Rapport

DERVISHI Sevi, HEOUAIRI Adrian

6 janvier 2024

Table des matières

1 Introduction

Il s'agit d'un projet réalisé en Go 1.21. Nous avons réussi à transférer un fichier de 698 `Mio` entre deux instances de notre pair à une vitesse de 120 `kio/s`. Le téléchargement de `jch.irif.fr/videos/r.mp4` prend 32 s. Le programme présente une interface en ligne de commande avec TAB *completion* pour les noms de commande et leurs arguments (voir la liste des commandes ci-dessous). On peut également lancer une commande avec par exemple `go run . wget jch.irif.fr/images/horse.jpg` (le programme se termine après l'avoir exécutée).

```
> help
PATH is PEER_NAME[PATH2] with PATH2 = /videos for example
    hello PEER: sends at least two Hellos to PEER
    exit: exits the program
    help: shows help message
    lspeers: shows the connected peers if --addr specified shows also addresses
    findrem PEER: shows the files shared by PEER
    curl PATH: downloads and shows the file at PATH
    wget PATH: downloads recursively the directory or file at PATH
```

Le projet est découpé en fichiers `.go` : `constants.go download.go main.go merkle_tree.go messages.go rest.go udp.go ui.go utils.go`.

2 Constantes

L'intégralité du code utilise des constantes, par exemple le nombre de réémissions, présentes dans `constants.go`.

3 Format des messages

Tous les messages sont représentés par la structure `udpMsg` :

```
type udpMsg struct {
    Id     uint32
    Type   uint8
    Length uint16
    Body   []byte
}
```

De plus, on *parse* le `Body` de certains messages avec d'autres structures :
— *Hello, HelloReply* → `struct hello`
— *Datum* → `datumChunk, datumTree (big file), datumDirectory`
Nous avons des fonctions de conversion `udpMsgToByteSlice` et `byteSliceToUdpMsg` pour convertir en `[]byte` afin de recevoir ou d'envoyer des messages par UDP.

4 Lister les fichiers d'un pair ou les télécharger

`getPeerPathHashMap...(peerName string, hash []byte, path string, currentMap map[string][]byte)` Permet de construire une map associant un chemin (`jch.irif.fr/images/horse.jpg`) à son hash de datum. Il faut passer le hash de la racine et une map vide au premier appel. On ajoute la paire (hash, path) à la map, on télécharge le datum décrit par hash, si c'est un dossier, on fait un appel récursif sur chaque fils avec un *path* égal à `path + "/" + nom de fichier du fils`.

`downloadRecursive(peerName string, hash []byte, path string)` utilise un principe similaire : Lorsque hash donne un *datum directory*, on crée le dossier, si c'est un *chunk* on écrit le fichier *single-chunk*, si c'est un *big file* on lance `writeBigFile`.

`writeBigFile(peerName string, datum datumTree, path string, depth int)` Il s'agit d'une fonction récursive qui écrit un *big file* sur disque : on itère sur les hashs enfants de *datum* : on télécharge l'enfant, si c'est un *chunk* on l'*append* au fichier sur disque, si c'est un *big file* on fait un appel récursif. Ceci effectue un parcours préfixe de l'arbre *big file* qui va écrire les *chunks* dans l'ordre.

On ne stocke jamais l'arbre d'un pair, il est retéléchargé à chaque commande.
Les fichiers téléchargés d'autres pairs sont stockés dans `PSI-download` à la racine du projet, par exemple `PSI-download/jch.irif.fr/images/horse.jpg`.

5 Réception et envoi de messages

Nous utilisons deux structures en variable globale :
— `peers map[string][]*net.UDPAddr` protégé par un `RWMutex` : associe un nom de pair à la liste de ses adresses supposées valides jusqu'à erreur.
— `msgQueue` liste doublement chainée (`list.List`) protégée par un `RWMutex` : stocke les réponses (type $\geqslant 128$) qui doivent être récupérées par le thread qui a envoyé une requête.
Nous bindons `:8449` (normalement `0.0.0.0:8449`) en IPv4 avec `ListenUDP` (le `UDPConn` est stocké dans la variable globale `connIPv4`) et on a un thread qui exécute `listenAndRespond` qui reçoit tous les messages (il est bloqué sur `ReadFromUDP`) et lance `handleMsg` sur chaque message. On ajoute manuellement notre adresse `127.0.0.1:8449` dans `peers` pour pouvoir communiquer avec nous-mêmes.

`handleMsg(addr, msg)` : Les messages qui sont des réponses (type $\geqslant 128$) sont simplement ajoutés à la `msgQueue`. Les messages qui sont des requêtes sont traités dans `handleMsg` (on y répond si cela est nécessaire avec `simpleSendMsgToAddr`). Si on reçoit un Hello d'un pair inconnu, on ajoute son nom et son adresse dans la map `peers` des pairs connus.

`simpleSendMsgToAddr` exécute simplement `connIPv4.WriteToUDP`.

`retrieveInMsgQueue(addr, msg)` parcourt la `msgQueue` en boucle à la recherche d'une réponse et retourne une erreur au bout de `MSG_QUEUE_MAX_WAIT` (3 s) si la réponse n'est pas trouvée.

`sendToAddrAndReceiveMsgWithReemissions` exécute une boucle 5 fois : envoyer la requête à une adresse, puis `retrieveInMsgQueue`.

`natTraversal()` exécute une boucle 10 fois : envoyer un *NatTraversalRequest* à `jch.irif.fr`, puis `sendToAddrAndReceiveMsgWithReemissions(addr, createHello())`.

Le thread `keepAliveMainPeer` maintient la connexion avec `jch.irif.fr` en envoyant *Hello* toutes les 30 s. L'API exposée pour envoyer et recevoir des messages :
`ConnectAndSendAndReceive(peerName string, toSend udpMsg)` : Si le pair est connu (son adresse est dans peers), on fait `sendToAddrAndReceiveMsgWithReemissions` (pas de `natTraversal`). Si aucune ne marche, on obtient les adresses du pair avec REST et jusqu'à ce qu'une adresse des adresses marche : `sendToAddrAndReceiveMsgWithReemissions` Hello, si ça ne marche pas `natTraversal`, puis si on a reçu HelloReply, on envoie enfin le message avec `sendToAddrAndReceiveMsgWithReemissions`, si cela fonctionne on ajoute l'adresse du pair à `peers`.

6 Partage de nos fichiers avec les pairs

Le téléchargement de l'arbre d'un pair et l'exportation de nos fichiers n'utilisent aucun code en commun. Nos fichiers à partager se situent dans `PSI-shared-files` à la racine du projet.

Nous représentons l'arbre de *Merkle* que nous partageons par une structure récursive `merkleTreeNode` :

```
type merkleTreeNode struct {
    // The parent node, useful for hash computation
    // Never nil except for the root node
    Parent *merkleTreeNode

    // Path of the file or directory this node represents
    Path string

    // Never nil (even for CHUNK)
    Children []*merkleTreeNode

    // Nil if not computed yet
    Hash []byte

    // CHUNK, TREE, DIRECTORY
    Type byte
```

```
    // -1 if not CHUNK
    ChunkIndex int
}
```

Nous construisons cet arbre une fois pour toutes au début du programme (il faut redémarrer le programme pour prendre en compte les modifications dans `PSI-shared-files`). La construction de notre arbre se produit en deux étapes : un parcours récursif de `PSI-shared-files` qui crée tous les `merkleTreeNode` correspondant aux fichiers et dossiers de `PSI-shared-files` et calcule le `merkleTreeNode.Hash` pour toutes les feuilles (dossier vide ou *chunk*, que les *chunks* soient dans *big file* ou non). Ensuite, on lance une fonction récursive qui calcule le hash de tous les nœuds internes.

6.1 Première étape (création de l'arbre)

```
func PathToMerkleTreeWithoutInternalHashes(path string, parent *merkleTreeNode) (*merkleTreeNode, error)
```

Le premier appel est effectué avec `path = PSI-shared-files` et `parent = nil`.
Fonctionnement : on crée `ret`, le `merkleTreeNode` à retourner.

- Si `path` est un dossier : `ret` prend le type `directory`. Si `path` est un dossier vide, on set son hash au hash du dossier vide. Si `path` est un dossier et contient des fichiers ou dossiers : on lance un appel récursif avec arguments (`path + "/" + nom de fichier, ret`) et on append le résultat à `ret.Children`.
  - Si `path` est un fichier de taille $\leqslant 1024$, `ret` prend le type *chunk*, `ret.ChunkIndex = 0`. On lit le *chunk*, on calcule son hash et on set `ret.Hash`. Les champs `Path` et `ChunkIndex` d'un `merkleTreeNode` de type *chunk* permettent de retrouver le contenu du fichier en lisant le fichier situé à `Path` au bon endroit selon `ChunkIndex`. De cette façon, on ne stocke pas les *chunks* en mémoire.
  - Si `path` est un fichier de taille > 1024, on set `type = big file` et on lance fillBigFile sur `ret` qui va créer tous les `merkleTreeNode` *big file* et *chunk* et les mettre en tant qu'enfant, petits-enfants, etc de `ret`.
- `fillBigFile` : cette fonction n'est pas récursive. On part d'une racine *big file* sans enfants.
  - Première étape qui crée uniquement les nœuds *big file* : La capacité courante est 32, le *big file* auquel on ajoute des enfants *big file* reste le même tant que sa liste de fils n'est pas pleine. On calcule le nombre de *chunks* dans le fichier. Tant que le nombre de *chunks* que l'arbre courant peut accueillir est strictement inférieure à la capacité courante, on ajoute un enfant *big file* et on augmente la capacité courante de 31 (32 nouvelles places moins la place prise par le nouveau *big file*). Lorsque la liste des enfants du *big file* auquel on est en train d'ajouter des enfants devient pleine, on cherche le prochain *big file* auquel on peut ajouter des enfants par *level order traversal* partant de la racine.
  - Deuxième étape qui ajoute les feuilles *chunk* La deuxième partie de `fillBigFile` ajoute les feuilles *chunk* à l'arbre *big file* par un appel à `root.addChunkLeaves(nbChunk, 0)`, `root` étant le *big file* racine, `nbChunk` étant le nombre de *chunks* dans le fichier et 0 étant l'indice du premier *chunk*.

`func (bigFile *merkleTreeNode) addChunkLeaves(nbChunkToCreate int, nextChunkIndex int)` est une méthode récursive. Pour tous les enfants de `bigFile`, on fait `nextChunkIndex = child.addChunkLeaves(nbChunkToCreate, nextChunkIndex)`. Ensuite, tant que `bigFile` a de la place dans ses enfants et qu'on ne dépasse pas `nbChunkToCreate`, on ajoute un fils `chunk` avec le hash calculé et `ChunkIndex = nextChunkIndex` et on incrémente `nextChunkIndex`. Enfin, on retourne `nextChunkIndex`.

6.2 Deuxième étape (calcul des hash des nœuds internes)

Une fois que tous les nœuds de l'arbre ont été créés, on calcule les hashs des nœuds internes avec : `func (node *merkleTreeNode) computeHashesRecursively()`
La méthode est lancée sur la racine (`merkleTreeNode avec Path = PSI-shared-files`).
Fonctionnement :
Si le hash existe déjà (`node` est une feuille `chunk` ou `directory`), `return`. Par la suite, node est donc de type `directory` ou `big file`. Pour chaque enfant de `node`, on lance un appel récursif sur l'enfant et on ajoute à la valeur à hasher le hash de l'enfant (et le nom de fichier paddé avec `\0` si c'est un `directory`). En fin de fonction, on set le hash de `node` au hash de `value`.

6.3 `map[string]*merkleTreeNode`

Pour répondre rapidement aux `GetDatum`, nous stockons une `map` qui associe chaque hash à son `merkleTreeNode`.

7 Cryptographie

Nous arrivons à signer les messages, mais la vérification de signature des messages entrants peut parfois échouer, c'est-à-dire qu'un message correctement signé est considéré comme étant mal signé et donc jeté.

8 Améliorations possibles

— Nous aurions dû utiliser des pointeurs vers des structures au lieu de valeurs (évite les copies lors du passage en argument).
— Il faudrait vérifier que `PSI-shared-files` ne contient pas de dossiers ayant plus de 16 entrées au début du programme.
— Le champ `Path` de `merkleTreeNode` devrait être un pointeur de `string` (si cela est possible en Go) afin de ne pas répéter en mémoire le même *path* pour tous les enfants *big file* et *chunk* d'un *big file*.
— Il faudrait gérer les pairs dont la racine n'est pas un *chunk* `directory` (*big file* ou *chunk*).
— Il faudrait vérifier que les noms de fichier dans un `datum directory` sont de l'UTF-8 valide.

— Il faudrait implémenter `IPv6` (avoir deux threads qui reçoivent des messages au lieu d'un, et utiliser le bon *socket* en fonction de l'adresse à laquelle on envoie un message).
— Il faudrait envoyer *ErrorReply* aux pairs qui nous envoient des messages invalides.
— Il faudrait remplacer `msgQueue` par une `map[*addrId]*udpMsg` dont la clé est une struct (adresse, port, ID du message) et la valeur est un pointeur vers le message. Ainsi, avant d'envoyer un message, on crée la clé dans la `map` avec une valeur `nil` pour indiquer qu'on attend le message. Lors de la réception d'un message, on vérifie si la clé est présente avec une valeur `nil`. Si la clé n'existe pas, on peut jeter le message car aucun thread ne va jamais le récupérer (actuellement, les réponses envoyées par un pair sans requête de notre part sont stockées dans la `msgQueue` pour toujours). Avec cette modification, `retrieveInMsgQueue` serait simplement en attente active tant que la valeur de la clé est `nil`.
— Il faudrait supporter les guillemets dans l'interface en ligne de commande pour gérer les chemins avec des espaces.
— Il faudrait supporter les noms de pair dont le nom contient "/".
— Le champ Length de `udpMsg` est redondant, il suffit d'une méthode qui fait `len(Body)`.
— Il faudrait utiliser les mêmes structures de données pour uploader et downloader des arbres.