2510 Project 2

Yuhang Wang, Yuxuan Zhang

Email: YUW199@pitt.edu, YUZ276@pitt.edu

Contribution: Both members have the same contribution to the project.

Overall design

- Language: Go
- We implemented a bi-directional streaming chat server based on gRPC.
  - `rpc ChatRoom(stream ClientRequest) returns (stream ServerResponse){}`
- We defined a broadcast function to send messages to multiple clients on every update.
  - On every broadcast, the server will send the latest 10 messages to every client in the group.
- All functions and features are tested and work properly.

Project structure

- Under the `chat` folder:
  - **chatserver.go:** main logic for a never ending loop receiving messages from clients and servers.
    - **Main service logic:** ChatRoom()
  - **replica.go:** functions for insert messages after partition healed, sending between replicas, anti-entropy functions.
    - **Anti-Entropy**: CheckForMerge(), Entropy(), SendMySaving()
    - **Merge messages:** Insertion(), InsertLike()
  - **HelperFunctions.go**: helper functions
  - **data.go:** Where we define our data structure and corresponding functions for linked list operation.

Client-Server interaction

gRPC Message and Service

Sercive

```
service Chat { rpc ChatRoom(stream ClientRequest) returns (stream ServerResponse){} }

func (sm *Server) ChatRoom(ct Chat_ChatRoomServer) error{} // our service function
```

- On the Client side, the program will read from the user input and set the corresponding **Request field inside** the client message ClientRequest sent to the server.
- On the Server side, the program will execute different functions given for different Request field. The server response message will also **set the Request field**, such that the client knows how to handle the message received. For example, whether the response contains a chat message to be printed, or the client needs to extract and store the User ID assigned by the server.

Message

```
message ClientRequest{
  string request = 1;
  int32 user_id = 2;
  string userName = 3;
  string ori_group_name = 4;  // join, switch
  string new_group_name = 5; // join, switch
  int32 like_number = 6;
  string Content = 7;
  int32 Replica = 8;
  int32 Timestamp = 9;
  int32 GroupTimestamp = 10;
  string FromServer = 11;
  int32 MapIndex = 12;
}

message ServerResponse{
  string request = 1;
  string username = 2;
  string content = 3;
  int32 like = 4;
  int32  msg_id = 5;
  int32 user_id = 6;
  string goodbye = 7;
  int32 local_message = 8;
}
```

- Not all field will be set for every message sent, we only set and read the certain fields for certain function.
- We will explain each field of the message in the following section

Main Data Structure

- Server

```
type Server struct {
    UnimplementedChatServer
```

```
    GroupMap  map[string]*Group
    Mu        sync.Mutex
    Me        int32
    StringMe  string
    UserCount int32
    StreamSet map[string]*ServerConfig
}

type ServerConfig struct {
    Client   ChatClient
    Stream   Chat_ChatRoomClient
    Context  context.Context
    LastTime time.Time
    isOnline int32
}
```

- This is the server object. We use a map to map the group name with its' `Group` struct pointer. `Me` and `StringMe` represents the int version and the string version of the replica Number(1, 2, 3...)
- StreamSet is used to store all other replicas' information(context, client....) so we can send messages to them or reconnect to them.

- Group

```
type Group struct {
    GroupName      string
    MessageList    *List
    UserList       map[int32]*User  // track the connecting clients to this group
    sameUserNameCount map[string]int32 //track the identities (UserNames) of this group
    MaxLamport      int32
    ReplicaInfo     map[string]*Information
}
```

- For each group client joining, we create a struct object and store all related information.
- `MessageList` is a list implemented using the `linked list`, we keep the head and tail for any insertion easily. All the messages appended to this group will be store in this variable. The first message of the group will be `Group.MessageList.Head.Message`

```go
type List struct {
    Head     *Node
    Tail     *Node
    TenStart *Node
    Count    int32
}
```

- You may notice that the type inside the `List` is `*Node`. For the whole program, we keep two different list, one is for message, another one is for like activities. So a Node may represent a message or a like event.

```go
type Node struct {
   Activity *Activity
   Message  *Message
   Next     *Node
}
```

- The ReplicaInfo variable is the most important variable an desgin in this replica version of chat room. I will explain it indepth in the Replica section of this report.

- Message:

Now, finally it comes to the message representation. Based on the above description. Message is a struct variblae inside struct `Node`.

```go
type Message struct {
    MsgId        int32            `json:"msg_id"`
    Lamport      int32            `json:"lamport"`   // lamport timestamp
    Replica      int32            `json:"replica"`   // which replica does this message come from
    Creator      string           `json:"creator"`   // the username of the creator
    Content      string           `json:"content"`   // the content of the message
    UserLike     map[string]int32 `json:"userLike"` // 1 indicates like, -1 indicates dislike
    ActivityList *List            `json:"-"`
    TotalLike    int32            `json:"totalLike"`
}
```

We keep every message's ID(Starting from 1, 2,...). The `Lamport` timestamp and the `Replica` indicates where this message is originally appended at. For example, if server 5 receives a message from a client. **When server 5 transmit this message to server 1**, server 1 will check `Message.Replica` such that it knows this message belongs to **server 5**. It is important for our causality to be realized. `ActivityList` is a list of all like event happening on this message. From this field, we can keep track any conflicting event happening during the partition such that we can calculate the true total like number of this message. I will explain all important fields again in the later section when we describe our replication logic.

Replication - Messages consistency

Summary

We utilize the lamport timestamp to maintain the causality between messages at all times. Each server periodcally send a `CheckForMerge` message to all other replicas. We only need **three** timestamp `int32` varibales to represent the chat history of local server. On the other side, if a replica receive the `CheckForMerge` message, it will **compare the three timestamp** varibales to **what it got**. It will then send all messages that the sender server miss. Basically, every one tell the other servers what it currently got. All other servers compare the three counters with their own, if find the guy misses something, they will send that server's all missing messages.

I will go in them in detail in the later section.

Lamport and Causality

- Each Message has two important fields: `Replica` and `Lamport`.
  - As previously mentioned, `Replica` denotes which replica this message comes from.
  - Lamport denotes the lamport timestamp.

When receiving a message from the client

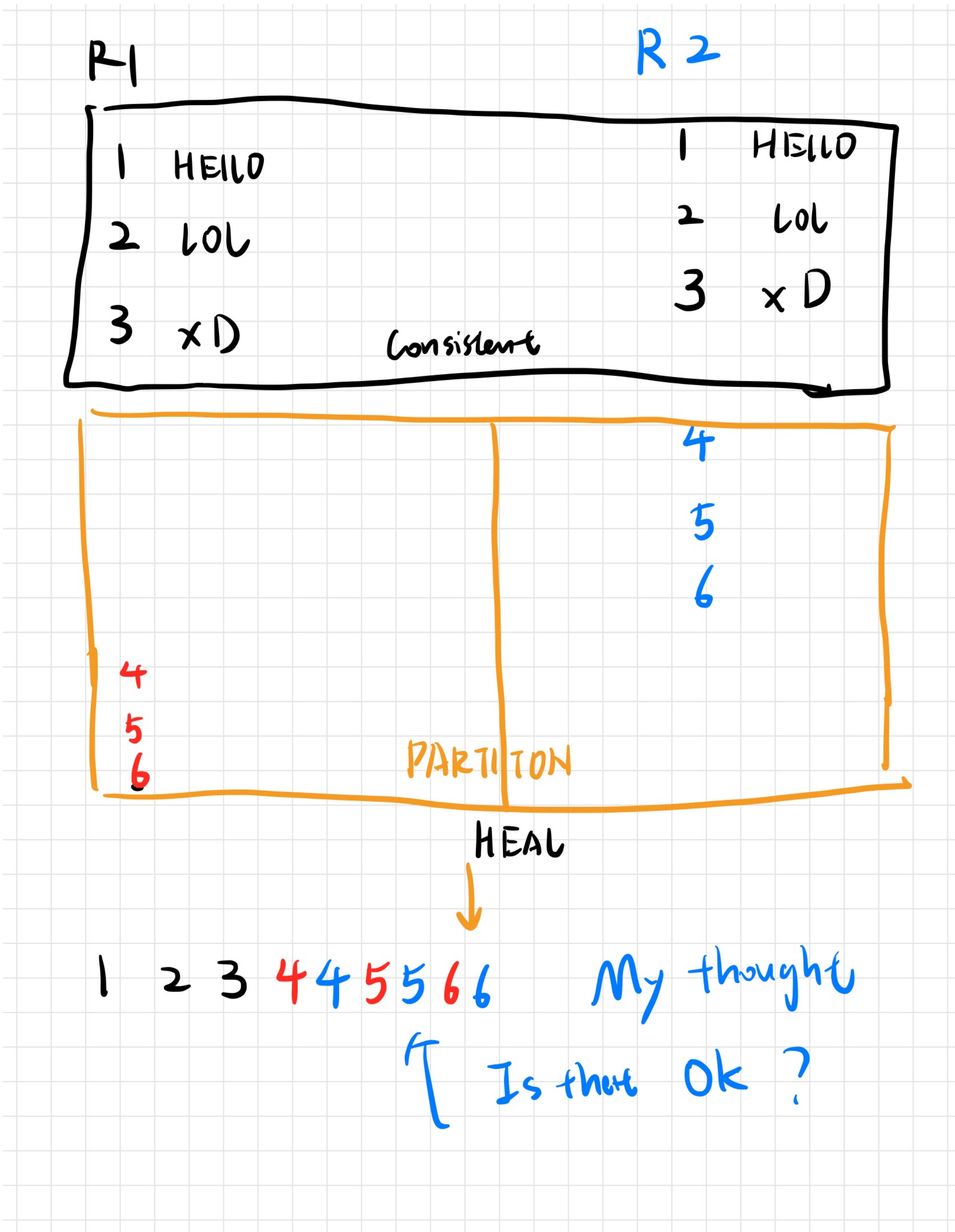
- From each replica's side, whenever **a client connecting to itself** appends a message, it append the message and increase the lamport timestamp by 1 and set it to this message. The replica number will be itself. This message is then regarded as its own message. Hence, the causality is guaranteed, as every message after a a message always has the highest lamport timestamp.

When receiving a message from other server

- We need to insert the message into the correct place.
- We conduct the insertion based on the lamport timestamp as well as the replica number.

Insertion logic

- We always place the message with smaller `Lamport` timestamp before the message with higher one.
- When we encounter two messages have the same Lamport:
  - From server 1 to server 5, server 1's message has the highest precedence, then server 2, server 3.....
  - That means when two messages has the same **lamport timestamp**, we always place the message with the higher precedence replica before the other one.



- You can see that when R1 has higher precedence than R2, we always place R1's message before R2's message when we have the same lamport timestamp.

The final order

Hence, every message has different `Replica` from different server. For a single server, every message has different `Lamport` timestamp. Therefore, combine those two fields, we can differentiate every message. As previously mentioned, we insert messages from other replicas using the two fields. As a result, the final order between messages between relicas is consistent.

Replication - Communicating logic

We don't want to simply let every replica sends its whole chat history periodcally to all other replicas. We think it is a waste of resources. Therefore, I come up with a solution.

```
type Group struct {
    GroupName         string
    MessageList       *List
    UserList          map[int32]*User  // track the connecting clients to this group
    sameUserNameCount map[string]int32 // track the identities (UserNames) of this group
    MaxLamport        int32
    ReplicaInfo       map[string]*Information
}

type Information struct {
    UserList           string
    GroupNameTimestamp int32 // last received message's timestamp from this server
    LikeTimestamp      int32
    MsgTimestamp       int32
    LikeEventLamport   map[int32]*Node
    MessageLamport     map[int32]*Node
}
```

- As previously mentioned, the `ReplicaInfo` is the most important data structure to send messages between replicas.
- The `ReplicaInfo` is just a map mapping the other server's name(1-5) with the `Information` struct
- The `Information` struct represents all the messages, message count, like events, like events counts for a given server. The server name will be used as the key in ReplicaInfo
- `MstTimestamp` is the counter of how many messages have been appended to the Server
    - If a client append a message to Server 1, it means that this message is appended and belong to Server 1

- For example, only Server 1 itself can increase the number of its message count, which is `Group.ReplicaInfo[1].MsgTimestamp`. Other server can only replace their old value by a higher one and accept the corresponding message content, and the higher one can only orignially comes from the Server itself. It means that maybe server 3 receives server 1's total message count through server 2 through eventual path propogation.

- `MessageLamport` is a map storing the message number of this replica as the key and the message Node as the value. When can then access the **message content** and lamport and othe stuff through this map.
  - Please notice that this message number does not equal to the **message ID of the whole group** chat history among all 5 replicas. The key only represents the i th message appended to a given server, not the number in the whole chat history.
- All fields will be set to 0 in all replica.

```
for i := int32(1); i <= 5; i++ { // iterate the map over 5 replicas
     s := strconv.Itoa(int(i))  // convert the server number to string
        the_group.ReplicaInfo[s] = &Information{UserList: "",
            MsgTimestamp: 0, LikeTimestamp: 0, GroupNameTimestamp: 0,
            LikeEventLamport: make(map[int32]*Node),
            MessageLamport: make(map[int32]*Node)}
    }
```

For example

- Let's say we are currently in Server 1.
- If Server 1 receives a message from a client connecting to itself. We know that this message belongs to server 1, since the operation is happened on Server 1's side.
- So we update `Group.ReplicaInfo[1]` , the 1 means that server 1. We will increase the MsgTimestamp inside the `Information` by 1. `Group.ReplicaInfo[1].MsgTimestamp++`.
- Then, storing the `Message` inside the `Node` to the map `MessageLamport.`
- At this time, `Group.ReplicaInfo[1].MsgTimestamp` is 1 on Server 1's side. However, on other replicas' side, this field is still 0.
- Then, when it comes to the anti-entropy phase, All other servers will iterate their data inside the map `Group.ReplicaInfo[ServerName]` using all server Names From 1-5. They will send all 5 `Info` struct to all ramaining 4 servers. We know that , all fields inside `Group.ReplicaInfo[1]` on server 2, 3, 4, 5 are 0.
- Then, when server 1 receives all other replicas' request, and insepct the certain fields, it will know that they don't have the latest information of `Group.ReplicaInfo[1]` since their value of `MsgTimestamp` and `LikeTimestamp` and `GroupNameTimestamp` are 0, which is of course, since only the Server 1 can modify this field. Therefore, Server 1 knows that all other replicas currently has 0 messages from Server 1. Then, it will iterate from 0 to its current message counter, and look in the map to find the corresponding message node, send it to all replicas.

- Moving on, then 2,3,4,5 receives messages from server 1. How do they know whether to perform the message insertion or not? Maybe it is just a waste of time since I may already has the latest message. They will firstly check the certain fields in the gRPC message to extract the `MessageLamport` such that it is higher than the value currently in its map. Only by then will the server update and trying to insert the message.
- I may not well explain the logic, please move to the next section. I have provide better example when explaining the eventual path progogation. **Thank you.**

Eventual path propagation

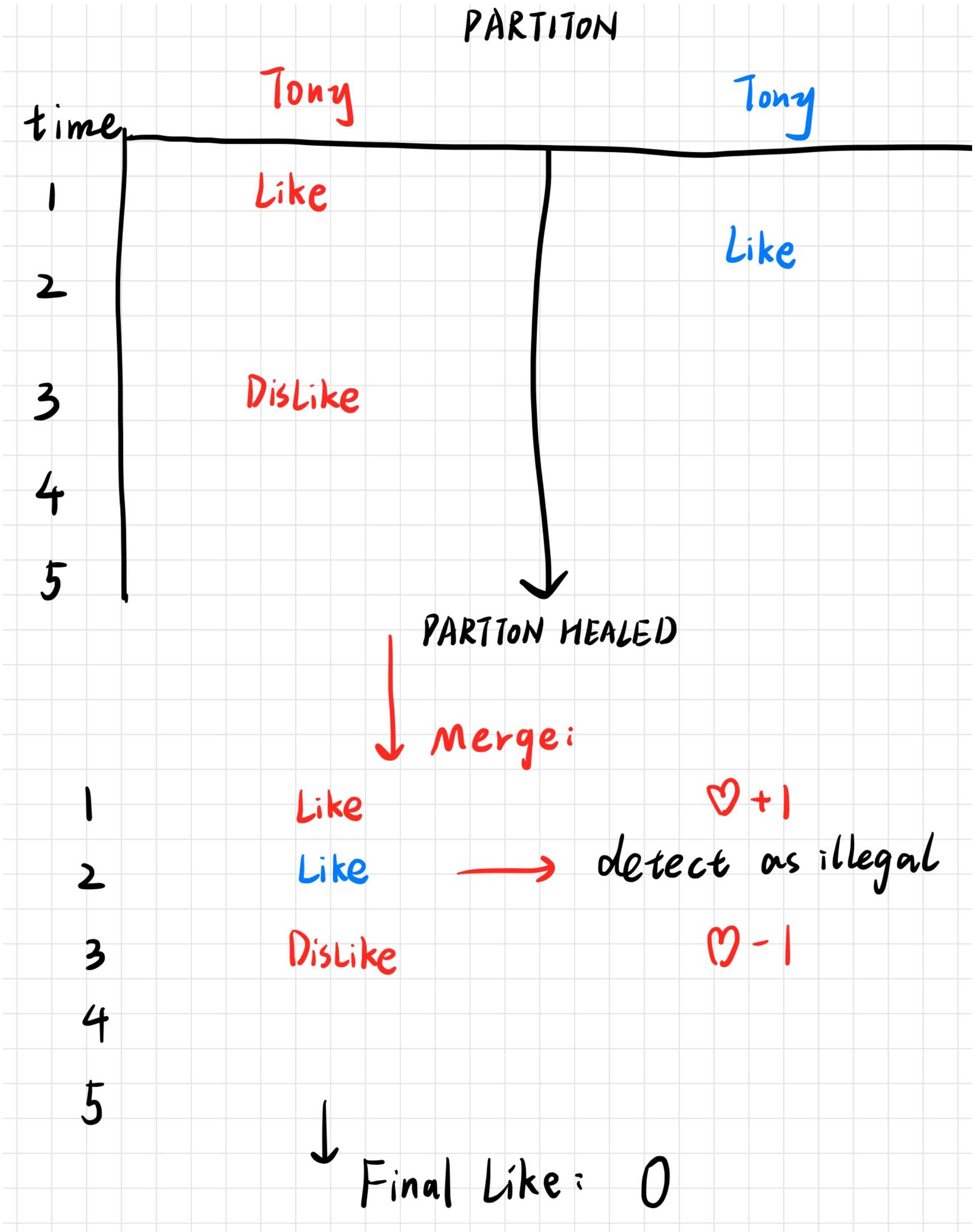
I divide the message of the group into 5 parts, each represents the origin servers. It is flexible when it comes to eventual path propagation. Since during a partition,

- At first, the counter for message belongs to Server 1 is 4. Let's assume all replicas have the same information.
- Let's say now we introduce a partition: [1,2] [2,3,4,5].
- After a client appends **two message** on Server 1, Server 1 will update its own Server 1's counter to 6. When Server 2 sends its own counter(which is 4 at this time) to server 1, Server 1 will detect that, and send message from 4 to 6 (exclude message 4), which are only two messages: message 5 and message 6 to Server 2.
- Server 2 then accept the two messages, since they have higher Server 1's MsgTimestamp(not the Lamport for the whole group). Server 2 will then insert the two messages to its whole group history based on the lamport and replica(notice that those two are the real Lampor timestamp among all replicas). After that, it will update its Server 1's message counter to 6.
- At this step, server 2 and server 1 are consistent. Hence, when **Server 3** sends **Server 3 's** Server 1's message counter to Server 2. Server 2 will detect that server 3 misses 2 messages from Server 1.
- The logic is the same for the Group connecting Userlist, each like/dislike event for a Message. All counter for all replicas will be kept in all replicas.
- In total, this is my design for the logic of the replicated chat room.

Ensuring correct like number after partition healed

```go
type Message struct {
    MsgId        int32            `json:"msg_id"`
    Lamport      int32            `json:"lamport"`  // lamport timestamp
    Replica      int32            `json:"replica"`  // which replica does this message come from
    Creator      string           `json:"creator"`  // the username of the creator
    Content      string           `json:"content"`  // the content of the message
    UserLike     map[string]int32 `json:"userLike"` // 1 indicates like, -1 indicates dislike
    ActivityList *List            `json:"-"`
    TotalLike    int32            `json:"totalLike"`
}
```

- I have a list to store every like event for a mesasge.
- After the partition is healed, when other replicas send their like activities during the partition to a server, this server will insert the like/dislike activity to the correct place based on the actual time.



- You can see that after merge, all activities will be sort in the ascedning order. We will then iterate over the like activity list for a message and calculate the final like count, eliminating all illegal behaviors.

Save on Disk / Restoring / View Implementation

- The data structure is too complicated to save on disk in a JSON file. I tried but failed. Therefore, I only store the group name for all group registered on the Server. After the partition is healed, the Server will broadcast all servers for all group information and restoring from the server crash.
- Although the server is able to get the group name list even though I don't put the name into the file. We have a go routine never stops, constantly asking other replcias to send their group list to itself. This go routine will also act like a heartbeat message.
  - When the client type the `view` command, the server will substract the now time and the last time receiving message from a certain server to detect whether the duration is longer than the timeout and the server is offline.

Other things

- We have included the report from project 1 under the folder. For this report, we mainly focusing on the logic of the replica. You may want to check out the previous report for reference if you wan to see our design for client-server logic, or the basic operation like `join`, `swtich`, ...etc.