SCIENCES SORBONNE UNIVERSITÉ

SRCS : Systèmes Répartis Client/Serveur

TD 3 : Implantation d'un protocole applicatif avec les sockets

Objectifs pédagogiques :
- **Protocole applicatif**
- **Connexion**
- **Gestion des erreurs**
- **Protocole avec/sans état**

Exercice 1 – Serveur Web

Le but de cet exercice est d'implanter une partie du protocole HTTP/1.0 (commandes GET et PUT) avec l'API socket du langage Java.

Question 1

Quelle méthode Java permet d'attendre l'arrivée de demandes de connexions sur un port TCP ?

Question 2

Comment lire des données sur une socket en Java ?

Question 3

Comment écrire des données sur une socket en Java ?

Question 4

À partir de l'extrait de la RFC décrivant le protocole *HTTP* fourni en annexe de ce TD, quel est le rôle de la commande `GET` ?

Question 5

Donner le format de la commande et préciser ce qu'envoient et reçoivent le client et le serveur ?

Question 6

Écrire le pseudo-code d'un serveur *HTTP* gérant la commande `GET`.

Question 7

La commande `PUT` permet à un client Web (navigateur Internet ou programme quelconque), d'envoyer à distance sur un serveur Web un fichier. De même que la commande FTP `PUT`, la commande *HTTP* `PUT` est une commande d'écriture d'un fichier sur un serveur (alors que `GET` est une commande de lecture). La commande `PUT` fait partie des extensions introduites par la version 1.1 du protocole *HTTP* 1.1. Le client envoie la commande suivante :

```
PUT nomDeFichier versionHTTP
en-tête1: valeur1
...
en-têten: valeurn
ligne blanche
Contenu du fichier
```

L'en-tête `Content-Length` est obligatoire et fourni la taille des données envoyées. Par exemple :

```
PUT /index.html HTTP/1.1
Content-Length: 10
ligne blanche
xxxyyyyzzz
```

Le serveur reçoit une telle commande. Si le fichier ne peut pas être écrit ou s'il y a une autre erreur (par exemple commande *HTTP* mal formée), il envoie un message d'erreur de la forme :

```
HTTP/version codeErreur signification
ligne blanche
Message d'erreur
```

Par exemple :

```
HTTP/1.1 404 Not Found
ligne blanche
Fichier inexistant
```

Sinon, si le fichier existe, il envoie un message de la forme :

```
HTTP/version 200 OK
en-tête1: valeur1
...
en-têten: valeurn
```

Par exemple :

```
HTTP/version 200 OK
```

Pourquoi l'en-tête Content-Length est-elle obligatoire ?

Question 8

Modifier le pseudo-code de la question 5 pour ajouter au serveur *HTTP* la gestion d'une commande `PUT`.

Question 9

Donner le pseudo-code d'un client Web envoyant une commande `PUT` à un serveur.

Exercice 2 – Serveur calculette

En utilisant un protocole de transport fiable (TCP), on souhaite réaliser un serveur qui implante les fonctionnalités d'une calculatrice élémentaire fournissant quatre opérations (`+ - * /`) . Des clients doivent donc pouvoir se connecter à distance et demander au serveur la réalisation d'une opération. Le serveur leur répond en fournissant le résultat ou un compte rendu d'erreur.

Question 1

Définir un protocole applicatif (i.e. un ensemble de messages et leur enchaînement) aussi simple que possible implantant cette spécification.

Question 2

Recenser les cas d'erreur pouvant se produire avec ce protocole applicatif et proposer un comportement à adopter lorsque ces erreurs surviennent (on ne s'intéresse ici ni aux erreurs de transport du style message perdu, ni aux pannes de machines).

Question 3

En utilisant les primitives fournies ci-dessous donner le pseudo-code du serveur et le pseudo-code d'un client demandant la réalisation de l'opération 8.6*1.75 :

- `ouvrirCx(serveur)` : demande d'ouverture de connexion vers le serveur
- `attendreCx()` : attente bloquante et acceptation d'une demande de connexion
- `envoyer(données)` : envoi de données
- `recevoir()` : attente bloquante et réception de données
- `fermerCx()` : fermeture de connexion

Question 4

Le serveur est-il avec ou sans état ?

Remarque : on se place au niveau des primitives définies à la question précédente. On ne s'intéresse pas au protocole de transport sous-jacent.

Question 5

Si vous avez répondu sans état à la question précédente, donner un exemple de serveur calculette avec état. Réciproquement, si vous avez répondu avec état, donner un exemple de serveur calculette sans état.

Question 6

On souhaite que plusieurs clients puissent se connecter simultanément sur le serveur. Cela pose-t-il un problème particulier ? Modifier le pseudo-code du serveur pour prendre en compte ce cas.

Question 7

On souhaite maintenant que les clients puissent enchaîner plusieurs demandes d'opérations au sein d'une même connexion. Proposer deux solutions pour cela (une en modifiant le protocole précédent et une sans modification du protocole).

RFC 1945

Network Working Group
Request for Comments: 1945
Category: Informational

T. Berners-Lee
MIT/LCS
R. Fielding
UC Irvine
H. Frystyk
MIT/LCS

May 1996

Hypertext Transfer Protocol -- HTTP/1.0

Status of This Memo

This memo provides information for the Internet community. This memo does not specify an Internet standard of any kind. Distribution of this memo is unlimited.

IESG Note:

The IESG has concerns about this protocol, and expects this document to be replaced relatively soon by a standards track document.

Abstract

The Hypertext Transfer Protocol (HTTP) is an application-level protocol with the lightness and speed necessary for distributed, collaborative, hypermedia information systems. It is a generic, stateless, object-oriented protocol which can be used for many tasks, such as name servers and distributed object management systems, through extension of its request methods (commands). A feature of HTTP is the typing of data representation, allowing systems to be built independently of the data being transferred.

HTTP has been in use by the World-Wide Web global information initiative since 1990. This specification reflects common usage of the protocol referred to as "HTTP/1.0".

Table of Contents

be listed, separated by whitespace. By convention, the products are listed in order of their significance for identifying the application.

```
product         = token ["/" product-version]
product-version = token
```

Examples:

```
User-Agent: CERN-LineMode/2.15 libwww/2.17b3

Server: Apache/0.8.4
```

Product tokens should be short and to the point -- use of them for advertizing or other non-essential information is explicitly forbidden. Although any token character may appear in a product-version, this token should only be used for a version identifier (i.e., successive versions of the same product should only differ in the product-version portion of the product value).

4. HTTP Message

4.1 Message Types

HTTP messages consist of requests from client to server and responses from server to client.

```
HTTP-message   = Simple-Request           ; HTTP/0.9 messages
               | Simple-Response
               | Full-Request             ; HTTP/1.0 messages
               | Full-Response
```

Full-Request and Full-Response use the generic message format of RFC 822 [7] for transferring entities. Both messages may include optional header fields (also known as "headers") and an entity body. The entity body is separated from the headers by a null line (i.e., a line with nothing preceding the CRLF).

```
Full-Request   = Request-Line             ; Section 5.1
                 *( General-Header        ; Section 4.3
                  | Request-Header        ; Section 5.2
                  | Entity-Header )       ; Section 7.1
                 CRLF
                 [ Entity-Body ]          ; Section 7.2

Full-Response  = Status-Line              ; Section 6.1
                 *( General-Header        ; Section 4.3
                  | Response-Header       ; Section 6.2
```



```
                  | Entity-Header )       ; Section 7.1
                 CRLF
                 [ Entity-Body ]          ; Section 7.2
```

Simple-Request and Simple-Response do not allow the use of any header information and are limited to a single request method (GET).

```
Simple-Request  = "GET" SP Request-URI CRLF

Simple-Response = [ Entity-Body ]
```

Use of the Simple-Request format is discouraged because it prevents the server from identifying the media type of the returned entity.

4.2 Message Headers

HTTP header fields, which include General-Header (Section 4.3), Request-Header (Section 5.2), Response-Header (Section 6.2), and Entity-Header (Section 7.1) fields, follow the same generic format as that given in Section 3.1 of RFC 822 [7]. Each header field consists of a name followed immediately by a colon (":"), a single space (SP) character, and the field value. Field names are case-insensitive. Header fields can be extended over multiple lines by preceding each extra line with at least one SP or HT, though this is not recommended.

```
HTTP-header    = field-name ":" [ field-value ] CRLF

field-name     = token
field-value    = *( field-content | LWS )

field-content  = <the OCTETs making up the field-value
                 and consisting of either *TEXT or combinations
                 of token, tspecials, and quoted-string>
```

The order in which header fields are received is not significant. However, it is "good practice" to send General-Header fields first, followed by Request-Header or Response-Header fields prior to the Entity-Header fields.

Multiple HTTP-header fields with the same field-name may be present in a message if and only if the entire field-value for that header field is defined as a comma-separated list [i.e., #(values)]. It must be possible to combine the multiple header fields into one "field-name: field-value" pair, without changing the semantics of the message, by appending each subsequent field-value to the first, each separated by a comma.

4.3 General Header Fields

There are a few header fields which have general applicability for both request and response messages, but which do not apply to the entity being transferred. These headers apply only to the message being transmitted.

```
General-Header = Date                     ; Section 10.6
               | Pragma                   ; Section 10.12
```

General header field names can be extended reliably only in combination with a change in the protocol version. However, new or experimental header fields may be given the semantics of general header fields if all parties in the communication recognize them to be general header fields. Unrecognized header fields are treated as Entity-Header fields.

5. Request

A request message from a client to a server includes, within the first line of that message, the method to be applied to the resource, the identifier of the resource, and the protocol version in use. For backwards compatibility with the more limited HTTP/0.9 protocol, there are two valid formats for an HTTP request:

```
Request        = Simple-Request | Full-Request

Simple-Request = "GET" SP Request-URI CRLF

Full-Request   = Request-Line             ; Section 5.1
                 *( General-Header        ; Section 4.3
                  | Request-Header        ; Section 5.2
                  | Entity-Header )       ; Section 7.1
                 CRLF
                 [ Entity-Body ]          ; Section 7.2
```

If an HTTP/1.0 server receives a Simple-Request, it must respond with an HTTP/0.9 Simple-Response. An HTTP/1.0 client capable of receiving a Full-Response should never generate a Simple-Request.

5.1 Request-Line

The Request-Line begins with a method token, followed by the Request-URI and the protocol version, and ending with CRLF. The elements are separated by SP characters. No CR or LF are allowed except in the final CRLF sequence.

```
Request-Line = Method SP Request-URI SP HTTP-Version CRLF
```


Note that the difference between a Simple-Request and the Request-Line of a Full-Request is the presence of the HTTP-Version field and the availability of methods other than GET.

5.1.1 Method

The Method token indicates the method to be performed on the resource identified by the Request-URI. The method is case-sensitive.

```
Method         = "GET"                    ; Section 8.1
               | "HEAD"                   ; Section 8.2
               | "POST"                   ; Section 8.3
               | extension-method

extension-method = token
```

The list of methods acceptable by a specific resource can change dynamically; the client is notified through the return code of the response if a method is not allowed on a resource. Servers should return the status code 501 (not implemented) if the method is unrecognized or not implemented.

The methods commonly used by HTTP/1.0 applications are fully defined in Section 8.

5.1.2 Request-URI

The Request-URI is a Uniform Resource Identifier (Section 3.2) and identifies the resource upon which to apply the request.

```
Request-URI    = absoluteURI | abs_path
```

The two options for Request-URI are dependent on the nature of the request.

The absoluteURI form is only allowed when the request is being made to a proxy. The proxy is requested to forward the request and return the response. If the request is GET or HEAD and a prior response is cached, the proxy may use the cached message if it passes any restrictions in the Expires header field. Note that the proxy may forward the request on to another proxy or directly to the server specified by the absoluteURI. In order to avoid request loops, a proxy must be able to recognize all of its server names, including any aliases, local variations, and the numeric IP address. An example Request-Line would be:

```
GET http://www.w3.org/pub/WWW/TheProject.html HTTP/1.0
```

The most common form of Request-URI is that used to identify a resource on an origin server or gateway. In this case, only the absolute path of the URI is transmitted (see Section 3.2.1, abs_path). For example, a client wishing to retrieve the resource above directly from the origin server would create a TCP connection to port 80 of the host "www.w3.org" and send the line:

```
GET /pub/WWW/TheProject.html HTTP/1.0
```

followed by the remainder of the Full-Request. Note that the absolute path cannot be empty; if none is present in the original URI, it must be given as "/" (the server root).

The Request-URI is transmitted as an encoded string, where some characters may be escaped using the "% HEX HEX" encoding defined by RFC 1738 [4]. The origin server must decode the Request-URI in order to properly interpret the request.

5.2 Request Header Fields

The request header fields allow the client to pass additional information about the request, and about the client itself, to the server. These fields act as request modifiers, with semantics equivalent to the parameters on a programming language method (procedure) invocation.

```
Request-Header = Authorization            ; Section 10.2
               | From                     ; Section 10.8
               | If-Modified-Since        ; Section 10.9
               | Referer                  ; Section 10.13
               | User-Agent               ; Section 10.15
```

Request-Header field names can be extended reliably only in combination with a change in the protocol version. However, new or experimental header fields may be given the semantics of request header fields if all parties in the communication recognize them to be request header fields. Unrecognized header fields are treated as Entity-Header fields.

6. Response

After receiving and interpreting a request message, a server responds in the form of an HTTP response message.

```
Response        = Simple-Response | Full-Response

Simple-Response = [ Entity-Body ]
```



```
Full-Response   = Status-Line             ; Section 6.1
                  *( General-Header       ; Section 4.3
                   | Response-Header      ; Section 6.2
                   | Entity-Header )      ; Section 7.1
                  CRLF
                  [ Entity-Body ]         ; Section 7.2
```

A Simple-Response should only be sent in response to an HTTP/0.9 Simple-Request or if the server only supports the more limited HTTP/0.9 protocol. If a client sends an HTTP/1.0 Full-Request and receives a response that does not begin with a Status-Line, it should assume that the response is a Simple-Response and parse it accordingly. Note that the Simple-Response consists only of the entity body and is terminated by the server closing the connection.

6.1 Status-Line

The first line of a Full-Response message is the Status-Line, consisting of the protocol version followed by a numeric status code and its associated textual phrase, with each element separated by SP characters. No CR or LF is allowed except in the final CRLF sequence.

```
Status-Line = HTTP-Version SP Status-Code SP Reason-Phrase CRLF
```

Since a status line always begins with the protocol version and status code

```
"HTTP/" 1*DIGIT "." 1*DIGIT SP 3DIGIT SP
```

(e.g., "HTTP/1.0 200 "), the presence of that expression is sufficient to differentiate a Full-Response from a Simple-Response. Although the Simple-Response format may allow such an expression to occur at the beginning of an entity body, and thus cause a misinterpretation of the message if it was given in response to a Full-Request, most HTTP/0.9 servers are limited to responses of type "text/html" and therefore would never generate such a response.

6.1.1 Status Code and Reason Phrase

The Status-Code element is a 3-digit integer result code of the attempt to understand and satisfy the request. The Reason-Phrase is intended to give a short textual description of the Status-Code. The Status-Code is intended for use by automata and the Reason-Phrase is intended for the human user. The client is not required to examine or display the Reason-Phrase.

The first digit of the Status-Code defines the class of response. The last two digits do not have any categorization role. There are 5 values for the first digit:

- o 1xx: Informational - Not used, but reserved for future use
- o 2xx: Success - The action was successfully received, understood, and accepted.
- o 3xx: Redirection - Further action must be taken in order to complete the request
- o 4xx: Client Error - The request contains bad syntax or cannot be fulfilled
- o 5xx: Server Error - The server failed to fulfill an apparently valid request

The individual values of the numeric status codes defined for HTTP/1.0, and an example set of corresponding Reason-Phrase's, are presented below. The reason phrases listed here are only recommended -- they may be replaced by local equivalents without affecting the protocol. These codes are fully defined in Section 9.

```
Status-Code    = "200"   ; OK
               | "201"   ; Created
               | "202"   ; Accepted
               | "204"   ; No Content
               | "301"   ; Moved Permanently
               | "302"   ; Moved Temporarily
               | "304"   ; Not Modified
               | "400"   ; Bad Request
               | "401"   ; Unauthorized
               | "403"   ; Forbidden
               | "404"   ; Not Found
               | "500"   ; Internal Server Error
               | "501"   ; Not Implemented
               | "502"   ; Bad Gateway
               | "503"   ; Service Unavailable
               | extension-code

extension-code = 3DIGIT

Reason-Phrase  = *<TEXT, excluding CR, LF>
```

HTTP status codes are extensible, but the above codes are the only ones generally recognized in current practice. HTTP applications are not required to understand the meaning of all registered status

codes, though such understanding is obviously desirable. However, applications must understand the class of any status code, as indicated by the first digit, and treat any unrecognized response as being equivalent to the x00 status code of that class, with the exception that an unrecognized response must not be cached. For example, if an unrecognized status code of 431 is received by the client, it can safely assume that there was something wrong with its request and treat the response as if it had received a 400 status code. In such cases, user agents should present to the user the entity returned with the response, since that entity is likely to include human-readable information which will explain the unusual status.

6.2 Response Header Fields

The response header fields allow the server to pass additional information about the response which cannot be placed in the Status-Line. These header fields give information about the server and about further access to the resource identified by the Request-URI.

```
Response-Header = Location                ; Section 10.11
                | Server                  ; Section 10.14
                | WWW-Authenticate        ; Section 10.16
```

Response-Header field names can be extended reliably only in combination with a change in the protocol version. However, new or experimental header fields may be given the semantics of response header fields if all parties in the communication recognize them to be response header fields. Unrecognized header fields are treated as Entity-Header fields.

7. Entity

Full-Request and Full-Response messages may transfer an entity within some requests and responses. An entity consists of Entity-Header fields and (usually) an Entity-Body. In this section, both sender and recipient refer to either the client or the server, depending on who sends and who receives the entity.

7.1 Entity Header Fields

Entity-Header fields define optional metainformation about the Entity-Body or, if no body is present, about the resource identified by the request.

```
    Entity-Header  = Allow                    ; Section 10.1
                   | Content-Encoding         ; Section 10.3
                   | Content-Length           ; Section 10.4
                   | Content-Type             ; Section 10.5
                   | Expires                  ; Section 10.7
                   | Last-Modified            ; Section 10.10
                   | extension-header

    extension-header = HTTP-header
```

The extension-header mechanism allows additional Entity-Header fields to be defined without changing the protocol, but these fields cannot be assumed to be recognizable by the recipient. Unrecognized header fields should be ignored by the recipient and forwarded by proxies.

7.2 Entity Body

The entity body (if any) sent with an HTTP request or response is in a format and encoding defined by the Entity-Header fields.

```
    Entity-Body    = *OCTET
```

An entity body is included with a request message only when the request method calls for one. The presence of an entity body in a request is signaled by the inclusion of a Content-Length header field in the request message headers. HTTP/1.0 requests containing an entity body must include a valid Content-Length header field.

For response messages, whether or not an entity body is included with a message is dependent on both the request method and the response code. All responses to the HEAD request method must not include a body, even though the presence of entity header fields may lead one to believe they do. All 1xx (informational), 204 (no content), and 304 (not modified) responses must not include a body. All other responses must include an entity body or a Content-Length header field defined with a value of zero (0).

7.2.1 Type

When an Entity-Body is included with a message, the data type of that body is determined via the header fields Content-Type and Content-Encoding. These define a two-layer, ordered encoding model:


```
    entity-body := Content-Encoding( Content-Type( data ) )
```

A Content-Type specifies the media type of the underlying data. A Content-Encoding may be used to indicate any additional content coding applied to the type, usually for the purpose of data compression, that is a property of the resource requested. The default for the content encoding is none (i.e., the identity function).

Any HTTP/1.0 message containing an entity body should include a Content-Type header field defining the media type of that body. If and only if the media type is not given by a Content-Type header, as is the case for Simple-Response messages, the recipient may attempt to guess the media type via inspection of its content and/or the name extension(s) of the URL used to identify the resource. If the media type remains unknown, the recipient should treat it as type "application/octet-stream".

7.2.2 Length

When an Entity-Body is included with a message, the length of that body may be determined in one of two ways. If a Content-Length header field is present, its value in bytes represents the length of the Entity-Body. Otherwise, the body length is determined by the closing of the connection by the server.

Closing the connection cannot be used to indicate the end of a request body, since it leaves no possibility for the server to send back a response. Therefore, HTTP/1.0 requests containing an entity body must include a valid Content-Length header field. If a request contains an entity body and Content-Length is not specified, and the server does not recognize or cannot calculate the length from other fields, then the server should send a 400 (bad request) response.

> Note: Some older servers supply an invalid Content-Length when sending a document that contains server-side includes dynamically inserted into the data stream. It must be emphasized that this will not be tolerated by future versions of HTTP. Unless the client knows that it is receiving a response from a compliant server, it should not depend on the Content-Length value being correct.

8. Method Definitions

The set of common methods for HTTP/1.0 is defined below. Although this set can be expanded, additional methods cannot be assumed to share the same semantics for separately extended clients and servers.

8.1 GET

The GET method means retrieve whatever information (in the form of an entity) is identified by the Request-URI. If the Request-URI refers to a data-producing process, it is the produced data which shall be returned as the entity in the response and not the source text of the process, unless that text happens to be the output of the process.

The semantics of the GET method changes to a "conditional GET" if the request message includes an If-Modified-Since header field. A conditional GET method requests that the identified resource be transferred only if it has been modified since the date given by the If-Modified-Since header, as described in Section 10.9. The conditional GET method is intended to reduce network usage by allowing cached entities to be refreshed without requiring multiple requests or transferring unnecessary data.

8.2 HEAD

The HEAD method is identical to GET except that the server must not return any Entity-Body in the response. The metainformation contained in the HTTP headers in response to a HEAD request should be identical to the information sent in response to a GET request. This method can be used for obtaining metainformation about the resource identified by the Request-URI without transferring the Entity-Body itself. This method is often used for testing hypertext links for validity, accessibility, and recent modification.

There is no "conditional HEAD" request analogous to the conditional GET. If an If-Modified-Since header field is included with a HEAD request, it should be ignored.

8.3 POST

The POST method is used to request that the destination server accept the entity enclosed in the request as a new subordinate of the resource identified by the Request-URI in the Request-Line. POST is designed to allow a uniform method to cover the following functions:

- o Annotation of existing resources;
- o Posting a message to a bulletin board, newsgroup, mailing list, or similar group of articles;
- o Providing a block of data, such as the result of submitting a form [3], to a data-handling process;
- o Extending a database through an append operation.

The actual function performed by the POST method is determined by the server and is usually dependent on the Request-URI. The posted entity is subordinate to that URI in the same way that a file is subordinate to a directory containing it, a news article is subordinate to a newsgroup to which it is posted, or a record is subordinate to a database.

A successful POST does not require that the entity be created as a resource on the origin server or made accessible for future reference. That is, the action performed by the POST method might not result in a resource that can be identified by a URI. In this case, either 200 (ok) or 204 (no content) is the appropriate response status, depending on whether or not the response includes an entity that describes the result.

If a resource has been created on the origin server, the response should be 201 (created) and contain an entity (preferably of type "text/html") which describes the status of the request and refers to the new resource.

A valid Content-Length is required on all HTTP/1.0 POST requests. An HTTP/1.0 server should respond with a 400 (bad request) message if it cannot determine the length of the request message's content.

Applications must not cache responses to a POST request because the application has no way of knowing that the server would return an equivalent response on some future request.

9. Status Code Definitions

Each Status-Code is described below, including a description of which method(s) it can follow and any metainformation required in the response.

9.1 Informational 1xx

This class of status code indicates a provisional response, consisting only of the Status-Line and optional headers, and is terminated by an empty line. HTTP/1.0 does not define any 1xx status codes and they are not a valid response to a HTTP/1.0 request. However, they may be useful for experimental applications which are outside the scope of this specification.

9.2 Successful 2xx

This class of status code indicates that the client's request was successfully received, understood, and accepted.