Reliable Publisher/Subscriber Service

SDLE 2021/2022 - Turma 2 - Grupo 13

Diogo Nunes
up201808546@up.pt

José Silva
up201705591@up.pt

Mariana Soares
up201605775@up.pt

Rita Mota
up201703964@up.pt

***Abstract*—The first project for the Large Scale Distributed Systems class was to develop a reliable Publisher/Subscriber service with exactly-once semantics. The report elaborates on the project requirements, design decisions, implementation and finally finishes with a discussion about the achieved results.**

***Index Terms*—publisher, subscriber, broker, reliability, exactly-once semantics, zeromq**

I. Introduction

The first assignment for the Large Scale Distributed Systems class was to design and implement a reliable publish-subscribe service that guarantees exactly-once delivery.

In a reliable Publisher/Subscriber service, Publishers attempt to share a message with all Subscribers of that message's topic. A great example can be newsletters since a newsletter writer sends one newsletter to all newsletter subscribers. In addition, a subscriber may only start receiving messages after subscribing to a topic and subscriptions must be durable, which means that a subscriber must receive all messages put into a topic after subscribing to it.

Another characteristic is that Publishers don't need to know how or when the information will be processed. This means that they don't need to wait for subscribers to receive the information, thus providing a more flexible and robust system.

II. Problem Specification

The Publisher/Subscriber service must be implemented on top of ØMQ[1], an embeddable message library. It must also implement four basic operations:

- *put()* used by the publisher to send messages to a topic
- *get()* used by a subscriber to receive messages, must specificy the topic
- *subscribe()* used by a subscriber to express interest in a topic
- *unsubscribe()* used by a subscriber to disregard a topic

In the context of this assignment, a topic is an arbitrary set of bytes and can be created when a subscriber tries to subscribe to a nonexistent topic.

Subscribers must identify themselves throughout all the message exchanges.

Finally, the service must reliable and guarantee an exactly-once delivery, meaning that it needs to be prepared for failure and that each subscriber should only receive each message one time.

[1]ZeroMQ

A. Exactly-once delivery

Over an unreliable channel it is impossible to design a system that can reliably reach **consensus** - the problem is better illustrated with the **Two Generals Problem** [4]. The problem shows that two generals in two different cities that can only communicate by sending a messenger through enemy fields **can agree** on a specific action, i.e an attack, but **can't agree** on the specific time to perform the action.

A similar situation is exposed with a exactly-once delivery system. Depending on the behaviour of the broker the exactly-once semantics are broken:

- If the broker decides to assume that when a message is sent then it is delivered, there's the risk the receiver doesn't get it due to network malfunction or system crash, thus the message is lost
- If the broker decides to wait for the confirmation from the receiver, then there's the possibility of the receiver requesting for a message twice, thus a message a is delivered twice breaking the exactly-once semantics.

It's up to the system designer to decide what fits better the problem.

B. Reliable service

Reliability in Publisher/Subscriber means that a broker does its utmost keep track of the messages flow while being *crash resistant*. This can be achieved through **persistence**, the problem is that it hurts performance.

A system that focus too much of persisting data data will have reduced throughput, while a system that persists data infrequently risks suffering a major data loss upon a crash.

Besides that it is important to keep track of how much items are kept in memory, if too many items are in memory the system risks to run into a **Out-of-Memory** crash, thus there must be a limit.

C. Scalibility

Even though scalibility is not the main goal of the project, it is important to develop the project with it in mind and discuss possible strategies how that can be achieved.

III. Proposed Solution

A. Architecture

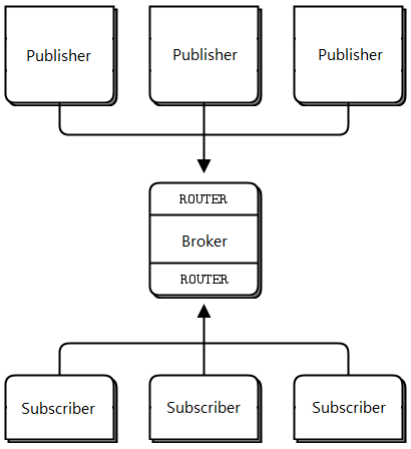

Fig. 1. Architecture following the Simple Pirate pattern

The broker exposes two *ROUTER* sockets, one for the publisher and one of the subscriber. Using two sockets allows the subscriber and publisher logic to be segregated and to be scaled independently. For example, the publisher socket can be in a company's intranet and the subscriber socket exposed to the whole internet.

The ROUTER sockets is the asynchronous version of the REP socket, which allows the broker to respond to requests out-of-order [2].

Additionally, the ROUTER sockets can talk natively with REQ and DEALER sockets [2] which gives flexibity to the developers to choose.

B. Other considered architectures

1) SUB and PUB based architecture: Even though ØMQ offers this socket pair for Publisher/Subscriber type services, they don't respect the restrictions imposed the the specification. A subscriber must connect to a publisher and inform it what topics it is interested in, it is not scalable because subscribers must be aware of all other publishers. Besides that a publisher can post a message without having subscribers [2], thus they are effectively lost.

2) XSUB and XPUB based architecture: They are lower level version of the PUB/SUB socket pair [2] and help to solve the discoverability problem exposed in the previous paragraph.

A broker would expose the a XPUB/XSUB pair and all the nodes would have to connect to it. The problem is that it suffers the same problems as the original pair, if the broker goes down the Publisher will still be able to send its messages but they will be lost. Additionally there's no way of sending messages to a specific node, thus if a subscriber goes down temporarily there is no way of recovering gracefully.

3) PUSH and PULL based architecture: It is an unfit pair for the problem, its main purpose it to evenly distribute jobs [2]. It offers doesn't allow a specific node to be targeted.

4) REQ and REP based architecture: Logically it's a good fit for the system. The broker can expose REP sockets and all other nodes must connect to them, since the broker must reply to every request than it allows the nodes to only move forward upon a response.

Sadly the pair is unfit for production use [2], its functionality is based on a finite state machine that can easily **lock**. The REP socket **must** reply to every request, thus if the requester goes down the system is locked. That's because even if the requester comes back up since it is using a REQ socket it must send a message before reading any message, leading to a deadlock.

C. Reliability mechanisms

Reliability has exposed in the section II-B is split into two categories: **system state persistence** and **resource maintenance**.

For system state persistence, *SQlite3* was chosen to store the data, due to being a well tested *DBMS*[2] with extremely good performance. Three periodic tasks were decided as crucial to maintain the system state:

1) **Message caching** - a set number of topics is chosen accordingly to last time their messages were cached, their queues are cached
2) **Subscribers caching** - a set number of topics is chosen accordingly to last time their subscribers were cached, current list of subscribers and their current position in message queue is cached
3) **Unsubscribers trimming** - it is not efficient to check for a given topic which subscribers have unsubscribed but are still persisted to the database. Because of that topics maintain a unsubscriber list that will clear the entries from the database

Regarding **resource maintenance** the following jobs were decided:

1) **Message queue trimming** - message queues can't grow forever and need to be trimmed according to a predefined size
2) **Topic Unloading** - the broker can't keep all the topics in memory and topics that are not accessed for a given period of time can be dumped to the database
3) **Stale message removal** - a message that has been read by all subscribers should be deleted
4) **Fast startup** - on startup the broker should load all topics in the database in an *unloaded* state, allowing for faster restarts and less memory pressure. This allows to reduce the effects of a server crash [3]

A message that is **sent** to the subscriber is **considered as delivered**, this guarantees that a **message isn't sent twice** but if there's a problem with the network or the subscriber crashes the message is lost. The main driver for this decision is that is allows the broker to focus on delivering fresher messages and the subscribers to move forward.

[2]Database Management System

IV. Implementation

All components of the system were developed in **Python**.

A. Broker

There are two main threads, the **Subscriber** and the **Publisher** handlers. Both of them deal with the incoming messages from their sockets. Messages are dispatched to handlers, the handlers respond with the response message.

Then there are auxiliary threads that are responsible of preserving the system's reliability

1) **Message queue caching and trimming** - responsible for caching and trimming the queues, this guarantees that whenever a message is removed from the queue it's guaranteed to already be persisted
2) **Subscriber caching** - saves the list of subscribers to the database
3) **Unsubscriber removal** - removes from the database the unsubscribers
4) **Read message removal** - removes messages that have been read by all subscribers
5) **Topic Unloader** - keeps the loaded topic list to a fixed size, by dumping to the database topics that haven't been accessed for a given period of time

1) Exception swallowing: All threads in the broker perform *exception swallowing*, if an unhandled exception bubbles up to the main loop it is caught, logged and swallowed. This allows the system to always move forward and not be brought down by a single point of failure. In case of any processing error the message handler sends an *INVALID* response, informing the receiver there was a problem with processing their request.

2) Database interface: To interface with the database **SQLAlchemy** [6] was used, it is an ORM[3] that operates with several flavors of DBMS, including *SQLite3*.

3) get()ing message not in queue: If a message has been written to the database and trimmed from the message queue then the only option of reading it is by reading from the database. The message is not preserved in memory afterwards which means the queue is only reserved for freshest messages.

B. Subscriber

A subscriber can instantiate the **Subscriber** class. It offers the following primitives which offers the following primitives:

- *subscribe()* - takes the topic identifier as an argument
- *unsubscribe()* - takes the topic identifier as an argument
- *get()* - takes the topic identifier and maximum number of messages to retrieve(the broker can have a lower limit) and returns the messages available. It is available in **blocking** and **non-blocking** mode.

C. Publisher

A publisher can instantiate the **Publisher** class. It offers the following primitives which offers the *put()* primitive that takes the topic identifier and message content as an argument also in **blocking** and **non-blocking** mode.

[3]Object Relational Mapping

D. Agnostic to content

Instead of being limited to strings, the broker operates on byte arrays. This allows the broker to support **all** types of string encodings and also non-valid strings. This provides extra flexibility to the subscribers and publishers, allowing to them to exchange arbitrary messages. Subscriber identifiers are also byte arrays.

E. Message encoding

A goal of the implementation was to support sending **arbitrary** messages over the network, while respecting the respective message type structure. The solution was to use **COBS**[4] encoding , it's a byte stuffing method that converts an arbitrary byte array into a byte array without null bytes (the size might increase).

Since null bytes won't show in encoded content, then they're used as separators. A message then can be split on null bytes and its contents interpreted.

F. Message Types

There are ten types of messages:

1) **SUBSCRIBE** - contains identifier of subscriber and desired topic to subscribe to
2) **UNSUBSCRIBE** - contains identifier of subscriber and desired topic to unsubcribe from
3) **GET** - contains identifier of subscriber and number of requested messages
4) **PUT** - contains topic identifier and message to be posted
5) **ACK** - acknowledge, used by the broker to let the receiver their request was succesful
6) **INVALID** - there was a problem with a problem
7) **NO_MESSAGES** - informs the subscriber there are no new messages for him for a given topic
8) **NOT_SUBSCRIBED** - informs the subscriber he's not subscribed to the given topic
9) **ALREADY_SUBSCRIBED** - informs the subscriber he's already subscribed to the topic
10) **NO_SUBSCRIBER** - informs the publisher there's no subscriber, thus the message won't be posted

G. Graceful Shutdown

Upon receiving a **SIGNINT** the broker will dump everything it has in memory to the database and wait for the threads to join. If the operator decides to kill it earlier, sending another SIGINT will terminate the broker immediately.

[4]Consistent Overhead Byte Stuffing

H. Broker customization

The broker was developed to be highly customizable, allowing it to adapt to different kinds of environment. The default values were also chosen to be as conservative as possible without hurting performance

1) **max_topics_in_memory** - soft limit for how many topics can be in memory at a given time - *default 2*
2) **topic_queue_size** - soft limit for how many messages can be in a topic message queue - *default 5*
3) **max_topics_per_cache_call** - how many topics are handled for each time the cache is triggered - *default 1*
4) **topic_unload_interval** - how often should the topic unloader run - *default 10 seconds*
5) **cache_interval** - how often the cache threads should run - *default 10 seconds*
6) **unsub_interval** - how often the unsubscriber thread should run - *default 30 seconds*
7) **stale_msg_interval** - how often the stale message removal should run - *default 30 seconds*

1) Soft limits: The limits are not *hard* in the sense that they are only enforced when the responsible thread runs, this allows to reduce the I/O stress and to scale according to the incoming traffic.

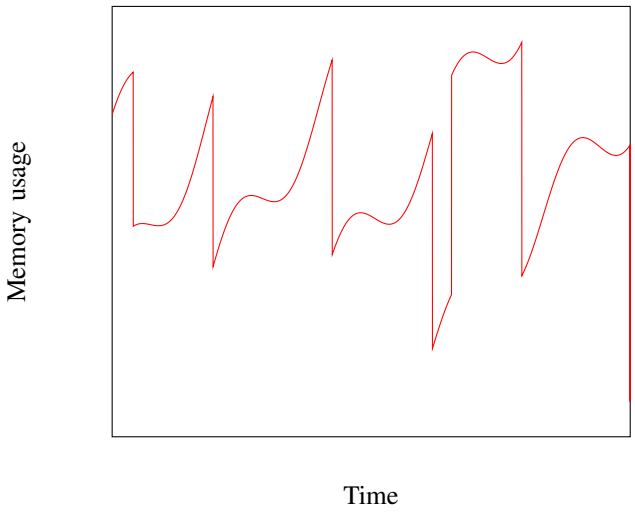

Fig. 2. Memory usage over time

Memory usage as seen in Figure 2 will operate won't have an hard limit, but overtime the usage will stabelize, so it's important for the operator to leave out some buffer space to let it scale.

I. Blocking and Non-Blocking

A major constraint of the project was that *get()* calls must block until there's a message in the queue for the subscriber. This is not scalable because a connection might hang for an undetermined period of time, if the subscriber goes down then the broker kept it open for nothing. The solution was for the broker to reply with *NO_MESSAGES* when there are no messages and for the subscriber to perform exponential back off between successive retries. Instead of busy waiting for messages the subscriber now tries to get messages at a more realistic and scalable rate. Besides that a developer can choose not to wait for new messages and get an immediate response - non-blocking mode.

The default values used for the exponential back-off were 2 for the base and 60 seconds[5] for maximum wait time, Figure 3.

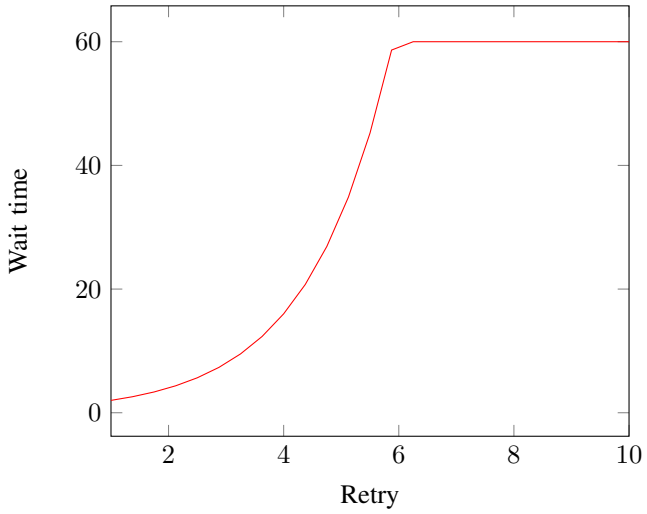

Fig. 3. Wait time per retry

J. Code structure

- **argument_parser.py** - offers a flexible interface to parse arbitrary arguments, highly customizable
- **message.py** - contains enum and classes of message wrappers
- **proxy.py** - source code of the broker(proxy)
- **publisher.py** - source code of the publisher
- **subscriber.py** - source code of the publisher
- **utils.py** - utilitaries, one off functions and COBS encoding handler
- **zeromq_entities.py** - ØMQ abstractions with support for exponential backoff retrials
- **tester.py** - throughput tester for the broker, measures time and message content
- **proxy.sh** - script that boots the broker(proxy.py) with non-default settings, used for testing

[5]The value was chosen because it only performs 60 attempts in an hour, which is low enough bandwidth and fast enough to get new commesages

V. Discussion

A. Asynchronous

Due to being powered by the *ROUTER* sockets the current implementation is asynchronous since messages can be processed out-of-order.

B. Exactly-once semantics

As mentioned in section III-C the broker considers that a sent message is delivered, which means that if there's a network issue or the receiver crashes while processing there's no way of getting that message again.

Not allowing the subscriber to decide whether or not they've actually received the message allows the system to progress forward without focusing too much on older messages and without breaking the semantics. It also improves the overall performance because there's a high chance an older message will need to be read from the database.

This consideration was the result of analyzing the use cases of a Publisher/Subscriber system, they are most useful when dealing with **recent** data, thus there are no incentives in focusing too much on lost messages.

C. Multi-threaded processing

Even though the broker heavily uses threads it is processing data in **parallel**. This is because of Python's **GIL**[6], which only allows a native thread to run at a time. Threads only perform context switch upon a I/O operation, such as reading and writing to the database or reading and writing to a socket.

A possible solution to overcome this problem would be to use a distribution of Python that doesn't contain the GIL, such as JPython. The default Python distribution[7], can shine in a multi-threaded environment given that it is I/O heavy. Since all auxiliary threads only operate with the database they're constantly allowing other threads to run, no thread is starving. Even for the main Subscriber and Publisher threads, all processing of data is localized and very small prioritizing responding as fast as possible.

Concluding, even though there are limitations due to of **CPython's**, the broker offers multi-threaded processing but not **parallel** processing. If a user wants true parallelism then switching the Python distribution is an option.

D. Reliability

The reliability mechanisms, section III-C, guarantee that the broker can have a speedy error recovery, i.e frequent data persistence with fast startup, and that during the broker's runtime the resource limits are enforced periodically.

Not having hard limits on resources allows the broker to keep a more consistent throughput. If they were implemented bad actors could force topics to be constantly loaded and unloaded leading to a **thrashing**[8] scenario, which would majorly slowdown the performance.

[6]Global Interpreter Lock
[7]CPython
[8]Spending more time in I/O than CPU tasks

The only downside of not having hard limits is that if the cleaner threads don't run frequent enough the broker might run out of memory. Because of this the broker contains insightful logs that can be fed into a tool such as *ElasticSearch* and the broker operators can analyze how data is flowing in the broker and customize its parameters according to the necessities.

E. Scalability

The broker as defined in section IV-H can be **vertically scaled** - adding more resources to the machine will allow the operator to set higher limits. It can also be **horizontally scaled** due to the fact it does not require to have all the topics in memory and can load them from a database. Since *SQLAlchemy* is being used the underlying DBMS can be replaced, e.g *SQLite* $\rightarrow$ *PostgreSQL* removing the need for each node to keep the database in their respective disk.

A possible scalability strategy is to use a load balancer that performs **consistent hashing** [5], since the topic identifier must be present in all message requests then it can be the input for the hashing algorithm. This also means the brokers will become specialized in only certain messages, thus allowing the bucketization of topics. Additionally, consistent hashsing allows the addition/removal of nodes in a graceful manner and since the brokers have their own clean-up and bootstrap routines the network will rebalance without causing a major destabilization of the system.

The broker could be incorporated into a auto-scaling strategy, that depending on the traffic would spin up more broker instances to handle the traffic.

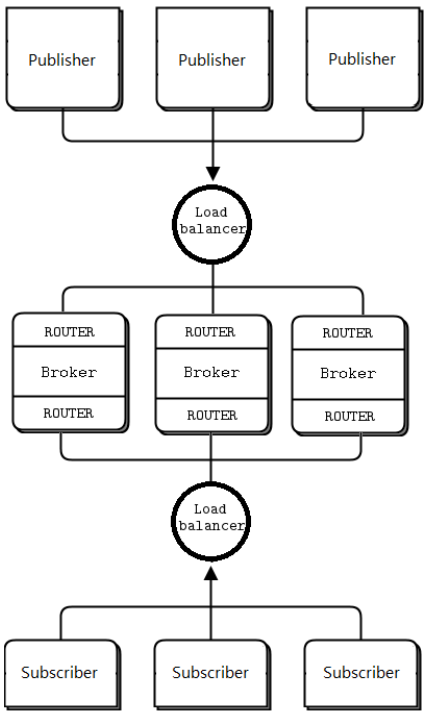

Fig. 4. Scalable Architecture

1) Availability: The broker offers high availability [1] due do the fact **fast start-up** was implemented which helps to minimize the down time. If consistent hashing is used a node going down will also be barely noticed by the requesters (the latency might increase momentarily).

2) Efficiency: The broker maintains high efficiency [1] with the aid of the helper threads, which isolate the I/O operations without impacting the response time.

VI. Validation

The broker was validated using a script that supported the following options:

- Number of unique messages to generate
- Number of subscribers to listen to all messages
- Flag to validate whether there are no duplicated messages

At the end of the execution it reports how much time did it took to complete[9].

To test the broker two types of test were run. First the **message scaling test** - how the system handles when more messages need to be delivered - and secondly **subscriber scaling test** - how the system handles when there's more subscribers. For all tests a single publisher was used and the broker used the following non-default settings:

1) **max_get_msgs** at 100, this allows the subscribers to retrieve up to 50 messages per *get()*
2) **topic_queue_size** at 400, the default topic queue size was 5 which when dealing with thousands of messages that read *all* the messages causes the messages to **only** be read from the database.

All the tests were run in a Thinkpad E595 with a *Ryzen 7 3700U*.

A. Subscriber Scaling Test

5000 messages were sent for all tests, only the number of subscribers increased.

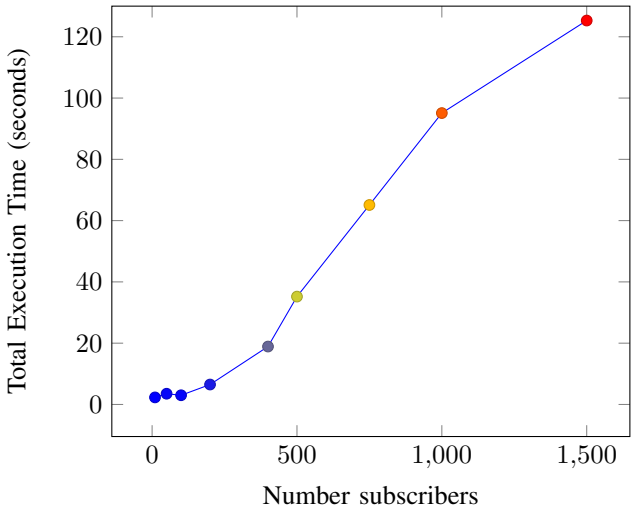

Fig. 5. Subscriber scaling graph

[9]All subscribers receive all messages

B. Message Scaling Test

100 subscribers were sent for all tests, only the number of messages increased.

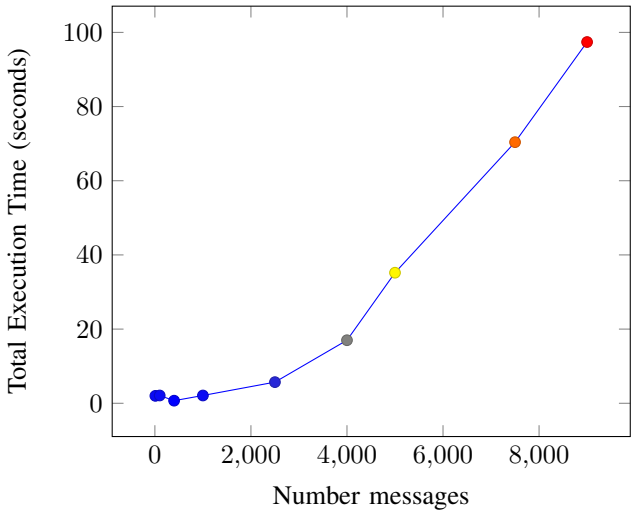

Fig. 6. Message scaling graph

C. Takeaways

According to Figure 5 subscriber scaling displays a **parabolic** growth, while according to Figure 6 message scaling displays **exponential scaling**. Has expected has the variable increases beyond the expected settings the performance degrades, but even under high stress situations[10] the execution time only reaches 2 minutes.

The tests were run in a under-powered laptop CPU that only exhibited major slowdowns when the traffic didn't meet the expectations, thus it can be safely concluded the broker is **adapted to deal with anomalous situations and exporadic bursts of traffic**.

VII. Closing Remarks

The proposed solution and respective implementation respect all the requirements from the Problem Specification, section II, with proper justfication for each design decision.

Even though *exactly-once semantics* is an unsolvable problem in Computer Science it doesn't mean it can't be "solved" for specific use cases, such a Publisher/Subscriber service with emphasis on delivering the freshest data.

References

[1] An Introduction to Distributed Systems - http://webdam.inria.fr/Jorge/html/wdmch15.htmlx21-30400014.3.1
[2] The ZeroMQ Guide - for Python Developers, Pieter Hintjens
[3] Reliability issues in distrubuted operation systems, Andrew S. Tanenbaum, Robbert van Renesse
[4] Two Generals' Problem - https://en.wikipedia.org/wiki/Two_Generals%27_Problem
[5] Consistent Hashing - https://en.wikipedia.org/wiki/Consistent_hashing
[6] SQLAlchemy - The Database Toolkit for Python - https://www.sqlalchemy.org/

[10]1 order of magnitude higher than expected load