Projet de programmation réseau
BROUILLON

Juliusz Chroboczek

10 Mars 2009

Introduction

Le but de ce projet est d'implémenter un système de messagerie instantanée *(chat)* en ligne, c'est à dire une application réseau permettant à des utilisateurs humains d'avoir une ou plusieurs conversations écrites à travers le réseau.

Le protocole qui vous est proposé est très flexible, et vous êtes libres de choisir l'interface utilisateur qui vous convient. Par exemple, vous pouvez implémenter une interface en mode texte ou une interface graphique, une interface permettant d'avoir une seule conversation ou plusieurs simultanément, une interface pour un seul utilisateur ou pour plusieurs. Vous êtes aussi libres de choisir le langage de programmation qui vous convient (du moment que je le connais et qu'il est implémenté sur les machines de l'UFR), mais le code qui vous est fourni est écrit en Java.

Par contre, le protocole est défini de façon précise par cet énoncé, et votre implémentation doit absolument être interopérable avec les autres implémentations du protocole (notamment la mienne). Pour ceux d'entre vous qui auraient envie d'aller au-delà du sujet, le protocole inclut un mécanisme d'extensions (voir paragraphe 4.1).

Un programme qui n'interopère pas avec mon implémentation du protocole ne sera pas accepté comme solution à ce projet.

Protocole minimal et étendu

Le projet est structuré en un protocole « minimal », décrit à la section 3, et un certain nombre d'extensions. Je m'attends à ce que tout le monde implémente le protocole minimal, voudrais croire que beaucoup d'entre vous s'attaqueront aux extensions, et j'espère que certains définiront des extensions auxquelles je n'ai pas pensé. Je vous conseille de lire la totalité de ce document avant de commencer à implémenter le protocole minimal.

Déroulement

Le projet sera normalement réalisé par groupes de trois étudiants. Nous accepterons les projets soumis par des groupes de deux ou même un seul étudiant, mais les projets soumis par de petits groupes seront jugés selon les mêmes critères que ceux des groupes de trois personnes.

Nous n'accepterons pas qu'un projet soit soumis par un groupe de plus de trois étudiants.

Nous vous demanderons de nous fournir avant la soutenance :

– une archive (`tar.gz`, `zip` ou `jar`) contenant votre code source ainsi qu'un fichier nommé `README` qui indique clairement comment le compiler ; votre source devra être compilable sur au moins une des machines de l'UFR[1] ;
– un rapport de 5 à 20 pages environ, dans un format qui peut être lu et imprimé avec des outils libres[2].

Le rapport devra donner une vision globale de votre projet, indiquer clairement quelles sont les parties du projet que vous aurez traitées et préciser les choix techniques non évidents que vous aurez faits (par exemple, vous pourriez nous indiquer comment vous avez résolu le problème de synchronisation indiqué au paragraphe 1.2, ou comment vous avez évité le *livelock* indiqué au paragraphe 3.4).

Il n'est pas absolument nécessaire de paraphraser l'énoncé du projet dans le rapport.

1 Structure du protocole

Le protocole est un protocole pair-à-pair complètement décentralisé et symétrique : il n'y a pas de serveur central servant à coordonner les pairs, et, à part au moment de l'établissement d'une connexions, où un pair joue un rôle passif (serveur) ou actif (client), tous les pairs jouent le même rôle.

Je vous propose d'implémenter votre pair de façon incrémentale : commencez par un pair assez simple, puis étendez ses fonctionnalités.

1.1 Vision générale

Chaque pair attend des connexions sur un ou plusieurs socket TCP (adresse IP et port), IPv4, IPv6 ou un mélange des deux.

Un pair peut aussi établir activement des connections TCP avec d'autres pairs. Lorsqu'il établit une connexion, il l'établit à partir d'un numéro de port sur lequel il attend des connexions : les données de couche transport de la connexion indiquent au pair passif le numéro de port sur lequel le pair actif accepte des connexions.

Un utilisateur, humain ou pas, est identifié par un *pseudonyme*, qui est une chaîne de caractères supposée globalement unique[3]. Un pair peut être *responsable* de zéro, un ou plusieurs pseudonymes : pour envoyer un message destiné à un pseudonyme ψ, il faut l'envoyer au pair responsable de ψ.

Lorsque l'utilisateur d'un pair A désire commencer une conversation avec un pseudonyme ψ, le pair A établit une connexion avec un pair B qui est supposé responsable pour ψ. Il envoie alors le message destiné à ψ à B. Si B est effectivement responsable pour ψ, il informe A du succès de

1 Si vous voulez utiliser des logiciels qui ne sont pas installés, prévenez-nous suffisamment à l'avance.

2 Par exemple le format PDF.

3 En d'autres termes, je ne m'attends pas à ce que vous résolviez le problème de l'unicité globale des pseudonymes — on les suppose uniques, et on espère pour le mieux.

l'opération ; sinon, il informe A de son échec (paragraphe 3.4). Dans ce dernier cas, A peut soit repéter l'opération avec un autre pair, ou retourner un message d'erreur à l'utilisateur[4].

Gestion des pairs Typiquement, un pair participant au protocole sera connecté simultanément à plusieurs pairs. De plus, il maintiendra probablement une liste de pairs « connus » auxquels il n'est pas connecté, par exemple des pairs auxquels il s'est connecté auparavant ou des pairs obtenus par la requête `_FIND_FIND` (paragraphe 4.2).

Par ailleurs, un pair devra maintenir une structure de données qui associe un pseudonyme au pair qui en est responsable.

La nature précise de ces structures de données dépend de l'implémentation ; en d'autres termes, vous êtes libres de choisir les structures de données qui vous conviennent. Personnellement, j'ai implémenté ce sujet avec les structures de données suivantes :

- une liste de pairs connectés ; elle est initialement vide, et est mise à jour lors de chaque établissement ou fermeture de connection ;
- une table de pairs connus ; elle est initialement con□gurée par l'utilisateur ; un pair en est ôté lorsqu'une tentative de connection échoue, et un nouveau pair y est ajouté lors d'une □n de connection ainsi que lors d'une réponse `_FIND_ACK` (paragraphe 4.2) ; cette liste sert à fournir une liste de pairs à essayer de contacter lorsqu'un pseudonyme est inconnu ;
- une table de pseudonymes, qui à un pseudonyme associe un ou plusieurs pairs qui sont soupçonnés d'en être responsables ; cette table est mise à jour à l'aide de la requête `LISTPSEUDOS` (paragraphe 3.5), ainsi qu'a chaque invocation du sous-protocole `FIND` (paragraphe 4.2).

D'autres choix sont bien-sûr possibles, et une version antérieure de mon code utilisait des structures de données di□érentes (la table de pseudos était scindée en deux tables, une table de pseudos connectés, et une table de pseudos connus).

1.2 Structure du protocole

Les messages du protocole sont divisés en requêtes, réponses et messages non-solicités. Une réponse est envoyée en réponse à une requête, et chaque requête solicite une réponse : il y a donc une correspondance biunivoque entre requêtes et réponses. Par contre, un message non-solicité peut être envoyé à tout moment, et ne solicite aucun autre message. Un message non-solicité a un rôle purement informatif : un message non-solicité peut toujours être ignoré.

À la di□érence des protocoles client/serveur, on utilise une seule connexion TCP pour la communication entre deux pairs ; de ce fait, il n'y a pas une direction pour les requêtes et une autre pour les réponses, requêtes et réponses peuvent transiter dans les deux sens. En particulier, si un pair A envoie une requête m_1 à un pair B, il n'est pas certain que le premier message reçu sur la connexion soit la réponse à m_1 : il se peut que B envoie une ou plusieurs requêtes à A avant de répondre à m_1 (voir □gure 1).

1.3 Qualité de l'implémentation

Même si le protocole est □xé, il est possible de l'implémenter de façon plus ou moins maline. En particulier, une implémentation de qualité sera non seulement capable d'interopérer avec des

4 Remarquez qu'une *race condition* nous empêche de garantir qu'on enverra toujours un message au bon pair.

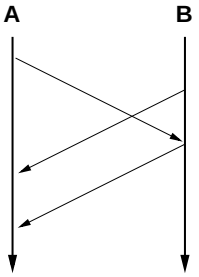

Figure 1: Une requête qui « double » une réponse

implémentations correctes du protocole, mais évitera aussi de planter en présence d'une implémentation incorrecte.

Le protocole est conçu pour permettre le *pipelining*, c'est-à-dire la possibilité d'avoir plusieurs requêtes « en vol » (qu'on a émises mais pour lesquelles on n'a pas encore reçu de réponse), ce qui permet de cacher partiellement la latence du réseau. Une implémentation de qualité utilisera le *pipelining* de façon aussi agressive que possible, mais pas davantage.

Vous pourrez aussi réfléchir à la taille des segments que votre implémentation permet à la couche TCP de produire, et en particulier à la façon dont vous interagissez avec l'algorithme de Nagle. (Si votre stratégie d'émission est maline, une approche raisonnable consiste à simplement éteindre l'algorithme de Nagle.)

2 Syntaxe des messages

Un formateur et un analyseur lexical vous sont fournis ; il n'est donc pas nécessaire de comprendre entièrement la structure syntaxique du protocole.

Le protocole est défini comme un protocole texte, ce qui facilite son déboggage (il est possible en particulier d'utiliser la commande `telnet` pour tester une implémentation). Toutes les chaînes manipulées par le protocole sont codées en UTF-8.

Chaque message est représenté sur le fil comme une ligne au sens de la commande `telnet`, soit une suite d'octets ne contenant pas l'octet NL de valeur 10 (représentant le caractère « `\n` », et terminés par l'octet NL.

> **Remarque :** à la différence du C, le langage Java sépare la notion d'octet de la notion de caractère. Une chaîne de caractères peut être convertie en suite d'octets à l'aide de la méthode `String.getBytes(String)`, une suite d'octets en chaîne de caractères à l'aide du constructeur `String(byte[], String)`. Cette dernière conversion peut bien-sûr échouer lorsque la suite d'octets passée en paramètre ne correspond pas à une suite de caractères dans le codage utilisé.
>
> Dans la suite de ce document, je me permets parfois de confondre suites d'octets et suites de caractères.

2.1 Structure lexicale

On définit les unités lexicales suivantes :

- Un *entier décimal* est une suite de caractères constituée entièrement de chiffres.
- Un *atome* est une suite de caractères ASCII commençant par une lettre majuscule ou un souligné « _ » et constitué entièrement de lettres majuscules, de chiffres ou du caractère souligné.
- Une *chaîne de caractères* consiste d'un caractère *parenthèse ouverte* « (» suivi d'un nombre arbitraire d'octets suivis d'une parenthèse fermée «) ». Les octets de valeur inférieure à 0x1F, ainsi que ceux correspondant au caractères « (», «) » et « \ » doivent être protégés par un *backslash*. Une telle « chaîne de caractères » peut représenter une vraie chaîne, codée en UTF-8, ou simplement un bloc de données binaires.
- Une *liste* est un caractère « [», suivi d'une suite d'unités lexicales séparées par des espaces, suivie d'un caractère «] ».
- Une *adresse de socket* a la forme « |*a*|*p*| » où *a* est une adresse IPv4 ou IPv6, et *p* est un numéro de port (un entier). Par exemple, une adresse de socket IPv4 peut s'écrire sous la forme « |169.254.191.168|9999| », et une adresse IPv6 « |2001:660:3301:8063::1|9999| ».

2.2 Structure syntaxique

Chaque message a la structure suivante :

- un caractère indiquant s'il s'agit d'une requête, d'une réponse, ou d'un message non-solicité ; il vaut « ? » pour une requête et « ! » pour une réponse, et « – » pour un message non-solicité ;
- un atome indiquant le type de message ;
- les paramètres du message, une suite d'unités lexicales séparées par des espaces ;
- un caractère de fin de ligne (10, « \n »).

Par exemple, une requête `HELLO` pourra consister d'une seule ligne contenant la chaîne de caractères suivante :

```
? HELLO (4.0) [FIND]
```

Les paramètres de la requête sont la sont la

Si B ne veut pas établir de connexion avec A, il répond par une réponse `HELLONAK` avec un message destiné à l'utilisateur humain pour donner la raison de ce refus. À la réception de `HELLONAK`, A doit fermer la connexion.

Si l'établissement de connexion échoue, ou si le pair B répond par autre chose qu'une réponse `HELLOACK`, indiquant qu'il n'implémente pas ce protocole, A enlève B de sa liste de pairs connus.

Format des messages

```
? HELLO version extensions
! HELLOACK version extensions
! HELLONAK version raison
```

Les champs utilisés dans ces messages sont définis comme suit.

- *version*, une chaîne, est la version du protocole, et vaut toujours `4.0`[5] ;
- *extensions* est une liste d'atomes ; dans le protocole minimal, on émet la liste vide, et on l'ignore dans les messages reçus ;
- *raison*, une chaîne, est la raison du refus.

> **Remarque :** Le champ *raison* de la réponse `HELLONAK` explique un problème, il est donc raisonnable de déranger l'utilisateur pour lui en faire part.

3.2 Fermeture de connexion

Un pair A qui désire fermer une connexion avec un pair B envoie une requête `CLOSE`. Si B est d'accord pour fermer la connexion, il répond par `CLOSEACK` ; A ferme la connexion TCP dès qu'il a reçu la réponse `CLOSEACK`. B peut aussi refuser de fermer la connexion (par exemple parce qu'il a encore des choses à dire) en envoyant une réponse `CLOSENAK`.

Ce *handshake* final évite les problèmes dans le cas où B a des requêtes en vol lorsqu'il reçoit une requête `CLOSE`.

Format des messages

```
? CLOSE
! CLOSEACK
! CLOSENAK
```

Les messages définis ici n'ont pas de paramètres.

3.3 Requête nulle

La requête nulle `NOOP` n'a aucun effet. Un pair qui reçoit la requête nulle doit répondre avec une réponse `NOOPACK`.

Le message non-solicité nul `NOOP` n'a aucun effet. Un pair qui reçoit le message non-solicité nul l'ignore.

5 Eh oui, c'est déjà la quatrième mouture de ce projet...

Remarque : il n'est pas nécessaire de générer des requêtes nulles, mais il est obligatoire d'y répondre.

Format des messages

? NOOP

! NOOPACK

– NOOP

3.4 Envoi d'un message à un pseudonyme

Lorsqu'un pair A décide d'envoyer un message à un pseudonyme ψ dont B est responsable, il envoie à B un message SEND. Si B accepte ce message, il répond avec SENDACK. Si B n'est pas responsable pour ψ, il répond normalement avec SENDUNKNOWN. Si B refuse ce message pour une raison quelconque, il répond avec SENDNAK.

Si B n'est pas responsable pour ψ, il peut optionnellement répondre SENDKNOWN au lieu de SENDUNKNOWN, indiquant l'adresse du pair dont il pense qu'il est responsable pour le pseudonyme demandé. Un pair implémentant le protocole minimal traite une telle réponse comme synonyme de SENDUNKNOWN ; un pair plus avancé peut réagir à une telle réponse en renvoyant la requête SEND au ou aux pairs indiqués par la réponse. (Il sera important dans ce cas d'éviter les boucles in□nies.)

Format des messages

? SEND orig pseudo message

! SENDACK

! SENDUNKNOWN

! SENDKNOWN pseudo addr-list

! SENDNAK raison

Le champ *message* (une chaîne) de la requête SEND contient le message destiné à l'utilisateur humain. Le champ *orig* contient le pseudo de l'émetteur, le champ *pseudo* celui du destinataire.

Le champ *pseudo* de la réponse SENDKNOWN contient le pseudo auquel était destiné le message ; il doit être identique au message correspondant de la requête SEND. Le champ *addr-list*, une liste d'adresses de socket, contient une liste d'adresses de pairs dont le pair émetteur pense qu'ils peuvent être responsables pour le pseudo considéré.

Le champ *raison* (une chaîne) de la réponse SENDNAK contient la raison du refus, sous une forme lisible par un être humain.

3.5 Liste de pseudonymes

Un pair A qui désire connaître la liste des pseudonymes pour lesquels est responsable un pair B, par exemple pour peupler sa liste de pseudonymes connus ou pour l'a□cher dans une boîte de dialogue, envoie à B une requête LISTPSEUDOS. B répond avec une réponse PSEUDOSLIST, qui contient les pseudonymes dont il est responsable.

Format des messages

? LISTPSEUDOS

! PSEUDOSLIST list

Le champ *list*, une liste de chaînes, contient la liste des pseudonymes dont est responsable le pair qui l'émet.

3.6 Mise à jour des pseudonymes

Un pair peut à tout moment informer un autre pair qu'il est responsable d'un nouveau pseudonyme, ou qu'il cesse d'être responsable d'un pseudonyme donné. Cela se fait, respectivement, à l'aide des messages non-solicités NEWPSEUDO et OLDPSEUDO.

Format des messages

– NEWPSEUDO list

– OLDPSEUDO list

Le champ *list*, une liste de chaînes, contient une liste de pseudonymes dont le pair émetteur devient responsable ou cesse d'être responsable.

> **Remarque :** il n'est pas nécessaire de générer des messages de mise à jour, et il n'est pas non plus nécessaire de se servir de l'information qu'ils contiennent. Cependant, votre pair ne devra pas planter s'il reçoit un tel message.

3.7 Erreur

Lorsqu'un pair reçoit une requête qu'il ne comprend pas (par exemple parce que son pair est boggué), il y répond par une réponse BAD.

Un pair peut à tout moment envoyer un message non-solicité BAD qui indique une situation erronée à ses pairs, par exemple lorsqu'il a reçu un message non-solicité qu'il ne comprend pas.

Format des messages

! BAD raison

– BAD raison

Le champ *raison* est une chaîne de caractères expliquant la raison pour laquelle la requête n'a pas pu être comprise. Il peut être vide.

4 Extensions

Cette section dé nit les extensions au protocole minimal que nous vous encourageons à implémenter.

4.1 Mécanisme d'extension

Le protocole dé□ni dans ce document est extensible. Lors du *handshake* initial, les pairs en présence envoient des *listes d'extensions* dans les messages `HELLO` et `HELLOACK`.

Chacune des extensions déclarées à la syntaxe d'un atome. Si une extension s'appelle *ext*, elle peut dé□nir des nouveaux messages ayant la forme *_ext_req*, où *req* a la syntaxe d'un atome.

4.2 Recherche de pseudonyme

Ce paragraphe dé□nit l'extension nommée `FIND`.

Lorsqu'un pair A veut déterminer le pair responsable pour un pseudonyme ψ, il peut envoyer à un pair B implémentant l'extension `FIND` une requête `_FIND_FIND`. Si B ne connaît pas ψ, il répond par `_FIND_NAK`. S'il est responsable pour ψ, il répond par `FIND_ACK`. S'il n'est pas responsable pour ψ, mais il connaît ψ, il répond par `_FIND_KNOWN`, qui inclut la ou les adresses du pair qu'il considère comme responsable.

> **Remarque :** tout comme la requête `HELLO`, la réponse `FIND_ACK` ne contient pas l'adresse de B, que A connaît déjà. Cette approche permet d'éviter à B de connaître sa propre adresse.

Format des messages

? `_FIND_FIND` pseudo

! `_FIND_NAK` pseudo

! `_FIND_ACK` pseudo

! `_FIND_KNOWN` pseudo addr-list

La requête `_FIND_FIND` a un paramètre *pseudo*, une chaîne, qui dé□nit le pseudonyme à rechercher. Les réponses incluent aussi ce champ, ce qui peut simpli□er l'implémentation (n'hésitez pas à l'ignorer si ce n'est pas le cas).

Le champ *addr-list* est une liste d'adresses de socket.

4.3 A□chage de texte enrichi

L'extension nommée `RICHTEXT` permet d'envoyer des messages contenant plus que du texte, par exemple du texte enrichi (avec gras, italiques, couleurs etc.) ou des images.

4.3.1 Négociation d'un format enrichi

Un pair ϕ qui désire envoyer du texte enrichi à un pair ψ supportant l'extension `RICHTEXT` doit tout d'abord déterminer les formats de texte enrichi supportés par ce dernier. Il le fait en envoyant une requête `_RICHTEXT_FORMATS`. Le pair ψ répond par une réponse `_RICHTEXT_FORMATS` contenant la liste des noms de formats supportés.

Format des messages

? _RICHTEXT_FORMATS

! _RICHTEXT_FORMATS liste

Le paramètre *liste* est une liste d'atomes, un pour chaque format enrichi supporté (voir paragraphe 4.3.3.

4.3.2 Envoi d'un message enrichi

Après avoir déterminé que le pair ψ supporte un format f, le pair ϕ peut lui envoyer un message enrichi à l'aide d'une requête _RICHTEXT_SEND. Le pair ψ répond à cette requête comme à une requête SEND (paragraphe 3.4), c'est à dire par l'une des réponses SENDACK, SENDUNKNOWN, SENDKNOWN ou SENDNAK.

Format des messages

? _RICHTEXT_SEND format orig pseudo message

Le paramètre *format* est un atome identifiant le format du message. Les paramètres *orig* et *pseudo* sont analogues à ceux de la requête SEND (paragraphe 3.4). Le paramètre *message* est le message à envoyer ; son interprétation dépend de la valeur de *format*.

4.3.3 Formats enrichis

Les formats enrichis suivants sont définis :

- GIF : le message est une image au format GIF ;
- PNG : le message est une image au format PNG ;
- JPEG : le message est une image au format JPEG ou JPEG progressif ;
- ENRICHED : le message est au format `text/enriched`, tel que défini par la RFC 1896, codé en UTF-8 ;
- HTML : le message est au format `text/html`, version 2.0 ou 4.0, codé en UTF-8 (les déclarations « META » ou ISO 2022 du jeu de caractères ne sont pas autorisées).

Attention, les formats GIF, PNG et JPEG sont des formats binaires ; il faudra prendre soin de citer correctement leur contenu (ce que fait automatiquement la classe `LexString` fournie).

4.4 Autres extensions

Toutes les autres extensions seront les bienvenues. On peut par exemple penser à des extensions qui permettent l'authentification cryptographique des pairs ou des pseudonymes, la communication cryptée, la recherche de pairs sur le lien local ou abonnés à un groupe multicast, le transfert de fichiers, le transfert de données multimédia (audio, vidéo) etc.

Comme le mécanisme d'extension dépend du fait que les identificateurs d'extension sont globalement uniques, je vous demanderai de me communiquer à l'avance le nom de toute extension que vous aurez définie.