CSC 4509 : Conception d'un Chat multi-serveurs

Par Benoit Tellier.

Fonctionnalités et exigeances

Nous cherchons à réaliser un Chat distribué avec plusieurs serveurs.

Un client pourra effectuer les actions suivantes :

- Se connecter à un serveurs en spécifiant un pseudo
- Envoyer un message à l'ensemble des personnes présentent sur le Chat
- Obtenir la liste des clients connectés sur notre réseau
- Obtenir la liste des serveurs connectés
- Envoyer un message à un, et un seul des clients connectés (message privé)
- établir une connection de spare avec un autre serveur, sur laquelle on basculera en cas de nécéssité.

Nous ferons attention aussi aux points suivants :

- Quand un client peut dialoguer avec un client avec qui il ne pouvait pas dialoguer avant, il reçoit une notification indiquant qu'il peut dialoguer avec ce client
- Inversement quand on ne peut plus dialoguer avec un client, on reçoit un message nous l'indicant.

De plus, un serveurs peu demander l'accès en exclusion mutuelle à une ressource de logging. Pour se faire il utilise un algorithme d'exclusion à base de jeton. Cet algorithme nous restraint au niveau topologie (c'est la raison pour laquelle nous ne pouvons pas fusionner deux réseaux).

Enfin, le serveur maitre peut demander l'arrêt de l'ensemble des serveurs, ce qui sera effectué une fois l'ensemble des clients bloqués en emmission, et l'ensemble des messages emis écoulés.

Les points mentionnés ci dessus surviennent lors des connections / déconnection de client ou de serveurs.

Nous souhaitons, autant que possible, que notre architecture supporte les changements de topologie aux niveau des serveurs (ajout ou départ d'un ou plusieurs serveurs). Voici les conditions simplificatrices dans lesquels nous nous placerons concernant ce problème :

1. Aucun changement de topologie n'est effectué pendant un changement de topologie
2. Aucun message n'est C broadcasté pendant un changement de topologie
3. Aucun C broadcast n'est en cours au moment du changement de topologie.
4. Il n'y a pas de changement topologique pendant la détection de terminaison lié à l'arrêt de notre application répartie.
5. Pas de transfert de jeton d'exclusion mutuelle pendant un changement de topologie.

De plus nous supportons les changements topologiques suivants :

1. Perte d'un serveur (et éventuel partitionnement de notre ensemble de serveurs)
2. Ajout des serveurs un par un.

Nous ne supportons

Nous porterons aussi une attention particulière à la détection et au traitement des erreurs réseau (on adopte un politique assez sévère...)

Commentaires sur les algorithmes

Élection

Question 1

L'algorithme d'élection est à insérer dans le code du serveur.

Le serveur a besoin d'annalyser le contenu des messages, pour notamment savoir si le message reçu est un Jeton d'élection ou un jeton gagnant.

Les clients n'ont pas à être importunés avec les messages d'élection. (Ils ne participent pas).

Question 2

L'algorithme de vague écho est écrit sous la forme orientée controle dans notre polycopié.

Le voici sous sa forme évenement :

```
Init:
Init p : #p, Init rec, lrec : 0, Init state : indecis, Init father, win, caw : Undefined

Pour lancer l'élection
# some init
Init win, father : undefined
Init rec, lrec : 0
Init state : indecis
send Jeton( p ) to all neightbours


On reception( m )
if m contain Jeton( r )
    if caw == undefined
        #First message received by a non candidate process. We need to do some init
        # init for new wave
        Init rec, lrec : 0, Init state : indecis, Init father, win : Undefined
    endif
    if caw == undefined || r < p
        caw = r
        rec = 0
        father = emmeteur( m )
        send m to all neightbours except father
    endif
    if caw == r
        increment rec
        if rec == #Neigh
            if caw == p
                send GAGNANT( p ) to all neightbours
            else
                send Jeton( r ) to father
            endif
        endif
    endif
else
    # m contain Gagnant( r )
    if lrec < #Neigh
        if lrec == 0 || r !=p
        send GAGNANT( r ) to all neightbours
    endif
    lrec ++
    win = r
    if lrec == #Neigh
        if win == p
            state = gagnant
            launch actions
        else
            state = perdant
        endif
        # to allow us to launch a new election
        caw = undefined
    endif
endif
```

Nous échangeons des messages de la classe ElectionToken. Nous savons que nous avons à faire à un message ElectionToken car le message est précédé de 4 octets (ajoutés à l'envoie) qui précisent son type.

Les actions sont lancés par le gagnant en fin d'élection.

Dans notre cas, il lui incombe d'informer chaque serveurs des autres serveurs présents dans la topologie, ainsi que de collecter la liste des pseudos afin de pouvoir envoyer les join / leave notifications liées au changement de topologie.

Question 3

Voici nos variables :

- rec : nombre de jetons reçus pour la vague en cours
- caw : identifiant du processus de la vague en cours
- lrec : nombre de réponses reçues pour la vague gagnante
- state : état électoral du serveur
- win : identifiant du vainqueur
- father : un moyen de communication avec son parent direct
- #Neigh : nombre de voisins directs

Question 4

L'algorithme est lancé quand :

- On le demande explicitement sur l'entrée clavier du serveur
- On a un changement de notre topologie serveur : ajout ou retrait d'un ou plusieurs serveurs

Question 5

Nous nous basons sur l'adresse du socket du serveur pour identifier nos serveurs de manière unique sur notre réseau. Pour comparer ces adresses de socket, nous comparons simplement la chaine de caractère obtenue par l'appel à la méthode toString().

Le gagnant est celui qui a la plus grande chaine de caractère (au sens de compareTo()).

Question 6

Il existe une méthode d'envoie de messages : NetManager.sendInterServerMessage

Il existe une méthode permettant d'envoyer un message à tous nos voisins : State.broadcastInterServerMessage

Il existe une méthode pour envoyer un message à tous nos voisins sauf un : State.broadcastInterServerMessageWithoutFather

Les deux dernières méthodes s'appuient sur la première.

Pour gérer l'accès à la machine à état en exclusion mutuelle, nous allons identifier les methodes appelé via les commandes clavier, et protéger les ressources accédées via des mutex.

Question 7

#Neigh est récupéré via la méthode State.getNbConnectedServers() qui récupère la cardinalité de l'ensemble des serveurs auquels nous sommes connectés.

Question 8

J'autorise plusieurs éxécutions de l'algorithme.

Question 9

Je l'ai testé sur les topologies suivantes :

Je relance une élection à chaque changement de topologie. Il suffit de déconnecter un serveur, par exemple dans la topologie triangle, et les voisins lancent alors une élection en même temps (donc 2 candidats).

Diffusion

Notre C diffusion s'appuie sur notre classe **CBroadcastManager**.

Question 1

Cet algorithme est à implémenter sur les serveurs seulement. Il permet de faire passer les évenements qu'un serveur

cherche à avertir les autres. Les clients reçoivent le dit évènement par l'intermédiaire de leur serveur si ils sont concernés.

Question 2

La ligne ** (p = q) ** signifie que ce serveur est à l'origine de ce message. Nous l'acceptons directement.

La ligne **v[q] = vp[q] + 1)** permet de vérifier que c'est bien le message suivant pour le noeud initiateur.

La ligne **v[k] <= vp[k]** permet de vérifier que nous n'avons pas besoin de délivrer des messages d'autres noeuds avant celui ci.

Question 3

Le vecteur d'orloge de chaque machine est contenu dans le CBroadcastManager. Il est transmis dans le champ **NeededData** des messages interserveurs.

Note : Le vecteur d'orloge est réinitialisé à chaque changement de topologie. La logique aurait été de les resynchroniser via un Echo, puis de les mettre à jour sur l'ensemble des noeuds à l'aide d'une diffusion fiable mais le temps me manque, et compte tenu de nos hypothèses de changement de topologie, ce scénario est largement acceptable.

Question 4

Nous avons deux autres structures de données :

- **messageBag** : collection de messages non encore C acceptés
- **cAcceptedMessages** : collection ordonée de messages c acceptés. L'idée est ici de respecter l'interface BroadcastManager, dont la méthode process input indique si au moins un message a été accepté. Il est de la responsabilité de l'utilisateur, en cas de message(s) C acceptés de les lire via la méthode **ArrayList getCAcceptedMessages()**

Les nouveaux messages sont passés via la méthode **processInput**.

Question 5

Les lignes 5 - 7 de l'algorithme correspondent à la méthode **public void launchBroadcast(InterServerMessage);**

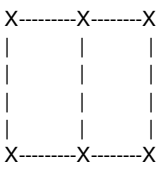
Question 6

Les lignes 8 - 13 de l'algorithme correspondent à la méthode **public Boolean manageInput(InterServerMessage message)**

Question 7

Dans mon cas il n'y a malheureusement pas encore de méthodes implémenter pour celà. Devrais être implémenté prochainement.

Une double boucle, par exemple un huit, devrais aider à observer des non respects de l'ordre causal. Par exemple :



Exclusion mutuelle

Question 1

Ce sont nos serveurs qui accèdent à une ressource de manière concurente. Ce sont donc eux qui doivent implémenter un mécanisme les protégeant des accès concurents. Les clients n'ont pas ce genre de problématiques, nous n'avons donc pas besoin de mettre en place ce genre de solutions pour eux.

Question 2

L'algorithme est écrit en suivant la convention orientée évenements.

Question 3

Les serveurs échangent des messages de type InterServerMessage, RBroadcasté (donc le champ type est à 3), soit

transportant une demande d'entrée en session critique (subtype = 8, avec nsp comme champ message, et l'identité du processus dans le champs identifier), soit un transfert de jeton (subtype=9, le jeton est dans le champ message, et le processus a qui est destiné le jeton est situé dans le champs NeededData).

Question 4

La demande de jeton se fait à l'aide de la méthode **askLock** du LockManager. Nous pouvons la lancer depuis le clavier, une commande est prévue à cet effet.

Question 5

La topologie du réseau sous tendue par l'algorithme de Ricart et Agrawala est un graphe fortement connexe (chaque serveur est relié à chaque autre serveur). Néanmoins,nous pouvons nous passer de cette hypothèse si chaque message est broadcasté (quitte à identifier le serveur de destination sur un message destiné à un seul serveur : le serveur en question saura se reconnaitre et les autres réaliserons qu'il ne leur est pas dédié). Avec cette modification, nous n'avons plus de contraintes topologiques. C'est cette solution là que je vais mettre en place.

Question 6

Liste des variables de l'algorithme :

- **nsp** : L'estamplille de nos demandes de jeton.
- **dem** : Le compte pour chaque serveur des demandes reçues par chaque serveur.
- **jet** : Le compte des demandes satisfaites par le jeton. Ce n'est pas à proprement parlé une variable propre à un serveur, mais une valeur qui transite avec le jeton.

Question 7

Le jeton est donné au processus élu, qui n'est pas en session critique. Le fait d'utiliser le serveur élu nous garanti l'unicité du jeton. Par ailleurs, nous nous assurons qu'il y a bien un et un seul serveur élu. Il semble donc logique de se placer dans cette optique.

Question 8

Le jeton est un tableau qui, à chaque processus associe le nombre de fois ou ce processus a accédé cette ressource. Au début de l'éxécution, les valeurs associées à chaque processus sont initialisées à 0.

Question 9

A la réception du jeton, nous sommes les seuls à pouvoir accéder la ressource distante. Nous pouvons donc commencer à l'utiliser.

Question 10

Nos serveurs, compte tenu de nos exigeances, connaissent déjà l'ensemble des serveurs situés sur leur réseau. Ils capable de les identifier, et donc d'échanger des messages via RBroadcast sans ajout d'un nouveau sous protocole de notre part.

Question 11

L'ensemble des demandes est une table de hashage qui associe à un identifiant serveur le nombre de demandes qu'il a fait pour ce verrou.

Question 12

Quand il reçoit une demande, le processus ajoute une demande au processus qui l'a fait. Ensuite, si il n'est pas en session critique mais possède le jeton, il le redistribue en utilisant la méthode de sortie de session critique.

Question 13

Pour pouvoir tester l'exclusion mutuelle, nous pouvons effectuer des demandes de verrou de la part de plusieurs processus (au moins 3) et des relachement de verrou. Il faut tester les cas de figures suivants :

- Un serveur demande le verrou et personne n'est en session critique
- Un serveur demande le verrou alors qu'un autre serveur est en session critique. On vérifie qu'il obtient bien le jeton à la sortie de la session critique du premier processus.
- Deux serveurs demandent le verrou alors qu'un autre utilise les ressources. On vérifie bien que chacun libère la ressource et transmet le jeton au suivant.

- Il nous faut nous assurer aussi de l'hypothèse de vivacité. Pour se faire, on peut réaliser une séquence de ce style :
  - Lock 1
  - Lock 2
  - Lock 3
  - Release 1
  - Lock 1
  - Ici disons que 2 obtienne le jeton
  - Release 2
  - Ici 3 doit obtenir le jeton. Si ce n'est pas le cas alors nous aurons un problème avec la propriété de vivacité.

Notes complémentaires

Architecture permettant de personnaliser le comportement du LockManager

J'ai implémenté le patron de conception du visiteur (voir les classes ResourceVisitor et LogResourceVisitor) afin de pouvoir personaliser le comportement de notre lock manager. Maintenant, l'utilisateur de cette classe peut spécifier le comportement de cette classe sans avoir à l'hériter.

Ce patron de conception aurait pu être utilisé pour personaliser l'EchoManager. Je n'y ai pensé qu'après, mais cela s'y preterai bien...

Système de log

J'ai ajouté un système de gestion des logs. Je me suis appuyé sur **log4j** (http://logging.apache.org/log4j/1.2/).

Changement de topologie

En cas de changement de topologie, le serveur élu lance une vague pour savoir qui possède le jeton. Si personne ne possède de jetons, il en régénère un. Un serveur isolé ne peut posséder de jetons, il n'y a donc pas de risques à l'ajouter à notre topologie.

Si deux (ou plus) jetons sont introduits, le serveur élu demande immédiatement la sortie de session critique aux autres serveurs, demande la destruction des jetons, et enfin régénère un jeton. C'est un cas critique que nous ne devons pas rencontrer suite aux contraintes topologiques choisies.

Détection de terminaison

Question 1

On doit insérer le code de détection de terminaison dans le code du serveur.

Question 2

Les cas d'utilisation de notre algorithme de terminaison est le suivant :

On souhaite empêcher nos clients d'envoyer des messages supplémentaires. Nous souhaitons ensuite que l'esemble des messages clients soient délivrés. Une fois qu'ils auront été tous délivrés, nous éteindront nos serveurs.

Donc pour permettre ceci, il nous faut :

1. Empêcher que les clients génèrent de nouveaux messages entre nos serveurs
2. Attendre la terminaison de notre application (tous les messages des clients sont arrivés à destination). Pour se faire, nous utiliserons une variante de l'algorithme de Safra.
3. Une fois le point précédent terminé, nous pourrons en toute sécurité éteindre nos serveurs.

La condition **« forall process p, statep = passif »** signifie donc, avec ce qui précède

aucun serveur n'accepte plus de messages de client; tous nos serveurs sont en attente de messages de client

Le client détecte juste que le serveur n'est plus disponible et s'arrête. Il n'est pas nécéssaire d'ajouter un protocole supplémentaire étant donné que couper la connection coté serveur entraine l'arrêt du client.

La ligne 17 de l'algorithme permet de savoir si un message a été reçu entre deux réceptions du jeton. On réinitialise à la fin de tout traitement du jeton la couleur à **white** et elle sera basculé à **black** en cas de réception d'un message *utilisateur*. Il suffit ensuite, quand on reçoit le jeton, de lire la couleur du serveur pour savoir si un message a été reçu depuis le dernier passage. Pour l'implémenter, il suffit d'avoir une variable correspondant à la couleur dans notre classe chargée de la détection de terminaison. Nous passons sa valeur à **white** après chaque passage du jeton, et la passons à **black** lors de toute réception de message utilisateur.

La variable **statep** permet de savoir si notre serveur est actif (si on travaille sur des requêtes utilisateur). Il faut le basculer

en active quand on commence à traiter un message utilisateur, et le rebasculer à passive une fois que l'on a finit cette tache.

Question 3

La variable **mcp** correspond à la variable **messageCount** de ntre classe **EndManager**. Nous comptons dedans les messages reçus par nos clients, donc les join et leave notifications, et les messages privés et publiques transitant entre nos serveurs.

Mon implémentation est grandement simplifié, étant donné que ces messages sont les seuls à utiliser le broadcast manager causal. Ce sont donc les seuls du type InterServerMessage à être estampillé avec le type 5.

Question 4

Comme nous nous sommes fixé comme contrainte une topologie sous-jacente quelconque, nous ne pouvons faire d'hypothèses sur notre réseau. Nous n'avons donc aucune certitude de réussir à extraire un anneau de notre topologie. J'ai choisis de mettre en place un anneau virtuel en utilisant les propriétés suivantes :

- Chaque serveur a connaissance de la même liste ordonnée de l'ensemble des autres serveurs.
- Chaque serveur peut communiquer avec n'importe quel autre serveur par l'intermédiaire d'autre serveurs, par exemple en utilisant de la diffusion fiable, et en rajoutant un identifiant qui adresse ce message à l'autre serveur.

Nous pouvons ainsi recréer un anneau virtuel si à chaque serveur on adresse le message suivant au serveur suivant dans notre tableau d'identifiants serveurs ordonné. Il sera le seul à le recevoir via R Broadcast puis reconnaissance de son identifiant.

Cette technique permet de s'affranchir facilement de toute contrainte topologique mais génère une augmentation de la complexité en nombre de messages.

Question 5

Le processus **p0** est le processus responsable de la détection de terminaison. Nous choisirons **p0** comme étant le processus élu (ce qui nous assure et son existance et son unicité).

Question 6

Compte tenu de notre scénario d'extinction, nous appelons la méthode **startEndingDetection** (qui correspond aux ligne 8 et 9 de l'algorithme). Elle est appelée quand l'administrateur système souhaite éteindre proprement son infrastructure. Elle a pour responsabilité de rendre les clients inactifs (le serveur n'accepte plus leurs messages), et d'envoyer le premier jeton.

Le jeton circule à une vitesse raisonnable (une attente de 10 ms est observée à chaque envoie de jeton afin de soulager les CPUs).

Question 7

Comme indiqué précédemment, on réinitialise à la fin de tout traitement du jeton la couleur à **white** et elle sera basculé à **black** en cas de réception d'un message *utilisateur*. Il suffit ensuite, quand on reçoit le jeton, de lire la couleur du serveur pour savoir si un message a été reçu depuis le dernier passage. Pour l'implémenter, il suffit d'avoir une variable correspondant à la couleur dans notre classe chargée de la détection de terminaison. Nous passons sa valeur à **white** après chaque passage du jeton, et la passons à **black** lors de toute réception de message utilisateur.

Question 8

Avec ce qui précède, l'action **Sp** est à appeler une fois par nombre de C broadcast propagé (que ce soit au sein du RBroadcast manager ou au sein de l'envoi).

Question 9

L'action **Rp** est à déclencher à chaqe réception de message relié à du C Broadcast.

Question 10

L'action **Ip** doit se déclencher dès que le traitement d'un message C Broadcasté est finit.

Question 11

L'action **Tp** doit être réalisé dès que le jeton est reconnu et accepté par le serveur.

Question 12

J'ai testé mon implémentation sur les structures suivantes :

```
X
```

```
X------X
```

```
X----X
|    /
|  /
X
```

```
X----X----X
|    / |
|  /   |
X----X
```

Question 13

J'ai mis en place un message RBroadcasté qui déclenche l'arrêt de tous nos serveurs.

Récolte d'informations via Echo

Pourquoi cet algorithme

Lors d'un ajout de serveur à notre réseau, nous prenons le risque de connecter deux réseaux entre eux. Si tel est le cas, il nous faut un moyen pour retrouver la liste des pseudos des personnes connectées, ainsi que la liste des serveurs connectés. Enfin, il faut que le serveur élu puisse s'assurer qu'un seul jeton d'exclusion mutuelle existe sur son réseau.

D'où le besoin de cet algorithme.

principe

Le concept est qu'on va effectuer un écho pour collecter des données sur chacun des noeuds.

Chaque noeud chargera dans son message de broadcast (envoyé à tous sauf à l'éventuel parent) les informations qu'on lui demande de transmettre. Chaque noeud écoute les données des noeuds voisins et les accumule pour cet écho. Enfin, dès que chacun des noeuds lui a répondu (il a reçu autant de messages qu'il y a de noeuds auquel il est relié), il renvoie l'ensemble des données récoltées (en faisant attention à d'imancables doublons) à son parent si il n'est pas initiateur, sinon si il est initiateur, il a collecté l'ensemble des données des noeuds du réseau.

Algorithme

```
Init p = #p, seq = 0, rcvMsg[p][seq] = 0, waveData[p][seq] = [], father = null

if initiateur
    seq ++
    set message p and seq
    broadcast message to all neightbours
fsi
tant que rcvMsg[p][seq] < #neightbours
    receive message from q
    Merge message.data and waveData[p][seq]
    if first time we spot this echo
        father = q
        repleice message.data by our node datas
        broadcast message to all neightbours except q
    rcvMsg[p][seq]++
done
if is Initiator
    return waveData[p][seq]
else
    message.data = waveData[p][seq]
    send message to father
fsi
```

Complexité :

En O(N^2) où N est le nombre de serveurs.

Preuve :

On a trivialement (grâce au broadcast) que chaque serveur reçoit le message. Il renverra donc l'information qu'il porte à son parent, qui le transmettra alors à son parent, et ainsi de suite. Le noeud initiateur reçoit donc bien les données de tous les noeuds, ce qui est le but souhaité.

Changements de topologie

Nous cherchons à avoir trois propriétés vraies ci possible à tout instant :

1. Il y a exactement un serveur élu par réseau comportant plus d'un serveur
2. Chaque serveur a connaissance de l'ensemble des clients connectés sur son réseau
3. Chaque serveur a connaissance de l'ensemble des autres serveurs présent sur son réseau
4. Un seul jeton d'exclusion mutuelle est présent.

Ces trois propriétés sont mises à mal par les changements de topologie (ajout ou retrait d'un ou plusieurs serveurs).

Par exemple :

- Un ajout de serveur peut entrainer deux réseaux à être relié, et ainsi avoir deux serveurs élus.
- Un ajout de serveur peut entrainer l'ajout de nouveau client / serveurs sur notre réseau.
- Un retrait de serveur peut entrainer la coupure de notre réseau en 2 réseaux. Ainsi un de ces deux réseau se retrouvera sans serveur élu.
- Un retrait de serveur peut entrainer des clients serveurs à qui nous ne pouvons pas parler.
- Perte de messages de diffusion FIFO ou causale.

Notre but est de revenir au plus vite à la normale...

Approche

Pour solutionner ce problème, voici la solution que j'ai choisi de mettre en place :

- En cas de changement de topologie, on lance une élection.
- En fin d'élection, il appartient au serveur élu de resynchroniser les clients présent ainsi que la liste des serveurs. Pour se faire :
  - Il lance deux echos afn de collecter les données (liste des serveurs et liste des pseudos). Les détails de l'algorithme sont disponibles ici
  - Une fois qu'il a reçu une de ces listes, il lance ensuite un R broadcast pour la communiquer aux noeuds du réseau. Idem à la réception de la deuxième liste.
  - Enfin, chaque noeud qui reçoit cette liste sort du changement topologique.

Rappel : on suppose qu'il n'y a pas de messages C boadcasté ni de transfert de jeton d'exclusion mutuelle pendant notre changement de topologie.

Notes sur la sécurité dans un environemment multithreadé

Nous utilisons deux threads sur notre serveur. Un de ces threads consiste à attendre la saisie d'un utilisateur, l'autre consiste à écouter le réseau et réagir aux demandes qui sont faites.

Concernant le thread "clavier", il est à noté qu'il y a deux types de commandes :

- Celles qui sont thread safe. Il s'agit :
  - D'établir une connection avec un nouveau serveur
  - De consulter, réserver, ou relacher une ressource
  - Demander la terminaison de l'application distribuée en toute sécurité
- Les autres commandes ne sont pas thread safe, et ne sont présentes que pour des raisons de debug. Elles disparaitraient dans un environnement de production.

Afin de garantir l'aspect thread safe du premier type de méthodes, différents mécanismes ont été mis en place :

- Pour les méthodes qui ne demandent qu'un envoie de message, nous garantissons que l'usage des méthodes d'envoie de messages du NetManager permet l'envoi de plusieurs messages en simultané (un verrou gère les écritures sur chaque channel). La seule méthode faisant appel à ce mécanisme est la demande de connection à un autre serveur.
- Pour les méthodes utilisant des métodes de State, il a été préféré de mettre en place une politique de Scheduling. La raison de ce choix est la simplification de la gestion du multithreading en utilisant ce paradigme. Notre selecteur ne reste donc maintenant que temporairement en attente d'évenements, puis ensuite il consulte, et le cas échéant éxécute les action schedulées.

Coté client, nous avons juste à assurer le coté thread safe de l'envoie de messages.