Td 1 - Node.js v0.10.33 - Irc like App

10 novembre 2017

1

Stream et Socket

L'objectif de ce td est de réaliser une version avancée du client/serveur de tchat.

1. Le client doit pouvoir *uploader* et *downloader* les fichiers sur le serveur. Ces opérations s'effectueront par l'usage d'un deuxième canal de communication à la `ftp`. En d'autres termes, un premier canal permettra de gérer le contrôle du deuxième canal dédié au transfert de fichier.

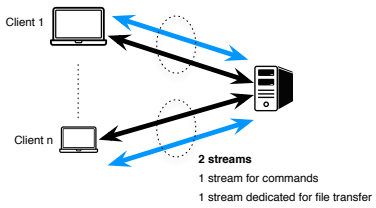

FIGURE 1 – Transfer de fichiers

En particulier,

- Définir deux nouvelles commandes `upload` et `download` ainsi que leurs syntaxes pour initier le transfert de fichier.

Base revision d87719f, Wed Oct 11 11:55:02 2017 +0200, David Bromberg.

- Mettre à jour la spécification du protocole de votre messagerie pour gérer ces deux nouvelles fonctionnalités. En particulier donner le format des messages `JSON` échangés.
- Spécifier la sémantique des nouveaux messages ajoutés à la spécification.
- Spécifier le diagramme de séquences des différents messages nécéssaires pour le transfert de fichier.

2. On souhaite dorénavant permettre l'échange de fichiers entre clients.

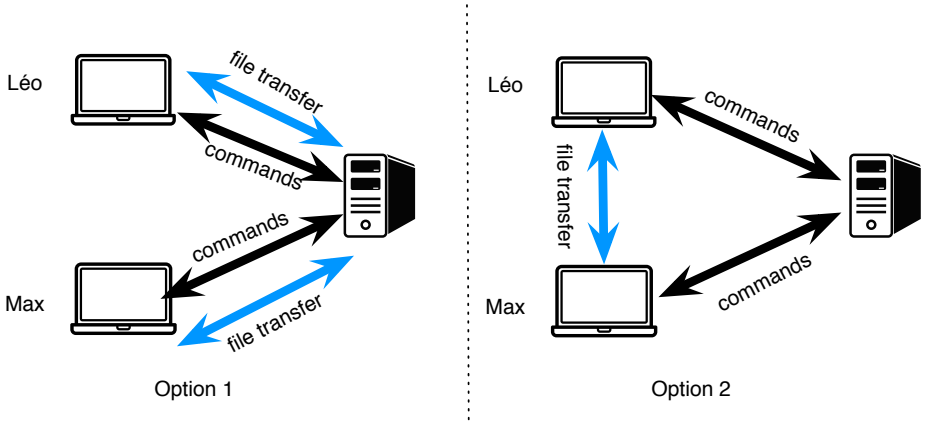

FIGURE 2 – Transfer de fichiers

Il existe deux techniques différentes :

- Transferer le fichier par l'intermédiaire du serveur.
- Transférer directement entre clients.

Implémenter les deux techniques, en particulier par l'ajout de la commande `dcc` (Direct Client-to-Client) et la commande `csc` (Client-to-Server-to-Client).

Historiquement, le transfer de fichier directement entre client a été implémenté pour la premiere fois dans `IRC`. Voir pour information http://www.irchelp.org/protocol/dccspec.html

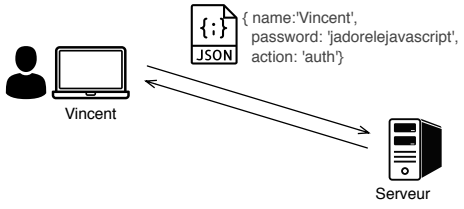

FIGURE 3 – Authentification basique

3. On souhaite rajouter de l'authentification. En particulier, on souhaite que le serveur puisse se souvenir des utilisateurs, pour notamment garder l'historique de leur conversation. Mettre en place un mécanisme d'authentification par mot de passe. Pour se faire, mettre à jour votre protocole de `tchat` afin que les utilisateurs puissent s'authentifier, voir Figure 3.

- Pour se souvenir de l'utilisateur, utiliser une base de données de type `sqlite`. En particulier, utiliser le module `sqlite3`.
- Les mots de passes ne doivent jamais être stockés en clair dans une base de données. Utiliser un algorithme de hash tel que `bcrypt` et utiliser le module node.js `bcrypt`. Voir https://fr.wikipedia.org/wiki/Bcrypt

4. Attention, **l'authentification ne rime pas avec sécurité**. On souhaite dorénavant tenter de renforcer un peu la sécurité entre le client et le serveur.

En particulier, on souhaite crypter les messages entre le client et le serveur sachant que le canal utilisé n'est pas sécurisé (Voir Figure 4).

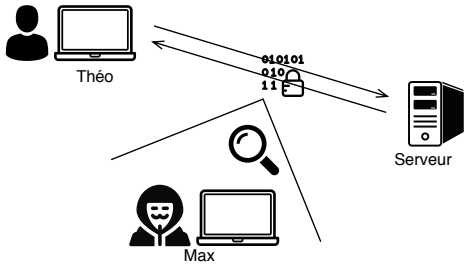

FIGURE 4 – Tchat sécurisé

Le client et le serveur ne se connaissent pas forcément à l'avance. En l'occurence `Théo` veut interagir avec le serveur sans que `Max` puisse interpréter et lire les messages échangés. Il est possible de crypter les messages (*cryptage symétrique*) en utilisant une clé ou un mot de passe, mais comment faire pour s'échanger le secret sans que `Max` l'intercepte ?

Pour résoudre ce problème, on va utiliser un mécanisme de *cryptage asymétrique*. Plus particulièrement, on va utiliser l'algorithme de Diffie-Hellman avec une courbe elliptique (`ECDH`). Voir https://fr.wikipedia.org/wiki/Échange_de_clés_Diffie-Hellman.

L'objectif d'utiliser `ECDH` est de permettre d'établir un secret entre deux parties à travers un canal non sécurisé sans s'échanger le secret !. `Théo` et le serveur génèrent tous deux une paire de clefs publique/privée en utilisant `ECDH`. Comme son nom l'indique la clé publique est publique et peut être partagée avec n'importe qui, mais par contre la clé privée doit rester confidentielle, et seul la personne qui l'a créé doit en avoir connaissance. `Théo` et le serveur peuvent alors s'échanger leur clé (publique). Enfin, un secret partagé pourra alors être généré de part et d'autre de la communication en combinant la clé privée et la clé publique reçues par le canal non sécurisé.

Si `Max` était en train d'écouter la communication, il a très certainement réussi à collecter la clé publique de `Théo` et/ou du serveur, mais il sera incapable de calculer le même secret sans les clés privées.

Une fois que `Théo` et le serveur disposent du même secret, ils sont alors en mesure d'utiliser un cryptage symétrique pour crypter leurs messages sans que `Max` ne puisse les comprendre.

- Mettre en oeuvre le protocole de sécurisation en utilisant `ECDH` entre les clients et votre serveur. Pour se faire, on utilisera uniquement le module nommé `crypto` de node.js. https://nodejs.org/api/crypto.html
- Pour utiliser `ECDH` voir la documentation https://nodejs.org/api/crypto.htmlcrypto_class_ecdh
- Pour le cryptage symétrique utiliser le cryptage `'aes-256-cbc'`. Voir https://fr.wikipedia.org/wiki/Advanced_Encryption_Standard
- Pour utilser `AES` avec `node.js`, voir la documentation https://nodejs.org/api/crypto.htmlcrypto_class_cipher et https://nodejs.org/api/crypto.htmlcrypto_crypto_createcipher_algorithm_password_options
- Il par ailleurs, il est très fortement recommandé d'utiliser un vecteur d'initialisation (`iv`). Voir https://en.wikipedia.org/wiki/Initialization_vector.
- Pour utiliser un `iv` en `node.js` voir la documentation https://nodejs.org/api/crypto.htmlcrypto_crypto_createcipheriv_algorithm_key_iv_options

5. Est-ce que cette solution est sécurisée et met-elle à l'abri `Théo` des intentions de `Max` de décrypter les messages échangés ? ;-)

6. Enfin on souhaite mettre en place la commande `sh` qui permet de créer une instance d'un `shell` côté serveur et d'interagir directement avec un `shell` dédié au client. Utiliser le module `node-pty` https://github.com/Tyriar/node-pty