Les bases du langage Go
Partie 6 : communication réseau

Programmation système

BUT informatique, deuxième année

Comme pour le cours d'initiation au développement, le langage Go sera utilisé pour le cours de programmation système. Ce TP a pour but de compléter les bases du langage Go étudiées l'an dernier. On part du principe que les bases du langage Go (5 parties) vues en initiation au développement sont maîtrisées.

Dans tout ce TP un texte avec cet encadrement est un texte contenant une information importante, il faut donc le lire avec attention et veiller à en tenir compte.

Un texte avec cet encadrement est une remarque.

Un texte avec cet encadrement est un travail que vous avez à faire.

Si vous souhaitez avoir un autre point de vue sur le contenu de ce TP, vous pouvez consulter, par exemple, le site *Go by Example*[1].

1. `https://gobyexample.com/`

1 Sockets Unix

Sur un ordinateur Linux on peut communiquer entre processus en utilisant un mécanisme appelé *sockets*. L'idée générale est la suivante pour la communication entre deux processus : un fichier particulier est créé et, quand un des processus écrit dans ce fichier, l'autre a la possibilité de lire ce qui vient d'être écrit.

En Go, le paquet net permet de créer des sockets et d'interagir avec. Pour mettre en place la communication on utilise une approche client/serveur, qu'on peut décomposer en trois étapes :

1. Le serveur se met en écoute (processus 1)
2. Le client demande à se connecter (processus 2)
3. Le serveur accepte la connexion (processus 1)

À l'issu de ces trois étapes, une voie de communication est ouverte entre les deux processus, permettant à chacun d'écrire et de lire des messages.

> Une fois la communication établie, les rôles de client et de serveur n'ont plus lieu d'être : cette communication est symétrique.

1.1 Listen : le serveur se met en écoute

La mise en écoute du serveur peut se faire par la fonction net.Listen (du paquet net). Le programme 1 donne un exemple de son utilisation : il crée un serveur se mettant en écoute avec le protocole unix à l'adresse test.sock. On obtient un objet (au sens strict du terme, il n'y a pas d'objets en Go, mais cela y ressemble beaucoup) de type net.Listener qui permettra ensuite d'accepter une connexion d'un client. Après ça, on utilise defer pour garantir la fermeture de listener a la fin du programme. Enfin, un time.Sleep effectue une attente de 10 secondes qui sert à simuler les traitements à faire par le serveur.

```go
package main

import (
    "log"
    "net"
    "time"
)

func main() {

    listener, err := net.Listen("unix", "test.sock")
    if err != nil {
        log.Println("listen error:", err)
        return
    }
    defer listener.Close()

    time.Sleep(10 * time.Second)

}
```

Programme 1 – half-server.go

> Récupérez le fichier half-server.go sur MADOC, compilez-le et exécutez-le. Pendant que le serveur tourne, regardez le contenu du dossier où vous l'avez lancé. Une fois que le serveur a terminé de tourner, regardez à nouveau le contenu de ce dossier. Qu'observez-vous ?

Faites tourner deux serveurs en même temps. Que remarquez-vous ? Qu'en déduisez-vous sur les sockets ?

1.2 Accept : le serveur accepte la connexion

Une fois qu'un client aura demandé à se connecter à notre serveur, il faudra que ce dernier accepte la connexion pour pouvoir mettre en place la communication. Le programme 2 est une évolution du programme 1 qui ajoute l'acceptation d'une demande de connexion avec listener.Accept(). Ceci crée un objet de type net.Conn qui permettra d'interagir avec le client. Il ne faut pas oublier de fermer la connexion en fin de programme.

```go
package main

import (
    "log"
    "net"
    "time"
)

func main() {
    listener, err := net.Listen("unix", "test.sock")
    if err != nil {
        log.Println("listen error:", err)
        return
    }
    defer listener.Close()

    conn, err := listener.Accept()
    if err != nil {
        log.Println("accept error:", err)
        return
    }
    defer conn.Close()
    log.Println("Le client s'est connecté")

    time.Sleep(10 * time.Second)

}
```

Programme 2 – server.go

Récupérez le fichier server.go sur MADOC, compilez-le et exécutez-le. Attendez une quinzaine de secondes. Que remarquez-vous ? Qu'en déduisez-vous sur la méthode Accept ? Arrêtez le serveur (avec Ctrl+c par exemple).

Relancez un serveur. Que remarquez-vous ? Supprimez à la main le fichier test.sock. Lancez encore un serveur. Qu'en déduisez-vous ?

1.3 Dial : le client demande à se connecter

Un autre processus (qu'on appelle en général client) peut demander à se connecter au premier processus (le serveur). Pour cela, on utilise la fonction net.Dial en précisant les mêmes informations que dans la fonction net.Listen du serveur. Le programme 3 donne un exemple de cela.

```go
package main

import (
    "log"
    "net"
)

func main() {

    conn, err := net.Dial("unix", "test.sock")
    if err != nil {
        log.Println("Dial error:", err)
        return
    }
    defer conn.Close()

    log.Println("Je suis connecté")

}
```

Programme 3 – client.go

Récupérez le fichier client.go sur MADOC. Compilez-le puis essayez de l'exécuter quand le serveur tourne et quand il ne tourne pas. Qu'observez-vous ?

1.4 Communiquer

Une fois la communication établie, il est possible de communiquer entre les deux processus au moyen de leurs connexions (net.Conn) respectives. Pour cela on dispose de méthodes Read (pour lire) et Write (pour écrire).

Modifiez server.go et client.go de façon à pouvoir échanger un message entre eux. Il faudra donc que l'un lise et l'autre écrive. Placez judicieusement des affichages dans votre code pour constater ce qui se passe. Pour en savoir plus sur Read et Write n'hésitez pas à aller voir la documentation de paquet net[2]. On remarque notamment que ces fonctions retournent des erreurs, qu'il faudra donc tester systématiquement.

On peut «transformer» une variable msg de type []byte en variable de type string en écrivant simplement string(msg).

L'utilisation de ces fonctions de lecture et d'écriture n'est pas des plus simples car elles ne manipulent que des flux d'octets : il faut traiter soit même le découpage de ces flux en messages en fonction du protocole de communication que l'on souhaite mettre en place.

Pour se simplifier la vie on peut interagir avec notre connexion en utilisant des méthodes de plus haut niveau qui se basent sur Read et Write mais sont plus intuitives à utiliser. On peut par exemple utiliser le paquet bufio, exactement comme nous l'avions fait dans la partie 3 de cet apprentissage du Go pour écrire et lire dans des fichiers.

Modifiez à nouveau votre serveur et votre client pour utiliser les méthodes de bufio pour échanger le message entre vos deux processus.

2. `https://golang.google.cn/pkg/net/`

2 À travers le réseau

Pour communiquer entre machines situées sur un même réseau on peut procéder de la même manière que pour communiquer entre processus sur une machine unique. Il suffit de changer le protocole de communication (actuellement *unix*) et l'adresse à laquelle on contacte le serveur (actuellement test.sock).

Les programmes 4 et 5 sont des adaptation des programmes précédents pour établir une connexion TCP à l'adresse localhost (donc toujours sur une seule machine) et sur le port 8080.

```go
package main

import (
    "log"
    "net"
    "time"
)

func main() {
    listener, err := net.Listen("tcp", ":8080")
    if err != nil {
        log.Println("listen error:", err)
        return
    }
    defer listener.Close()

    conn, err := listener.Accept()
    if err != nil {
        log.Println("accept error:", err)
        return
    }
    defer conn.Close()
    log.Println("Le client s'est connecté")

    time.Sleep(10 * time.Second)

}
```

Programme 4 – server-tcp.go

Récupérez sur MADOC les fichiers server-tcp.go et client-tcp.go, compilez-les, exécutez-les. Constatez que tout fonctionne bien, éventuellement en les modifiant pour échanger quelques messages.

Avec votre voisin ou votre voisine, mettez en place une communication entre vos machines : le serveur sera sur une machine et le client sur l'autre. Testez. Échangez quelques messages.

```go
package main

import (
    "log"
    "net"
)

func main() {

    conn, err := net.Dial("tcp", "localhost:8080")
    if err != nil {
        log.Println("Dial error:", err)
        return
    }
    defer conn.Close()

    log.Println("Je suis connecté")

}
```

Programme 5 – client-tcp.go