Rapport PSTL :
Cryptographie asynchrone en Rust

Guo Ning
Traore Djadji

24 mai 2019

1 Introduction

Lors de notre projet nous avons fait face à plusieurs nouvelles notion, propres à Rust, nous allons en définir quelques une.

1.1 "Ownership"

Rust n'as pas de "garbage collector", et contrairement au C en Rust on ne doit pas allouer et libérer explicitement de la mémoire. Rust utilise une approche alternative et c'est là qu'intervient ce principe d'ownership, selon le Rust book, il y a trois règles principales à l'ownership :

— Chaque valeur est possédée par une variable "owner"

— Il ne peut y avoir qu'un seul et unique "owner"

— Quand la variable owner sors de son *scope*, la valeur est perdu

Ce qui permet à Rust d'avoir une gestion de la mémoire efficace et sure. Car Rust alloue automatiquement l'espace nécessaire sur la pile pour les variables par défault, et toute les données stockée sur la pile doivent avoir une taille connu à la compilation. Et donc, lorsqu'une variable est défini, celle ci va sur la pile et lorsque l'on sort de son "scope", cette variable est détruite. En Rust, il y a aussi une notion de "tas" mémoire, lorsque l'on met une variable, dont nous ne connaissons pas la taille à la compilation, sur le tas, on demande au système de nous allouer de la mémoire pour cette variable et le système nous renvoi un pointeur à l'adresse où sera la variable.

```
1  fn main() {
2      let y = 5;
3      {
4          let x = 10
5      }
6      foo(y);
7      let s = String::from("hello");
8  }
9
10 fn foo(y : u8){
11     let a = 12;
12     let b = 45;
13 }
```

Décomposons l'exemple si dessus :
Dans notre main, y possède la valeur 5, on l'ajoute à la pile et à la *ligne 3*, on entre dans un autre scope, on ajoute donc x qui possèdent la valeur 10 à la pile. Mais à la *ligne 5*, on sors du scope, donc on supprime x de la pile.
Lorsqu'à la *ligne 10*, on entre dans la fonction foo, à qui on passe y en argument, on stocke sur la pile a et b, possédant les variables 12 et 45 respectivement. Et lorsque l'on sort du scope de foo, on supprime a et b du stack mais on pert aussi l'ownership de y car, il ne peut y a voir qu'un seul owner par variable et lorsque l'on passe y en argument de la fonction, le y présent dans le main perd l'ownership de la variable, on ne peut donc plus utiliser y dans le main après la

fonction foo.
Lorsque l'on arrive à la *ligne 7*, on écrit s possèdant un pointeur, pointant vers l'adresse mémoire de ce string sur la pile mais, la chaine de caractères sera écrite sur le tas.

1.2 "References" et "Borrowing"

Mais comment faire si nous voulons passer une varaible en argument de fonction, sans perdre l'ownership de cette variable ? Rust proposes de "prêter" la variable grâce à son principe de "borrowing", que l'on note grâce à **&**, que l'on appelle "references". Il y a deux type de prêt, le prêt en lecture seule que l'on note grâce à une references qui précèdent, le type de la variable (correspond à x dans l'exemple si dessous) ou la variable qu'elle réferencies (correspond à b dans notre exemple), et le prêt qui nous permet de modifier la variable que l'on note avec &mut avant, le type de la variable (correspond à z dans notre exemple) ou le nom de la variable (correspond à c dans notre exemple). Pour faire un prêt avec modifications, la variable référenciées doit être elle même mutable, que l'on définie grâce au mot clef **mut**, car en rust les variables sont de base immutable.

```
1 fn foo(y : u8, x : &u8, z : &mut u8){
2     let mut a = 10;
3     let b = &a;
4     let c = &mut a;
5 }
```

1.3 Comptage de référence atomique avec Arc<T>

Le type Arc<T> permet de partager d'une valeur de type T entre les threads. La partie "Atomique" signifie qu'on peut accéder à Arc<T> en toute sécurité à partir de plusieurs threads.
Invoquer un clone sur Arc produit une nouvelle instance Arc qui pointe sur la même valeur en augmentant le nombre de références. Exemple :

```
1 let foo = Arc::new(vec![1.0, 2.0, 3.0]);
2 // les deux syntaxes dessous sont equivalents.
3 let a = foo.clone();
4 let b = Arc::clone(&foo);
```

Ici, a, b et foo sont tous des Arcs qui pointent vers le même emplacement mémoire. Ce code crée une nouvelle référence plutôt que de copier le contenu entier de foo.
Voici un exemple de partage de données immuables entre les threads :

```
1 let five = Arc::new(5);
2 for _ in 0..10 {
3     let five = Arc::clone(&five);
4     thread::spawn(move || {
5         println!("{:?}", five);
6     });
7 }
```

Les références partagées dans Rust interdisent des données modifiables par défaut, et Arc ne fait pas exception : on ne peut pas obtenir de référence modifiable à quelque chose à l'intérieur d'un arc. Si on veut pouvoir modifier la valeur, on a besoin d'un type capable d'assurer qu'une seule personne à la fois puisse modifier ce qui est à l'intérieur. Pour cela, on doit utiliser Mutex, RwLock ou l'un des types Atomic.
Voici un exemple d'utilisation d'un Mutex<T> type pour nous permettre de modifier en toute sécurité une valeur partagée entre les threads :

```
fn main() {
    let data = Arc::new(Mutex::new(vec![1, 2, 3]));

    for i in 0..3 {
        let data = data.clone();
        thread::spawn(move || {
            let mut data = data.lock().unwrap();
            data[0] += i;
        });
    }
}
```

1.4 Read-write locks : RwLock<T>

Envisagez une situation où on a plusieurs lecteurs et un seul auteur. On peut protéger cette ressource avec un **Mutex**, mais le problème est que le mutex ne fait aucune distinction entre lecteurs et écrivain. Chaque thread sera obligé d'attendre, peu importe ce que leurs intentions.
RwLock<T> est une alternative au concept mutex, permettant deux types de verrous : lecture et écriture. Il ne peut y avoir qu'un seul verrou en écriture à la fois, mais plusieurs verrous de lecteur. Le paramètre de type T représente les données protégées par ce verrou. Exemple :

```
let lock = RwLock::new(5);
// plusieurs lecteurs peuvent lire a la fois
{
    let r1 = lock.read().unwrap();
    let r2 = lock.read().unwrap();
    assert_eq!(*r1, 5);
}

// un seul verrou en ecriture peut etre maintenu
{
    let mut w = lock.write().unwrap();
    *w += 1;
    assert_eq!(*w, 6);
}
```

Les verrous de lecture sont déposés à ligne 8 et le verrou en écriture est déposé quand il sort de la portée en ligne 14 à la fin.
La fonction read a une signature :

```
pub fn read(&self) -> LockResult<RwLockReadGuard<T>>
```

Elle verrouille ce rwlock avec un accès en lecture partagé, le thread d'appel sera bloqué jusqu'à ce qu'il n'y ait plus d'écrivains qui détiennent le verrou. Il se peut que d'autres lecteurs se trouvent actuellement dans le verrou.
La fonction write a une signature similaire :

```
pub fn write(&self) -> LockResult<RwLockWriteGuard<T>>
```

Cette fonction verrouille ce rwlock avec un accès exclusif en écriture, bloquant le thread actuel tant que d'autres auteur ou lecteurs auront actuellement accès au verrou. Retourne un garde RAII qui libérera l'accès partagé de ce thread une fois qu'il sera déposé.

2 Implémentation de différentes structure

Nous devions implémenter trois structures différentes à partir d'un pseudo code et d'un code C :

```
struct manager {
  struct aes_args args;
  int lens[8];
  long unused_lanes
  struct aes_job *job_in_lane[8]
}

struct aes_args {
  char *input[8];
  char *output[8];
  char iv[8][16];
  char *keys[8];
}

struct aes_job {
  char *plaintext;
  char* ciphertext;
  char iv[16];
  char *keys;
  int len;
  int status; // enum = {BeingProcessed, Completed }
}
```

La structure job contient la chaine de caractère que nous voulons crypter, son vecteur d'initialisation, sa clef aisni qu'un status, pour nous permettre de recupérer le message crypter lorsque le cryptage a été effectuer.
La structure args stocke les différentes entrées des Jobs dans des tableaux.
La structure manager quand a elle, permet de gérer les différents job, grâce à un aes_args et, grâce à sa fonction submit_job permet de lancer le cryptage sur tout les jobs, lorsque l'on a atteint le nombre de job requis.
Pseudo code de submit_job :

```
submit_job(struct manager *state, struct aes_job *job){
  lane <- getUnusedLane(sate->unused_lanes())
  state -> job_in_lane[lane] <- job;
  remove(state->unused_lanes, lane);

```

```
6   state -> args.input[lane] <- job.plaintext
7   state -> output[lane] <- job.ciphertext
8   state -> args.keys[lane] <- job.keys
9   state -> args.iv[lane] <- job.iv
10
11  job -> status <- BeingProcessed;
12  if state -> unused_lane != 0 :
13    retour
14  (min, minIdx) <- getMin(state -> lens)
15  loop 0..min :
16    EncryptX8(input[i],
17      ouput[i], ...)
18  state -> lens[i] -= min for all i
19  for i in 0..8 :
20    if state->lens[i] == 0 :
21      state -> job_in_lane[i] -> status <- Completed
22      state -> job_in_lane[i] = NULL
23      state -> unused_lanes = state -> unused_lanes U {i};
24 }
```

La fonction submit_job, permet d'envoyer un job au manager, qui le stockera dans le aes_args, à l'indice que unused_lane indiquera, étant donnée que unused_lane contient le ou les indices du tableau qui sont vide et peuvent être utilisé. Si unused_lanes est vide, cela siginifie que nous avons huit job, le nombre requis pour commencer le processus de cryptage. Nous lançons donc une boucle sur args qui cryptera chaque input, jusqu'à l'indice de la taille de la chaine ayant la plus petite taille parmi les huit job.
Par exemple, si nous avons huit job et que les tailles des jobs sont [15 ; 25 ; 36 ; 10 ; 58 ; 60 ; 70 ; 35], nous ferons une boucle qui cryptera chaque input de l'indice 0 à l'indice 10, car 10 est la plus petite taille parmi les jobs.
Ensuite on met à jour toute les tailles des chaine de caratère , en leur soustrayant la taille de la plus petite chaine. Si nous reprenons notre exmple, nous allons soustraire 10 à toute les autres tailles, ce qui nous donnerai, [5 ; 15 ; 26 ; 0 ; 54 ; 50 ; 60 ; 25].
Si une des taille est à 0, on modifie son statut et on retire ce job de notre manager.
La fonction encryptX8, prend 8 chaine de même taille et les crypte en parrallèle, mais dans notre implémentation, nous avons utilisé une fausse fonction de cryptage.

2.1 Notre approche

2.1.1 Utilisation de Arc<T>

Au première abord, nous voulions avoir une approche qui ressemble plus à la structure, que nous devions implémenter. Mais nous avions rapidement rencontré des erreurs de "borrow", causé par le ciphertext et le status d'un job, que nous devions modifié dans la fonction submit_job. Et M.Dagand nous a ensuite orienté vers l'utilisation d'Arc et a séparé les attributs ciphertext et status de leur structure Job, en les mettant dans une structure à part, la structure Receipt. Ce qui nous permet, de nous passer aussi de la structure args,

car nous n'avons plus besoin de stocker les valeurs du job dans une structure étant donnée que nous avons accès au ciphertext et au status grâce au Arc.

```
enum Status {
    BeingProcessed,
    Completed
}


struct Job{
    plaintext: Vec<u8>,
    iv: [u8;16],
    keys: Vec<u8>,
    len: usize
}

struct Receipt {
    ciphertext: Vec<u8>,
    status: Status,
}
```

2.1.2 Utilisation de Arc<RwLock<T> >

```

struct Manager {
    jobs: Vec<Job>,
    min_len: usize,
    receipts: Vec<Arc<RwLock<Receipt>>>,
}
```

Pour la structure de Manager, on a utilisé le type Arc<RwLock<T> > pour représenter Receipt, l'avantage d'utiliser ce type est que cela permet à un nombre quelconque de lecteurs d'acquérir Receipt pour lire le ciphertext dans Receipt tant qu'un écrivain ne le détiendra pas.

2.1.3 Implementation de Manager

Dans implementation de Manager, d'abord, on a associée une new() fonction qui fait l'initialisation comme ci-dessous :

```
impl Manager {
    fn new () -> Manager {
        return Manager {
            jobs: Vec::new(),
            receipts: Vec::new(),
             min_len: usize::max_value()
        }
    }
    ...
```

Cela permet à l'utilisateur de créer une nouvelle instance de manager avec Manager : : new(). On initialise la valeur de min_len à une valeur maximum (usize : : max_value ()).

Et puis on implémente la fonction submit_job :

```
1     fn submit_job(&mut self, job: Job) -> Arc<RwLock<Receipt>> {
2
3         let mut submitted = Receipt { ciphertext: Vec::new(),
4                                       status: Status::BeingProcessed };
5
6         submitted.ciphertext = vec![0; job.len];
7         let p = Arc::new(RwLock::new(submitted));
8         self.receipts.push(p.clone());
9         self.min_len = cmp::min(self.min_len, job.len);
10        self.jobs.push(job);
11
12        if self.jobs.len() == 8 {
13            for (i, job) in self.jobs.iter().enumerate() {
14                fake_encrypt(&job.plaintext,
15                             &mut Arc::clone(&self.receipts[i]).
16                             write().unwrap().ciphertext,
17                             &job.keys,
18                             &job.iv,
19                             self.min_len);
20            }
21
22            for i in 0..self.jobs.len() {
23                self.jobs[i].len -= self.min_len;
24                if self.jobs[i].len == 0 {
25                 Arc::clone(&self.receipts[i]).
26                write().unwrap().status = Status::Completed;
27                }
28            }
29
30            self.jobs.retain(|x| x.len != 0);
31
32            self.receipts.retain(|x| x.read().unwrap().status ==
      Status::BeingProcessed);
33
34            self.min_len = self.min_len();
35
36            if self.jobs.len() == 0 {
37                self.min_len = usize::max_value();
38            }
39
40        return p;
41    }
42 }
```

Dans les grandes lignes, cette fonction, fait la même chose que le code en pseudo-code que l'on a vu au-dessus, elle envoie un job au manager, une fois on a obtenu 8 jobs dans manager, on commence le processus de cryptage, quand on finit le cryptage, on met à jour la taille de tous les jobs. Quand un job a fini (ça veut dire son taille est 0), on met le status de receipt correspondant à "Completed".
Mais il y a différences majeur autour du Receipt. Nous créons un reciept, dans la fonction, auquel nous ajoutons un arc. Ce receipt est d'une part stocké dans le manager, mais il est aussi retourner par la fonction pour permettre de récupérer le ciphertext par la suite lorsque que le cryptage aura été effectuer.
Nous utilisons à la *ligne 30*, une fonction sur les vecteurs rust qui garde tous

les éléments qui satisfais la condition. Donc, ici, on garde dans job, tout les arguments qui ont une taille différente de 0.
Nous faisons la même chose avec le vecteur de receipts en fonctions du status des receipts, et retirons, les receipts qui ont terminé leur cryptage.
On met à jour le min_len et si tous les jobs sont finis, on reinitialise min_len.

3 Implémentation de fonctions

3.1 Fonction Poll

On a implémenté une fonction poll comme ci-dessous :

```
fn poll(receipt : & mut Arc<RwLock<Receipt>>) -> Vec<u8>{
    let mut text : Vec<u8> = Vec::new();
    loop {
         if receipt.read().unwrap().status == Status::Completed {
            text = receipt.read().unwrap().ciphertext.to_vec();
            break;
        }else {
            continue;
        }
    }
    return text;
}
```

Cette fonction est bloquante, il fait une attente active, ils vérifient de façon répétée si le status de Receipt est Completed.
Si le receipt est "BeingProcessed", il continue à attendre.
Une fois le status de Receipt est Completed il obtient le verrou en lecture, lire ciphertext dans Receipt et sort de la boucle et envoyer le résultat ensuite.

3.2 Fonction flush_job

Nous devions aussi implémenter une fonction flus_job qui, quand on l'appelle force tout les jobs à se terminer. Nous l'avons donc implémenter ainsi :

```
    fn flush_job(&mut self) {
        let max : usize = self.max_len();

        for job in &mut self.jobs {
            job.plaintext.resize(max, 0 as u8);
            job.keys.resize(max, 0 as u8);
        }

        for rec in &self.receipts {
            Arc::clone(&rec).write().unwrap().ciphertext.resize(max
    ,0);
        }

        for (i, job) in self.jobs.iter().enumerate() {
            fake_encrypt(&job.plaintext,
                         &mut Arc::clone(&self.receipts[i]).write()
    .unwrap().ciphertext,
```

```
                            &job.keys,
                            &job.iv,
                            max);
        Arc::clone(&self.receipts[i]).write().unwrap().ciphertext.
    resize(job.len,0);
        Arc::clone(&self.receipts[i]).write().unwrap().status =
    Status::Completed;
        }

        self.jobs = Vec::new();
        self.receipts = Vec::new();
        self.min_len = usize::max_value();
    }

}
```

Etant donnée que pour la fonction EncryptX8, toute les chaines doivent avoir la même taille, une des intuition que nous avions suivit est que nous tout les inputs devait avoir la taille de la chaine de taille la plus grande. La fonction max_len(), renvoit la taille de la chaine la plus grande, ensuite nous utilisons une fonction de Rust sur les vecteurs, resize qui permet de modifié la taille du vecteur jusqu'à l'indice du première argument de la fonction avec le deuxième argument si la taille passé en argument est plus grande que celle du vecteur sur lequel on applique la fonction, sur le plaintext, le ciphertext et la clef.

Etant donnée que pour la fonction EncryptX8, toute les chaines doivent avoir la même taille, une des intuition que nous avions suivit est que nous tout les inputs devait avoir la taille de la chaine de taille la plus grande. La fonction max_len(), renvoit la taille de la chaine la plus grande, ensuite nous utilisons une fonction de Rust sur les vecteurs, resize qui permet de modifié la taille du vecteur jusqu'à l'indice du première argument de la fonction avec le deuxième argument si la taille passé en argument est plus grande que celle du vecteur sur lequel on applique la fonction, sur le plaintext, le ciphertext et la clef.
Nous utilisons ensuite notre fonction de cryptage sur chaque job. Ensuite nous faisons un resize sur le ciphertext de manière à récuperer seulement la partie du message crypter qui nous intéresse.
Et nous finissons en remettant à vide, le vecteur de Job, de Receipts de job et mettons la taille du min_len à sa valeur par défaut dans notre structure.