Storm

Distributed and fault-tolerant realtime computation

Nathan Marz
Twitter

Basic info

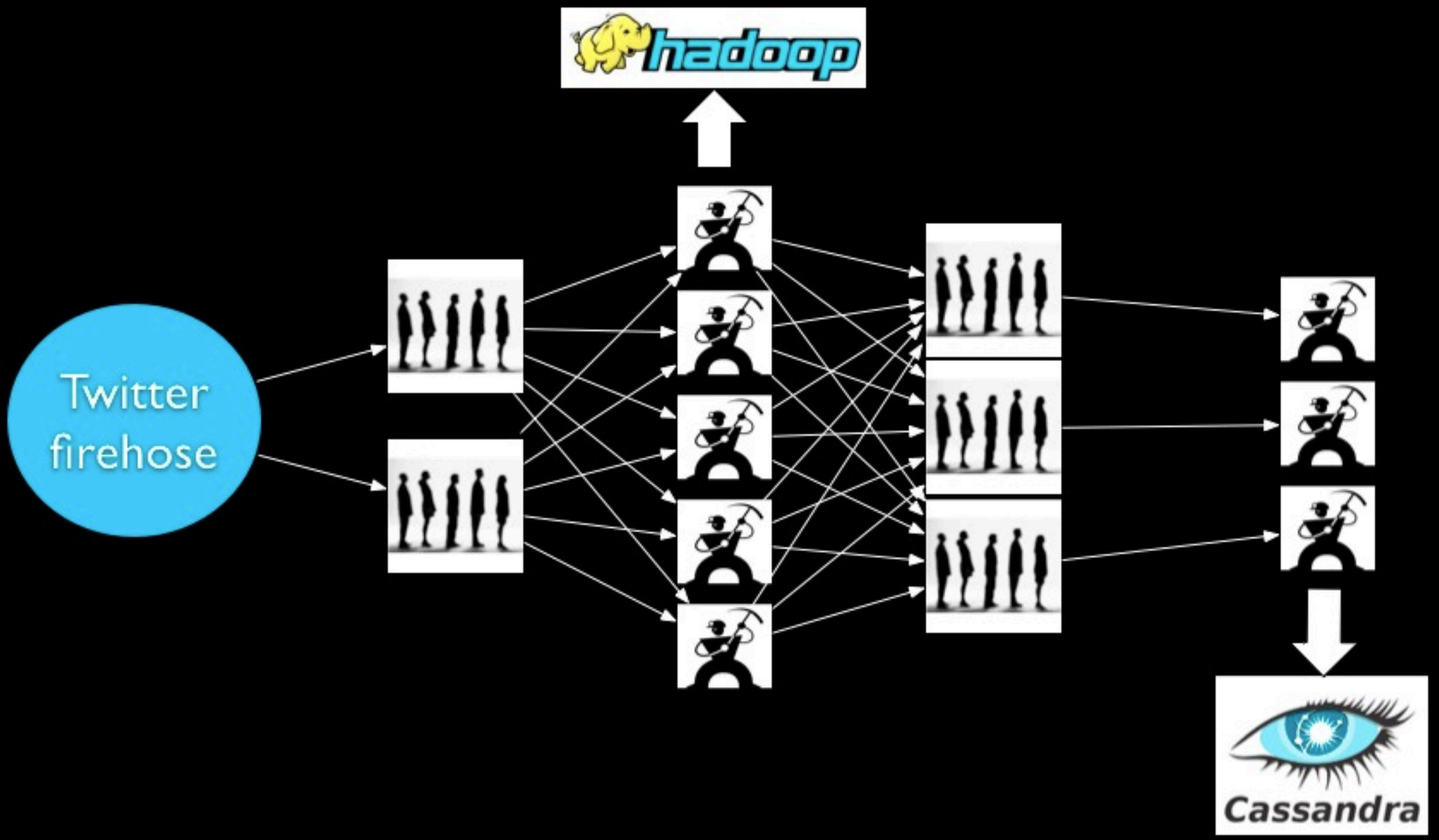
- Open sourced September 19th
- Implementation is 15,000 lines of code
- Used by over 25 companies
- >2700 watchers on Github (most watched JVM project)
- Very active mailing list
  - >2300 messages
  - >670 members

Before Storm

Queues

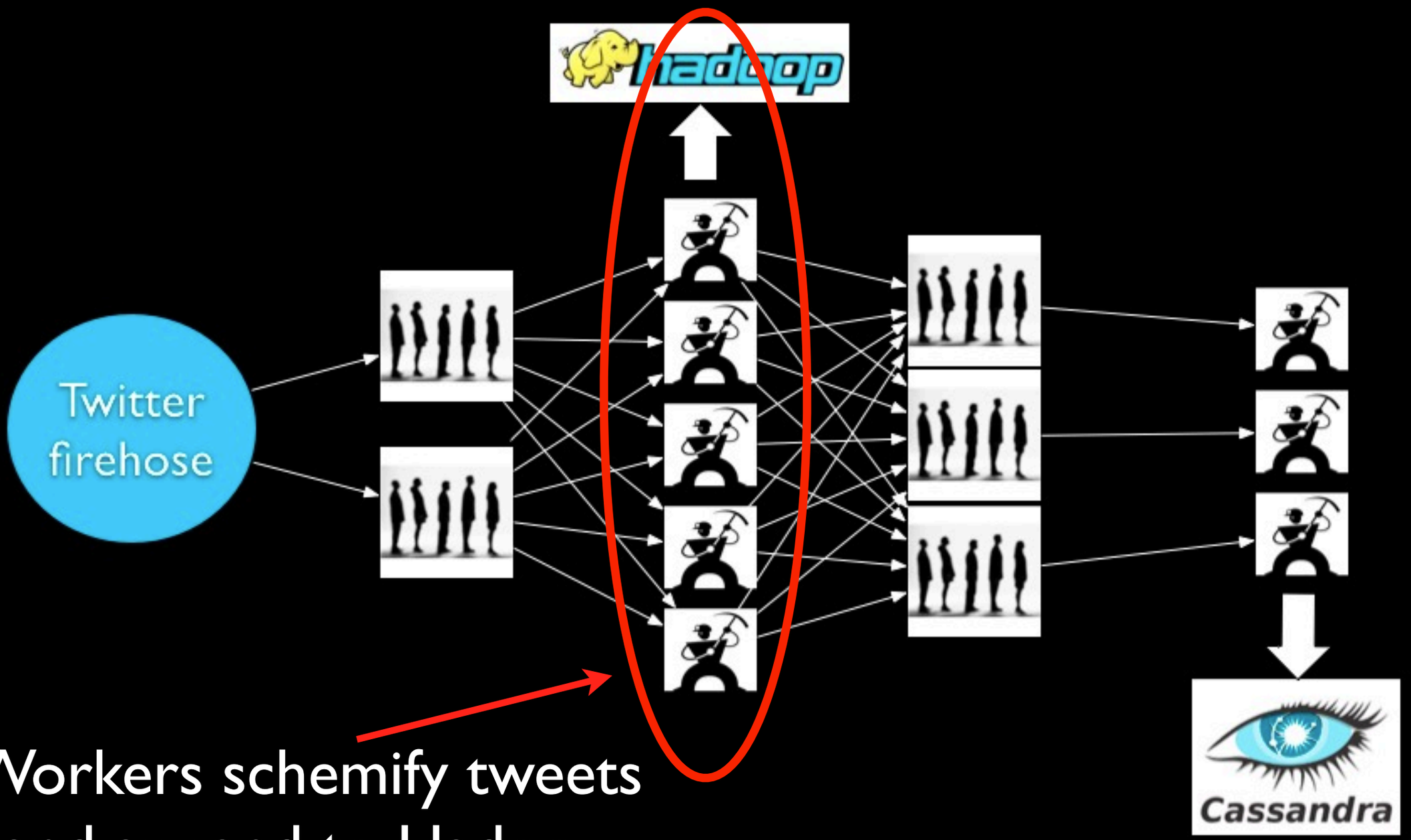
Workers

Example



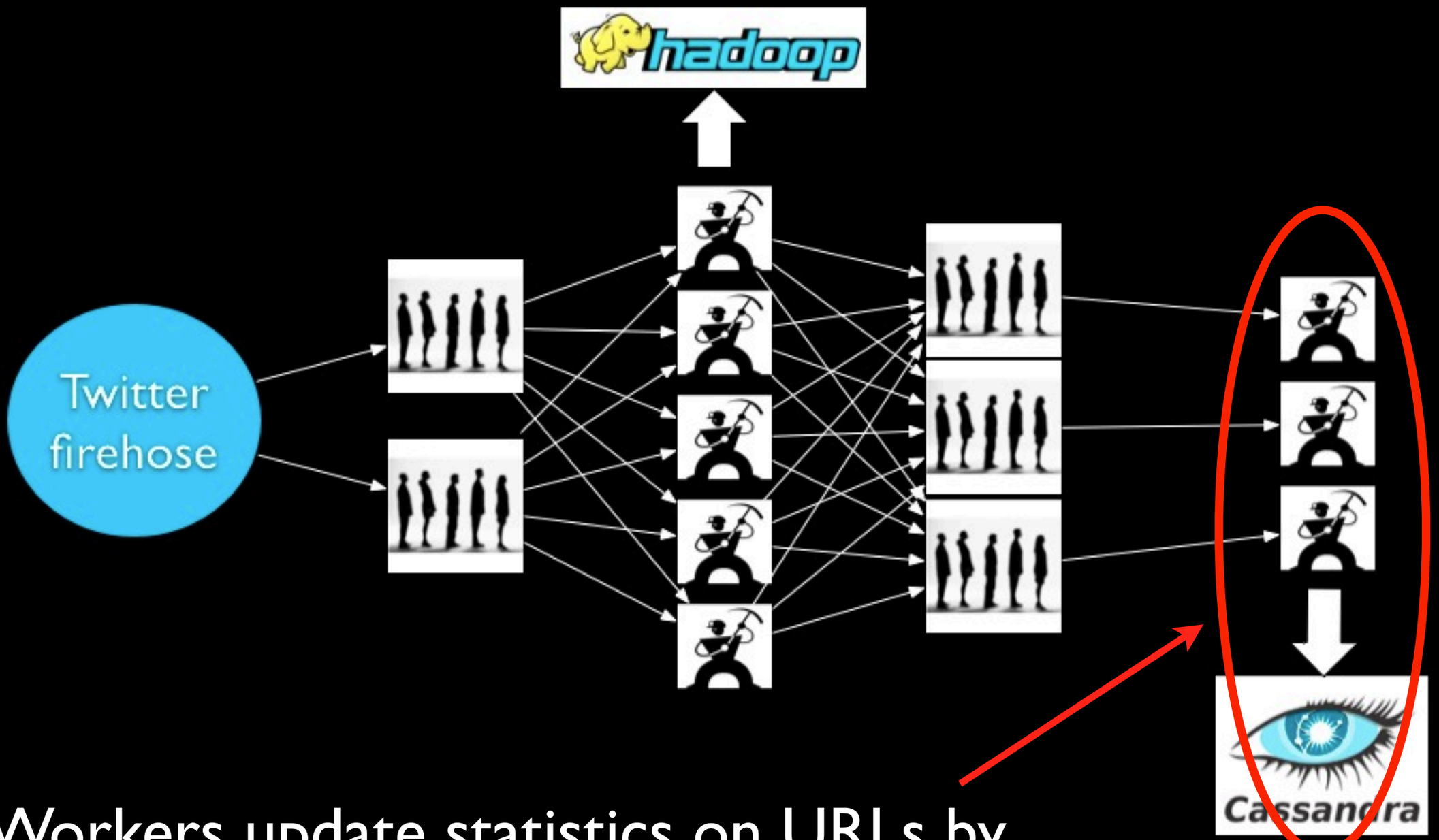
(simplified)

Example



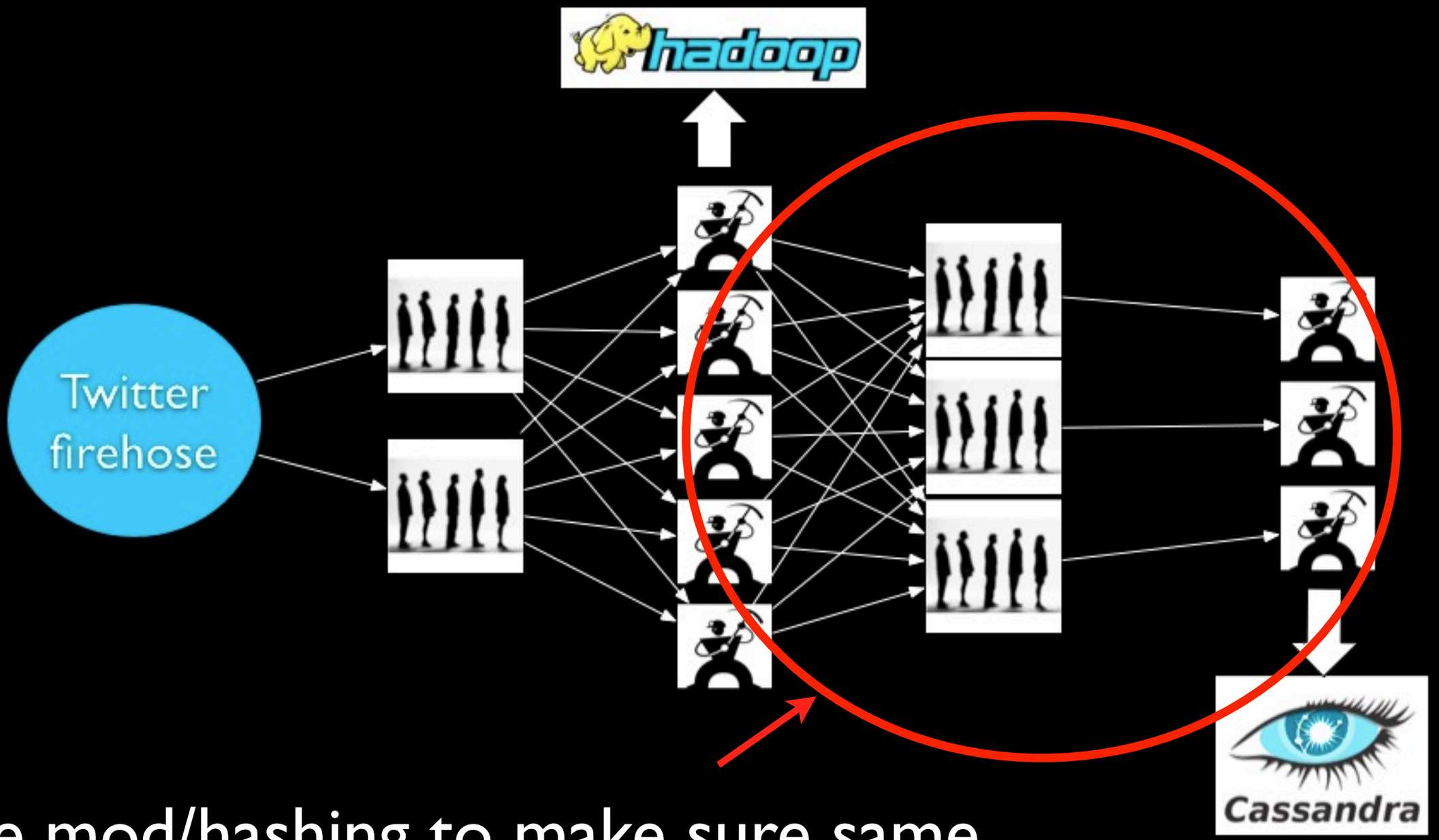
Workers schemify tweets and append to Hadoop

Example



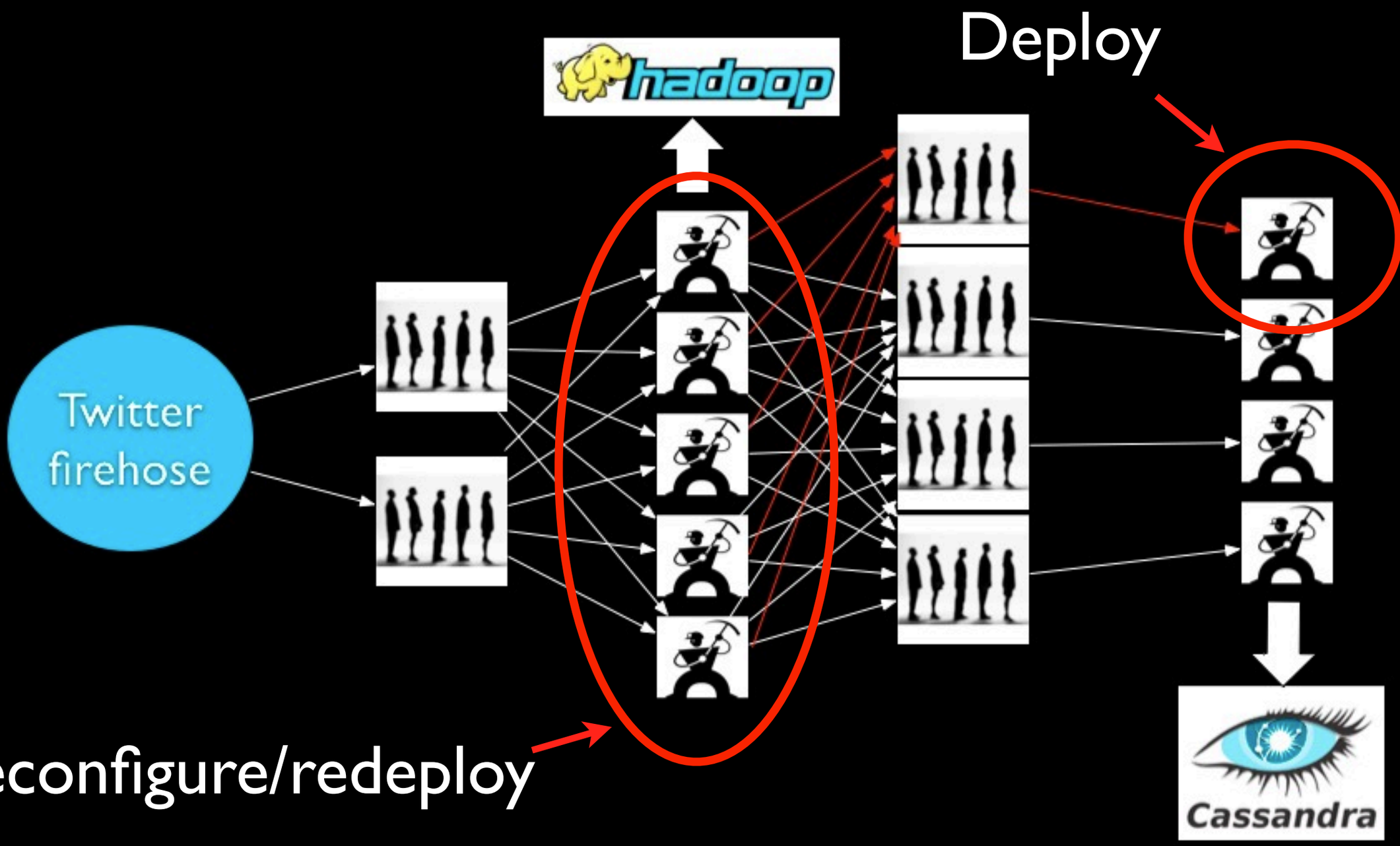
Workers update statistics on URLs by incrementing counters in Cassandra

Example



Use mod/hashing to make sure same URL always goes to same worker

Scaling


Deploy
hadoop
Twitter firehose
Reconfigure/redeploy
Cassandra

Problems

- Scaling is painful
- Poor fault-tolerance
- Coding is tedious

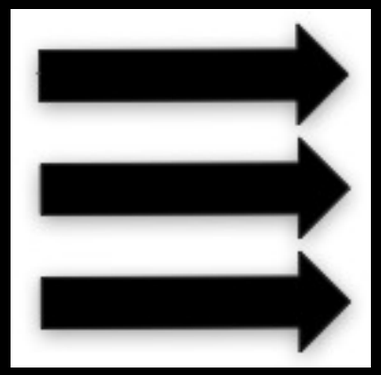
What we want

- Guaranteed data processing
- Horizontal scalability
- Fault-tolerance
- No intermediate message brokers!
- Higher level abstraction than message passing
- “Just works”

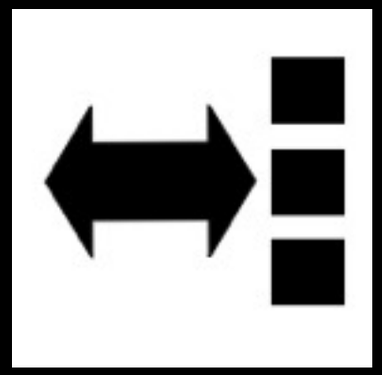
Storm

- ✔ Guaranteed data processing
- ✔ Horizontal scalability
- ✔ Fault-tolerance
- ✔ No intermediate message brokers!
- ✔ Higher level abstraction than message passing
- ✔ “Just works”

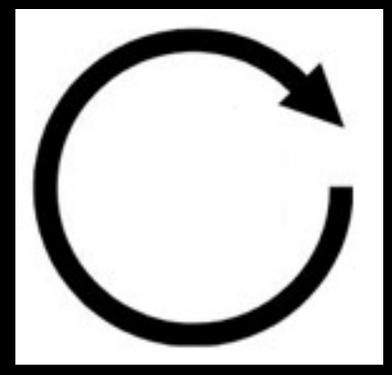
Use cases



Stream processing

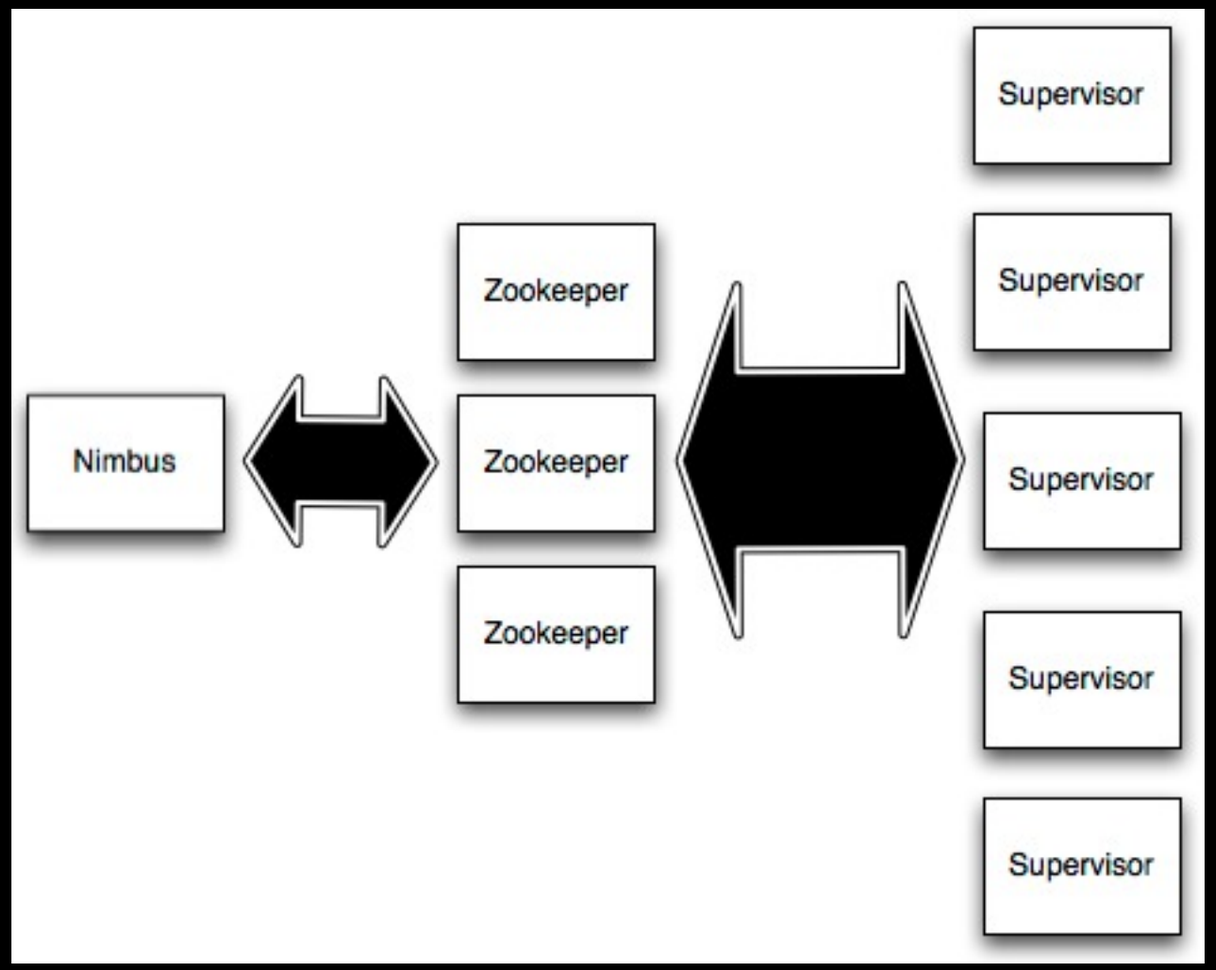


Distributed RPC

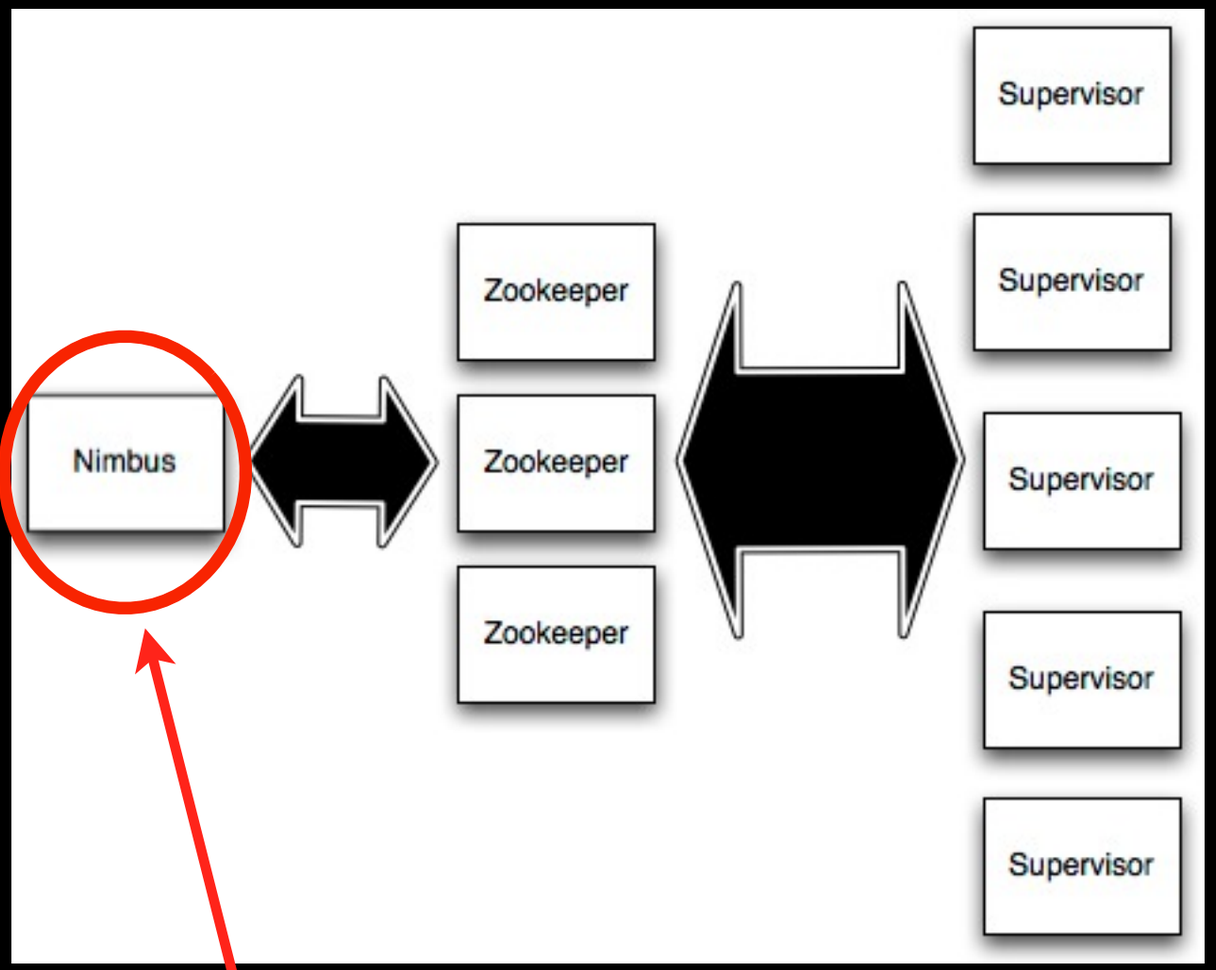


Continuous computation

Storm Cluster

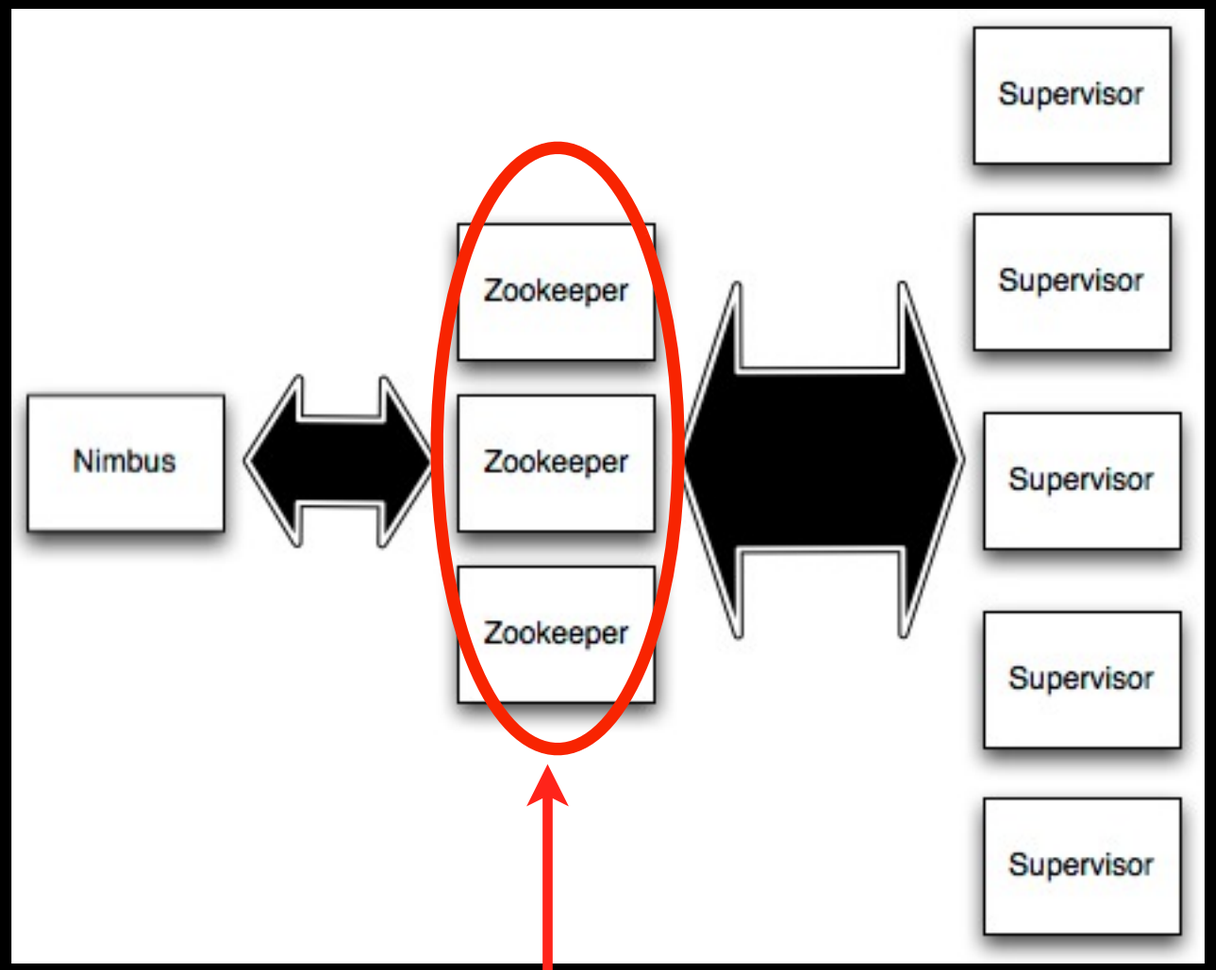


Storm Cluster



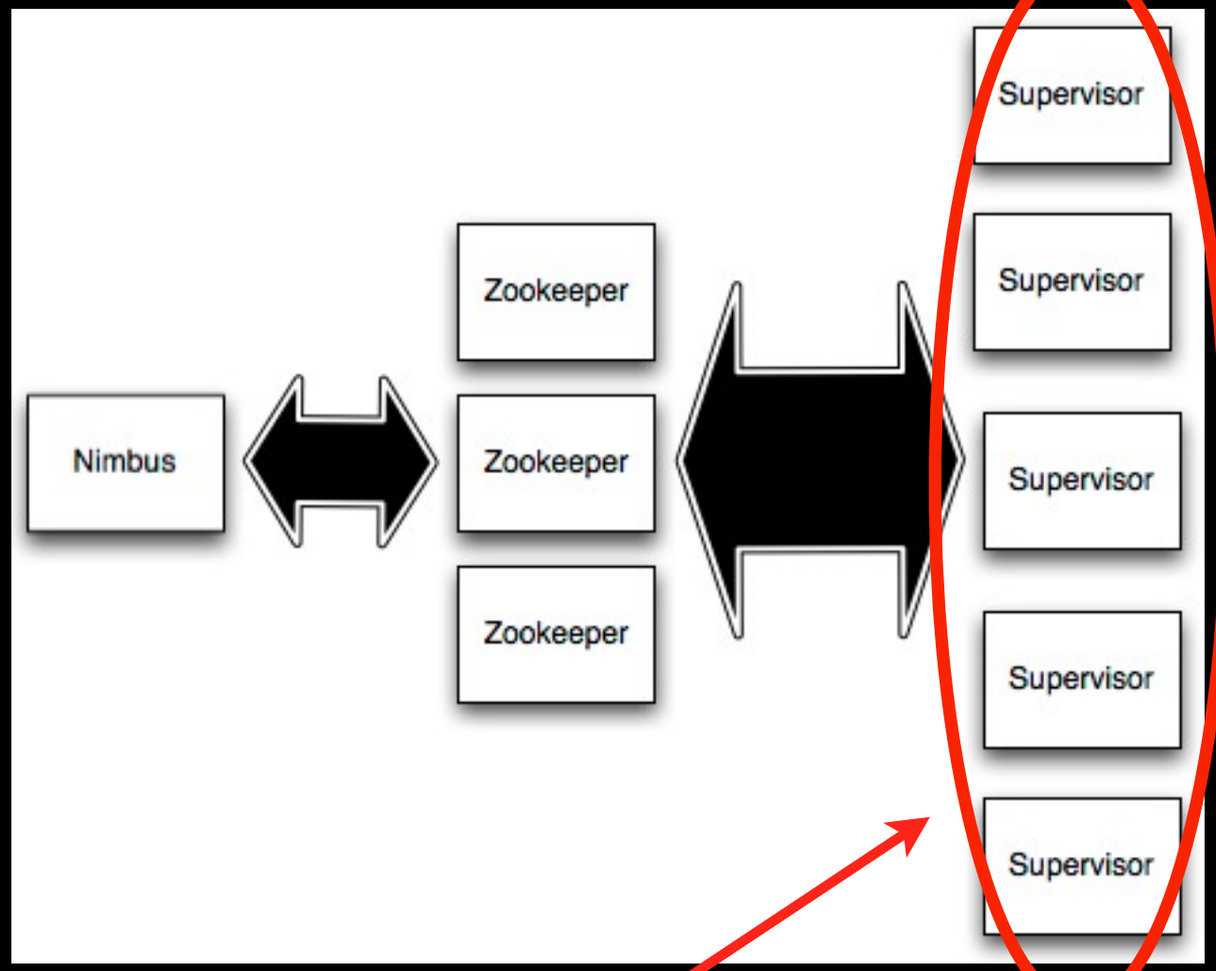
Master node (similar to Hadoop JobTracker)

Storm Cluster



Used for cluster coordination

Storm Cluster



Run worker processes

Starting a topology

```
storm jar mycode.jar twitter.storm.MyTopology demo
```

Killing a topology

```
storm kill demo
```

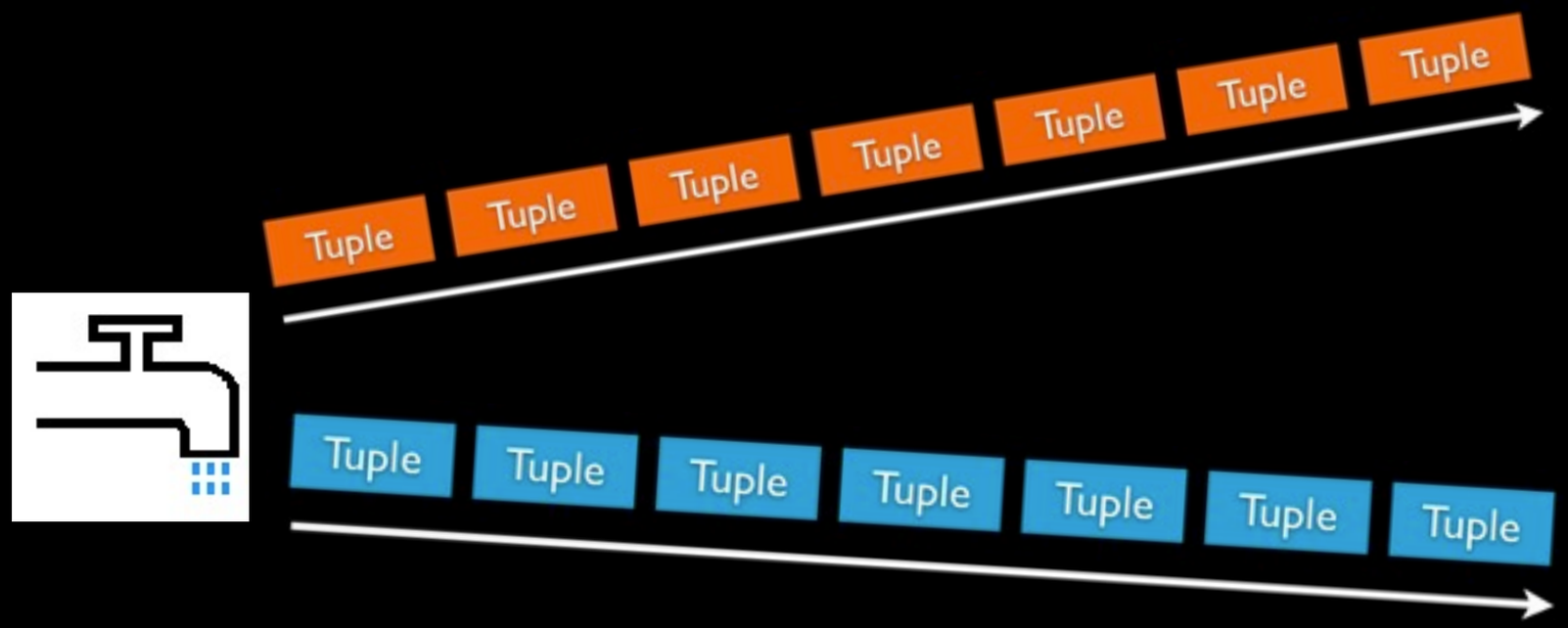
Concepts

- Streams
- Spouts
- Bolts
- Topologies

Streams

Unbounded sequence of tuples

Spouts

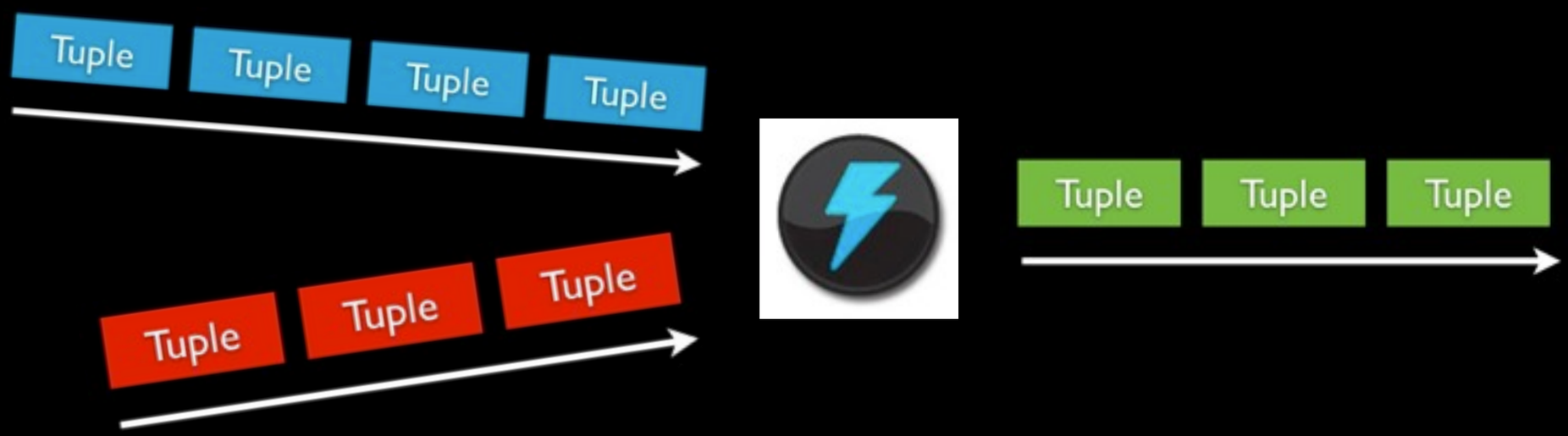


Source of streams

Spout examples

- Read from Kestrel queue
- Read from Twitter streaming API

Bolts

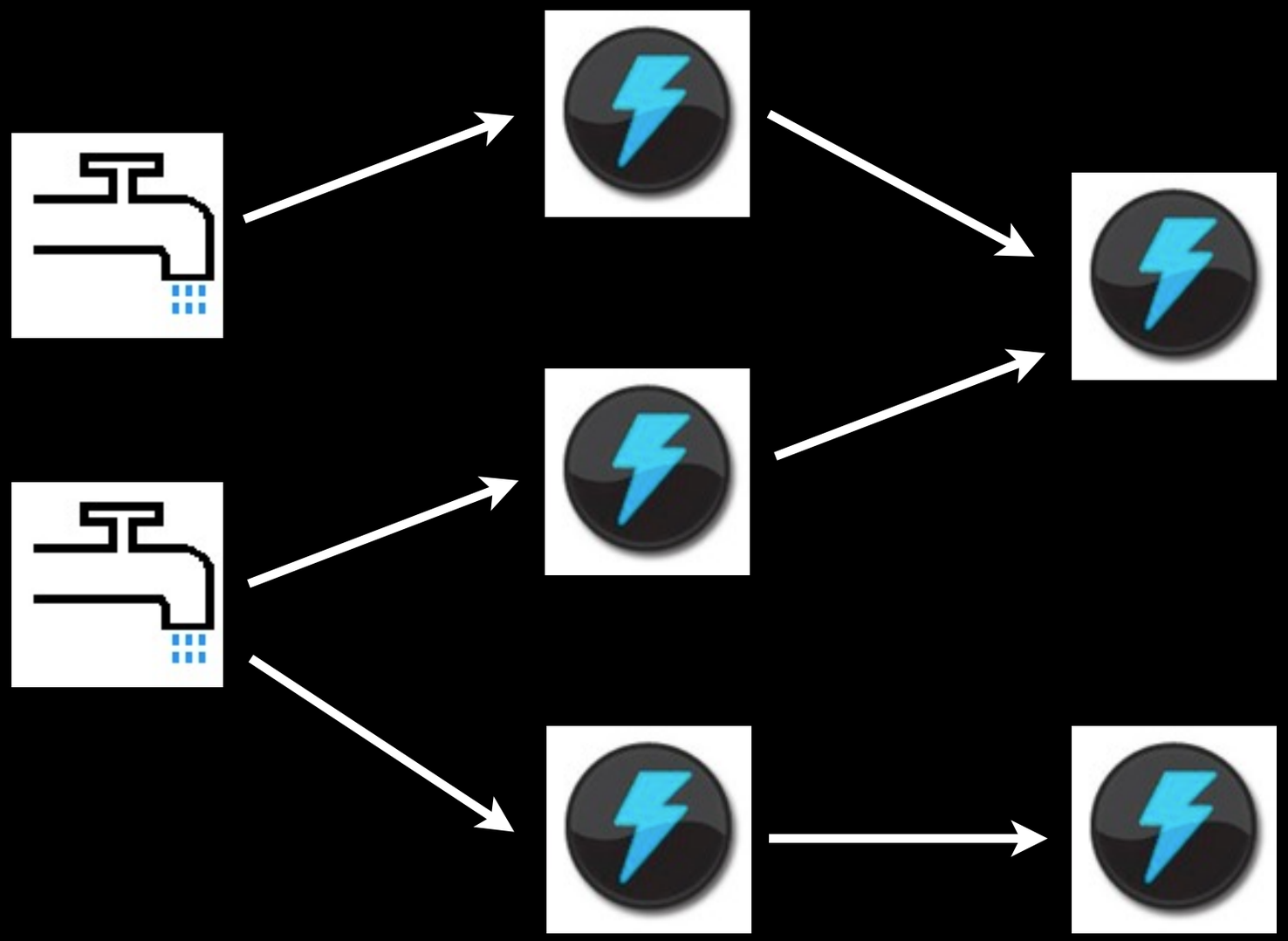


Processes input streams and produces new streams

Bolts

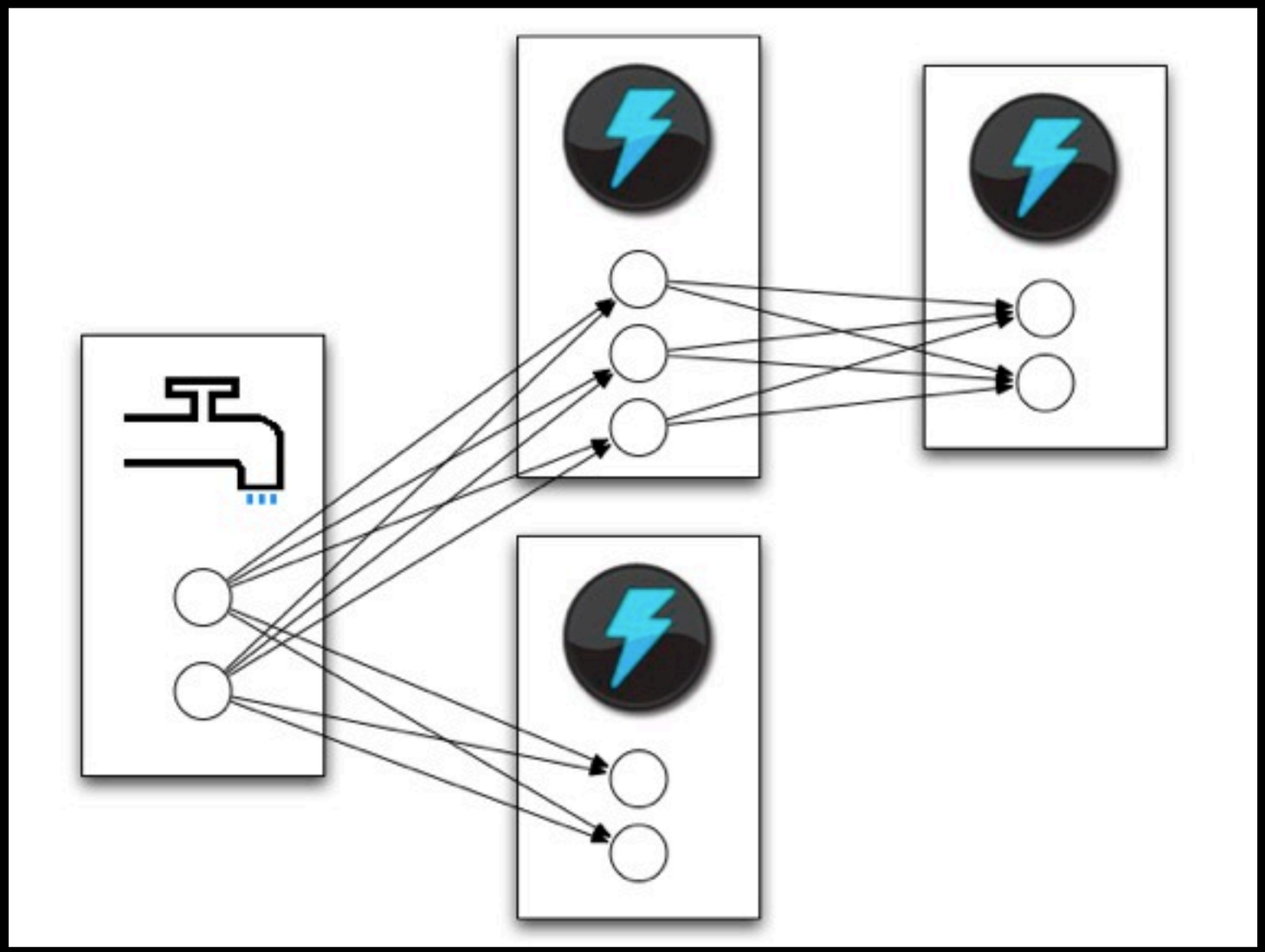
- Functions
- Filters
- Aggregation
- Joins
- Talk to databases

Topology



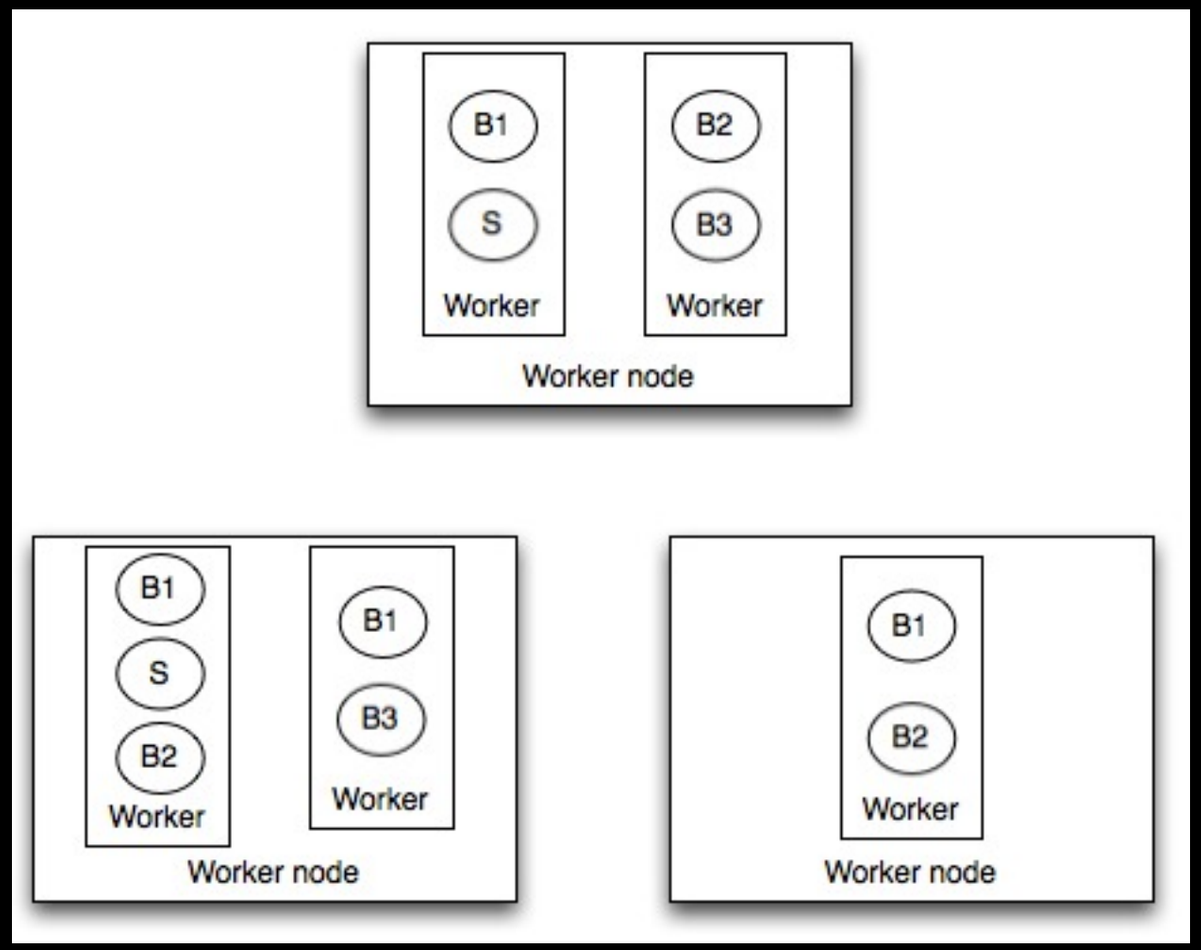
Network of spouts and bolts

Tasks



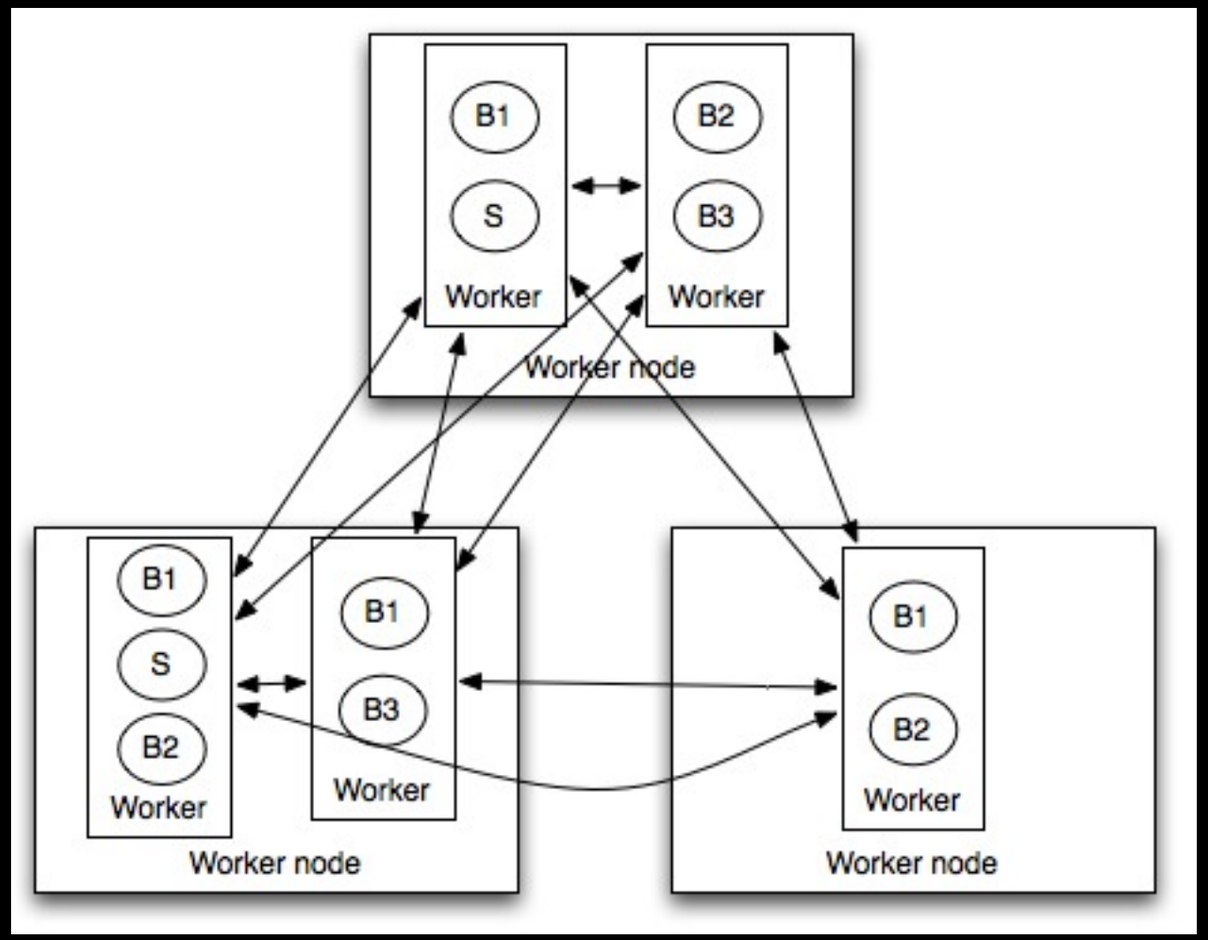
Spouts and bolts execute as
many tasks across the cluster

Task execution



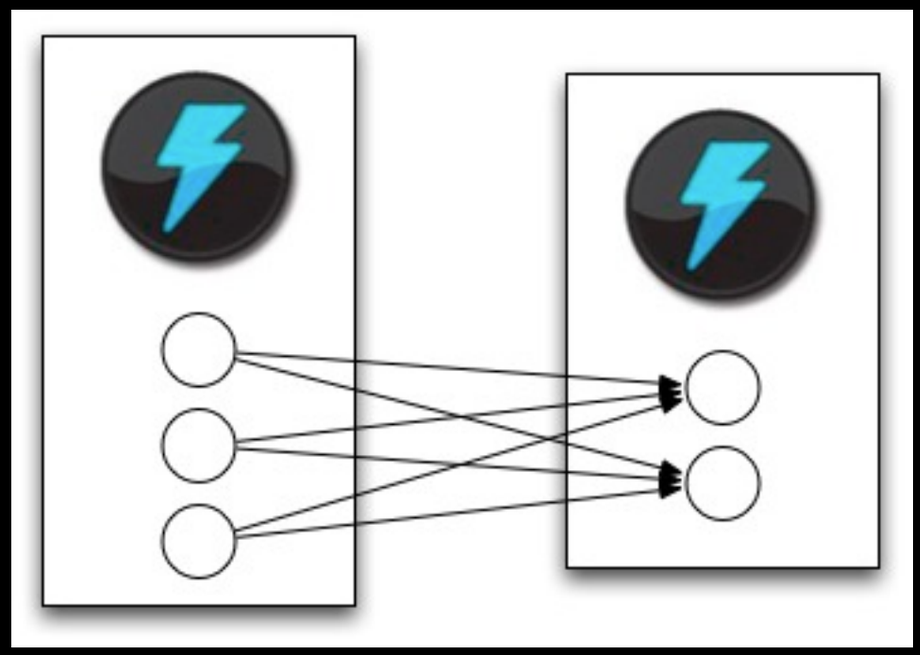
Tasks are spread across the cluster

Task execution



Tasks are spread across the cluster

Stream grouping

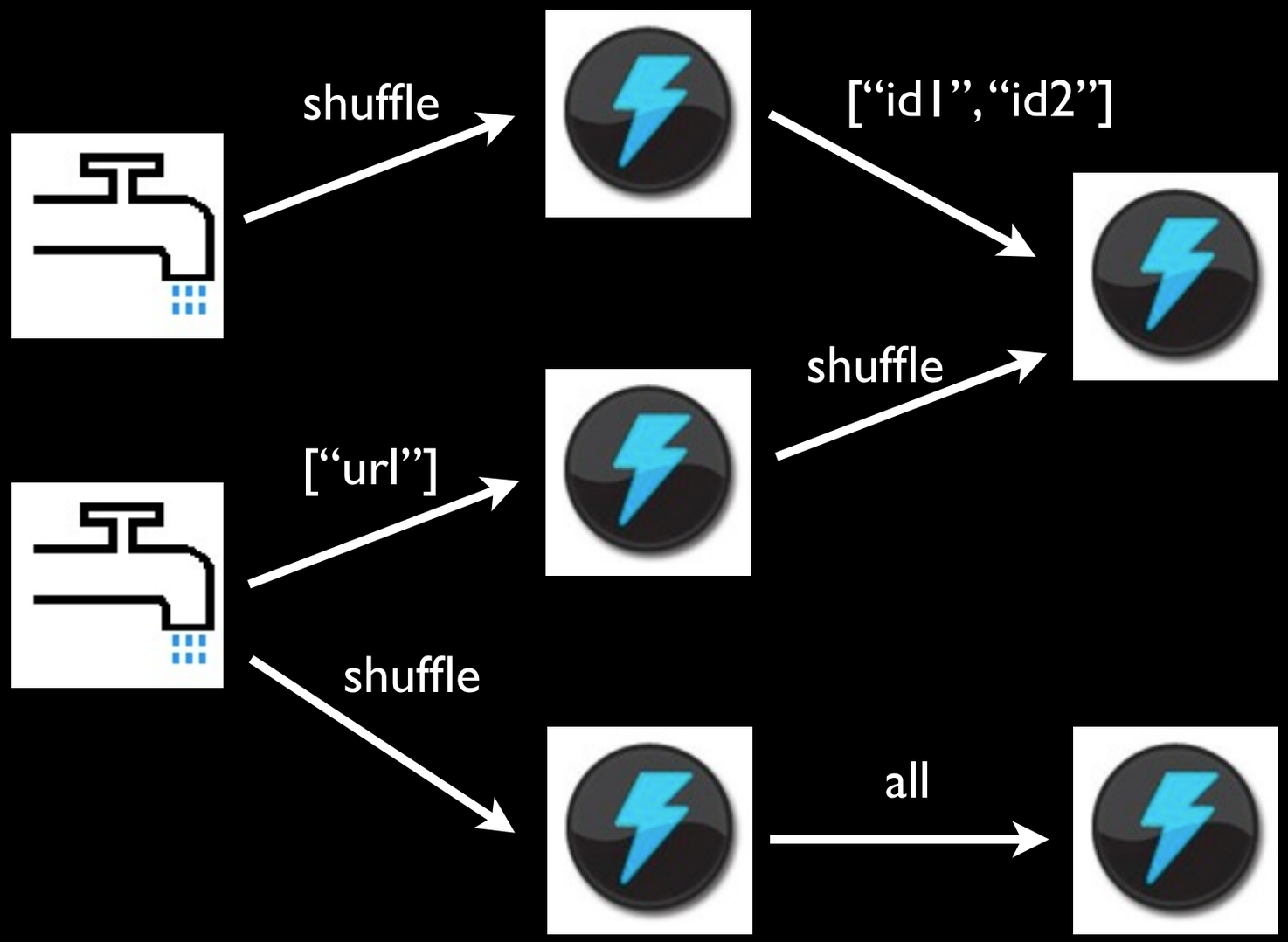


When a tuple is emitted, which task does it go to?

Stream grouping

- **Shuffle grouping:** pick a random task
- **Fields grouping:** mod hashing on a subset of tuple fields
- **All grouping:** send to all tasks
- **Global grouping:** pick task with lowest id

Topology



Streaming word count

```
(defn mk-topology []
  (topology
   {"spout" (spout-spec sentence-spout)}
   {"split" (bolt-spec {"spout" :shuffle}
                       split-sentence
                       :p 5)
    "count" (bolt-spec {"split" ["word"]}
                       word-count
                       :p 6)}))
```

Topology definition for streaming word count

Streaming word count

```
(defn mk-topology []
  (topology
   {"spout" (spout-spec sentence-spout)}
   {"split" (bolt-spec {"spout" :shuffle}
                       split-sentence
                       :p 5)
    "count" (bolt-spec {"split" ["word"]}
                       word-count
                       :p 6)}))
```

Define a spout for the topology

Streaming word count

```
(defn mk-topology []
  (topology
   {"spout" (spout-spec sentence-spout)}
   {"split" (bolt-spec {"spout" :shuffle}
                       split-sentence
                       :p 5)
    "count" (bolt-spec {"split" ["word"]}
                       word-count
                       :p 6)}))
```

Define “word splitter” bolt

Streaming word count

```
(defn mk-topology []
  (topology
   {"spout" (spout-spec sentence-spout)}
   {"split" (bolt-spec {"spout" :shuffle}
                       split-sentence
                       :p 5)
    "count" (bolt-spec {"split" ["word"]}
                       word-count
                       :p 6)}))
```

Consumes from “spout” with a shuffle grouping

Streaming word count

```
(defn mk-topology []
  (topology
   {"spout" (spout-spec sentence-spout)}
   {"split" (bolt-spec {"spout" :shuffle}
                       split-sentence
                       :p 5)
    "count" (bolt-spec {"split" ["word"]}
                       word-count
                       :p 6)}))
```

Has a parallelism of five tasks

Streaming word count

```
(defn mk-topology []
  (topology
   {"spout" (spout-spec sentence-spout)}
   {"split" (bolt-spec {"spout" :shuffle}
                       split-sentence
                       :p 5)
    "count" (bolt-spec {"split" ["word"]}
                       word-count
                       :p 6)}))
```

Define word counting bolt

Streaming word count

```
(defn mk-topology []
  (topology
   {"spout" (spout-spec sentence-spout)}
   {"split" (bolt-spec {"spout" :shuffle}
                       split-sentence
                       :p 5)
    "count" (bolt-spec {"split" ["word"]}
                       word-count
                       :p 6)}))
```

Consumes from “split” with a fields grouping on “word”

Streaming word count

```
(defspout sentence-spout ["sentence"]
  [conf context collector]
  (let [sentences ["a little brown dog"
                   "the man petted the dog"
                   "four score and seven years ago"
                   "an apple a day keeps the doctor away"]]
    (spout
     (nextTuple []
       (Thread/sleep 100)
       (emit-spout! collector [(rand-nth sentences)])
       ))))
```

Spout that emits randomly chosen sentences

Streaming word count

```
(defbolt split-sentence ["word"] [tuple collector]
  (let [words (.split (.getString tuple 0) " ")]
    (doseq [w words]
      (emit-bolt! collector [w] :anchor tuple))
  (ack! collector tuple)
  ))
```

Word splitting bolt

Streaming word count

```
(defbolt word-count ["word" "count"] {:prepare true}
  [conf context collector]
  (let [counts (atom {})]
    (bolt
     (execute [tuple]
       (let [word (.getString tuple 0)]
         (swap! counts (partial merge-with +) {word 1})
         (emit-bolt! collector [word (@counts word)] :anchor tuple)
         (ack! collector tuple)
         )))))
```

Word counting bolt

Streaming word count

```
(StormSubmitter/submitTopology
 "word-count"
 {TOPOLOGY-WORKERS 3}
 (mk-topology))
```

Submitting topology to a cluster

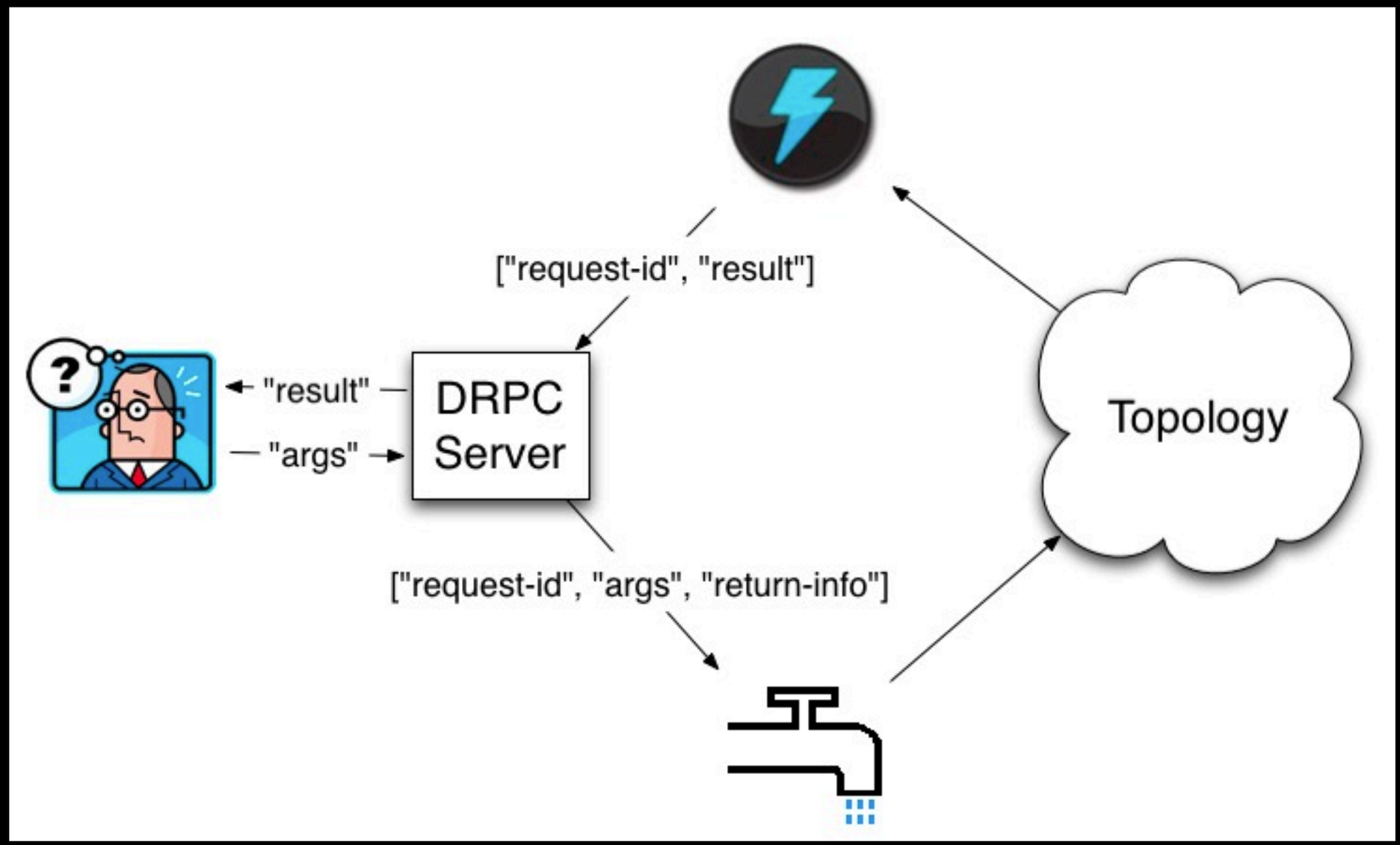
Streaming word count

```
(let [cluster (LocalCluster.)]
  (.submitTopology cluster
                   "word-count"
                   {TOPOLOGY-DEBUG true}
                   (mk-topology)))
```

Running topology in local mode

Demo

Distributed RPC



Data flow for Distributed RPC

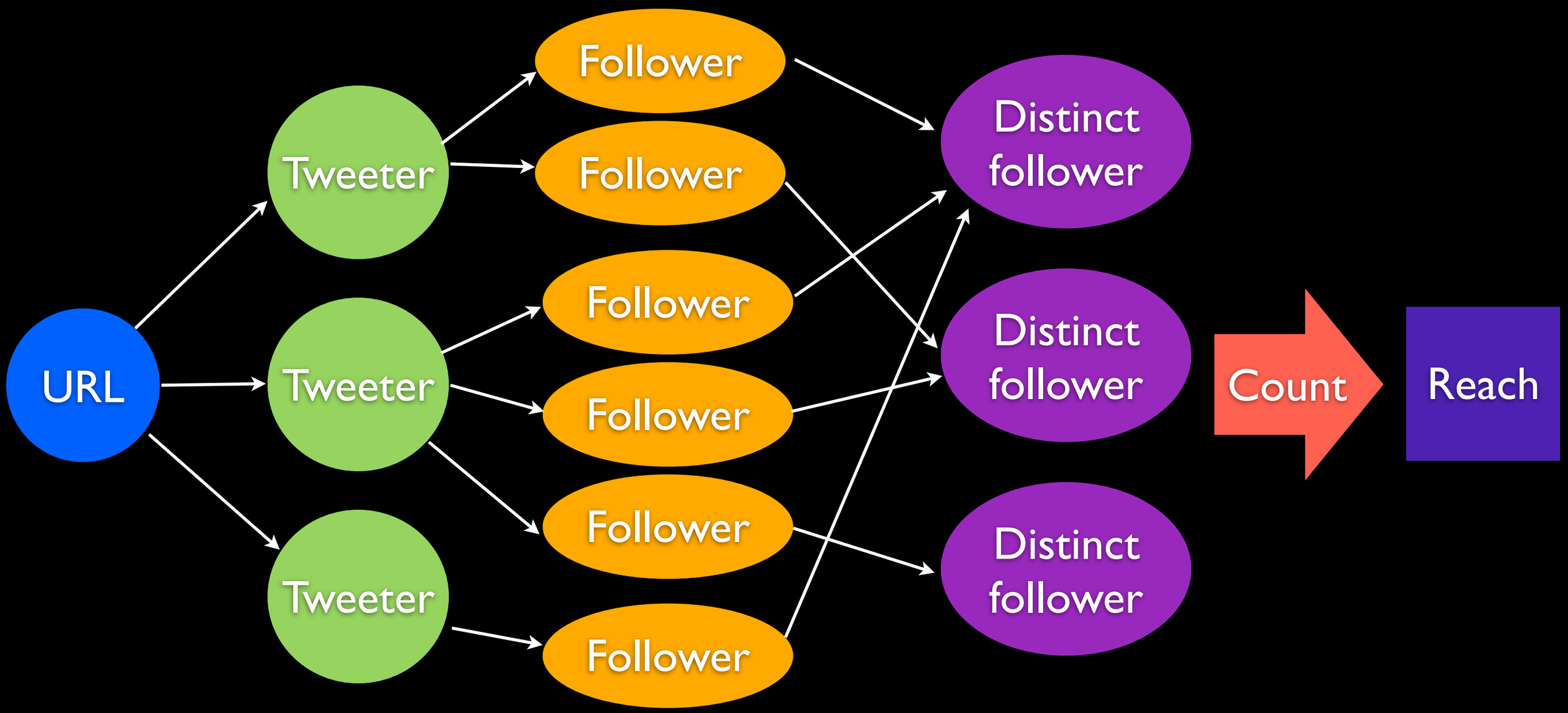
DRPC Example

Computing “reach” of a URL on the fly

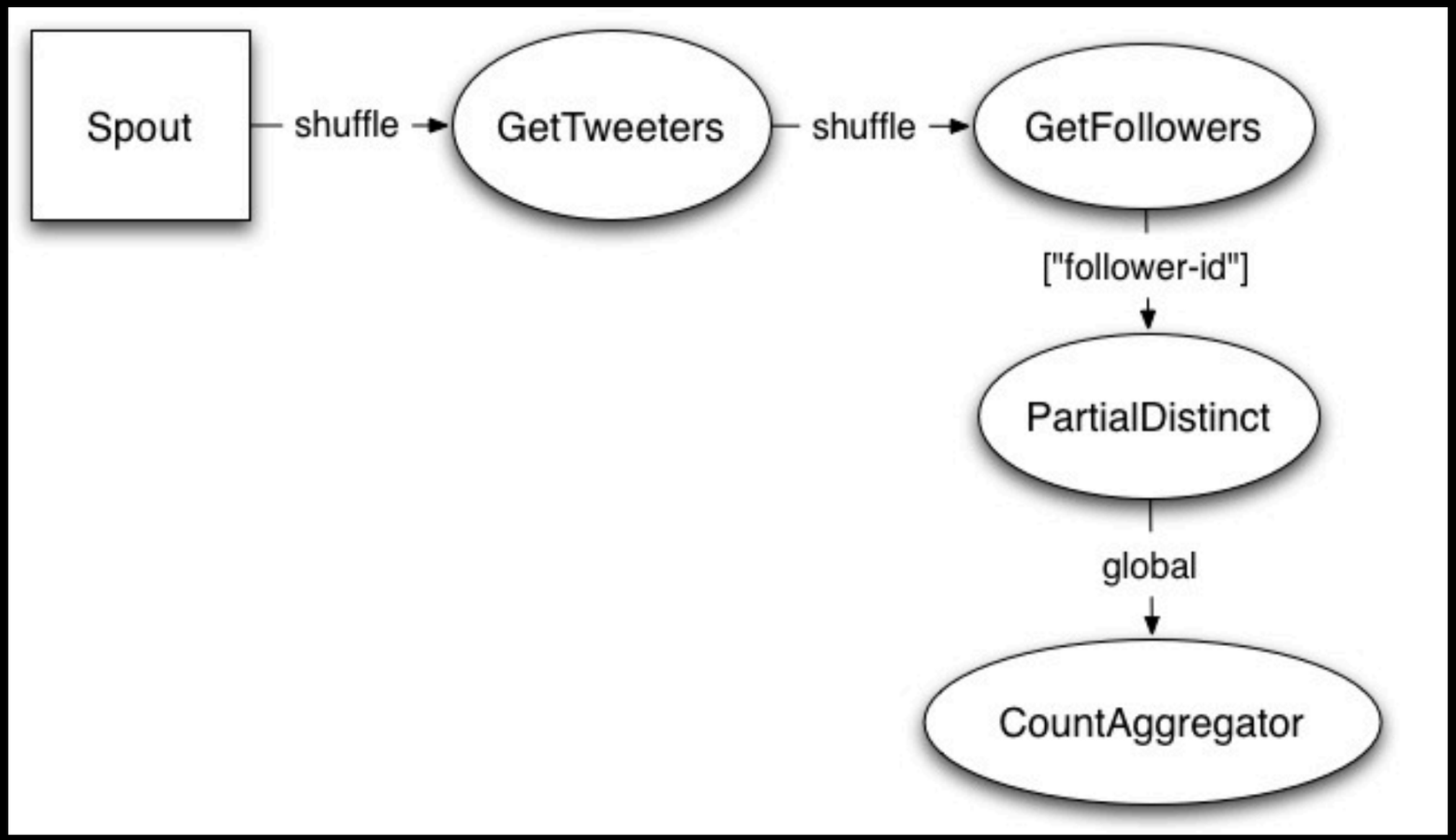
Reach

Reach is the number of unique people exposed to a URL on Twitter

Computing reach



Reach topology



Reach topology

```
(linear-drpc-topology
 [get-tweeters :p 3]
 [get-followers :grouping :shuffle :p 12]
 [partial-distinct :grouping ["follower-id"] :p 8]
 [count-aggregator :grouping :global]
 )
```

Reach topology

```
(defbatchbolt partial-uniquer ["id" "partial-count"]
  [conf context collector id]
  (let [followers (atom #{})]
    (batchbolt
     (execute [tuple]
       (swap! followers conj (.getValue tuple 1)))
     (finishBatch []
       (emit-batch-bolt! collector [id (count @followers)])))))
```

Reach topology

```
(defbatchbolt partial-uniquer ["id" "partial-count"]
  [conf context collector id]
  (let [followers (atom #{})]
    (batchbolt
      (execute [tuple]
        (swap! followers conj (.getValu
      (finishBatch []
        (emit-batch-bolt! collector [id (count @followers)])))))
```

Keep set of followers for each request id in memory

Reach topology

```
(defbatchbolt partial-uniquer ["id" "partial-count"]
  [conf context collector id]
  (let [followers (atom #{})]
    (batchbolt
      (execute [tuple]
        (swap! followers conj (.getValue tuple 1)))
      (finishBatch []
        (emit-batch-bolt! collector [id (count @followers)])))))
```

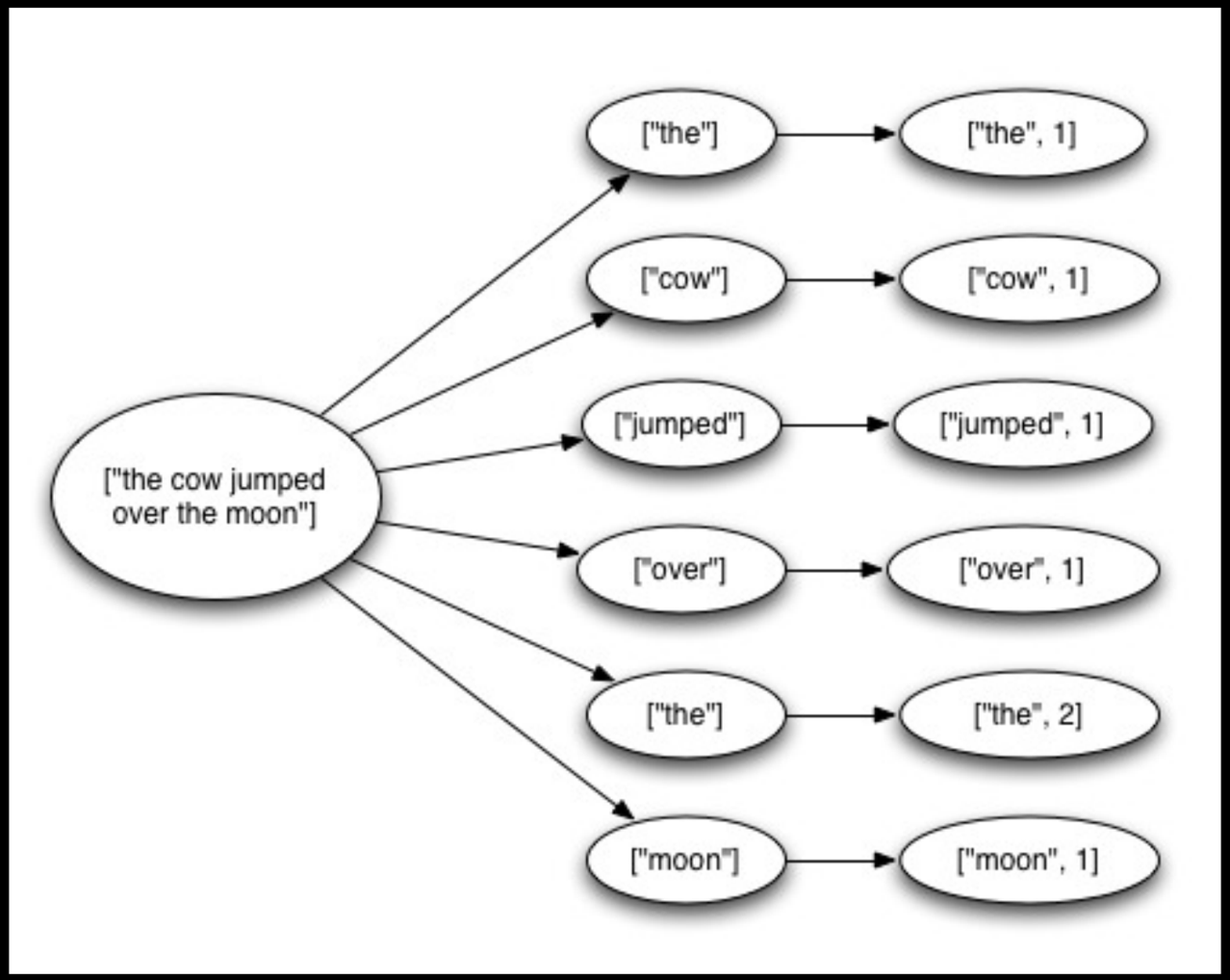
Update followers set when receive a new follower

Reach topology

Emit partial count after receiving all followers for a request id

```
(defbatchbolt partial-uniquer ["id"
  [conf context collector id]
  (let [followers (atom #{})]
    (batchbolt
     (execute [tuple]
       (swap! followers conj (.getValue tuple 1)))
     (finishBatch []
       (emit-batch-bolt! collector [id (count @followers)])))))
```

Guaranteeing message processing



"Tuple tree"

Guaranteeing message processing

- A spout tuple is not fully processed until all tuples in the tree have been completed

Guaranteeing message processing

- If the tuple tree is not completed within a specified timeout, the spout tuple is replayed

Guaranteeing message processing

```
(defbolt split-sentence ["word"] [tuple collector]
  (let [words (.split (.getString tuple 0) " ")]
    (doseq [w words]
      (emit-bolt! collector [w] :anchor tuple))
    (ack! collector tuple)
    ))
```

Reliability API

Guaranteeing message processing

```
(defbolt split-sentence ["word"] [tuple collector]
  (let [words (.split (.getString tuple 0) " ")]
    (doseq [w words]
      (emit-bolt! collector [w] :anchor tuple))
    (ack! collector tuple)
    ))
```

"Anchoring" creates a new edge in the tuple tree

Guaranteeing message processing

```
(defbolt split-sentence ["word"] [tuple collector]
  (let [words (.split (.getString tuple 0) " ")]
    (doseq [w words]
      (emit-bolt! collector [w] :anchor tuple))
    (ack! collector tuple)
    ))
```

Marks a single node in the tree as complete

Guaranteeing message processing

- Storm tracks tuple trees for you in an extremely efficient way

Transactional topologies

How do you do idempotent counting with an at least once delivery guarantee?

Transactional topologies

Won’t you overcount?

Transactional topologies

Transactional topologies solve this problem

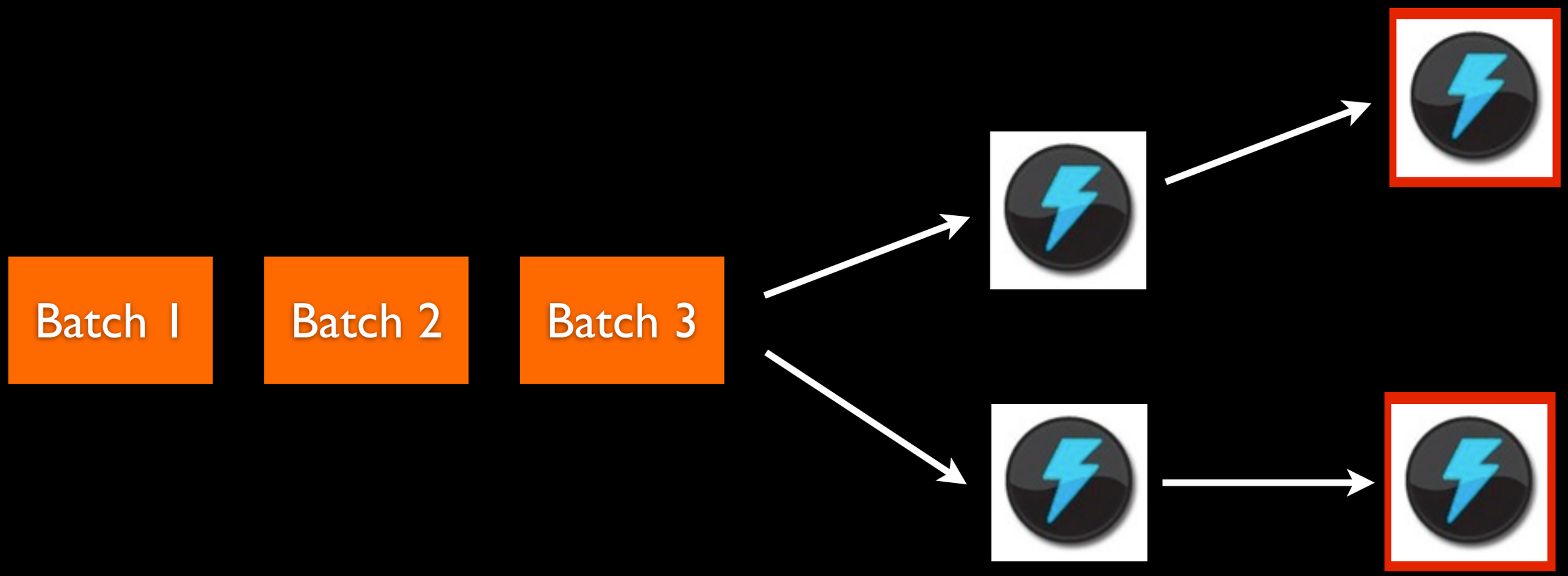
Transactional topologies

Built completely on top of Storm's primitives
of streams, spouts, and bolts

Transactional topologies

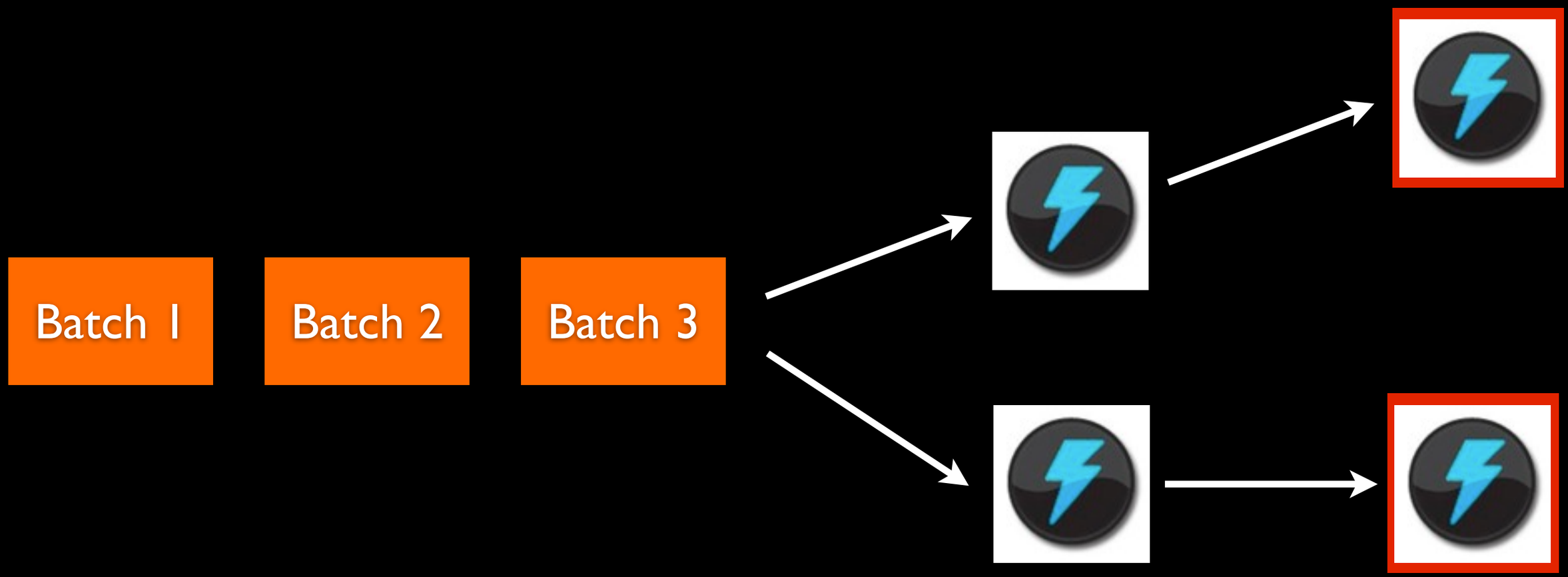
Enables fault-tolerant, exactly-once messaging semantics

Transactional topologies



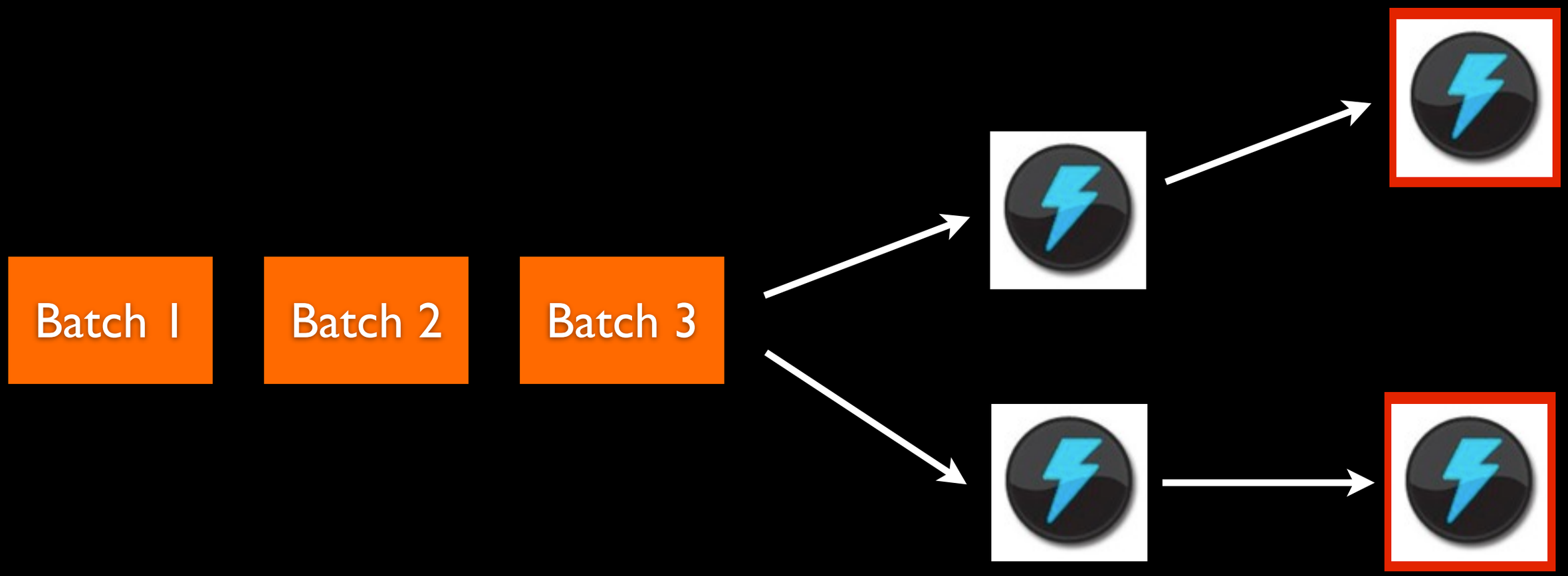
Process small batches of tuples

Transactional topologies



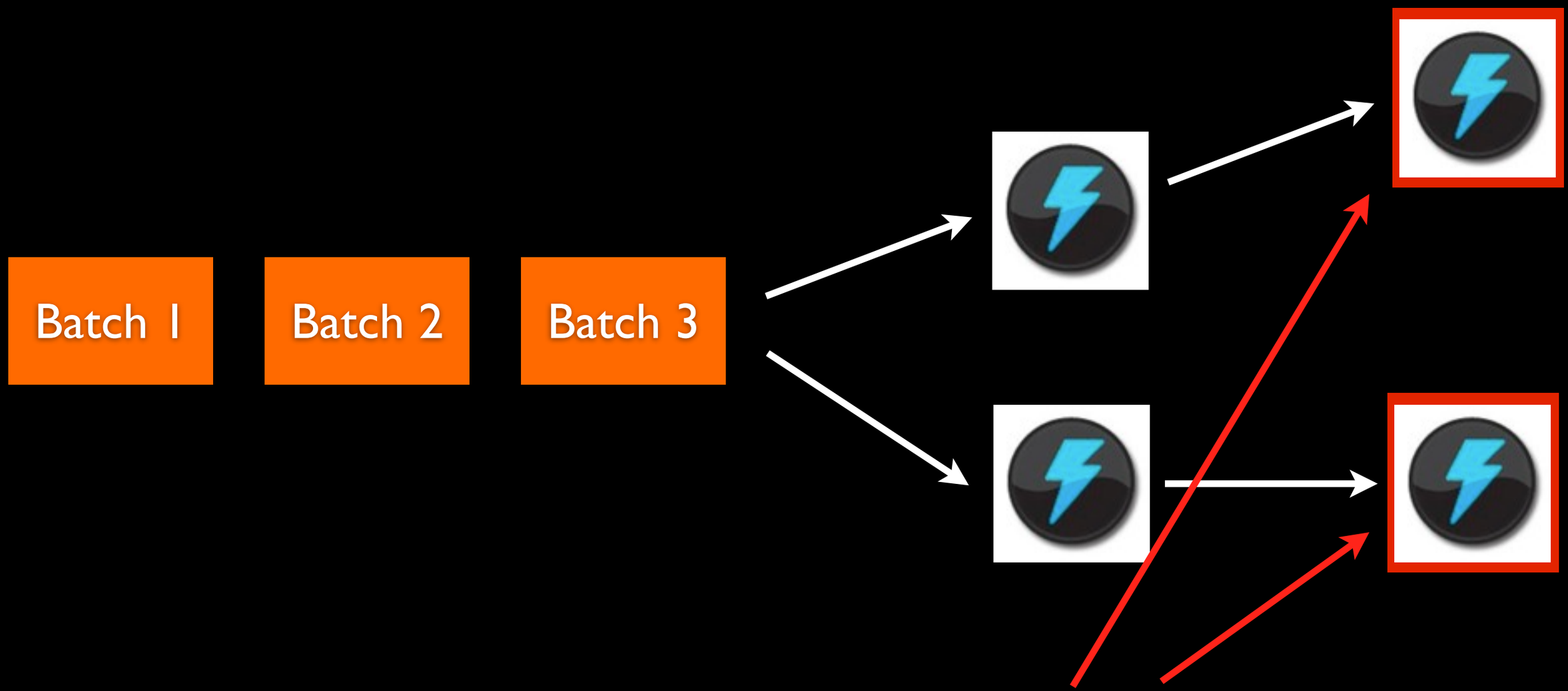
If a batch fails, replay the whole batch

Transactional topologies



Once a batch is completed, commit the batch

Transactional topologies



Bolts can optionally be “committers”

Transactional topologies

Commit 1 | Commit 1 | Commit 2 | Commit 3 | Commit 4 | Commit 4

Commits are ordered. If there's a failure during commit, the whole batch + commit is retried

Example

```
(defbatchbolt idempotent-counter {} {:committer true}
  [conf context collector attempt]
  (let [partial-count (atom 0)]
    (batchbolt
     (execute [tuple]
       (swap! partial-count inc))
     (finishBatch []
       (let [curr-val (get-db-val)
             txid (txid attempt)]
         (if-not (= txid (:txid curr-val))
           (set-db-val (+ (:val curr-val)
                          @partial-count)
                       txid)
           )))))
```

Example

```
(defbatchbolt idempotent-counter {} {:committer true}
  [conf context collector attempt]
  (let [partial-count (atom 0)]
    (batchbolt
     (execute [tuple]
       (swap! partial-count inc))
     (finishBatch []
       (let [curr-val (get-db-val)
             txid (txid attempt)]
         (if-not (= txid (:txid curr-val))
           (set-db-val (+ (:val curr-val)
                          @partial-count)
                       txid)
           )))))
```

New instance of this object for every transaction attempt

Example

```
(defbatchbolt idempotent-counter {} {:committer true}
  [conf context collector attempt]
  (let [partial-count (atom 0)]
    (batchbolt
      (execute [tuple]
        (swap! partial-count inc))
      (finishBatch []
        (let [curr-val (get-db-val)
              txid (txid attempt)]
          (if-not (= txid (:txid curr-val))
            (set-db-val (+ (:val curr-val)
                           @partial-count)
                        txid)
            ))))))
```

Aggregate the count for this batch

Example

```
(defbatchbolt idempotent-counter {} {:committer true}
  [conf context collector attempt]
  (let [partial-count (atom 0)]
    (batchbolt
     (execute [tuple]
       (swap! partial-count inc))
     (finishBatch []
       (let [curr-val (get-db-val)
             txid (txid attempt)]
         (if-not (= txid (:txid curr-val))
           (set-db-val (+ (:val curr-val)
                          @partial-count)
                       txid)
           ))))))
```

Only update database if transaction ids differ

Example

```
(defbatchbolt idempotent-counter {} {:committer true}
  [conf context collector attempt]
  (let [partial-count (atom 0)]
    (batchbolt
     (execute [tuple]
       (swap! partial-count inc))
     (finishBatch []
       (let [curr-val (get-db-val)
             txid (txid attempt)]
         (if-not (= txid (:txid curr-val))
           (set-db-val (+ (:val curr-val)
                          @partial-count)
                       txid)
           ))))))
```

This enables idempotency since commits are ordered

Example

```
(defbatchbolt idempotent-counter {} {:committer true}
  [conf context collector attempt]
  (let [partial-count (atom 0)]
    (batchbolt
     (execute [tuple]
       (swap! partial-count inc))
     (finishBatch []
       (let [curr-val (get-db-val)
             txid (txid attempt)]
         (if-not (= txid (:txid curr-val))
           (set-db-val (+ (:val curr-val)
                          @partial-count)
                       txid)
           ))))))
```

(Credit goes to Kafka devs for this trick)

Transactional topologies

Multiple batches can be processed in parallel,
but commits are guaranteed to be ordered

Transactional topologies

- Requires a more sophisticated source queue than Kestrel or RabbitMQ
- storm-contrib has a transactional spout implementation for Kafka

Storm UI

Storm UI

Topology summary

Name	Id	Uptime	Num workers	Num tasks
poseidon	poseidon-1-1314658150	23h 17m 0s	80	765

Topology stats

Window	Emitted	Transferred	Complete latency (ms)	Acked	Failed
10m 0s	24786020	24786000	4131.688	2338940	0
3h 0m 0s	621695800	621694600	4463.830	59353840	0
1d 0h 0m 0s	4447725560	4447716960	4278.459	438710100	0
All time	4447725560	4447716960	4278.459	438710100	0

Spouts (All time)

Id	Parallelism	Emitted	Transferred	Complete latency (ms)	Acked	Failed	Last error
1	160	877453060	877453060	4278.459	438710100	0	

Bolts (All time)

Id	Parallelism	Emitted	Transferred	Process latency (ms)	Acked	Failed	Last error
-1	4	438716440	438716440	0.009	2223890060	0	
2	160	877451720	877451720	0.320	438725980	0	
3	160	1264258160	1264258160	5.438	438724980	0	
4	18	55946080	55946080	0.215	55946040	0	
5	18	55947280	55947280	0.121	55947280	0	
6	18	55945660	55945660	0.229	55945660	0	
7	18	55946480	55946480	0.145	55946580	0	
8	18	81512620	81512620	0.209	81512620	0	
9	30	438710060	438710060	4205.639	438710140	0	
10	80	163024580	163024580	0.194	81512200	0	

Storm on EC2

https://github.com/nathanmarz/storm-deploy

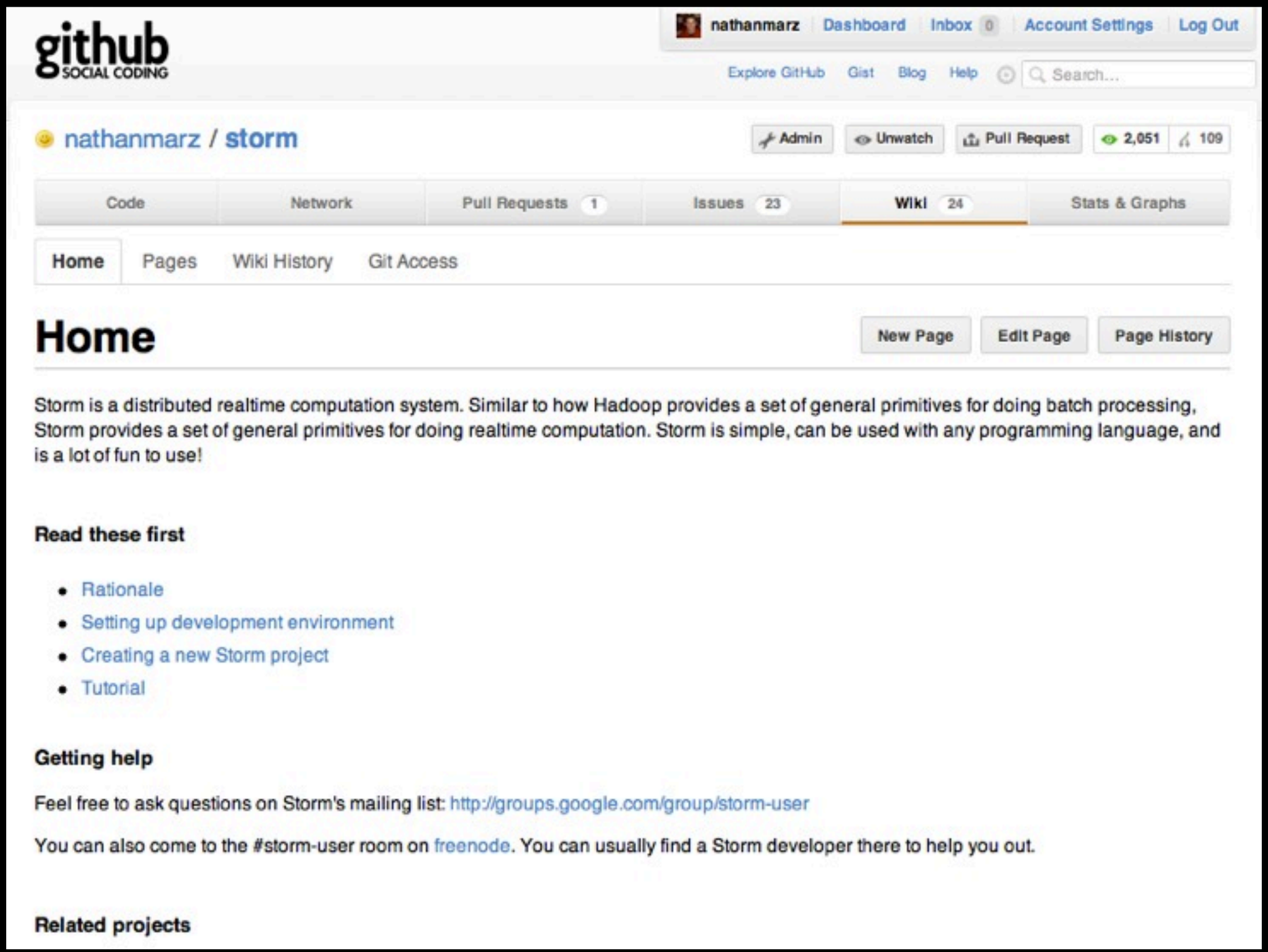
One-click deploy tool

Starter code

https://github.com/nathanmarz/storm-starter

Example topologies

Documentation



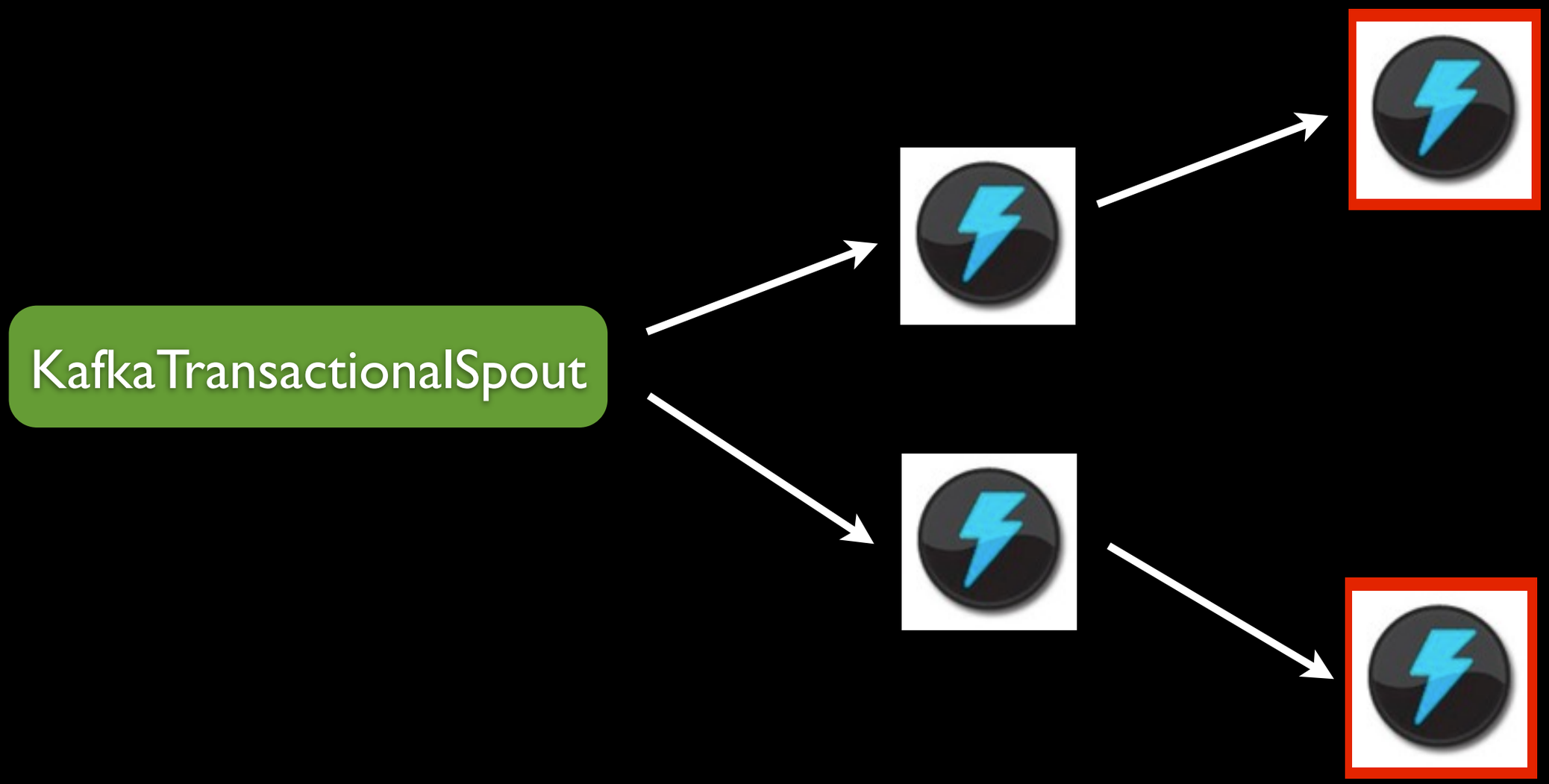
Questions?

http://github.com/nathanmarz/storm

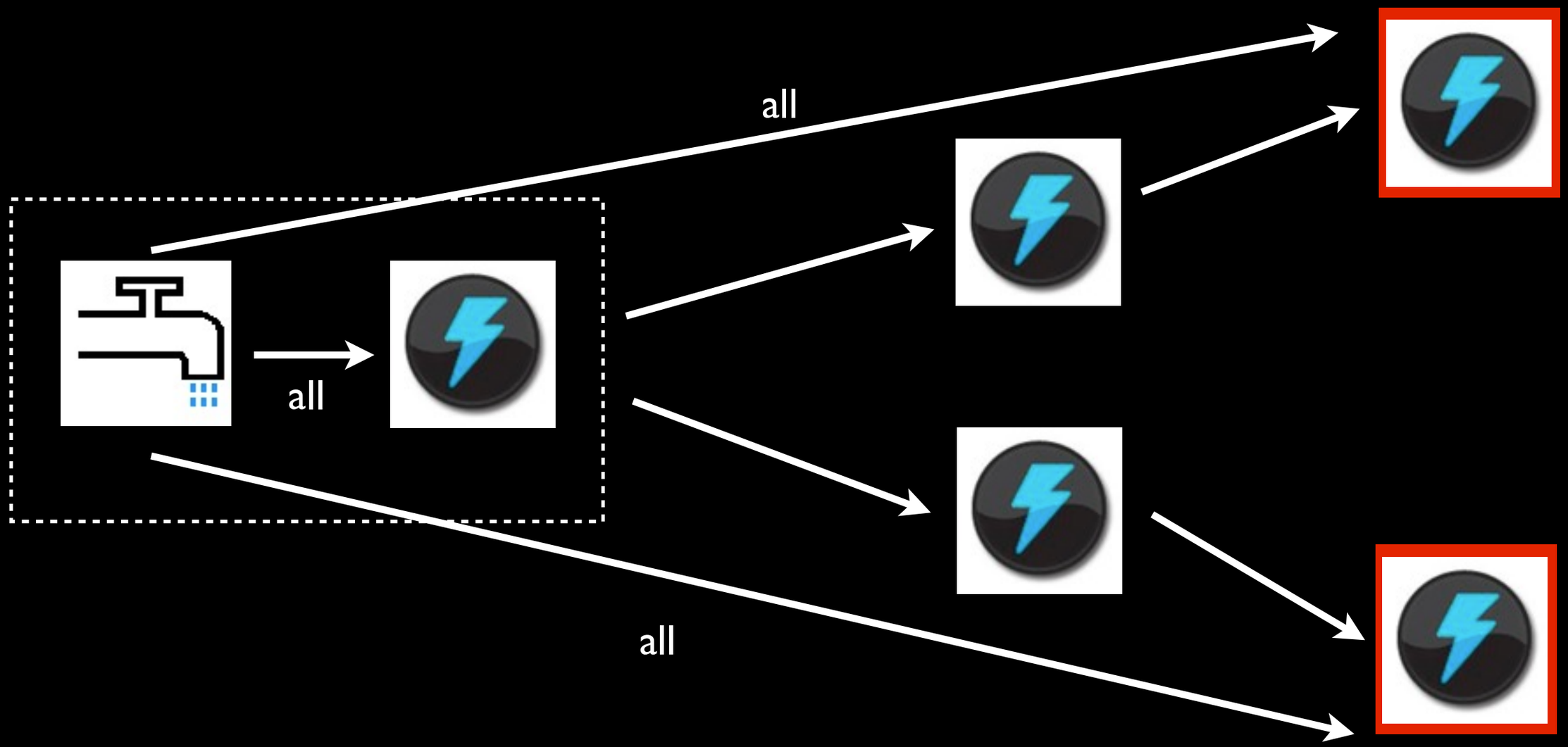
Future work

- State spout
- Storm on Mesos
- “Swapping”
- Auto-scaling
- Higher level abstractions

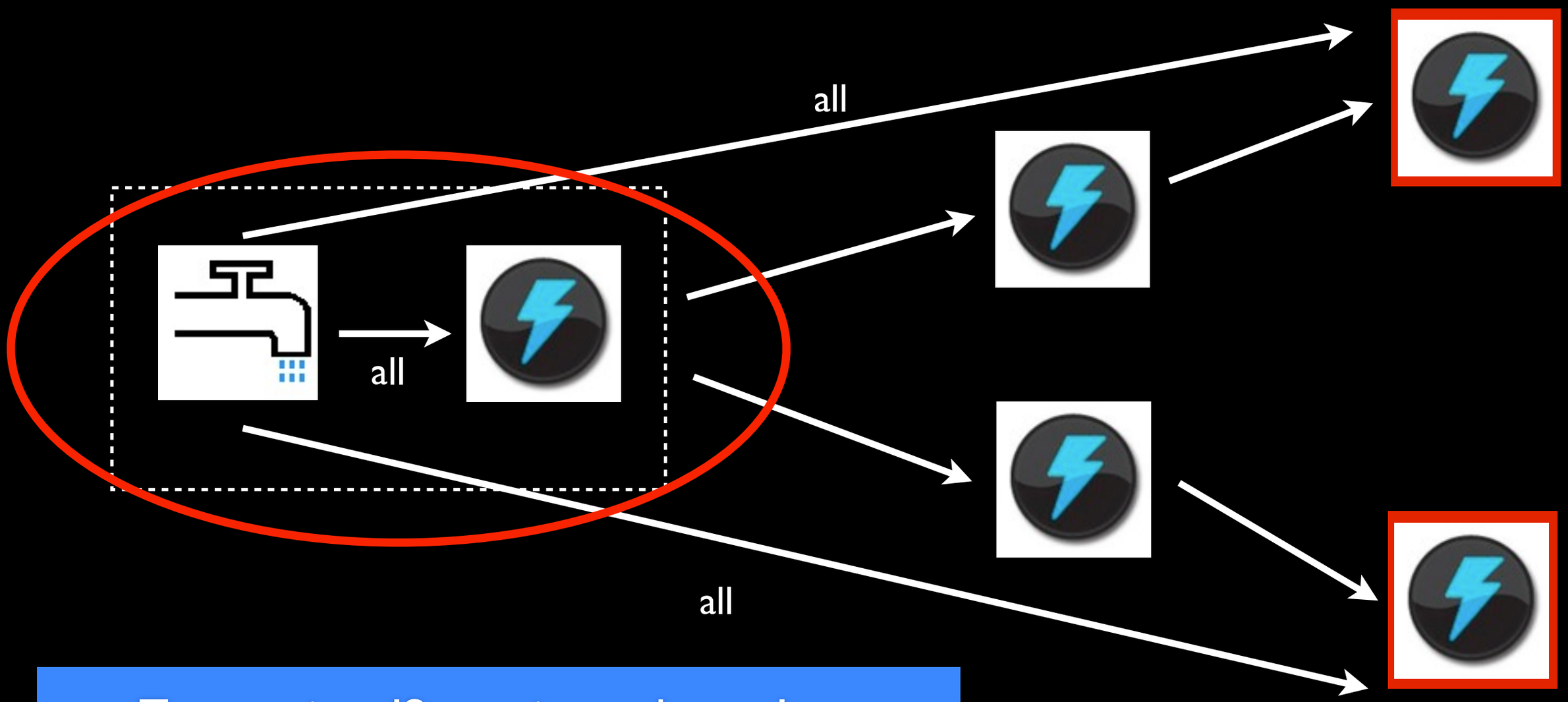
Implementation



Implementation

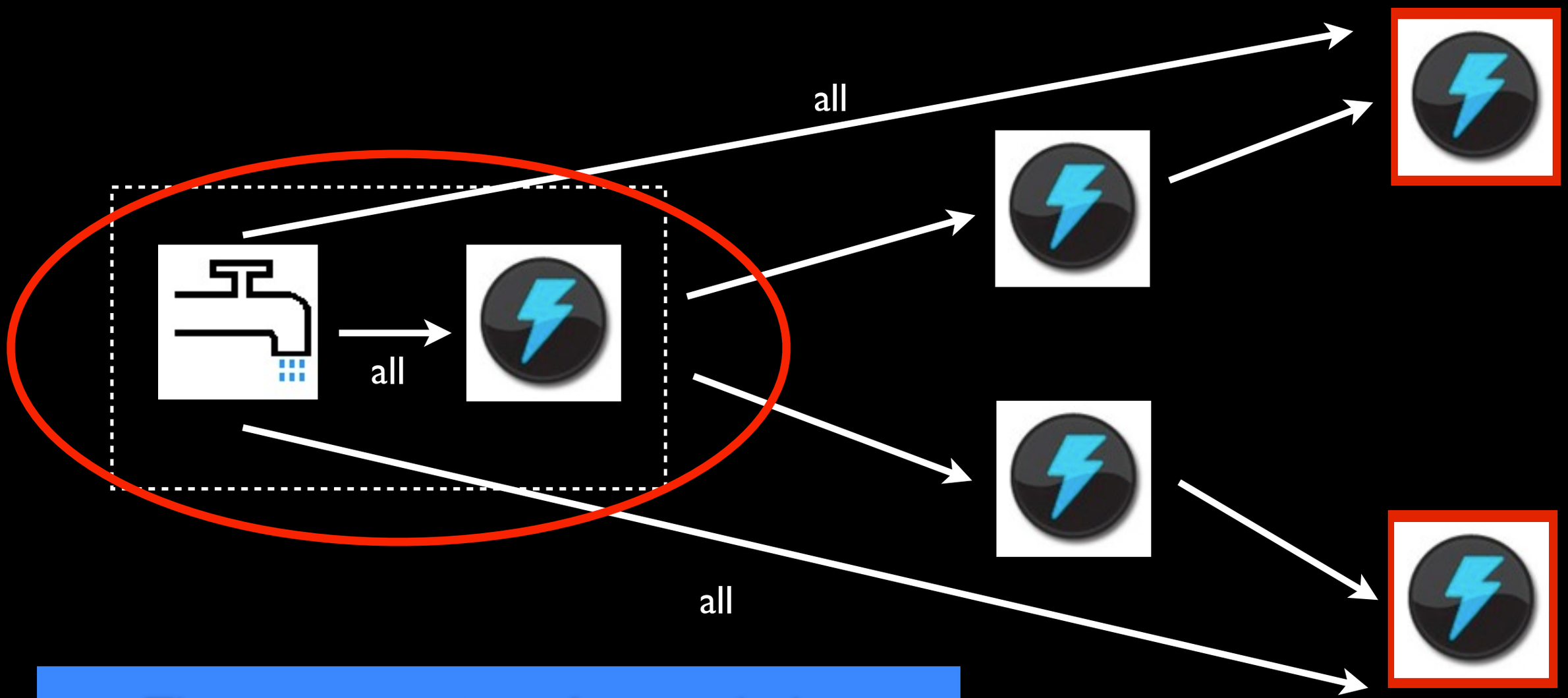


Implementation



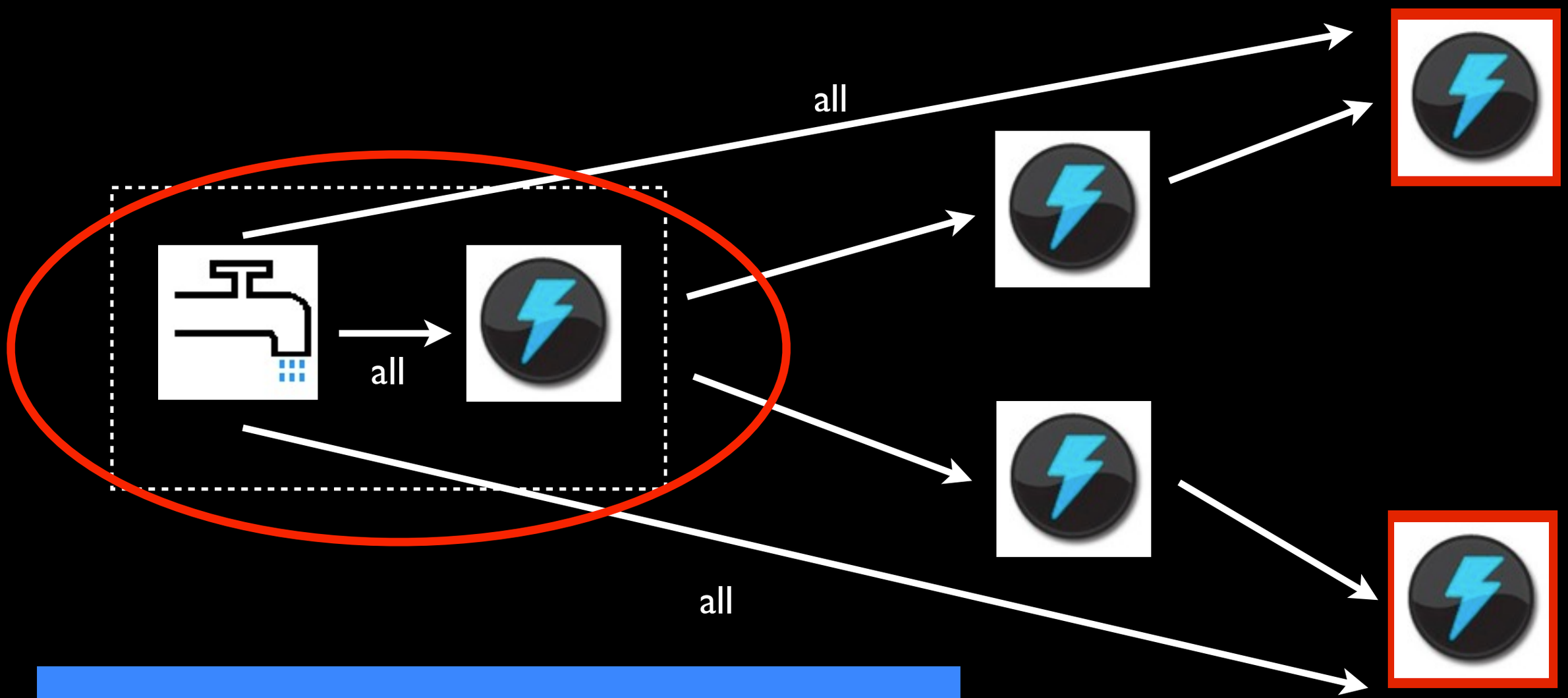
TransactionalSpout is a subtopology consisting of a spout and a bolt

Implementation



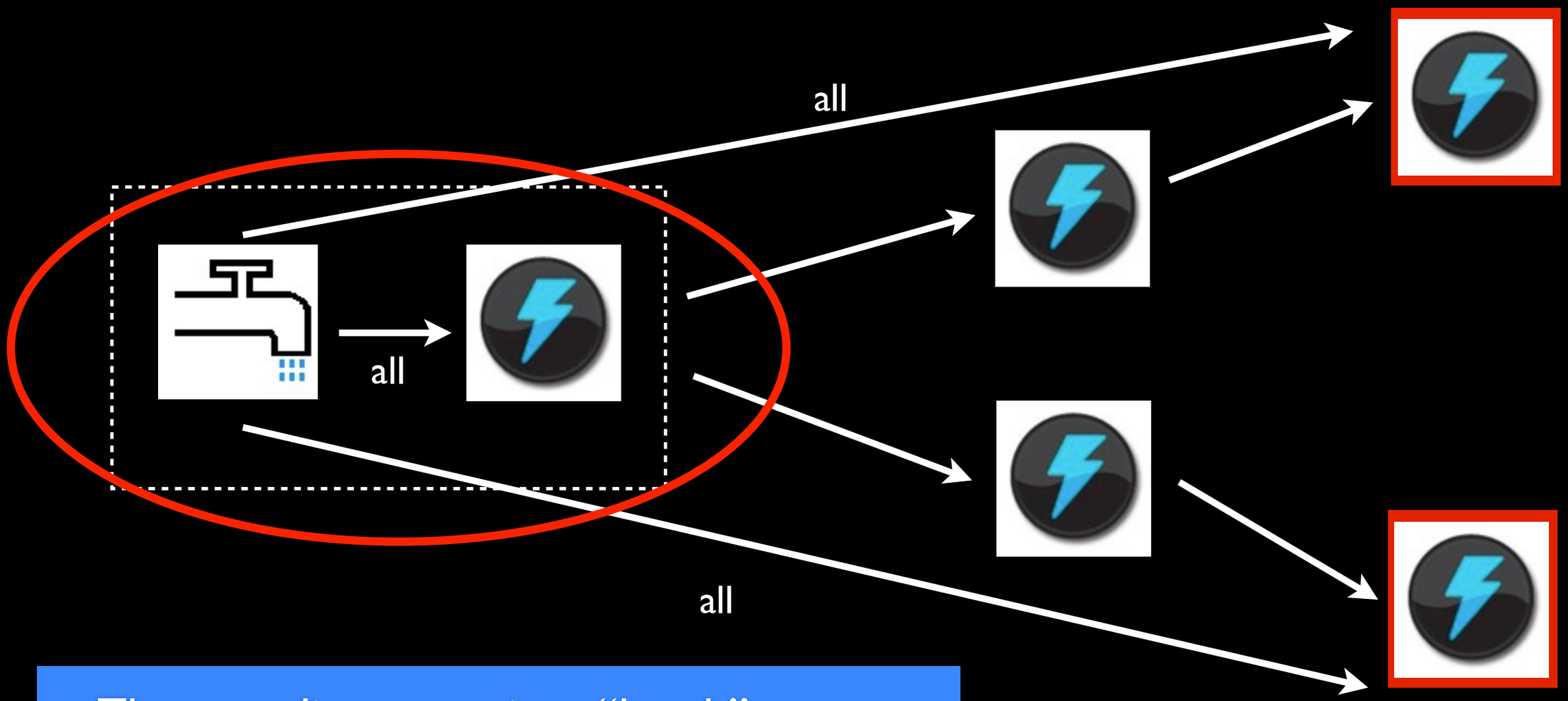
The spout consists of one task that coordinates the transactions

Implementation



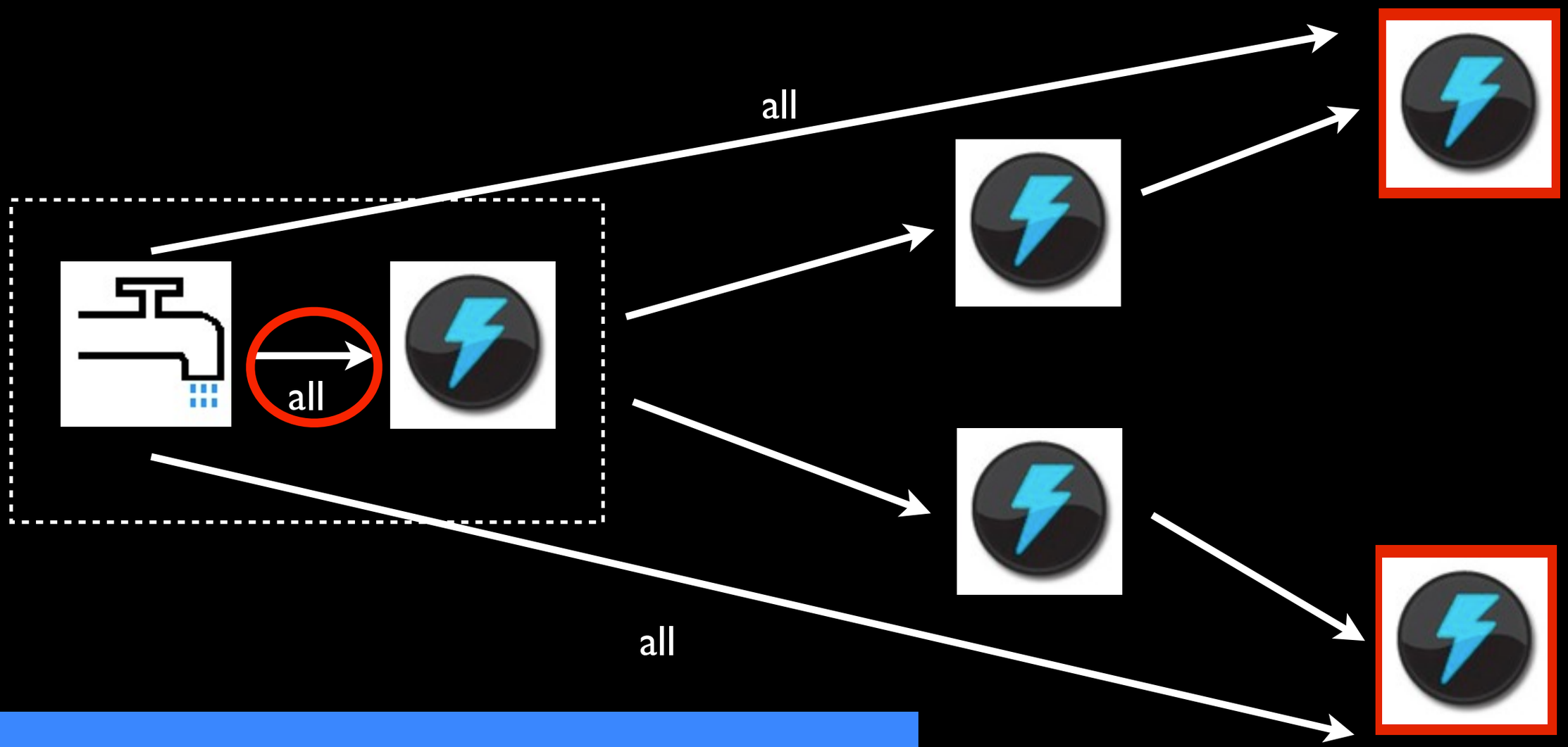
The bolt emits the batches of tuples

Implementation



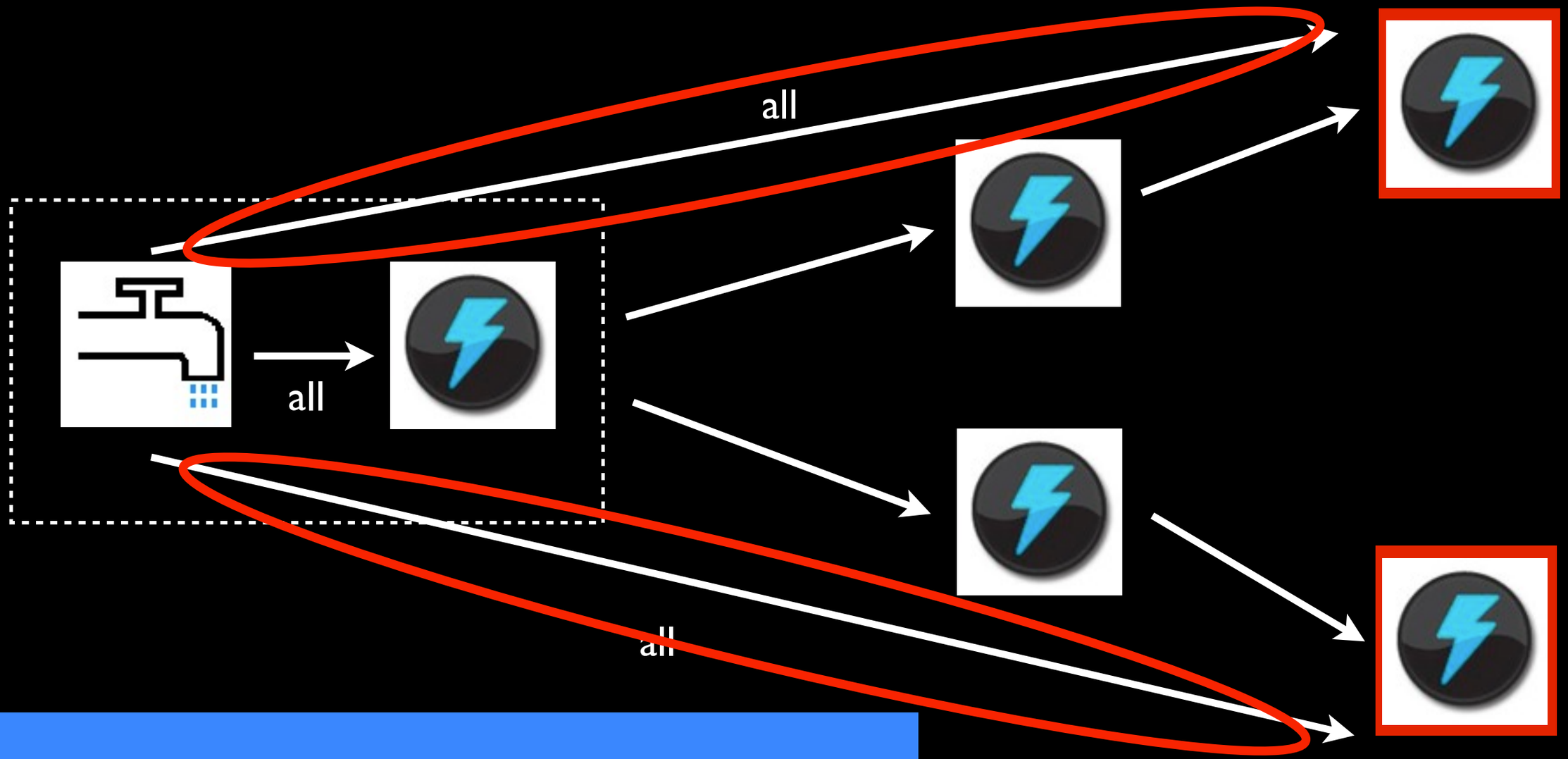
The coordinator emits a “batch” stream and a “commit stream”

Implementation



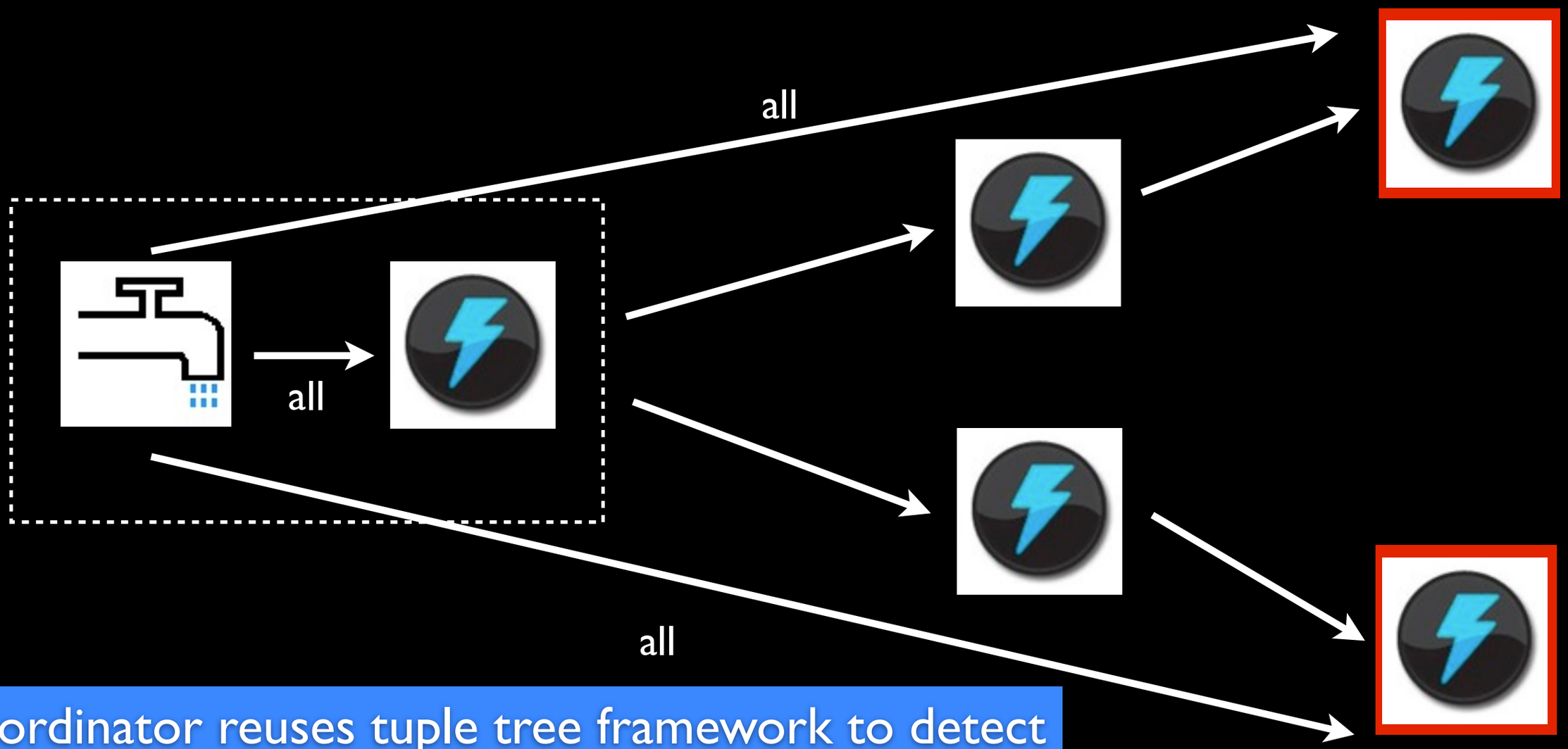
Batch stream

Implementation



Commit stream

Implementation



Coordinator reuses tuple tree framework to detect success or failure of batches or commits and replays appropriately