CHATHACK

Manuel Développeur

MARTI Emilie & MECHOUK Lisa – ESIPE INFO2

SOMMAIRE

Présentation du protocole

Le protocole Chathack est un projet de Programmation Réseau concernant un Serveur et Client de messagerie instantanée suivant un protocole décris dans la RFC MARTIMECHOUK. Il permet à des utilisateurs d'échanger des messages entre eux en privé ou en public.

Architecture Générale

L'architecture globale du projet est séparée en **ChatHackClient**, **ChatHackServer** et classes utilitaires, chacun dans son package.

Les classes considérées utilitaires sont principalement celles dont les interfaces implémentées sont communes pour le serveur et le client, c'est à dire :

- **Context** : les contextes du serveur et du client
- **Frame** : les classes des trames
- **Reader** : les classes qui lisent le contenu d'un ByteBuffer reçu
- **FrameVisitor** : Les classes du visiteur

Diagramme des classes UML : cf. Annexes

Client ChatHack

Le client du protocole ChatHack est implémenté en **TCP non-bloquant**.

Pour gérer les deux types de connexion du client ChatHack, qui se caractérise par un alias avec ou sans mot de passe, on utilise deux constructeurs. Le nombre d'arguments passés lors de l'exécution du client déterminera de quel type de client il s'agit.

Pour les échanges, nous avons découpé ces parties en deux. Nous avons le traitement des échanges clients serveurs grâce à une socket channel qui permet d'envoyer nos messages au serveur pour qu'il l'envoie à tous les clients connectés. Il possède également un context : **ClientToServerContext** qui prend une key et un client afin de réaliser les traitements dessus.

La partie des échanges privé client/client possède également son propre contexte : **ClientToClientContext** qui prend en argument une key et un client. Dans ce cas le client fait une demande de connexion privée et fait office de serveur pour accueillir la connexion client-client, via une ServerSocketChannel.

Chaque contexte **ClientToServerContext** et **ClientToClientContext** possèdent leurs propres visiteurs : **ClientToServerFrameVisitor** et **ClientToClientFrameVisitor**.

Le client est concerné par les trames suivantes :

- ConnectionFrame
- GlobalMessageFrame
- PrivateConnectionFrame
- SimpleFrame

Serveur ChatHack

Le serveur ChatHack est un serveur de chat en TCP non-bloquant.

A l'exécution, on lui renseigne une adresse et deux ports libres de la machine : Le premier, qui sera le port d'écoute pour accueillir les connexions des clients puis le deuxième, le port d'écoute du serveur MDP fourni.

Le serveur ChatHack agit comme un serveur vis-à-vis des clients qui vont vouloir utiliser le service de chat et va également agir comme un client vis à vis du serveur MDP.

Un seul selector est utilisé pour les deux types de connexion, avec une classe contexte chacun. Toutes les classes des contextes implémentent l'interface Context.

Dans ce projet, on utilise le Patron de Conception du Visiteur pour gérer les différentes trames qui circuleront à travers l'environnement de ChatHack. Tous les classes des visiteurs implémentent l'interface FrameVisitor. Il y a une classe visiteur pour chaque contexte.

Pour chaque type de trame on utilise une classe. Les champs des classes des trames correspondent aux champs de la trame et à leur équivalent en ByteBuffer. Toutes les classes des trames implémentent l'interface ChatHackFrame.

En tant que serveur :

La classe contexte concernée est ServerToClientContext et son visiteur est la classe ServerToClientFrameVisitor. Les trames concernées sont les suivantes :

- ConnectionFrame
- GlobalMessageFrame
- LoginPasswordFrame
- PrivateConnectionFrame
- SimpleFrame

Les clients du serveur sont identifiés par un ID et cette association se fait via une HashMap dans les champs (HashMap<Long, ServerToClientContext>). Pour chaque client, il y a un contexte.

En tant que client :

La classe contexte concernée est ServerToBDDContext et son visiteur est la classe ServerToBDDFrameVisitor. Les trames concernées sont les suivantes :

- BDDServerFrame
- BDDServerFrameWithPassword

Ces trames ci-dessus vont être créées à partir de la trame de requête de connexion pour demander la validité du login demandé (avec ou sans mot de passe).

- BDDServerResponseFrame

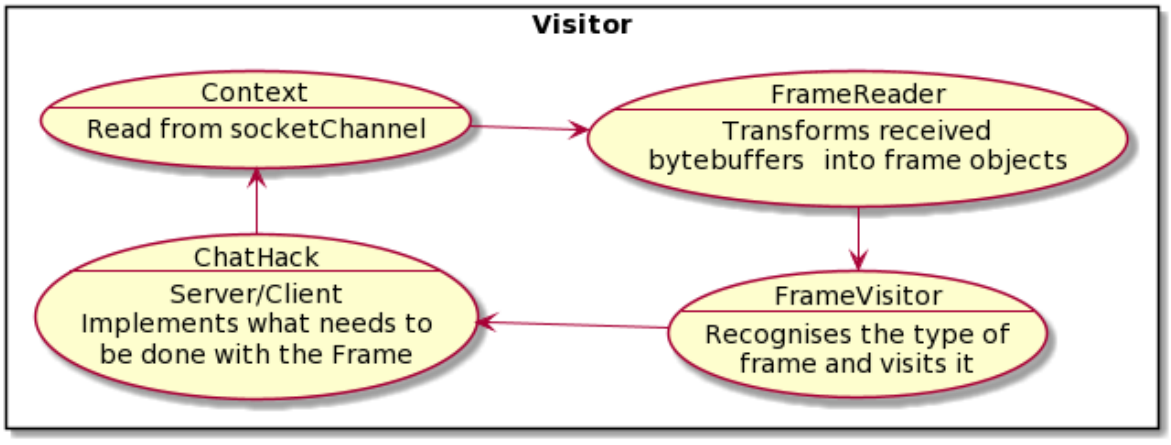
Cette trame va être reçue par le serveur ChatHack lorsque le serveur va lui faire une requête de validation d'un login avec/sans mot de passe (voir ci-dessus).

Le Visiteur

Ce patron de conception permet de gérer chaque frame séparément lorsqu'elles sont lues par un serveur ou un client. Lors de la lecture, nous ne pouvons pas savoir de quelle trame il s'agit tant qu'on ne les aura pas décodées via nos classes qui implémentent Reader.

Pour ce faire, nous utilisons le visiteur qui va envelopper les opérations de traitement et vont être appelées selon le type exact de la classe de la trame sur laquelle on veut réaliser le traitement. Chaque visiteur va implémenter les méthodes de traitement pour toute les trames qui existent dans l'environnement de ChatHack.

Comme il y a un visiteur par contexte, certaines trames ne sont pas concernées, et dans ce cas on envoie une exception de type UnsupportedOperationException. C'est un contrat donc on doit, en tant que développeur, ne pas appeler ces méthodes. Une représentation visuelle de comment fonctionne le visiteur dans ce projet est proposée ci-dessous



Soutenance Bêta

Lors de la soutenance, environ 80% du développement était fait, ce pourquoi cette documentation est semblable à celle donnée le jour de la démonstration le 20/05/2020. Cependant un bug bloquant empêchait toute connexion au serveur ChatHack, donc la démonstration n'a pu se réaliser. Ce qui nous a fait défaut est que nous avions beaucoup développé, mais très peu débugué. Le gros point que nous devions traiter était le débogage de l'application afin de faire fonctionner ce qui était déjà implémenté mais non fonctionnel avant de nous lancer dans le développement des autres fonctionnalités

Ce bug a été corrigé dans la journée avec une séance d'intégration du travail des deux binômes.

Deux HashMap ont été rajoutées dans le serveur pour lier contexte, id et login. À la suite d'une remarque de l'enseignant, le parcours des clés à réaliser pour broadcaster un message à destination du chat global se fait désormais sur ces hashmaps, et non sur le sélecteur.

Pour la suite du développement, nous avons fait attention à bien déboguer régulièrement afin d'avoir des fonctionnalités qui fonctionnent correctement.

Les fonctionnalités du ChatHack

Fonctionnalités	Etat
Message en Broadcast	
Message privé	
Envoie de fichier privé	
Déconnexion	
Connexion au serverMDP	

Fonctionnel

Non Fonctionnel

Difficultés que vous avez rencontrées

Les plus grosses difficultés que nous avons rencontrées concernaient les problèmes de taille des buffers, des oublis ou des flips en trop.

Le fait de réaliser ce projet en distanciel a compliqué la chose lors des périodes de débogage, c'est d'ailleurs ce qui nous a fait défaut lors de la première soutenance, nous avions beaucoup développé de notre côté sans avoir mis en commun ou tester nos fonctionnalités. Pour arranger cette situation nous avons organiser plusieurs points débogage par téléphone et partage d'écran afin de déboguer ensemble le projet et cela a très bien fonctionné et nous a permis de réaliser l'intégralité des fonctionnalités attendues.

Nous avons aussi beaucoup échangé afin de se tenir un maximum au courant sur nos avancées concernant le développement ou la mise à jour de la RFC.

En difficulté, nous avons été obligés à plusieurs de modifier notre RFC afin de l'améliorer car en développant nous nous rendions compte que certaines Frames devaient être changées ou encore ajouter des nouvelles Frames. Nous avons donc souvent modifié notre RFC avant de trouver les bonnes Frames permettant d'implémenter proprement notre protocole ChatHack.

Encore une fois sur ce point la communication a été un point primordial, nous avons beaucoup échangé entre nous afin de toujours se tenir à jour sur l'avancé du projet et de chacune des parties qui le compose.

Annexes

Diagramme des classes UML :