Tey Shi Ying (1003829)

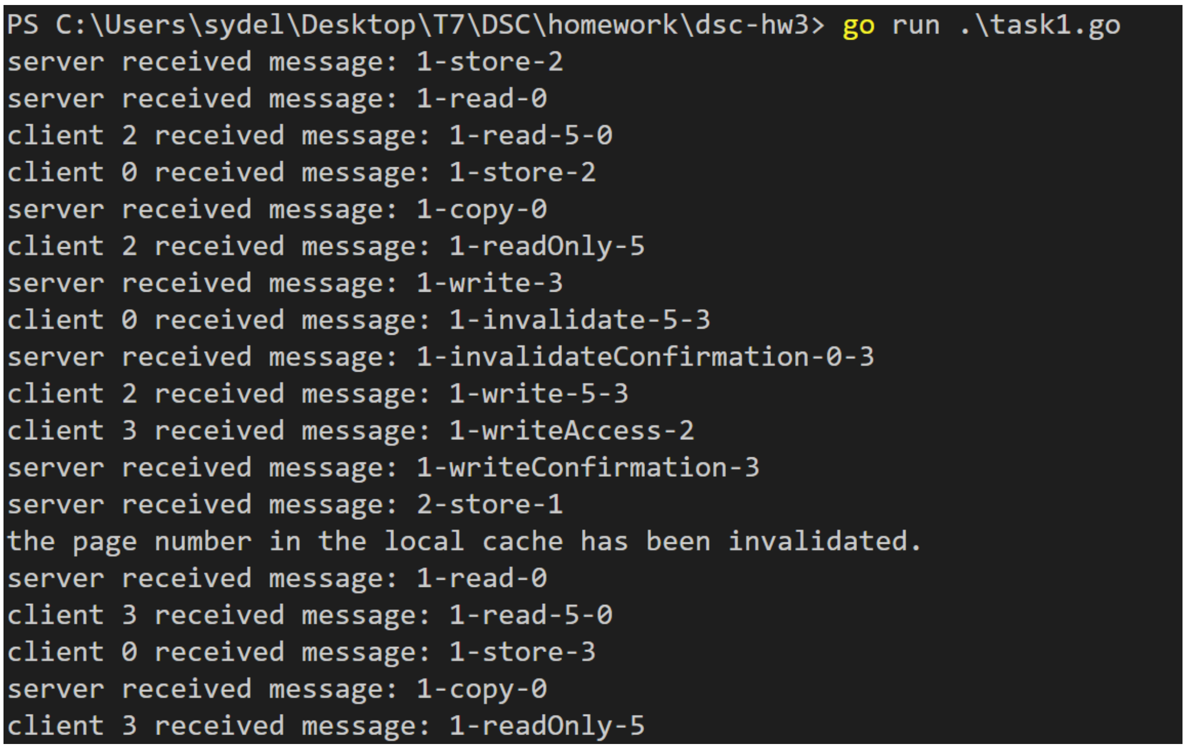
DSC PA3

Task1: Implement Ivy Architecture

The messages are written in the format : <page_number,action,sendingClientId,requestClientId>, where actions are explained in the comments for the select cases in client function and CM function respectively, the sendingClientId refers to the id of the client who sent the current message, requestClientId refers to the id of the client who initiated the sequence – for example line 9 invalidateConfirmation has requestClientId 3 which corresponds to the third test case where "node 3 issues a write request to page_number 1".

The following shows the output for test case where

1. node 2 inserts page_number 1 corresponds to print_lines 1
2. node 0 issues a read request of page_number 1 to server corresponds to print lines 2-6
3. node 3 issues a write request to page_number 1 to server corresponds to print lines 7-12
4. node 1 inserts page_number 2 corresponds to print lines 13
5. node 0 issues a readRequest to page number 1 corresponds to print lines 14-19



```
PS C:\Users\sydel\Desktop\T7\DSC\homework\dsc-hw3> go run .\task1.go
server received message: 1-store-2
server received message: 1-read-0
client 2 received message: 1-read-5-0
client 0 received message: 1-store-2
server received message: 1-copy-0
client 2 received message: 1-readOnly-5
server received message: 1-write-3
client 0 received message: 1-invalidate-5-3
server received message: 1-invalidateConfirmation-0-3
client 2 received message: 1-write-5-3
client 3 received message: 1-writeAccess-2
server received message: 1-writeConfirmation-3
server received message: 2-store-1
the page number in the local cache has been invalidated.
server received message: 1-read-0
client 3 received message: 1-read-5-0
client 0 received message: 1-store-3
server received message: 1-copy-0
client 3 received message: 1-readOnly-5
```

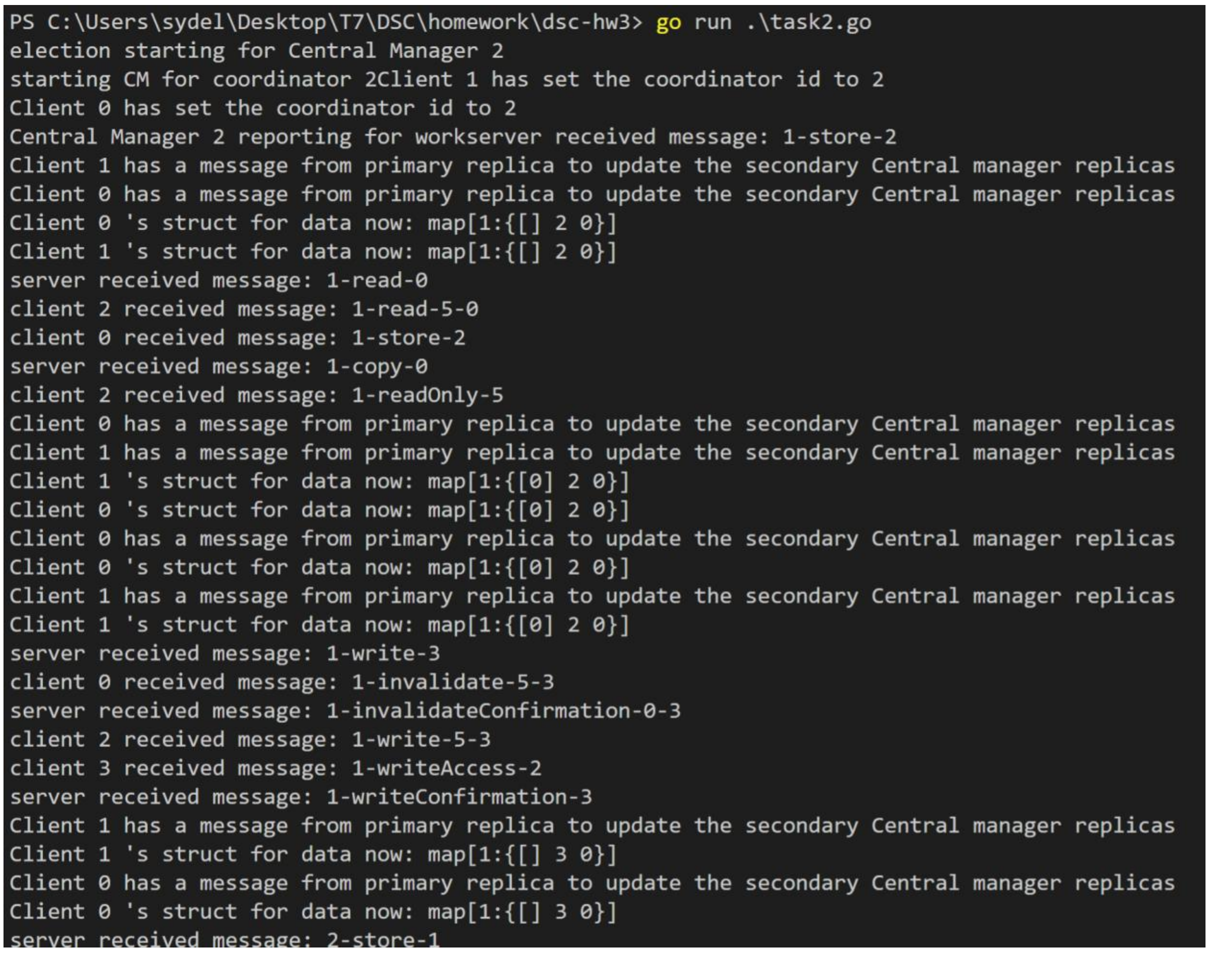
To get the above output,

1. Change the package to main in task1.go
2. In terminal type, go run task1.go

Task2: Fault Tolerant Ivy Architecture with Bully Algorithm

I implement fault tolerance for the central managers by selecting primary central manager via bully algorithm's election process. This central manager would start a goroutine `CentralManager.CM()` to read client requests via server channel that would be held by the primary channel. This go routine `CentralManager.CM()` will come with a kill_node channel in order to stop the goroutine when central manager is supposed to be dead. Also this goroutine which would spin up another goroutine `broadcastDataCM()` which routinely sends the updated map of CentralManagerRecords to all other replicas of central managers which would update their own CentralManagerRecords.

First, we test the `broadcastDataCM()` in a non-fault environment. Below, we can see the map of CentralManagerRecords for replica central manager 0 and 1, where client 0's request to read page 1 has updated the copyset of the page in the primary central manager and has been replicated onto the map of CentralManagerRecords for replica central manager 0 and 1. This can be seen from line 16-17. Similar is the case when node 3's request to write to page 1 changed the copyset and the owner for page 1. In a non-fault environment.

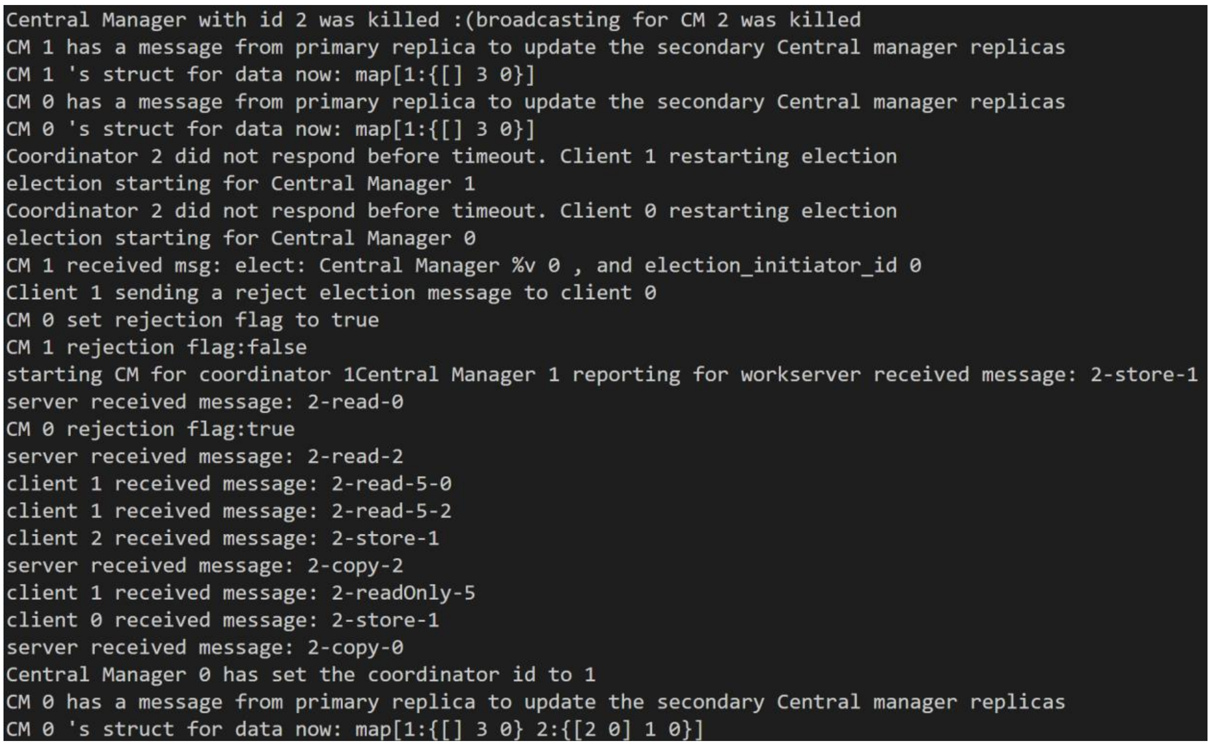


```
PS C:\Users\sydel\Desktop\T7\DSC\homework\dsc-hw3> go run .\task2.go
election starting for Central Manager 2
starting CM for coordinator 2Client 1 has set the coordinator id to 2
Client 0 has set the coordinator id to 2
Central Manager 2 reporting for workserver received message: 1-store-2
Client 1 has a message from primary replica to update the secondary Central manager replicas
Client 0 has a message from primary replica to update the secondary Central manager replicas
Client 0 's struct for data now: map[1:{[] 2 0}]
Client 1 's struct for data now: map[1:{[] 2 0}]
server received message: 1-read-0
client 2 received message: 1-read-5-0
client 0 received message: 1-store-2
server received message: 1-copy-0
client 2 received message: 1-readOnly-5
Client 0 has a message from primary replica to update the secondary Central manager replicas
Client 1 has a message from primary replica to update the secondary Central manager replicas
Client 1 's struct for data now: map[1:{[0] 2 0}]
Client 0 's struct for data now: map[1:{[0] 2 0}]
Client 0 has a message from primary replica to update the secondary Central manager replicas
Client 0 's struct for data now: map[1:{[0] 2 0}]
Client 1 has a message from primary replica to update the secondary Central manager replicas
Client 1 's struct for data now: map[1:{[0] 2 0}]
server received message: 1-write-3
client 0 received message: 1-invalidate-5-3
server received message: 1-invalidateConfirmation-0-3
client 2 received message: 1-write-5-3
client 3 received message: 1-writeAccess-2
server received message: 1-writeConfirmation-3
Client 1 has a message from primary replica to update the secondary Central manager replicas
Client 1 's struct for data now: map[1:{[] 3 0}]
Client 0 has a message from primary replica to update the secondary Central manager replicas
Client 0 's struct for data now: map[1:{[] 3 0}]
server received message: 2-store-1
```

To get the above output,

1. Change the package to main2 in task1.go
2. Change the package to main in task2.go
3. In terminal type, go run task2.go

Below, we shows the print logs for a fault tolerant. When central manager with highest id, id 2 was killed, the election starts for central managers with id 0 and 1. Central manager with id 1 was elected. It starts the goroutine to access the stored buffer for the server to process the requests which also starts the goroutine to broadcast the data of map[pageNumber]<copyset,ownerId,invalidateRequest> to all the replica central managers – which is this case is central manager 0. As we can see in the last line, the data struct of the replica node central manager 0 has been changed.



```
Central Manager with id 2 was killed :(broadcasting for CM 2 was killed
CM 1 has a message from primary replica to update the secondary Central manager replicas
CM 1 's struct for data now: map[1:{[] 3 0}]
CM 0 has a message from primary replica to update the secondary Central manager replicas
CM 0 's struct for data now: map[1:{[] 3 0}]
Coordinator 2 did not respond before timeout. Client 1 restarting election
election starting for Central Manager 1
Coordinator 2 did not respond before timeout. Client 0 restarting election
election starting for Central Manager 0
CM 1 received msg: elect: Central Manager %v 0 , and election_initiator_id 0
Client 1 sending a reject election message to client 0
CM 0 set rejection flag to true
CM 1 rejection flag:false
starting CM for coordinator 1Central Manager 1 reporting for workserver received message: 2-store-1
server received message: 2-read-0
CM 0 rejection flag:true
server received message: 2-read-2
client 1 received message: 2-read-5-0
client 1 received message: 2-read-5-2
client 2 received message: 2-store-1
server received message: 2-copy-2
client 1 received message: 2-readOnly-5
client 0 received message: 2-store-1
server received message: 2-copy-0
Central Manager 0 has set the coordinator id to 1
CM 0 has a message from primary replica to update the secondary Central manager replicas
CM 0 's struct for data now: map[1:{[] 3 0} 2:{[2 0] 1 0}]
```

To get the above output,

1. Change the package to main2 in task2.go
2. Change the package to main in task3.go
3. In terminal type, go run task3.go

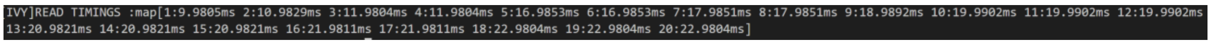
Task3: Does New Design Still Preserve Sequential Consistency

Yes, I think the new design preserves sequential consistency between primary central manager and client. When the primary central manager goes down, requests to the central manager are blocked while the election process for bully algorithm is in place. After the coordinator is elected, the coordinator spins up a goroutine to connect to the channel that connects with the client nodes. The coordinator is then able to process the requests of the client via FIFO. Hence, new design still preserves sequential Consistency.

Experiment1: Compare Ivy to Fault Tolerant

For this experiment, I first made a store page request from a random client and then ran 20 read requests for this page by random clients. Below are the timings for each read request where key-value <x:y> represents: at y time there are x requests reached . Because of goroutines the timing will be different with each time you run the code.

IVY PERFROMANCE

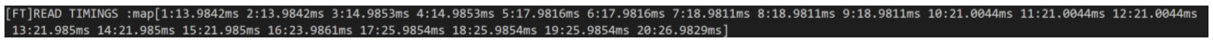


```
IVY]READ TIMINGS :map[1:9.9805ms 2:10.9829ms 3:11.9804ms 4:11.9804ms 5:16.9853ms 6:16.9853ms 7:17.9851ms 8:17.9851ms 9:18.9892ms 10:19.9902ms 11:19.9902ms 12:19.9902ms
13:20.9821ms 14:20.9821ms 15:20.9821ms 16:21.9811ms 17:21.9811ms 18:22.9804ms 19:22.9804ms 20:22.9804ms]
```

To get the above output,

1. Change the package to main2 in task3.go
2. Change the package to main in task4.go
3. In terminal type, go run task4.go

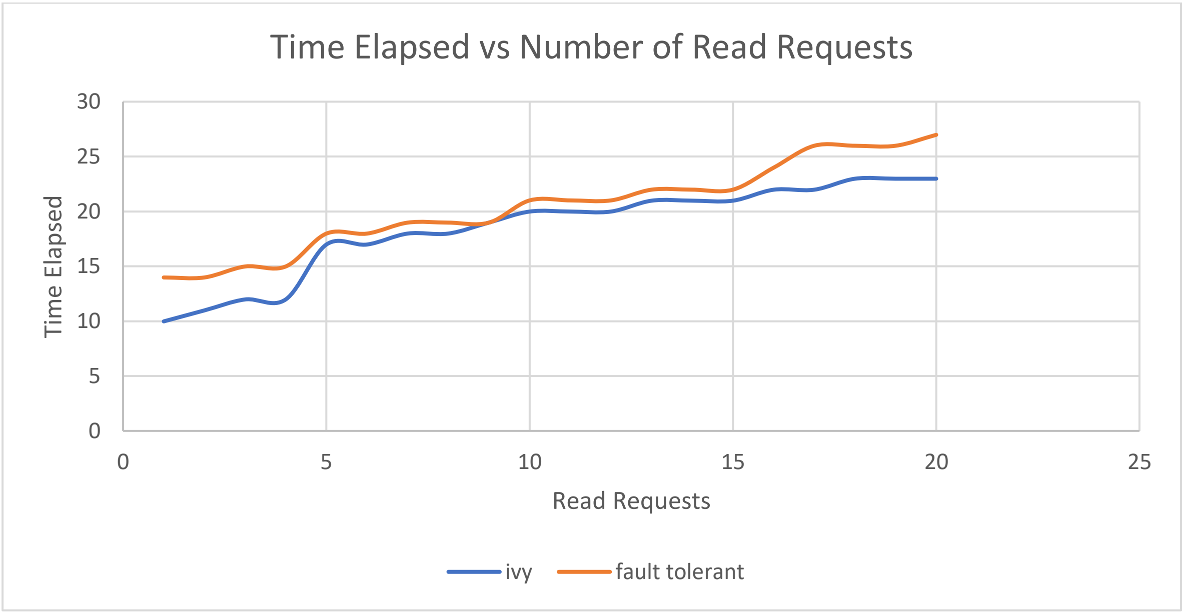
FAULT TOLERANT PERFORMANCE



```
[FT]READ TIMINGS :map[1:13.9842ms 2:13.9842ms 3:14.9853ms 4:14.9853ms 5:17.9816ms 6:17.9816ms 7:18.9811ms 8:18.9811ms 9:18.9811ms 10:21.0044ms 11:21.0044ms 12:21.0044ms
 13:21.985ms 14:21.985ms 15:21.985ms 16:23.9861ms 17:25.9854ms 18:25.9854ms 19:25.9854ms 20:26.9829ms]
```

To get the above output,

1. Change the package to main2 in task4.go
2. Change the package to main in task5.go
3. In terminal type, go run task5.go



By observation, fault tolerant implementation takes a longer time for the same number of read requests. This is probably because there is a performance bottleneck on the primary central manager as the central manager also has to send out map[pageNumber]centralManagerRecord to all the other central manager replicas.