UQAC

DirectChat

Systèmes Répartis TP2

Cours : [8INF843]
Professeur : Hamid Mcheick

Fabien OGLI
Mathieu REGNARD

Fabien OGLI
Mathieu REGNARD

Installation

Les éléments suivants sont requis pour lancer l'application:

- Docker
- Docker-Compose
- Node
- Npm,
- Yarn (pour le développement)

Les instructions sont écrites pour un environnement Linux/UNIX. Sur un environnement Windows, il faudra adapter les commandes mais elles devraient rester très similaires.

Pour lancer le serveur ouvrez un terminal et tapez la commande suivante (Attention, sur macOS et Windows, il faut s'assurer que docker soit lancé au préalable)

```
$ docker-compose up -d
```

Pour l'arrêter, entrez cette commande:

```
$ docker-compose down
```

Une fois le serveur lancé (sur le port 8000), on peut démarrer l'application

Pour lancer l'application en mode développement il faut d'abord récupérer les dépendances du *core* peer2peer et du code d'affichage. Depuis la racine du projects, exécutez les commandes suivantes.
Dans le cas du mode développement, il est possible de changer l'URL du serveur dans le fichier "views/vue-frontend/src/renderer/env.js".

```
$ cd app/ && npm i && cd ../
```

```
$ cd views/vue-frontend && yarn install
```

```
$ yarn dev
```

Ou exécutez l'application compilé, en mode production directement depuis le dossier 'build/'. L'application utilisera localhost:8000 comme URL de base pour le serveur !

Ce dossier contient 3 dossiers :

- un dossier macOS
- un dossier linux
- un dossier windows

Déplacez vous dans le dossier correspondant à votre plateforme, et exécutez le binaire :
p2p-chat.app pour macOS
p2p-chat pour linux (Attention, il faut surement changer les permissions avec chmod +x …)
p2p-chat.exe pour Windows

Un double clique sur l'exécutable devrait lancer l'application.

Architecture

P2P-Chat est une application de messagerie instantanée en Pair-à-Pair. Cela signifie que chaque utilisateur de l'application est directement connecté à ses amis en ligne. L'architecture est un mélange à la fois d'une architecture Client-Serveur et d'une architecture Décentralisée. Notre souhait initial était de faire une messagerie instantanée complètement décentralisée, c'est à dire sans serveur central.
Cependant l'authentification et la gestion d'amis se révèlèrent complexe :

- Comment faire l'authentification ?
- Comment identifier un utilisateur sur le réseau ?
- Comment avoir l'adresse IP d'un ami que l'on veut ajouter ?

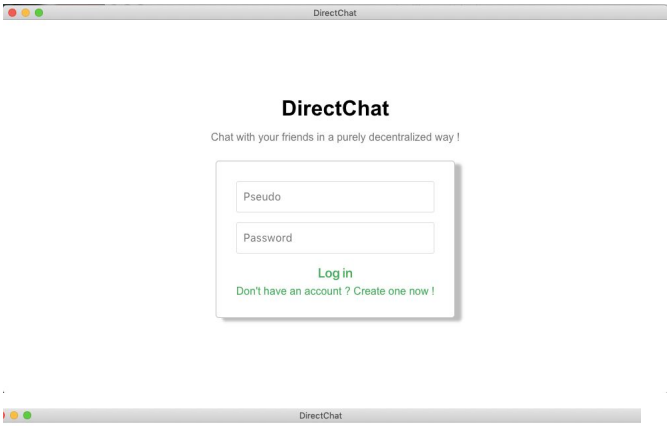
Plusieurs méthodes existent pour répondre à ces problèmes, mais la difficulté et le temps d'implémentation sont augmentés !
Nous avons étudié ces méthodes, comme par exemple les *Distributed Hash Table*, qui vont permettre de stocker et d'accéder à des éléments dans le réseau sans avoir à parcourir chaque noeud. On peut par exemple connaître l'IP d'une personne en O(log(n)), avec n le nombre de noeuds du réseau.
Cependant, d'autres problèmes peuvent surgir : que se passe-t-il par exemple lorsque l'information n'est présente dans aucun des noeuds ?
Afin de faciliter le développement, nous avons souhaité nous orienter vers une approche hybride. Un serveur central architecturé en api REST permet de gérer l'authentification et l'identification des utilisateurs. Ce serveur sert également de routeur et stocke les adresses IPs des clients connectés. Lorsqu'un noeud se connecte, il pourra connaître l'adresse IP d'un client particulier et se connecter en communication directe.

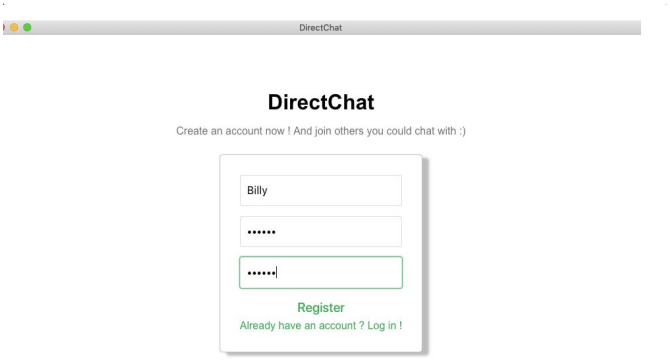
L'architecture du serveur à été mise en place grâce à *Docker* et *Docker Compose*. Ces deux outils permettent de créer un environnement de travail commun à tous, et donc permet d'éviter les problèmes de systèmes d'exploitations différents.

Le serveur à été écrit avec le langage *Golang*.
L'application *peer-to-peer* à été entièrement réalisée en *JavaScript* grâce à *NodeJS*.
La couche de présentation est faite grâce aux frameworks *Electron* (qui permet de créer des applications *cross-platform* **natives** en *JavaScript*) et *VueJS*.

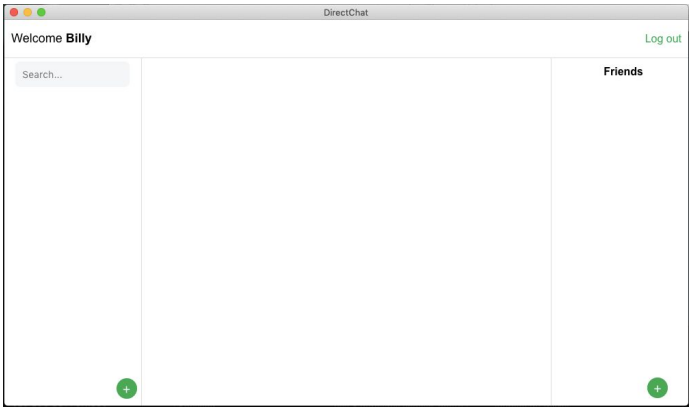
Trace d'exécution

Au démarrage de l'application, il est demandé de s'authentifier sur le réseau. Comme c'est la première exécution, il faut créer un compte !



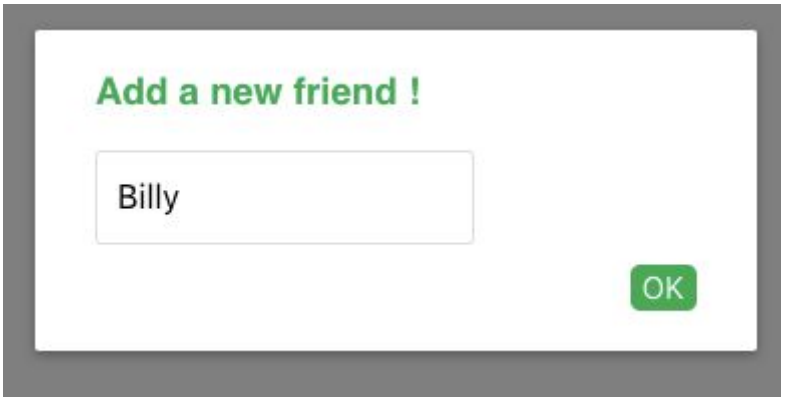


Ensuite, nous arrivons sur la page d'accueil. Celle ci est composée de 3 parties. La partie à gauche, qui va permettre de gérer la liste des conversations, la partie centrale, qui va servir à afficher une conversation en particulier et la partie de droite, qui permet de gérer la liste d'amis.

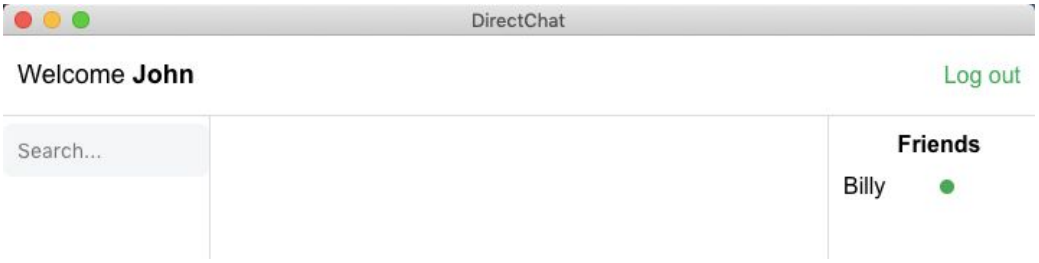


Pour la suite de cette démonstration, nous allons créer un deuxième compte (en suivant les mêmes étapes) : le compte de John.

Bien sûr, l'application de Billy n'est pas fermée, et il est toujours connecté !
Une fois celui-ci créé, nous allons ajouter Billy dans la liste d'amis de John. John est alors automatiquement ajouté dans la liste des amis de Billy.

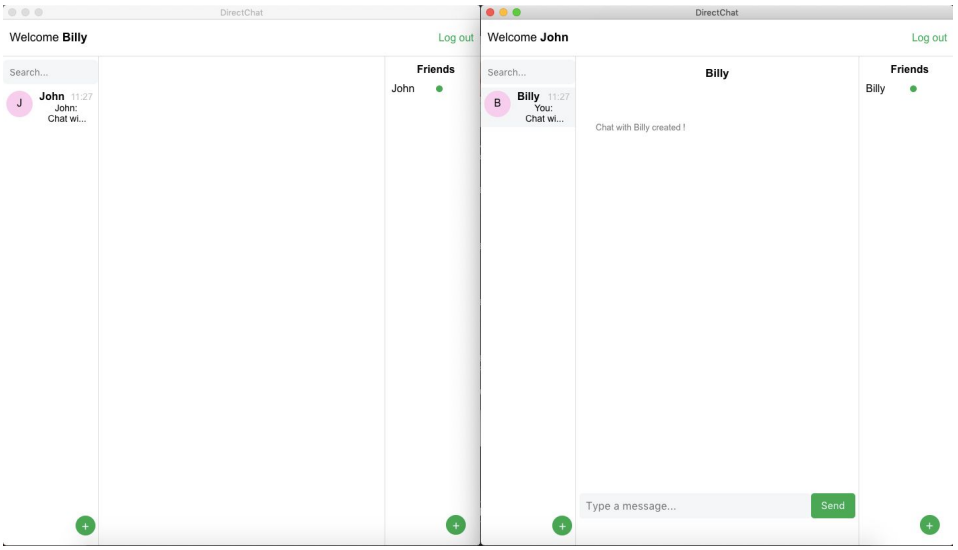


On peut remarquer ici que Billy est bien connecté (petit cercle vert à côté de son pseudo)

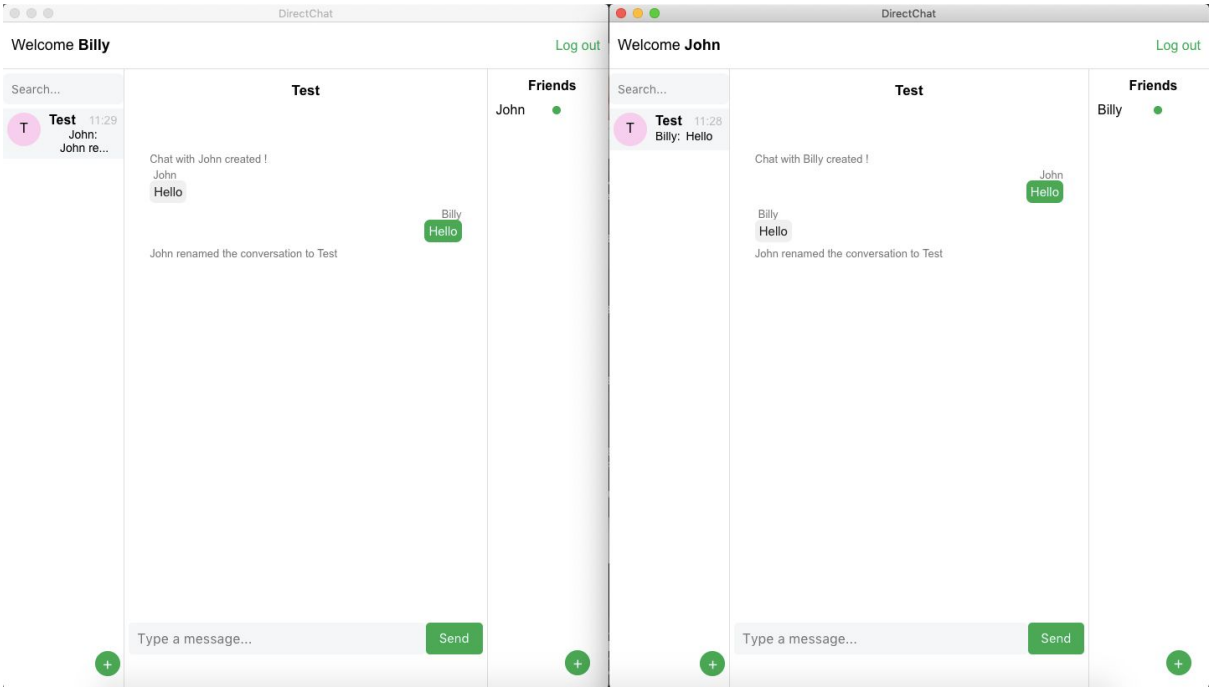


On peut ensuite créer une conversation avec lui, en cliquant sur le bouton + en bas à gauche de l'écran.

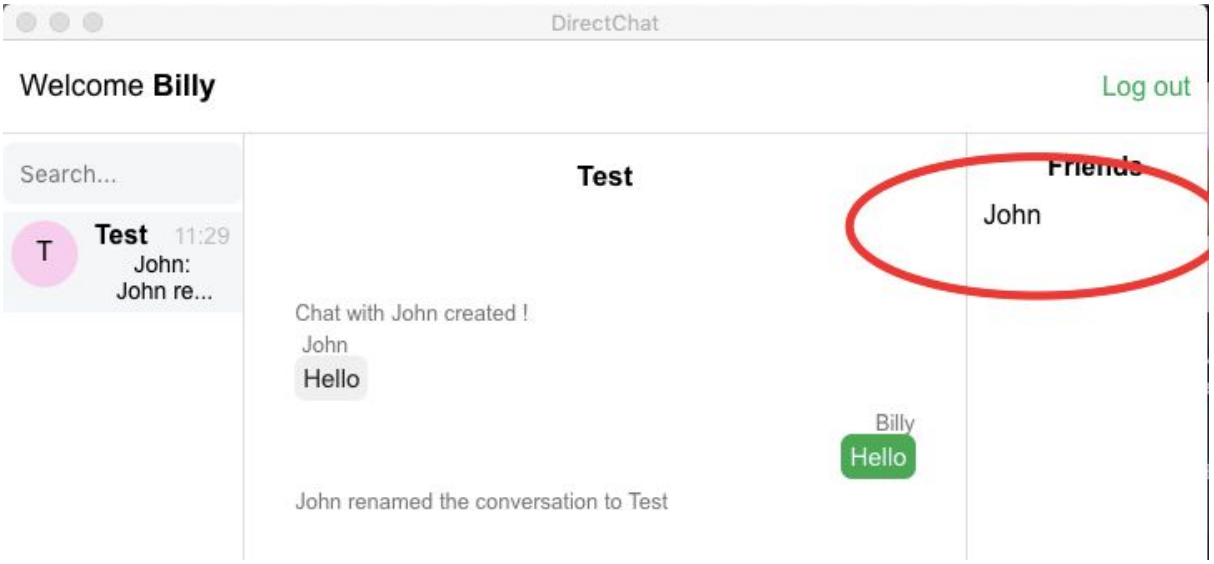
La conversation est alors créée instantanément chez les deux clients;



Enfin, on peut discuter ! Il est également possible de changer le nom de la conversation (pseudo de la ou les personnes avec qui on discute par défaut). Ce changement est propagée à tous les participants de la conversation comme on peut le voir sur la capture d'écran ci dessous.



Pour finir, lorsqu'un utilisateur est déconnecté (dans notre cas John), la petite icône verte disparaît ce qui nous permet de savoir en temps réel qui est connecté ou non.



Fabien OGLI
Mathieu REGNARD

Détails de l'application

P2P

Que se cache-t-il derrière toute cette belle présentation ?
La logique de l'application bien sûr !

Lorsque l'utilisateur est connecté, il ouvre un port sur la machine et attends des connexions.
En parallèle, il demande au serveur la liste de ses amis, avec leurs adresses IPs et port.
Le client se connecte alors à tous ses amis dont l'adresse IP est disponible (ce qui signifie qu'ils sont connectés).

Une fois la connexion TCP établie entre les deux noeuds, un moyen d'authentification à été mis en place. Nous référencerons à ce mécanisme par '*handshake*'.
Le noeud qui fait office de serveur attends un *handshake* avant d'accepter la connection.
Le client envoie alors le message suivant : *'handshake* clientId Pseudo'.
Le noeud serveur peut alors connaître le client qui communique avec lui et il lui répond un message similaire afin que le client s'assure de communiquer avec la bonne personne.

Par la suite, nous souhaitions encrypter la communication, et donc cet *handshake* permettrait de se mettre d'accord entre le serveur et le client sur les algorithmes de cryptage.
De plus, il serait également possible d'assurer l'intégrité de la communication : *le noeud serveur correspond bien à cette personne et j'ai la certitude que ce n'est pas quelqu'un qui se fait passer pour lui*.

Une fois la partie *handshake* terminée, la communication peut commencer ! Chaque noeud va pouvoir envoyer et recevoir des messages.
La gestion de la réception et de l'envoie de message est fait grâce à une architecture *Publisher/Subscriber* implémentée dans le module *NodeJS 'net'*. On enregistre des événements, et des actions correspondant à ces événements : *Lorsque tel événement occurre, exécute telle méthode*.
Le message est traitée par une liste chainée d'opérations, ce qui permet d'ajouter plusieurs types d'opérations de manière modulaire !
Idée d'opérations : gestion des unicodes, ajout des '*emojis*' en transformant les caractères, cryptage etc…
Les opérations inverse sont exécutées à la réception du message.
Pour l'instant, la seule opération implémentée est l'ajout d'un retour chariot '\r\n' à la fin du message.

Afin de faciliter la gestion des utilisateurs, chaque communication est associé à un socket, qui correspond à un client. Ces associations sockets/clients sont stockés afin d'y accéder facilement.

Fabien OGLI
Mathieu REGNARD

Serveur REST

L'utilité du serveur dans notre application est la gestion de client dans la base de données. Il permet donc d'enregister son adresse IP, son pseudo et son mot de passe. De plus, le serveur gère également les liens d'amités de client à client. C'est donc lui qui permet au client de connaître ses amis et leurs adresses IPs.
Par soucis de sécurité, à chaque fois que le client se déconnecte, son adresse IP est supprimée de la base de donnée.

Route	Méthode	Requis	Détails
"api/clients/new"	POST	Pseudo, mot de passe et adresse	Créer un nouveau client
"api/clients/login"	POST	Pseudo, mot de passe et adresse	Log le client
"api/clients/{id}"	GET	JWT Token	Retourn le client
"api/clients/{id}"	PUT	JWT Token, Id, champs modifiés	Met à jour le client
"api/clients/{id}"	DELETE	JWT Token, Id du client	Supprime le client
"api/clients/{id}/friends"	POST	JWT Token, pseudo de l'ami	Ajoute un ami au client
"api/clients/{id}/logout"	PUT	JWT Token	Déconnecte le client

Fabien OGLI
Mathieu REGNARD

Liste des fonctionnalités

- Protocole de communication directe en P2P créé.
- Gestion des utilisateurs
- Stockage local des conversations par sécurité
- Chat implémenté sur le protocole
- Ajout d'amis
- Discussion 1-to-1 : on discute avec une seule personne
- Discussion de groupe
- Message de type informationnels (Changement de nom de la conversation etc)
- Recherche de conversation
- Chaîne de traitement (opérations) sur le message

Améliorations possibles

- Cryptage de la communication
- Cryptage des conversations (le contenu du message est crypté, avec un cryptage asymétrique de type RSA)
- Cryptage N-to-N : On crypte la conversation de groupe avec des algorithmes différents
- Ajout des différentes opérations sur la liste chaînée
- Appels audio (communication directe)
- Appels vidéos (communication directe)
- Envoyer un message lorsqu'une personne est déconnectée
- Gérer la demande d'amis (pas d'acceptation automatique)
- Gestion de plusieurs interface pour un même utilisateur (recevoir sur le cellulaire ainsi que sur l'ordi)