1 Cahier des charges

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

La date de remise du projet est :
— vendredi 8 décembre 2023 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 *meneveaux_subrenat_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.

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*.

Référez-vous à la section 4 (page 6) 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 *master*, des *workers* et des *clients*.

Le *master* est un programme qui tourne en permanence. Il attend que des *clients* le contactent pour leur rendre le résultat de calculs ou la confirmation d'actions.
Les *clients* sont des programmes indépendants.
Les *workers* sont des programmes indépendants, mais lancés par le *master* ou d'autres *workers*.

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 ; ils se font donc concurrence pour communiquer avec le *master*.

De même, il n'y a qu'un seul code source pour les *workers*, mais ils sont lancés en plusieurs exemplaires, et ils formeront un arbre binaire de recherche (non équilibré pour être précis).

Un *client* s'adresse exclusivement au *master* qui se décharge alors sur l'arbre de *workers* pour les calculs, avant de renvoyer la réponse au *client*.

Le but est de stocker un ensemble de nombres *float* dans un ABR (arbre binaire de recherche non équilibré), chaque noeud de l'arbre est un processus lourd *worker*.

Chaque *worker* gère donc un élément (*float*), avec sa cardinalité, de l'ensemble trié et est connecté à son père (généralement un *worker*, sauf pour la racine où le père est le *master*), aux deux *workers* fils (s'ils existent) ainsi qu'au *master* par seul un canal commun à tous les *workers*.

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

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

Le *master* est exécuté en premier et dans un premier temps ne lance pas de *worker* car l'ensemble est vide au départ.
Lorsque des éléments seront rajoutés :

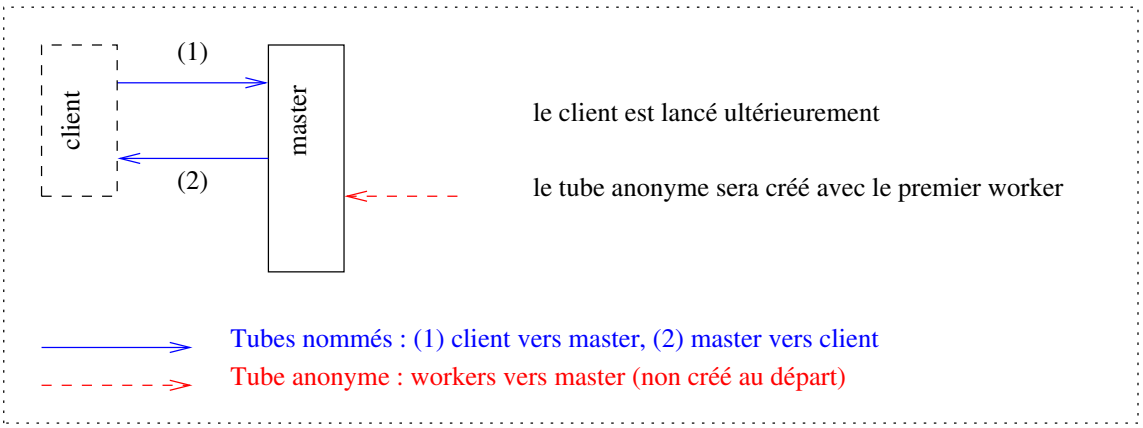
- le *master* ne pourra envoyer des données qu'au premier *worker* (i.e. la racine qui elle enverra des données à ses fils, et ainsi de suite tout au long de l'arbre).
- le *master* pourra, en tant que parent, recevoir des données du premier *worker*.
- le *master* pourra, en tant que *master*, recevoir des résultats de n'importe quel *worker* (via un tube commun, cf. ci-dessous).
- un *worker* peut communiquer bidirectionnellement avec son père (qui peut être le *master*) et ses fils (s'ils existent).
- un *worker* peut envoyer des données au *master* mais pas en recevoir de lui.

Les communications entre *workers* et *workers*, et entre *master* et *workers* se font aux moyens de tubes anonymes.

Un *client* et le *master* communiquent :

- par une paire de tubes nommés (*mkfifo*, *open*, ...) pour échanger données et résultats,
- deux sémaphores IPC (*semget*, *semop*, ...) pour se synchroniser.

Voici le schéma au début de la vie du *master* :



Le *client* envoie une demande de calcul ou d'action au *master* sur un tube nommé (1) et attend la réponse sur l'autre tube nommé (2).

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, ...*).

3.2 *client*

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

Voici le déroulement d'un *client* :

— Le *client* envoie une demande au *master* (sur le premier tube). Il y a plusieurs ordres possibles :
  1. arrêt du *master*
  2. demander le nombre d'éléments et le nombre d'éléments distincts
  3. demander le minimum de l'ensemble
  4. demander le maximum de l'ensemble
  5. tester l'existence d'un élément, avec sa cardinalité
  6. demander la somme des éléments
  7. ajouter un élément à l'ensemble
  8. ajouter plusieurs éléments aléatoires à l'ensemble
  9. demander aux *workers* d'afficher leurs éléments (dans l'ordre croissant)

  Dans le cas des ordres 5, 7 et 8, le *client* envoie des informations supplémentaires.
— Le *client* attend et reçoit la réponse du *master* sur le deuxième tube ; dans tous les cas il reçoit un accusé de réception (ack) et éventuellement il reçoit des informations supplémentaires.
— Le *client* débloque le *master* grâce à un sémaphore ; en effet, le *master*, avant de s'occuper d'une nouvelle demande, doit s'assurer que le *client* a complètement terminé.

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 (i.e. tous les points sauf le dernier) 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 a été préalablement créé par le *master*.

De même, le sémaphore de synchronisation en fin de *client* a été créé et initialisé par le *master*.

Dans le code fourni (fichier *client.c*), une proposition d'organisation de l'implémentation est détaillée.

3.3 *client* (bis)

Cette partie est complètement indépendante du reste du projet.

On rajoute une fonctionnalité au *client* : calculer le nombre de fois qu'un élément est présent dans un tableau ; une même valeur peut donc être présente en plusieurs exemplaires.
Le calcul est complètement local (i.e. sans le *master*) et en mode multi-thread.

Voici l'algorithme demandé :
— soient T un tableau de N cases (généré aléatoirement) et E l'élément à compter.
— on lance K (nombre choisi par l'utilisateur) threads.
— chaque thread est associée à une portion du tableau disjointe des autres et fait l'algo suivant :
  . il compte le nombre de fois où l'élément est présent dans sa portion
  . il met à jour un compteur partagé par tous les threads
— il ne faut pas que plusieurs threads modifient le compteur en même temps : l'accès au compteur sera en section critique gérée par un mutex de type *ptread_mutex_t*.
— il ne faut pas de variables globales.
— lorsque tous les threads ont fini, la fonction *main* affiche le résutat.

3.4 *worker*

Bien que décrit ici, il est préférable d'implémenter le code du *worker* une fois les codes du *client* et du *master* fonctionnels.

Le premier *worker* est issu d'un *fork/exec* du *master*. La création de ce *worker* n'est effectuée que lorsqu'un *client* demande l'insertion du premier élément dans l'ensemble. Ensuite les *workers* se créent en cascade en formant un ABR. À part le premier, les *workers* sont créés par d'autres *worker*.

Un *worker* est responsable d'un seul élément (un *float*) de l'ensemble, mais avec sa cardinalité (i.e. le nombre de fois où l'élément est présent dans l'ensemble). Bref il y aura autant de *workers* s'exécutant que d'éléments distincts dans l'ensemble.

En résumé le rôle du *worker* est le suivant :

— il reçoit un ordre via un tube anonyme provenant du père
— selon l'ordre et l'élément dont il a la charge, il peut :
  . transmettre l'ordre à un de ses fils (et il a fini son traitement)
  . transmettre l'ordre à ses fils et attendre leurs réponses
  . donner la réponse à son père
  . donner la réponse au *master*.
— il recommence au premier point

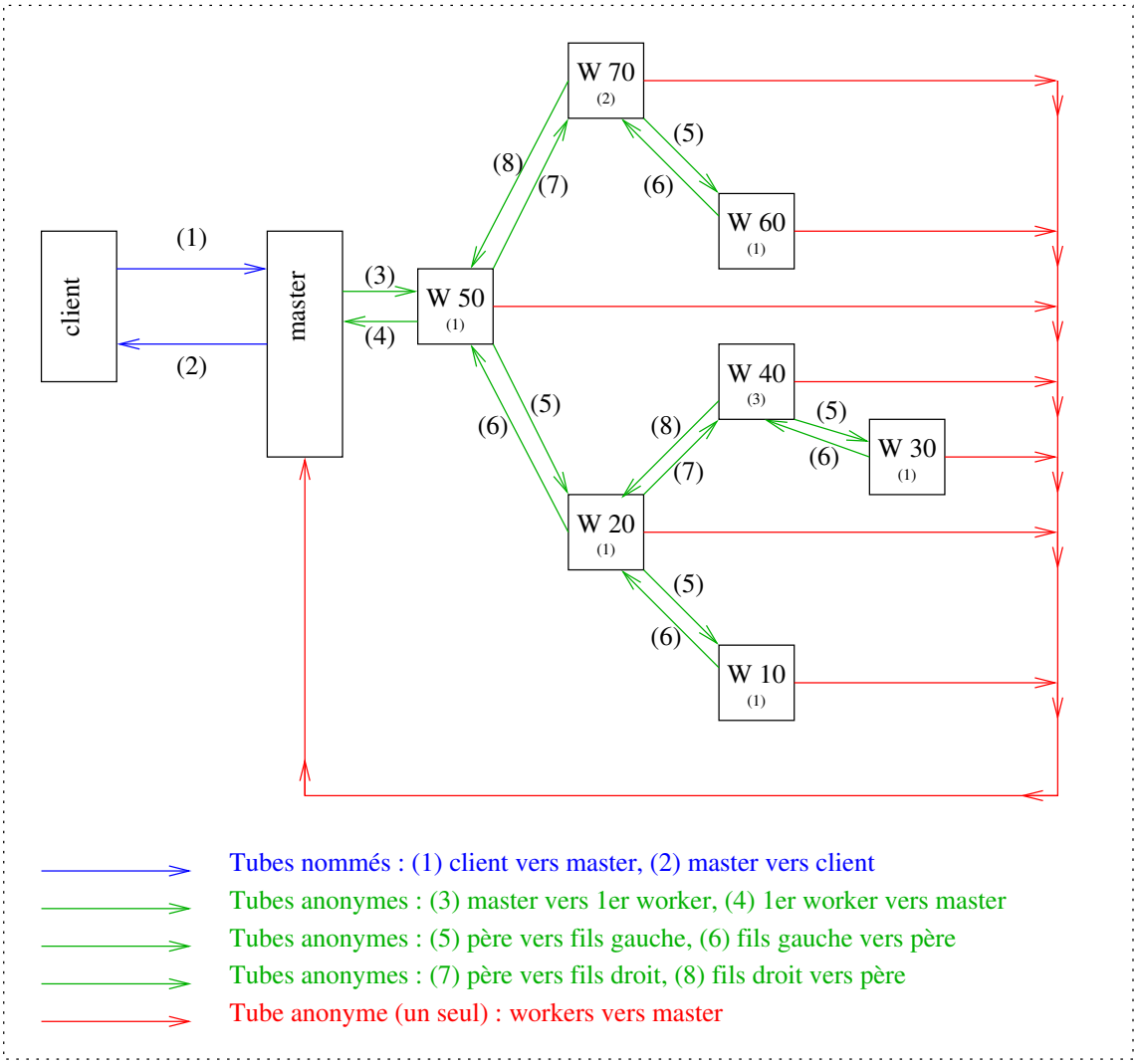
Plus précisément, un *worker* effectue les opérations suivantes :
— c'est un processus indépendant lancé par un *fork/exec*, soit du *master* pour le premier *worker*, soit par le *worker* parent.
— il reçoit en ligne de commande (i.e. *argv*) l'élément dont il a la charge, ainsi que les *file descriptors* pour communiquer avec son père et le *master*.
  Note : lorsqu'un *worker* vient d'être créé il est par définition une feuille de l'arbre.
— il prévient le *master* qu'il vient d'être créé.
— répéter à l'infini
  — attendre un ordre du *worker* parent [1]
  — si c'est un ordre d'arrêt :
    — transmettre cet ordre aux *workers* suivants (s'ils existent)
    — attendre la fin de ces derniers (s'ils existent)
    — sortir de la boucle
  — sinon c'est ordre classique à traiter, et il y a 4 cas :
    a. on a directement la réponse et il faut la transmettre au père (c'est un cas particulier de *d*).
    b. on a directement la réponse et il faut la transmettre au *master*.
    c. on transmet l'ordre à un des fils (gauche ou droite) et on ne s'occupe plus de rien (c'est la descendance qui s'occupera de donner le résultat au *master*).
    d. on transmet l'ordre à son fils gauche et on attend sa réponse. On fait de même avec le fils droit. On fait un calcul et on envoie le résultat final au père.
— se terminer proprement

Voici le principe de création d'un nouveau *worker* :
— création de deux tubes anonymes
— *fork/exec* en passant, en ligne de commande, au nouveau *worker* : l'élément dont il a la charge, le file descriptor pour recevoir des données du *worker* courant, le file descriptor pour envoyer des données au *worker* courant, le file descriptor du tube vers le *master*.
— on en déduit qu'il n'y a qu'un seul tube anonyme vers le *master* et que tous les *workers* écrivent dedans ; mais les algorithmes sont faits de telle sorte qu'un seul *worker* à la fois a la possibilité d'écrire, et donc ce n'est pas un problème.

Prenons l'exemple de l'insertion des éléments suivants : 50, 70, 20, 40, 70, 40, 60, 30, 10, 40 :
— 50 : création du *worker* W50
— 70 : création du *worker* W70 à droite de W50
— 20 : création du *worker* W20 à gauche de W50
— 40 : création du *worker* W40 à droite de W20
— 70 : incrémentation de la cardinalité de W70 (passe à 2)
— 40 : incrémentation de la cardinalité de W40 (passe à 2)
— 60 : création du *worker* W60 à gauche de W70
— 30 : création du *worker* 30 à gauche de W40
— 10 : création du *worker* 10 à gauche de W20
— 40 : incrémentation de la cardinalité de W40 (passe à 3)

Le schéma complet et général de communication est alors le suivant :

1. ou du *master* dans le cas du premier *worker* : nous ne préciserons désormais plus ce cas particulier, d'autant plus qu'un *worker* n'a aucune idée de qui lui envoie des informations.



Et donc les étages se rajoutent au fur et à mesure. Par exemple :

- insertion de 12 : W10 créerait un fils droit
- insertion de 65 : W60 créerait un fils droit
- insertion de 24 : W30 créerait un fils gauche
- ...

Dans le code fourni (fichier *worker.c*), une proposition d'organisation de l'implémentation est détaillée.

3.5 *master*

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

- attendre la connexion d'un *client*
- analyser la demande du *client*
- éventuellement faire intervenir les *workers*
- renvoyer le résultat au *client*

De manière plus précise ;

- initialisations diverses dont création des tubes nommés et des deux sémaphores
- boucle infinie de travail :
  - ouvrir les tubes de communication avec le futur *client*
  - attendre l'envoi d'un ordre d'un *client*
  - si ordre de fin
    - envoi ordre de fin au premier *worker*
    - attendre la fin du premier *worker*
    - envoyer accusé de réception au *client*

        - sortir de la boucle
    - sinon si le premier *worker* n'existe pas
        - si c'est une insertion, lancement du premier *worker* avec les trois tubes anonymes
        - sinon transmettre le résultat au *client*
    - sinon
        - envoi de l'ordre au premier *worker* pour qu'il effectue le travail
        - attendre la réponse (soit du premier *worker*, soit d'un des des *workers*)
        - transmettre le résultat au *client*
    - fermer les tubes nommés de communication
    - attendre le déblocage du *client*
- fin répéter
- libération des ressources
- destruction des tubes

Dans le code fourni (fichier *master.c*), une proposition d'organisation de l'implémentation est détaillée.

4 Travail à rendre

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

Documents à rendre :

- Le code du projet (chaque fichier créé doit comporter en commentaire vos noms et prénoms),
- Notez qu'il y a un *Makefile* pour compiler le projet. Si vous avez ajouté des fichiers, il faudra le compléter. Votre projet doit compiler avec le *Makefile* fourni.
- 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.

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.

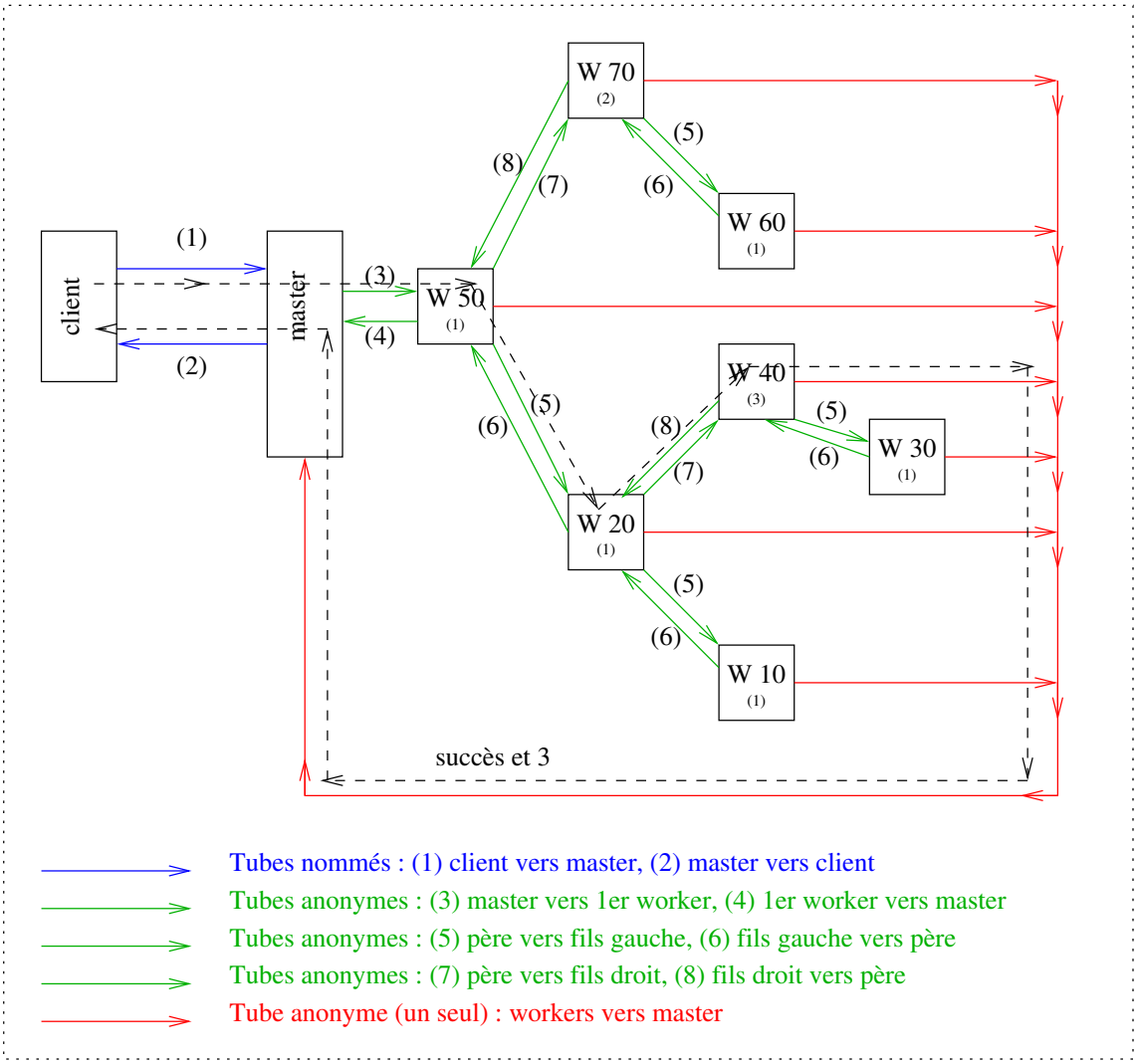
2. sauf s'il dure plus de 24 heures

5 Schémas explicatifs

5.1 Cheminement : existence de l'élément 40

Cet ordre transite le long de l'ABR (en se servant de ses propriétés) :

- W50 : envoyé par le master
- W20 : car 40 est inférieur à 50, donc on va à gauche
- W40 : car 40 est supérieur à 20, donc on va à droite ; on a trouvé et on envoie un succès au master avec la cardinalité qui est 3



5.2 Cheminement : existence de l'élément 42

On a exactement le même cheminement que pour 40. Les différences sont :

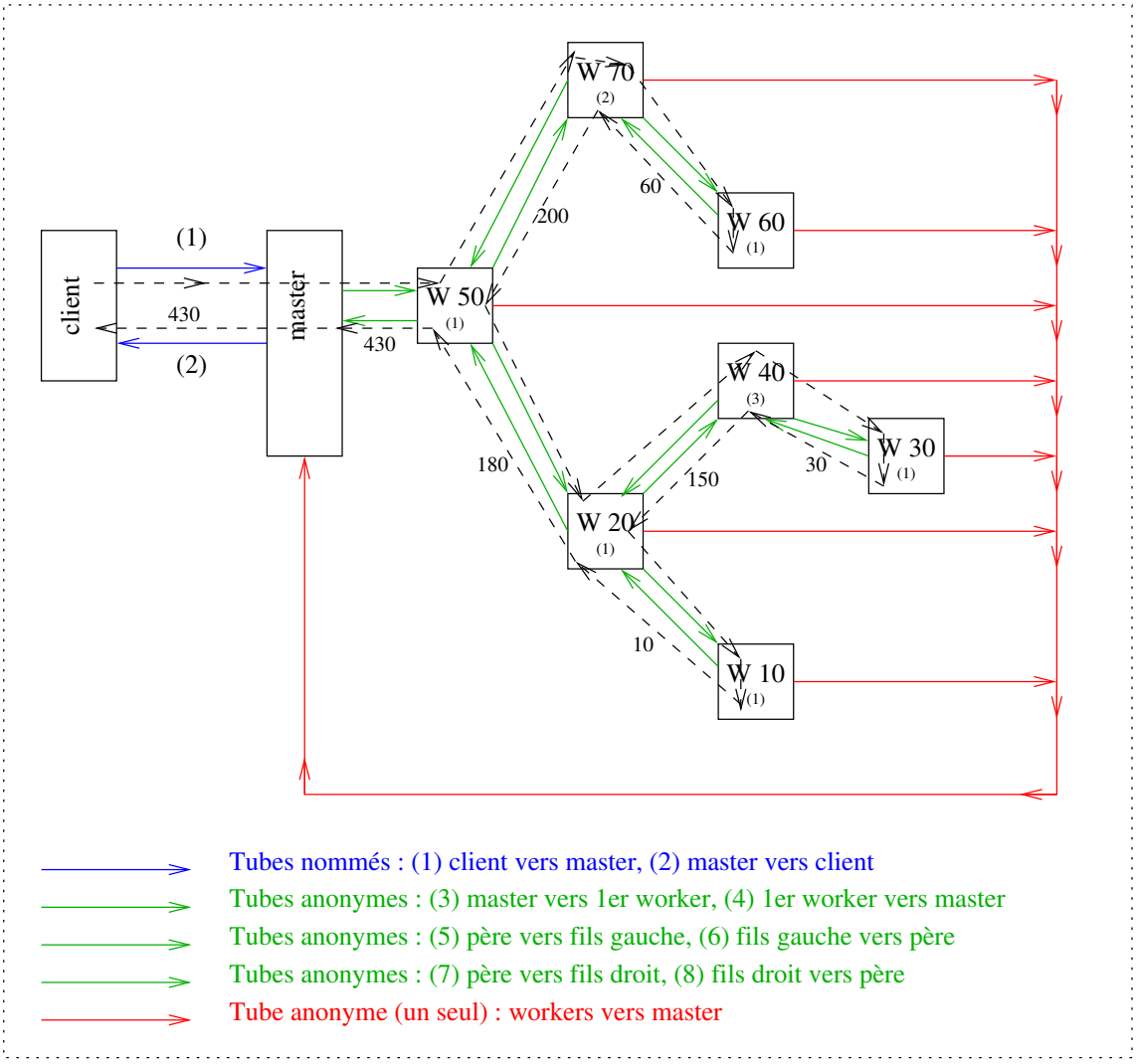
- lorsque W40 reçoit le test d'existence de 42 il voudrait le transmettre à son fils droit
- le fils droit n'existant pas, il déduit que 42 n'est pas présent dans l'arbre
- il envoie au *master* un résultat d'échec

Tournez SVP ...

5.3 Cheminement : somme des éléments

Il s'agit d'un parcours complet en profondeur, chaque élément renvoyant à son père la somme de son sous-arbre. On obtient au final le résultat 430.

Dans le dessin il s'agit d'un parcours à droite d'abord (ce qui correct mais peu habituel) uniquement pour éviter que les flèches se croisent.



Tournez SVP ...

5.4 Cheminement : insertion de 64

Comme on n'équilibre pas l'arbre, on fait le même parcours que pour la recherche. Soit l'élément est présent et il suffit d'incrémenter son compteur, soit il est absent (c'est le cas pour 64) et on ajoute un *worker* à l'endroit où on a détecté son absence (à droite de 60 dans notre cas).