Projet MIRC

Antonin BRETAGNE

1 Exécuter le projet

1.1 Dépendances

Le projet est programmé en Rust. C'est un langage qui, tout en restant bas niveau, offre des garanties de sécurité largement supérieures au C et au C++.

La sécurité est une dimension importante des programmes, d'autant plus quand le programme reçoit et traite des données inconnues provenant d'autres utilisateurs.

Le programme requiert:

- Cargo (Rust, toolchain *nightly*)
- Testé sous Linux (Linux Mint, Ubuntu 22.04.1 LTS) et Windows 10

1.2 Compilation

Le projet peut être compilé avec `cargo build --release` ou plus simplement `make` si le logiciel *Make* est installé.

1.3 Exécution

L'exécutable est situé à l'emplacement "target/release/projet" (projet.exe sous Windows).

La liste des arguments possibles peut être obtenue en exécutant :

```
target/release/projet --help
```

Ces arguments sont détaillés ci-dessous.

Arguments positionnels (requis) Le programme doit être exécuté avec les arguments suivants :

```
target/release/projet <port> <neighbours...>
```

- `port` est le port UDP sur lequel le socket doit envoyer et recevoir les messages. Le port doit être valide et disponible, sinon le programme ne démarrera pas.
- `neighbours` contient les adresses des premiers voisins à contacter, séparées par des espaces. L'ordre dans lequel sont spécifiées les adresses est indifférent. L'argument doit contenir au moins une adresse valide, sinon le programme ne démarrera pas. Si l'argument contient par ailleurs des adresses invalides, ces adresses sont ignorées et un message d'erreur (non fatal) est affiché.

Note: il est parfois nécessaire de spécifier plusieurs voisins initiaux pour que le programme fonctionne. Voir la section "Invitation et auto-invitation" ci-dessous pour les explications.

Exemple pour écouter le port 40000 et communiquer avec un voisin local sur le port 40001 en IPv6 :

```
target/release/projet 40000 [::1]:40001
```

Arguments optionnels

- `-v` ou `--verbose` indique le niveau de verbosité du programme.
- `-n` ou `--nickname` est le pseudo de l'utilisateur. S'il est spécifié, il sera ajouté au début de tous les messages.

Niveaux de verbosité: Les niveaux de verbosité sont définis dans le fichier *src/api/logging.rs*. Dans les arguments de la ligne de commande, il est possible d'utiliser soit la position du niveau (déconseillé, car la valeur n'est pas explicite et peut changer lorsque des niveaux sont ajoutés ou supprimés), le nom complet du niveau et parfois des alias plus courts.

Niveau	Position	Nom (en ligne de commande)	Alias
Disabled	0	disabled	off
InternalError	1	internal-error	error, err
Warning	2	warning	warn
Anomaly	3	anomaly	
Important	4	important	
Information	5	information	info
Trace	6	trace	
Debug	7	debug	
Full	8	full	all

Si l'argument `-v` est omis, ou que la valeur spécifiée soit invalide, le niveau "Anomaly" est utilisé par défaut. Si une valeur numérique plus grande que 8 est donnée, le niveau "Full" est utilisé.

2 Test

Le fichier `tests/test.py` automatise le lancement de plusieurs clients "en cercle" dans des terminaux différents, mais n'effectue aucun test supplémentaire.

3 Documentation et choix d'implémentation

Le dépot GitHub du projet est disponible à l'adresse : `https://github.com/E-Mans-Application/ProjetReseau`.

Le code est entièrement documenté. La documentation est disponible soit directement dans les fichiers sources, soit à l'adresse `https://eman-mirc-doc.000webhostapp.com/`.

3.1 Synchronisation

Le choix a été fait d'utiliser le moins de `threads` possibles afin d'éviter le surcoût dû aux structures de synchronisation telles que les mutex.

Le thread principal lit les messages que l'utilisateur entre sur l'entrée standard. Un thread d'arrière plan s'occupe de tout le reste. Le socket est passé en mode "non bloquant".

Pour offrir une interface confortable à l'utilisateur, la bibliothèque `rustyline` est utilisée.

Pour que le thread principal communique de manière sûre avec le thread d'arrière plan, une "Multi-producer multi-consumer" channel est utilisée. L'implémentation utilise la bibliothèque `crossbeam`.

3.2 IPv4

Le protocole principal utilisé par le socket est IPv6, mais le programme peut envoyer et recevoir des messages vers/depuis des adresses IPv4 sur les systèmes qui supportent le "dual-stack". Cette fonctionnalité peut être désactivée en mettant la constante "`IPV6_ONLY`" à `true` dans le fichier `src/api/lib.rs`.

La structure `Addr` (fichier `src/api/addresses.rs`) permet d'automatiser la gestion des adresses IPv4.

3.3 Nombre maximal de données récentes

Le nombre maximal de données récentes peut être spécifié par la constante `MAX_RECENT_DATA_COUNT` dans le fichier `src/api/lib.rs`.

Plutôt qu'une liste circulaire, la combinaison d'un `HashMap` et d'un `VecDeque` est utilisée pour conserver les performances d'accès par clé du `HashMap` tout en mémorisant l'ordre d'insertion des données.

Par ailleurs, la constante `MAX_RECENT_DATA_AGE` permet de limiter la durée d'inondation d'une donnée.

3.4 Invitation et auto-invitation

Le programme peut recevoir des messages de tous ses voisins **invités**.

L'expression voisin invité est synonyme de voisin potentiel, mais insiste sur le caractère restrictif de l'invitation.

Un voisin est invité si :

- Son adresse est spécifiée au démarrage dans l'argument `neighbours`
- Son adresse est communiquée (dans un TLV Neighbour) par un voisin déjà invité
- La constante `ALLOW_SELF_INVITATION` est "true" et le voisin envoie un message UDP quelconque

Les messages des voisins non invités sont ignorés (ils ne sont même pas "parsés").

La version du protocole IP utilisé par l'adresse est importante, notamment dans le cas des adresses locales. Un voisin invité avec l'adresse `[::1]:port` ne

pourra pas envoyer de messages avec l'adresse `127.0.0.1:port`, et réciproquement.

Autoriser l'auto-invitation peut être dangereux, car cela peut permettre à un utilisateur malveillant de rejoindre la conversation.

Néanmoins, la désactivation de l'auto-invitation peut compliquer la communication entre trois clients ou plus. Pour connecter plusieurs clients "en cercle", il est nécessaire de spécifier à la fois les adresses du voisin suivant et du voisin précédent.

Pour des raisons de clarté, l'invitation d'un voisin est représentée au niveau du typage. Les adresses des voisins invités ont en effet le type `& 'arena Addr`.

La liste des voisins potentiels utilise un HashSet. Les éléments du HashSet ne sont jamais supprimés. Il n'est pas possible d'annuler une invitation ; en revanche, les voisins qui violent le protocole peuvent être temporairement bloqués.

3.5 Découverte des voisins

Le programme envoie périodiquement des "Hellos" courts à tous ses voisins **invités** et non symmétriques. Le programme envoie également immédiatement un "Hello" court à tous les voisins invités par un TLV "Neighbour".

Le programme envoie régulièrement (toutes les 30 secondes) la liste de ses voisins symmétriques à tous ses voisins actifs. Lors du démarrage du programme, le premier envoi est anticipé afin d'accélérer la liaison.

Le programme n'ajoute pas sa propre adresse à la liste de ses voisins, mais il peut devenir voisin de lui-même lors de la communication avec des voisins non-locaux.

Sous Unix, une méthode expérimentale est mise en place pour limiter ce problème.

3.6 Gestion des erreurs

Si une erreur survient au démarrage lors de la création du socket, le programme est immédiatement arrêté avec un message d'erreur.

Par la suite, si une erreur survient lors de l'envoi ou de la réception d'un message, l'erreur peut être affichée si le niveau de verbosité est suffisamment élevé, mais aucune autre action n'est entreprise.

3.7 Agrégation

Le programme utilise des files d'attente de TLVs à envoyer à chaque voisin, ce qui permet d'agréger plusieurs TLVs dans le même message et d'envoyer des datagrammes contenant plusieurs TLVs

La file d'attente est vidée régulièrement, ce qui force l'envoi des TLvs même si les datagrammes ne sont pas pleins.

Si le message entré par l'utlisateur est trop gros pour entrer dans un seul TLV, il est divisé en plusieurs TLVs. Cependant, il n'y a alors aucune garantie que les TLVs soient reçus dans le bon ordre par les voisins, d'autant plus qu'il n'y a aucune raison que les délais d'inondation tirés au sort pour le TLV qui contient le début du message soient inférieurs à ceux des autres TLVs.

Le protocole n'est pas adapté à l'envoi de messages longs. Le protocole ne spécifie pas que `Nonce` dans l'id du message doit reproduire l'ordre des messages, donc il n'est pas possible d'utiliser l'id des messages pour les trier dans l'ordre.

3.8 Délais d'inondation

Les délais d'inondation sont tirés aléatoirement pour chaque voisin. La date de la prochain inondation est une fonction du couple (id du message, voisin à inonder).

Le point négatif est, comme indiqué plus haut, que les TLVs associés à un même message (trop long pour un seul TLV) peuvent être inondés dans le désordre, mais cette situation a été jugée trop anecdotique pour remettre en question le procédé de tirage au sort des délais d'inondation.

3.9 Adresses multiples

La table des voisins actifs est indexée par l'adresse uniquement. Rien n'empêche deux voisins actifs différents d'avoir la même ID.

Si un voisin a plusieurs adresses mais une seule ID, le voisin communique avec toutes les adresses actives de ce voisin (celles qui ont envoyé "Hello" récemment).

Le fait que 2 voisins aient la même ID peut seulement poser des problèmes de collision dans les IDs des messages qu'ils envoient ; mais il est de la responsabilité de l'utilisateur qui utilise plusieurs adresses de synchroniser les IDs de ses messages.

En supposant que la génération des IDs aléatoires suive une loi uniforme parfaite, la probabilité que plusieurs utilisateurs différents se retrouvent avec la même ID aléatoire est de $2,7 \cdot 10^{-14}$ dans le cas de 1000 clients. Bien qu'en pratique, la génération ne soit pas parfaitement aléatoire, il semble raisonnable de négliger l'éventualité de collisions entre IDs.

3.10 Sécurité

Rust assure qu'il n'est pas possible de lire la mémoire au-delà du message reçu.

Si l'en-tête du message spécifie une taille plus grande que le message effectivement reçu, le message entier est ignoré. Il en est de même si un TLV déborde du message.

Les TLVs Data envoyés par des voisins non symmétriques sont ignorés.

De plus, chaque fois qu'un voisin viole le protocole, il est exclu de la liste des voisins actifs et bloqué pendant une durée qui augmente exponentiellement en fonction du nombre de violations passées. Le voisin ne peut plus redevenir actif tant qu'il est bloqué. Tous les messages reçus d'un voisin bloqué sont ignorés, de la même manière que si le voisin n'était pas invité.

3.11 Bibliothèques ("crates") utilisées

- `derive_more`
- `rand`
- `bumpalo`

- `clap` (avec fonctionnalité `derive`)
- `crossbeam`
- `chrono` (avec fonctionnalité `clock`)
- `rustyline`
- `socket2`
- `pnet` (Unix uniquement)