TP CRÉER UNE APPLICATION DE CHAT DANS LE LANGAGE DE PROGRAMMATION GO

Présentation du TP

Nous avons vu beaucoup de notions et je conçois qu'il soit difficile dans certains cas de comprendre l'intérêt de certaines notions à travers des exemples basiques, et donc c'est pour ces raisons que j'ai décidé de vous concocter un Tp pour utiliser les éléments que nous avons pu voir dans les chapitres précédents.

Le but de ce tp, est de créer une **application de discussion**, permettant de communiquer avec la personne que l'on souhaite et peu importe l'endroit où on se trouve.

Les exigences

Voici une liste d'exigences pour ce tp :

- Le serveur accepte plusieurs utilisateurs
- Le serveur possède un fichier de logs avec les connexions entrantes et sortantes.
- L'utilisateur doit définir un pseudo avant de pouvoir se connecter et le pseudo ne doit dépasser 20 caractères.
- L'utilisateur ne peut pas prendre un pseudo déjà utilisé

- L'utilisateur doit envoyer le message au serveur et le message sera diffusé à tous les autres utilisateurs.
- Utilisez la notion de classe avec des structures (une structure pour le serveur et une autre pour le client)
- Prévenir les autres utilisateurs qu'un utilisateur s'est connecté/déconnecté
- Le client et le serveur communiquent via le protocole **TCP**

Quelques conseils pour bien démarrer

Je ne vais pas vous laissez sans aucunes informations (sauf si vous le souhaitez, dans ce cas lancez vous sur le tp directement), ci-dessous je vais vous présenter les étapes à suivre pour créer un serveur et plusieurs clients en utilisant la bibliothèque `net`

Le serveur

Je vous ai mis des commentaires sur toutes les lignes de code, mais je vous tout de même vous fournir plus tard quelques explications en plus. Commencez d'abords par créer un fichier et nommez le **server.go**.

Voici à quoi il doit ressembler :

```
package main

import (
    "fmt"
    "net"
)

func gestionErreur(err error) {
    if err != nil {
        panic(err)
    }
```

```
}

const (
    IP   = "127.0.0.01" // IP local
    PORT = "3569"       // Port utilisé
)

func main() {

    fmt.Println("Lancement du serveur ...")

    // on écoute sur le port 3569
    ln, err := net.Listen("tcp", fmt.Sprintf("%s:%s", IP, PORT))
    gestionErreur(err)

    // On accepte les connexions entrantes sur le port 3569
    conn, err := ln.Accept()
    if err != nil {
        panic(err)
    }

    // Information sur les clients qui se connectent
    fmt.Println("Un client est connecté depuis", conn.RemoteAddr())

    gestionErreur(err)

    // boucle pour toujours écouter les connexions entrantes (ctrl-c pour quitter)
    for {
        // On écoute les messages émis par les clients
        buffer := make([]byte, 4096)       // taille maximum du message qui sera envo
        length, err := conn.Read(buffer)   // lire le message envoyé par client
        message := string(buffer[:length]) // supprimer les bits qui servent à rien e

        if err != nil {
            fmt.Println("Le client s'est déconnecté")
        }

        // on affiche le message du client en le convertissant de byte à string
        fmt.Print("Client:", message)

        // On envoie le message au client pour qu'il l'affiche
        conn.Write([]byte(message + "\n"))
    }
}
```

on a commencé par importer la bibliothèque `net`, c'est cette bibliothèque qui va nous aider, à créer notre client et serveur

```
ln, err := net.Listen("tcp", fmt.Sprintf("%s:%s", IP, PORT))
```

La fonction `Listen()` permet de créer un serveur, elle prend comme premier paramètre le protocole à utiliser ici on utilise le protocole TCP/IP sans rentrer dans les détails c'est un protocole qui va gérer pour nous les règles de communication. Ensuite comme deuxième paramètre elle prend une valeur de type string sur laquelle on associera notre IP (ici c'est l'ip local de notre machine) et notre port (ici 3569)

```
conn, err := ln.Accept()
```

La fonction `Listen()` nous a permis de créer notre serveur sauf que pour le moment il n'accepte aucun client encore. De ce fait on utilise la fonction `Accept()` pour accepter les connexions entrantes.

Un bon serveur se doit d'être disponible 24h/24h 7j/7j "Pas de serveur, pas de clients " c'est pour cela qu'on utilise la boucle `for` pour être toujours à l'écoute des connexions entrantes.

```
buffer := make([]byte, 4096)
```

la taille du buffer est de de 4096, donc le message qu'on recevra du client ne peut dépasser les 4096 bits, le surplus ne sera pas reçu par le client (il est possible d'éviter ce problème avec la fonction `NewReader()` de la bibliothèque `bufio`, je vais utiliser cette méthode sur le client)

```
>message := string(buffer[:length])
```

Si notre message est de taille 16 et notre buffer de taille 4096, il serait plus approprié d'adapter le buffer selon la taille du message en supprimant le surplus. Cette étape est très importante si vous souhaitez faire des comparaisons de votre message avec une autre string (je pense notamment à la vérification du pseudo par le serveur).

Enfin on ne peut envoyer et ne recevoir que des bits (des 0 ou 1) via le protocole TCP (tous les protocoles d'ailleurs), il est donc important de convertir les messages avant de les envoyer de string en byte et inversement quand on souhaite les afficher sur notre écran.

Le client

Côté client ça reste un peu près le même code avec quelques petites modifications.

Créez un fichier et nommez le **client.go** avec le code suivant :

```
package main

import (
 "bufio"
 "fmt"
 "net"
 "os"
)

func gestionErreur(err error) {
 if err != nil {
  panic(err)
 }
}

const (
 IP   = "127.0.0.01" // IP local
 PORT = "3569"       // Port utilisé
)

func main() {

 // Connexion au serveur
 conn, err := net.Dial("tcp", fmt.Sprintf("%s:%s", IP, PORT))
 gestionErreur(err)

 for {
  // entrée utilisateur
  reader := bufio.NewReader(os.Stdin)
  fmt.Print("client: ")
  text, err := reader.ReadString('\n')
  gestionErreur(err)

  // On envoie le message au serveur
  conn.Write([]byte(text))
```

```
  // On écoute tous les messages émis par le serveur et on rajouter un retour à la li
  message, err := bufio.NewReader(conn).ReadString('\n')
  gestionErreur(err)

  // on affiche le message utilisateur
  fmt.Print("serveur : " + message)
 }
}
```

```
conn, err := net.Dial("tcp", fmt.Sprintf("%s:%s", IP, PORT))
```

Ici on n'utilise pas la fonction `Listen()` mais la fonction `Dial()` qui nous permet créer un client et de le connecter à notre serveur, le reste du code ressemble beaucoup au code du serveur.

Résultat :

Côté serveur :

```
> go run server.go
Lancement du serveur ...
Un client est connecté depuis 127.0.0.1:448
Client:salut
Client:ça va ?
Client:je parle seul :'(
Le client s'est déconnecté
```

Côté client :

```
> go run client.go
client: salut
serveur : salut
client: ça va ?
serveur : ça va ?
client: je parle seul :'(
serveur : je parle seul :'(
client: ^Csignal: interrupt
```

Gérer plusieurs clients

Vous pensez que c'est fini ? Je vous assure que non car si vous souhaitez rajouter un deuxième client voila ce qui se passe :

```
> go run client.go
client: test
```

On touche à la limite de ce type de serveur, il n'est pas capable de gérer **simultanément** plusieurs clients pour pallier ce problème il suffit de créer dans notre serveur une **goroutine** par client connecté. Voici à quoi va ressembler notre serveur en rajoutant cette fonctionnalité :

```
package main

import (
    "bufio"
    "fmt"
    "net"
)

func gestionErreur(err error) {
    if err != nil {
        panic(err)
    }
}

const (
    IP   = "127.0.0.01"
    PORT = "3569"
)

func read(conn net.Conn) {
    message, err := bufio.NewReader(conn).ReadString('\n')
    gestionErreur(err)

    fmt.Print("Client:", string(message))

}

func main() {

    fmt.Println("Lancement du serveur ...")

    ln, err := net.Listen("tcp", fmt.Sprintf("%s:%s", IP, PORT))
    gestionErreur(err)
```

```
    var clients []net.Conn // tableau de clients

    for {
        conn, err := ln.Accept()
        if err == nil {
            clients = append(clients, conn) //quand un client se connecte on le rajou
        }
        gestionErreur(err)
        fmt.Println("Un client est connecté depuis", conn.RemoteAddr())

        go func() { // création de notre goroutine quand un client est connecté
            buf := bufio.NewReader(conn)

            for {
                name, err := buf.ReadString('\n')

                if err != nil {
                    fmt.Printf("Client disconnected.\n")
                    break
                }
                for _, c := range clients {
                    c.Write([]byte(name)) // on envoie un message à chaque client
                }
            }
        }()
    }
}
```

Il faut savoir que par défaut la fonction `Accept()` est bloquante, elle ne s'exécutera seulement si le serveur reçoit une nouvelle connexion. Une goroutine est alors créée propre à chaque nouveau client connecté connexion.

Je vais créer deux clients et voici le résultat obtenu :

Côté serveur :

```
> go run server.go
Lancement du serveur ...
Un client est connecté depuis 127.0.0.1:44976
Un client est connecté depuis 127.0.0.1:44980
```

Client 1 :

```
> go run client.go
client: slt
```

```
serveur : slt
client: toto
serveur : ah
```

Client 2 :

```
> go run client.go
client: ah
serveur slt
client: tata
serveur ah
```

Le serveur gère bien plusieurs clients mais par contre côté clients c'est du n'importe au niveau de l'affichage des messages .

Le serveur envoie bel et bien les données aux clients, mais les clients ne semblent pas bonnement synchroniser l'affichage des messages. Pour remédier ce problème il suffit de créer deux goroutines dans notre client, une goroutine pour gérer la réception des données et une goroutine pour gérer l'envoi des données.

Voici à quoi va ressembler notre nouveau client :

```
package main

import (
    "bufio"
    "fmt"
    "net"
    "os"
    "sync"
)


func gestionErreur(err error) {
    if err != nil {
        panic(err)
    }
}

const (
    IP   = "127.0.0.01" // IP local
    PORT = "3569"       // Port utilisé
)
```

```
func main() {

    var wg sync.WaitGroup

    // Connexion au serveur
    conn, err := net.Dial("tcp", fmt.Sprintf("%s:%s", IP, PORT))
    gestionErreur(err)

    wg.Add(2)

    go func() { // goroutine dédiée à l'entrée utilisateur
        defer wg.Done()
        for {
            reader := bufio.NewReader(os.Stdin)
            text, err := reader.ReadString('\n')
            gestionErreur(err)

            conn.Write([]byte(text))
        }
    }()

    go func() { // goroutine dédiée à la reception des messages du serveur
        defer wg.Done()
        for {
            message, err := bufio.NewReader(conn).ReadString('\n')
            gestionErreur(err)

            fmt.Print("serveur : " + message)
        }
    }()

    wg.Wait()

}
```

Résultat :

Client 1 :

```
> go run server.go
slt
serveur : slt
serveur : hoho
serveur : haha
serveur : bla bla
coco
serveur : coco
```

Client 2 :

```
> go run server.go
hoho
serveur : hoho
haha
serveur : haha
bla bla
serveur : bla bla
serveur : coco
```

à vous de jouer

Vous ne vous êtes peut être pas rendu compte mais mais on vient de créer une application de chat ! Par contre ça ne respecte pas encore nos exigences ;). Je vous ai passé les informations nécessaires pour mener à bien ce TP. Vous n'allez pas être noté ! Donc prenez le temps d'utiliser au maximum les notions vues sur les chapitres précédents et bien sûr amusez-vous :).

Solution

Mon petit message

Avant de vous partager le code, je tiens à préciser plusieurs choses.

Tout d'abord mon code sera en anglais, la raison est que j'ai l'habitude de coder et d'écrire mes documentations en anglais, je vous conseille de faire pareil car ça va vous permettre d'apprendre la langue anglaise avec de nouveaux termes techniques anglais, cette langue est très importante car la majorité des réponses de vos recherches sur un moteur de recherche (Google ou autres) seront en anglais.

Deuxièmement, je ne vais pas vous expliquer comme j'ai l'habitude de faire ligne par ligne mon code. vous avez largement les connaissances nécessaires pour comprendre mon code mais Je vous ai tout de même mis des commentaires sur

certaines lignes de code et sur toutes les méthodes de mes structures.

Troisièmement, il existe différentes façons de faire ce type d'application, donc n'hésitez pas à le modifier selon votre guise.

Voici le lien pour télécharger mon code ici

Lancer le programme

Avant de lancer mon code, voici à quoi doit ressembler votre arborescence :

```
$GOPATH/src/
|___ chat-application
|   |___ client
|   |
|___ client.go
|
|___ server
|
|___ server.go

|___ main.go
```

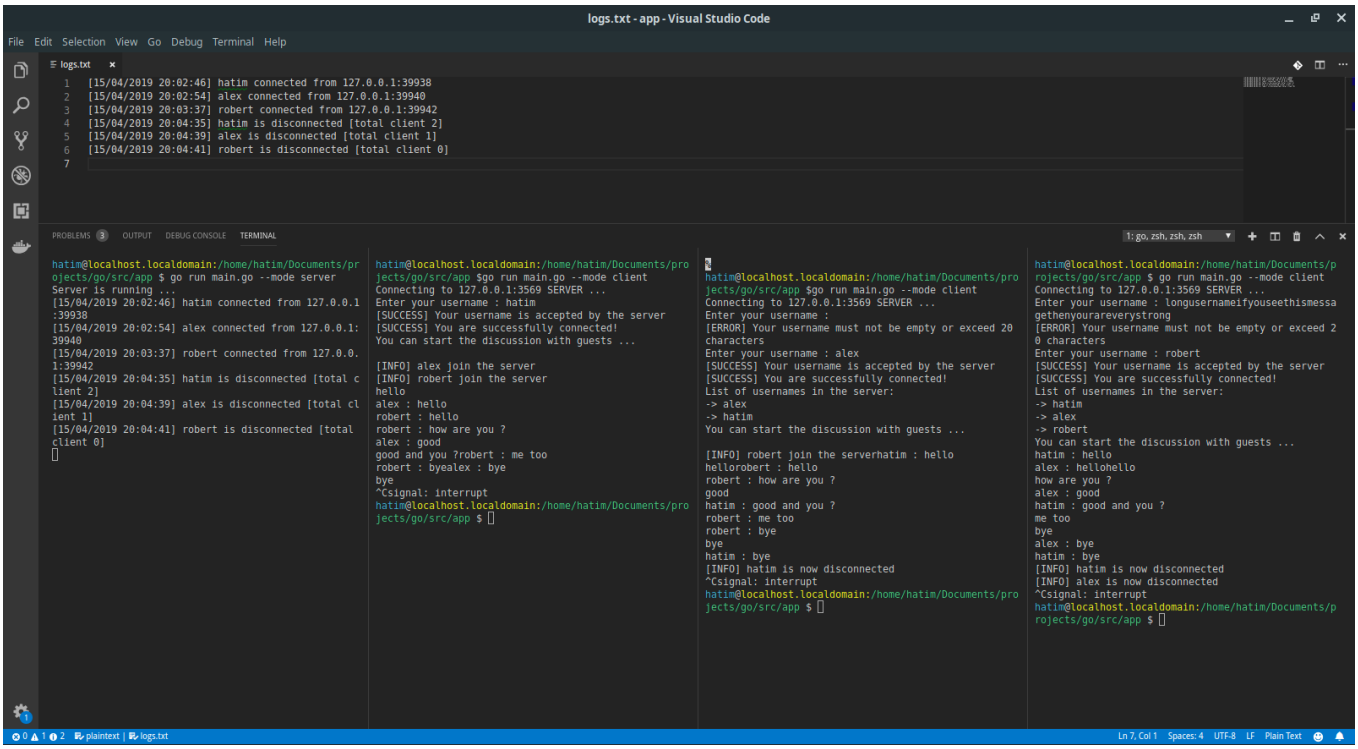
Lancer le serveur :

```
go run main.go --mode server
```

Lancer le client :

```
go run main.go --mode client
```

Screenshot



Agrandir l'image

Compiler votre programme pour le Partager !

Pour compiler votre programme afin de le partager avec les autres utilisateurs il suffit de lancer la commande suivante :

```
go build main.go
```

Si vous êtes sur windows ça vous créera un fichier **main.exe** et un fichier **main** sur linux. Une fois compilé les utilisateurs qui utiliseront votre programme n'auront pas besoin d'installer le compilateur go vu que sera un fichier binaire.