IFT2245 Rapport TP2

Charlie Gauthier
Maud Moerel-Martini

1 avril 2019

Pour les futurs références nous prendrons la couleur bleue pour le client et le rouge pour le serveur.

Lire et comprendre cette donnée :

cf. Annexe schéma 1

Les sockets du serveur et du client s'ouvrent sur le port indiqué en comme premier argument en ligne de commande des deux main.c. On comprends que la logique du code suit ce schéma. Le serveur et le client commencent. Le serveur ouvre un socket, puis attend le client.

a. Le client fait BEGIN et CONF. CONF envoie le nombre de ressources indiquées dans les arguments en ligne de commande du client/main.c.
b. Le serveur reçoit ça. Il prend les ressources indiquées par CONF comme étant les maximums de ressources partagées entre les clients. Puis, il renvoie ACK pour indiquer que c'est validé. Après, il lance autant de threads qu'indiqué selon l'argument en ligne de commande de server/main.c.
c. Les petits clients se font créer. Ils envoient leur INIT.
d. Boucle principale :
    i. Les threads serveur reçoivent ça. Ils prennent les ressources indiquées par INIT comme étant les maximums de ressources du client qui a envoyé le INIT. Ils renvoient ACK pour indiquer que c'est validé.
    ii. Les petits clients entrent dans leur boucle principale. Ils envoient autant de REQ qu'indiqué dans les arguments en ligne de commande du client/main.c. Ces REQ sont soit négatifs ou positifs. Si positifs ils vont de 0 à leur maximum de ressources. Si négatifs, ils vont de 0 à leur nombre de ressources utilisées.
    (1) Si c'est la dernière REQ, ils libèrent toutes leurs ressources.
    iii. Les threads serveur traitent ces REQ avec l'algorithme du banquier. Si la commande ne va pas créer d'interblocage, ils renvoient ACK. Sinon, ils envoient WAIT. Si la commande est tout simplement invalide, ils envoient ERR.
    iv. Les petits clients reçoivent ces codes de réponses. Si c'est WAIT, ils attendent le nombre de secondes indiquées par le WAIT et renvoient la REQ. Si c'est ACK, ils se souviennent qu'ils utilisent maintenant (leur ancienne utilisation += la requête) ressources. Si c'est ERR, ils passent à la prochaine requête.
e. Lorsque les petits clients ont terminé d'envoyer toutes leurs REQ (pas nécessairement en même temps), ils envoient CLO
f. Le serveur renvoie ACK si le client qui CLO n'a pas de ressources utilisées et ERR sinon.
g. Lorsque un petit client a CLO, il pthread_exit. Lorsque tous les petits clients sont sortis, le client parent envoie END.
h. Le serveur reçoit ce END et envoie un ACK si tous les petits clients ont été fermés avec un CLO. Sinon, il envoie ERR.

Parties importantes du code fourni (voir correspondances aux points en 1) :

cf. Annexe schéma 2

a. Ceci est avant le ct_code, ou pendant et on s'assure qu'une seule thread envoie cela. Pas de fonction pré-faite pour ceci.

b. St_init : on y lit BEGIN et CONF et on configure les ressources max du programme. On retourne ACK si tout s'est bien passé.

c. Ct_code : c'est dans cette fonction qu'on a la boucle principale des clients, mais avant de commencer cette boucle, il faut faire les INIT. On les envoie au serveur et on écoute la réponse.

d. Boucle principale :

   i. St_code : c'est dans cette fonction qu'on a la boucle principale des serveurs. St_code ne fait que boucler tant que le serveur peut encore recevoir des connections. Quand on détecte qu'un client se connecte au grand socket principale du serveur (server_socket_fd), on "dispatch" le traitement de la connection à st_process_requests. Dans st_process_requests, on retrouve le code pour le traitement de : INIT, REQ, CLO, END. Si on reçoit d'autres commandes, on retourne une erreur.

   ii. Les petits clients entrent dans la partie boucle de ct_code (boucle principale des clients). Ils envoient autant de REQ qu'indiqué dans les arguments en ligne de commande du clientmain.c. On crée les REQ (positives ou négatives comme mentionné à 1.d.ii) dans ct_code, puis on "dispatch" le traitement de l'envoie à send_request. (1) Si c'est la dernière REQ, ils libèrent toutes leurs ressources

   iii. St_process_requests traite ces REQ avec l'algorithme du banquier. Si la commande ne va pas créer d'interblocage, on envoie ACK. Sinon, ils envoient WAIT. Si la commande est tout simplement invalide, ils envoient ERR.

   iv. Les petits clients reçoivent ces codes de réponses dans send_request ou dans ct_code. Si c'est WAIT, ils attendent le nombre de secondes indiquées par le WAIT et renvoient la REQ. Si c'est ACK, ils se souviennent qu'ils utilisent maintenant (leur ancienne utilisation += la requête) ressources. Si c'est ERR, ils passent à la prochaine requête.

e. Ct_end : lorsque les petits clients ont terminé d'envoyer toutes leurs REQ (pas nécessairement en même temps), ils envoient CLO dans ct_end.

f. Encore dans st_code et st_process_requests, le serveur reçoit les CLO et les traire comme décrit dans 1.f.

g. Fin de ct_end : on exit le thread client. Le client main attend que tous les clients aient exit, puis entre dans ct_wait_server. Il envoie END au server.

h. Encore dans st_code et st_process_requests, le serveur reçoit le END et envoie un ACK si tous les petits clients ont été fermés avec un CLO. Sinon, il envoie ERR.

i. Lorsque le client main reçoit ce ACK, il sait que le serveur s'est fermé. Il peut donc se fermer lui-même.

Compléter le code fourni :

cf. Annexe schéma 3

Points généraux :
Client et serveur : on utilise __atomic_add_fetch pour incrémenter les counter partagés utilisés dans le journal (documentation) . On s'est aussi inspiré de cette réponse SO pour le memory model à utiliser. Ces opérations atomiques sont beaucoup plus simples à gérer que d'utiliser un mutex pour chaque variable.
Client :

- On a modifié la struct client_thread : on a ajouté les int* max_ressources et used_ressources. Ces tableaux indiquent la quantité de ressources maximales et utilisées, respectivement, des petits clients. Ils sont initialisés aux bonnes valeurs dans ct_create_and_start et dans le début de ct_code.

- On a ajouté un mutex "random_mutex", qui protège les appels à random() car random n'est pas thread safe : il se re-seed avec la valeur de retour pour le prochain appel. Donc, si on ne le protégeait pas, on aurait souvent les mêmes valeurs dans différentes REQ si plusieurs clients font l'appel a random() simultanément

Serveur :

- On a créé une nouvelle struct, client. Client comprends simplement m_ressources et u_ressources, les ressources maximales et utilisées du client représenté. On a aussi créé des fonctions d'opérations sur les client.

    ◦ Create_clients crée (malloc) une variable globale clients**, un tableau de client*. On assume que les ID des clients reçus dans le serveur on été générés séquentiellement dans le client. Autrement dit, on assume que si on reçoit un client d'ID 3000, il y aura AUSSI des clients d'ID de 0 à 3000. Comme ca, on peut facilement générer un tableau pour les clients dont l'index est l'ID : si on veut process un REQ du client 3, on a qu'a accéder à clients[3] et on reçoit le client du client 3. Ceci nous évite de chercher le client dans une liste chaînée ou dans un tableau à chaque fois qu'on veut accéder au client X. De plus, c'est logique, puisque la façon la plus simple de faire des ID uniques est simplement d'incrémenter un counter dans clientmain.c.

    ◦ Resize_clients(int new_max_index) s'assure que new_max_index est un accès valide selon clients[new_max_index]. Si ce ne l'est pas, on agrandit clients.
    Note : pour les deux points précédents, les client non-init ou fermés sont NULL.

    ◦ New_client(id, ressources) crée un client à clients[id]. On appelle resize_clients avant pour s'assurer que c'est valide.

    ◦ Close_client(index) ferme le client à clients[index] en libérant la mémoire prise par ses int* et par le client lui-même. Puis, on assigne NULL à clients[index] pour indiquer que le client a été fermé.

- On a crée les fonctions pour l'algorithme du banquier.

    ◦ Lock_bankers verrouille les clients et l'algorithme du banquier (voir le point sur les mutex plus bas). Puis, on retourne l'appel à call_bankers.

    ◦ Call_bankers gère les états hypothétiques. On y change available et le u_ressources du client concerné pour faire comme si la REQ avait été acceptée. Puis, on appelle bankers. Si bankers dit que l'état est "safe", on laisse les changements comme ça. Sinon, on les défait. Dans les deux cas, on retourne le code retourné par bankers.

    ◦ Bankers applique l'algorithme du banquier sur un état de available et des clients.

- On a ajouté deux mutex, client_mutex et bankers_mutex.

    ◦ Client_mutex contrôle l'accès aux clients (dans les fonctions d'opérations sur les clients et dans bankers) pour empêcher les conditions de course.

    ◦ Bankers_mutex contrôle l'accès à bankers, ce qui s'assure que l'évaluation des REQ se fasse séquentiellement. Sans cela, on commencerait l'analyse (2) d'une REQ pendant qu'une autre se fait analyser (1) et donc l'état hypothétique de 1 contaminerait l'analyse 2.

Common :

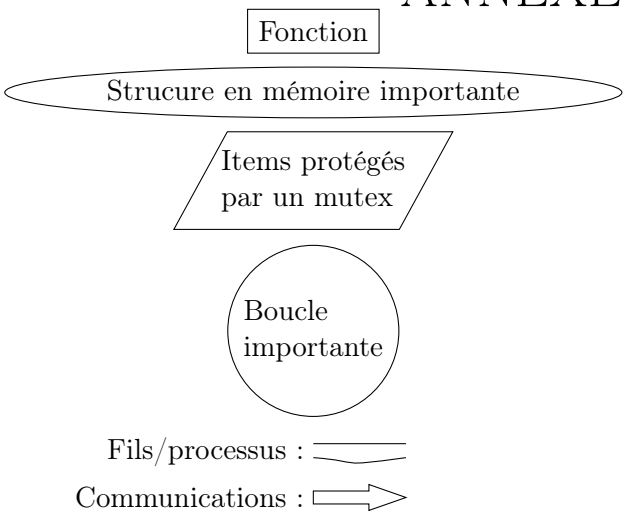
- Le read_socket fourni avec le TP a été modifié en op_socket_code et fait maintenant la lecture et l'écriture sur les sockets.
- On a créé err_h_socket qui fait le "error handling" pour l'appel de op_socket_code : si on y a écritlu moins que la grosseur de buffer demandée, on traite l'erreur.
- On a créé read_socket et write_socket, qui appellent err_h_socket avec les bons arguments pour la lecture et l'écriture, respectivement.

- On a créé read_compound et write_compound, qui appellent read_socket et write_socket, respectivement, de façon à faire la transation sur le socket d'une tête de message suivi d'un corps de message.

a. On a créé init_server dans clientmain.c. On y envoie et traite la réponse de BEGIN et CONF.
b. St_init : on y malloc available et on y place le nombre de chaque ressources indiquées par CONF. On y appelle create_clients.
c. Ct_code : avec les INIT on malloc used_ressources et max_ressources qui prennent les valeurs maximales et utilisées du client.
d. Boucle principale :
    i. St_code n'a presque pas été changé. St_process_requests a beaucoup changé. On y a placé un grand switch case qui traite les codes des requests. Si on a autre chose que INIT, REQ, CLO, END, on lance une erreur. Sinon :
        i. INIT : on appelle new_client et on y crée la représentation du client et de ses ressources max.
        ii. REQ : on appelle lock_bankers et on retourne un message approprié au client
        iii. CLO : on vérifie que le client a bien free ses ressources (on envoie ERR sinon), puis appelle close_client et on envoie ACK.
        iv. END : on vérifie que tous les clients ont été fermés, on envoie ACK si oui (ERR sinon), puis on ferme le serveur.
    ii. On génère les REQ comme décrit plus haut. Send_request fait le traitement du code de retour :
        i. ACK : on change le used_ressources du thread client (qui nous est passé en paramètre).
        ii. WAIT : on attend le nombre de secondes indiqué.
        iii. ERR : on ne fait rien Puis, ct_code rappelle la même REQ tant qu'on a pas autre chose que WAIT.
    iii. textcolorredVoir 3.d.i
    iv. textcolorblueVoir 3.d.ii
e. Ct_end : voir 2.e. Si on reçoit ERR, on exit.
f. Voir 3.d.i
g. Dans ct_wait_server : on busy wait en attendant que tous les clients soient fermés. Lorsque c'est fait, on sait que le serveur a reçu les CLO de tous les clients, et on envoie END.
h. Encore dans st_code et st_process_requests, le serveur reçoit le END et envoie un ACK si tous les petits clients ont été fermés avec un CLO. Sinon, il envoie ERR.
i. Lorsque le client main reçoit ce ACK dans ct_wait_server, il sait que le serveur s'est fermé. Il peut donc se fermer lui-même.

ANNEXE

Légende



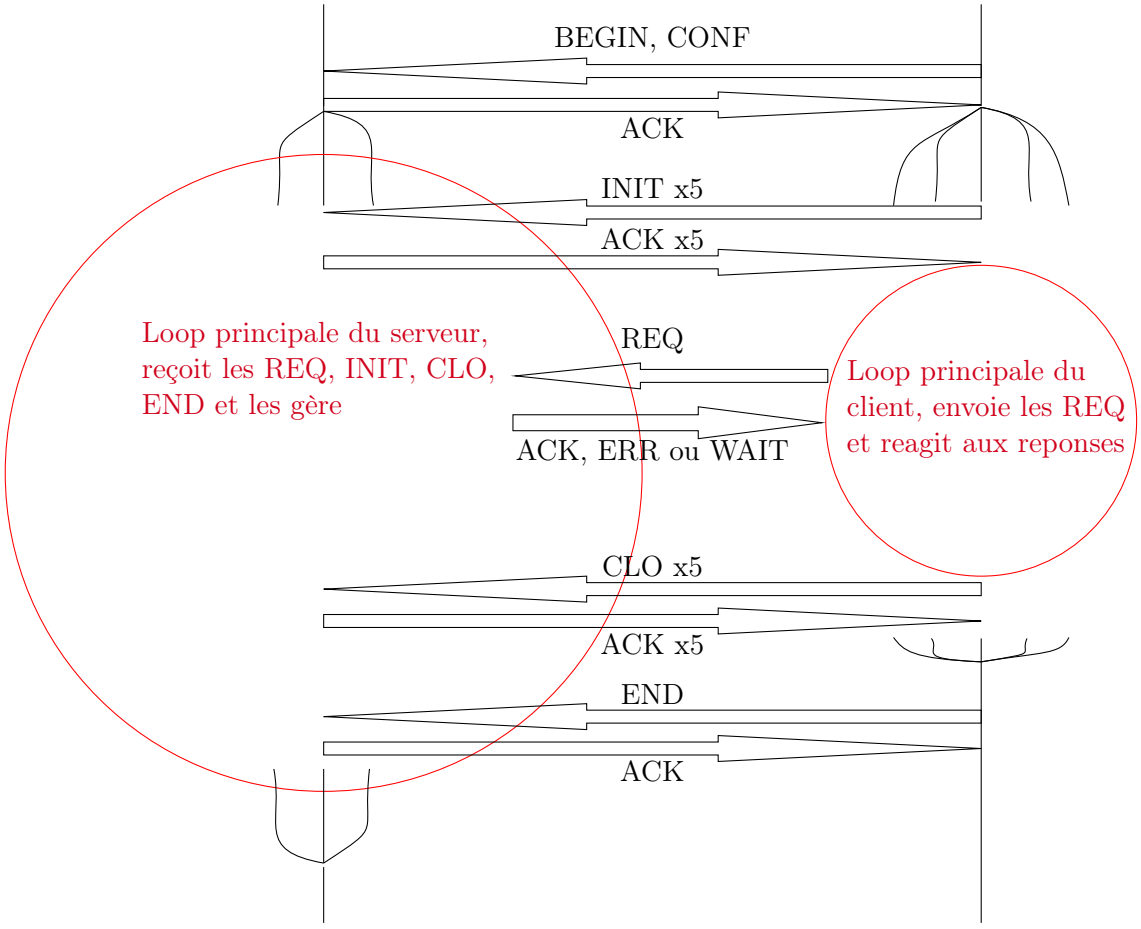


Schéma 1

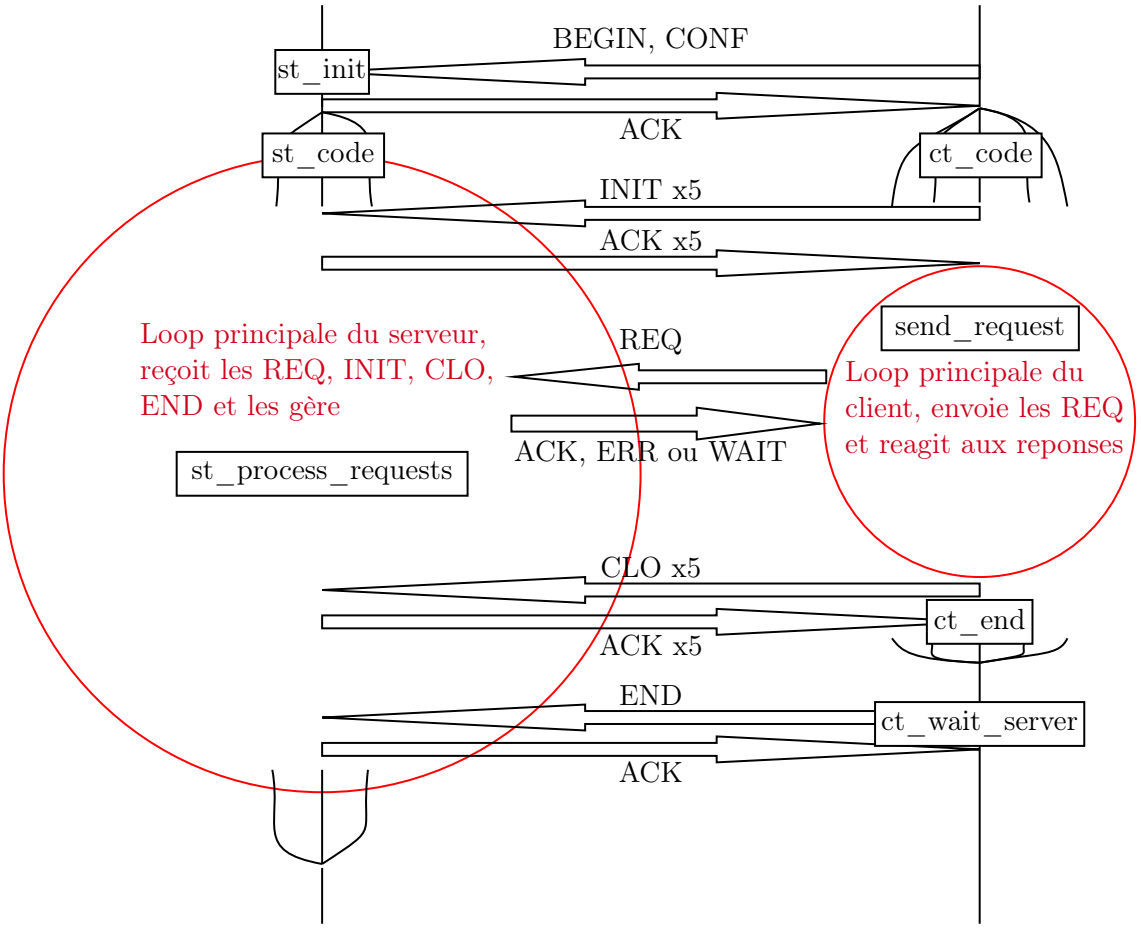

BEGIN, CONF
st_init
ACK
st_code
ct_code
INIT x5
ACK x5
Loop principale du serveur, reçoit les REQ, INIT, CLO, END et les gère
REQ
send_request
Loop principale du client, envoie les REQ et reagit aux reponses
ACK, ERR ou WAIT
st_process_requests
CLO x5
ct_end
ACK x5
END
ct_wait_server
ACK

Schéma 2

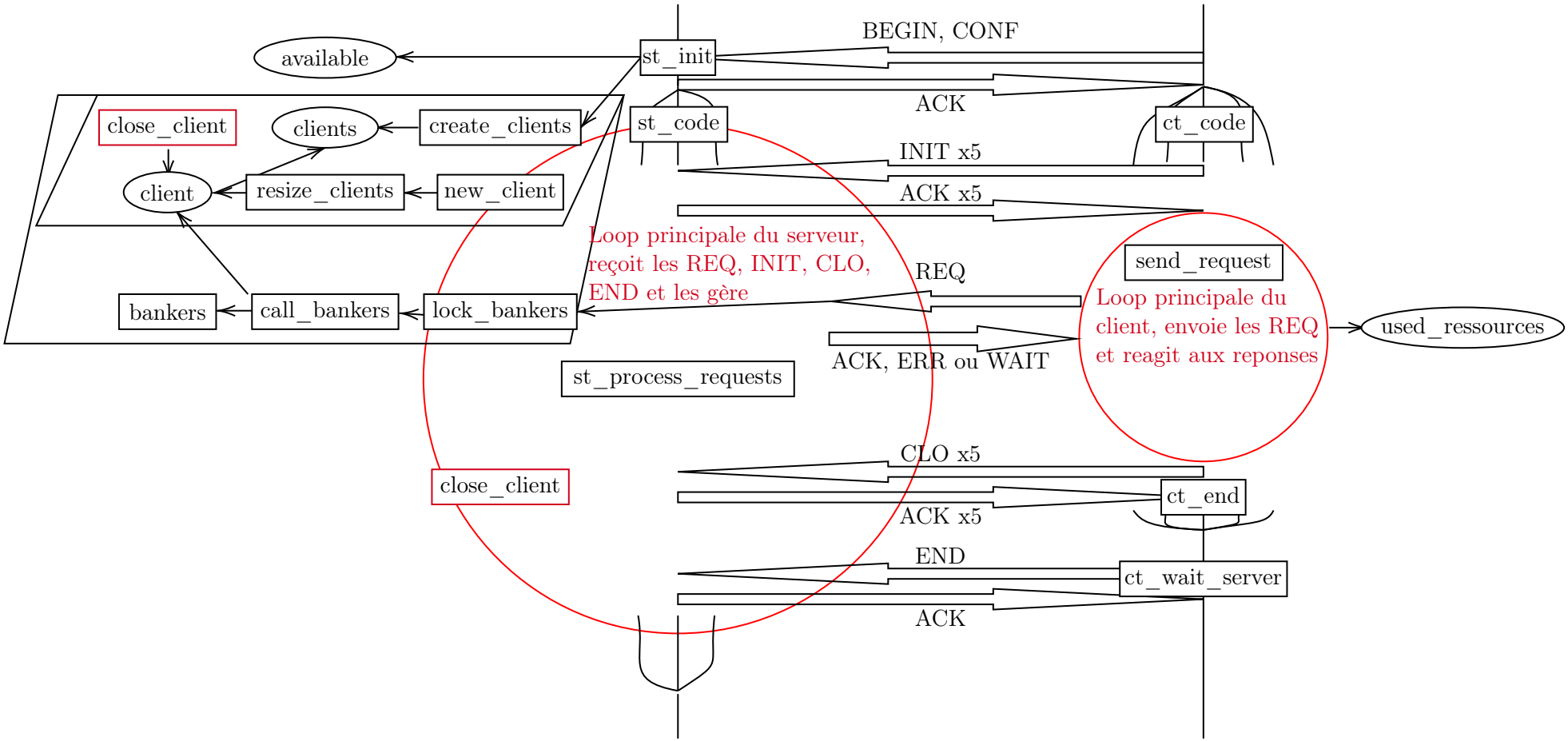

available
st_init
BEGIN, CONF
ACK
close_client
clients
create_clients
st_code
ct_code
INIT x5
client
resize_clients
new_client
ACK x5
Loop principale du serveur,
reçoit les REQ, INIT, CLO,
END et les gère
send_request
REQ
bankers
call_bankers
lock_bankers
Loop principale du
client, envoie les REQ
et reagit aux reponses
used_ressources
ACK, ERR ou WAIT
st_process_requests
CLO x5
close_client
ct_end
ACK x5
END
ct_wait_server
ACK

Schéma 3