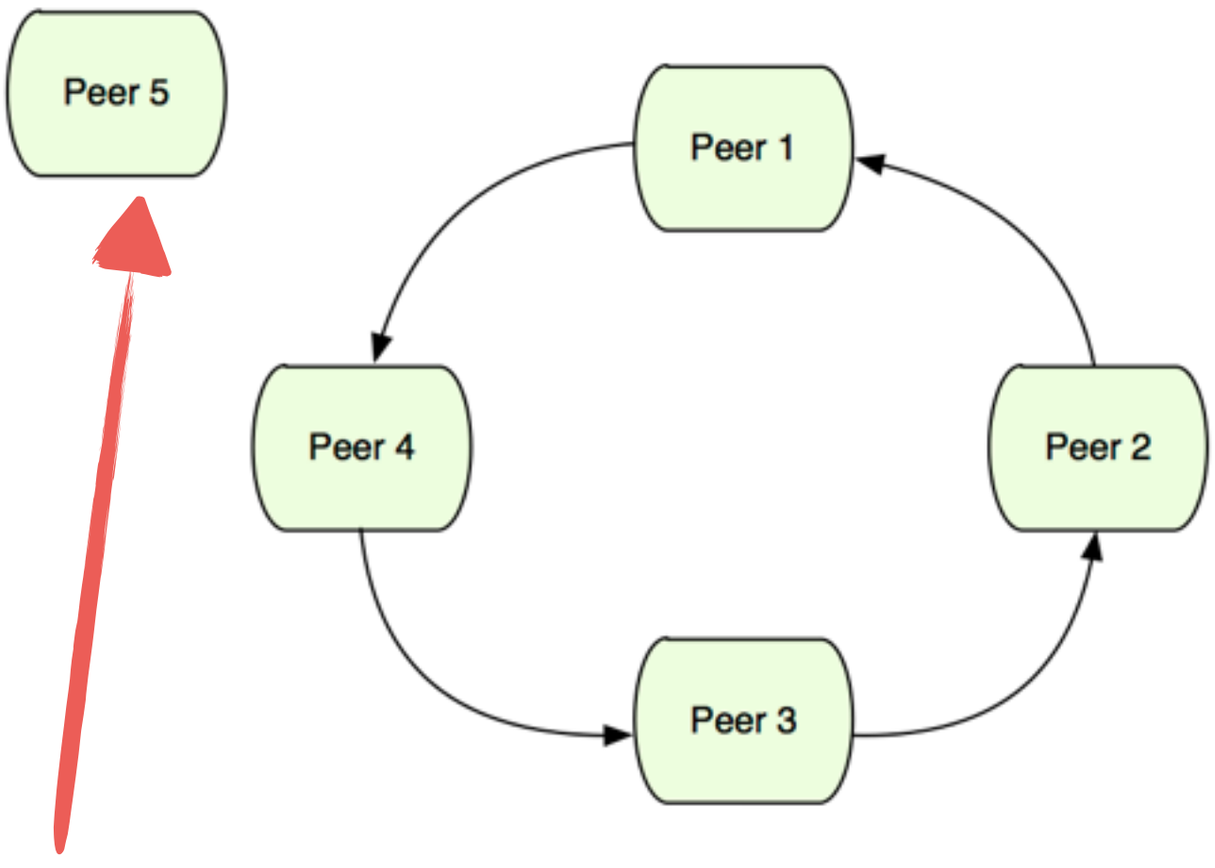


t = 45

```
{:pairs {:peer-1 :peer-4
         :peer-4 :peer-3
         :peer-3 :peer-2
         :peer-2 :peer-1}}
```

replica @ t=45

Detecting New Peers

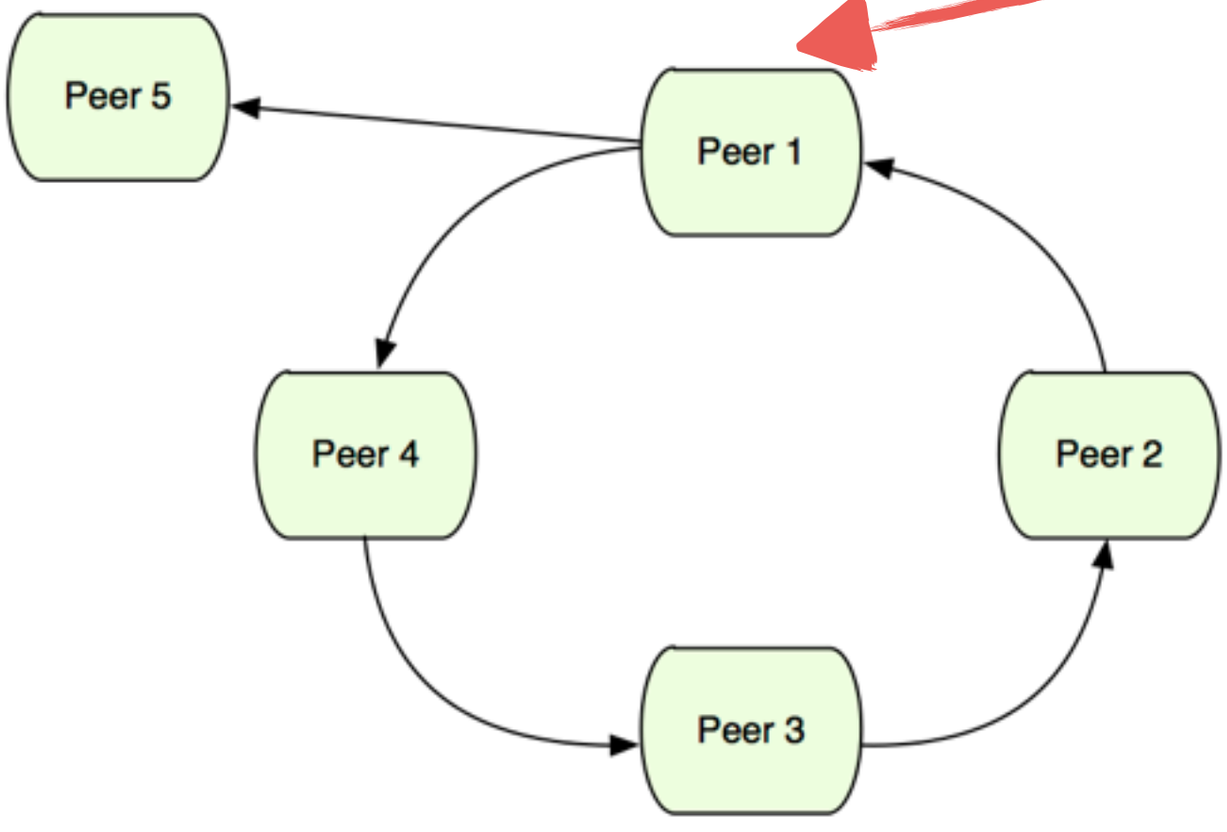


t = 45

```
{:pairs {:peer-1 :peer-4
         :peer-4 :peer-3
         :peer-3 :peer-2
         :peer-2 :peer-1}}
```

writes log entry

Phase 1: Prepare

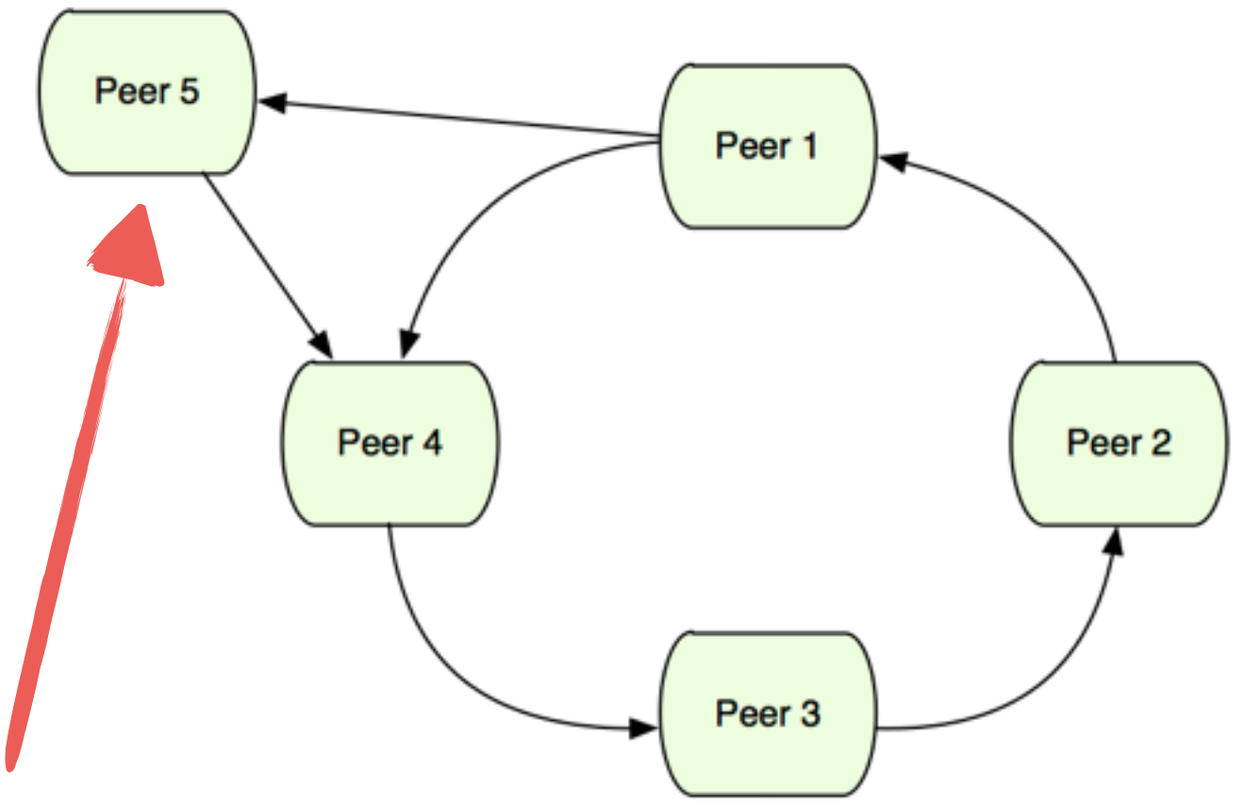


reads entry,
fires side effects,
writes entry

t = 47

```
{:pairs {:peer-1 :peer-4
         :peer-4 :peer-3
         :peer-3 :peer-2
         :peer-2 :peer-1
 :prepared {:peer-1 :peer-5}}
```

Phase 2: Accept

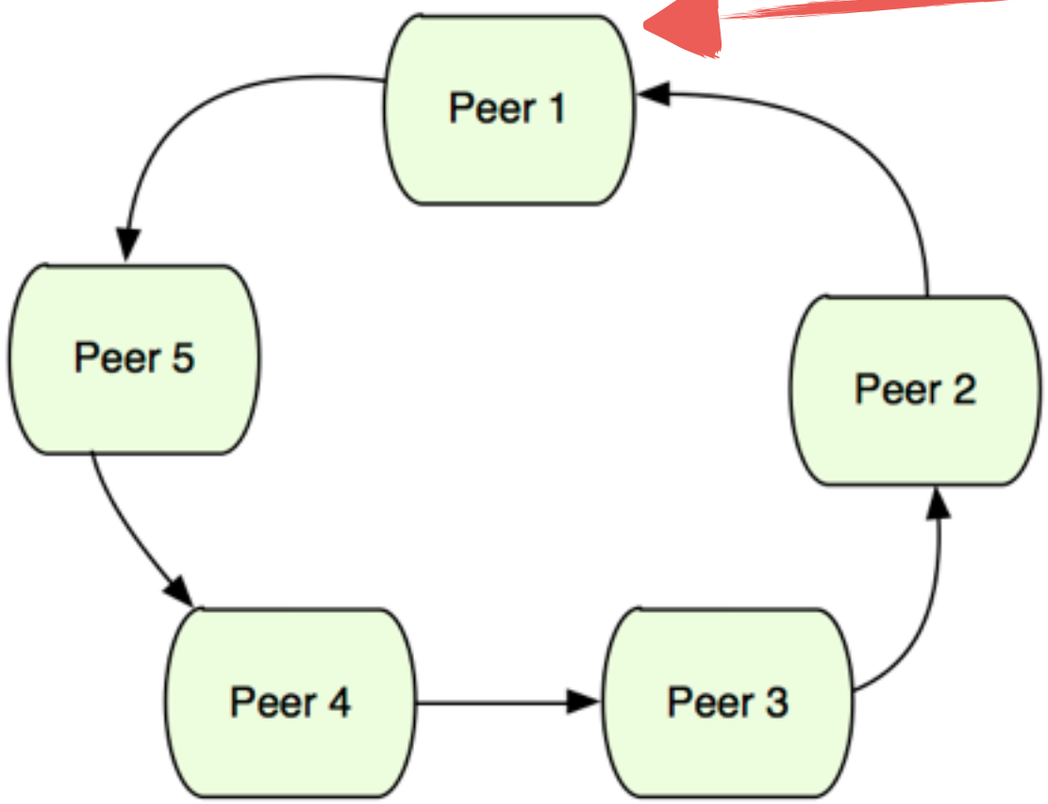


t = 50

```
{:pairs {:peer-1 :peer-4
         :peer-4 :peer-3
         :peer-3 :peer-2
         :peer-2 :peer-1
 :accepted {:peer-1 :peer-5}}
```

reads entry,
fires side effects,
writes entry

Phase 3: Stitch



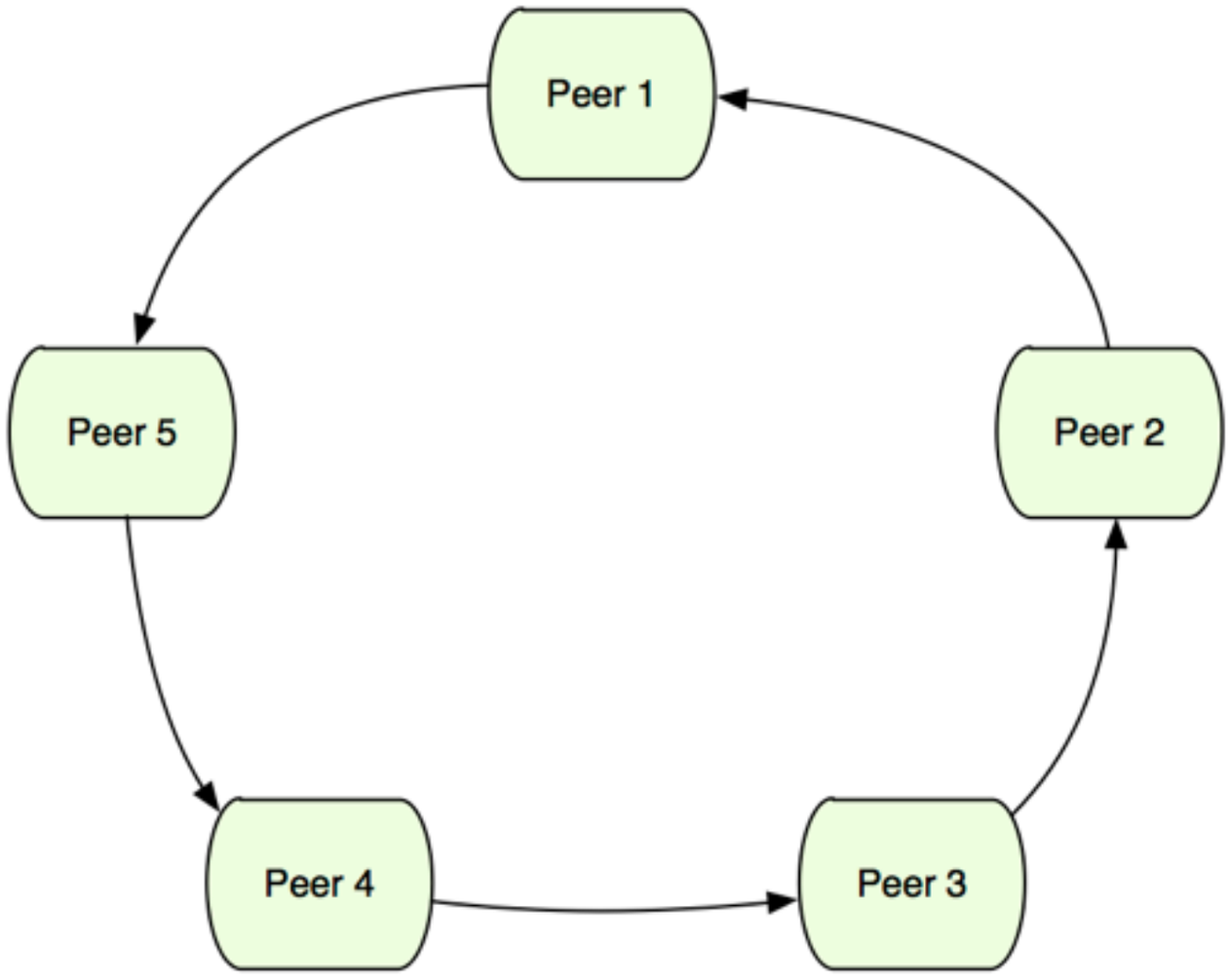
reads entry,
fires side effects

t = 51

```
{:pairs {:peer-1 :peer-5
         :peer-5 :peer-4
         :peer-4 :peer-3
         :peer-3 :peer-2
         :peer-2 :peer-1}}
```

balanced ring

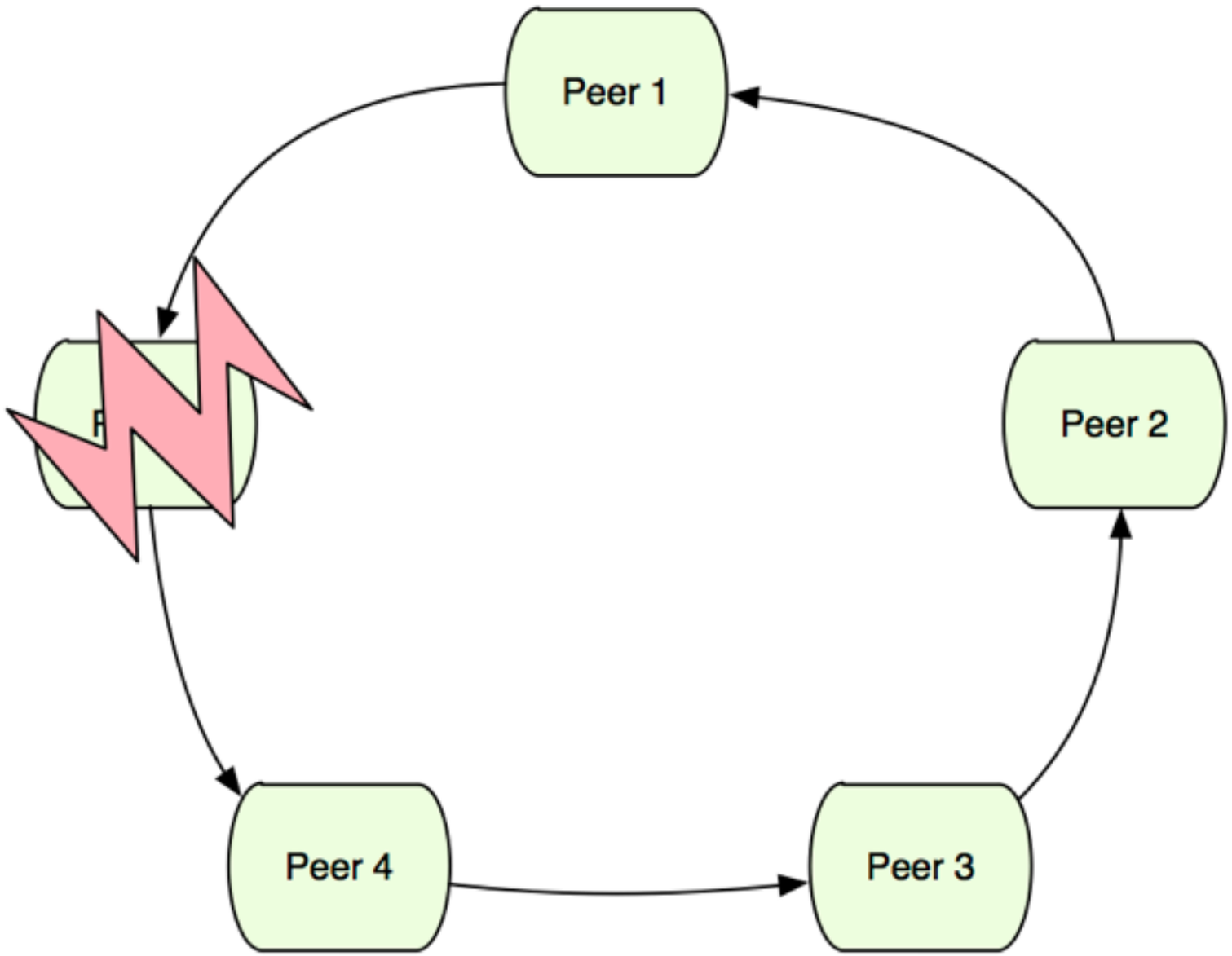
Failure Detection



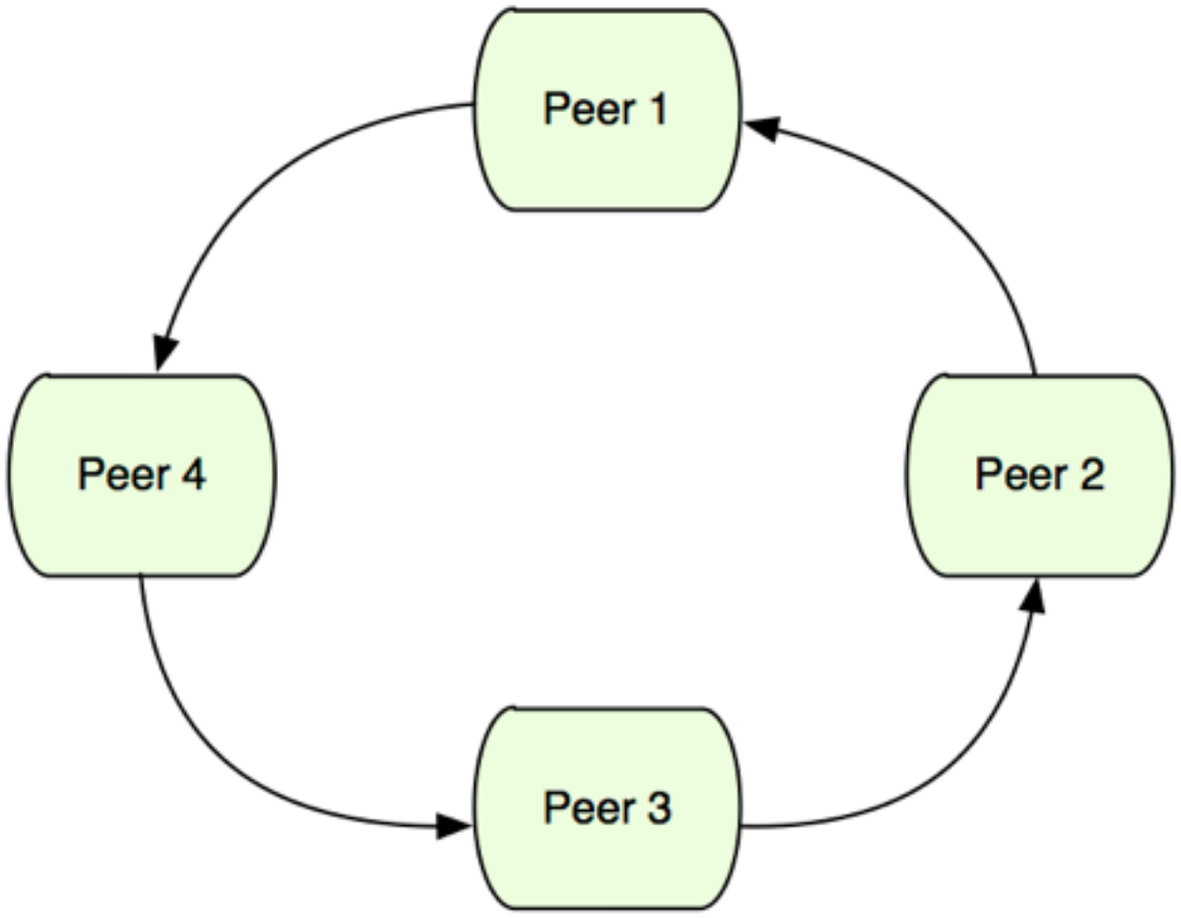
t = 42

```
{:pairs {:peer-1 :peer-5
         :peer-5 :peer-4
         :peer-4 :peer-3
         :peer-3 :peer-2
         :peer-2 :peer-1}}
```

Failure Detection



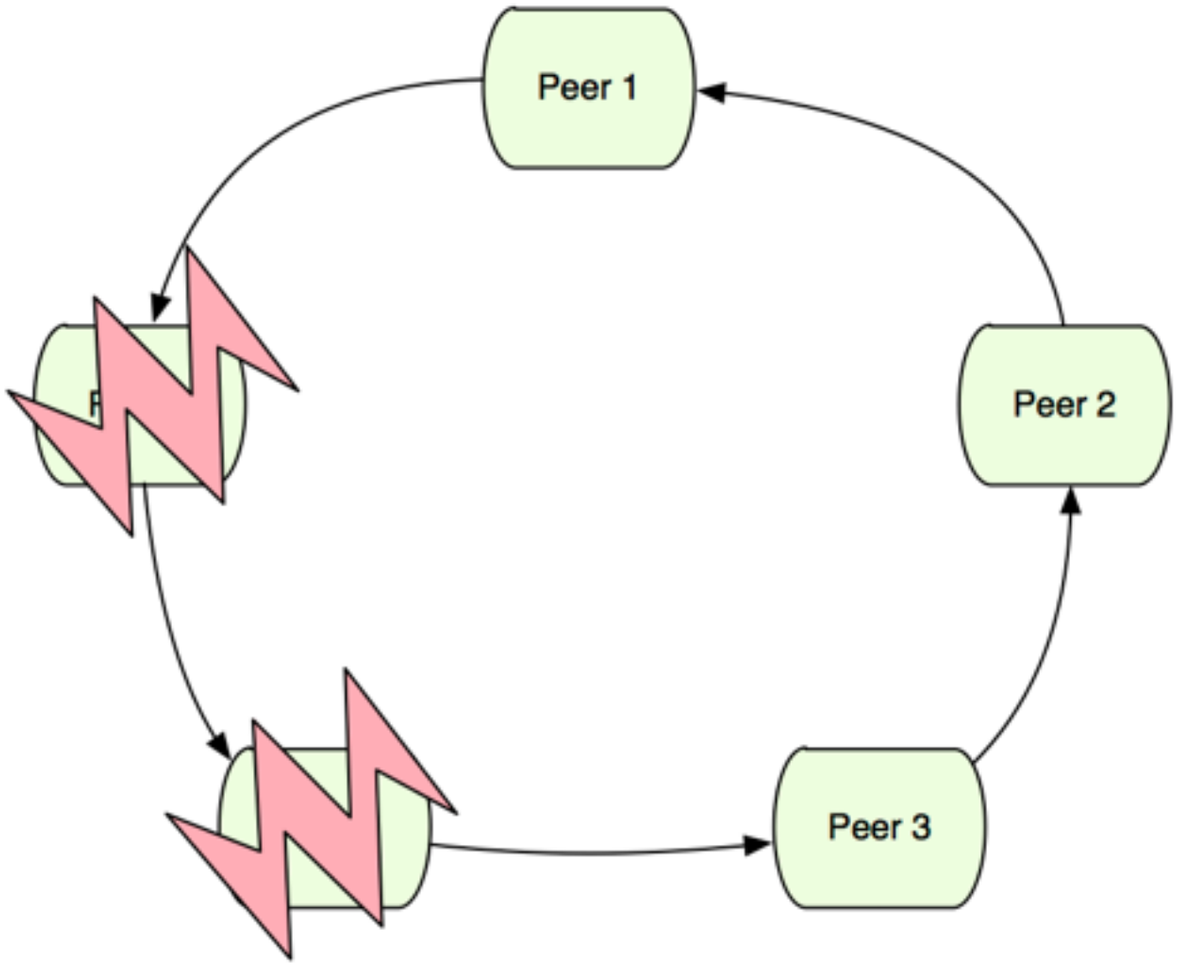
Failure Detection



t = 45

```
{:pairs {:peer-1 :peer-4
         :peer-4 :peer-3
         :peer-3 :peer-2
         :peer-2 :peer-1}}
```

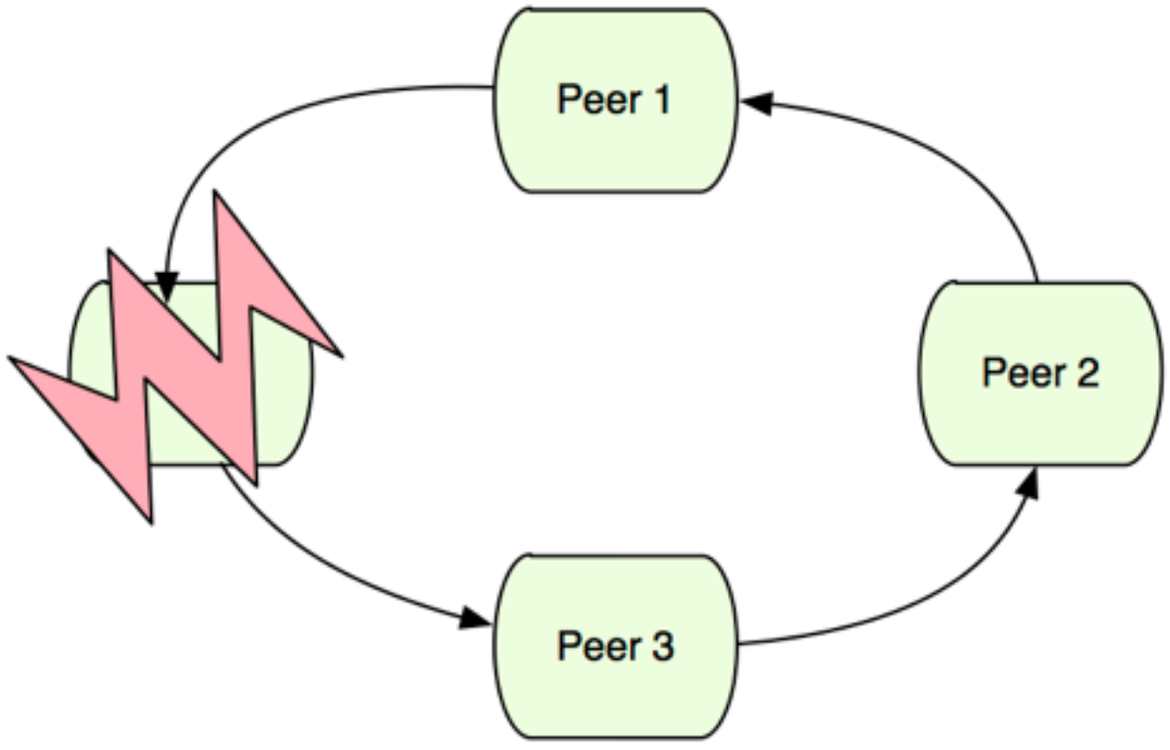
Failure Detection



t = 42

```
{:pairs {:peer-1 :peer-5
         :peer-5 :peer-4
         :peer-4 :peer-3
         :peer-3 :peer-2
         :peer-2 :peer-1}}
```

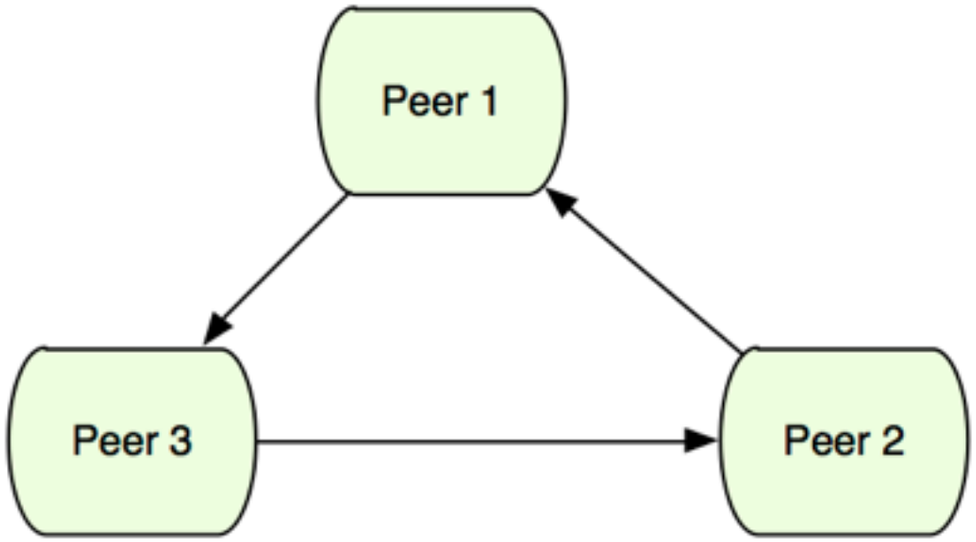
Failure Detection



t = 48

```
{:pairs {:peer-1 :peer-4
         :peer-4 :peer-3
         :peer-3 :peer-2
         :peer-2 :peer-1}}
```

Failure Detection



t = 50

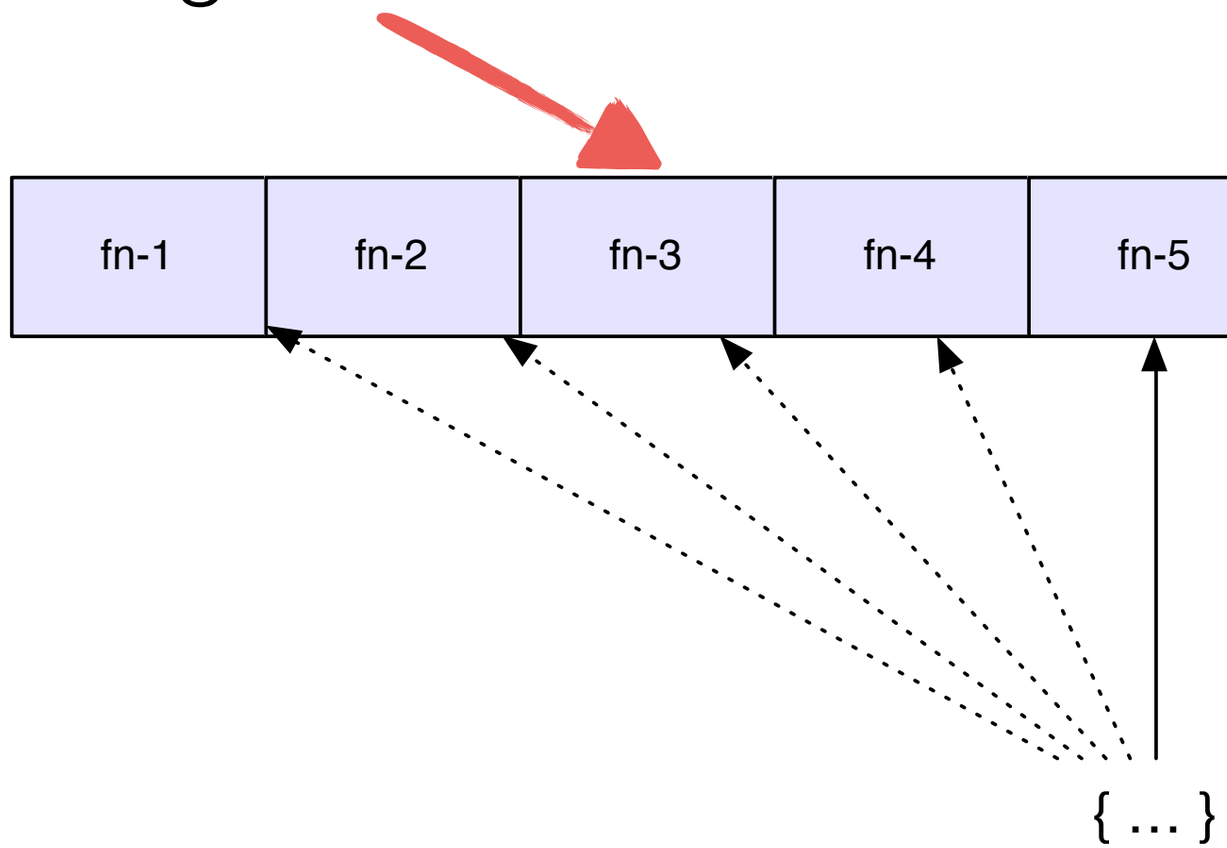
```
{:pairs {:peer-1 :peer-3
         :peer-3 :peer-2
         :peer-2 :peer-1}}
```

How do we test it?

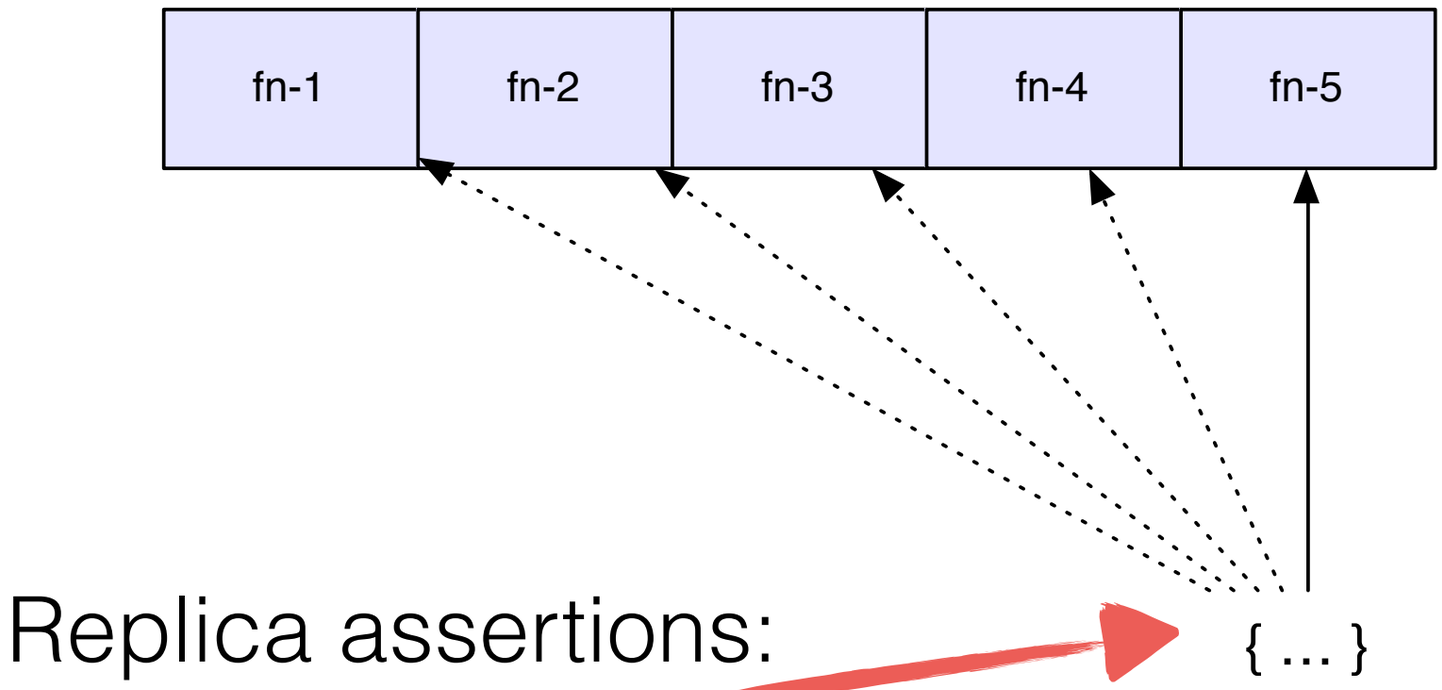
- Generative testing
- Generate seed log entries
- Allow peers to react until peers stop appending to log
- Property-based assertions on final replica
- Transcends time - no concurrency in model
- Our program is a time machine

Successful Generative Test

Generated log entries



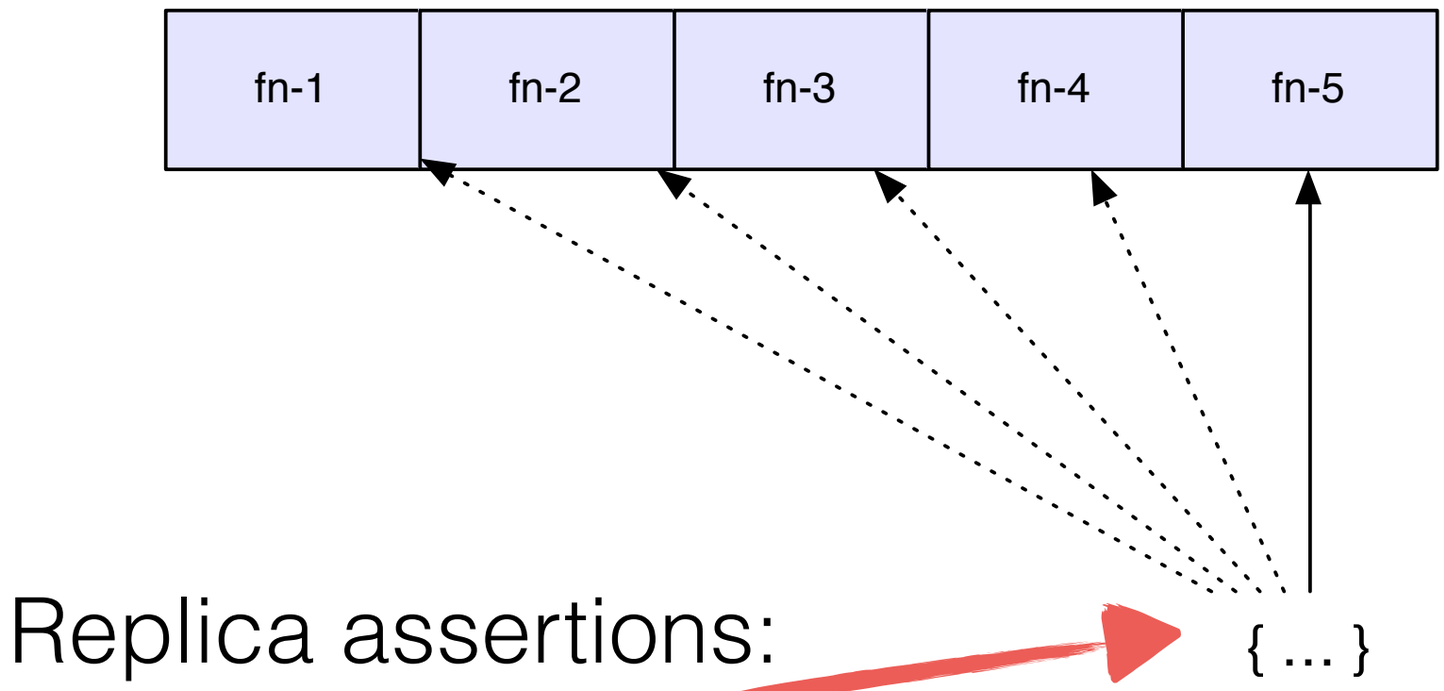
Successful Generative Test



Replica assertions:

pass

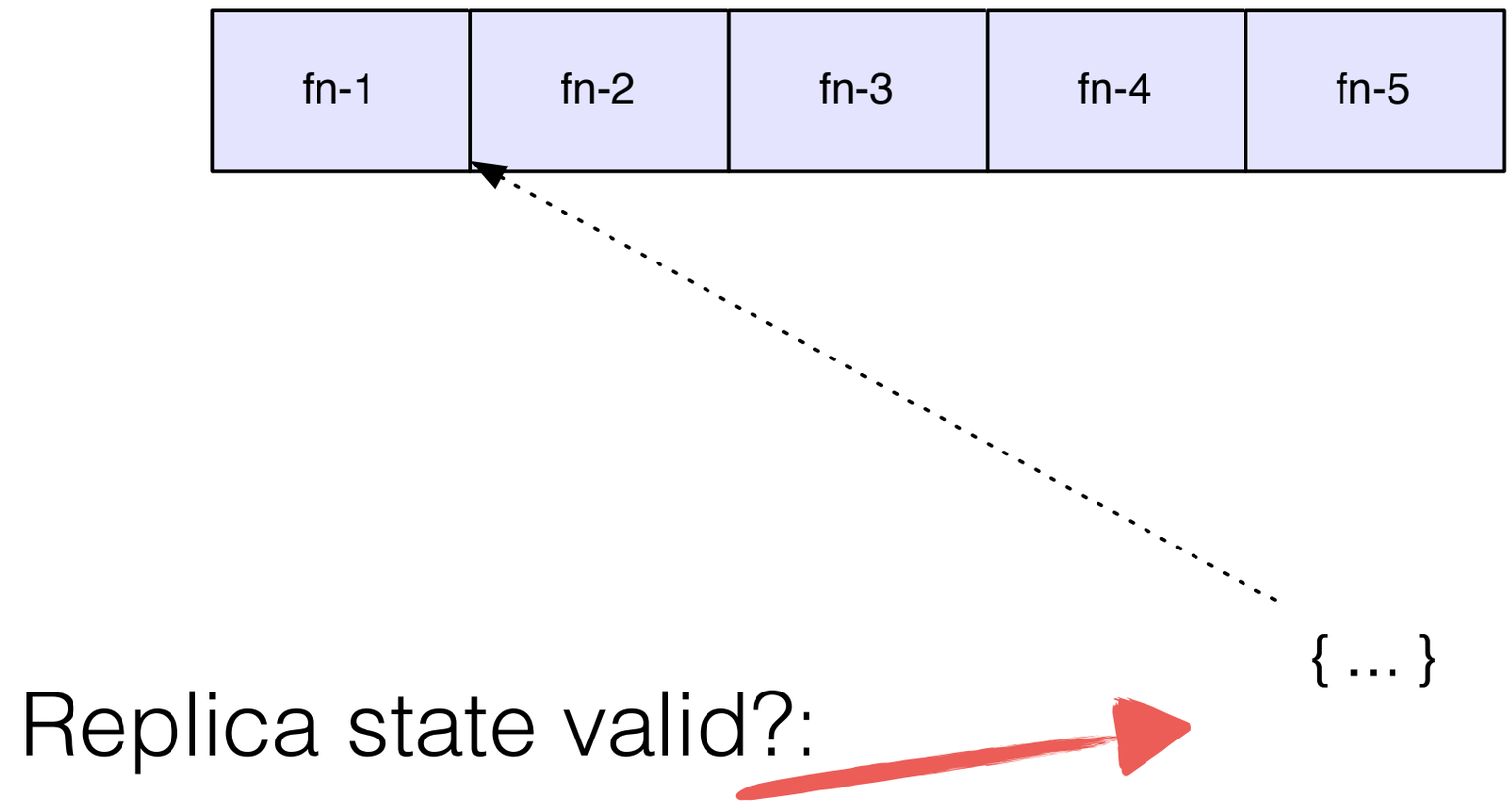
Time Machine Debugging



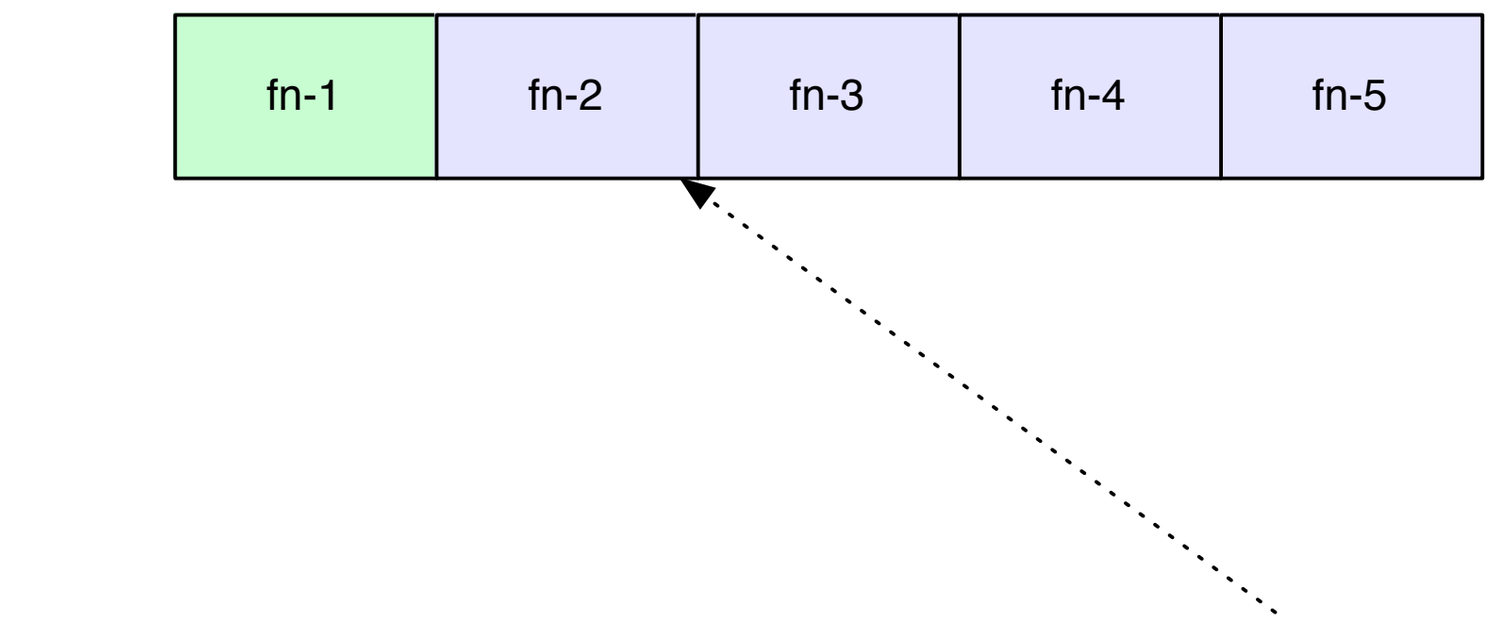
Replica assertions:

fail

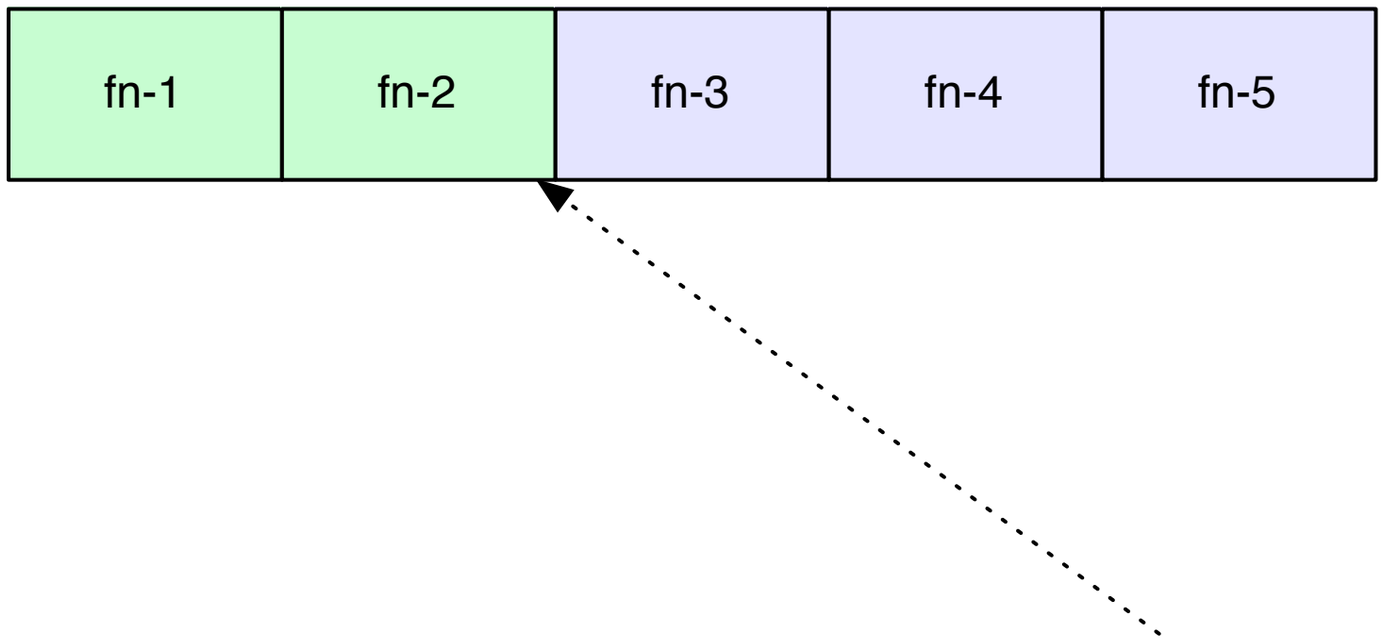
Time Machine Debugging



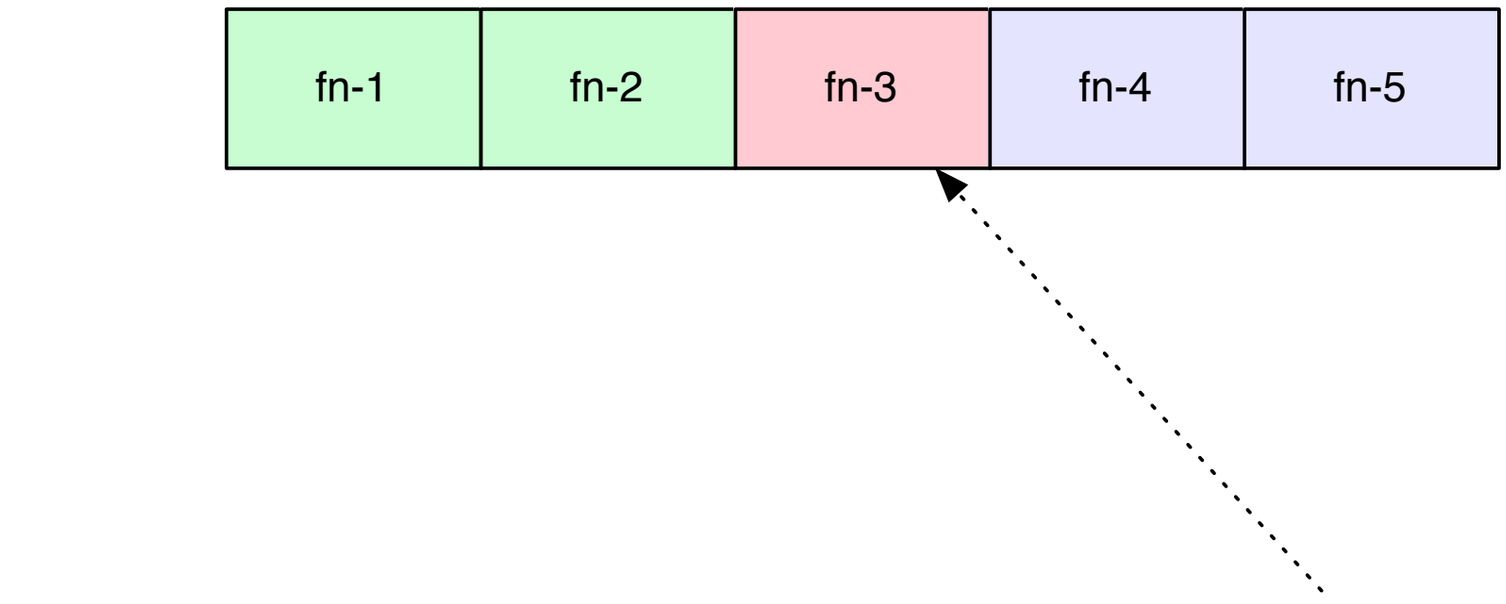
Time Machine Debugging



Time Machine Debugging



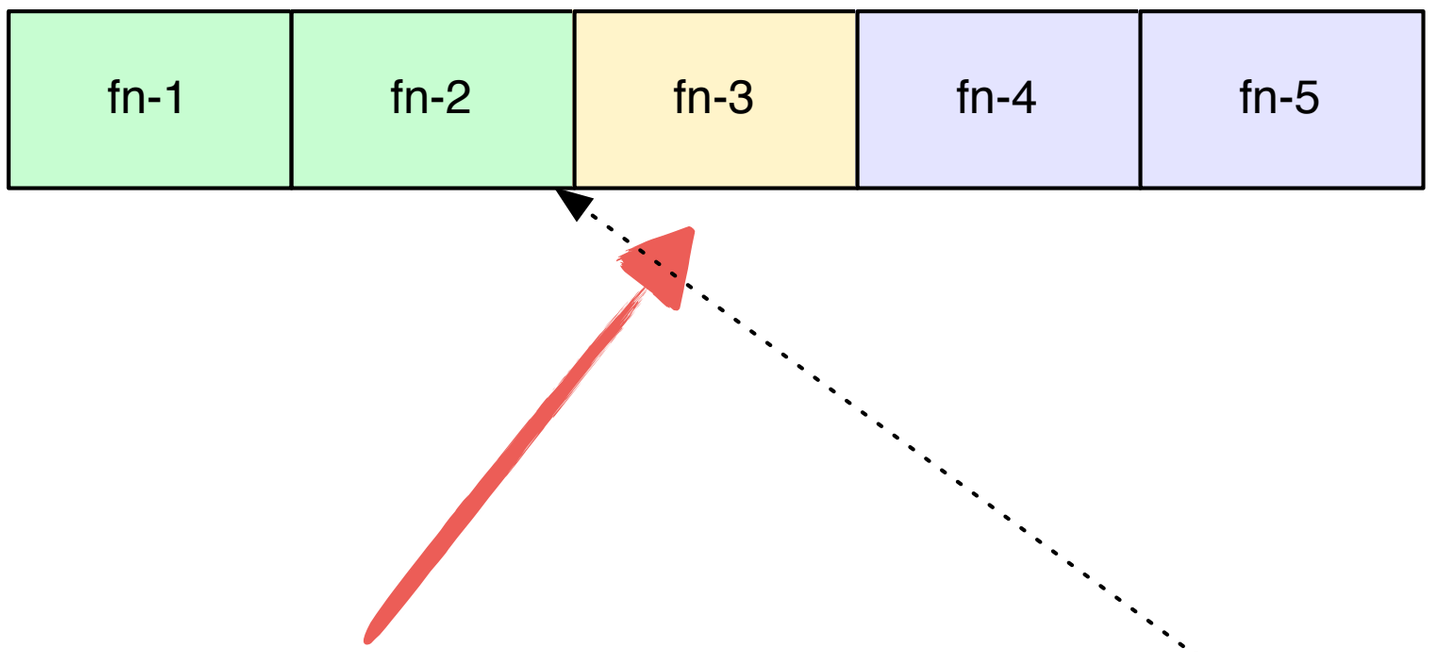
Time Machine Debugging



Replica state valid?:

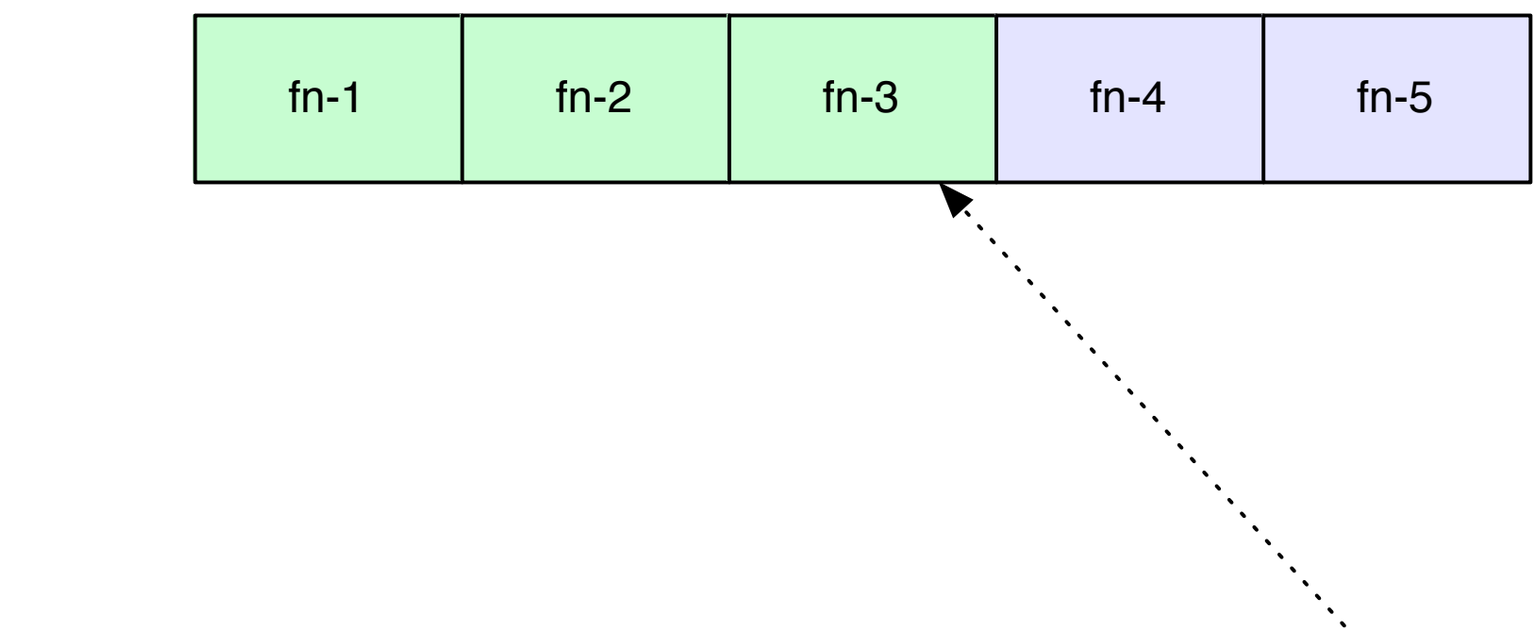
no

Time Machine Debugging

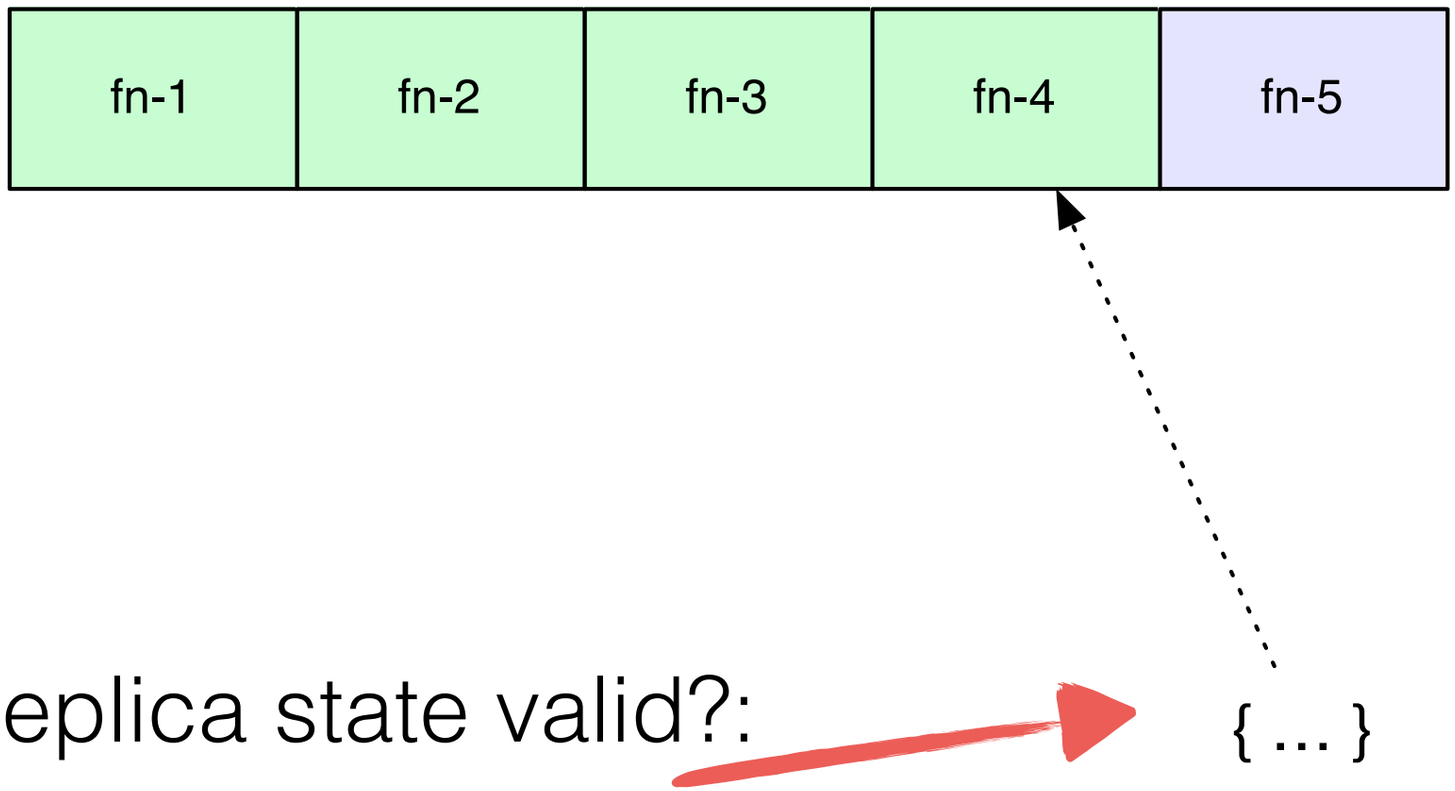


Back up the log and
patch the fn-3 code,
make a unit test

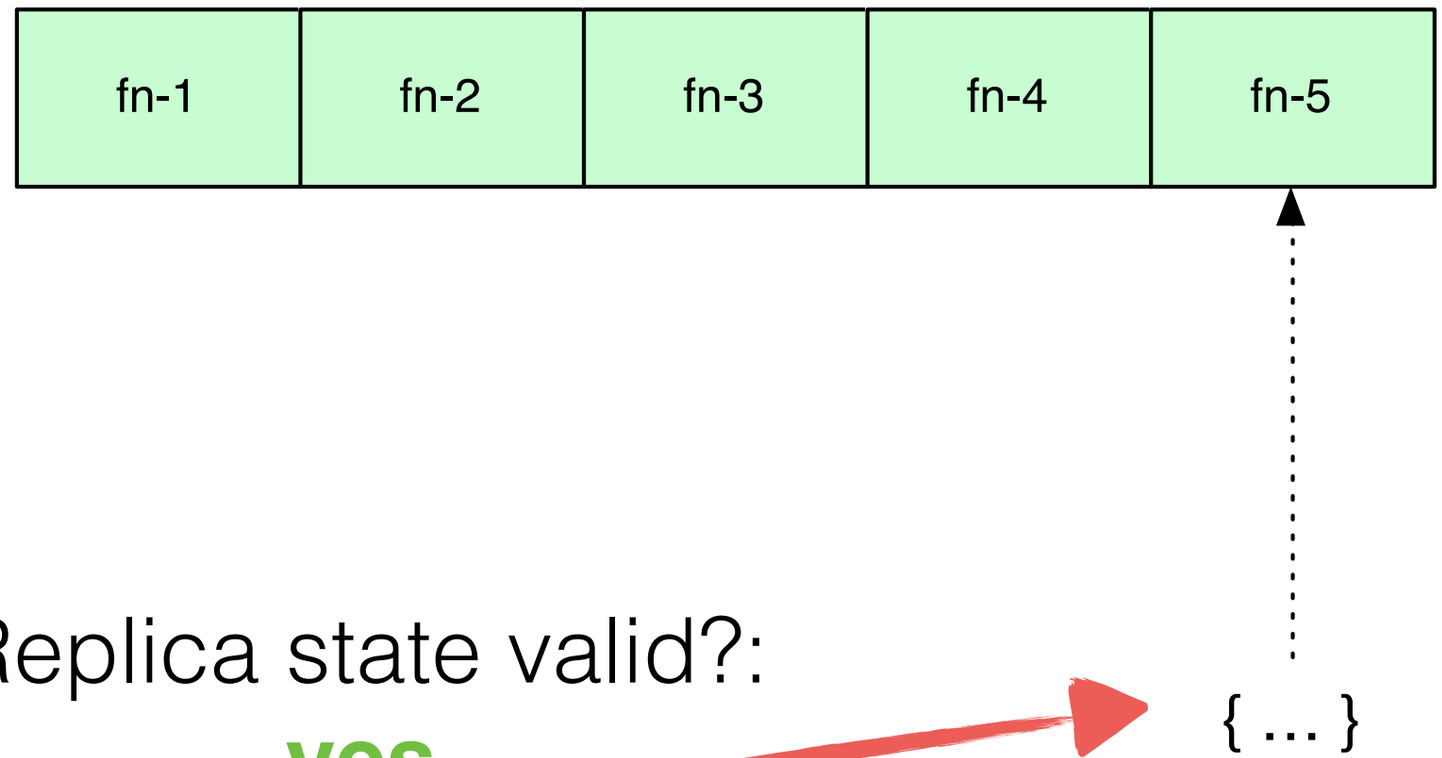
Time Machine Debugging



Time Machine Debugging



Time Machine Debugging



Replica state valid?:

yes

Bug fixed

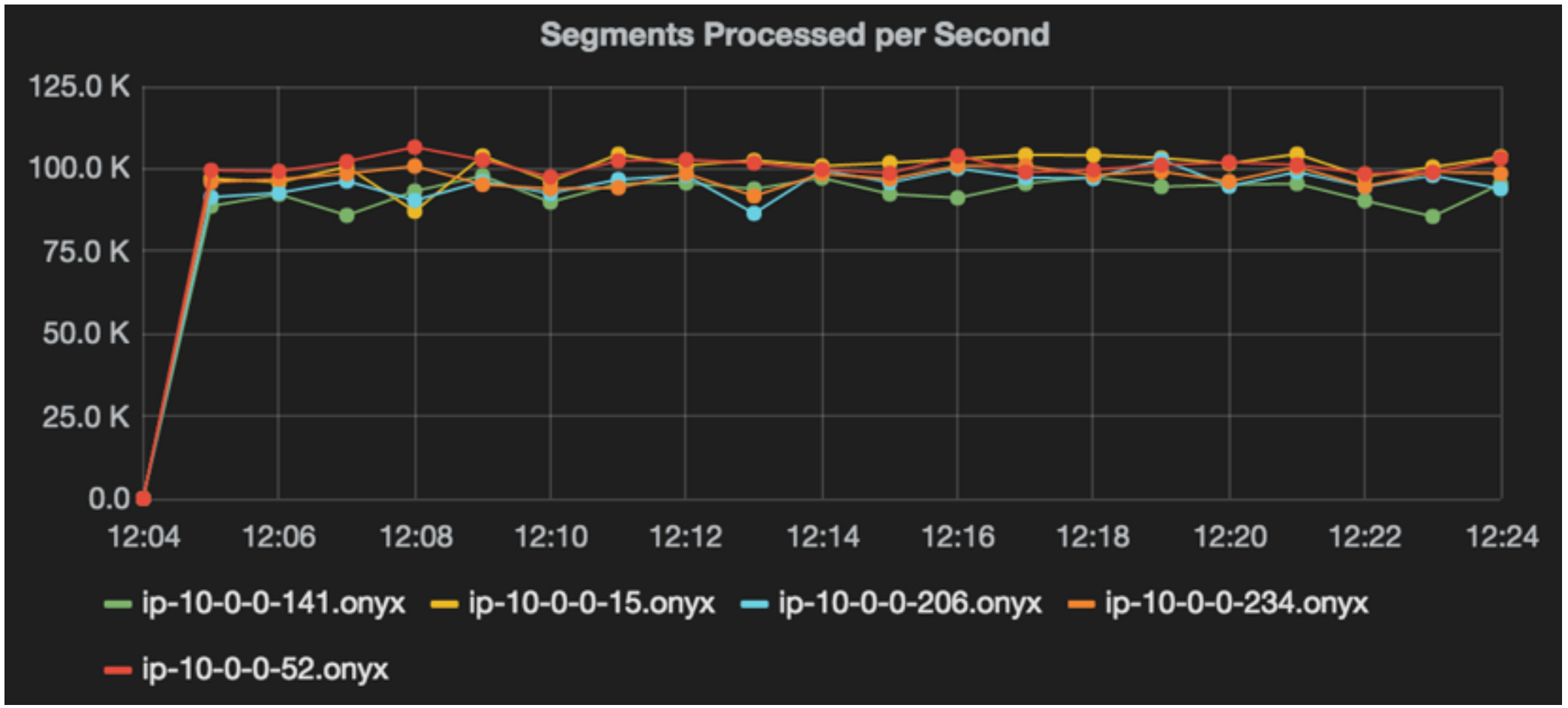
Other uses of the Log

- Allocation of peers to task is a pure function
- Distributed, coordinated Blade garbage collector
- Coordinated backpressure with high water marks
- User-level CloudWatch alarms

State of Onyx

- Talks to: Datomic, SQL, Kafka, S3, Durable Queue, core.async
- Emits metrics to Onyx Dashboard & streams
- Automated benchmark suite
- Production-grade performance

Onyx is Fast



5*c4.xlarge AWS VMs

Thanks

- Try: `lein new onyx-app my-master-piece`
- Learn: https://github.com/onyx-platform/onyx
- Chat: https://gitter.im/onyx-platform/onyx