1 Cahier des charges

Ce devoir est à faire en binôme de préférence, pas de trinôme.

La date de remise du projet est :

- mercredi 4 décembre 2024 à 18h00

Le projet sera déposé sur la plate-forme UPDAGO, sous forme d'un fichier archive au format *tar* compressé avec l'utilitaire *gzip*.

Les enseignants se réservent le droit de convoquer les étudiants à un oral pour des précisions sur le travail.

Le nom de l'archive sera IMPÉRATIVEMENT composé de vos noms de famille (ou d'un seul nom en cas de monôme) en minuscules dans l'ordre lexicographique, d'un underscore, du mot "projet", par exemple *subrenat_zrour_projet*, suivi des extensions classiques (i.e. *".tar.gz"*).
Le désarchivage devra créer, dans le répertoire courant, un répertoire s'appelant : *PROJET*.
Ces directives sont à respecter SCRUPULEUSEMENT (à la minuscule/majuscule près). Un script-shell est à votre disposition pour vérifier votre archive.
Les archives ne passant pas avec succès le vérificateur seront refusées (considérées hors-sujet).

Le langage utilisé est obligatoirement le C. Un programme doit compiler, sans erreur ni warning, sous le Linux des machines des salles de TP avec les options suivantes :
-Wall -Wextra -pedantic -std=c99 (voire *-Wconversion* si vous avez le courage).
De même un programme doit s'exécuter avec *valgrind* sans erreur ni fuite mémoire.

Vous n'êtes pas autorisés à utiliser des bibliothèques ou des composants qui ne sont pas de votre fait, hormis les bibliothèques système. En cas de doute, demandez l'autorisation.

Le répertoire *src* doit être sous *git* que vous devez utiliser. Vous pouvez vous contenter d'une utilisation basique (i.e. uniquement la branche principale) ou bien entendu aller plus loin. Un tutoriel est présent sur l'espace dédié au cours. Dans l'archive à rendre, vous laisserez le répertoire *.git*.
Synchronisez votre dépôt local avec un dépôt distant (*gitlab*, *github*, ...). Vous éviterez ainsi toute perte de travail : à votre niveau, un effacement accidentel ou un crash disque n'est pas une raison valable pour ne pas rendre le projet.

Référez-vous à la section 4 (page 7) pour des directives précises.

Il vous est demandé un travail précis. Il est inutile de faire plus que ce qui est demandé. Dans le meilleur des cas le surplus sera ignoré, et dans le pire des cas il sera sanctionné.

2 Présentation générale du projet

Le but est d'implémenter un *orchestre*, des *services* et des *clients*.

L'*orchestre* est un programme qui tourne en permanence [1]. Il attend que des *clients* le contactent pour leur rendre des *services*.
Les *clients* sont des programmes indépendants.
Les *services* sont des programmes indépendants, mais lancés par l'*orchestre*.

Il n'y a qu'un seul code source, et donc un seul exécutable, pour les *clients*, mais on peut en lancer plusieurs en même temps (dans des consoles de commandes différentes) :
- cela permet d'utiliser plusieurs *services* en même temps
- mais ils se font donc concurrence pour communiquer avec l'*orchestre*.

De même, il n'y a qu'un seul code source (et un seul exécutable) pour tous les *services* et cet exécutable sera lancé plusieurs fois, par l'*orchestre*, en autant d'exemplaires qu'il y a de *services*.

Un *client* s'adresse toujours en premier lieu à l'*orchestre* qui se décharge alors sur un des *services* de

1. on parle de démon/daemon

son choix. À partir de ce moment le *client* échangera directement avec le *service*, indépendamment de l'*orchestre*.

Il y a une série préétablie de *services* pouvant être rendus, i.e. ils sont codés en dur dans des fichiers sources distincts (mais réunis dans le même exécutable). On rappelle que chaque *service* est un processus indépendant, et que tous ces processus font tourner le même exécutable.
Un fichier de configuration, utilisé uniquement par l'*orchestre*, liste les *services* disponibles.

Tous ces processus utilisent, pour communiquer, les sémaphores IPC, les tubes anonymes et les tubes nommés fournis par les bibliothèques du C.

Le fonctionnement est complexe à appréhender et ne vous attendez pas à avoir une idée claire au bout de quelques minutes. Il faut lire plusieurs fois le sujet et n'hésitez pas à discuter entre vous pour confronter l'ordre et la nature des informations qui circulent.
En fin de document plusieurs schémas illustrent les connexions entre les processus.

3 Fonctionnement détaillé

Une partie du code est déjà fournie et est abondamment commentée. Cela complète les explications ci-dessous. Autrement dit toutes les informations nécessaires ne sont pas dans le sujet, il faut lire en parallèle le code source fourni.

3.1 IPC et autres communications

L'*orchestre* est exécuté en premier. Il lance, dès sa création, tous les *services* (fonctions *fork*, *exec*). Chaque *service* est associé à une paire de tubes nommés pour les futures communications avec les *clients*.

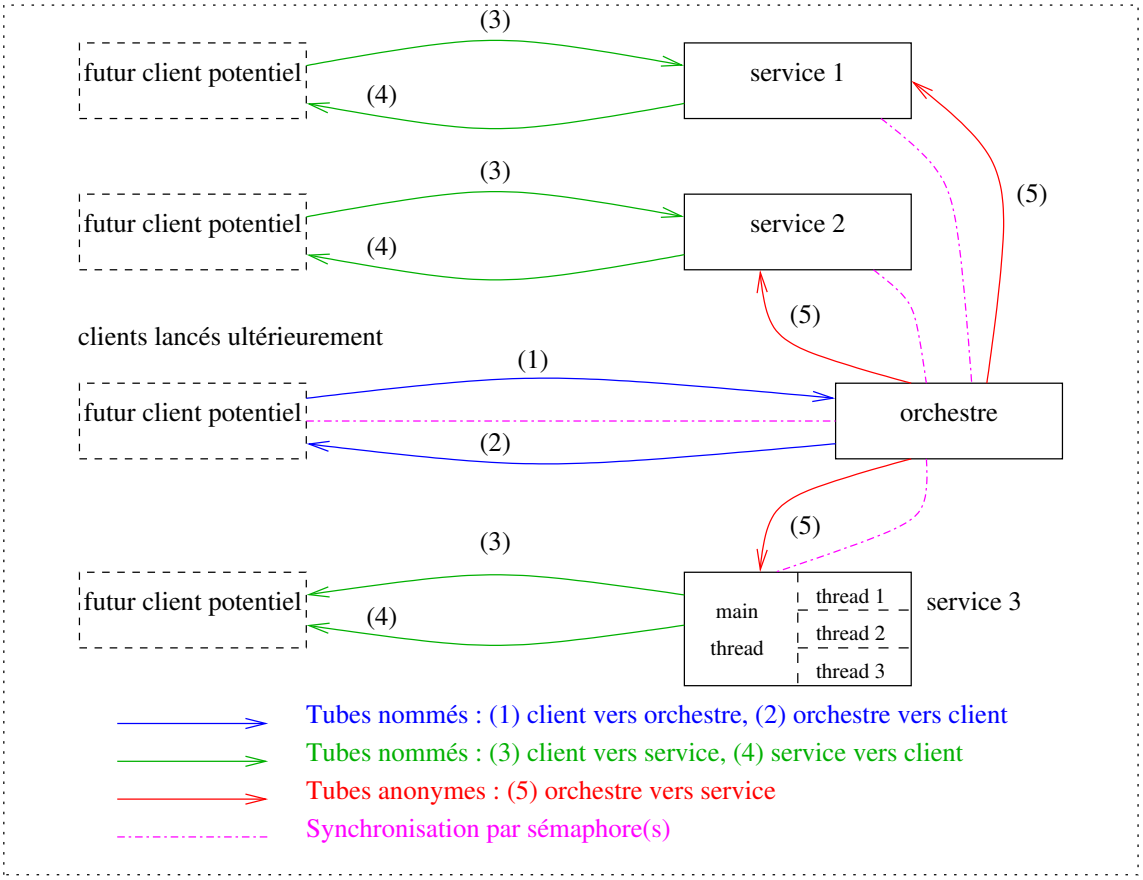
L'*orchestre* et un *service* communiquent :
- par un seul tube anonyme (fonction *pipe*) dans le sens *orchestre* vers *service*,
- un sémaphore IPC (fonctions *semget*, *semop*, ...) pour se synchroniser.
- il y a donc autant de sémaphores et de tubes qu'il y a de *services*.

Un *client* et l'*orchestre* communiquent :
- par une paire de tubes nommés (*mkfifo*, *open*, ...) pour amorcer la discussion,
- un sémaphore IPC (*semget*, *semop*, ...) pour se synchroniser.

Une fois la demande du *client* validée, l'*orchestre* :
- fournit au *client* les noms des tubes nommés associés au *service*, ainsi qu'un mot de passe.

Le *client* et le *service* sont alors complètement indépendants de l'*orchestre* et communiquent via cette paire de tubes qui resteront en place à la fin du traitement.

Voici le schéma au début de la vie de l'*orchestre* :



Il y a des synchronisations et des sections critiques (accès restreints à des portions de code) qui utiliseront obligatoirement les sémaphores IPC (*semget, ...)* pour une gestion entre processus lourds (cf. détails ci-dessous).

Les entrées-sorties seront effectuées avec les fonctions de bas niveau (*open, write, ...*),, sauf pour la lecture du fichier de configuration par l'*orchestre* qui utilisera les fonctions de haut niveau (*fopen, fgets, ...*).

3.2 *client*

Il y a deux tubes nommés, pré-créés par l'*orchestre*, pour obtenir une communication bidirectionnelle entre *clients* et *orchestre*.

Dans un premier temps :

- *client* envoie une demande à l'*orchestre* (sur le premier tube), cette demande étant le numéro du service souhaité (le numéro -1 indiquera à l'*orchestre* de s'arrêter),
- *client* reçoit en retour (sur le deuxième tube) un code indiquant si l'*orchestre* accepte ou non de répondre à la demande,
- si la demande est acceptée, *client* reçoit (sur le deuxième tube) un mot de passe et les noms des deux tubes nommés par lesquels vont désormais transiter toutes les autres communications (i.e. entre le *client* et le *service*).
- dans tous les cas le *client* prévient l'*orchestre* de la fin de la transaction ; l'*orchestre* attend cette information sur le premier tube.
- Le *client* débloque l'*orchestre* grâce à un sémaphore ; en effet, l'*orchestre*, avant de s'occuper d'une nouvelle demande (i.e. un nouveau *client*), doit s'assurer que le *client* courant a complètement terminé sa communication avec lui.

Attention, cette phase est délicate et il ne faudrait pas que deux *clients* se télescopent. Le plus simple (et cette solution est imposée) est que la portion de code gérant cette communication soit exécutée par un seul *client* à la fois (en mettant le code en section critique). Cette exclusivité est sous la responsabilité des *clients*, mais le mutex (i.e un sémaphore) a été préalablement créé par l'*orchestre*.

De même, le sémaphore de synchronisation (utilisé en fin de cette phase du *client*) a été créé et initialisé par l'*orchestre*.

Les tubes nommés entre les *services* et les *clients* sont créés une seule fois par l'*orchestre* au lancement de ce dernier, et seront détruits par ce même *orchestre* à sa terminaison.

Dans un second temps :

- *client* envoie les données à traiter au *service* créé au lancement de l'*orchestre* (par exemple deux nombres, ou une image, ...),
- *client* reçoit le résultat en provenance du *service*,
- dans tous les cas le *client* envoie au *service* un code de fin de communication.

À part le dernier point qui est commun à tous les *clients*, les deux autres ont un protocole complètement libre qui dépend du service demandé. Cependant pour simplifier, on supposera qu'il n'y a qu'un seul aller-retour entre le *client* et le *service*, un aller pour envoyer toutes les données, un retour pour récupérer tous les résultats.

Dans le cas particulier du *client* qui indique à l'*orchestre* de se terminer, ce "second temps" n'existe pas.

Dans le code fourni (répertoire *CLIENT*), une proposition d'organisation de l'implémentation est détaillée.

3.3 *service*

Bien que décrit ici, il est préférable d'implémenter le code des *services* une fois les codes du *client* et de l'*orchestre* fonctionnels.

Un *service* est issu d'un *fork/exec* de l'*orchestre* dès le lancement de ce dernier. Le rôle du *service* est de répondre à un seul *client* à la fois.
C'est le premier paramètre de la ligne de commande (i.e. *argv[1]*) qui indique au *service* quel service il doit gérer.

Un *service* effectue les opérations suivantes :

- attendre, sur le tube anonyme, l'ordre de l'*orchestre* de gérer un *client*, ainsi qu'un mot de passe,
- récupérer (sur le tube nommé 1) le mot de passe envoyé par le *client* et vérifier qu'il correspond à celui envoyé par l'*orchestre*,
- envoyer (sur le tube nommé 2) si le mot de passe est accepté ou non,
- récupérer (sur le tube nommé 1) les données (envoyées par le *client*) à traiter,
- calculer le résultat,
- envoyer (sur le tube nommé 2) le résultat au *client*.
- attendre le code de fin provenant du *client*.
- prévenir l'*orchestre*, via le sémaphore, qu'il a terminé son traitement,
- recommencer.

Comme indiqué dans la section précédente, tous les *services* ont le même fonctionnement ; seules les phases de réception, calcul et envoi du résultat sont propres à chaque *service*.

Dans le code fourni (répertoire *SERVICE*), une proposition d'organisation de l'implémentation est détaillée.

3.4 Services à implémenter

On impose un minimum de 3 services.
Dans le code fourni (fichier *SERVICE/0README*), il y a des explications détaillées.

3.4.1 Service 1 : somme de deux *float*

Les données à traiter sont deux *float*, le résultat est un *float*.

3.4.2 Service 2 : compression

La donnée à traiter est une chaîne de caractères, le résultat est une chaîne de caractères qui est "compressée" selon un algorithme spécial.

Chaque suite de caractères identiques est remplacée par un nombre (la longueur de la suite) et le caractère. On supposera que la chaîne initiale ne comporte pas de chiffre.

Par exemple, la chaîne "ccccccccccchhhattttt" est transformée en "12c3h1a5t". La chaîne "chat" est transformée en "1c1h1a1t".
On voit alors que la "compression" peut conduire à un doublement de la taille (ce qui est le pire des cas) !

3.4.3 Service 3 : somme d'un tableau

Le calcul est une simple somme des cases d'un tableau de *float*. La difficulté est que l'on impose une résolution multi-threads.

Les données à traiter sont le tableau et le nombre de threads (choisi par le *client* donc). Le résultat est un *float*.

Chaque thread s'occupe d'une portion distincte du tableau et calcule sa somme locale.
Chaque thread met à jour une une variable partagée (attention aux conflits d'accès).
Lorsque tous les threads de calcul ont terminé, le thread principal envoie le résultat au *client*.

Par exemple, soit le tableau suivant :

15	3	12	8	2	7	8	1	10	12	9	4	5

Voici une résolution avec 3 threads :

- *thread1* s'occupe des 4 premières cases et calcule 38 (en local)
- *thread2* s'occupe des 4 cases suivantes et calcule 18 (en local)
- *thread3* s'occupe des 5 dernières cases et calcule 40 (en local)

Si on suppose que l'ordre des tentatives de modification de la variable partagée est *thread2*, *thread1*, *thread3*, alors :

- *thread2* dépose 18
- *thread1* dépose 56 (i.e. $18 + 38$)
- *thread3* dépose 96 (i.e. $56 + 40$)

3.5 *orchestre*

3.5.1 Fichier de configuration

Le fichier de configuration a une structure fixe et simplifiée à l'extrême :

- *ligne 1 :* nombre de *services* disponibles,
- *ligne 2 :* nom de l'exécutable du *service*,
- puis une ligne par *service* :
  - le numéro du service (de 0 à nombre services - 1)
  - mot indiquant si le service est ouvert ou fermé
- les lignes suivantes sont ignorées (i.e. non lues)
- on supposera que les données du fichier sont correctes sans avoir à faire de tests d'erreurs.

Ce fichier n'est lu et connu que par l'*orchestre*. Notamment un *service* ne sait pas s'il est ouvert ou fermé. S'il est fermé il ne sera simplement jamais sollicité par l'*orchestre*.

Dans le code fourni (répertoire *CONFIG*) il y a :

- un exemple de fichier de configuration correct
- un programme permettant de tester les fichiers de configuration

— le fichier *0README* qui explique la structure d'un fichier de configuration et qui montre les sorties écran du programme de test

Pour lire et exploiter un fichier de configuration il y a une API imposée. Les deux fichiers sont dans le répertoire *CONFIG* :

— *config.h* : signature des fonctions de l'API, vous n'avez pas le droit de modifier ce fichier.

— *config.c* : implémentation de l'API qui est à votre charge (mais à faire en tout dernier).

Le programme de test utilise cette API.

Seul l'*orchestre* (et le programme de test) utilisera cette API.

3.5.2 processus *orchestre*

Le principe général est le suivant sur une boucle "infinie" :

- attendre la connexion d'un *client*
- analyser la demande du *client*
- si elle est correcte, déclencher le *service* et le laisser gérer le *client*

Un *service* ne peut gérer qu'un seul *client* à la fois. Si un *client* demande un *service* qui est déjà occupé par un autre *client*, alors le deuxième *client* demandeur recevra un code d'échec de l'*orchestre*.

De manière plus précise, voici l'algorithme de l'*orchestre* ;

- initialisations diverses (dont lecture du fichier de configuration)
- lancement de tous les *services* (qu'ils soient ouverts ou non) avec les tubes nommés associés, les tubes anonymes et les sémaphores
- répéter (jusqu'à ordre de fin)
    - se connecter sur les tubes de communication avec le futur *client*
    - attendre l'envoi d'un numéro de *service* par un *client*
    - détecter des fins éventuelles de *services* en cours de traitement
    - si le numéro est correct
        - générer un mot de passe et l'envoyer au *service* visé
        - indiquer au *client* le mot de passe et les noms des tubes préexistants
    - attente de l'accusé de réception du *client*
- fin répéter
- attente de la fin des traitements en cours
- envoi aux *services* d'un code de terminaison
- attente de la terminaison des *services*
- destruction des tubes

Dans le code fourni (répertoire *ORCHESTRE*), une proposition d'organisation de l'implémentation est détaillée.

3.5.3 Fin de l'*orchestre* (hors projet)

Cette question NE doit PAS être faite (i.e. ne doit pas être présente dans le code rendu). Elle n'est là que pour vous montrer les fonctionnalités d'un "vrai" démon.

Pour terminer l'*orchestre*, plutôt que d'avoir un *client* dédié à cette usage, il suffirait d'envoyer un signal (par exemple *SIGURS1*) à l'*orchestre*.

À la réception de ce signal, le *orchestre* ne doit plus gérer les *clients* (il ne répond plus aux demandes de connexion) et se termine de la même façon que lorsque il recevait l'ordre du *client* dédié.

Attention, la gestion des signaux est délicate, notamment du fait que le signal peut arriver n'importe quand (par exemple au moment où l'*orchestre* reçoit un message).

3.5.4 Rechargement du fichier de configuration (hors projet)

Cette question NE doit PAS être faite (i.e. ne doit pas être présente dans le code rendu). Elle n'est là que pour vous montrer les fonctionnalités d'un "vrai" démon.

Si l'on change le fichier de configuration alors que le *orchestre* tourne, il faut prévenir l'*orchestre* de relire le fichier.

Pour cela on envoie un signal (par exemple *SIGURS2*) à l'*orchestre*.

Attention, la gestion des signaux est délicate, notamment du fait que le signal peut arriver n'importe quand (par exemple au moment où l'*orchestre* accepte un *service* qui change d'état avec le nouveau fichier de configuration).

4 Travail à rendre

Note : il y a un fichier *OREADME* dans le répertoire du code.

Documents à rendre :

- Le code du projet (les 3 fichiers principaux doivent comporter en commentaire vos noms et prénoms, cf. entêtes),
- Notez qu'il a des scripts-shell pour compiler les programmes et non des *Makefile* ; vous pouvez laisser comme cela.
- Un rapport au format pdf, nommé "rapport.pdf" (disons 2 pages hors titre et table des matières) qui contient :
  - l'organisation de votre code : liste des fichiers avec leurs buts,
  - tous les protocoles de communication.
  - et presque le plus important : n'hésitez pas à préciser ce qui ne marche pas correctement dans votre code.

  Soignez l'orthographe, la grammaire, ...

Le fichier "rapport.pdf" doit être directement dans le dossier *PROJET* (racine de votre archive).
Le code du projet sera dans un sous-répertoire nommé *src*.
La hiérarchie des répertoires fournie pour le code, s'il y en a une, doit être conservée (dans *src* donc).
De même les noms des fichiers du répertoire *src* ne doivent pas être changés.
Rappelez-vous que vous avez à disposition un script-shell de vérification, et que les archives ne passant pas avec succès cette vérification seront refusées (i.e. seront considérées hors-sujet car ne respectant pas le cahier des charges).

Tous les retours des appels système doivent être testés. Une assertion fera amplement l'affaire ; privilégiez la bibliothèque *myassert* fournie. N'hésitez pas à faire des wrappers pour rendre le code plus lisible.

Attention, dans l'archive à rendre, ne mettez que les fichiers sources. Tous les autres fichiers (*.o* et autres cochonneries) NE doivent PAS être dans l'archive.

Remarques et/ou rappels très importants :

- indiquez clairement dans le rapport ce qui ne marche pas.
- laissez, dans l'archive, le sous-répertoire *.git* du répertoire *src*. Seule la branche *main* (ou *master*) sera considérée pour la correction.
- un projet n'étant pas validé par le script de vérification sera considéré comme ne répondant pas au cahier des charges.
- une architecture (répertoires, fichiers) est fournie, vous devez la respecter.
- la date de remise est impérative et le dysfonctionnement du site de dépôt ne sera pas une excuse recevable pour un retard [2]. Si vous voulez travailler jusqu'au dernier moment, déposez une première version au moins 24 heures avant ; et vous pourrez l'écraser avec une nouvelle version.
- le projet doit compiler sans warning et fonctionner sur les machines Linux des salles de TP.

Méthodologie de travail :

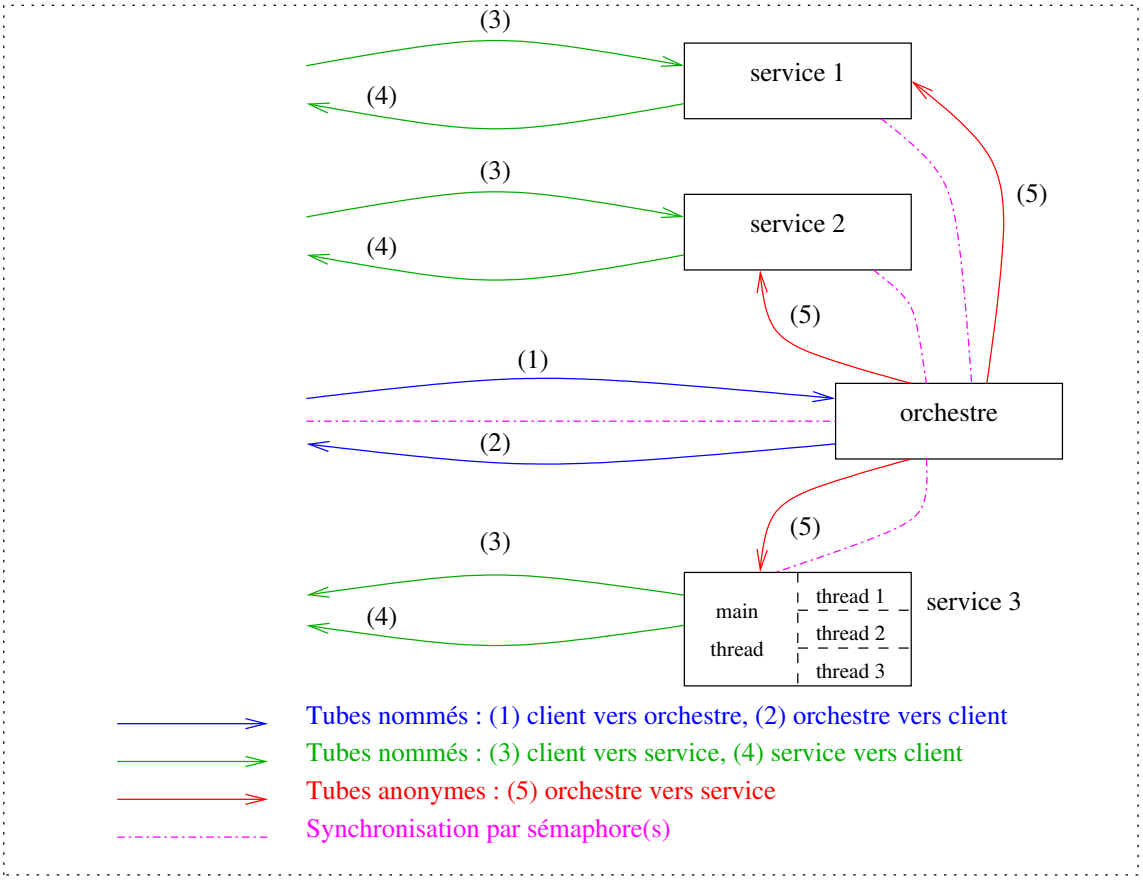
2. sauf s'il dure plus de 24 heures

- Compilez régulièrement pour éviter d'avoir plus de 50 erreurs à la fois.
- Procédez par étapes. Par exemple c'est une très [3] mauvaise idée de programmer tout l'*orchestre* avant d'attaquer les autres programmes. Une bonne idée est par exemple d'implémenter en premier l'envoi du numéro du service par le *client* et la réponse par l'*orchestre* ; puis ajouter l'envoi du mot de passe ; et ainsi de suite.
- Autre façon d'expliquer le point précédent : à chaque nouvelle fonctionnalité (même petite), votre code doit compiler et s'exécuter sans erreur.
  Si vous ne faîte pas ainsi, vous n'arriverez (très vraisemblablement) pas à faire fonctionner votre projet.
- N'hésitez pas à créer plusieurs de fonctions annexes (envoi/réception de mot de passe, envoi du numéro de service, ...) de façon à ce que les codes principaux (i.e. les *main*) des trois programmes soient les plus clairs possibles.
- Lisez bien les codes sources fournis et les fichiers *0README* qui apportent beaucoup d'information.

5 Schémas explicatifs

5.1 L'*orchestre* et les *services* au repos

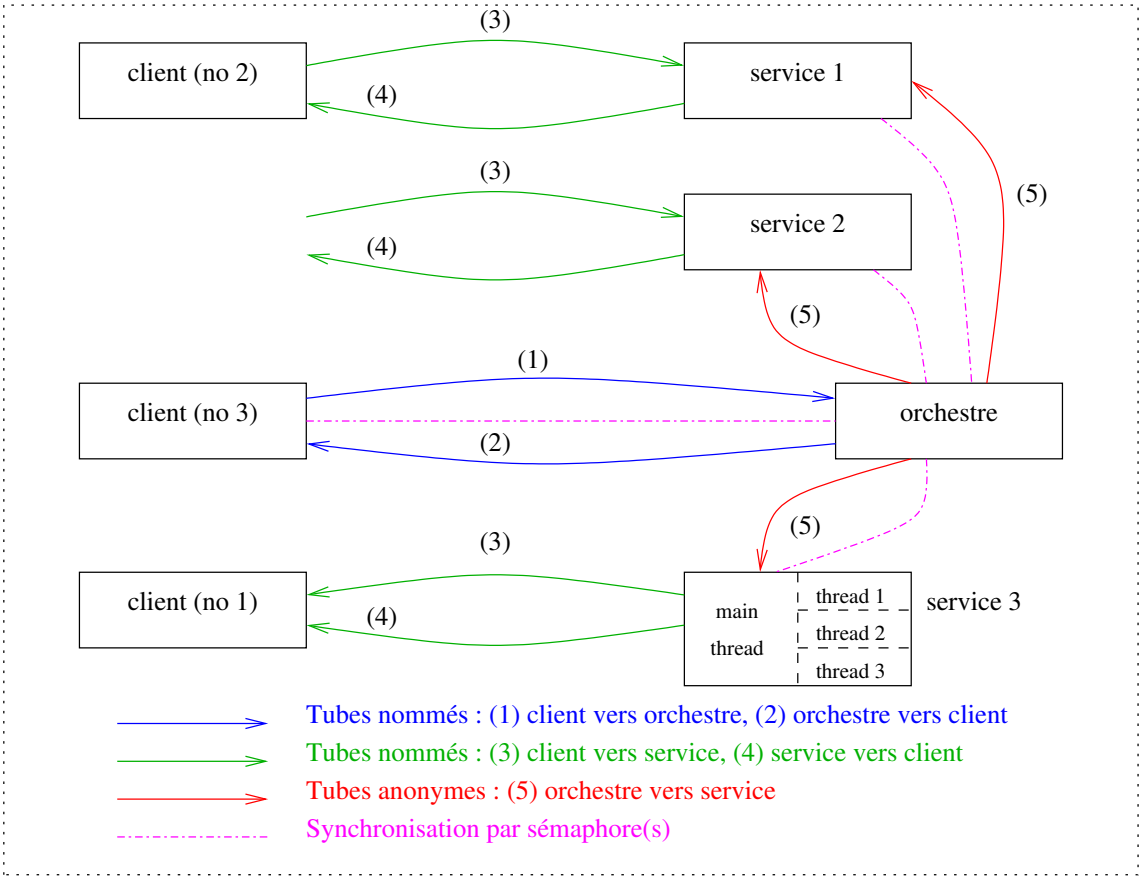
Voici l'état des programmes lorsqu'aucun *client* n'est en action, donc l'*orchestre* et les *services* (que l'on suppose au nombre de trois) tournent mais sont en attente.



5.2 Ensemble des processus avec *clients* en action

Voici l'état des programmes lorsque deux *clients* sont en action et un troisième établit la connexion avec l'*orchestre*.

3. mais vraiment très

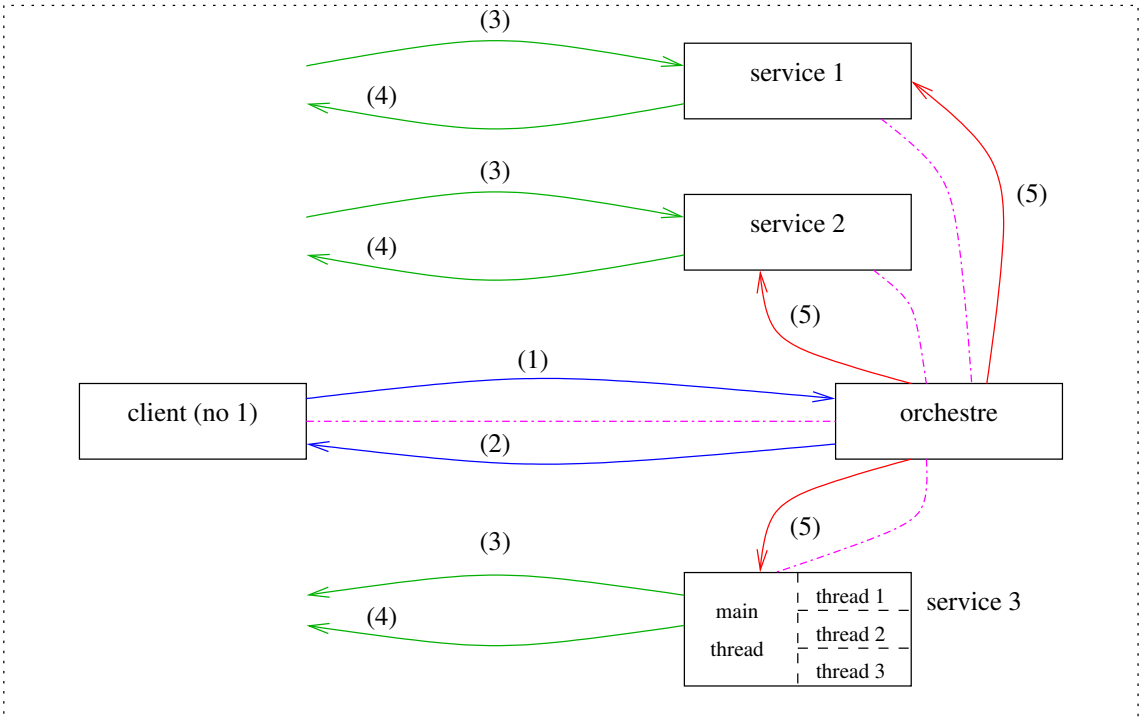


5.3 Succession d'étapes

Voici un enchaînement possible qui mène à l'état précédent.

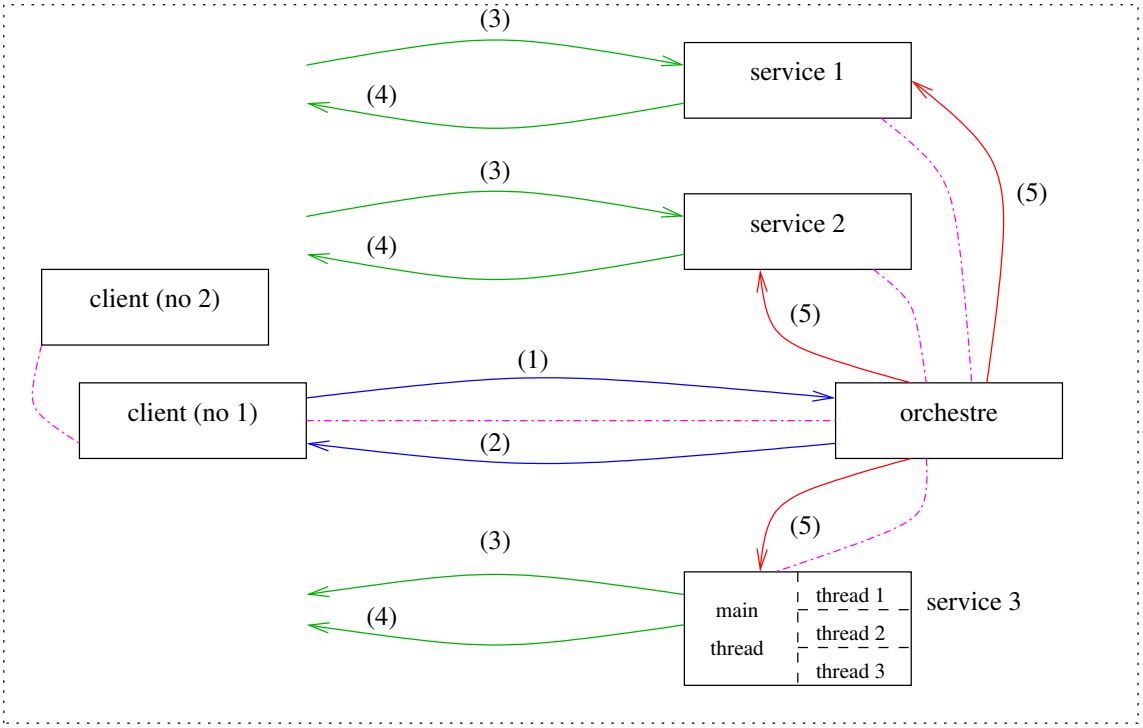
5.3.1 client 1

Le *client* 1 se connecte à l'*orchestre*, pour demander le *service* 3.



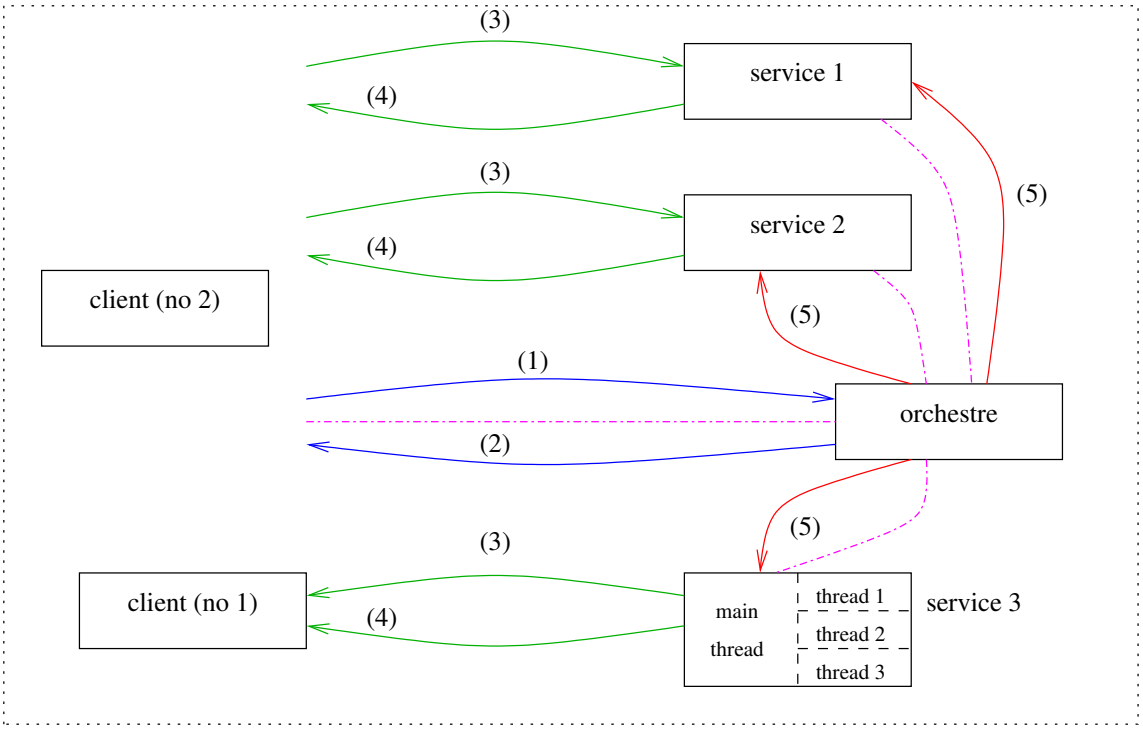
5.3.2 client 2

Le *client* 2 veut se connecter à l'*orchestre* mais doit attendre, via un sémaphore, que le *client* 1 ait fini sa communication avec l'*orchestre*.



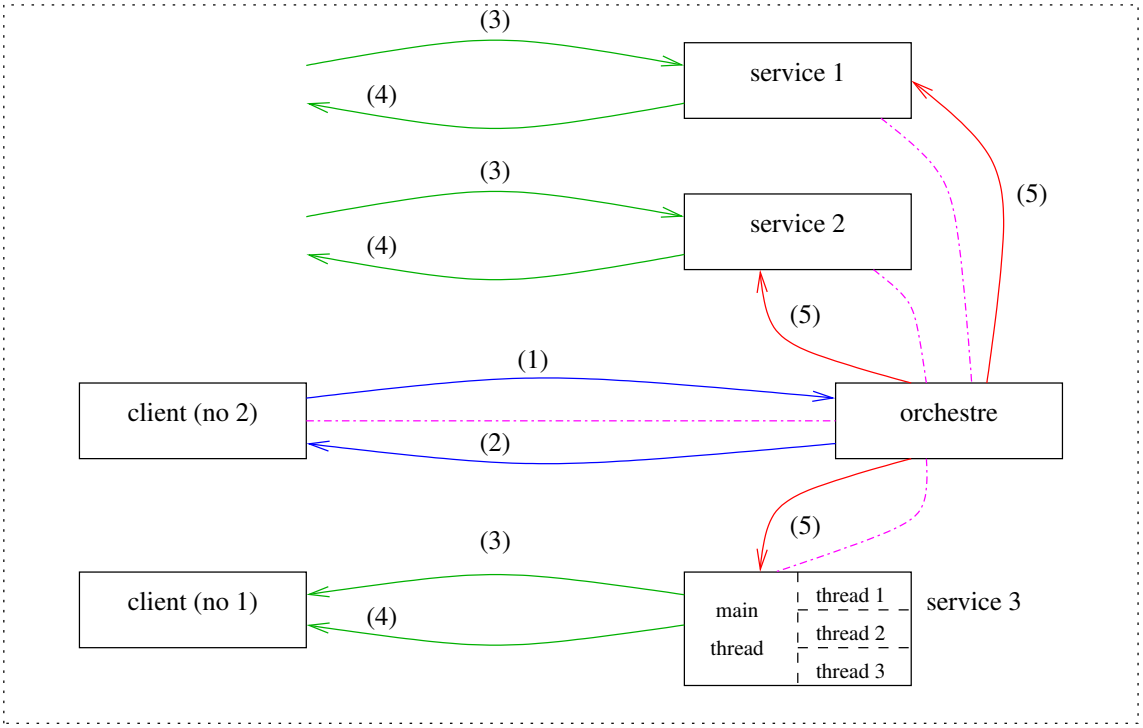
5.3.3 client 1

Le *client* 1 débloque le *client* 2.
Le *client* 1 se connecte au *service* 3.



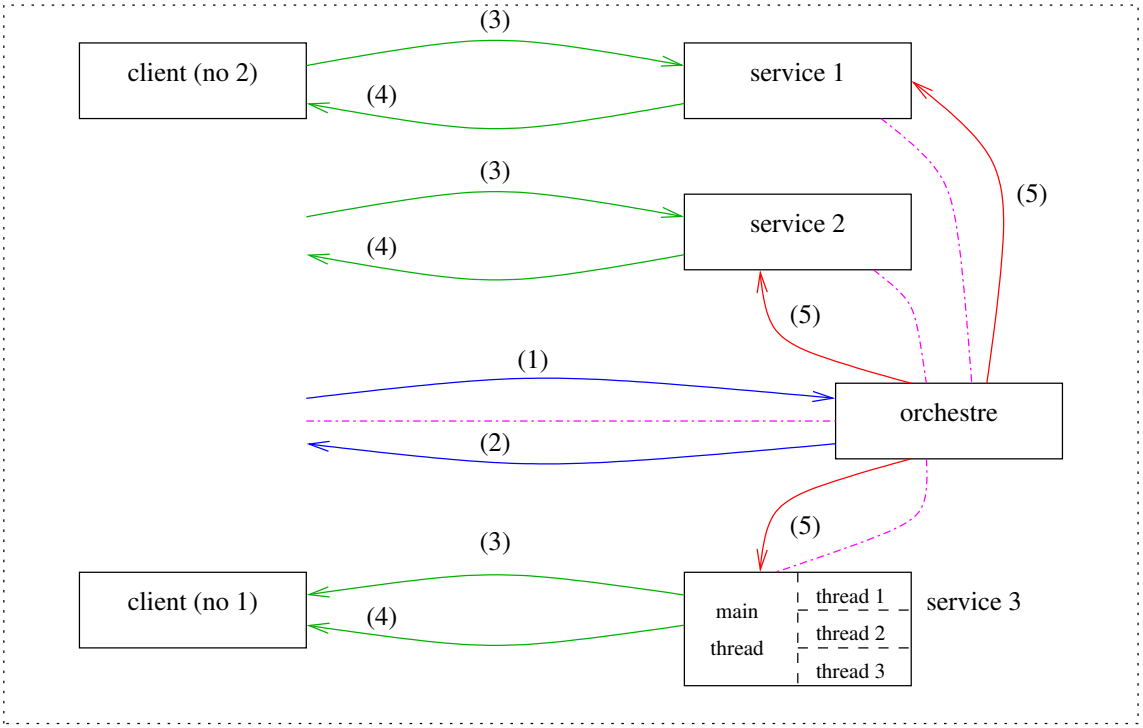
5.3.4 client 2

Le *client* 2 se connecte à l'*orchestre*, pour demander le *service* 1.
Le *client* 1 est toujours connecté au *service* 3.



5.3.5 client 2

Le *client* 2 se connecte au *service* 1.
Le *client* 1 est toujours connecté au *service* 3.



5.3.6 client 3

Le *client* 3 se connecte à l'*orchestre*.
Le *client* 1 est toujours connecté au *service* 3.
Le *client* 2 est toujours connecté au *service* 1.