# Applications TCP et UDP
## HTTP Link Checker et Dice Roll
Sys 1 : Programmation réseau (et système)

L'objectif de ce premier projet est d'écrire deux applications interagissant avec le réseau. D'une part, un outil permettant de trouver les liens morts sur un site web (Partie 1 et 2) ; et d'autre part, un serveur de jets de dés, qui permet à un client d'envoyer un nombre de face (pas nécessairement réaliste), et d'obtenir le résultat d'un jet d'un tel dé.

**Objectifs pédagogiques:**

- Faire communiquer des programmes par des sockets réseau;
- Architecturer puis écrire un programme lisible de quelques centaines de lignes;
- Implémenter une partie d'un protocole textuel complexe (HTTP coté client) basé sur TCP.
- Implémenter complètement un protocole réseau binaire simple (Dice Roll coté client et serveur), basé sur UDP.
- Observer les différences entre TCP et UDP, et leurs intérêts.

## ★ Première partie : Vérifier les liens d'une page web (12pts)

Pour cette première étape, il vous faut écrire un programme qui télécharge une page web passée en paramètre, cherche les liens HTML qu'elle contient et vérifie qu'ils pointent vers des pages existantes.

■ **Première étape :** télécharger une page web d'après son URL

Dans l'URL `http://people.irisa.fr/Martin.Quinson/`, on distingue trois parties.

- `http://` indique le protocole à utiliser sur le serveur.
- `people.irisa.fr` est le nom de domaine, c'est-à-dire le nom de la machine à contacter pour récupérer la page en question
- `/Martin.Quinson/` désigne la page sur ce serveur à récupérer.

En pratique, il faut ouvrir une connexion cliente TCP sur le port 80 de la machine `people.irisa.fr` (utilisez le DNS pour trouver l'IP correspondante), puis envoyer le contenu suivant.

```
GET /Martin.Quinson/ HTTP/1.0\r\nHost: people.irisa.fr\r\n\r\n
```

Il est important de spécifier le nom de domaine depuis lequel on veut récupérer le document, et de bien terminer chaque ligne par `\r\n` (au lieu de seulement `\n`), sous peine de recevoir une erreur "Malformed request" du serveur. Si la requête est bien formée, le serveur renvoie un document débutant ainsi:

```
HTTP/1.1 200 OK
Date: Fri, 12 Nov 2021 14:47:35 GMT
Server: Apache
Last-Modified: Sat, 09 Oct 2021 16:32:48 GMT
ETag: "1673-5cdee0bd6d1df"
Accept-Ranges: bytes
Content-Length: 5747
Vary: Accept-Encoding
X-Content-Type-Options: nosniff
X-Frame-Options: sameorigin
Connection: close
Content-Type: text/html

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
...
```

À la première ligne, on voit que la réponse est de type `200 OK`, c'est-à-dire une réussite. Nous aurions pu avoir par exemple une réponse `404 Not Found` à la place, indiquant que le document demandé n'existe pas. L'objectif général du projet est donc de tester pour chaque lien de la page si cela correspond à une ressource existante sur le serveur correspondant (code de retour 200), ou non (code de retour différent de 200, le plus souvent 404). Mais nous n'en sommes pas encore là.

Si le code de retour est 200, la page demandée est envoyée après un ensemble de lignes d'entête suivi d'une ligne vide (ici, le document débute donc à la ligne 13). Sur la ligne 7, on trouve la ligne d'entête `Content-Length`, qui indique que le document téléchargé contient exactement 5747 octets.

▷ **Question 1:** Faites un programme téléchargeant une page web, puis affichant tout son contenu. Les entêtes HTTP ne doivent pas être affichés à l'écran.

■ **Seconde étape :** extraire les liens de la page

Le format HTML est assez verbeux, mais très simple au final. Les liens sont de la forme suivante.

```
<a href="http://people.irisa.fr/Martin.Quinson/blog">Mon blog</a>
```

On distingue le texte à afficher (placé entre le tag ouvrant `<a ...>` et le tag fermant `</a>`, càd "Mon blog") du lien vers lequel pointer (la valeur de l'attribut `href`, `http://people.irisa.fr/Martin.Quinson/blog`).

▷ **Question 2:** Limitez maintenant l'affichage à la seule liste des liens contenus dans la page.

■ **Troisième étape :** Robustesse de l'analyse

HTML est un formalisme extrêmement permissif. Les lettres des tags peuvent être en majuscule, et qu'on peut avoir n'importe quel caractère blanc entre le `a` du tag et le `href`. Les trois liens suivants sont donc valides:

```
<A hReF="http://people.irisa.fr/Martin.Quinson/blog">Mon blog</a>
<a      href="http://people.irisa.fr/Martin.Quinson/blog">Mon blog</a>
            <a
href="http://people.irisa.fr/Martin.Quinson/blog">Mon blog</a>
```

▷ **Question 3:** Modifiez votre code pour accepter les différentes formes d'un lien.

■ **Étape intermédiaire :** nettoyage

Retirez le code mort, donnez les meilleurs noms possibles à vos fonctions et variables, faites maintenant les petits nettoyages à faire "plus tard", et commitez cette version fonctionnelle dans votre git avant de continuer.

Un bon code n'a pas besoin de beaucoup de commentaires. N'écrivez pas en commentaire à quoi sert une fonction ou une variable, donnez des noms explicites pour rendre tout commentaire redondant. Si vous ne trouvez pas de nom explicite, c'est probablement que vos fonctions sont mal découpées. N'indiquez pas non plus en commentaire comment ça marche: le code est là pour ça, et il doit se suffire à lui-même, sans paraphrase. En revanche, indiquez pourquoi ça marche: conditions sur les paramètres à respecter, invariant de boucle, etc. Ne commentez pas un code mal pensé. Réécrivez-le.

■ **Quatrième étape :** vérification des liens d'une page

Pour chaque lien trouvé sur la page web passée en paramètre, il faut chercher à la télécharger depuis son serveur afin de vérifier son existance. Les liens résultants en un code 200 seront ignorés, mais il faut faire un message d'erreur adapté pour erreur rencontrée.

▷ **Question 4:** Implémentez la détection de lien cassés et affichez les erreurs.

■ **Dernière étape :** nettoyage

Une fois que cela fonctionne, refaites une passe de nettoyage sur votre code. Cela simplifiera votre travail par la suite. Prenez le temps de réorganiser votre code pour le rendre le plus lisible possible. Par exemple, il faut absolument éviter de dupliquer trop de code entre le téléchargement de la page initiale et celui des différents liens à vérifier ensuite.

Il serait raisonnable d'écrire des tests, par exemple pour la fonction d'analyse du texte, pour vous assurer que vous pouvez utiliser votre code sans crainte quand vous écrivez les autres modules. Vous pouvez utiliser `https://framagit.org/mquinson/simple-unit-testing/` pour cela.

Ces nettoyages reviennent à vous assurer que le code déjà écrit constituera une fondation solide pour le code à venir. Ne pas les faire est un pari audacieux et assez risqué.

## ★ (Optionnelle) Deuxième partie : vérification d'un site web (3 pts bonus)

Il s'agit maintenant de faire la recherche récursive de toutes les pages d'un domaine. Pour cela, on traitera différement les liens trouvés en fonction qu'il s'agit de liens internes (pointant vers l'une des pages du même site), ou des liens externes. En pratique, un lien interne se trouve sur le même serveur (ici, `people.irisa.fr`).

On peut aussi décider que ne vérifier que les liens présentant un préfixe donné. Par exemple ici, on cherchera les liens morts de la page `http://people.irisa.fr/Martin.Quinson/blog` car la page initiale est un préfixe de cette URL. Ce comportement peut être utile, en particulier dans mon cas, car je ne veux pas connaître les liens morts des pages de mes collègues, même si on trouve un lien de ma page vers la leur.

▷ **Question 5:** Écrivez le programme vérifiant un site web.

Libre à vous d'organiser votre code comme vous le souhaitez. Une façon serait de maintenir un grand buffer du texte à analyser, de l'analyser peu à peu, et de vérifier tous les liens qu'on y trouve. Quand on vérifie un lien interne, on ajoute le contenu de la page pointée à la fin du buffer. On peut aussi analyser le texte dès qu'on le reçoit au lieu de le stocker dans un buffer. Une autre façon est de d'abord construire un vecteur des liens trouvés dans une page avant de les vérifier les uns après les autres. Réfléchissez avant d'écrire votre code, et cherchez la solution vous permettant d'écrire le code le plus simple possible.

Si vous avez réussi à identifier différents modules, il devient vite fastidieux de recompiler votre projet à la main après chaque modification. Écrivez un `CMakeLists.txt` en vous inspirant des TP précédents.

## ★ Nettoyage final

Pensez à documenter un peu votre projet avant de passer à la suite. Rédigez un README présentant rapidement le projet. Après une introduction de quelques lignes, notez comment compiler et exécuter votre projet. Vous donnerez également la vision d'ensemble de votre programme en listant les différents modules et fonctions, ainsi que les grandes lignes de leurs interactions. S'il reste des problèmes dans votre code, indiquez-le.

## ★ Troisième partie : Dice Roll (8 points + bonus)

L'objectif de ce mini-projet est d'écrire un serveur et un client communiquant en UDP, relativement simple. On propose d'utiliser des ports UDP 3549 (`0xddd`) et suivants pour les serveurs de dés, mais ces numéros de ports ne sont qu'une convention entre nous, et ne font pas partie de la spécification du protocole.

Une requête de jet de dés se compose de deux octets :

Le nombre magique `0xde`, suivi du nombre de face, sur un octet.

Lorsqu'un serveur reçoit un datagramme comportant ce format, il génère un nombre aléatoire entre 1 et le nombre de faces, et renvoie celui-ci.

La réponse est alors de la forme octet magique `d0` suivi du résultat (sur un octet). En cas de requête mal formée, votre serveur peut renvoyer `df` suivi du nombre `0`, mais votre serveur peut aussi ignorer les requêtes manifestement mal formées.

`0` n'étant pas un nombre de face ou un résultat valide, celui-ci sera interprété comme le premier nombre de face non-représentable sur un octet.

```
Requête :

0      1      2 octets
+------+------+
| 0xde |  Nf  |
+------+------+

Nf est un nombre 8-bit non-signé, où 0 est interprété de façon particulière.

Réponse :

- Succès :

0      1      2 octets
+------+------+
| 0xd0 |   r  |
+------+------+

r est compris entre 1 et Nf inclus, distribution uniforme.

- Erreur :
```

```
0      1      2 octets
+------+------+
| 0xdf |   0  |
+------+------+
```

■ **Première étape :** Requête

Les fonctions auxiliaires de CS:APP ne fonctionnent que pour TCP, il va donc falloir implémenter l'ouverture des sockets UDP nécessaire à la main.

▷ **Question 6:** Écrivez la gestion d'argument pour le client et le serveur. De quels paramètres le serveur et le client ont besoin ?

▷ **Question 7:** Ajoutez au serveur l'ouverture d'une socket pour réceptionner les paquets UDP sur un port donné, ajoutez au client le code nécessaire pour envoyer une requête au serveur, dans le serveur émettez un message sur la console à chaque data-gramme reçu. Choisissez si votre client permet de faire plusieurs requêtes en mode interactif ou s'il effectue une seule requête avant de quitter.

■ **Deuxième étape :** Réponse

▷ **Question 8:** Quel est le nombre de face maximum supporté ? Implémentez une fonction générant un nombre aléatoire en 1 et le nombre de faces demandées. On considèrera que rand() de stdlib.h présente est uniforme de qualité suffisante entre `0` et `RAND_MAX`, mais votre fonction doit générer une distribution uniforme sur l'intervalle souhaité. Il est possible de faire plusieurs appels à rand pour ce faire.

▷ **Question 9:** Implémentez la génération du nombre réponse à la requête, implémentez le code d'envoi de la réponse au client.

▷ **Question 10:** Ajoutez au client le code nécessaire pour réceptionner le data-gramme UDP de réponse en provenance du serveur.

■ **Troisième étape :** Gestion d'erreur

▷ **Question 11:** Que se passe-t-il se un paquet est perdu. Implémentez la gestion d'erreur coté client pour gérer la parte de paquet. Vous pouvez ajouter au serveur une probabilité de perdre la requête pour effectuer des tests. (Celle-ci peut être passé en argument optionnel au lancement du serveur ou configuré à la compilation). On considère que la perte de paquet équivaut à un dé cassé.

▷ **Question 12:** Ajoutez un gestion d'erreur pour les data-grammes mal-formés coté client et serveur.

■ **Quatrième étape :** Nettoyage

De la même façon que précédemment, prenez le temps de nettoyer votre code.

■ **Optionnel :** Critique du protocole (2 pts bonus)

▷ **Question 13:** Ce protocole est très simpliste, pouvez-vous identifier des limitations ou problèmes de celui-ci.

▷ **Question 14:** Proposez une spécification d'une meilleur version du protocole, vous pourriez utilise 0x0d ou 0xdd comme nombre magiques d'un octet si besoin.

Ces réponses permettront une discussion en classe autour des protocoles.