

Programmation Concurrente, Réactive et Répartie
Jeu de Lettres
Jeudy Jordan - Ta Michael
2017-2018

Sommaire

Introduction.....	3
Serveur.....	4
Hiérarchie	4
Serveur de jeu.....	4
Serveur de mots.....	4
Serveur HTTP journalisation.....	4
Algorithmes	5
Structure de données	6
Concurrence	6
Bonus	8
Vérification immédiate.....	8
Chat	8
Dictionnaire	8
Journalisation	8
Client.....	9
Hiérarchie.....	9
Algorithmes.....	9
Manuel d'utilisateur :.....	13

Introduction

Le but du devoir était de réaliser une application clients-serveur permettant à des utilisateurs de jouer à un jeu de lettre multijoueurs nommé Boggle. Le principe du jeu Boggle est simple, chaque joueurs connectés au serveur reçoit une grille identique (une grille est un plateau de lettre de 4x4 lettres). Le but des joueurs est d'envoyer des mots construits à partir des lettres du plateau, à la fin d'un tour le serveur calcul les points obtenu par les joueurs en fonction de la taille du mot et de la difficulté à trouver le mot (par exemple un mot trouvé par plusieurs joueurs donnera moins de points qu'un autre trouvé par uniquement une personne). De plus, pour qu'un mot soit valide et donne donc des points, il faut qu'il soit former des mots à partir de lettres adjacents (horizontalement, verticalement ou en diagonale), les mots sont de tailles 3 minimum et les lettres ne peuvent être utilisé qu'une seule fois par mots, la taille maximum d'un mot et donc de 16, la grille étant composé de 16 lettres. Bien évidemment, les mots doivent exister, pour cela nous utiliserons un dictionnaire afin de vérifié leurs existence.

Dans la suite du projet, nous avons suivi le protocole proposé par le sujet du projet, nous avons toutefois ajouté quelques précisions :

- le format de bilan envoyé par le serveur sous la forme VAINQUEUR/bilan/ est identique à celui de scores. Il est donc sous la forme `n*usr1*scr1*usr2*sc2*...` où n peut correspondre au numéros de la session précédente.

- lors de l'extension du chat, lors d'un envoi d'un message publique, le serveur envoi lui aussi le message publique a l'envoyeur.

De plus, les langages choisis sont le C pour le serveur et le Java pour le client. Nous avons aussi réalisé les extensions du dictionnaire distant et du journal, ceux ci ont été fait en Java et en Python.

Ainsi, les extensions qui ont été réalisé pour ce projet sont le chat, la vérification immédiate, le dictionnaire distant, le journal, le client autonome et le client graphique.

Voici un lien vers le github de notre projet <https://github.com/jordanupmc/bogglePC2R>

Il contient un README.md qui fait office de manuel d'utilisation.

Serveur

Hierarchie

Le dossier serveur contient les fichiers sources des différents serveurs.

Src contient le serveur de jeu, words le serveur de mots et journal le serveur HTTP pour la journalisation des dernières parties. Le dossier test contient un fichier goodGrilles contenant des grilles dont il existe beaucoup de mots. Et un script startWithGrid qui permet de lancer facilement le serveur avec un fichier contenant des grilles. Le fichier doit contenir le nombre de tour, et les différentes grilles. Le fichier goodGrilles peut servir d'exemple.

Serveur de jeu

serveur.c est le main du serveur, il contient les différentes fonctions qui permettent de parser une requête, les tâches exécutées par les différents threads.

game.c contient la définition de la structure joueur, elle contient donc toutes les fonctions qui permettent de manipuler la liste des joueurs présents.

grille.c contient les fonctions liées à la grille de jeu, que ce soit la génération d'une grille aléatoire ou bien la vérification d'une trajectoire.

synchro.h contient les définitions des mutex et condition (ce sont des variables globales) utilisés par les threads.

threadSafeList.c contient les fonctions permettant de manipuler la structure nodePropose qui représente un nœud d'une liste chaînée (de proposition de mot). Ce sont des fonctions classiques le seul changement est la prise d'un mutex avant la modification/consultation de la liste.

Serveur de mots

DicoServer.java c'est une simple reprise du serveur multi-threadé du TME, il charge dans une HashMap tous les mots d'un dictionnaire qui se trouve dans le même dossier.

Serveur HTTP journalisation

journal.jou c'est le journal, le résultat est écrit dans un certain format décrit plus tard.

server.py c'est le main du serveur HTTP, il utilise une interface CGI et écoute sur le port 8888. Pour éviter d'avoir à réinventer la roue on utilise le module BaseHTTPServer de python.

cgi.py c'est le programme exécuté par server.py pour obtenir le HTML. Il prend journal.jou en entrée et génère une HTML.

Algorithmes

Vérification de trajectoire

Pour vérifier que la trajectoire proposée par un client est correcte on utilise une matrice d'adjacence. En effet la grille peut être vue comme un graphe non orienté, les dés sont les sommets. Les sommets adjacents sont reliés par des arêtes. Les dés sont donc représentables par des points, le dé 0 a la position (0, 0) le dé 15 a la position (3, 3)

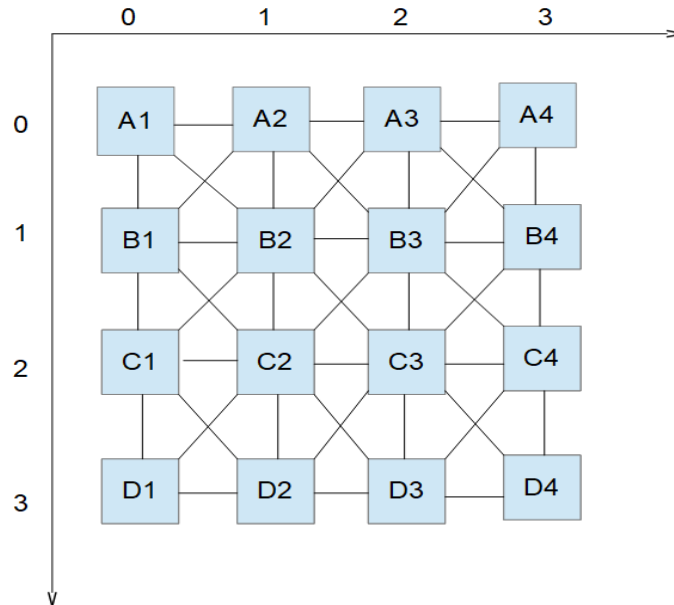


Fig. 1. graphe de la grille

On crée donc la matrice adjacence du graphe avec la fonction `createMatriceAdjacence`, elle crée une matrice 16×16 , $mAdjacence[i][j] = 1 \rightarrow$ dés i et dés j sont adjacents sinon $mAdjacence[i][j] = 0$. Il y a adjacence quand $distance(dés\ i, dés\ j) = 1$.

Enfin pour vérifier la trajectoire d'un joueur (fonction `checkTrajectoire`), il suffit de convertir les cases sélectionnées en position (A0 \rightarrow 0, D4 \rightarrow 15). Dans un premier temps on teste si la trajectoire ne contient pas de cases en doublon. Ensuite on parcourt la matrice adjacence pour savoir si la trajectoire est correcte.

Pseudo-code C d'une partie de la fonction `checkTrajectoire`.
traj contient les numéros de dés sélectionnés par le joueur,
nbCase le nombre de dés sélectionnés.

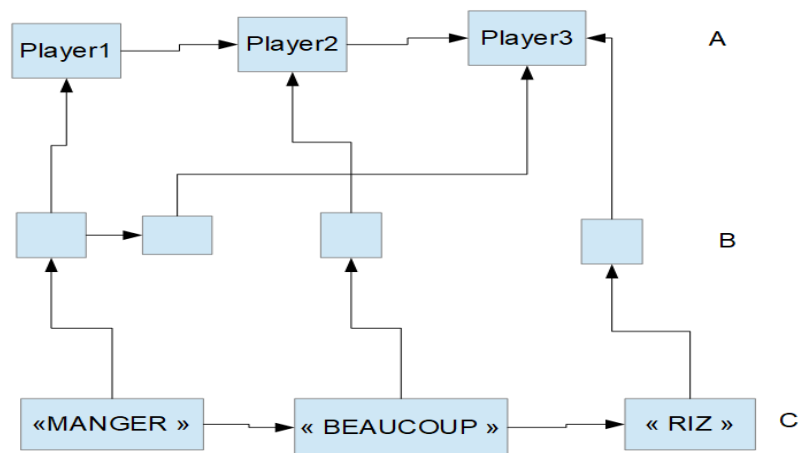
```
tmp = traj[0];
for( i = 1 ; i < nbCase ; i++){
    if( traj[i] < 0 || traj[i] >= 16 || mAdjacence[tmp][ traj[i] ] == 0 )
        return false ;
    tmp = traj[i] ;
}
return true ;
```

L'avantage de cette méthode c'est que le test de la trajectoire est simple, le point négatif c'est qu'au lancement du serveur il faut calculer la matrice adjacence. Même si on pourrait la calculer une fois pour toute, et initialisé mAdjacence de manière statique avec le résultat obtenus. Le calcul n'étant pas très demandant (matrice 16*16), on se permet de la calculer une fois au démarrage du serveur.

Structure de données

Le serveur repose sur une liste simplement chaînée de joueur, un joueur contenant un socket, son nom, un score, ses proposition de mots/trajectoires et un champs pour savoir si il a quitté le jeu sans ce déconnecter.

Il y a également une liste simplement chaînée de proposition, une proposition contenant le mot proposé, une liste de pointeurs de joueur qui ont proposé ce mot et le nombre de joueur de la liste de joueur.



A la liste des joueurs connecté

B les différentes listes de pointeurs de joueur

C la liste des proposition de mot

Fig 2. Structures de données

Concurrence

Le serveur de jeu est multi-threadé, il repose uniquement sur la librairie pthread pour tout ce qui concerne la concurrence, et suit le standard X/Open 7 pour les fonctions standard de C.

Le serveur est composé de quatre pool de thread :

- Connexion, gère les messages de type CONNEXION.
- Trouve, gère les messages de type TROUVE.
- Vérificateur, actif lorsque le temps de réflexion est écoulé.
- Chat, gère le chat

De plus il y a un thread appelé Timer qui est un « médiateur » qui va débloquent les vérificateurs, bloqué les threads de la pool Trouve. Dans la boucle infini du processus main il y a un select sur la socket de connexion et sur l'ensemble des sockets des joueurs connecté. Le main parse la requête et l'affecte au bon pool.

Un thread qui appartient à un pool prend le mutex de la pool auquel il appartient, il teste si il y a un travail à réaliser. S'il n'y a pas de travail il s'endort sur une condition, sinon il décrémente le nombre de travail et réalise un traitement spécifique. Même si les thread sont des processus léger, l'avantage d'avoir des pool c'est d'avoir un nombre constant de thread. En terme de problème de concurrence c'est un simple producteur consommateur. Le désavantage c'est que le main doit gérer lui même les déconnexion car il écoute simultanément sur un ensemble de socket avec la primitive select, il faut retirer le socket de l'ensemble dès réception d'une requête SORT. C'est un point que l'on a pas réussis à améliorer.

Connexion

Quand le main reçoit une connexion sur sa socket de connexion, il ajoute à un tableau la socket du client, Une thread du pool de connexion récupère la socket et attend le message de CONNEXION et ajoute le joueur à une liste. Enfin il envoie le message de BIENVENUE puis broadcaste à tous les joueurs le message de CONNECTE.

Trouve

Quand le main reçoit une requête TROUVE il ajoute dans un tableau de type data dont les champs utiles pour le pool Trouve sont la socket du client, la trajectoire et le mot. Une thread du pool récupère les informations puis teste si le temps de réflexion est écoulé (avec un flag géré par le Timer) pour ajouter ou non la proposition à une liste de proposition. Le défaut c'est que même si le temps est écoulé une thread se réveille juste pour « jeter » la proposition.

Vérificateur

Les thread de ce pool sont liés de près au Timer et la gestion de ce pool diffère du cas général. Ici, quand le Timer a broadcaste le RFIN, il débloquent les vérificateur avec un nombre de « job » a effectué égale au nombre de joueurs. Le principe c'est que les vérificateurs traite toutes les propositions d'un joueur. En vérifiant la trajectoire, le mot, et en envoyant un MVALIDE ou MINVALIDE en fonction. Le dernier thread vérificateur doit réveiller le Timer pour qu'il puisse broadcaste le bilan. Pour y parvenir on rajoute un compteur partagé qui est incrémenter après avoir finis de vérifier toute les proposition d'un joueur. Si le compteur est égale au nombre de joueur c'est qu'on est le dernier vérificateur (les autres thread sont bloqué sur la condition de tableau vide du problème producteur consommateur).

Timer

Il porte ce nom car ce thread gère le temps de réflexion, mais il fait bien plus qu'un simple sleep. Au lancement du serveur il est bloqué sur une condition d'initialisation, c'est à dire tant qu'un joueur ne s'est pas connecté le temps de réflexion ne démarre pas. Ensuite dès qu'un joueur c'est connecté le comportement de ce thread est toujours le même. Il dort pendant un certains temps, à la fin de ce temps il change la valeur d'un flag qui signifie que le temps de réflexion est terminé. Il broadcaste le message RFIN. Les threads du pool Trouve arrête d'ajouter des proposition de mots. Il réveille les vérificateurs pour qu'ils traitent tous les clients. Il s'endort sur une condition, pour que les vérificateurs terminent de travailler. Il calcule les scores de tous les joueurs, broadcaste la requête de BILANMOTS, il incrémente le numéros du tour en fonction de sa valeur il broadcaste les requêtes VAINQUEUR ou SESSION.

Bonus

Vérification immédiate

On « fusionne » le pool Trouve et celui des Vérificateurs, on a donc un pool TrouveImmédiat. Un thread de ce pool va directement vérifier si les arguments sont correcte. Si c'est le cas le thread appelle la fonction containsMotThenAdd qui est threadSafe. Cette fonction, dans une section protégée par un mutex teste si le mot a déjà été proposé si ce n'est pas le cas on ajoute le mot à la liste. Donc on a la garantie de ne pas avoir d'entrelacement possible lors du teste « containsMot » et de l'ajout du mot à la liste. Selon la valeur retournée on envoie la requête MINVALIDE/PRI. Le timer n'a pas besoin d'attendre la fin du traitement des threads du pool TrouveImmédiat, il change la valeur de son flag et liste tous les mots et récupère les scores directement. Le défaut c'est que si un thread était en train de vérifier une proposition et qu'entre temps le temps de réflexion est terminé, la proposition ne sera pas comptabilisé.

Chat

C'est un pool de thread classique, il n'y a pas de spécificité particulière. A part que le pool traite les requêtes de chat publique et privé. Pour le chat publique on fait un simple broadcast, pour le privé on retrouve la socket du joueur à partir du nom précisé dans la requête et on lui envoie le message directement.

Dictionnaire

Le serveur de mots est codé en Java, pour éviter d'avoir à coder une structure de données assez intéressante en C pour permettre une recherche rapide d'un mot. Java offre déjà des implémentations de structure très efficace. On utilise une HashMap<String> pour représenter le dictionnaire qui est chargé à partir d'un fichier à la création du serveur. Les vérificateurs communiquent avec ce serveur pour tester l'existence d'un mot avec le protocole suivant :

C → S CHECK MOT\n est-ce que MOT existe ?

S → C OK\n le mot existe

S → C KO\n le mot n'existe pas

Journalisation

À la fin d'une session le Timer écrit dans le fichier journal.jou les informations de la session dans ce format.

S numerosTour date-heure

U scores

F

F correspond à Fin, S Session, U Users

Le serveur HTTP de journalisation, parse ce fichier et génère le contenu de tableau HTML. On peut y accéder depuis un navigateur sur le port 8888 de la machine qui a lancé le server.py,

Client

Hierarchie

Le package `boggle.client` contient les fichiers sources des clients :

- `BoggleClient.java` correspond aux informations du client connectés, son nom, s'il est en jeux, s'il est connecté, la grille sur laquelle il joue.

- Nous avons divisé les client en deux threads, une threads sera dédié à la réception des messages venant du serveur (`ClientLecteur.java`) il est celui du modifie l'interface graphique, l'autre sera utilisé afin d'envoyer des requêtes aux serveur (`ClientEcrivain.java` est l'interface de celui ci, `ClientJoueur.java` et `ClientTricheur.java` l'implémente) il est celui qui récupère les informations de l'interface graphique. Ainsi, il le client pourra recevoir des requêtes du serveur a tout moment et pourra en envoyer a aussi à n'importe quel moment.

- `UserScore.java` est utilisé pour construire la table des scores dans l'interface graphique.

Le package `boggle.ui` contient les fichiers sources en rapport à l'interface graphique ainsi que le main :

- `GameWindow.java` est l'interface graphique du jeu, elle a été faite avec `javafx`.

- `UIMain.java` est le main du client, il lance les deux threads clients (Lecteur et Ecrivain) et lance aussi l'interface graphique.

Le package `boggle.dico` contient les fichiers sources afin de construire un dictionnaire pour l'extension du client autonome.

Algorithmes

Le Client Lecteur contient une boucle infini, qui attend les requêtes du serveur, lors de la réception d'un message de type :

- BIENVENUE, il vérifie que le tirage reçu contient 16 lettres, met à jour la table des scores et met aussi à jour la grille permettant ainsi le joueur d'envoyer des mots.

- CONNECTE (ou DECONNEXION, SESSION), affiche un message informant le client qu'un nouveau joueur s'est connecté (ou déconnecté, ou qu'une nouvelle session commence).

- VAINQUEUR, met à jour la table des scores.

- TOUR, vérifie que le tirage est une chaîne de caractère de taille 16 et met à jour la grille avec ce nouveau tirage et autorise le joueur d'envoyer de nouveau mots.

- MVALIDE (et MINVALIDE), affiche un message informant que le mot a été validé (ou invalidé).

- RFIN, informe le client que le tour est fini et empêche le client d'envoyer de nouveau mots au serveur.

-BILANMOTS, met à jour la table des scores et informe le client des mots qui ont été accepté par le serveur.

-RECEPTION (et PRECEPTION) affiche dans le chat le message (privé) reçu.

Dans le cas où la commande n'est pas reconnu ou qu'elle n'est pas correctement formée, on affiche un message dans la console et on ignore la commande.

Le client Joueur contient une liste de commandes qu'il doit envoyé au serveur, au lancement il attend que le joueur entre son nom, lorsque c'est fait il envoi la requête CONNEXION au serveur suivi de son nom. Puis il boucle indéfiniment, il attend que le client compose des mots ou des messages pour les envoyer au serveur, si le client décide d'envoyer une requête SORT, il sort de sa boucle infini et se termine. La concurrence a été géré à l'aide de wait() en java, le client attend qu'on lui notify() qu'une commande doit être envoyé, cela est fait dans l'interface graphique.

De plus, lorsque le client Joueur se termine signe que le client arrête de jouer, il ferme la socket, le client lecteur attrape donc l'exception provoqué par le readln sur une socket fermée et se termine, la terminaison de la thread client lecteur est ainsi géré.

Le client tricheur construit dans un premier temps un dictionnaire à partir d'un fichier, ce fichier a été construit à partir du GLAFF tel qu'il a été proposé dans le sujet. Puis il parcourt la grille et forme des mots, qu'il ajoute dans la liste des commandes a envoyé le mot trouvé s'il est de taille 3 ou plus, s'il existe dans le dictionnaire et si le mot n'a pas déjà été envoyé pour cette grille.

La construction des mots se déroule de la façon suivante, on initialise un tableau a deux dimension initialisé a false a toute les cases, il correspondra aux cases qu'on a déjà visité. Puis, on construit une chaîne de caractère correspondant a un mot de façon récursive en visitant les cases adjacents a la notre (horizontalement, verticalement ou en diagonal) en vérifiant que la case n'a pas été déjà visitée.

Si le chat a été activé, le tricheur a 40 % de chance d'envoyé un message publique pré-définie lors du début du tour, et a encore 40 % de chance d'en envoyé un second à la fin du tour. Après cela, il parcourt sa liste des commandes a envoyé, il en envoi une toutes les 0,5s à 2s pour le rendre plus « humain », de plus, si le joueur décide d'envoyer un message ou de trouver lui même un mot ou de se déconnecter, la requête sera la prochaine a être envoyé au serveur. De même que le client joueur, s'il n'a pas de requêtes a envoyé, il attendra avec un wait le notify de l'interface graphique.

BuildDico est une classe qui construit le fichier « Dictionnaire.txt », en supprimant les accents et en mettant en majuscule les mots du GLAFF, il s'assure aussi de l'unicité des mots.

Dictionnaire est une classe qui parcourt un fichier pour construire une hashmap représentant un dictionnaire. Nous avons choisi une hashmap car il est plus rapide de construire cette structure et il est aussi plus rapide de chercher l'existence d'un mot dans la hashmap. Toutefois, nous perdons en complexité mémoire mais cela peut être négligeable face au temps gagné à la construction et à la recherche.

L'interface graphique (GameWindow) est au départ composé d'un champ ou le joueur peut décider de son nom (Fig 3.), d'ailleurs nous vérifions que le nom ne contient pas de caractères spéciaux pouvant nuire à notre protocole notamment des '/', '*' ou des '\n' pour ce faire nous limitons les noms aux lettres majuscules et minuscules et aux chiffres, de plus un nom est de taille 1 minimum et 16 au maximum.

Après cela, il se trouve sur une page d'attente en attendant le message de BIENVENUE du serveur (Fig. 4). Une fois reçu, l'interface sera composée d'un champ défilant correspondant à des information non pertinente pour le déroulement d'une partie (joueur qui se connecte/déconnecte, etc), d'un champ affichant le mot qui est en train d'être composé, la grille de jeu, un tableau des scores, un bouton ouvrant le navigateur pour afficher le journal et le chat (voir Fig 5), il s'agit de la description de l'interface où toutes les options sont activées, par exemple si l'option journal n'est pas activée, le bouton ne sera pas présent.

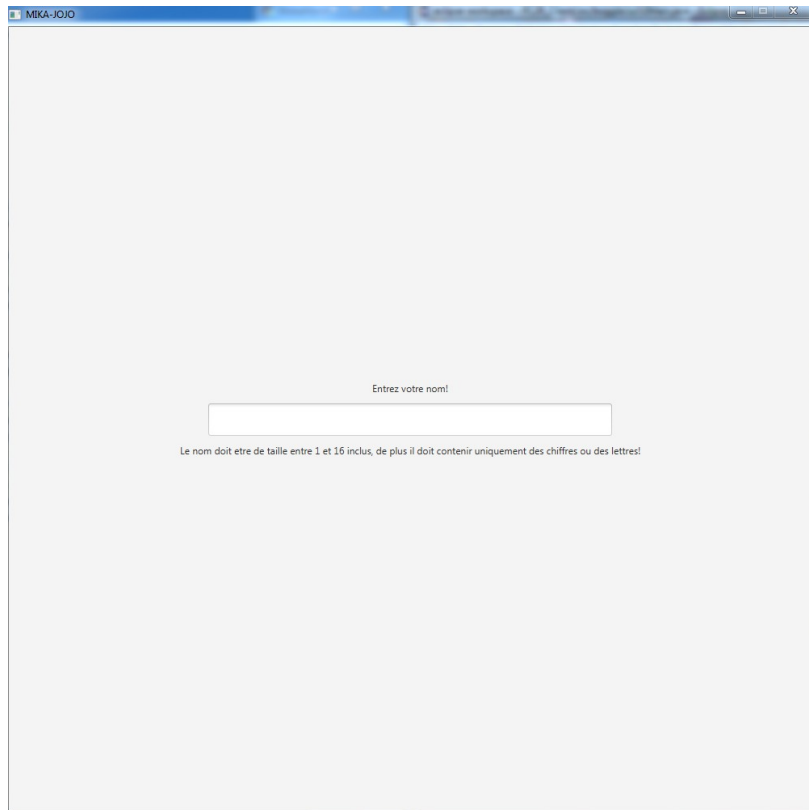
The image shows a web browser window with the title bar 'MIKA-JOJO'. The main content area is light gray. In the center, there is a text input field with the placeholder text 'Entrez votre nom!'. Below the input field, there is a small line of text in red: 'Le nom doit etre de taille entre 1 et 16 inclus, de plus il doit contenir uniquement des chiffres ou des lettres!'. The browser window has standard Windows-style window controls (minimize, maximize, close) on the right side of the title bar.

Fig 3 : Interface graphique pour entrer le nom du joueur

La grille de jeu est jouable, c'est à dire que lorsque l'on souhaite envoyer un mot, nous commençons par cliqué sur la première lettre du mot puis en maintenant enfoncé le clique, nous déplaçons la souris sur le seconde lettre du mot et ainsi de suite. Lors du relâchement de la souris, le mot est envoyé au client joueur (ou client tricheur) et on le notify qu'une nouvelle commande est prête à être envoyé.

Pour envoyer un message publique, il suffit de l'écrire dans le champ et d'appuyer sur entrer, l'interface graphique s'occupera de créer la commande ENVOI et de signaler au client joueur qu'une commande est prête. De même, lors d'un envoi d'un message privé, il suffit d'écrire dans le champ le nom de la personne, si le champ est vide, c'est qu'il s'agit d'un message publique.

Manuel d'utilisateur :

Pour lancer compiler le client, placez vous dans le répertoire du projet et lancez la commande :
`ant compile`

Pour lancer le client, lancez la commande :

`ant run -Dargs='-serveur|-port|-chat|-cheat|-journal''`

- -serveur hostname : indiquant le nom du serveur ou son adresse IP
- -port numport : précisant le numéro de port du serveur
- - chat : pour activer le chat
- -cheat : pour activer le client autonome
- -journal website : indiquant l'adresse web du journal