Dive into Redis

A Quick Look at Redis

Anıl Can Aydın

December 2, 2017

Zetaops

Table of contents

Introduction

About

- Izmir Institute of Technology - Computer Engineering
  - B.Sc. 2010 - 2016
  - M.Sc 2016 - Ongoing
    - **Interests:** Biometric Recognition, Image Processing, Artificial Intelligence, Machine Learning, Neural Networks
- Sava Consultancy, Izmir.
  - Android Developer 2015 - 2016, Part-time
- Zetaops, Izmir.
  - Software Developer 2017 - Ongoing
    - **Interests:** Problem Solving, High Availability, Scalability, RESTful API Design, Data Modeling, Caching, Microservices
    - **Stack:** Python, RabbitMQ, Redis, Riak
- Trying to:
  - keep up with the state-of-the-art tech
  - learn continuosly to be able to see the next level of this challenging game which we all trying to survive in it
  - contribute open source projects to make the world a better place
  - wake up early to catch things up, couldn't make it yet

Contact

Find me on

- github/anlcnydn
- twitter/anlcnydn

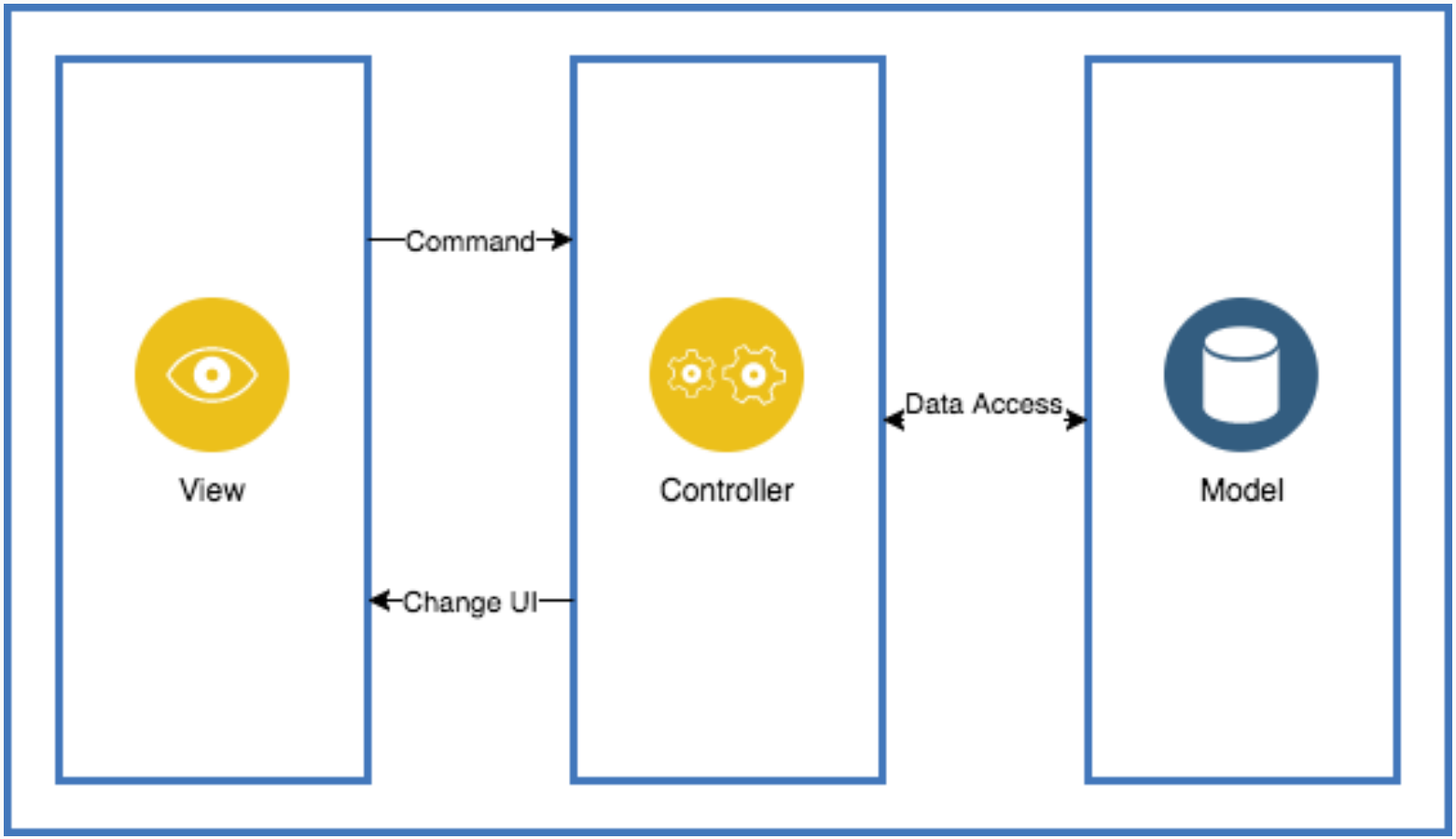
or mail me at anil@zetaops.io

Links

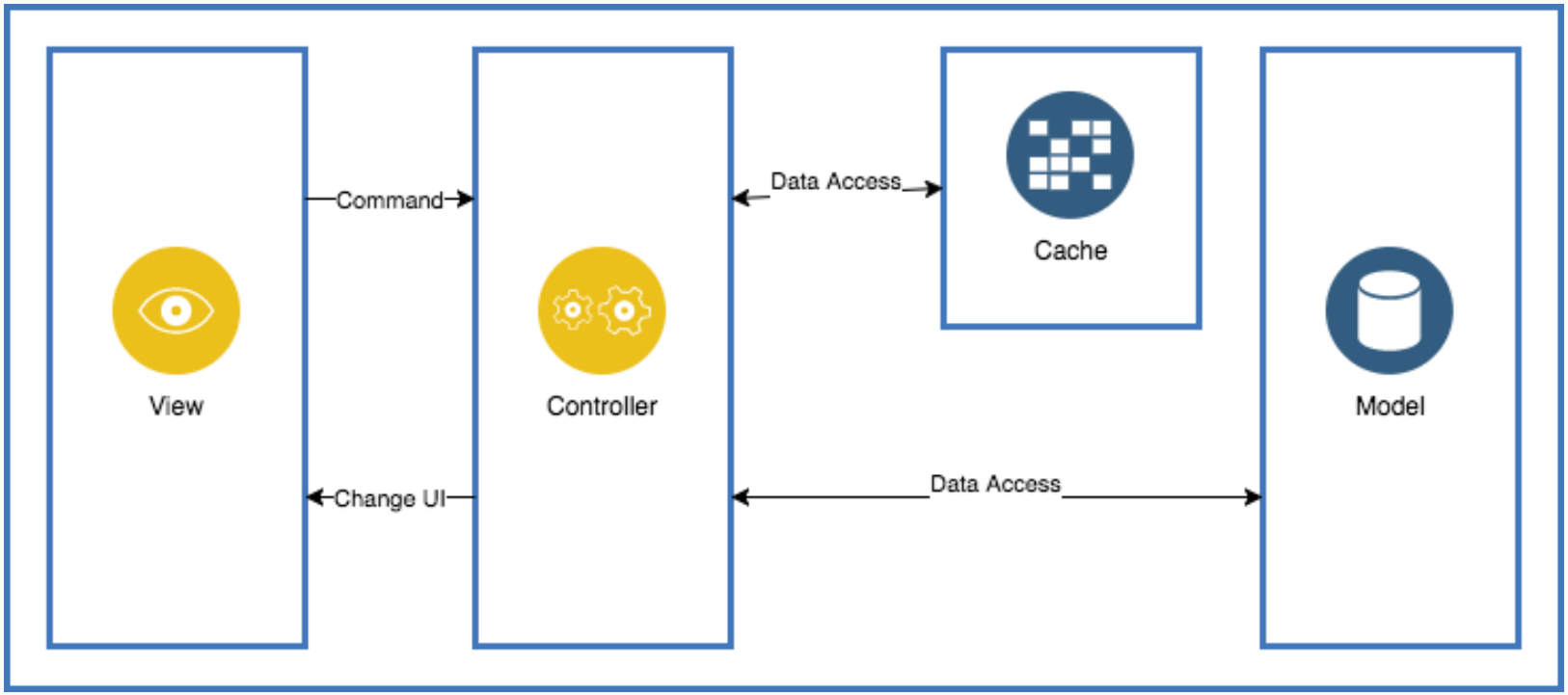
- github/anlcnydn/gdg2017-dive-into-redis
- github/anlcnydn/gdg2017-slides

Overview on Caching and Redis

Caching



Caching


View
Command
Change UI
Controller
Data Access
Cache
Data Access
Model

Caching

- Improves scalability by distributing query workload
- Allows flexibility in the processing of data
- Caching can improve availability of data
- Improved data access speeds

Why Redis

- Open source
- In-memory data structure store, used as database, cache and message broker
- Provides high performance and low latency
- Blazingly fast
- Written in C
- Optimized to handle millions of operations in a second with less than 1 ms latency in a single server
- Pre-built data structures
- Client libraries in almost every language and active developer community and contributors

Redis

- redis-cli
- redis-py

Redis Data Types

Data Types

Not a *plain* key-value store, it is actually a *data structures server*

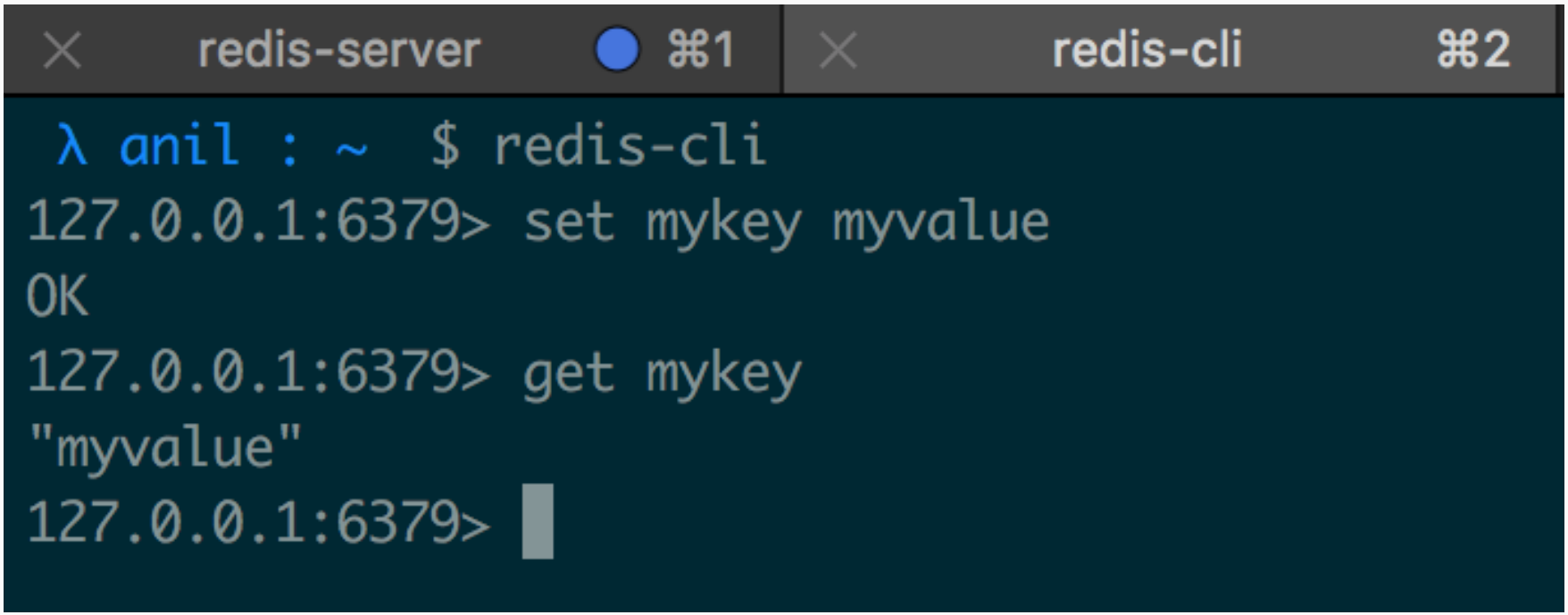
List of all the data structures supported by Redis:

- Binary-safe strings.
- Lists: sorted order of insertion, linked lists.
- Sets: unique, unsorted
- Sorted sets: Sorted by their **score**. Unlike Sets it is possible to retrieve a range of elements (for example you may ask: give me the top 10, or the bottom 10).
- Hashes, map, dict
- Bit arrays (or simply bitmaps): it is possible, using special commands, to handle String values like an array of bits
- HyperLogLogs: this is a probabilistic data structure which is used in order to estimate the cardinality of a set, also not in the scope of this presentation.

Redis Keys

- Redis keys are binary safe, this means that you can use any binary sequence as a key
- Very long keys are not a good idea.
  - Not memory efficient.
  - May increase the burden of the key look-up.
- Very short keys are often not a good idea. user:1000:followers beats u1000flw.
- Try to stick with a schema. object-type:id is a good idea, user:1000.
- The maximum allowed key size is 512 MB.

Strings


redis-server
⌘1
redis-cli
⌘2
λ anil : ~ $ redis-cli
127.0.0.1:6379> set mykey myvalue
OK
127.0.0.1:6379> get mykey
"myvalue"
127.0.0.1:6379>

SET and GET in redis-py

```
import redis

cache = redis.Redis()

cache.set("mykey", "myvalue")

print(cache.get("mykey"))
# b'myvalue' // All returning keys and values are "byte" in
                                Python 3, "unicode" in Python
                                2
```

SET

- Note that SET will replace any existing value already stored into the key
- Values can be strings (including binary data) of every kind, for instance you can store a jpeg image inside a value.
- A value can't be bigger than 512 MB.
- The SET command has interesting options, that are provided as additional arguments.
- For example, I may ask SET to fail if the key already exists, or the opposite, that it only succeed if the key already exists.

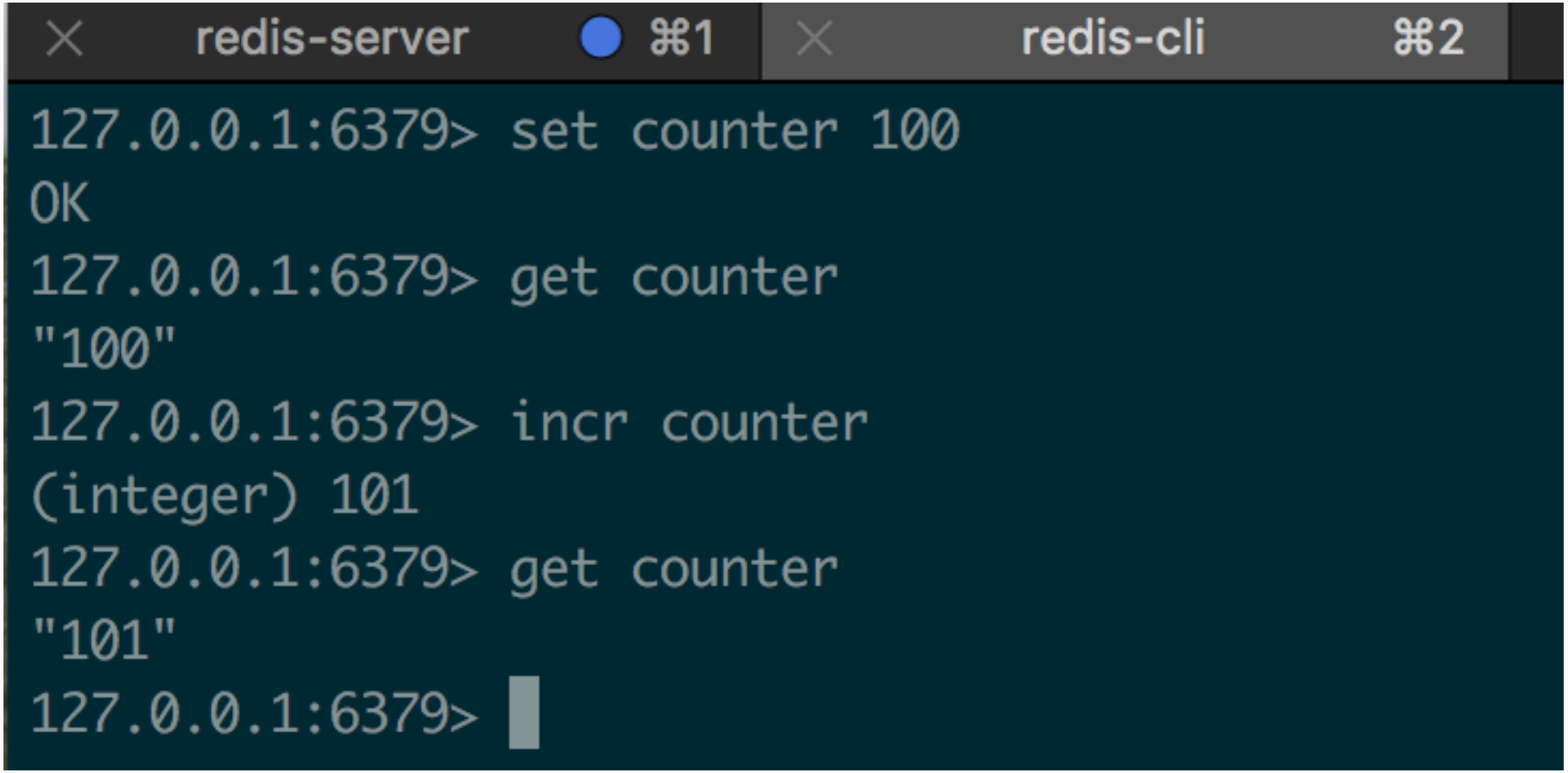
NX and XX

```
127.0.0.1:6379> get mykey
"myvalue"
127.0.0.1:6379> set mykey mynewvalue nx
(nil)
127.0.0.1:6379> get mykey
"myvalue"
127.0.0.1:6379> set mykey mynewvalue xx
OK
127.0.0.1:6379> get mykey
"mynewvalue"
127.0.0.1:6379>
```

NX and XX in redis-py

```
print(cache.get("mykey"))
# b'myvalue'

print(cache.setnx("mykey", "mynewvalue"))
# False

print(cache.get("mykey"))
# b'myvalue'

print(cache.setxx("mykey", "mynewvalue"))
# True

print(cache.get("mykey"))
# b'mynewvalue'
```

INCR


redis-server
⌘1
redis-cli
⌘2
127.0.0.1:6379> set counter 100
OK
127.0.0.1:6379> get counter
"100"
127.0.0.1:6379> incr counter
(integer) 101
127.0.0.1:6379> get counter
"101"
127.0.0.1:6379>

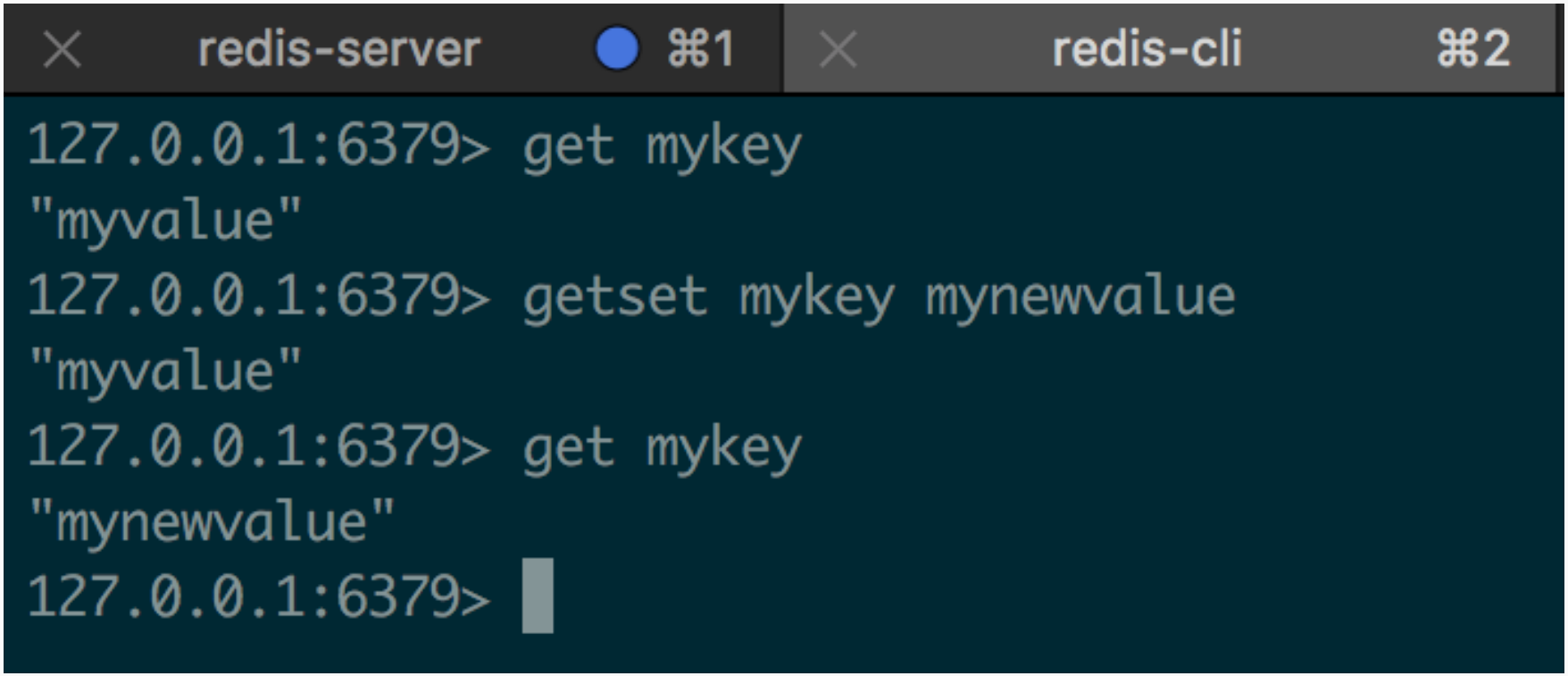
INCR in redis-py

```
cache.set("counter", 100)

print(cache.incr("counter"))
# 101 // integer

print(cache.get("counter"))
#  b '101' // byte
```

INCR, INCRBY, DECR and DECRBY

All are atomic. No race condition.

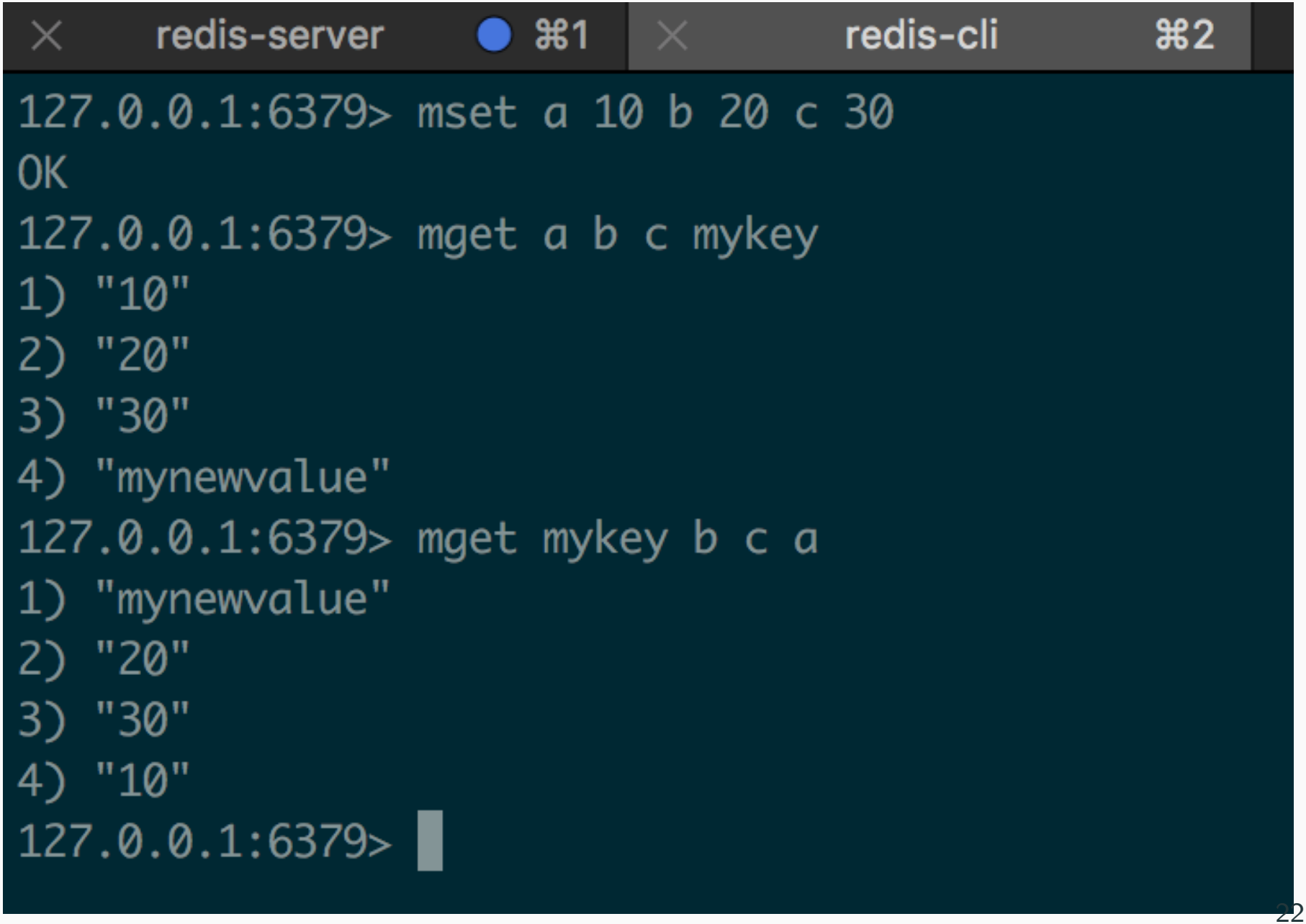
GETSET


redis-server
⌘1
redis-cli
⌘2
127.0.0.1:6379> get mykey
"myvalue"
127.0.0.1:6379> getset mykey mynewvalue
"myvalue"
127.0.0.1:6379> get mykey
"mynewvalue"
127.0.0.1:6379>

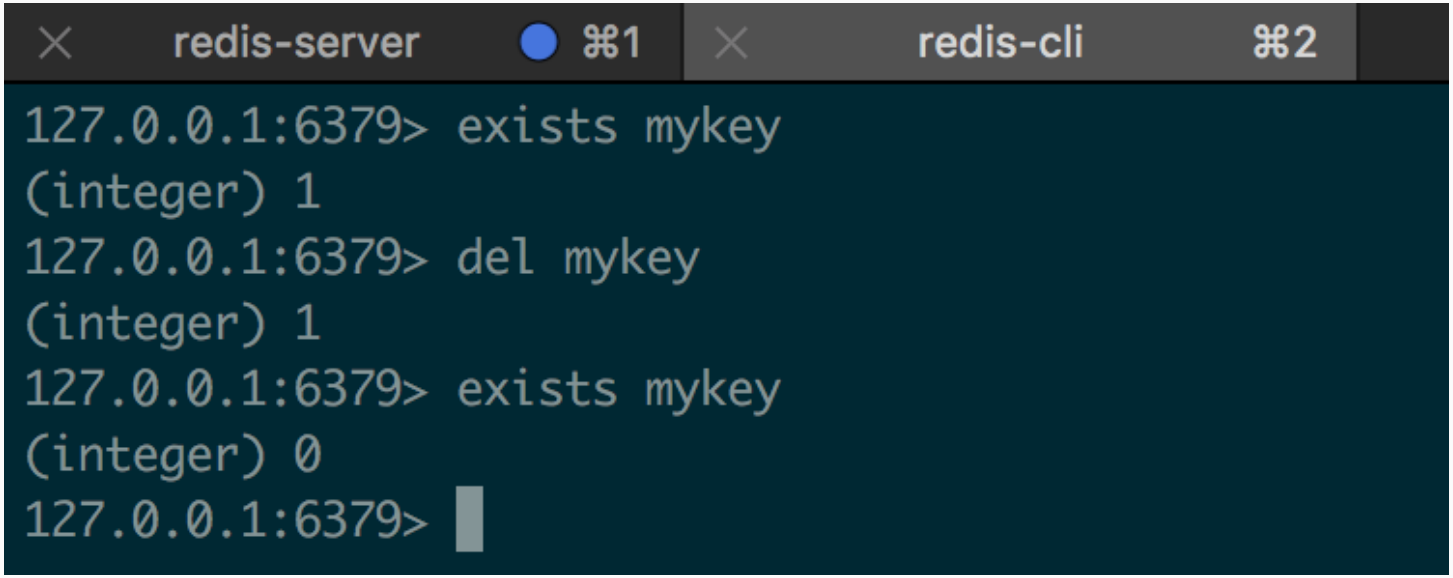
GETSET Case Story

If you have a system that increments a Redis key using INCR every time your web site receives a new visitor. You may want to collect this information once every hour, without losing a single increment. You can GETSET the key, assigning it the new value of "0" and reading the old value back.

MSET and MGET



EXISTS and DEL



Lists

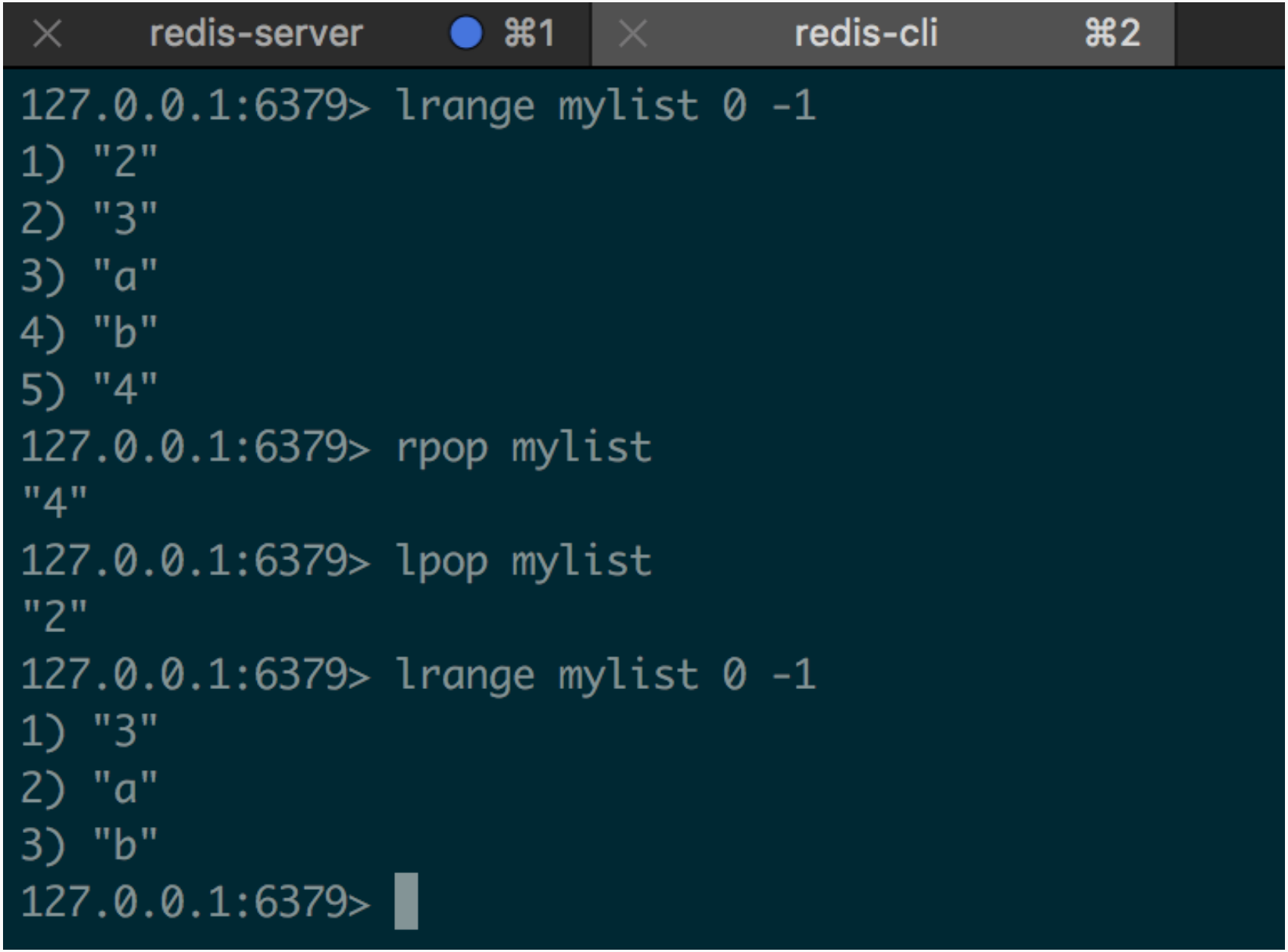
- Redis lists are implemented via Linked Lists
- The operation of adding a new element in the head or in the tail of the list is performed in constant time.
- What's the downside? Accessing an element by index is very fast in lists implemented with an Array (constant time indexed access) and not so fast in lists implemented by linked lists (where the operation requires an amount of work proportional to the index of the accessed element).
- When fast access to the middle of a large collection of elements is important, there is a different data structure that can be used, called sorted sets.

First Steps with Lists

LRANGE

- Note that LRANGE takes two indexes, the first and the last element of the range to return. Both the indexes can be negative, telling Redis to start counting from the end: so -1 is the last element, -2 is the penultimate element of the list, and so forth
- LPUSH and RPUSH are variadic commands, meaning that you are free to push multiple elements into a list in a single call
- You can pop elements from left and right, similarly to how you can push elements in both sides of the list

LPOP and RPOP



Common Use Cases for Lists

- Remember the latest updates posted by users into a social network
- Communication between processes, using a producer-consumer pattern where the producer pushes items into a list, and a consumer (usually a worker) consumes those items and executed actions. Redis has special list commands to make this use case both more reliable and efficient.
- For example both the popular Ruby libraries resque and sidekiq use Redis lists under the hood in order to implement background jobs
- The popular Twitter social network takes the latest tweets posted by users into Redis lists.

Capped Lists

- Redis allows us to use lists as a capped collection, only remembering the latest N items and discarding all the oldest items using the LTRIM command.
- The LTRIM command is similar to LRANGE, but instead of **displaying the specified range of elements** it sets this range as the new list value. All the elements outside the given range are removed.

LTRIM


redis-server
⌘1
redis-cli
⌘2
127.0.0.1:6379> rpush mylist 1 2 3 4 5
(integer) 5
127.0.0.1:6379> ltrim mylist 0 2
OK
127.0.0.1:6379> lrange mylist 0 -1
1) "1"
2) "2"
3) "3"
127.0.0.1:6379>

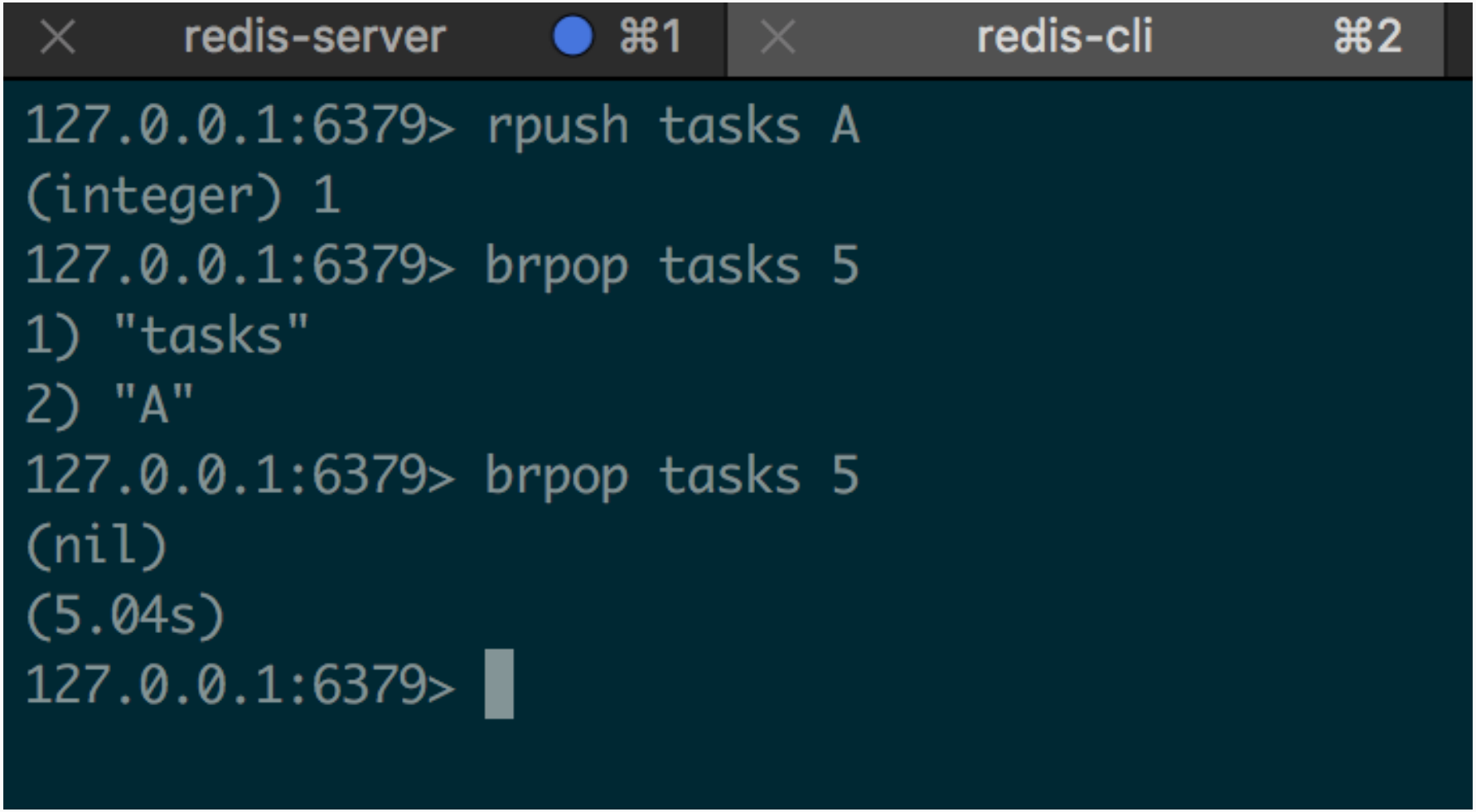
Blocking Operations on Lists

- The usual producer/consumer setup
- To push items into the list, producers call LPUSH
- To extract/process items from the list, consumers call RPOP
- If the list is empty and there is nothing to process, so RPOP just returns NULL
- In this case a consumer is forced to wait some time and retry again with RPOP
- This is called polling, and is not a good idea

BRPOP and BLPOP

BRPOP and BLPOP are versions of RPOP and LPOP able to block if the list is empty: they'll return to the caller only when a new element is added to the list, or when a user-specified timeout is reached.

BRPOP


redis-server
⌘1
redis-cli
⌘2
127.0.0.1:6379> rpush tasks A
(integer) 1
127.0.0.1:6379> brpop tasks 5
1) "tasks"
2) "A"
127.0.0.1:6379> brpop tasks 5
(nil)
(5.04s)
127.0.0.1:6379>

More on BRPOP

- Note that you can use 0 as timeout to wait for elements forever, and you can also specify multiple lists and not just one, in order to wait on multiple lists at the same time, and get notified when the first list receives an element.
- Clients are served in an ordered way: the first client that blocked waiting for a list, is served first when an element is pushed by some other client, and so forth.
- The return value is different compared to RPOP: it is a two-element array since it also includes the name of the key, because BRPOP and BLPOP are able to block waiting for elements from multiple lists.
- If the timeout is reached, NULL is returned.
- Suggested reading on RPOPLPUSH and BRPOPLPUSH

Automatic Creation and Removal of Keys

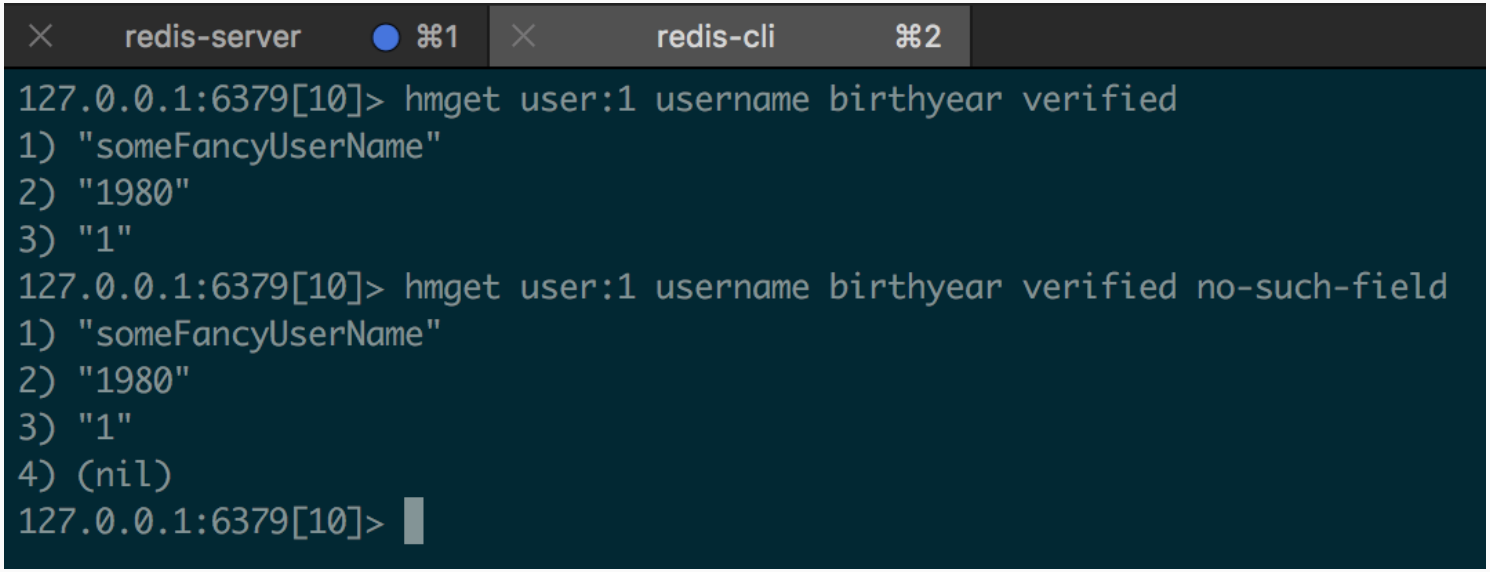
- When we add an element to an aggregate data type, if the target key does not exist, an empty aggregate data type is created before adding the element.
- When we remove elements from an aggregate data type, if the value remains empty, the key is automatically destroyed.
- Calling a read-only command such as LLEN (which returns the length of the list), or a write command removing elements, with an empty key, always produces the same result as if the key is holding an empty aggregate type of the type the command expects to find.

HMSET, HGET, HGETALL

redis-server ⌘1 | redis-cli ⌘2

```
127.0.0.1:6379[10]> hmset user:1 username someFancyUserName birthyear 1980 verified 1
OK
127.0.0.1:6379[10]> hget user:1 username
"someFancyUserName"
127.0.0.1:6379[10]> hget user:1 birthyear
"1980"
127.0.0.1:6379[10]> hget user:1 verified
"1"
127.0.0.1:6379[10]> hget user:1 veriverified
(nil)
127.0.0.1:6379[10]> hgetall user:1
1) "username"
2) "someFancyUserName"
3) "birthyear"
4) "1980"
5) "verified"
6) "1"
127.0.0.1:6379[10]>
```

HMGET


redis-server ⌘1
redis-cli ⌘2
127.0.0.1:6379[10]> hmget user:1 username birthyear verified
1) "someFancyUserName"
2) "1980"
3) "1"
127.0.0.1:6379[10]> hmget user:1 username birthyear verified no-such-field
1) "someFancyUserName"
2) "1980"
3) "1"
4) (nil)
127.0.0.1:6379[10]>

More on Hashes

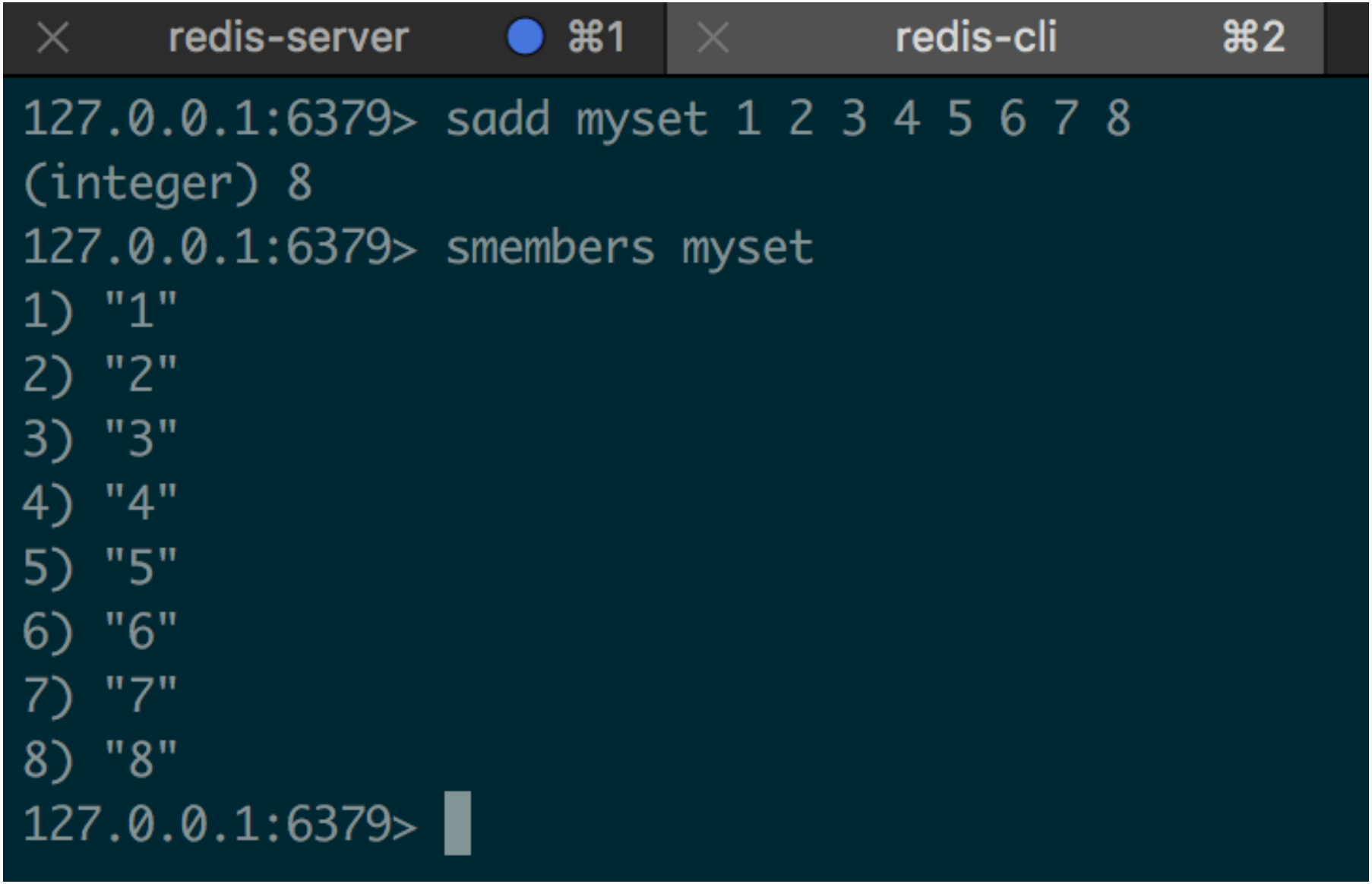
While hashes are handy to represent objects, actually the number of fields you can put inside a hash has no practical limits (other than available memory), so you can use hashes in many different ways inside your application.

More on hashes.

Sets

- Redis Sets are unordered collections of strings
- The SADD command adds new elements to a set.
- It's also possible to do a number of other operations against sets like testing if a given element already exists, performing the intersection, union or difference between multiple sets, and so forth.

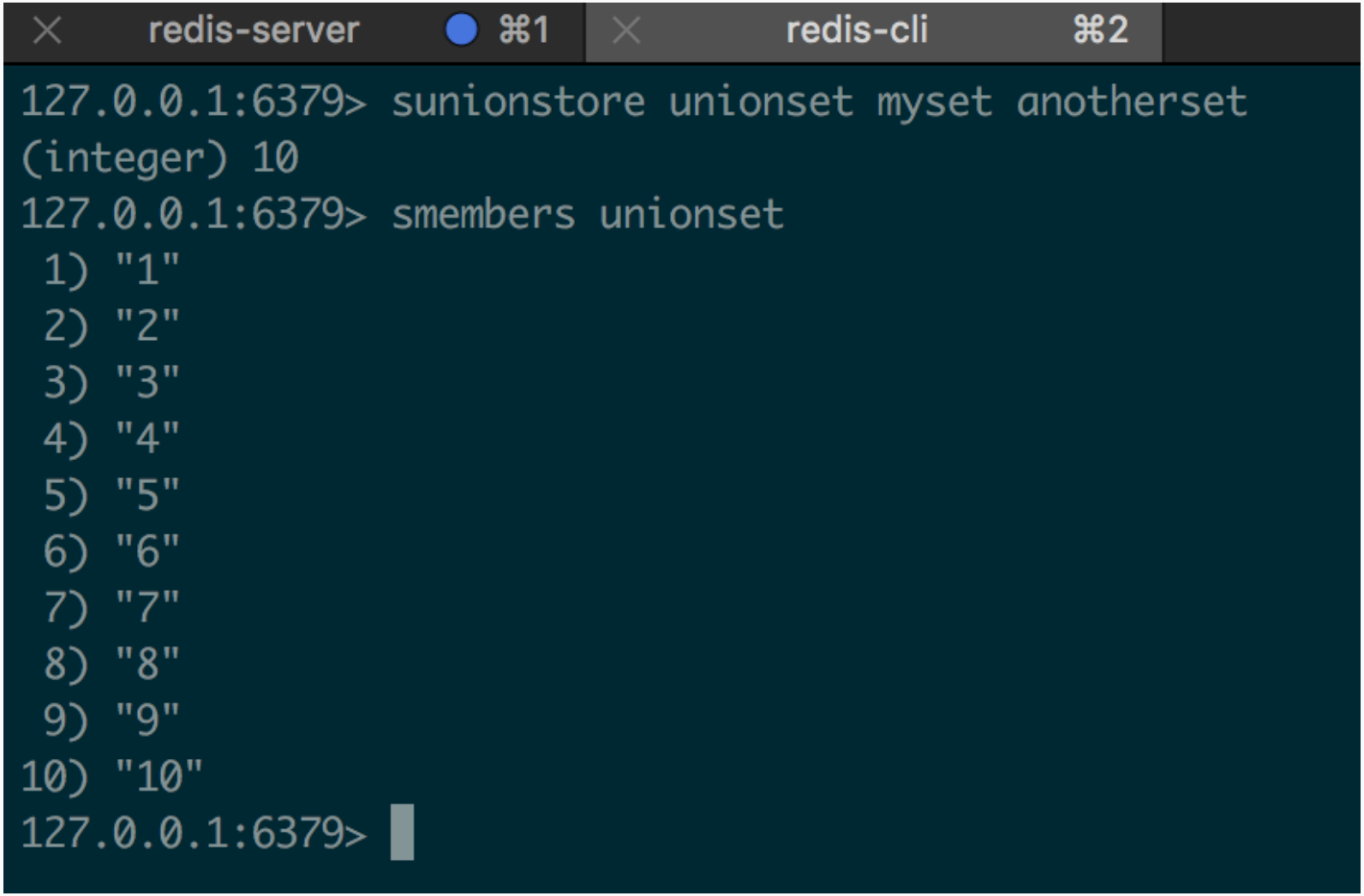
SADD


redis-server
⌘1
redis-cli
⌘2
127.0.0.1:6379> sadd myset 1 2 3 4 5 6 7 8
(integer) 8
127.0.0.1:6379> smembers myset
1) "1"
2) "2"
3) "3"
4) "4"
5) "5"
6) "6"
7) "7"
8) "8"
127.0.0.1:6379>

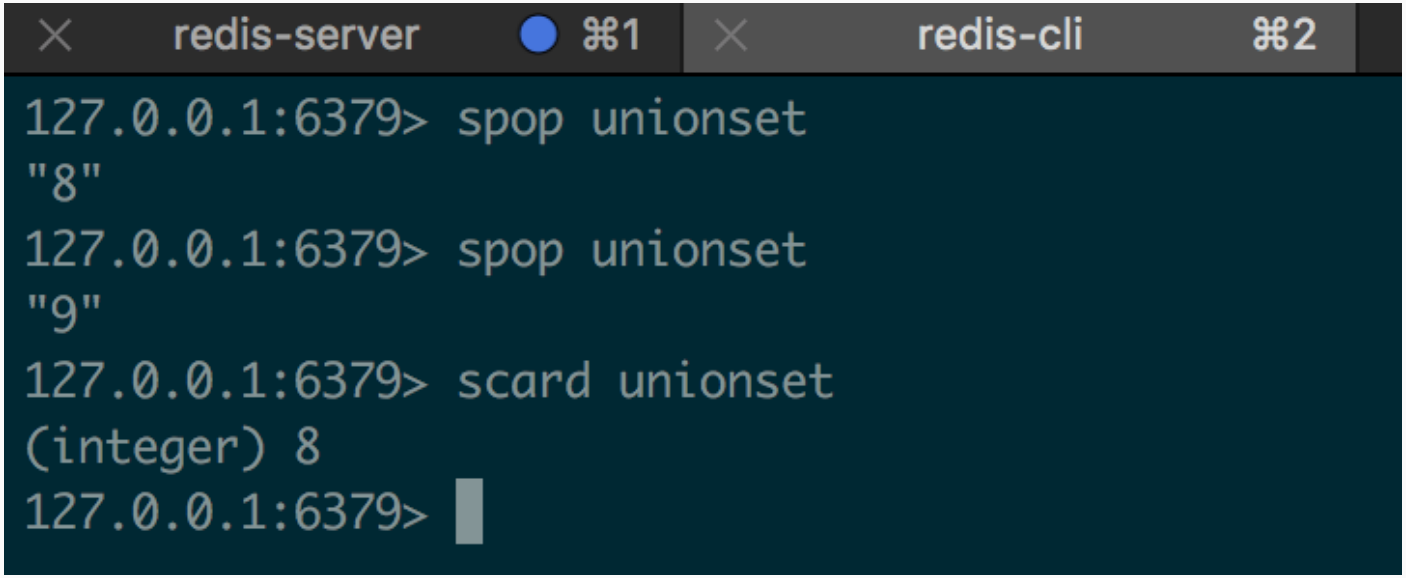
SINTER


redis-server ⌘1
redis-cli ⌘2
127.0.0.1:6379> sadd myset 1 2 3 4 5 6 7 8
(integer) 8
127.0.0.1:6379> smembers myset
1) "1"
2) "2"
3) "3"
4) "4"
5) "5"
6) "6"
7) "7"
8) "8"
127.0.0.1:6379> sadd anotherset 7 8 9 10
(integer) 4
127.0.0.1:6379> sinter myset anotherset
1) "7"
2) "8"
127.0.0.1:6379>

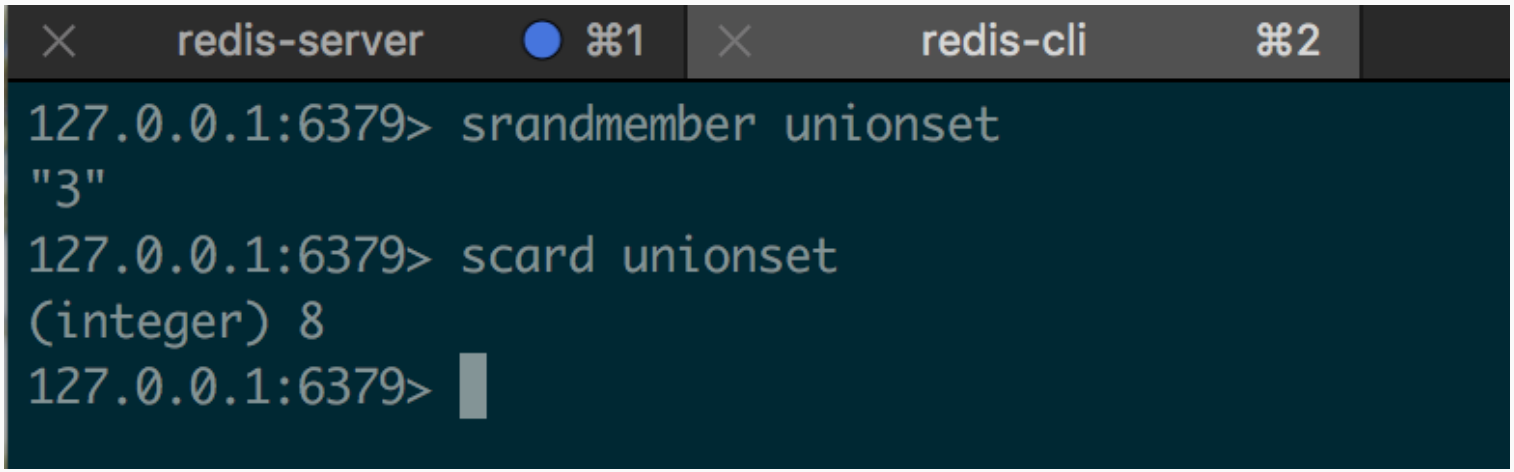
SUNIONSTORE



SPOP and SCARD



SRANDMEMBER



Sorted Sets

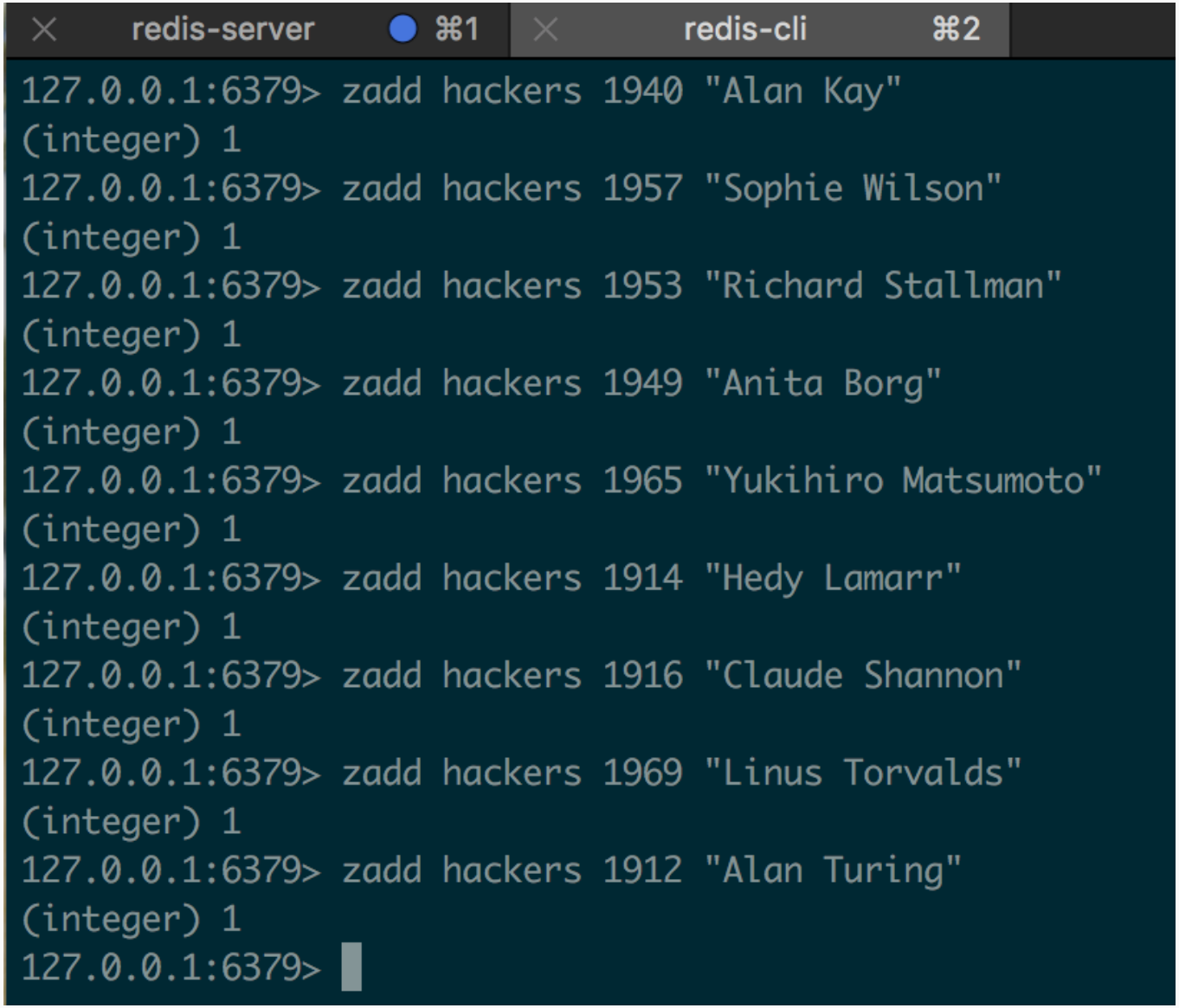
- Sorted sets are a data type which is similar to a mix between a Set and a Hash
- Like sets, sorted sets are composed of unique, non-repeating string elements, so in some sense a sorted set is a set as well
- However while elements inside sets are not ordered, every element in a sorted set is associated with a floating point value, called the score (this is why the type is also similar to a hash, since every element is mapped to a value)
- Moreover, elements in a sorted sets are taken in order (so they are not ordered on request, order is a peculiarity of the data structure used to represent sorted sets).

Sorted Sets: Ordering Rules

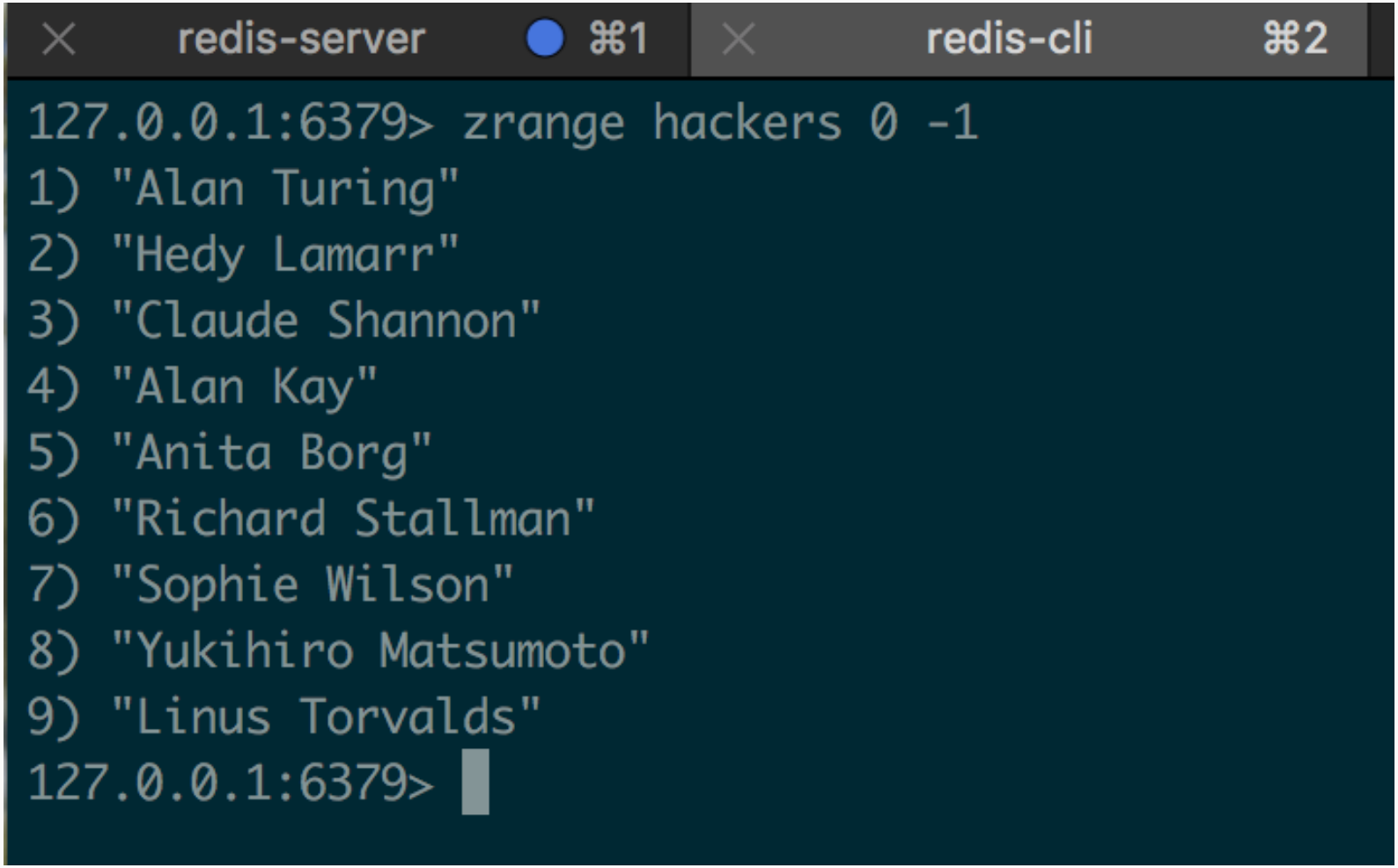
They are ordered according to the following rule:

- If A and B are two elements with a different score, then A $>$ B if A.score is $>$ B.score.
- If A and B have exactly the same score, then A $>$ B if the A string is lexicographically greater than the B string. A and B strings can't be equal since sorted sets only have unique elements.

ZADD


redis-server ⌘1
redis-cli ⌘2
127.0.0.1:6379> zadd hackers 1940 "Alan Kay"
(integer) 1
127.0.0.1:6379> zadd hackers 1957 "Sophie Wilson"
(integer) 1
127.0.0.1:6379> zadd hackers 1953 "Richard Stallman"
(integer) 1
127.0.0.1:6379> zadd hackers 1949 "Anita Borg"
(integer) 1
127.0.0.1:6379> zadd hackers 1965 "Yukihiro Matsumoto"
(integer) 1
127.0.0.1:6379> zadd hackers 1914 "Hedy Lamarr"
(integer) 1
127.0.0.1:6379> zadd hackers 1916 "Claude Shannon"
(integer) 1
127.0.0.1:6379> zadd hackers 1969 "Linus Torvalds"
(integer) 1
127.0.0.1:6379> zadd hackers 1912 "Alan Turing"
(integer) 1
127.0.0.1:6379>

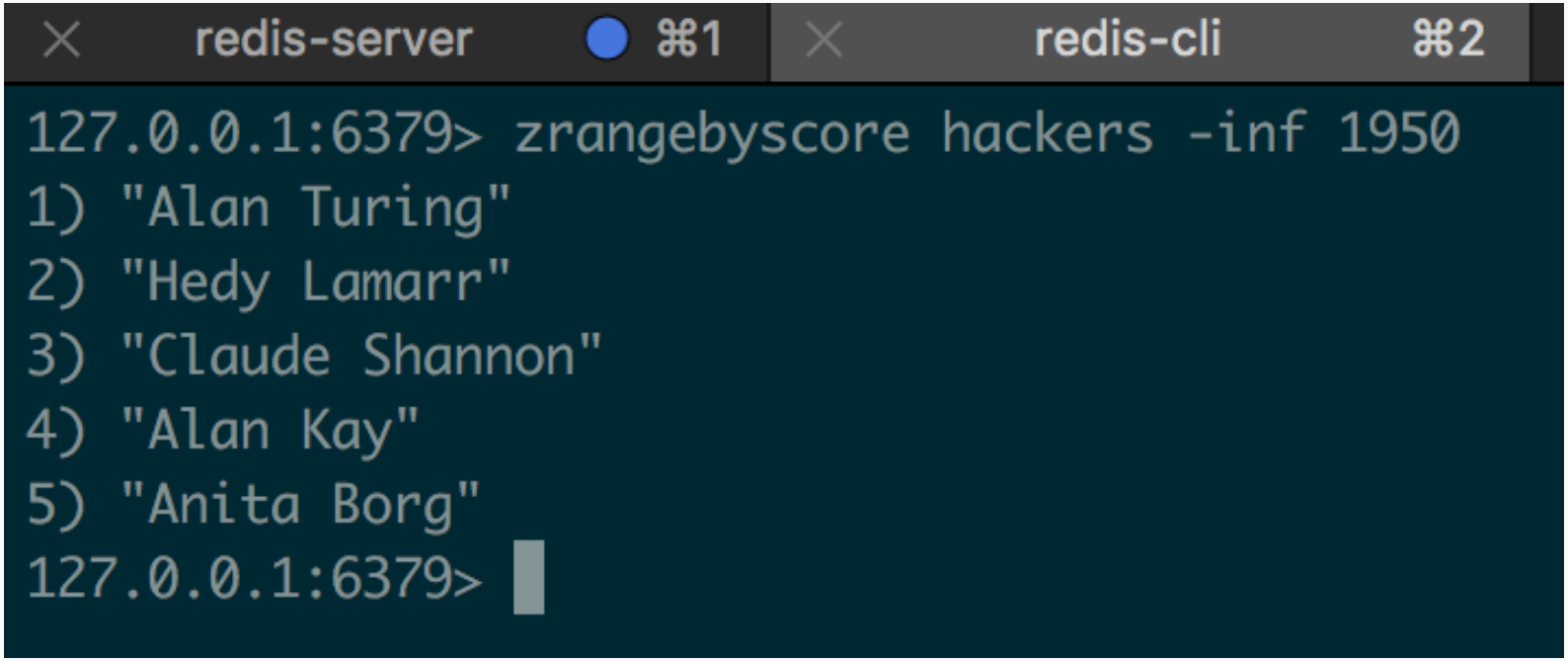
ZRANGE


redis-server
⌘1
redis-cli
⌘2
127.0.0.1:6379> zrange hackers 0 -1
1) "Alan Turing"
2) "Hedy Lamarr"
3) "Claude Shannon"
4) "Alan Kay"
5) "Anita Borg"
6) "Richard Stallman"
7) "Sophie Wilson"
8) "Yukihiro Matsumoto"
9) "Linus Torvalds"
127.0.0.1:6379>

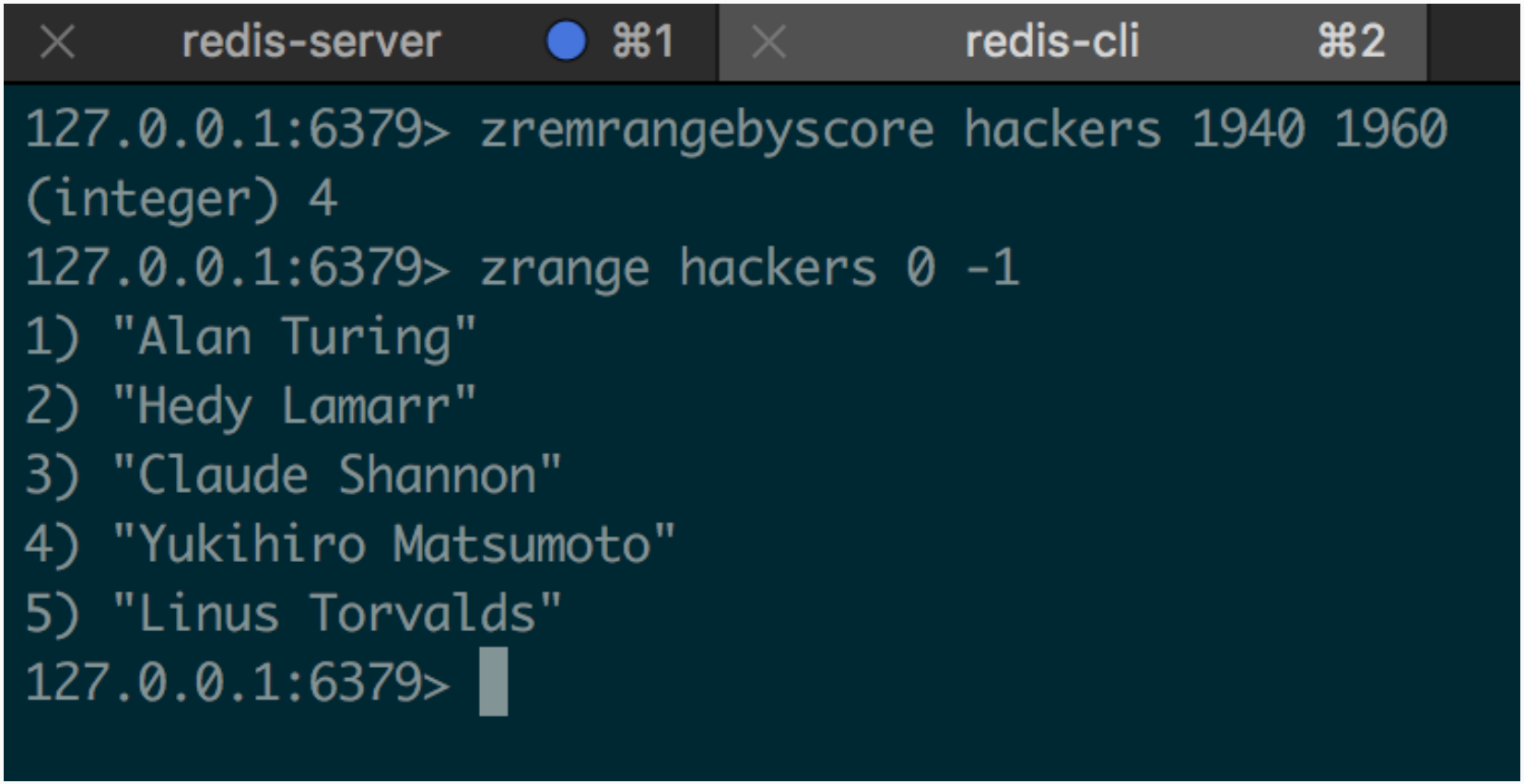
More on Sorted Sets

- With sorted sets it is trivial to return a list of hackers sorted by their birth year because actually they are already sorted.
- Implementation note: Sorted sets are implemented via a dual-ported data structure containing both a skip list and a hash table, so every time we add an element Redis performs an O(log(N)) operation. That's good, but when we ask for sorted elements Redis does not have to do any work at all, it's already all sorted.
- What if I want to order them the opposite way, youngest to oldest? Use ZREVRANGE instead of ZRANGE
- It is possible to return scores as well, using the WITHSCORES argument

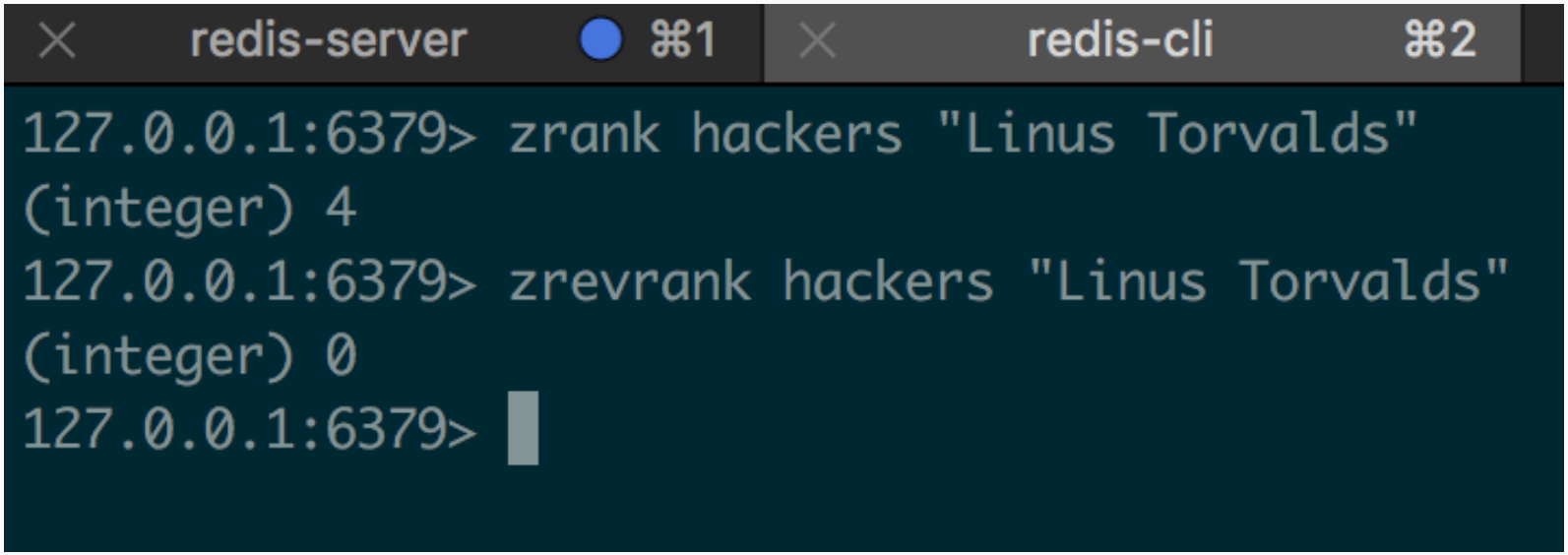
Operating on Ranges: ZRANGEBYSCORE



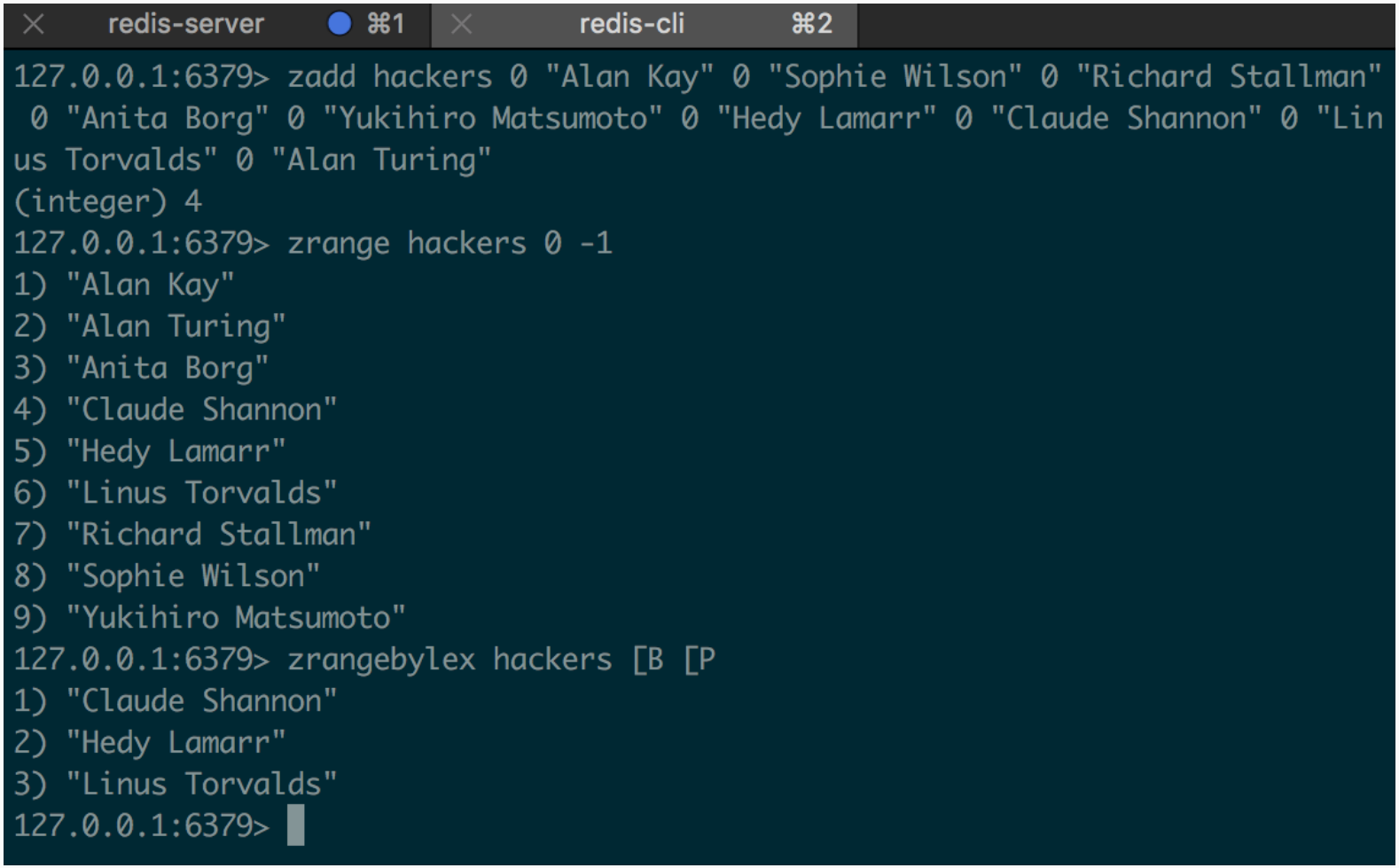
Operating on Ranges: ZREMRANGEBYSCORE



Operating on Ranges: ZRANK and ZREVRANK



Lexicographical Scores: ZRANGEBYLEX



Bitmaps

Bitmaps are not an actual data type, but a set of bit-oriented operations defined on the String type.

Data Types Conclusion

- Binary-safe strings.
- Lists
- Sets
- Sorted sets
- Hashes
- Bitmaps

Redis Pipelining

Request/Response protocol

Redis is a TCP server using the client-server model and what is called a Request/Response protocol [1].

This means that usually a request is accomplished with the following steps:

- The client sends a query to the server, and reads from the socket, usually in a blocking way, for the server response.
- The server processes the command and sends the response back to the client.

[1] Redis Clients Handling

Request/Response protocol

Round Trip Time(RTT)

The round-trip delay time (RTD) or round-trip time (RTT) is the length of time it takes for a signal to be sent plus the length of time it takes for an acknowledgment of that signal to be received [2].

[2] Round Trip Time - Wikipedia

Round Trip Time(RTT)

For instance if the RTT time is 250 milliseconds (in the case of a very slow link over the Internet), even if the server is able to process 100k requests per second, we'll be able to process at max four requests per second.

If the interface used is a loopback interface[3], the RTT is much shorter (for instance my host reports 0,044 milliseconds pinging 127.0.0.1), but it is still a lot if you need to perform many writes in a row.

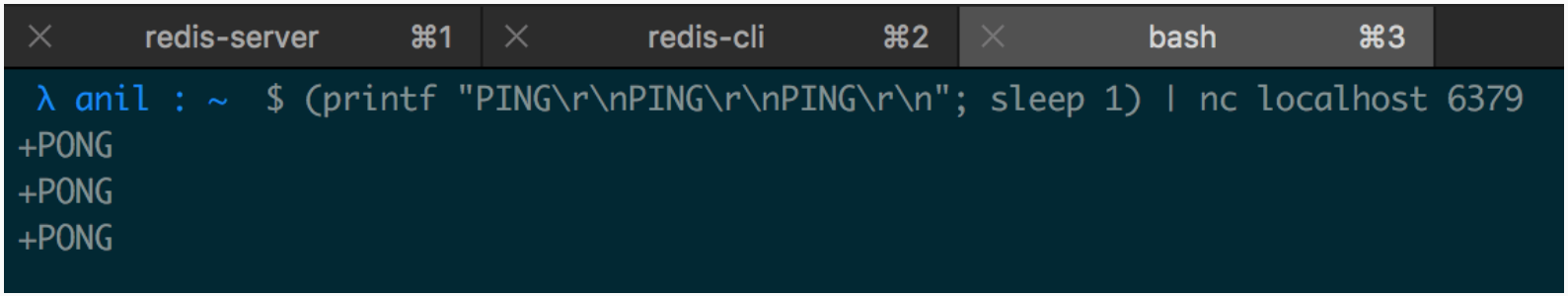
Fortunately there is a way to improve this use case.

[3] Loopback, or loop-back, refers to the routing of electronic signals, digital data streams, or flows of items back to their source without intentional processing or modification. Loopback - Wikipedia

Redis Pipelining

It is possible to send multiple commands to the server without waiting for the replies at all, and finally read the replies in a single step.

Redis Pipelining



Redis Pipelining

```
 λ anil : ~  $ (printf "incr a\r\nincr a\r\nincr a\r\n"; sleep 1) | nc localhost 6379
:1
:2
:3
--
 ~
 λ anil : ~  $ (printf "incr a\r\nincr a\r\nincr a\r\n"; sleep 1) | nc localhost 6379
:4
:5
:6
```

Redis Pipelining in pyredis

```
import redis

cache = redis.Redis()

cache.set('bing', 'baz')

pipe = cache.pipeline()

pipe.set('foo', 'bar')
pipe.get('bing')
# the EXECUTE call sends all buffered commands to the server
                                , returning a list of
                                responses, one for each
                                command.
pipe.execute()
# [True, 'baz']
```

Redis Pipelining

IMPORTANT NOTE:

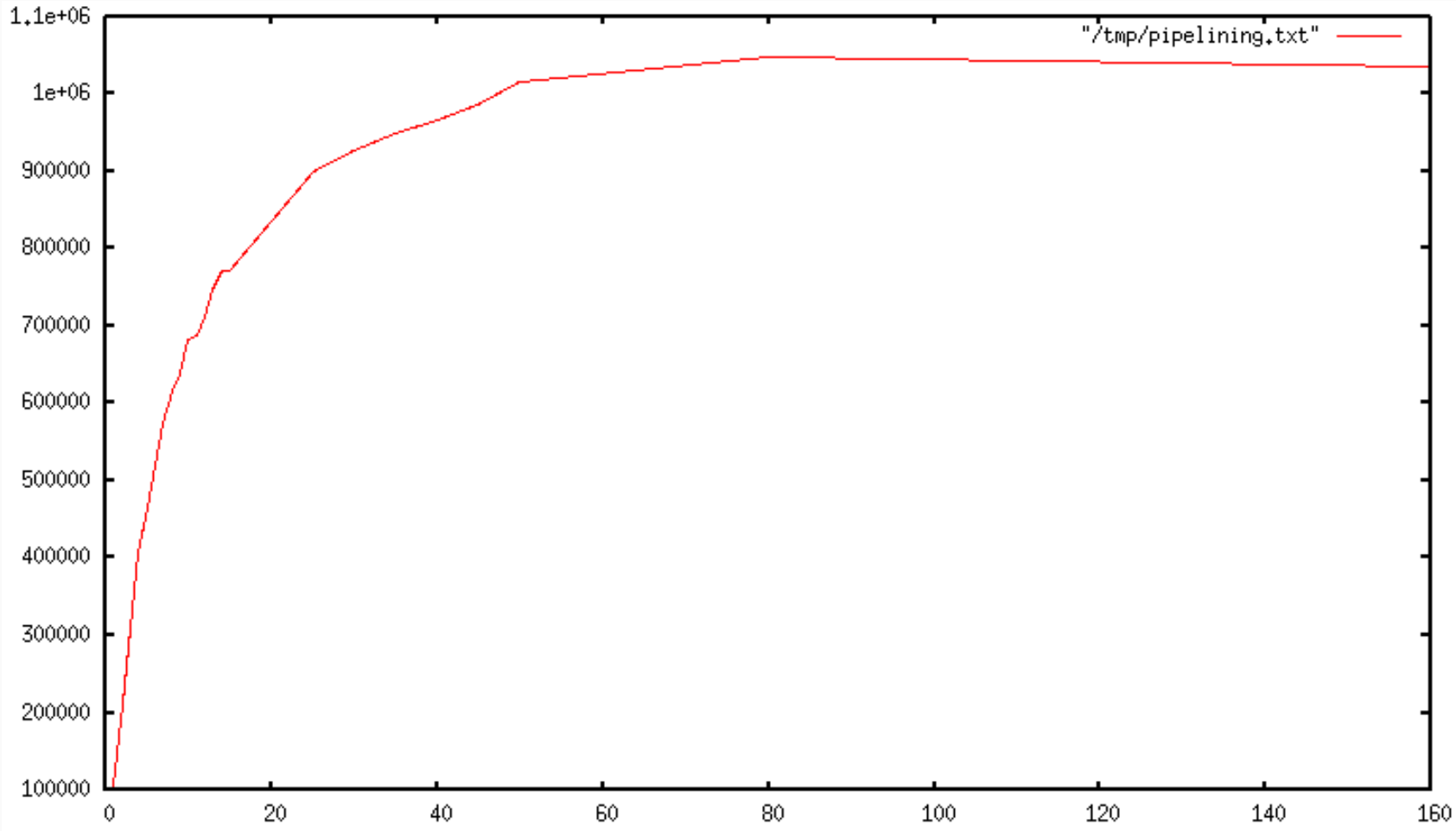
- While using pipelining, the server will be forced to queue the replies, using memory.
- So if you need to send a lot of commands with pipelining, it is better to send them as batches having a reasonable number, for instance 10k commands, read the replies, and then send another 10k commands again, and so forth.
- The speed will be nearly the same, but the additional memory used will be at max the amount needed to queue the replies for this 10k commands.

It's not just a matter of RTT

- Without using pipelining, serving each command is very **cheap** from the point of view of accessing the data structures and producing the reply, but it is very **costly** from the point of view of **doing the socket I/O**.
- This involves calling the read() and write() syscall, that means going from user land to kernel land. The context switch [4] is a huge speed penalty.
- When pipelining is used, many commands are usually read with a **single** read() system call, and multiple replies are delivered with a **single** write() system call.

[4] Context Switching - Wikipedia

It's not just a matter of RTT

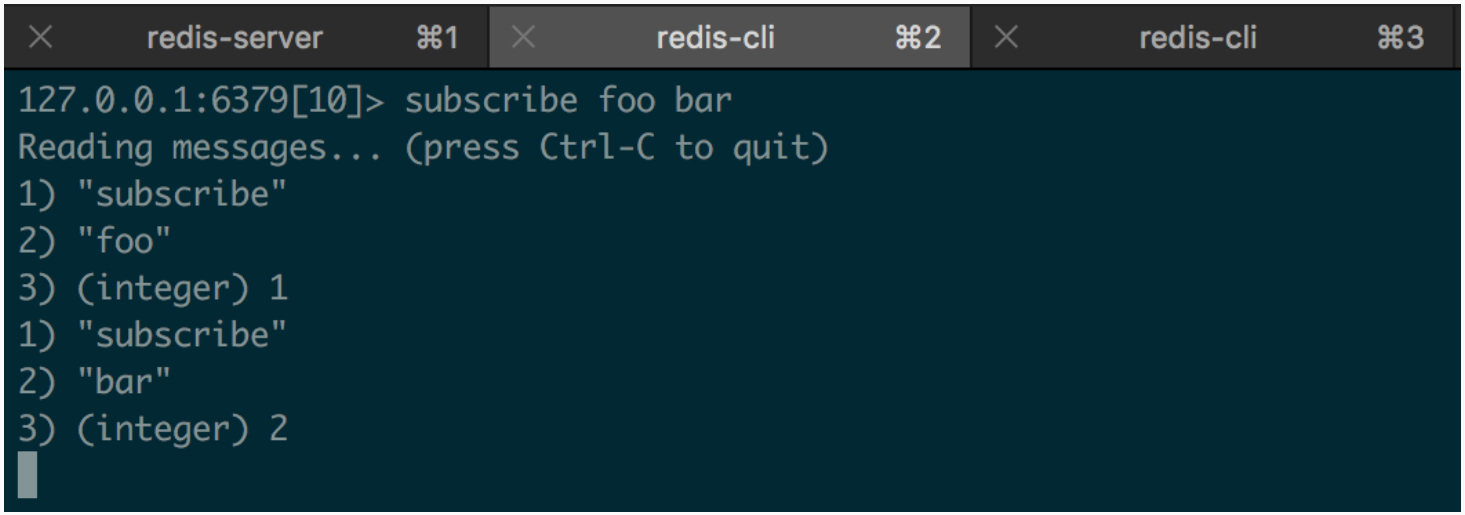


Redis Pub/Sub

Pub/Sub

Senders (publishers) are not programmed to send their messages to specific receivers (subscribers). Rather, published messages are characterized into **channels**, without knowledge of what (if any) subscribers there may be.

SUBSCRIBE


redis-server ⌘1
redis-cli ⌘2
redis-cli ⌘3
127.0.0.1:6379[10]> subscribe foo bar
Reading messages... (press Ctrl-C to quit)
1) "subscribe"
2) "foo"
3) (integer) 1
1) "subscribe"
2) "bar"
3) (integer) 2

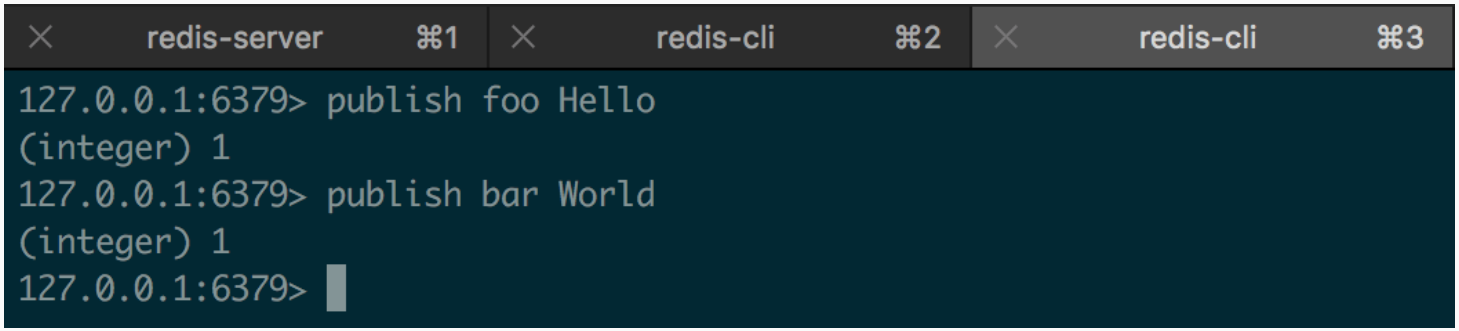
SUBSCRIBE Cont...

- A client subscribed to one or more channels should not issue commands, although it can subscribe and unsubscribe to and from other channels.
- The commands that are allowed in the context of a subscribed client are SUBSCRIBE, PSUBSCRIBE, UNSUBSCRIBE, PUNSUBSCRIBE, PING and QUIT.

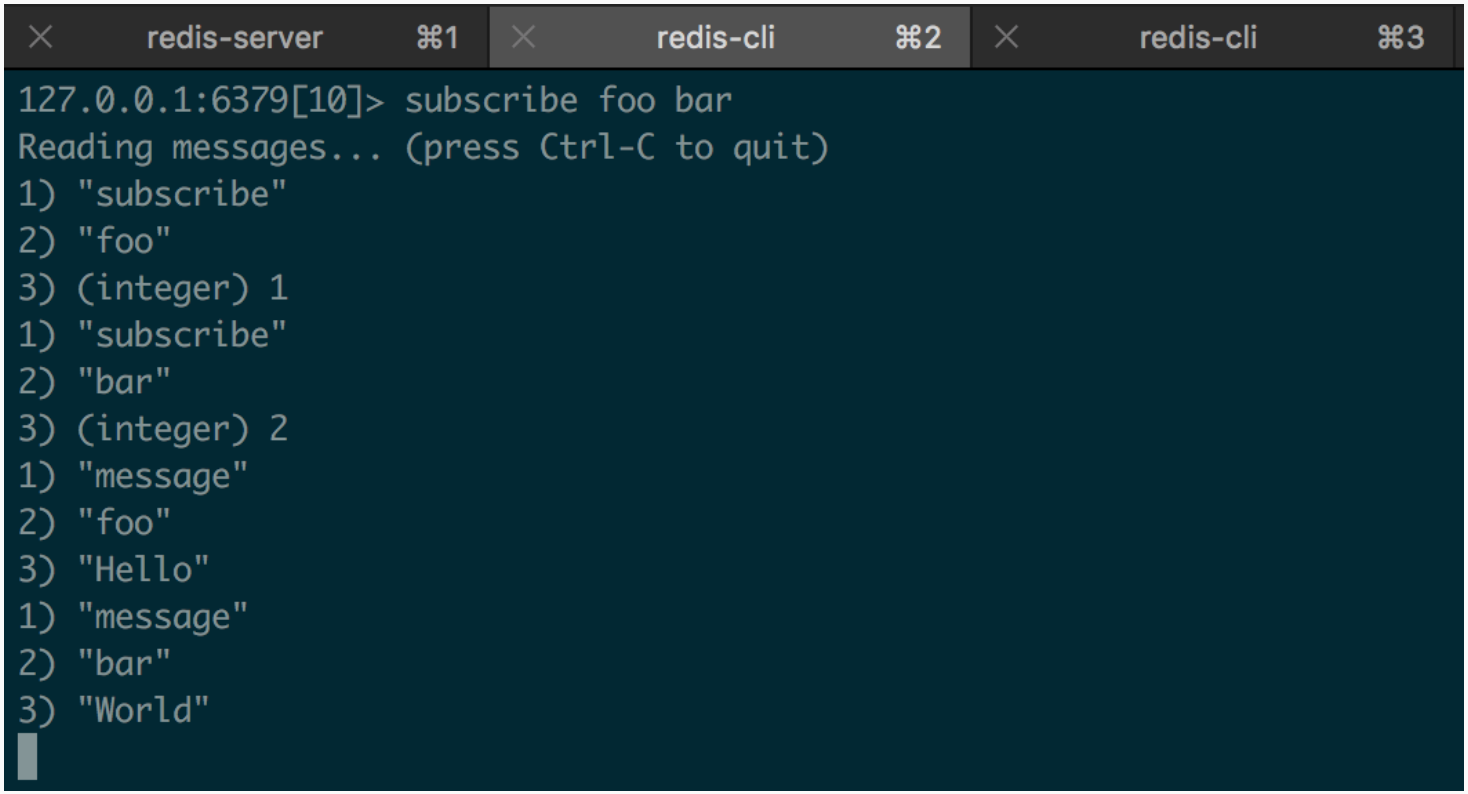
SUBSCRIBE Cont...

- It is issued a subscription event for channels foo and bar, and both succeed.
- So, client at tab 2 is listening channels foo and bar.
- Let's publish some messages and see what happens.

PUBLISH


redis-server ⌘1
redis-cli ⌘2
redis-cli ⌘3
127.0.0.1:6379> publish foo Hello
(integer) 1
127.0.0.1:6379> publish bar World
(integer) 1
127.0.0.1:6379>

Messages


redis-server ⌘1
redis-cli ⌘2
redis-cli ⌘3
127.0.0.1:6379[10]> subscribe foo bar
Reading messages... (press Ctrl-C to quit)
1) "subscribe"
2) "foo"
3) (integer) 1
1) "subscribe"
2) "bar"
3) (integer) 2
1) "message"
2) "foo"
3) "Hello"
1) "message"
2) "bar"
3) "World"

Messages Cont...

- So, Hello is published to channel foo and World is published to channel bar.
- Subscriber client received the messages in a specific format.

Messages Cont...

- A message is an Array reply with three elements.
- The first element is the kind of message:
  - **subscribe**: means that we successfully subscribed to the channel given as the second element in the reply. The third argument represents the number of channels we are currently subscribed to.
  - **unsubscribe**: means that we successfully unsubscribed from the channel given as second element in the reply. The third argument represents the number of channels we are currently subscribed to. When the last argument is zero, we are no longer subscribed to any channel, and the client can issue any kind of Redis command as we are outside the Pub/Sub state.
  - **message**: it is a message received as result of a PUBLISH command issued by another client. The second element is the name of the originating channel, and the third argument is the actual message payload.

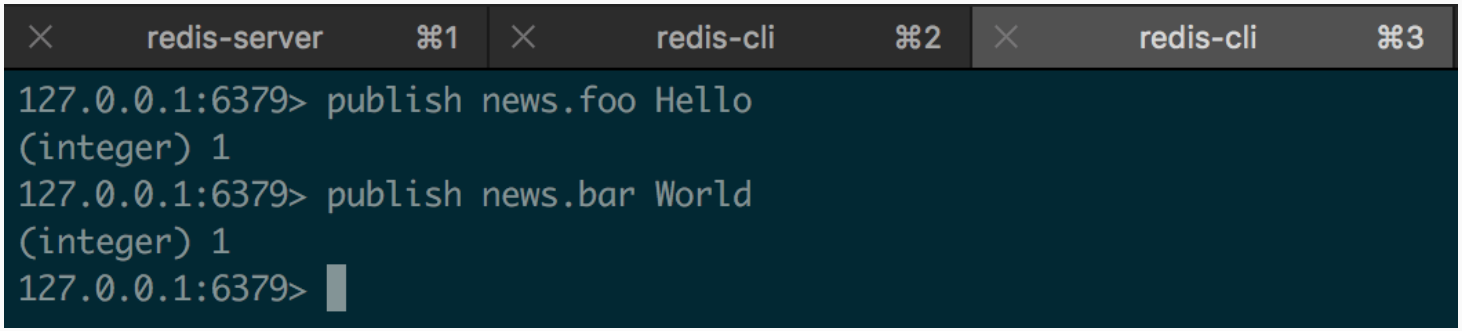
Pattern-matching subscriptions

- The Redis Pub/Sub implementation supports pattern matching.
- Clients may subscribe to glob-style patterns in order to receive all the messages sent to channel names matching a given pattern.

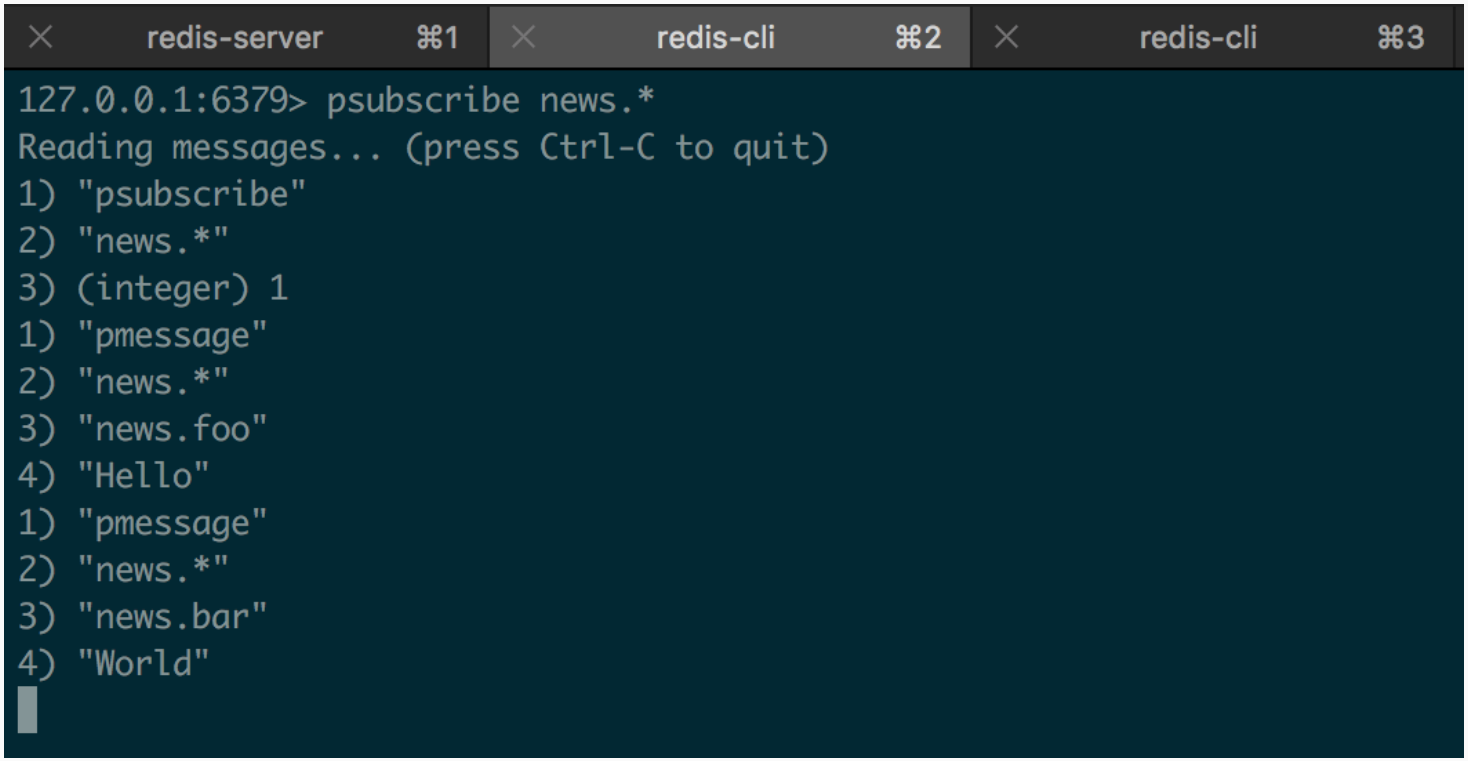
PSUBSCRIBE


redis-server ⌘1
redis-cli ⌘2
redis-cli ⌘3
127.0.0.1:6379> psubscribe news.*
Reading messages... (press Ctrl-C to quit)
1) "psubscribe"
2) "news.*"
3) (integer) 1

PSUBSCRIBE Cont...



PSUBSCRIBE Cont...



PMessage

Messages received as a result of pattern matching are sent in a different format:

- **pmessage**: it is a message received as result of a PUBLISH command issued by another client, matching a pattern-matching subscription. The second element is the original pattern matched, the third element is the name of the originating channel, and the last element the actual message payload.

Redis Pub/Sub Conclusion

Redis gives a simple and handy usage of pub/sub pattern. Only thing to do is sending commands to it.

Questions

```
while questions:
    question = questions.pop()
    try:
        answer(question)
    except Exception:
        pass

demo_app()
```

Demo

Demo - Vocabulary Application

Simple command line application that stores the words with their translations and give a chance to test user itself with pop-quiz option.

- add
- update
- read
- delete
- list
- quiz

Data-structures, pipelining and pub/sub mechanism are used.

Conclusion

Conclusion

Done.

- Data types
- Pipelining
- Pub/sub

Further.

- Keyspace Notifications
- Memory Optimization

Suggested Reading and Media

- Real Time Delivery Twitter
- RPOPLPUSH
- BRPOPLPUSH
- Redis Clients Handling
- Keyspace Notifications
- Memory Optimization
- Memory Optimization Case Story
- Messaging at Scale at Instagram

Contact

Find me on

- github/anlcnydn
- twitter/anlcnydn

or mail me at anil@zetaops.io

References

- Data Types
- Pipelining
- Pub/Sub
- Commands