Advanced usage and Distribution Redis

http://redis.io

Expiry

- EXPIRE command, an existing key, and a time to live (in seconds).
- Set ice to expire after 10 seconds
- SET ice "I'm melting..."
- EXPIRE ice 10

- EXISTS ice
- Wait 10 secs
- EXISTS ice

- Shortcut command:
- SETEX ice 10 "I'm melting...“
- Check time to live:
- TTL ice
- Make it persistent
- PERSIST ice

Database Namespaces

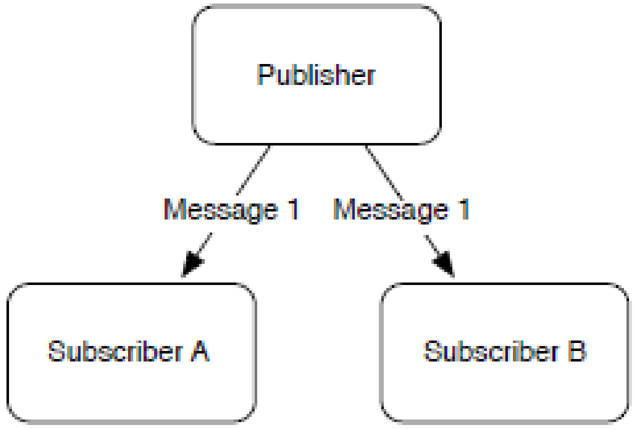
- In Redis a namespace is called a *database* and is keyed by number
- So far we've always interacted with the default database (namespace 0)
- SET greeting hello
- GET greeting

- SELECT 1
- GET greeting
- SET greeting hola

- SELECT 0
- GET greeting
- Since all databases are running in the same server instance, you can shuffle keys around using MOVE
- MOVE greeting 2
- SELECT 2
- GET greeting

Publish-subscribe

- Multiple subscribers



- Start two clients:
- SUBSCRIBE comments

- Start publisher:
- PUBLISH comment “Thanksgiving is next week!”

- UNSUBSCRIBE disconnects client – in redis-cli console press Ctrl+C to break the connection

Redis configuration

- Server info: INFO
- daemonize no – starts in the foreground
- port 6379
- loglevel verbose | notice | warning
- logfile stdout /*filename req if daemonize mode*/
- database 16

Redis configuration (cont.)

- save 300 1 (snapshotting, save every 5 mins if any keys change at all)
- appendonly yes (keeps record of all write commands)
  - appendfsync always | everysec | no

Security

- Redis is not natively built to be a fully secure server
- Plaintext password not really safe
- Use SSH security
- Redis allows you hide or suppress commands
  - Include rename-command in the conf file:
  - rename-command FLUSHALL c123456789
  - rename-command FLUSHALL “”

Master-slave replication

- First make a copy of the redis.conf file
  - cp redis.conf redis-s1.conf
  - Change port and slaveof
    - port 6380
    - Slaveof 127.0.0.1 6379
  - Start both servers
    - Redis-server redis-s1.conf
  - Add to the server
    - SADD meetings “Initial group meeting” “ECE potluck”
  - Query in the slave
    - SMEMBERS meetings

Redis cluster

- Many Redis clients provide an interface for building a simple ad hoc distributed Redis cluster.
- Unlike the master-slave setup, both of servers take the master (default) configuration.

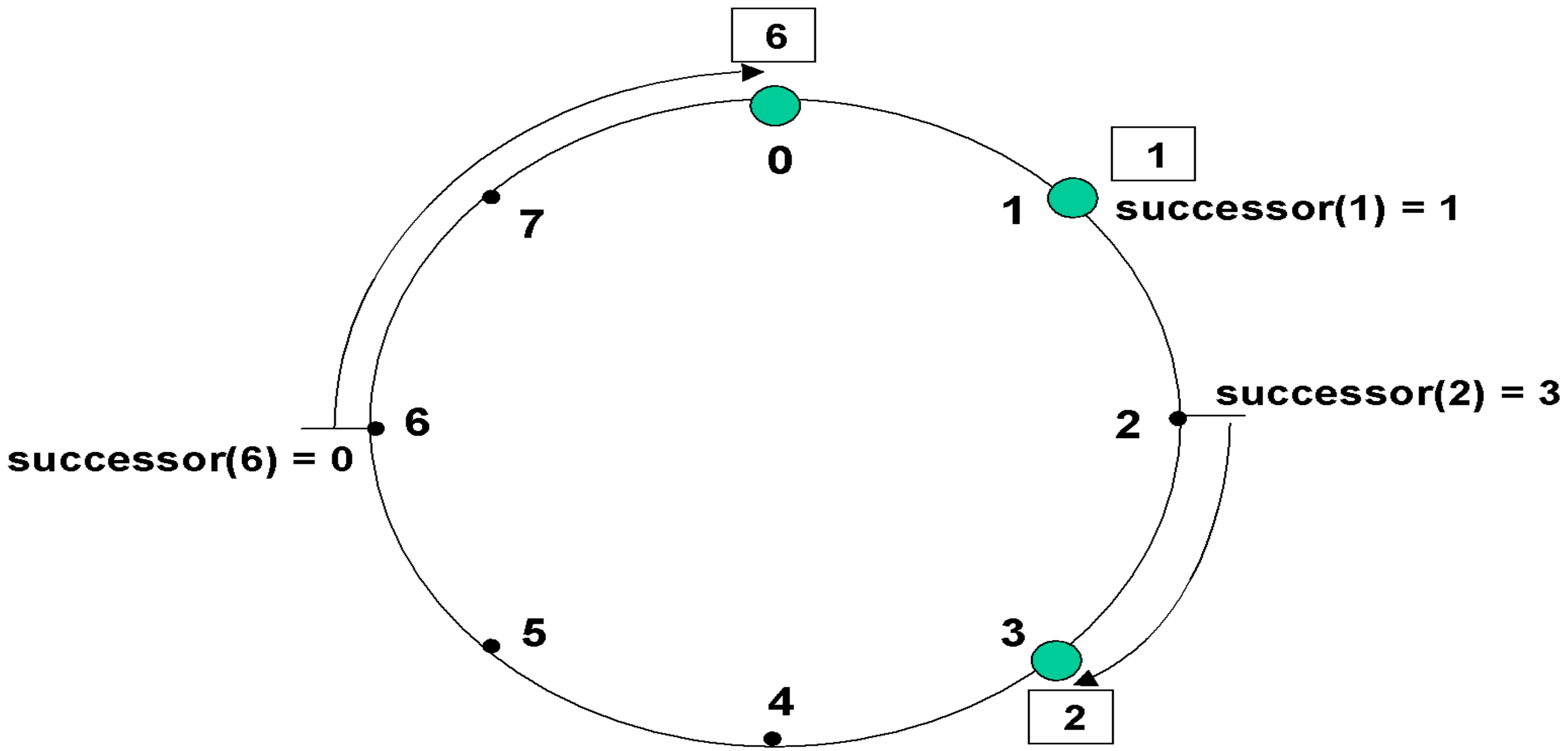
```
redis = Redis::Distributed.new([
"redis://localhost:6379/",
"redis://localhost:6380/” ])
```

- Consistent hashing is used to manage the cluster

Consistent Hashing System

- Given *k*, every node can locate n_k
- Hash every node's IP address
  - map these values on a circle
- Given a key *k*, hash k
  - k is assigned to closest node on circle, moving clockwise.

Consistent Hashing System



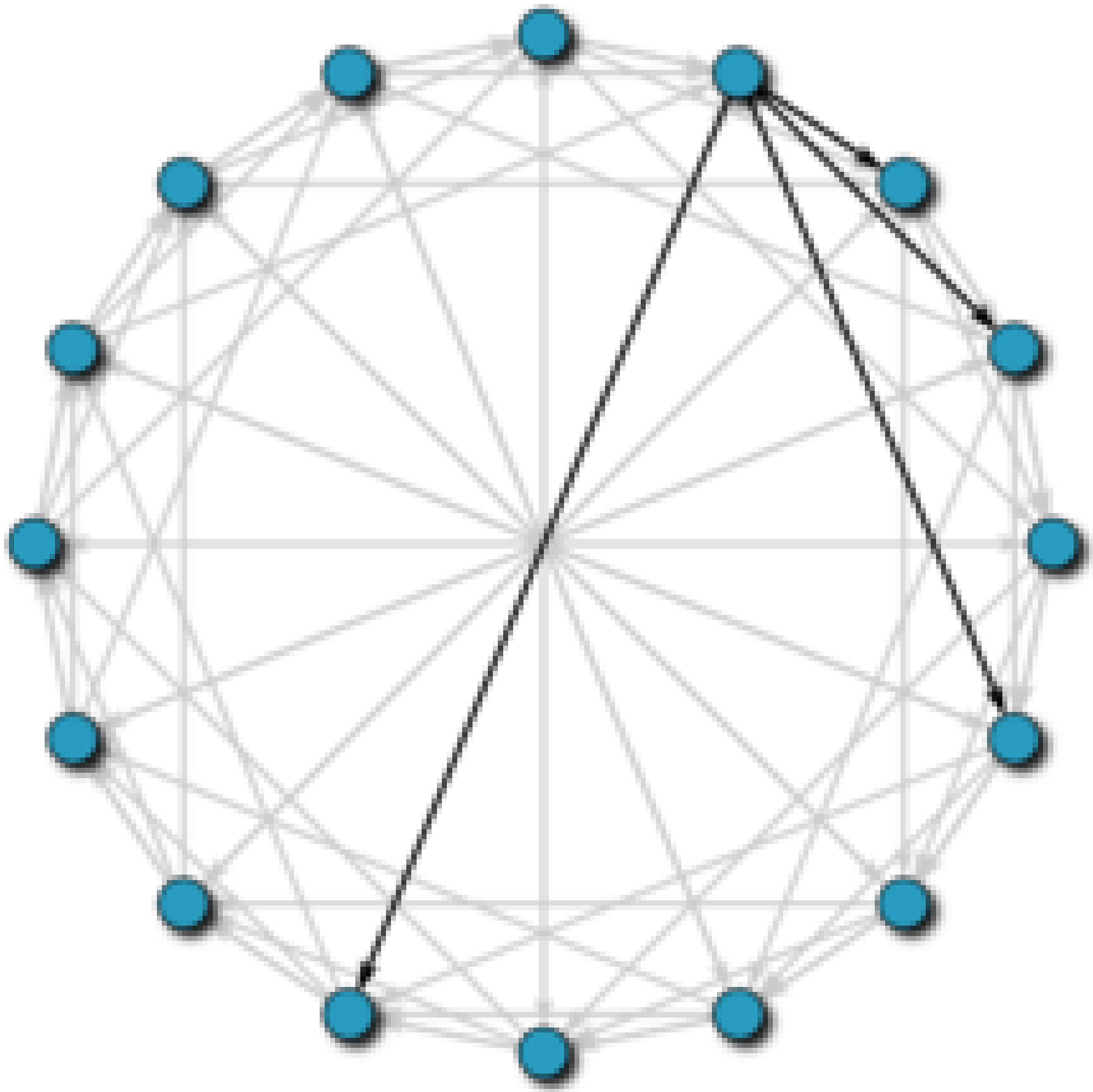
Consistent Hashing System

- Pros:
  - Load Balanced
  - Dynamic membership
    - when Nth node joins network, only O(1/N) keys are moved to rebalance
- Con:
  - Every node must know about every other node
  - O(N) memory, O(1) communication
    - Not scalable in number of nodes

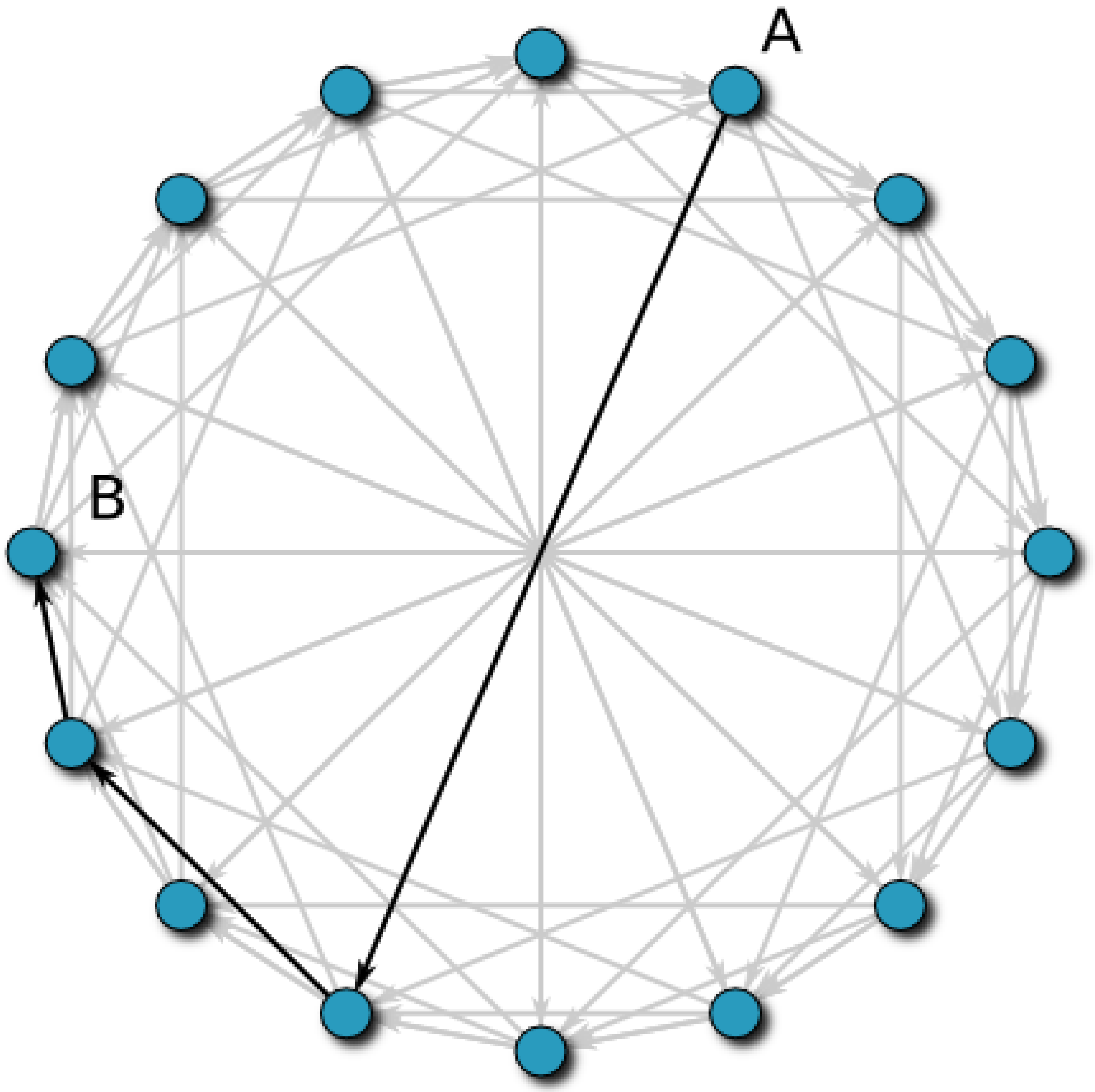
Scaling Consistent Hashing

- Approach 0:
  - Each node keeps track of only their successor
  - Resolution of hash function done through routing
  - O(1) memory
  - O(N) communication
- Approach 1:
  - Each node keeps track of O(log N) successors in a “finger table”
  - O(log N) memory
  - O(log N) communication

Finger Table Pointers



Routing with Finger Tables



Incremental Scalability

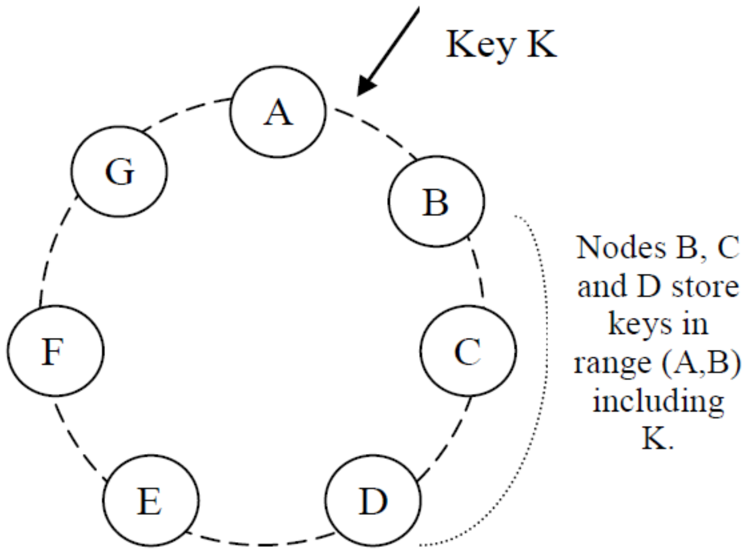
- Utilize “virtual nodes” along ring
    - Many virtual nodes per physical node
    - larger machines can hold more virtual nodes
    - Heterogeneous hardware is properly load balanced

Replication

Each data item is replicated at N hosts.

preference list: The list of nodes that is responsible for storing a particular key.

Some fine-tuning to account for virtual nodes



Preference Lists

- List of nodes responsible for storing a particular key.
- Due to failures, preference list contains more than N nodes.
- Due to virtual nodes, preference list skips positions to ensure distinct physical nodes.

Replication: Sloppy Quorum

- **Quorum System**: R + W > N, W > N/2
  - R, W, N are tunable

- Each node maintains a "preference list" of replicas for its own data
- Replicas are made on first N *healthy* nodes from preference list
  - require R nodes to respond for get()
  - require W nodes to respond for put()

Data Versioning

- A put() call may return to its caller before the update has been applied at all the replicas
- A get() call may return many versions of the same object.
- Challenge: an object may have distinct versions
- Solution: use vector clocks in order to capture causality between different versions of same object.

Vector Clock

- A vector clock is a list of (node, counter) pairs.
- Every version of every object is associated with one vector clock.
- If the all counters on the first object’s clock are less-than-or-equal to all of the counters in the second clock, then the first is an ancestor of the second and can be forgotten.
- Application reconciles divergent versions and collapses into a single new version.

High Availability for Writes

- Clients write to first node they find
  - Vector clocks timestamp writes
  - Different versions of key's value live on different nodes

- Conflicts are resolved during reads
  - Like git: "automerge conflict" is handled by end application

Using Redis from your program

- Java Clients for Redis
  - **Lettuce** or **Jedis**
    - https://redislabs.com/lp/redis-java/
- Python Client for Redis
  - **Redis-py**
    - https://redislabs.com/lp/python-redis/

For today…

- Install your favorite programming language driver and connect to the Redis server.
- Insert and increment a value within a transaction.
- Submit the file with your code to ICON.