Broker Project

Amir Fakharzadeh

Table of Contents

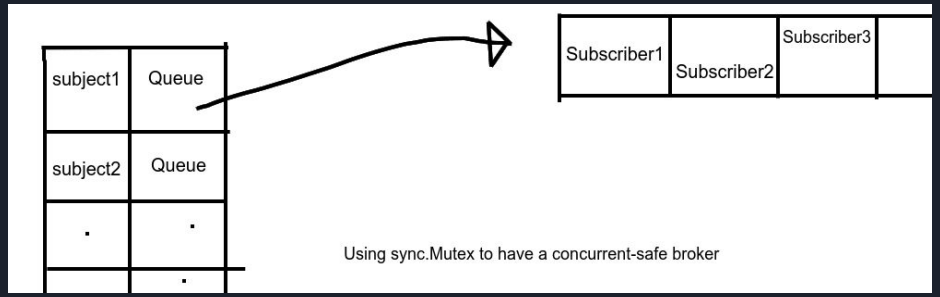
- Project Description
- Implementation Architecture
- gRPC
- Prometheus/Grafana
- Jaeger
- k6/golang client
- Postgres
- Cassandra
- Optimizing
- Envoy-Proxy
- Docker
- Kubernetes

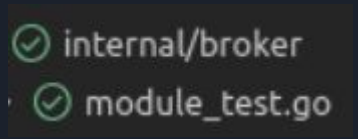
Project Description

- With Exp. Time
- Without Exp. Time

Implementation Architecture

- Could use sync.Map because of its thread-safe operators.
- Use map for broker -> map[subject]Queue
- Each Queue has a list of its subscriber and each subscribe has a channel with type broker.Message for returning messages to users.
- storageType field for using different ways of storing messages.(GOLANG_MAP, POSTGRES, CASSANDRA)
- Use mutex to make the broker concurrent-safe.
- Check expiration of messages in another goroutines not to block the whole broker (using timer.Ticker to wake the blocked goroutine)
- Again for subscribing a subject retrieving messages are being done in another goroutine not to block the functionality of Subscribing.





Challenges in Implementation Architecture

- Delete messages in the given expiration time
- Retrieving messages when new subscriber adds
- Overhead of thinking complex has impact on methods performance. Think Simple!

gRPC

- First of all we had a proto file which we used it to implement the server-side of gRPC (using protoc)
- Proto compiler creates the translation of proto file in the language that we want.
- After that broker.pb.go and broker_grpc.pb.go had been created, I created the RPC of each methods like Publish, Subscribe and Fetch.
- Also at the end to test my gRPC using
- In broker_server.go, I implemented some basic functionality of what each API must have (get requests and convert them to appropriate type and give it to my broker module, and return status code (using Code type in gRPC) and appropriate responses)
- At last I had to register the gRPC server which i had written before.
- And at the end, I tested my gRPC APIs using insomnia.
- No huge challenge just stream type in APIs.

Prometheus\Grafana

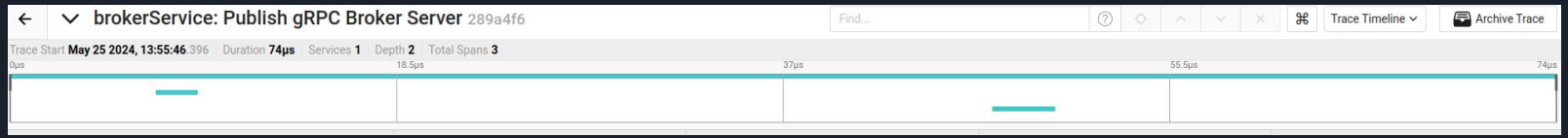
- I defined some metrics for RPC and Env metrics to monitor my broker.’
- Using different types of metrics (Gauge, Counter, Summary, Histogram)
- I had to expose these metrics on a http server from broker on port 9901.
- So I set a prometheus config file with 5s scrape interval
- active_subscriber , memory_usage, cpu_utilization, cpu_load are implemented as Gauge metric.
- GarbageCollection is implemented as Counter, method_duration is implemented as Summary and method_count is implemented with Histogram.
- I have to use some defined metrics in my RPCs like active_subsriber, method_duration and method_count.
- For active_subscriber for example in Subscribe RPC you should Increament the counter and whenever you want to exit the method you should decreament to show the exact number of active subscribers.
- For method_duration and method_count you should use observe method to track the value of in the evaluated elapsed time.
- After that I tested one time in Prometheus UI and then I fixed a data source for monitoring prometheus in grafana and make dashboard.

Challenges in Prometheus/Grafana

- Wrong usage of Counter for method_count
- Problem in pulling prometheus and grafana images based on restriction that dockerhub has put on Iran. so I decided to use Arvan docker registry docker.arvancloud.ir
- Unfamiliar with interacting the prometheus config file with the http server that broker was using it to expose the declared metrics.
- There was a misunderstanding about Env metrics which led me to use node exporter to monitor some factors of my broker app. So there is a little over-engineering through configuring containers and prometheus.yml config file.

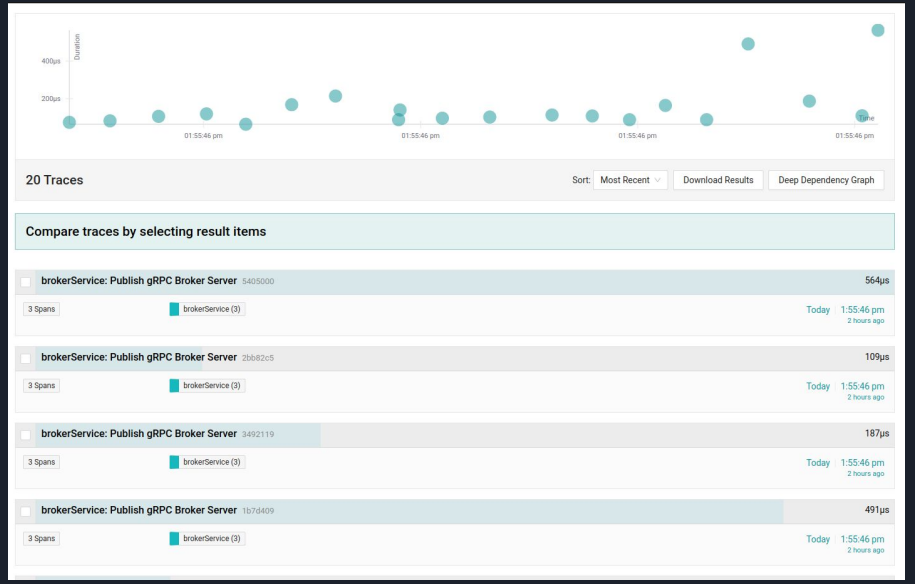
Jaeger

- Usage of jaeger in my broker was tracing deeper through different layer of my app like broker module, database operations and … to find out which part of my app can be a bottleneck.
- First of all I set configuration of jaeger using its host and port. Then create new tracer object.
- You should start a span and give the context of each span to lower layer to figure out which span takes more time to execute.
- After tracer registration, I traced different parts of the broker from the outer to inner layer to find bottlenecks of my app.
- Each part that you start tracing needs a context which comes from the outer layer which is created with ContextWithSpan method which holds a reference to above span.
- For example for publishing new message in the broker module which decides what to do and where to store the message I use 2 span. First one for subscribers to get published message, and second one for the way of storing the published message and both of these 2 spans are related to Publish RPC and can include more span of lower layer like SavingMessageDB.



Challenges in Jaeger

- I wanted to use logrus as the log system of jaeger so i should define an adapter interface to implement the needed method.
- First of all I use for each layer (RPC, module, database) just one span which could not help me to find the bottlenecks properly. After that I decided to separate each layer to smaller fractions to monitor better.



k6/golang client

- First I implement k6 load test to evaluate performance of my broker app which made me to implement it and test it (javascript).
- After that due to being costful I was to write the load test using golang client to make some lighter tests using goroutines.
- I implemented my golang load test using gRPC Client object and use their method to test functionality and monitoring performance of each RPC.
- I used workerpool to test performance of my broker.

Challenges in k6/golang client

- First of all k6 needs a heavy resource so you wouldn't be able to get a huge rate on your machine and it got crashed soon.
- After that you figure out that using k6 is not logical, it made me to write the load test with multiple goroutines which couldn't get the max rate that it should so at the end I implemented the load test using workerpool with less goroutines.
- Despite all these changes, My machine got crashed too many times which forced me to think about hiring a VPS with better config and resources.
- I got a VPS from Arvancloud to test my broker with better resources.
- After each crashing all containers went down and must be configured again.

Postgres

- I stored messages in postgres using raw query.
- I added a removed(bool) column. Whenever I want to delete a record i update it and set the removed flag True and do not remove it completely from my database.
- I used logrus for observing better what happened in my app during the querying database.
- Also I considered and index based on subject and id for retrieving with better performance.
- At the end I defined whenever the app starts running it creates a job to change the flag of removed for those messages which their createdAt + expiration_time greater and equal than current_time.
- Consider that for retrieving messages from Database there is a removed = false in its WHERE clause.
- There is no challenge in using pgx driver.

Cassandra

- Implementation of cassandra is very similar to postgres except the creation of Keyspace is done in the code using raw cql.
- The main challenge of implementing cassandra for persisting messages was knowing concepts of cassandra like keyspace, replicaFactor, consistencyLevel and … .
- What I implemented is single-cluster single-node due to lack of resources.
- What I did here for removing expired messages is exactly like postgres by not removing it completely and just update the removed flag and set it to True whenever it must be expired.

Optimizing

- NOT_PERSISTED
  - What I did about optimizing the performance of the broker, first of all I decided to think simpler to the problem so I decrease the complexity of my code in NOT_PERSISTED mode and also decrease some unnecessary Lock which was not necessary and being thread-safe was not a main factor and you could omit it.
- POSTGRES
  - I was thinking about the whole process which was being done between postgres and broker. And generally about what I could do to decrease factors which had huge impact on make performance and latency worse like RTT and … . What I could do to use batch approach in deleting and inserting message from and to database with a time or count threshold.
- CASSANDRA
  - What I did in cassandra is as like as POSTGRES by using batch approach for some operations like deleting and inserting using gocql.Batch and for both I declared a scheduledBatchOperation which execute queries which are available in batch

Optimizing

CASSANDRA

POSTGRES

NOT_PERSISTED

Challenges in Optimizing

- I had to use all tools that I had configured (jaeger, grafana, prometheus) so the very basic challenge that I had was figuring out where is the boundary of being optimized or not.
- I had to change and rewrite functionality of some operations like add messages, deleting and … to get better result in monitoring and tracing which led to think about implementing using SOLID principles which must consider the idea of Open for Extension and Close for Modification.
- Add some fields to structs of DBs to perform batch approach correctly including thread-safe operations for them.
- I had to run the load test multiple times which increased the probability of getting crashed the bought VPS because of uncertainty about its performance.

Envoy-proxy

- The wanted usage in envoy-proxy was rate limiting, but if had more nodes we could implement load balancing and other features of envoy-proxy.
- I had to set up both listener (downstream) and cluster (upstream) to put the envoy between them and work appropriately.
- After that I declared a route “/” to send requests to <ip_address>:10000 and after that envoy give it to cluster or upstream which we defined its config in cluster section in envoy.yml


```
p_filters:
 name: envoy.filters.http.local_ratelimit
 typed_config:
   "@type": type.googleapis.com/envoy.exte
   stat_prefix: http_local_rate_limiter
   token_bucket:
     max_tokens: 1000
     tokens_per_fill: 800
     fill_interval: 1s
```

Challenges in Envoy-proxy

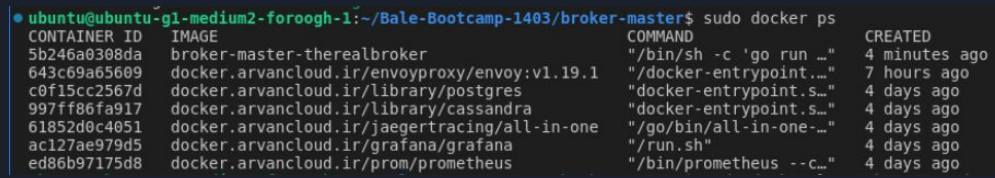
- The most annoying part of setting up the envoy-proxy was interacting with its document which was similar to an unsolvable problem and every new section, I were referred to a new world.
- I had to complete its config step by step and in an incremental approach and I tried and failed so many times by following its documentation but its architecture and concept was too simple the hard part was “Implementation”.
- I had to be careful about its config when i wanted to deploy it on kubernetes or when I should set a docker network to make my containers interact with each other.

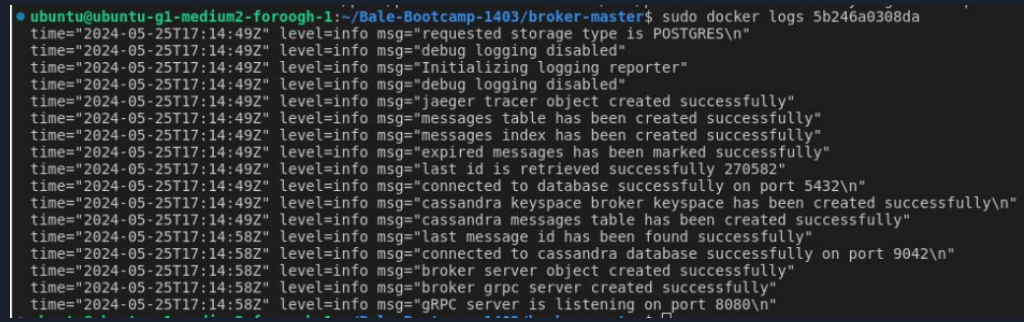
Docker

- For deploying on docker, I created 7 container using docker-compose with arvancloud as my docker registry.
- There's a dependency between broker container and postgres, cassandra, prometheus and jaeger and broker container restarts until there is no failure in running grpc server on the given port.
- For env variables and some configs that might change in the future, I used .env file.
- You could give your env vars using env_file or with this syntax ${field_in_env_file}
- About Broker Image there are some obstacle in building image fluently, like 403 forbidden and old versions of packages. My approach was using `go install -v ./...` but best practice of golang based images are copying go.mod and go.sum first and then `go mod download` which do not download like go install repeatedly until there is a new added package. Also you can use `go mod vendor` to make a directory to store packages directly in your project root and use from it to decrease the overhead of downloading these packages from go proxies.

Challenges in Docker

- Pulling images from docker hub is unavailable in iran and I had to set another DNS or use VPN or I must change the docker registry.
- Go proxies are not available to install packages which you need for your project and it gives you 403 forbidden error due to sanctions.
- Another challenge was when I want to use go mod vendor it may not be successful because of incompatibility between packages.
- The very basic challenge that I had when I was dirting my hand with docker was making containers able to watch each other using docker network.
- Using go install compiles and installs packages but go mod vendor (when you're offline) and go mod download works different by downloading dependencies to the local module cache.





Kubernetes

- First of all, There must be a cluster to be able to run pods on a node in that cluster. For that we could use minikube, kubeadm , kind but i use k3s because it's a lightweight kubernetes and for my condition and resources that is the best tool that I could have.
- I used different types of kind in the process of deploying the broker on k8s. Also each kind has its own fields in k8s API.
- Also I created Stateful pod because of some reasons.
- I used ConfigMap to store some config files (envoy, prometheus) and environments variables. I used PersistenVolume and PersistentVolumeClaim too to provide an abstraction for physical storage resources in a cluster and also for pvc request storage resources which defined by PV.

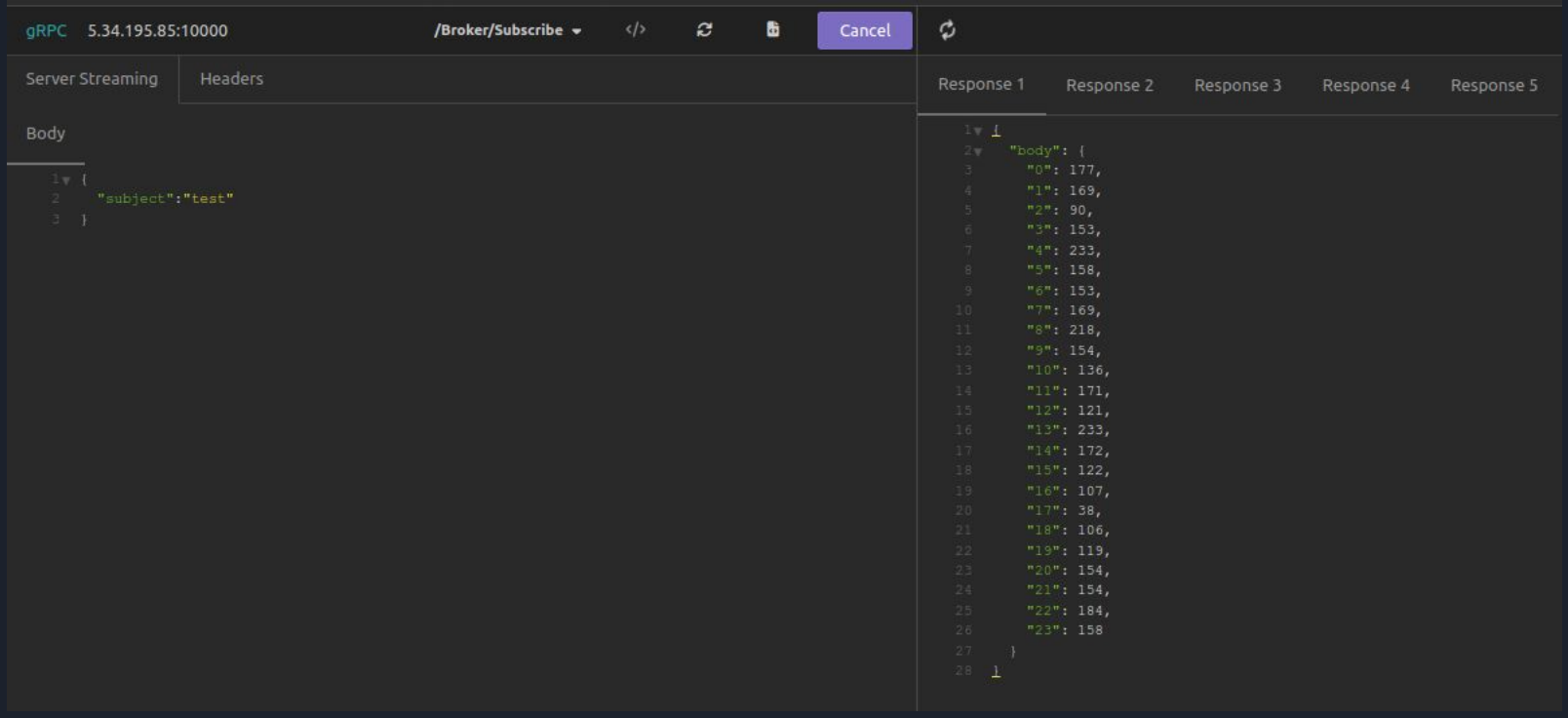
Challenges in Kuberenetes

- Working with kind, minikube, kubeadm was complex and not installable due to the location so I decided to use k3s which is a lightweight kubernetes.
- At first I was using Deployment, and I wanted to give each pod its own storage which is preserved across restarts So I changed it to StatefulSet.
- I had to push the image of broker on a docker registry so I pushed it on docker hub which was not successful and after that I pulled an image of registry and pushed my image on it and in my kuber configuration for the container part i use the image which I had pushed on the registry container on my machine.

```
amirox@amirooox:~$ kubectl get nodes
NAME       STATUS   ROLES                  AGE     VERSION
amirooox   Ready    control-plane,master   3d21h   v1.29.4+k3s1
amirox@amirooox:~$ kubectl get pods --field-selector spec.nodeName=amirooox
NAME                READY   STATUS    RESTARTS        AGE
envoy-0             1/1     Running   2 (28h ago)     2d16h
prometheus-0        1/1     Running   2 (28h ago)     2d18h
grafana-0           1/1     Running   2 (28h ago)     2d19h
cassandra-0         1/1     Running   2 (28h ago)     2d13h
postgres-0          1/1     Running   1 (28h ago)     2d1h
jaeger-0            1/1     Running   2 (28h ago)     2d23h
therealbroker-0     1/1     Running   23 (27h ago)    2d1h
```

```
amirox@amirooox:~$ kubectl get configmap
NAME                   DATA   AGE
kube-root-ca.crt       1      3d21h
postgres-config        6      3d4h
jaeger-config          2      2d23h
grafana-config         2      2d19h
prometheus-config      5      2d18h
cassandra-config       5      2d13h
therealbroker-config   26     2d1h
envoy-config           1      2d16h
```


gRPC 5.34.195.85:10000
/Broker/Publish
Send
0 OK
Unary
Headers
Response 1
Body
{
"subject":"test",
"body":"salamememanamoireemsemsmaneamrie",
"expirationSeconds":"10000"
}
[
"id": 270577
]


gRPC 5.34.195.85:10000
/Broker/Subscribe
Cancel
Server Streaming
Headers
Response 1
Response 2
Response 3
Response 4
Response 5
Body
{
"subject":"test"
}
[
"body": {
"0": 177,
"1": 169,
"2": 90,
"3": 153,
"4": 233,
"5": 158,
"6": 153,
"7": 169,
"8": 218,
"9": 154,
"10": 136,
"11": 171,
"12": 121,
"13": 233,
"14": 172,
"15": 122,
"16": 107,
"17": 38,
"18": 106,
"19": 119,
"20": 154,
"21": 154,
"22": 184,
"23": 158
}
]