Master 2 **Protocoles des services Internet**

Projet

Implémentation d'un protocole

Service de micro-blogging distribué

Léonard **Namolaru** Hugo **L.**

1 Fonctions implémentées

- **Sujet minimal**
    - Notre pair est capable de s'enregistrer auprès du serveur.
    - Notre pair est capable de télécharger les messages publiés par un autre pair et de les afficher (nous vérifions le hash d'un nœud avant de s'en servir).
    - Notre pair met à disposition des autres pairs une liste de messages.
- **Si nous envoyons un datagramme de type requête et que nous n'avons pas reçu de réponse**, nous renvoyons le datagramme plusieurs fois, le timeout entre chaque renvoi augmentant de façon exponentielle.
- **Interface utilisateur via la ligne de commande** : l'utilisateur peut décider avec quel pair il veut communiquer et quelle action il veut effectuer avec ce pair. En même temps, à travers un fil d'exécution (*thread*) séparé, nous attendons l'arrivée de nouveaux messages. Tout ce qui se passe s'affiche clairement à l'écran.
- **Nous sauvegardons en mémoire les sessions que nous avons ouvertes avec d'autres pairs** : cela permet d'éviter de refaire des *handshake* (**Hello**, **HelloReply**) à chaque fois qu'on contacte un pair.
- **Nous sauvegardons en mémoire les sessions que d'autres pairs ont ouvertes avec nous.** Si un pair nous demande des informations avant l'échange **Hello**, **HelloReply** avec nous, ou si la durée de la session est dépassée, il recevra un datagramme **Error** (type 254) avec un message d'erreur clair.

- **Nous sauvegardons en mémoire les arbres de Merkle que nous avons obtenus à partir d'autres pairs, avec une mise à jour incrémentale** (l'idée est de déterminer rapidement ce qui a changé et ne télécharger que les parties de l'arbre qui ont changé).
- **Nous implémentons l'authentification entre le client et le serveur ainsi qu'entre les pairs.** Nous partageons une clé publique, nous stockons une clé privée, et nous vérifions l'identité du pair ou du serveur à travers les signatures. De même les pairs et le serveur peuvent vérifier notre identité car nous signons nos messages en début de session.

2 Implémentation des arbres de Merkle

merkleTree.go

Nous trouvons que l'implémentation que nous avons faite pour les arbres de Merkle est intéressante car avec des modifications mineures elle peut également être utilisée pour d'autres projets, puisque nous avons essayé de créer une implémentation aussi générique que possible. C'est pourquoi nous avons décidé de vous présenter ici ses grands principes :

- **Construire un arbre de Merkle à partir des feuilles (dans notre cas, les messages)**

```
func CreateTree(messages [][]byte, maxArity int) *MerkleTree
func (merkleTree *MerkleTree) createLeafNodes(messages [][]byte) []*MerkleNode
func (merkleTree *MerkleTree) createNodes(leafNodes []*MerkleNode) *MerkleNode
```

À partir d'une liste prête de messages, il est possible de produire un arbre de Merkle qui permettra aux autres paires d'accéder à ces messages. Dans un tel cas, la construction de l'arbre de Merkle s'effectue de haut en bas, c'est-à-dire : des feuilles à la racine.

Pour cela, nous avons une fonction récursive : dans un premier temps, pour chaque groupe de 32 messages (ou moins, à condition que le groupe ait au moins 2 messages), nous créons un nœud parent. Nous transférons ces nouveaux nœuds à un autre appel de la fonction. S'il reste un seul message au-delà des groupes de 32 messages, il est également transféré à l'étape suivante. Maintenant, nous répétons le processus : créer un nouveau nœud pour chaque groupe contenant entre 2 et 32 nœuds et les

transférer à un autre appel de la fonction. La aussi, s'il reste un seul élément au-delà des groupes de 32 nœuds, cet élément est également passer à l'appel suivant. Et ainsi de suite, jusqu'à ce qu'un appel à la fonction reçoive un groupe qui ne dépasse pas 32 éléments, dans ce cas on crée un élément parent pour ces éléments, cet élément devient la racine de l'arbre. Le code que nous avons écrit peut correspondre à n'importe quelle limite du nombre d'enfants par nœud, tant que ce nombre est supérieur à 1.

- **Construire un arbre de Merkle de haut en bas**

```
func CreateEmptyTree(maxArity int) *MerkleTree
func (merkleTree *MerkleTree) AddNode(hash []byte, nodeData []byte) bool
```

Dans le cas où l'on reçoit les nœuds de l'arbre d'un autre pair, on construit l'arbre au fur et à mesure que les nœuds arrivent par le réseau.

Dans ce cas, puisque chaque nœud interne comprend une liste avec le hash de chacun de ses enfants, de cette façon pour chaque nœud, nous savons sous quel nœud existant nous devons ajouter chaque nouveau nœud qui arrive. Le seul nœud qui n'a pas de parent est bien sûr la racine.

La même fonction nous aide également à mettre à jour l'arborescence sans retélécharger tous les nœuds. Lorsque nous obtenons la racine, nous parcourons la liste qui inclut les hachages de ses enfants et vérifions si la racine existante a actuellement des enfants avec un hash différent. Les enfants de la racine dont le hash n'apparaît pas dans la nouvelle racine sont supprimés, et il nous reste maintenant que les parties de l'arbre qui n'ont pas changé. Avant d'envoyer chaque datagramme de type **GetDatum**, nous allons d'abord vérifier si le hash est déjà dans l'arbre. Si tel est le cas, nous pouvons renoncer à envoyer ce datagramme, puisque les données existent déjà chez nous.

- **Parcours en profondeur de l'arbre**

De nombreuses opérations que nous venons d'évoquer nécessitent de parcourir en profondeur l'arbre : rechercher si un certain hash se trouve dans l'arbre et rechercher si le père d'un certain nœud se trouve dans l'arbre en sont de bons exemples. Au-delà de cela, nous voulons parfois imprimer l'arbre entier ou des parties de celui-ci. Nous effectuons toutes ces

actions en utilisant la méthode suivante :

```
func (merkleTree *MerkleTree) DepthFirstSearch(nodesHeightCountInitialization
int, function func(int, *MerkleNode, []byte) bool, hashSearch []byte, arg
...*MerkleNode) *MerkleNode
```

```
function func(int, *MerkleNode, []byte) bool
```

`function` est une fonction qui sera appelée sur chacun des nœuds de l'arbre. Il s'agit d'une fonction qui renvoie *true* si nous sommes intéressés qu'après l'avoir appelée, nous arrêtons le parcours et que la fonction de parcours **`DepthFirstSearch`** renvoie un pointeur vers le nœud sur lequel nous venons d'appeler la fonction. **`*MerkleNode`** est donc un pointeur vers le nœud sur lequel on lance la fonction, et **`[]byte`** est le hash que nous recherchons dans l'arborescence, et est en fait une copie du paramètre **`hashSearch []byte`**. Si nous ne recherchons pas un hash particulier, mais par exemple que nous voulons simplement imprimer l'arbre, nous utiliserons la valeur **`nil`** pour le paramètre **`hashSearch`**.

Enfin, l'**`int`** de **`function func(int, *MerkleNode, []byte) bool`** est en fait la hauteur du nœud sur lequel on lance la fonction par rapport au début du parcours. La valeur au début du parcours est déterminée en fonction du paramètre **`nodesHeightCountInitialization int`** et elle augmente de 1 à chaque passage à un "étage" inférieur de l'arbre. Par conséquent, pour la plupart des cas, nous utiliserons le nombre 0 pour le paramètre **`nodesHeightCountInitialization`**. Ceci est particulièrement utile si nous voulons imprimer l'arbre entier alors qu'avant d'imprimer chaque nœud, nous imprimons X fois le caractère spécial **`\t`** en fonction de la hauteur à laquelle le nœud est localisé afin d'obtenir un *output* selon la hiérarchie de l'arbre.

Dernier point : le paramètre **`arg ...*MerkleNode`** de la fonction parcours **`DepthFirstSearch`** est facultatif et permet de donner un pointeur sur le nœud à partir duquel la recherche va démarrer. En l'absence de ce paramètre, la recherche commence à partir de la racine.

3 Communication avec le serveur et avec les autres pairs

networking.go

Quand on parle de communication il faut penser à 2 aspects : d'une part : envoyer des datagrammes au serveur ou à un autre pair, et d'autre part : recevoir des datagrammes du serveur ou d'un autre pair. Dans le deuxième cas, il est possible que nous ayons reçu une réponse suite à un datagramme que nous avons envoyé, mais il est également possible que ce soit un datagramme qui nous oblige à envoyer une réponse.

Tout d'abord, nous définissons une socket non connectée dans la fonction **`main()`** du fichier **client.go** :

```
conn, errorMessage := net.ListenPacket("udp", UDP_LISTENING_ADDRESS)
```

Et immédiatement après, nous exécutons un nouveau *thread* qui sera actif tant que notre client est actif. Le rôle de ce *thread* est de lire les datagrammes qui nous seront envoyés, il s'agit de la fonction **`UdpRead()`**, une fonction qui, comme les autres fonctions que nous traiterons dans cette partie du rapport, est définie dans le fichier **networking.go**.

- **Si nous envoyons un datagramme de type requête et que nous n'avons pas reçu de réponse, nous renvoyons le datagramme plusieurs fois, le timeout entre chaque renvoi augmentant de façon exponentielle.**

  Pour cela, nous avons défini une nouvelle structure de données qui représente une adresse dont nous attendons une réponse :

```
type WaitingResponse struct {

    // The UDP address from which we are waiting for a reply

    FullAddress *net.UDPAddr

    // A list of the type numbers of the datagrams we are waiting to receive from this address. For example: HELLO_REPLY_TYPE, DATUM_TYPE, NO_DATUM_TYPE

    DatagramTypes []int

    // The id that should be in the datagram of the answer we will receive (the same as the id in the datagram of our request)

    Id []byte

}
```

  Et nous avons défini une liste globale qui comprendra toutes les **`WaitingResponse`** dont nous attendons toujours des réponses :

```
var waitingResponses []WaitingResponse
```

Ainsi, au début de la fonction **`UdpWrite()`**, pour chaque type de datagramme que nous pouvons envoyer, nous vérifions s'il s'agit d'un datagramme auquel nous devons recevoir une réponse. Et si oui - quelle type de réponse ?

```
var responseOptions []int
switch datagramType {
case HELLO_TYPE:
    responseOptions = append(responseOptions, HELLO_REPLY_TYPE)
case GET_DATUM_TYPE:
    responseOptions = append(responseOptions, NO_DATUM_TYPE, DATUM_TYPE)
  ..
}
```

Si la liste **`responseOptions`** est pas vide, cela signifie que nous attendons une réponse. Nous insérons donc une nouvelle instance de la structure de données **`WaitingResponse`** dans la liste **`waitResponses`**. Si l'adresse à laquelle nous voulons envoyer un datagramme est déjà dans cette liste (c'est-à-dire que nous attendons déjà des messages de la même adresse), nous ajouterons la liste **`responseOptions`** au champ **`DatagramTypes`** de l'instance de la structure de données **`WaitingResponse`** qui existe déjà pour cette adresse.

Nous envoyons le datagramme une première fois puis nous envoyons le "*thread* principal" en veille pendant 2 secondes. Cette durée augmente de façon exponentielle pour chaque envoi supplémentaire (4, 8 et 16 secondes). Nous faisons un maximum de 4 tentatives car pendant nos tests, dans la plupart des cas d'autres pairs que nous voulions contacter étaient (probablement) inactifs et donc faire trop de tentatives aurait empêché le client d'effectuer d'autres actions.

Avant chaque tentative d'envoi de message, nous vérifions si l'adresse à laquelle nous avons envoyé un datagramme est toujours dans la liste **`WaitingResponse`**, si c'est le cas, nous en envoyons un autre.

En parallèle, le *thread* chargé de lire les datagrammes qui nous parviennent vérifie à chaque fois qu'un datagramme arrive s'il s'agit du datagramme que nous attendions. Si tel est le cas, l'adresse est supprimée de la liste **`waitingResponses`**. Le "*thread* principal" verra cela lorsqu'il se réveillera et saura qu'il peut arrêter d'essayer d'envoyer des messages vu qu'une réponse a été reçue.

L'utilisation d'une liste de données pouvant être lue et modifiée par 2 *threads* distincts peut créer des problèmes et nous utilisons donc un **`mutex`** afin que la lecture et l'écriture de cette liste se fassent de manière atomique.

- **Réponses aux requêtes d'autres pairs (ou du serveur)**

  Après réception d'un nouveau datagramme, la fonction **`ReadUdp()`** vérifie s'il s'agit d'un datagramme qui nécessite une réponse :

```
 switch buf[TYPE_BYTE] {

        case byte(HELLO_TYPE):
            ..
            UdpWrite(conn, string(buf[ID_FIRST_BYTE:ID_FIRST_BYTE+ID_LENGTH]),
HELLO_REPLY_TYPE, udpAddress, nil, privateKey))

        case byte(ROOT_REQUEST_TYPE):

            UdpWrite(conn, string(buf[ID_FIRST_BYTE:ID_FIRST_BYTE+ID_LENGTH]),
ROOT_TYPE, udpAddress, nil, privateKey)

        ..
```

4 Sécurité

Authentification

- Le client crée une clé privée qu'il stocke dans un fichier chiffré (AES).

```
func Encrypt(key []byte, plainText []byte) []byte {

    block, err := aes.NewCipher(key)

    ...

    cipherText := make([]byte, aes.BlockSize+len(plainText))

    iv := cipherText[:aes.BlockSize]

    ...

    stream := cipher.NewCFBEncrypter(block, iv)

    stream.XORKeyStream(cipherText[aes.BlockSize:], plainText)

    return cipherText

}
```

- Le client crée une clé publique à partir de ce fichier.

```
func CreateOrFindPrivateKey(fileInfo fs.FileInfo, err error) *ecdsa.PrivateKey {
    keyForFile := []byte("asuperstrong32bitpasswordgohere!") //32 bit key for AES-256

    if err != nil || fileInfo.Size() == 0 {
        privateKey, err := ecdsa.GenerateKey(elliptic.P256(), rand.Reader)
        privateKeyDr, err := x509.MarshalECPrivateKey(privateKey)

        ...
        privPEM := pem.EncodeToMemory(
            &pem.Block{
                Type:  "EC PRIVATE KEY",
                Bytes: privateKeyDr,
            },
        )

        cipherPrivPEM := Encrypt(keyForFile,privPEM)
        err = ioutil.WriteFile(NAME_FILE_PRIVATE_KEY, cipherPrivPEM, 0644)
        ...
    }

    cipherData, err := ioutil.ReadFile(NAME_FILE_PRIVATE_KEY)
    ...
    plainData := Decrypt(keyForFile, cipherData)
    lines := strings.Split(string(plainData), "\n")
    lines = lines[1 : len(lines)-2]
    privateKeyString := strings.Join(lines, "")
    ...
    privateKey, err := ecdsa.GenerateKey(elliptic.P256(),
bytes.NewReader([]byte(privateKeyString)))
    ...
    return privateKey
}
```

- Le client partage sa clé publique.
- Le client signe ses premiers messages au début d'une session

```
(func CreateSignature(datagram []byte, datagramLength int, privateKey *ecdsa.PrivateKey) []byte ).
```

- Le client récupère la clé publique du serveur.
- Le client récupère les clés publiques des pairs s'ils en ont.

- Le client vérifie les signatures du serveur et des pairs en début de session

```
(func VerifySignature(buf []byte, publicKey *ecdsa.PublicKey) bool ).
```

Chiffrement

- Le client crée une clé privée et une clé publique associée à une session lorsque deux pairs ont le flag **8** dans leur **Hello**.

```
func CreatePrivateKeyForEncryption() *ecdsa.PrivateKey

func GeneratePublicEncodedKeyForEncryption(privateKey *ecdsa.PrivateKey) string

type SessionWeOpened struct {
    FullAddress             *net.UDPAddr
    LastDatagramTime        time.Time
    Merkle                  *MerkleTree
    Buffer                  []byte
    sharedKey               []byte
    privateKeyForSession    *ecdsa.PrivateKey
    myPublicKeyForSession   *ecdsa.PublicKey
}
```

- Le client échange sa clé publique avec le type **SEND_KEY_HELLO_TYPE** s'il est le premier à envoyer sa clé publique, et **SEND_KEY_HELLO_REPLY_TYPE** s'il est le deuxième.
- Ensuite chaque pair calcule la clé partagée de chiffrement/déchiffrement qu'il stocke dans une session.

```
func GenerateSharedKey(publicKeyForEncryptionFromPeer ecdsa.PublicKey,
myPrivateKeyForEncryption *ecdsa.PrivateKey) []byte
```

- Le client chiffre et déchiffre les messages en AES avec la fonction **Encrypt()** et **Decrypt()** si la session contient une clé partagée.

```
if sessionsWeOpened[i].sharedKey != nil {
    datagram = Encrypt(sessionsWeOpened[i].sharedKey, datagram)
}
if sessionsWeOpened[i].sharedKey != nil {
    buf = Decrypt(sessionsWeOpened[i].sharedKey, buf)
}
```