Jepsen

Distributed Systems Safety Research

Andi Skrgat

Content of this deck

1. A short introduction to Clojure
2. Jepsen
    a. Theoretically
    b. Interaction with Memgraph
    c. Networking
3. Failures
4. Demo

01

Clojure

Clojure as functional language

- Functional paradigm ⇒ functions as values
- Referential transparency
- Usually pure functions
- Functions as 1st class citizens
  - Return
  - Compose
  - Pass
  - ...

Linguistic perspective on Clojure

- Based on JVM stack ⇒ very easy, bidirectional integration with Java
- Compiles to bytecode on the JVM
- Compiles to Common Intermediate Language (CIL) on the CLR (used for .NET)
- Dynamically typed language
- Based on Lisp ⇒ Metaprogramming

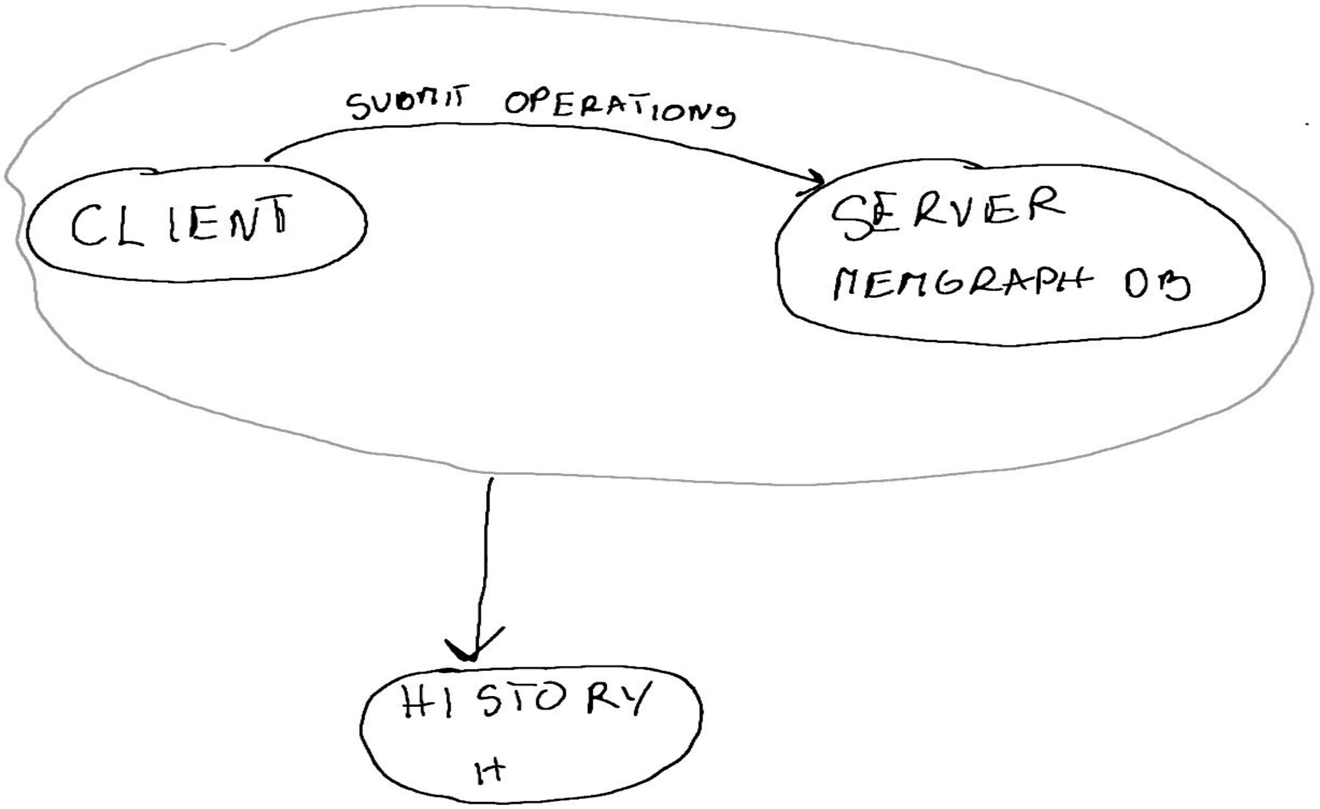
Algorithmic perspective

- Immutable data structures
- Concurrency (Software transactional memory support)
- Pass by value ⇒ efficiency?

02

What Is Jepsen?

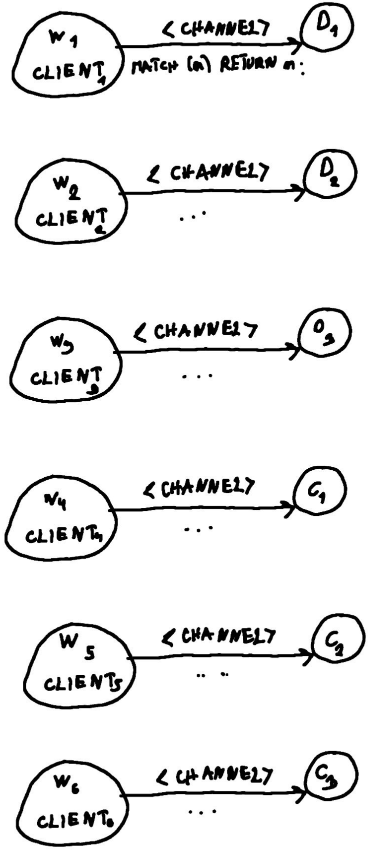
- A testing tool for distributed systems
- Injecting faults
- A collection of libraries
- Written in Clojure
- Working on binaries, not on source code ⇒ bugs in production
- Cannot prove correctness, only failures

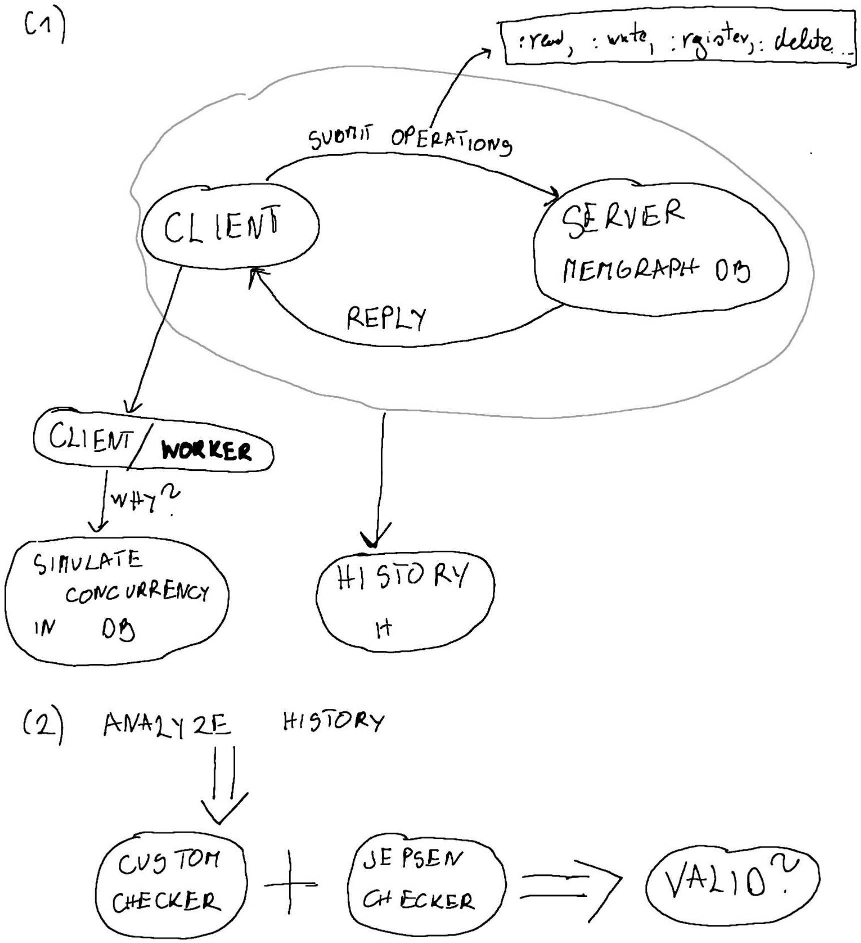

SUBMIT OPERATIONS
CLIENT
SERVER
MEMGRAPH DB
HISTORY
H

W= WORKER

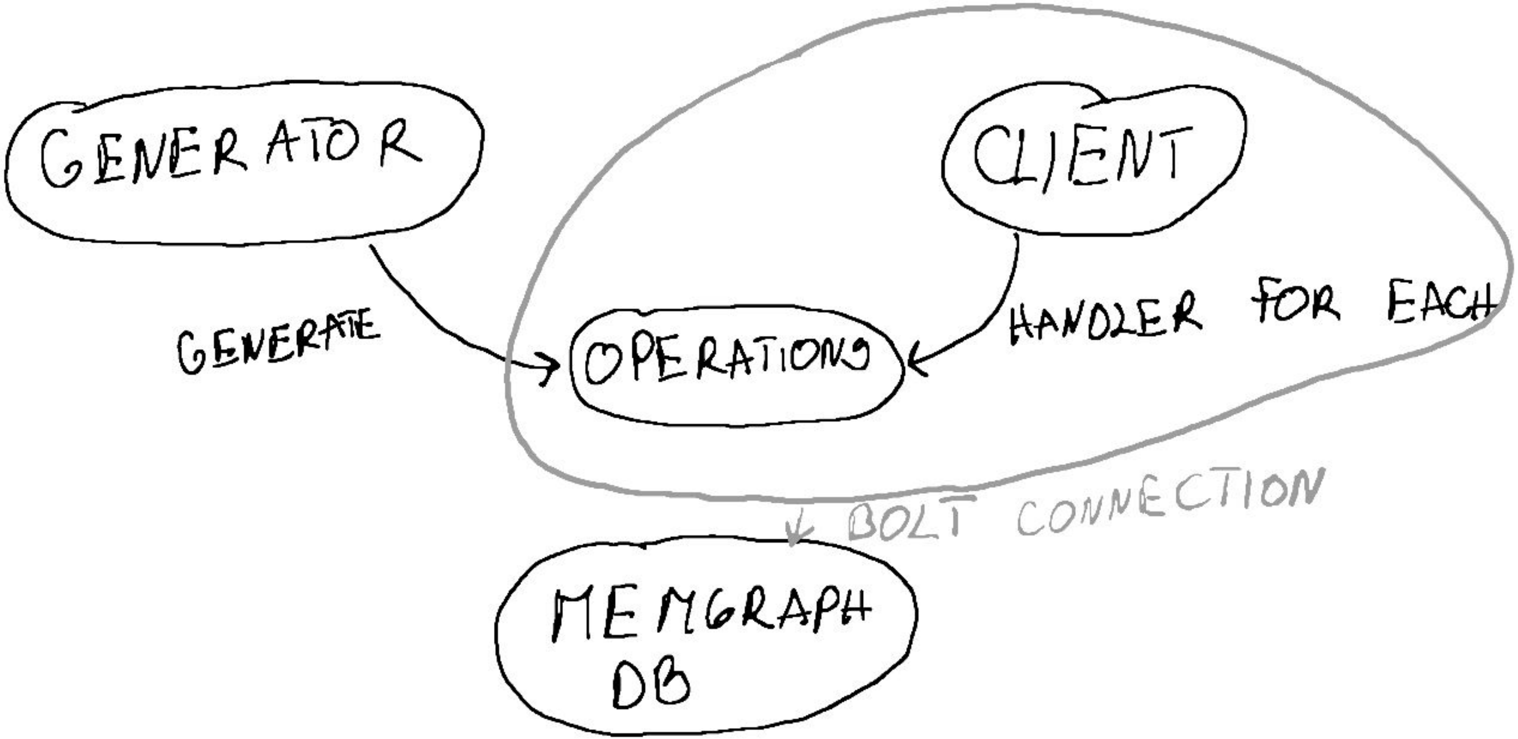
D= DATA INSTANCE = MEMGRAPH

C= COORDINATOR = MEMGRAPH



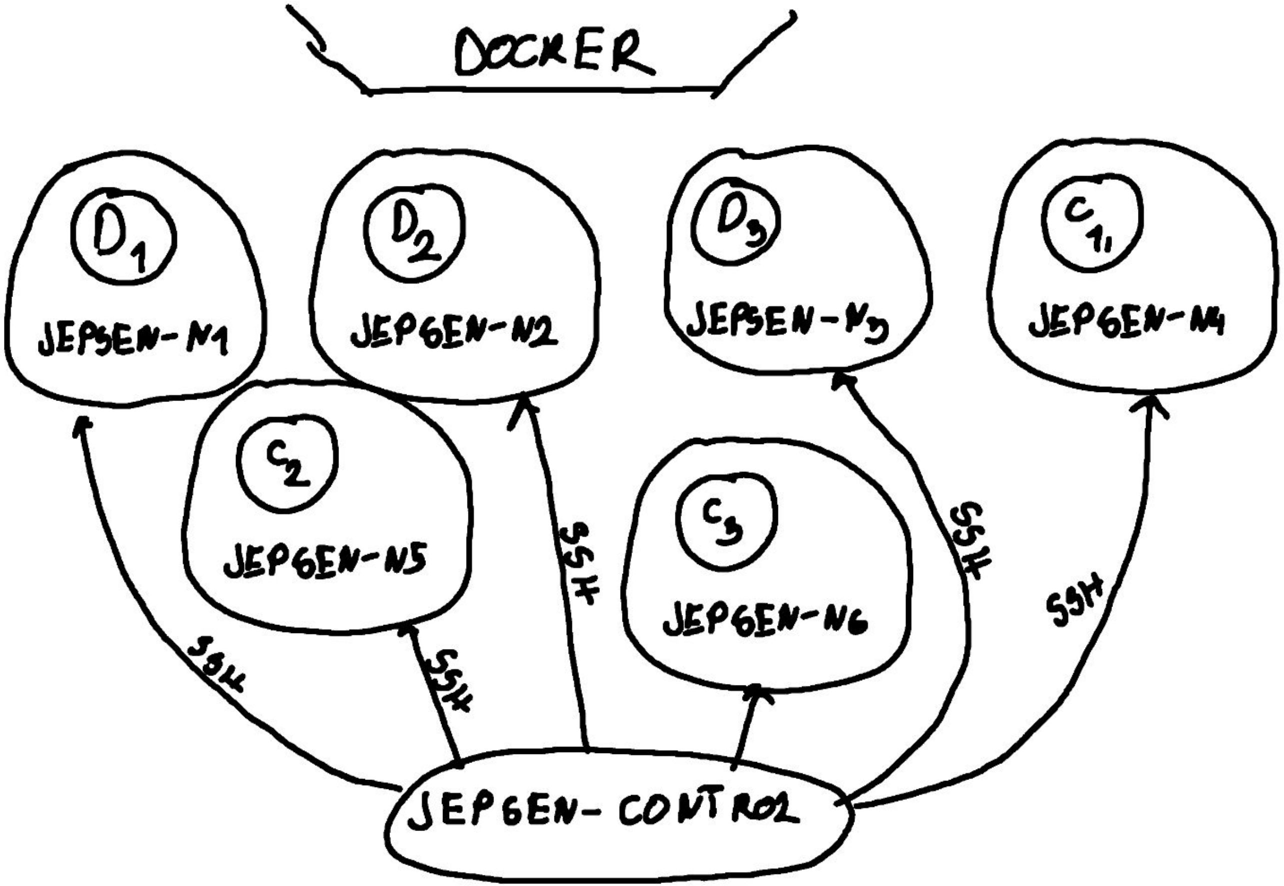

(1)
:read, :write, :register, :delete ..
SUBMIT OPERATIONS
CLIENT
SERVER
MEMGRAPH DB
REPLY
CLIENT/WORKER
WHY?
SIMULATE CONCURRENCY IN DB
HISTORY
H
(2) ANALYZE HISTORY
CUSTOM CHECKER
+
JEPSEN CHECKER
VALID?

(1) SUBMITTING OPERATIONS



Summary

- Client per worker
- Num of workers = num of instances
- Generator generates sequences of operations
- Client has a handler for each operation
- Cypher query is sent to Memgraph DB using Bolt connection
- Result is written to history
- History analyzed at the end


DOCKER
D1
JEPSEN-N1
D2
JEPSEN-N2
D3
JEPSEN-N3
C1
JEPSEN-N4
C2
JEPSEN-N5
C3
JEPSEN-N6
SSH
SSH
SSH
SSH
SSH
SSH
JEPSEN-CONTROL

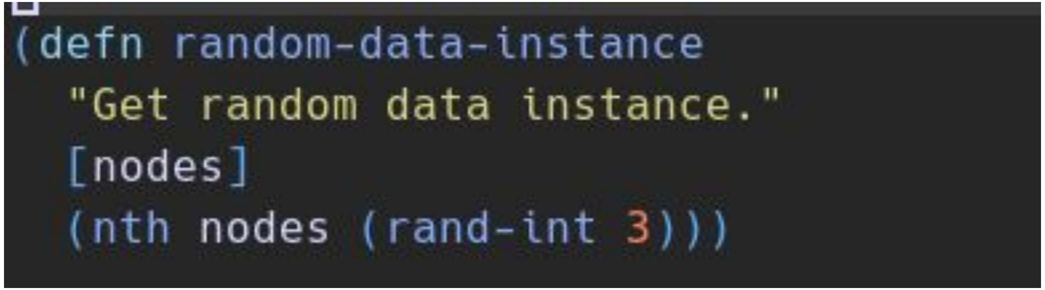
INITIALIZATION

(1) CHOOSE 1ST LEADER

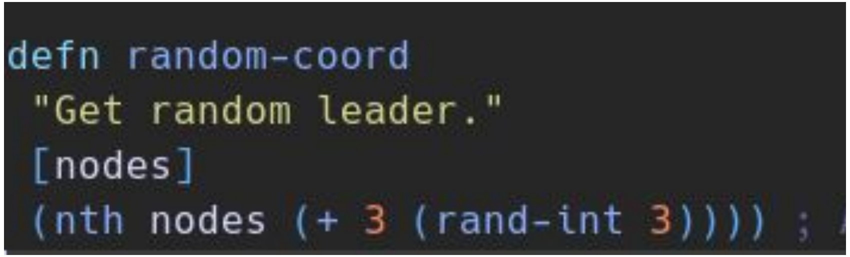
(2) CHOOSE 1ST MAIN

(3) SETUP CLUSTER

(4) INITIALIZE DATA

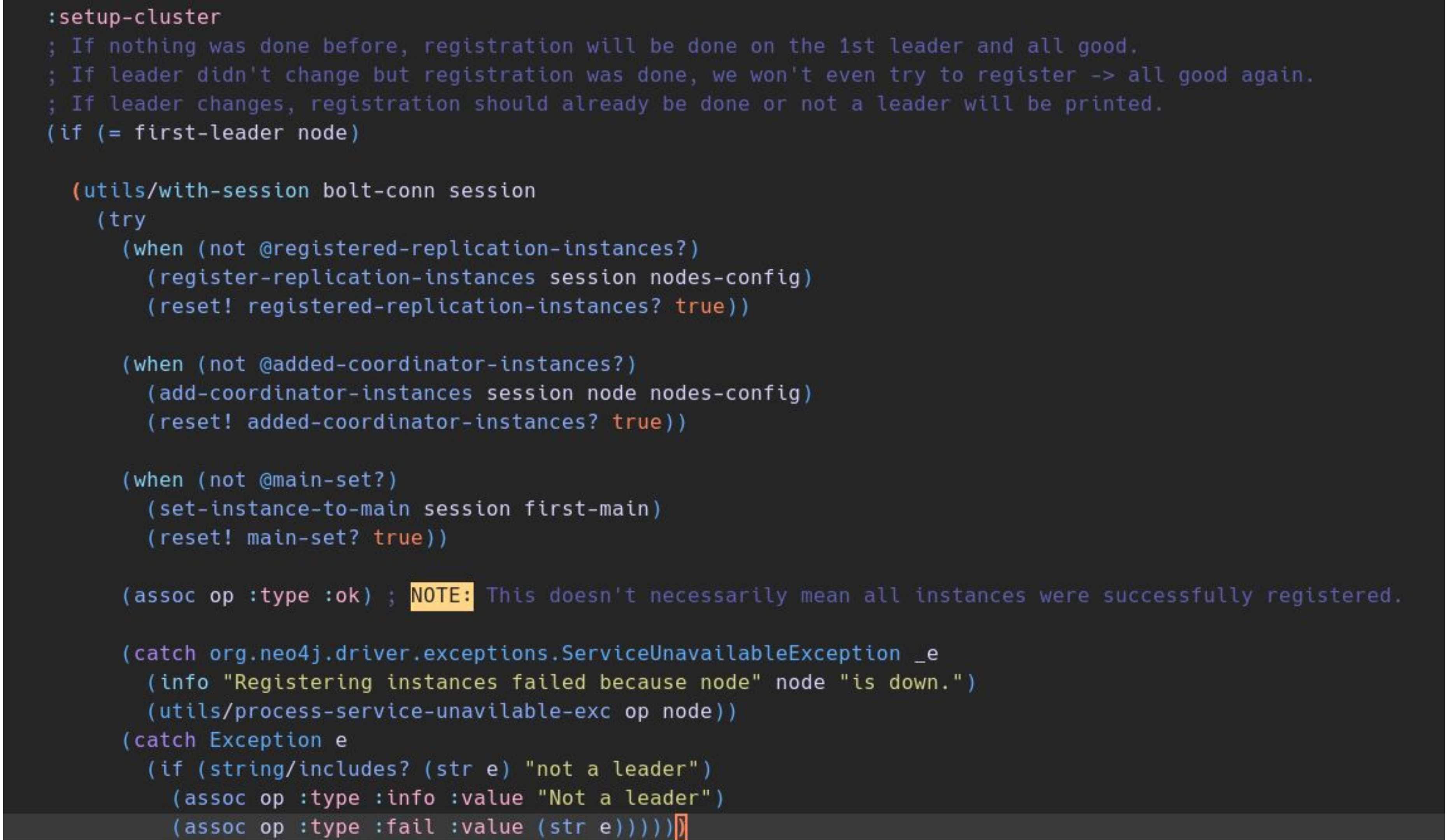


```clojure
(defn random-data-instance
  "Get random data instance."
  [nodes]
  (nth nodes (rand-int 3)))
```

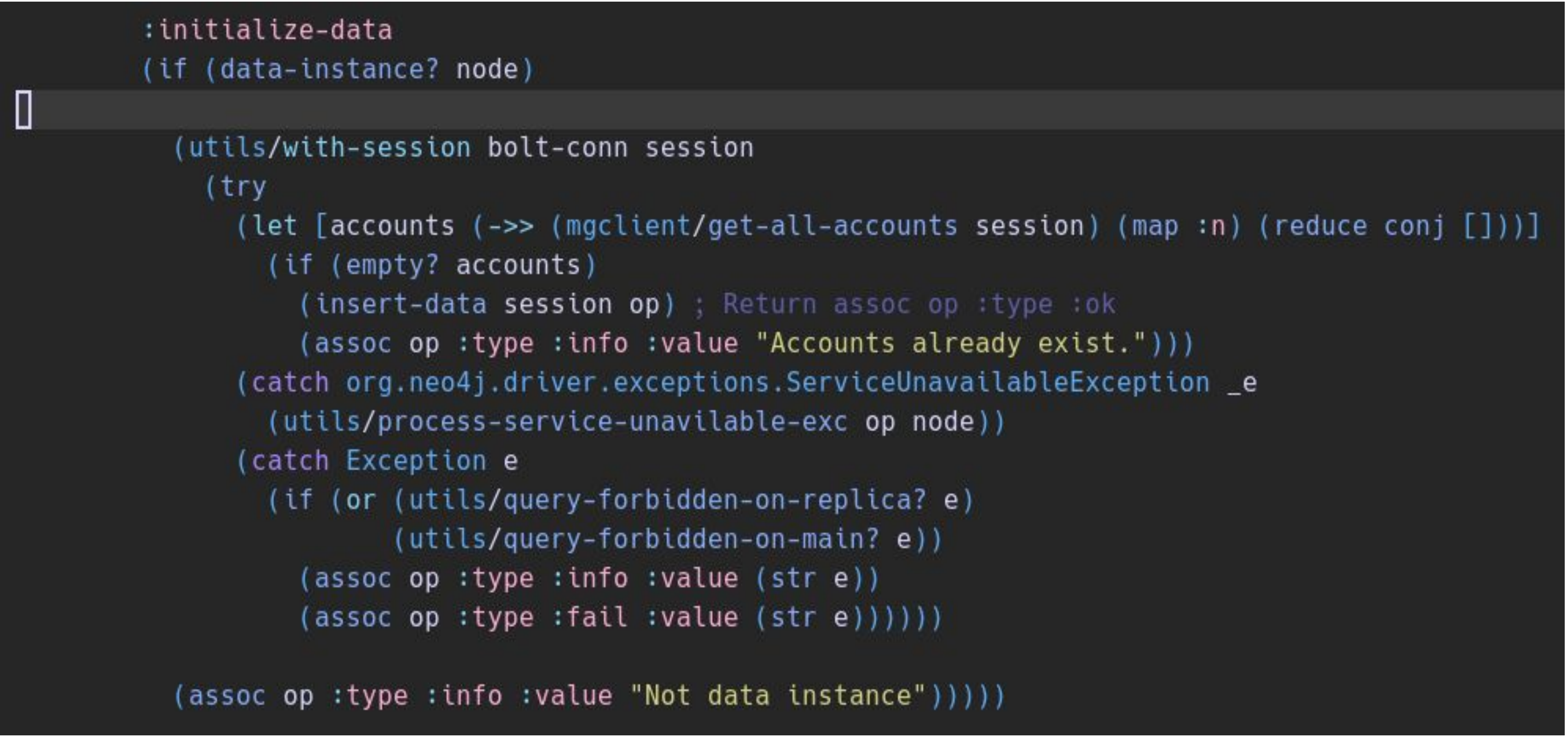


```clojure
defn random-coord
 "Get random leader."
 [nodes]
 (nth nodes (+ 3 (rand-int 3)))) ;
```

$[D_1, D_2, D_3, C_1, C_2, C_3]$



```
:setup-cluster
; If nothing was done before, registration will be done on the 1st leader and all good.
; If leader didn't change but registration was done, we won't even try to register -> all good again.
; If leader changes, registration should already be done or not a leader will be printed.
(if (= first-leader node)

  (utils/with-session bolt-conn session
    (try
      (when (not @registered-replication-instances?)
        (register-replication-instances session nodes-config)
        (reset! registered-replication-instances? true))

      (when (not @added-coordinator-instances?)
        (add-coordinator-instances session node nodes-config)
        (reset! added-coordinator-instances? true))

      (when (not @main-set?)
        (set-instance-to-main session first-main)
        (reset! main-set? true))

      (assoc op :type :ok) ; NOTE: This doesn't necessarily mean all instances were successfully registered.

      (catch org.neo4j.driver.exceptions.ServiceUnavailableException _e
        (info "Registering instances failed because node" node "is down.")
        (utils/process-service-unavilable-exc op node))
      (catch Exception e
        (if (string/includes? (str e) "not a leader")
          (assoc op :type :info :value "Not a leader")
          (assoc op :type :fail :value (str e))))))
```



```
        :initialize-data
        (if (data-instance? node)

          (utils/with-session bolt-conn session
            (try
              (let [accounts (->> (mgclient/get-all-accounts session) (map :n) (reduce conj []))]
                (if (empty? accounts)
                  (insert-data session op) ; Return assoc op :type :ok
                  (assoc op :type :info :value "Accounts already exist.")))
              (catch org.neo4j.driver.exceptions.ServiceUnavailableException _e
                (utils/process-service-unavilable-exc op node))
              (catch Exception e
                (if (or (utils/query-forbidden-on-replica? e)
                        (utils/query-forbidden-on-main? e))
                  (assoc op :type :info :value (str e))
                  (assoc op :type :fail :value (str e))))))

          (assoc op :type :info :value "Not data instance")))))
```

Summary

- Initialization is done through two operations = **:setup_cluster** and **:initialize_data**
- First leader is chosen randomly
- First main is chosen randomly

HA Phase III-IV-V

Healthy cluster state

Legend & port specs

data instance

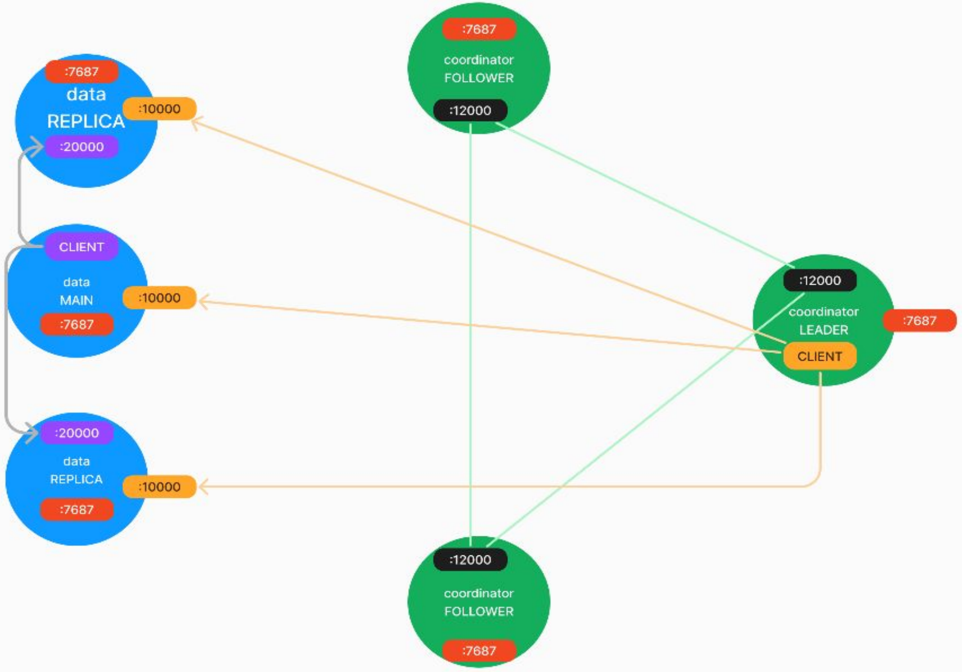
coordinator in...

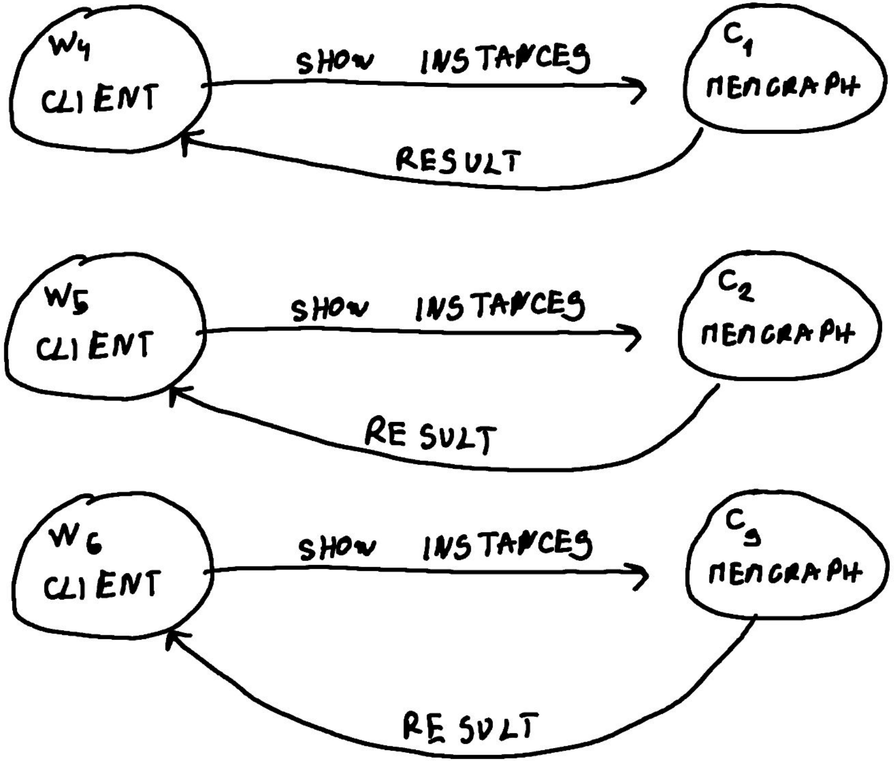
:7687
Bolt
FLAG: --bolt-port
ENV : MEMGRAPH_BOLT_PORT

:20000
Replication
FLAG: --replication-port

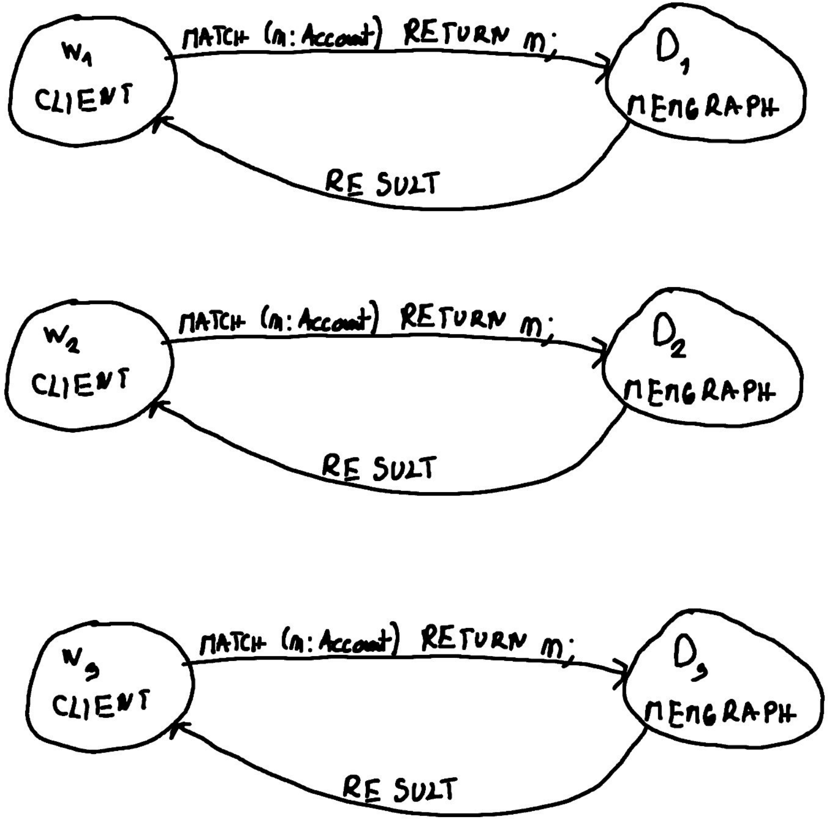
:10000
Management
FLAG: --management-port
ENV : MEMGRAPH_MANAGEMENT_PORT

:12000
Raft
FLAG: --coordinator-port
ENV : MEMGRAPH_COORDINATOR_PORT





RESULT ENDS IN HISTORY


W1
CLIENT
MATCH (m:Account) RETURN m;
D1
MEMGRAPH
RESULT
W2
CLIENT
MATCH (m:Account) RETURN m;
D2
MEMGRAPH
RESULT
W3
CLIENT
MATCH (m:Account) RETURN m;
D3
MEMGRAPH
RESULT

RESULT WRITTEN TO HISTORY

TRANSFER MONEY =

```
MATCH (m: Accont {id: id13)
SET m.balance = n.balance + x
RETURN m;
```

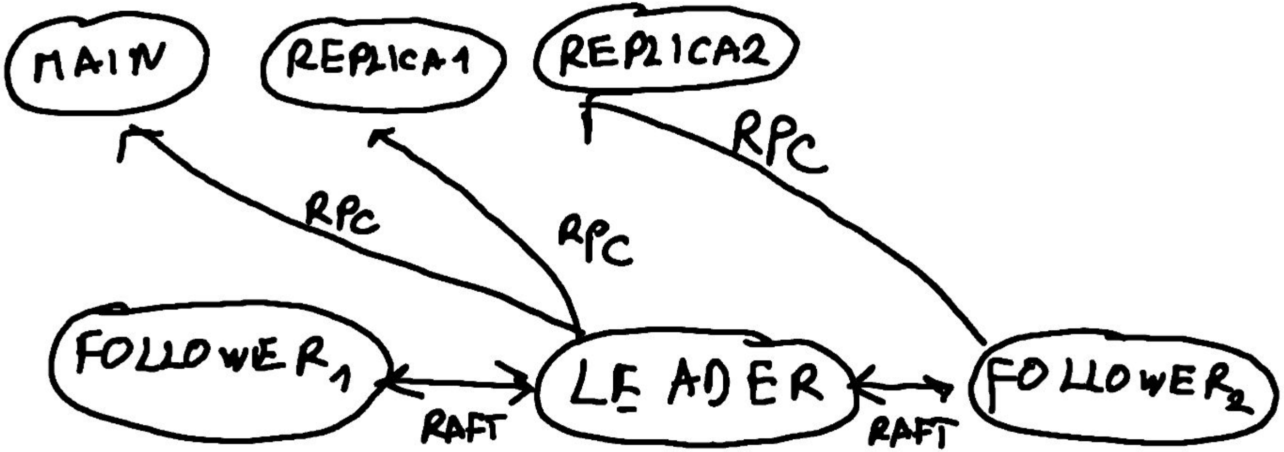
Summary

- **:read-balances** ⇒ read balances of all accounts, write results to history
- **:show-instances** ⇒ run `SHOW INSTANCES`, write results to history
- **:transfer-money** ⇒ update accounts, write results to history

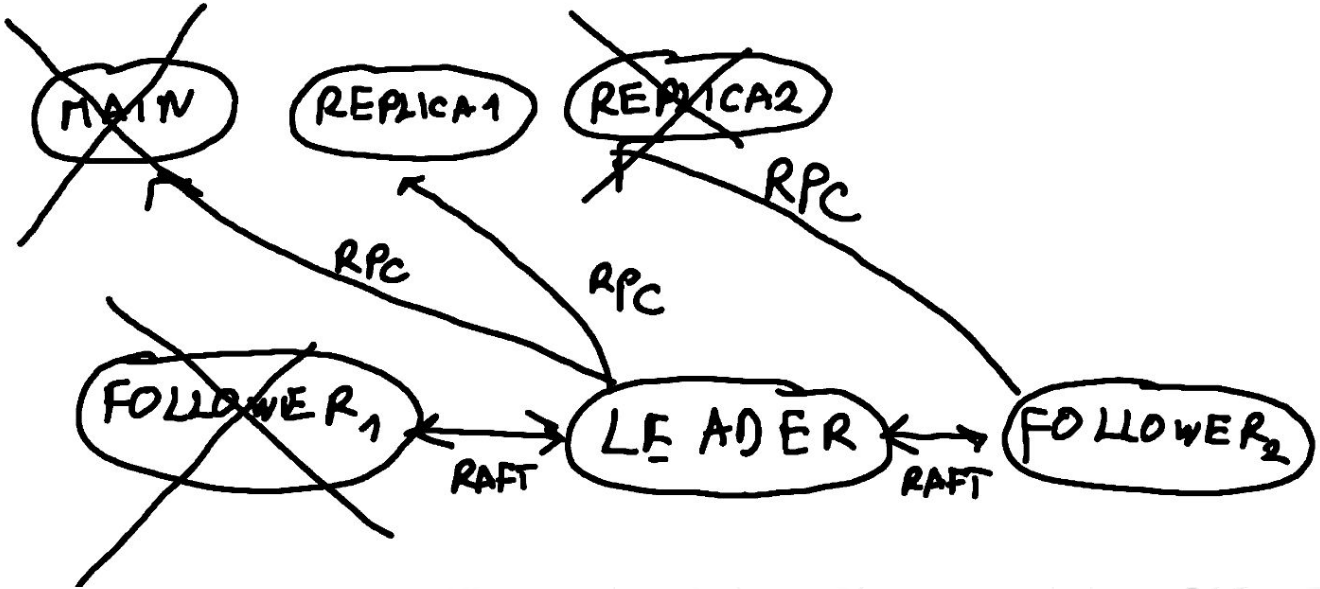
03

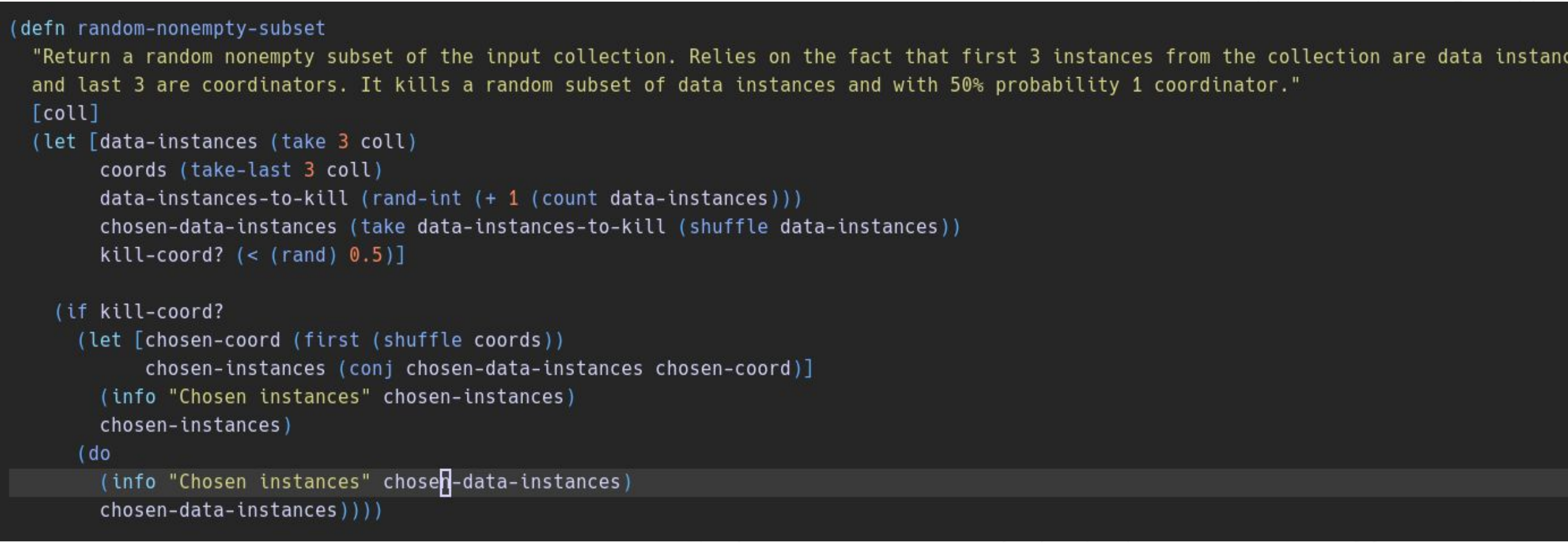
Introduce failures

(1) KILL A SUBSET OF NODES



(1) KILL A SUBSET OF NODES





```clojure
(defn random-nonempty-subset
  "Return a random nonempty subset of the input collection. Relies on the fact that first 3 instances from the collection are data instanc
  and last 3 are coordinators. It kills a random subset of data instances and with 50% probability 1 coordinator."
  [coll]
  (let [data-instances (take 3 coll)
        coords (take-last 3 coll)
        data-instances-to-kill (rand-int (+ 1 (count data-instances)))
        chosen-data-instances (take data-instances-to-kill (shuffle data-instances))
        kill-coord? (< (rand) 0.5)]

    (if kill-coord?
      (let [chosen-coord (first (shuffle coords))
            chosen-instances (conj chosen-data-instances chosen-coord)]
        (info "Chosen instances" chosen-instances)
        chosen-instances)
      (do
        (info "Chosen instances" chosen-data-instances)
        chosen-data-instances))))
```

(2) PARTITION RANDOM HALVES

$[\, D_1 \;\; D_2 \;\; D_3 \;\; C_1 \;\; C_2 \;\; C_3 \,]$

⇓ SHUFFLE

$[\, D_2 \;\; C_1 \;\; C_3 \;\; D_1 \;\; D_3 \;\; C_2 \,]$

⇓ BISECT

$(\; [\, D_2 \;\; C_1 \;\; C_3 \,] \quad [\, D_1 \;\; D_3 \;\; C_2 \,] \;)$

⇓

CUT THE NETWORK IN HALF

04

Demo

Thank you for your time!

www.memgraph.com

References

(1) https://www.cs.cmu.edu/~rwh/students/okasaki.pdf

(2) https://www.manning.com/books/clojure-in-action

(3) https://jepsen.io/

(4) https://stackoverflow.com/questions/5669933/is-clojure-compiled-or-interpreted