Programmation avec les sockets, en go

1 Généralités sur les sockets, en go

`go` offre le package `"net"` pour la programmation réseau. Il faut donc importer ce package en plus des packages de base comme `fmt`, `os` et `bufio` pour les entrées-sorties.

```
import (
    "net" // for socket resources
    "os"  // for OS resources
    "fmt" // general formatting
)
```

2 Principales fonctions

Côté Serveur : Création d'un socket, puis écoute des connexions La fonction `Listen` crée un nouveau socket en précisant son type de protocole (tcp,...), son adresse IP et son port. La fonction `Listen` renvoie :
- un socket d'écoute, sur lequel le serveur attend les demandes de connexions des clients, et
- un code d'erreur qui est null ou non null (en cas erreur).

```
    // attempt to create a connection tcp sock,  and bind it to serverIP+port
    sockListener, errCode := net.Listen("tcp", "serverIP:port")
    // if errCode != nil { // there is an error
    ...
```

Pour les tests, utiliser un numéro de port non réservé (au moins supérieur à 4096).

Côté Serveur : fonction d'acceptation des connexions Après avoir effectué `listen`, un serveur en mode connecté peut attendre une demande de connexion des clients (avec `Accept`).

La fonction **`Accept`** prend à chaque fois la première demande de connexion en attente. Elle crée un nouveau socket ayant les mêmes propriétés que la socket d'écoute. S'il n'y a pas de demande en attente, `Accept` est blocante. Le descripteur du nouveau socket créé est retourné comme résultat.

En retour, la fonction fournit un socket sur lequel client et server vont échanger ; et aussi un code d'erreur qui est null ou non null (en cas erreur).

```
    // try to accept clients on the connection sockect => communic socket
    commSock, err := sockListener.Accept()
    if err != nil {
      ...
```

Côté Client : fonction de connexion à une machine via un socket Dans le mode connecté, un client doit se connecter au serveur avant les interactions. La fonction **`Dial`** du package **`net`** établit une connexion avec un serveur. Cette fonction est blocante ; elle ne rend la main que lorsque la connexion est effectivement établie ou s'il y a une erreur de connexion.

```
    // try to send 'resquests' on port "2056"
    commSock, errCode  := net.Dial("tcp", "127.0.0.1:2056") //IP+port as for server
    //if errCode != nil { //there is an error
    ...
```

Côtés Serveur et Client : fonctions de transfert de données : Read, Write

```
// writing on the communication socket
commSock.Write([]byte("aMessageToBeSent")) // send to the server/client

// reading from the communication socket; bufc will contain the result
bufLen, errCode := commSock.Read(bufc) // try a read (answer) from the Sock
//if errCode != nil { // there is an error
  ...
```

Côtés Serveur et Client : fonction de fermeture de socket Une fonction `Close` du package net, permet de fermer un socket.

```
     commSock.Close() // commSock closed; pay attention to not still reading it!
```

3 Fonctions auxilliaires, Go

```
   // how to read text input from stdin, using the bufio package
    reader := bufio.NewReader(os.Stdin) // create a reader
    fmt.Print("Hit the text: ") // write a prompt mesage on the screen
    text, textLen := reader.ReadString('\n') // read until '\n'
    // Now, the text+"\n" can be sent to a comm socket
```

4 Introduction aux **Goroutines**

Une goroutine est un 'thread' léger, une fonction d'exécution indépendante, avec sa propre pile d'appels, gérée par go au moment de l'exécution. Lorsqu'une fonction est lancée comme goroutine, elle s'exécute simultanément avec le reste des instructions.

Les goroutines s'exécutent dans le même espace d'adresses, donc elles doivent se synchroniser pour accéder à la mémoire.

...don't communicate by sharing memory; share memory by communicating (Rob Pike, co-créateur de Go).

Voici un exemple de lancement de fonctions comme des goroutines.

```
func myFunction1(s string) {   ... }
func myFunction2(s string) {   ... }
func anotherFunction() {   ... }
func main() {
        go myFunction1("hello")  // launch myFunction1 and continue the main
        go myFunction2("bonjour") // launch myFunction2 and continue the main
        anotherFunction("coucou") // call anotherFunction
}
```

Exercice 1 - prise en main

> Dans toute la suite du TP, et lors de vos futurs projets de programmation répartie, vous devez toujours tester les programmes en local (sur une machine : 2/n terminaux) ; quand tout marche comme il faut, vous pouver passer à 2/n machines !!

1. Récuperez et expérimentez le programme serveur fourni sur la page du cours. Enregistrez l'exemple donné dans un fichier, nommé `serveur1-1.go`

   Dans un terminal, compilez et exécutez le programme. Dans un deuxième terminal, exécutez en ligne de commande un client à l'aide de la commande `telnet` ou `ssh` (voir le `readme` associé à l'exemple).
2. Modifiez le programme (serveur1-1.go), en y mettant l'adresse IP de votre machine à la place de `localhost`. Recompiler. Tester.
3. Entre binômes, testez le programme à partir d'une autre machine distante, toujours avec `telnet` mais en donnant cette fois-ci l'adresse IP du serveur (l'autre machine).

Maintenant nous avons un *serveur* qui marche ; nous allons écrire et tester un *client*.

Exercice 2 - construction d'un client (en go)

Ecrivez en go un programme client, qui interagit avec le serveur précédent. Réfléchissez bien au protocole d'échange entre client et serveur ; utilisez par exemple un diagramme de séquence, pour clarifier qui envoie/reçoit quoi.

Exercice 3 - construction d'un serveur de ressources

Admettons maintenant que le serveur dispose d'une ressource qui est structurée comme une hashmap $\{(clé_i, valeur_i)\}$. Un client se connecte au serveur, puis envoie successivement des clés, pour récupérer des valeurs correspondantes. Ecrivez une (autre) version du serveur qui permet de répondre aux clients lui envoyant des clés pour obtenir les valeurs.
Imaginez un protocole pour l'arrêt de l'interaction entre le client et le serveur.

☞ Testez d'abord en local (2 terminaux), puis entre machines distantes.

Structurer bien vos programmes à l'aide de fonctions, de façon à en garantir la réutilisation. Par exemple, pouvez-vous servir des pages web demandées par des clients ? des fichiers demandés par des clients ?
Normalement, il suffira de remplacer les fonctions d'exploitation de la hashmap $\{(clé_i, valeur_i)\}$ par celles traitant les nouvelles ressources, sans modifier la structure générale de vos programmes.

Exercice 4 - un serveur et n clients

Généralisez l'exercice précédent (avec la hashmap $\{(clé_i, valeur_i)\}$) à une application qui est composée d'un serveur et de n clients. ☞ **Utilisez les goroutines**.

Expérimentez en exécutant les clients en local, puis à partir de machines distantes.

Exercice 5 - un serveur et n clients (`peer-to-peer`)

Généralisez l'exercice précédent à une application qui est composée d'un serveur et de n clients. Cette fois-ci un serveur ne disposant pas de la donnée demandée, peut s'adresser à un autre serveur pour la récupérer avant de la renvoyer. Posez une limite au nombre de serveurs et il n"y a pas de cycle dans les demandes.

Expérimentez en exécutant les pairs (`peer`) à partir de machines distantes.

Mise en œuvre et compte-rendu

Vous devez construire une série de programmes qui marchent (bien) tout le temps, sans installation/configuration particulière. Vous devez prévoir les configurations nécessaires (machines, adresses, paramètres, ...)
Vous fournirez une archive qui, une fois décompressée, permette d'exécuter directement ou à travers un menu (texte simple), chacune de vos applications.

```
*** Travaux réalisés par Mah Ray Chasles, DUT Info 2021/2022 ***
 Choisissez l'application que vous voulez tester, et laissez-vous guider
 1.  Une application 1client/1serveur en local (des indications ...)
 2.  Une application 1client/1serveur à distance (des indications ...)
 3.  Une application 1client/1serveur en local avec service de donnee <k,v>
 4.  ...
 9. Quitter
```

Vous devez préparer un compte-rendu de TP (1-2 pages) sans fautes, contenant la liste des exercices traités ; pour ceux qui marchent, comment ça marche et pour ceux qui ne marchent pas, pourquoi ça ne marche pas. Vous pouvez illustrer avec des copies d'écrans ou de programmes.

Le compte-rendu doit être fait simplement avec le formattage `markdown` (donc du simple ascii), afin de générer un document en pdf. N'oubliez pas, en en-tête de votre document, le *qui-quoi-quand-où*.

Rudiments pour la hashmap, et la lecture fichier

Créer une hashmap en `go`

```
var tamap map[string]string // clé et valeurs sont des 'string'

tamap = make(map[string]string)
tamap["fa"]  = "sill"
tamap["mi"]  = "lion"
tamap["li"] = "nez"
```

Lire à partir d'un fichier bibliothèque `ioutil`

```
ioutil.ReadFile("nomfichier")
```

Divers

Débarassez une chaîne des caractères spéciaux avant et après.

```
func main() {
   fmt.Print(strings.Trim("\n\n!!Hello, Bonjour, Yep !!\n\n", "\n"))
}
```

affichera `!!Hello, Bonjour, Yep !!`

Attention à la structure de contôle if_then_else,
le mot-clé `else` placé sur la même ligne : `if condition {partieAlors} else {partieSinon}`

Quelques références bibliographiques et "webographiques"

`https://golang.org/pkg/net/` (Sockets en go)
`https://tour.golang.org/concurrency/1` (Introduction à la concurrente)
`http://pagesperso.ls2n.fr/~attiogbe-c/mespages/programmation-applis-reparties.html`
`https://gobyexample.com/mutexes`