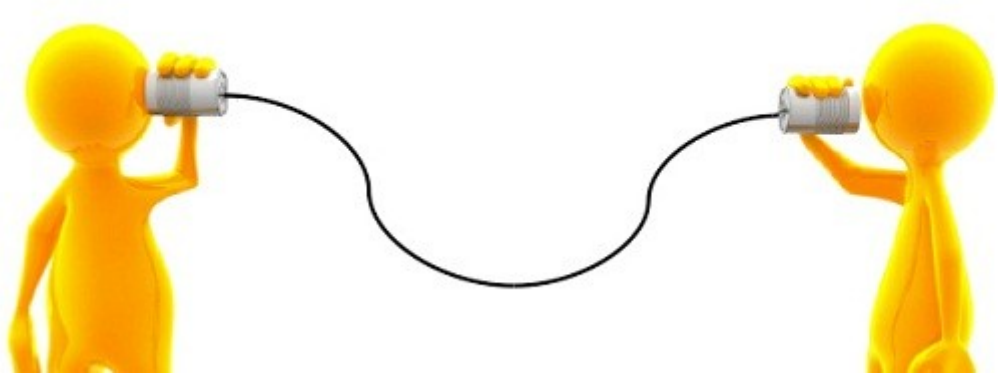
Rapport de BE

API de communication entre threads



Bonneval Fabien — **Dzieciol** Nicolas

30/11/2015

Tables des matières

1. Analyse du problème

1.1 Présentation du problème

Le but de ce BE est de concevoir une API qui fournira un service de communication par messages entre threads d'un même processus dans le cadre d'une application multitâche. En effet ce système peut être mis en place directement dans le code des tâches mais cela implique une mise en place plus lourde et impactant les performances.

La communication entre threads se fera sur le prince de boîte à lettre, pour utiliser le service il devra tout d'abord s'abonner avec un identifiant unique, pour pouvoir ensuite envoyer et recevoir des messages.

Cette API permettra à chaque abonné d'envoyer des messages du type de son choix à tout autre abonné avec l'identifiant de celui-ci ou a tous les abonnés avec un identifiant de broadcast. La lecture des messages se fera par une file FIFO.

1.2 Analyse des besoins

Nous allons à présent voir les cas d'utilisation correspondant aux fonctions décrites dans le cahier des charges. Chaque fonction est bloquante le temps de faire des vérifications élémentaires.

1) Initialisation

Cette étape est indispensable pour pouvoir utiliser l'API, la fonction initMsg() peut être appelée par n'importe quel thread et lancera le service, à condition que celui-ci ne soit pas déjà lancé.

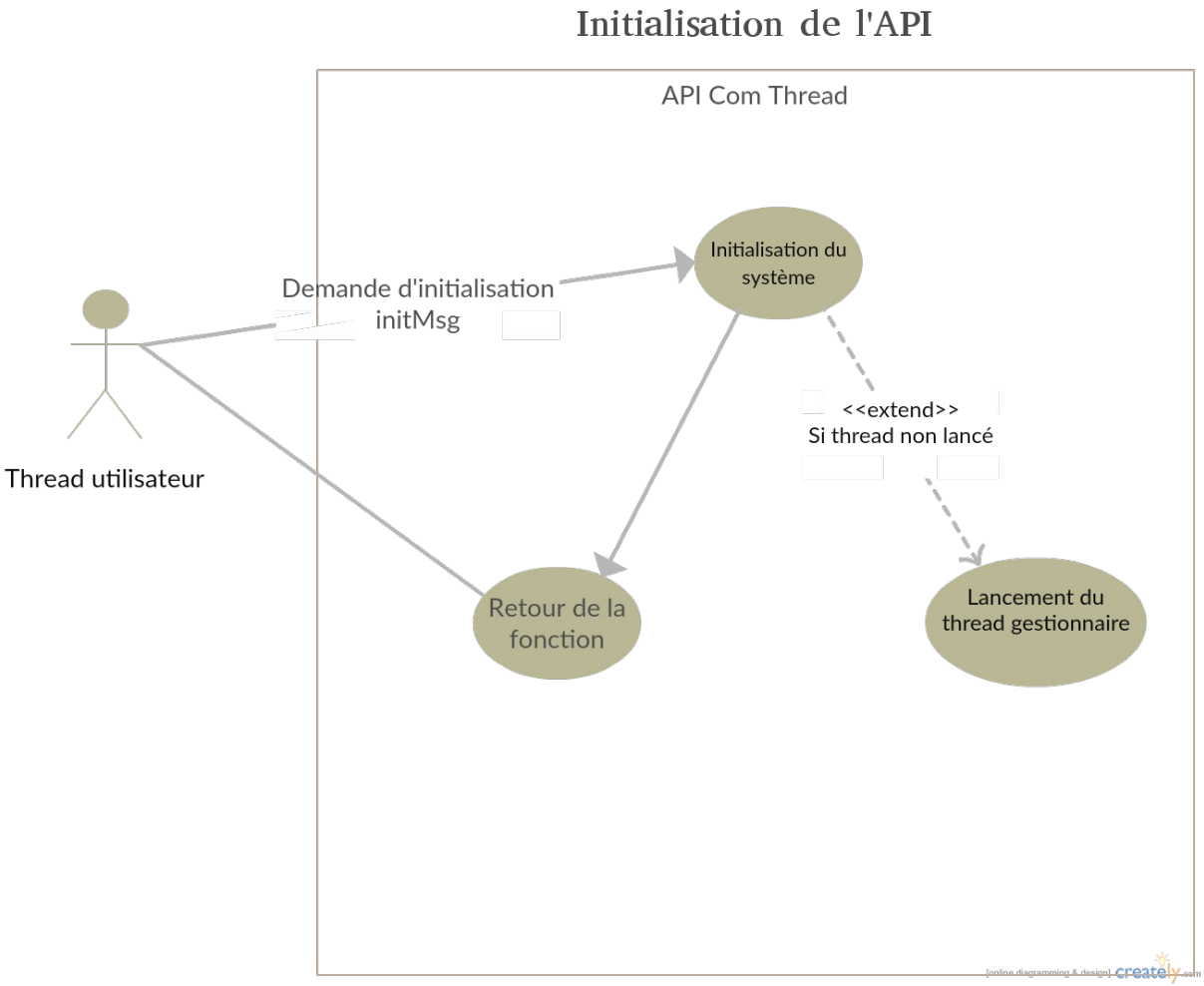


Illustration 1: Use case initialisation

2) Abonnement d'une tâche au service

Pour pouvoir communiquer un thread doit s'abonner au service, pour cela le thread devra formuler une demande d'abonnement avec la fonction aboMsg() de l'API en fournissant un identifiant (nombre entier) qui ne doit pas être déjà utilisé.

Par défaut l'identifiant 0 caractérisera « l'adresse de broadcast » et il ne sera donc pas utilisable pour un thread.

Lors de l'abonnement, une zone de communication entre le gestionnaire et l'utilisateur sera crée ainsi qu'une messagerie accessible uniquement par le thread gestionnaire.

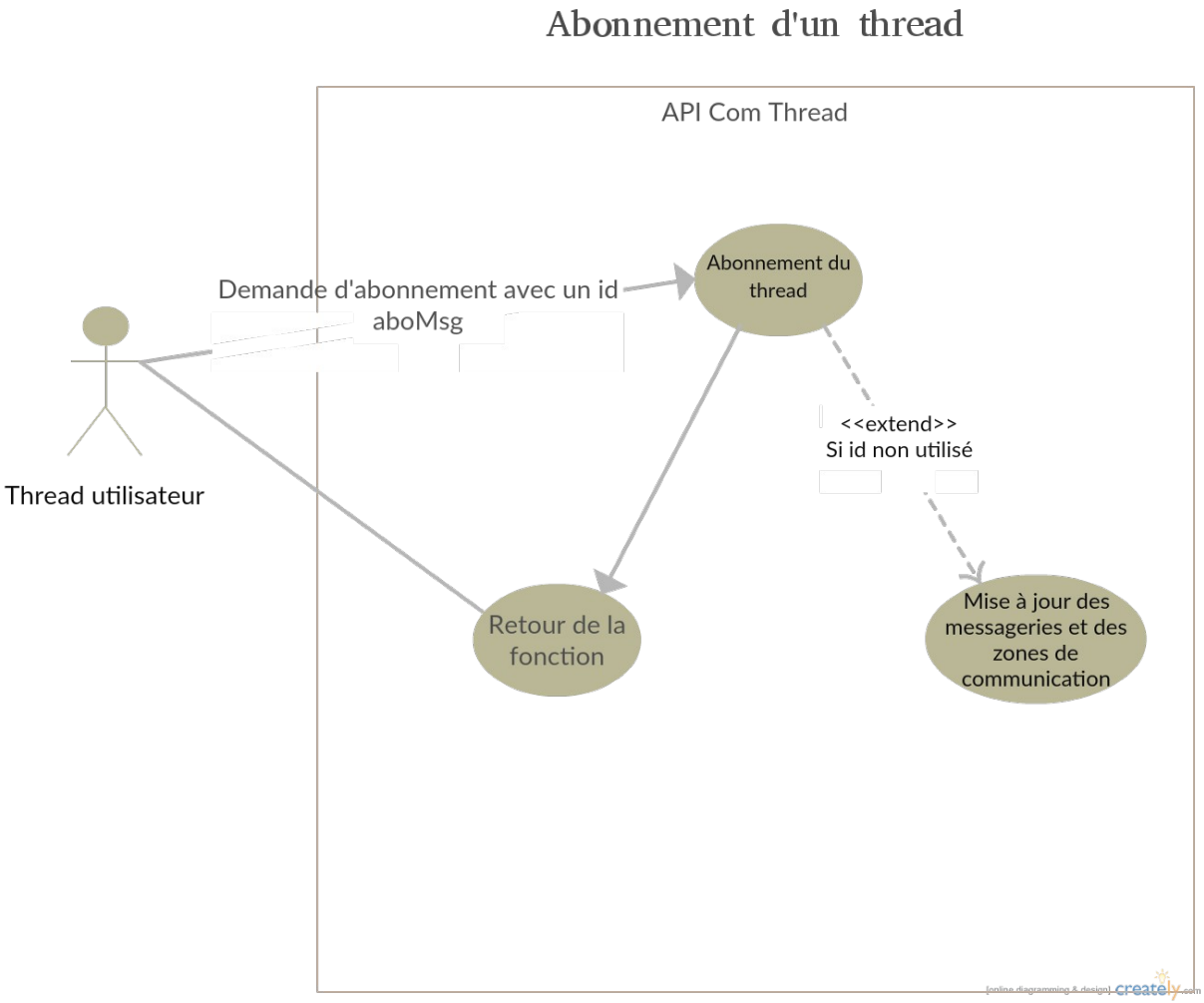


Illustration 2: Use case abonnement

3) Émission d'un message

Pour envoyer un message l'expéditeur et le destinataire doivent être abonnés au service. Si ce n'est pas le cas l'API retournera un message d'erreur correspondant.

L'expéditeur indiquera dans la zone de communication l'identifiant du destinataire ou 0 pour distribuer à tous les abonnés.

Le message pourra être de n'importe quel type et nécessite donc que l'expéditeur renseigne la taille des données à envoyer.

Le thread gestionnaire devra vérifier des informations avant de renvoyer le message de retour à l'expéditeur, cette fonction sera donc bloquante le temps de ces vérifications (expéditeur et destinataire abonnés au service) mais pas si la messagerie est pleine.

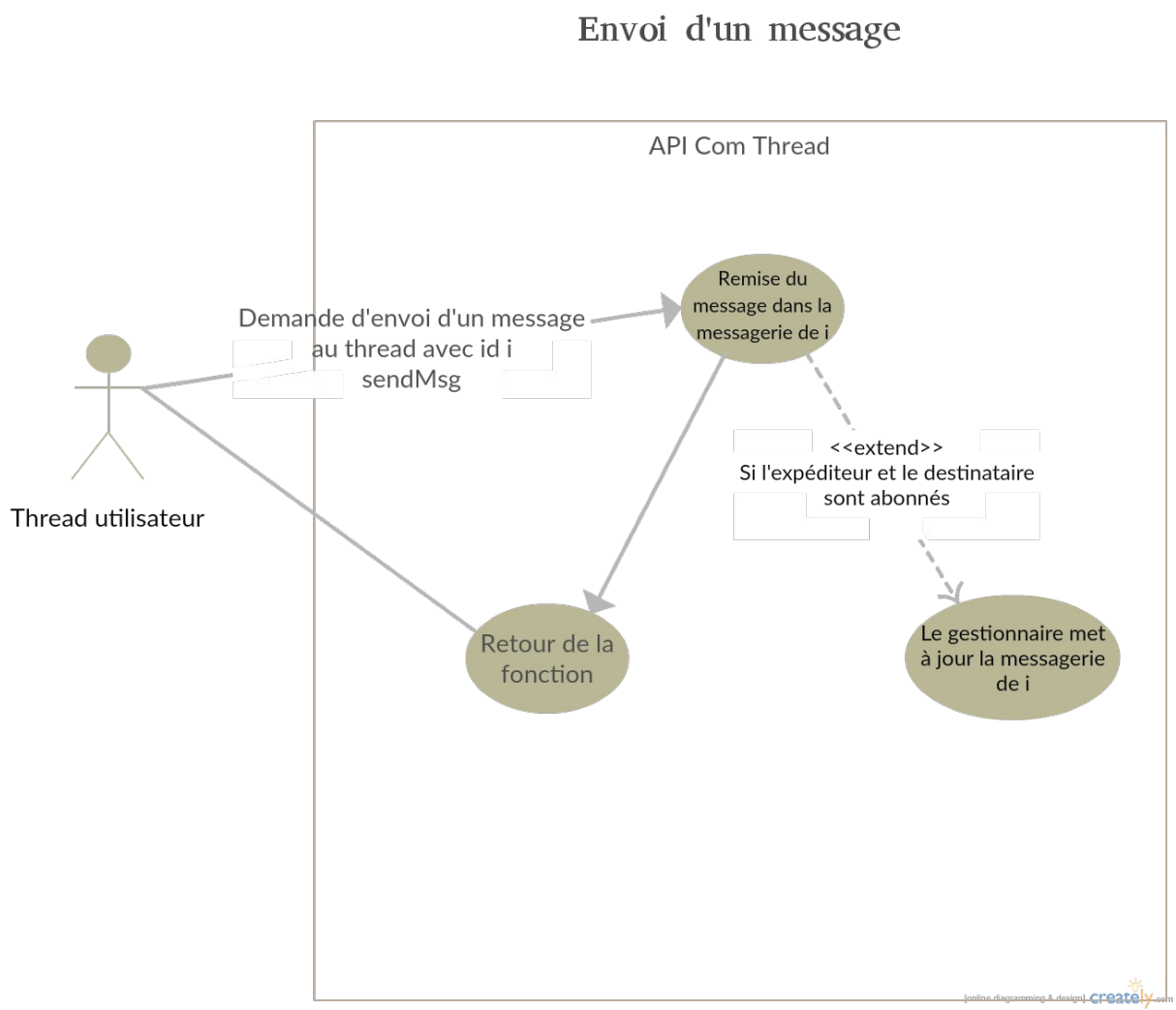


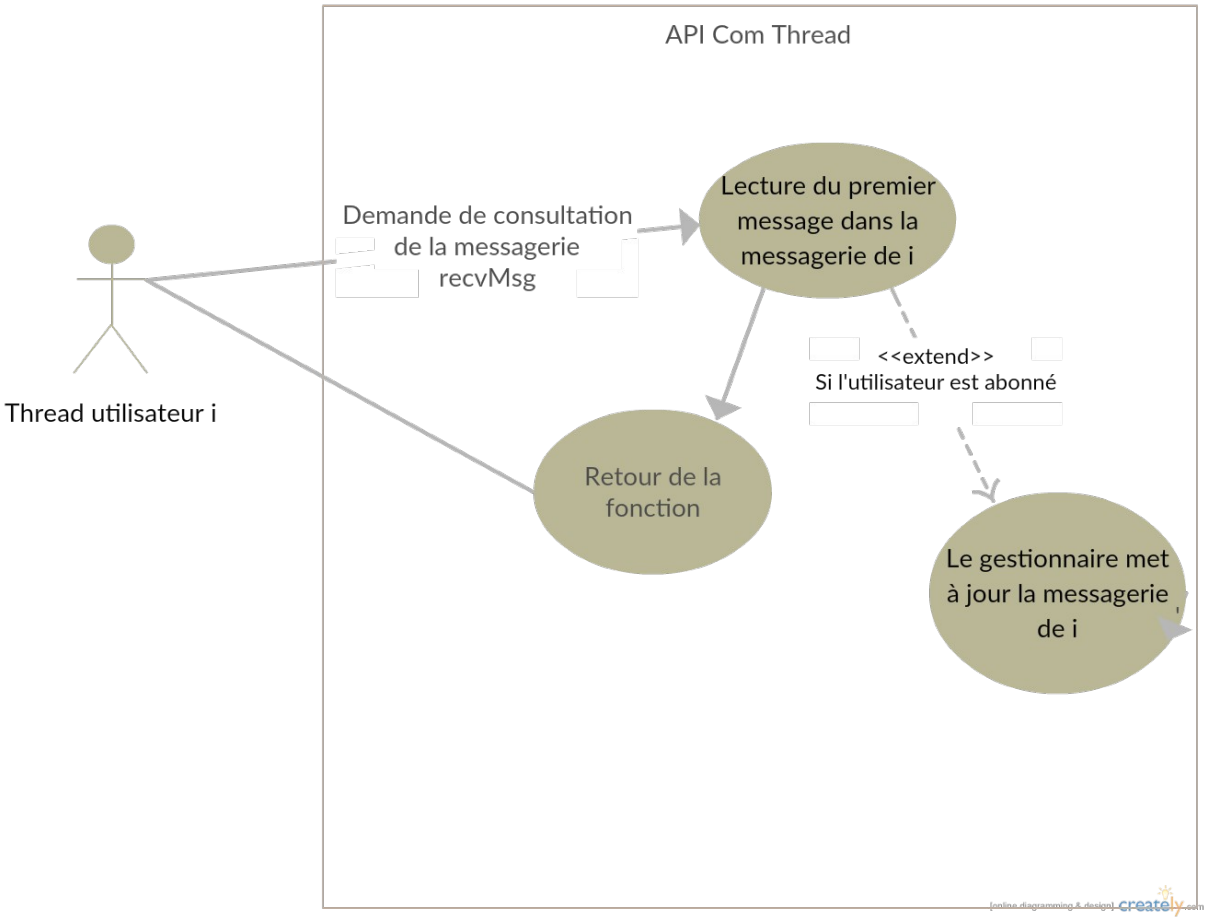
Illustration 3: Use case envoi d'un message

4) Récupération d'un message

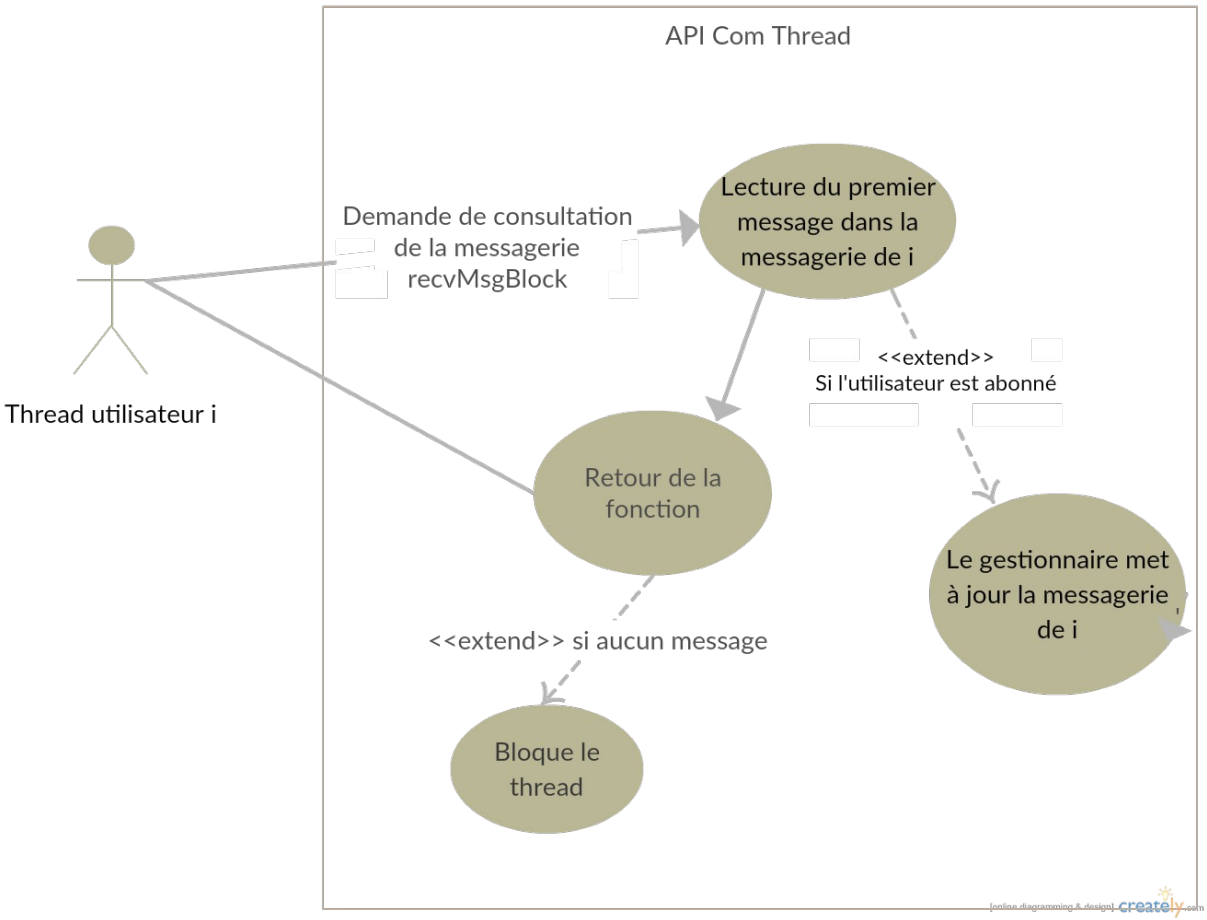
Dans le cahier des charges cette fonction avait deux utilisations et nous avons décidés de scinder cette fonction en deux et d'avoir une fonction pour chaque utilisation, qui sont : recevoir un message, et consulter le nombre de messages dans sa messagerie (recvMsg et getNbMsg).

Lorsqu'un utilisateur fait une requête de consultation de sa messagerie, nous avons également décidés de faire deux versions de cette fonction, une qui est bloquante et l'autre non (recvMsg et recvMsgBlock) quand il y a un message le gestionnaire renvoi le plus ancien message non lu quand la messagerie est vide la fonction recvMsgBlock retiens le thread tant qu'il n'a pas de message.

Réception d'un message version non blocante



Réception d'un message version blocante



5) Consultation du nombre de messages dans la messagerie

Un thread doit pouvoir consulter le nombre de messages en attente dans sa messagerie sans effectuer de retrait, dans ce cas la fonction n'est pas bloquante et retourne le nombre de messages en attente dans la boite aux lettres

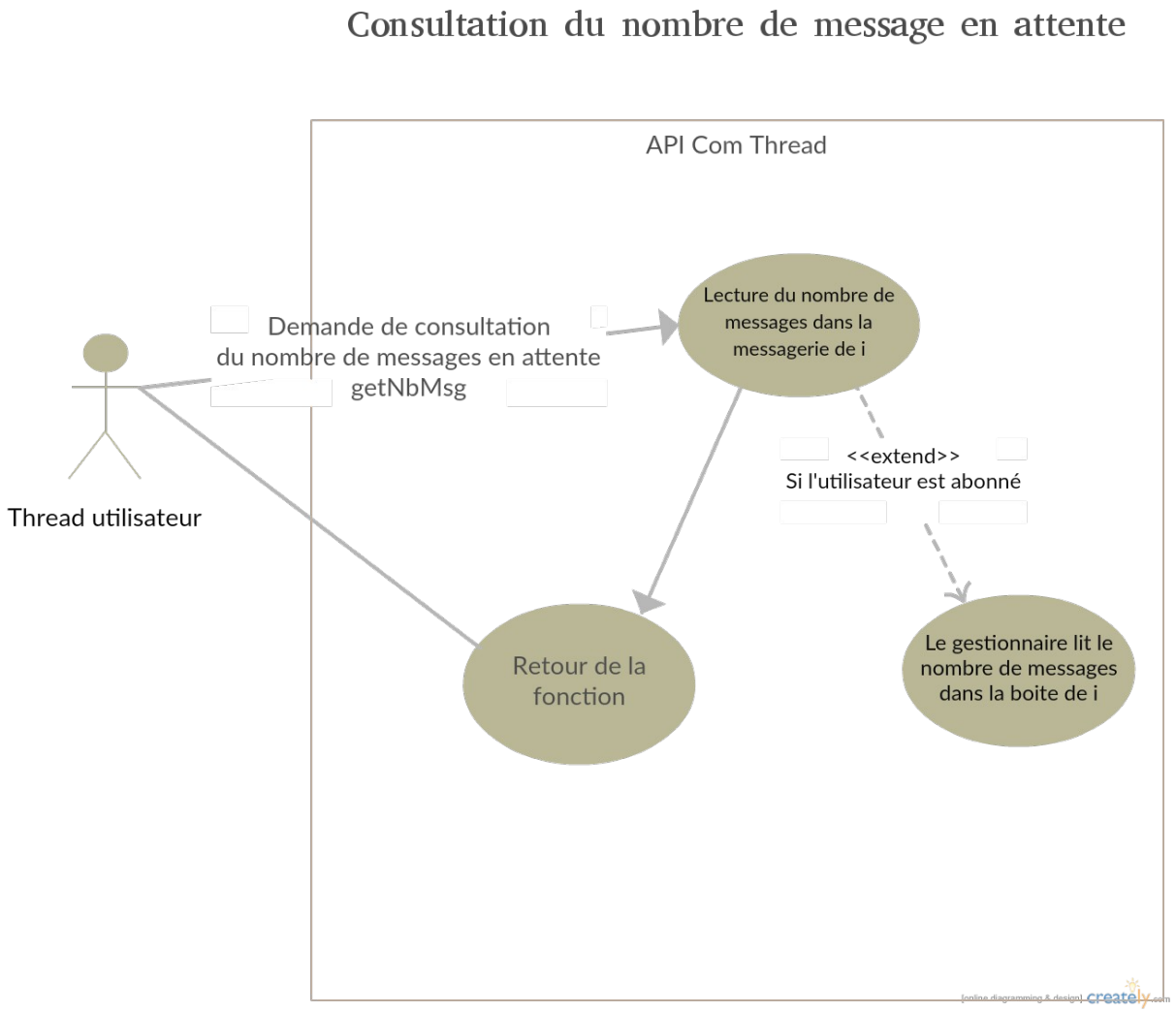


Illustration 4: Use case lecture du nombre de messages en attente

6) Désabonnement au gestionnaire

Un thread doit pouvoir désabonner un de ses identifiants auprès du gestionnaire, l'identifiant correspondant ne pourra plus recevoir de messages, si sa messagerie n'est pas vide lors de la demande les messages seront alors perdus.

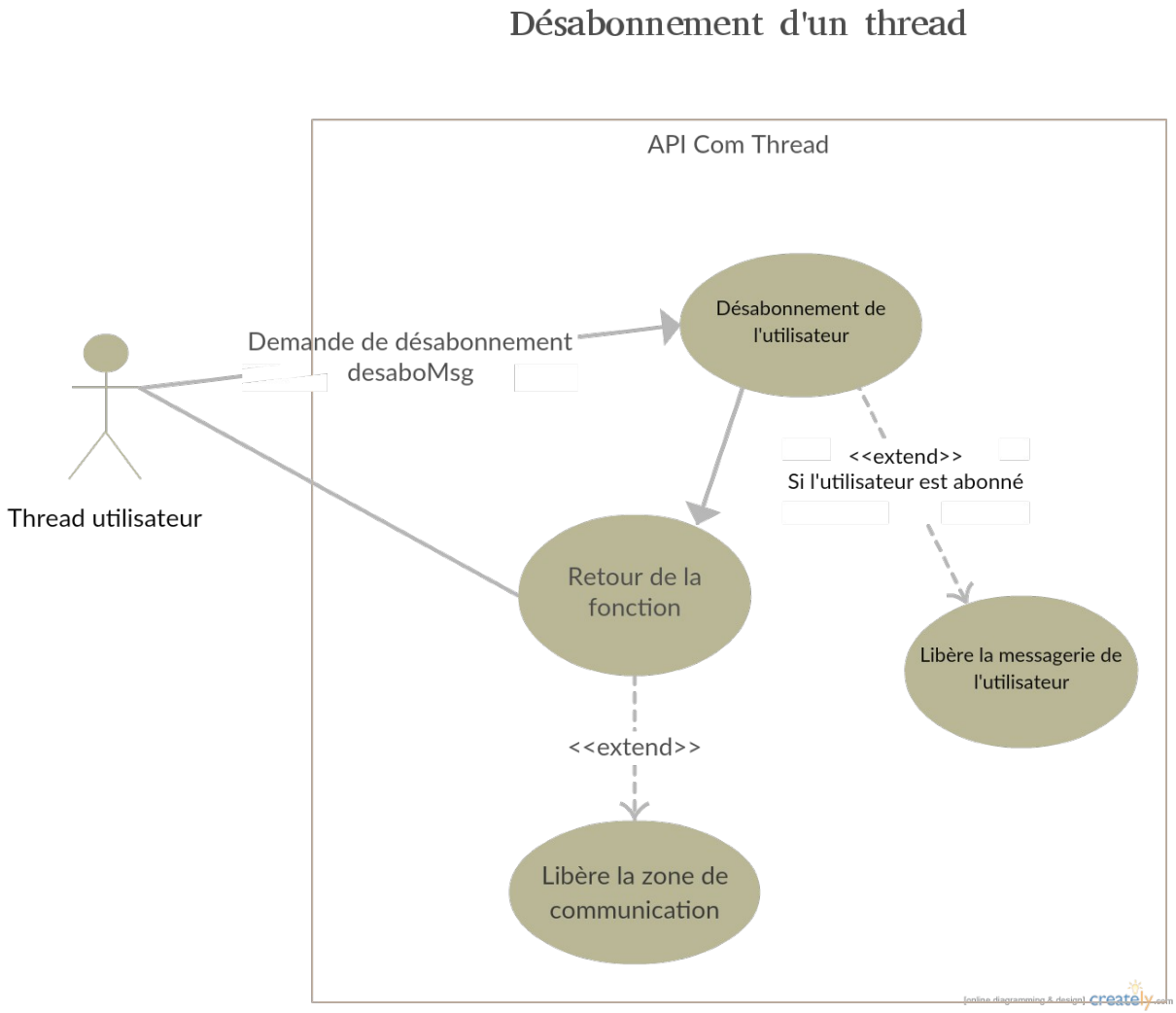


Illustration 5: Use case désabonnement d'un thread

7) Terminaison du gestionnaire

La terminaison du service doit pouvoir se faire de deux manières, la méthode « normale » qui ne peut marcher que si tous les utilisateurs sont désabonnés du service, et une méthode forcée qui désabonne automatiquement tous les utilisateurs.

Cette fonction doit bien libérer tous les espaces mémoires utilisés par l'API.

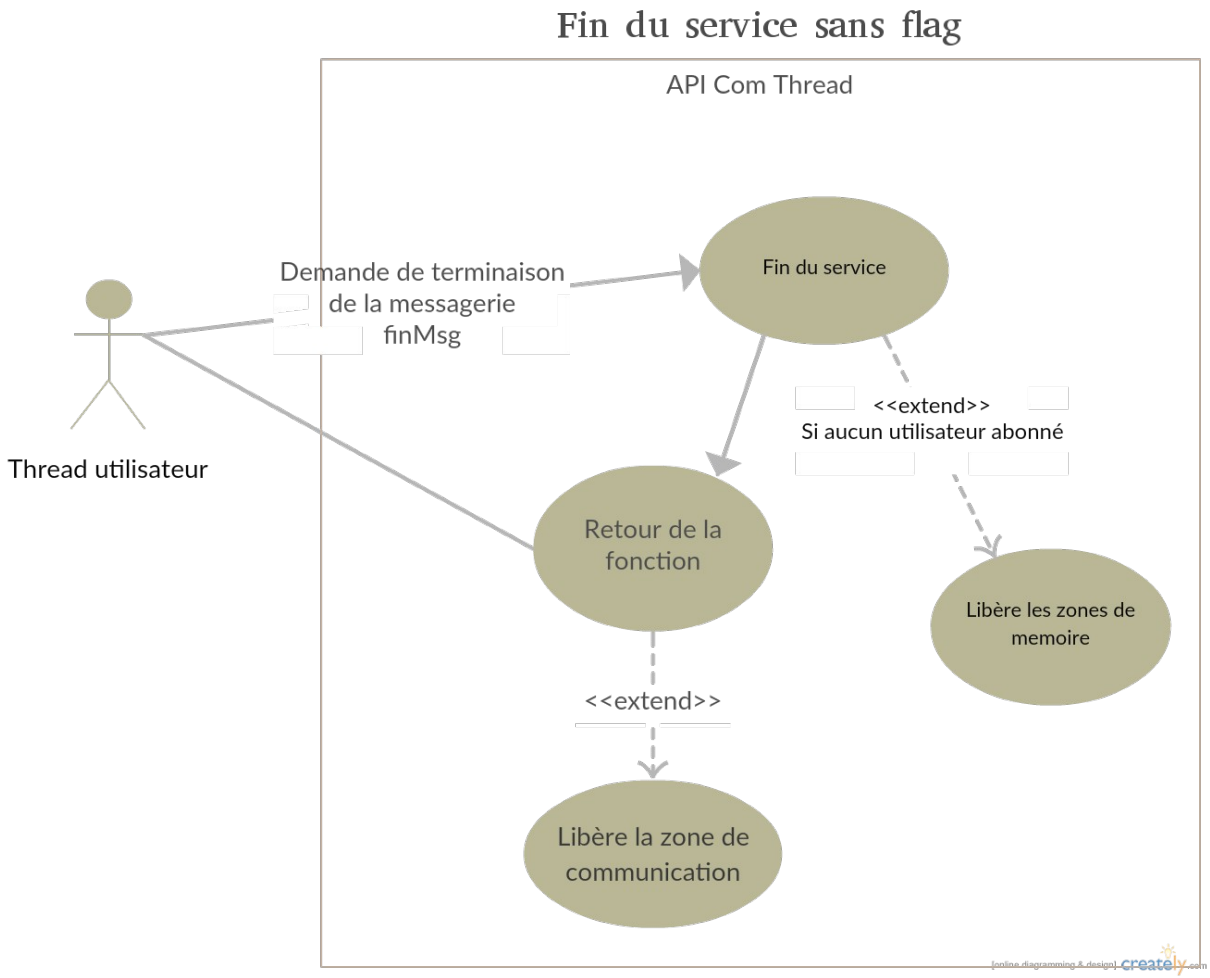


Illustration 6: Use case fin du gestionnaire méthode normale

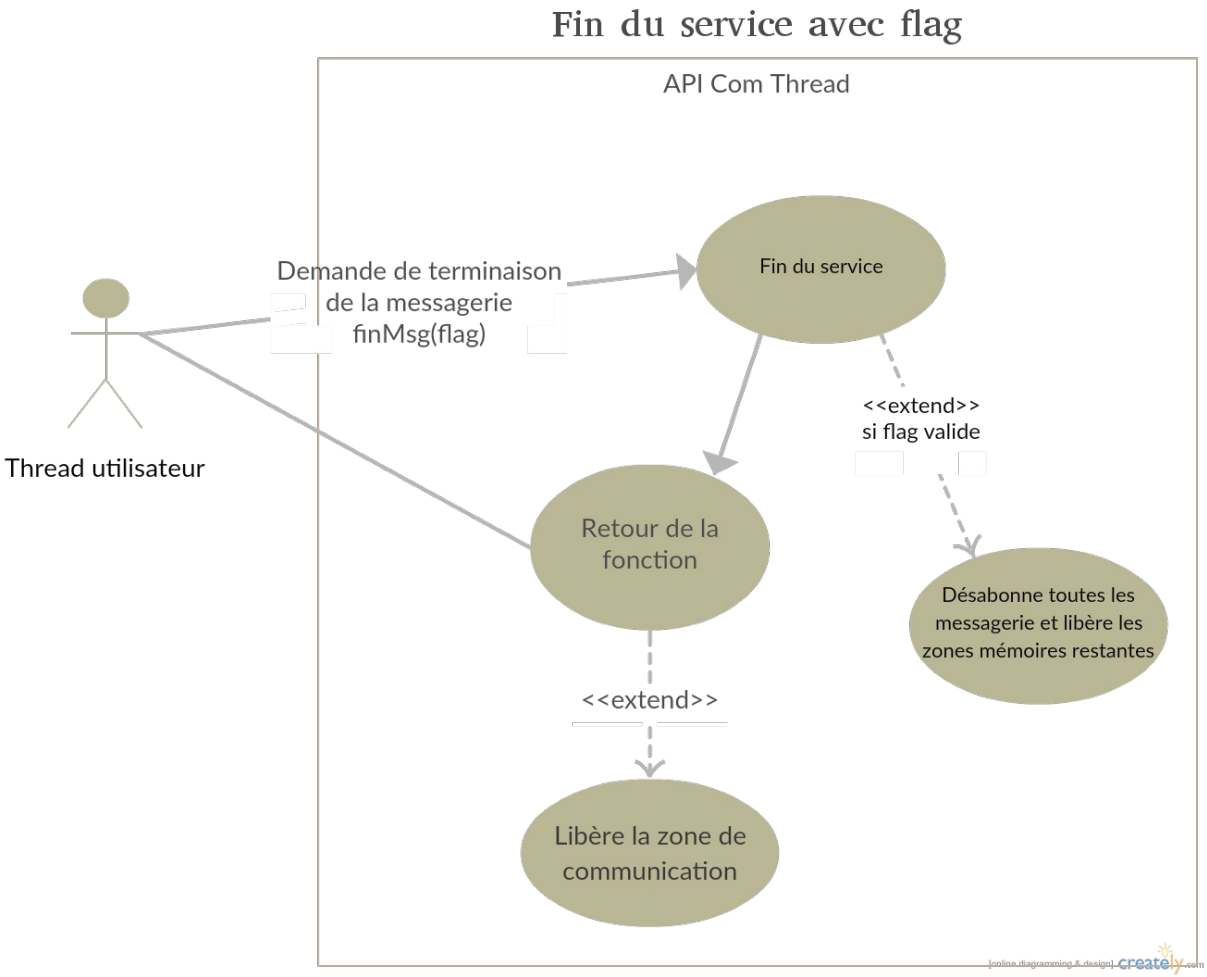


Illustration 7: Use case fin du gestionnaire méthode brutale

2. Conception et étude technique

Dans cette partie nous allons décrire comment utiliser les fonctions de l'API accessible à l'utilisateur et définir les codes d'erreur de ces fonctions. Ainsi que les choix techniques effectués pour la structure et les fonctions et discuter du choix de ceux-ci.

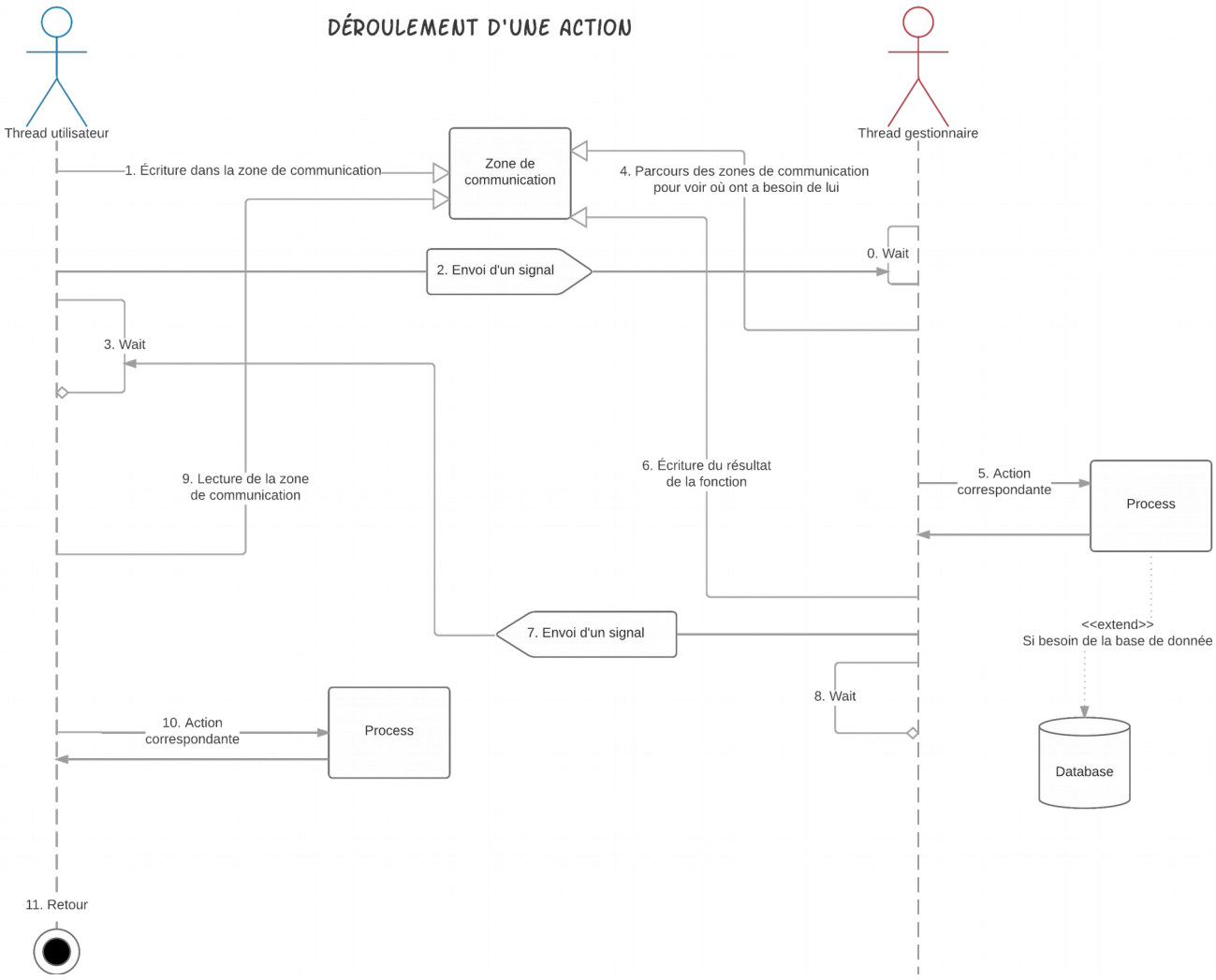


Illustration 8: Déroulement d'une action

L'illustration ci-dessus représente comment se déroule une action, toutes les fonctions suivent ce déroulement similaire.

2.1 Comment utiliser l'API ?

Notre API contient 8 fonctions utilisables par les clients:

1) initMsg

Description :Cette fonction permet de lancer le service et il doit être appelé une fois avant tout autre fonction. Cela crée le thread gestionnaire ainsi que les zones de stockage.

Construction: int initMsg (int nb_abo_max) ;

- nb_abo_max représentant le nombre max d'abonnés souhaité.

Retour :

- SUCCESS = 0
- ALREADY_LAUNCH = 1
- INIT_ERROR = 2
- TECH_ERROR = 10

2) aboMsg

Description : Cette fonction permet à un thread de s'abonner au service pour ensuite utiliser la messagerie (envoyer et recevoir des messages). Pour s'abonner il y a en variable globale une zone de communication utilisée juste pour l'abonnement des clients et un flag (_abo_traite) qui permet de savoir si ce qu'il y a dans cette zone peut être effacé ou non.

Construction: int aboMsg(communication * mycom, int id);

- mycom : lorsqu'un utilisateur veut s'abonner au service il doit allouer une zone de **communication** et passer un pointeur sur cette zone, le gestionnaire s'occupera de renseigner les paramètres comme l'identifiant
- id : identifiant souhaité par le client

Retour :

- SUCCESS = 0
- NO_SERVICE = 3
- ID_IN_USE = 4
- MAX_ABO = 5
- TECH_ERROR = 10

3) sendMsg

Description : Cette fonction permet à un client d'envoyer un message à un autre client, pour cela il faut passer à cette fonction plusieurs arguments :

Construction: int sendMsg(communication * mycom, int id_dest, void * contenu, int data_size);

- mycom : pointeur sur la structure **communication** de l'utilisateur, pour que la fonction puisse remplir les données au bon endroit, cela permet d'éviter de passer l'id d'un autre client
- id_dest : L'identifiant du destinataire
- contenu : Contenu du messager
- data_size : Taille du contenu du message, cette information est nécessaire du fait que le contenu du message n'est pas typé

Retour :

- SUCCESS = 0
- NO_SERVICE = 3
- NO_ABO = 6
- ID_UNKNOWN = 7
- TECH_ERROR = 10

4) recvMsg

Description : Cette fonction permet à un client de consulter sa messagerie et de récupérer le plus ancien message non lu.

Construction: int recvMsg(communication * mycom, message **msg);

- mycom : pointeur sur la structure **communication** de l'utilisateur
- msg : Pointeur sur un pointeur sur son message.

Retour :

- SUCCESS = 0
- NO_SERVICE = 3
- NO_ABO = 6

- NO_MSG = 8
- TECH_ERROR = 10
- INBOX_FULL = 12

5) recvMsgBlock

Description : Meme fonction que recvMsg sauf que celle-ci est bloquante si la messagerie est vide.

Construction : int recvMsgBlock(communication *, message **msg);

- mycom : Pointeur sur la structure **communication** de l'utilisateur
- msg : Pointeur sur un pointeur sur son message.

Retour :

- SUCCESS = 0
- INBOX_FULL = 2
- NO_SERVICE = 3
- NO_ABO = 6
- NO_MSG = 8
- TECH_ERROR = 10

6) getNbMsg

Description : Cette fonction permet à un client de connaître le nombre de messages non lus dans sa messagerie.

Construction : int getNbMsg(communication * mycom, int* nb_msg);

- mycom : Pointeur sur la structure **communication** du client
- nb_msg : Pointeur où le client pourra lire le nombre de messages non lus

Retour :

- SUCCESS = 0
- NO_SERVICE = 3
- NO_ABO = 6
- TECH_ERROR = 10

7) desaboMsg

Description : Cette fonction permet à un client de se désabonner du service

Construction: int desaboMsg(communication * mycom);

- mycom : Pointeur sur la structure **communication** du client

Retour :

- SUCCESS = 0
- NO_SERVICE = 3
- NO_ABO = 6
- TECH_ERROR = 10

8) finMsg

Description : Cette fonction permet à un thread (abonné ou non) de mettre fin à la messagerie, le flag permet de le faire de manière douce ou brutale

Construction : int finMsg (int flag);

- flag : 0 pour mettre fin à la messagerie de manière douce
  1 pour mettre fin à la messagerie de manière brutale

Retour :

- SUCCESS = 0
- NO_SERVICE = 3
- TECH_ERROR = 10
- STILL_REMAINS_ABOS = 9

2.2 Structure globale de l'API

Dans cette partie nous allons voir comment sont traités les différentes requêtes, comment sont protégés certaines zones et le cheminement des donnés.

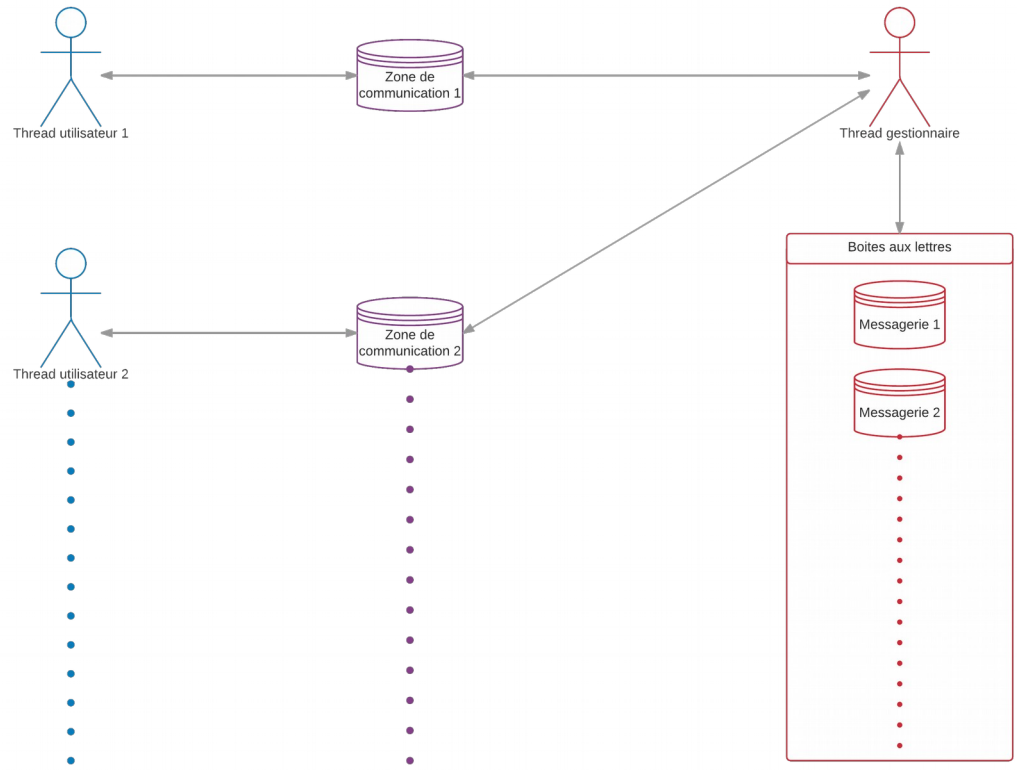


Illustration 9: Représentation des échanges

L'illustration ci-dessus représente de façon simpliste comment sont organisés les échanges dans l'API, chaque thread à une zone de communication partagée avec le thread gestionnaire et le gestionnaire à accès à toutes les messagerie et c'est le seul à y avoir accès.

1) Les variables globales

Nous allons décrire ici les différentes variables globales que l'on a définis pour que notre API fonctionne :

- pthread_mutex_t _mutex_abo : mutex utilisé pour protéger les variables globales utilisés pour l'abonnement mais également pour savoir si le gestionnaire est libre.
- pthread_cond_t _client_signal : condition qui permet de réveiller le gestionnaire.

- communication * _com_abo : pointeur vers une **communication** utilisé lorsqu'un client veut s'abonner au service, ce pointeur est à **NULL** si la dernière demande d'abonnement à été traitée.
- pthread_t * _thread_gest : thread gestionnaire, est à **NULL** si le service n'est pas lancé
- int service_ready : Permet de savoir si le service est prêt
- int fin : flag qui permet au gestionnaire de savoir si la fin du service est demandée.
- int _nb_max_abo : nombre maximum d'abonnés défini par le thread qui à lancé le gestionnaire

Notre API à également quelques paramètres :

- NB_ABO_MAX : limite d'abonnés fixée par l'API
- NB_LETTER_MAX : nombre maximal de lettre dans une messagerie

2) Zones de communications

Ces zones sont partagées entre les clients et le gestionnaire, comme il peut y avoir un nombre variable de clients et qu'il y a une zone de communication par client il a un tableau de pointeur sur ces zones communications qui est globale et le client doit allouer cette zone quand il s'abonne. Ces variables globales sont protégés par un mutex.

Elle est définie sous forme d'une structure ***communication*** qui comporte plusieurs paramètres :

- int client_id :L'identifiant du client
- int opération : Le code opération de l'opération souhaitée qui peut être
  - 0 : Qui représente aucune opération
  - 1 : Pour envoyer un message
  - 2 : Pour recevoir un messager
  - 3 : Pour avoir le nombre de messages dans la messagerie
  - 4 : Pour se désabonner
  - 7 : Pour mettre fin au service
  - 8 : Pour mettre fin au service de manière forcée
  - 9 : Pour s'abonner au service

Ces codes seront définis par une enumération qui reprendra tous les codes opérations

- int retour : Le code de retour de la dernière fonction réalisée qui peuvent être
    - 0 : SUCCESS
    - 1 : ALREADY_LAUNCH
    - 2 : INBOX_FULL
    - 3 : NO_SERVICE
    - 4 : ID_IN_USE
    - 5 : MAX_ABO
    - 6 : NO_ABO
    - 7 : ID_UNKNOWN
    - 8 : NO_MSG
    - 9: STILL_REMAINS_ABOS
    - 10 : TECH_ERROR
- int dest_id : L'identifiant du destinataire
- void * contenu : le contenu d'un message qui peut être de n'importe quel type cette donnée peut être destinée au gestionnaire ou au client suivant la fonction appelée
- pthread_mutex_t * mutex : Un mutex protégeant la zone de requête
- pthread_cond_t * signal_gestionnaire : permettant d'envoyer un signal au gestionnaire et lui dire qu'il y a une requête pour lui.

La zone de requête sera renseignée par les fonctions de l'API quand un client fait une demande et par le thread gestionnaire pour remettre les informations de retour d'une fonction utilisée par un client.

Nous avons choisi de faire une zone de communication par client, cela permet d'éviter des erreurs dans l'identifiant fournis car un thread ne peut pas connaître les pointeurs sur communication d'autres threads

3) Zone de stockage

La zone de stockage est une zone qui comporte toutes les messageries des clients, cette zone est uniquement connue par le gestionnaire.

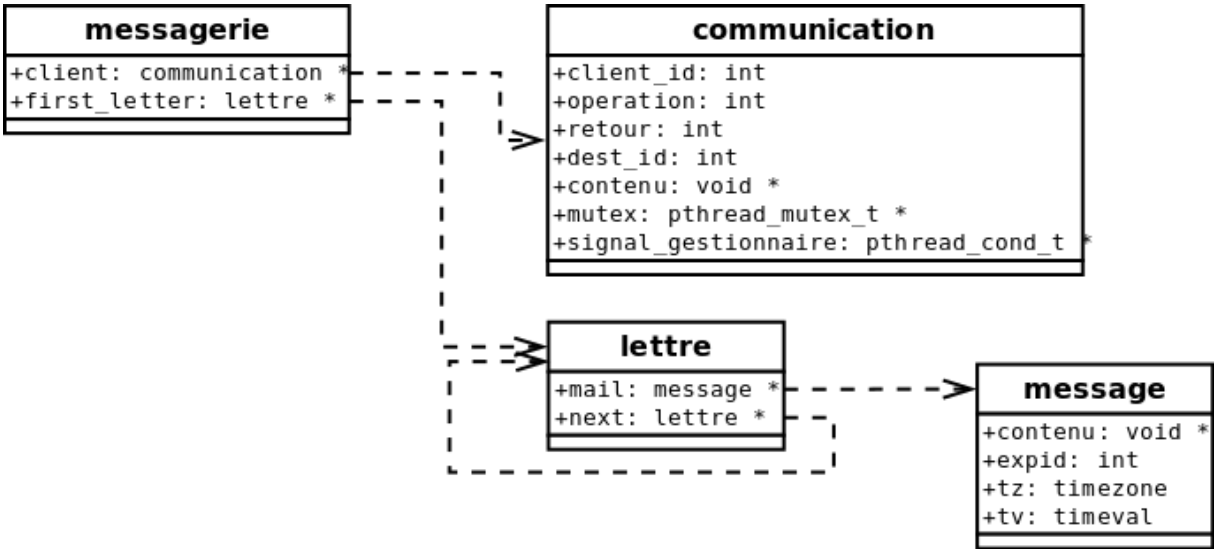


Illustration 10: Diagramme de classes messagerie

Le gestionnaire tient à jour un tableau de **messageries**. Chaque messagerie correspond à un abonné. La structure **messagerie** contient un pointeur sur **communication** et un pointeur sur la première **lettre** destinée a l'abonné dont c'est la messagerie. Chaque lettre contient un pointeur sur **message** et un pointeur sur la **lettre** suivante. Cela forme donc une liste chaînée.

Un message est constitué d'un pointeur sur la donnée que le message transporte, de l'identifiant de l'expéditeur du message, et de la date d'envoi du message.

La structure communication est essentielle pour la communication entre les threads clients et le gestionnaire.

Elle contient :

- l'identifiant du client
- l'opération que le client désire effectuer
- le code de retour du gestionnaire
- l'identifiant du destinataire du message lors de l'envoi d'un message
- un pointeur vers la donnée que le client et le gestionnaire s'échangent
- un mutex protégeant ces données
- une condition servant à réveiller le client.

3. Dossier de test

3.1 Tests nominaux

Dans un premier temps nous testons chaque fonction de l'API avec des tests unitaires pour vérifier que chaque message ou requête suit le cheminement souhaité et donne le résultat attendu. Pour cela nous avons défini des appels à effectuer à chaque fonction et nous consulterons les résultats avec des affichages écran.

Initialisation :

- Appel de la fonction initialisation par un thread
- Appel de la fonction initialisation par plusieurs thread à des instants proches

Abonnement :

- Appel de cette fonction alors que le service n'est pas initialisé
- Abonnement de plusieurs threads avec des identifiants différents
- Abonnement de plusieurs threads avec des identifiants identiques
- Un thread s'abonne plusieurs fois avec des identifiants différents
- Un thread s'abonne plusieurs fois avec des identifiants identiques
- Appel de la fonction abonnement alors que le nombre maximal d'abonnés est atteins

Envoi d'un message :

- Appel de cette fonction alors que le service n'est pas lancé
- Appel de cette fonction alors que l'on est pas abonné
- Envoi d'un message à un identifiant inexistant
- Envoi d'un message à un identifiant existant
- Envoi de message alors que l'on est pas abonné
- Envoi d'un message à un destinataire qui a une boite pleine
- Envoi d'un message en broadcast

Réception de messages :

Pour cette partie chaque test sera à effectuer avec les deux fonctions de réception (recvMsg et recvMsgBlock)

- Appel de cette fonction alors que le service n'est pas lancé
- Appel de cette fonction par un thread non abonné
- Réception d'un message alors que la boite est vide

Lecture du nombre de messages non lus :

- Appel de cette fonction alors que le service n'est pas lancé
- Appel de cette fonction par un thread non abonné
- Appel de cette fonction par un thread abonné

Désabonnement :

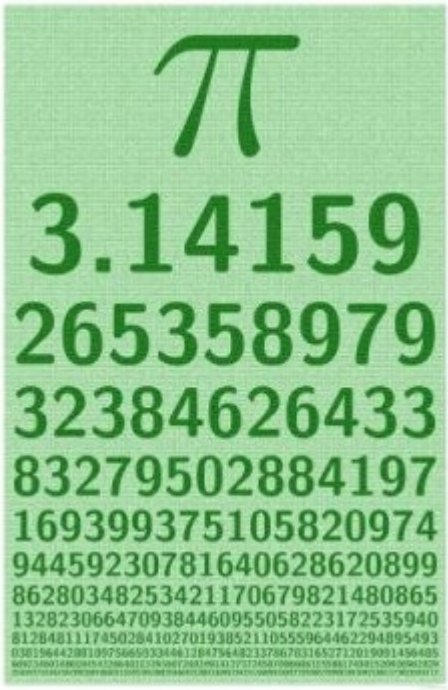
- Appel de cette fonction alors que le service n'est pas lancé
- Appel de cette fonction par un thread non abonné
- Appel de cette fonction par un thread abonné

Arrêt du service :

- Appel de cette fonction alors que le service n'est pas lancé
- Appel de cette fonction sans le flag alors que le service est toujours en utilisation
- Appel de cette fonction sans le flag alors que le service n'est plus utilisé
- Appel de cette fonction avec le flag

3.2 Scénario de test

Les scénarios de tests ont pour but de mettre en avant des cas d'utilisation se rapprochant plus d'un contexte d'utilisation « réel ». Il ne suffit pas de tester chaque fonction seule pour garantir le bon fonctionnement de l'API pour cela nous avons défini des scénarios se basant sur des événements pouvant survenir lors de l'utilisation de cette API. Nous allons utiliser l'API pour une tâche simple, calculer les décimales de pi.



Nous allons prendre une série qui converge très lentement car ici le but n'est pas de calculer précisément et très profondément les décimales, mais de mettre en avant le bon fonctionnement de l'API.

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} + \frac{1}{13} - \frac{1}{15} + \frac{1}{17} - \frac{1}{19} + \dots$$

Nous allons créer une fonction qui crée cinq thread, un de ceux-là sera utilisé pour l'affichage des résultats et les quatre autres servirons aux calculs.

Ce scénario pourra mettre en avant plusieurs contextes :

- Bonne communication entre les thread, on pourra imaginer que les threads s'envoient des messages pour connaître l'avancement du calcul et savoir quelles valeurs calculer.
- Simulation de congestion d'une boite aux lettres en effectuant des envois rapprochés de la part des threads de calcul ou un temps de lecture long de la part du thread de calcul.

4. Résultat des tests

Légende :

- **OK** : Test réussi
- **A** : Anomalie dans le cas testé
- **T** : Testé
- **NT** : Non testé
- **ATT** : En attente

№ du test	Fonction testée	Cas	État d'avancement			Résultats (OK / A)	Remarque
			T	NT	ATT		
1	**initMsg**	Appel par un thread	X			OK	
2		Appel par plusieurs thread	X			OK	
3	**aboMsg**	Appel alors que le service n'est pas initialisé	X			OK	
4		Abonnement de plusieurs thread avec id différents	X			A	
5		Abonnement de plusieurs thread avec id identiques		X			
6		Un thread s'abonne plusieurs fois avec des identifiants différents		X			
7		Un thread s'abonne plusieurs fois avec des identifiants identiques		X			
8		Appel de la fonction abonnement alors que le nombre maximal d'abonnés est atteins		X			
9	**sendMsg**	Appel de cette fonction alors que le service n'est pas lancé		X			
10		Appel de cette fonction alors que l'on est pas abonné		X			
11		Envoi d'un message à un identifiant inexistant		X			
12		Envoi d'un message à un identifiant existant		X			
13		Envoi de message alors que l'on est pas abonné		X			
14		Envoi d'un message à un destinataire qui a une boite pleine		X			
15		Envoi d'un message en broadcast		X			

№ du test	Fonction testée	Cas	État d'avancement			Résultats (OK / A)	Remarque
			T	NT	ATT		
17		Appel de cette fonction par un thread non abonné		X			
18		Réception d'un message alors que la boite est vide		X			
19	recvMsgBlock	Appel de cette fonction alors que le service n'est pas lancé		X			
20		Appel de cette fonction par un thread non abonné		X			
21		Réception d'un message alors que la boite est vide		X			
22	getNbMsg	Appel de cette fonction alors que le service n'est pas lancé		X			
23		Appel de cette fonction par un thread non abonné		X			
24		Appel de cette fonction par un thread abonné		X			
25	desaboMsg	Appel de cette fonction alors que le service n'est pas lancé		X			
26		Appel de cette fonction par un thread non abonné		X			
27		Appel de cette fonction par un thread abonné		X			
28	finMsg	Appel de cette fonction alors que le service n'est pas lancé		X			
29		Appel de cette fonction sans le flag alors que le service est toujours en utilisation		X			
30		Appel de cette fonction sans le flag alors que le service n'est plus utilisé		X			
31		Appel de cette fonction avec le flag		X			

Tableau 1: Résultats des tests nominaux

5. Conclusion

Pour le moment, nous avons fini l'architecture de l'application et coder la plupart des fonctions. Mais nous avons récemment rencontré un souci sur notre ancienne architecture et cela à remis en cause beaucoup de fonctionnement, ce problème devrait être résolu pour la remise finale du projet.

Lors de la remise du code nous joindrons une annexe dans laquelle se trouvera les éventuelles fonctions ajoutés et modifications, le tableau de résultat des tests nominaux mis à jour ainsi que les résultats des tests sur le scénario de calcul des décimales de pi. Nous ajouterons également un bilan global sur le projet.