Chapter 1. Lua et le réseau

Table of Contents

1.1. Utiliser Lua pour des applications réseaux

Etant donné sa nature extensible, Lua dispose de sa panoplie de modules permettant de s'interfacer avec le réseau au sens large. Cependant, selon l'utilisation du réseau que vous souhaitez réaliser, Lua n'est pas forcément le meilleur choix. Ainsi, si vous souhaitez simplement apporter des capacités de contrôle à distance à votre applicaiton ou à échanger des données avec une application tierce, Lua peut effectivement convenir et vous permettre des réalisations rapides et fiables. Si par contre vous cherchez à développer un gros service réseau, proposant par exemple des API *SOAP* ou *JSON* et destinés à des milliers d'utilisateurs, alors Lua n'est probablement pas le bon choix. Non pas que Lua ne serait pas capable de tenir la charge - bien au contraire -, mais plutôt car l'écosystème *web* de Lua est assez pauvre comparé à des langages plus spécialisés comme PHP, Perl ou Ruby.

Le module phare de Lua dans le domaine réseau est *LuaSocket*. C'est le plus connu de la communauté et le plus souvent utilisé et cité dans la mailing-list et les sites. Il est d'ailleurs utilisé dans le manuel d'apprentissage associé au présent manuel.

1.2. `LuaSocket`

`LuaSocket` est à l'heure actuelle la librairie réseau la plus connue et la plus utilisée par la communauté Lua. Elle permet d'adresser directement les protocoles *TCP* et *UDP*, ainsi

que des facilités pour réaliser des clients *SMTP*, *HTTP*, *DNS* et *FTP*. Lorsque c'est possible, la librairie reprend directement les noms de fonctions de l'API C mais y ajoute des abstractions qui en simplifie l'utilisation. La lecture d'un stream peut par exemple se faire ligne à ligne, par blocs de données ou bien encore jusqu'à ce que la connexion soit fermée. L'ensemble des fonctions manipulant des adresses peuvent les recevoir soit sous forme IP, soit sous forme de noms, le module se chargeant d'effectuer les résolutions nécessaires.

En compléments des protocoles eux-mêmes, la librairie intègre le traitement des *URL* et des codages *MIME*. Enfin, dans le but de créer facilement des chaînes de traitement des données, le module *LTN12* permet de créer des chaînes de filtres, chaque chaîne ayant une source et une destination.

1.3. Un serveur TCP/IP avec `LuaSocket`

L'exemple ci-dessous (issu du manuel d'apprentissage) montre en quelques lignes comment réaliser un serveur web minimaliste. Si vous faites pointer votre navigateur vers l'adresse `http://localhost:8080`, vous recevrez en retour le message du script.

```
-- Chargement du module socket de LuaSocket
local socket = require 'socket'
-- Création server TCP, sur la machine locale et sur le port 8080
local server = socket.bind("localhost", 8080)
-- Attente d'une connexion et renvoi du client connecté
local client = server:accept()
-- Reception des données du client
local line = client:receive()
-- Et renvoi d'une réponse
client:send("Serveur Lua:\r\nBonjour !\r\n");
-- Fermeture de la connexion
client:close()
```

1.4. Un client TCP/IP avec `LuaSocket`

On peut tout aussi très simplement créer une connection et échanger des données avec un serveur.

```
-- Chargement du module socket de LuaSocket
local socket = require 'socket'
-- Connection à yahoo.fr, sur le port HTTP 80
local mysock = socket.connect("www.yahoo.fr",80)
-- Envoi d'une requête HTTP
local page, code = mysock:send([[GET /index.html HTTP/1.1
```

```
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 6.1; en-US;
 rv:1.9.1.5) Gecko/20091102 Firefox/3.5.5 (.NET CLR 3.5.30729)
]] )
-- Récupération de la réponse (une erreur dans ce cas, la requête
 HTTP étant trop simple)
local page, code = mysock:receive('*a')
print(page, code)
```

1.5. Envoyer et recevoir des données UDP avec LuaSocket

Le module ne serait pas complet sans la possibilité d'utiliser UDP. Les exemples de code ci-dessous montrent l'échange de données via UDP sur une même machine.

```
-- Chargement du module socket de LuaSocket
local socket = require 'socket'
--
print"LISTENING"
-- Creation d'une socket UDP
local res = socket.udp()
-- On attache la socket à l'adresse locale, port 8700
res:setsockname("127.0.0.1", 8700)
-- On attend des données
local data = res:receive()
-- On affiche
print(data)
-- On libère les ressources
res:close()
```

Le code d'envoi des données:

```
-- Chargement du module socket de LuaSocket
local socket = require 'socket'
-- Le datagramme UDP à envoyer
local data = "COUCOU"
-- On utilise ici directement la méthode sendto, qui ne prend que
 des adresses IP
socket.udp():sendto( data, "127.0.0.1", 8700)
```

L'envoi des données peut aussi se faire avec la méthode `send` en convertissant la `socket` en mode connectée:

```
-- Chargement du module socket de LuaSocket
local socket = require 'socket'
-- Le datagramme UDP à envoyer
```

```
local data = "COUCOU"
local sender = socket.udp()
-- On peut ici spécifier un nom de machine
sender:setpeername("localhost", 8700)
-- On utilise ici directement la méthode sendto
sender:send( data)
```

Par défaut, les appels à `receive` et `receivefrom` sont bloquants, mais on peut utiliser la méthode `settimeout(value_in_seconds)` pour les convertir en non-bloquants.

1.6. Effectuer des recherches DNS avec LuaSocket

Le module `socket` intègre le sous-module `dns` permettant d'éffectuer des recherches DNS et des conversions de noms vers des adresses IP et vice-versa.

- `name = socket.dns.gethostname()` : retourne le nom de la machine hôte;
- `name, resolvetab = socket.dns.tohostname(address)` : retourne le nom de la machine correspondant à `address` ainsi qu'une table contenant les champs `name`, `alias` et `ip`;
- `ipaddr, resolvetab = socket.dns.toip(address)` : retourne l'adresse IP correspondant à `address` ainsi qu'une table contenant les champs `name`, `alias` et `ip`.

```
local socket = require 'socket'
-- Affichage du nom du host courant
print("HOST :", socket.dns.gethostname() )
```

1.7. Fonctions FTP de LuaSocket

Les fonctions FTP de `LuaSocket` permettent d'effectuer les opérations d'un client FTP en Lua. Les deux fonctions sont `ftp.get` et `ftp.put`. Comme pour le module `http`, les fonctions acceptent deux modes d'invocation : un mode simplifié et un mode complet.

Le mode simplifié permet de télécharger ou d'envoyer le contenu d'un fichier au serveur FTP, en donnant l'adresse comme argument.

```
local ftp = require 'socket.ftp'
-- Récupère le contenu d'un README
local myreadme = ftp.get("ftp://download.nvidia.com/opensuse/
README")
-- Upload du contenu sur un FTP publique
ftp.put("ftp://ftp.mypubftp.com/README", myreadme)
```

Le mode complet permet de contrôler l'ensemble des arguments FTP, tels que mot de passe, commande et port. Dans ce mode, les méthodes sont invoquées avec une table comme argument.

1.8. Le module `ltn12`

TODO

1.9. Récupérer le contenu d'une page web avec `LuaSocket`

Le module `http` de `LuaSocket` offre la méthode `request` qui permet de récupérer une URL via le protocole HTTP. La méthode supporte deux types d'invocation, l'une simplifiée et l'autre complète. L'invocation simple s'effectue sous la forme `http.request( url [,body])`. La fonction récupère la page pointée par l'adresse `url` avec la méthode `GET`. Si `body` est renseigné, alors c'est la méthode `POST` qui est utilisée. Dans tous les cas, la fonction retourne quatre valeurs : la page, le code de retour, les entêtes de la réponse et la ligne de statut, comme vu dans l'exemple qui suit.

```
local http = require"socket.http"
local p, c, h, l = http.request("http://luajit.org/
ext_ffi_api.html")
-- p contient la page récupérée
print("HTTP code : ", c)
print("Returned headers", h)
print("Ligne de statut", l)
```

Affiche:

```
HTTP code :  200
Returned headers  table: 0x419237b0
Ligne de statut HTTP/1.1 200 OK
```

La méthode complète permet d'effectuer des opérations plus complexes sur les URL. Son invocation se fait en passant comme argument une table contenant les entrées suivantes:

- `url` : l'adresse de la page à récupérer. C'est le seul paramètre obligatoire;
- `sink` : la fonction de récupération des données selon le mécanisme du module `LTN12`;
- `method` : chaîne de caractère contenant la méthode à utiliser `GET`, `POST` ou `HEAD`;

- `headers` : table contenant des entêtes HTTP complémentaires;
- `source` : une fonction destinée à fournir la section `BODY` de la requête HTTP;
- `step` : une fonction d'itération, comme utilisée par le module `LTN12`;
- `proxy` : chaîne contenant l'URL du proxy;
- `redirect` : booléen, à mettre à `false` pour ignorer les directives HTTP de redirection;
- `create` : une fonction optionnelle à utiliser au lieu de `socket.tcp`.

Dans le cas de l'utilisation de la méthode complète, les valeurs de retour sont identiques à la version simplifiée, mis à part que la première valeur vaut `1`, puisque le contenu de l'adresse est envoyé à la fonction `sink`.

```
local http = require"socket.http"
local ltn12 = require"ltn12"
local p, c, h, l = http.request{url = "http://luajit.org/
ext_ffi_api.html",
                                sink =
 ltn12.sink.file(io.open("ext_ffi_api.html", "w"))
                               }
print("SAVED : ", p, c)
```

1.10. Autres fonctions de `LuaSocket`

La librairie `LuaSocket` permet aussi :

- de formater et de vérifier des URL avec `socket.url`
- manipuler les formats MIME et l'encodage `b64` associé, avec `socket.mime`