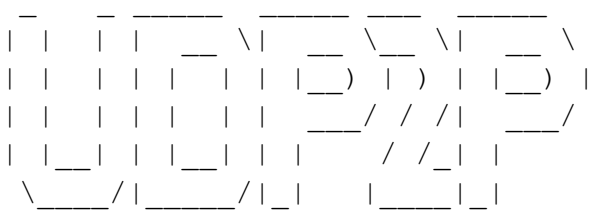
Document d’architecture technique

M2 MIC

Projet de Protocoles Internet



Rayane BAIT, Guilhem MIZRAHI

1. La représentation du file system

1.1 L'arbre de Merkle

Le file system est représenté par la structure `MktFsNode` dans la librairie `lib-file`. Sa définition est la suivante :

```
pub struct MktFsNode {
   pub path: PathBuf,
   pub ntype: MktFsNodeType,
   pub children: Option<Vec<MktFsNode>>,
   pub hash: [u8; 32],
}
```

C'est donc un arbre de Merkle. Chaque nœud contient ses enfants potentiels dans l'attribut `children`.

L'attribut `ntype` est du type enum `MktFsNodeType`. En rust les enums peuvent notamment prendre des arguments. La définition de l'enum est la suivante :

```
pub enum MktFsNodeType {
    DIRECTORY { path: PathBuf },
    CHUNK { file: File, offset: u64 },
    BIGFILE { path: PathBuf },
}
```

Les arguments dans la variante `CHUNK` permettent notamment d'ouvrir les fichiers à l'avance et de savoir à quel offset lire le contenu du fichier pour le `CHUNK` en question. Ces arguments sont calculés au moment de la création de l'arbre.

1.2 L'optimisation du nombre d'enfants

Lors de la création de l'arbre de Merkle plusieurs choix sont possibles pour l'organisation des enfants d'un `BIGFILE`. Une solution naïve serait de répartir les données de façon équitable entre les enfants. Cette solution peut s'avérer sous optimale d'un point de vue du réseau en nécessitant l'envoi de plus de paquets. Par exemple, avec une taille maximum de `CHUNK` autorisée à 1024 octets et 2 enfants par nœud, un fichier de $3 \times 1024 = 3072$ octets serait réparti en 4 enfants de 768 octets, alors que 3 enfants devraient suffire à contenir les données.
L'approche utilisée dans `UDP2P` détermine à l'avance le nombre de nœuds optimal ainsi que la profondeur de l'arbre nécessaire pour stocker ces nœuds. Ainsi la profondeur de l'arbre est

$$profondeur = floor(log_{nb\ max\ d'enfants\ par\ nœud}(nombre\ de\ chunks\ optimal))$$

cette valeur est décrémentée si le logarithme est entier. Chaque enfant est alors chargé de $taille\ des\ chunks \times (nb\ max\ d'enfants\ par\ nœud)^{profondeur}$ octets (le dernier étant chargé du reste).
Cette approche permet de générer un nombre minimal d'enfants et d'optimiser le nombre de paquets a envoyer.

1.3 La génération de l'arbre et le calcul des hashes

L'arbre est généré de façon récursive à partir d'un chemin sur le file system du peer exportant son arborescence. De la même façon, les hashes sont calculées de façon récursive selon le format de la spécification.

2. La communication avec le serveur REST

2.1 Le client HTTP

Le client HTTP est implémenté avec la librairie `reqwest`. Un unique client est utilisé pour faire toutes les requêtes lorsque l'application demande les informations des pairs pour profiter des fonctionnalités de reprise de session. Aussi le header User-Agent est mis à la valeur "Projet M2 protocoles Internet" pour identifier que le potentiel spam du serveur lors de la phase de développement venait bien d'un groupe d'élèves légitime.

3. La structure interne du protocole

3.1. Le système de queues.

Le pair le plus simple n'a pas besoin d'état pour renvoyer des données valide. Étant donné un paquet (valide) reçu, il est censé pouvoir renvoyer le paquet attendu avec seulement les données contenues dans le paquet. Le système de queues exploite cela en séparant l'implémentation en queues et tâches. Qui permettent de notifier les différentes tâches lorsqu'une queue n'est pas vide.
Il y a 4 queues principales:

- La queue de réception.
- La queue de traitement.
- La queue d'action.
- La queue d'envoi.

Ainsi que 6 tâches indépendantes (`tokio::tasks`) qui manipulent ces queues:

- La tâche de réception, qui réceptionne, sérialise et envoie les paquets (ainsi que l'adresse de socket d'où ils proviennent) dans la queue de réception.
- La tâche de manipulation des paquets, qui lit la queue de réception, détermine si le paquet lu est valide, analyse le corps et envoie une action de type process (structure de type `Action::Process`) correspondante dans la queue de traitement.
- La tâche de traitement, qui analyse l'action de type process et effectue des actions correspondantes (Par exemple, lorsque l'action est de type `ProcessHelloReply`, le pair créé, s'il n'existe pas, le pair et rempli son nom et ses extensions).

- La tâche d'envoi, qui lit la queue d'envoi (queue qui contient des paquets associés à des adresses de socket), détermine le type d'ip et envoie le paquet à l'adresse correspondante avec une socket adaptée (de type ipv6 ou ipv4).
- La tâche de renvoi, qui détermine les paquets à renvoyer et les traversées de NAT à effectuer.

3.2. L'implémentation des queues

Les queues sont des vecteurs à deux sens sur un type générique `T` :

```
struct Queue<T> {
	data: VecDeque<T>,
}
```

Elles sont mises dans un `Arc` (Atomic reference counter, pour passer les queues à différents threads) et un `Mutex` (Pour que chaque thread puisse les modifier sans *race condition*). La queue de traitement est mise dans un `RwLock`. Le `RwLock` permet d'avoir plusieurs lecteurs à la fois mais un seul thread qui écrit. Cela permet d'avoir des tâches auxiliaires qui lisent seulement la queue de traitement (Par exemple, lors du téléchargement, on peut recevoir des paquets de plusieurs personnes en même temps, on lit alors la queue de traitement jusqu'à voir le paquet voulu). Les queue sont accompagnée d'états de queue

```
struct QueueState {
	is_not_empty: (Mutex<bool>, Condvar),
}
```

Lorsqu'une queue est vide, les tâches la lisant se mettent à l'arrêt en appelant une méthode `QueueState::wait()`. Alors les tâches remplissant ces queues peuvent appeler `QueueState::set_non_empty_queue()` pour notifier les lecteurs de la queue que celle ci n'est plus vide.

3.3. L'implémentation des tâches

L'implémentation est basée sur la librairie asynchrone *tokio*, qui permet de faire des manipulations "en parallèle" sans pour autant monopoliser des ressources en utilisant des threads. L'exécution est alors concurrente et non parallèle. *Tokio* met à disposition des taches *`tokio::spawn()`* qui ne demandent que 64 octets et une allocation, elles sont donc bien moins gourmandes que des threads. Les tâches ont toujours la forme suivante: On lit une queue jusqu'à lire le paquet qui nous intéresse (`peek_until_datum_with_hash_from()`), on effectue une action (enregistrer le datum si il est valide et renvoyer un `getdatum`, par exemple) puis réitérer ou terminer si la condition de fin est atteinte.

3.4. Les structures PendingIds et ActivePeers

Pour permettre de gérer l'enregistrement des pairs et la vérification des paquets entrants. On maintient deux structures `PendingIds` et `ActivePeers` qui permettent de garder en mémoire des informations concernant les paquets envoyés et les pairs.

```
struct PendingIds {
      id_to_addr: HashMap<
      [u8; 4],
      ( SocketAddr, PacketType, Instant, usize, bool)
      >,id_to_packet: HashMap<[u8; 4], (Packet, SocketAddr)>
}
```

Permet d'associer un id de paquet a :

- l'adresse à laquelle il a été envoyé
- le type du paquet envoyé
- l'instant où il a été envoyé
- le nombre de tentatives d'envoi
- si une traversée de NAT a été essayée

Cette structure est remplie au moment de l'envoi par la tâche d'envoi, modifiée par la tâche de renvoi lorsque trop de tentatives ont été effectuées. Elle est vidée par la tâche de manipulation lorsqu'elle confirme que le paquet correspond à un paquet envoyé.

```
struct ActivePeers {
      pub addr_map: HashMap<SocketAddr, String>,
      pub peer_map: HashMap<String, Peer>,
}
```

Qui associe une adresse de socket à un nom (les pairs étant identifiés par leur nom, et associe un nom à une structure de pair:

```
struct Peer {
      name: Option<String>,
      addresses: Vec<SocketAddr>,
      root: Option<[u8; 32]>,
      public_key: Option<[u8; 64]>,
      extensions: Option<[u8; 4]>,
      timer: Option<Instant>,
}
```

Qui contient les informations associées à un pair.
La structure est remplie à chaque réception de paquet de type Hello/HelloReply, Root/RootReply et PublicKey/PublicKey reply. Les timeouts sont gérés à chaque réception de paquets du pair.

3.5. La tâche de renvoi

La tâche de renvoi appelle périodiquement `PendingIds::packets_to_resend()` pour obtenir tous les paquets à renvoyer ainsi que les traversées de NAT à amorcer.

3.6. Schéma de la structure