Compte-rendu du projet de
Programmation des communications réseaux
UE3 : ARCHITECTURE ET SYSTÈMES

COMMUNICATION RÉSILIENTES AU BRUIT : INTRODUIRE ET CORRIGER DES ERREURS SUR CONTENU

23 mai 2024

CERISARA Nathan
JACQUET Ysée
JAYAT Adrien

Sommaire

1) Présentation du projet

a) Introduction / Description Rapide

Nous avons développé une application de messagerie avec des fonctionnalités très basiques que nous avons nommé ***NyanChat***.
C'est une messagerie où les clients peuvent s'inscrire et se connecter afin de discuter en temps réel avec les autres clients connectés.

Nous avons aussi développé la détection et la correction d'erreurs, que l'on introduit virtuellement par le proxy.

Nous avons utilisé la librairie *poll* pour gérer les connexions multi-clients, et l'écoute de l'entrée standard (*stdin*).

L'application se décompose en 3 parties bien distinctes:

- **Le serveur**, qui va gérer la réception des messages côté client, et s'occuper de renvoyer les messages reçu à tous les autres clients, tout en essayant de détecter des erreurs à la réception des messages grâce à des codes polynomiaux.
- **Le proxy**, qui sert juste d'intermédiaire entre les clients et le serveur. Il va aussi brouiller certains messages lors de l'envoi d'un message d'un client au serveur. Il y a aussi une petite détection basique de détection de messages problématiques, il ne va pas transmettre de messages vides par exemple.
- **Le client**, qui se découpe en 2 versions: une version terminal graphique, et une version très simple et basique. Les clients vont pouvoir se connecter grâce à un pseudonyme, et un code pour permettre d'empêcher d'autres clients de lui piquer son pseudonyme. Ensuite, les clients sont connectés à un caneau de messagerie publique, où tous les messages sont transmis à tous les clients connectés.

L'application est aussi disponible sur github: https://github.com/nath54/NyanChat.

b) Comment utiliser l'application (installation + utilisation)

1 - Installation

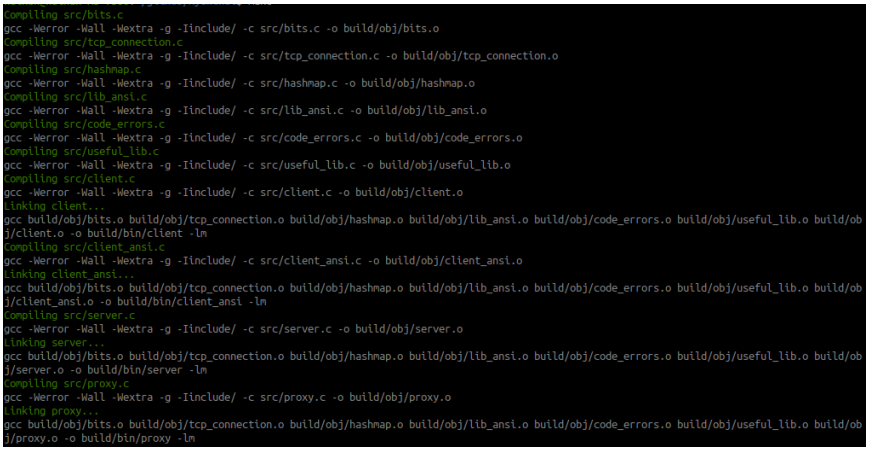
• La première étape est de récupérer les fichiers de l'application: soit depuis l'archive sur moodle, soit depuis la dernière release github (**#TODO: faire une release github, et mettre le lien ici, dès que tout le code sera bon**).

• Attention! Si vous avez cloné le répertoire depuis github, il faut aussi cloner les sous-modules (*unity* est un sous-module utilisé pour les tests) avec la commande `` `git submodule init && git submodule update` ``.

• Si vous avez récupéré une archive, il faut la décompresser, en utilisant soit votre explorateur de fichier, soit la commande `` `unzip` `` sur linux.

• Un fois que vous avez bien récupéré le dossier du projet, et qu'il est bien décompressé, il faut aller dedans depuis un terminal. Si vous utilisiez un explorateur de fichier jusque là, vous pouvez faire un clic droit, et cliquer sur l'option *ouvrir dans un terminal*. Sinon, vous pouvez directement ouvrir un terminal et utiliser la commande `` `cd chemin/de/NyanChat/` ``.

• Il faut maintenant bien compiler l'application, donc pour cela, il faut utiliser la commande `` `make` ``. Normalement, l'application devrait bien se compiler et vous devriez avoir le résultat suivant:



Après cette étape, l'application est donc bien mise en place et compilée. Il ne reste plus qu'à l'utiliser.

2 - Création d'un serveur

Pour créer un serveur, il faut lancer la commande :
`./build/bin/server *PORT_SERVEUR*`.

Le serveur est donc lancé, et écoute les connexions entrantes depuis le port demandé.
Il est possible que vous receviez l'erreur suivante : *Error during bind call -> : Address already in use*.
Cela signifie que le port d'écoute demandé n'est pas disponible, il faut donc changer de port d'écoute.

Si vous avez un peu de chance, si vous tapez `\quit` sur la console du serveur, le serveur se termine, mais lors des tests, cela ne marchait pas tout le temps.

3 - Création d'un proxy

Pour lancer un proxy, il faut lancer la commande :
`./build/bin/proxy *IP_SERVEUR PORT_SERVEUR PORT_ECOUTE_CLIENTS*`.

Si vous testez cette application en local sur votre ordinateur, l'ip du serveur sera 127.0.0.1, et ce sera aussi celle du proxy.

Le proxy est donc lancé, il va d'abord essayer de se connecter au serveur, et si il échoue, il va directement se quitter avec un message d'erreur, sinon, il va lancer l'écoute des connexions des clients.

Il est aussi possible que vous receviez l'erreur suivante : *Error during bind call -> : Address already in use*.
Cela signifie que le port d'écoute des clients demandé n'est pas disponible, il faut donc changer de port d'écoute pour les clients.

4 - Utilisation du client graphique

Pour lancer un client graphique, il faut lancer la commande :
`./build/bin/client_ansi *IP_PROXY PORT_PROXY 2> log.txt*`.

Si vous testez cette application en local sur votre ordinateur, l'ip du proxy sera 127.0.0.1.
Il faut bien faire attention à ne pas utiliser le même port pour le serveur et le proxy.

Le `2> log.txt` redirige la sortie erreur vers le fichier `log.txt`. Il faut absolument le faire, car sinon, les sorties d'erreurs ne seront d'une part pas lisibles, et d'autre part vont dégrader l'affichage de l'interface graphique.

Si l'application ne se lance pas, ou s'interromp à cause d'une erreur, vous pouvez voir l'erreur en regardant le contenu du fichier `log.txt`, par exemple avec la commande : `cat log.txt`.

Mais principalement, la principale raison d'un client graphique qui ne se lance pas, c'est qu'il n'arrive pas se connecter au proxy, il faut donc vérifier que vous avez bien donné la bonne adresse et le bon port du proxy, et que ce dernier est bien lancé.

Après avoir lancé l'application, vous arriverez sur une page de connexion :

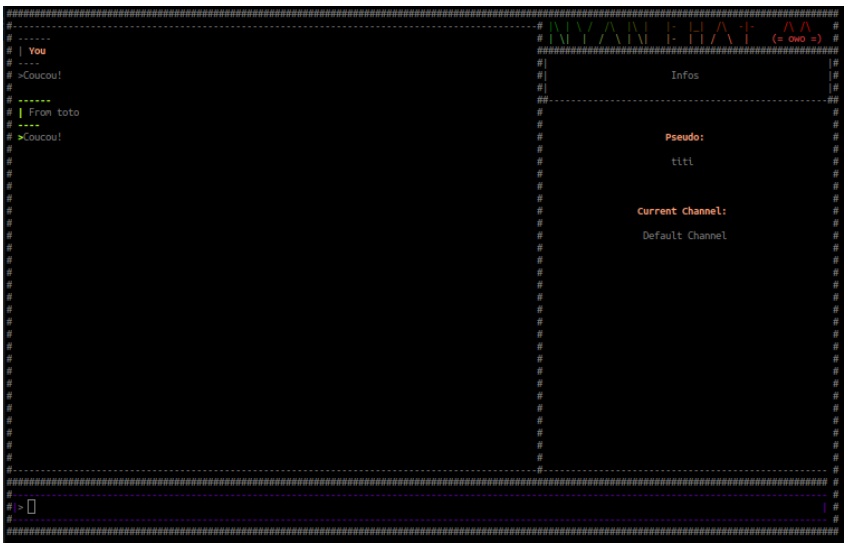
Il faudra donc ensuite rentrer le pseudonyme que vous voulez utiliser pour le client.

S'il ne se passe rien lorsque vous appuyez sur la touche *ENTRÉE* du clavier, c'est que soit:

- le pseudonyme demandé est trop court (il doit faire entre 4 et 64 caractères)
- le pseudonyme demandé est déjà utilisé par un autre client
- le client s'est déconnecté du proxy (le proxy a été fermé, manuellement ou à cause d'une erreur)
    - Il faudra dans ce cas là relancer le proxy (et attendre de pouvoir à nouveau accéder au port, ou utiliser un nouveau port d'écoute pour le proxy, et ensuite de relancer le client graphique).
- Il y a un bug au niveau du client.

Dans les deux derniers cas, n'hésitez pas à examiner le fichier `log.txt`.

Si la connexion s'est bien effectuée, vous devriez arriver sur une fenêtre qui ressemble à :



Vous avez, en bas, l'entrée utilisateur pour écrire les messages que vous voulez envoyer, et dans la partie en haut à gauche de la fenêtre, les messages reçus depuis le serveur. Pour envoyer un message, il suffit d'appuyer sur la touche *ENTRÉE* de votre clavier, vous pouvez naviguer parmi les messages reçus avec les flèches haut et bas du clavier, et vous pouvez quitter l'application en appuyant sur *Ctrl*-*Q*.

Comme nous n'avons pas eu beaucoup de temps pour développer l'interface graphique, il se peut qu'il y ait quelques bugs/erreurs.

5 - Utilisation du client très très basique

Pour lancer un client graphique, il faut lancer la commande :
`` `./build/bin/client *IP_PROXY PORT_PROXY*` ``.

Si vous testez cette application en local sur votre ordinateur, l'ip du proxy sera 127.0.0.1.
Il faut bien faire attention à ne pas utiliser le même port pour le serveur et le proxy.

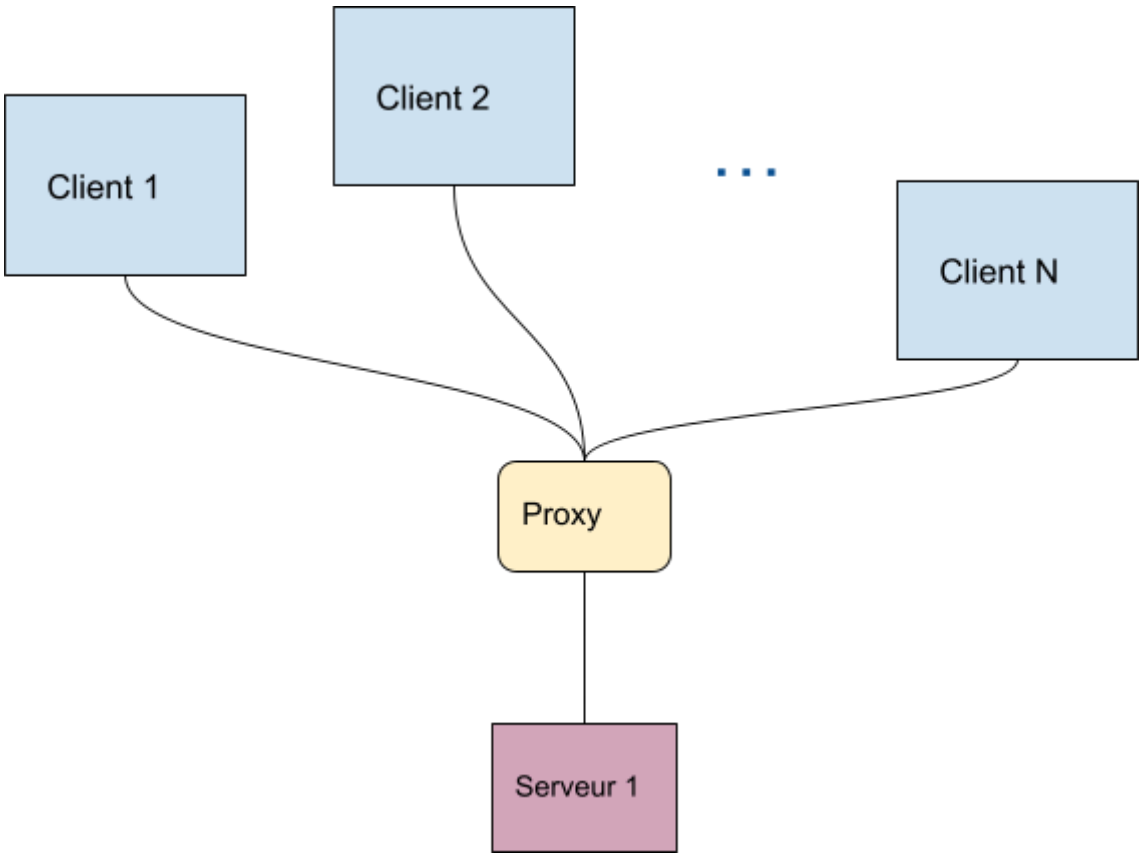
Il faudra ensuite envoyer le pseudonyme que vous souhaitez utiliser.

Ensuite, vous pourrez envoyer et recevoir des messages à votre guise.

c) Architecture Précise

1 - Graphe de l'architecture réseau

On a réalisé un réseau connectant un serveur, un proxy et plusieurs clients. Les échanges de données se feront uniquement selon les arcs du graphe ci-dessous (i.e. les clients et le serveur ne peuvent pas échanger directement entre eux).



2 - Explication de l'architecture des fichiers du projet

```
.
├── include (fichiers d'en têtes des codes C)
│   ├── bits.h (contient des fonctions de manipulation de bits d'octets)
│   ├── client.h (contient des structures pour garder les différents états du client)
│   ├── code_errors.h (contient des fonctions )
│   ├── hashmap.h (dictionnaire)
│   ├── lib_ansi_colors.h (définition de couleurs)
│   ├── lib_ansi.h (fonctions pour faire du graphisme sur terminal)
│   ├── lib_chks.h (définitions des CHK pour vérifier le bon résultat de fcts systèmes)
│   ├── lib_client_server.h (structures utiles pour le client et le serveur)
│   ├── server.h (structures pour le serveur)
│   ├── tcp_connection.h (connection socket)
│   └── useful_lib.h (petites fonctions utiles)
├── Makefile (Fichier de compilation)
├── README.md (Readme)
├── res (Ressources, logo, …)
│   ├── …
├── src (Codes sources)
│   ├── bits.c (fonctions de manipulation de bits d'octets)
│   ├── client_ansi.c (Client graphique)
│   ├── client.c (Client)
│   ├── code_errors.c (Gestion des codes d'erreurs)
│   ├── hashmap.c (Dictionnaire)
│   ├── lib_ansi.c (Fonctions graphiques)
│   ├── proxy.c (Proxy)
│   ├── server.c (Serveur)
│   ├── tcp_connection.c (connection socket)
│   └── useful_lib.c (petites fonctions utiles)
├── test (Tests (compiler avec `make test_…`)
│   ├── test_ansi.c ()
│   ├── test_code_errors.c ()
│   └── test_hashmap.c ()
```

d) Description détaillée de la partie sockets / TCP

1 - Polling, explication des fonctions `tcp_connection_...(...)`

Nous avons donc utilisé *poll* pour gérer plusieurs clients et l'entrée standard en même temps.
Comme beaucoup de parties de codes étaient très similaires entre un serveur et un client, nous avons réuni toute la partie connexion dans le fichier `tcp_connection.c`.

On peut donc initialiser une connexion avec `tcp_connection_server_init(...)` ou bien `tcp_connection_client_init(...)`.

Une fois connecté, ou en écoute, on peut lancer la boucle principale de la connexion socket avec la fonction `tcp_connection_mainloop(TcpConnection* con, fn_on_msg on_msg, void* on_msg_custom_args, fn_on_stdin on_stdin, void* on_stdin_custom_args);`

La boucle principale va écouter tous les événements reçus par *poll*. On peut distinguer plusieurs cas:

- Une connexion entrante (pour connexion de type serveur)
- Un message reçu du socket.
- Un message reçu de l'entrée standard.

Dans les deux derniers cas, on va appeler respectivement la fonction donnée en argument `on_msg_custom_args` avec comme argument supplémentaire `on_msg_custom_args`, et la fonction donnée en argument `on_stdin` avec comme argument supplémentaire `on_stdin_custom_args`.

Les codes respectifs aux serveurs, proxy et clients vont donc avoir comme points d'entrée les fonctions passées en arguments de `tcp_connection_mainloop`.

2 - Explication de la structure d'un message

Un message va avoir la structure suivante:

```c
typedef struct __attribute__((packed, aligned(4))) Message {
    /*
    Type of message: see MSG_.....
    */
    int msg_type;

    /*
    Identifier which allows the client to know which message
     the server is talking about when an acknowledgment is sent
    */
    int msg_id;
```

```
    // Pseudo of the client
    char src_pseudo[MAX_NAME_LENGTH];

    // Client socket of the proxy (filled by the proxy)
    int proxy_client_socket;

    /*
    Flag for destination type:
    0 = default channel
    1 = private channel
    2 = private message
    */
    int dst_flag;

    // Pseudo of the client destination, or name of the channel
    char dst[MAX_NAME_LENGTH];

    // Length of the message
    uint32 msg_length;

    // Message from the user
    char msg[MAX_MSG_LENGTH];

    // (for detection & correction)
    char control[MAX_MSG_LENGTH];

    // error[i] is true if msg[i] has an error not corrected
    bool error[MAX_MSG_LENGTH];
    // Number of retransmissions due to NEGATIVE ACK
    int nb_retransmission;

} Message;
```

Les valeurs les plus importantes de cette structure sont :

- msg_type : indique le type du message, permet de différencier un message de connexion, un message d'erreur, un acquittement, …
- msg: le contenu textuel d'un message, la donnée importante à transmettre
- src_ pseudo: le pseudo du client lorsqu'il envoie un message au serveur

Les autres valeurs sont aussi importantes et très utiles, mais nous n'avons pas le temps de toutes les expliquer ici, il faudra lire les commentaires dans le code directement.

3 - Explication de la connexion d'un client

Un client va être défini par un pseudo, et son id du socket de polling dans le proxy.
Pour le pseudo, s'il essaie de l'utiliser pour la première fois, il va générer un code qu'il va enregistrer dans un fichier, puis va envoyer un message de type connexion au serveur, qui va comparer s'il a déjà vu un code associé au pseudo demandé, et si c'est le cas, il va

comparer ce code avec le code envoyé par le client. Si c’est différent, le serveur va envoyer un message d’erreur au client. Sinon, le serveur va envoyer un message indiquant une connexion réussie.

e) Description de la partie codes polynomiaux

Voir les réponses aux questions du sujet.

2) Réponses aux questions du sujet

Question 1

Quel est le rendement du code polynomial que vous avez choisi ? Et son pouvoir détecteur ? Correcteur ? Enfin, ses capacités relatives ? Justifiez l'ensemble de ses performances.

Polynôme : $P(X) = X^8 + X^7 + X^5 + X^4 + X^3 + 1$
On a 8 bits d'information pour 16 bits encodés (8 bits de contrôle donc), on a donc un rendement de 50%. Sa capacité de détection s'élève à 3 bits erronés maximum, et sa capacité correctrice est de 1. Cela s'explique par sa distance de Hamming, qui vaut 4 :

- $4 \geq k + 1$ avec k le nombre d'erreurs détectées
- $4 \geq 2 \times k + 1$ avec k le nombre d'erreurs corrigées

Question 2

Décrivez votre architecture et plus particulièrement les modalités de votre procédure de décodage : correction et/ou détection et les mécanismes de retransmission mis en œuvre.

Question 3

Pour simplifier la prise en main de votre travail, sa lecture, et le débogage, commencez par écrire quatre fonctions utilitaires : `set_nth_bit`, `read_nth_bit`, `print_word` *et* `chg_nth_bit`.

Cf. fichier `bits.c`.

Question 4

Vous implémenterez maintenant une fonction `uint16_t encode_G(uint16_t m)` *[...]. Expliquez votre implémentation. Est-ce que votre code est circulaire ?*

Il s'agit de la fonction `uint16_t encode(uint16_t G[K][N], uint16_t m)` dans le fichier `code_errors.h`.

La fonction `encode` correspond essentiellement à un produit matriciel entre la matrice de contrôle et le message à encoder. Comme on est dans F2, les * sont remplacés par des & et les + par des |.
Pour chaque colonne des bits de contrôle, on calcule le bit de parité et on modifie le bit correspondant.

Le code n'est pas circulaire car les mots encodés ne sont pas cycliques (ie une permutation circulaire d'un mot ne donne pas toujours un nouveau mot possible).

Question 5

Vous implémenterez le calcul du cardinal des bits à 1 dans un mot de votre code.
Écrivez ensuite une fonction qui calcule la distance de Hamming de votre code polynomial (représenté par la matrice génératrice G dérivée de celui-ci). Expliquez votre algorithme et vérifiez que votre analyse était bonne à la question 1.

Cf fonctions `int weight(uint16_t m)` dans le fichier `bits.h` et `int code_hamming_distance(uint16_t G[K][N])` dans le fichier `code_errors.h`.

La fonction `code_hamming_distance` va, pour chaque mot (de 1 à 255), encoder ce dernier grâce à la matrice génératrice G avant de calculer le poids du mot encodé (i.e. distance de Hamming au mot nul). Parmi tous ces poids, on renverra le plus petit, qui correspond à la distance de Hamming de notre code polynomial.

Question 6

Après avoir réalisé le codage et décodage sous forme matricielle, faites en de même sous forme d'une division émulant du mieux possible un registre à décalage avec rétro-action logiciel.

Cf `uint16_t rem_lfsr(uint16_t p, uint16_t x)` dans le fichier `code_errors.h`. `p` est le polynôme d'encodage faisant office de diviseur, et `x` le dividende. La fonction retourne le reste de la division.

Question 7

Finalisez l'implémentation de votre client, serveur et proxy pour qu'ils supportent chacun la fonctionnalité attendue : codage, bruitage et décodage. Enfin, ajoutez un mécanisme de retransmission en cas de détection d'erreur non corrigeable.

3) Conclusion

Nous avons donc réalisé une application de messagerie multi-client avec détection d'erreurs.
Nous avons voulu faire une application très ambitieuse en quelques jours, nous avons réussi à faire beaucoup de choses, mais malheureusement, il y a beaucoup de choses que nous avons préparées dans le code, et que nous n'avons pas pu implémenter (par exemple: un système de salon privé). Et il peut rester quelques bugs dans l'application.

A cause du manque de temps, nous ne sommes pas certains que la détection/correction d'erreur fonctionne bien, mais elle est théoriquement correcte.

Le registre à décalage est malheureusement curieusement non opérationnel : il fonctionne comme attendu sur un test complexe qui a été en parallèle vérifié à la main (cf `make test`) ; mais lorsqu'on l'utilise, par exemple, à la place de l'encodage avec matrice génératrice lors de la recherche de la distance de Hamming du code polynomial, la valeur renvoyée pour cette distance n'est pas la bonne…