Assignment 2 README

Content

> Number of clients you want to test with = Number of terminals

Lamport’s Shared Priority Queue

Running the code

This part has been implemented using RPCs. So for each client you need to spin a separate terminal. For your ease of testing you may spin up less than 10 terminals to see how long it takes for all the clients to edit and release the CS. time.Sleep() has been introduced in multiple places to magnify the effects of time vs number of nodes.

In order to run the code, firstly ensure that the `clientlist.json` file contains an empty json string:

```
{}
```

> The above requirement is only for the genesis node in the network.

Then run the following command in `./distributed_mutual_exclusion/lamport_priority_queue_mutex`:

```
go run main.go
```

Run the same command in `n` terminals if you want to spin up `n` nodes.

Modifying the critical section

In order to modify the critical section, enter 1, then press enter. The critical section is just an integer value in our case. You do not need to explicitly enter anything into the critical section, for the sake of ease of testing.

Once you hit enter, the modification will take place exactly at some 10th second of the minute— this has been done to make testing easier as well, because otherwise you will have to manually enter some integer into every terminal. Plus, even if you press 1 at different times, they will all still execute concurrently!

```
for {
    timeNow := time.Now()
    if timeNow.Second() % 10 == 0 {break}
  } // Even if I press one at slightly different times, they will all send concurrently at the same time.

// execute the rest of the modify CS code.
```

To edit CS concurrently in multiple nodes, press 1 and enter on all the terminals before the tenth second of the minute.

For example: if your iphone clock shows `13:54:11`, then before it reaches the tenth second- `13:54:20`, press 1 and hit enter on that terminal. All the terminals will concurrently execute at `13:54:20`

Reading the logs

The time taken is always printed in red ← this is what we will be looking out for when making the table.

Logs related to critical section modification is printed in purple.

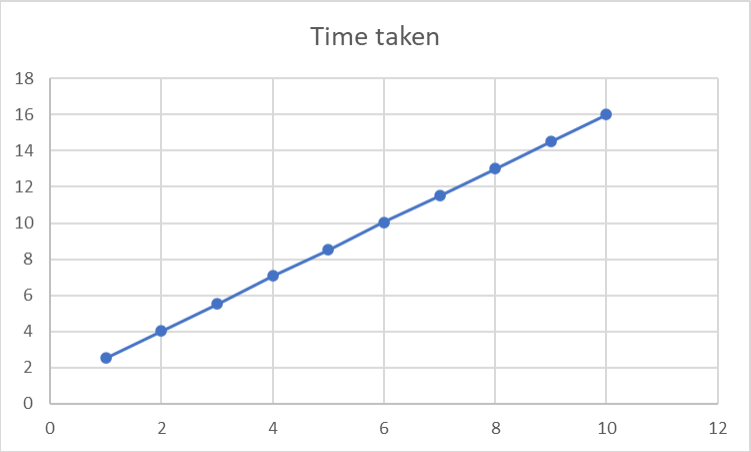
Logs related to response handling (handling of requests once you receive them) is printed in green.

In order to see the amount of time taken, check the logs in red.

The time taken for last node to get out of the critical section and for the process to get over is the node which printed out the time last → this will have no print statements underneath it. This is the same for all the protocols implemented here.

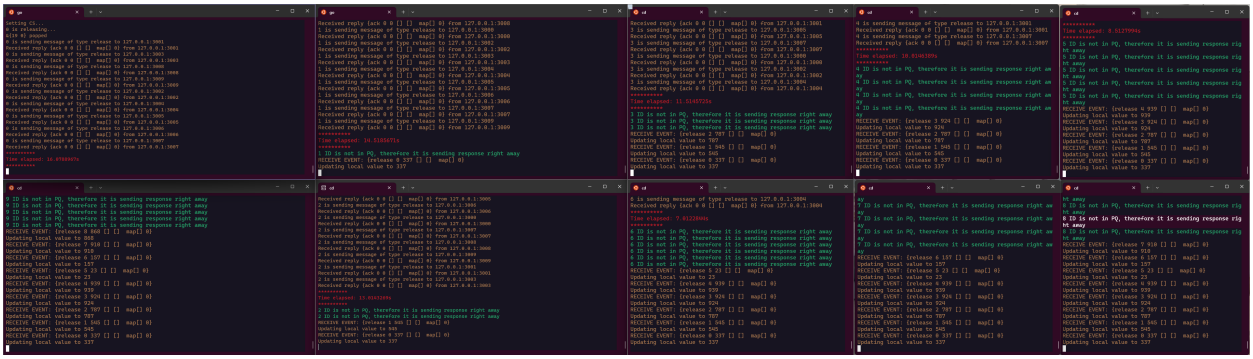
Results for lamport's shared priority queue

Number of clients	Time taken
1	2.5159691
2	4.0111192
3	5.5115263
4	7.0736346
5	8.5133836
6	10.0471589
7	11.5136595
8	13.0148754
9	14.5159476
10	16.0153349



The results are relatively linear, as the number of requesters go up.

Sample result with 10 nodes:



Zoom in for more clarity

From the above, we can see that the time taken to execute 10 clients simultaneously is 16 seconds → as the last node printed out the time at last.

Lamport's Shared Priority Queue with Ricart & Agrawala Optimization

With the R&A optimization, the code looks very similar to the first part. However, the key difference here is the way `ResponseHandler()` is implemented and the receival of `RESPONSE` type messages are received.

- Clients that do not have a request of their own directly respond to the `REQUEST` message of the requestor.

- Clients that have a request, will respond directly to the `REQUEST` message of the requestor, iff the incoming request is smaller than the local request.
- In all other cases, the client will put requests into its priority queue. And will only respond once it has done executing it's CS.
- The requestor can receive responses in two ways:

```
// within the modifyCS code:
func (client *Client) ModifyCriticalSection() {
  // Code to make request to make request and send to everybody here...

  for id, responseChannel := range responseChannels {
    if id == client.Id {
      continue
    }
    reply := <-responseChannel
    if reply.Type == RESPONSE { // receiving response here
      client.listOfResponses.Store(reply.From, 1)
      systemmodify.Println("Received a response from", reply.From, " through local call.")
    }
    client.UpdateLocalClock()
  }

  // Code to check number of responses and execute CS here...
}
```

The other way is if some receiver of the request also had it's own request, then I will receive that nodes request via RPC incoming message:

```
func (client *Client) HandleIncomingMessage(msg Message, reply *Message) error {
switch msg.Type {
case RESPONSE:
    systemmodify.Println("Received a response")
    if *client.RequestTimeStamp < INF { // If I had an active request, then update list of responses.
      client.listOfResponses.Store(msg.From, 1)
      systemmodify.Println("Received a response from", msg.From, "from RPC.")
    }
    client.CS = msg.UpdatedValue
    reply.Type = ACK + RESPONSE
}
}
```

Running the code

Like the other parts, this one has also been implemented using RPCs. In order to spin up ten clients in the directory `distributed_mutual_exclusion/lamport_ricart_agrawala/` in ten terminals (however many nodes you want); then run the node using

```
go run main.go
```

You also need to ensure that when spinning up the genesis node, the `clientlist.json` is an empty json string:

```
{}
```

The time taken for last node to get out of the critical section and for the whole process to get over is the node which printed out the time last → this will have no print statements underneath it. This is the same for all the protocols implemented here.

Modifying the Critical Section

Modifying the critical section follows the same logic as in the previous parts.

In order to modify the critical section, enter 1, then press enter. The critical section is just an integer value in our case. You do not need to explicitly enter anything into the critical section, for the sake of ease of testing.

Once you hit enter, the modification will take place exactly at some 10th second of the minute — this has been done to make testing easier as well, because otherwise you will have to manually enter some integer into every terminal. Plus, even if you press 1 at different times, they will all still execute concurrently!

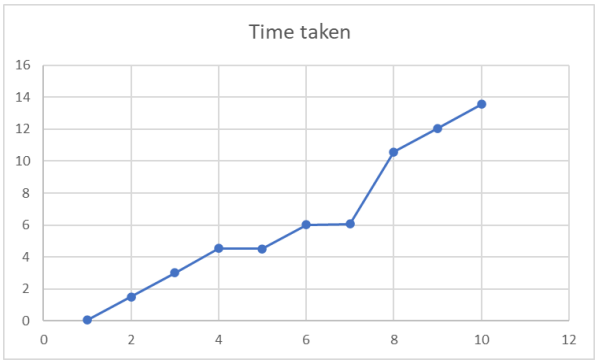
```
for {
    timeNow := time.Now()
    if timeNow.Second() % 10 == 0 {break}
  } // Even if I press one at slightly different times, they will all send concurrently at the same time.

// execute the rest of the modify CS code.
```

To edit CS concurrently in multiple nodes, press 1 and enter on all the terminals before the tenth second of the minute.

For example: if your iphone clock shows `13:54:11`, then before it reaches the tenth second- `13:54:20`, press 1 and hit enter on that terminal. All the terminals will concurrently execute at `13:54:20`

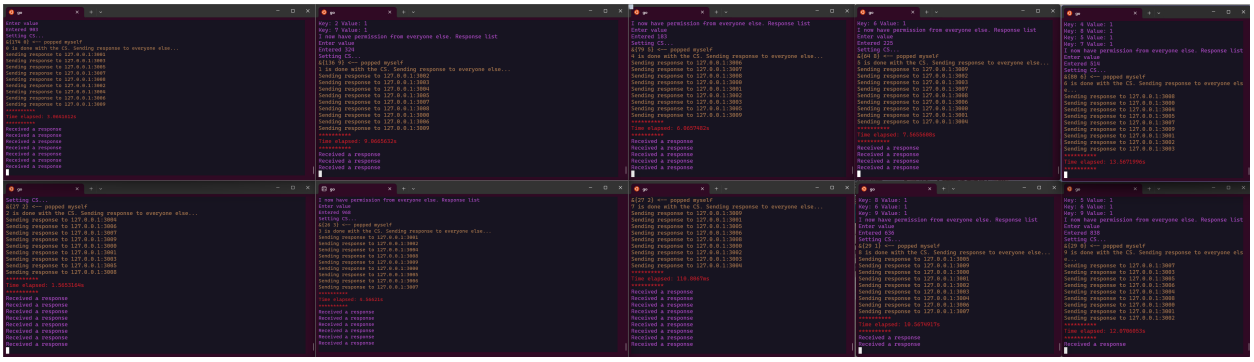
Results

Lamport's Shared Priority Queue with Ricart and Agrawala optimization	
Number of clients	Time taken
1	0.0499748
2	1.5099366
3	3.0107272
4	4.5407253
5	4.5118003
6	6.0142896
7	6.0666157
8	10.5780867
9	12.0516811
10	13.5671996



We can see that the time taken is lower than that of Lamport's Shared Priority Queue without R&A optimization.

Sample result with 10 nodes:



Zoom into the picture for more clarity

From the above, we can see that the time taken to execute 10 clients simultaneously is 13 seconds → as the last node printed out the time at last.

Voting Protocol

This part has also been implemented using RPCs. The voting protocol with deadlock avoidance works as follows:

When a node requests to enter the critical section, it sends the request out to all the nodes. On the sending side, the nodes will simply create a request, append a timestamp, and send it out to all nodes in the `clientlist`.

On the receiving side, the nodes will put the request in a priority queue. Then it will trigger a `ResponsHandler` - which will put the request in a priority queue, ordered by the timestamp, and tie broken by their respective IDs. The `ResponseHandler` loops through all the elements in the priority queue, and sends rescind to all the elements but the head. It will also send a response (affirmative vote) to the head of the min priority queue. The priority queue keeps changing as more requests come in. Therefore, it runs a for loop to send responses to everyone.

```
func (client *Client) ResponseHandler(from []int, fromId int, remoteTime int) {
    isPriorityQueueEmpty := false
    for !isPriorityQueueEmpty {
        count := 0

        // Get a copy of the priority queue entries
        entriesCopy := client.PQ.GetEntries()
    if len(entriesCopy) == 0 {
            isPriorityQueueEmpty = true
      break
        }


        // send rescind to everyone but the top entry in your PQ
        for _, PQEntry := range entriesCopy {
            if count == 0 {
                // don't send rescind to the first element in the queue
                count++
                continue
            }
            msg := Message{Type: RESCIND, From: client.Id}
            go client.CallRPC(msg, client.Clientlist[PQEntry.ID], make(chan<- Message))
        }

        // send a response to the first entry in your PQ
        msg := Message{Type: RESPONSE, From: client.Id}
        go client.CallRPC(msg, client.Clientlist[entriesCopy[0].ID], make(chan<- Message))
        time.Sleep(1500 * time.Millisecond)
    }
}
```

The code has also been designed in a way that you cannot vote for yourself. This has been done because of tie breaking in certain scenarios. Example there are three nodes- all requestors. If the nodes vote for themselves, they only need to get one more vote to get into CS. As this might cause a race condition, we can remove the requirement that clients vote for themselves. So the clients will only respond to you if you are at the top of their priority queue.

Running the code

The code is run in the same way as the previous parts. Just ensure that `clientlist.json` has an empty json string:

```
{}
```

> Like in the previous parts, the above is only a requirement for the genesis node.

And run the following command in your terminal in `../distributed_mutual_exclusion/voting_protocol`:

```
go run main.go
```

The logs are colored in the same way as the previous parts as well.

The time taken for last node to get out of the critical section and for the whole process to get over is the node which printed out the time last → this will have no print statements underneath it. This is the same for all the protocols implemented here.

Modifying the critical section

Modifying the critical section follows the same logic as in the previous parts.

In order to modify the critical section, enter 1, then press enter. The critical section is just an integer value in our case. You do not need to explicitly enter anything into the critical section, for the sake of ease of testing.

Once you hit enter, the modification will take place exactly at some 10th second of the minute — this has been done to make testing easier as well, because otherwise you will have to manually enter some integer into every terminal. Plus, even if you press 1 at different times, they will all still execute concurrently!

```
for {
    timeNow := time.Now()
    if timeNow.Second() % 10 == 0 {break}
  } // Even if I press one at slightly different times, they will all send concurrently at the same time.

// execute the rest of the modify CS code.
```

To edit CS concurrently in multiple nodes, press 1 and enter on all the terminals before the tenth second of the minute.

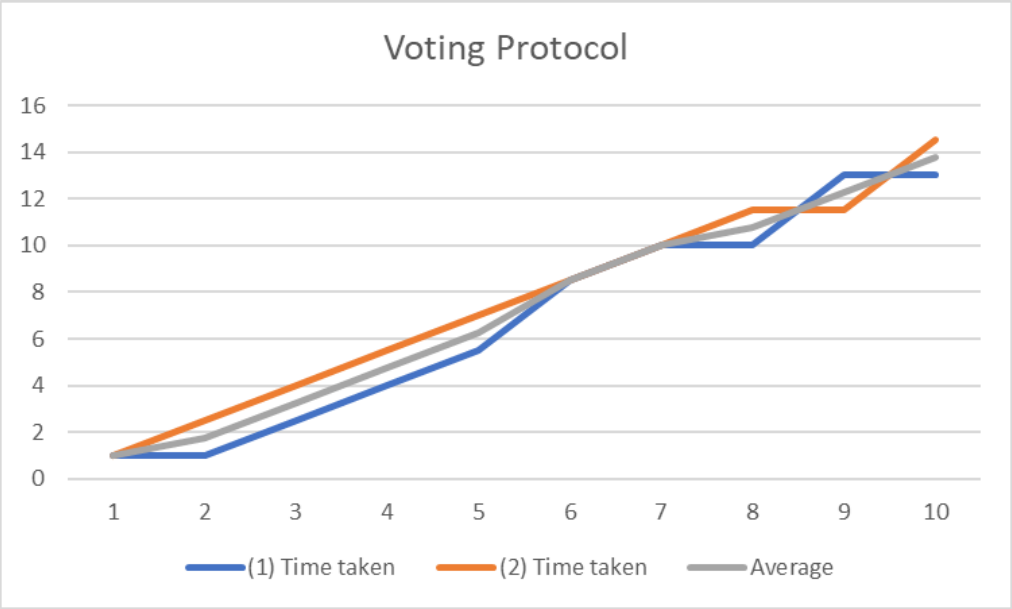
For example: if your iphone clock shows `13:54:11`, then before it reaches the tenth second- `13:54:20`, press 1 and hit enter on that terminal. All the terminals will concurrently execute at `13:54:20`

Results

The results were averaged over 2 runs, and averaged. As can be seen from the result table below, the time taken for the 1 client and 2 clients in run 1 is very close to each other, whereas in run 2 it is more

differentiated. In order to even the differences out, the average was taken and plotted in a graph, as shown below.

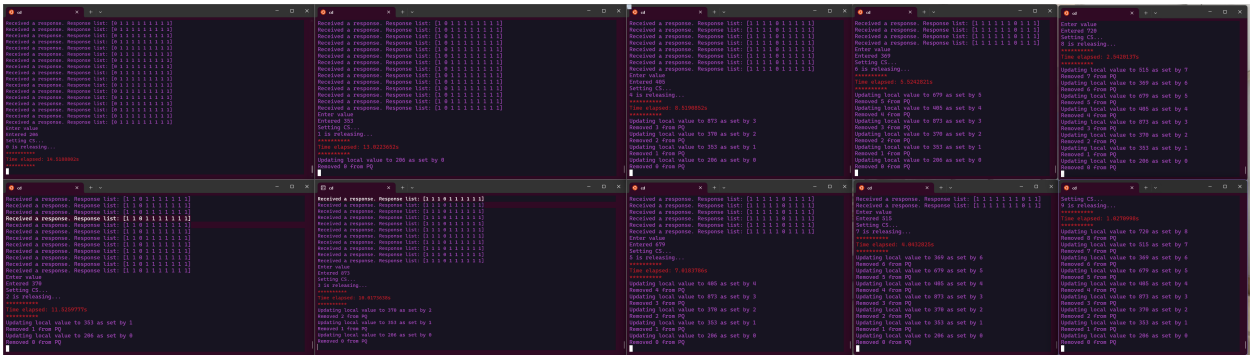
Voting Protocol			
Number of clients	(1) Time taken	(2) Time taken	Average
1	1.0106872	1.0089929	1.00984005
2	1.0114089	2.509934	1.76067145
3	2.5093088	4.0116012	3.260455
4	4.0113538	5.510904	4.7611289
5	5.5106495	7.0122452	6.26144735
6	8.5141356	8.5128002	8.5134679
7	10.0153503	10.016977	10.01616365
8	10.0143246	11.5175983	10.76596145
9	13.0203808	11.5194679	12.26992435
10	13.0353711	14.5534512	13.79441115



A regression line for these points yields a linear relationship in this case as well.

We see that on average, the CS execution times are lesser for the voting protocol, as compared to lamport shared priority queue protocol.

Sample run with 10 nodes



Zoom into see more clearly

From the above, we can see that the time taken to execute 10 clients simultaneously is 14 seconds → as the last node printed out the time at last.