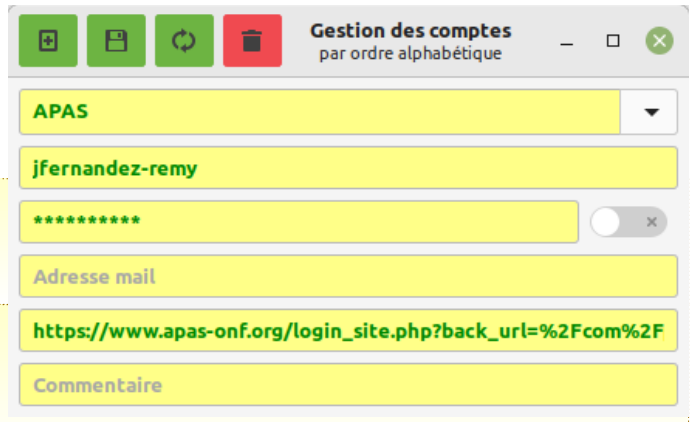
Après avoir acquis toutes les compétences nécessaires, je vous propose de mettre en œuvre une application concrète avec un système client-serveur qui permet de gérer les différents comptes importants avec des pseudos, des mots de passe associés, des liens Internet éventuels, des adresses mail, suivi de commentaires annexes.



*Vu qu'il s'agit d'un système **client-serveur**, il est nécessaire de **crypter** les échanges dans la communication réseau. La nouveauté ici, c'est que les informations transitées sont d'une capacité relativement importante.*

*Nous devrons donc proposer un **cryptage symétrique** pour tout l'ensemble du transfert. Par contre, nous rajouterons un **cryptage asymétrique** uniquement pour chiffrer les mots de passes. Pour cela, nous donnerons les **clés publiques** directement dans les **messages cryptés symétriquement**.*

*Le service sera placé dans une **Raspberry** derrière une **Box**. Du coup, nous pourrons exploiter pleinement cette gestion des comptes depuis **Internet**. Il faudra faire en sorte que ce service soit monté automatiquement à chaque redémarrage.*

Mise en place du service sur la 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 la base de données) à l'aide des commandes classiques de copie sécurisée :*

```
> scp service-comptes comptes.bd pi@192.168.1.25:/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 proposer un **fichier de configuration** dans le répertoire prévu à cet effet « **/etc/init/{nom du fichier.conf}** » à l'intérieur duquel vous placez les lignes suivantes :*

/etc/init/service-comptesconf

```
#!/bin/sh
description "gestion des comptes personnels"
start on startup
task
exec /home/pi/rust/service-comptes
```

*Pour que ce service soit parfaitement opérationnel, vous devez rajouter l'exécutable dans le système de lancement automatique local « **/etc/rc.local** » qui prendra en compte alors la configuration précédente,*

/etc/rc.local

```
#!/bin/sh -e
#
# rc.local
#
# This script is executed at the end of each multiuser runlevel.
# Make sure that the script will "exit 0" on success or any other
# value on error.
```

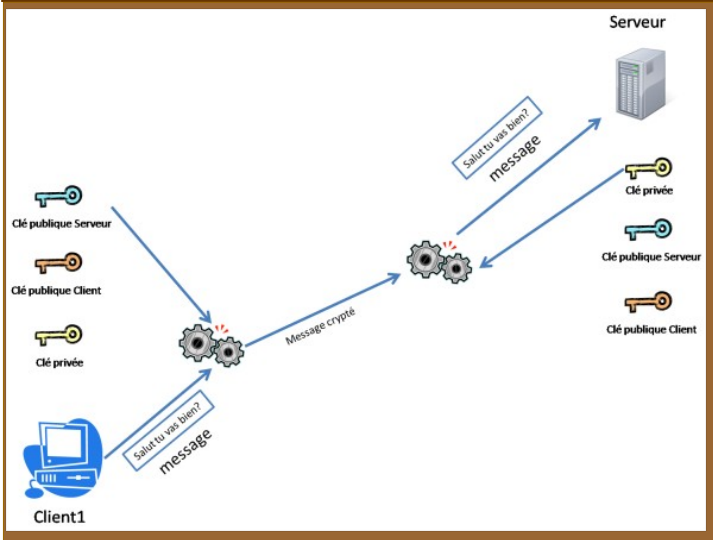
```
_IP=$(hostname -I) || true
if [ "$_IP" ]; then
  printf "My IP address is %s\n" "$_IP"
fi

/home/pi/rust/service-comptes
```

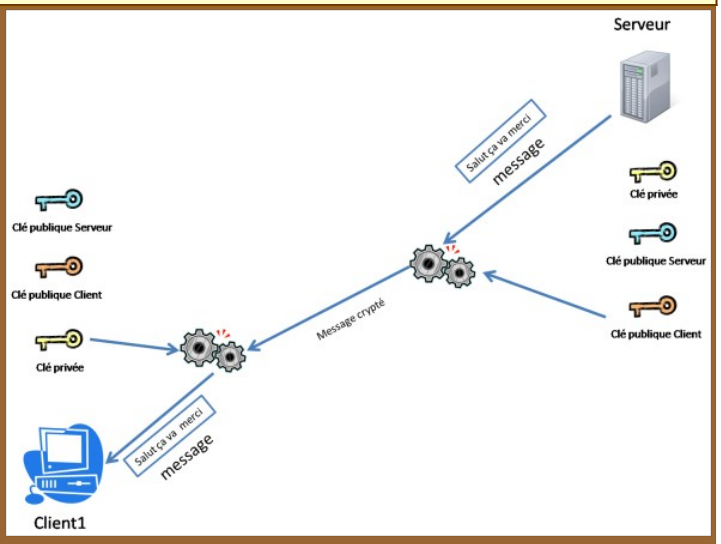
SERVICE DE GESTION DE COMPTES

Maintenant que nous venons de constituer toute l'architecture nécessaire au déploiement, nous allons nous intéresser au code du service. Nous devons prendre en compte la gestion du réseau avec la communication des différentes informations sous forme de trames **JSON**, le tout étant encapsulé par deux systèmes de cryptage. Par ailleurs, la gestion de la base de données se fait côté serveur bien entendu.

Cargo.toml

```
[package]
name = "service-comptes"
version = "0.1.0"
edition = "2021"

[dependencies]
rusqlite = {version="0.27.0", features=["bundled"] }
serde = {version="1.0.136", features=["derive"]}
serde_json = "1.0.79"
openssl = {version="0.10.38", features=["vendored"]}
```





chiffrement.rs

```
use openssl::rsa::{Padding, Rsa};
use std::net::TcpStream;
use std::io::{Read, Write};
use openssl::base64::{decode_block, encode_block};

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

impl Cryptage {
  pub fn generer(pair: TcpStream) -> Cryptage {
    let rsa = Rsa::generate(512).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()).unwrap();
  }
  pub fn crypter(&mut self, transfert: Vec<u8>) {
    self.pair.write(encode_block(transfert.as_slice()).as_bytes()).unwrap();
  }
```

```
  pub fn decrypter(&mut self) -> Vec<u8> {
    let trame= &mut [0; 2048];
    let nombre = self.pair.read(trame).unwrap();
    let trame = String::from_utf8(trame[..nombre].to_vec()).unwrap();
    decode_block(trame.as_str()).unwrap()
  }

  pub fn crypter_mdp(&mut self, mdp: String) -> Vec<u8> {
    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(mdp.as_bytes(), &mut cryptage, Padding::PKCS1).unwrap();
    cryptage
  }

  pub fn decrypter_mdp(&mut self, mdp: &[u8]) -> String {
    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(mdp, &mut recuperation, Padding::PKCS1).unwrap();
    String::from_utf8_lossy(&recuperation[..taille]).to_string()
  }
}
```

*Pour le chiffrement de la trame globale, nous passons par un cryptage symétrique **Base64**. La technique est vraiment très simple puisque deux fonctions permettent de chiffrer et de déchiffrer sans réglage préalable à l'aide des fonctions respectives **encode_block()** qui génère une chaîne cryptée et **decode_block()** qui retourne un vecteur d'octets.*

*Le **cryptage du mot de passe** se fait par un **cryptage asymétrique** en générant **les clés privées et publiques locales** et en récupérant la **clé publique de l'ordinateur en communication**. Puisqu'il s'agit d'un cryptage du seul mot de passe à l'intérieur d'un autre cryptage, la **clé privée** est générée en **512** octets.*

comptes.rs

```
use rusqlite::{params, Connection};
use serde::{Serialize, Deserialize};

#[derive(Debug, Clone, Serialize, Deserialize)]
pub struct Compte {
  pub action: String,
  pub comptes: Vec<String>,
  compte: String,
  pseudo: String,
  pub mdp: String,
  mail: String,
  internet: String,
  commentaire: String
}

#[derive(Debug, Clone, Serialize, Deserialize)]
pub struct Liste {
  pub comptes: Vec<String>
}

impl Liste {
  pub fn copier(liste: Vec<String>) -> Liste {
    Liste { comptes: liste }
  }
}

#[derive(Debug, Clone, Serialize, Deserialize)]
pub struct Transfert {
  pub compte: Compte,
  pub mdp_crypte: Vec<u8>,
  pub cle_publique: String
}

impl Compte {
  pub fn enregistrer(&mut self) {
    let bdd = Connection::open("/home/pi/rust/comptes.bd").unwrap();
    bdd.execute("INSERT INTO COMPTES (compte, pseudo, mdp, messagerie, internet, commentaire) \
                 VALUES(?1, ?2, ?3, ?4, ?5, ?6)",
            params![self.compte, self.pseudo, self.mdp, self.mail, self.internet, self.commentaire]).unwrap();
    self.liste();
  }

  pub fn modifier(&self) {
    let bdd = Connection::open("/home/pi/rust/comptes.bd").unwrap();
    bdd.execute("UPDATE COMPTES SET pseudo=?2, mdp=?3, messagerie=?4, internet=?5, commentaire=?6 WHERE
compte=?1",
        params![self.compte, self.pseudo, self.mdp, self.mail, self.internet, self.commentaire]).unwrap(); }
```

```
  pub fn supprimer(&mut self) {
    let bdd = Connection::open("/home/pi/rust/comptes.bd").unwrap();
    bdd.execute("DELETE FROM COMPTES WHERE compte=?1", params![self.compte]).unwrap();
    self.liste();
  }

  pub fn liste(&mut self) {
    let bdd = Connection::open("/home/pi/rust/comptes.bd").unwrap();
    self.comptes = Vec::new();
    let mut requete = bdd.prepare("SELECT compte FROM COMPTES ORDER BY compte").unwrap();
    let lignes = requete.query_map([], move |ligne| { Ok(ligne.get(0)?) }).unwrap();
    for ligne in lignes {
      if let Ok(compte) = ligne { self.comptes.push(compte); }
    }
  }

  pub fn choisir(&mut self) {
    let bdd = Connection::open("/home/pi/rust/comptes.bd").unwrap();
    let mut requete = bdd.prepare("SELECT * FROM COMPTES  WHERE compte=?1").unwrap();
    let resultat = requete.query_row(params![&self.compte], |ligne| {
      let pseudo: String = ligne.get(1).unwrap();
      let mdp: String = ligne.get(2).unwrap();
      let mail: String = ligne.get(3).unwrap();
      let internet: String = ligne.get(4).unwrap();
      let commentaire: String = ligne.get(5).unwrap();
      Ok((pseudo, mdp, mail, internet, commentaire))
    });
    if let Ok((pseudo, mdp, mail, internet, commentaire)) = resultat {
      self.pseudo = pseudo;
      self.mdp = mdp;
      self.mail = mail;
      self.internet = internet;
      self.commentaire = commentaire;
    }
  }
}
```

Le service doit gérer complètement la base de données en permettant d'enregistrer de nouveaux comptes, de les modifier ultérieurement, ou de les supprimer définitivement. Nous pouvons aussi connaître la liste de tous les comptes déjà enregistrés et de choisir un compte particulier afin qu'il puisse être visualisé dans l'application cliente.

*Vous disposez donc de méthodes pour réaliser cette gestion complète de comptes associées à une structure spécialisée **Compte**. Il est à noter que cette structure dispose d'un attribut supplémentaire qui permet de déterminer l'action à réaliser qui sera précisée par l'application cliente.*

*Nous disposons d'une structure supplémentaire **Liste** qui permet de retourner la liste des comptes enregistrés suivant la demande du client, notamment au début du lancement de l'application cliente. Cette structure permet d'éviter de retourner des champs vides associé à un compte particulier.*

*Toutes ces structures sont **sérialisables** afin de permettre la communication réseau au format **JSON**. Il reste une structure **Transfert**, qui comme son nom l'indique, permet d'effectuer le transfert complet dans les deux sens de communication entre le service et l'application cliente.*

*Cette structure factorise la structure **Compte** afin d'avoir toutes les informations nécessaires pour enregistrer un nouveau compte par exemple, mais dispose en plus d'un attribut qui représente la clé publique du client (qui est fourni à chaque requête) ainsi que le cryptage du mot de passe si ce dernier est présent.*

client.rs

```
use std::net::TcpStream;
use crate::chiffrement::Cryptage;
use crate::comptes::{Liste, Transfert};

pub fn traitement_client(client: TcpStream) {
  println!("Nouveau client [adresse : {}]", client.peer_addr().unwrap().ip());
  let mut cryptage = Cryptage::generer(client);
  cryptage.envoyer_cle_publique();
  match serde_json::from_slice::<Transfert>(cryptage.decrypter().as_slice()) {
     Ok(mut requete) => {
       println!("{:?}", requete);
       cryptage.cle_publique_pair = requete.cle_publique.as_bytes().to_vec();
       if !requete.mdp_crypte.is_empty() {
         requete.compte.mdp = cryptage.decrypter_mdp(requete.mdp_crypte.as_slice());
       }
       match requete.compte.action.as_str() {
          "enregistrer" => requete.compte.enregistrer(),
          "modifier" => requete.compte.modifier(),
          "supprimer" => requete.compte.supprimer(),
          "choisir" => requete.compte.choisir(),
```

```
            _ => requete.compte.liste(),
        }
        let resultat = match requete.compte.action.as_str() {
          "enregistrer" | "supprimer" | "liste" => serde_json::to_vec::<Liste>(&mut
Liste::copier(requete.compte.comptes)).unwrap(),
          _ => {
            requete.cle_publique = "".to_string();
            requete.mdp_crypte = cryptage.crypter_mdp(requete.compte.mdp);
            requete.compte.mdp = "".to_string();
            println!("Envoi : {:?}", &requete);
            serde_json::to_vec::<Transfert>(&requete).unwrap()
          }
        };
        cryptage.crypter(resultat);
      }
      Err(_) => { println!("Votre format JSON n'est pas correct !"); }
    }
}
```

*Ce code représente toute la communication avec l'application cliente. La remarque à faire, c'est que après chaque échange, la communication est définitivement interrompue. Ce qui veut dire que pour chaque **requête-réponse** nous aurons un cryptage différent, ce qui permet d'avoir un maximum de protection surtout pour ce genre de données.*

*Le protocole choisi pour chaque échange est le suivant : le service délivre tout de suite sa **clé publique** au client. Ensuite, le client soumet sa requête grâce à la trame de type **Transfert**. Si un mot de passe doit être transmis au serveur, il est alors chiffré grâce à la **clé publique fournie par le service**. Dans le même temps, dans la requête, nous trouvons systématiquement **la clé publique du serveur** pour que le service puisse l'utilisée au cas où pour crypter également le mot de passe s'il est présent.*

*Je le rappelle, toutes ces communications dans un sens ou dans l'autre sont systématiquement chiffrées par une **clé symétrique** au travers du cryptage de type **Base64**.*

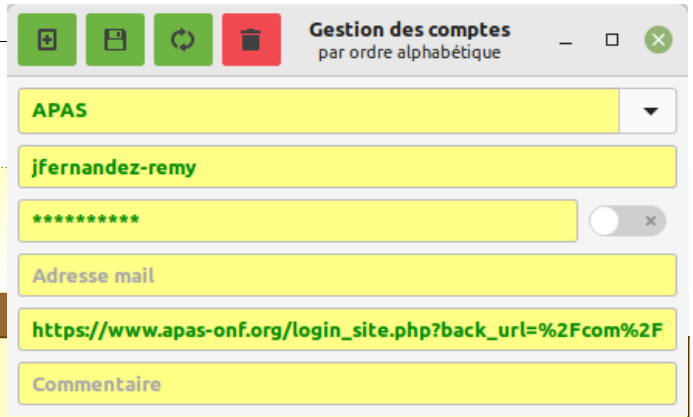
client.rs

```
mod client;
mod chiffrement;
mod comptes;

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

fn main() {
   let adresse = "0.0.0.0";  // prend en compte l'@ IP du poste
   let port = 7777;
   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 !")
   }
}
```

RÉALISATION DE LA PARTIE CLIENTE

L'application cliente concerne essentiellement l'IHM, mais nous avons également une bonne partie commune au service puisque nous devons également établir la communication avec des trames JSON cryptées.

Vu que nous devons respecter le même protocole, sinon aucune communication n'est possible, nous retrouverons les mêmes types de fichiers sources pour la partie chiffrement ainsi que pour l'organisation des structures en conformité au protocole.



Cargo.toml

```
[package]
name = "client-comptes"
version = "0.1.0"
edition = "2021"
```

```
[dependencies]
gtk = "0.15.4"
serde = {version="1.0.136", features=["derive"]}
serde_json = "1.0.79"
openssl = {version="0.10.38", features=["vendored"]}
```

*Nous retrouvons pratiquement le même fichier de configuration de projet que le service, puisque beaucoup de modules similaires doivent être installés. La seule différence, ce qui paraît logique, concerne les modules **gtk** (**IHM**) et **rusqlite** (**BDD**).*

chiffrement.rs

```
use openssl::rsa::{Padding, Rsa};
use std::net::TcpStream;
use std::io::{Read, Write};
use openssl::base64::{decode_block, encode_block};

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

impl Cryptage {
  pub fn generer(pair: TcpStream) -> Cryptage {
    let rsa = Rsa::generate(512).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 recuperer_cle_publique(&mut self) {
    let recuperation = &mut  [0; 512];
    let taille = self.pair.read(recuperation).unwrap();
    self.cle_publique_pair = recuperation[..taille].to_vec();
  }

  pub fn crypter(&mut self, transfert: Vec<u8>) {
    self.pair.write(encode_block(transfert.as_slice()).as_bytes()).unwrap();
  }

  pub fn decrypter(&mut self) -> Vec<u8> {
    let trame= &mut [0; 2048];
    let nombre = self.pair.read(trame).unwrap();
    let trame = String::from_utf8(trame[..nombre].to_vec()).unwrap();
    decode_block(trame.as_str()).unwrap()
  }

  pub fn crypter_mdp(&mut self, mdp: String) -> Vec<u8> {
    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(mdp.as_bytes(), &mut cryptage, Padding::PKCS1).unwrap();
    cryptage
  }

  pub fn decrypter_mdp(&mut self, mdp: &[u8]) -> String {
    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(mdp, &mut recuperation, Padding::PKCS1).unwrap();
    String::from_utf8_lossy(&recuperation[..taille]).to_string()
  }
}
```

*Là aussi, ce code est très similaire à celui du même nom associé au service. La différence toutefois, c'est que côté client nous disposons de la méthode **recuperer_cle_publique()** alors que côté serveur, nous avions la méthode **envoyer_cle_publique()**.*

communication.rs

```
use serde::{Serialize, Deserialize};
use std::net::TcpStream;
use crate::chiffrement::Cryptage;

#[derive(Debug, Clone, Serialize, Deserialize)]
pub struct Compte {
  pub action: String,
  pub comptes: Vec<String>,
```

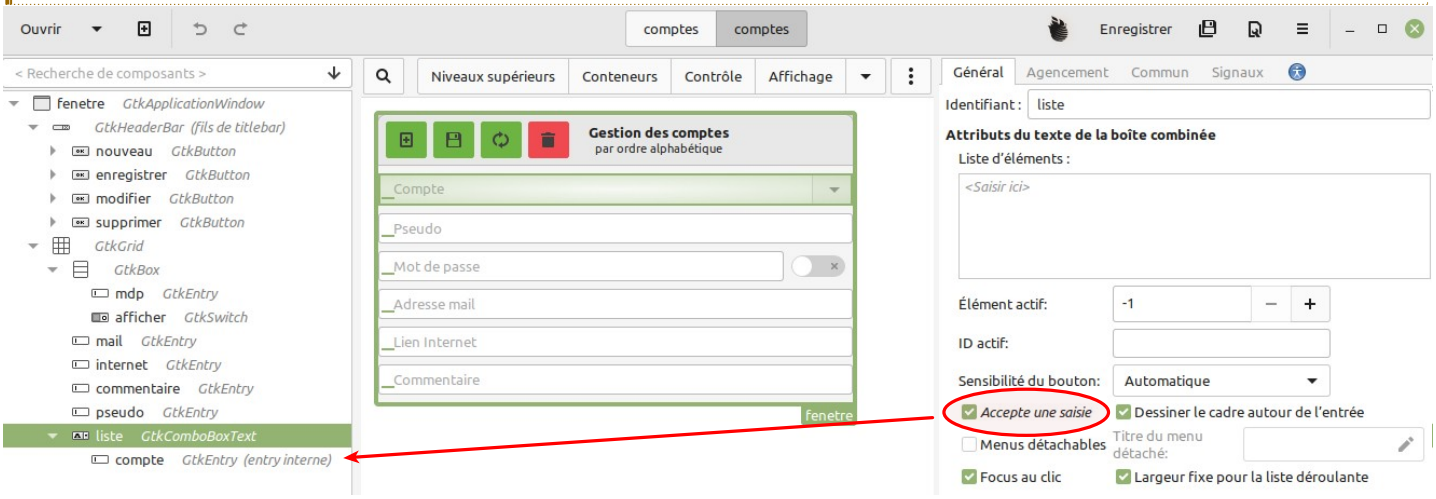
```
  pub compte: String,
  pub pseudo: String,
  pub mdp: String,
  pub mail: String,
  pub internet: String,
  pub commentaire: String
}

#[derive(Debug, Clone, Serialize, Deserialize)]
pub struct Transfert {
  compte: Compte,
  mdp_crypte: Vec<u8>,
  cle_publique: String
}

impl Transfert {
  fn new(compte: &Compte, cle: Vec<u8>) -> Transfert {
    Transfert {
      compte: compte.clone(),
      cle_publique: String::from_utf8(cle).unwrap(),
      mdp_crypte: vec![]
    }
  }
}

#[derive(Debug, Clone, Serialize, Deserialize)]
pub struct Liste {
  pub comptes: Vec<String>
}

impl Compte {
  pub fn new() -> Compte {
    Compte {
      action: String::new(),
      comptes: Vec::new(),
      compte: String::new(),
      pseudo: String::new(),
      mdp: String::new(),
      mail: String::new(),
      internet: String::new(),
      commentaire: String::new()
    }
  }

  pub fn soumettre(&mut self) {
    let adresse = "192.168.1.25"; // "remy-manu.no-ip.biz";
    let port = 7777;
    match TcpStream::connect((adresse, port)) {
      Ok(service) => {
        let mut cryptage = Cryptage::generer(service);
        cryptage.recuperer_cle_publique();
        let mut transfert = Transfert::new(&self, cryptage.cle_publique_locale.clone());
        if !self.mdp.is_empty() {
          transfert.mdp_crypte = cryptage.crypter_mdp(self.mdp.clone());
          transfert.compte.mdp = "".to_string();
        }
        cryptage.crypter(serde_json::to_vec::<Transfert>(&transfert).unwrap());
        match self.action.as_str() {
          "enregistrer" | "supprimer" | "liste" => {
            let resultat = serde_json::from_slice::<Liste>(cryptage.decrypter().as_slice()).unwrap();
            self.comptes = resultat.comptes;
          },
          _ => {
            let transfert = serde_json::from_slice::<Transfert>(cryptage.decrypter().as_slice()).unwrap();
            *self = transfert.compte;
            self.mdp = cryptage.decrypter_mdp(transfert.mdp_crypte.as_slice());
          },
        }
      },
      Err(_) => println!("Non connecté au service")
    }
  }
}
```

*Ce code est très important et représente la communication réseau. Nous retrouvons les mêmes structures que pour le **service** bien entendu afin que le **protocole** soit bien compris **pour les deux parties**. Le **client** tente de se connecter au **service** et dès que la connexion est établie, il reçoit la **clé publique du serveur** qui va servir au cryptage du mot de passe s'il est présent.*

*De son côté, le client fournit systématiquement sa propre **clé publique** intégrée dans la trame associée à la structure **Transfert**. Si un mot de passe est crypté, il est également intégré dans cette même trame **Transfert**.*

*Suivant le type de requête, le client reçoit une trame au format **Liste** ou une trame au format **Transfert**. Si le client reçoit une trame de type **Transfert**, c'est que nous devons afficher la totalité du compte sélectionné, ce qui sous entend que nous avons un **mot de passe** que nous devons donc **décrypter systématiquement**.*

*La suite des sources concerne plus précisément l'interface graphique de l'application. Encore une fois, la conception de **l'IHM** se fait au travers de l'outil **GLADE**. J'utilise ici une boîte **Combo** qui permet de faire une saisie dans sa zone principale ce qui correspond finalement à une zone d'édition classique.*



comptes.rs

```
use gtk::prelude::*;
use gtk::*;
use crate::communication::Compte;

#[derive(Clone)]
pub struct IHM {
  nouveau: Button,
  enregistrer: Button,
  modifier: Button,
  supprimer: Button,
  liste: ComboBoxText,
  compte: Entry,
  pseudo: Entry,
  mdp: Entry,
  afficher: Switch,
  mail: Entry,
  internet: Entry,
  pub commentaire: Entry
}

impl IHM {
  pub fn placer(vue: Builder) -> Self {
    let ihm = IHM {
      nouveau: vue.object("nouveau").unwrap(),
      enregistrer: vue.object("enregistrer").unwrap(),
      modifier: vue.object("modifier").unwrap(),
      supprimer: vue.object("supprimer").unwrap(),
      liste: vue.object("liste").unwrap(),
      compte: vue.object("compte").unwrap(),
      pseudo: vue.object("pseudo").unwrap(),
      mdp: vue.object("mdp").unwrap(),
      afficher: vue.object("afficher").unwrap(),
      mail: vue.object("mail").unwrap(),
      internet: vue.object("internet").unwrap(),
      commentaire: vue.object("commentaire").unwrap()
    };
    let mdp = ihm.mdp.clone();
    ihm.afficher.connect_changed_active(move |switch| {
       mdp.set_visibility(switch.is_active());
    });
    ihm
  }

  pub fn effacer(&self) {
    let ihm = self.clone();
```

```
  self.nouveau.connect_clicked(move |_| {
    ihm.compte.set_text("");
    ihm.pseudo.set_text("");
    ihm.mdp.set_text("");
    ihm.mail.set_text("");
    ihm.internet.set_text("");
    ihm.commentaire.set_text("");
  });
}

pub fn enregistrement(&self) {
  let ihm = self.clone();
  self.enregistrer.connect_clicked(move |_| {
    let mut json = Compte::new();
    json.compte = ihm.compte.text().to_string();
    json.pseudo = ihm.pseudo.text().to_string();
    json.mdp = ihm.mdp.text().to_string();
    json.mail = ihm.mail.text().to_string();
    json.internet = ihm.internet.text().to_string();
    json.commentaire = ihm.commentaire.text().to_string();
    json.action = "enregistrer".to_string();
    json.soumettre();
    ihm.liste(&mut json);
  });
}

pub fn modification(&self) {
  let ihm = self.clone();
  self.modifier.connect_clicked(move |_| {
    let mut json = Compte::new();
    json.compte = ihm.compte.text().to_string();
    json.pseudo = ihm.pseudo.text().to_string();
    json.mdp = ihm.mdp.text().to_string();
    json.mail = ihm.mail.text().to_string();
    json.internet = ihm.internet.text().to_string();
    json.commentaire = ihm.commentaire.text().to_string();
    json.action = "modifier".to_string();
    json.soumettre();
  });
}

pub fn suppression(&self) {
  let ihm = self.clone();
  self.supprimer.connect_clicked(move |_| {
    let mut json = Compte::new();
    json.compte = ihm.compte.text().to_string();
    json.compte = ihm.compte.text().to_string();
    json.action = "supprimer".to_string();
    json.soumettre();
    ihm.liste(&mut json);
  });
}

pub fn selection(&self) {
  let ihm = self.clone();
  self.compte.connect_changed(move |_| {
    let mut json = Compte::new();
    json.compte = ihm.compte.text().to_string();
    json.action = "choisir".to_string();
    json.soumettre();
    ihm.pseudo.set_text(json.pseudo.as_str());
    ihm.mdp.set_text(json.mdp.as_str());
    ihm.mail.set_text(json.mail.as_str());
    ihm.internet.set_text(json.internet.as_str());
    ihm.commentaire.set_text(json.commentaire.as_str());
  });
}

pub fn liste(&self, compte: &mut Compte) {
  let ihm = self.clone();
  ihm.liste.remove_all();
  if compte.comptes.is_empty() {
    let mut json = Compte::new();
    json.action = "liste".to_string();
    json.soumettre();
    for compte in &json.comptes {
      ihm.liste.append_text(compte.as_str());
    }
```

```
    }
    else {
      for compte in &compte.comptes {
        ihm.liste.append_text(compte.as_str());
      }
    }
    ihm.liste.set_active(Some(0));
  }
}
```

*Ce source est relativement volumineux puisqu'il prend en compte toute la gestion événementielle de l'application. Pour des projets un peu conséquent, comme c'est le cas ici, il est préférable d'interfacer le code de **l'IHM** à partir d'une structure **IHM** qui représente parfaitement tous les éléments la constituant.*

*Nous retrouvons ensuite l'ensemble des méthodes correspondant aux événements souhaités. Pour pratiquement chacune de ces méthodes, nous créons un objet représentant la structure **Compte** que nous complétons suivant les événements de **l'IHM**.*

*Nous spécifions ensuite l'attribut **action** avec l'une des requête souhaitée (**« enregistrer »**, **« modifier »**, **« supprimer »**, **« choisir »** et **« liste »**). Une fois que que le compte est parfaitement complété, nous le soumettons au service.*

main.rs

```
mod comptes;
mod communication;
mod chiffrement;

use gtk::prelude::*;
use gtk::*;
use comptes::IHM;
use crate::communication::Compte;

fn main() {
    gtk::init().unwrap();
    feuille_de_style();
    let vue = Builder::from_string(include_str!("../comptes.glade"));
    let fenetre: Window = vue.object("fenetre").unwrap();
    let ihm = IHM::placer(vue);
    ihm.selection();
    ihm.effacer();
    ihm.enregistrement();
    ihm.modification();
    ihm.suppression();
    ihm.liste(&mut Compte::new());
    fenetre.connect_destroy(|_| { gtk::main_quit() });
    fenetre.show();
    gtk::main();
}

fn feuille_de_style() {
    let css = CssProvider::new();
    css.load_from_data(include_bytes!("../comptes.css")).unwrap();
    StyleContext::add_provider_for_screen(
        &gdk::Screen::default().unwrap(),
        &css,
        gtk::STYLE_PROVIDER_PRIORITY_APPLICATION
    );
}
```

*Nous disposons dans ce code de la fonction principale deux macros magiques **include_bytes!()** et **include_str!()** qui permettent d'intégrer directement dans l'exécutable des ressources externes comme c'est le cas ici avec **« comptes.glade »** qui représente le document associé à la structure de **l'IHM**, ainsi que la feuille de style **« comptes.css »**.*

Cela veut dire que nous n'aurons plus à déployer ces deux fichiers ressources lorsque nous déploierons l'exécutable sur d'autres postes dans le réseau local ou par Interner. Nous disposons de deux variantes d'inclusion, soit pour la récupération des octets brutes, soit directement sous forme de texte lorsque la ressource est elle-même sous forme de fichier texte. Cela dépend des structures qui utilisent ces données.