Programmation client/serveur avec les sockets - protocoles -

Module
Systèmes Communicants et Synchronisés

Master Informatique
1ère année

L. Philippe & V. Felea

Client / Serveur

- présentation (interface utilisateur)
- persistance (données de l'application)
- traitements (service métier)

découpage en 3 parties

qui fait quoi ?

- déploiement (contexte distribué) - tiers
  - 2 tiers : Client/Serveur
  - client : présentation
  - serveur : persistance

? traitements ?

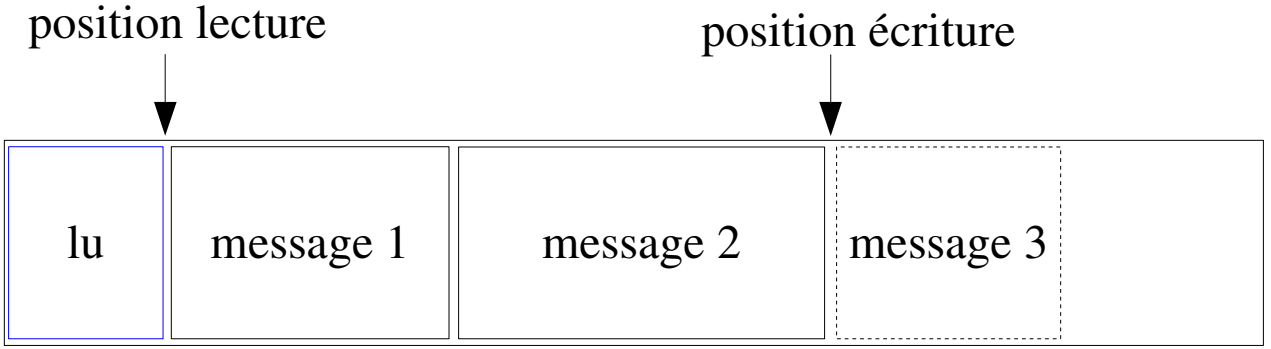
- permettre la collaboration : définir les moyens d'accès

Principe d'échange

- protocole
  - interface du service
  - ensemble des règles qui régissent un échange
    - synchronisation des échanges
    - modes d'échange
    - codage des données échangées
- support : sockets
  - flux d'octets
  - non typé
  - non structuré
- but : définir une interface pour accéder à un serveur sans connaître son implémentation (~ objets)

Exemple – service de buffering (1)

- client
  - envoie des données de type caractère
  - demande à lire *n* caractères (consommation)



Protocole d'échange

- plusieurs possibilités en fonction de l'architecture de la communication
- architecture Client / Serveur
  - mode d'échange : requêtes / réponses
  - simplicité de la synchronisation : limite les risques
  - requête
    - demande à effectuer un traitement
    - paramètres dépendant de la requête
  - réponse
    - retour de requête : statut
    - valeurs résultats

Définition de l'interface (1)

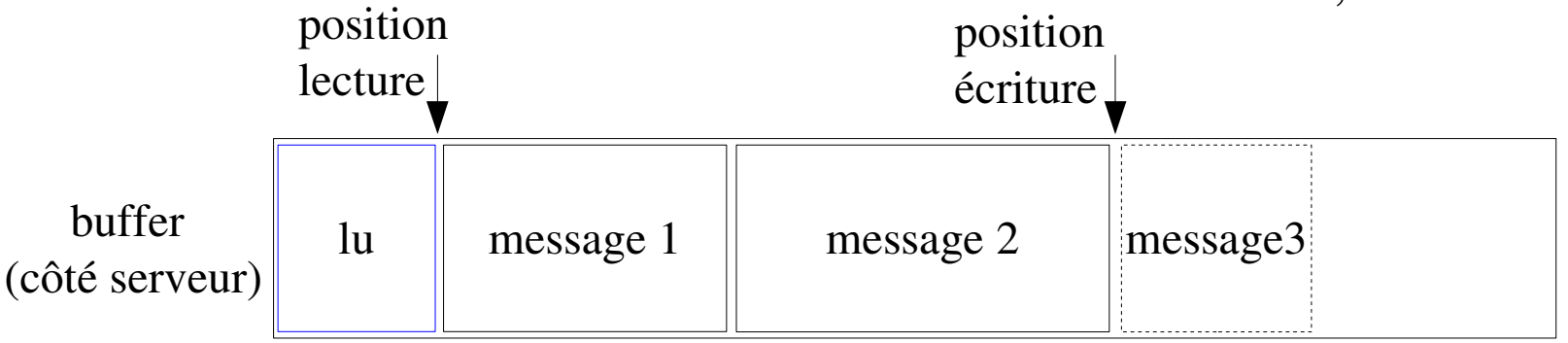
- liste des requêtes et réponses
  - une réponse par requête
  - données échangées
  - cas d'erreur
- raisonnement en terme de fonctions
  - appel de fonction = envoi de requête
  - retour de fonction = envoi de réponse
  - paramètres sur le réseau plutôt que sur la pile
  - intérêt
    - découpage clair
    - modèle de programmation connu
  - interface = liste des fonctions avec paramètres

Définition de l'interface (2)

- interface affichée par le serveur : publication
- définition / structuration des échanges
- paramètres
  - entrée
  - sortie
  - entrée / sortie
  - erreurs

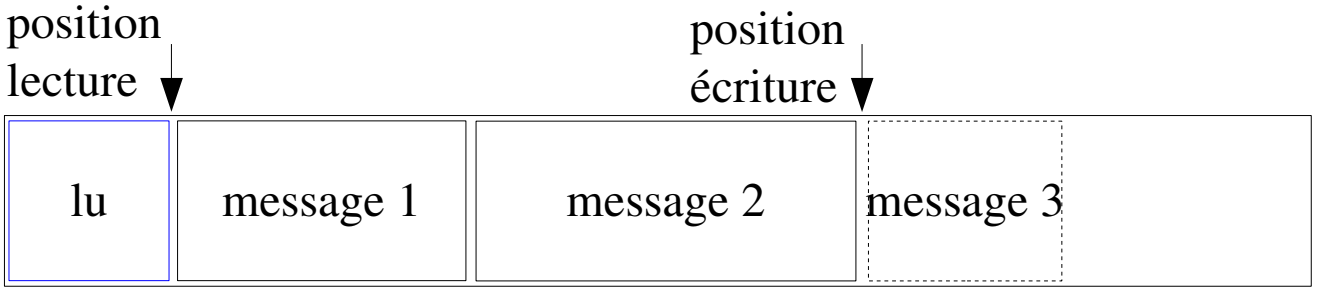
Exemple – service de buffering (2)

- serveur : buffer qui mémorise des caractères
- requêtes / réponses
  - lecture
    - appel : taille des données à lire
    - retour : nombre de données lues, données, code erreur
  - écriture
    - entrée : taille des données, données
    - sortie : nombre de données écrites, code erreur



Exemple – service de buffering (3)

- principe de fonctions



  - *int lecture (char* donneesLues, int* taille)*
    - retourne un code d'erreur
    - recopie les données depuis le buffer en *donneesLues*
    - essaie de lire *taille* caractères et retourne le nombre de caractères lus dans *taille*
  - *int ecriture (char* donneesAEcrire, int* taille)*
    - retourne un code d'erreur
    - envoie les données en *donneesAEcrire* à être écrites dans le buffer
    - envoie *taille* caractères et retourne le nombre de caractères écrits, dans *taille*

Synchronisation des échanges

- attention - l'interface ne donne pas d'informations sur la façon dont ont lieu les échanges
- échanges requête/réponse simples
  - client : send puis recv
  - serveur : recv puis send
- schémas plus complexes
  - suppose de bien suivre en symétrique pour les deux
  - si *n* clients pour un serveur : suivi des requêtes
  - si *n* clients pour *m* serveurs
- risques
  - interblocages

Modes d'échange

- connecté (généralement)
- requête / réponse
- synchrone / asynchrone
  - synchrone : attente de l'établissement de la communication et que le message soit effectivement arrivé à destination

Codage des messages

- sockets = flux non structuré d'octets
  - non typé
  - pas de délimiteur
- établir une convention (interface) entre client et serveur
- format
  - binaire
    - + : rapidité / - : hétérogénéité des représentations
  - texte
    - + : codage ASCII universel et utilisation de protocole existants (HTTP) / - : rapidité
- l'adapter au type de l'application

Protocole d'échange – mise en œuvre

- indépendant du mode de codage
- fixer des types de requêtes
  - identifier une requête entrante
  - 1 par fonction / 1 global / 1 pour plusieurs fonctions ?
- fixer les données associées
  - requêtes : code de requête + données en entrée
  - réponses : code d'erreur + données en sortie
- fixer les cas (codes) d'erreur
  - plus d'infos ?
- l'adapter à l'application

Protocole d'échange – proposition de mise en œuvre : interface (1)

- définition de types de données énumérés (*enum*)
  - codes des requêtes
  - codes d'erreur
- définition de types de données structurés (*struct*)
  - ordre des données
  - typedef struct { … } MaRequete ;
  - requête
    - code de requête (**en premier**)
    - paramètres d'appel (données envoyées par le client)
    - taille variable des requêtes
    - PAS DE POINTEURS
  - réponse
    - code d'erreur **(en premier)**
    - données retournées par le serveur

Protocole d'échange – proposition de mise en œuvre : interface (2)

- données communes (constantes)
  - numéro de port
  - taille buffer, etc.
- fichier partagé
  - .h
  - #include
  - commenté
- ne pas mettre
  - données (structures de données) spécifiques
  - ex : TAIL_BUF, du message mais pas la taille interne du buffer du service

Exemple – service de buffer : l'interface (4)

Rappel des services (sous la forme de fonctions)

```
int lecture (char* donnees, int* taille);
int ecriture (char* donnees, int* taille);
```

codes de requêtes

```
typedef enum {LECT, ECRI} TypCodReq;
 ou
#define LECT 1
#define ECRI 2
```

données communes

```
#define PORT_SERV 6767
#define TAIL_DATA 256
#define ...
```

codes de retour

```
typedef enum {ERR_OK, ERR_BUF_PLEIN, ERR_BUF_VIDE, ERR_CODE_REQ} TypErr;
 ou
#define ERR_OK 0
#define ERR_BUF_PLEIN -1
#define ERR_BUF_VIDE -2
```

Exemple – service de buffer : l'interface (5)

- requêtes

```
typedef struct {
  TypCodReq codeReq;
  int tailLect;
} TypLectReq;
```

```
typedef struct {
  TypCodReq codeReq;
  int nbEcri;
  char donnees[TAIL_DATA];
} TypEcriReq;
```

- réponses

```
typedef struct {
  TypErr codeErr;
  int tailLue;
  char donneesLu[TAIL_DATA];
} TypLectRep;
```

```
typedef struct {
  TypErr codeErr;
  int tailEcri;
} TypEcriRep;
```

Protocole d'échange – mise en œuvre du client

- composition des requêtes
  - obtenir les paramètres de requête
  - constitution du message
    - remplir la structure correspondante
    - encapsulation = message en 1 seul morceau
  - envoi du message
    - ATTENTION PAS DE POINTEUR
- utilisation des réponses
  - réception d'un message de réponse
  - gestion des erreurs
  - délivrer les résultats

Protocole d'échange – mise en œuvre du serveur (1)

- réception des requêtes : mécanisme d'identification de requête
  - si plus d'un type, on ne sait pas quelles données (correspondant à quelle requête) sont dans la socket
- solution
  - envoi du code de requête séparément
    - recevoir d'abord le code de requête
    - allouer la structure adaptée
    - en mode non-connecté : problème réception multiple (ordre et réception non garantis)
    - efficacité

Protocole d'échange – mise en œuvre du serveur (2)

- solutions si envoi de requête (code et données) en un seul paquet
  - déterminer le type de la requête à recevoir
    - réception type de requête
    - option de réception (flags) : MSG_PEEK, laisse les données dans la socket
    - recevoir le type de données dans la structure adaptée
  - définir un seul type de requête (requêtes similaires)
    - définir une union regroupant les types de requêtes
    - réception identique
    - identifier le type

Protocole d'échange – mise en œuvre du serveur (3)

- réception de la requête
- branchement selon le type de requête
  - un cas par type
- appel de code de traitement (fonction)
- génération de la réponse
- renvoi de la réponse sur la même connexion
- conserver la connexion pour les requêtes/réponses ultérieures ?

Exemple – service de buffer : le serveur (6)

- réception des requêtes : MSG_PEEK

```
TypCodReq codeReq;
TypLectReq lectReq;
TypEcriReq ecriReq;
…
/* réception du code de requête, maintien dans la socket */
err = recv(sockTrans, &codeReq, sizeof(codeReq), MSG_PEEK);
if (err <= 0) { … // erreur }
else {
  /* réception en fonction du code */
  switch(codeReq) {
     case LECT :
        err = recv(sockTrans, &lectReq, sizeof(lectReq), 0);
        if (err < 0) {… // erreur }
        …
        break;
     case ECRI :
        err = recv(sockTrans, &ecriReq, sizeof(ecriReq), 0);
        …
        break;
```

```
typedef struct {
  TypCodReq codeReq;
  int tailLect;
} TypLectReq;
```

```
typedef struct {
  TypCodReq codeReq;
  int nbEcri;
  char donnees[TAIL_DATA];
} TypEcriReq;
```

Exemple – service de buffer : l'interface (7)

- mise en œuvre : union des requêtes
  - requêtes : code requête TypCodReq

```
typedef struct {
  TypCodReq codeReq;
  int tailLect;
} TypLectReq;
```

```
typedef struct {
  TypCodReq codeReq;
  int nbEcri;
  char donnees[TAIL_DATA];
} TypEcriReq;
```

  - unions

```
typedef union {
  TypLectReq lr;
  TypEcriReq er;
} TypRequest;
```

```
typedef struct {
  TypCodReq codeReq;
  union {
    TypLectReq lr;
    TypEcriReq er;
  } params;
} TypRequest;
```

Exemple – service de buffer : le serveur (8)

- mise en œuvre : exemple union dans la structure

```
TypRequest req;
TypLectRep lectRep;
TypEcriRep ecriRep;
…
/*  réception du code de requête, maintient dans la socket  */
err = recv(sockTrans, &req, sizeof(req), 0);
if (err <= 0) { … // erreur }
/* réception de la requête */
switch(req.codeReq) {
    case LECT :
        traitLectReq(req.params.lr, &lectRep);
        err = send(sockTrans, &lectRep, sizeof(lectRep), 0);
        if (err != sizeof(lectRep)) { … // erreur }
        break;
   case ECRI :
       traitEcriReq(req.params.er, &ecriRep);
       err = send(sockTrans, &ecriRep, sizeof(ecriRep), 0);
       if (err != sizeof(ecriRep)) {… // erreur }
```

```
typedef struct {
  TypCodReq codeReq;
  union {
    TypLectReq lr;
    TypEcriReq er;
  } params;
} TypRequest;
```

```
typedef struct {
  int tailLect;
} TypLectReq;
```

```
typedef struct {
  int nbEcri;
  char donnees[TAIL_DATA];
} TypEcriReq;
```

Protocole d'échange – gestion des erreurs

- définitions interface / protocole
- validation des données : client ou serveur
  - taille de buffer
  - valeurs bornées : vérification
- gestion des connexions, déconnexions
- vider les sockets ?

Protocoles d'échange - exemplification

- protocoles en mode binaire : problème des représentations
- protocoles en mode texte : internet
  - définition des requêtes et des réponses
  - exemples
    - HTTP = HyperText Transmission Protocol
    - SMTP = Simple Mail Transfer Protocol
    - POP = Post Office Protocol

Protocole d'échange HTTP

- sockets en mode connecté
- commandes en mode texte
- échange de fichiers entre un client et un serveur
- version actuelle est HTTP/1.1 (W3C)
  - décrite par le RFC (Request For Comments) 2616
  - W3C (06/1999) 175 pages
  - http://www.w3.org/Protocols
  - sur le port 80
  - URL : http://nom_machine:no_port/path
  - se contente de transférer le contenu des fichiers entre les machines

Protocole d'échange HTTP - requêtes (1)

- forme générale d'une requête

  commande URL version_HTTP

  entête_1_HTTP : valeur

  …

  entête_n_HTTP : valeur

  CR

  corps

- principales *commandes* des requêtes HTTP
  - *GET* demande une représentation d'une ressource
  - *HEAD* demande d'information sur une ressource sans demander la ressource elle-même
  - *POST* met à jour des données sur la ressource (incluses dans le corps de la requête)

Protocole d'échange HTTP - requêtes (2)

- entêtes
  - informations sur le client : from, host, user-agent, ...
  - informations sur la page contenant le lien : Referer
  - conditions : if-modified-since, if-unmodified-since, ...
  - préférences pour le document : Accept-language, ...
  - la dernière entête est terminée par deux CR
    - marque de fin
- exemple

  GET http://www.google.fr HTTP/1.0

  Accept : text/html

  If-Modified-Since : Sunday, 15-January-2011 14:30:00 GMT

Protocole d'échange HTTP - réponses

- forme générale d'une réponse

  version_HTTP code_erreur message

  entête_1_HTTP : valeur

  ...

  entête_n_HTTP : valeur

  CR

  corps

- exemple

  HTTP/1.0 200 OK

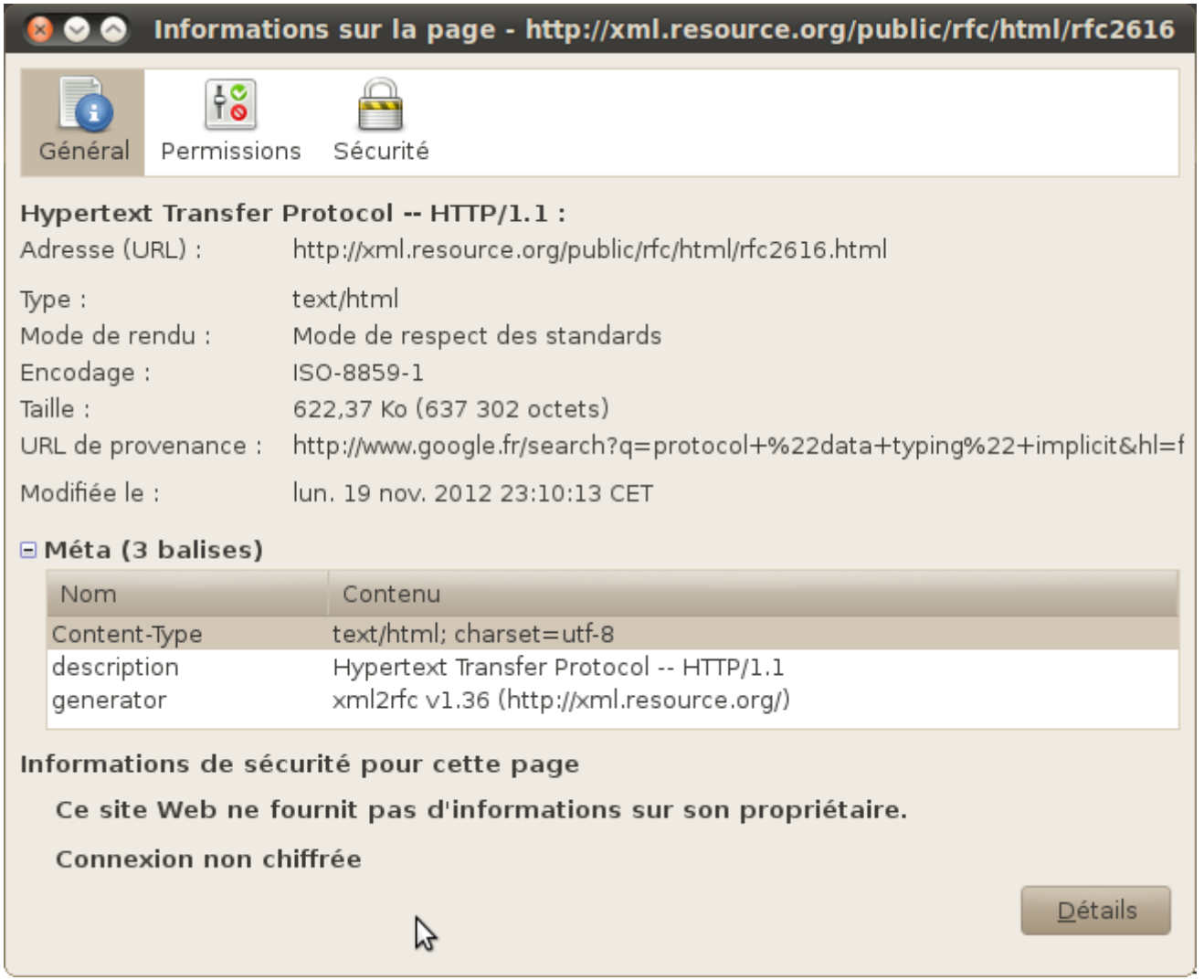
  Date : Tue, 06 Dec 2011 ...

  Content-Type : text/HTML ; charset=...

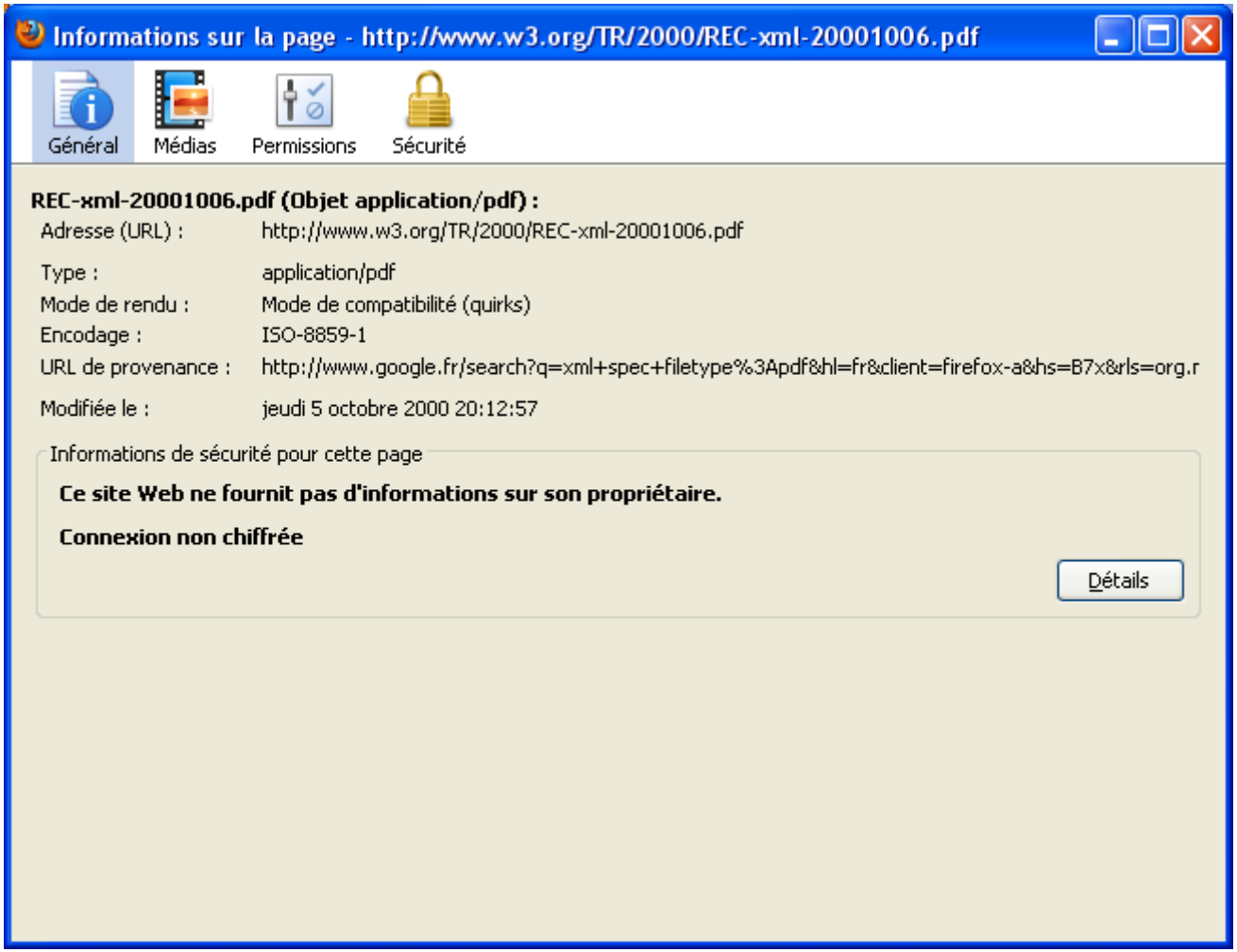
  <!doctype html> ...

- codes d'erreur
  - 2xx : success (200 : OK)
  - 3xx : redirection
  - 4xx : error (400 : Bad Request ; 404 : Not Found)
  - 5xx : internal server-error

Format MIME (1)



Format MIME (2)



Protocole d'échange SMTP (1)

- suite de plusieurs requêtes et messages encodés envoyés par le client et des réponses envoyées par le serveur
- format des requêtes
  - verbe (caractères ASCII alphabétiques)
  - paramètre optionnel
  - CRLF=\015\012 (octal) 13 10 (décimal)
- exemples de verbes
  - *EHLO*
  - *MAIL*
  - *RCPT*
  - *DATA*
  - *QUIT*

Protocole d'échange SMTP (2)

- format des réponses
  - code (3 chiffres ASCII) 200-399 = acceptance, 400-499 = temporary rejection, 500-599 = permanent rejection
  - espace (dernière ligne de la réponse), - (les autres)
  - le texte
  - CRLF
- échanges multiples
  - plus complexe

Protocole d'échange SMTP - exemple

C: EHLO access.univ-fcomte.fr
S: 220 access.univ-fcomte.fr SMTP Ready
C: EHLO pelleas.univ-fcomte.fr
S: 250 access.univ-fcomte.fr
C: MAIL FROM:<*expediteur@hote.domaine.tld*>

S: 250 OK
C: RCPT TO:<mon.dest@giganet.net>

S: 250 OK
C: RCPT TO:<dest@nowhere.fr>

S: 550 No such user here
C: DATA
S: 354 Start mail input; end with <CRLF>.<CRLF>

C: Subject: le projet de scs
C: J'ai bientôt fini
C: la communication avec l'arbitre
C :
C: A+
C: <CRLF>.<CRLF>

S: 250 OK
C: QUIT
S: 221 access.univ-fcomte.fr closing transmission

Protocole d'échange

- typage des données
  - implicite
    - proposition
    - HTTP→HTML
    - problème : comprendre le message
    - avantage : transferts minimums
  - explicite
    - définition du type avec les données
    - exemple : <xml>
    - avantage : identification des données

Protocole d'échange – Java ou .Net

- sur la base des objets
  - encapsule les données
    - = structures
    - délimiteurs = structuré
    - type des requêtes
      - instance of
      - valeur définie dans l'objet
    - type des réponses : codes, chaînes, ...
  - besoin du code (.class) ?
- utilisation sockets idem