Problem 3

I implement the transaction module to do broadcasting and monitoring by using GO.

I choose to use GO because it supports for data type like uint64 and its built-in Goroutines makes it easy for developers to write concurrent code.

I use the OOP because it is a way to build the complex software and provides many design patterns that serve as guideline for improving maintainability.

First, I declare struct Transaction, it is like the class in Java, to hold the method for broadcasting transaction and monitoring transaction status.

```go
type Transaction struct {
}
```

```go
func (t Transaction) BroadcastTransaction(request BroadcastTransactionRequest) (BroadcastTransactionResponse,error) {
    jsonData,err:=json.Marshal(request)
    if err!=nil{
        return BroadcastTransactionResponse{} ,err
    }

    fmt.Printf("%s Broadcasting...\n",string(jsonData))
    req, err := http.NewRequest("POST", "https://mock-node-wgqbnxruha-as.a.run.app/broadcast", bytes.NewBuffer(jsonData))
    if err!=nil{
        return BroadcastTransactionResponse{} ,err
    }

    req.Header.Set("Content-Type","application/json")
    client:=&http.Client{}
    resp,err:=client.Do(req)
    if err!=nil{
        return BroadcastTransactionResponse{} ,err
    }
    defer resp.Body.Close()

    var response BroadcastTransactionResponse
    err=json.NewDecoder(resp.Body).Decode(&response)
    if err!=nil{
        return BroadcastTransactionResponse{} ,err
    }

    fmt.Printf("%s Broadcasting Successfully\n",string(jsonData))
    return response,err
}
```

```go
func (t Transaction) MonitorTransactionStatus(request MonitorTransactionStatusRequest)(MonitorTransactionStatusResponse,error) {
    fmt.Printf("%s Monitoring...\n",request.TxHash)

    resp,err:=http.Get(fmt.Sprintf("https://mock-node-wgqbnxruha-as.a.run.app/check/%s",request.TxHash))
    if err!=nil{
        return MonitorTransactionStatusResponse{} ,err
    }
    defer resp.Body.Close()

    var response MonitorTransactionStatusResponse
    err=json.NewDecoder(resp.Body).Decode(&response)
    if err!=nil{
        return MonitorTransactionStatusResponse{} ,err
    }

    fmt.Printf("%s Monitoring Successfully\n",request.TxHash)
    return response,err
}
```

For integration into another application, developers can import this module and utilize its function to use the broadcast and status monitoring functions.

Additionally, I provide the transaction pipeline function for applications that prefer not to implement pipeline themselves. The output of the pipeline is the transaction status, represented as an Enum, which helps the application understand the possible outputs.

The Enum is used because the number of possible output variants is limited.

```go
type TransactionStatus uint8
const(
    CONFIRMED TransactionStatus=iota+1
    FAILED
    PENDING
    DNE
)
```

```go
func (t Transaction) InvokeTransactionPipeline(pipelineInput TransactionPipelineInput)(TransactionPipelineOutput,error){
    broadcastRequest := BroadcastTransactionRequest(pipelineInput)

    resBroadcast,err:=t.BroadcastTransaction(BroadcastTransactionRequest(broadcastRequest))
    if(err!=nil){
        return TransactionPipelineOutput{},err
    }

    monitorRequest:=MonitorTransactionStatusRequest(resBroadcast)
    resTransactionStatus,err:=t.MonitorTransactionStatus(MonitorTransactionStatusRequest(monitorRequest))

    return TransactionPipelineOutput{convertStatus(resTransactionStatus.TxStatus)},err
}
```

Example of using this module in the application:

```
transactionModule := transaction.Transaction{}
var wg sync.WaitGroup

for _,input:= range inputList{
    go func(){
        response,err:= transactionModule.InvokeTransactionPipeline(tran
        if err!=nil{
```

For performance, the bottle neck of every functions is the speed of a network.

I assume that this module will be used to monitor the status of each transaction in a large transaction list.

Therefore, if developers do not implement it correctly, processing all transactions may be slow and result in poor user experience.

To solve this problem, Goroutine is used to write the concurrent code. By processing each transaction concurrently, the developers can create a more efficient application.

```
for _,input:= range inputList{
    go func(){
        response,err:= transactionModule.InvokeTransactionPipeline(transaction.TransactionPipelineInpu
        if err!=nil{
            fmt.Println(err)
        }else{
            display(input,response.TxStatus)
            switch response.TxStatus{
                case transaction.CONFIRMED:
                    //Notify the user that transaction is completed
                case transaction.FAILED:
                    //Notify the user that transaction has failed to process
                case transaction.PENDING:
                    //Notify the user that transaction is currently being processed
                case transaction.DNE:
                    //Notify the user that transaction does not exist
                default:
                    fmt.Println("Wrong Transaction Status")
            }
        }
        wg.Done()
    }()
    wg.Add(1)
}
wg.Wait()
```

For reliability, if errors occur during the process, this module can notify the application that the errors are occurring.

```
if err!=nil{
    return BroadcastTransactionResponse{} ,err
}
```

```go
for _,input:= range inputList{
    go func(){
        response,err:= transactionModule.InvokeTransactionPipeline(transaction.TransactionPipelineInput
        if err!=nil{
            fmt.Println(err)
        }else{
            display(input,response.TxStatus)
            switch response.TxStatus{
                case transaction.CONFIRMED:
                    //Notify the user that transaction is completed
                case transaction.FAILED:
                    //Notify the user that transaction has failed to process
                case transaction.PENDING:
                    //Notify the user that transaction is currently being processed
                case transaction.DNE:
                    //Notify the user that transaction does not exist
                default:
                    fmt.Println("Wrong Transaction Status")
            }
        }
        wg.Done()
    }()
    wg.Add(1)
}
wg.Wait()
```

After obtaining the transaction status from this module, we can handle it in the following ways:

CONFIRMED : Notify the user that transaction is completed

FAILED : Notify the user that transaction has failed to process. If necessary, rerun the process.

PENDING : Notify the user that transaction is currently being processed.

DNE : Notify the user that transaction does not exist.