

DistributedSystemsPSETS50.041

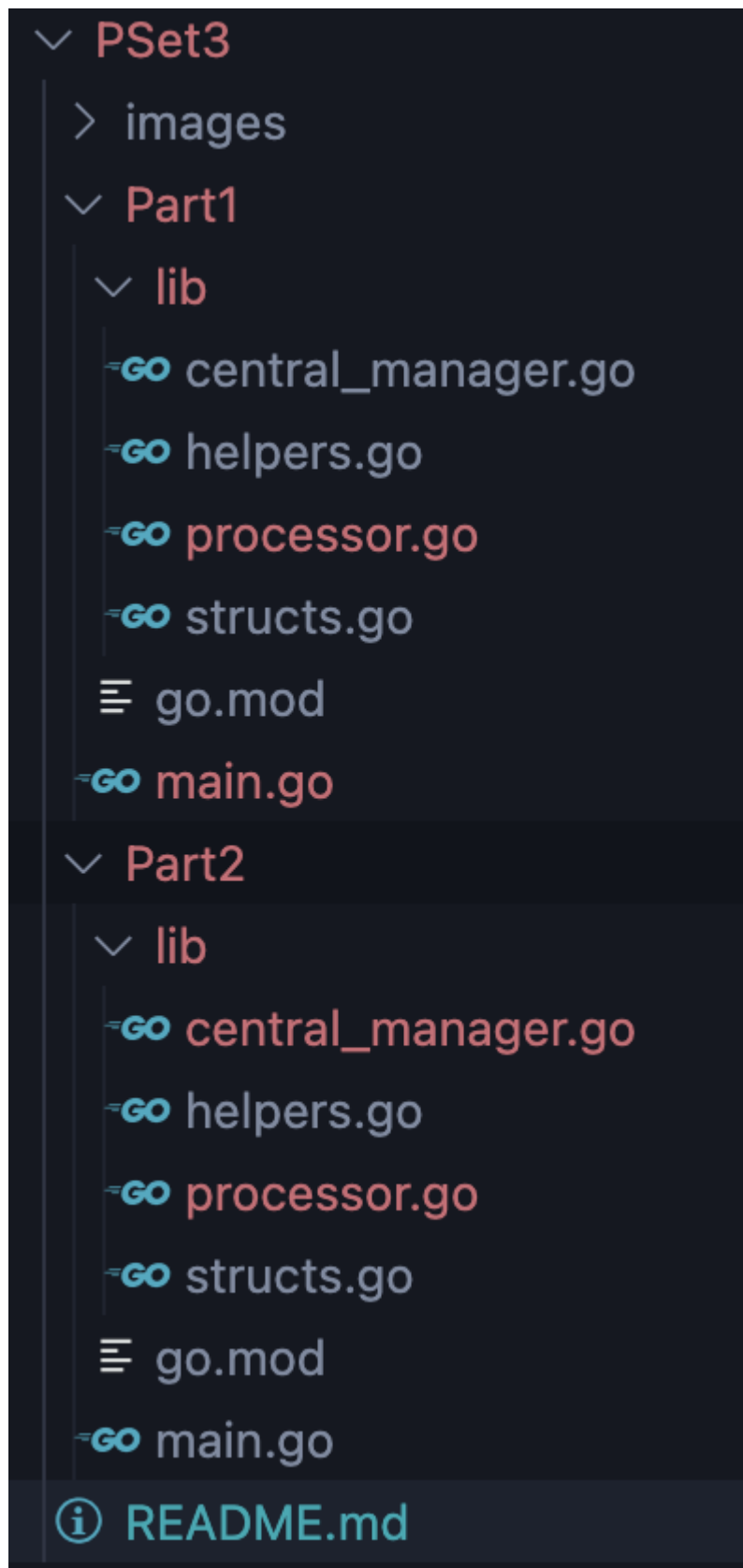
Assignment 3

Toh Kai Feng

1004581

Introduction

This is the following file structure used:



1. To run non-fault tolerant version (Part 1):

```
cd Part1
go run -race main.go
```

2. To run fault tolerant version (Part 2):

```
cd Part2
go run -race main.go
```

Part 1 Basic Ivy Protocol

This would be the output when part 1 is ran:

```
3: WRITE_REQUEST request (1) sent for page Id 1
8: cache for pageId 1 invalidated
7: cache for pageId 1 invalidated
CM: updated entries map to map[0:{{[] 5}} 1:{{[] 3}} 2:{{[] 5}} 3:{{[7 5 2] 6}}]
8: WRITE_REQUEST request (1) sent for page Id 3
7: cache for pageId 3 invalidated
5: cache for pageId 3 invalidated
2: cache for pageId 3 invalidated
6: cache for pageId 3 invalidated
CM: updated entries map to map[0:{{[] 5}} 1:{{[] 3}} 2:{{[] 5}} 3:{{[] 8}}]
5: WRITE_REQUEST request (1) sent for page Id 3
8: cache for pageId 3 invalidated
CM: updated entries map to map[0:{{[] 5}} 1:{{[] 3}} 2:{{[] 5}} 3:{{[] 5}}]
```

What we are looking for is that:

1. When a write request is made by processor A for pageId X,
2. All entries in the copyset, and entries held by the owner are invalidated.
3. Central Manager updates Processor A as the new owner.

In the picture above,

Processor A making write request: 8

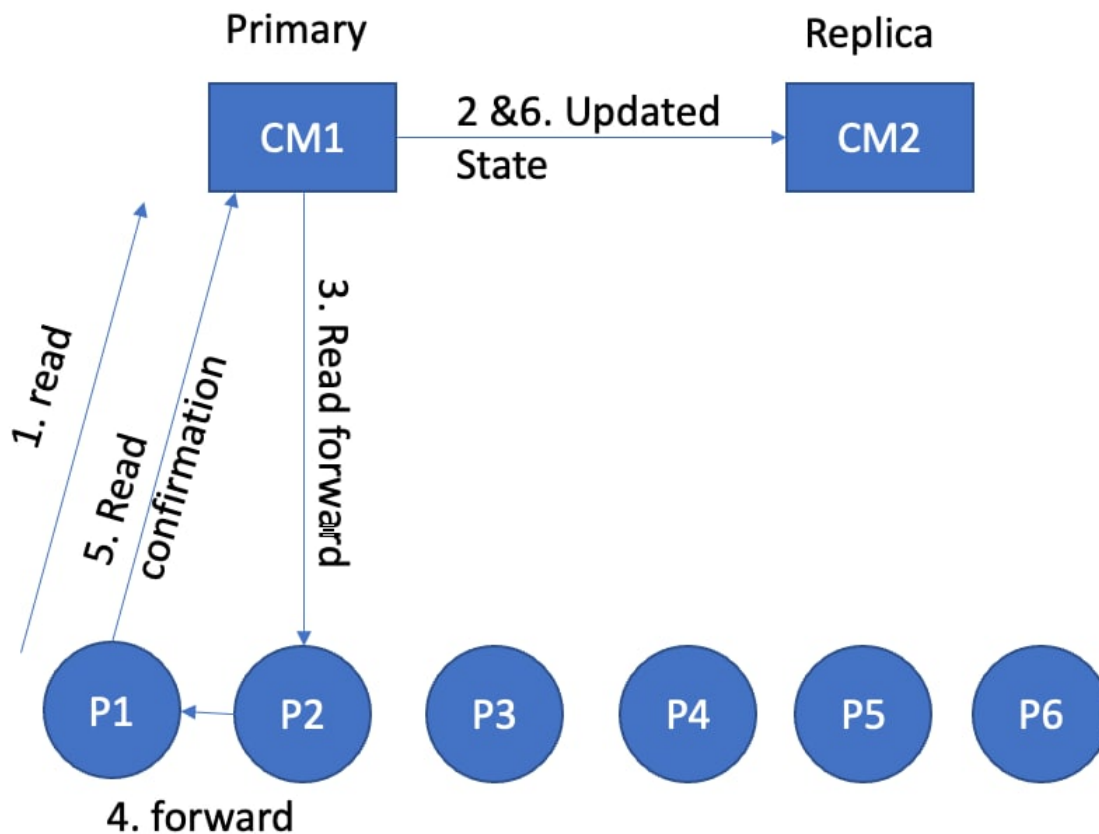
Processors in Copyset: [7,5,2]

Processor previously owner of page: 6

Part 2 Fault Tolerant Ivy Protocol

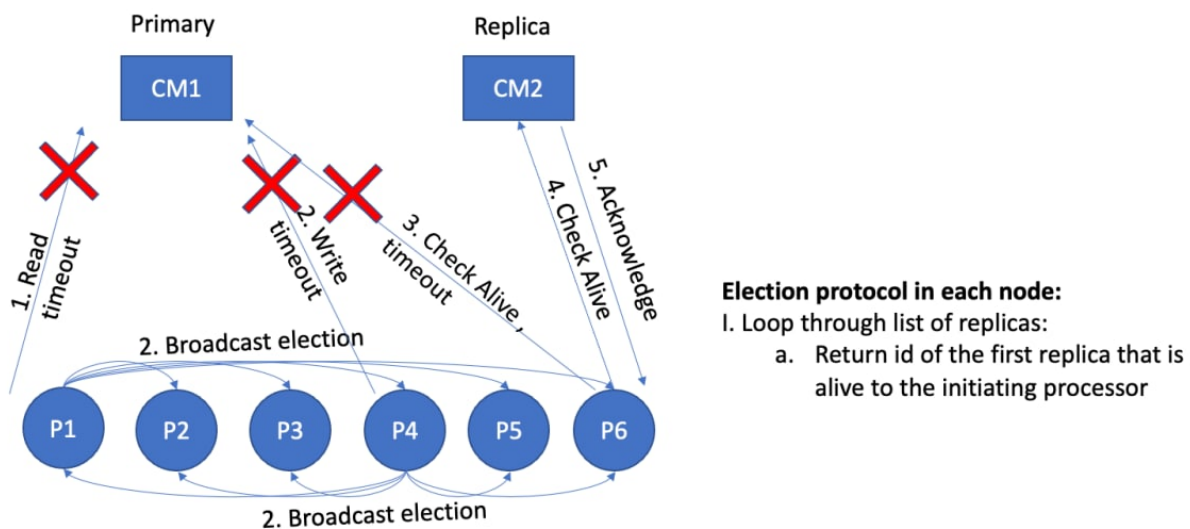
2.1 Primary and Secondary Replicas with consistency

Every time a Primary CM handles a message from a processor, the CM would forward it's state of Entries, Queued requests, and Invalidation Progress to the Secondary replica.



2.2 Election to choose Replica

The Election mechanism can be visualised in the following diagram:



- Processors 1 and 4 concurrently realise a timeout has been reached for their request.
- Processors 1 and 4 broadcast to all other processors that election is in progress so that all nodes can stop their requests and so that the processor with the highest ID can run the election protocol.
- Processor 6 sends a CHECK_ALIVE message to all central managers, and waits for a duration before electing the central manager with the highest ID.

4. NOT SHOWN IN DIAGRAM: once a CM 2 has been elected, CM 2 announces to all Processors that it is the new primary replica, and continues the message handling.

This election trigger in part 2 can be seen in this output:

```
CM 0: request map before removing 7: map[0:{{0 0 {0 0 0 0 []}} []} 1:{{0 1 {0 0 0 0 []}} [{7 1 1 7 []}]} 2:{{0 0 {0 0 0 0 []}} []} 3:{{0 0 {0 0 0 0 []}} []}]
CM 0: request map: map[0:{{0 0 {0 0 0 0 []}} []} 1:{{0 0 {0 0 0 0 []}} []} 2:{{0 0 {0 0 0 0 []}} []} 3:{{0 0 {0 0 0 0 []}} []}]
CM 0: updated entries map to map[0:{{7 8} 4} 1:{{[] 7} 2:{{4} 7} 3:{{[] 6}}]
5: READ_REQUEST request (0) sent for page Id 2
CM 0: updated entry map map[0:{{7 8} 4} 1:{{[] 7} 2:{{4 5} 7} 3:{{[] 6}}]
CM 0: dead
CM 0: DIED -- Time Elapsed: 2738 ms
2: WRITE_REQUEST request (1) sent for page Id 2
4: READ_REQUEST request (0) sent for page Id 3
0: WRITE_REQUEST request (1) sent for page Id 2
3: READ_REQUEST request (0) sent for page Id 1
6: WRITE_REQUEST request (1) sent for page Id 2
4: READ_REQUEST request (0) sent for page Id 1
8: READ_REQUEST request (0) sent for page Id 3
9: READ_REQUEST request (0) sent for page Id 0
7: READ_REQUEST request (0) sent for page Id 3
7: WRITE_REQUEST request (1) sent for page Id 0
5: WRITE_REQUEST request (1) sent for page Id 3
8: WRITE_REQUEST request (1) sent for page Id 0
1: READ_REQUEST request (0) sent for page Id 0
4: WRITE_REQUEST request (1) sent for page Id 2
0: READ_REQUEST request (0) sent for page Id 3
6: WRITE_REQUEST request (1) sent for page Id 0
3: WRITE_REQUEST request (1) sent for page Id 2
8: READ_REQUEST request (0) sent for page Id 2
5: WRITE_REQUEST request (1) sent for page Id 0
2: WRITE_REQUEST request (1) sent for page Id 0
8: WRITE_REQUEST request (1) sent for page Id 1
1: WRITE_REQUEST request (1) sent for page Id 3
0: WRITE_REQUEST request (1) sent for page Id 1
6: READ_REQUEST request (0) sent for page Id 1
2: WRITE_REQUEST request (1) sent for page Id 3
5: WRITE_REQUEST request (1) sent for page Id 2
9: WRITE_REQUEST request (1) sent for page Id 2
1: READ_REQUEST request (0) sent for page Id 1
4: timeout for pageId: 3, operation: READ_REQUEST
4: broadcasting election
4: starting election
8: starting election
```


In the picture above, we can see that processors are still making their requests despite CM 0 dying. Since each request has a timeout to trigger an election, an election would eventually occur which causes the processor experiencing the timeout to broadcast to all other processors to enter election mode.

The election results look like this:

```
7: READ_REQUEST request (0) sent for page Id 0
0: WRITE_REQUEST request (1) sent for page Id 1
7: READ_REQUEST request (0) sent for page Id 1
5: WRITE_REQUEST request (1) sent for page Id 2
9: WRITE_REQUEST request (1) sent for page Id 2
1: READ_REQUEST request (0) sent for page Id 1
2: timeout for pageId: 2, operation: WRITE_REQUEST
2: broadcasting election
2: starting election
9: starting election
1: starting election
3: timeout for pageId: 3, operation: READ_REQUEST
6: starting election
7: starting election
3: broadcasting election
8: starting election
3: starting election
0: starting election
4: starting election
5: starting election
9: CM 1 alive, chosen
9: CM 2 alive, chosen
9: CM 3 alive, chosen
9: new Cm elected is CM 1
CM 1: elected as new primary
2: cache for pageId 2 invalidated
5: cache for pageId 2 invalidated
8: cache for pageId 2 invalidated
CM 1: request map before removing 0: map[0:{{0 0 {0 0 0 0 []}} [{6 1 0 6 []} {
1 0 0 1 []} {5 1 0 5 []} {3 1 0 3 []} {7 0 0 7 []} {9 0 0 9 []} {8 1 0 8 []}]}
1:{{0 0 {0 0 0 0 []}} [{1 0 1 1 []} {2 0 1 2 []} {6 1 1 6 []} {0 1 1 0 []} {4
0 1 4 []} {7 0 1 7 []}]} 2:{{0 1 {0 0 0 0 []}} [{0 1 2 0 []} {5 1 2 5 []} {2
1 2 2 []} {4 1 2 4 []} {6 0 2 6 []} {7 1 2 7 []} {3 0 2 3 []} {9 1 2 9 []}]} 3
:{{0 0 {0 0 0 0 []}} [{5 1 3 5 []} {0 0 3 0 []} {2 1 3 2 []} {1 1 3 1 []} {6 0
3 6 []} {3 0 3 3 []} {8 0 3 8 []}]}]
CM 1: request map: map[0:{{0 0 {0 0 0 0 []}} [{6 1 0 6 []} {1 0 0 1 []} {5 1 0
```

2.3 Changes to Ivy Protocol

The following are the few changes made to the IVY protocol:

1. Timeouts for messages that are sent to the CM.

```
func (p *Processor) HandleTimeout() {  
    /**  
    after a timeout, check for requests that are NOT IDLE, and have exceeded specified timeout duration  
    if request is not Idle and exceeded timeout duration ⇒ start election  
    */  
  
    if p.InElection {  
        // check for acknowledgement messages and elect  
        p.ElectNewCM()  
        return  
    }  
    // p.InElection = true  
  
    for key := range p.RequestMap {  
        requestState := p.RequestMap[key]  
        if requestState.State == IDLE {  
            // Don't check if request is in idle mode (means the request has completed)  
            continue  
        }  
  
        //Check whether : currentTime ≥ requestTimestamp + timeoutDuration  
        if time.Now().UnixNano() ≥ requestState.Timestamp+int64(p.TimeoutDur*int(time.Second)) {  
            p.log(false, "timeout for pageId: %v, operation: %v", requestState.Message.PageId,  
                requestState.Message.Type.toString())  
            p.StartElection()  
        }  
    }  
}
```

2. Forwarding of state of CM to all CM replicas.

```

38 func (cm *CentralManager) Start() {
39     cm.log(true, "starting %v", "test")
40     cm.Die = make(chan int, 20)
41     startTime := time.Now().UnixNano()
42
43     // ticker to make regular requests
44     for {
45         select {
46             case <-cm.Die:
47                 cm.log(false, "dead")
48                 cm.log(false, "DIED -- Time Elapsed: %v ms", float32((time.Now().UnixNano()-startTime)/int64
49                     (time.Millisecond)))
50                 return
51             case m := <-cm.Incoming:
52                 cm.EnqueueRequest(m)
53                 cm.ForwardState() ←
54
55             case m := <-cm.ConfirmationChan:
56                 cm.log(true, "%v message (%v) from %v for pageId %v", m.Type.toString(), m.Type, m.Sender, m.
57                     PageId)
58                 cm.HandleMessage(m)
59                 cm.ForwardState() ←
60
61             default:
62                 if !cm.IsPrimary {
63                     break
64                 }
65
66                 pageId := cm.LongestQueue()
67                 if pageId == -1 {
68                     break
69                 }
70                 cm.log(true, "pageId: %v, requestMap: %v", pageId, cm.CurrentState.RequestMap[pageId].Queue)
71                 queuedMessage := cm.CurrentState.RequestMap[pageId].Queue[0]
72                 cm.HandleMessage(queuedMessage)
73                 cm.ForwardState() ←
74         }
75     }
76 }

```

3. Election stage to elect a new CM during a request Timeout. (as shown in part 2.1)

Part 3 Experiments

For my experiment, I test the time taken for 10000 messages to be handled by the CM. The 10000 messages are all random read or write messages with a random delay (0 ~ 2 seconds).

I also used the following parameters for the comparison:

1. Number of Processors: 10
2. Number of Pagelds: 4

Performance of Basic Ivy Protocol


```
CM: updated entries map to map[0:{{[] 2}} 1:{{[5 9] 0}} 2:{{[] 5}} 3:{{[] 7}}]
1: READ_REQUEST request (0) sent for page Id 2
CM: updated entry map map[0:{{[] 2}} 1:{{[5 9] 0}} 2:{{[1] 5}} 3:{{[] 7}}]
5: WRITE_REQUEST request (1) sent for page Id 1
9: cache for pageId 1 invalidated
0: cache for pageId 1 invalidated
CM: DIED -- Time Elapsed: 236934 ms
6: READ_REQUEST request (0) sent for page Id 3
7: READ_REQUEST request (0) sent for page Id 2
5: WRITE_REQUEST request (1) sent for page Id 0
```

CM: DIED -- Time Elapsed: 236934 ms

Average Time taken per request = 2.370 ms

Performance of Fault Tolerant Ivy with 1 fault

When CM Fails at a random time

```
98% done cm 1: Counter:100 cm 1:
D -- Time Elapsed: 305482 ms
```

cm 0: DIED -- Time Elapsed: 155452

ms

cm 1: DIED -- Time Elapsed: 305482 ms

Average Time taken per request = 3.054 ms

When CM Fails and wakes back up

```
CM 0: request map: map[0:{{0 0 0 0 0 0}} 1:{{0 0 0 0 0 0}} 2:{{0 0 0 0 0 0}} 3:{{0 0 0 0 0 0}}]
CM 0: updated entries map to map[0:{{[1] 6}} 1:{{[0] 3}} 2:{{[] 9}} 3:{{[5 4] 1}}]
CM 0: DIED -- Time Elapsed: 59744 ms
6: READ_REQUEST request (0) sent for page Id 2
0: READ_REQUEST request (0) sent for page Id 0
```

CM 0: DIED -- Time Elapsed: 59744 ms

Death rate: 1 in 1000

Average Time take per request = 5.97 ms

Performance of Fault Tolerant Ivy with 3 faults.

```
cm 3: Counter:400
cm 3: Counter:300
cm 3: Counter:200
cm 3: Counter:100
cm 3: Counter:0
cm 3: DIED -- Time Elapsed: 758445 ms
```

cm 0: DIED --

Time Elapsed: 194195 ms

cm 1: DIED -- Time Elapsed: 382273 ms

cm 2: DIED -- Time Elapsed: 564827 ms

cm 3: DIED -- Time Elapsed: 758445 ms

Death rate: 1 in 2500

Average Time take per request = 7.58 ms

Evaluation

For the performance of the Fault Tolerant Ivy with multiple faults, I believe the significant increase in delay is due to the higher overhead in having multiple central managers.