Distributed key-value store with data partitioning and replication

Course project, Distributed Systems 1, 2016

Overview

- A key-value store (data base). Every data item is identified by a unique key:

```
- update(key, value)

- get(key) -> value
```

Reliability and accessibility

- Replication — several nodes store the same item

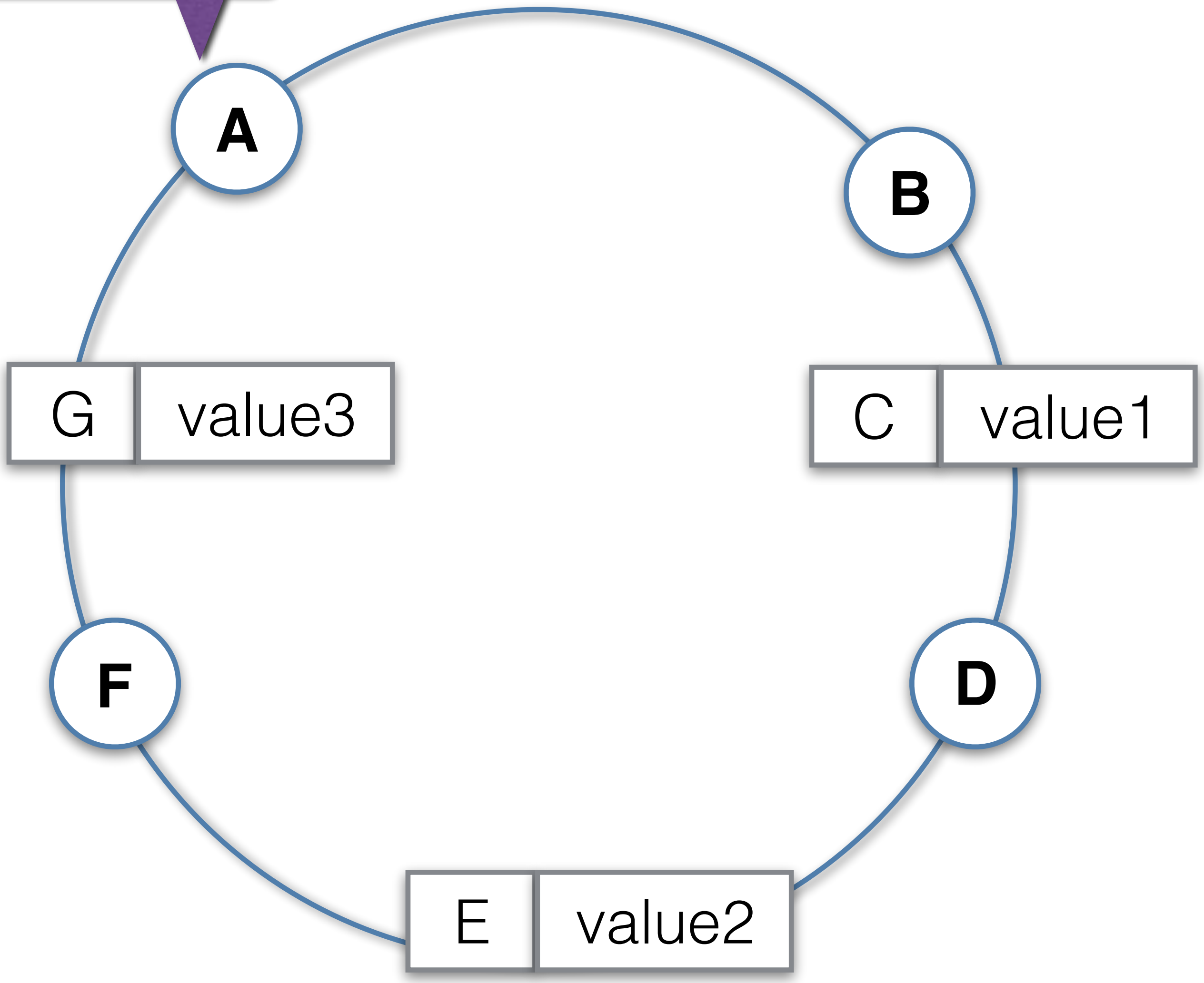
- Data partitioning — not all the nodes store all the items

Load balancing

One ring…

- DHT-based (distributed hash table), peer-to-peer system
- Inspired by Amazon Dynamo but **much** simpler
- Both the storage nodes and data items have associated keys that form a circular space (**ring**)
- For the keys we'll use integer numbers

Storage node with key A

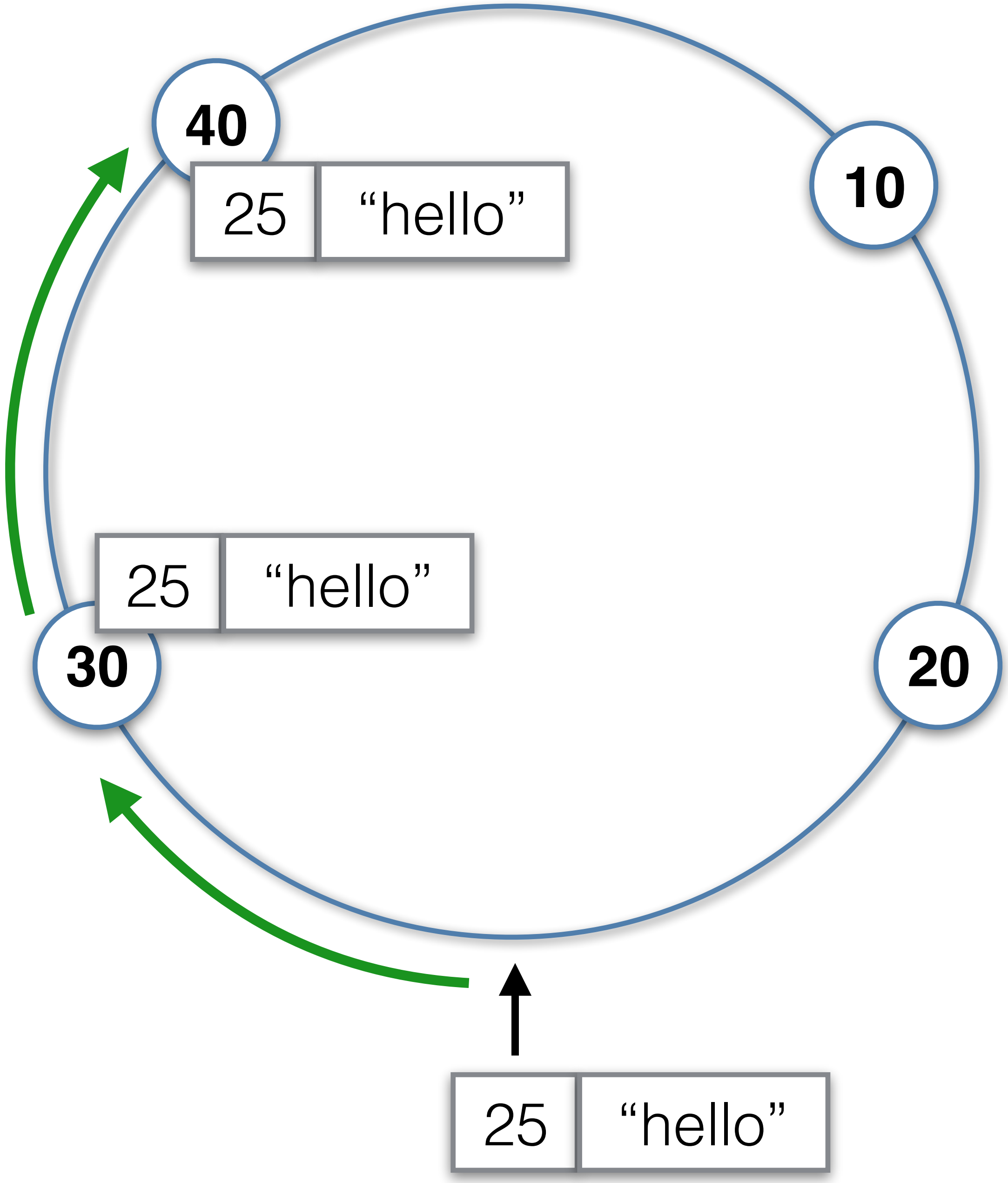


Data item with key E

… to rule them all

A data item with key K is stored by N nearest clockwise nodes

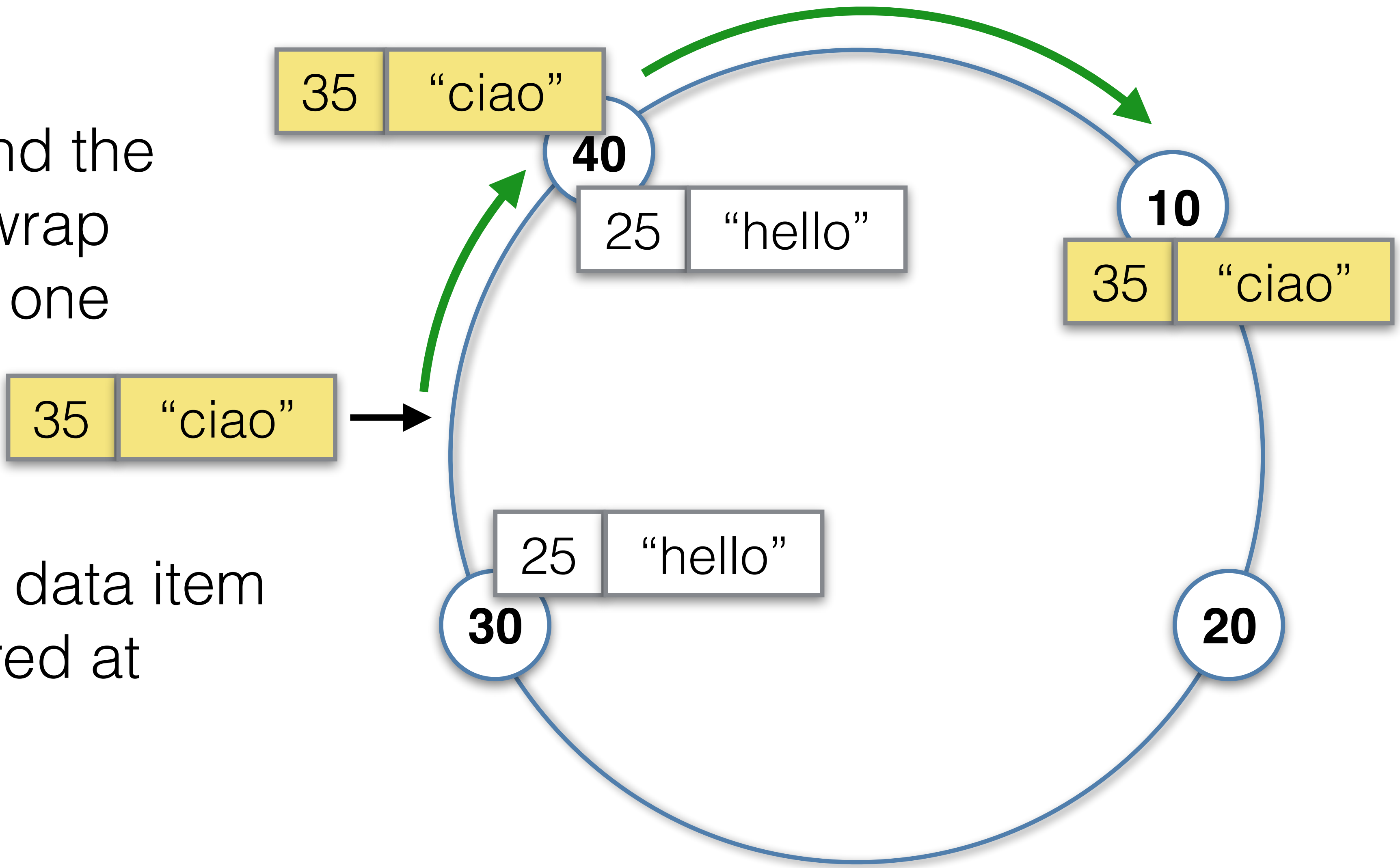
For example, if N=2, a data item with key **25** will be stored at nodes **30** and **40**



… to rule them all

If we need to go beyond the largest-key node, we wrap around to the smallest one

For example, if N=2, a data item with key **35** will be stored at nodes **40** and **10**

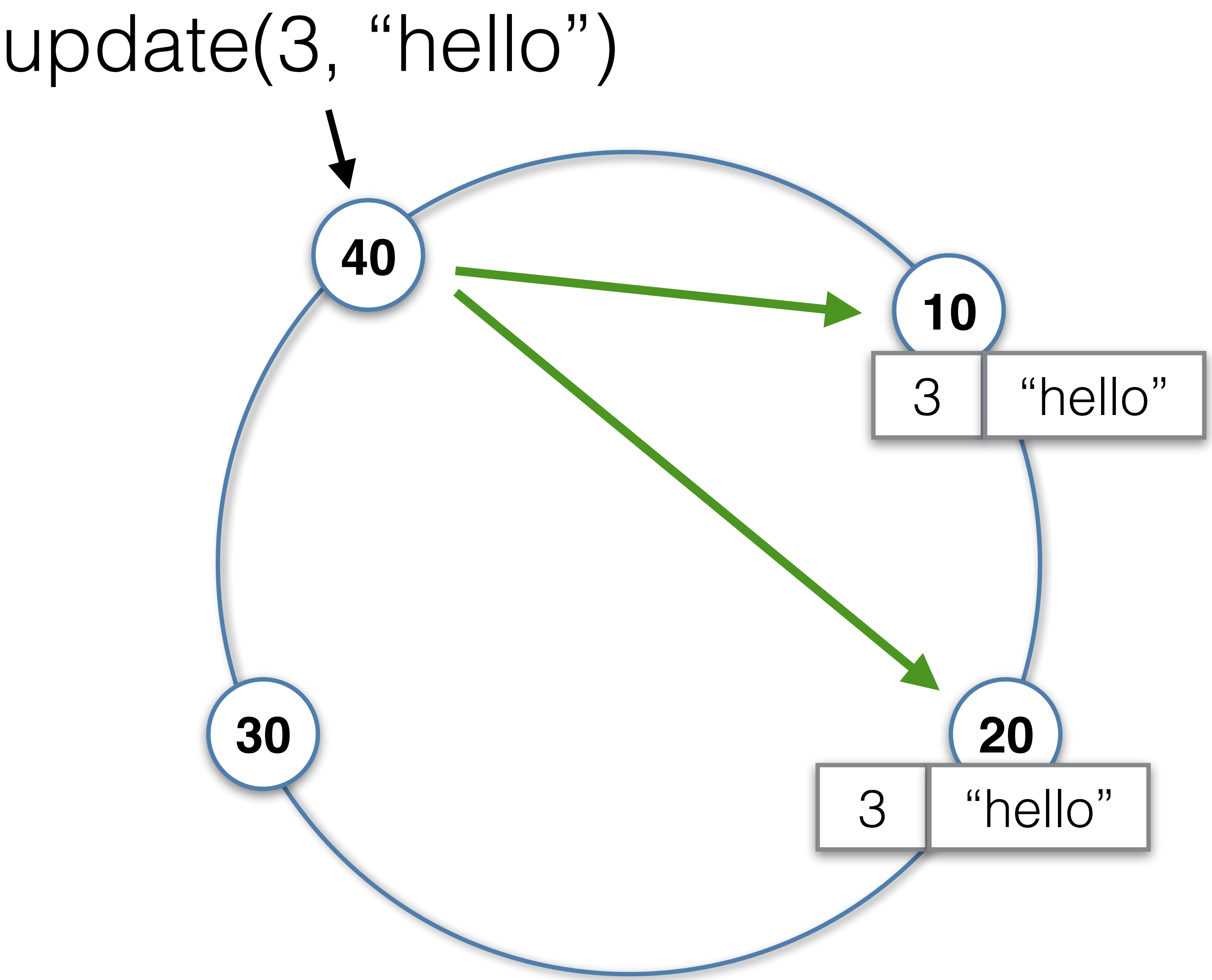


Request coordinator

Clients may contact *any* nodes to read/write data with *any* key

The node the client sends its request to is called a *request coordinator*

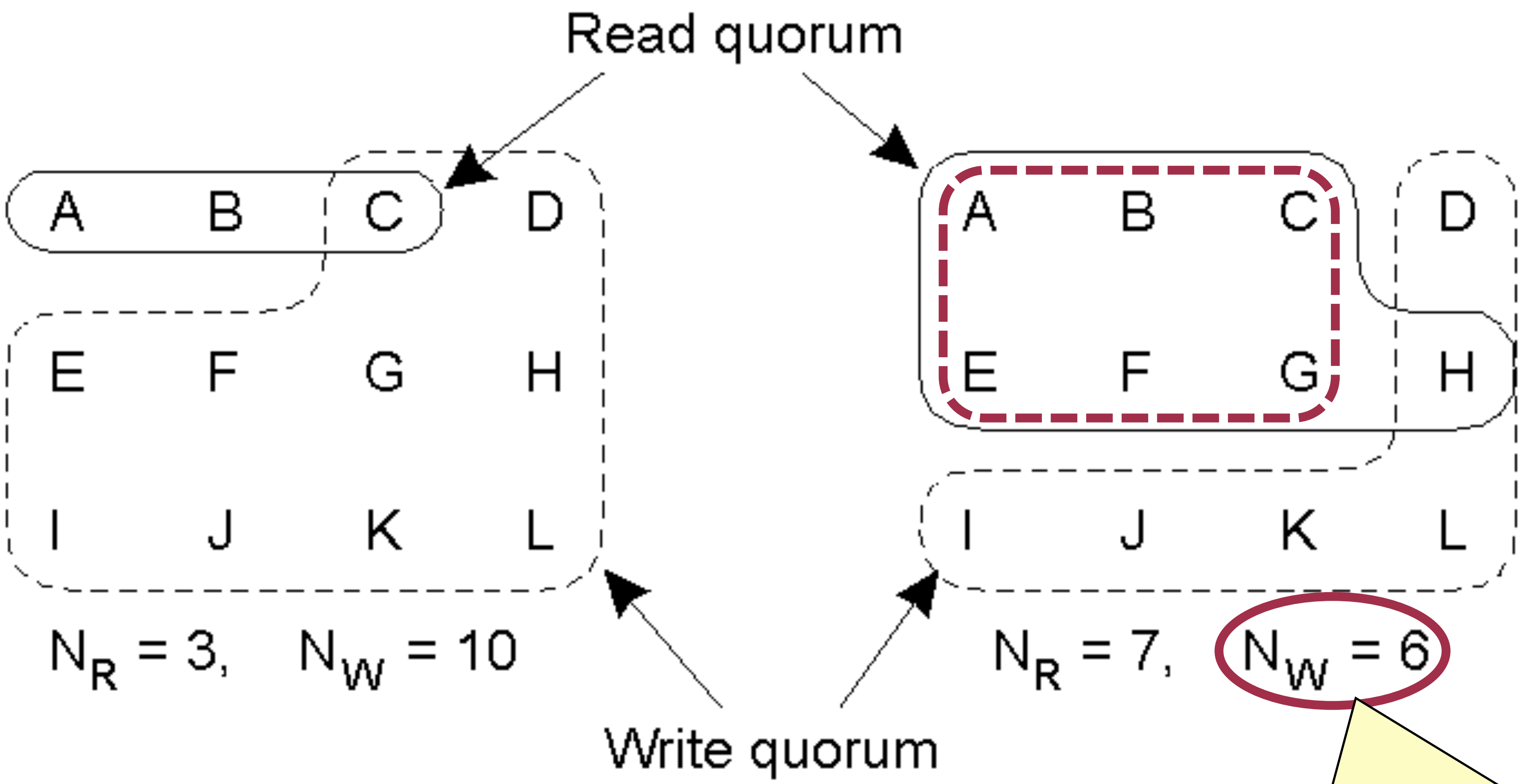
The nodes know all their peers and, therefore, can compute who is responsible for the key and pass the item to them



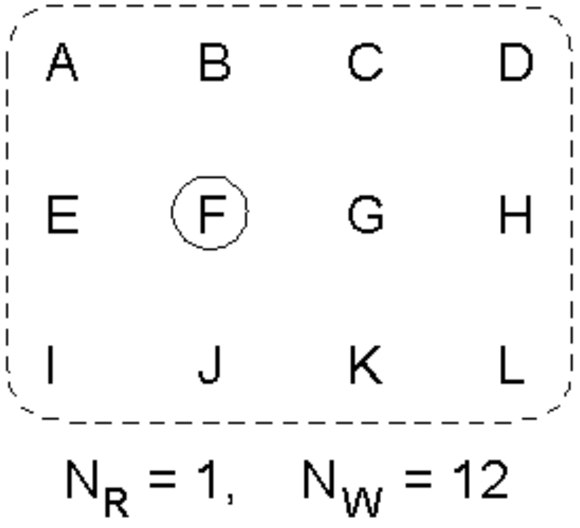
Replication

- Same data item is stored by N nodes
- When reading an item, the coordinator requests data from all N nodes, but answers to the client as soon as **R<N** of them reply
- When writing, the coordinator tries to contact all N nodes, but completes the write even if only **W<N** of them reply
- R — read quorum, W — write quorum. **R+W>N**
- We also need to assign a version to every data item

Replicated-Write: Quorum-Based



Different sets could have the same version but different data



Read one, write all (ROWA)

Read

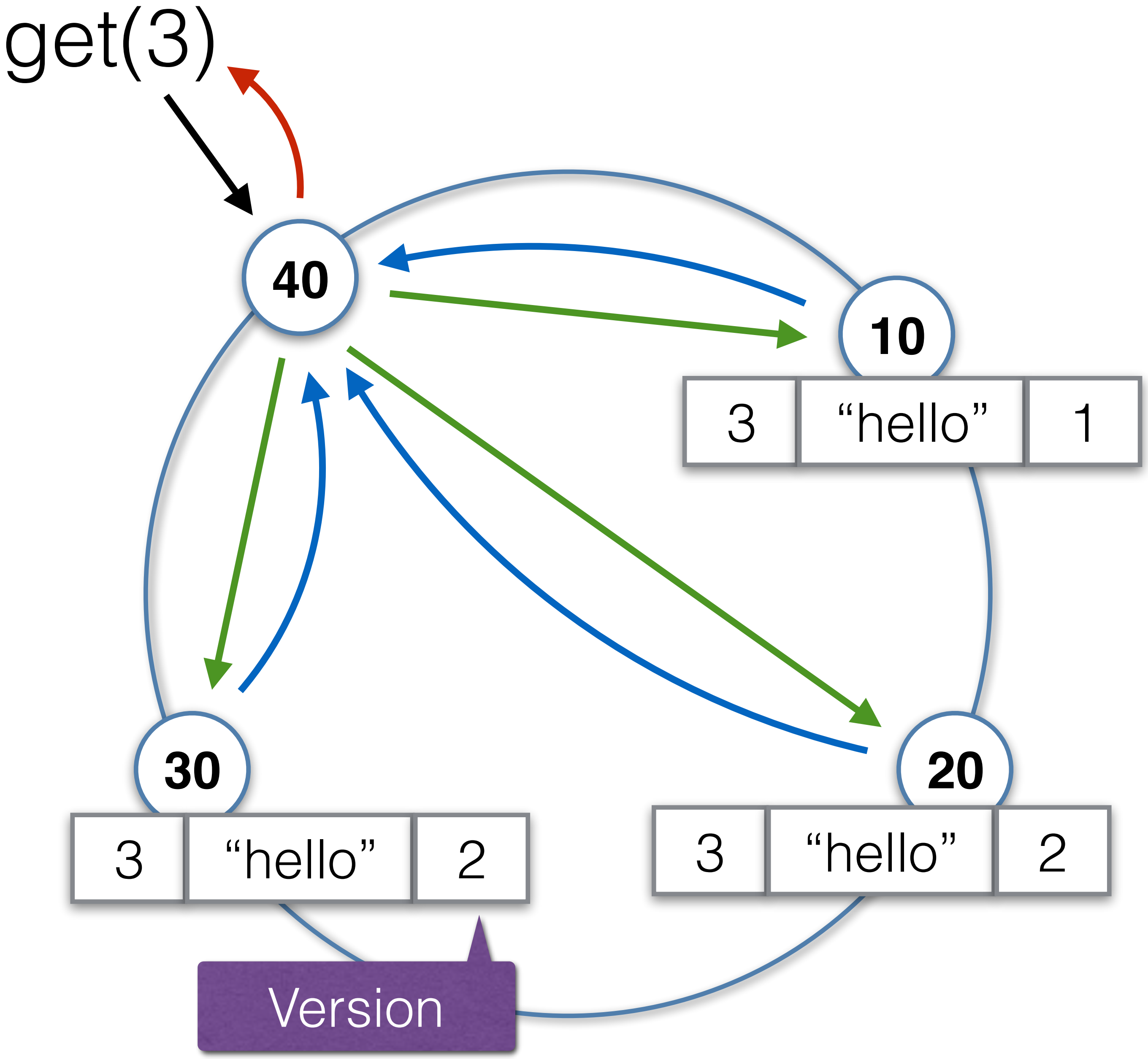
Let N=3, R=2

1. A client contacts node 40 to read the item with key 3

2. Node 40 requests the item from replicas 10, 20 and 30

3. The nodes reply

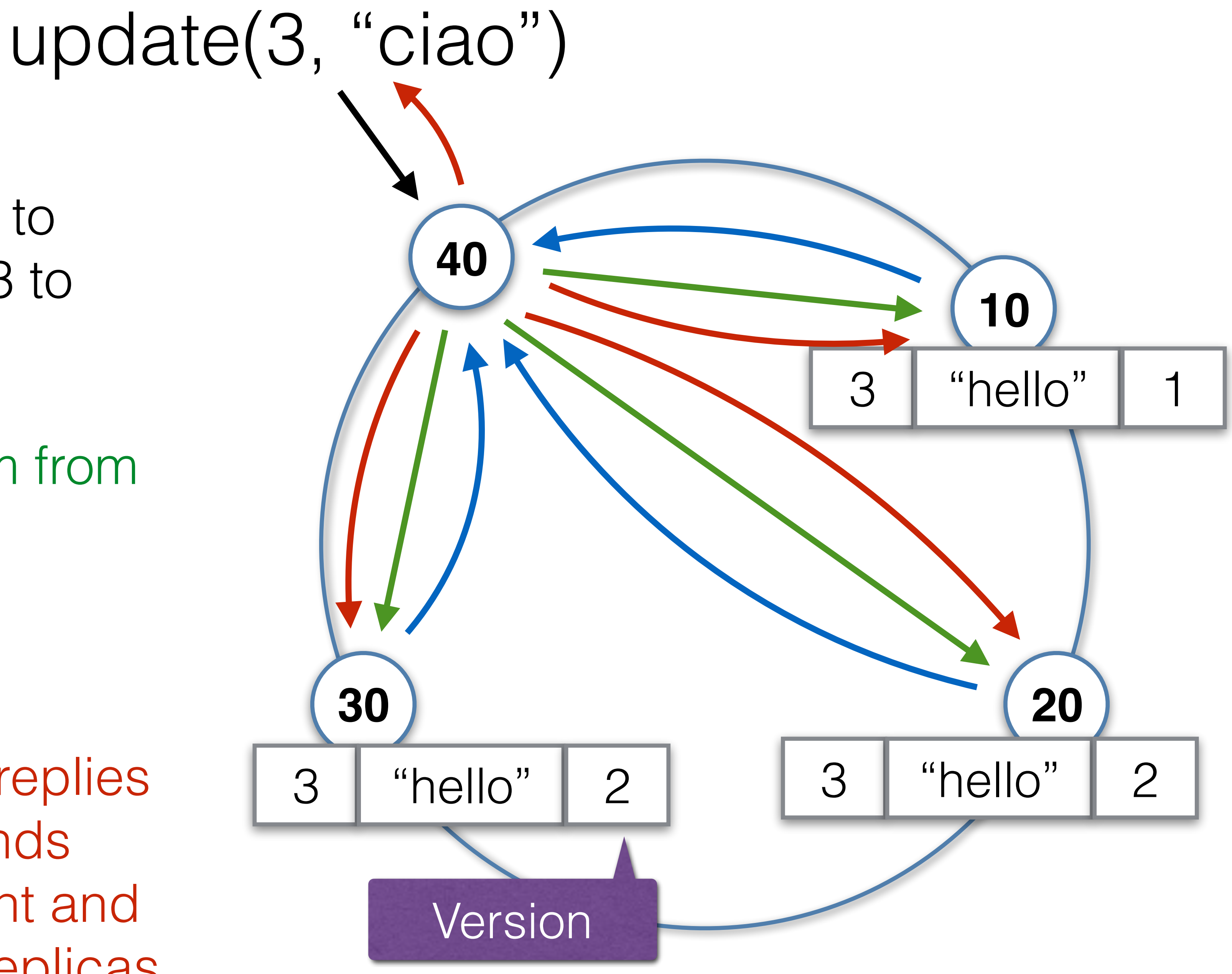
4. As soon as R=2 replies are received, node 40 sends the item with the highest version back to the client



Write

Let N=3, W=2

1. A client contacts node 40 to update the item with key 3 to “ciao”
2. Node 40 requests the item from replicas 10, 20 and 30
3. The nodes reply
4. As soon as Q=max(R,W) replies are received, node 40 sends success status to the client and sends [3, “ciao”, 3] to N replicas



Write

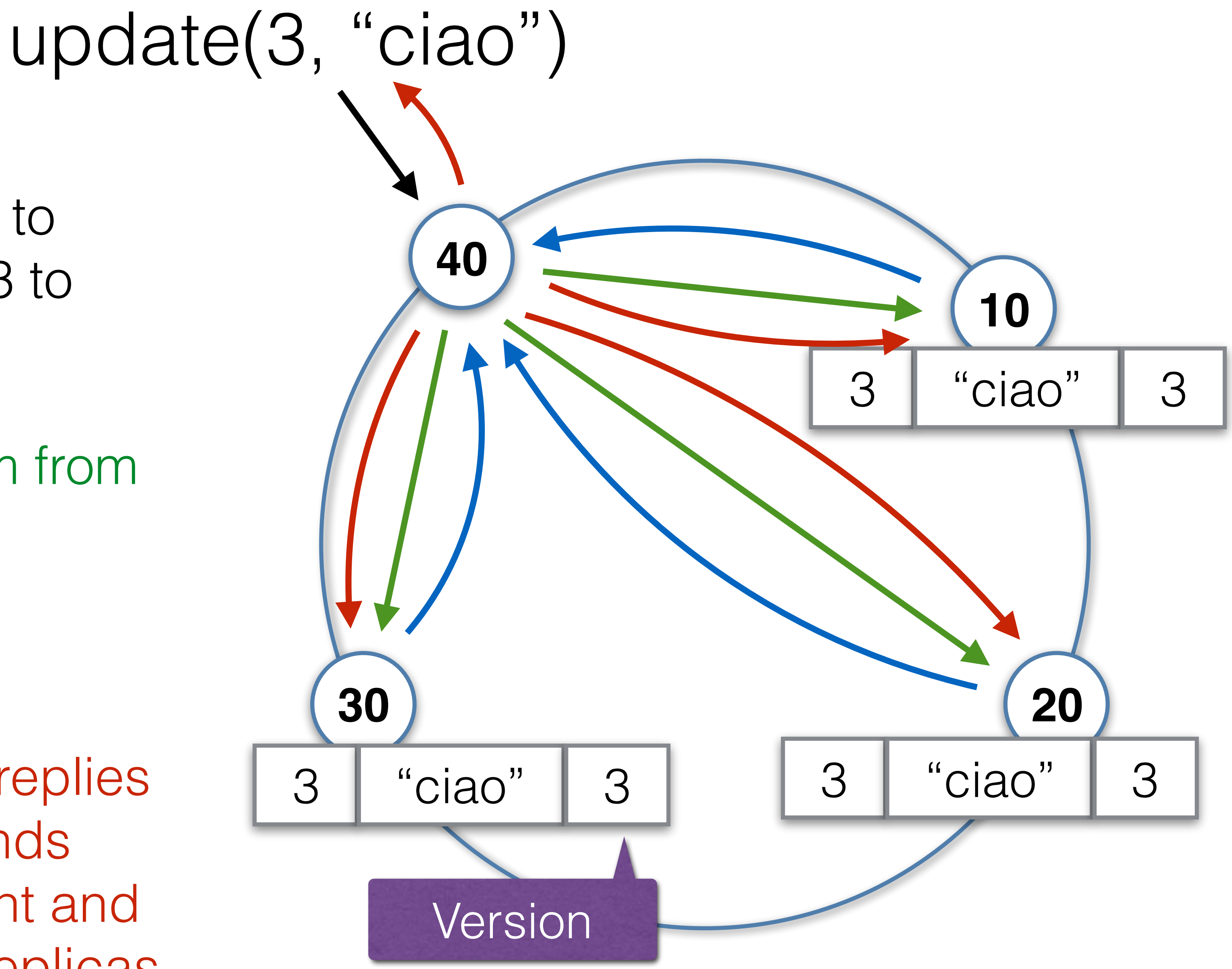
Let N=3, W=2

1. A client contacts node 40 to update the item with key 3 to “ciao”

2. Node 40 requests the item from replicas 10, 20 and 30

3. The nodes reply

4. As soon as Q=max(R,W) replies are received, node 40 sends success status to the client and sends [3, “ciao”, 3] to N replicas



No quorum?

When reading or writing, if if the quorum is not reached after a timeout T, the coordinator replies with a failure to the client

Repartitioning

When a node joins or leaves, the system should move data accordingly

Crashes do not count as leaves! No need to detect crashes and move data if some node cannot be reached

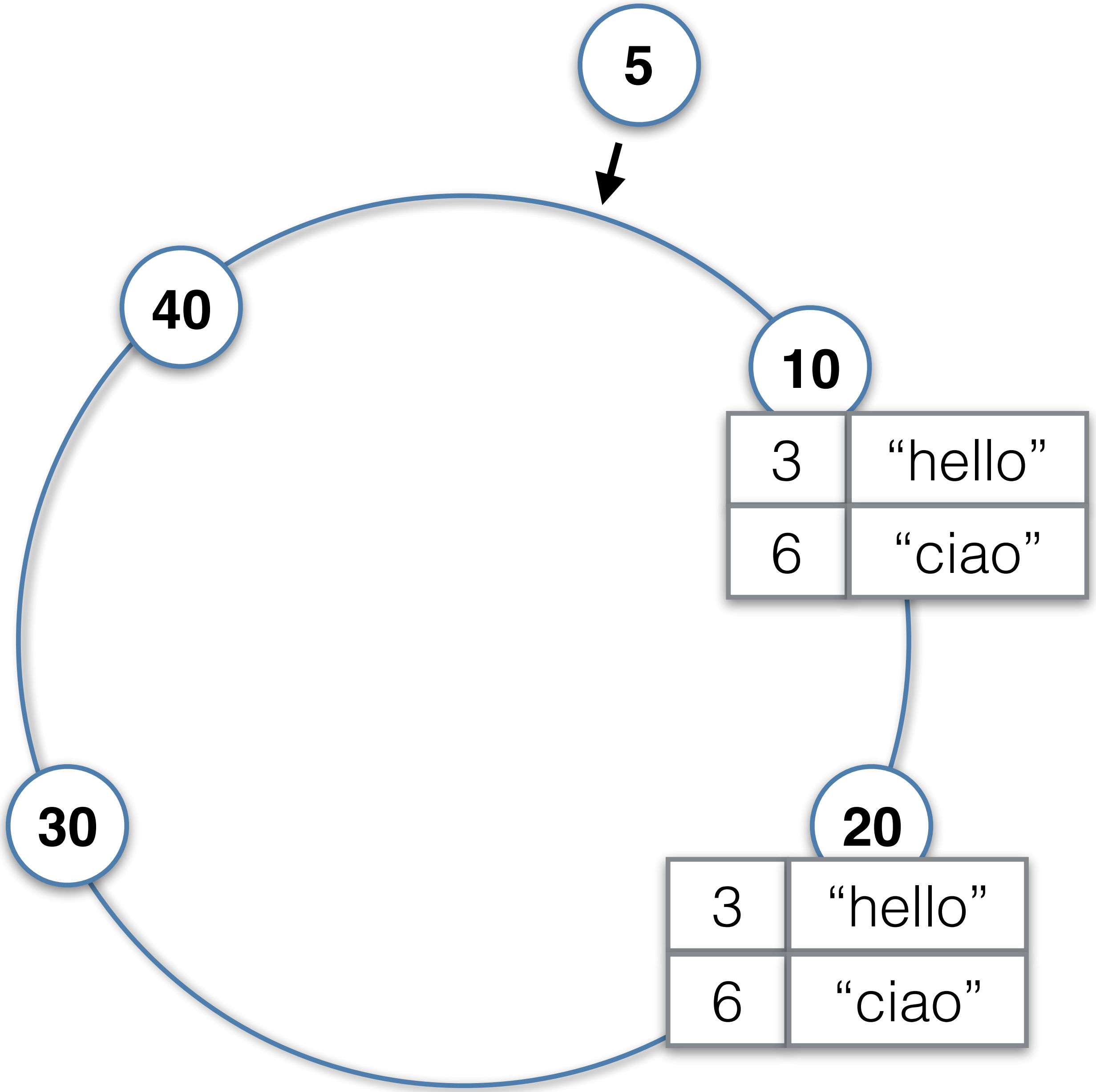
Joining

1. Contact one of the nodes (specified as the command line argument) to request the list of nodes

2. Ask the next clockwise node to send back all the data the new node is responsible for

3. Announce its presence to every node in the system

4. The other nodes should remove data they are not responsible for any more



3	“hello”
6	“ciao”

3	“hello”
6	“ciao”

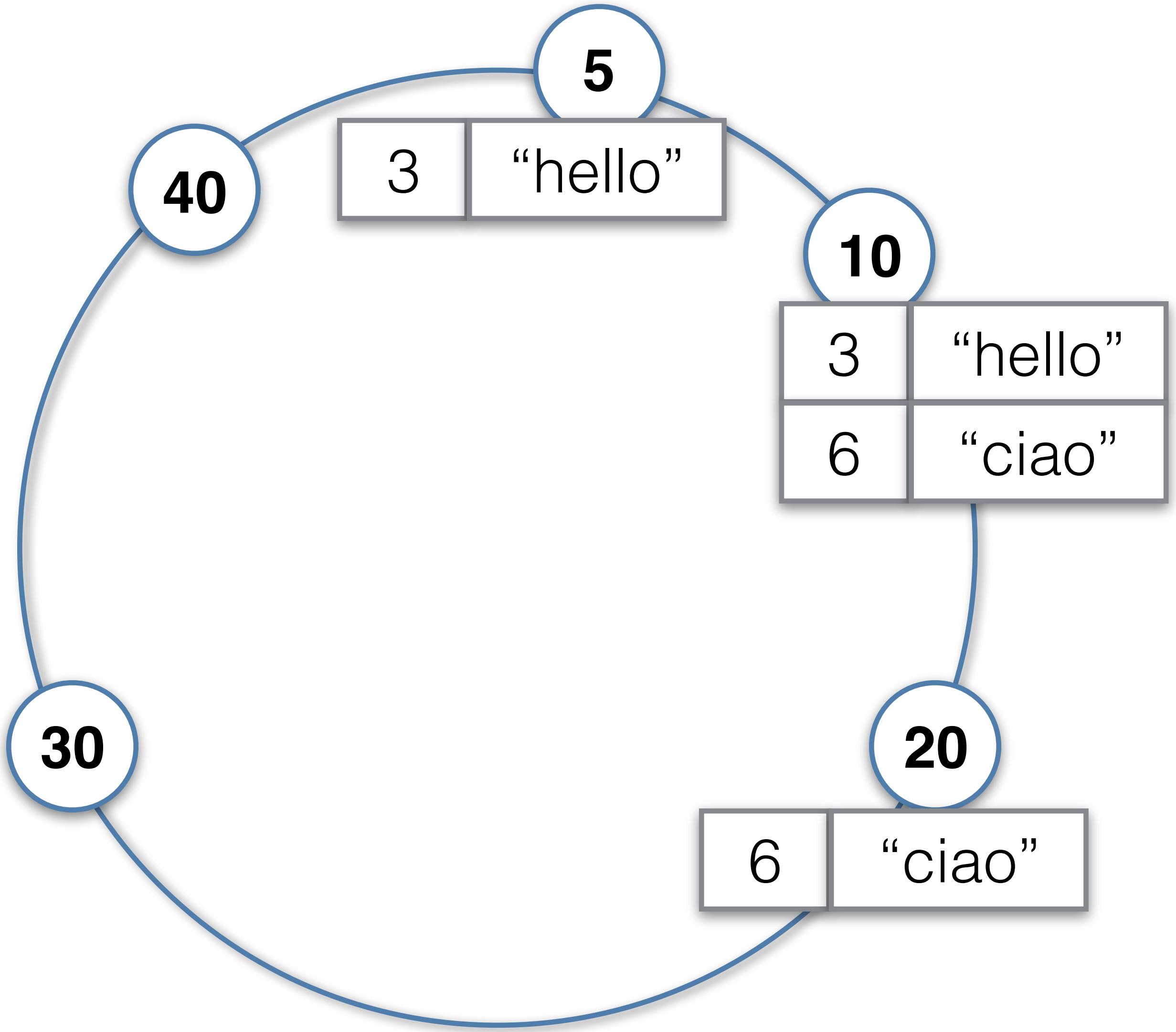
Joining

1. Contact one of the nodes (specified as the command line argument) to request the list of nodes

2. Ask the next clockwise node to send back all the data the new node is responsible for

3. Announce its presence to every node in the system

4. The other nodes should remove data they are not responsible for any more

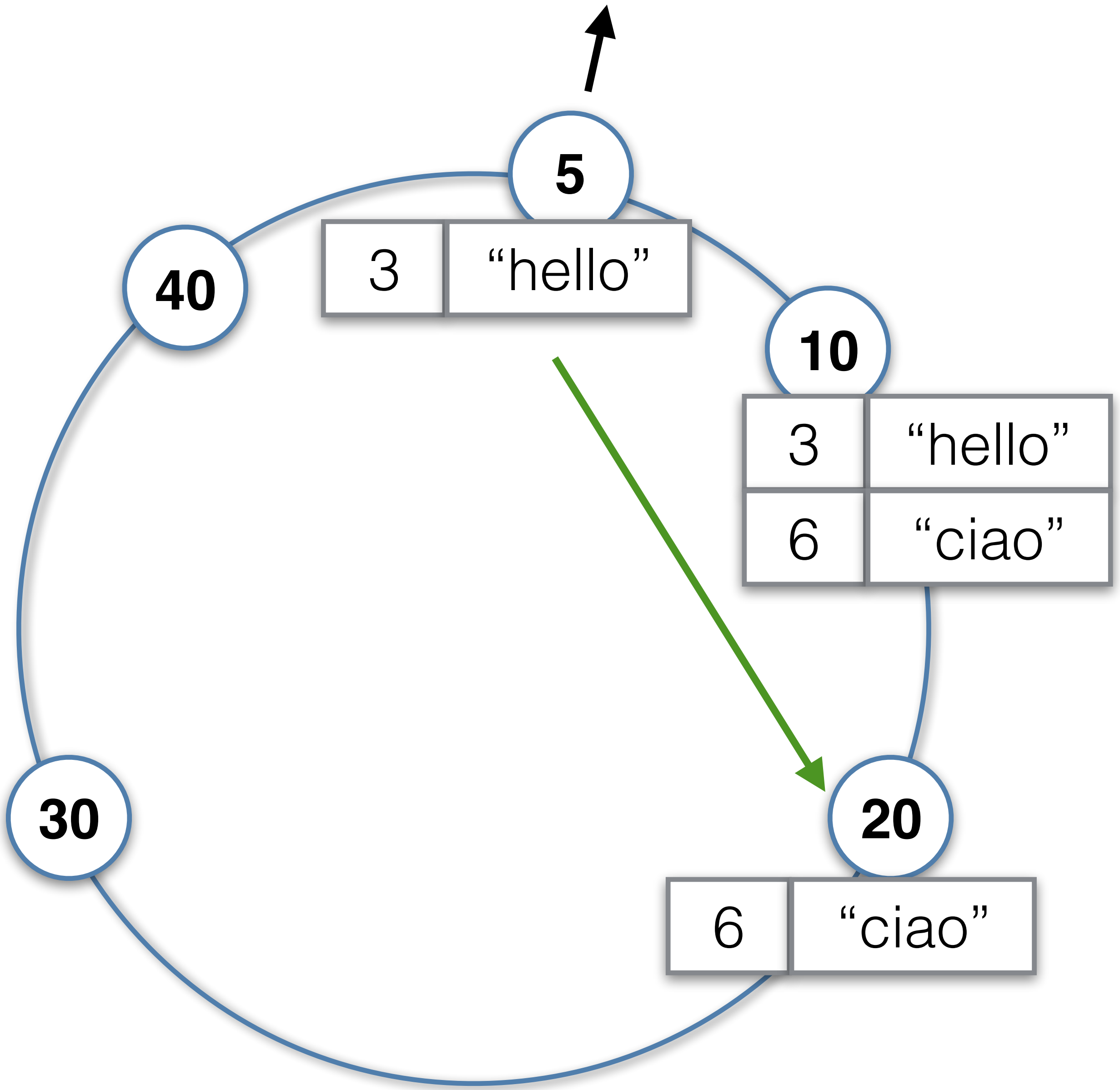


Leaving

1. A client may request a node to leave

2. The node announces that it is leaving to all the other nodes

3. The node passes its data to N clockwise peers (if needed)

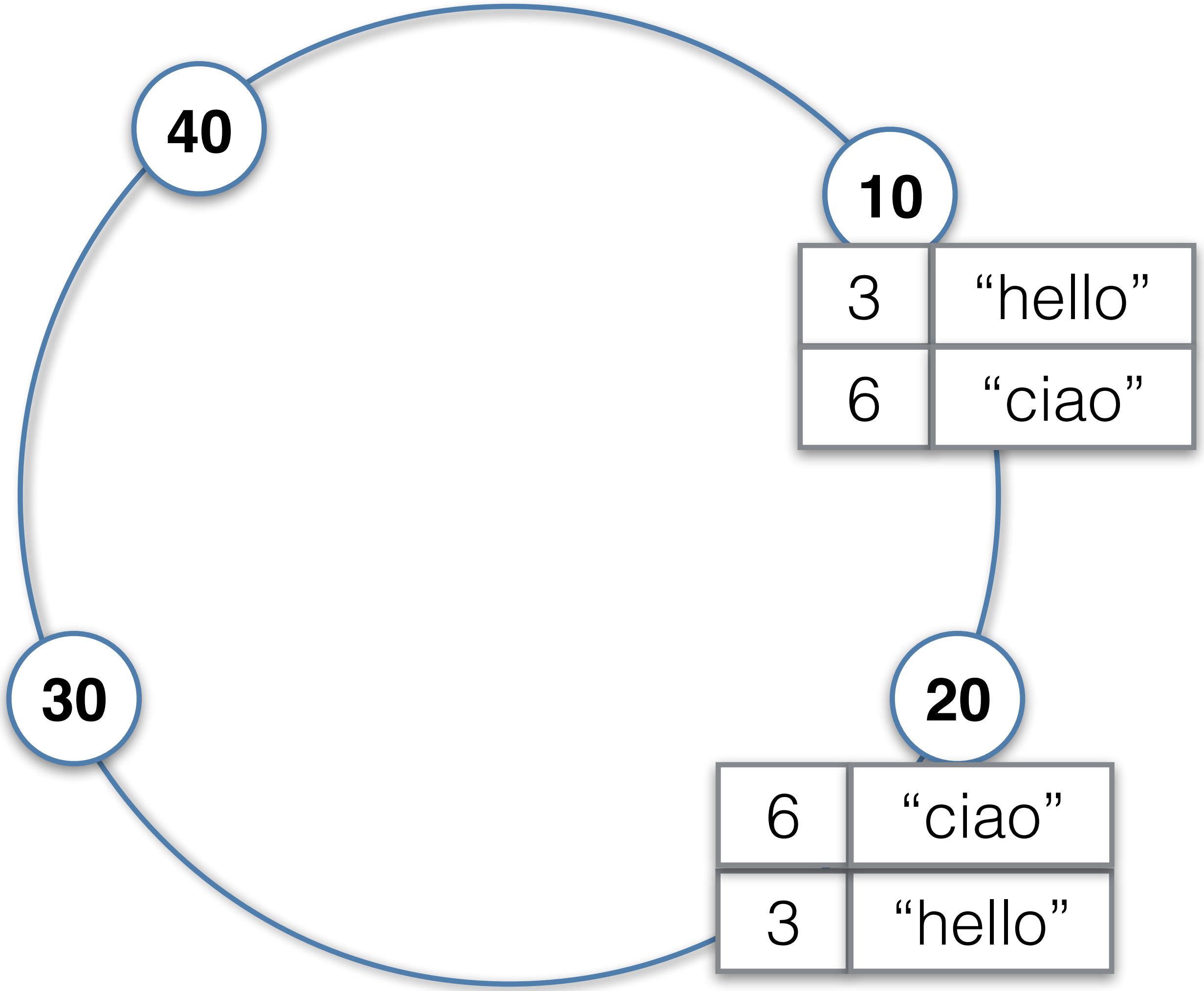


Leaving

1. A client may request a node to leave

2. The node announces that it is leaving to all the other nodes

3. The node passes its data to N clockwise peers (if needed)



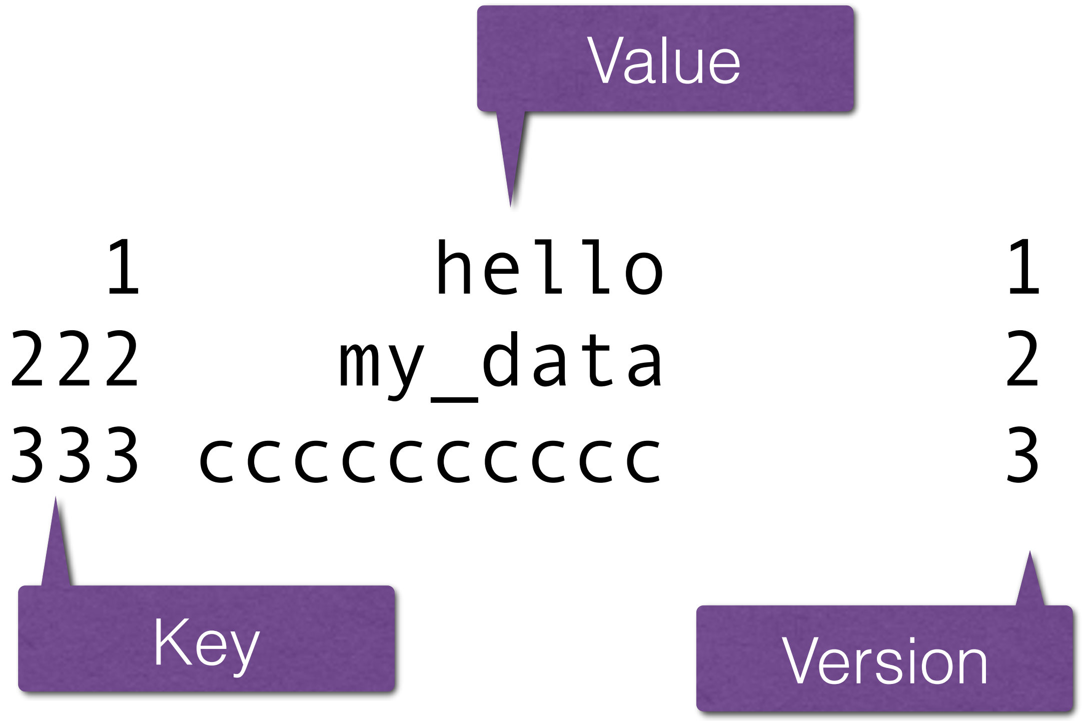
3	“hello”
6	“ciao”

6	“ciao”
3	“hello”

Local storage

- The nodes should keep their data items in a file

- Rewrite it when an item is updated



```
  1      hello       1
222    my_data       2
333 cccccccccc       3
```

Recovery

1. Request list of nodes from a specified node

2. Load the items from the local file

3. Forget the items the node is not responsible for (if a new node joined while this one was down)

User interface

- The node and the client are separate applications
- Command line arguments — the only interface needed for both the node and the client

Node

```
$ java Node join remote_ip remote_port
$ java Node recover remote_ip remote_port
```

Client

```
$ java Client ip port read key
$ java Client ip port write key value
$ java Client ip port leave
```

Assumptions

- A node serves one client at a time
- No parallel user requests affecting the same key
- Nodes join, leave or crash one at a time when the network is idle
- Replication parameters are specified at compile time
- Nodes should be able to talk over the network among themselves and with clients
- More things in the description document

Grades

- The whole feature set is worth **6 points**
- A reduced implementation without replication, versions and support for recovery is worth **3 points**
- Work in pairs! (still grades are individual)

Networking with Akka

Configuration file

application.conf

```
akka {
  actor {
    provider = remote
  }
  remote {
    enabled-transports =
                     ["akka.remote.netty.tcp"]
    netty.tcp {
      hostname = "127.0.0.1"
      port = 10001
    }
  }
}
```

IP address to bind to

Port to listen to

Additional parameters

You may put arbitrary parameters to the config file. In our case it is convenient to put the node ID (key) there:

application.conf

```
akka {
 …
}
nodeapp {
  id = 1
}
```

Reading the config

In the main() function:

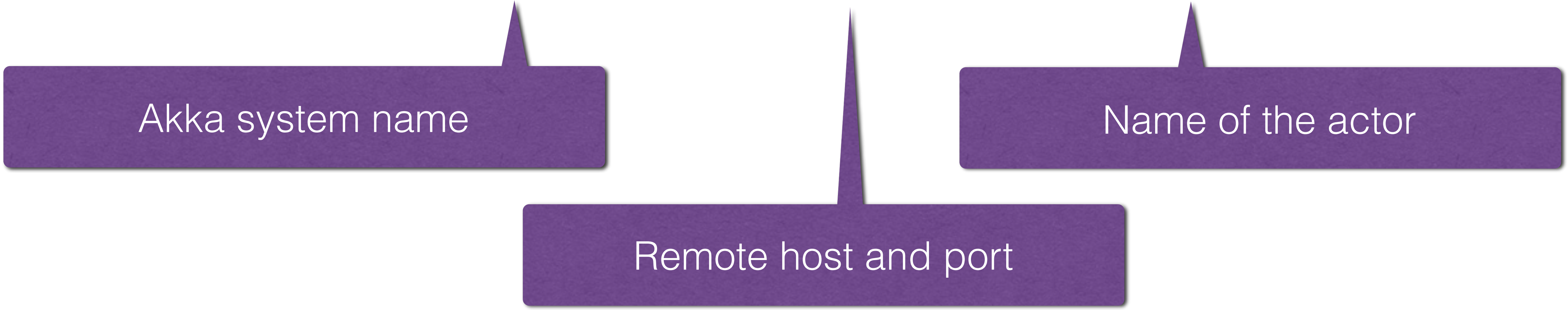
Read our custom parameter

```
Config config =
ConfigFactory.load("application");

int myId = config.getInt(“nodeapp.id");

ActorSystem system =
  ActorSystem.create("mysystem", config);
```

Feed other parameters to Akka

Accessing remote actors

- To access the remote actors it is needed to either know the reference or use the “remote path”:
- “akka.tcp://mysystem@*host*:*port*/user/node”



Akka system name

Name of the actor

Remote host and port

```
getContext().actorSelection(path).tell(
                 message, getSelf() );
```

Accessing remote actors with a reference

- Once you’ve learned the ActorRef of a remote actor it is convenient to use it exactly the way we did with local actors

- To learn it you should either receive a message from the remote actor and use getSender() or receive the reference from someone else

Example

- On Moodle!
- Contains the program and configurations for three nodes
  - Starts the Node actor. If remote ip and port are specified, requests the list of currently joined nodes
  - Joins the group itself

Starting multiple nodes on a single computer

- For every node you will need a separate configuration file
- You may put them to separate directories (as in the example)
- To launch:

```
$ cd 1
$ java -cp $AKKA_CLASSPATH:.:.. NodeApp
```