AARHUS UNIVERSITET

Microservices and DevOps

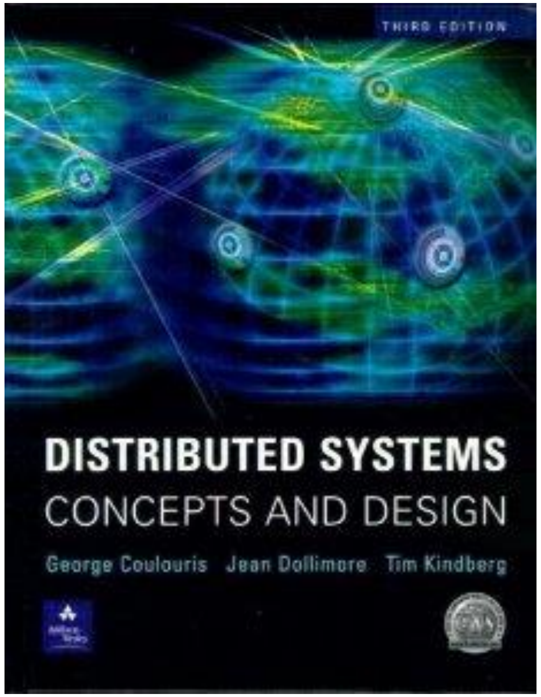
Scalable Microservices

Replication and Redundancy

Henrik Bærbak Christensen

Literature

- Colouris, Dollimore, Kindberg, 2001
  - Gets deep into the details of reliable communication, byzantine failures, etc.
  - We approach it from the architectural level
    - Terminology
    - Overall protocol



AARHUS UNIVERSITET

Replication

- Replication: Maintenance of copies of data at multiple computers

- As always – not only one architectural quality is affected but several
  - Availability increase:
    - Our primary concern in MSDO
    - *If one man is down, another man can take over*
  - Performance increase:
    - More men can pull more load

Bass Tactic: Maintain multiple copies of data

Replication

- But
  - Cost:
    - Two nodes cost more than one
  - Reliability:
    - More complex algorithms increase probability of error (fail-over, ping-echo, voting, …)
  - Data consistency
    - What if A reads from node X while B writes to node Y ?
    - CAP theorem
      - Consistency: All see same data
      - Availability: Every request is served
      - Partition Tolerance: Operational despite arbitrary failures
    - RDB goes for C while NoSQL (generally) goes for A

Discussion

- *Redundancy:* In engineering, redundancy is the duplication of critical components or functions of a system with the intention of increasing reliability of the system, usually in the form of a backup or fail-safe.
  [Wikipedia]

- What is the difference from *replication?*

- Which Nygard pattern covers ‘redundancy’?

Availability Calculations

- If a system of *n* replicated servers in which each server has a probability, *p*, of failing, then the system has total probability
- p^n of failing
- Ex
  - p = 5% (0.05)
    - (72 minutes every 24h)
  - n = 3
  - Overall failure rate:
  - $0.05^3 = 0.000125$ = 0,125 per mille
    - (10 seconds every 24h)

That is:
$1 - p^n$ availability

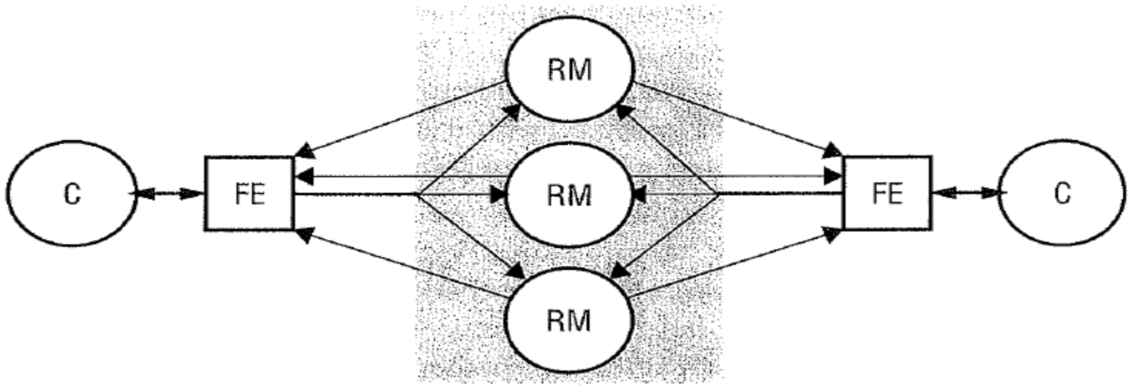
Active Replication

Fully redundant primaries

Architecture

- Clients *multicast* requests to all nodes
- All nodes process identically but independently and reply
- Front-End receives answers
  - May do one of several things: Use first one, compare, vote…

Active replication



Active Replication

- Benefits
  - No performance penalty for failures
    - (no promotion of a slave to become primary, all are primary)
  - (Potentially) Simpler servers
    - Less need for hand-shaking
    - Still need *state resynchronization* code in case a node has failed and need to get up-to-date
- Liabilities
  - No better performance than if using only one node
  - Complexity in front-end
    - Multicast, voting, ...
  - Consistency amongst nodes...

AARHUS UNIVERSITET

Passive Replication

Master/slave

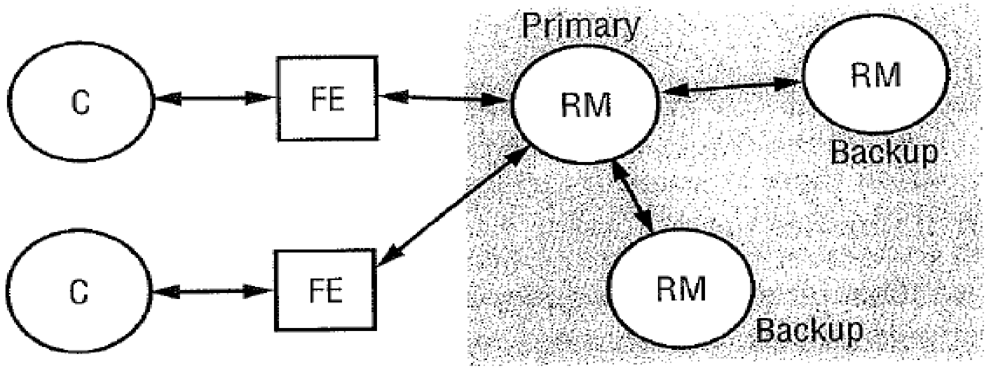
Primary/backup

Architecture

- One node is *primary*
  - executes operations (notably writes/updates)
  - sends copies to slaves (in case of write/update)
- Primary failure
  - One slave is promoted to become primary

What is the FrontEnd = FE?

The passive (primary-backup) model for fault tolerance



Passive Replication

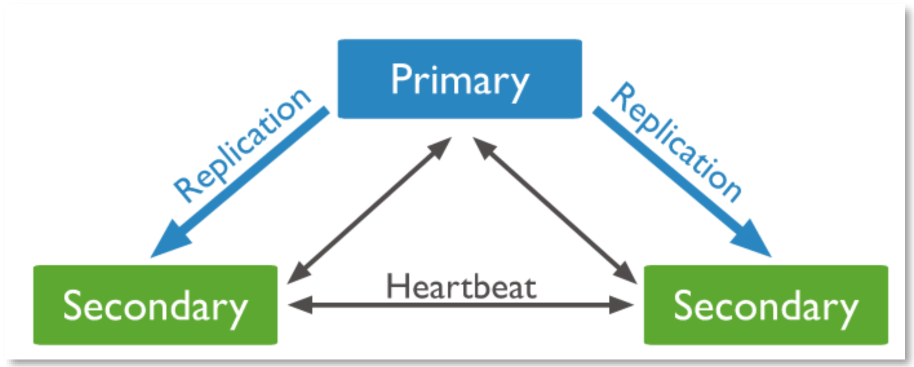
- Liabilities
  - Additional complexity in algorithms to keep slaves up-to-date, ensuring consistency, etc...
  - It takes time for the slaves to note that the primary is gone and promote one as new primary
    - MongoDB – up to one minute
    - Meanwhile the clients are waiting – *slow response*
  - Primary takes all requests
    - No obvious performance gain from balancing load
  - Storage requirements
    - N nodes require N times storage
    - EcoSense: for every 1TB extra diskspace we pay for 3 TB

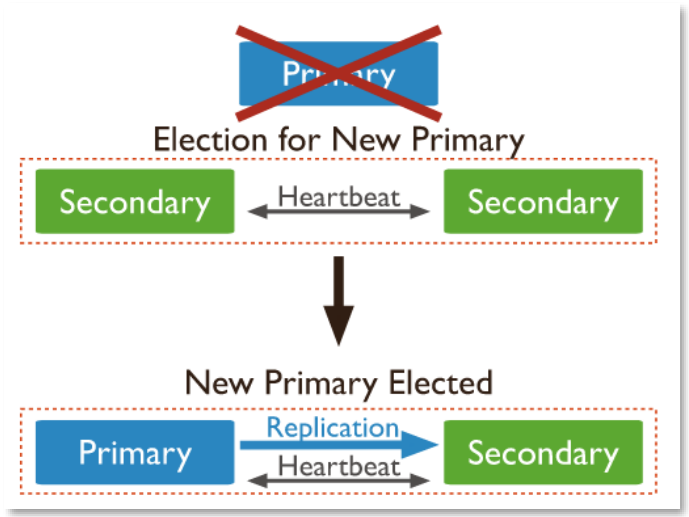
Example: MongoDB

- MongoDB: **Replica Sets**
  - Built to run on commodity hardware
    - That is, the stuff that you buy in a shop, not necessarily the stuff the IT department has in the server room
    - Commodity disks, however, have higher failure rates
  - But – avoid failures by using replication
    - Three is a recommendation
    - Two + arbiter is another option
      - Arbiter does not store anything but participate in electing a new primary

Example: MongoDB

- Use heart beat to monitor the replica set





AARHUS UNIVERSITET

Example: MongoDB

- Highly configurable
  - Write concern

Unacknowledged	happy go lucky
Acknowledged	primary has received write request
Journaled	primary has journaled the write request
Replica Acknowledged	n replicas have received write request

  - Read concerns
    - Writes go to primary, but all reads may go to slaves
      - Boosts performance
      - Sacrifice consistency, you may get old data!
      - MongoDB is *eventual consistent*

AARHUS UNIVERSITET

Redis Cluster

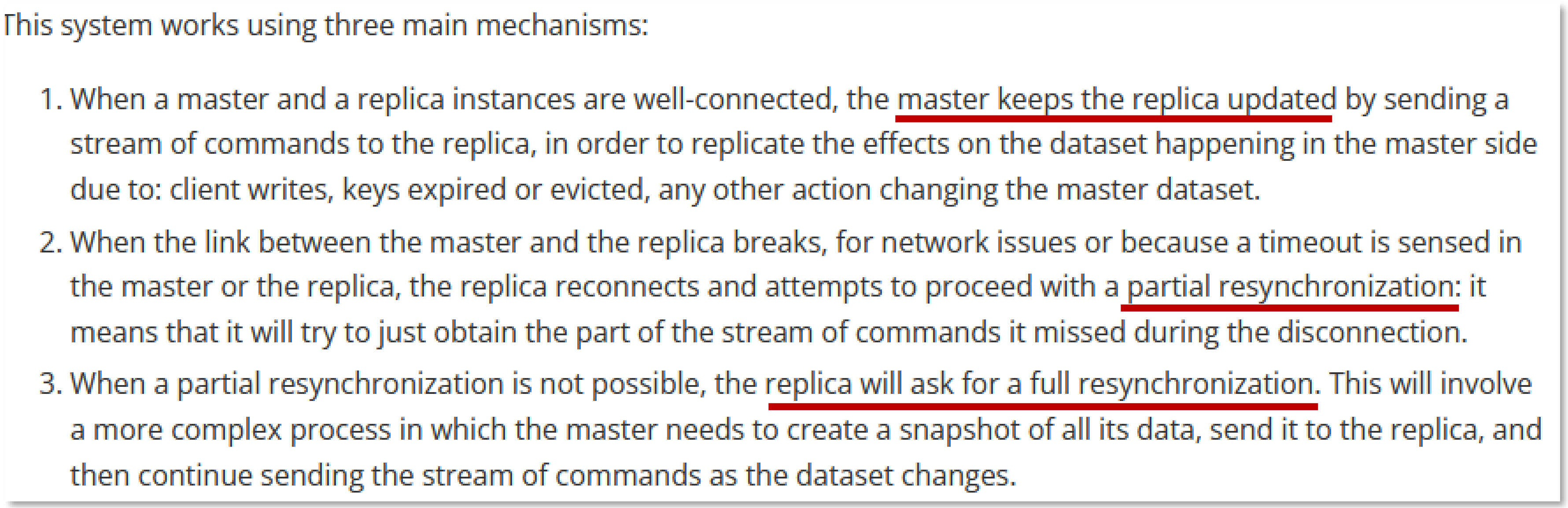
Replication and Sharding and Availability
“Det er jo hele tre ting?!?”

Replication

- The replication model of Redis is very simple
- One instance acts as the ‘master’
  - Just like we have used it so far..
    - Assume it is on node ‘redis1:6379’
- Then we can start another instance and make it *slave*
  - Start another instance, open the ‘redis-cli’ and issue
    - ‘replicaof redis1 6379’

AARHUS UNIVERSITET

Replication

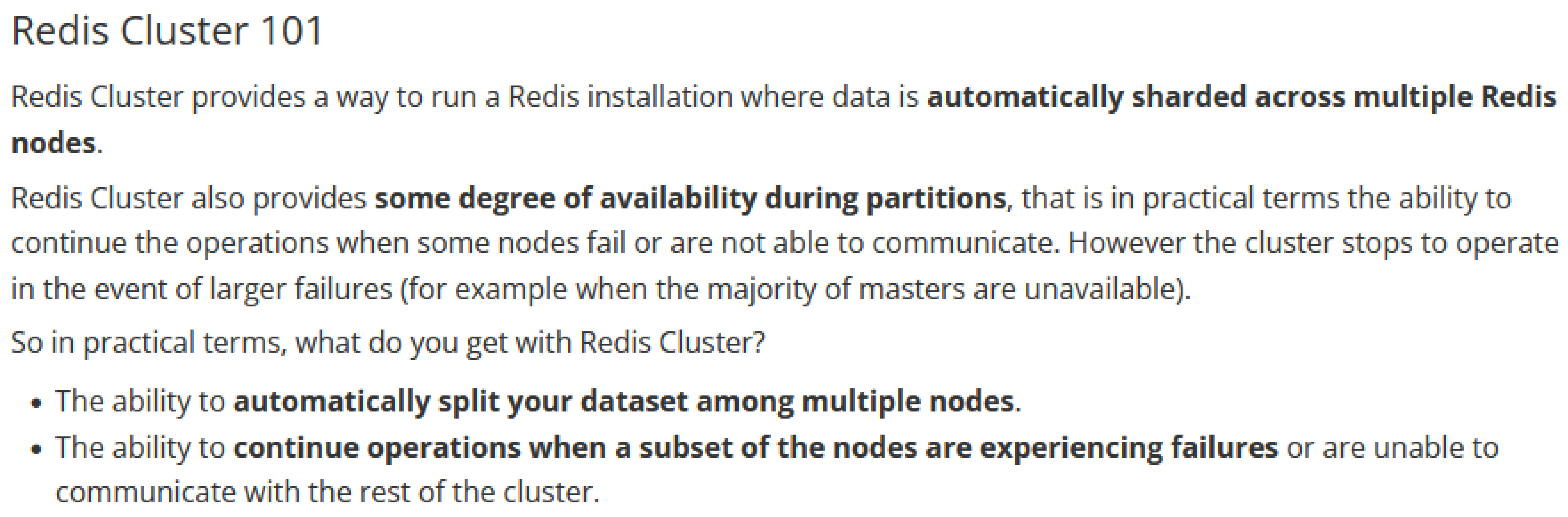

This system works using three main mechanisms:

1. When a master and a replica instances are well-connected, the master keeps the replica updated by sending a stream of commands to the replica, in order to replicate the effects on the dataset happening in the master side due to: client writes, keys expired or evicted, any other action changing the master dataset.
2. When the link between the master and the replica breaks, for network issues or because a timeout is sensed in the master or the replica, the replica reconnects and attempts to proceed with a partial resynchronization: it means that it will try to just obtain the part of the stream of commands it missed during the disconnection.
3. When a partial resynchronization is not possible, the replica will ask for a full resynchronization. This will involve a more complex process in which the master needs to create a snapshot of all its data, send it to the replica, and then continue sending the stream of commands as the dataset changes.

- You can read from the replica *but you cannot write to it*
  - And you have to connect to 'the one you need to speak to'
    - Redis1 ? Redis2?

Redis Cluster

- Redis Cluster provides
  - Sharding and Replication and FailOver



Redis Cluster 101

Redis Cluster provides a way to run a Redis installation where data is **automatically sharded across multiple Redis nodes**.

Redis Cluster also provides **some degree of availability during partitions**, that is in practical terms the ability to continue the operations when some nodes fail or are not able to communicate. However the cluster stops to operate in the event of larger failures (for example when the majority of masters are unavailable).

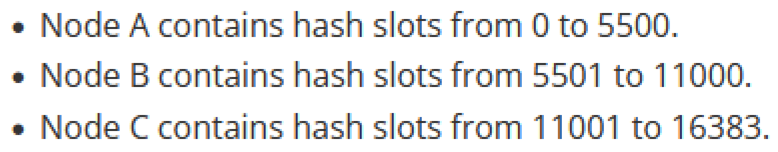
So in practical terms, what do you get with Redis Cluster?

- The ability to **automatically split your dataset among multiple nodes**.
- The ability to **continue operations when a subset of the nodes are experiencing failures** or are unable to communicate with the rest of the cluster.

https://redis.io/topics/cluster-tutorial

Sharding

- Every key is part of a **hash slot**
  - Redis has exactly 16384 hash slots
    - Every key is mapped to one of these hash slots
- Every node (master) is responsible for a subset, e.g.



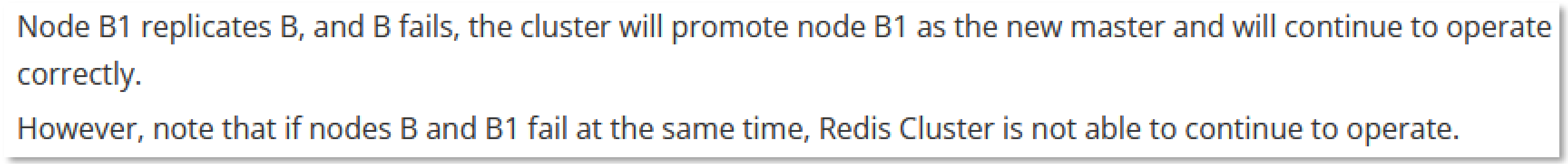
- Node A contains hash slots from 0 to 5500.
- Node B contains hash slots from 5501 to 11000.
- Node C contains hash slots from 11001 to 16383.

- If we add two more nodes (masters), we just move the relevant hash slots to the new masters…
  - Uhum, transactions need to cover only keys in the same slot ☹
    - So there is a mechanism to guaranty that… anyway…

Master-Slave

- So if we loose master B, then all keys in hash slot 5501 – 11000 are lost ☹

- Solved by having a replica of B, let us call it B1

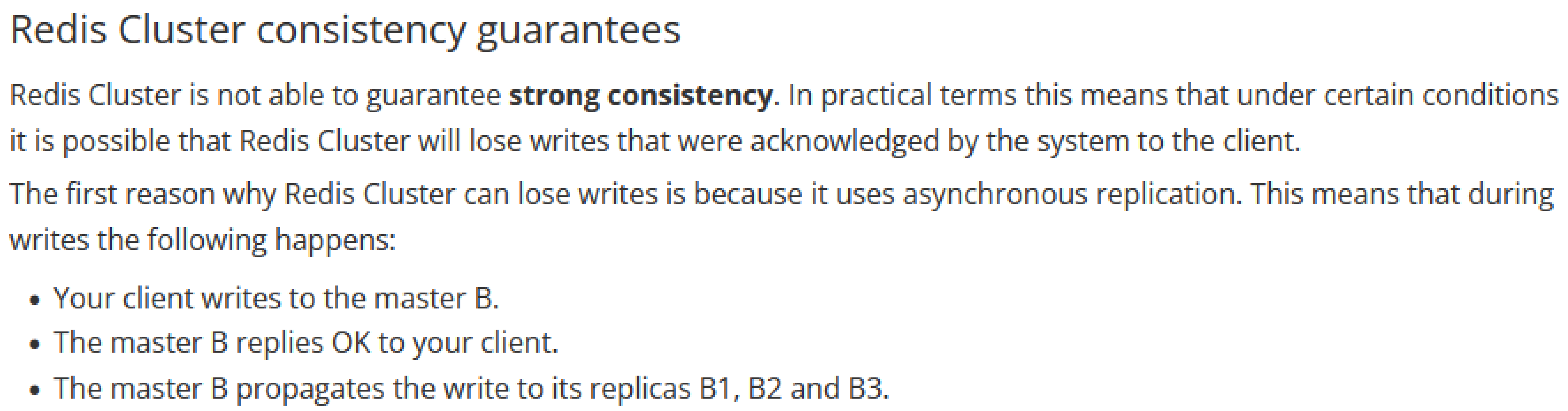


Node B1 replicates B, and B fails, the cluster will promote node B1 as the new master and will continue to operate correctly.

However, note that if nodes B and B1 fail at the same time, Redis Cluster is not able to continue to operate.

Eventual Consistency (?)

- Only a weak consistency…



Redis Cluster consistency guarantees

Redis Cluster is not able to guarantee **strong consistency**. In practical terms this means that under certain conditions it is possible that Redis Cluster will lose writes that were acknowledged by the system to the client.

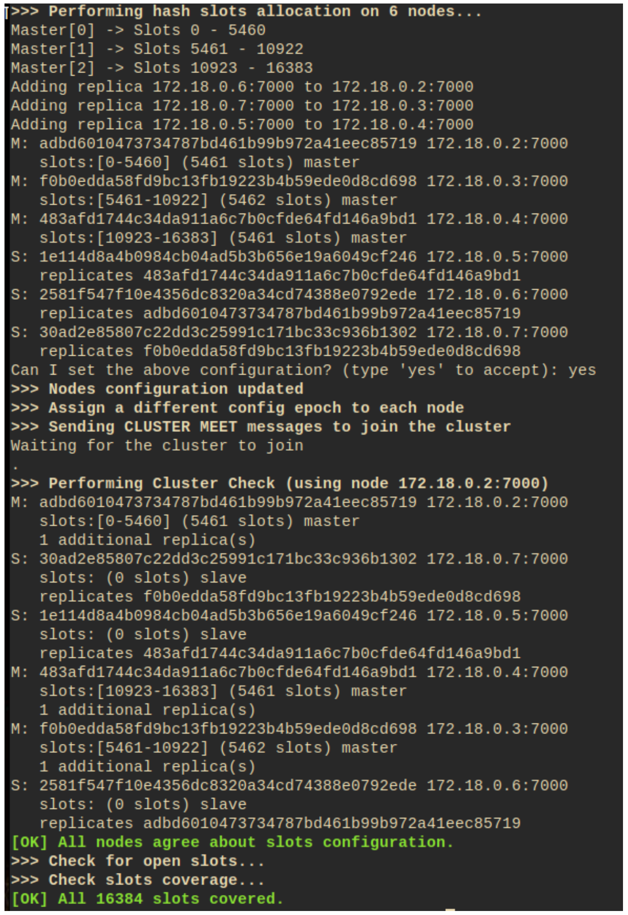
The first reason why Redis Cluster can lose writes is because it uses asynchronous replication. This means that during writes the following happens:

- Your client writes to the master B.
- The master B replies OK to your client.
- The master B propagates the write to its replicas B1, B2 and B3.

- If B fails before propagating the write, the data is lost!

Redis cluster

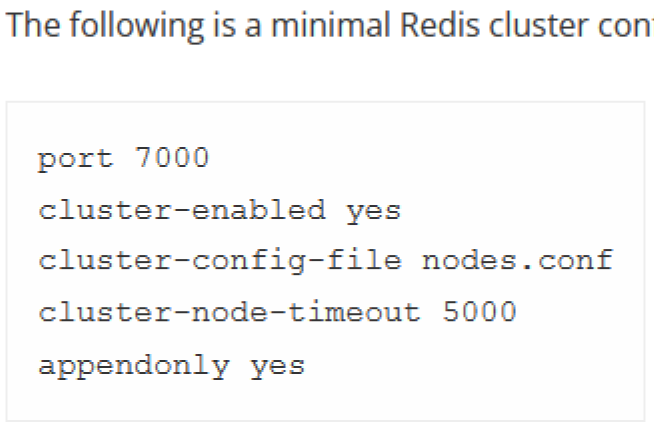
- The **minimal cluster** requires at least three masters…
- So, you need 6 instances!
  - Three masters (minimum)
  - And each one with a replica
- Ideally, the machines are geographically distributed, of course



```
>>> Performing hash slots allocation on 6 nodes...
Master[0] -> Slots 0 - 5460
Master[1] -> Slots 5461 - 10922
Master[2] -> Slots 10923 - 16383
Adding replica 172.18.0.6:7000 to 172.18.0.2:7000
Adding replica 172.18.0.7:7000 to 172.18.0.3:7000
Adding replica 172.18.0.5:7000 to 172.18.0.4:7000
M: adbd6010473734787bd461b99b972a41eec85719 172.18.0.2:7000
   slots:[0-5460] (5461 slots) master
M: f0b0edda58fd9bc13fb19223b4b59ede0d8cd698 172.18.0.3:7000
   slots:[5461-10922] (5462 slots) master
M: 483afd1744c34da911a6c7b0cfde64fd146a9bd1 172.18.0.4:7000
   slots:[10923-16383] (5461 slots) master
S: 1e114d8a4b0984cb04ad5b3b656e19a6049cf246 172.18.0.5:7000
   replicates 483afd1744c34da911a6c7b0cfde64fd146a9bd1
S: 2581f547f10e4356dc8320a34cd74388e0792ede 172.18.0.6:7000
   replicates adbd6010473734787bd461b99b972a41eec85719
S: 30ad2e85807c22dd3c25991c171bc33c936b1302 172.18.0.7:7000
   replicates f0b0edda58fd9bc13fb19223b4b59ede0d8cd698
Can I set the above configuration? (type 'yes' to accept): yes
>>> Nodes configuration updated
>>> Assign a different config epoch to each node
>>> Sending CLUSTER MEET messages to join the cluster
Waiting for the cluster to join
.
>>> Performing Cluster Check (using node 172.18.0.2:7000)
M: adbd6010473734787bd461b99b972a41eec85719 172.18.0.2:7000
   slots:[0-5460] (5461 slots) master
   1 additional replica(s)
S: 30ad2e85807c22dd3c25991c171bc33c936b1302 172.18.0.7:7000
   slots: (0 slots) slave
   replicates f0b0edda58fd9bc13fb19223b4b59ede0d8cd698
S: 1e114d8a4b0984cb04ad5b3b656e19a6049cf246 172.18.0.5:7000
   slots: (0 slots) slave
   replicates 483afd1744c34da911a6c7b0cfde64fd146a9bd1
M: 483afd1744c34da911a6c7b0cfde64fd146a9bd1 172.18.0.4:7000
   slots:[10923-16383] (5461 slots) master
   1 additional replica(s)
M: f0b0edda58fd9bc13fb19223b4b59ede0d8cd698 172.18.0.3:7000
   slots:[5461-10922] (5462 slots) master
   1 additional replica(s)
S: 2581f547f10e4356dc8320a34cd74388e0792ede 172.18.0.6:7000
   slots: (0 slots) slave
   replicates adbd6010473734787bd461b99b972a41eec85719
[OK] All nodes agree about slots configuration.
>>> Check for open slots...
>>> Check slots coverage...
[OK] All 16384 slots covered.
```

Process

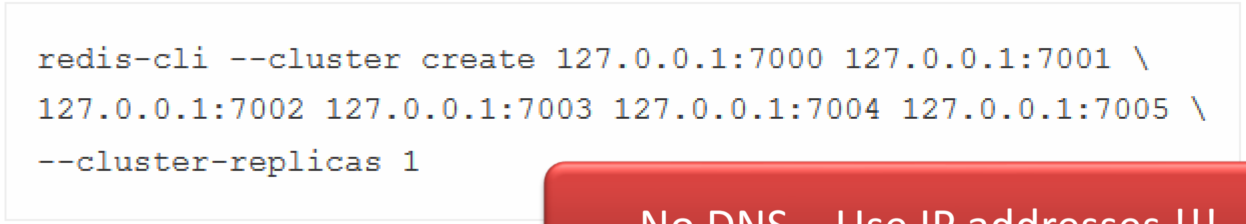
- Start 6 instances, configured to *cluster mode*



The following is a minimal Redis cluster configuration file:

```
port 7000
cluster-enabled yes
cluster-config-file nodes.conf
cluster-node-timeout 5000
appendonly yes
```

- Start the ‘redis-cli’ in any node and issue the cluster command



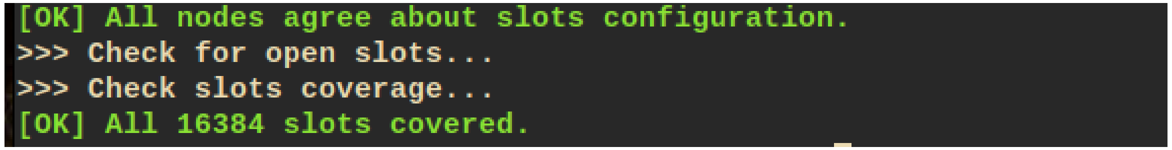
```
redis-cli --cluster create 127.0.0.1:7000 127.0.0.1:7001 \
127.0.0.1:7002 127.0.0.1:7003 127.0.0.1:7004 127.0.0.1:7005 \
--cluster-replicas 1
```

No DNS – Use IP addresses !!!

AARHUS UNIVERSITET

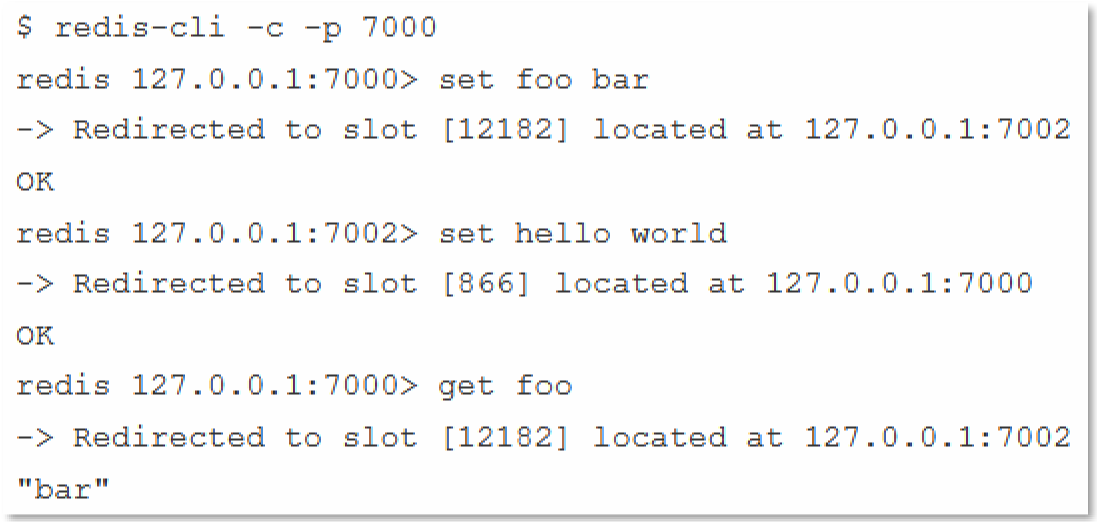
And…

- That is it!



```
[OK] All nodes agree about slots configuration.
>>> Check for open slots...
>>> Check slots coverage...
[OK] All 16384 slots covered.
```

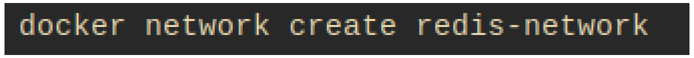
- Open a cli in *any node* in **cluster mode (-c)**



```
$ redis-cli -c -p 7000
redis 127.0.0.1:7000> set foo bar
-> Redirected to slot [12182] located at 127.0.0.1:7002
OK
redis 127.0.0.1:7002> set hello world
-> Redirected to slot [866] located at 127.0.0.1:7000
OK
redis 127.0.0.1:7000> get foo
-> Redirected to slot [12182] located at 127.0.0.1:7002
"bar"
```

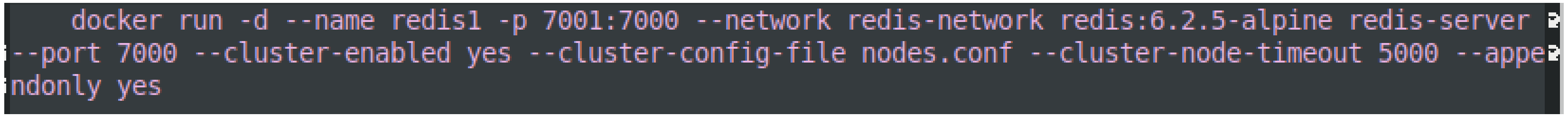
Dockerizing a Manual Test

- The fine tutorial requires redis installed on you machine.
- It is (almost) as easy to run using Docker but…
  - You need to create a docker network
    - So the instances can talk to each other…



```
docker network create redis-network
```

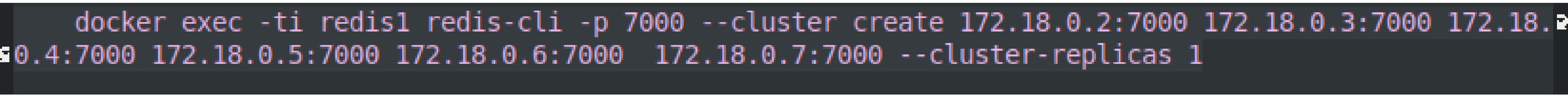
  - Then create the instances
    - 6x the one below, varying the parameters



```
    docker run -d --name redis1 -p 7001:7000 --network redis-network redis:6.2.5-alpine redis-server 
--port 7000 --cluster-enabled yes --cluster-config-file nodes.conf --cluster-node-timeout 5000 --appe
ndonly yes
```

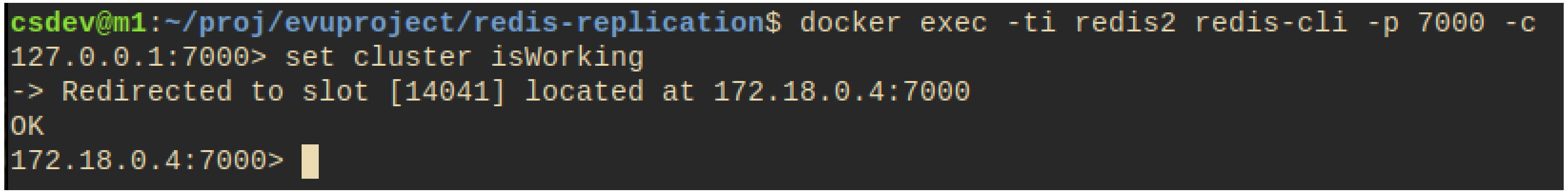
Dockerizing

- Initialize the Cluster
  - You need the IP addresses on the internal network
    - docker exec –ti redis1 sh
      - And issue ‘ifconfig’ to find that
    - Fortunately Docker assigns them consecutive numbers
- And then



```
    docker exec -ti redis1 redis-cli -p 7000 --cluster create 172.18.0.2:7000 172.18.0.3:7000 172.18.
0.4:7000 172.18.0.5:7000 172.18.0.6:7000  172.18.0.7:7000 --cluster-replicas 1
```

- Testing



```
csdev@m1:~/proj/evuproject/redis-replication$ docker exec -ti redis2 redis-cli -p 7000 -c
127.0.0.1:7000> set cluster isWorking
-> Redirected to slot [14041] located at 172.18.0.4:7000
OK
172.18.0.4:7000>
```

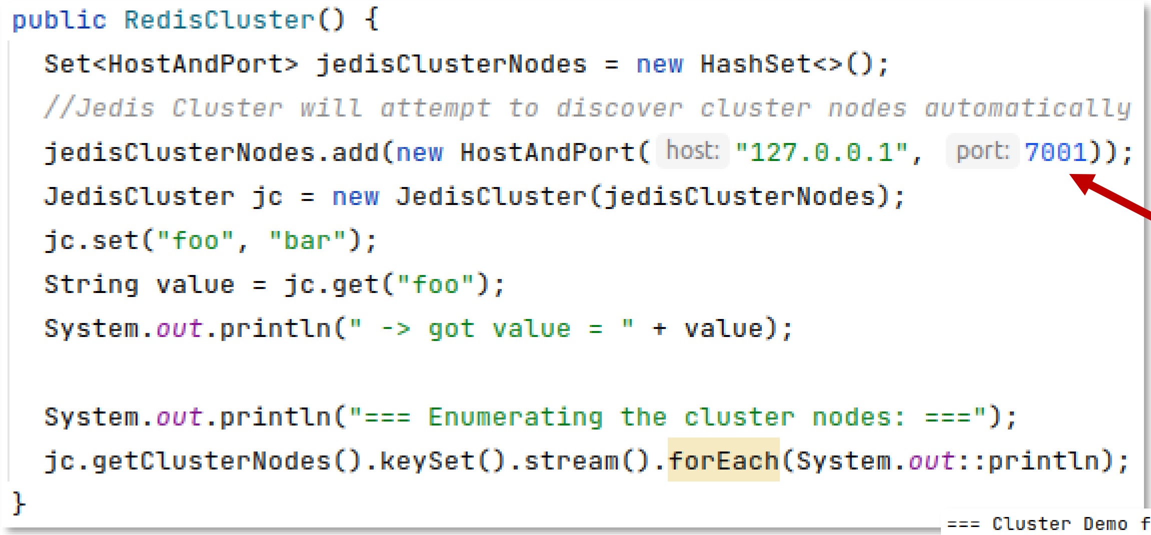
AARHUS UNIVERSITET

Java Side

AARHUS UNIVERSITET

Was…

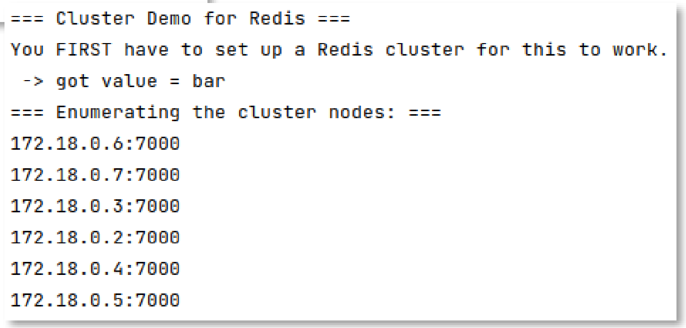
- … actually really easy, as Jedis supports it directly…



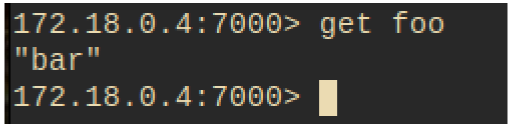
```java
public RedisCluster() {
  Set<HostAndPort> jedisClusterNodes = new HashSet<>();
  //Jedis Cluster will attempt to discover cluster nodes automatically
  jedisClusterNodes.add(new HostAndPort( host: "127.0.0.1",  port: 7001));
  JedisCluster jc = new JedisCluster(jedisClusterNodes);
  jc.set("foo", "bar");
  String value = jc.get("foo");
  System.out.println(" -> got value = " + value);

  System.out.println("=== Enumerating the cluster nodes: ===");
  jc.getClusterNodes().keySet().stream().forEach(System.out::println);
}
```

Reason for portmapping to localhost



```
=== Cluster Demo for Redis ===
You FIRST have to set up a Redis cluster for this to work.
 -> got value = bar
=== Enumerating the cluster nodes: ===
172.18.0.6:7000
172.18.0.7:7000
172.18.0.3:7000
172.18.0.2:7000
172.18.0.4:7000
172.18.0.5:7000
```



```
172.18.0.4:7000> get foo
"bar"
172.18.0.4:7000>
```