. RAPPORT FINAL

• VALENTIN LERAY
• RT22 CYBER FI

Table des matières

I. Architecture Globale

1) Client

Le fichier source client.py a pour objectif de permettre la connexion et l'envoi de code à un serveur et de récupérer le résultat de l'exécution de ce dernier. Pour ce faire, le code a été réalisé pour fonctionner de la manière suivante :

- **Interface Utilisateur :**

Le client comprend une interface utilisateur permettant l'utilisation simple des fonctions du programme.

- **Connexion au serveur et affichage de l'état de la connexion :**

La connexion avec le serveur maitre se fait via le bouton « Se connecter au serveur », un fois les champs « IP Serveur » et « Port » complétés, correspondant respectivement à l'adresse IP du serveur maître et le port réservé aux clients sur ce dernier. Le champs texte situé sous le bouton de connexion au serveur permet l'affichage de l'état de la connexion avec le serveur, ou les éventuelles erreurs.

- **Sélection et ouverture des fichiers :**

La sélection d'un fichier se fait via le bouton « Ouvrir fichier », qui permet la sélection de fichier sources (.py, .c, .java). Une fois le fichier sélectionné, le contenu de ce dernier est affiché dans la zone de texte sous le bouton, afin de le lire ou de le modifier.

- **Envoi des programme et affichage des résultats :**

L'envoi des programmes se fait via le bouton « Envoyer Programme », qui envoie le programme ouvert au serveur maître. L'affichage du résultat d'exécution se fait dans la zone de texte sous ce bouton.

- **Fermeture propre du programme**

La fermeture propre du programme se fait via le bouton « Quitter » situé au bas de la fenêtre.

2) Serveur Maître

Le fichier source serveur_maitre.py a pour objectif de mettre en place une connexion vers de multiples clients ainsi qu'une connexion vers de multiples serveurs esclaves. Il doit être capable de recevoir les programmes envoyés par le client, les redistribuer à un serveur esclave en vérifiant qu'il ne soit pas surchargé, de récupérer le résultat de l'exécution, et enfin de le renvoyer au client approprié. Pour ce faire, le code a été réalisé pour fonctionner de la manière suivante :

- **Gestion des connexions :**

Le serveur maitre met en place deux connexions, une pour les clients et une pour les serveurs esclaves, sur deux ports distincts donnés en argument lors du démarrage (--pc : port pour les connexions des clients, --pe : port pour les connexions des serveurs esclaves). Toutes les connexions sont gérées en parallèle pour permettre plusieurs traitements en simultané.

- **Répartition des charges :**

Le serveur maître suit la charge actuelle (en nombre de programmes exécutés simultanément) de chaque serveur esclave connecté. Lorsqu'un programme est reçu d'un client, il le redistribue a un serveur esclave dont la charge est inférieure au nombre de programmes maximum passé en argument au démarrage du serveur maître (--nbr_p : nombre maximum de programmes simultanés par esclave)

- **Redistribution des programmes :**

Lorsqu'un client envoie un programme, le maître enregistre la requête et l'adresse du client, puis envoie le programme à l'esclave sélectionné. La logique utilisée permet que chaque programme soit accompagné d'un identifiant unique permettant de suivre et d'associer la réponse de l'esclave au client approprié.

- **Récupération et redistribution des résultats :**

Après exécution du programme par un serveur esclave, le maître reçoit le résultat et l'envoie directement au client correspondant.

- **Gestion des erreurs :**

Le serveur maître détecte et gère les erreurs de connexion ou les interruptions (client ou esclave) en supprimant automatiquement les connexions concernées tout en maintenant le fonctionnement des autres.

3) Serveur Esclave

Le fichier source **serveur_esclave.py** a pour objectif de recevoir les programmes envoyés par le serveur maître, d'exécuter ces programmes, et de renvoyer les résultats d'exécution. Pour ce faire, le code a été réalisé pour fonctionner de la manière suivante :

- **Connexion au serveur maître** :

Lors de son démarrage, le serveur esclave établit une connexion avec le serveur maître grâce à l'adresse IP du serveur maître et son port réservé aux esclaves, que l'on donnera en argument en le démarrant (--ip_m : IP du serveur maître, --pm : port du serveur maître). Dès que la connexion est établie, il informe le serveur maître des compilateurs qu'il possède.

- **Réception des programmes** :

Chaque programme envoyé par le maître est reçu sous forme d'une chaîne de caractères accompagnée d'un identifiant de client. Le serveur esclave renvoie cet identifiant dans sa réponse pour que le serveur maître soit capable d'identifier le client associé au programme

- **Exécution des programmes :**

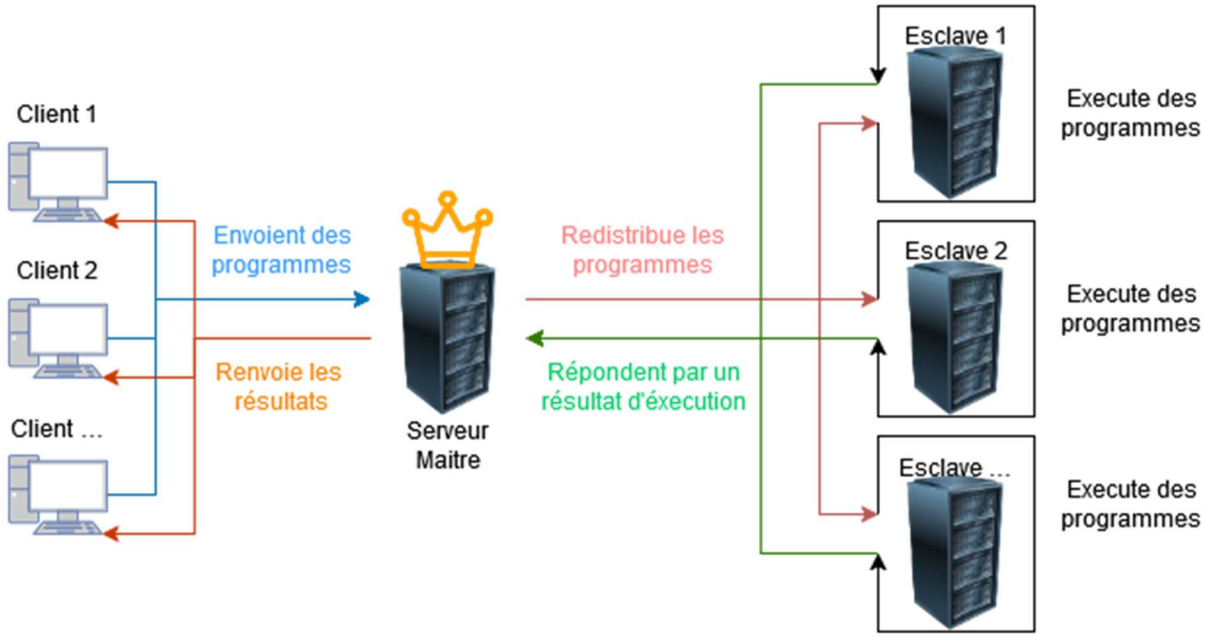
Le serveur esclave exécute les programmes reçus après avoir identifié le type de langage de programmation auquel ils correspondent.

- **Gestion des erreurs** :

Le serveur esclave est conçu pour détecter les erreurs de communication ou d'exécution et renvoyer un message d'erreur au serveur maître dans ce cas.

4) Structure Finale

Une fois la structure complète, voici comment fonctionnent les différents codes les uns par rapport aux autres :



II. Choix des Technologies

1) Client

Afin de réaliser l'interface graphique sur le client, nous nous sommes servis du module Python PyQT6 qui permet assez simplement de mettre en place une interface utilisateur. Nous avons pour cela défini plusieurs objets tels que des labels, des zones de texte ou encore des boutons en fonction de l'affichage souhaité, en prenant soin de modifier leurs dispositions à l'aide d'une Grid.

```python
def init_ui(self):
    """
    Configure l'interface utilisateur de la fenêtre principale.
    """
    widget = QWidget()
    self.setCentralWidget(widget)
    grid = QGridLayout()
    widget.setLayout(grid)

    label_serveur = QLabel("IP Serveur :")
    label_port = QLabel("Port :")
    self.label_connected = QLabel('')
    self.label_connected.setObjectName('label_connected')
    self.label_connected.setMaximumHeight(20)
    self.bouton_connexion = QPushButton("Se connecter au serveur")
    self.input_serveur = QLineEdit('127.0.0.1')
    self.input_port = QLineEdit('4200')
    self.text_editor = QTextEdit('')
    self.bouton_ouvrir = QPushButton('Ouvrir fichier')
    self.bouton_envoyer_fichier = QPushButton('Envoyer Programme')
    self.affichage = QTextEdit('')
    self.affichage.setReadOnly(True)
    self.quit = QPushButton("Quitter")

    # Configuration de la disposition
    grid.addWidget(label_serveur, 0, 0)
    grid.addWidget(self.input_serveur, 0, 1)
    grid.addWidget(label_port, 0, 2)
    grid.addWidget(self.input_port, 0, 3)
    grid.addWidget(self.bouton_connexion, 1, 0, 1, 4)
    grid.addWidget(self.label_connected, 2, 0, 1, 4)
    grid.addWidget(self.bouton_ouvrir, 3, 0, 1, 2)
    grid.addWidget(self.bouton_envoyer_fichier, 3, 2, 1, 2)
    grid.addWidget(self.text_editor, 5, 0, 2, 2)
    grid.addWidget(self.affichage, 5, 2, 2, 2)
    grid.addWidget(self.quit, 7, 0, 1, 4)
```

Enfin les boutons ont été reliés aux actions qu'ils déclenchent en leur associant une fonction qui sera exécutée lorsque le bouton est cliqué.

```python
# Connexions des boutons à leurs méthodes
self.quit.clicked.connect(self.quitter)
self.bouton_connexion.clicked.connect(self.connect)
self.bouton_ouvrir.clicked.connect(self.ouvrir_fichier)
self.bouton_envoyer_fichier.clicked.connect(self.envoyer_fichier)

self.setWindowTitle("Client Graphique")
self.resize(400, 400)
```

En ce qui concerne la gestion de la connexion au serveur, c'est le module intégré à Python « socket » qui a été utilisé, afin de permettre un échange asynchrone des informations. De plus, afin d'éviter le blocage de l'interface graphique lors de l'attente d'une réponse, le module « threading » a été utilisé, permettant de traiter plusieurs tâches simultanément.

La fonction « connect », déclenchée lorsque l'on appuie sur le bouton se connecter au serveur , Établi ou interrompt la connexion avec le serveur maître en fonction de l'état actuel de la connexion. Si la connexion a été établie, un thread est lancé en parallèle pour réceptionner les messages entrants grâce à la fonction « recevoir_message ».

```python
def connect(self):
    """
    Établit ou interrompt la connexion avec le serveur.
    """
    global client_connected, client, addresse_serveur, port
    if client_connected:
        self.deconnecter()
    else:
        try:
            addresse_serveur = self.input_serveur.text()
            port = int(self.input_port.text())
            client = socket.socket()
            client.connect((addresse_serveur, port))
        except Exception as e:
            self.label_connected.setText(f"Erreur : {e}")
        else:
            client_connected = True
            self.label_connected.setText('Connecté au serveur')
            self.bouton_connexion.setText("Se déconnecter du serveur")
            thread_reception = threading.Thread(target=self.recevoir_messages)
            thread_reception.start()
```

Un autre point important à préciser à propos du fonctionnement du client, et à propos de la manière donc on va être traité le programme envoyé, se situe dans la fonction « ouvrir_fichier ». Cette dernière va ajouter, en fonction de l'extension du fichier, un commentaire dans le langage de programmation qui correspond, commentaire qui permettra d'identifier le type de langage reçu sur les serveurs.

```python
def ouvrir_fichier(self):
    """
    Ouvre un fichier source et ajoute un commentaire correspondant au langage de ce dernier.
    """
    file_path, _ = QFileDialog.getOpenFileName(self, "Ouvrir un fichier", "", "Source Files (*.py *.c *.java);;All Files (*)")
    if file_path:
        with open(file_path, 'r') as file:
            contenu = file.read()
        if file_path.endswith('.c'):
            commentaire = "//c\n"
        elif file_path.endswith('.java'):
            commentaire = "//java\n"
        elif file_path.endswith('.py'):
            commentaire = "#python\n"

        contenu_modifie = commentaire + contenu
        self.text_editor.setText(contenu_modifie)
```

2) Serveur Maître

En ce qui concerne les solutions technologiques mises en place sur le serveur maître, il y a plusieurs points importants à souligner.

Tout d'abord le module « socket » est utilisé pour gérer la connexion des clients et des esclaves. C'est bien 2 connexions qui sont créées, une pour tous les clients et une pour les esclaves, afin de séparer ces derniers.

```
serveur_clients = socket.socket()
serveur_esclaves = socket.socket()
```

De plus, « argparse » a aussi été utilisé pour permettre l'intégration d'arguments en ligne de commande pour notre code. Ces arguments ont pour but d'éviter l'écriture en dur dans le code des variables qui pourraient être menées à changer en fonction des installations.

```
parser = argparse.ArgumentParser(description="Serveur maître pour clients et esclaves.")
parser.add_argument("--pc", type=int, default=4200, help="Port pour les connexions clients (par defaut : 4200).")
parser.add_argument("--pe", type=int, default=4300, help="Port pour les connexions esclaves (par defaut : 4300).")
parser.add_argument("--nbr_p", type=int, default=2, help="Nombre maximum de programmes simultanés par esclave (par defaut : 2).")
args = parser.parse_args()
```

Dans notre cas cela nous permet d'adapter les ports clients et esclaves du serveur maître ainsi que le nombre de programmes maximum par esclave en fonction de la situation. Pour plus d'informations sur les arguments, démarrez le serveur suivi de l'argument « -h » ou « --help ».

Sur le serveur maître, on peut retrouver différents dictionnaires qui ont pour but de stocker respectivement les esclaves actifs ainsi que leur charge, et les clients qui sont connectés au serveur maître. Ces dictionnaires sont définis globalement afin d'être accessible par toutes les fonctions.

```
esclaves_actifs = {}
charge_esclaves = {}
clients_connectes = {}
lock = threading.Lock()
```

L'utilisation de threads lock découle de différentes erreurs que j'ai pu observer lorsque des threads modifiaient des mêmes listes simultanément. Cela permet d'éviter que plusieurs threads effectue des actions sur les mêmes dictionnaires en même temps.

Après avoir eu quelques difficultés à propos de la redistribution des messages au bon client en raison d'inversion de sorties, la solution qui a été décidée et d'inclure le port du client dans le message qui va être redistribué aux esclaves afin de s'en servir comme d'un identifiant unique que l'esclave renverra au serveur maître. En effet, en utilisant une fonction qui écoute constamment les esclaves dans un thread, on peut extraire le port retransmis par l'esclave afin de redistribuer la sortie d'exécution au bon client en cherchant dans le dictionnaire des connexions.

En ce qui concerne l'équilibrage de charge et la vérification de la présence des compilateurs sur les esclaves, c'est le serveur maître qui stocke les informations nécessaires dans ses dictionnaires. L'information de la présence des compilateurs êtes envoyé au serveur maître dès l'initialisation d'une connexion avec un esclave et est stocké sous la forme de booléen. Les compilateurs en question sont GCC pour le code C et Javac pour du code Java. Comme le serveur maître se base sur les informations envoyées par l'esclave, et que l'esclave ne vérifie que la présence de ces 2 compilateurs-ci, il est important d'utiliser ces 2 derniers sur les serveurs esclaves. Au moment de

redistribuer le script aux esclaves, le serveur maître n'a plus qu'à vérifier le type de code reçu, afin de l'envoyer vers un esclave capable de traiter ce programme et qui n'est pas surchargé.

3) Serveur Esclave

Le serveur esclave a été pensé pour tourner sur une distribution Linux. Il a pour but d'exécuter et/ou de compiler du code, c'est pourquoi il a été pensé pour fonctionner avec 2 compilateurs en particulier, qui sont GCC et Javac, pour du code C et Java respectivement. De plus, en raison de l'utilisation du module « shutil » pour son appel à la fonction shutil.which(), qui vérifie la présence ou non des compilateurs dans les variables d'environnement, il est important de s'assurer que les exécutables des compilateurs se trouve en effet bien dans les variables d'environnement $PATH.

De plus, comme le serveur esclave tente de se connecter au serveur maître dès son démarrage , il est important de considérer qu'il faut d'abord démarrer le serveur maître avant de vouloir démarrer les esclaves.

En outre, tout comme pour le serveur maître, le module « argparse » a été choisi pour permettre dans ce cas-ci de préciser l'adresse IP du serveur maître ainsi que le port réservé aux esclaves sur le serveur maître. Pour plus d'informations sur les arguments, démarrez le serveur suivi de l'argument « -h » ou « --help ».

```
parser = argparse.ArgumentParser(description="Serveur esclave qui exécute le code envoyé par le serveur maître.")
parser.add_argument("--ip_m", type=str, default='127.0.0.1', help="Adresse IP du serveur maître (par defaut : localhost).")
parser.add_argument("--pm", type=int, default=4300, help="Port utilisé pour la connexion au master (par defaut : 4300).")
args = parser.parse_args()
```

Le serveur esclave est pensé pour pouvoir traiter plusieurs programmes en même temps, et C'est pourquoi la réception d'un message du serveur maître déclenche la création d'un nouveau thread pour exécuter le code et renvoyer un retour d'exécution. Afin d'éviter que ces différents threads n'interférent entre eux, le module « subprocess » a été choisi, car il permet d'isoler ces exécutions dans différents sous-processus.

Comme la compilation des langages Java et C requiert l'utilisation de fichiers temporaires, il a été choisi d'utiliser le module « random » afin de générer des noms de fichiers aléatoires pour éviter tout type d'erreur lié à des interférences en raison d'un nom de fichier similaire.

```
try:
    nom_temporaire = f"temp_{random.randint(0, 9999)}.c"
    nom_sans_extension = nom_temporaire.split(".")[0]
```

Enfin comme précisé dans la partie sur le serveur maître, le port du client qui a été extrait du message reçu par le serveur maître est réinjecté dans le résultat d'exécution que l'on va lui renvoyer, afin que ce dernier puisse le redistribuer au client qui correspond

III. Déploiement et Tests

Afin de procéder au déploiement du système, je vous invite à suivre la procédure d'installation que vous pouvez retrouver au lien suivant : https://github.com/LerayValentin/R309-SAE302, dans le répertoire /SAE302.

Une fois cette procédure d'installation terminée, il faut tester l'installation du système. Pour ce faire, démarrez votre serveur maître puis vos serveurs esclaves. Si le serveur maître a correctement démarré et que les serveurs esclaves ont par la suite réussi à se connecter au serveur maître, vous pouvez maintenant connecter un client au serveur maître. Pour ce faire, démarrez le client et, entrez l'adresse IP du serveur maître dans la case prévue à cet effet ainsi que le port réservé aux clients que vous avez défini au démarrage de votre serveur maître, puis appuyez sur le bouton « Se connecter ». Un message devrait maintenant être apparu vous indiquant que vous êtes correctement connecté au serveur.

Il ne reste plus qu'à charger un fichier à l'aide du bouton « Ouvrir un fichier », puis de l'envoyer à l'aide du bouton prévu à cet effet. Après un court instant, le résultat d'exécution devrait apparaître dans la partie droite de la fenêtre.

En cas d'erreur ou de dysfonctionnement, reprenez le guide d'installation en veillant bien à suivre toutes les étapes.

IV. Réflexion à propos du projet

1) Difficultés rencontrées

Au cours de ce projet j'ai eu l'occasion de rencontrer bon nombre de difficultés. Qu'il s'agisse de la restructuration de mon code en raison d'erreurs, ou bien de difficultés à imaginer comment fonctionnerait mon système, il y a plusieurs points à soulever.

Ayant décidé de commencer par une architecture plus simple pour faciliter la mise en place de mon projet, ma première grosse difficulté est apparue lorsque j'ai voulu connecter plusieurs clients avec un unique serveur, et concerne la gestion des sorties d'exécutions. En effet lorsque j'exécutais plusieurs codes simultanément à partir de différents clients, et notamment lorsqu'un code prenait plus de temps à s'exécuter que l'autre, j'ai observé une inversion de sortie qui menait à l'envoi du mauvais résultat au client.

Le problème étant que, même une fois ce problème résolu dans le cadre de mon architecture simple, j'ai pu observer des problèmes similaires lorsque j'ai voulu réaliser l'architecture complexe, c'est-à-dire un serveur maître et des serveurs esclaves. En effet, en attendant une réponse du serveur esclave dans le thread qui lui avait envoyé le message, j'observais à nouveau cette inversion de sortie qui causait le renvoi de mauvais résultats au client. C'est pour cela que j'ai opté pour une structure qui redistribue le message au serveur esclave en fournissant le port en tant que identifiant unique du client, pour s'assurer que le bon résultat parvienne jusqu'au bon client.

Un autre problème que j'ai rencontré et due à mon manque de connaissances du langage de programmation Java. En effet, je ne savais pas que la classe publique définie dans le code devait portez le même nom que le nom du fichier. C'est pourquoi après avoir décidé de donner des noms aléatoires au fichier temporaire enregistré sur le serveur esclave, j'observais de nombreuses erreurs, ce qui m'a poussé à me renseigner un peu plus sur le sujet.

2) Améliorations possibles

Au vu de l'état final de mon projet, me viennent des idées d'amélioration qui pourrait être apportée à ce dernier à mon niveau.

La première idée qui me vient à l'esprit s'orienterai vers la question de sécurité du projet. En effet dans l'état actuel des choses, aucune communication n'est chiffrée et les informations passent en clair, ce qui soulève des grandes questions de sécurité et laisse le système vulnérable à des attaques très basiques.

De plus le fait que j'ai utilisé le port du client comme l'identifiant unique le reliant à son résultat d'exécution pour elle là encore causer des problèmes, d'usurpation d'identité par exemple. Il pourrait être intéressant de modifier cet identifiant unique par 1 ID généré aléatoirement qui serait stocké en lien avec le client sur le serveur maître.

De plus dans l'état actuel des choses, lorsque tous les serveurs esclaves sont surchargés, le serveur maître renvoie un message d'erreur au client lui indiquant qu'aucun esclave n'est disponible pour exécuter son code. Il pourrait être intéressant de mettre au point un système de file d'attente, qui garderait donc en mémoire le code envoyé par le client en attendant qu'un esclave soit disponible pour son exécution.