COP 5615 – FALL 2020
Project 4 Part II – Twitter Clone (WebSocket Interface and Cryptography)

Yu-Peng Chen
7904-3193

1. Overview
   This document contains the following items:
   A. Instructions to run the code
   B. JSON-based API
   C. The WebSocket interface
   D. Authentication and Cryptography for connection and digital signature
   E. The client that is implemented using an AKKA actor
   Note that some of the information related to Part I can be found in the **Appendix**.
2. Instructions
   Two independent processes are implemented. One is the server (engine) process, the other is the client process. For the client process, I implemented a simulator to activate multiple clients and simulate the clients' behaviors. The simulation process is similar to the one in Part I, with **the communication method changed (i.e., the WebSocket interface)** and **operations that are related to cryptography added**. There is no interactive mode provided yet. The program only allows for setting the following parameters: **1) number of simulated clients, 2) simulation mode, 3) whether to output log files for each client, and 4) log level on the server side**. In addition to the two phases implemented in Part I, a **log level** parameter is added on the server side to test the functionalities, especially the ones related to authentication and cryptography. Please refer to the Appendix for the details of the Testing phase and the Zipf phase implemented in Part I.
   A. Logging in the Testing phase
   On the server side, a new parameter is added, which is the **LOG_LEVEL** parameter with three options: 1) trace, 2) tweets, and 3) security. This parameter allows for logging data with different levels to check the correctness of the functionalities data with different to check the correctness of the functionalities. On the simulator side, the "testing" phase is used.
   - First, navigate to the "**Server**" folder and run the following commands:
     - dotnet build
     - dotnet run [LOG_LEVEL]
       - "trace": This option will allow the program to output every operations.
       - "tweets" : This option will allow the program to output tweets and security information.
       - "security": This option will only allow the program to output security information
       - Example : dotnet run "tweets"

     This we keep the process running and listening to requests.
   - Then, navigate to the "**Simulator**" folder and run the following commands:
     - dotnet build
     - dotnet run [NUMBER_OF_USER] [MODE] [DEBUG]
       - Example: dotnet run 50 "testing" "false"
       - NUMBER_OF_USER: An integer indicating the number of simulated users.

        - MODE: A string that should be set to **“testing”** in this phase.
        - DEBUG: A string that can be set to “true” or “false”.

B. Zipf phase

- First, navigate to the “**Server**” folder and run the following commands:
    - dotnet build
    - dotnet run [LOG_LEVEL]

  This we keep the process running and listening to request.
- Then, navigate to the “**Simulator**” folder and run the following commands:
    - dotnet build
    - dotnet run [NUMBER_OF_USER] [MODE] [DEBUG]
        - Example: dotnet run 10000 “zipf” “false”
        - NUMBER_OF_USER: An integer indicating the number of simulated users.
        - MODE: A string that should be set to **“zipf”** in this phase.
        - DEBUG: A string that should be set to “false” in this phase

C. Note

- To test the correctness of the program, it is recommended to set a small number of clients (e.g., 50). It will be easier to check by reading the log in the terminal.

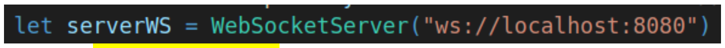
3. The WebSocket Interface

A. Third-Party Package

- The **WebSocketSharp** package is used for the implementation of the WebSocket interface.
- Note that there will be a warning message saying that the package may not be fully compatible with the project. The framework used for the project is **net5.0**. Since the **WebSocketSharp** package has not been updated since 2016, it is expected to have issues related to the compatibility. However, there has been no issues occurred so far.

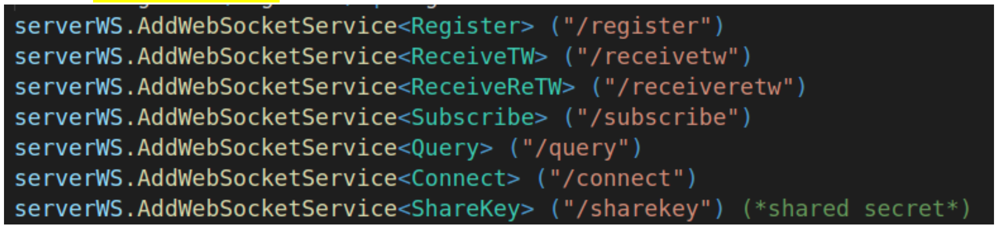
B. Implementation

- On the server side, the following items are implemented
    - Set up a WebSocket Server with the following code:



```
let serverWS = WebSocketServer("ws://localhost:8080")
```

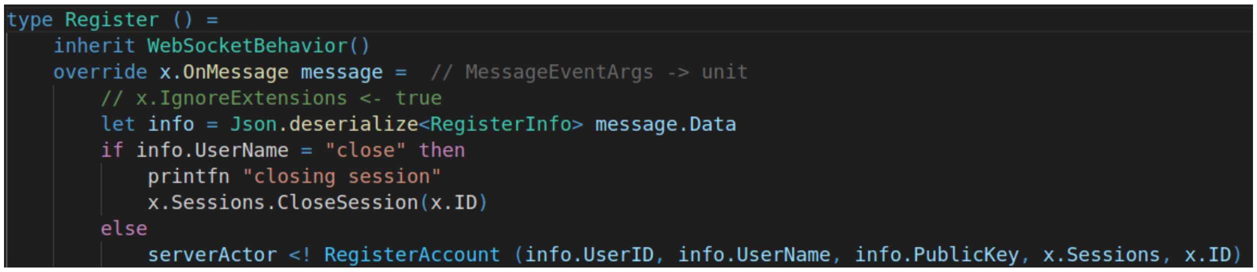
    - Set up seven services with corresponding URIs



```
serverWS.AddWebSocketService<Register> ("/register")
serverWS.AddWebSocketService<ReceiveTW> ("/receivetw")
serverWS.AddWebSocketService<ReceiveReTW> ("/receiveretw")
serverWS.AddWebSocketService<Subscribe> ("/subscribe")
serverWS.AddWebSocketService<Query> ("/query")
serverWS.AddWebSocketService<Connect> ("/connect")
serverWS.AddWebSocketService<ShareKey> ("/sharekey") (*shared secret*)
```

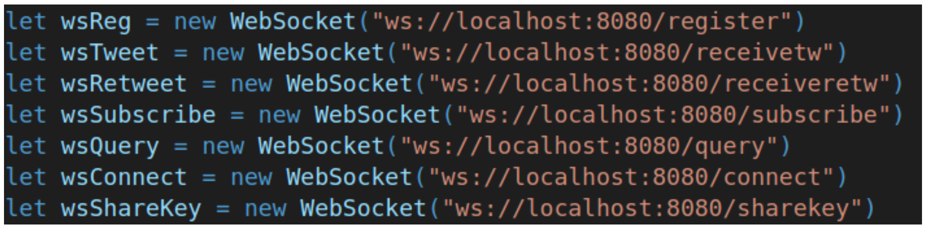
        - The ReceiveTW service is used to receive plain tweets, tweets with hashtags, and tweets with mentions. The ReceiveReTW service is used specifically for re-tweeting.
        - The Query service covers the query for mentions, hashtags, and subscriptions.
        - The Connect service is used to connect or disconnect from the server.
        - The ShareKey service is used to establish a secret key between the client and the server.

- Upon receiving requests from the client, the WebSocket Server sends corresponding messages to the Server actor for further operations

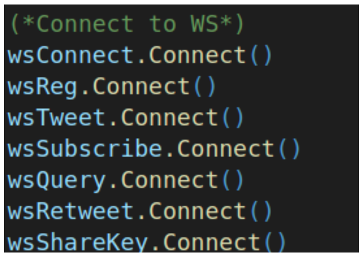


```
type Register () =
    inherit WebSocketBehavior()
    override x.OnMessage message =  // MessageEventArgs -> unit
        // x.IgnoreExtensions <- true
        let info = Json.deserialize<RegisterInfo> message.Data
        if info.UserName = "close" then
            printfn "closing session"
            x.Sessions.CloseSession(x.ID)
        else
            serverActor <! RegisterAccount (info.UserID, info.UserName, info.PublicKey, x.Sessions, x.ID)
```

- On the client side, the following items are implemented **for each Client actor**.
  - Upon spawning each client actor, a **WebSocket instance** corresponding each of the seven services is initiated and connected.

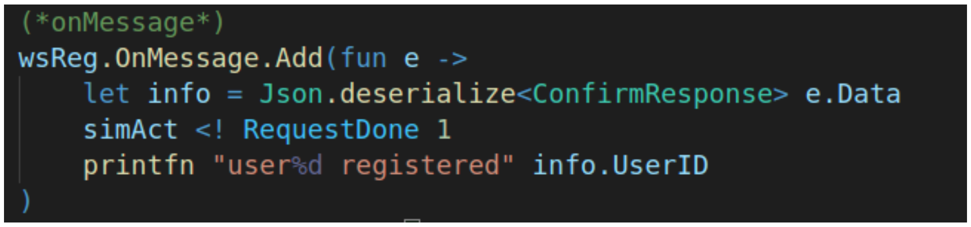


```
let wsReg = new WebSocket("ws://localhost:8080/register")
let wsTweet = new WebSocket("ws://localhost:8080/receivetw")
let wsRetweet = new WebSocket("ws://localhost:8080/receiveretw")
let wsSubscribe = new WebSocket("ws://localhost:8080/subscribe")
let wsQuery = new WebSocket("ws://localhost:8080/query")
let wsConnect = new WebSocket("ws://localhost:8080/connect")
let wsShareKey = new WebSocket("ws://localhost:8080/sharekey")
```



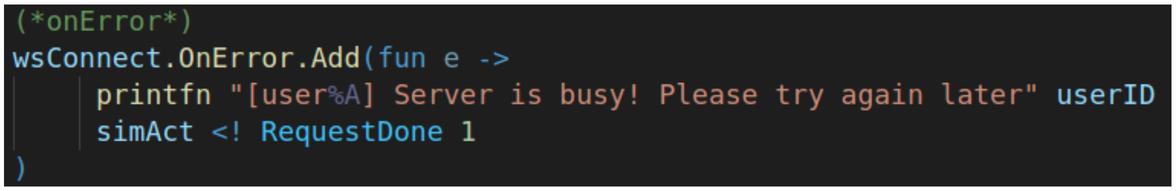
```
(*Connect to WS*)
wsConnect.Connect()
wsReg.Connect()
wsTweet.Connect()
wsSubscribe.Connect()
wsQuery.Connect()
wsRetweet.Connect()
wsShareKey.Connect()
```

  - Then, call-back functions are added to handle responses from the server (i.e., OnMessage) and error messages (i.e., OnError). The following figures provide an example for each case.
    - OnMessage



```
(*onMessage*)
wsReg.OnMessage.Add(fun e ->
    let info = Json.deserialize<ConfirmResponse> e.Data
    simAct <! RequestDone 1
    printfn "user%d registered" info.UserID
)
```

    - OnError



```
(*onError*)
wsConnect.OnError.Add(fun e ->
        printfn "[user%A] Server is busy! Please try again later" userID
        simAct <! RequestDone 1
)
```

4. JSON-based API

Messages are sent back and forth between the server and the clients in the **JSON format**. A third-party package (i.e., FSharp.json) is used to provide the functionalities of **serializing and deserializing** the JSON-format messages.

A. Third-Party Package

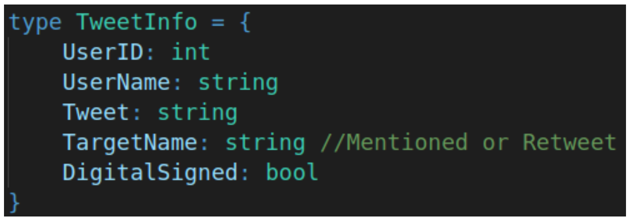
- The **FSharp.json** package is used for the implementation of the JSON-based API

B. Implementation

- Message Types
  - Both the Server folder and the Simulator folder include a file called **Messages.fs**. This file specifies all the necessary message types for the JSON-based API. It also includes the message type for the Simulator, Client, and Server actors. Note that the JSON message types should be the same on both the server side and the client side for them to communicate.
  - The following table specifies all the message types

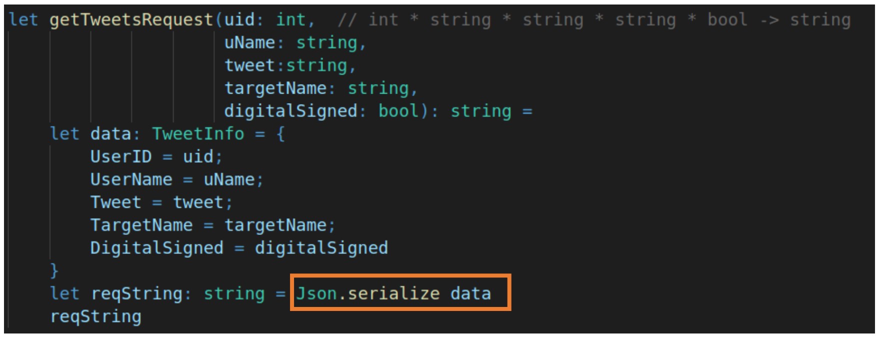
Message Types	Purpose
SharedKeyInfo	Used for establishing a shared secret key between the server and each client.
SignedMessageInfo	Used for sending messages with a digital signature.
RegisterInfo	Used for registration.
ConnectInfo	Used for connection.
SubscribeInfo	Used for subscribing.
QueryInfo	Used for querying.
TweetInfo	Used for sending tweets or re-tweets.
TweetResponse	Used for responding queries with tweets
ConfirmResponse	Used for general confirming.

  - Here is an example for one of the message types



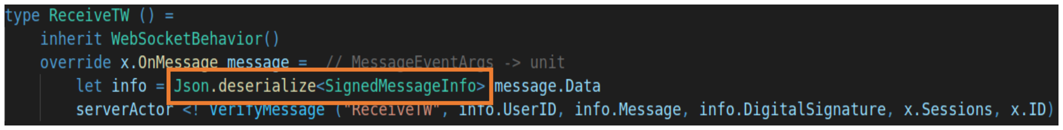
```
type TweetInfo = {
    UserID: int
    UserName: string
    Tweet: string
    TargetName: string //Mentioned or Retweet
    DigitalSigned: bool
}
```

- Serializing and Deserializing
  - Both the Server folder and the Simulator folder include a file called **Util.fs**. This file specifies all the necessary methods for serializing and deserializing JSON-format messages. Note that the methods should be the same on both the server side and the client side for them to communicate. The following figures provide an example for each case.
  - Serializing



```
let getTweetsRequest(uid: int,  // int * string * string * string * bool -> string
                     uName: string,
                     tweet:string,
                     targetName: string,
                     digitalSigned: bool): string =
    let data: TweetInfo = {
        UserID = uid;
        UserName = uName;
        Tweet = tweet;
        TargetName = targetName;
        DigitalSigned = digitalSigned
    }
    let reqString: string = Json.serialize data
    reqString
```

- Deserializing



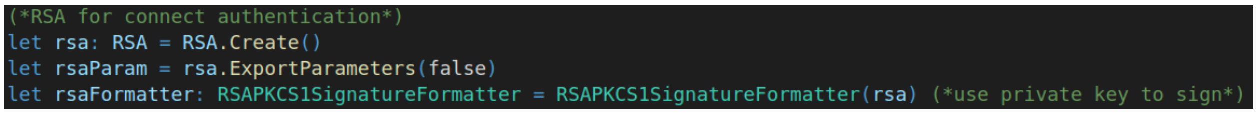
```
type ReceiveTW () =
    inherit WebSocketBehavior()
    override x.OnMessage message =  // MessageEventArgs -> unit
        let info = Json.deserialize<SignedMessageInfo> message.Data
        serverActor <! VerifyMessage ("ReceiveTW", info.UserID, info.Message, info.DigitalSignature, x.Sessions, x.ID)
```

5. Authentication and Cryptography

In this bonus part, asymmetric algorithms provided by .NET's **System.Security.Cryptography** package are used for two purposes as described below.

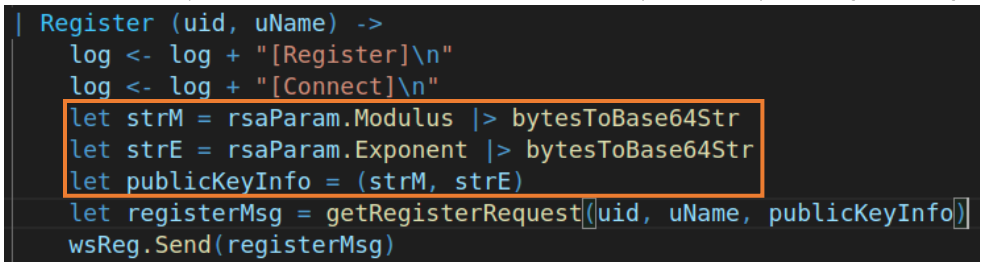
A. Authentication when re-connecting

- **RSA** is used to create a public/private key pair



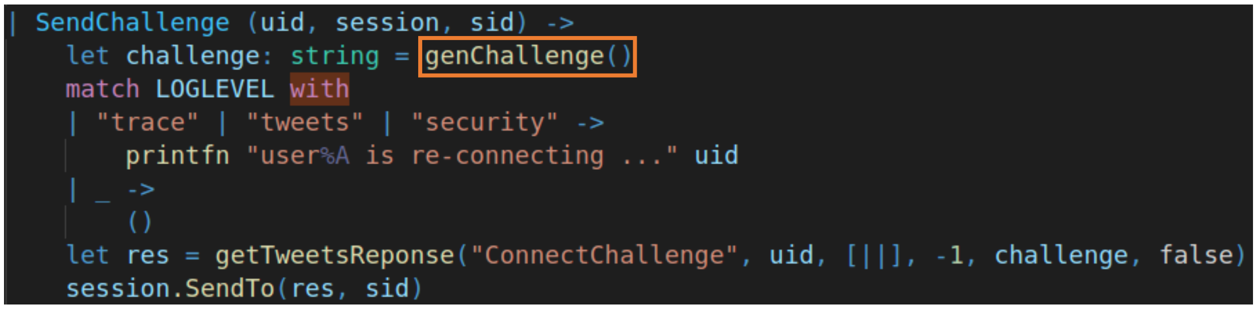
```
(*RSA for connect authentication*)
let rsa: RSA = RSA.Create()
let rsaParam = rsa.ExportParameters(false)
let rsaFormatter: RSAPKCS1SignatureFormatter = RSAPKCS1SignatureFormatter(rsa) (*use private key to sign*)
```

- The client provides the server with his or her public key during the registration



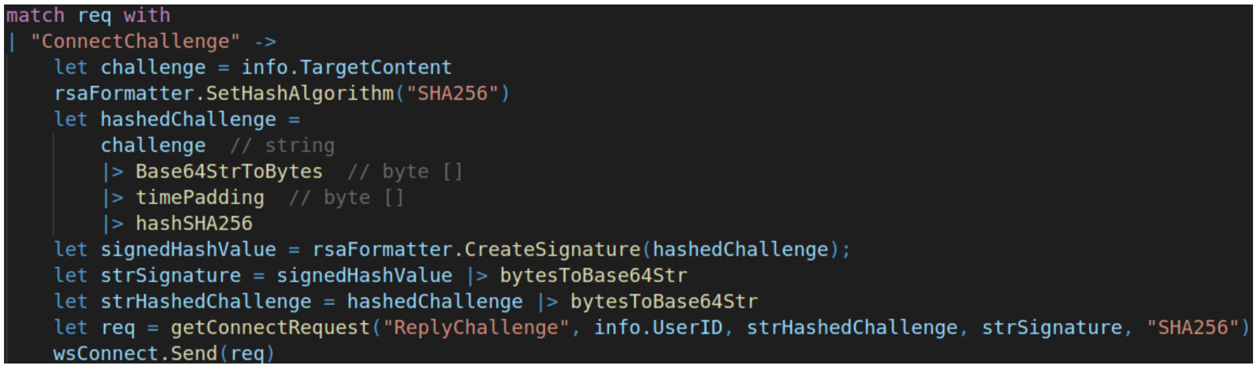
```
| Register (uid, uName) ->
    log <- log + "[Register]\n"
    log <- log + "[Connect]\n"
    let strM = rsaParam.Modulus |> bytesToBase64Str
    let strE = rsaParam.Exponent |> bytesToBase64Str
    let publicKeyInfo = (strM, strE)
    let registerMsg = getRegisterRequest(uid, uName, publicKeyInfo)
    wsReg.Send(registerMsg)
```

- Note that the client automatically connects without authentication right after registration, which is characterized by the AKKA actor message "**FirstConnect**"
- When the client is trying to re-connect, the server sends a 256-bit challenge



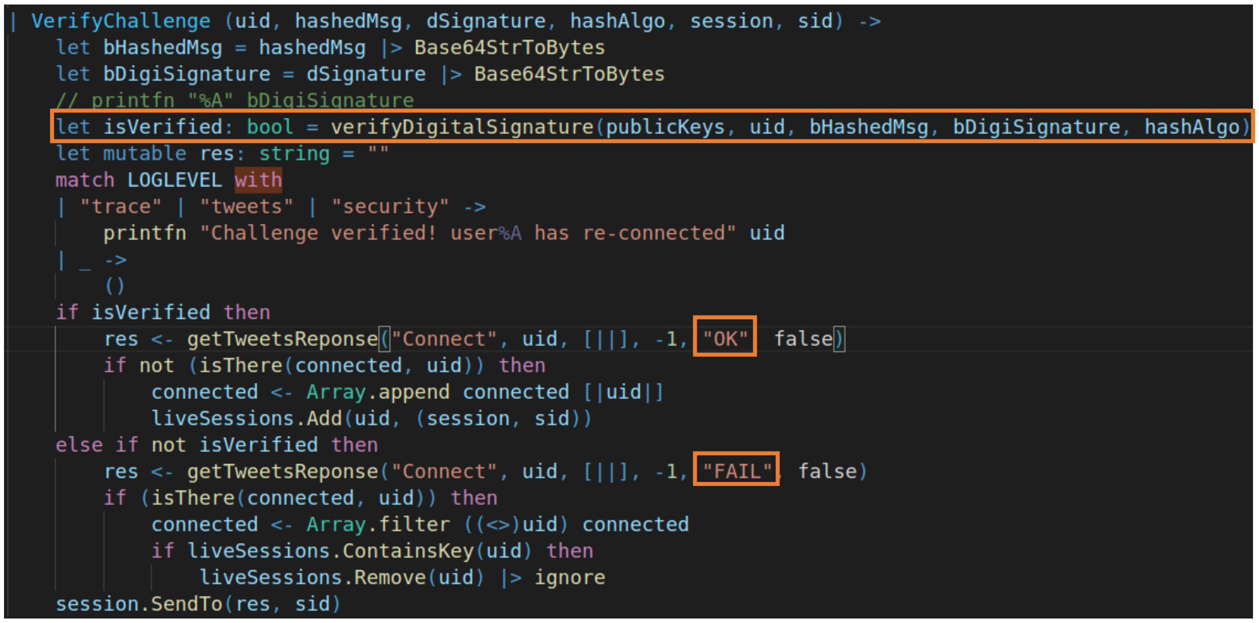
```
| SendChallenge (uid, session, sid) ->
    let challenge: string = genChallenge()
    match LOGLEVEL with
    | "trace" | "tweets" | "security" ->
        printfn "user%A is re-connecting ..." uid
    | _ ->
        ()
    let res = getTweetsReponse("ConnectChallenge", uid, [||], -1, challenge, false)
    session.SendTo(res, sid)
```

- Upon receiving the challenge, the client forms a message containing the challenge, the current time, and digitally signs it.



```
match req with
| "ConnectChallenge" ->
    let challenge = info.TargetContent
    rsaFormatter.SetHashAlgorithm("SHA256")
    let hashedChallenge =
        challenge  // string
        |> Base64StrToBytes  // byte []
        |> timePadding  // byte []
        |> hashSHA256
    let signedHashValue = rsaFormatter.CreateSignature(hashedChallenge);
    let strSignature = signedHashValue |> bytesToBase64Str
    let strHashedChallenge = hashedChallenge |> bytesToBase64Str
    let req = getConnectRequest("ReplyChallenge", info.UserID, strHashedChallenge, strSignature, "SHA256")
    wsConnect.Send(req)
```

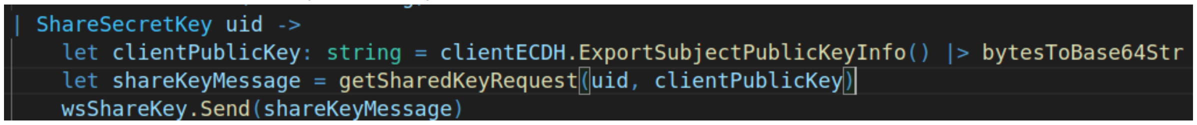
- The server sends “OK” or “FAIL” based on the verification



```
| VerifyChallenge (uid, hashedMsg, dSignature, hashAlgo, session, sid) ->
    let bHashedMsg = hashedMsg |> Base64StrToBytes
    let bDigiSignature = dSignature |> Base64StrToBytes
    // printfn "%A" bDigiSignature
    let isVerified: bool = verifyDigitalSignature(publicKeys, uid, bHashedMsg, bDigiSignature, hashAlgo)
    let mutable res: string = ""
    match LOGLEVEL with
    | "trace" | "tweets" | "security" ->
        printfn "Challenge verified! user%A has re-connected" uid
    | _ ->
        ()
    if isVerified then
        res <- getTweetsReponse("Connect", uid, [||], -1, "OK", false)
        if not (isThere(connected, uid)) then
            connected <- Array.append connected [|uid|]
            liveSessions.Add(uid, (session, sid))
    else if not isVerified then
        res <- getTweetsReponse("Connect", uid, [||], -1, "FAIL", false)
        if (isThere(connected, uid)) then
            connected <- Array.filter ((<>)uid) connected
            if liveSessions.ContainsKey(uid) then
                liveSessions.Remove(uid) |> ignore
    session.SendTo(res, sid)
```

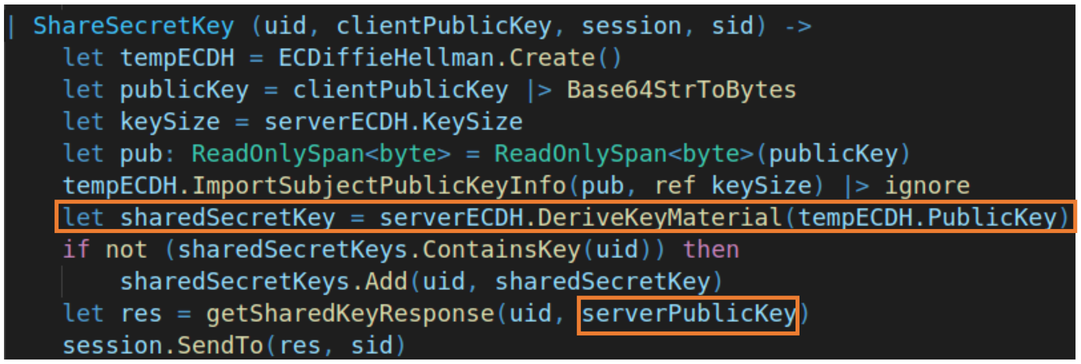
B. <u>Digital Signature for Tweets</u>

- For this part, the Elliptic Curve Diffie-Hellman (**ECDH**) algorithm is used to establish a shared secret between the client and the server. Then, **HMACSHA256** is used to sign the tweets and the shared secret on the server side is used to verify the digital signature
- The client and the server **establish a shared secret using ECDH**
  - The client sends its public key to the server



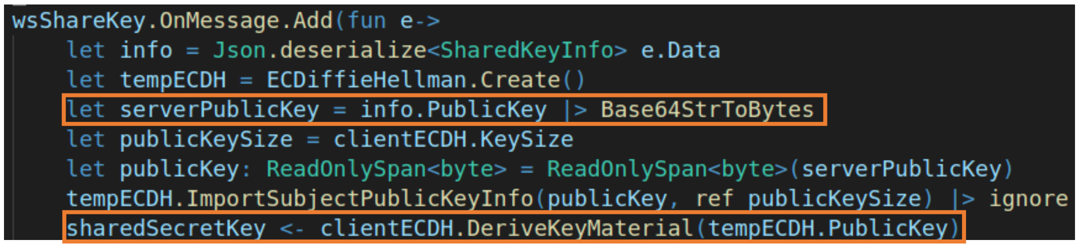
```
| ShareSecretKey uid ->
    let clientPublicKey: string = clientECDH.ExportSubjectPublicKeyInfo() |> bytesToBase64Str
    let shareKeyMessage = getSharedKeyRequest(uid, clientPublicKey)
    wsShareKey.Send(shareKeyMessage)
```

  - The server 1) sends its public key to the client and 2) generates the shared secret based on the client’s public key



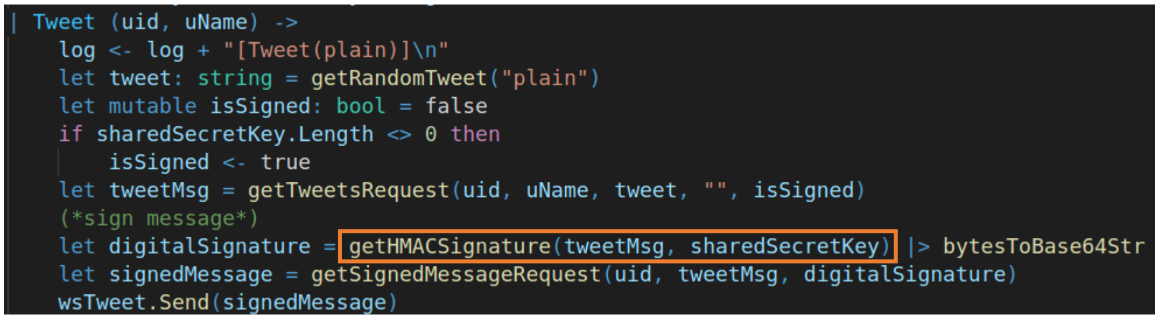
```
| ShareSecretKey (uid, clientPublicKey, session, sid) ->
    let tempECDH = ECDiffieHellman.Create()
    let publicKey = clientPublicKey |> Base64StrToBytes
    let keySize = serverECDH.KeySize
    let pub: ReadOnlySpan<byte> = ReadOnlySpan<byte>(publicKey)
    tempECDH.ImportSubjectPublicKeyInfo(pub, ref keySize) |> ignore
    let sharedSecretKey = serverECDH.DeriveKeyMaterial(tempECDH.PublicKey)
    if not (sharedSecretKeys.ContainsKey(uid)) then
        sharedSecretKeys.Add(uid, sharedSecretKey)
    let res = getSharedKeyResponse(uid, serverPublicKey)
    session.SendTo(res, sid)
```

- The client receives the server’s public key and generates the shared secret key, which should be the same as the one generated on the server side.



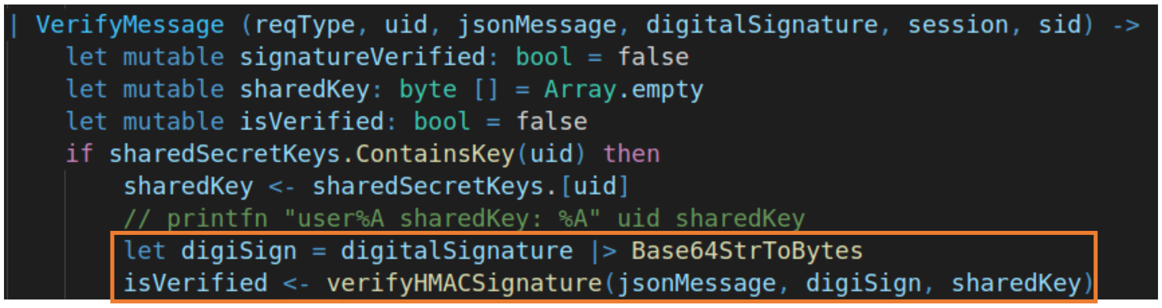
```
wsShareKey.OnMessage.Add(fun e->
    let info = Json.deserialize<SharedKeyInfo> e.Data
    let tempECDH = ECDiffieHellman.Create()
    let serverPublicKey = info.PublicKey |> Base64StrToBytes
    let publicKeySize = clientECDH.KeySize
    let publicKey: ReadOnlySpan<byte> = ReadOnlySpan<byte>(serverPublicKey)
    tempECDH.ImportSubjectPublicKeyInfo(publicKey, ref publicKeySize) |> ignore
    sharedSecretKey <- clientECDH.DeriveKeyMaterial(tempECDH.PublicKey)
```

- After the shared secret key is established, the client signs the tweets and re-tweets



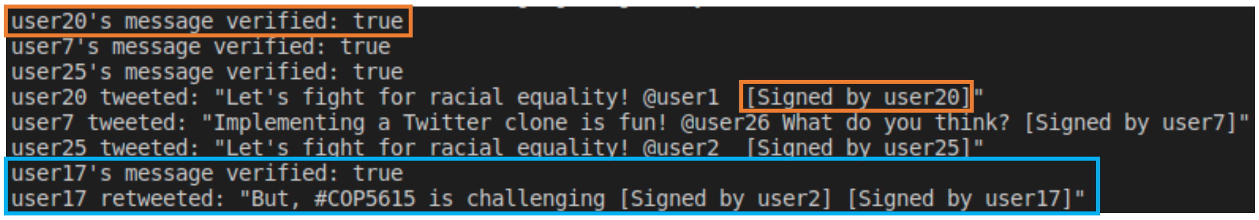
```
| Tweet (uid, uName) ->
    log <- log + "[Tweet(plain)]\n"
    let tweet: string = getRandomTweet("plain")
    let mutable isSigned: bool = false
    if sharedSecretKey.Length <> 0 then
        isSigned <- true
    let tweetMsg = getTweetsRequest(uid, uName, tweet, "", isSigned)
    (*sign message*)
    let digitalSignature = getHMACSignature(tweetMsg, sharedSecretKey) |> bytesToBase64Str
    let signedMessage = getSignedMessageRequest(uid, tweetMsg, digitalSignature)
    wsTweet.Send(signedMessage)
```

- Upon receiving the tweets or re-tweets, the server verifies the digital signature.



```
| VerifyMessage (reqType, uid, jsonMessage, digitalSignature, session, sid) ->
    let mutable signatureVerified: bool = false
    let mutable sharedKey: byte [] = Array.empty
    let mutable isVerified: bool = false
    if sharedSecretKeys.ContainsKey(uid) then
        sharedKey <- sharedSecretKeys.[uid]
        // printfn "user%A sharedKey: %A" uid sharedKey
        let digiSign = digitalSignature |> Base64StrToBytes
        isVerified <- verifyHMACSignature(jsonMessage, digiSign, sharedKey)
```

- If the digital signature is verified, the server puts a **[Signed by USERNAME]** signature on the tweet



```
user20's message verified: true
user7's message verified: true
user25's message verified: true
user20 tweeted: "Let's fight for racial equality! @user1  [Signed by user20]"
user7 tweeted: "Implementing a Twitter clone is fun! @user26 What do you think? [Signed by user7]"
user25 tweeted: "Let's fight for racial equality! @user2  [Signed by user25]"
user17's message verified: true
user17 retweeted: "But, #COP5615 is challenging [Signed by user2] [Signed by user17]"
```

- As an example, we can see in the above log, user20’s message is verified, and then there is a signature **[Signed by user20]** on the client’s message
- The digital signature is attached to the tweet in the case of re-tweeting. As we can see in the last line, user17 retweeted user2’s message with user2’s digital signature on it. Since user17’s message is also verified, another digital signature of user17’s is also attached to the tweet.

6. Akka-based Client Actor

The client in this program is simulated using the AKKA actors. A Simulator actor is implemented to spawn a specified number of Client actors. Then the simulator triggers the clients’ behaviors in a fixed order with a set of numbers as the parameters. These parameters are calculated based on the specified number of clients. More details can be found in the README file of Part I or in the Appendix.

A. Code Structure

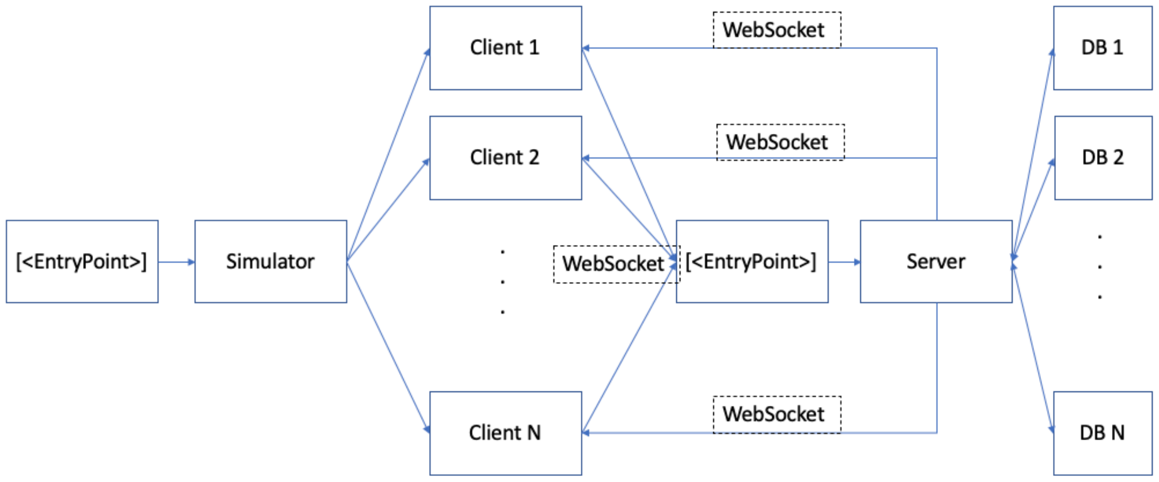
- The Simulator folder

Files	Purpose
Program.fs	[<EntryPoint>] for reading the specified arguments via the terminal and specifying a fixed set of behaviors.
Simulator.fs	Implementation of the Simulator actor that triggers multiple Client actors.
Client.fs	Implementation of the Client actor that sends and receives messages from the server
Util.fs	Implementation of all the common functions and JSON APIs.
Messages.fs	This file specifies all the message types

- The Server folder

Files	Purpose
Program.fs	[<EntryPoint>] for reading the specified arguments via the terminal and receiving messages from the clients via Websockets.
Server.fs	Implementation of the Server actor that communicates with multiple Client actors.
Database.fs	Implementation of the DB actor that stores tweets and other information for each client.
Util.fs	Implementation of all the common functions and JSON APIs.
Messages.fs	This file specifies all the message types.

B. Code Structure and Task Flow



- As shown in the above figure, the Simulator actor triggers N Client actors to communicate with the server back and forth via WebSockets. The server can also deliver live tweets to the Client actors (e.g., mentioned) without waiting for a request.

7. Dependencies

   A. Part I

- Akka.Remote
  - Version: 1.4.10
  - To allow the communication between two independent processes.
- FSharp.json
  - Version: 0.4.0
  - Messages are sent back and forth between two processes in the **JSON format**. This package provides the functionalities of **serializing and deserializing** the JSON format messages.

In addition to the two packages used in part I, I added the following WebSocket package that allows for client-server communication.

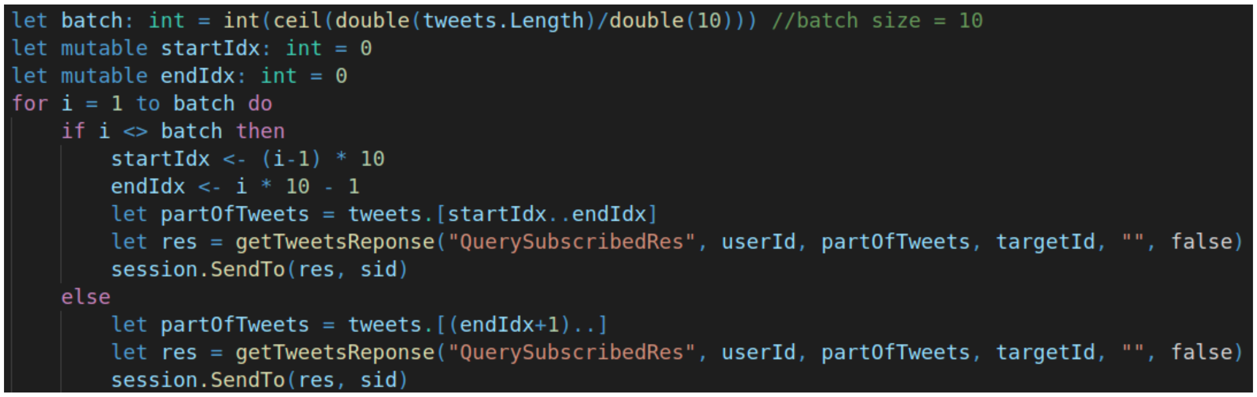
   B. Part II

- System.Security.Cryptography
  - Supported by the net5.0 framework
- WebSocketSharp
  - Version: 1.0.3-rc11

8. Discussion

   A. Solved payload issue

- Previously in part I, I encountered the problem of oversized payload. Therefore, in this version of the program, I modified the mechanism of replying to a tweet query. Now the tweets that are going to be sent based on the query will be divided into batches with each batch containing 10 tweets only. This way the server does not have to send thousands of tweets in a single message when the number of clients scales up. Below is the screenshot of this implementation.



```
let batch: int = int(ceil(double(tweets.Length)/double(10))) //batch size = 10
let mutable startIdx: int = 0
let mutable endIdx: int = 0
for i = 1 to batch do
    if i <> batch then
        startIdx <- (i-1) * 10
        endIdx <- i * 10 - 1
        let partOfTweets = tweets.[startIdx..endIdx]
        let res = getTweetsReponse("QuerySubscribedRes", userId, partOfTweets, targetId, "", false)
        session.SendTo(res, sid)
    else
        let partOfTweets = tweets.[(endIdx+1)..]
        let res = getTweetsReponse("QuerySubscribedRes", userId, partOfTweets, targetId, "", false)
        session.SendTo(res, sid)
```

B. Verifying every single tweets

- This implementation significantly slows down the process, which is expected. Although I didn't provide a systematic analysis for this finding, it is obvious that the time needed for termination increased significantly.

9. Appendix

This appendix provides the information in part I's README as a reference for the readers.

[1] Instructions

A. Testing phase

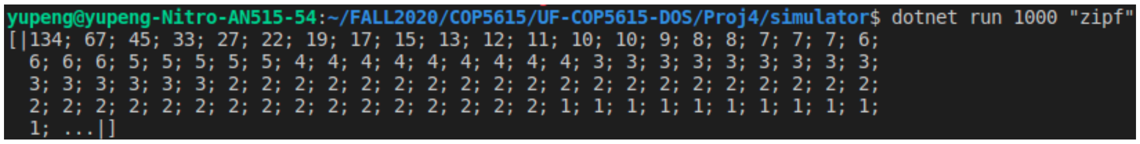
The goal of this phase is to test the correctness of the server APIs. Thus, **there is an option of printing the debugging information to the pre-existing folder "log"**, which is placed under the same directory as the "Client.fs" file. The logging option will print out a .txt file with the name of "log[ID].txt" for each client, so it is recommended that this option should be run with a small number of clients, otherwise there will be a lot of text files written to the "log" folder. The following is the command line for this phase:

- First, navigate to the "**server**" folder and run the following commands:
  - dotnet build
  - dotnet run

  This we keep the process running and listening to requests.
- Then, navigate to the "**simulator**" folder and run the following commands:
  - dotnet build
  - dotnet run [NUMBER_OF_USER] [MODE] [DEBUG]
    - Example: dotnet run 50 "testing" "true"
    - NUMBER_OF_USER: An integer indicating the number of simulated users.
    - MODE: A string that should be set to **"testing"** in this phase.
    - DEBUG: A string that can be set to "true" or "false".

B. Zipf phase

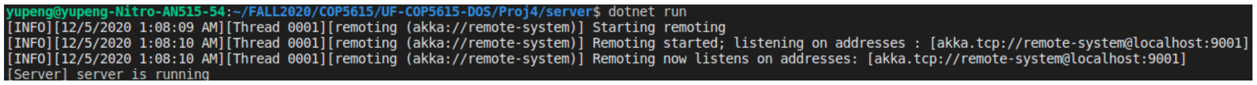
The goal of this phase is to simulate the Zipf distribution ($X \sim Zipf(\alpha, n)$) with the probability mass function: $f(x) = \frac{1}{x^{\alpha} \sum_{i=1}^{n} (\frac{1}{i})^{\alpha}}, where\ x = 1, \dots n$. In the Zipf simulation for this project, n is the number of total registered clients, and the distribution provides **the number of subscribers** for each client. Here is an example when n = 1000 and $\alpha = 1$:



```
yupeng@yupeng-Nitro-AN515-54:~/FALL2020/COP5615/UF-COP5615-DOS/Proj4/simulator$ dotnet run 1000 "zipf"
[|134; 67; 45; 33; 27; 22; 19; 17; 15; 13; 12; 11; 10; 10; 9; 8; 8; 7; 7; 7; 6;
  6; 6; 6; 5; 5; 5; 5; 5; 4; 4; 4; 4; 4; 4; 4; 4; 4; 3; 3; 3; 3; 3; 3; 3; 3; 3;
  3; 3; 3; 3; 3; 3; 2; 2; 2; 2; 2; 2; 2; 2; 2; 2; 2; 2; 2; 2; 2; 2; 2; 2; 2; 2;
  2; 2; 2; 2; 2; 2; 2; 2; 2; 2; 2; 2; 2; 2; 2; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1;
  1; ...|]
```

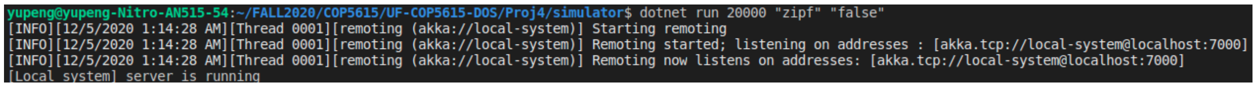
- First, navigate to the "**server**" folder and run the following commands:
  - dotnet build
  - dotnet run

  This we keep the process running and listening to request.
- Then, navigate to the "**simulator**" folder and run the following commands:
  - dotnet build
  - dotnet run [NUMBER_OF_USER] [MODE] [DEBUG]
    - Example: dotnet run 10000 "zipf" "false"
    - NUMBER_OF_USER: An integer indicating the number of simulated users.
    - MODE: A string that should be set to **"zipf"** in this phase.
    - DEBUG: A string that should be set to "false" in this phase
- The terminal will print out the following messages for the server and the simulator respectively to show that the communication path is successfully built:
  - Server:



```
yupeng@yupeng-Nitro-AN515-54:~/FALL2020/COP5615/UF-COP5615-DOS/Proj4/server$ dotnet run
[INFO][12/5/2020 1:08:09 AM][Thread 0001][remoting (akka://remote-system)] Starting remoting
[INFO][12/5/2020 1:08:10 AM][Thread 0001][remoting (akka://remote-system)] Remoting started; listening on addresses : [akka.tcp://remote-system@localhost:9001]
[INFO][12/5/2020 1:08:10 AM][Thread 0001][remoting (akka://remote-system)] Remoting now listens on addresses: [akka.tcp://remote-system@localhost:9001]
[Server] server is running
```

  - Simulator:



```
yupeng@yupeng-Nitro-AN515-54:~/FALL2020/COP5615/UF-COP5615-DOS/Proj4/simulator$ dotnet run 20000 "zipf" "false"
[INFO][12/5/2020 1:14:28 AM][Thread 0001][remoting (akka://local-system)] Starting remoting
[INFO][12/5/2020 1:14:28 AM][Thread 0001][remoting (akka://local-system)] Remoting started; listening on addresses : [akka.tcp://local-system@localhost:7000]
[INFO][12/5/2020 1:14:28 AM][Thread 0001][remoting (akka://local-system)] Remoting now listens on addresses: [akka.tcp://local-system@localhost:7000]
[Local system] server is running
```

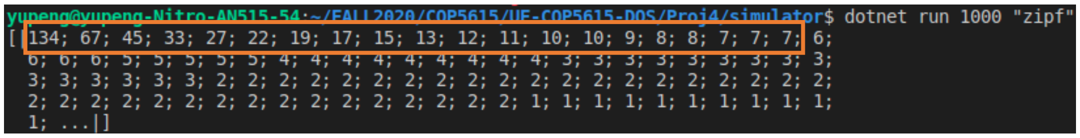
C. Note

- the commands should be **run in the folder by the above order** (i.e., first in the server folder and then in the simulator folder), otherwise the simulator will not be able to find the server.
- When the simulator disconnects, which means that all the clients have terminated their processes, there will be some error messages showing up on the server side that I'm still not able to address. But this should not affect the functionality of the engine.
- **The process has to be start from the beginning to run the simulator again**. What I mean by this is, although the server will keep listening to events, the data stored in the previous run will not be cleaned, so **the server process should also be terminated manually (i.e., Ctrl + C or Cmd + C) to start another run**.
- **The folder log has to exist in the first place to save the log file**. The program does not automatically generate the folder. The reason is that I couldn't figure out how to create a folder with "write" permission. I can only create a new folder with "ReadOnly" attribute, so the workaround is to create the folder named "log" in advance.

[2] Functionality

The simulator is able to test 11 functionalities. Based on my design, the total number of requests depends on the number of clients, which in my opinion is similar to the real-world case. The following items describe how the numbers of requests are decided for each function, where **N** is the **number of clients**:

- Register users
  - The number of registered users equals **N**.
- Send tweets
  - The number of tweets sent equals **N**.
- Send tweets with hashtags
  - The number of tweets sent with hashtags equals $\left\lceil \frac{N}{4} \right\rceil$.
  - This number is larger than the number of tweets with mentions because I assume that people use hashtags more often than mentioning others based on my personal experience with the real Twitter.
- Send tweets with mentions
  - The number of tweets sent with mentions equals $\left\lceil \frac{N}{10} \right\rceil$.
- Subscribe to celebrities
  - A celebrity is defined as the client who has many subscribers.
  - The number of celebrities is $\left\lceil \frac{N}{50} \right\rceil$.
  - For example, in the Zipf distribution simulation with N = 1000, 20 clients will be assigned as the celebrity and will be followed by a certain number of clients. The specific number is selected from the first 20 entries of the Zipf distribution array calculated based on the formula provided in the previous section:



```
yupeng@yupeng-Nitro-AN515-54:~/FALL2020/COP5615/UF-COP5615-DOS/Proj4/simulator$ dotnet run 1000 "zipf"
[|134; 67; 45; 33; 27; 22; 19; 17; 15; 13; 12; 11; 10; 10; 9; 8; 8; 7; 7; 7; 6;
  6; 6; 6; 5; 5; 5; 5; 5; 4; 4; 4; 4; 4; 4; 4; 4; 4; 3; 3; 3; 3; 3; 3; 3; 3; 3;
  3; 3; 3; 3; 3; 3; 2; 2; 2; 2; 2; 2; 2; 2; 2; 2; 2; 2; 2; 2; 2; 2; 2; 2; 2; 2;
  2; 2; 2; 2; 2; 2; 2; 2; 2; 2; 2; 2; 2; 2; 2; 2; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1;
  1; ...|]
```

- Send Retweets
  - The number of retweets sent equals $\left\lceil \frac{N}{20} \right\rceil$.
  - Retweets are chosen from the tweets received by the client who queries them. The intuition is that before retweeting other people’s tweets, the client has to query the tweets first to know exactly which tweet he or she wants to retweet, which is similar to the behavior of surfing other people’s tweets first and then retweeting the interesting ones.
  - **Celebrities send more tweets and retweets**.
- Query tweets with hashtags
  - $\left\lceil \frac{N}{4} \right\rceil$ clients are randomly selected to query tweets with a certain hashtag, which is also randomly selected.
- Query tweets with mentions
  - $\left\lceil \frac{N}{10} \right\rceil$ clients are randomly selected to query tweets that mention himself of herself (i.e., my mentions).
- Query the tweets sent by the subscribed users
  - $\left\lceil \frac{N}{10} \right\rceil$ clients are randomly selected to query tweets sent by one of the users that he or she subscribed to.

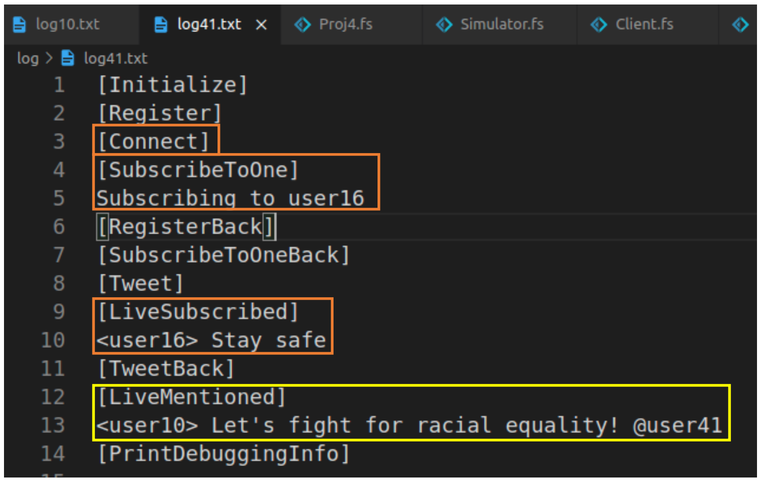
- Connect and Disconnect
    - Upon registering, all the users are by default connected to the server, which is usually the case in the real world.
    - In the simulation, $\left\lceil \frac{\boldsymbol{N}}{\mathbf{10}} \right\rceil$ of the clients will disconnect first, and then after simulating some of the other functionalities, $\left\lceil \frac{\boldsymbol{N}}{\mathbf{10}} \right\rceil$ of the clients will connect to the server again.
    - When the client is connected, the tweets with the following two conditions will be delivered automatically without querying:
      1) the tweets sent by a subscribed client and 2) the tweets that mention the client.

[3] Performance

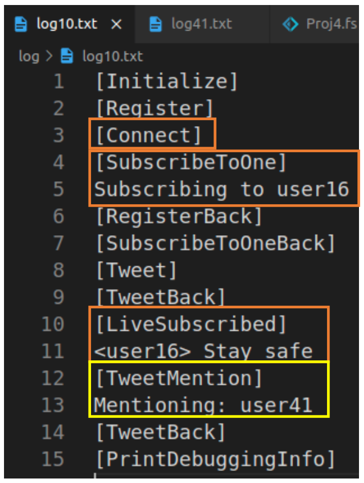
A. Correctness

A small number of users (e.g., 50) is used to test the correctness of the engine API with the help of the debugging files written to the **"log"** folder as described above. We can look at several examples in one of the trials:

- Some general explanations of the symbols
    - Square brackets include the operation that is performed.
    - The <> brackets include the user who tweeted the tweet that follows the brackets.
    - Each client[ID] has the name of user[ID]. In the real-world case, the name user[ID] should be replaced with real names such as Dennis or Alex.
    - As mentioned above, the file name "log[ID].txt" saves the operation log of client[ID].
- Live delivery for subscribed and mentioned
    - As shown in the figures below, in the scenario of 50 people, only one celebrity will be selected, which is **user16** in this case. We can see that client41 and client10 never disconnect from the server, and that they both subscribe to user16. Therefore, they both received the **[LiveSubscribed]** message when user16 tweeted "Stay safe". Also, since client41 is connected when client10 sends tweet with @user41, client41 received the **[LiveMentioned]** message with the tweet: "<user10> Let's fight for racial equality! @user41 "

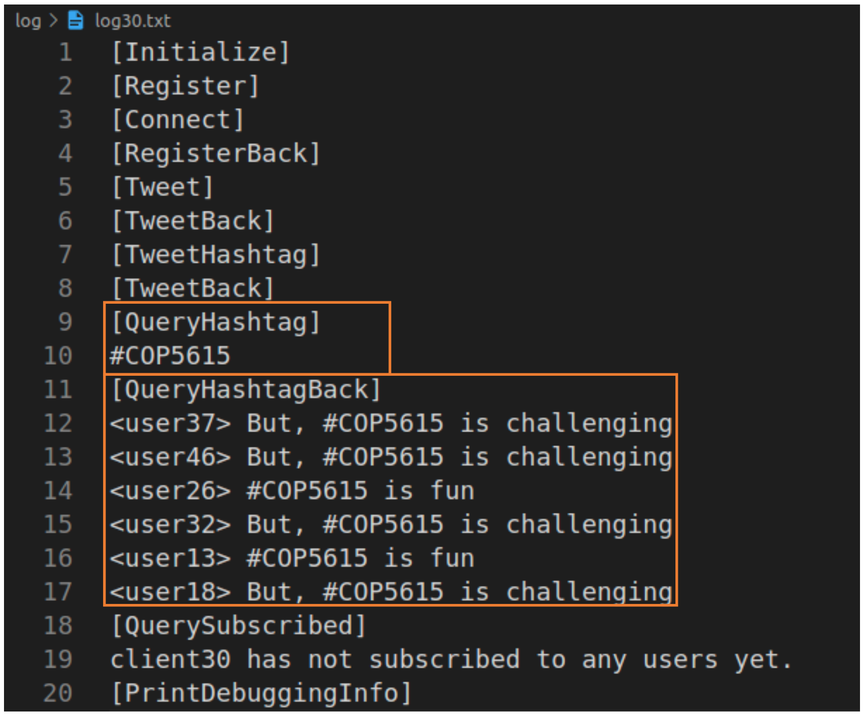


```
[Initialize]
[Register]
[Connect]
[SubscribeToOne]
Subscribing to user16
[RegisterBack]
[SubscribeToOneBack]
[Tweet]
[LiveSubscribed]
<user16> Stay safe
[TweetBack]
[LiveMentioned]
<user10> Let's fight for racial equality! @user41
[PrintDebuggingInfo]
```



```
[Initialize]
[Register]
[Connect]
[SubscribeToOne]
Subscribing to user16
[RegisterBack]
[SubscribeToOneBack]
[Tweet]
[TweetBack]
[LiveSubscribed]
<user16> Stay safe
[TweetMention]
Mentioning: user41
[TweetBack]
[PrintDebuggingInfo]
```

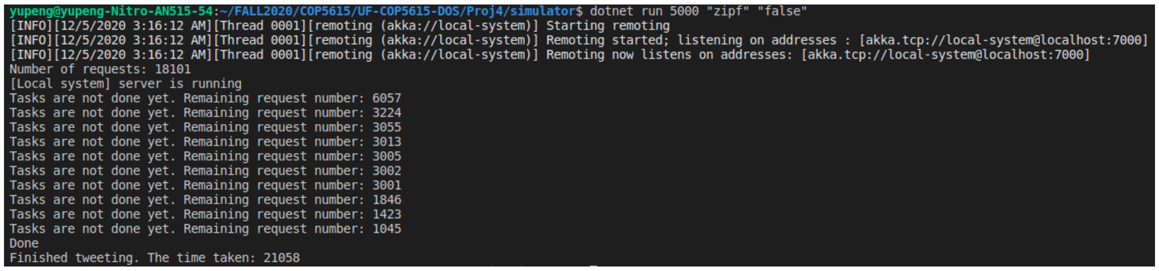
- As shown in the figure below, hashtags (and mentions) can be placed in any positions, which is similar to the real Twitter. The figure also shows the functionality of querying tweets with a specific hashtag, which **is #COP5615** in this case.



```
[Initialize]
[Register]
[Connect]
[RegisterBack]
[Tweet]
[TweetBack]
[TweetHashtag]
[TweetBack]
[QueryHashtag]
#COP5615
[QueryHashtagBack]
<user37> But, #COP5615 is challenging
<user46> But, #COP5615 is challenging
<user26> #COP5615 is fun
<user32> But, #COP5615 is challenging
<user13> #COP5615 is fun
<user18> But, #COP5615 is challenging
[QuerySubscribed]
client30 has not subscribed to any users yet.
[PrintDebuggingInfo]
```

B. Setting

- The order of each simulated functions
  - Register N clients (connected) → subscribe to $\left\lceil \frac{N}{50} \right\rceil$ celebrities → $\left\lceil \frac{N}{10} \right\rceil$ disconnect → N send tweets → $\left\lceil \frac{N}{4} \right\rceil$ send tweets with hashtags → $\left\lceil \frac{N}{10} \right\rceil$ connect → $\left\lceil \frac{N}{10} \right\rceil$ send tweets with mentions → $\left\lceil \frac{N}{4} \right\rceil$ query tweets with a certain hashtag → $\left\lceil \frac{N}{10} \right\rceil$ query my mentions → $\left\lceil \frac{N}{20} \right\rceil$ retweet → $\left\lceil \frac{N}{10} \right\rceil$ query the tweets subscribed to → Done
- **Stopping criteria**
  The simulator keeps track of the total number of requests sent. The server replies the "Back" messages to the client who sent the request after the request is performed. For example, for the **[QueryHashtag]** message, the corresponding "Back" message is the **[QueryHashtagBack]** message. Then, upon receiving the "Back" messages, the clients will notify the simulator to reduce the recorded number of requests. The program stops after all the requests are completed, as shown in the figure below.

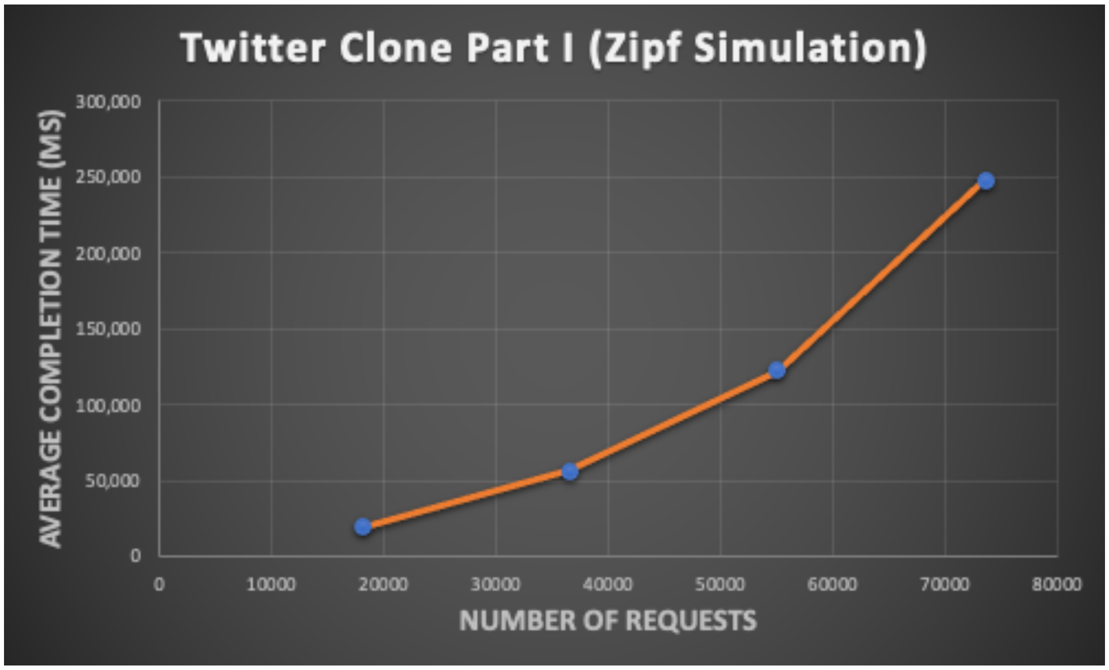


```
yupeng@yupeng-Nitro-AN515-54:~/FALL2020/COP5615/UF-COP5615-DOS/Proj4/simulator$ dotnet run 5000 "zipf" "false"
[INFO][12/5/2020 3:16:12 AM][Thread 0001][remoting (akka://local-system)] Starting remoting
[INFO][12/5/2020 3:16:12 AM][Thread 0001][remoting (akka://local-system)] Remoting started; listening on addresses : [akka.tcp://local-system@localhost:7000]
[INFO][12/5/2020 3:16:12 AM][Thread 0001][remoting (akka://local-system)] Remoting now listens on addresses: [akka.tcp://local-system@localhost:7000]
Number of requests: 18101
[Local system] server is running
Tasks are not done yet. Remaining request number: 6057
Tasks are not done yet. Remaining request number: 3224
Tasks are not done yet. Remaining request number: 3055
Tasks are not done yet. Remaining request number: 3013
Tasks are not done yet. Remaining request number: 3005
Tasks are not done yet. Remaining request number: 3002
Tasks are not done yet. Remaining request number: 3001
Tasks are not done yet. Remaining request number: 1846
Tasks are not done yet. Remaining request number: 1423
Tasks are not done yet. Remaining request number: 1045
Done
Finished tweeting. The time taken: 21058
```

C. <u>Table for the Zipf distribution simulation</u>

dotnet run NUMBER_OF_CLIENTS “zipf” “false”

Number of clients	5,000	10,000	15,000	20,000
Number of requests	18,101	36,504	54,997	73,534
Average completion time (ms)	19,667	56,368	122,923	248,320



D. <u>Maximum number of clients and requests</u>

Based on my design, the maximum number of clients and the corresponding requests are **20,000 clients and 73,534 requests**. When I tried to increase the number of simulated clients, I received the error message of “**oversized payload**”. According to the error message, the messages I tried to send exceeded the maximum allowed size of 128,000 bytes. My hypothesis is that the queries asking for multiple tweets (e.g., querying tweets with a certain hashtag) are the main factors which caused the problem.