GraphCRDT

A portable on-memory conflict-free replicated graph database.

Note: This project has already only been tested on macOS Monetary.

This is a simple implementation of a conflict-free replicated graph database. The graph operations have been developed based on the Last-Writer-Wins element set, which contains several essential functions as follow:

- add a vertex/edge
- remove a vertex/edge
- check if a vertex/edge is in the graph
- query all vertices connected to a vertex
- find any path between two vertices
- merge with concurrent changes from other graphs/replicas

This project comes with a full decentralization fashion which can merge data without any coordination between replicas. The core idea here is that each replica can work independently. When the replica connects to the database network, they can merge or receive updates from other replicas via the connection in the network. If a replica wants to join the network, it should be assigned an address and know exactly one friend (replica) in the network. After a replica in the network receives a message that its friend has just registered to the network, it will broadcast information of this newcomer to the whole network. This message will be sent to all replicas since the network is always connected. Likewise, when a replica sends a merge request to its friends, this message will also be sent to all other replicas. The Last-Writer-Wins data type will solve any conflict.

Installation

Requirements:

- Docker version 20.10.6+
- Python 3.8+

Build docker image for the database instance:

```
docker build -t gcrdt .
```

Run the first database instance. It should be noted that the first instance of the network has no friend here, so we set `FRIEND_ADDRESS=-1` :

```
docker run -d --name cluster_1 -p 8081:8000 -e ADDRESS=http://host.docker.internal:8081 \
           -e FRIEND_ADDRESS=-1 gcrdt
```

For example:

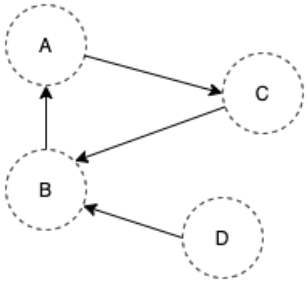
```
docker run -d --name cluster_2 -p 8082:8000 -e ADDRESS=http://host.docker.internal:8082 \
           -e FRIEND_ADDRESS=http://host.docker.internal:8081 gcrdt

docker run -d --name cluster_3 -p 8083:8000 -e ADDRESS=http://host.docker.internal:8083 \
           -e FRIEND_ADDRESS=http://host.docker.internal:8082 gcrdt
```

After executing these commands, **cluster_1**, **cluster_3**, **cluster_3** are connected. We can also run the sample script (provided in the project repository) to have a network with 5 replicas (instances):

```
chmod +x run.sh
./run.sh
```

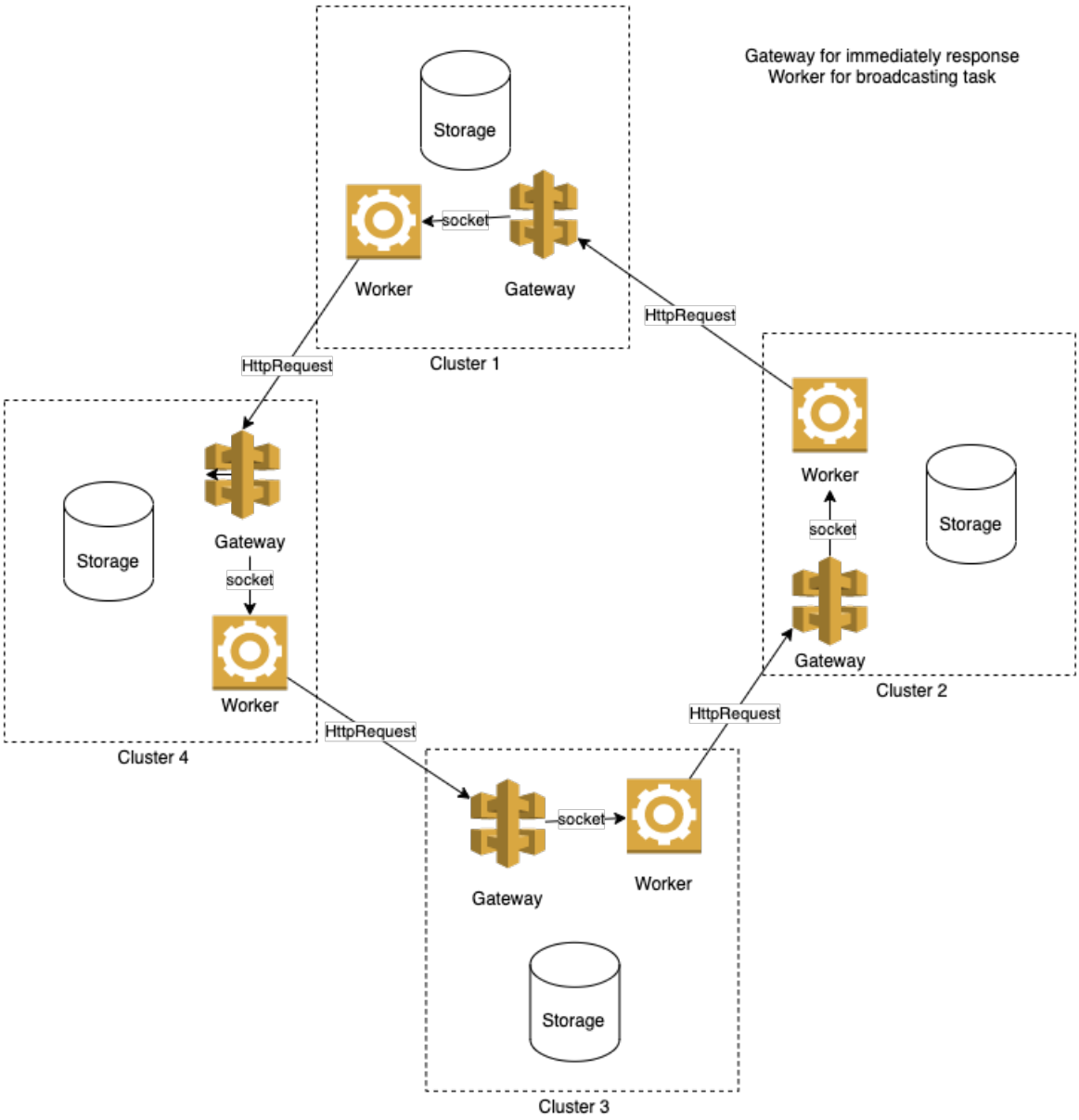
Architecture

In this type of peer-to-peer communication, latency and switching loop (https://en.wikipedia.org/wiki/Switching_loop) is the big problems. As the figure below shows, D lets its friend B know that he wants to join when D connects to the network. Then B will send to its friend A the information of D. Likewise, A will send the information to its friend C, and C is also a friend of B, so that C will send another request to B. In any REST-like communication type, when a service sends a request, it will wait for a response. That is why the switching loop problem arises here: B wait for A to respond, A wait for C, and C is also waiting for B.



There are several ways to deal with this problem:

- Use TTL or Timeout for a request: This is not a good idea since TTL/Timeout will increase the latency of the whole network.
- Use fire-and-forget protocols like Apache Thrift, UDPSocket: The main drawback of these protocols is that we must keep the connection between 2 services if we want them can talk to each other. So it will be another problem with network connection in a large network.
- Use asynchronous message queue like RabbitMQ, Kafka: We can do that, but unfortunately, we don't want any coordination between services, and our network must be fully decentralized among replicas, which means any centralized data will not be accepted.
- Separate architecture into two layers: Yes, at least it works for our case. We can use two layers, the first for communication among networks, and the other is responsible for performing logical queries and broadcasting among replicas. These two layers can be connected by exactly one inner UDPSocket tunnel in the container; this will be more stable than holding a bunch of connections as in the second option. Since the communication gateway will immediately respond to each request, workers will perform other jobs so that we can imagine this as a fire-and-forget gateway.



API Client

Simply install the API client of this project in Python (a PyPi version will come soon):

```
python setup.py install
```

Usage:

- Connect to a database instance:

```
from gcrdt_client import CRDTGraphClient

connection_string = "http://127.0.0.1:8081"
instance = CRDTGraphClient(connection_string)
```

- Clear database (A `broadcast()` request should be sent after any operation in the database instances to keep the data real-time synchronized among replicas. Although, of course, we can also send `broadcast()` request after a list of operations in the database, the rule of Last-Writer-Wins will solve any

conflicts):

```
instance.clear() # clear all edges and vertices from the database
instance.broadcast() # send new update to the network
```

- Add/remove/check a vertex:

```
instance.add_vertex(1)
instance.add_vertex(2)
instance.broadcast()

print(instance.exists_vertex(1)) # True
print(instance.exists_vertex(2)) # True

instance.remove_vertex(1)
instance.remove_vertex(2)
instance.broadcast()

print(instance.exists_vertex(1)) # False
print(instance.exists_vertex(2)) # False
```

- Add/remove/check an edge:

```
instance.add_vertex(1)
instance.add_vertex(2)
instance.add_edge(1, 2)
instance.broadcast()

print(instance.exists_edge(1, 2)) # True

instance.remove_edge(1, 2)
instance.broadcast()

print(instance.exists_edge(1, 2)) # False
```

- Find path between two vertices:

```
instance.add_vertex(1)
instance.add_vertex(2)
instance.add_vertex(3)
instance.add_edge(1, 2)
instance.broadcast()

print(instance.find_path(1, 2)) # [1, 2]

instance.add_edge(1, 3)
print(instance.find_path(2, 3)) # [2, 1, 3]
```

- Query all vertices connected to a vertex:

```
instance.add_vertex(1)
instance.add_vertex(2)
instance.add_vertex(3)
instance.add_edge(1, 2)
instance.add_edge(1, 3)
instance.broadcast()

print(instance.get_neighbors(1)) # [2, 3] the returned result is on the sorted order
```

Apart from the client API in Python, we can also directly use REST API of the instance gateway to perform queries. Check it out at `{host_name}:{port}/redoc` or `{host_name}/{port}/docs` (For example: http://127.0.0.1:8081/redoc or http://127.0.0.1:8081/docs)

Testing

This project is also provided some pre-defined tests to validate our system. For example, to test the functionalities of a database instance, start an instance with the

listing port `8081` first.

```
docker run -d --name cluster_1 -p 8081:8000 -e ADDRESS=http://host.docker.internal:8081 \
           -e FRIEND_ADDRESS=-1 gcrdt
```

Then run the unit test as below:

```
python -m unittest test/unit.py
```

If you want to check the consistency between replicas, we start 5 instances and then run another test as below:

```
./run.sh # to start 5 database instances

python -m unittest test/integration.py
```

Sample testcase:

```
import unittest
from gcrdt_client import CRDTGraphClient

class CRDTGraphIntegrationTestCase(unittest.TestCase):
    cluster_1 = "http://127.0.0.1:8081"
    cluster_2 = "http://127.0.0.1:8082"
    cluster_3 = "http://127.0.0.1:8083"
    cluster_4 = "http://127.0.0.1:8084"
    cluster_5 = "http://127.0.0.1:8085"

    def test_get_neighbors_and_find_paths(self):
        instance_1 = CRDTGraphClient(CRDTGraphIntegrationTestCase.cluster_1)
        instance_2 = CRDTGraphClient(CRDTGraphIntegrationTestCase.cluster_2)
        instance_3 = CRDTGraphClient(CRDTGraphIntegrationTestCase.cluster_3)
        instance_4 = CRDTGraphClient(CRDTGraphIntegrationTestCase.cluster_4)
        instance_5 = CRDTGraphClient(CRDTGraphIntegrationTestCase.cluster_5)

        self.assertTrue(instance_1.clear())
        self.assertEqual(instance_1.broadcast(), "Success")

        self.assertEqual(instance_1.add_vertex(1)["status"], "Success")
        self.assertEqual(instance_2.add_vertex(2)["status"], "Success")
        self.assertEqual(instance_3.add_vertex(3)["status"], "Success")

        self.assertEqual(instance_1.broadcast(), "Success")
        self.assertEqual(instance_2.broadcast(), "Success")
        self.assertEqual(instance_3.broadcast(), "Success")

        self.assertEqual(instance_1.add_edge(1, 2)["status"], "Success")
        self.assertEqual(instance_1.broadcast(), "Success")

        self.assertEqual(instance_2.find_path(1, 2)["data"], [1, 2])
        self.assertEqual(instance_3.add_edge(1, 3)["status"], "Success")
        self.assertEqual(instance_3.broadcast(), "Success")

        self.assertEqual(instance_4.find_path(2, 3)["data"], [2, 1, 3])
        self.assertEqual(instance_5.get_neighbors(1), [2, 3])
        self.assertEqual(instance_5.add_vertex(4)["status"], "Success")
        self.assertEqual(instance_5.broadcast(), "Success")

        self.assertEqual(instance_1.find_path(1, 4)["status"], "Error")
```

Improvement directions

- Back up data on disk to preserve the data even when all database instances is crashed.
- Add authentication and authorization layer to the gateway.
- Reduce network latency and broadcasting operations. We can develop a smarter routing algorithm to reduce the number of broadcasting operations. Our

main objective is to minimize the longest path between any replica in the network, but we also don't want to keep too many connections (edge) in the network. That is why we need a "smart" routing algorithm.