Durant cette étude, nous aborderons le développement réseau qui permet de faire communiquer plusieurs machines distantes entre elles. **Rust** fournit deux niveaux d'accès aux services réseau.

*À bas niveau, vous pouvez accéder au support des **sockets** de base du système d'exploitation sous-jacent, ce qui vous permet d'implémenter les clients et les serveurs pour les protocoles orientés connexion.*

***Rust** procure également des bibliothèques qui offrent un accès supérieur à des protocoles réseaux spécifiques au niveau des applications, telles que le **FTP**, le **HTTP**, **SMTP**, etc.*

Principe général d'une communication réseau – les sockets

Pour que nous puissions communiquer en réseau, il faut qu'une machine puisse rendre un ou plusieurs services. Dans ce cas de figure, cette machine est considérée comme un serveur. Bien entendu, sur le même réseau local, plusieurs machines peuvent être des serveurs. Voici quelques principes simples sur la notion de communication réseau :

- *Pour localiser le serveur qui vous intéresse, il faudra connaître son **adresse IP**. Ensuite, sur ce serveur, pour utiliser le bon service auquel vous souhaitez soumettre une question, vous devez l'identifier par son **numéro de port**.*
- *Toutes les autres machines qui souhaitent utiliser ce service sur ce serveur sont considérées comme des machines clientes.*
- *Pour que le client et le service puissent se comprendre et communiquer correctement, il faut qu'ils parlent le même langage, c'est-à-dire qu'ils utilisent le même **protocole**.*
- *Enfin, un serveur doit normalement répondre à tous les clients qui se connectent. Il doit donc prendre en compte une gestion multi-tâche (**multi-threading**).*

Le développement réseau implique la mise en œuvre d'un canal de communication bidirectionnel (dialogue dans les deux sens) entre deux machines dont les deux extrémités (chacune étant en contact avec une machine) représentent les sockets (points de connexion).

Il faudra donc mettre en place une socket côté client, mais aussi une socket côté serveur pour que le canal de communication puisse s'établir. Il va de soit que si le serveur veut répondre à plusieurs clients en même temps, il devra disposer d'autant de sockets qu'il y a de clients (chaque socket représente un client spécifique).

*Nous avons évoqué la notion d'adresse IP (**@IP**). Elle doit être unique dans le réseau local, ce qui permet d'identifier une machine (un système numérique : ordinateur, smartphone, raspberry, etc.), à l'image de notre adresse personnelle pour recevoir correctement notre courrier.*

Au dessus de l'adresse IP, il existe plusieurs modes de transport de l'information. Deux sont couramment utilisés :

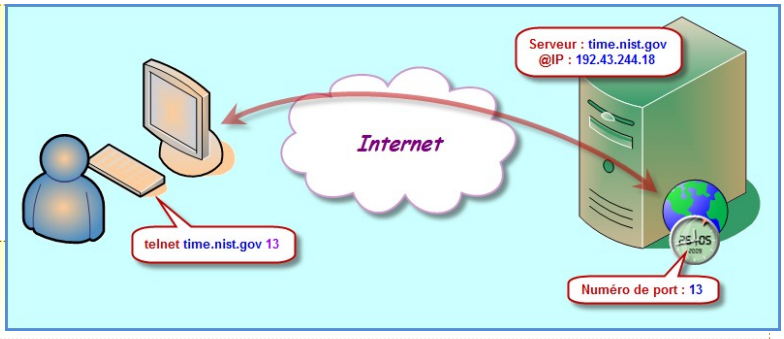
- ***TCP** : (**T**ransfert **C**ontrol **P**rotocol) qui est mode de transport sûr, qui contrôle par un système d'accusé-réception que l'information est bien parvenue à son destinataire. Ce principe est vraiment bien constitué, la contre partie c'est que ce protocole est très verbeux et prend pas mal de temps supplémentaires. Toutefois, c'est celui qui est le plus utilisé.*

  *Ce protocole **TCP** est dit « orienté connexion », c'est-à-dire que les applications sont connectées pour communiquer et que nous pouvons être sûr, quand nous envoyons une information au travers du réseau, qu'elle a bien été réceptionnée par l'autre application. Si la connexion est rompue pour une raison quelconque, les applications doivent rétablir la connexion pour communiquer de nouveau sur le réseau.*
- ***UDP** : (**U**ser **D**atagram **P**rotocol) Il existe des cas de figure, où le temps de communication est primordial, notamment pour les flux audio (téléphone) ou les flux vidéo. Ici, même si certains passages ne passe pas très bien tant-pis, il faut que le flux ne soit pas interrompu. Dans ce cas là, nous n'avons pas besoin de vérifier que chaque paquet envoyé (petite portion d'information) soit bien reçu par le destinataire. Une fois que le canal de communication est mis en place, nous laissons passer le flux sans interruption quoi qu'il arrive, le transfert est alors beaucoup plus rapide. Ainsi, le protocole **UDP** envoie des informations au travers du réseau sans se soucier de savoir si elles seront bien réceptionnées par la cible. Ce protocole n'est pas connecté, une application envoie quelque chose au travers du réseau en spécifiant une cible. Ce type de protocole est utile si vous avez besoin de faire transiter beaucoup d'informations au travers du réseau mais qu'une petite perte occasionnelle d'informations n'est pas très handicapante. Nous trouvons ce type de protocole dans les flux importants de données (musique ou vidéo). Cela fait beaucoup à transmettre mais ce n'est pas dramatique s'il y a une petite perte d'informations de temps à autre puisque, quelques millisecondes plus tard, le serveur renverra de nouveau les informations.*

Première application cliente - connaître la date et l'heure exacte avec le service ITS

Je vous propose de mettre en œuvre notre première application cliente qui interroge un service sur le réseau (ici le réseau Internet). Le service que nous allons interroger est le service de date standard **ITS** (**I**nternet **T**ime **S**ervice) qui utilise le port **« 13 »** et qui se trouve aux États-Unis avec l'URL suivante : **« time.nist.org » (@IP 192.43.244.18)**.

*La plupart des serveurs **UNIX** implémentent en permanence ce système de date. Le serveur auquel nous allons nous connecter se trouve au **NIST** à **Boulder** dans le **Colorado**, et fournit l'heure d'une horloge atomique au césium. Naturellement, le temps affiché n'est pas parfaitement précis à cause des délais de propagation des informations sur le réseau. Par convention, le service date est toujours rattaché au port **13**.*



Le programme (le service) intégré au serveur fonctionne en permanence sur la machine distante, attendant un paquet du réseau qui tente de communiquer avec le port

« 13 ». Lorsque le système d'exploitation de l'ordinateur distant reçoit le paquet contenant une requête de connexion sur le port « 13 », le processus d'écoute du serveur est activé et la connexion est alors établie. Cette connexion demeure jusqu'à ce qu'elle soit arrêtée par l'une des deux parties.

Une fois que la connexion est établie, le service (le programme) de l'ordinateur distant renvoie un ensemble de données, ici une chaîne de caractères correspondant à la date et à l'heure « temps universel », puis il clôture la connexion, puisque le client a obtenu sa réponse. Les ressources utilisées (mémoire interne, tâche concurrente, etc.) sont libérées pour répondre le plus tôt possible à un nouveau client.

CONNEXION AVEC LE SERVICE ITS

La mise en place d'une connexion réseau par le protocole **TCP** se fait au travers de la structure **TcpStream** qui fait partie de la bibliothèque standard. Du coup, il n'est pas nécessaire de gérer de nouvelles dépendances. On imagine bien qu'il existe une structure équivalente pour le protocole **UDP**, elle s'appelle **UdpSocket**.

*La structure **TcpStream** possède un certain nombre de méthodes bien utiles pour gérer la communication avec le service distant. La première démarche consiste à établir la connexion, grâce à la méthode statique **connect()**. Elle propose plusieurs scenarii pour cette mise en communication. Celle que nous proposons ici est de spécifier un seul argument sous forme de chaîne de caractères qui comporte à la fois l'adresse suivi du numéro de port, séparé par un deux-points « **:** ».*

Connexion avec le service (uniquement)

```
use std::net::TcpStream;

fn main() {
  match TcpStream::connect("time.nist.gov:13") {
    Ok(_) => println!("Connexion réussie"),
    Err(_) => println!("Non connecté avec le service")
  }
}
```

Résultat

```
Connexion réussie
```

*Dans l'exemple précédent, l'adresse est une adresse **DNS**. Il est possible de proposer à la place la suite des quatre nombres de l'adresse **Ipv4**, toujours dans la même chaîne unique. Une autre alternative possible est de séparer les deux informations, l'adresse et le port sous la forme d'un tuple, toujours grâce à cette méthode statique **connect()**.*

Connexion avec le service (uniquement)

```
use std::net::TcpStream;

fn main() {
  let adresse = "time.nist.gov";
  let port = 13;
  match TcpStream::connect((adresse, port)) {
    Ok(_) => println!("Connexion réussie"),
    Err(_) => println!("Non connecté avec le service")
  }
}
```

Résultat

```
Connexion réussie
```

*Une autre solution consiste à spécifier chaque nombre séparé d'une adresse **IPv4**. Pour cela, vous devez utiliser la structure **Ipv4Addr** et **SocketAddrV4** chacune ayant la méthode statique **new()** :*

Connexion avec le service (uniquement)

```
use std::net::{TcpStream, Ipv4Addr, SocketAddrV4};

fn main() {
  let ip = Ipv4Addr::new(132, 163, 97, 1);
  let port = 13;
  match TcpStream::connect(SocketAddrV4::new(ip, port)) {
    Ok(_) => println!("Connexion réussie"),
    Err(_) => println!("Non connecté avec le service")
  }
}
```

Résultat

```
Connexion réussie
```

Quelque soit la stratégie choisie, le résultat de la méthode connect() renvoie systématiquement un Result<_>, ce qui me paraît logique puisque la connexion peut bien sûr ne pas aboutir pour plein de raisons différentes. Par exemple, le service n'existe pas ou tout simplement n'est pas encore démarré.

Connaître les adresses IP des différents intervenants

La structure **std::net ::TcpStream** possède bien des méthodes, notamment celles qui permettent de connaître les informations de connexion, l'adresse **IP** et le numéro de **port** utilisé. La méthode pour connaître ces informations sur le **poste local** se nomme **local_addr()**, et la méthode pour connaître le **serveur distant** (appairé) se nomme **peer_addr()**.

Connexion avec le service (uniquement)

```
use std::net::TcpStream;

fn main() {
  let adresse = "time.nist.gov";
  let port = 13;
  match TcpStream::connect((adresse, port)) {
    Ok(connexion) => {
      let local = connexion.local_addr().unwrap();
      let distant = connexion.peer_addr().unwrap();
      println!("PC local => {}", local);
      println!("Serveur  => {}", distant);
      println!("Local    => @IP({}) port({})", local.ip(), local.port());
      println!("Distant  => @IP({}) port({})", distant.ip(), distant.port());
    },
    Err(_) => println!("Non connecté avec le service")
  }
}
```

Résultat

```
PC local  => 192.168.1.23:39024
Serveur  => 128.138.140.44:13
Local    => @IP(192.168.1.23) port(39024)
Distant  => @IP(128.138.140.44) port(13)
```

*Avec ces deux méthodes d'information, nous pouvons discriminer les deux attribut de connexion à l'aide les deux méthodes spécifiques, **ip()** et **port()**. L'identification de l'adresse **IP** se fait ici en **Ipv4**. Si votre réseau local accepte **l'Ipv6**, c'est ce dernier qui sera privilégié dans le résultat affiché.*

Communiquer avec le service

Une fois que la connexion est établie, avec de chaque côté l'implémentation d'une socket, le tube de communication est alors effectif, vous pouvez dès lors communiquer par ce tube. Je rappelle que la communication entre les deux intervenants suppose qu'ils parlent le même langage, c'est-à-dire qu'ils respectent le protocole établi à la fois pour la requête et pour la réponse associée.

*Ici, pour ce service **ITS**, nous n'avons pas besoin de requête venant du client, puisque le fait de se connecter sous-entend que nous désirons connaître l'heure et la date à cet instant précis. Dès la connexion établie, le service renvoie sous forme de texte les informations relatives à ce moment précis de l'intervention.*

La mise en œuvre de cette réponse se fait au travers des flux d'entrée-sortie que nous avons déjà abordé lors d'une étude précédente (normalement, les sockets sont intégrés dans les entrées-sorties puisqu'ils sont similaires aux fichiers par exemple, mais j'ai préféré en faire un cours à part entière).

*Avec la prise en compte de flux d'entrée-sortie, vous possédez de méthodes spécifiques qui gèrent les différents situations de façon relativement simple, soit sous forme d'une suite d'octets, soit sous forme de chaîne de caractères. Dans ce protocole, c'est une chaîne qui est renvoyée, nous utilisons alors la méthode **read_to_string()** de la structure **TcpStream**.*

Communication avec le service – retour de l'instant présent

```
use std::net::TcpStream;
use std::io::prelude::*;

fn main() {
  let adresse = "time.nist.gov";
  let port = 13;
  match TcpStream::connect((adresse, port)) {
    Ok(mut service) => {
      // Lecture de la trame venant du service
      let mut reponse = String::new();
      let _ = service.read_to_string(&mut reponse);
      println!("Réponse du service : {}", reponse.trim());
    },
    Err(_) => println!("Non connecté avec le service")
  }
}
```

Résultat

```
Réponse du service : 59482 21-09-25 12:46:04 50 0 0 540.6 UTC(NIST) *
```

*Le texte de la réponse est relativement riche. Nous allons rajouter des traitements qui permettent de bien récupérer la date et l'heure exacte en France, sachant que la réponse est une heure **UTC** (temps universel).*

*Pour cela, nous utilisons la bibliothèque **chrono** qui permet de bien gérer les dates, les heures, soit en **UTC**, soit en prenant en compte la localité du pays. Pour bénéficier de toutes ces richesses, pensez à proposer la dépendance adéquat.*

Cargo.toml

```
[package]
name = "test-rust"
version = "0.1.0"
authors = ["manu"]
edition = "2018"

[dependencies.chrono]
version = "0.4.19"
features = ["unstable-locales"]
```

Communication avec le service – retour de l'instant présent

```
use std::net::TcpStream;
use std::io::prelude::*;
use chrono::prelude::*;

fn main() {
  let adresse = "time.nist.gov";
  let port = 13;
  match TcpStream::connect((adresse, port)) {
    Ok(mut service) => {
      // Lecture de la trame venant du service
      let mut reponse = String::new();
      let _ = service.read_to_string(&mut reponse);
      println!("Réponse du service : {}", reponse.trim());

      //  Analyse pour récupérer l'heure actuelle en France initialement donné au format UTC
      let decoupe = reponse.trim().split_whitespace().collect::<Vec<_>>();
      let instant = format!("{} {}", decoupe[1], decoupe[2]);
      println!("UTC : {}", instant);
      let utc = Utc.datetime_from_str(instant.as_str(), "%y-%m-%d %H:%M:%S").unwrap();
      let maintenant = Local.from_utc_datetime(&utc.naive_utc());
      println!("{}", maintenant.format_localized("Il est %Hh %Mmn %Ss, le %A %d %B %Y", Locale::fr_FR));
    },
    Err(_) => println!("Non connecté avec le service")
  }
}
```

Résultat

```
Réponse du service : 59482 21-09-25 12:46:04 50 0 0 540.6 UTC(NIST) *
UTC : 21-09-25 12:46:04
Il est 14h 46mn 04s, le samedi 25 septembre 2021
```

Deuxième application qui permet de récupérer une page Web brute sans interprétation

Notre deuxième application nous permet de communiquer au travers du protocole **« HTTP »** et de consulter un site Web sans passer par un navigateur. Un service Web est standardisé, et il est toujours sur le port **« 80 »**.

*Nous verrons ainsi la réponse **« HTTP »** suivi de son contenu qui est une simple page Web, directement avec le balisage **« HTML »** sans interprétation. Cette fois-ci le protocole est différent puisque nous devons soumettre une requête de type **GET** pour avoir la réponse adaptée.*

*Toutefois, le processus est similaire au projet précédent, c'est-à-dire que vous devez d'abord établir la connexion avant de soumettre votre requête. Le protocole **HTTP** offre une communication de base sous forme de chaînes de caractères.*

Connexion à mes cours en ligne, récupération du document HTML

```
use std::net::TcpStream;
use std::io::prelude::*;

fn main() {
  let adresse = "remy-manu.no-ip.biz";
  let port = 80;
  match TcpStream::connect((adresse, port)) {
    Ok(mut service) => {
      service.write(b"GET HTTP/1.0 /\n\n").unwrap();
      let mut html = String::new();
      service.read_to_string(&mut html).unwrap();
      println!("{}", html);
    },
    Err(_) => println!("Non connecté avec le service")
  }
}
```

Résultat

```
HTTP/1.1 400 Bad Request                                  // En-tête de la réponse du protocole HTTP
Server: nginx
Date: Sat, 25 Sep 2021 16:10:46 GMT
Content-Type: text/html
Content-Length: 150
Connection: close

<html>                                                    // Document HTML
<head><title>400 Bad Request</title></head>
<body>
<center><h1>400 Bad Request</h1></center>
<hr><center>nginx</center>
</body>
</html>
```

*Pour ce projet, nous avons besoin de la méthode **write()** de **TcpStream** sachant que pour que la communication soit correcte, n'oubliez pas que, même s'il s'agit d'un texte, il doit être envoyé sous forme de flux d'octets, d'où le préfixe « **b** ».*

*Ici, nous demandons à consulter une page Web (la page d'accueil **« / »**), c'est donc plutôt la méthode **GET** qui est la plus adaptée. C'est ce qu'utilise un navigateur (c'est la requête par excellence).*

PREMIER SERVICE – MONO-PASSE – MONO-UTILISATEUR

Avec **Rust** nous pouvons également créer nos propres services, ce qui est bien sûr beaucoup plus intéressant. Comme nous restons dans le domaine de la connexion réseau, nous continuons à utiliser les sockets que nous connaissons déjà. Je dis des sockets puisqu'il en faut une pour le service lui-même, et ensuite autant de sockets qu'il y a de clients connectés. Afin de bien maîtriser tous les mécanismes en jeu, je vous propose de réaliser un service très rudimentaire qui contrôle uniquement la phase de connexion avec un client.

*Il existe la structure **TcpListener** spécialisée pour la mise en œuvre d'un service. La méthode **bind()** de cette structure permet de se mettre à l'écoute d'une connexion possible sur un **port** donné. Elle prend exactement les mêmes types de paramètre que pour la méthode **connect()** de **TcpStream**.*

*Il ne reste ensuite plus qu'à attendre la connexion d'un client à l'aide de la méthode bloquante **accept()**. En cas de réussite, cette méthode renvoie un **tuple** contenant un **TcpStream** (la socket du client) et un **SocketAddr** (l'adresse du client).*

*Pour tester ce processus, vous devez d'abord lancer le programme du service et ensuite l'application cliente (par exemple **telnet**). Une fois que la communication est établie, le programme du service s'arrête puisque nous n'avons pas mis en place une boucle pour écouter le réseau en permanence.*

Création d'un service mono-passe, mono-utilisateur

```rust
use std::net::TcpListener;

fn main() {
  let adresse = "192.168.1.23";
  let port = 7878;
  match TcpListener::bind((adresse, port)){
    Ok(service) => {
      println!("En attente d'un client...");
      match service.accept() {
        Ok((client, adresse)) => println!("Nouveau client [adresse : {}]", adresse),
        Err(_) => println!("Un client a tenté de se connecter")
      }
    },
    Err(_) => println!("Le service ne peut être activé (port déjà utilisé) !")
  }
}
```

Résultat

```
En attente d'un client...
Nouveau client [adresse : 192.168.1.23:50030]
```

SERVICE MULTI-UTILISATEUR

Gérer un seul client, c'est bien, mais qu'en est-il si nous voulons en gérer plusieurs ? La solution évidente est qu'il vous suffit de boucler sur l'appel de la méthode **accept()** et de gérer chaque client dans un thread séparé. **Rust** fournit aussi une nouvelle méthode i**ncoming()** qui permet de gérer cela de façon plus élégante, en utilisant une boucle **for** pour prendre en compte chaque client connecté.

Création d'un service multi-passe, multi-utilisateur

```rust
use std::net::TcpListener;

fn main() {
  let adresse = "192.168.1.23";
  let port = 7878;
```

```
  match TcpListener::bind((adresse, port)){
    Ok(service) => {
      println!("En attente d'un client...");
      for connexion in service.incoming() {
        match connexion {
          Ok(client) => println!("Nouveau client [adresse : {}]", client.peer_addr().unwrap()),
          Err(_) => println!("Un client a tenté de se connecter")
        }
      }
    },
    Err(_) => println!("Le service ne peut être activé (port déjà utilisé) !")
  }
}
```

Résultat

```
En attente d'un client...
Nouveau client [adresse : 192.168.1.23:55132]
Nouveau client [adresse : 192.168.1.23:55140]
Nouveau client [adresse : 192.168.1.23:55146]
```

*La méthode **incoming()** renvoie un **Result<TcpStream>**. Après vérification, nous sommes donc directement avec la socket qui représente le client qui vient de se connecter. Nous pouvons alors prendre exactement les mêmes méthodes que nous avons utilisées lors de l'implémentation d'une application cliente connectée à un service.*

*Si le client se connecte, envoie sa requête, récupère la réponse et se déconnecte tout de suite après, le temps de traitement du service étant généralement très court, nous n'avons pas besoin d'implémenter un système **multi-threading**. Par contre, si nous mettons en œuvre un service tchat par exemple, les clients restent constamment connectés, vous devez alors prévoir un **thread** pour chaque client.*

SERVICE MULTI-UTILISATEUR MULTI-TÂCHE

Je vous propose maintenant de réaliser un traitement spécifique pour chacun des clients connectés. Ce service réalise un calcul de financement d'un crédit à la consommation. Il fournit alors le calcul des mensualités, du nombre de mensualités, du coût total du crédit avec les intérêts en spécifiant côté client, le capital, le nombre d'années et le taux d'intérêt.

*Le protocole utilisé pour échanger ces différentes informations se traduit par l'envoi et le retour de chaînes de caractères. Ensuite, pour séparer les différentes informations, à la fois pour la requête et pour la réponse, nous choisissons le caractère « **:** ».*

Création d'un service multi-passe, multi-utilisateur multi-tâche

```
mod client;

use std::net::{TcpListener, TcpStream};
use std::thread::spawn;
use client::traitement_client;

fn main() {
  let adresse = "172.16.20.31";
  let port = 7878;
  match TcpListener::bind((adresse, port)){
    Ok(service) => {
      println!("En attente d'un client...");
      for connexion in service.incoming() {
        match connexion {
          Ok(client) => { spawn(move || { traitement_client(client)}); },
          Err(_) => println!("Un client n'a pas réussi à se connecter")
        }
      }
    },
    Err(_) => println!("Le service ne peut être activé (port déjà utilisé) !")
  }
}
```

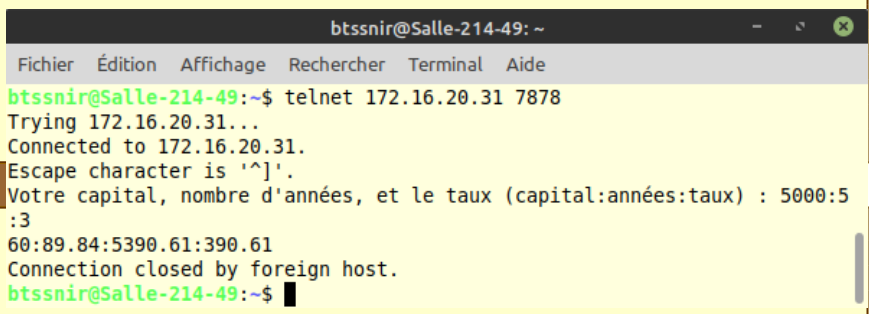
*Nous retrouvons le même code précédent avec cette fois-ci la mise en œuvre d'une tâche concurrente lorsqu'une nouvelle connexion se présente. Cela se fait vraiment très simplement, il suffit de lancer la fonction **std::thread ::spawn()** que nous connaissons déjà et que nous avons abordé lors de l'étude précédente. Le choix du multi-tâche me paraît ici judicieux puisque si nous testons la communication avec le client « **telnet** », l'utilisateur met pas mal de temps pour confectionner sa requête.*

*Dans cette fonction **spawn()**, vous précisez la clôture du traitement à réaliser pour résoudre la requête venant du client. Afin de bien séparer le rôle du service et le traitement en coulisse, il est plus judicieux de proposer une fonction, ici **traitement_client()**, qui réalise le traitement spécifique pour chacun des clients connectés. Dans l'absolu, cette fonction peut être placée dans un fichier séparé, « **client.rs** ».*

client.rs

```
use std::str::*;
use std::io::{Write, Read};
use std::net::TcpStream;
```

```rust
pub fn traitement_client(mut client: TcpStream) {
    // Informations sur le client
    println!("Nouveau client [adresse : {}]", client.peer_addr().unwrap().ip());
    // Communication avec le client (envoi d'un message et retour de la requête
    client.write("Votre capital, nombre d'années, et le taux (capital:années:taux) : ".as_bytes());
    let mut reponse = &mut [0; 30];
    let nombre = client.read(reponse).unwrap();
    println!("Nombre d'octets : {}", nombre);
    // Mise en oeuvre des informations relatives de la requête
    let texte = from_utf8(&reponse[0..nombre]).unwrap().trim();
    println!("{}", texte);
    let valeurs =  texte.split(':').collect::<Vec<_>>();
    let capital = valeurs[0].parse::<f64>().unwrap();
    let ans = valeurs[1].parse::<f64>().unwrap();
    let taux = valeurs[2].parse::<f64>().unwrap() / 100.;
    println!("Capital {} sur {} ans avec un taux de {} %", capital, ans, taux);
    // Traitements spécifiques, calcul du finacement
    let mois = ans*12.;
    let mensualite = capital*taux/12./(1.-f64::powf(1.+taux/12., -ans*12.));
    let total = mois*mensualite;
    let interets = total-capital;
    println!("Mois={}, Mensualité={:.2}, Coût={:.2}, Intérêts={:.2}", mois, mensualite, total, interets);
    // Mise en oeuvre du protocole et envoi de la réponse
    let mut resultat = format!("{}:{:.2}:{:.2}:{:.2}\n", mois, mensualite, total, interets);
    client.write(resultat.as_bytes());
}
```



Résultat

```
En attente d'un client...
Nouveau client [adresse : 172.16.20.16]
Nombre d'octets : 10
5000:5:3
Capital 5000 sur 5 ans avec un taux de 0.03 %
Mois=60, Mensualité=89.84, Coût=5390.61, Intérêts=390.61
```

*La seule remarque par rapport à la communication précédente sur la récupération de l'heure universelle, c'est que cette fois-ci, nous utilisons la méthode **read()** de **TcpStream** plutôt que la méthode **read_to_string()**. Ce choix est justifié dans le fait que la deuxième méthode peut prendre plusieurs lignes de texte en une seule fois. Ici, cela pose problème puisque depuis le client **telnet** par exemple, nous devons tout-de-suite soumettre la requête dès que l'utilisateur appui sur la touche « **Entrée** ».*

*Je rappelle que la méthode **read()** attend un argument correspondant à un tableau d'octets mutable, afin qu'il soit automatiquement rempli par l'ensemble des octets de la requête venant du réseau, et donc de l'application cliente à distance. La capacité du tableau doit être suffisante pour absorber la totalité de la requête.*

*Dans le même registre, lorsque nous soumettons une chaîne de caractères dans un flux réseau, il faut qu'elle soit considérée comme un flux d'octets, d'où l'utilisation systématiquement de la méthode **as_bytes()** de la structure **String**.*

COMMUNIQUER EN FORMAT JSON

JSON est un format léger pour l'échange de données structurées complexes. Il est à l'image des documents **XML** en moins verbeux. Il est très utile lorsque vous devez transférer toutes les informations relatives à une structure ou à un objet persistant par exemple. Voici ci-dessous un exemple de document **JSON** représentant un objet de type **Personne** qui peut disposer de plusieurs numéros de téléphones :

```
{
  "id" : 51,
  "nom" : "PÊCHEUR",
  "prénom" : "Martin",
  "naissance" : 18/11/1969,
  "téléphones" : ["04-45-18-99-77", "06-89-89-87-23", "04-72-33-55-84"]
}
```

L'ossature du document ressemble à une structure dont le début et la fin sont désignés par des accolades. Chaque élément du document possède une clé et une valeur associée. L'ensemble des éléments sont séparés par des virgules. Pour la définition de la clé et de la valeur, vous devez l'écrire entre guillemets, sauf éventuellement pour les valeurs numériques. Enfin, si une clé possède plusieurs valeurs, vous devez les spécifier entre des crochets séparées par des virgules et toujours écrites entre guillemets.

*Dans **Rust**, l'implémentation du format **JSON** se fait au travers de la librairie « **serde** ». L'idée générale est de travailler avec le type **struct** dont la mise en œuvre doit donner image précise du document **JSON** à soumettre ou à récupérer. La technique*

utilisée est une sérialisation (et une dé-sérialisation dans l'autre sens) où chacune des valeurs de chaque attribut de la structure est automatiquement transformé dans le bon type interne (là aussi dans les deux sens).

*Afin de bien maîtriser le principe de fonctionnement, nous reprendrons le projet précédent auquel nous allons rajouter une soumission de la requête au format **JSON** avec également la réponse dans ce type de format.*

Cargo.toml

```
[package]
name = "test-rust"
version = "0.1.0"
authors = ["manu"]
edition = "2018"

[dependencies]
serde = {version = "1.0.130", features = ["derive"]}
serde_json = "1.0.68"
```

json.rs

```
use serde::{Serialize, Deserialize};

#[derive(Debug, Deserialize)]
pub struct Requete {
  pub capital: f32,
  pub annees: f32,
  pub taux: f32
}

#[derive(Debug, Serialize)]
pub struct Reponse {
  pub mois: f32,
  pub mensualite: f32,
  pub total: f32,
  pub interets: f32
}
```

*Pour notre projet, je vous propose de rajouter un fichier source qui permet de créer les deux structures pour formaliser les documents **JSON**, une spécifique pour la requête, l'autre bien entendu pour la réponse.*

*Vous remarquez la présence de dérivations qui permettent de stipuler dans quel sens s'effectue la sérialisation. Pour la requête, nous devons transformer la chaîne au format **JSON** directement en une variable de type « **Requete** » (**dé-sérialisation**). Pour la réponse, le traitement est inversé. À partir d'une variable de type « **Reponse** », nous générons automatiquement une chaîne de caractères au format **JSON** (**sérialisation**).*

main.rs

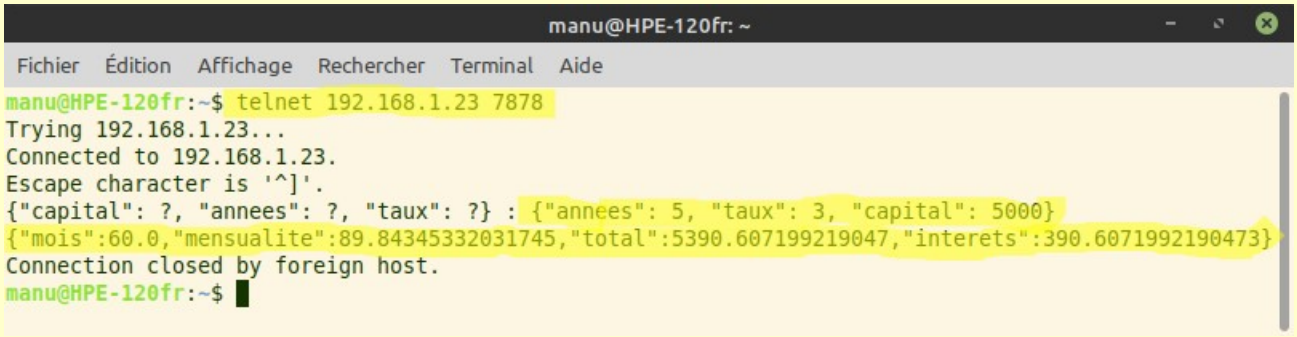
```
mod client;
mod json;

use std::net::{TcpListener};
use std::thread::spawn;
use client::traitement_client;

fn main() {
  let adresse = "0.0.0.0";
  let port = 7878;
  match TcpListener::bind((adresse, port)){
    Ok(service) => {
      println!("En attente d'un client...");
      for connexion in service.incoming() {
        match connexion {
          Ok(client) => { spawn(move || { traitement_client(client)}); },
          Err(_) => println!("Un client n'a pas réussi à se connecter")
        }
      }
    },
    Err(_) => println!("Le service ne peut être activé !")
  }
}
```

*Le fichier source de la fonction principale reste identique si ce n'est la prise en compte du nouveau module « **json.rs** ». Vous remarquez également que les champs de l'adresse IP sont tous à zéro. Cela permet de spécifier que c'est l'adresse du poste hôte qui est pris comme référence.*

client.rs

```
use std::str::*;
use std::io::{Write, Read};
use std::net::TcpStream;
```

```rust
use crate::json::{Requete, Reponse};

pub fn traitement_client(mut client: TcpStream) {
    // Informations sur le client
    println!("Nouveau client [adresse : {}]", client.peer_addr().unwrap().ip());
    // Communication avec le client (envoi d'un message et retour de la requête
    client.write(r#"{"capital": ?, "annees": ?, "taux": ?} : "#.as_bytes());
    let reponse = &mut [0; 100];
    let nombre = client.read(reponse).unwrap();
    println!("Nombre d'octets : {}", nombre);

    // Mise en oeuvre des informations relatives de la requête
    let texte = from_utf8(&reponse[0..nombre]).unwrap().trim();
    println!("{}", &texte);
    let requete : Requete = serde_json::from_str(&texte).unwrap();
    println!("{:?}", requete);
    let capital = requete.capital;
    let ans = requete.annees;
    let taux = requete.taux / 100.;

    // Traitements spécifiques, calcul du financement
    let mois = ans*12.;
    let mensualite = capital*taux/12./(1.-f32::powf(1.+taux/12., -ans*12.));
    let total = mois*mensualite;
    let interets = total-capital;
    let reponse = Reponse {mois, mensualite, total, interets};
    println!("{:?}", &reponse);

    // Mise en oeuvre du protocole et envoi de la réponse
    let resultat = serde_json::to_string(&reponse).unwrap();
    client.write(resultat.as_bytes());
}
```



*Une fois que les structures sont créées, il est très facile de récupérer les informations d'un document **JSON** au moyen de la fonction **serde_json ::from_str()**. Pour l'opération inverse, il existe la fonction **serde_json::to_string()**.*

*Dans l'application cliente « **telnet** », vous devez donc soumettre votre nouvelle requête cette fois-ci au format **JSON**. Vous remarquez toutefois que l'ordre des attributs n'est pas fixé, il suffit que tous les éléments soient bien pris en compte.*

Résultat

```
En attente d'un client...
Nouveau client [adresse : 192.168.1.23]
Nombre d'octets : 43
{"annees": 5, "taux": 3, "capital": 5000}
Requete { capital: 5000.0, annees: 5.0, taux: 3.0 }
Reponse { mois: 60.0, mensualite: 89.84345332031745, total: 5390.607199219047, interets: 390.6071992190473 }
```

*Le code précédent fonctionne très bien, en supposant toutefois que le format **JSON** de la requête soit parfaitement bien formé. Si ce n'est pas le cas, le service risque d'avoir un comportement assez aléatoire. Il faut donc tenir compte de cette problématique.*

client.rs

```rust
use std::str::*;
use std::io::{Write, Read};
use std::net::TcpStream;
use crate::json::{Requete, Reponse};

pub fn traitement_client(mut client: TcpStream) {
    // Informations sur le client
    println!("Nouveau client [adresse : {}]", client.peer_addr().unwrap().ip());
    // Communication avec le client (envoi d'un message et retour de la requête
    client.write(r#"{"capital": ?, "annees": ?, "taux": ?} : "#.as_bytes());
    let reponse = &mut [0; 100];
```

```rust
    let nombre = client.read(reponse).unwrap();
    println!("Nombre d'octets : {}", nombre);

    // Mise en oeuvre des informations relatives de la requête
    let texte = from_utf8(&reponse[0..nombre]).unwrap().trim();
    println!("{}", &texte);
    match serde_json::from_str::<Requete>(&texte) {
        Ok(requete) => {
            println!("{:?}", requete);
            let capital = requete.capital;
            let ans = requete.annees;
            let taux = requete.taux / 100.;

            // Traitements spécifiques, calcul du financement
            let mois = ans*12.;
            let mensualite = capital*taux/12./(1.-f32::powf(1.+taux/12., -ans*12.));
            let total = mois*mensualite;
            let interets = total-capital;
            let reponse = Reponse {mois, mensualite, total, interets};
            println!("{:?}", &reponse);

            // Mise en oeuvre du protocole et envoi de la réponse
            let resultat = serde_json::to_string(&reponse).unwrap();
            client.write(resultat.as_bytes());
        }
        Err(_) => { client.write(b"Votre format JSON n'est pas correct !"); }
    }
}
```

```
manu@HPE-120fr: ~
Fichier  Édition  Affichage  Rechercher  Terminal  Aide
manu@HPE-120fr:~$ telnet 192.168.1.23 7878
Trying 192.168.1.23...
Connected to 192.168.1.23.
Escape character is '^]'.
{"capital": ?, "annees": ?, "taux": ?} : {"capital": 5000, "taux":3}
Votre format JSON n'est pas correct !Connection closed by foreign host.
manu@HPE-120fr:~$ telnet 192.168.1.23 7878
Trying 192.168.1.23...
Connected to 192.168.1.23.
Escape character is '^]'.
{"capital": ?, "annees": ?, "taux": ?} : {"capital": 5000, "annees": 5, "taux": 3}
{"mois":60.0,"mensualite":89.84345332031745,"total":5390.607199219047,"interets":390.6071992190473}
Connection closed by foreign host.
manu@HPE-120fr:~$
```

Résultat

```
En attente d'un client...
Nouveau client [adresse : 192.168.1.23]
Nombre d'octets : 29
{"capital": 5000, "taux":3}
Nouveau client [adresse : 192.168.1.23]
Nombre d'octets : 43
{"capital": 5000, "annees": 5, "taux": 3}
Requete { capital: 5000.0, annees: 5.0, taux: 3.0 }
Reponse { mois: 60.0, mensualite: 89.84345332031745, total: 5390.607199219047, interets: 390.6071992190473 }
```

RÉALISATION DE LA PARTIE CLIENTE

À titre d'exercice, je vous propose de réaliser une application cliente qui utilisera ce service de façon beaucoup plus simple que l'application « **telnet** ». Nous renseignerons uniquement les données utiles au service en générant automatiquement le protocole souhaité avec le transfert de documents **JSON** adaptés.

> *Bien entendu, pour réaliser cette application, nous avons de nouveau besoin de la librairie « **serde** », d'une part et nous utiliserons le même type de fichier source « **json.rs** » afin de respecter le choix du protocole souhaité par le service. Seul le sens des sérialisations est inversé par rapport au service.*

Cargo.toml

```
[package]
name = "client-finance"
version = "0.1.0"
edition = "2018"

[dependencies]
serde = {version = "1.0.130", features = ["derive"]}
serde_json = "1.0.68"
```

json.rs

```rust
use serde::{Serialize, Deserialize};

#[derive(Debug, Serialize)]
pub struct Requete {
  pub capital: f32,
  pub annees: f32,
  pub taux: f32
}

#[derive(Debug, Deserialize)]
pub struct Reponse {
  pub mois: f32,
  pub mensualite: f32,
  pub total: f32,
  pub interets: f32
}
```

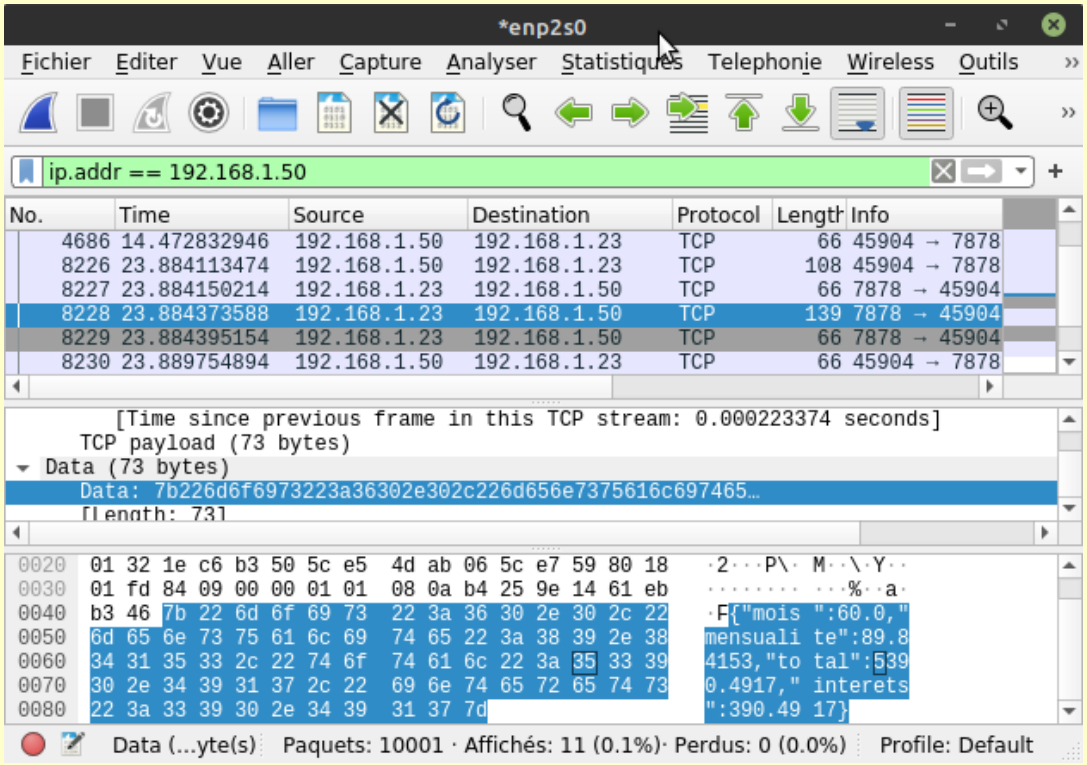
main.rs

```rust
mod json;

use std::io::{stdin, stdout, Write, Read};
use std::net::TcpStream;
use crate::json::{Requete, Reponse};
use std::str::from_utf8;

fn main() {
    let adresse = "192.168.1.23";
    let port = 7878;
    match TcpStream::connect((adresse, port)) {
        Ok(mut service) => {
            println!("Projet de financement");
            println!("---------------------");
            envoi_requete(&mut service);
            recuperation_reponse(&mut service);
        },
        Err(_) => println!("Non connecté avec le service")
    }
}

fn envoi_requete(service: &mut TcpStream) {
    let capital = saisie("Capital");
    let annees = saisie("Nombre d'années");
    let taux = saisie("Taux");
    let requete = Requete {capital, annees, taux};
    let envoi = serde_json::to_string(&requete).unwrap();
    service.write(envoi.as_bytes());
}

fn recuperation_reponse(service: &mut TcpStream) {
    let mut reponse = String::new();
    service.read_to_string(&mut reponse).unwrap();
    match serde_json::from_str::<Reponse>(reponse.as_str()) {
        Ok(resultat) => {
            println!("Nombre de mois : {}", resultat.mois);
            println!("Mensualités : {:.2} €", resultat.mensualite);
            println!("Coût total : {:.2} €", resultat.total);
            println!("Intérêts : {:.2} €", resultat.interets);
        },
        Err(retour) => println!("Format réponse non valide ! {:?}", retour)
    }
}

fn saisie(intitule: &str) -> f32 {
    let mut valeur = String::new();
    loop {
        print!("{} : ", intitule);
        stdout().flush().unwrap();
        stdin().read_line(&mut valeur).unwrap();
        match valeur.trim().parse() {
            Ok(x) =>  return x,
            Err(_) => { println!("Il faut une valeur numérique ! ") }
        }
    }
}
```

Résultat

Projet de financement

Capital : 5000
Nombre d'années : 5
Taux : 3
Nombre de mois : 60
Mensualités : 89.84 €
Coût total : 5390.49 €
Intérêts : 390.49 €



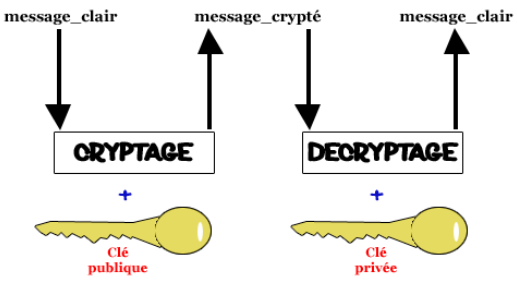
*Notre application est découpée en quatre parties, le programme principal, une fonction pour soumettre la requête en formatant le contenu au format **JSON**, une fonction pour récupérer le réponse au format **JSON** interprété pour un affichage circonstancié, et une fonction qui permet de contrôler la saisie pour que les valeurs proposées soient bien des valeurs numériques.*

*À la récupération de la réponse, nous sollicitons directement la méthode **read_to_string()** sans passer par un tableau d'octets puisque nous sommes sûr que le service envoi bien un texte complet au format **JSON**.*

*Grâce à un analyseur de trame, nous pouvons remarquer le format **JSON** lors de la soumission de la réponse du serveur avec les différentes valeurs intéressantes du financement. Ces informations apparaissent en clair, ce qui peut poser des problèmes pour des données sensibles, notamment pour les mots de passes associés à des comptes particuliers. Il serait souhaitable de passer par un système de communications cryptées.*

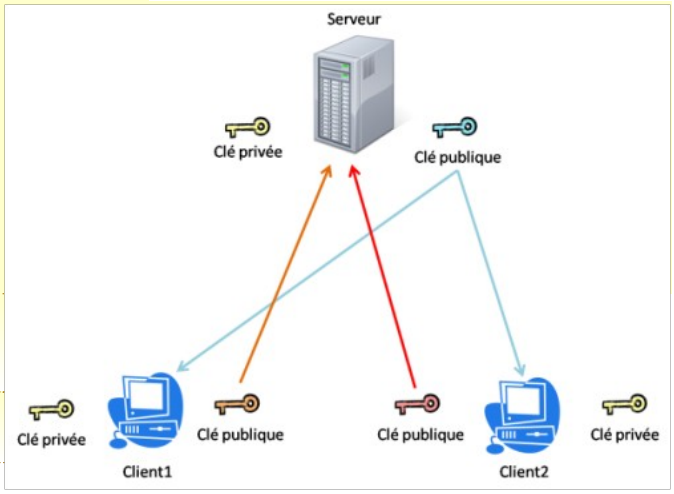
CRYPTAGE

Afin d'éviter que les messages de communication soit directement visible dans un analyseur de trame, il serait souhaitable de crypter les informations avant de les envoyer afin qu'elles ne soient pas compréhensible par un être humain. Après réception, bien entendu, ces informations doivent être décrypter afin de retrouver les contenus originaux. Nous trouvons principalement deux grandes familles de cryptographie : la **cryptographie symétrique** (ou dite à clé secrète) et la **cryptographie asymétrique** (dite aussi à clé publique).



***La cryptographie symétrique :** Nous parlons de cryptographie symétrique lorsqu'un texte, document, etc. est chiffré et déchiffré avec la même clé, la clé secrète. Ce procédé est à la fois simple et sûr. Principal inconvénient : étant donné que nous possédons qu'une clé, si vous la donnez à quelqu'un pour qu'il puisse vous envoyer des messages chiffrés avec celle-ci, il pourra aussi bien déchiffrer tous les autres documents que vous avez chiffrés avec cette dernière.*

***La cryptographie asymétrique :** Contrairement à la cryptographie symétrique, ici nous avons besoin de deux clés, une clé publique et une clé privée. La clé publique, tout le monde peut la posséder, il n'y a aucun risque, vous pouvez la transmettre à n'importe qui. Elle sert à chiffrer le message. Mais, il existe aussi la clé privée que seul le récepteur possède. Elle sert à déchiffrer le message chiffré avec la clé publique.*



*La **clé privée** doit absolument rester **privée**, c'est pour cela qu'elle est **quelquefois** stockée dans un fichier crypté par un mot de passe appelée « **passphrase** ».*

*La **clef privée** n'est jamais transmise à personne alors que la clef publique est transmissible sans restrictions. Ainsi, à l'établissement*

de la connexion, les différents acteurs s'échangent leurs clés publiques afin d'avoir une communication totalement chiffrée, ***nous*** *parlons alors de transaction.*

La technique de mise en œuvre se fait au travers du système de chiffrement ***RSA*** *qui fait partie des algorithmes de cryptographie asymétrique, celui utilisé notamment par l'API* ***openssl****.* ***RSA*** *utilise un algorithme très sophistiqué donc plus sûr qui prend en compte* ***plusieurs*** *nombres premiers de plus de* ***100*** *chiffres chacun. Comme nous ne connaissons toujours pas la progression des nombres premiers, il est alors extrêmement difficile de retrouver le message original non crypté.*

Avant de voir comment communiquer entre deux systèmes numériques différents sur le réseau avec le chiffrage des conversations, je vous propose de réaliser des expériences avec la librairie interne de ***Rust*** *qui utilise intrinsèquement les compétences de* ***openssl****. Pour cela, vous devez intégrer cette librairie au sein de votre projet.*

Cargo.toml

```
[package]
name = "test-rust"
version = "0.1.0"
authors = ["manu"]
edition = "2018"

[dependencies]
openssl = { version ="0.10.36", features = ["vendored"] }
```

main.rs

```
use openssl::rsa::{Padding, Rsa};
use openssl::symm::Cipher;
use std::fs::File;
use std::io::Write;

fn main() {
  // Cryptée une données à partir de RSA
  let rsa = Rsa::generate(512).unwrap();
  let message = "bonjour";
  let mut cryptage = vec![0; rsa.size() as usize];
  let _ = rsa.public_encrypt(message.as_bytes(), &mut cryptage, Padding::PKCS1).unwrap();
  println!("Message initial : '{}'", message);
  println!("Cryptage vecteur : ({} éléments) {:?}", cryptage.len(), cryptage);
  println!("Cryptage caractères : {}", String::from_utf8_lossy(cryptage.as_slice()));

  // Décrypter le message crypté
  let mut recuperation = vec![0; rsa.size() as usize];
  let taille = rsa.private_decrypt(&cryptage, &mut recuperation, Padding::PKCS1).unwrap();
  let resultat = &recuperation[..taille];
  println!("Résultat du décryptage : {}\n", String::from_utf8_lossy(resultat));

  // Retrouver un RSA à l'aide de la clé privée d'un RSA déjà généré
  let cle_privee = rsa.private_key_to_pem().unwrap();
  println!("{}", String::from_utf8(cle_privee.to_vec()).unwrap());
  println!("Taille RSA : {}", rsa.size());
  let rsa = Rsa::private_key_from_pem(cle_privee.as_slice()).unwrap();

  // Décrypter le message crypté au tout début avec le nouvel RSA
  let mut recuperation = vec![0; rsa.size() as usize];
  let taille = rsa.private_decrypt(&cryptage, &mut recuperation, Padding::PKCS1).unwrap();
  let resultat = &recuperation[..taille];
  println!("Résultat du décryptage : {}", String::from_utf8_lossy(resultat));

  // Génération du fichier associé à la clé privée
  if let Ok(mut sauvegarde) = File::create("serveur.crt") {
    sauvegarde.write(cle_privee.as_slice()).unwrap();
  }

  // Création d'une nouvelle clé privée avec une passphrase
  let passphrase = "manu".as_bytes();
  let rsa = Rsa::generate(512).unwrap();
  let cle_privee = rsa.private_key_to_pem_passphrase(Cipher::aes_128_cbc(), passphrase).unwrap();
  println!("\n{}", String::from_utf8(cle_privee.to_vec()).unwrap());

  // Cryptage du message et décryptage du message avec la passphrase
  let _ = rsa.public_encrypt(message.as_bytes(), &mut cryptage, Padding::PKCS1).unwrap();
  println!("Cryptage caractères : {}", String::from_utf8_lossy(cryptage.as_slice()));
  let rsa = Rsa::private_key_from_pem_passphrase(cle_privee.as_slice(), passphrase).unwrap();
  let taille = rsa.private_decrypt(&cryptage, &mut recuperation, Padding::PKCS1).unwrap();
  let resultat = &recuperation[..taille];
  println!("Résultat du décryptage avec passphrase : {}", String::from_utf8_lossy(resultat));

  // Génération d'une clé publique pour diffusion
  let cle_publique = rsa.public_key_to_pem().unwrap();
  println!("\n{}", String::from_utf8(cle_publique.to_vec()).unwrap());
```

```
    // Cryptage du message cette fois-ci à l'aide de la clé publique
    let rsa = Rsa::public_key_from_pem(cle_publique.as_slice()).unwrap();
    let mut cryptage = vec![0; rsa.size() as usize];
    let _ = rsa.public_encrypt(message.as_bytes(), &mut cryptage, Padding::PKCS1).unwrap();
    println!("Cryptage caractères : {}", String::from_utf8_lossy(cryptage.as_slice()));

    // Décryptage du messae avec la passphrase
    let rsa = Rsa::private_key_from_pem_passphrase(cle_privee.as_slice(), passphrase).unwrap();
    let taille = rsa.private_decrypt(&cryptage, &mut recuperation, Padding::PKCS1).unwrap();
    let resultat = &recuperation[..taille];
    println!("Résultat du décryptage avec passphrase : {}", String::from_utf8_lossy(resultat));
}
```

Résultat

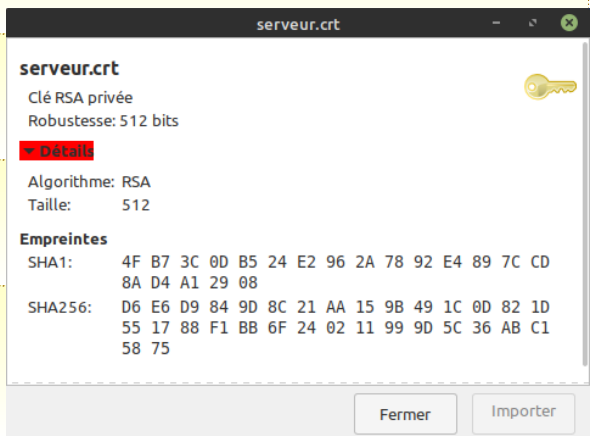
```
Message initial : 'bonjour'
Cryptage vecteur : (64 éléments) [109, 106, 148, 155, 209, 46, 81, 58, 151, 102... ]
Cryptage caractères : mj���.Q:�f#X>�2?�(��f2#Q�!8B�}.R����Deq��#]W��35�6��#J#��G�qj�b�
Résultat du décryptage : bonjour

-----BEGIN RSA PRIVATE KEY-----
MIIBOgIBAAJBAOa0ZLAIf28PdeNBDb8KUZ85o01KRgPDrzJbFTWDlFMAq7qU5Cf7
7UJXc9lWyf9tLOM6LWVO2pH8ITcQR0KqjWUCAwEAAQJAa1eR5an+iRE7e8J//9xw
P9KUndiqHQRGlvp0MtMHB1rNOo+1RM/GhY9wGMvI9YAnw11QE9D1/pEbiDh/8R2D
vQIhAPyXxbtdUqmEtxyZgIKTtXAlgxBdzSZD2OM7S+7Vm4O3AiEA6dEJ8fUGGzAy
wknR8vPbJ7Xj4ddEPCm+LYaZ1ppuj8MCICO+bnK3IA1xBGB9sFvJ/zDhfgyA2ZkC
nNx69aHUhG89AiAcDJ26zwAaht+v+xcapDJoQyPBVgezh/Zuk7Ukz4Xv7QIhAJ7x
oy7OrOGNJxSBiajeEaZ4PTo3FZPrR9xwSJUiKSh8
-----END RSA PRIVATE KEY-----

Taille RSA : 64
Résultat du décryptage : bonjour

-----BEGIN RSA PRIVATE KEY-----
Proc-Type: 4,ENCRYPTED
DEK-Info: AES-128-CBC,C878DD68B50DC68A80485C34F3E75337

ySEVSg3/pnnhbAlnbF41715pX6Z22Vtt7fb7Kq1C0QanUWaEtNiGwEkm0xGq1NAm
tsgC0WXOUE9wZkob0c9ntBtwZp0GPf6hAXhtMilM+BzgHqnM6eyZXZUK+K7+vCNE
W6QKiU6f9jKjzNOsS3pzZk+qrqi1jTbyOCw1EDVFJyRRBOTqgTmeIRgi1o55k6wI
fHOn4j1KXbW9kNYKh26MD0LbDdEGH1EPq1VkNJoLK7m2QCKDEkimZkOovahykvOT
VSv4floY17Q8LeMdzouRfv4SYN8uFkH8O7VPfTgwYKlELnD+PTIpOPupeqkyNVfD
Udjv8ztqhAh2Vpl1cWw2PNydXULZlzzVDRSW1bafEe4ARCFeTZ6kfO87JvKaaWsR
qPFm5l1xCVcOgmzwV2DGguTbKEh1xdF1t2+h/a6QXL4=
-----END RSA PRIVATE KEY-----

Cryptage caractères : ÔnMÔV Ôd#Ô#\+}Ôl#sÔÔÔÔDÔkÔTK"#\8^Ô|xSl#!aJÔeÔÔÔ#Ô2ÔÔ#y9Y#Ô(#
Résultat du décryptage avec passphrase : bonjour

-----BEGIN PUBLIC KEY-----
MFwwDQYJKoZIhvcNAQEBBQADSwAwSAJBAK+PzF+9aqmEhayPXryjUpv0ODo0P7UN
PF2Ee/a+eTMMjvCVIyhez0A1la8uwl2Q6hwZ2fRDfXkDiZZawyXEnw8CAwEAAQ==
-----END PUBLIC KEY-----

Cryptage caractères : ÔÔ#Ôv_ÔÔÔ!ÔRr8Ô9Ô9Ô2Ô          #M1ÔÔ-#JÔÔÔÔÔÔÔsÔ^Ô#r#APÔÔ1oNÔ9Tf#y\
ÔÔ
Résultat du décryptage avec passphrase : bonjour
```

*Ce programme est séparé en plusieurs parties afin de réaliser plusieurs types de tests suivant les scenarii à adopter pour avoir la meilleure stratégie de protection. La particularité et l'originalité de cette librairie **openssl** intégrée à **Rust** c'est que la génération des clés publiques et privées se fait directement à l'intérieur du code.*

Dans les autres API, nous devons générer nos clés à l'extérieur à partir du système d'exploitation et ensuite intégrer ces deux fichiers dans les ressources de l'exécutable. Ici, c'est beaucoup plus intéressant, nous n'avons pas besoin de fichiers annexes.

*Dans ce code, nous utilisons le système de chiffrage **RSA** qui est une partie intégrante de **openssl**. Pour cela, il existe la structure **Rsa** avec toutes les méthodes associées nécessaires pour chiffrer, déchiffrer, récupérer les clés privées et publiques associées.*

*La première partie de ce code nous montre la génération du chiffrage **RSA** à l'aide de la méthode **generate()**. Il est possible de choisir la robustesse du cryptage afin de choisir le degré de protection, **512, 1024, 2048, 4096, etc.** Ici, pour que le résultat ne prenne pas trop de place, j'ai pris le minimum. Le plus fréquent est d'utiliser un chiffrage **RSA** de **2048**.*



*Je rappelle que la génération du code **RSA** consiste finalement à générer la **clé privée**, celle qui sert au décryptage. En fait, elle fait partie de la variable issue de la structure **Rsa**.*

*Toujours dans cette première partie, nous pouvons dès lors crypter et décrypter nos messages sans se préoccuper des clés privées et publiques grâce aux méthodes respectives **public_encrypt()** et **private_decrypt()**. Les clés existent bel et bien, mais elle sont intégrées directement dans la structure **Rsa**.*

*Il est possible de retrouver ces clés à l'aide des méthodes respectives **private_key_to_pem()** et **public_key_to_pem()**. Nous pouvons dès lors crypter et décrypter les messages à partir de la récupération de ces clés, grâce à la génération d'un nouvel **Rsa** et des méthodes statiques **private_key_from_pem()** et **public_key_from_pem()**.*

*Comme nous l'avons abordé dans l'introduction de ce chapitre, nous pouvons aussi intégrer une « **passphrase** » à la génération de la clé privée. Nous utilisons pour cela les méthodes associées respectives **private_key_to_pem_passphrase()** et **private_key_from_pem_passphrase()**.*

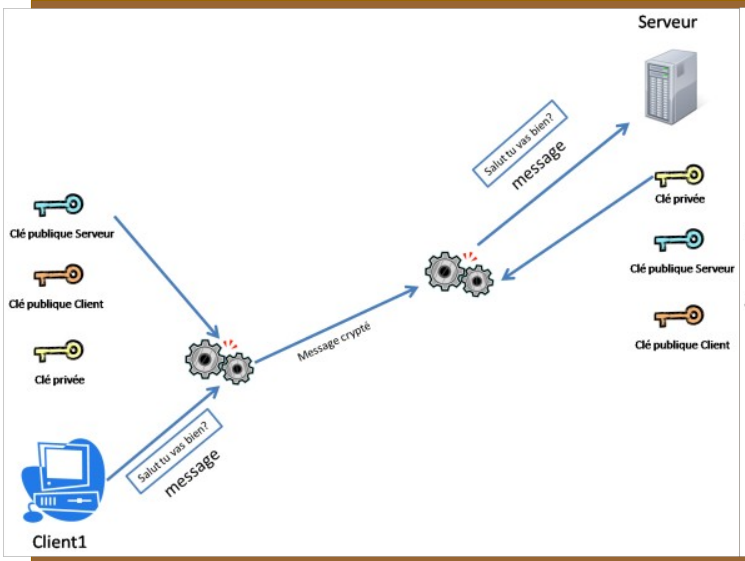
*Pour terminer sur ce sujet, nous pouvons remarquer que sur le fichier généré à partir de la **clé privée**, nous n'avons aucune spécification sur la durée de vie, ce qui sous-entend que le décryptage peut durer indéfiniment.*

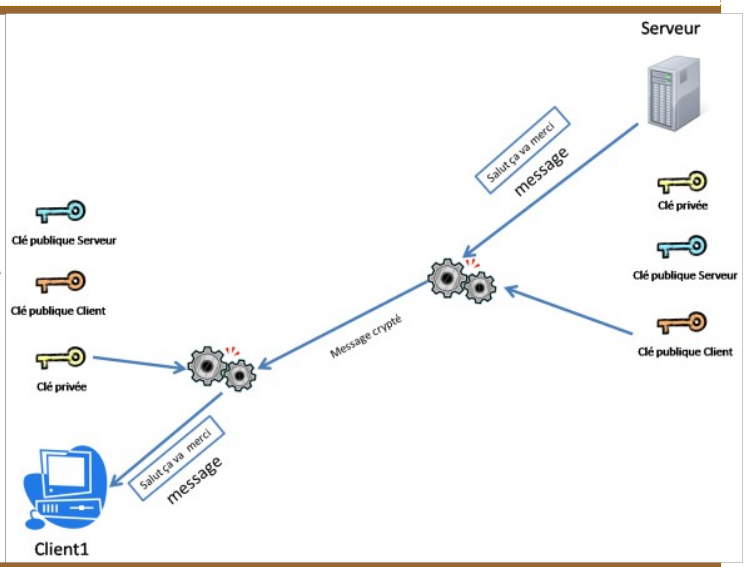
CHIFFRER LES COMMUNICATIONS ENTRE LE SERVICE ET LES DIFFÉRENTS CLIENTS

Maintenant que nous connaissons l'implémentation possible pour le cryptage de l'information, nous allons nous en servir et l'implémenter pour notre projet de financement. Par rapport aux différentes possibilités que nous venons de découvrir, il s'agit de proposer la meilleure stratégie afin de permettre une communication parfaitement protégée, avec des données chiffrées qui transitent, sans qu'il soit possible pour un pirate éventuel de retrouver l'information originale.

*La solution la plus simple consiste à respecter ce qui est prévu par la cryptographie asymétrique, en donnant la **clé publique** à l'autre interlocuteur afin qu'il puisse chiffrer son message à soumettre, tout en conservant la **clé privée** pour déchiffrer ce message envoyé. **Rappelez-vous qu'avec la clé publique, il est impossible de retrouver la clé privée**.*

*Donc, d'un point de vue pratique, chaque interlocuteur envoi sa propre **clé publique** à l'autre communiquant, ce qui fait que chaque interlocuteur possède à un instant donné deux clés publiques, celle qu'il envoie et celle qu'il reçoit de l'autre. Par principe, nous avons donc deux chiffrages différents. Par ailleurs, vu que les clés sont générées à chaque établissement de connexion, nous avons là aussi des cryptages différents pour chacun des clients connectés, ce qui offre un haut niveau de protection.*





chiffrement.rs

```
use openssl::rsa::{Padding, Rsa};
use std::net::TcpStream;
use std::io::{Read, Write};

pub struct Cryptage {
  pub pair: TcpStream,
  cle_privee: Vec<u8>,
  cle_publique_locale: Vec<u8>,
  cle_publique_pair: Vec<u8>
}

impl Cryptage {
  pub fn generer(pair: TcpStream, bits: u32) -> Cryptage {
    let rsa = Rsa::generate(bits).unwrap();
    Cryptage {
      pair,
      cle_privee: rsa.private_key_to_pem().unwrap(),
      cle_publique_locale: rsa.public_key_to_pem().unwrap(),
      cle_publique_pair: vec![]
    }
```

```
  }
  pub fn envoyer_cle_publique(&mut self) {
    self.pair.write(self.cle_publique_locale.as_slice());
  }
  pub fn recuperer_cle_publique(&mut self) {
    let recuperation = &mut  [0; 1024];
    let taille = self.pair.read(recuperation).unwrap();
    self.cle_publique_pair = recuperation[..taille].to_vec();
  }
  pub fn afficher_cle_publique_pair(&self) {
    println!("{}", String::from_utf8(self.cle_publique_pair.to_vec()).unwrap());
  }
  pub fn crypter(&mut self, message: String) {
    let rsa =  Rsa::public_key_from_pem(self.cle_publique_pair.as_slice()).unwrap();
    let mut cryptage = vec![0; rsa.size() as usize];
    let _ = rsa.public_encrypt(message.as_bytes(), &mut cryptage, Padding::PKCS1).unwrap();
    self.pair.write(cryptage.as_slice()).unwrap();
  }
  pub fn decrypter(&mut self) -> String {
    let trame= &mut [0; 1024];
    let nombre = self.pair.read(trame).unwrap();
    let rsa = Rsa::private_key_from_pem(self.cle_privee.as_slice()).unwrap();
    let mut recuperation = vec![0; rsa.size() as usize];
    let taille = rsa.private_decrypt(&trame[..nombre], &mut recuperation, Padding::PKCS1).unwrap();
    String::from_utf8_lossy(&recuperation[..taille]).to_string()
  }
}
```

*Ce fichier source factorise tout le fonctionnement du cryptage qui doit se trouver côté serveur et sur le client. Afin de respecter notre analyse de l'introduction, ce fichier source est composé d'une structure **Cryptage** avec deux **clés publiques** pour le cryptage, une envoyée et une reçue, une **clé privée** pour le décryptage. Vu que ce cryptage sert à la communication réseau, cette structure possède également un dernier élément qui représente l'autre ordinateur, soit le serveur, soit le client.*

*Suive un certain nombre de méthodes qui implémentent cette structure **Cryptage**. La première **generer()** est une méthode statique qui permet de créer la structure qui servira à tout le processus de cryptage durant toute la phase de communication entre les deux ordinateurs appairés. Nous nous servons de cette méthode pour créer la **clé privée** pour décrypter les messages et la **clé publique** qui sera envoyée à l'autre ordinateur connecté pour qu'il puisse lui-même crypter les différents messages. Cette générations de clés se fait suivant le niveau de robustesse choisi par l'utilisateur, avec des données sensibles ou pas.*

*Nous en profitons pour récupérer le point de connexion de type **TcpStream** qui sera utile pour pratiquement toutes les autres méthodes. Le cryptage ne sert effectivement que pour la communication réseau. La méthode suivante **envoyer_cle_publique()** se sert de ce point de connexion pour soumettre **la clé publique** pour l'autre ordinateur, sachant que la **clé privée** adaptée reste sur le poste en cours. Le cheminement **cryptage-décryptage** est donc assuré.*

*Il faut bien entendu que les deux postes respectent le même processus de ce principe là. La méthode **recuperer_cle_publique()** doit être utilisée par l'autre ordinateur en même temps que le premier soumet la méthode précédente. Nous comprenons très bien le rôle de cette méthode qui, je le rappelle, sert à crypter le message pour l'autre ordinateur.*

*Pour que ce système de communications de **clés publiques** soit parfait, il faut que l'autre ordinateur fasse le processus inverse, c'est-à-dire qu'il doit d'abord être en attente de la **clé publique** du premier avant que lui-même soumette sa propre **clé publique**. La méthode suivante **afficher_cle_publique_pair()** ne sert à rien, mais elle nous est utile en phase de test pour visualiser les **clés publiques** récupérées dans l'ordinateur distant et valider ainsi la transaction des clés.*

*Une fois que les deux **clés publiques** ont été échangées, il est maintenant possible de réaliser le cryptage des messages à envoyer à l'aide de la **clé publique** de l'interlocuteur grâce à la méthode **crypter()** et le décryptage des messages reçus avec la **clé privée** grâce cette fois-ci à la méthode **decrypter()**.*

Il est à noter que si vous désirez améliorer la robustesse de votre cryptage, il est possible de rajouter une « passphrase » sur la clé privée qui est répercutée bien évidemment sur les clés publiques envoyées. Par contre, pour que cela fonctionne parfaitement, vous devez avoir la même sur les deux postes en connexion, sinon c'est très difficile à implémenter.

Cargo.toml (pour les deux projets – service et application cliente)

```
[package]
name = "client-finance"
version = "0.1.0"
edition = "2018"

[dependencies]
serde = {version = "1.0.130", features = ["derive"]}
serde_json = "1.0.68"
openssl = { version ="0.10.36", features = ["vendored"] }
```

client.rs (côté serveur)

```
use std::str::*;
use std::io::{Write, Read};
use std::net::TcpStream;
use crate::json::{Requete, Reponse};
```

```rust
use crate::chiffrement::Cryptage;

pub fn traitement_client(mut client: TcpStream) {
    // Informations sur le client
    println!("Nouveau client [adresse : {}]", client.peer_addr().unwrap().ip());
    // Génération des clés, soumettre la clé publique au client et récupérer celle du client
    let mut cryptage = Cryptage::generer(client,777);
    cryptage.envoyer_cle_publique();
    cryptage.recuperer_cle_publique();
    cryptage.afficher_cle_publique_pair();
    // Communication avec le client (Attente de la requête)
    match serde_json::from_str::<Requete>(cryptage.decrypter().as_str()) {
        Ok(requete) => {
            let capital = requete.capital;
            let ans = requete.annees;
            let taux = requete.taux / 100.;

            // Traitements spécifiques, calcul du financement
            let mois = ans*12.;
            let mensualite = capital*taux/12./(1.-f32::powf(1.+taux/12., -ans*12.));
            let total = mois*mensualite;
            let interets = total-capital;
            let reponse = Reponse {mois, mensualite, total, interets};

            // Mise en oeuvre du protocole et envoi de la réponse
            let resultat = serde_json::to_string(&reponse).unwrap();
            cryptage.crypter(resultat);
        }
        Err(_) => { cryptage.pair.write(b"Votre format JSON n'est pas correct !"); }
    }
}
```

client.rs (côté serveur)

```rust
mod json;
mod chiffrement;

use std::io::{stdin, stdout, Write, Read};
use std::net::TcpStream;
use json::{Requete, Reponse};
use std::str::from_utf8;
use chiffrement::Cryptage;

fn main() {
    let adresse = "192.168.1.23";
    let port = 7878;
    match TcpStream::connect((adresse, port)) {
        Ok(mut service) => {
            println!("Projet de financement");
            println!("----------------------");
            let mut cryptage = Cryptage::generer(service,888);
            tranferts_cles_publiques(&mut cryptage);
            envoi_requete(&mut cryptage);
            recuperation_reponse(&mut cryptage);
        },
        Err(_) => println!("Non connecté avec le service")
    }
}

fn tranferts_cles_publiques(cryptage: &mut Cryptage) {
    cryptage.recuperer_cle_publique();
    cryptage.envoyer_cle_publique();
    cryptage.afficher_cle_publique_pair();
}

fn envoi_requete(cryptage: &mut Cryptage) {
    let capital = saisie("Capital");
    let annees = saisie("Nombre d'années");
    let taux = saisie("Taux");
    let requete = Requete {capital, annees, taux};
    let envoi = serde_json::to_string(&requete).unwrap();
    cryptage.crypter(envoi);
}

fn recuperation_reponse(cryptage: &mut Cryptage) {
    match serde_json::from_str::<Reponse>(cryptage.decrypter().as_str()) {
        Ok(resultat) => {
            println!("Nombre de mois : {}", resultat.mois);
            println!("Mensualités : {:.2} €", resultat.mensualite);
```

```
            println!("Coût total : {:.2} €", resultat.total);
            println!("Intérêts : {:.2} €", resultat.interets);
        },
        Err(retour) => println!("Format réponse non valide ! {:?}", retour)
    }
}

fn saisie(intitule: &str) -> f32 {
    let mut valeur = String::new();
    loop {
        print!("{} : ", intitule);
        stdout().flush().unwrap();
        stdin().read_line(&mut valeur).unwrap();
        match valeur.trim().parse() {
            Ok(x) =>  return x,
            Err(_) => { println!("Il faut une valeur numérique ! ") }
        }
    }
}
```

*De chaque côté, nous utilisons la structure **Cryptage**. Vous remarquez pour les deux sources entre les deux interlocuteurs que nous prenons un niveau de robustesse différent, respectivement **777** et **888**, ce qui est tout à fait possible.*

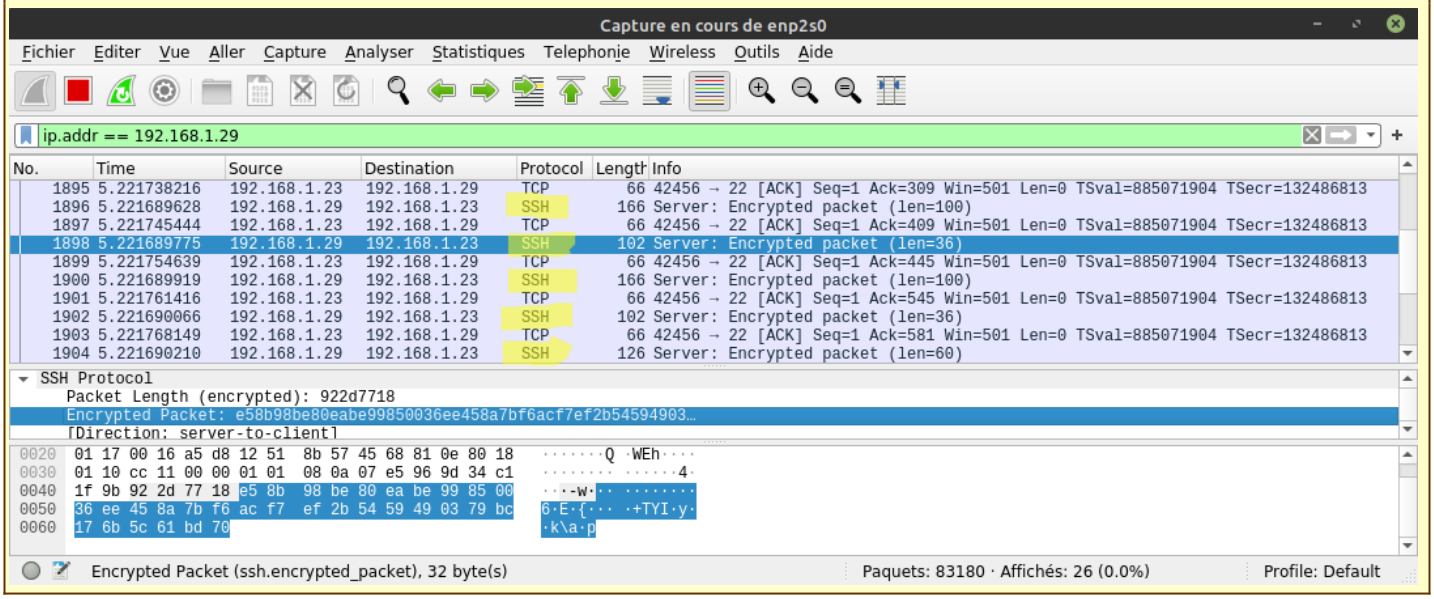
*Si les données sensibles sont plutôt envoyées d'un côté, vous pouvez appliquer un nombre de bits beaucoup plus conséquent. De l'autre côté, vous pouvez choisir une robustesse plus faible afin d'avoir un temps de réaction le plus rapide possible. À ce sujet, si vous prenez un nombre de **4096** pour les deux interlocuteurs, le temps de réaction peut durer plusieurs secondes. Jusqu'à **2048** bits, le temps de réponse est tout-à-fait acceptable.*

Résultat (service)

```
En attente d'un client...
Nouveau client [adresse : 192.168.1.23]
-----BEGIN PUBLIC KEY-----
MIGLMA0GCSqGSIb3DQEBAQUAA3oAMHcCcAC1Y5t6q1e4Ju/EzsLpfgSXUS+MOP1Q
45RWu1IRcrJZj0mwO96tuoUsyEPpc/Dwh0jCAYW7DZmKN1kV1rXv9IXouYA0rKIO
fP+7+KEnEZ8KRq11KH22tSiEUXThOLamH+9fDHabYTvqk9oMWjeSBJMCAwEAAQ==
-----END PUBLIC KEY-----
```

Résultat (client)

```
Projet de financement
---------------------
-----BEGIN PUBLIC KEY-----
MH0wDQYJKoZIhvcNAQEBBQADbAAwaQJiASQuaIxkWoIGNax8H8O6FTlrB9HYZEbM
cSO1gurQ8mneAzZ0/O+0kwDSeLxBkGKf05ldHBOseSY9j6FpbCi47FO0pW+wOTa9
oAq39AqUlUhDjecGas6NIc8uTJtxPIQGjvUCAwEAAQ==
-----END PUBLIC KEY-----

Capital : 5000
Nombre d'années : 5
Taux : 3
Nombre de mois : 60
Mensualités : 89.84 €
Coût total : 5390.49 €
Intérêts : 390.49 €
```



MISE EN PLACE DU SERVICE SUR UNE RASPBERRY - CROSS-COMPILATION

Les outils de **Rust** sont relativement puissants et complets. Ils permettent notamment de réaliser des programmes pour **plusieurs cibles différentes** tout en restant sur notre poste de développement. C'est ce que nous appelons faire de la **compilation croisée** sur le même poste de travail.

*Il est nécessaire au préalable d'installer les compilateurs pour chacune des cibles souhaitées et ceci toujours grâce à outil **rustup**. Dans la documentation de **Rust**, vous avez une très grande liste de cibles. Pour compiler un programme pour qu'il fonctionne sur une **Raspberry**, vous devez prendre le compilateur **armv7-unknown-linux-gnueabihf** :*

```
> rustup target add armv7-unknown-gnueabihf
```

*Pour que cela fonctionne parfaitement, il ne faut pas oublier d'installer également le **cross-compilateur C** pour votre cible choisie, ce qui se fait dans l'environnement **Linux** avec la commande suivante.*

```
> sudo apt install crossbuild-essential-armhf
```

*Les commandes précédentes, bien entendu, sont à faire qu'une seule fois sur votre ordinateur hôte. Par contre, par défaut, lorsque vous exécutez un projet, la cible reste votre ordinateur hôte. Si vous souhaitez changer de cible, vous devez rajouter dans votre projet un répertoire « **.cargo** » à l'intérieur duquel vous placez le fichier « **config.toml** » avec le script suivant :*

.cargo/config.toml

```
[build]
target = "armv7-unknown-linux-gnueabihf"

[target.armv7-unknown-linux-gnueabihf]
linker = "arm-linux-gnueabihf-gcc"
```

*Lorsque dès lors vous construisez votre projet, le système tient compte de ce fichier, et la compilation correspond par exemple à un exécutable de type **Raspberry**. Il suffit alors de le déployer avec les commandes classiques de copie sécurisée :*

```
> scp service-finance pi@192.168.1.29:/home/pi/rust
```

*Une fois que votre service est placé dans la **Raspberry**, vous devez ensuite faire en sorte que ce service soit opérationnel dès le démarrage du **nano-PC**. Pour cela, vous devez inscrire un **script** dans le répertoire prévu à cet effet « **/etc/init.d/{nom du service}** » à l'intérieur duquel vous placez votre ligne de commande correspondant au lancement du service souhaité.*

/etc/init.d/service-finance

```
#!/bin/sh
/home/pi/rust/service-finance
```

Une fois que ce script est enregistré, il faut qu'il soit considéré comme un exécutable. Tapez alors la commande suivante :

```
> sudo chmod +x /etc/init.d/service-finance
```

*Grâce à la ligne précédente, nous avons intégré dans la liste des services notre service de financement. Pour qu'il soit démarré automatiquement au lancement de la **Raspberry**, il faut qu'il soit alors activé par la commande suivante :*

```
> sudo update-rc.d service-finance default
```

*Si votre **Raspberry** est accessible par Internet derrière votre **Box**, pensez à changer le mot de passe « **raspberry** » par un autre plus sécurisant grâce à la commande « **passwd** ». Si après coup, vous souhaité arrêter ce service actif, tapez finalement la commande suivante :*

```
> sudo update-rc.d -f service-finance remove
```

TRANSFERT DE GROS BLOCS DE DONNÉES SUR LE RÉSEAU

Tous les projets que nous avons réalisés jusqu'à présent manipulaient des données de petites quantités. Nous allons donc vois dans ce dernier sujet comment transférer de plus grosses quantités de données sur le réseau, par exemple, des fichiers images, de la musique ou des vidéos.

*L'idée générale est de prévoir un tableau tampon de **1024** octets qui est souvent utilisé dans tous les « **buffers** » intermédiaires et de récupérer au fur et à mesure l'ensemble des octets venant du réseau dans une boucle jusqu'à ce que l'ensemble du fichier soit transmis.*

Généralement, dans ce genre de situation, vous stockez ces octets dans un fichier, que vous avez créé au lancement de la procédure, au fur et à mesure de leurs arrivés. Cela évite d'avoir une utilisation de la mémoire relativement conséquente.

*Ici, pour visualiser le fonctionnement de ce principe, nous allons prendre un « **buffer** » de quelques octets de capacité et nous soumettrons un simple texte bien plus grand pour bien comprendre le comportement du service de traitement.*

main.rs

```
use std::net::{TcpListener, TcpStream};
use std::io::Read;

fn main() {
  match TcpListener::bind("0.0.0.0:7777") {
    Ok(service) => {
```

```
    println!("En attente d'un client...");
    match service.accept() {
      Ok((client, _adresse)) => { traitement_client(client) },
      Err(_) => println!("Un client n'a pas réussi à se connecter")
    }
   },
   Err(_) => println!("Le service ne peut être activé !")
  }
}

pub fn traitement_client(mut client: TcpStream) {
  const TAILLE: usize = 7;
  let mut message = vec![];
  let octets = &mut [0; TAILLE];
  loop {
    let nombre = client.read(octets).unwrap();
    message.extend_from_slice(&octets[..nombre]);
    if nombre<TAILLE { break }
  }
  println!("Message complet = '{}'", String::from_utf8(message).unwrap().trim());
}
```

Résultat

En attente d'un client...

Message complet = 'Voici un message beaucoup plus grand que la capacité du tableau tampon'