Scalable Dialog Subsystem

Purpose:

As a result of H/W, a basic microservice skeleton was created, which will be developed in future h/w.

The project is located here:

https://github.com/maleykovichdim/my_chat_websocket_sharding

Run: docker-compose up --build

localhost:8086

Chat server is implemented in the golang language, based on the gorilla websocket server.

Folder: /chat

Запуск: ./start.sh

Chat Client is desined to testing server operation.

It sends and receives messages.

Folder: /chat_client

Run: go run main.go

Four Mysql servers, run in docker-compose, shard the message table:

```
mysql> show columns from message;
+--------------+--------------+------+-----+-------------------+----------------+
| Field        | Type         | Null | Key | Default           | Extra          |
+--------------+--------------+------+-----+-------------------+----------------+
| id           | int(11)      | NO   | PRI | NULL              | auto_increment |
| id_author    | int(11)      | NO   |     | NULL              |                |
| id_recipient | int(11)      | NO   |     | NULL              |                |
| text         | varchar(255) | NO   |     | NULL              |                |
| time         | timestamp    | NO   |     | CURRENT_TIMESTAMP |                |
+--------------+--------------+------+-----+-------------------+----------------+
5 rows in set (0.00 sec)
```

Run: docker-compose up --build

The message id is currently in each table independently.

In fact, consistent hashing was used according to the derived (calculated) dialog id specified below.

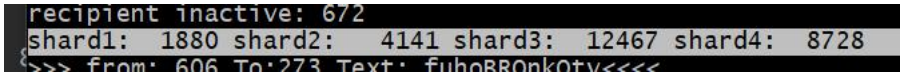
Main ideas:

Messages should be grouped by chat ID and message time (To remove the effect of "Lady Gaga"). In this version of sharding, an artificially derived chat ID was used:

```
var idChat string
if AUTHOR ID < RECIPIENT ID  {
        idChat = AUTHOR ID + "_" + RECIPIENT ID
} else {
        idChat = RECIPIENT ID + "_" + AUTHOR ID
}
```

The collection of author and recipient ids is a monotonic sequence, , so we use a hashed key under the hood. The time for sending messages is also a monotonous sequence, and it would be possible to additionally distribute messages in each additionally created group of shards in according to the border-threshold of the timestamp.

An example of filling shards with messages from a random message flow:



Since sharding itself is carried out at the application level, it accordingly allows the possibility of resharding without downtime:

The GetShardDbBody function returns a pointer to the holder of the shard's database.

The function can be modified to add special conditions.

You can also add new databases to the ring or remove corrupted databases from the context.

In consistent hashing, when adding new nodes, you only shuffle a small portion of the keys.

When a shard is disconnected, it must be removed from the ring and messages will be distributed to other shards. Also, if the shard is very full, then it is worth adding new shards next to the filled ones and taking care of getting messages from the already filled shards on which the old rules, determined by the old set of shards, were in effect. Perhaps one solution is to add new shards with a new timestamp. Or the inclusion of rules related to the timestamp, if requests allow it ...