RAPPORT SAE 3.02

JAEGER Bastian

UNIVERSITÉ HAUTE-ALSACE

1. Introduction SAE 3.02

Objectif :

L'objectif principal de cette SAE, était de concevoir et développer une architecture distribuée permettant de gérer la compilation et l'exécution de programmes envoyés par des clients. Le projet combinait des concepts comme la communication réseau via sockets, le threading pour la gestion simultanée de clients, et une interface graphique PyQt6 pour offrir une expérience utilisateur intuitive.

Architecture :

L'architecture se compose des éléments suivants :

Serveur Maître : Responsable de la gestion des connexions, de la répartition des tâches et du monitoring des ressources. Le serveur maître peut également compiler les programmes en local et mettre en attente les clients si tous les serveurs secondaires sont surchargés. Si aucun serveur secondaire n'est disponible, il traite les tâches dans les limites de ses ressources.

Serveurs Secondaires : Chargés de l'exécution des programmes reçus et de la communication des résultats au serveur maître.

Client : Fournit une interface utilisateur pour soumettre des programmes et afficher les résultats.

2. Processus du Développement

2.1 Connexion entre le client et le serveur maitre

J'ai d'abord travaillé sur le client et le serveur maître afin d'établir une connexion fiable entre eux avec des sockets et voir si je peux envoyer des messages entre les deux.

2.2 Compilation et Retour des résultats

Une fois la connexion établie, je me suis concentré sur la possibilité pour le client d'envoyer un programme à compiler et d'obtenir le résultat. Avec l'utilisation du module **subprocess** pour exécuter les commandes de compilation et d'exécution sur le serveur maître. Puis les messages sont retourné grâce aux stdout et si il y'a des erreurs par stderr générés par les commandes qui sont capturés grâce à subprocess.run avec les arguments stdout=subprocess.PIPE et stderr=subprocess.PIPE.

2. Processus du Développement

```python
# ————————————
# -FONCTION COMPILATION / EXECUTION PROGRAMME-
# ————————————

def execution_programme(language_code, fichier, adresse_client, programme=None):
    try:
        systeme = platform.system()

        # ————————————
        # -PYTHON-
        # ————————————

        if language_code == "py":
            resultat_programme = subprocess.run(
                ['python3' if systeme != 'Windows' else 'python', fichier],stdout=subprocess.PIPE,stderr=subprocess.PIPE,text=True
            )

        # ————————————
        # -JAVA-
        # ————————————

        elif language_code == "java":
            classname = os.path.splitext(os.path.basename(fichier))[0]
            subprocess.run(['javac', fichier], check=True)
            resultat_programme = subprocess.run(
                ['java', classname],stdout=subprocess.PIPE,stderr=subprocess.PIPE,text=True
            )
            os.remove(classname + ".class")

        # ————————————
        # -C-
        # ————————————

        elif language_code == "c":
            executable_sortie = f"prog_{adresse_client[1] if adresse_client else 'default'}_{threading.get_ident()}"
            executable_sortie += ".exe" if systeme == 'Windows' else ""
            subprocess.run(['gcc', fichier, '-o', executable_sortie], check=True)
            resultat_programme = subprocess.run(
                [f'./{executable_sortie}' if systeme != 'Windows' else executable_sortie],stdout=subprocess.PIPE,stderr=subprocess.PIPE,text=True
            )
            os.remove(executable_sortie)

        # ————————————
        # -C++-
        # ————————————

        elif language_code == "cpp":
            executable_sortie = f"prog_{adresse_client[1] if adresse_client else 'default'}_{threading.get_ident()}"
            executable_sortie += ".exe" if systeme == 'Windows' else ""
            subprocess.run(['g++', fichier, '-o', executable_sortie], check=True)
            resultat_programme = subprocess.run(
                [f'./{executable_sortie}' if systeme != 'Windows' else executable_sortie],stdout=subprocess.PIPE,stderr=subprocess.PIPE,text=True)
            os.remove(executable_sortie)
        else:
            return "", f"Langage '{language_code}' non supporté."

        return resultat_programme.stdout, resultat_programme.stderr

    except subprocess.CalledProcessError as e:
        return "", f"Erreur d'exécution : {str(e)}"
    except Exception as e:
        return "", f"Erreur : {str(e)}"
```

2. Processus du Développement

```python
# ------------
# -GESTION SAUVEGARDE / LANCEMENT PROGRAMME-
# ------------

def sauvegarde_execution(socket_client, language_code, fichier, programme):
    logging.info(f"Serveur Maitre : Lancement de l'exécution du programme en {language_code}.")
    global active_programs
    active_programs += 1
    try:
        with open(fichier, "wb") as f:
            f.write(programme)
        stdout, stderr = execution_programme(language_code, fichier, None, programme)

        if stdout:
            socket_client.sendall(f"SORTIE:\n{stdout}".encode())
        if stderr:
            socket_client.sendall(f"ERREURS:\n{stderr}".encode())
        if not stdout and not stderr:
            socket_client.sendall("Aucune sortie.".encode())
    except Exception as e:
        envoie_erreur(socket_client, f"Erreur d'exécution : {e}")
    finally:
        active_programs -= 1
        nettoyage(socket_client, fichier)


# ------------
# -FONCTION RENVOIE DU RESULTAT-
# ------------

def envoie_sortie(socket_client, stdout, stderr):
    if stdout: socket_client.sendall(f"SORTIE:\n{stdout}".encode())
    if stderr: socket_client.sendall(f"ERREURS:\n{stderr}".encode())
    if not stdout and not stderr: socket_client.sendall("Aucune sortie.".encode())

def envoie_erreur(socket_client, message):
    if socket_client.fileno() != -1:
        try:
            socket_client.sendall(message.encode())
        except OSError as e:
            logging.warning(f"Impossible d'envoyer une erreur au client : {e}")


# ------------
# -FONCTION NETTOYAGE FICHIER TEMPORAIRE-
# ------------

def nettoyage(socket_client, fichier):
    if socket_client and socket_client.fileno() != -1:
        socket_client.close()
    if fichier and os.path.exists(fichier):
        os.remove(fichier)
```

2. Processus du Développement

```python
def gestion_client(socket_client, adresse_client):
    fichier = None
    try:
        header_data = socket_client.recv(1024).decode()
        secret_key, language_code, taille_programme = header_data.split(':')
        taille_programme = int(taille_programme)
        if secret_key != SECRET_KEY:
            raise ValueError("Clé secrète invalide")

        socket_client.sendall("HEADER_RECUE".encode())
        programme = reception_données(socket_client, taille_programme)
        fichier = prepare_fichier(language_code, programme, adresse_client)

        if delegation_programme():
            sauvegarde_execution(socket_client, language_code, fichier, programme)
        elif delegation_serveurs_autres(socket_client, adresse_client, language_code, taille_programme, programme):
            logging.info(f"Client {adresse_client} délégué à un serveur secondaire.")
        else:
            try:
                socket_client.sendall(b"ATTENTE")
                logging.info(f"Client {adresse_client} mis en attente faute de ressources.")
                request_queue.put((socket_client, adresse_client, programme, header_data))
            except (socket.error, BrokenPipeError) as e:
                logging.warning(f"Erreur lors de l'envoi de l'attente au client {adresse_client} : {e}")
                raise
    except Exception as e:
        envoie_erreur(socket_client, f"Erreur : {e}")
    finally:
        if fichier:
            nettoyage(None, fichier)
```

2. Processus du Développement

2.3 Délégation sur les Serveurs Secondaires

Puis je me suis penché, sur le fait d'élaboré un système pour permettre au serveur maître de déléguer les tâches à des serveurs secondaires. J'ai du mettre en place la communication entre le serveur maitre et le serveur secondaire à l'aide de Socket puis je me suis penché sur l'implémentation d'une commande STATUS permettant au serveur maître de connaître l'état des serveurs secondaires (CPU, RAM disponibles) ce qui permet la délégation des tâches en fonction des ressources disponibles.

```python
# ------------
# -FONCTION DELEGATION AUX AUTRES SERVEURS-
# ------------

def delegation_serveurs_autres(socket_client, adresse_client, language_code, taille_programme, programme):
    meilleur_serveur = choisir_meilleur_serveur()
    if not meilleur_serveur:
        return False

    ip_serveur, port_serveur = meilleur_serveur
    try:
        socket_autres = socket.create_connection((ip_serveur, port_serveur))
        header = f"{SECRET_KEY}:{language_code}:{taille_programme}"
        socket_autres.sendall(header.encode())

        response = socket_autres.recv(1024).decode()
        if response != "HEADER_RECUE":
            return False

        socket_autres.sendall(programme)
        while True:
            data = socket_autres.recv(4096)
            if not data:
                break
            socket_client.sendall(data)

        socket_autres.close()
        return True
    except Exception as e:
        logging.warning(f"Délégation échouée vers {ip_serveur}:{port_serveur} - {e}")
        return False
```

```python
active_programs = 0

def delegation_programme():
    global active_programs
    return (active_programs < MAX_PROGRAMMES and psutil.cpu_percent(interval=1) < MAX_CPU_USAGE and psutil.virtual_memory().percent < MAX_RAM_USAGE)
```

2. Processus du Développement

2.4 Monitoring

Je me suis aussi penché sur le fait d'ajouter des fonctionnalités de monitoring sur le serveur maître et les serveurs secondaires ce qui m'a permis aussi de renvoyer l'état des serveurs au serveur maitre. Cela permet la suivi en temps réel des ressources CPU et RAM utilisées et l'affichage des programmes actifs sur chaque serveur.

```python
# ------------
# -FONCTION MONITEUR CPU / RAM-
# ------------

def Moniteur_CPU_RAM():
    while True:
        CPU = psutil.cpu_percent(interval=1)
        RAM = psutil.virtual_memory().percent
        RAM_MB = psutil.virtual_memory().used / (1024 ** 2)
        PROGRAMMES = active_programs
        logging.info(f"Utilisation CPU : {CPU}% / {MAX_CPU_USAGE}% | Utilisation RAM : {RAM_MB:.2f} MB ({RAM}%) / {MAX_RAM_USAGE}% | Programmes en cours : {PROGRAMMES}/{MAX_PROGRAMMES}")
        time.sleep(0.5)
```

2. Processus du Développement

2.5 File d'Attente et Priorité pour la Délégation

Puis à la fin j'ai conçu une file d'attente pour gérer les clients, ce qui ce qui permet de mettre en attente les clients lorsque tous les serveurs sont occupés ou lorsque seul le serveur maître est opérationnel. Dès qu'un serveur se libère, le système sélectionne automatiquement le serveur avec la plus faible utilisation des ressources pour exécuter la prochaine tâche en attente. Cette partie a été particulièrement complexe à gérer et a nécessité de nombreux ajustements. À un moment donné, mon code est devenu assez désordonné, mais avec persévérance, j'ai réussi à structurer et faire fonctionner cette fonctionnalité correctement.

```python
def stauts_serveurs(ip, port):
    try:
        with socket.create_connection((ip, port), timeout=2) as s:
            s.sendall("STATUS".encode())
            response = s.recv(1024).decode()
            parts = dict(item.split(":") for item in response.split("/"))
            charge = int(parts["CHARGE"])
            max_programmes = int(parts["MAX_PROGRAMMES"])
            return charge, max_programmes
    except Exception as e:
        logging.warning(f"Impossible de récupérer l'état du serveur {ip}:{port} - {e}")
        return float('inf'), None


def choisir_meilleur_serveur():
    charges = {}
    for ip, port in SERVEUR_AUTRES:
        charge, max_programmes = stauts_serveurs(ip, port)
        if max_programmes and charge < max_programmes:
            charges[(ip, port)] = charge
    return min(charges, key=charges.get) if charges else None

def gestion_client(socket_client, adresse_client):
    fichier = None
    try:
        header_data = socket_client.recv(1024).decode()
        secret_key, language_code, taille_programme = header_data.split(':')
        taille_programme = int(taille_programme)
        if secret_key != SECRET_KEY:
            raise ValueError("Clé secrète invalide")

        socket_client.sendall("HEADER_RECUE".encode())
        programme = reception_données(socket_client, taille_programme)
        fichier = prepare_fichier(language_code, programme, adresse_client)

        if delegation_programme():
            sauvegarde_execution(socket_client, language_code, fichier, programme)
        elif delegation_serveurs_autres(socket_client, adresse_client, language_code, taille_programme, programme):
            logging.info(f"Client {adresse_client} délégué à un serveur secondaire.")
        else:
            try:
                socket_client.sendall(b"ATTENTE")
                logging.info(f"Client {adresse_client} mis en attente faute de ressources.")
                request_queue.put((socket_client, adresse_client, programme, header_data))
            except (socket.error, BrokenPipeError) as e:
                logging.warning(f"Erreur lors de l'envoi de l'attente au client {adresse_client} : {e}")
                raise
    except Exception as e:
        envoie_erreur(socket_client, f"Erreur : {e}")
    finally:
        if fichier:
            nettoyage(None, fichier)
```

2. Processus du Développement

```python
request_queue = Queue()

def gestion_file_attente():
    while True:
        socket_client, adresse_client, programme, header_data = request_queue.get()
        try:
            if socket_client.fileno() == -1:
                logging.warning(f"Socket du client {adresse_client} fermé. Retiré de la file.")
                continue

            secret_key, language_code, taille_programme = header_data.split(':')
            taille_programme = int(taille_programme)

            if delegation_programme():
                fichier = prepare_fichier(language_code, programme, adresse_client)
                sauvegarde_execution(socket_client, language_code, fichier, programme)
                logging.info(f"Client {adresse_client} exécuté localement depuis la file d'attente.")
            elif delegation_serveurs_autres(socket_client, adresse_client, language_code, taille_programme, programme):
                logging.info(f"Client {adresse_client} délégué depuis la file d'attente à un serveur secondaire.")
            else:
                if not socket_client._closed:
                    try:
                        socket_client.sendall(b"ATTENTE")
                        logging.info(f"Client {adresse_client} mis en attente faute de ressources.")
                    except (socket.error, BrokenPipeError) as e:
                        logging.warning(f"Erreur lors de la tentative de mise en attente pour {adresse_client} : {e}")
                        continue
                time.sleep(1)
                request_queue.put((socket_client, adresse_client, programme, header_data))
        except Exception as e:
            logging.error(f"Erreur dans la gestion de la file d'attente pour {adresse_client} : {e}")
            envoie_erreur(socket_client, f"Erreur : {e}")
        finally:
            request_queue.task_done()
```

3. Limites du Projet

Le projet, bien que fonctionnel et robuste, présente certaines limites qui peuvent être améliorées à l'avenir. Tout d'abord, la sécurité des communications est relativement rudimentaire. Actuellement, les échanges entre les clients, le serveur maître, et les serveurs secondaires sont protégés par une simple clé partagée. Ce niveau de protection peut suffire pour des scénarios de tests ou de démonstration, mais il reste inadéquat pour une utilisation en production, où des méthodes de chiffrement plus avancées comme AES ou RSA devraient être mises en œuvre. Ensuite, le système de monitoring est minimaliste. Bien que des informations sur l'utilisation des ressources CPU et RAM soient disponibles, ces données ne sont accessibles que via des logs ou des retours en ligne de commande. L'absence d'une interface graphique dédiée rend le suivi des performances en temps réel moins intuitif et peut compliquer le diagnostic en cas de problème. Enfin, la gestion des surcharges, bien qu'améliorée par l'ajout d'une file d'attente et d'un système de priorité, pourrait encore être perfectionnée. Les erreurs liées à une surcharge complète des serveurs ne sont pas toujours remontées de manière claire ou détaillée au client. De plus, la coordination entre les différents modules, bien que fonctionnelle, pourrait être optimisée pour mieux gérer les pics de charge et garantir une réactivité accrue du système dans ces situations.

3.2 Amélioration Possibles

- Intégration de chiffrement avancé (AES ou RSA) pour sécuriser les communications.
- Développement d'une interface web pour le monitoring en temps réel.
- Optimisation de l'algorithme de répartition des charges en utilisant des prédictions basées sur l'historique.

4. Conclusion

Le projet m'a permit de développer mes compétences :

- Utilisation avancée des sockets en Python pour la communication réseau.
- Gestion multi-threadée pour optimiser les connexions simultanées.
- Création d'interfaces utilisateur avec PyQt6 pour améliorer l'expérience utilisateur.
- Utilisation de subprocess pour la compilation et l'execution des programmes.
- Délégation des tâches aux serveurs secondaires avec le monitoring où le serveur maître récupère des informations sur l'utilisation des ressources CPU et RAM et le max de progammes pour chaque serveurs avec l'utilisation de psutil.

4.2 Schéma