Rapport BE - Programmation Système et Concurrente

API de communication entre threads

Guilhem Buisan & Léopold Désert-Legendre

Table des matières

I-Analyse du problème

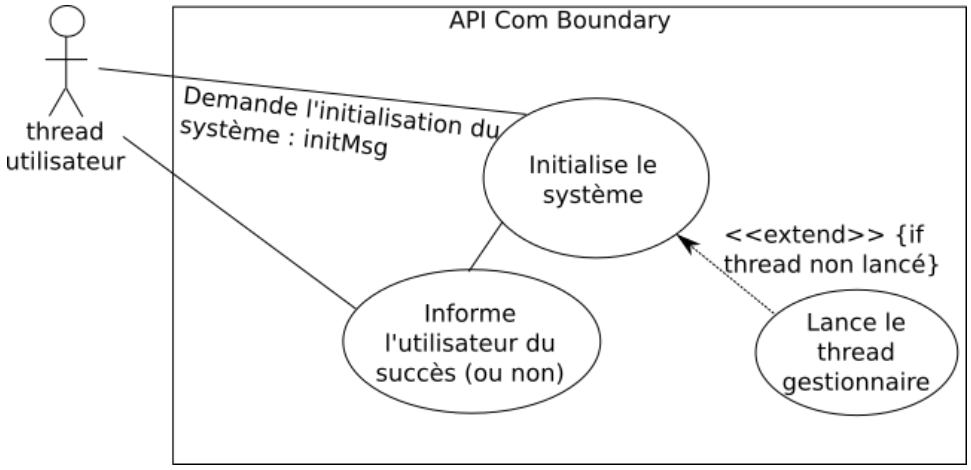
Cette API a pour but de transmettre des messages de type chaîne de caractères de taille maximale 60 caractères entre threads d'un même processus. En effet, la communication entre threads est souvent lourde au niveau du code et impacte les performances. Grâce à cette API, l'utilisateur peut assurer une communication simple et efficace.

La communication est basée sur un système de boîte à lettres. Il faut avant tout lancer l'initialisation de l'API pour pouvoir l'utiliser. Pour pouvoir envoyer et recevoir des messages, chaque thread doit s'abonner.

Chaque thread est identifié par l'API par un nombre unique choisi par l'utilisateur au moment de l'abonnement, 0 est pris par défaut et désigne tous les abonnés. La lecture de message est de type FIFO.

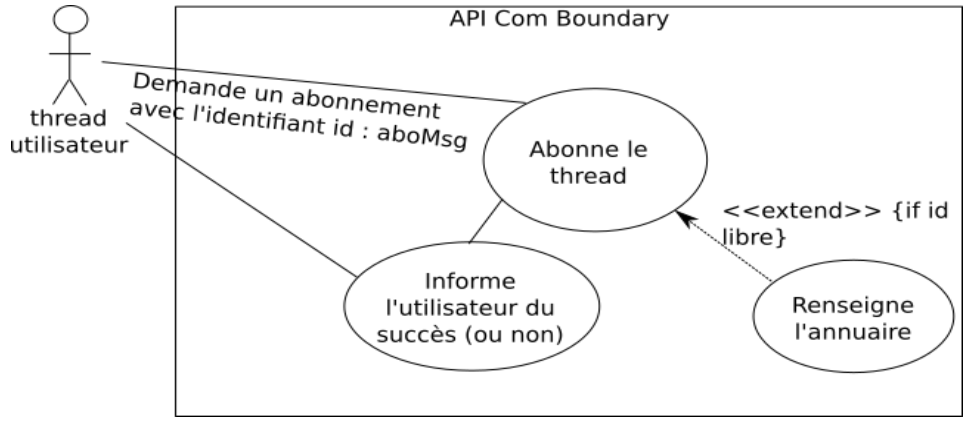
Initialisation :

Cette étape est nécessaire avant tout autre appel de fonctions de l'API.



Abonnement :

Il est nécessaire pour chaque thread de s'abonner pour pouvoir communiquer avec les autres. L'utilisateur doit choisir un identifiant unique sous forme de nombre (un entier) pour chaque abonnement. Un même thread peut s'abonner plusieurs fois avec des identifiants différents (et ainsi avoir plusieurs boites à lettres indépendantes). Les identifiants doivent être différents de 0.

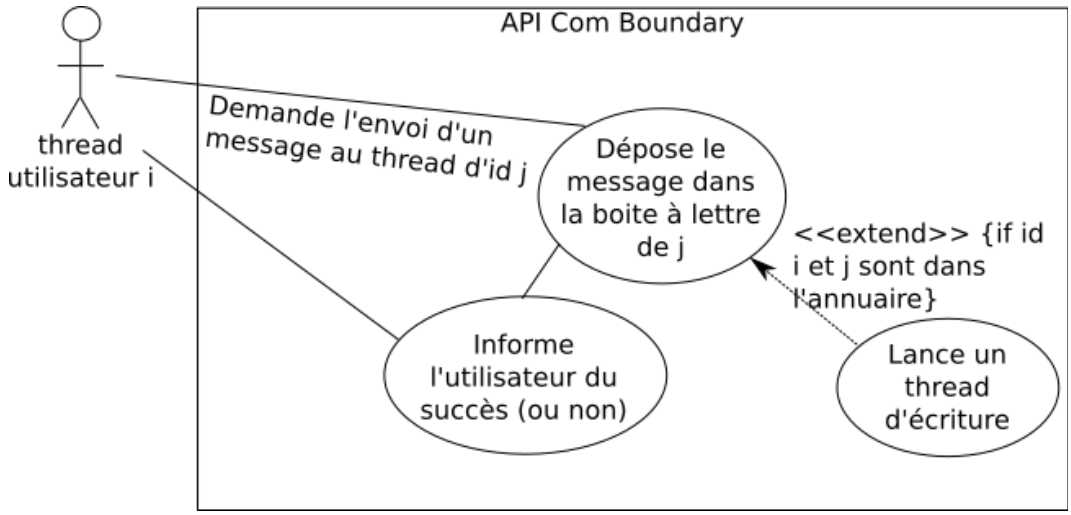


Envoi d'un message :

Pour envoyer un message le thread émetteur et le thread destinataire doivent être abonnés. Le message est une chaîne de caractères d'au maximum 60 caractères. L'utilisateur devra préciser l'identifiant source, l'identifiant destinataire (0 si l'utilisateur veut émettre un message à destination de tous les abonnés) et bien sûr le message. L'API vérifiera si le thread émetteur est bien abonné avec l'identifiant renseigné en source.

L'envoi en broadcast, envoie le message à tous les threads abonnés à la prise en compte de la requête sauf le thread source.

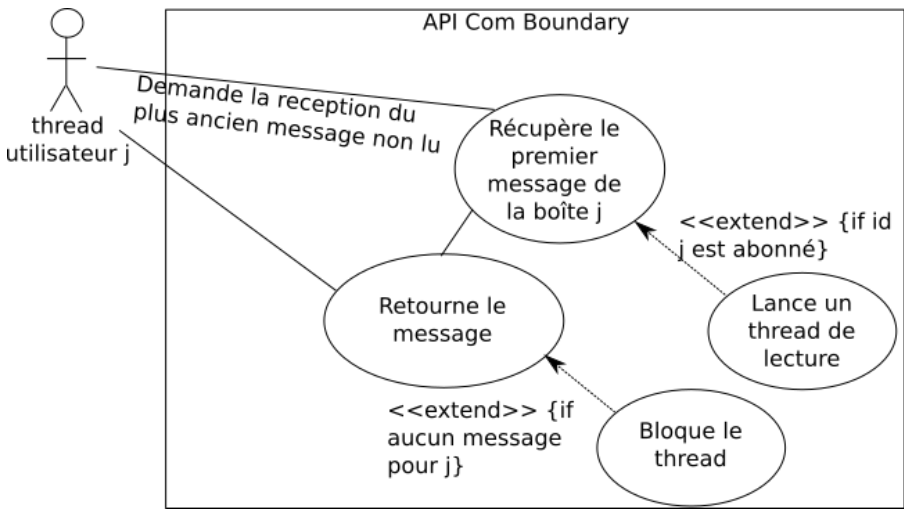
Cette fonction n'est bloquante que le temps de faire les vérifications élémentaires (source et destinataires abonnés,…), elle ne l'est pas lors de la remise du message. Ainsi, si la boite à lettres du destinataire est pleine, le thread émetteur continue quand même son exécution. Lorsque la boîte à lettre destinataire est pleine, les messages reçus ne sont pas perdus, il seront délivrés au fur et à mesure que la boite se vide, cependant, l'ordre de réception (FIFO) n'est plus garanti pour ces messages.



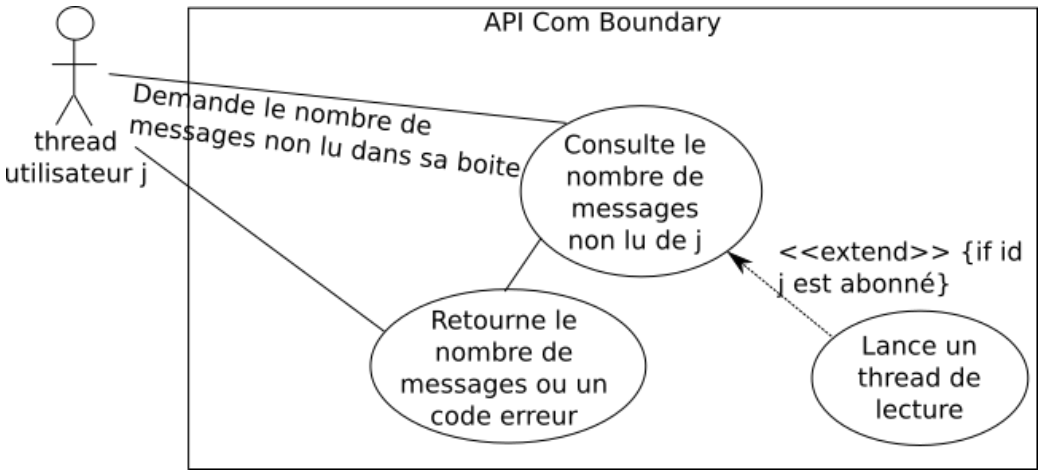
Réception :

Il y a deux façons d'accéder à sa boite à lettres :

- La lecture d'un message : Lors de la demande de lecture de sa boite à lettres, la fonction est bloquante tant que la boite à lettre est vide, sinon, elle renvoie le plus ancien message reçu (fonctionnement en FIFO). L'API vérifiera que le thread demandant la lecture est bien le thread qui s'est abonné avec cet identifiant (on évite que n'importe quel thread puisse lire dans n'importe quelle boite).



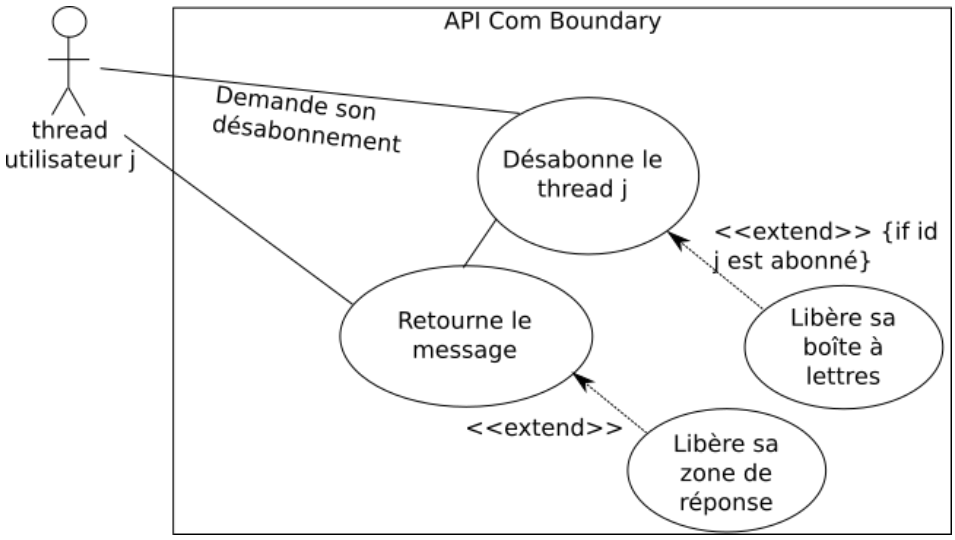
- Le nombre de messages : Lors de la lecture du nombre de messages disponibles, la fonction est bloquante uniquement durant les vérifications essentielles (id existant,…), puis retourne le nombre de messages disponibles (non lu) dans la boite à lettre. L'API vérifiera que le thread demandeur est bien le thread abonné avec l'identifiant demandé.



Désabonnement :

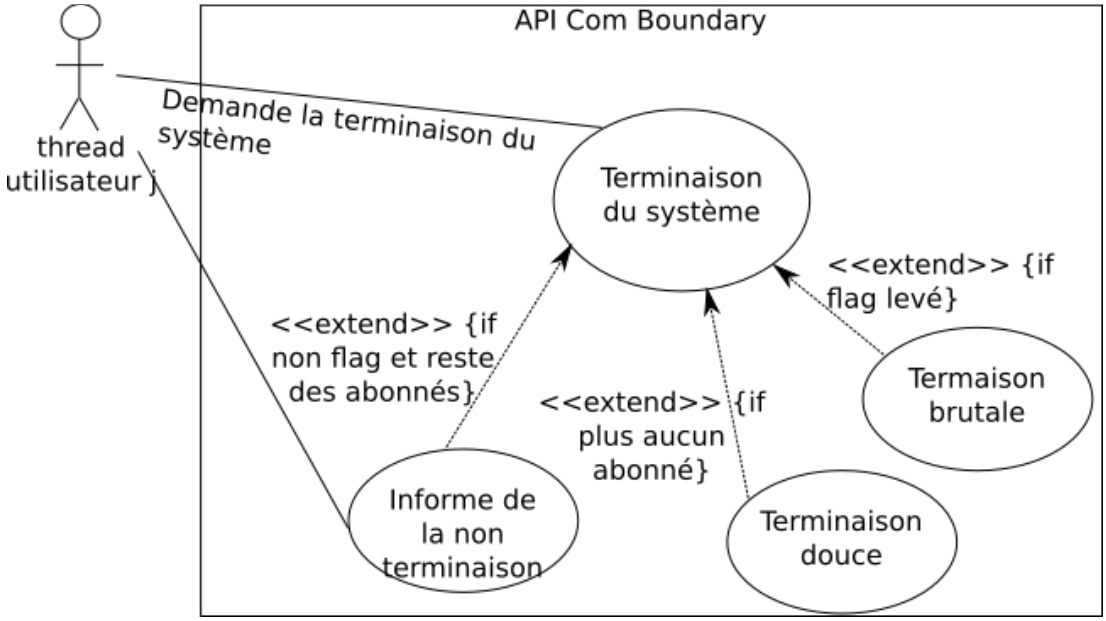
Lorsque un thread se désabonne d'un identifiant, ceci signifie qu'il ne souhaite plus être connu de l'API, il ne pourra donc plus envoyer ou recevoir de message sous cet identifiant. Cette fonction est bloquante jusqu'au désabonnement ou retour de code erreur. Si un thread est abonné avec plusieurs identifiants, s'il se désabonne que d'un, il pourra quand même continuer d'utiliser l'API avec les autres identifiants qu'il possède. Un thread ne peut désabonner que les identifiants avec lesquels il s'est abonné (un thread ne peut pas désabonner un autre thread).

Un thread peut se désabonner même s'il lui reste des messages non lus, dans ce cas les messages non lus ne seront plus accessibles.



Terminaison :

La terminaison du service de communication peut se produire de deux manières : soit la manière douce, le service vérifie qu'il n'y a plus de threads abonnés avant de se terminer, soit le manière forte, le service s'arrête sans aucune vérification.



II-Étude technique

Dans la suite nous nommerons id système l'identifiant alloué par le système d'exploitation à chaque thread et id API l'identifiant utilisé par l'utilisateur pour l'abonnement, l'envoi et la réception de messages.

II-1 Manuel d'utilisation

Notre API comportera 6 fonctions :

- **int initMsg(int nombremax)** : Permet la création du service (thread, mémoire, mutex, …). Doit être appelée une et une seule fois avant toute autre fonction de l'API.

  Argument : nombremax : Nombre maximal de threads pouvant être abonné au service.

  Retour :

  - 0 : service lancé
  - -1 : service déjà lancé
  - -2 : erreur création thread gestionnaire

- **int aboMsg(int id)** : Permet à un thread de s'abonner au service (d'envoyer et de recevoir des messages) et d'être connu comme le numéro **id**. Doit être appelée par chaque thread avant d'utiliser une autre fonction de l'API. **L'id doit être différent de 0 (broadcast).**

  Argument : id : Identifiant du thread sur le service.

  Retour :

  - 0 : identifiant abonné avec l'id API passé en argument
  - -1 : service non lancé
  - -2 : identifiant id existant
  - -3 : nombre maximal de thread abonné atteint
  - -4 : erreurs techniques (impossible d'allouer de la mémoire, ...)

- **int sendMsg(char * message, int dest_id, int source_id)** : Permet à un thread de déposer le message **message** dans la boîte à lettres du destinataire d'adresse **dest_id**.

  Arguments : message : Le message à faire passer (chaîne de caractères de 60 caractères max)

  dest_id : L'identifiant API de la boite à lettres cible.

  source_id : L'identifiant API de l’émetteur du message (utile en cas de plusieurs abonnements à l'API par un même thread)

Retour :

- 0 : message envoyé
- -1 : service non lancé
- -2 : tâche émettrice non abonnée (id API source inexistant)
- -3 : thread émetteur non associé avec cet id API (id API n'a pas été associé à ce thread)
- -4 : tâche destinataire non abonnée (id API destinataire inexistant)
- -5 : erreurs techniques

- **int recvMsg(int flag, int id, char ** message)** : Permet à un thread d'identifiant API **id** de consulter sa boite à lettres. Soit de lire le plus ancien message non lu (fonction bloquante si la boite à lettre est vide), soit de consulter le nombre de messages non lus.

  Arguments :

  - flag : 0 : lire le plus ancien message (**bloquant**)

    1 : consulter le nombre de messages non lus
  - id : id API de la boite à lettres à consulter
  - message : pointeur où l'utilisateur récupérera le message en cas de lecture

  Retour :

  - 0-* : nombre de messages non lu (dans le cas de la consultation du nombre de messages)
  - 0 : message lu et présent dans *message (*dans le cas d'une lecture de message)
  - -1 : service non lancé
  - -2 : id non abonné
  - -3 : thread demandeur non associé à cet id API (ce thread ne s'est pas abonné avec cet id)
  - -4 : *flag* (argument) invalide
  - -5 : Autres erreurs techniques

- **int desaboMsg(int id)** : Permet à un thread de se désabonner de l'id API **id**. Un thread désabonné ne pourra plus ni envoyé ni recevoir de message, sa boite à lettres est détruite même s'il reste des messages non lus. L'identifiant **id** libéré peut être a nouveau utilisé pour s'abonner.

  Argument :

  - id : identifiant API à désabonner

  Retour :

- 0 : désabonné
- -1 : service non lancé
- -2 : id non abonné
- -3 : id ne correspondant pas au thread (ce thread n'a pas été abonné avec cet id)
- -4 : erreurs techniques

- **int finMsg(int force)** : Termine le service de communication. Si **force** = 0 : le service vérifiera qu'il n'y plus de threads abonnés avant de se terminer, si **force** = 1 : le service s'arrêtera sans vérifications.

  Argument :

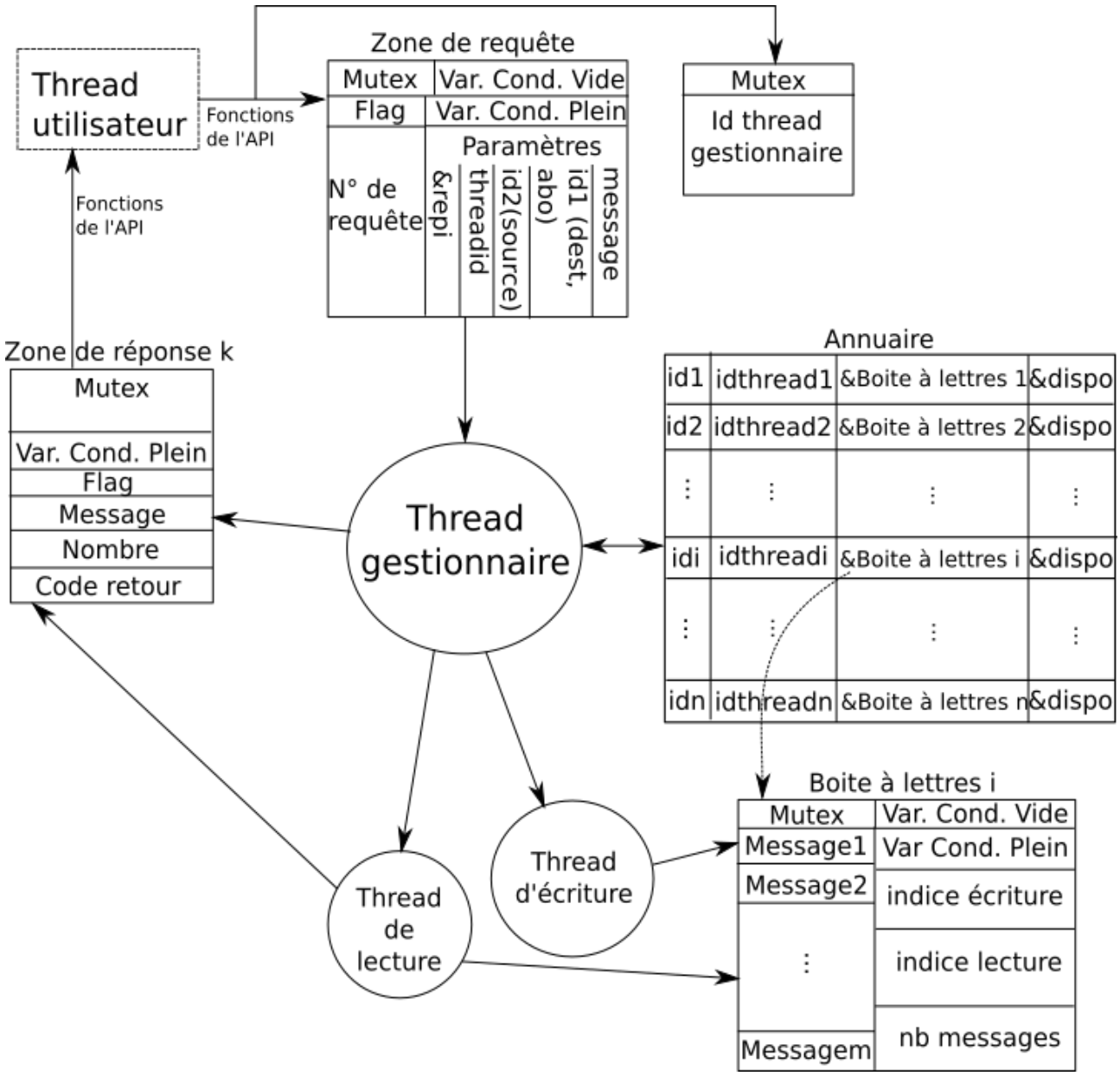
  - force : 0 : le service se termine s'il n'y a plus de tâches abonnées

    1 : le service se termine

  Retour :

  - 0 : le service s'est terminé
  - -1 : service non lancé
  - -2 : il reste des threads abonnés, le service ne s'est pas arrêté
  - -3 : erreurs techniques

II-2 Structure générale de l'API



II-2-a Zone de requête

C'est une variable globale protégée par un **mutex** et **deux variables conditionnelles** (une condition « vide », une condition « plein ») ; permet aux différentes fonctions de l'API de transmettre une requête au thread gestionnaire. On se retrouve ainsi dans une situation producteur (thread utilisateur via les fonctions de l'API)/consommateur (le thread gestionnaire en attente d'une requête).

Elle est déclarée dans le header de l'API et est toujours présente. Elle est définie sous forme d'une **structure *requestZone*** comportant :

- **int num_req** : définit le code de requête
  - 1 : demande abonnement
  - 2 : demande d'envoi de message
  - 3 : demande de reception de message
  - 4 : demande de nombre de messages
  - 5 : demande de desabonnement
  - 6 : terminaison douce
  - 7 : terminaison brutale
- **int user_id1** : Permet de passer un identifiant (id demandé dans le cas d'un abonnement, id destinataire dans le cas d'un envoi, id demandeur dans le cas d'une demande de recpetion, id à désabonner dans le cas d'un désabonnement)
- **int user_id2** : Permet de passer un deuxième identifiant (id source dans le cas d'un envoi)
- **repZone * adr_rep** : Adresse de la zone réponse.
- **pthread_t id_thread** : Permet de renseigner un identifiant de thread (identifiant du thread demandeur dans le cas d'un abonnement, identifiant du thread source dans le cas d'un envoi, identifiant du thread voulant se désabonner)

  Ceci permet d’empêcher un thread d'usurper l'identifiant d'un autre thread. De plus, en implémentant une fonction à nombre d'arguments variables, on peut se passer de l'id source pour un envoi si le thread abonné n'a qu'un identifiant.
- **char * msg** : Permet de renseigner un message(message à envoyer dans le cas d'un envoi)
- **int flag_req** : Permet de savoir si la requête à été prise en compte ou non par le gestionnaire
- **pthread_mutext_t mutex_req** : Mutex associé à la zone de requêtes.
- **pthread_cond_t var_cond_req_full** : Variable conditionnelle associée à la zone de requêtes pleine.

- **pthread_cond_t var_cond_req_empty** : Variable conditionnelle associée à la zone de requêtes vide.

La zone de requête ne sera renseignée uniquement par les fonctions de l'API invoquées par les threads utilisateurs, et sera lue uniquement par le thread gestionnaire. Le mutex permet d'assurer la synchronisation de cette ressource, et les variables conditionnelles permettent d'éviter l'attente active de la part des threads utilisateurs si elle est pleine ou du thread gestionnaire si elle est vide.

L'unicité de cette zone de requête est choisie car le temps de traitement du thread gestionnaire est très rapide. On évite donc les attentes en écriture de la part des threads utilisateur sans consommer de mémoire superflue en faisant une file d'attente de requêtes.

II-2-b Thread gestionnaire

Le thread gestionnaire est créé lors de l'initialisation. Son rôle est de lire les requêtes et d'y répondre. Pour ce faire, il dispose d'un annuaire en variable locale que lui seul peut consulter ou renseigner (nul besoin de mutex).

Sa structure est basée sur un « switch » qui traite les numéros de requêtes.

Lors d'un abonnement, il cherche une place libre dans l'annuaire, alloue la boîte à lettre dans le tas, ainsi qu'un flag d'existance de boite à lettre et renseigne leur adresse dans l'annuaire.

Pour une demande d'envoi de message, il lit l'adresse de boite à lettres correspondante à l'id API (passé en requête) dans l'annuaire (en vérifiant si l'id API existe et si l'id API correspond bien à l'id système) et créé un thread d'écriture dédié avec un type d’ordonnancement FIFO pour assurer l'ordre dans lequel les messages sont délivrés.

Pour une demande de lecture de message, il lit l'adresse de boite à lettre correspondante à l'id (passé en requête) dans l'annuaire (en vérifiant si l'id existe et si l'id système demandeur a bien accès à cette boîte à lettre) et créé un thread de lecture dédié avec un type d’ordonnancement FIFO pour assurer la lecture de messages dans le bon ordre.

Lors d'un désabonnement, il cherche l'id API correspondant dans l'annuaire, met le flag d’existence à 0, libère la zone mémoire de la boîte à lettre et met l'entrée correspondante à 0 dans l'annuaire. Il y a ici un problème de fuite mémoire, en effet le flag d’existence des boîtes aux lettres n'est jamais libéré, mais est pourtant indispensable pour garder un contrôle minimal sur les threads d'écriture/lecture et éviter le tant redouté *segmentation fault* si un de ces threads tente d’accéder à un boîte à lettre n'existant plus. Par exemple, si un thread d'écriture est lancé sur une boite à lettre i, mais qu'il n'est pas exécuté de suite, que le thread gestionnaire détruit la boite à lettre suite à un désabonnement puis que le thread d'écriture reprend son exécution et essaie d’accéder à la boite à lettre, les choses vont mal se passer. D'où cette fuite mémoire indispensable, pour la limiter nous recommandons l'utilisation d'un *char* pour le flag (uniquement un octet). Ainsi, s'il y a beaucoup d'abonnements, désabonnements successifs, la fuite de mémoire peut devenir importante.

Il est clair que nous avons essayé de privilégier la vitesse de traitement des requêtes à l'économie de mémoire.

Pour la terminaison, si elle est douce, le gestionnaire vérifie qu'il n'y a plus d'abonnement avant de se stopper. Si elle est brutale, le gestionnaire désabonne un à un les id API restants ceci afin d'assurer que tous les thread d'écriture/lecture se suiciderons lorsque il verront que la boîte à lettre n'existe plus.

II-2-c Thread d'écriture

A chaque requête d'envoi de message, un thread d'écriture est créé. Celui-ci prend le mutex de la boîte à lettre cible et est en attente si elle est pleine. Ce thread doit vérifier le flag indiquant si la boîte à lettre existe toujours avant d'essayer d'y accéder. Si elle n'est plus disponible, il se suicide. Si elle l'est, il écrit le message au bon endroit dans le buffer tournant incrémente le nombre de messages, et réveille les threads de lecture en attente.

En cas d'envoi en broadcast, si une erreur survient, les envois devant se produire après l'erreur sont annulés, seuls les destinataires qui ont pu être livrés avant l'erreur reçoivent le message.

II-2-d Thread de lecture

A chaque requête de lecture ou de nombre de messages, un thread de lecture est créé. Celui-ci doit prendre le mutex de la boite à lettre cible et attendre si elle est vide. De même il doit tester le flag assurant que la boite à lettre existe toujours, et se suicider sinon. Si la boite existe et qu'il y a un message, il le lit, décrémente le nombre de message réveille les threads d'écritures en attente de place libre et répond lui même dans la zone de réponse. C'est le seul cas où le thread gestionnaire n'écrit pas dans la boite réponse, et ceci afin de rendre la lecture de message bloquante sans bloquer le thread gestionnaire et le reste du système.

II-2-e Id du thread gestionnaire

Cette variable globale suit la **structure *Id_thread_gest*** décrite comme suit :

- **pthread_t id_gest** : Identifiant système du thread gestionnaire. Il est initialisée à 0, puis renseigné à l'initialisation, lors de l'appel de la fonction initMsg, ce qui permet de savoir s'il a déjà été lancé ou non. Il est consulté uniquement lors d'une initialisation (MsgInit) et écrit lors de la création du thread gestionnaire.
- **pthread_mutex_t mutex_gest** : Mutex associé à l'id du thread gestionnaire.

  Elle permet de savoir si le thread gestionnaire est lancé où non.

II-2-f Annuaire

L'annuaire est un tableau local au thread gestionnaire. La taille est fixée à la création du thread gestionnaire et est égale au nombre max de threads pouvant être abonnée (argument de la fonction initMsg). Chaque cellule du tableau contient la **structure *Annuaire*** suivante :

- **int id** : l'identifiant demandé par le thread abonné passé en argument lors de l'appel de la

fonction aboMsg,

- **pthread_t idThread** : identifiant système du thread abonné. Permet de s'assurer qu'un thread n'essaye pas d'usurper l'id d'un autre thread abonné.
- **BaL* bal** : adresse de la boîte à lettres du thread abonné.
- **Char* flag_exist** : Flag permettant aux threads de lecture et d'écriture de savoir si une BaL a été supprimée ou non.

L'annuaire est consulté par le thread gestionnaire à l'appel de chaque fonction. Il sera potentiellement modifié avec les fontions aboMsg (ajout d'un champ) et desaboMsg (retirer un champ).

A l'initialisation tous les *id* sont nuls (O(n)) puis à chaque écriture/lecture dans l'annuaire on rempli/vérifie depuis l'index 0 jusqu'à trouver un *id* nul/recherché (O(n)). On a donc une complexité linéaire à l'initialisation et à la lecture/écriture. Le tri aurait pu être une solution, mais un tri à chaque écriture ou désabonnement ne serait pas rentable pour un nombre de thread maximal assez faible.

II-2-g Boîtes à Lettres

Les boîtes à lettres sont des variables globales qui sont associées aux threads abonnés dans l'annuaire. Elles sont protégées par un **mutex** et **deux variables conditionnelles** (une condition « vide », une condition « plein »). Elles fonctionnent sous la forme d'un buffer tournant. Une boîte à lettre est sous la forme d'une structure *BaL* comme suit :

- **char** msg [taille_bal][taille_max_msg]** : Tableau contenant les messages à destination du thread correspondant. Il s'agit d'un buffer tournant. Le tableau a une capacité de 10 messages.
- **Int iecriture** : indice d'écriture dans le buffer tournant.
- **Int ilecture** : indice de lecture dans le buffer tournant.
- **Int nb_msg** : nombre de messages disponibles dans la boîte à lettres. Cette argument est utile lorsqu'un thread appelle la fonction recvMsg.
- **pthread_mutext_t mutex_bal** : Mutex associé à la boîte à lettres.
- **pthread_cond_t var_cond_bal_full** : Variable conditionnelle associée à la boîte à lettres pleine.
- **pthread_cond_t var_cond_bal_empty** : Variable conditionnelle associée à la boîte à lettres vide.

Une boîte à lettres peut être sollicitée au même moment par différents threads lancés par le gestionnaire, c'est pourquoi elles sont protégées par mutex et variable conditionnelle : Si la boîte est

pleine, les threads d'écritures de message dans cette boîte à lettre attendront.

II-2-h Zones de réponse

Les zones de réponse sont des variables globales qui sont créées avant chaque requête et détruite après chaque réponse (à chaque appel de fonction de l'API). Elles sont protégées par un **mutex** et **une variable conditionnelle** (permettant d'éviter que l'attente de la réponse soit active). Il s'agit d'une **structure *repZone*** composée comme suit :

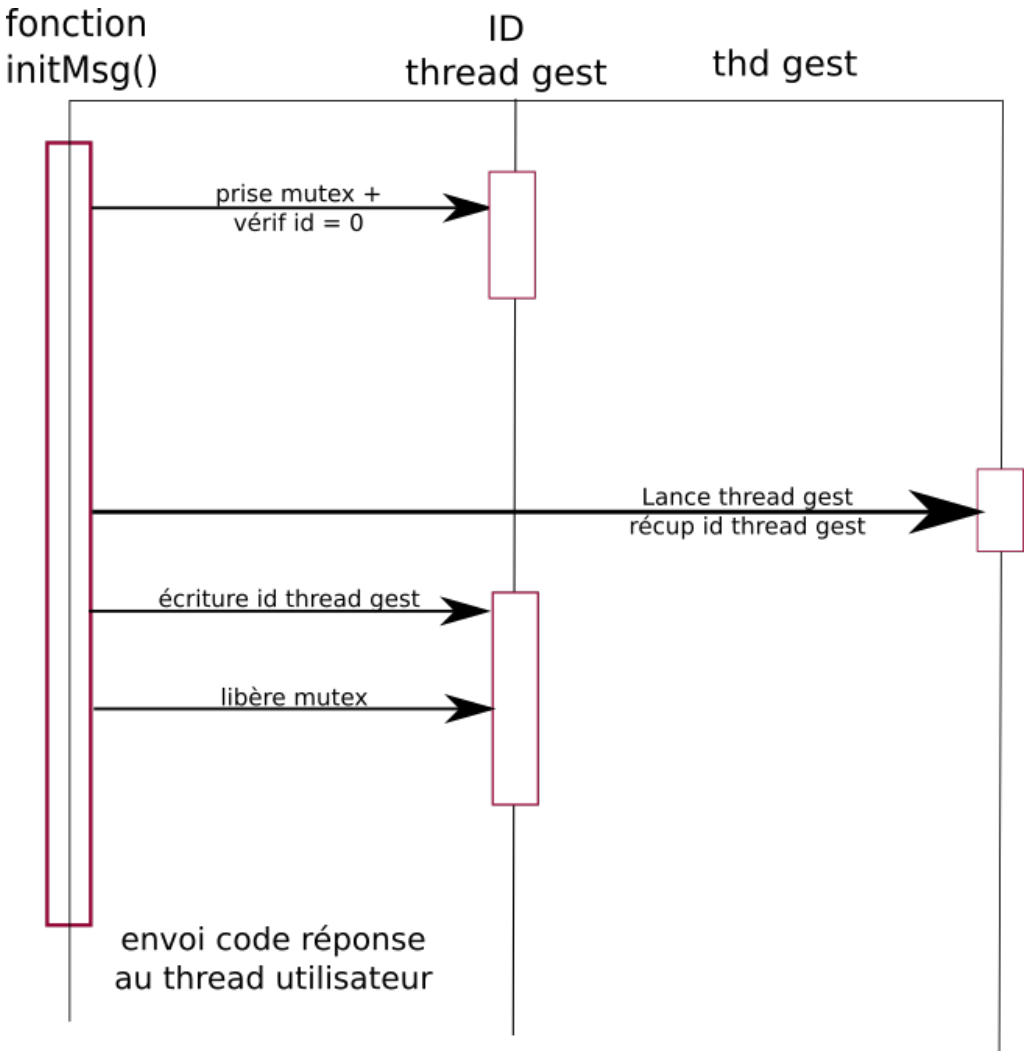
- **int flag_rep** : Permet à la fonction de l'API de savoir si la réponse a été envoyée. Permet également aux threads lancés par le gestionnaire de savoir s'ils peuvent écrire la réponse ou non.
- **int code_err** : Code d'erreur de la fonction de l'API appelée. Par défaut, 0 s'il n'y a pas d'erreur,
- **Union** :
  - **char * msg [taille_max_message]** : Dernier message de la Boîte à lettre du thread correspondant. Renvoyé après l'appel de la fonction recvMsg.
  - **Int nb_msg** : Entier représentant le nombre de message dans la boîte à lettres du thread correspondant. Renvoyé lors de l'appel de la fonction recvMsg.
- **pthread_mutex_t mutex_rep** : Mutex associé à la zone réponse
- **pthread_cond_t var_cond_rep** : Variable conditionnelle associée à la zone de réponse (évitant que l'attente active de la réponse par la fonction).

  Cette zone de réponse sera consultée par la fonctions de l'API qui l'a créée afin de renvoyer le message correspondant aux spécifications décrites dans la partie « Manuel d'utilisation » en II-1.

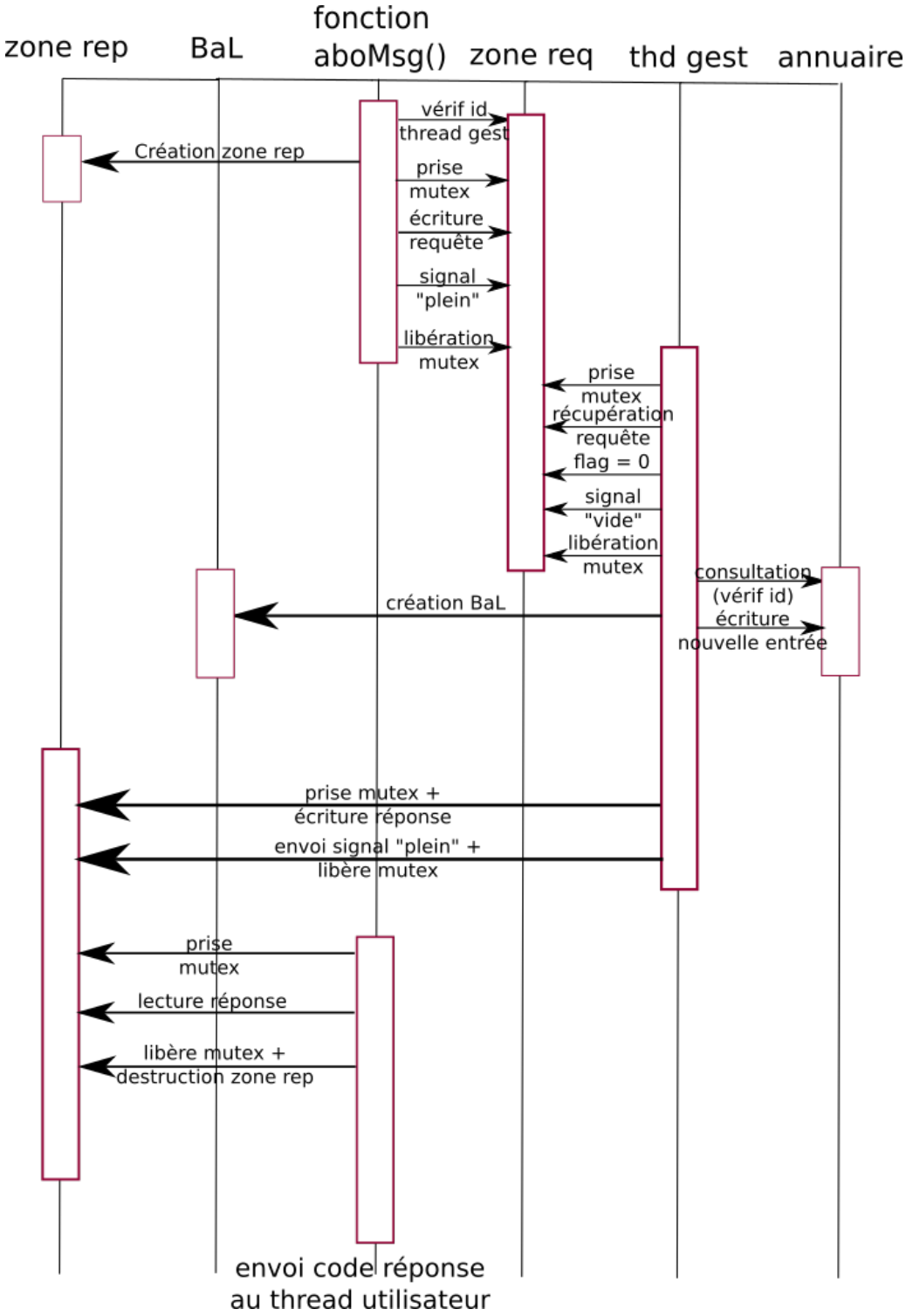
II-3 Description temporelle de l'API

Il s'agit ici de décrire plus précisément le déroulement de chaque fonction (en fonctionnement nominal) grâce à un diagramme de séquence.

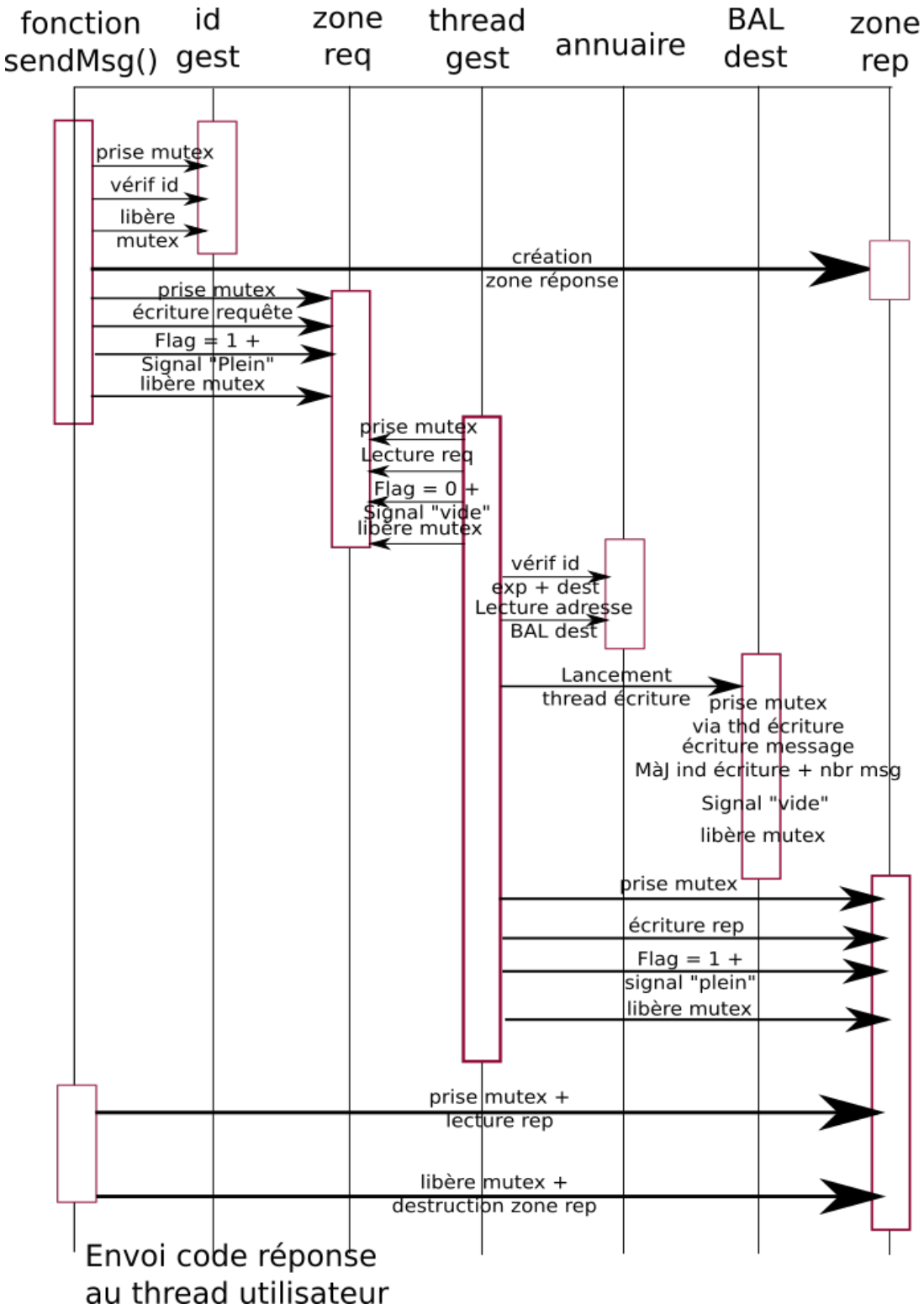
- InitMsg :



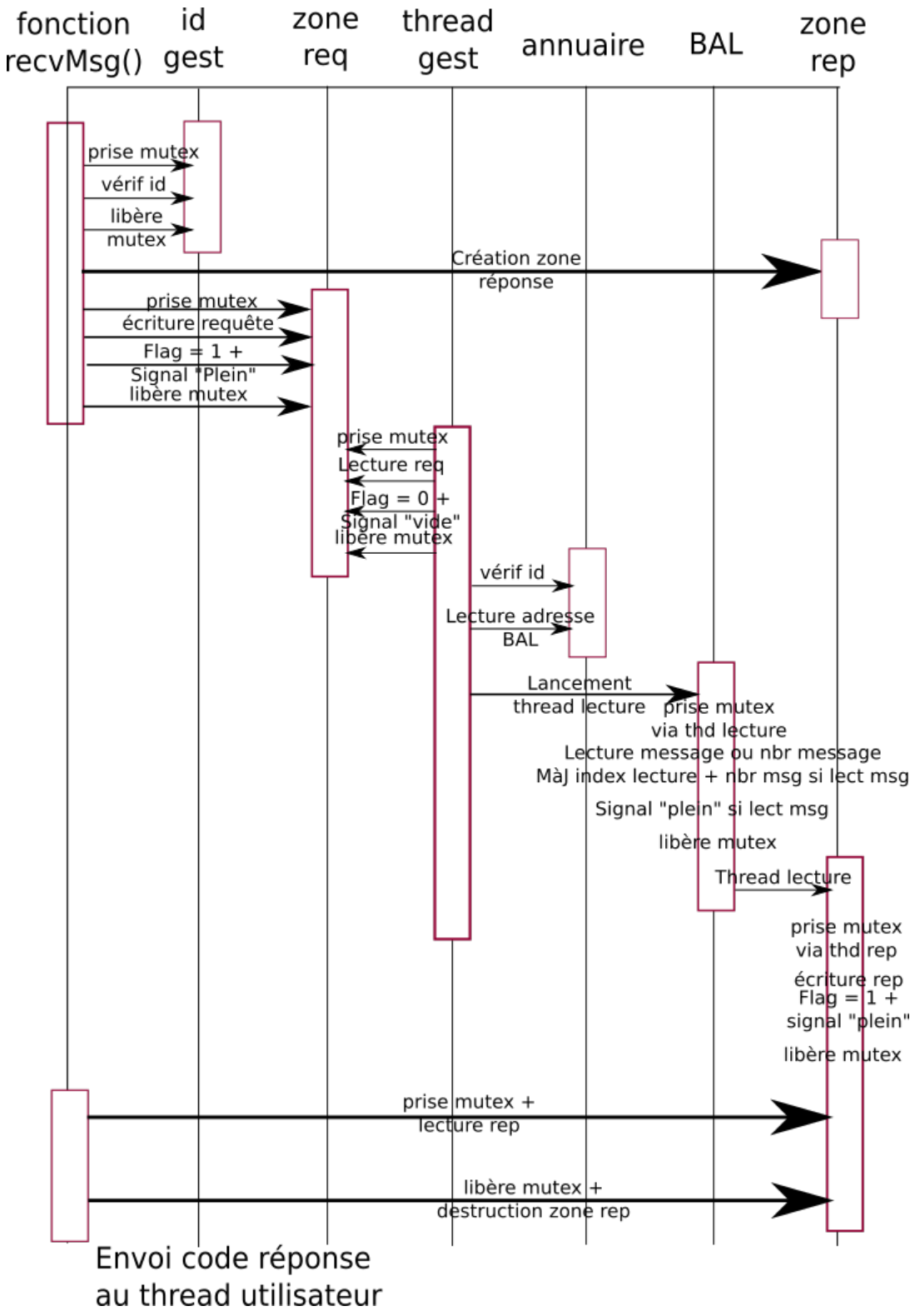
- aboMsg :


zone rep
BaL
fonction aboMsg()
zone req
thd gest
annuaire
vérif id thread gest
Création zone rep
prise mutex
écriture requête
signal "plein"
libération mutex
prise mutex
récupération requête
flag = 0
signal "vide"
libération mutex
consultation (vérif id)
écriture nouvelle entrée
création BaL
prise mutex + écriture réponse
envoi signal "plein" + libère mutex
prise mutex
lecture réponse
libère mutex + destruction zone rep
envoi code réponse au thread utilisateur

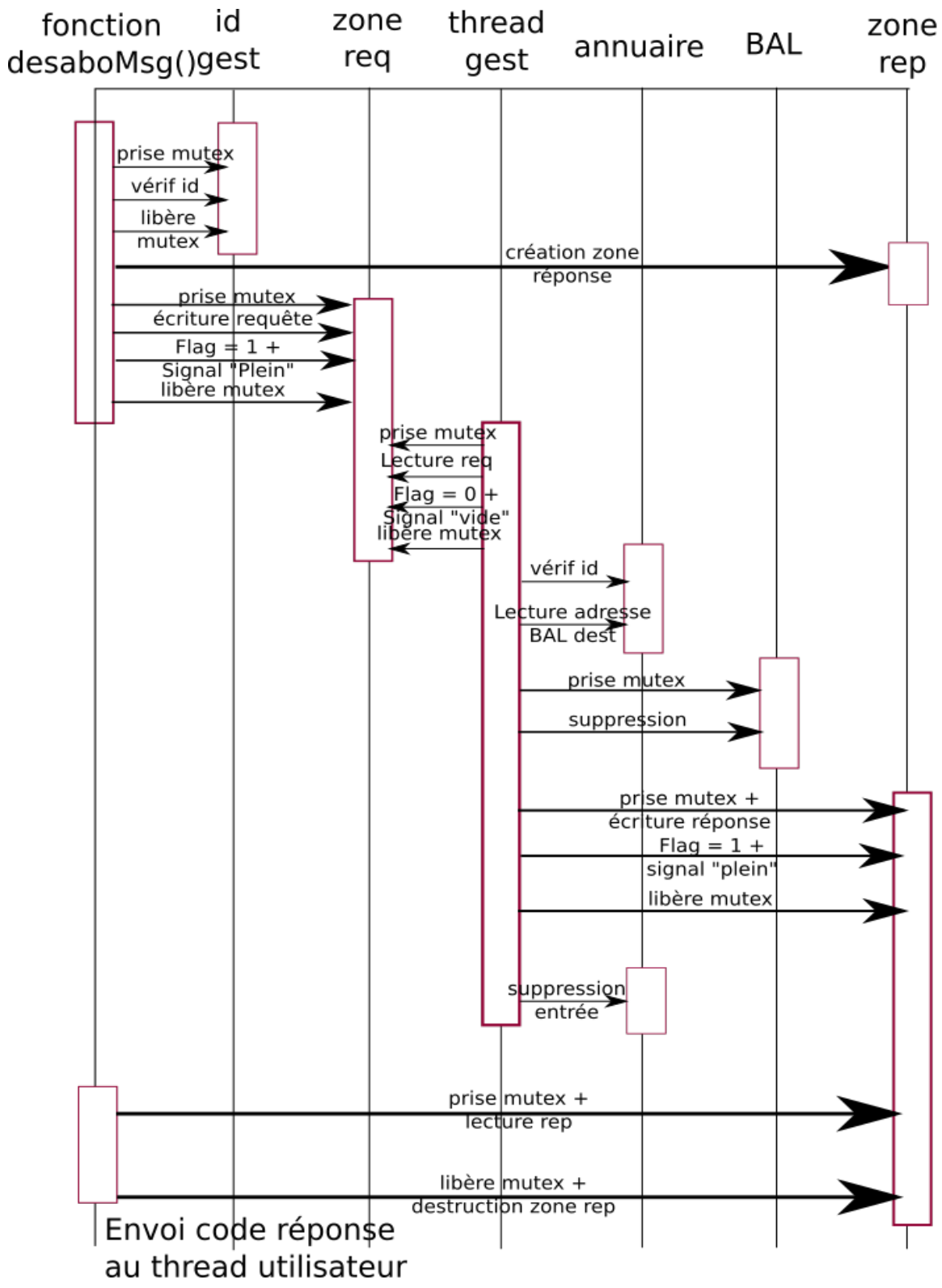
- sendMsg :


fonction sendMsg()
id gest
zone req
thread gest
annuaire
BAL dest
zone rep
prise mutex
vérif id
libère mutex
création zone réponse
prise mutex
écriture requête
Flag = 1 +
Signal "Plein"
libère mutex
prise mutex
Lecture req
Flag = 0 +
Signal "vide"
libère mutex
vérif id
exp + dest
Lecture adresse
BAL dest
Lancement thread écriture
prise mutex
via thd écriture
écriture message
MàJ ind écriture + nbr msg
Signal "vide"
libère mutex
prise mutex
écriture rep
Flag = 1 +
signal "plein"
libère mutex
prise mutex + lecture rep
libère mutex + destruction zone rep
Envoi code réponse au thread utilisateur

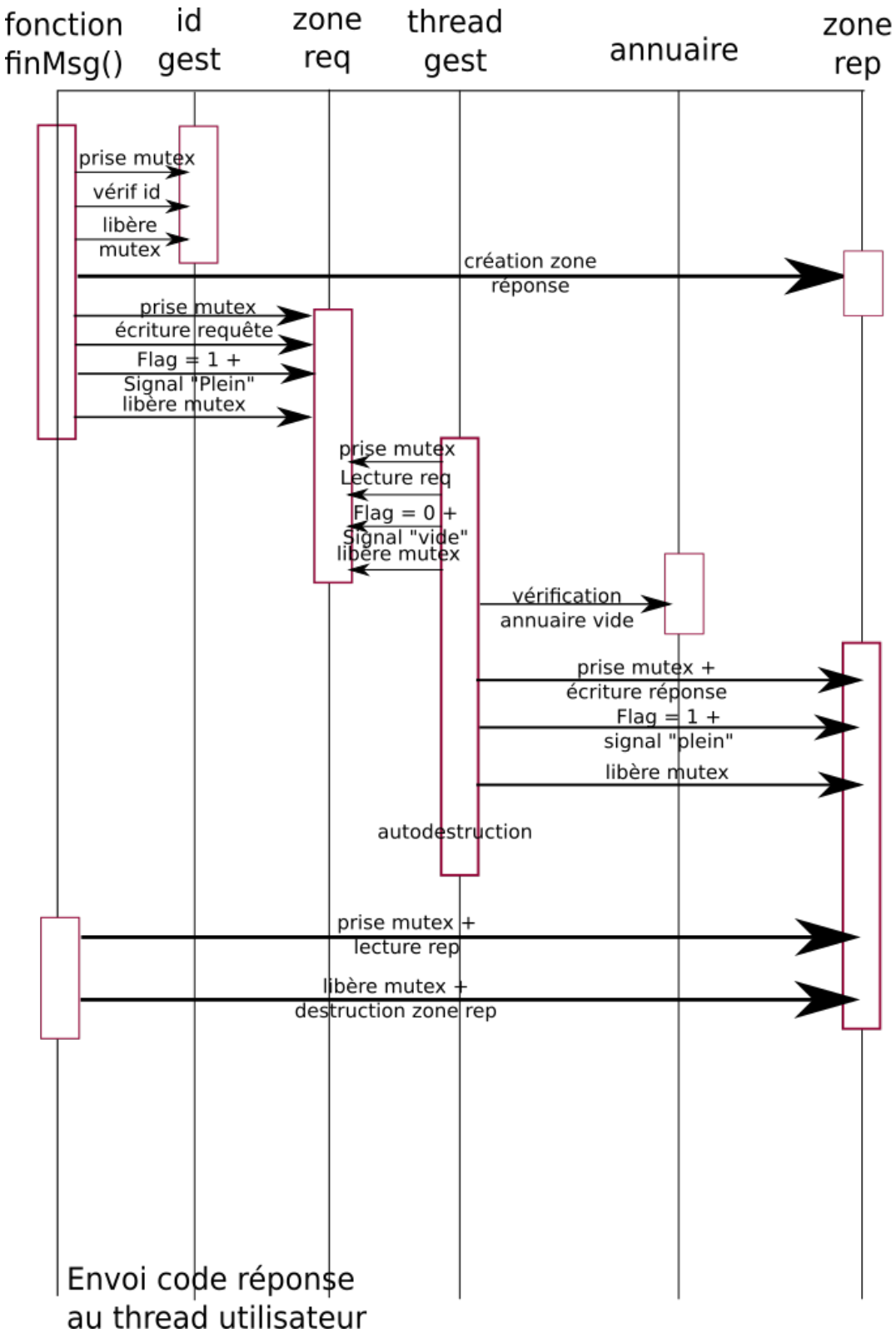
- recvMsg :


fonction recvMsg()
id gest
zone req
thread gest
annuaire
BAL
zone rep
prise mutex
vérif id
libère mutex
Création zone réponse
prise mutex
écriture requête
Flag = 1 +
Signal "Plein"
libère mutex
prise mutex
Lecture req
Flag = 0 +
Signal "vide"
libère mutex
vérif id
Lecture adresse BAL
Lancement thread lecture
prise mutex
via thd lecture
Lecture message ou nbr message
MàJ index lecture + nbr msg si lect msg
Signal "plein" si lect msg
libère mutex
Thread lecture
prise mutex
via thd rep
écriture rep
Flag = 1 +
signal "plein"
libère mutex
prise mutex + lecture rep
libère mutex + destruction zone rep
Envoi code réponse au thread utilisateur

- desaboMsg :


fonction desaboMsg()
id gest
zone req
thread gest
annuaire
BAL
zone rep
prise mutex
vérif id
libère mutex
création zone réponse
prise mutex
écriture requête
Flag = 1 +
Signal "Plein"
libère mutex
prise mutex
Lecture req
Flag = 0 +
Signal "vide"
libère mutex
vérif id
Lecture adresse BAL dest
prise mutex
suppression
prise mutex +
écriture réponse
Flag = 1 +
signal "plein"
libère mutex
suppression entrée
prise mutex +
lecture rep
libère mutex +
destruction zone rep
Envoi code réponse au thread utilisateur

- finMsg :


fonction finMsg()
id gest
zone req
thread gest
annuaire
zone rep
prise mutex
vérif id
libère mutex
création zone réponse
prise mutex
écriture requête
Flag = 1 +
Signal "Plein"
libère mutex
prise mutex
Lecture req
Flag = 0 +
Signal "vide"
libère mutex
vérification annuaire vide
prise mutex +
écriture réponse
Flag = 1 +
signal "plein"
libère mutex
autodestruction
prise mutex +
lecture rep
libère mutex +
destruction zone rep
Envoi code réponse
au thread utilisateur

III-Dossier de tests

III-a-Tests unitaires

Les premiers tests effectués sont les tests unitaires de chaque fonction proposée par l'API. Pour se faire, nous vérifions que chaque entité est créée et manipulée conformément à la description temporelle de cette fonction (cf II-3). Pour suivre les étapes de manière temporelle, nous utilisons des affichages écrans (fonction printf).

Initialisation :

1. Un thread appelle la fonction initialisation.
2. Plusieurs threads appellent la fonction d'initialisation « en même temps ».
3. Le même thread appelle la fonction d'initialisation plusieurs fois.

Abonnement :

1. Un ou plusieurs threads essaient de s'abonner sans que la fonction d'initialisation n'est été appelée.
2. Plusieurs threads s'abonnent les uns après les autres avec des id différents.
3. Un de ces threads s'abonne plusieurs fois avec des id différents.
4. Un de ces threads essaie de s'abonner plusieurs fois avec le même id.
5. Plusieurs threads essaient de s'abonner avec le même id.
6. Cinq threads veulent s'abonner avec des id différents, alors que le nombre de threads maximal déclaré est trois.

Envoi :

1. Un ou plusieurs threads essaient d'envoyer un message sans que la fonction d'initialisation est été appelée.
2. Un thread non abonné essaie d'envoyer un message à un thread abonné (ou non).
3. Un thread abonné essaie d'envoyer un message à un thread non abonné.
4. Un thread non abonné essaie d'envoyer un message en broadcast.
5. Un thread abonné envoie en boucle un message à un thread qui s'abonne plus tard.
6. Un thread abonné essaye d'envoyer un message à partir d'un id API qui ne lui appartient pas.
7. Un thread abonné envoie un message à un thread abonné.
8. Un thread abonné envoie un message vers un id abonné avec une boite à lettres pleine.

9. Un thread abonné envoie un message en broadcast.

Réception :

1. Un ou plusieurs threads essaient d'envoyer un message sans que la fonction d'initialisation est été appelée.
2. Un thread non abonné essaye de récupérer un message (ou le nombre) sur un id existant.
3. Un thread non abonné essaye de récupérer un message (ou le nombre) sur un id non existant.
4. Un thread abonné essaye de récupérer un message (ou le nombre) sur un id qui ne lui appartient pas.
5. Un thread abonné essaye de récupérer un message alors qu'il n'y en a pas.
6. Un thread abonné essaye de lire le nombre de message alors qu'il n'y en pas.
7. Un thread abonné récupère un message (et il y en a au moins un).
8. Un thread abonné lit le nombre de message (et il y en a au moins un).

Désabonnement :

1. Un ou plusieurs threads essaient de se désabonner sans que la fonction d'initialisation ait été appelée.
2. Un ou plusieurs threads essaient de se désabonner sans avoir été abonné (id inexistant)
3. Un thread essaie de désabonner d'un id qui ne lui appartient pas.

Terminaison :

1. Un ou plusieurs threads essaient de terminer l'API sans que le fonction d'initialisation est été appelée.
2. Un thread essaie de terminer de manière douce l'API alors qu'il reste un ou plusieurs threads abonnés.
3. Un thread essaie de terminer (manière douce ou non) l’application alors qu'un thread de lecture/écriture est lancé.

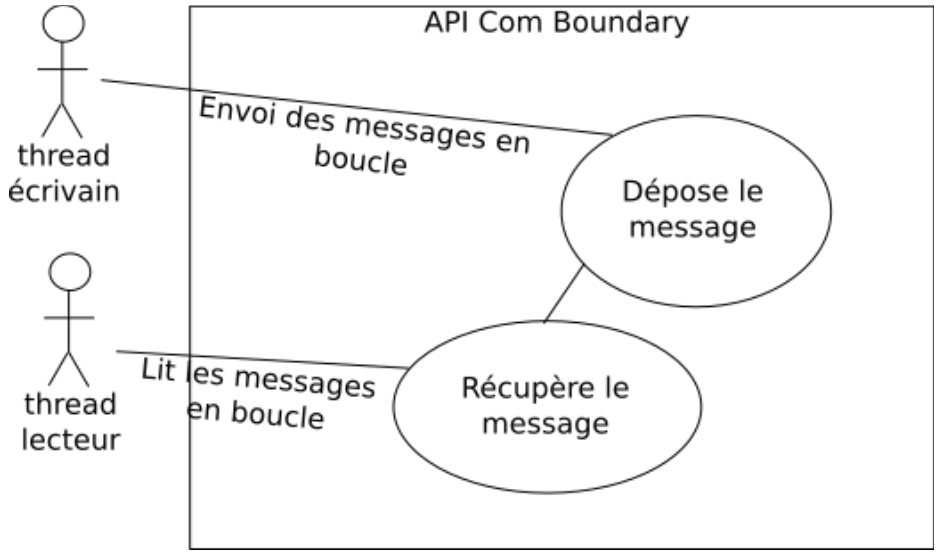
III-b-Tests par scénarii

Les tests par scenario ont pour but de vérifier et confirmer le bon déroulement d'une opération complète de gestions de threads par l'API. En effet, bien que chaque fonction de l'API soit

unitairement opérationnelle, le bon fonctionnement de la gestion des threads n'est pas garanti tant que ces fontions n'auront pas été testés en contexte d'utilisation.

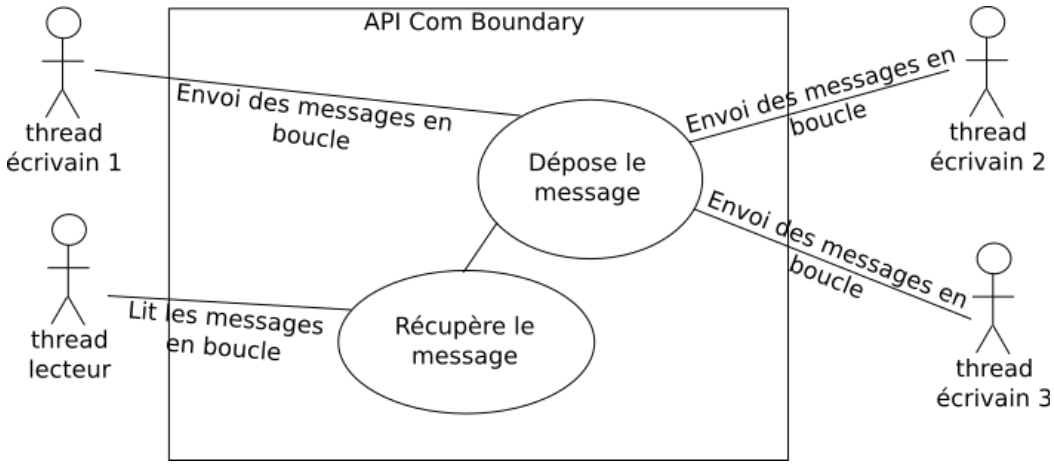
Scénario 1 : Serveur d'affichage :

Pour notre premier test, nous avons choisi de reprendre le TD du serveur d'affichage vu en cours, sans la tâche de supervision. Il s'agit donc ici d'un cas minimal d'utilisation de l'API, avec seulement deux threads qui tentent de communiquer : un thread d'écriture qui écrit des messages destinés à être affichés et un thread de lecture qui affiche les messages.



Scénario 2 : Test de congestion

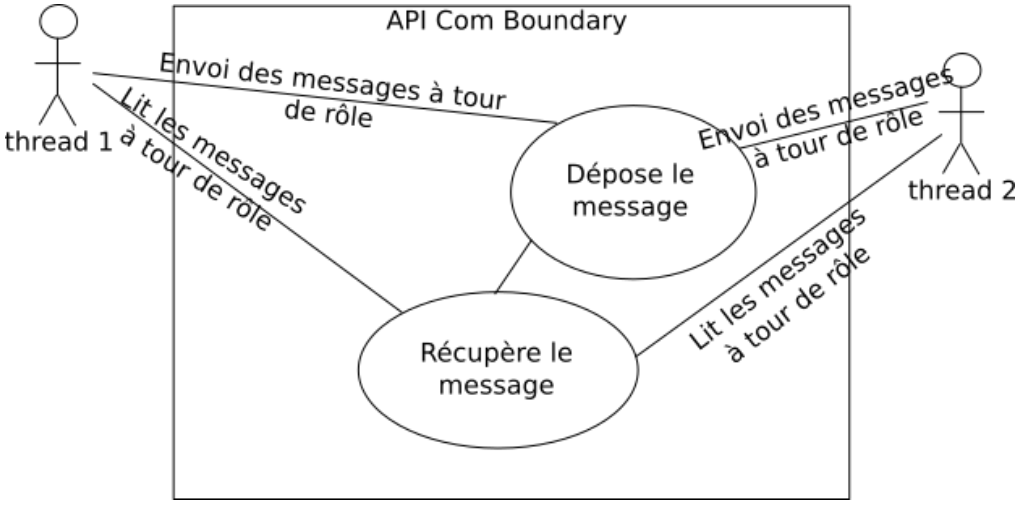
Pour notre second test, nous avons choisi de vérifier la robustesse de l'API face à un flux important de messages à destination d'un même thread. Ainsi, il s'agira donc de créer 3 threads qui tentent d'envoyer simultanément des messages à un thread cible qui les affichera et les lira à un rythme plus faible en utilisant l'API.



Scénario 3 : Test de communication

Pour ce troisième test, nous souhaitons nous assurer qu'un thread est capable à la fois

d'envoyer et de recevoir des messages. Pour cela, nous créons deux threads qui communiquent l'un avec l'autre. Lorsqu'un thread reçoit un message, il renverra un message vers l'autre thread et ainsi de suite.



IV-Résultats des tests

IV-a Tests unitaires

Initialisation :

1. L'initialisation fonctionne correctement, le thread gestionnaire se lance et son identifiant est renseigné dans id_thread_gest. OK
2. Le premier thread prend le mutex, les autres attendent alors, puis initialise le service. Une fois le mutex libérés les autres threads s'aperçoivent que l'id_thread_gest n'est pas égal à 0 et renvoie donc -1. OK
3. Tous les appels d'initialisation supplémentaire renvoient -1. OK

Abonnement :

1. La fonction retourne -1. OK
2. Le thread gestionnaire traite les requêtes une part une et abonne chaque thread avec l'id demandé, les fonctions retournent 0 (jusqu'au nombre threadmax atteint). OK
3. Le thread gestionnaire abonne le même thread avec des id différents, les fonctions retournent 0 (jusqu'au nombre threadmax atteint). OK
4. Au premier abonnement avec l'id, l'abonnement se passe bien et la fonction retourne 0, mais pour tous les autres, l'abonnement ne fonctionne pas et la fonction retourne -2. OK
5. Le premier thread à s'abonner avec cet identifiant est bien abonné et à le code retour 0. Les autres ne sont pas abonnés et on le code retour -2. OK
6. Les trois premiers threads s'abonnent et ont le code retour 0, les deux d'après ne sont pas abonnés et ont le code retour -3. OK

Envoi :

1. Les fonctions retournent le code retour -1. OK
2. Le message n'est pas envoyé et on a le code retour -2. OK
3. Le message n'est pas envoyé et on a le code retour -4. OK
4. Les messages ne sont pas envoyés et on a le code retour -2. OK
5. Les messsages ne sont pas envoyés et l'envoyeur à le code retour -4 jusqu'à ce que le destinataire s'abonne. Tous les envois pris en compte après l'abonnement fonctionnent et renvoie le code retour 0. OK
6. Le message n'est pas envoyé et on a le code retour -3. OK
7. Le message est délivré et l'envoyeur à le code retour 0. OK

8. Le thread d'écriture reste en attente et l'envoyeur à le code retour 0. OK
9. Le temps de traitement est un plus long, mais tous les messages sont envoyés et la fonction renvoie 0.

Réception

Non codé : les résultats seront dans l'annexe.

Désabonnement

Non codé : les résultats seront dans l'annexe.

Terminaison

Non codé : les résultats seront dans l'annexe.

IV-b- Tests par scénarii

Scénario 1 : Non codé : les résultats seront dans l'annexe.

Scénario 2 : Non codé : les résultats seront dans l'annexe.

Scénario 3 : Non codé : les résultats seront dans l'annexe.

V-Conclusion

A ce jour, nous avons terminé l'architecture de l'application et entamé la majorité du code. En particulier, la fonction initMsg(), aboMsg() et sendMsg() sont déjà fonctionnelles et répondent bien aux tests spécifiés ainsi qu'aux spécifications définies plus haut.

Jusqu'à maintenant, le seul défaut architectural que nous n'avons pas su corriger est le problème de fuite mémoire défini ci-dessus (II-2-b). Ce problème ayant été décelé tardivement, nous n'avons pas pu définir de nouvelles structures permettant de régler ce problème dans un délai raisonnable.

A l'issue des deux semaines supplémentaires à partir du jour de la publication de ce rapport, nous joindrons à celui-ci une annexe regroupant toutes les fonctionnalités ajoutées (tests, modifications, bilan...) au cours de ces deux semaines. Le projet est en bonne voie et il devrait respecter les délais imposés.