



MODULAR PROGRAMME ASSESSED COURSE-WORK SPECIFICATION

Module Details:

Module Code: uqc109s2	Module Title: Computer Networks & Operating Systems	
Module Leader: Nigel Gunton		
Module Tutors: Ian Johnson John Counsell Nigel Gunton Graham Darke		
Assignment CW1	Element Number: Weighting 25%	Total Assignment Time: 12 hrs + lab time

Dates:

Date assignment issued to students: w/b 3rd Nov.	Date for return of marked work: w/b ?? Jan
Submission Place: post-box in N foyer, below the North stairs	Date of Submission: w/b ?? Dec.
	Time of Submission: 10.00am

Deliverables:

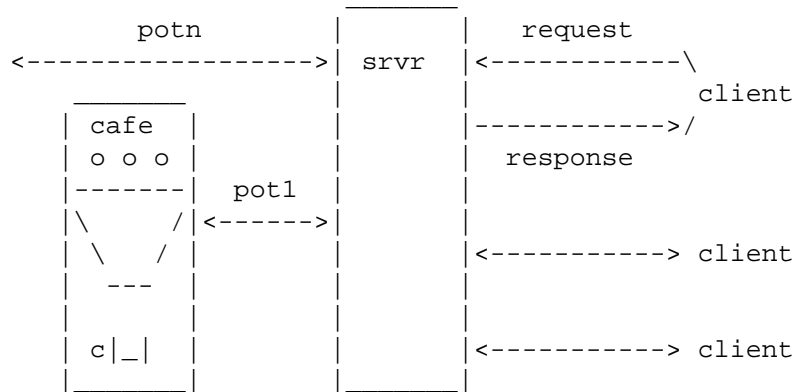
As listed on the Assignment spec sheet

You May Work In Pairs For This Assignment

Requirements :

The design and development of an RFC2324 compliant Coffee Pot Server.

Most modern applications include network or Internet connectivity. In order to gain a better understanding of the role of an operating system and networking software in supporting such applications, you are to design and implement a standards compliant server and a client. This will support HTCP, the Hyper-Text Coffee-Pot Control Protocol. This protocol is designed to allow the remote control of coffee brewing machines via the internet. An HTCP server accepts requests from remote clients and uses these requests to manage the coffee machine. The server provides responses based on the status of the coffee brewing machine.



In order to achieve this you will need to :-

- i) Read and understand RFC2324. This provides an introduction to RFCs (Request For Comments). RFCs are the standards documents of the Internet, all key protocols are described and defined in them. Networking software is required to comply with the defined standards if it is to interact correctly with other software using the same protocols. Note that, as with all protocols during their development phase, there are some ambiguities in RFC2324. It is up to you to recognize and interpret these ambiguities and omissions in such a way that reflects your understanding of the intent of RFC2324. This should be documented as the first part of the deliverables. Credit will be given for the quality of decisions made at this stage.
- ii) Compare the set of request messages and responses provided below, in BNF, with the written descriptions shown in RFC2324. Decide which of this set of messages and responses should be generated by the client and which by the server. These messages will be encapsulated within HTTP and should include any extensions to HTTP that may be required at both client and server end. Your decisions and modifications should be documented and again credit will be given for the standard of your documentation.
- iii) Develop a full top-level design for both the client and the server. You will be expected to use a recognized design methodology such as UML. It is recommended that you consider designing both the server and the client as state machines, (statechart diagrams in UML). Credit will be given for the quality of the design and can be offset against implementation, although at least a basic implementation will be expected. Note that the most complex areas are
 - Parsing user input in the client and constructing a valid request.
 - Parsing the request message in the server and building an appropriate response.

and your design should show how you undertake the parsing.

- iv) Implement the design using either C or Java. There will be lab support for C only. Other languages will be considered but must be agreed upon by your lab tutor. You will be expected to use the code examples provided, it will be up to you to decide which of the examples are the most appropriate. These are all available via the links on Ian Johnsons web-pages ... <http://www.cems.uwe.ac.uk/~irjohnso> . Note that these pages are only available on-site. The majority of examples are in C. If you use code from other sources then it must be clearly identified as such and you must be able to explain its functionality to your lab tutor.

Note that in the absence of a coffee machine you will have to simulate it by either

- a) implementing variables in the server to represent the coffee machine status
or
- b) reading and writing to a file holding the coffee machine status
or
- c) implementing a coffee machine which is controlled by the server

NOTE:

Your coffee-pot server is not required to provide all additions but must recognize the additions list and respond with an appropriate message.

Deliverables:

- 1) A short (< 500 words) description in your own words that shows your understanding of RFC2324. 0% to 10%.
- 2) Documentation describing the messages and responses of your system. 0% to 10%
- 3) Your design(s). 0% to 30%
- 4) Your code for the client and the server. This must be signed and dated by your lab tutor to whom it must have been demonstrated successfully. The lab tutor must indicate the degree to which s/he considers it to have met the requirements. 0% to 30%

You should demonstrate the following stages :-

- i) A stand-alone client with user input from the keyboard and output to the screen.
- ii) A stand-alone server with input from either the keyboard or a file and output to the screen or to a file.
- iii) The combination of your client and server. The server should print out the request message and the client should print out the response message.

A further 20% is available for quality of design or quality of code or for demonstrating your client/server against a third party client/server. Factors to be considered will include clarity of the design, the degree to which the design matches the code, commenting, robustness and testing.

Note that the overall quality of the documentation that you hand in will affect the mark allocated. Spelling, layout, code comments etc are all important.

```

coffee-url = "coffee" ":" [ "//" host ] [ "/" pot-designator ] [ "?" additions-list ]

pot-designator = "pot-" integer ; for machines with multiple pots
additions-list = #( addition )

HTCPCP-message = Request | Response ; HTCPCP/0.1 messages

generic-message = start-line
                  *(message-header CRLF)
                  CRLF
                  [ message-body ]
start-line       = Request-Line | Status-Line

message-header = field-name ":" [ field-value ]
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, separators, and quoted-string>

message-body = entity-body

Method        = "PROPFIND" ; Section 2.1.3
              | "GET" ; Section 2.1.2
              | "BREW" ; Section 2.1.1
              | "POST" ; Section 2.1.1
              | "WHEN" ; Section 2.1.4
              | extension-method

extension-method = token

Request       = Request-Line
              *( (request-header
                  | entity-header ) CRLF )
              CRLF
              [ message-body ]

Request-Line = Method SP Request-URI SP HTCPCP-Version CRLF

Request-URI  = "*" | absoluteURI | abs_path | authority

request-header = Accept-Additions ; Section 2.2.2.1
                | Safe-Condition ; Section 2.2.1.1

```

Server responses

```
Response = Status-Line
          *((response-header
            | entity-header ) CRLF)
          CRLF
          [ message-body ]
```

Status-Line = HTCPCP-Version SP Status-Code SP Reason-Phrase CRLF

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 - Request received, continuing process
- 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

```
Status-Code = "100" ; Continue
              | "200" ; OK
              | "400" ; Bad Request
              | "401" ; Unauthorized
              | "402" ; Payment Required
              | "403" ; Forbidden
              | "404" ; Not Found
              | "406" ; Not Acceptable
              | "410" ; Gone
              | "501" ; Not Implemented
              | "503" ; Service Unavailable
```

```
response-header = Age
                 | Retry-After
                 | Safe
```

entity-header = Accept-Additions

```
Retry-After = "Retry-After" ":" ( HTTP-date | delta-seconds )
Safe = "Safe" ":" safe-nature
safe-nature = "yes" | "no" | conditionally-safe
conditionally-safe = "if-" safe-condition
safe-condition = "user-awake" | token
```

Entity Body

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

The entity body of a POST or BREW request MUST be of Content-Type "message/coffeepot". Since most of the information for controlling the coffee pot is conveyed by the additional headers, the content of "message/coffeepot" contains only a coffee-message-body:

```
coffee-message-body = "start" | "stop"
```

Accept-Additions header field

```
Accept-Additions = "Accept-Additions" ":"
                  "#"(addition-type [accept-params])

addition-type    = ( "*"
                    | milk-type
                    | syrup-type
                    | sweetener-type
                    | spice-type
                    | alcohol-type
                    ) * ( ";" parameter)

milk-type        = ( "Cream" | "Half-and-half"
                    | "Whole-milk" | "Part-skim"
                    | "Skim" | "Non-dairy" )

syrup-type       = ( "Vanilla" | "Almond" | "Raspberry")

sweetener-type   = ( "White-sugar" | "Sweetener"
                    | "Raw-cane" | "Honey")

spice-type       = ( "Cinnamon" | "Cardamon" )

alcohol-type     = ( "Brandy" | "Rum" | "Whiskey"
                    | "Aquavit" | "Kahlua" )

parameter        = number | volume

number           = ( "1" | "2" | "3" | "4" | "5" )
volume           = ( "dash" | "splash" | "little" | "medium" | "lots" )
```

Safe-Condition header field

```
Safe-Condition = ("user-awake" | token )
```