HTTP

Sur 2 séances

Objectifs

Pour le côté théorique :

- description générale du protocole
- notions d’URL et de ressource
- format des messages et des requêtes
- en-têtes
- code des réponses

Pour le côté pratique :

- écriture d’un serveur HTTP minimal pour débuggage
- implémentation d’un gestionnaire de session
- gestion de cookie

HTTP : description générale

- HyperText Transfer Protocol
- créé au CERN en 1990 pour la documentation scientifique
- Protocole Client/serveur, basé sur TCP
- Client habituel : navigateur web
- Désignation de ressources sur le serveur : URL
- normalisation : 1996 HTTP/1.0, RFC 1945, 1997,1999 HTTP/1.1 RFC 2068, RFC 2616

Ressource et URL

Ressource

- Ressource : entité hébergée par le serveur
- le client interagit sur des représentations des ressources
- ce qui constitue la ressource effectivement : fichier (web statique), résultat de l'exécution d'un programme (web dynamique)

URL

- Uniform Resource Locator
- cas particulier d'URI (Identifier)

```
type_connexion://[login:pass@]serveur[:port]/path/to/resource[#id_fragment][?liste_param]
```

Exemples :

```
http://gamba.perso.enseeiht.fr/appliweb
```

```
http://gamba.perso.enseeiht.fr/doc?param1=val1&p2=val,val2
```

Messages du protocole

Requêtes : du client au serveur

- ligne de commande : `type_requête ressource_path HTTP/version`
- en-têtes optinnels, `nom: valeur`, un par ligne
- une ligne blanche, obligatoire
- un corps, optionnel

Réponse : du serveur au client

- ligne de statut : `HTTP/version code-réponse phrase réponse`
- en-têtes optionnels, format identique à une requête
- ligne blanche obligatoire
- un corps, optionnel

Type de requête

- GET : demande la récupération de la représentation d’une ressource
- HEAD : idem GET, mais sans le corps éventuel
- PUT : fournit une répresentation pour une ressource, à gérer par le serveur
- DELETE : demande d’effacer la ressource spécifiée
- PATCH : application de modification partielle à une ressource
- POST : soumission de données, pouvant créer une nouvelle ressource ou modifier une ressource existante

En-têtes

Spécifique requête

- `Host` : indique le nom DNS du serveur, obligatoire en HTTP/1.1
- `Connection` : persistence de la connexion TCP (`Keep-Alive` ou non)

Spécification du corps d'un message

- `Content-Length` : longueur en octets
- `Content-Type` : type MIME de la ressource

Négociation de contenu

- `Accept` : liste des types MIME acceptés par le client
- `Accept-Charset`, `Accept-Language`

Le serveur utilise ces entêtes pour choisir la représentation adéquate de la ressource.

Code de retour de la réponse

- 1xx Information
- 2xx succès
- 3xx redirection
- 4xx : erreur dans la requête
- 5xx : erreur serveur

Les codes à connaître :

- 200 ok : code standard pour indiquer que l'opération demandée s'est bien déroulée, typiquement pour un GET
- 201 Created : une ressource a été crée (réponse à un PUT ou un POST)
- 303 See other : redirection vers une autre ressource
- 304 Not Modified
- 400 Bad request : requête impossible à analyser (erreur de syntaxe)
- 401 Unauthorized
- 403 Forbidden
- 404 Not Found
- 500 Erreur interne du serveur

Exemple de dialogue :

Requête

```
<pre 'code'>
GET /index.html HTTP/1.1
Host: www.example.com

```

Réponse

```
<pre 'code'>
200 HTTP/1.1 ok
Content-Length: 4
Content-Type: text/plain

toto
```

Tester ce genre de requête en utilisant telnet sur un site connu :

```
telnet google.com 80
```

TP : écriture d'un serveur minimal

À partir du code de base, compléter pour analyser le contenu d'une requête HTTP. Le serveur doit retourner le contenu de la requête comme réponse.

Pour tester votre serveur, vous utiliserez :

- firefox
- curl

```
curl -X GET
http://localhost:8080/toto
```

```
curl -X POST -d nom=valeur
http://localhost:8080/toto
```

Faites pour le moment du code procédural (pas de

```
<pre 'code'>
```

```ruby
require 'socket'

port = 8080
server = TCPServer.open(port)
process_number = 1
trap('EXIT'){ server.close }
process_number.times do
  fork do
    trap('INT'){ exit }
    loop do
      socket = server.accept
      # socket.gets : lit une ligne entière
      # sockets.read(n) : lit n octets
      # socket.write string : écrit
      # socket.puts string : idem plus retour ligne
      #
      #request parsing
      #response code
      #response headers
      #response body
    end
    exit
  end
end

trap('INT') { puts "\nexiting" ; exit }
```

classes)

```
# on attend la fin des fils
Process.waitall
```

tcp_server.rb

Cookies

- cookie : suite d'information envoyée par un serveur HTTP à un client, et stocké sur le client
- positionnement par le serveur avec l'en-tête `Set-Cookie: name=value;attr=val;attr=val…`
- rappel de la valeur par le client avec l'en-tête `Cookie: name=value`
- attributs à la mise en place: date d'expiration, chemin, domaine, chiffré ou non

```
Set-Cookie: nom=nouvelle_valeur;
expires=date; path=/;
domain=.exemple.org
```

- persistant : date d'expiration donnée et dans le futur, sinon cookie de session (persiste jusqu'à l'arrêt du navigateur)

Pour une référence complète, voir les RFC2109 et RFC2965

- supprimer un cookie : positionner une date d'expiration dans le passé
- les attributs `Path` et `Domain` déterminent le chemin et les sites pour lesquels le cookie sera renvoyé.

Les dates d'expiration sont données sous le format `Wdy, DD-Mon-YYYY HH:MM:SS GMT`, par exemple `Mon, 15-Mar-2042 22:42:12`

Gestion de session

- au niveau serveur, une session sert à repérer les connexions émises par l'instance particulière d'un navigateur
- non géré au niveau HTTP

Principe

1. on crée un identifiant au début de la session au niveau serveur
2. le serveur donne l'identifiant au client
3. le client rappelle l'identifiant dans les requêtes suivants

L'identifiant sert à indexer des ressources au niveau du serveur.

Gestion de l'identifiant

- cookie
- paramètre de la requête, soit dans l'URL, soit dans le corps

Sécurité ...

- session hijacking

- HTTPS assure le chiffrement des cookies, donc à ne pas réserver aux quelques pages sensibles.

Rack : une couche d'abstraction

`gem install rack`

rack est une librairie ruby qui se greffe sur un serveur HTTP et fournit une interface haut niveau :

Une application rack est un objet ruby répondant à la méthode `call(env)`.

env représente l'environnement correspondant à l'analyse de la requête HTTP reçue. C'est un Hash dont les élément correspondent au résultat de l'analyse d'une requête HTTP.

La méthode `call` retourne un triplet `[status, headers, body]` qui servira à générer la réponse HTTP :

- `status` le code d'erreur HTTP
- `headers` : doit répondre à `each{ |key,value| … }`, typiquement un Hash.
- `body` : doit répondre à `each`, chaque élément devant être de type String, typiquement un tableau de String.

```
<pre 'code'>
class HelloWorld
  def call(env)
    [200, {"Content-Type" => "text/plain"}, ["Hello world!"]]
  end
```

```
end
```

Utiliser Rack

On utilise généralement les classes Rack::Request et Rack::Response

```
<pre 'code'>
require 'rack/request'
require 'rack/response'

class RackAppTemplate
  def call(env)
    req = Request.new(env)
    …
    res = Response.new
    res.write …
    res.finish # génère le triplet résultat
  end
end
```

Lancement d’une application Rack

```
<pre 'code'>

# app.ru
require '…'
run ObjectRespondingToCall.new
```

Utiliser rackup :

```
<pre 'code'>
rackup app.ru
```

L’application s’exécute alors avec le serveur web `webrick`. Pour l’exécuter au dessus d’un autre serveur, il faut suivre une procédure spécifique à chaque serveur.

Pour Apache et Nginx, il faut par exemple utiliser le module Passenger.

Middleware Rack

2 applications Rack sont facilement combinables:

<pre 'code'>

```ruby
class Middleware
  def initialize(app, …)
    @app = app
  end

  def call(env)
    # do stuff on env
    status, headers, body = @app.call(env)
    # do stuff on status, headers, body
    [status, headers, body]
  end
end
```

<pre 'code'>

```ruby

# app.ru
require '…'
use Middleware # , arg1, arg2 : argument passed to Middleware#initialize
run ObjectRespondingToCall.new
```

En sous-main, rackup utilise Rack::Builder