Je propose un nouveau projet qui va permettre de stocker des informations importantes comme par exemple les commandes utiles pour bien utiliser Linux. Côté interface, l'application cliente est plus modeste que la précédente puisque seuls trois éléments sont à prendre en compte, la rubrique principal, le sujet associé ainsi que la description complète souhaité.

*Il s'agit de nouveau d'un système **client-serveur**. Les informations qui transites ne sont pas des données sensibles. Nous effectuerons un simple cryptage symétrique pour éviter tout de même que les données soient visibles directement.*

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

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-ressources.conf

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

*Pour que tous ces services soient parfaitement opérationnels, vous devez rajouter l'exécutable dans le système de lancement automatique local « **/etc/rc.local** » qui prendra en compte alors les configurations des différents services comme le précédent,*

/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 &

/home/pi/rust/service-ressources &
```

SERVICE DE GESTION DE RESSOURCES

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 le système de cryptage **symétrique** en **Base64**. Par ailleurs, la gestion de la base de données se fait côté serveur bien entendu. Nous n'avons plus besoin du fichier source **« chiffrement.rs »**.

Cargo.toml

```
[package]
name = "service-ressources"
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"]}
```

ressources.rs

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

#[derive(Debug, Clone, Serialize, Deserialize)]
pub struct Ressource {
  pub action: String,
  pub rubrique: String,
  sujet: String,
  description: String,
  liste: Vec<String>
}

impl Ressource {
  pub fn enregistrer(&mut self) {
    let bdd = Connection::open("/home/pi/rust/important.bd").unwrap();
    bdd.execute("INSERT INTO FICHE (rubrique, sujet, description) VALUES(?1, ?2, ?3)",
            params![self.rubrique, self.sujet, self.description]).unwrap();
    self.liste_rubriques()
  }

  pub fn modifier(&self) {
    let bdd = Connection::open("/home/pi/rust/important.bd").unwrap();
    bdd.execute("UPDATE FICHE SET description=?3 WHERE rubrique=?1 AND sujet=?2",
        params![self.rubrique, self.sujet, self.description]).unwrap();
  }

  pub fn supprimer(&mut self) {
    let bdd = Connection::open("/home/pi/rust/important.bd").unwrap();
    bdd.execute("DELETE FROM FICHE WHERE rubrique=?1 AND sujet=?2", params![self.rubrique,
self.sujet]).unwrap();
    self.liste_rubriques()
  }

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

  pub fn liste_sujets(&mut self) {
    let bdd = Connection::open("/home/pi/rust/important.bd").unwrap();
```

```
    let mut requete = bdd.prepare("SELECT DISTINCT sujet FROM FICHE WHERE rubrique=?1").unwrap();
    let lignes = requete.query_map([&self.rubrique], move |ligne| { Ok(ligne.get(0)?) }).unwrap();
    for ligne in lignes {
      if let Ok(sujet) = ligne { self.liste.push(sujet); }
    }
  }

  pub fn description(&mut self) {
    let bdd = Connection::open("/home/pi/rust/important.bd").unwrap();
    let mut requete = bdd.prepare("SELECT * FROM FICHE  WHERE rubrique=?1 AND sujet=?2").unwrap();
    let resultat = requete.query_row(params![&self.rubrique, &self.sujet], |ligne| {
      let rubrique: String = ligne.get(1).unwrap();
      let sujet: String = ligne.get(2).unwrap();
      let description: String = ligne.get(3).unwrap();
      Ok((rubrique, sujet, description))
    });
    if let Ok((rubrique, sujet, description)) = resultat {
      self.rubrique = rubrique;
      self.sujet = sujet;
      self.description = description;
    }
  }
}
```

Le service doit gérer complètement la base de données en permettant d'enregistrer de nouvelles ressources, de les modifier ultérieurement, ou de les supprimer définitivement. Nous pouvons aussi connaître la liste de toutes les ressources déjà enregistrées et de choisir une ressource particulière afin qu'elle puisse être visualisée dans l'application cliente.

*Vous disposez donc de méthodes pour réaliser cette gestion complète des ressources associées à une structure spécialisée **Ressource**. Cette structure dispose d'un attribut supplémentaire qui permet de déterminer l'action à réaliser qui sera précisée par l'application cliente. Cette structure est **sérialisable** afin de permettre la communication réseau au format **JSON**.*

client.rs

```
use std::io::{Read, Write};
use std::net::TcpStream;
use crate::ressources::Ressource;
use openssl::base64::{decode_block, encode_block};

pub fn traitement_client(mut client: TcpStream) {
  println!("Nouveau client [adresse : {}]", client.peer_addr().unwrap().ip());
  let trame = &mut [0; 2048];
  let nombre = client.read(trame).unwrap();
  print!("nombre={} : ", nombre);
  let trame = String::from_utf8(trame[..nombre].to_vec()).unwrap();
  match serde_json::from_slice::<Ressource>(decode_block(trame.as_str()).unwrap().as_slice()) {
     Ok(mut requete) => {
       println!("{:?}", requete);
       match requete.action.as_str() {
          "enregistrer" => requete.enregistrer(),
          "modifier" => requete.modifier(),
          "supprimer" => requete.supprimer(),
          "description" => requete.description(),
          "sujets" => requete.liste_sujets(),
          _ => requete.liste_rubriques(),
       }
       println!("Envoi : {:?}", &requete);
       let resultat = serde_json::to_vec::<Ressource>(&requete).unwrap();
       client.write(encode_block(resultat.as_slice()).as_bytes()).unwrap();
     }
     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 symétrique différent, ce qui permet d'avoir un maximum de protection, même si les données échangées ne sont pas sensibles.*

*Le protocole choisi pour chaque échange est plus simple que l'application précédente : le client soumet sa requête grâce à la trame de type **Resource**. La réponse se fait avec la même structure. 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**.*

main.rs

```
mod client;
mod ressources;

use std::net::{TcpListener};
use std::thread::spawn;
```

```rust
use client::traitement_client;

fn main() {
   let adresse = "0.0.0.0";  // prend en compte l'@ IP du poste
   let port = 8888;
   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 de la structure en conformité du 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).*

Ressources importantes
Quelques commandes utiles

Android

SDK sous Linux

```
sudo dpkg --add-architecture i386
sudo apt-get update
sudo apt-get install libc6:i386 libncurses5:i386
libstdc++6:i386
sudo ./adb

// revoir les architectures autre que x64
dpkg --print-foreign-architectures

// Android sdk offline
$ https://androidsdkoffline.blogspot.fr/p/android-sdk-
tools.html

// NBAndroid netbeans 8.1
$ http://nbandroid.org/release81/updates/updates.xml

// SDK et AVD Manager
$ ./sdkmanager --no_https --list
$ ./sdkmanager --no_https  "platform-tools"
"platforms;android-26" "build-tools;28.0.0"
$ ./sdkmanager --no_https --update

// Particularités Ubuntu 18.04
https://linoxide.com/linux-how-to/install-android-sdk-
manager-ubuntu/
```

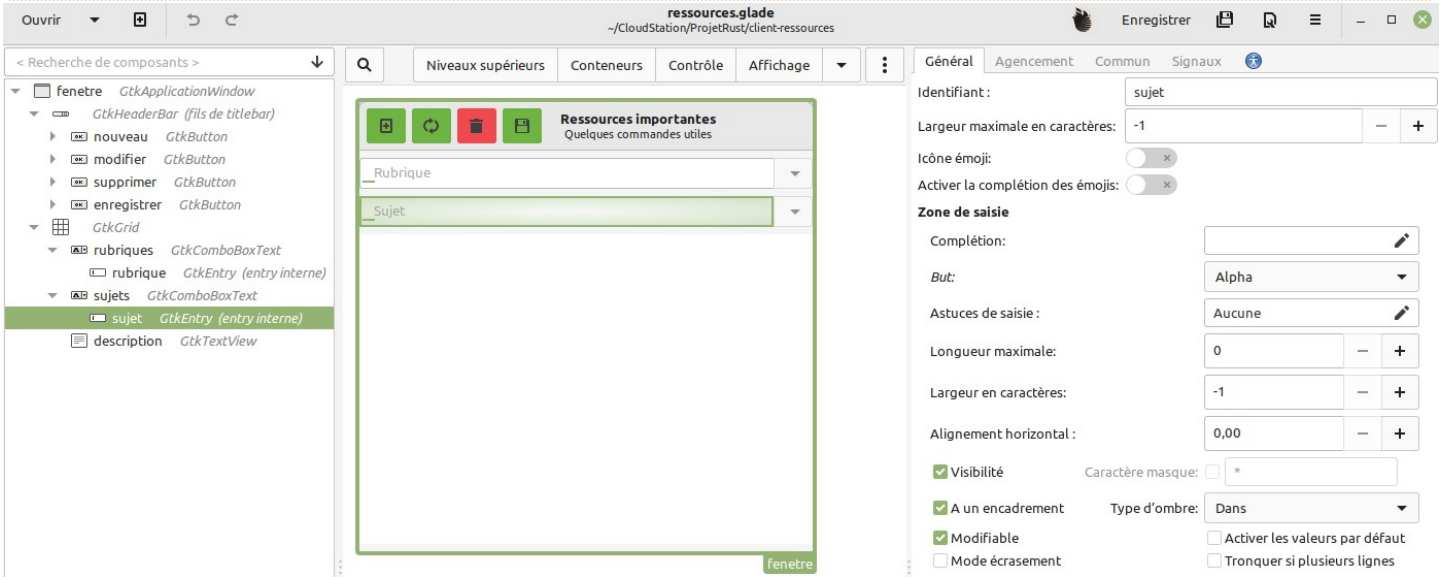
communication.rs

```rust
use std::io::{Read, Write};
use serde::{Serialize, Deserialize};
use std::net::TcpStream;
use openssl::base64::{encode_block, decode_block};

#[derive(Debug, Clone, Serialize, Deserialize)]
pub struct Ressource {
  pub action: String,
  pub rubrique: String,
  pub sujet: String,
  pub description: String,
  pub liste: Vec<String>
}

impl Ressource {
  pub fn new() -> Ressource {
    Ressource {
      action: String::new(),
      rubrique: String::new(),
      sujet: String::new(),
      description: String::new(),
      liste: Vec::new()
    }
  }

  pub fn soumettre(&mut self) {
    let adresse = "remy-manu.no-ip.biz";
    let port = 8888;
```

```
    match TcpStream::connect((adresse, port)) {
      Ok(mut service) => {
        let requete = serde_json::to_vec::<Ressource>(self).unwrap();
        service.write(encode_block(requete.as_slice()).as_bytes()).unwrap();
        let trame = &mut [0; 4096];
        let nombre = service.read(trame).unwrap();
        println!("nombre={}", nombre);
        let trame = String::from_utf8(trame[..nombre].to_vec()).unwrap();
        let ressource =
serde_json::from_slice::<Ressource>(decode_block(trame.as_str()).unwrap().as_slice()).unwrap();
        *self = ressource;
      },
      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**.*

*Le client soumet une requête en utilisant la structure **Ressource** et reçoit une trame avec le même format. 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 deux boîtes **Combo** qui permet de faire une saisie dans sa zone principale ce qui correspond finalement à une zone d'édition classique. La partie description est représenté par un composant de type **GtkTextView**.*



ressources.rs

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

#[derive(Clone)]
pub struct IHM {
  nouveau: Button,
  enregistrer: Button,
  modifier: Button,
  supprimer: Button,
  rubriques: ComboBoxText,
  rubrique: Entry,
  sujets: ComboBoxText,
  sujet: Entry,
  description: TextView
}

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(),
      rubriques: vue.object("rubriques").unwrap(),
      rubrique: vue.object("rubrique").unwrap(),
      sujets: vue.object("sujets").unwrap(),
```

```
    sujet: vue.object("sujet").unwrap(),
    description: vue.object("description").unwrap()
  };
  ihm
}

pub fn effacer(&self) {
  let ihm = self.clone();
  self.nouveau.connect_clicked(move |_| {
    ihm.rubrique.set_text("");
    ihm.sujet.set_text("");
    ihm.description.buffer().unwrap().set_text("");
  });
}

pub fn enregistrement(&self) {
  let ihm = self.clone();
  self.enregistrer.connect_clicked(move |_| {
    let mut json = Ressource::new();
    json.rubrique = ihm.rubrique.text().to_string();
    json.sujet = ihm.sujet.text().to_string();
    let texte = ihm.description.buffer().unwrap();
    let (debut, fin) = texte.bounds();
    json.description = texte.text(&debut, &fin, true).unwrap().to_string();
    json.action = "enregistrer".to_string();
    json.soumettre();
    ihm.liste_rubriques(&mut json);
  });
}

pub fn modification(&self) {
  let ihm = self.clone();
  self.modifier.connect_clicked(move |_| {
    let mut json = Ressource::new();
    json.rubrique = ihm.rubrique.text().to_string();
    json.sujet = ihm.sujet.text().to_string();
    let texte = ihm.description.buffer().unwrap();
    let (debut, fin) = texte.bounds();
    json.description = texte.text(&debut, &fin, true).unwrap().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 = Ressource::new();
    json.rubrique = ihm.rubrique.text().to_string();
    json.sujet = ihm.sujet.text().to_string();
    json.action = "supprimer".to_string();
    json.soumettre();
    ihm.liste_rubriques(&mut json);
  });
}

pub fn selection_rubrique(&self) {
  let ihm = self.clone();
  self.rubrique.connect_changed(move |_| {
    let mut json = Ressource::new();
    json.rubrique = ihm.rubrique.text().to_string();
    json.action = "sujets".to_string();
    json.soumettre();
    ihm.sujets.remove_all();
    for sujet in json.liste {
      ihm.sujets.append_text(sujet.as_str());
    }
    ihm.sujets.set_active(Some(0));
  });
}

pub fn selection_sujet(&self) {
  let ihm = self.clone();
  self.sujet.connect_changed(move |_| {
    let mut json = Ressource::new();
    json.rubrique = ihm.rubrique.text().to_string();
    json.sujet = ihm.sujet.text().to_string();
    json.action = "description".to_string();
```

```
      json.soumettre();
      ihm.description.buffer().unwrap().set_text(json.description.as_str());
    });
  }

  pub fn liste_rubriques(&self, ressource: &mut Ressource) {
    let ihm = self.clone();
    ihm.rubriques.remove_all();
    if ressource.liste.is_empty() {
      let mut json = Ressource::new();
      json.action = "rubriques".to_string();
      json.soumettre();
      for rubrique in json.liste {
        ihm.rubriques.append_text(rubrique.as_str());
      }
    }
    else {
      for rubrique in &ressource.liste {
        ihm.rubriques.append_text(rubrique.as_str());
      }
    }
    ihm.rubriques.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.*

*La particularité de ce code c'est que nous faisons référence à une **TextView** qui respecte le modèle **MVC**, et à ce titre sépare l'aspect visuel du modèle associé représenté par un **buffer** de texte. C'est ce dernier que nous utilisons pour intervenir avec le texte saisi ou récupéré.*

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

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

main.rs

```
mod communication;
mod ressources;

use gtk::prelude::*;
use gtk::*;
use ressources::IHM;
use crate::communication::Ressource;

fn main() {
    gtk::init().unwrap();
    feuille_de_style();
    let vue = Builder::from_string(include_str!("../ressources.glade"));
    let fenetre: Window = vue.object("fenetre").unwrap();
    let ihm = IHM::placer(vue);
    ihm.selection_rubrique();
    ihm.selection_sujet();
    ihm.effacer();
    ihm.enregistrement();
    ihm.modification();
    ihm.suppression();
    ihm.liste_rubriques(&mut Ressource::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!("../ressources.css")).unwrap();
    StyleContext::add_provider_for_screen(
        &gdk::Screen::default().unwrap(),
        &css,
        gtk::STYLE_PROVIDER_PRIORITY_APPLICATION
    );
}
```