CNT4007C - **Project 1**
02/18/2019
David Cabrera
dacabdi89@ufl.edu

NOTE: This PDF file was automatically generated from the README.md file of the **private** GitHub repository that hosts the code.

QUIZNET

USAGE

BUILDING

To build the solution you can either,

- `$ make qclient`,
- `$ make qserver`,
- or `$ make all` to build both applications.

The `makefile` provides comprehensive recipes to run tests, compile in debug mode with no optimizations, or produce manual entries. To run in debug mode set `MODE=debug` before calling on `make`. See `makefile` for more details.

RUNNING

The applications can be run as per the specifications of the project. After building the solution, you can run the server application by issuing the command

```
$ ./qserver
```

standing on the same subdirectory where the `make` command was issued.

Similarly, the client application can be started by issuing the command,

`./qclient TARGETHOST PORT`.

Please note that both `qclient` and `qserver` have been implemented using POSIX command line options with `getopt` and `getopt_long`.
Use option `-h` on both executables for further usage details.

Run `qclient` with the option `--nonpersistent` or `-n` for non persistent mode (See *Additional Features* section).

The server application provides logging and verbosity options.
Run with `--verbose` or `-v` for complete escaped reporting of serialized representations of both requests and responses.

Refer to the corresponding manual entries for details.

MANUAL ENTRIES

Due to the inability to run `man -l manual-entry` on the `storm.cise.ufl.edu` host, the command to properly visualize the manual entry will be,

```
$ nroff -Tascii -man [file] | less
```

PROTOCOL SPECIFICATION

Before any request or response, the client MUST perform an application level *handshake* by which it gives notice to the server on the type of session (persistent vs non persistent) it wishes to establish, as well as any other relevant information required to maintain the session.

The client should initiate the handshake on the FIRST request made to the server for each connection. It will send a type `s` request containing the parameters in the `body`.

REQUESTS

GENERAL REQUEST SPECIFICATION

All *Requests* must adhere to the following syntactic specification

```
type length\n
body\n
```

Field	Syntax	Semantics	Optional
`type`	Only **one** character from a-z	Indicates the type of request	No, all requests must have a `type` field
`length`	A decimal nonnegative integer	Describes the length of the request's `body` in bytes	No, an empty request is expected to include a 0 length
`body`	Any number of ASCII characters	Request's body and payload	Yes, but 0 length must be indicated

More precisely,

```
request = type , ' ' , length, '\n' , (body)? '\n'
        ;
type    = a-z
        ;
length  = [0-9]+
        ;
body    = ? any sequence of chars ?
        ;
```

If the server *knows* how to handle the incoming type of request and the request is processed succesfully, a type `o` OK response will be sent to the client carrying any requested data.

Otherwise, a type `e` ERROR response will be returned. If the error is due to an unrecognized type of request, the error type will be numbered `UNKREQ`.

TYPES OF REQUESTS

HANDSHAKE

Also know as *greeting* or *session negotionation* this is the FIRST request made over each session. It lets the server get ready for the upcoming requests.

```
s length\n
parameters\n
```

POST

Upload a question to the server

```
p length\n
body\n
```

This type of request must include a `length` field and a `body` field containing,

```
tags\n
question-title\n.\n
choices.\n
(correct-answer)\n
```

1. `tags` field with zero or more comma separated tags.

2. `question-title` with a string of alphanumeric and white space characters terminated by the sequence `\n.\n`.
3. Any number of `choices` of the form `(choice-character) choice-text\n.\n` followed by an additional `\n.\n` that ends the section.
4. The correct answer `correct-answer` followed by an end of line.

Please observe that the server will enforce the following rules

1. A question must contain at least 2 choices.
2. Every choice must be of the format `(choice-character) choice-text\n.\n`.
3. The `choice-character` value is a unique a-z character, the choices must be in alphabetic order, and the first choice must be `a`.
4. The `correct-answer` provided must exist in the set of choices.

Expect an error response if any of this rules is violated. The error is likely to indicate what kind of violation took place by providing information in the `extra` field of the error response.

DELETE

Request a deletion using type `d` and indicating the question's identifier in the body of the message.

```
d length\n
question-id\n
```

Expect an `NOTFND` error response if question doest not exist, or a `INVQID` if the the question id is not a numeric value.

GET

Request a question using type `g` and indicating the question's identifier in the body of the message.

```
g length\n
question-id\n
```

Expect an `NOTFND` error response if question doest not exist, or a `INVQID` if the the question id is not a numeric value.

GETR

Request a random question.

```
```

```
r 0\n
\n
```

Expect an `EMPTYQ` error response if the server has an empty quiz book.

CHECK

Check if a choice is the valid response to a question identified by its question id.

```
c length\n
question-id\n
choice\n
```

Expect a `NOTFND` error if question does not exist or a `CHNFND` if the question does not contain the provided choice.

KILL

Kill server process.

```
k 0\n
\n
```

QUIT

Inform the server that the client wishes to leave.

```
q 0\n
\n
```

This request is relevant only to *persistent* connections.

RESPONSES

All responses will come in two types, OK (type `o`) responses indicating a successful request and ERROR (type `e`) responses along with information on the failure.

TYPES OF RESPONSES

OK

```
o length\n
body\n
```

The content fo the body of the response will depend on the particular request that caused the response.

ERROR

Error responses along with error codes and extra information.

```
e length\n
[number]\n
[symbol]\n
[message]\n.\n
[extra]\n.\n
```

no	symbol	message	extra
0	`UNKERR`	Unkown error	Exception message, if any
1	`NOTFND`	Question not found	`[question-id]` provided
2	`MALQUE`	Malformed question body	Exception message, if any
3	`CHNFND`	Choice does not exist in question	`[choice]` and `[question-id]`
4	`UNKREQ`	Server does not know how to handle request type	Received request type
5	`INVQID`	Invalid format for question id provided	Provided id
6	`EMPTYQ`	Empty quiz book	None
7	`REQTSZ`	Request type length must be one character	Type provided

IMPLEMENTATION

General description

The solution was implemented in C++. Wrappers interfaces, and data models were written around the networking routines and the data representation. This approach allowed to develop the application abstracting the data management and ensuring integrity across the code.

The implementation is divided in three major areas, networking, data models, and other types (exceptions, escaping functions, etc).

Networking

Objects `Socket` and `Host` provide wrappers with all types of operations around the file descriptors and hosts representations used by the operating system. Using this abstractions allowed to gain some leverage over the complexity of the structures used at the Operating System level.

`QuizServer`

`QuizServer` is an abstraction of the set of services provided by the server application. Upon construction it receives it receives a host
host representation and a socket object to work with. Internally, when the method `run` is called, the object takes care of the typical initialization routing of the `Socket` object, namely, it calls methods that wrap around `bind`, `listen`, and `accept`.

Internally, the object uses a hash map pairing characters to callables. This approach allowed to drop new request/response procedures at-will if the specifications of the protocol so required. That is, the procedure that takes care of forming the response to a type `p` request, is a callable keyed on the char `'p'` of the `_handlers` map. Similarly, each corresponding procedure takes in a `Request` object and produces a `Response` object that is then serialized and sent back to the client. The handlers are typically *lamba expressions*, this provides *ad-hoc* request handling emplacement.

If any error were to occur, it is properly identified and reported back to the client by building a type `e Response` containing, usually, content that mirrors the exception that was raised on the server side.

`QuizServer` also takes a `QuizBook` object with a data model representation of the questions read/written to/from the question bank file (refer to the section 5 of the manual for file format information). All the entity manipulations are mediated by the `QuizBook` interface.

`QuizServer`, as per the protocol specifications, expects the client to perform a brief handshake, where it specifies the type of connection it desires to establish. This handshaking can be used in the future to communicate other relevant parameters.

`QuizClient`

On the other hand, the `QuizClient` utilizes a similar pattern of matching characters to callables. Each handler will take in the line and data introduced over standard input and build a `Request` object to relay back to the server. Before returning the object, it sets a `std::function` typed method with a *lambda expression* that will handle the response once the servers replies.

`QuizClient` will create a `Socket` object if it functions on *persistent* mode. Otherwise, it will instantiate a new one over each request.

Please notice that `QuizClient` performs a brief application level handshake (or session negotiation) with the server to give notice of the connection mode and other relevant information.

Data Models

For every component of a question, the question itself, and the entire quiz book, objects defining its interfaceable contracts were implemented. Using this approach the serialization and deserialization of the models became a centralized issue, facilitating bug tracking and standardization.

Each object validates its input and enforces data requirements. All cases provide overloaded stream operators to allow easy output, verification, and serialization.

To avoid data loss on server crashes, the `QuizBook` works on the file using a writethrough approach.

Data models include `Tag`, `TagCollection`, `Choice`, `ChoiceCollection`, `Question`, `QuestionSolved` (containing a question along with a verified solution), etc.

Exceptions

Customized exception objects have been provided for `ProtocolException` and `NetworkException`. These exception objects provide granularity in the catching of errors, allowing to distinguish between the two types of problems. Both client and server are robust and can survive (graciously) the other party leaving the session. These endpoints will use protocol specifications to communicate each other exceptions and errors and handle them appropiately. Refer to the *Protocol Specification* session for more details.

ADDITIONAL FEATURES

Although the specifications did not require implementing both persistent and non persistent mode, the client takes a flag `-n` or `-nonpersistent` to allow non persistent mode.