Go Assignment 1

50.041 Distributed Systems and Computing Go Assignment 1
Name: Guo Yuchen
Student ID: 1004885

Question 1

Compilation

For quetsion 1 and 2:
To build:

```
go build .\lamport.go
```

To execute：

```
.\lamport.exe -clients=?
```

For quetsion 3: To build:

```
go build .\vector.go
```

To execute：

```
.\vector.exe -clients=?
```

- `-clients` : `int` , indicates the number of clients. The default number is 15.

External package

`log` : used for output logging and debugging purpose.

Implementation

1. In the program, Message is defined to be a struct with 3 fields:

- `sender_id` : `int` , the id of the sender client/server
- `message` : `int` , the i-th message being sent by a client
- `clock` : `int` for lamport clock, `[]int` for vector clock

2. Clients send a message every 5 seconds.
3. There's a 50-50 chance of dropping or forwarding received message for the server. (flag can be 0 or 1, if 0, forward message)
4. Clock is updated every time an action (i.e. receiving message, send message) is performed.
5. In `vector.go` , causality violation is checked every time a message is received (`clockIsGreaterThan(c1, c2)`) If the local vector clock of the receiving machine is more than the vector clock of the message, then a potential causality violation is detected.

Output interpretation

The `logs.txt` in each folder contains the sample outputs with 15 clients. Refer to the files for more logs.

Lamport

```
 08:59:22 ===============START===============
08:59:22 Client 7's clock:  1.
08:59:22 Client 7 sending the 0-th message at clock 1.
08:59:22 Server received Client 7's 0-th message of clock 1.
08:59:22 Server's clock: 2.
08:59:22 Server forward to Client 0 Client 7's 0-th message.
...
08:59:22 Client 3 sending the 0-th message at clock 1.
08:59:22 Client 0 received Server's message. Sent from Client 7 at clock 2.
...
```

Vector

```
09:03:31 ===============START===============
09:03:31 Client 7's clock: [0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0].
...
09:03:31 Client 3 sending the 0-th message at clock [0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0].
09:03:31 Client 9's clock: [0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0].
09:03:31 Client 9 sending the 0-th message at clock [0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0].
...
09:03:31 Client 6 sending the 0-th message at clock [0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0].
09:03:31 Client 14's clock: [0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0].
09:03:31 Client 7 sending the 0-th message at clock [0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0].
...
09:03:31 Server received Client 3's 0-th message of clock [0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0].
09:03:31 Server's clock: [0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1].
09:03:31 Drop Client 3's 0-th message of clock [0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0].
...
09:03:31 Server's clock: [0 0 0 1 0 0 1 1 0 1 1 1 0 0 0 6].
09:03:31 Server forward to Client 0 Client 10's 0-th message.
09:03:31 Server forward to Client 1 Client 10's 0-th message.
...
09:03:31 Server forward to Client 14 Client 0's 1-th message.
09:03:31 Client 13 received Server's message. Sent from Client 0 at clock [10 8 5 1 6 5 1 1 2 1 1 1 4 3 6 16].
...
09:03:31 Client 14 received Server's message. Sent from Client 0 at clock [10 8 5 1 6 5 1 1 2 1 1 1 4 3 6 16].
09:03:31 Potential violation detected!
09:03:31 Local clock: [10 8 5 1 6 5 1 1 2 1 1 1 4 8 6 16].
09:03:31 Message clock: [10 8 5 1 6 5 1 1 2 1 1 1 4 3 6 16].
...
```

Others

Assumptions

1. Network is reliable.
2. Network is asynchoronous.
3. Channels won't congest. (But it actually happens when node number increases.)

Question 2

Compilation

To build:

```
go build .\run.go .\common.go .\logger.go .\message.go .\node.go
```

To execute：

```
.\run.exe -nodes=? -sync=? -timeout=? -failcoor=? -failworker=? -failcoorvic=? -failworkervic=?
```

- -nodes `int` number of nodes (default 3)
- -sync `int` the time interval to sync in second (default 1)
- -timeout `int` 2T(m) + T(p) in second (default 6)
- -failcoor `bool` if set to true, simulate coordinator fails (default FALSE)
- -failcoorvic `bool` if set to true, simulate newly elected coordinator fails while announcing (default FALSE)
- -failworker `bool` if set to true, simulate coordinator fails (default FALSE)
- -failworkervic `bool` if set to true, simulate worker node fails while announcing (default FALSE)

External package

`log` : used for output logging and debugging purpose.

Implementation

Message

Message is defined to be a struct with 4 fields:

- `sender_id` : `int` , the id of the sender machine
- `message_type` : `string` , the type of the message, including:
  - `SYNC` Coordinator sync data with worker.
  - `ELECT` A node elect itself to be the new coordinator.
  - `ACK` A node refuse one's self-elect request.
  - `VICTORY` A node broadcast that it is the now coordinator.
  - `STOP` To stop a goroutine.
- `message` : `any` , usually a sentence about the content, which is readable by human.
- `timestamp` : `time.Time` , the physical clock of the message. currently not used in the program.

Node

Each node has a state, a role, 4 channels.

- Role:
  - `WORKER`
  - `COORDINATOR`
- State:
  - `NORMAL` node performs normal work, not during election
  - `SELF_ELECTING` node is sending self-electing messages
  - `BROADCATING` new coordinator broadcasting victory
  - `DOWN` node fails
- Channels:
  - `ch_sync` for SYNC message
  - `ch_elect` for ELECT, ACK, VICTORY message
  - `ch_stop_elect` for STOP mesage to stop self_elect() or broadcast_victory()
  - `ch_role_switch` for STOP mesage to switch between `COORDINATOR` and `WORKER`

Synchonization and Election

- **Synchonization**
  If Coordinator is alive, it sends SYNC message at certain interval to Worker. If Worker receives SYNC message, update its data; if not for a timeout, start a election to elect itself as the Coordinator.
- **Election by bully algorithm**
  A node sends ELECT message to nodes with higher ids. If receives ACK or STOP, go back to NORMAL state and stops election. If not for a certain time, broadcasts its victory to all. After broadcasting, switch role to Coordinator and start sending SYNC messages.

Failure Implementation

Set node state to `DOWN` and stop preocessing received message,sending message, and electing. It can still receive messages but won't take care of it.

For implementation details, refer to the code.

Output

The `logs.txt` in each folder contains the sample outputs with 3 nodes, and the coordinator fails after 4 seconds.

Simulation

1. Multiple GO routines / Best case / Worst case

These conditions are implemented inside the system. If a worker hasn't received SYNC message from the Coordinator after timeout, it will start an election no matter whether there's any ongoing election or not. This election, if fails, will be terminated by receiving either ACK or STOP message eventually.

The following is part of a sample log in `logs_coor_fails.txt` for 4 nodes with Coordinator fails:

```
 15:51:13 run.go:55: ===============START===============
15:51:13 run.go:22: Node 0 starts to execute worker_tasks.
15:51:13 run.go:22: Node 2 starts to execute worker_tasks.
15:51:13 run.go:22: Node 1 starts to execute worker_tasks.
15:51:13 run.go:32: Node 3 starts to execute coordinator_tasks.
15:51:13 node.go:341: Coordinator 3 sent a sync message {}.
15:51:13 node.go:153: Node 2 received a message {}.
15:51:13 node.go:153: Node 0 received a message {}.
15:51:13 node.go:153: Node 1 received a message {}.
...
15:51:17 node.go:131: Node 3 failed.
15:51:22 node.go:178: Node 2 detected coordinator 3 failed.
15:51:22 node.go:47: Node 2 started an election.
15:51:22 node.go:65: Node 2 sent elect msg to 3.
15:51:22 node.go:178: Node 0 detected coordinator 3 failed.
15:51:22 node.go:47: Node 0 started an election.
15:51:22 node.go:65: Node 0 sent elect msg to 1.
15:51:22 node.go:65: Node 0 sent elect msg to 2.
15:51:22 node.go:65: Node 0 sent elect msg to 3.
15:51:22 node.go:178: Node 1 detected coordinator 3 failed.
15:51:22 node.go:47: Node 1 started an election.
15:51:22 node.go:200: Node 1 received a message Elect 0 as new coordinator.
15:51:22 node.go:232: Node 1 sent ACK to node 0.
15:51:22 node.go:200: Node 2 received a message Elect 0 as new coordinator.
15:51:22 node.go:200: Node 0 received a message Request from node 0 is refused by node 1.
15:51:28 node.go:84: Node 0 Stop the goroutine broadcast_victory.
15:51:28 node.go:200: Node 0 received a message Request from node 0 is refused by node 2.
15:51:28 node.go:200: Node 0 received a message Node 2 is the new coordinator
15:51:28 node.go:232: Node 2 sent ACK to node 0.
15:51:28 node.go:101: Node 2 is broadcasting a victory to 0.
15:51:28 node.go:200: Node 2 received a message Elect 1 as new coordinator.
15:51:28 node.go:65: Node 1 sent elect msg to 2.
15:51:28 node.go:65: Node 1 sent elect msg to 3.
15:51:28 node.go:200: Node 1 received a message Node 2 is the new coordinator
15:51:28 node.go:101: Node 2 is broadcasting a victory to 1.
15:51:28 node.go:101: Node 2 is broadcasting a victory to 3.
15:51:28 node.go:119: Node 2 sent role switch messgae.
15:51:28 node.go:200: Node 2 received a message Stop the goroutine
15:51:28 node.go:153: Node 2 received a message Stop the goroutine.
15:51:28 run.go:47: Node 2 switch role.
15:51:28 run.go:32: Node 2 starts to execute coordinator_tasks.
15:51:28 node.go:341: Coordinator 2 sent a sync message {}.
15:51:28 node.go:153: Node 1 received a message {}.
15:51:28 node.go:153: Node 0 received a message {}.
...
```

TL;DR for above log
Normal data sync between node 1,2,3->
Coordinator fails ->
Node 0 and 1 both detect failure and start election ->
Node 1 refuse Node 0's request ->
Node 0 ends election ->
Node 1 won election and become coordinator ->
Normal data sync between node 1,2

2. Coordinator silently leaves the network.

Set `failcoor= true` when execute the program. This will trigger the following goroutine in `run.go`:

```
    // Simulate coordinator failure after 4 seconds
    if *fail_coordinator {
        go func() {
            time.Sleep(4 * time.Second)
            for i := range nodes {
                if nodes[i].role == COORDINATOR {
                    nodes[i].fail()
                    break
                }
            }
        }()
    }
```

For the output, please refer to the previous case.

3. Worker silently leaves the network.

Set `failworker=true` when execute the program. This will trigger the following goroutine in `run.go`:

```
    // Simulate worker failure after 4 seconds
    if *fail_worker {
        go func() {
            time.Sleep(4 * time.Second)
            for {
                random_node_idx := rand.Intn(len(nodes))
                if nodes[random_node_idx].role == WORKER {
                    nodes[random_node_idx].fail()
                    return
                }
            }
        }()
    }
```

The following is a sample log in `logs_worker_fails.txt` for 4 nodes with a random worker fails:

```
 15:53:49 run.go:55: ===============START===============
15:53:49 run.go:22: Node 1 starts to execute worker_tasks.
15:53:49 run.go:22: Node 0 starts to execute worker_tasks.
15:53:49 run.go:22: Node 2 starts to execute worker_tasks.
15:53:49 run.go:32: Node 3 starts to execute coordinator_tasks.
15:53:49 node.go:153: Node 0 received a message {}.
15:53:49 node.go:153: Node 1 received a message {}.
15:53:49 node.go:341: Coordinator 3 sent a sync message {}.
15:53:49 node.go:153: Node 2 received a message {}.
15:53:50 node.go:341: Coordinator 3 sent a sync message {}.
15:53:50 node.go:153: Node 0 received a message {}.
15:53:50 node.go:153: Node 1 received a message {}.
15:53:50 node.go:153: Node 2 received a message {}.
```

4. The newly elected coordinator fails while announcing.

Set `failcoorvic=true` and `fail_coordinator=true` when execute the program, i.e. run command `.\run.exe -nodes=4 -sync=1 -timeout=6 -failcoor=true -failcoorvic=true`. This will trigger the following goroutine in `run.go`:

```
    // Simulate coordinator candicate (newly seleted coordinator) failure duing broadcasting
    if *fail_coor_during_broadcasting {
        go func() {
            time.Sleep(4 * time.Second)
            for {
                for i := range nodes {
                    if nodes[i].state == BROADCATING {
                        nodes[i].coordinator_fail_during_broadcasting()
                        return
                    }
                }
            }
        }()
    }
```

The following is part of a sample log in `logs_coor_fail_announcing.txt.txt` of 4 nodes:

```
22:59:48 run.go:61: ===============START===============
22:59:48 run.go:22: Node 0 starts to execute worker_tasks.
22:59:48 run.go:22: Node 2 starts to execute worker_tasks.
22:59:48 run.go:32: Node 3 starts to execute coordinator_tasks.
22:59:48 node.go:170: Node 0 received a message {}.
22:59:48 run.go:22: Node 1 starts to execute worker_tasks.
22:59:48 node.go:170: Node 1 received a message {}.
22:59:48 node.go:358: Coordinator 3 sent a sync message {}.
22:59:48 node.go:170: Node 2 received a message {}.
...
23:00:03 node.go:101: Node 2 is broadcasting a victory to 0.
23:00:03 node.go:65: Node 1 sent elect msg to 3.
23:00:03 node.go:65: Node 1 sent elect msg to 3.
23:00:03 node.go:101: Node 2 is broadcasting a victory to 1.
23:00:03 node.go:217: Node 1 received a message Node 2 is the new coordinator
23:00:03 node.go:101: Node 2 is broadcasting a victory to 3.
23:00:03 node.go:123: Node 2 fails during broadcasting.
23:00:09 node.go:195: Node 0 detected coordinator 2 failed.
23:00:09 node.go:47: Node 0 started an election.
...
23:00:15 run.go:53: Node 1 switch role.
23:00:15 run.go:32: Node 1 starts to execute coordinator_tasks.
23:00:15 node.go:170: Node 1 received a message Stop the goroutine.
23:00:15 node.go:358: Coordinator 1 sent a sync message {}.
23:00:15 node.go:170: Node 0 received a message {}.
```

Notice that this situation does not always happen due to racing between broadcasting and failure notification.

5. A node The failed node that is not the newly elected coordinator fails while announcing.

Set `failworkervic=true` and `fail_coordinator=true` when execute the program, i.e., run command `.\run.exe -nodes=3 -sync=1 -timeout=6 -failcoor=true -failworkervic=true`.

This will trigger the following goroutine in `run.go`:

```
// Simulate non-coordinator failure duing someone else broadcasting
    if *fail_worker_during_broadcasting {
        go func() {
            time.Sleep(4 * time.Second)
            for {
                for i := range nodes {
                    if nodes[i].state == BROADCATING {
                        for {
                            random_node_idx := rand.Intn(len(nodes))
                            if random_node_idx != i && nodes[random_node_idx].state != DOWN {
                                nodes[random_node_idx].worker_fail_during_broadcasting()
                                return
                            }
                        }
                    }
                }
            }
        }()
    }
```

The following is part of a sample log in `logs_worker_fail_announcing.txt` for 3 nodes:

```
 20:41:26 run.go:55: ===============START===============
 20:41:26 run.go:22: Node 1 starts to execute worker_tasks.
 20:41:26 run.go:32: Node 2 starts to execute coordinator_tasks.
 20:41:26 run.go:22: Node 0 starts to execute worker_tasks.
 20:41:26 node.go:162: Node 0 received a message {}.
 20:41:27 node.go:350: Coordinator 2 sent a sync message {}.
 20:41:27 node.go:162: Node 0 received a message {}.
 20:41:27 node.go:162: Node 1 received a message {}.
...
 20:41:30 node.go:130: Node 2 failed.
 20:41:35 node.go:187: Node 1 detected coordinator 2 failed.
 20:41:35 node.go:47: Node 1 started an election.
 20:41:35 node.go:65: Node 1 sent elect msg to 2.
 20:41:35 node.go:187: Node 0 detected coordinator 2 failed.
 20:41:35 node.go:47: Node 0 started an election.
 20:41:35 node.go:65: Node 0 sent elect msg to 1.
 20:41:35 node.go:209: Node 1 received a message Elect 0 as new coordinator.
 20:41:35 node.go:65: Node 0 sent elect msg to 2.
 20:41:35 node.go:241: Node 1 sent ACK to node 0.
 20:41:35 node.go:209: Node 0 received a message Request from node 0 is refused by node 1.
 20:41:41 node.go:84: Node 0 Stop the goroutine broadcast_victory.
 20:41:41 node.go:209: Node 0 received a message Node 1 is the new coordinator
 20:41:41 node.go:101: Node 1 is broadcasting a victory to 0.
 20:41:41 node.go:136: Node 0 fail_during_broadcasting.
 20:41:41 node.go:101: Node 1 is broadcasting a victory to 2.
 20:41:41 node.go:118: Node 1 sent role switch messgae.
 20:41:41 node.go:209: Node 1 received a message Stop the goroutine
 20:41:41 node.go:162: Node 1 received a message Stop the goroutine.
 20:41:41 run.go:47: Node 1 switch role.
 20:41:41 run.go:32: Node 1 starts to execute coordinator_tasks.
 20:41:41 node.go:350: Coordinator 1 sent a sync message {}.
 20:41:42 node.go:350: Coordinator 1 sent a sync message {}.
 20:41:43 node.go:350: Coordinator 1 sent a sync message {}.
...
```

Others

Issues

1. The current program usually supports <=4 nodes running, maybe due to limited computing power of my laptop or imperfectness of my implementation. Channels get stuck when too many messages arrive. Thus the next step may be to create separate channels for ELECT, ACK, VICTORY messages.
2. Some of the unexpected behaviours such as data races, delay of message transmission, etc. are not considered, especially when there are too many nodes.

Assumptions

1. Network is reliable.
2. Network is asynchoronous.
3. Channels won't congest. (But it actually happens when node number increases.)