Concurrent Systems Fall 2008
Mandatory Assignment 2

ITTP Server

Problem Description

Robin Sharp and Hans Henrik Løvengreen
Informatics and Mathematical Modelling, DTU.

November 2008

1 Task Requirements

Your task is to implement a simplified Web server which can be controlled via a management interface. The general architecture of the system in which the server is to operate is shown in Figure 1.

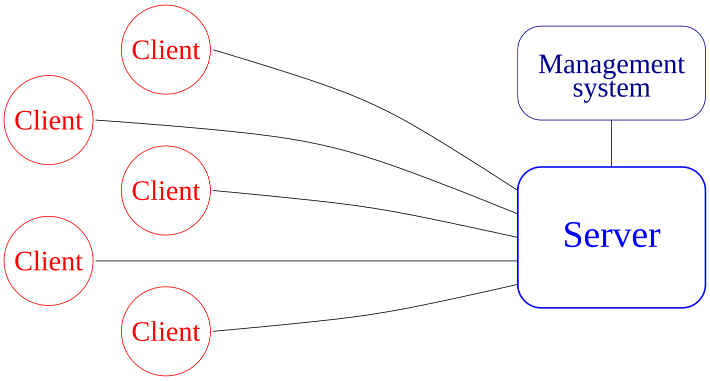

Figure 1: General system architecture

The clients are to communicate with the server via TCP connections to a specific port. The server must be able to handle several simultaneous connections on this port at any time. There should, however, be a limit to the number of simultaneous connections allowed. Since connections are supposed to be persistent (see below), the server should be multi-threaded such that each connection is dealt with by a separate thread in the server. Furthermore, the management interface will have to be handled by its own thread(s). As you will see, the threads dealing with the active connections and the management interface need to access the same data structures, so suitable attention must be paid to how to synchronise their operation, in order to avoid inconsistent behaviour.

The client and server are to communicate by means of a simple client-server protocol in which the client as initiator sends a `request`, to which the server replies with a `reponse`. This protocol is described in more detail in Section 2 below.

Connections should generally be be *persistent*, i.e. a client may use a connection for several request/response interactions. However a client may close the connection if it likes or it may request the server to close it after the response (e.g. if it is known to be the last request). Also the server may close the connection after a period of non-activity or if it needs to stop its service.

The management interface must allow the manager to monitor and control the operation of the server. It is entirely up to you to define the appearance of the interface, i.e. whether it should be a plain text interface where a dialogue appears in an ordinary window, or whether it should be a graphical interface with buttons, text fields, menus or whatever. However, it should offer the functionality described in Section 3 below to the manager.

You are required to implement all the necessary software for this task in Java. You are expected to document your program in an appropriate manner, explaining the structure of the program, the nature and purpose of the classes, methods and variables which it uses, and the tests which you have carried out to ensure its correctness. More detailed instructions and practical hints are available at the *assignment page*:

`http://www.imm.dtu.dk/courses/02152/mandat2.html`

1.1 Syntax notation

The form of the messages to be exchanged by the client and server is described below in an Extended Backus-Naur Form (EBNF) syntax notation, using the following conventions:

- The form of the syntactic elements is defined by a set of syntax rules, each with the form: `a ::= b`, where `a` is the name of the class of syntactic elements being defined, and `b` describes the form of elements of this class, as described below.
- On the right-hand side of a syntax rule, the names of syntax classes, such as `request`, `host`, `pchar`, are used to indicate that any element from that class can be used.
- A sequence of one or more characters enclosed in `" "`, such as `"/"`, `"GET"` and `"ittp://"`, stands for exactly the sequence of characters enclosed in `" "`.
- Notations of the form `x|y` indicate *alternatives* ("either x or y").
- Notations of the form `[x]` indicate *optional elements* ("possibly x").
- Notations of the form `{x}*` indicate *a sequence of 0 or more elements* ("0 or more x's").
- Notations of the form `{x}+` indicate *a sequence of 1 or more elements* ("1 or more x's").

As a notational convenience, the symbol `NL` is used to denote a newline (Carriage Return-Linefeed combination), and the symbol `SP` to denote blank space (one or more spaces or tabulator characters). In addition, the syntax class `alpha` contains the letters of the English alphabet (in both upper and lower case), and the class `digit` contains the decimal digits.

2 The ITTP client-server protocol

The clients are to communicate with the server using a protocol which is actually a variant of the Hypertext Transfer Protocol (HTTP) used between Web clients and servers. To avoid confusion, we call it the **Infratext Transfer Protocol, ITTP**.

2.1 Client requests

The ITTP client is to send requests described by the syntax:

```
request      ::=  request_line NL {request_head NL}* NL [body]
request_line ::=  method  SP  request_URI  SP  ITTP_version
method       ::=  "GET" | "HEAD" | "PUT"
request_URI  ::=  [scheme ":"] ["//" host [":" port]] "/" path
scheme       ::=  "ittp"
host         ::=  hostname | IPv4address
port         ::=  {digit}+
path         ::=  {pathseg "/"}* pathseg
ITTP_version ::=  "ITTP/2.8.3"
pathseg      ::=  {pchar}+
pchar        ::=  alpha | digit
                | "-" | "_" | "." | "'" | "$"
```

Here it is assumed that `hostname` and `IPv4address` are respectively hostnames and IPv4 addresses given in the usual dot-notation. Thus a request consists of a request line followed by zero or more header lines and possibly a request body. The request line:

- Selects a method to be used on the server,
- Specifies the resource to be accessed on the server by supplying a URI, which may specify the protocol, the host, or the port number, and must specify a path to the file containing the resource.
- Specifies which version of the ITTP protocol is to be used. (As can be seen, this is currently version 2.8.3)

The following rules apply to the URI:

1. If a host is given, it must denote the host on which the server is running.
2. If a port number is given, it must equal the port currently used by the server.
3. The `path` is relative to a *base directory* on the server machine. For instance, if the base directory is set to `/home/bill/ittp`, then the path `02152/exam.txt` denotes the file `/home/bill/ittp/02152/exam.txt`. For security reasons (to prevent access to files "below" the base directory in your directory tree), you should not allow the use of `".."` in paths.

Since the `request_URI` follows the general URI-syntax, it may be analyzed as such.

A `body` is only included in PUT requests. It must contain a sequence of octets which make up the intended new content of the resource. If the resource does not already exist, the server must create it. The request will be successful if and only if the user has authorisation to modify (or if necessary create) the resource described by the URI. The user is required to supply authorisation information in the PUT request, as described below.

2.2 Client request header lines

The header lines in ITTP requests follow the syntax:

```
request_head ::=  Authorisation
                | Connection
                | Content-length
                | Date
                | Host
                | If-Modified-Since
                | Range
```

The detailed syntax and semantics of these alternatives is as follows:

Authorisation: Specifies credentials used to identify the client user, following the syntax:

```
Authorisation ::= "Authorisation:" [SP] "Basic" SP uidp64
```

where `uidp64` is the base64 encoding of the userid/password combination `uidpass`:

```
uidpass ::= userid ":" password
userid  ::= alpha { pchar }*
passwd  ::= { pchar }+
```

This header line is mandatory in PUT requests and must be checked by the server. If incorrect credentials are given for access to a resource, a response with response code 401 (see below) must be given.

(You may assume for simplicity that at any instant the server requires the same credentials for *all* files relative to its *base directory*, so that the client must provide exactly these credentials in order to be allowed to execute a PUT method.)

Connection: Indicates that the client does not need the connection after the current request:

```
Connection ::= "Connection:" [SP] "close"
```

As part of the response, the server should include a similar line among the headers. It should then close the connection after the response has been completed.

Content-length: Specifies the length in octets of the resource referred to in the request, following the syntax:

```
Content-Length   ::= "Content-Length:" [SP] {digit}+
```

This header line is mandatory in PUT requests, where the content length must be equal to the length of the request body. For all other requests (where there is no request body, so the length is always zero), the line may be omitted.

Date: Specifies the date and time at which the request was made, following the syntax:

```
Date       ::= "Date:" [SP] datetime
datetime   ::= wkday "," SP day SP month SP year SP
                         hour ":" min ":" sec SP zone
zone       ::= alphazone | numzone
alphazone  ::= alpha alpha alpha
numzone    ::= "GMT" sign hour ":" min
sign       ::= "+" | "-"
```

Where:

- `wkday` is a 3-letter abbreviation of the weekday,
- `day` is a 2-digit form of the day of the month,
- `month` is a 3-letter abbreviation of the name of the month,
- `year` is a 4-digit year number.
- `hour`, `min`, `sec` are all 2-digit numbers giving the hour, minute and second respectively.
- `zone` is a code for the time zone. Two formats are allowed: The traditional *3-letter codes* (such as GMT, CET, PST, etc.), and the so-called *offset codes*, which express the time zone as an signed offset relative to GMT, where the offset is specified as two 2-digit numbers giving the number of hours and the number of minutes respectively.

For example: `Fri, 07 Nov 2008 16:25:01 CET` corresponds to 16:25:01 (Central European Time) on Friday, 7th November 2008 (using the 3-letter time zone code). Using the offset code, this time would be specified as `Fri, 07 Nov 2008 16:25:01 GMT+01:00`, since Central European Time is one hour ahead of GMT.

Host: Specifies the host from which the resource is being requested, following the syntax:

```
Host     ::= "Host:"  [SP] hostname
```

where `hostname` is given using the standard dot-notation for host names. If the host name does not correspond to the host on which the server is running, the request is not valid, and a response with response code 404 (see below) should be returned.

If-Modified-Since: Is used to make the requested method conditional: if the resource has been modified since the time specified, the method is to be carried out as usual. Otherwise, a response code 304 (see below) must be returned.

```
If-Modified-Since ::= "If-Modified-Since:" [SP] datetime
```

where `datetime` is as decribed above. This header line applies only to GET and HEAD methods.

Range: Specifies the portion of the resource referred to, following the syntax:

```
Range    ::= "Range:" [SP] first "-" last
```

where `first` and `last` are both decimal numbers (≥ 0), which respectively give the number of the first and last octets of the portion within the resource. For example: `Range: 123-456` specifies the portion of the resource from octet 123 to octet 456 (both inclusive). The first octet of the entire resource has number 0. It is considered a syntax error (code 400) if `last` is less than `first`, or if a range is specified in a PUT request.

As indicated by the syntax, the request header lines are to be terminated by a blank line, which separates them from the request body, if present.

2.3 Server responses

The ITTP server is to send responses described by the syntax:

```
response      ::=  response_line NL {response_head NL}* NL [body]
response_line ::=  ITTP_version  SP  response_code
```

Thus a response consists of a response line followed by zero or more header lines, and possibly a response body. An extra, blank line terminates the header lines, and separates them from the response body, if present. The `body` is any sequence of octets (bytes) taken directly from the resource (file). Note that the body needs not insist on `CR LF` as the new-line indication. Therefore, the body should not be transferred line-by-line, but byte-wise.

The response code consists of a 3-digit numeric code and a text explaining the significance of the code. The possibilities are as follows:

```
response_code ::=  "200 OK"
                 | "201 Created"
                 | "206 Partial content"
                 | "304 Not modified"
                 | "400 Syntax error in request"
                 | "401 Unauthorised"
                 | "404 Resource not found"
                 | "405 Method not allowed"
                 | "416 Requested range not satisfiable"
                 | "500 ITTP version not supported"
                 | "501 Service unavailable"
                 | "503 Method not implemented"
                 | "505 Internal server error"
```

The situations in which these responses should be given are as follows:

200 The request has succeeded. The header lines and body of the response contains appropriate information, depending on the method used in the request, as specified below.

201 The request has been fulfilled and has resulted in a new resource being created.

206 As for 200, except that only a part of the resource was requested and returned.

304 The request contained an `If-Modified-Since` header line, but the resource has not been modified since the specified time.

400 The server cannot understand the request due to a syntax error.

401 The credentials used for authorisation are incorrect.

404 The resource cannot be found on the server. Also used to signal inconsistent host or port values.

405 The server is (currently) unable to execute the requested method on the resource specified by the client.

416 The start of the range specified in the request lies outside the limits of the resource.

500 The server does not support the version of ITTP specified in the request.

501 The server is currently unable to handle the request due to temporary overloading or maintenance.

503 The server does not know the method specified in the request.

505 The server has encountered an unexpected condition which prevents it from fulfilling the request.

Note that the ITTP response is only required to contain useful information in the form of header lines and possibly a response body if the response code is 200, 201 or 206. When this is the case, the rest of the response contains information which depends on the requested method, as follows:

GET: Header lines describing the resource, as described below, followed by a response body containing the requested content of the resource. (Note that this will not necessarily be the entire content of the resource, as the `Range` request header line can be used to select part of the resource.)

HEAD: Header lines describing the resource, exactly as if a GET request had been used, but without any response body.

PUT: Header lines describing the resource after execution of the PUT method. There is no response body.

For response codes $\geq$ 300, no header lines are mandatory, but you may optionally include header lines that you find relevant.

2.4 Server response header lines

The header lines in ITTP responses follow the syntax:

```
response_head::=  Connection
                | Content-Length
                | Content-MD5
                | Content-Range
                | Content-Type
                | Date
                | Last-Modified
                | Server
```

For non-error responses (code numbers $2xx$), the header lines are mandatory, optional or disallowed depending on the method request according to the following scheme:

Header type	**GET**	**HEAD**	**PUT**
`Connection`	O	O	O
`Content-Length`	M	M	—
`Content-MD5`	O	O	O
`Content-Range`	O	O	—
`Content-Type`	O	O	M
`Date`	M	M	M
`Last-Modified`	M	M	M
`Server`	O	O	O

where M $\sim$ mandatory, O $\sim$ optional, — $\sim$ disallowed. For partial responses (code 206) to GET and HEAD requests, `Content-Range` is also mandatory. The presence of optional header lines is controlled by management variables (as described in section 3).

The detailed syntax and semantics of these alternatives is as follows:

Connection: Independently of the response code, this header line indicates that the connection will be closed by the server immediately after the response. This may be due to a client request or due to the server stopping its service.

```
Connection       ::= "Connection:" [SP] "close"
```

Content-Length: Specifies the length in octets of the portion of the resource referred to in the request, following the syntax:

```
Content-Length   ::= "Content-Length:" [SP] {digit}+
```

Note that if the `Range` request header line is included in the request, then the content length may be smaller than the length of the entire resource. In a response to a GET request, the content length must be equal to the length of the response body. For a HEAD response, the content length should be the length of the body that a GET response *would* have sent.

Content-MD5 [EXTENSION A]: Specifies the MD5 message digest (checksum) of the content of the requested resource, following the syntax:

```
Content-MD5      ::= "Content-MD5:" [SP] digest
```

where `digest` is a base64 representation of the MD5 digest evaluated from the content of the (partial) resource requested. Note that in the response to a PUT request, this digest is evaluated on the content of the resource *after* it has been created or modified.

Content-Range: Specifies the portion of the resource being sent as a response, following the syntax:

```
Content-Range    ::= "Content-Range:" [SP] first "-" last "/" length
```

where `first` and `last` are both decimal numbers, which respectively give the number of the first and last octets of the portion within the resource as described for the `Range` request header. If the requested range starts within, but exceeds the limit of the resource, the range is cut off at the end of the resource. `length` is the total length of the resource.

Content-Type: Specifies the media type of the resource, following the syntax:

```
Content-Type     ::=  "Content-Type:" [SP] media-type
media-type       ::=  type "/" subtype
type             ::=  "text" | "image" | "application"
subtype          ::=  "plain" | "html" | "gif" | "jpeg"
                    | "octet-stream"
```

In all cases where this response header line is generated, the server will determine the combination of type and subtype from the “suffix” of the filename of the resource, in the following way:

Suffix	Type/subtype
`.html, .htm`	`text/html`
`.txt`	`text/plain`
`.gif`	`image/gif`
`.jpeg, .jpg`	`image/jpeg`
other	`application/octet-stream`

Date: Specifies the date and time at which the response was made, following the syntax:

```
Date            ::= "Date:" [SP] datetime
```

Last-modified: Specifies the date and time at which the resource was last modified, following the syntax:

```
Last-Modified   ::= "Last-Modified:" [SP] datetime
```

Server: Specifies the server software version in use, following the syntax:

```
Server          ::= "Server:" [SP] description
```

where `description` can be any text (without `NL`) describing the software.

3 Server management functions

The operation of the server is controlled by a number of *management variables*, whose values can be controlled from the management interface.

The server is also required to maintain a number of *status variables*, which are used to store statistics about the running of the server. It is necessary to be able to keep track of:

- The number of connections established.
- The number of currently open connections.
- The number of incoming ITTP requests
- The number of error responses (numeric response code ≥ 400).
- The number of octets transferred in the response bodies.

You may refine this information and add other parameters if you like.

3.1 Management interface functionality

The management interface must allow the manager to perform the following tasks:

1. Display the current value of all management variables.
2. Start and stop the server. Initially, the server should be be in the started state where it must accept new connections (up to some default limit, say 5). When stopped, no further connections should be accepted and open connections should be closed as soon as any active requests have been served.
3. **[EXTENSION A]** Properly shut down the server. When requested through the management interface **or when the JVM is terminated by an external signal**, the server should finish any active requests and then shut down properly. For this, a `ShutdownHook` should be set in the JVM.
4. **[EXTENSION A]** Set a management variable which determines the maximum number of simultaneous connections (up to some fixed upper limit, say 20). Suitable action must be taken if the number is set below the current number of connections.
5. Set the management variables which determine which of the methods GET, HEAD and PUT the client may request from the server.
6. Set the management variables which determine which of the optional header lines (`Content-MD5` (if implemented), `Content-Type`, and `Server`) are to be returned in responses.
7. Set the management variable which determines the maximum permitted period of inactivity for a connection. (If a connection is inactive for longer than this period, the server should automatically close the connection.) The new value needs only be used after subsequent requests.
8. Set management variables defining the credentials which the client must supply in order to be allowed to perform a PUT operation.
9. Continuously display the current values of all status variables.
10. Reset (to zero) the status variables.

All methods and header lines should be enabled initially.

The management interface may be implemented either

- as an integral part of the server program, or
- **[EXTENSION B]** as a separate client program that controls the server remotely, e.g. via a dedicated TCP connection or using RMI.

In the remote control case, you must specify what should happen if more than one management client attempts to control the server.