Report

Link to github repository of solution:

https://github.com/frejabv/Exam

I Freja Bruun Vanggaard (frev) hereby declare that this submission was created in its entirety by me and only me. I have used parts of the code from Mini projects 2 and 3, on which I collaborated with Jacob Møller Jensen (jacj) and Jeppe Korgaard (korg).

Multiple choice

1. Assume that a blockchain network has been attacked, and that at least 40% of the nodes have Byzantine faults. What is correct

   i. **The network consistency is compromised since the Byzantine generals problem cannot be solved**

2. The *Token Ring* distributed mutual exclusion algorithm on N Nodes

   ii. **has a continuos bandwidth consumption**

3. The happened before relation

   iii. **is transitive**

4. In a network (I assume this question is supposed to be: In a network that is synchronous)

   ii. **the propagation delay can never be bounded**

5. A design goal for Microservices is

   iii. **loosely coupled, but use the same underlying technology stack**

6. We definitely have a deadlock when

   i. **a process P1 has a lock on a resource A while waiting to get a lock on another resource B; and, at the same time a process P2 has a lock on resource B while waiting to get a lock on resource A**

7. The TCP protocol

   ii. **guarantees that messages are delivered in the order they were sent**

8. In the Raft consensus algorithm

   iii. **the leader uses the heartbeat for notifying it is still alive and for sending updates about the log**

9. Sequential Consistency does not imply that

   i. **requests from all clients are processed in the same order that they were received**

10. An operation is idempotent when

    i. **applying it several consecutive times yields the same result**

Implementation discussion

Starting the system

Navigate to the folder Exam.

And run:

```
go run main.go
```

You need to start a total of 3 servers, so you need to write the above command in three separate terminals. When starting the servers you need to provide the ports as prompted. For the first server write 0 and press enter, for the second write 1 and press enter, for the third write 2 and press enter.

Then run:

```
go run frontend.go
```

To start 2 frontends. You need to write the above command in two separate terminals. In the first terminal write 0 and press enter. In the second terminal write 1 and press enter.

Then run:

```
go run client.go
```

To start a client. This will give you a message of the kind of inputs you can provide.

To put - write put followed by the key and value like:

```
put 4 5
```

To get - write get followed by the key like so:

```
get 4
```

It is only possible to access keys from 0-99.

All outputs from put and get are displayed in the log.txt file.

1. What system model are you assuming in your implementation. Write a full description.

I assume a system model with asynchronous network and crash-failures. This means that there is no upper bound on delays and nodes can fail with crash-failures. (I do not assume recovery but this has been somewhat implemented for servers in my solution)

2. What is the minimal number of nodes in your system to fullfill the requirements? Why?

I have chosen to do active replication, which means my system includes a client, 2 frontends and 3 servers. A total of 6 nodes. I use the words server and replica interchangeably to refer to the same thing.

In active replication the frontend receives calls to Put and Get from the client and forwards these requests to the servers. In active replication the frontend has more responsibility since it has to know of all the servers, and in my case is also responsible for doing some computations.

The server ports are hardcoded into the frontend, and as such rely on being initiated correctly as described in Starting the system. The same goes for the frontends.

I have 2 frontends to tolerate the crash-failure of 1 frontend. And 3 replicas to tolerate the crash-failure of 1 replica and still maintain a majority in a quorum.

If a replica crashes the frontend will keep sending requests to all replicas, but will ignore the answers of failed replicas. It will check to see if a majority of replicas have performed the put request correctly, which will still hold with the crash-failure of 1 replica. It will check all responses from get requests and only accept the one with the highest lamport timestamp, ensuring we get the most recent entry. So as long as 1 replica is a alive, the get requests will succeed. It does not make a big difference when during a request the replica crashes, as its answer will be ignored because it is not able to return anything valid.

If a frontend crashes the client should register it, by pinging the frontend in intervals, and if there is no response, it should connect to the other frontend ports that it knows of. Which means the client will again be able to connect to a frontend which will forward the put or get request to the servers.

3. Explain how your system recovers from crash failure.

If a replica crashes nothing is done, since there will still be a majority of replicas left for the frontend to access the system can continue as it did.

As described above the failure of a frontend will be registered by the client, when there is no response on the ping/heartbeat. Then the client will connect to another frontend and the system will continue as previously.

4. Explain how you achieve the Property 1 requirement.

(Property 1 - Repeatable Read) if a call *put(key,val)* is made, with no subsequent calls to put, then a new call *get(key)* to that node will return *val*.

In the absence of failures the call to put will return true, and as such confirm that the value has been persisted on all servers. Hence a get operation since it queries all replicas, will return the new value.

In the presence of a crash failure, if a replica crashes during put, its answer will be ignored since a majority, 2 replicas, still return answers the put will have gone through, and a subsequent get will return this value.

In the presence of a crash failure, if a frontend crashes before put is sent to the servers, the client will be alerted because the ping did not succeed, and connect to another frontend to retry.

In the presence of a crash failure, if a frontend crashes after put is sent to the servers, but before the reply is sent to the client, the client will be alerted because the ping did not succeed, and connect to another frontend to retry. They will not be alerted to the success of the put, but if they get on the key it will return the previously put value. If they retry with a different value, this will be what get returns.

If the call to put did not succeed, the user will be alerted in the log, and see that a retry is necessary.

5. Explain how you achieve the Property 2 requirement.

(Property 2 - Initialization to zero) if no call to *put* has been made for a given key, any *get* call returns 0 (zero).

I have decided to make the datastructure of the hash table a slice of type int, and initialized it with a length of 100. In GO this means every value in the slice is initialized to 0, hence querying a key, corresponding to an index in the slice that has not yet been overwritten by a put operation, will return 0.

6. Explain how you achieve the Property 3 requirement.

(Property 3 - Consistency) given a non-empty sequence of successful (returning **true**) put operations performed on the hash table, any subsequent get operation at any node should return the same value

Since i use active replication we then have to get/read on a quorum of nodes to see ensure we are reading the latest added value in case some of the nodes are lagging behind. In a system with 3 nodes, we need 2 nodes to agree, which is also why we can handle the failure of 1 node, and still ensure this property.

I use Lamport timestamps to make sure i return the most recently added value. This is done in the frontend, which after receiving a call to get from the client, forwards this request to all 3 replicas. If one of the replicas have crashed, the answer will be ignored. The Lamport timestamp of the responses from all 3 is then compared to see who has the highest. The value from the replica with the highest Lamport timestamp is the value that will be returned to the client. If several replicas have the same Lamport timestamp, they should also have the same value and this is disregarded. If some of the replicas do not have the same value as the replica with the highest

Lamport timestamp, a put request is sent to them to update their value on the key for the get request.

This means that if a replica comes back to life, its values will be updated by seeing that they don’t correspond to that of the other servers. However it is only the single value at the index of the key from the get request that will be updated. The hash table will therefore not be completely consistent, if several replicas fail and come back to life.

7. Explain how you achieve the Property 4 - Liveness requirement.

(Property 4 - Liveness) every call to *put* and *get* returns a value. If a call *put(k,v)* returns **false**, then some call in a repeated sequence of *put(k,v)* will return **true** (eventually, *put* and *get* succeed).

If all replicas are alive then operations will successfully return on having been distributed to all replicas.

If 1 replica crashes then put operations to that individual replica will return false. But we will still have a quorum of replicas left which can accept the requests, since we only have to tolerate the crash-failure of 1 replica. Therefore we can still achieve a majority of successes, and so in the end return true.

If a frontend crashes, the request will return false. The client will then connect to another frontend, and the next requests will return true. Since we only have to tolerate the crash-failure of 1 node some call will eventually return true.

8. Explain how you achieve the Property 5 - Reliability requirement.

(Property 5 - Reliability) your system can tolerate a crash-failure of 1 node.

Since we need a quorum and we start out with 3 replicas, we can tolerate one replica failing and still be up and running, since the majority required is 2. We can also tolerate 1 frontend failing as the client knows of 1 other frontend it can connect to instead.

It is described in the previous sections how the system handles crashes of replicas and frontends if they happen during put and get, and the different phases in the execution of the those operations.