Préparation à l'agrégation d'Informatique

Web 2 : le protocole HTTP

kn@lri.fr
http://www.lri.fr/~kn

université
PARIS-SACLAY

Plan

Modèle client/serveur

Architecture logicielle pour un programme *distribué*

- Un composant unique fourni l'accès à des ressources (serveur)
- Des compostants multiples accèdent aux ressources (clients)

Avantages

- Accès unique et contrôlé à la ressource
- Mutualisation des ressources (temps de calcul, stockage, ...)

Inconvénient

- Programme distribué
- Pas de mémoire partagée (asynchronisme)

Application Web

Une application qui utilise *HTTP* comme protocole d'échange entre le client et le serveur.

Optionnellement, elle utilise HTML/CSS (et Javascript) comme langage côté client.

Plan

Historique

1989 : Projet WorldWideWeb de Tim Berners-Lee (CERN)

- Établissement d'une connexion TCP/IP
- Client : `GET` suivi du nom du fichier HTML
- Serveur : renvoie le HTML
- Fermeture de la connexion

1996 : HTTP/1.0

- Spécification des méthodes (`GET`, `POST`, ...)
- Spécification des codes de retour

1999 : HTTP/1.1

- *keep-alive* (plusieurs paires requêtes/réponse par connexion TCP)
- *Virtual hosting* (le même serveur peut héberger plusieurs sites sur plusieurs noms de domaines)
- Caches, proxy, ...

Historique (2)

2015 : HTTP/2

- Compression
- Multiplexage des requêtes (client peut envoyer plusieurs requêtes sans attendre les réponses)
- *Server push* (le serveur envoie des données non demandées)

2020 : HTTP/3

- Basé sur UDP (QUIC)

Format des messages HTTP

Les messages ont la forme suivante

- Ligne initiale CR LF
- zéro ou plusieurs lignes d'en-tête CR LF
- CR LF
- Corps du message (document envoyé, paramètres de la requête, ...)

- *Requête* (client → serveur) la première ligne contient un nom de *méthode* (GET, POST, HEAD, ...), le paramètre de la méthode et la version du protocole
- *Réponse* (serveur → client) la version du protocole, le code de la réponse (200, 404, 403, ...) et les données de la réponse.

Demos

On utilise `telnet` pour envoyer manuellement une requête HTTP basique à un serveur.

On utilise `netcat -l` et on demande à `firefox` de s'y connecter.

Méthodes HTTP

`GET` : demande une ressource en lecture seule
`POST` : transmission de données pour obtenir une ressource
`PUT` : ajout/remplacement d'une ressource sur le serveur
`DELETE` : supprime une ressource sur le serveur
`HEAD` : demande les informations sur une ressource
`PATCH` : modifie une ressource existante
`CONNECT` : utilisé pour les connexions à un *proxy*
`TRACE` : utilisé pour du débuggage

Les codes de réponse

Les codes ont une signification spécifique et permettent au client de gérer la situation correspondante :

`1xx:` Messages informatifs : renvoyés alors que la requête n'est que partiellement reçu, pour indiquer l'état actuel de la communication. Exemple :
`100 Continue` : les en-têtes ont été reçus, en attente du corps la requête.
`2xx:` la requête du client a été reçue, validée et acceptée. Exemple :
`200 OK`.
`3xx:` redirections, le client doit faire une action supplémenptaire pour mener à bien la requête. Exemple :
`308 Permanent Redirect` : la requête devrait être refaite sur l'URL mise dans la réponse du serveur.
`4xx:` erreurs du client. Exemple
`404 Not found`.
`5xx:` erreurs du serveur. Exemple
`505 HTTP version not supported`.

Formulaire HTML (rappel)

L'élément *<form>* permet de créer des formulaires HTML. Un formulaire est constitué d'un ensemble de widgets (zones de saisies de textes, boutons, listes déroulantes, cases à cocher, ...) et d'un bouton submit. Lorsque l'utilisateur appuie sur le bouton, les données du formulaires sont envoyées au serveur. Exemple, fichier `nom.html` :

```
  <body>
    <form method="get" action="https://foo.com/foo.php" >
      Entrez votre nom :
      <input type="text" name="val_nom"/>
      <input type="submit" />
    </form>
  </body>
</html>
```

Requête de formulaire

Lorsqu'un formulaire est validé, tous les éléments de type `<input name="`x_i`">` sont considérés et :

- une chaîne de caractères :

$$S \equiv x_1=v_1\&\ldots\ \&x_n=v_n$$

est formée, où v_i est la valeur du $i^{\text{ème formulaire}}$ encodé (les caractères spéciaux, i.e. non imprimables ou dont le code est > 127 sont échapés).

- Si l'attribut `method` vaut `get`, et l'attribut `action` vaut URL, alors, une requête HTTP `GET` est faite la ressource `URL?S`
- Si l'attribut `method` vaut `post`, et l'attribut `action` vaut URL, alors, une requête HTTP `POST` est faite la ressource `URL`, la chaîne S est passé dans les données de la requête.

Plan

Stateless

Le protocole HTTP est un protocole *sans état*. Lorsque le serveur ferme une connexion TCP après avoir envoyé sa réponse, il « oublie » le client (i.e. rien dans le protoocle n'indique de conserver un historique des requêtes).

Exemple

- Un utilisateur se connecte à un site
- Il saisit sont login/mot de passe et clique « me connecter »
- Le site authentifie l'utilisateur et lui affiche sa page personnelle (et ferme la connexion)
- L'utilisateur recharge la page (nouvelle connexion et requête). Comment pouvoir s'assurer que c'est le même utilisateur ?

Problème Aucun mécanisme dans HTTP (ou dans TCP ou IP) ne permet d'identifier de façon exacte le client, i.e. l'utilisateur qui a fait la première requête.

Cookie

Un *cookie* est un petit paquet d'information, renvoyé par le serveur grâce à l'en-tête `Set-Cookie:`

```
HTTP/1.1 200 OK
Server: nginx/1.14.0 (Ubuntu)
Date: Tue, 19 Oct 2021 17:33:53 GMT
Content-Type: text/html
Content-Length: 2
Set-Cookie: foo=superdata42; Max-Age=3600
...
```

Le site crée un cookie appelé `foo`, contenant la valeur `superdata42` et d'une durée de vie de 3600 secondes.

Cette donnée est stockée *par le navigateur et renvoyée pour chaque requête (tant qu'il n'a pas expiré)*.

Utilité ?

On peut faire persister de l'information entre plusieurs requêtes.

Problème : l'information est sous le contrôle du client.

Session HTTP

Une *session* HTTP est un ensemble de paires requêtes/réponses entre *un* client et le serveur. Un état est maintenu côté serveur pour toute la durée de la session. Exemple :

- Un utlisateur se connecte sur un site (début de session)
- Il saisit son login/mot de passe.
- Si valide, le site définit côté serveur un booléen `connecté`
- À chaque chargement de page, le serveur vérifie la valeur du booléen pour voir si l'utilisateur est connecté. Il y a autant de booléens que de sessions.
- Au bout d'un certain temps d'inactivité ou en cas de déconnection explicite, le booléen est supprimé (fin de session)

Implémentation des sessions

On peut implémenter des sessions en utilisant des cookies.

On suppose que le serveur maintient un dictionnaire de dictionnaires appelé `S`.

1. Lorsque le client se connecte, regarder la valeur d'un cookie particulier (par exemple `MySessionID`)
2. Si ce cookie n'existe pas, on choisit un identifiant aléatoire (*i*), et on crée une nouvelle entrée : `S[i] = {}`
3. On envoie au client l'en-tête `Set-Cookie: MySessionID=`*i*`;Max-Age=600`.
4. Lorsque le client se reconnecte et fournit le cookie, on va chercher `S[i]` et le serveur peut alors y stocker des données propres au client.
5. Si le client ne se connecte pas au bout de 10 minutes, le navigateur détruit le cookie, et on reprend en 1.

Considérations de sécurité pour les sessions

- L'ID de session doit être difficile à deviner/énumérer (⇒ entier aléatoire de taille conséquente >= 128 bits)
- Les cookies peuvent être *signés* avec une clé du serveur (⇒ vérifie que le client ne modifie pas un cookie)
- Les entrées inutilisées dans la table de session doivent être régulièrement vidée. Une adresse IP effectuant trop de connexions doit être bannie temporairement (DOS).

Plan

Éléments de cryptographie (1)

Alice et Bob veulent échanger des données confidentielles.

1. Chiffrement symétrique:
   - Ils se mettent d'accord sur une *clé commune*
   - Alice *chiffre* son message avec la clé et l'envoie à Bob
   - Bob déchiffre le message avec *la clé*

Non sûr (Alice et Bob doivent se mettre d'accord sur une clé en « clair », par email par exemple) ou non pratique (ils doivent se rencontrer physiquement pour échanger la clé).

Efficace: on peut implanter plusieurs algorithmes de chiffrements en utilisant uniquement des opérations logiques de bases (`AND`, `OR`, `XOR`). Il est facile de les implanter sur des puces dédiées (cartes de crédit, processeurs mobiles). Ils sont « sûrs » tant que la clé reste secrète.

Éléments de cryptographie (2)

Alice et Bob veulent échanger des données confidentielles.

2. Chiffrement *asymétrique*:

- Bob crée une *clé publique* $\mathrm{K}^{\mathrm{B}}_{\mathrm{pub}}$ et une *clé secrète* $\mathrm{K}^{\mathrm{B}}_{\mathrm{priv}}$, telle que

$$\forall msg, K^B_{priv}(K^B_{pub}(msg)) = K^B_{pub}(K^B_{priv}(msg)) = msg$$

Bob *diffuse* sa clé publique (sur sa page Web par exemple, ou dans un annuaire de clé) et garde sa clé privée *secrète*.

- Alice *chiffre* son message m avec la *clé publique* de Bob ($\mathrm{K}^{\mathrm{B}}_{\mathrm{pub}}(\mathrm{m})$) et l'envoie à Bob
- Bob déchiffre le message avec sa clé privée: $\mathrm{K}^{\mathrm{B}}_{\mathrm{priv}}(\mathrm{K}^{\mathrm{B}}_{\mathrm{pub}}(\mathrm{m}))=\mathrm{m}$

Sûr et pratique

Peu efficace: repose sur des problèmes mathématiques difficiles. Chiffrer et déchiffrer un message n'est pas réaliste pour des grands messages (vidéo en streaming, requêtes Web, ...).

Éléments de cryptographie (3)

On combine les deux méthodes. (Alice envoie un message à Bob)

- Alice choisit une clé symétrique secrète *s*
- Elle l'envoie à Bob en utilisant la clé publique de ce dernier ($K^B_{pub}(s)$)
- Bob décrypte le message et obtient $s=K^B_{priv}(K^B_{pub}(s))$
- Bob et Alice se sont mis d'accord *de manière sûre* sur une clé commune s! Ils peuvent utiliser un algorithme de chiffrement symétrique pour le reste de la conversation

⇒ Ceci est à la base de protocoles tels que HTTPS

Éléments de cryptographie (4)

Le chiffrement asymétrique permet aussi d'avoir *la preuve* que quelqu'un est bien Bob!

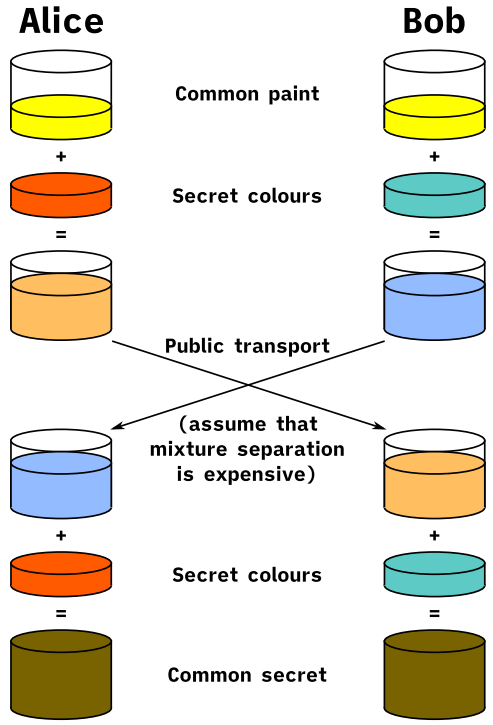
- Alice choisit un message secret aléatoire *m*, sans le divulguer (appelé *challenge*)
- Alice calcule $K^B_{pub}(s)$ et l'envoie à la personne qui prétend être Bob
- Seule la personne qui possède la clé privée de Bob (donc Bob ...) peut déchiffrer le message et renvoyer l'original à Alice.

⇒ Comment garantir que la personne qui a généré les clés au départ est bien Bob ?

Échange de clé

Une autre méthode permettant de se mettre d'accord sur un secret (sans authentification) est le protocole d'échange de clé de Diffie-Hellman.
On suppose une fonction binaire M telle que :

- Étant donnés x, y et z, $M(M(x, y), z) = M(M(y, z), x)$
- Étant donné x et $M(x, y)$ il est difficile de deviner y.



Autorités de certifications

Une *autorité de certification* (ou AC) est une entité habilitée à délivrer des certificats. Une AC est un tiers de confiance. Parmi les différentes AC on retrouve :

- des associations à but non lucratif (comme l'initiative *Let's Encrypt*, libre et gratuite)
- des états
- des entreprises spécialisées

Les systèmes d'exploitation et les navigateurs Web possèdent les clés publiques de ces entités (quelques centaines).

Ces clés sont régulièrement mises à jour.

Format X.509

Les ACs fournissent des certificats. Le standard est le format X.509

```
Certificate:
Data:
    Version: 3 (0x2)
    Issuer: C=US, O=Let's Encrypt, CN=Let's Encrypt Authority X3
    Validity
        Not Before: May  4 08:03:48 2020 GMT
        Not After : Aug  2 08:03:48 2020 GMT
    Subject: CN=nsi-terminale.fr
    Subject Public Key Info:
        Public Key Algorithm: rsaEncryption
            Public-Key: (2048 bit)
            Modulus:
                00:ce:1d:f4:48:24:bd:0d:64:3a:74: ...
            Exponent: 65537 (0x10001)
Signature Algorithm: sha256WithRSAEncryption
     38:90:e9:9d:20:12:9d:35:b7:db:4f:64:14:fd:0e: ...
```

HTTPS

HTTP*Secure* :

1. Le serveur fournit au client sont certificat
2. Le client vérifie (avec la clé publique de l'AC) que le certificat est valide
3. Le client et le serveur se mettent d'accord sur une clé de session symmétrique
4. Le client et le serveur échangent en HTTP standard, mais en chiffrant les messsages.

Les étapes 1 à 3 sont implémentées par exemple par le protocole TLS (Transport Layer Sécurity) et constitue la poignée de main TLS.

Niveaux de certification

Il existe trois niveaux de certification possibles :

- DV (domain validated):le niveau le plus basique. L'AC a juste vérifié que la personne qui a demandé le certificat est bien la personne qui contrôle le nom de domaine certifié.
- OV(organisation validated): le niveau intermédiaire. Des vérifications sont faites sur l'existence légale de la personne morale ou physique correspondant à l'entité certifiée. L'AC effectue des vérifications manuelles auprès de l'entité (généralement une entreprise) qui demande la certification.
- EV (extended validated) : le niveau le plus haut de certification. L'AC vérifie les documents légaux fournis par l'entreprise à certifier et recoupe ces vérifications avec les registres publics (registres des entreprises par exemple), en plus des vérifications faites au niveau OV.

Plan

Service Web

Un service Web est un moyen pour une application d'exposer son API au moyen du protocole HTTP

Il existe plusieurs méthodologies :

- Web API : utilise différents standards tels que WSDL (fichier XML décrivant le service), SOAP (protocole avec des messages XML au dessus de HTTP), UDDI (annuaire de service), ...
- REST : Representational State Transfer. Approche plus simple basée sur des URL et les méthodes HTTP (GET, POST, PUT, DELETE)

REST

Architecture issue de la thèse de Roy Thomas Fielding (2000, U. of Cal. Irvine)

Un service Web REST se caractérise de manière suivante :

- Une ou plusieurs URL servant de point d'entrée
- Un comportement implémenté pour les requêtes GET, POST, PUT, DELETE.

Les contraintes de méthode sont les suivantes

- GET : le traitement du serveur doit être sans effet (read-only)
- PUT/DELETE : le traîtement doit être idempotent (l'état exposé par le système ne doit pas varier après un appel à la méthode avec un paramètre donné)
- *stateless* chaque requête transporte l'état nécessaire, stocké côté client

Le contenu renvoyé par le service Web peut être quelconque, mais on privilégiera le JSON dans ce cours

Classe utilitaire/Appel de service Web

La classe `HTTPUrlConnection` permet de créer des requêtes HTTP en Java (voir la Javadoc). Pour des requêtes simples (GET) on poura utiliser uniquement la classe `URL`.

Exemples d'organisation URI/Méthodes HTTP (1)

Un site veut exporter une *collection* d'objets ainsi que des opérations sur ces derniers.
Url de collection : `http://site.com/api/collection/`

- GET : liste les objets, la taille de la collection, ...
- POST : ajoute un élément (des informations sur l'éléments à créer peuvent être données en paramètres de requête)
- PUT : remplace toute la collection par une autre
- DELETE : supprime la collection
- PATCH : inutilisé

Url d'un objet : `http://site.com/api/collection/object123`

- GET : récupère l'objet et ses méta-données
- POST : inutilisé
- PUT : remplace l'objet existant ou le crée
- DELETE : supprime l'objet
- PATCH : modifie l'objet

Exemples d'organisation URI/Méthodes HTTP (2)

S'il y a une base de donnée sous-jacente alors :
Url de collection : `http://site.com/api/collection/`

- GET → `SELECT id FROM COLLECTION`
- POST → `INSERT INTO COLLECTION VALUES`
- PUT → `DROP TABLE; CREATE TABLE; INSERT`
- DELETE → `DELETE FROM COLLECTION`

Url d'un objet : `http://site.com/api/collection/object123`

- GET → `SELECT * FROM COLLECTION WHERE id=123`
- PUT → `DELETE/INSERT`
- DELETE → `DELETE FROM COLLECTION WHERE id=123`
- PATCH → `UPDATE … SET … WHERE id=123`

Stateless ?

Dans une architecture REST, le serveur ne conserve *aucun état* spécifique à chaque client (i.e. pas de session, ni de cookies).

Chaque client est responsable de conserver son état localement.

Un mecanisme d'authentification/authorisation doit être mis en place

Authentification

Il existe plusieurs méthodes pour s'authentifier auprès d'un service Web :

Basic Auth : Ajout du `user:password` dans un header HTTP `Authorization`
- *Avantage* ⇒ simple
- Inconvénient ⇒ peut sûr

API Key : Le client récupère auprès du serveur un secret qu'il ajoute à chaque requête (le secret peut être renouvelé fréquement). *Avantage* ⇒ simple
- Inconvénient ⇒ peut sûr, non standard

OAuth : *Avantage* ⇒ assez sûr, standardisé
- Inconvénient ⇒ mise en place complexe