ESISAR

NE302 – Projet Réseau
Projet « HTTP Server » - part 3

1 Objectifs de l'étape 3:

L'objectif de l'etape 3 est la prise en compte des requêtes via le réseau et la conception et la réalisation de votre serveur HTTP.

Comme indiqué au lancement du projet, la partie gestion des sockets réseau est fournie dans une bibliothèque externe, vous ne devrez vous concentrer que sur la gestion du protocole HTTP.

Dans un premier temps votre serveur devra gérer les requêtes POST et surtout GET afin de renvoyer des **pages présentes localement sur le serveur** (fichier locaux, de type html, css, javascript, gif, jpg, png, etc...)

2 Détail du travail demandé

Votre programme devra réaliser plusieurs taches :

1 - Réception des requêtes HTTP venant du réseau. (Cf. 3 – gestion des requêtes HTT)

2 - Vérification syntaxique, les champs seront trouvés grâce à votre parseur réalisé en partie2 .

3 - Vérification sémantique de la requête (Est-ce une requête valide au sens de la RFC) ?

4 – Traitement de la requête. (Cf. 6 – Détails des règles sémantiques)

5 – Génération d'une réponse et envoi de la réponse et gestion de la connexion.

Vous devrez lister dans un document les règles sémantiques que vous mettrez en œuvre dans votre serveur

Par exemple: Si version HTTP vaut 1.1 et champ Host manquant → réponse 400 (bad request)

La tâche 5 est sans doute la plus complexe !!!!

3 Gestion des requêtes HTTP

Votre serveur utilisera une bibliothèque gérant les requêtes des différents clients, cette bibliothèque n'effectue aucun filtrage, ne fait que tamponner les segments TCP reçus pour « remonter » à votre serveur une requête la plus complète possible. En effet la gestion des flux TCP et leur programmation est assez complexe et nécessite des connaissances que vous n'avez pas encore acquises. Cette bibliothèque fait le travail à votre place sur les aspects réseau..

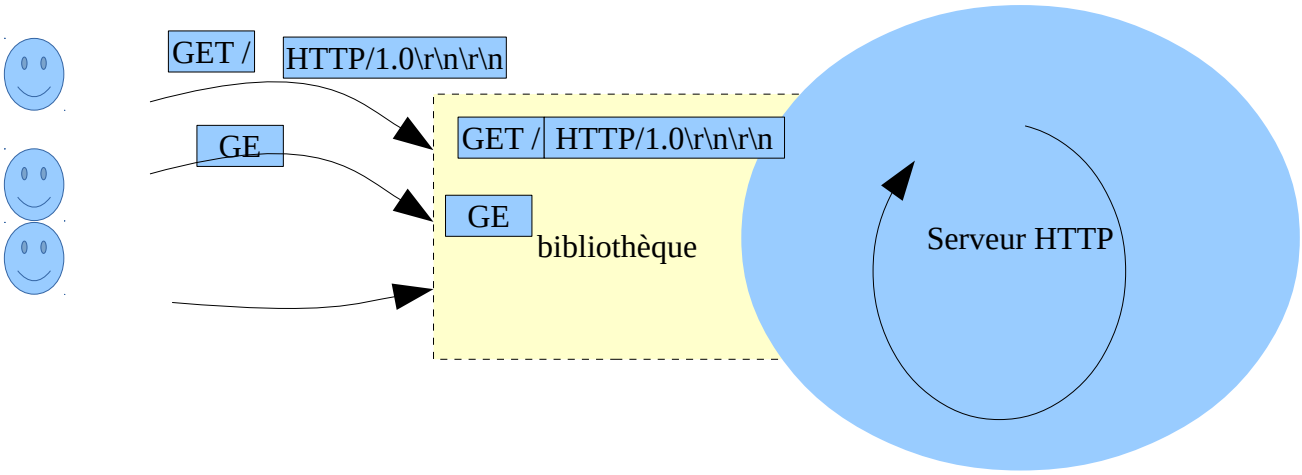
Pour plus de performance la bibliothèque est multi-connexions (vous remarquerez qu'un navigateur ouvre souvent plusieurs connexions vers le même serveur HTTP), et remonte parmi tous ses clients/connexions la première requête considérée comme «complète», puis la deuxième, etc...Donc votre programme recevra séquentiellement les requêtes.

Notion de complétude de la requête :

Savoir si une requête est complète ou non n'est pas chose si simple que cela, la librequest possède donc un parseur simple qui lui permet de savoir si la requête est complète ou non. Dans le cas ou librequest ne reconnaît pas une requête, un timeout est associé à la requête: s'il n'y a pas de nouvelles données pendant 5 secondes, la librequest remonte cette requête non valide à votre serveur (un warning est indiqué sur la sortie standad du programme).

Bien sûr cette bibliothèque ne fait pas tout le travail à votre place, elle vous remontera généralement ce qu'elle pense être une requête complète, mais elle peut aussi remonter une requête « non valide » si c'est ce que le client envoie réellement. Vous devrez donc être attentif à la validation syntaxique et sémantique de la requête reçue….

Exemple simplifié de fonctionnement pour illustrer les problèmes de gestion des flux….
On considère que chaque case représente un segment TCP ….



Ici l'appel à getRequest retournera la seule requête que la bibliothèque considère comme complète :

GET / HTTP/1.0\r\n\r\n

les autres clients doivent envoyer la fin de leur requête, qui seront remontées une fois « complète » à votre serveur au prochain appel de getRequest..

4 Analyse syntaxique

L'analyse syntaxique sera réalisée avec le travail effectué en etape 2, la stabilité de votre parseur est essentiel pour pouvoir aborder l'étape3. Si l'avis donné par l'enseignant est négatif su rvotre parseur, vous êtes fortement incité à prendre le parseur « enseignant » (libparser). Un retard sur l'étape 2 ne peut être une justification à un retard sur l'étape 3.

5 Détail des règles sémantiques.

Vous pouvez notamment vous appuyer sur les éléments suivants :

- Vous devrez notamment traiter les cas suivants selon le champ « method » :

  [RFC 7231 4.3.1] GET - A payload within a GET request message has no defined semantics; sending a payload body on a GET request might cause some existing implementations to reject the request.

  [RFC 7231 4.3.2] HEAD - A payload within a HEAD request message has no defined semantics sending a payload body on a HEAD request might cause some existing implementations to reject the request.

  [RFC 7231 4.3.3] POST - The POST method requests that the target resource process the representation enclosed in the request according to the resource's own specific semantics.
  Le traitement de la requête POST sera traitée en partie 4 du projet.

- pour le champ « Request-target » :

  [RFC 7230 2.7.3] Normalisation de la request-taget conformement à la rfc3986, notamment le « percent encoding » et le « dot segment removal ».

- pour le champ HTTP-version :

  [RFC 7230 2.7.3]

- pour les champs d'entête :

  Les entêtes possèdent des informations qu'il faut utiliser pour comprendre la requête, la traiter et eventuellement construire la réponse, c'est le cas des entêtes suivants :

  Transfer-Encoding-header [rfc 7230 – 4]
  Cookie-header / [rfc6265]
  Referer-header [rfc7231 – 5.5.2]
  User-Agent-header [rfc7231 -5.5.3]
  Accept-header [rfc7231 – 5.3.2]
  Accept-Encoding-header [rfc 7231 – 5.3.4]

  Une attention particulière sera portée sur les entêtes suivants :

  Content-Length-header [rfc7230 – 3.3.2 et 3.3.3]
  Host-header [rfc7230 – 5.3 et 5.4]
  Connection-header [rfc7230 – 6]

6 Utilisation de la bibliothèque.

La librequest.so est donnée sous forme compilée exclusivement, et fonctionne sur les stations des salles projet.

Ainsi votre serveur est une boucle traitant les requêtes séquentiellement qui lui arrivent via la bibliothèque, ces requêtes sont récupérées par votre serveur qui devra effectue en boucle : (Cf doc API)

- Appel de la méthode ***getRequest*** (Cf Doc API) // cette méthode est bloquante.
- Vérification et traitement de la requête
- Décision de renvoi d'une réponse via ***sendReponse*** ou **writeDirectClient et endWriteDirectClient** (Cf Doc API) // ces méthodes ne sont pas bloquantes.
- Décision de clôture ou non de la connexion TCP via ***requestShutdownSocket*** (Cf Doc API) // ces méthodes ne sont pas bloquantes.
- Libération de la mémoire associée à la requête via ***freeRequest*** (Cf Doc API)

[Doc API] refman.pdf ou site web du projet sur chamilo. La lecture de la doc API est nécessaire.

Votre programme doit inclure le fichier d'entête « request.h »

Un exemple est donné sur chamilo (écoute sur le port 8080)

L'option de compilation doit inclure -L. -lrequest si la bibliothèque « librequest.so » est dans « . » et request.h est dans ./api

gcc -g -I./api -c main.c
gcc -o sock main.o -L. -lrequest

Pour exécuter le programme vous serez peut être amené à effectuer la commande **export LD_LIBRARY_PATH=.** avant de pouvoir lancer le programme.

7 Exemples

Vous trouverez sur chamilo 2 exemples :

- un premier exemple qui indique comment utiliser la librequest.so

- un deuxième exemple qui réalise un serveur HTTP qui reçoit des requêtes par librequest.so, en vérifie la syntaxe vial le parseur enseignant (libparser.so), si celle-ci n'est pas correcte, renvoie un « 400 bad request », sinon récupère du parseur le champ HTTP-message et la renvoie en réponse au client dans le contenu d'une réponse »200 OK » . (yes this is a ping pong HTTP server)

8 Tests

Vous développerez quelques pages web, incluant des images et des scripts javascript, des fichier css. La démonstration de la fin d'étape 3 se fera avec votre site. L'évaluation pourra se faire avec un autre site (pages web)