UNIVERSITEIT•STELLENBOSCH•UNIVERSITY
jou kennisvennoot • your knowledge partner

Computer Programming 143 Project

2015

Instructions

1. All assignments for this practical must be submitted on learn.sun.
2. You must do all parts of the project **on your own**. If you are found to have **copied** any part of the project from any other source, unless instructed to do so in the assignment, the whole assignment will be considered **incomplete**. Copying can lead to suspension from the university. Special software will be used to check for copying, so don't take any chances!
3. An empty Ecplise project is provided – use this Eclipse project and add your code for all three parts of the project to it. Read the instructions in Section 2 to help you get started with the Eclipse project.
4. **Your workspace for the project must be on the C drive (for ex. on the Desktop) while working on your project, otherwise it will not work! Store your work in H:\CP143 or H:\RP143 before logging out of your PC, and copy it from there before you start working on it again.**
5. Include a comment block at the top of each source file according to the format given. It must include the correct filename and date, your name and student number, the copying declaration, and the title of the source file.
6. **Indent your code correctly!** Making your code readable is not beautification, it is a time and life saving habit. Adhere to the standards. You can use Ctrl+A and then Ctrl+I to auto-indent.
7. You will have to demonstrate your project and answer random questions on the project at the last practical session. Your mark for the project will partially be based on which sections you successfully complete, and partially on your answers to the questions. Successful completion of each part (parts 1 - 3) is a necessary but not sufficient condition – you will also have to answer the questions during the demonstration.
8. A lot of thought and effort has been put into this project to ensure that you learn something from it. Be sure to use this opportunity to learn and enjoy it!

1 Project Overview

The aim of the project is to write a computer program for communicating with another computer on the internet. The purpose of the communication is to retrieve encrypted messages from the other computer. However, before the other computer will send the messages, your computer must first be authenticated by generating the correct response to a challenge posed by the other computer. Only after being successfully authenticated will the encrypted messages be sent.

Your program will be executed on your PC (in the lab or at home), which is called the client, whereas the other computer (called the server) can be almost anywhere else in the world, as long as it is connected to the internet. This model is known as the client-server computing model and is used for many applications, including browsing websites on the internet.

The client sends a request to the server, to which it replies with a response – it is always the client that initiates the communication. For this project you will use the Hypertext Transfer Protocol (HTTP), which is the same request-response protocol used by your browser for communicating with web servers. The HTTP protocol has different methods for different requests (for example GET, POST, PUT, etc.), however, for this project all requests will be performed using the POST method. The POST request method is designed to request that a web server accepts the data enclosed in the request's message body, to which the server can then respond with data or just a fail/success message. It is often used when uploading a file or submitting a completed web form on a website.

In this project we will use the JavaScript Object Notation (JSON) for sending data to the server in the request's message body, as well as for receiving data in the responses from the server. JSON is a lightweight data-interchange format that is easy for humans to read and write and also easy for machines to parse (break the data down into its individual parts) and generate.

The user operating the client needs authorisation in order to get access to the server, which will be granted if the user enters the correct username and password. The sequence of network request-response transactions between the client and the server is called an HTTP session. The HTTP session for this application can be described as follows.

- Before the client can send or receive messages it needs to request a (random) session ID, which is generated by the server. The server generates the session ID, associates it with the requesting user's username, and sends it to the client in its response.
- The client takes the session ID and the user's username and password, and generates an encryption key that will be used to encrypt and decrypt messages going to and coming from the server for this session. The server already knows what the password for this particular user is, so it can generate the same encryption key to encrypt and decrypt messages going to and coming from the client. The client then encrypts and sends a predetermined secret message to the server to verify its authenticity, to which the server should respond with a message indicating success or failure.
- If the client is successfully authenticated, the server will allow the client to send other commands, otherwise it discards the current session ID and does not do anything the client says, unless the client requests for a new session ID.

The project has been divided into three parts as follows:

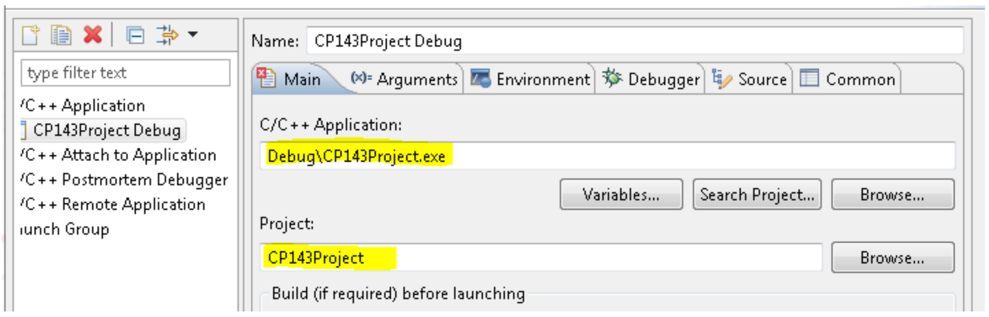
1. JSON objects (see Section 3.3.1) and number conversion - write code to create JSON objects and write/read data to/from JSON objects, as well as to convert single digit hexadecimal numbers to decimal numbers.
2. Encryption (see Section 4.3.2), checksums (see Section 4.3.1), and file IO - write code to implement a given algorithm to encrypt and decrypt strings, calculate checksums of strings, and keep record of the communication in a log file.
3. HTTP communication (see Section 5.3.1) - write code to implement the client side of the project's HTTP session specifications.

Each part of the project should take about a week to complete.

2 Getting Started

Follow these instructions to set up your Eclipse project:

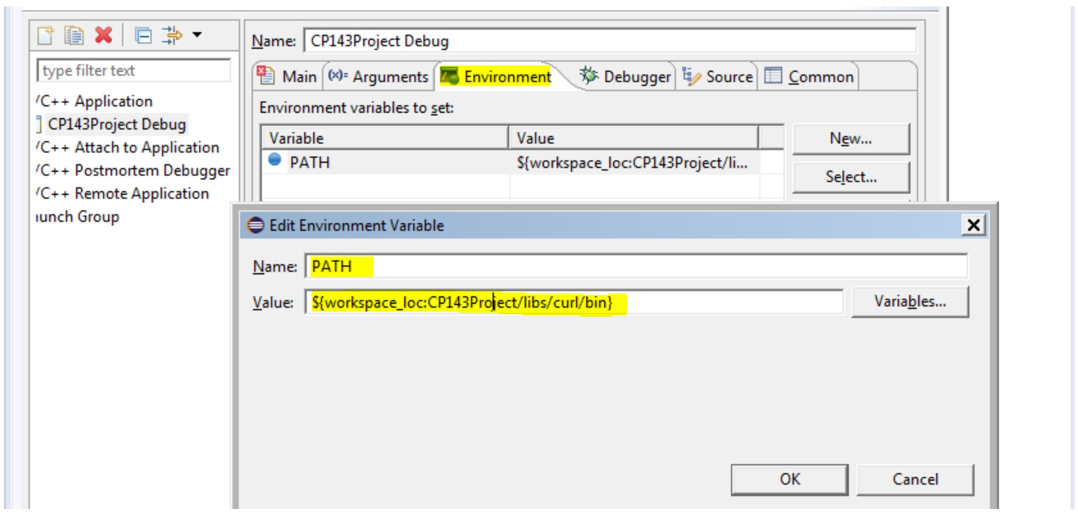
1. Create a folder called “CP143ProjectWorkspace” on your Desktop.
2. Download the “CP143Project_Release.zip” archive from learn.sun into your “CP143ProjectWorkspace” folder and extract it there. After extracting it your directory structure should be “CP143ProjectWorkspace\CP143Project” and not “CP143ProjectWorkspace\CP143Project_Release\CP143Project” (or anything else). If the directory structure is incorrect, Eclipse will not find the correct libraries when running the project executable.
3. Open Eclipse and select your “CP143ProjectWorkspace” folder as workspace.
4. Once Eclipse is open, click on `File -> Import...`
5. In the Import window, click on `General -> Existing Project into Workspace`, and then click on `Next`.
6. Next to “Select root directory:”, click on `Browse...` and select the “CP143Project” Folder inside your “CP143ProjectWorkspace” folder. Click on `OK` and then `Finish`.
7. Click on `Run -> Debug Configurations...`
8. In the Debug Configurations window, double-click on `C/C++ Application`, and enter the following text under the `Main` tab:


Name: CP143Project Debug

C/C++ Application: Debug\CP143Project.exe

Project: CP143Project

and under the `Environment` tab:
copy and paste this text: `${workspace_loc:CP143Project/libs/curl/bin}`


Name: CP143Project Debug

Environment variables to set:

Variable	Value
PATH	${workspace_loc:CP143Project/li...

Edit Environment Variable

Name: PATH

Value: ${workspace_loc:CP143Project/libs/curl/bin}

9. Click `OK`, `Apply`, and `Close`, and then you are ready to start!

3 Part 1: JSON Objects

3.1 Introduction

The aim of this project is for you to write a computer program that can communicate with another computer (the server) on the internet. The communication will be done by sending and receiving data in a specific format that both your program and the program on the server can easily generate and interpret. The format that will be used for this project is called JSON, and for this part of the project you will implement functions that your program will use to generate and interpret JSON objects. Some other useful functions will also be implemented, such as functions for determining the length of a string, removing white space from a string, and converting single digit hexadecimal numbers to decimal numbers.

3.2 Goals

In this part of the project you will write code to perform the following:

- Create JSON objects
- Read a name/value pair from a given JSON object
- Write a name/value pair to a given JSON object
- Get string representation of a JSON object
- Calculate the length of a null-terminated string (character array)
- Remove white space from a string (character array)
- Convert a hexadecimal digit to a decimal number

3.3 Background

3.3.1 JavaScript Object Notation (JSON)

A JSON object is an unordered set of name/value pairs. An object begins with { (left brace) and ends with } (right brace). Each name is followed by **:** (colon) and the name/value pairs are separated by **,** (comma) which is illustrated as follows:

```
{ name1: value1, name2: value2 }
```

The data in this object, `value1` and `value2`, can be accessed simply by using their corresponding names, `name1` and `name2`. Some programming languages have built-in support for this, whereas in C we will be using a library (called "parson") to create and interpret these types of objects.

A value can be a string in double quotes, or a number, or true or false or null, or an object or an array. These structures can be nested. A string is a sequence of zero or more Unicode characters, wrapped in double quotes, using backslash escapes. A character is represented as a single character string. A string is very much like a C or Java string. White space can be inserted between any two name/value pairs (also called tokens) without any effect. Except for a few details on encoding, the JSON language has now been completely described. Another example of a JSON object containing two name/value pairs is as follows:

```
{ "sessionID": "unauthed", "message": "<username>" }
```

We will be using a C library called parson for parsing (extracting data from) JSON objects, since the C programming language does not have built-in support for JSON. For this object, getting the value associated with the name "`sessionID`", which will be done with a call to a function in the C parson library, will return the string "`unauthed`". Similarly, the data associated with the name "`message`" is a username (the < and > indicate that it is a variable). The following shows how this is done in more detail using the parson library.

- **Creating JSON objects:** Before using a JSON object, it first needs to be declared and initialised, which can be done using the following two statements:

```
JSON_Value *rootValue = json_value_init_object();
JSON_Object *rootObject = json_value_get_object(rootValue);
```

The `JSON_Value` and `JSON_Object` data types are defined in the parson library. The statement in the first line creates a JSON value and initializes a JSON object that is stored inside the JSON value. This is necessary to allow for arrays of JSON objects and nested JSON objects to be accessed using the pointer to a root value (`rootValue`). The function call to `json_value_get_object` in the second line simply retrieves the JSON object inside the JSON value and returns a pointer to it. Note that these functions allocate the memory needed for the JSON value and object and return pointers to the allocated memory. When the JSON value is no longer needed, its memory must be freed, which for the code above can be done with the following function call:

```
json_value_free(rootValue);
```

- **Reading name/value pair from JSON object:** The parson library contains functions for getting the data (value) associated with a name, given a reference to the JSON object containing the name/value pair. Suppose the JSON object above, `root_object`, contains a name/value pair with an unknown string value with the name "`testString`". The unknown string value can then be retrieved by using the following statements:

  ```
  char *value;
  value = json_object_dotget_string(rootObject, "testString");
  ```

  Note that the `json_object_dotget_string` function returns a pointer to some of the memory allocated to the JSON value containing the JSON object, which will be freed when the memory for the JSON value is freed.

- **Writing name/value pair to JSON object:** The value associated with a name can be updated in a similar way to reading from it. Assuming a JSON object has been declared and initialised properly, a new name/value pair can be created, or the existing value of the pair updated, by using the following statement:

  ```
  json_object_set_string(rootObject, "sessionID", "unauthed");
  ```

  This will set the value associated with the name "`sessionID`" equal to "`unauthed`" in the JSON object referenced by `root_object`.

- **Getting the string representation of a JSON object (serialization):** A very useful property of JSON objects is that they can be represented, and thus also stored and transferred, using strings. After a JSON object has been declared, initialised and its name/value pairs added, its string representation can be obtained using the following statements (this is also called object serialization):

  ```
  char *serializedObject = NULL;
  serializedObject = json_serialize_to_string_pretty(rootValue);
  ```

  Note that the `json_serialize_to_string_pretty` function takes the JSON **value** as argument, and not the JSON object. Depending on the contents of the JSON object, `serializedObject` will then be a string in a similar format to

  ```
  { name1: value1, name2: value2 }
  ```

  Also note that the `json_serialize_to_string_pretty` function allocates the memory needed for the string and returns a pointer to it. This memory must be freed, and for the above example this is done with a call to

```
    json_free_serialized_string(serializedObject);
```

3.3.2 Length of null-terminated string

Strings in C are represented as arrays of characters (char[]). The end of the string is indicated by the '\0' character. The length of a string can thus be determined by beginning at the start of the array and counting the number of characters until a '\0' character is reached (the count excludes the '\0').

3.3.3 Hexadecimal Numbers

A hexadecimal number has a base of 16, whereas a decimal number has a base of 10. The following table shows the equivalent decimal numbers for the first 16 hexadecimal numbers (single hexadecimal digits):

Hexadecimal	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Decimal	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Note that the characters A – F can also be lower case.

3.4 Instructions

You will write a number of functions to implement the goals of this part of the project. A test program has been provided that must be used to test the written functions. The main function of the test program implements a number of tests that verifies the correct behaviour of each function.

3.4.1 Creating JSON objects

Write a function to create and initialize a JSON object. The function must return a reference to a JSON value that contains the JSON object, and its prototype must be as follows:

```
    JSON_Value* createJSONObject();
```

3.4.2 Read value for a specified name from a given JSON object

Write a function that reads the value associated with a specific name in a given JSON value containing an object. The function prototype must be as follows:

```
    const char* getValueFromName(JSON_Value *rootValue, const char *name);
```

3.4.3 Write name/value pair to JSON object

Write a function to write a name/value pair to a given JSON value containing an object. The function prototype must be as follows:

```
    void setNameValuePair(const JSON_Value *rootValue, const char *name,
            const char *value);
```

3.4.4 Get string representation of a JSON object

Write a function to get the string representation of a given JSON value that contains a JSON object. The function prototype must be as follows:

```
    char *convertToString(const JSON_Value *rootValue);
```

3.4.5 Determine length of string (character array)

Write a function to determine the length of a given null-terminated string. The function prototype must be as follows:

```
int getStringLength(const char *string);
```

3.4.6 Remove white space from a string (character array)

Write a function to remove all white space from a given string. The output of the function should be a copy of the input string but with white space removed. For this function, the characters that should be classified as white space are: `' '`, `'\t'`, `'\n'` (not `'\0'`!). The function prototype must be as follows:

```
void removeWhiteSpace(char *stringNoSpace, char *string);
```

Remember to allocate memory for `stringNoSpace` before calling this function!

3.4.7 Convert hexadecimal digit to a decimal number

Write a function that will determine and return the equivalent decimal number of a single hexadecimal digit **given as a character**. The function prototype must be as follows:

```
int hexDigit2Dec(char hexDigit);
```

Use the information in Section 3.3.3 to help you to implement this function. Hint: Use the ASCII table. Also keep in mind that the characters in hexadecimal numbers (A - F) can be either upper or lower case characters.

3.4.8 Verification

To verify that the functions that you have written are correct, you must build your program and run it. The project template that you must use already contains a main function that will run a number of tests and indicate whether each test has 'PASSED' or 'FAILED'. All the tests need to have passed before you continue with the next part of the project. A successful verification of all the tests is shown below:

```
========================================================================
Testing JSON Utilities
========================================================================
Creating JSON Value                                            : PASSED
Setting Key-Value pair of JSON Object                          : PASSED
Reading value for specific key from JSON Object                : PASSED
Get string representation of JSON object                       : PASSED
Getting length of array                                        : PASSED
Removing whitespace from string                                : PASSED
Converting hex to decimal                                      : PASSED
```

4 Part 2: Cryptography and Checksums

4.1 Introduction

One of the requirements for this project is that the communication between your program and the server must be encrypted. For this part of the project you will implement functions that will be used by your program to encrypt and decrypt the data that will be sent and received. Additionally, you will implement a function that you can use to log events in your program, which will be useful for debugging and keeping track of requests and responses to and from the server in part 3 of this project.

4.2 Goals

In this part of the project you will write code to perform the following:

- Append a line describing an event to a log file
- Calculate a checksum of a string (character array)
- Generate an encryption key from a given sessionID, username and password
- Encrypt a message that is represented as an ASCII string (character array)
- Decrypt an encrypted message that is represented as an ASCII string (character array)

4.3 Background

4.3.1 Checksum

A checksum is a value (sum) that is sent with data so that the program on the receiving end can check the integrity of the received data. For this project, we will use a very simple checksum function which, starting with a sum of zero, just multiplies the ascii value of each character in the data (the data will be a string of characters) with a constant value and adds it to the previous sum. This sum will then be sent with the data to the server, after which it will follow the same steps to calculate a sum from the data it has received, and compare its sum with the one that was sent with the data.

4.3.2 Encryption Algorithm

The first step that is needed for the encryption algorithm is to generate an encryption key, which will be used to encrypt and decrypt messages. For this project, a special function called a hash function will be used to generate an encryption key by using three strings as inputs: A session ID, a username, and a password. The hash function will concatenate the three strings, in the order they are listed, and then apply an algorithm to calculate a number. This number will then be converted to a hexadecimal number and stored as a string (this can easily be done using the `sprintf` function and the `“I64x”` formatting option). A message can then be encrypted by converting its characters to their integer values (using ASCII) and adding the first character of the encryption key to the first character of the message, the second character of the encryption key to the second character of the message, and so on, starting from the beginning of the encryption key after every 16 characters. The encrypted message can then be decrypted by subtracting the bytes of the same encryption key from the encrypted message and converting the integer values back to characters.

4.4 Instructions

You will write a number of functions to implement the goals of this part of the project. A test program has been provided that must be used to test the written functions. There is a call to the `testJSONUtils` function in the body of the `main` function in your `client.c` file. This function call is what causes the tests for part 1 of this project to be performed. Add a call to the `testEncryptionUtils` function below the call to `testJSONUtils`, so that the tests for part 2 will also be performed.

4.4.1 Append a line describing an event to a log file

Write a function to append a line that describes an event in your program to a log file. The name of the log file must be "`log.txt`", and the function prototype must be as follows:

```
void appendToLogFile(int level, const char *location, const char *part1, const
    char *part2);
```

The `level` parameter can have three different values: 0 for an ERROR, 1 for a WARNING, and 2 for INFO. The `location` parameter should contain the name of the function where the call is made to this function. The `part1` and `part2` parameters should contain a description of the error/warning/info, and the text to be appended to the log file should be formatted as follows:

```
[<level>] in <location>: <part1><part2>\n
```

An example of such a line:

```
[INFO] in getAuthRequestString: Auth request string generated for user: 15407756
```

4.4.2 Calculate a checksum of a string (character array)

Write a function that will calculate the checksum of an input string. The function prototype must be as follows:

```
void generateChecksum(char *checksum, char *message);
```

The memory for the `checksum` string must be allocated before calling this function. The function must perform the following steps in order to calculate the checksum:

```
initialize sum to zero (type unsigned int)
remove white space from message
for each character in the message with no white space
        value = 65521 * (ASCII value of character)
        sum = (sum + value) modulus 65535
convert sum to string containing the 4 digits of the equivalent hexadecimal number
```

Note that the white space is first removed from the message before its checksum is calculated. This is necessary, because the white space in a JSON object may be different when generated on different systems using different libraries. Hint: Use the `sprintf` function with the `"%x"` format specifier to get the hexadecimal representation.

4.4.3 Generate an encryption key

Write a function to generate an encryption key from a given sessionID, username and password. The sessionID, username and password must all be input parameters of the function and given as strings. The memory for the `key` string must be allocated before calling this function. The function prototype must be as follows:

```
void generateEncryptionKey(char *key, char *sessionID, char *username,
        char *password);
```

The function must perform the following steps in order to generate the encryption key:

```
initialize sum to zero (type unsigned long long)
concatenate string inputs in order: sessionID, username, password
for each character in the concatenated string
        value = 18014398509481984 * (ASCII value of character)
        sum = (sum + value) modulus 9146744073709551614
convert sum to string containing the 16 digits of the equivalent hexadecimal number
```

Hint: 8 Bytes are needed to store the value and sum. You can use the `unsigned long long` type for this. You can use the `sprintf` function with the `"%I64x"` format specifier to get the hexadecimal representation.

4.4.4 Encrypt a message

Write a function to encrypt a message given as a string input using the given encryption key. The memory for the `encryptedMessage` string must be allocated before calling this function. The function should output the encrypted message as a string to `encryptedMessage` in the argument list. The function prototype must be as follows:

```
void encryptMessage(char *encryptedMessage, char *message, char *encryptionKey);
```

The function must take the encryption key and convert each of its characters, which represent hexadecimal digits, to their equivalent decimal values as integers. This conversion can be done using the function in Section 3.4.7. The message must then be encrypted by adding the first of these integer values to the ASCII value of the first character in the message, and the second of the integer values to the second character in the message, and so on, and start again with the first integer value after every 16. This will be necessary if the message is longer than the encryption key, which will usually be the case.

4.4.5 Decrypt an encrypted message

Write a function to decrypt a previously encrypted message given as a string input using the given encryption key. The memory for the `decryptedMessage` string must be allocated before calling this function. The function should output the decrypted message as a string to `decryptedMessage` in the argument list. The function prototype must be as follows:

```
void decryptMessage(char *decryptedMessage, char *message, char *encryptionKey);
```

The decryption of a message is similar to its encryption – the only difference is that instead of adding the integer values of the encryption key they are subtracted.

4.4.6 Verification

To verify that the functions that you have written are correct you must run the program executable that has been compiled. The program will run a number of tests and indicate whether each test has 'PASSED' or 'FAILED'. All the tests need to have passed before you continue with the next part of the project. A successful verification of all the tests is shown below:

```
========================================================================
Testing JSON Utilities
========================================================================
Creating JSON Value                                             : PASSED
Setting Key-Value pair of JSON Object                           : PASSED
Reading value for specific key from JSON Object                 : PASSED
Get string representation of JSON object                        : PASSED
Getting length of array                                         : PASSED
Removing whitespace from string                                 : PASSED
Converting hex to decimal                                       : PASSED

========================================================================
Testing Encryption Utilities
========================================================================
Appending event description to log                              : PASSED
Generating hash                                                 : PASSED
Generating encryptionKey from sessionID, username and password  : PASSED
Encrypting message                                              : PASSED
Decrypting message                                              : PASSED
```

5 Part 3: HTTP communication

5.1 Introduction

For parts 1 and 2 you implemented a collection of functions that can perform various actions that are needed by your program. In this part of the project you will be putting these functions together to perform high level tasks, like getting authorisation from the server, sending and receiving other requests and responses to and from the server, etc.

5.2 Goals

In this part of the project you will write code to performing the following:

- Generate the content of an authorisation request sent to the server
- Process the response received from the server after sending the authorisation request
- Generate the content of the challenge request sent to the server
- Process the response received from the server after sending the challenge request
- Generate the content of the ’Why’ command request sent to the server
- Process the response received from the server after sending the ’Why’ command request
- Display a menu to the user showing the ’Login’, ’Why’ and ’Quit’ options
- Implement the functionality of ’Login’ menu option
- Implement the functionality of the ’Why’ menu option
- Implement the functionality of the ’Quit’ menu option
- Log all request/response transactions between the client and server to a log file

5.3 Background

5.3.1 HTTP Communication

As described earlier, the client-server computing model is used for this project, where the client is your PC and the server is another PC somewhere else. The communication between the client and the server will be done using the HTTP request-response protocol, and more specifically, using the HTTP POST method. A function for performing the POST requests has been implemented and is provided with the following prototype:

```c
    void postRequest(JSON_Value **responseValue, const char *host,
                     int port, const char *path, const char *content);
```

The output of the postRequest function is the *responseValue pointer to a JSON_Value containing the response object from the server. The memory for `responseValue` is allocated inside the function, and must be freed outside the function once it is no longer needed (see Section 3.3.1). The input parameters are the hostname (for ex. an I.P address), port number (typically 80), resource path (either /auth for the authorisation requests or /command for other requests), and the content of the HTTP request as a string. The contents of the HTTP request will be generated by using functions that return JSON values containing objects, and then getting their string representations.

5.3.2 Base64 Encoding and Decoding

When the encryption algorithm of Part 2 of this project is applied to a string, the result is binary data. This is problematic for data structures like JSON objects, but the issue can be resolved by taking an additional step before storing an encrypted string in a JSON object, and then also before decrypting a string retrieved from a JSON object. For storing encrypted data in a JSON object, the additional step is to apply base64 encoding to the data after it has been encrypted, but before storing it in the JSON object. The base64 encoding converts the binary data that results from the encryption into ascii characters. Similarly, for decrypting data from a JSON object, the additional step is then to first apply base64 decoding after retrieving the data from the JSON object, but before decrypting it using your decryption algorithm. The base64 decoding converts the ascii characters back to the binary data that resulted from the initial encryption and can then be decrypted.

Functions that perform the base64 encoding and decoding have been implemented and are available with the following prototypes:

```
    void base64EncodeString(char *encodedString, const char *string);
    void base64DecodeString(char *decodedString, const char *string);
```

Both of these functions require you to first allocate memory for the encoded and decoded strings before they are called.

5.3.3 HTTP Session Authorisation Sequence

Before the server will send any of its encrypted messages, the client first needs to get authorisation. The server has a username and password for each user that may request authorisation. In order to get authorisation, the client needs to request a new session ID and then send a specially formatted message using this session ID, the login details of the user, and a secret message only known to the client and the server (and not to anyone listening to your communications). The table below describes the interaction between the client and server that is needed for authorisation in more detail:

HTTP Session Authorisation Sequence

Client	Server
1) Requests authorisation by sending a username to the server using HTTP POST (user enters username US number)	
	2) Generates session ID (random) 3) Associates session ID with username in database 4) Responds to request with session ID in JSON (no encryption applied)
5) Generates encryption key using session ID, username and password (user enters password) 6) Calculates a checksum of the secret message 7) Encrypts the secret message using the encryption key 8) Requests verification by sending session ID, encrypted message, and checksum to server using HTTP POST	
	9) Finds session using session ID 10) Attempts to decrypt message using encryption key for this user 11) Calculates checksum of the decrypted message 12) Compares calculated checksum to the one received from the client 13) Compares decrypted message to the predetermined message 14) Discards session ID if failure 15) Responds with message in JSON (no encryption applied)
16) Displays success/failure message to user	

If this sequence is successfully completed, the server will now respond to other requests, such as the 'Why' command request (and the optional 'sendMessage' and 'getMessage' requests).

5.3.4 HTTP Command Request Sequence

The table below illustrates the interaction between the client and server for command requests (for ex. the 'Why' command):

HTTP Command Request Sequence

Client	Server
1) Performs request by sending the session ID and an encrypted message (the message contains the command and parameters)	
	2) Finds session using the session ID 3) Decrypts the message containing the command and its parameters 4) Executes command and generates a response message 5) Encrypts the response message using this user's encryption key for this session 6) Sends response in JSON
7) Decrypts response 8) Displays response message to user	

5.3.5 Authorisation Request Format

The client sends an authorisation request to the server to start the session using the `postRequest` function with the URL "*host*/auth". The contents of the HTTP request must be the string representation of a JSON object that is in the format shown below. The value of the "sessionID" field should be "unauthed" and the value of the message field (`<username>`) should be the user name (US student number) of the user. Values specified between "<" and ">" indicate values that will be specific to each user and session.

Authorisation Request:

```
{ "sessionID": "unauthed", "message": "<username>"  }
```

5.3.6 Challenge Request Format

After a response to an authorisation request was received from the server, the client must send a challenge request to the server using the `postRequest` function with the URL "*host*/auth". The contents of the HTTP request must be the string representation of a JSON object that is in the format shown below. The value of the "sessionID" field should now be the sessionID received from the server in its response to the authorisation request. The value of the message field (`<message>`) should be the secret message that has been encrypted (see Section 5.3.2). The value of the checksum field (`<checksum>`) should be the checksum of the unencrypted message.

Challenge Request:

```
{ "sessionID": "<sessionID>", "message": "<encrypted message>", "checksum": "<checksum>" }
```

5.3.7 Command Request Format

The requests generated by the client must be in the format shown below. The encrypted message in the message field must be sent using the `postRequest` function with the URL *host*/command. The encrypted message field (`<encrypted command message>`) should contain the encrypted string representation of the JSON object in the format specific to the command (see Section 5.3.8 for the 'Why' command). The value of the checksum field (`<checksum>`) should be the checksum of the string representation of the command before it is encrypted (see Section 5.3.2).

```
{ "sessionID": "<sessionID>", "message": "<encrypted command message>", "checksum": "<checksum>" }
```

5.3.8 'Why' Command Format

The text as shown below should be encrypted and put in the message field of the command request sent to the server. The server will then decrypt the command message and try to create a JSON object. If successful, the server can identify the requested command and its parameters, execute the command and send a response back to the client. The format for the 'Why' command is shown below:

```
{ "command": "why" }
```

5.3.9 Server JSON Response Format

The responses generated by the server is in the format shown below. Some of the result fields are encrypted (they are indicated). The server will respond with a Failed Command Response with the result being an unencrypted string. Depending on the request, the format of the response object will differ slightly. The response formats corresponding to each request type are shown below:

(a) **Authorisation Request Response:**

```
{ "sessionID": "<sessionID>"}
```

(b) **Successful Challenge Response:**

```
{ "sessionID": "<sessionID>", "result": "success" }
```

(c) **Failed Challenge Response:**

```
{ "sessionID": "<sessionID>", "result": "failure" }
```

(d) **Successful Why Command Response:** (Note <answer> is encrypted)

```
{ "sessionID": "<sessionID>", "result": "<answer>" }
```

(e) **Failed Command Response:**

```
{ "sessionID": "<sessionID>", "result": "failure" }
```

5.4 Instructions

You will write a number of functions to implement the goals of this part of the project. A test program has been provided that must be used to test some of the written functions. Add a call to the `testHttpUtils` function below the call to `testEncryptionUtils` in the `main` function of your `client.c` file, so that the tests for part 3 will also be performed. The functions that involve sending HTTP requests will require manual testing. You can test your program by trying to connect to the server at `ml.sun.ac.za` with a port number of `8888`.

5.4.1 Generate the content of an authorisation request

Write a function to generate the content of the authorisation request that will be sent to the server. The content is specified as a JSON_Value that must adhere to the specified format for authorisation requests in Section 5.3.5. The function prototype must be as follows:

```c
void generateAuthorisationContent(JSON_Value *content, const char *username);
```

5.4.2 Process the response received from the server after sending the authorisation request

Write a function that will process the response received from the server after the authorisation request was sent to the server. The response is obtained from the postRequest function as a JSON_Value and adheres to the format for authorisation responses in Section 5.3.9 (a). The function prototype must be as follows:

```
void processAuthorisationResponse(char *sessionID,
                const JSON_Value *responseValue);
```

5.4.3 Generate the content of the challenge request

Write a function to generate the content of the challenge request that will be sent to the server. The content is specified as a JSON_Value that must adhere the specified format for challenge requests in Section 5.3.6. The function prototype must be as follows:

```
void generateChallengeContent(JSON_Value* content, const char *sessionID,
                const char *message, const char* encryptionKey);
```

The secret message is "A long time ago in a galaxy far, far away".

5.4.4 Process the response received from the server after sending the challenge request

Write a function that will process the response received from the server after sending the challenge request to the server. The response is obtained from the postRequest function as a JSON_Value and adheres to the format for Challenge Responses in Section 5.3.9 (b) and (c). The function must return an integer value of 1 when authorisation was successful and a 0 if authorisation was unsuccessful. The function prototype must be as follows:

```
int processChallengeResponse(const JSON_Value *responseValue);
```

5.4.5 Generate the content of the 'Why' command request

Write a function to generate the content of the command request that must be sent to the server for the 'Why' command. The content is specified as a JSON_Value that must adhere the specified format for command requests and the 'Why' command in Sections 5.3.7 and 5.3.8. The function prototype must be as follows:

```
void generateWhyCommandContent(JSON_Value *content, const char* sessionID,
        const char* encryptionKey);
```

5.4.6 Process the response received from the server after sending the 'Why' command request

Write a function that will process the response received from the server after sending the 'Why' command request to the server. The response is obtained from the postRequest function as a JSON_Value and adheres to the format for command responses in Section 5.3.9 (d) and (e). The function prototype must be as follows:

```
void processWhyCommandResponse(char *answer, const JSON_Value *responseValue,
        const char *encryptionKey);
```

5.4.7 Verification

To verify that the functions that you have written are correct you must run the program executable that has been compiled. The program will run a number of tests and indicate whether each test has 'PASSED' or 'FAILED'. All the tests need to have passed before you continue with the next part of the project. A successful verification of all the tests is shown below:

```
========================================================================
Testing JSON Utilities
========================================================================
Creating JSON Value                                                : PASSED
Setting Key-Value pair of JSON Object                              : PASSED
Reading value for specific key from JSON Object                    : PASSED
Get string representation of JSON object                           : PASSED
Getting length of array                                            : PASSED
Removing whitespace from string                                    : PASSED
Converting hex to decimal                                          : PASSED

========================================================================
Testing Encryption Utilities
========================================================================
Appending event description to log                                 : PASSED
Generating hash                                                    : PASSED
Generating encryptionKey from sessionID, username and password     : PASSED
Encrypting message                                                 : PASSED
Decrypting message                                                 : PASSED

========================================================================
Testing Http Utilities
========================================================================
Generating Authorisation Request Content                           : PASSED
Processing Authorisation Response                                  : PASSED
Generating Challenge Request Content                               : PASSED
Processing Challenge Response                                      : PASSED
Generating Why Command Content                                     : PASSED
Processing Why Command Response                                    : PASSED
```

Note that the program will still run the tests from the previous parts of the project, indicating whether those functions are still working correctly or not.

5.4.8 Display a menu in the console showing the 'Login', 'Why', and 'Quit' options

Extend the main function to display a menu to the user. The menu should display the following three options:

```
====================
=== Menu Options ===
====================
1: Login
2: Why command
Q: Quit

Please input selection:
```

The user must input a character to choose one of the menu options. A character of '1' should be input to select the 'Login' menu option. A character of '2' should be input to select the 'Why' menu option. A character of ’Q’ or ’q’ should be input to select the ’Quit’ menu option. Invalid input should result in the menu being displayed again and the user again being prompted to make a selection. Verification of this function will need to be done manually.

5.4.9 Implement the functionality of ’Login’ menu option

Write a function that will implement the functionality of the ’Login’ menu option. The function should ask the user to input his/her user name and password. The function should then start the authentication session by sending an Authorisation request to the server and processing the received responses. The login function must return whether login has been successful or not. If the login was successful, the function should display the message ’Login: Success!’ and return an integer value of 1. If the login was unsuccessful for any reason, the function should display the message ’Login: Failed!’ and return an integer value of 0. It is suggested that the function prototype for the login function be the following:

```
    int login(char* sessionID, char* encryptionKey);
```

Note that the memory for `sessionID` and `encryptionKey` must be allocated before this function is called. Hint: Use the functions in Sections 5.4.1-5.4.4. Verification of this function will need to be done manually.

5.4.10 Implement the functionality of ’Why’ menu option

Write a function that will implement the functionality of the ’Why’ menu option. If the user has not been authorised, the function should display the message ’User has not yet been authenticated. Please login first!’. The function should display the result of the ’Why’ command, which if successful will be a random message generated by the server. If the ’Why’ command is unsuccessful for any reason, the function should display the message ’Command Failed!’. It is suggested that the function prototype for the login function be the following:

```
    void why(const char* sessionID, const char* encryptionKey);
```

Hint: Use the functions in Sections 5.4.5 and 5.4.6. Verification of this function will need to be done manually.

5.4.11 Implement the functionality of ’Quit’ menu option

Write a function that will implement the functionality of the ’Quit’ menu option. The function must exit the program. Verification of this function will need to be done manually.

5.4.12 Log all request/response transactions between the client and server to a logfile

Use the function in Section 4.4.1 to keep a log of all the requests and responses that are sent and received. Verification of this function will need to be done manually.

6 Optional Part: Additional Server Commands

6.1 Goals

This part of the project is optional. In this part you will write code to perform the following:

- Expand the menu to include the different commands that can be sent to the server after authentication
- Implement the functionality of the 'Send Message' menu option
- Implement the functionality of the 'Get Message' menu option

6.2 Background

6.2.1 Server Command List

In addition to the 'Why' command, as shown in Section 5.3.8, the server also understands the following commands:

'sendMessage' command

```
{ "command": "sendMessage"," to": "<username>", "content": "<message>" }
```

'getMessage' command:

```
{ "command": "getMessage" }
```

6.2.2 Server JSON Response Format

The responses generated by the server for the additional commands are:

Failed Command Response:

```
{ "sessionID": "<sessionID>", "result": "failure" }
```

Successful Sendmessage Command Result: (Note result is encrypted)

```
{ "sessionID": "<sessionID>", "result": "success" }
```

Successful Why GetMessage Response: (Note result is encrypted)

```
{ "sessionID": "<sessionID>", "result": { "from": "<username>", "content":  "<content>" } }
```

6.3 Instructions

6.3.1 Display a menu to the user showing all the options

Extend the menu that is displayed to the user. The menu should display the following options:

```
====================
=== Menu Options ===
====================
1: Login
2: Why
3: Send Message
4: Get Message
Q: Quit

Please input selection:
```

The user must input a character to choose one of the menu options. A character of '2' should be input to select the 'Why' menu option. A character of '3' should be input to select the 'Send Message' menu option. A character of '4' should be input to select the 'Get Message' menu option. Invalid input should result in the menu being displayed again and the user again being prompted to make a selection.

6.3.2 Implement the functionality of the ’Send Message’ menu option

Write a function that will implement the ’Send Message’ menu option. The function should ask the user to enter the username of the person to whom the message should be sent, followed by the contents of the message that should be sent. The message must be able to contain spaces. If the message is successfully sent, the text ’Message Successfully Sent!’ should be displayed. If the message sending fails for any reason, the text ’Command Failed!’ should be displayed. It is suggested that the function prototype for the ’Send Message’ function be the following:

```
    void sendMessage(const char* sessionID, const char* encryptionKey);
```

6.3.3 Implement the functionality of the ’getMessage’ menu option

Write a function that will implement the ’Get Message’ menu option. The function should display the contents of the message, followed by the username of the person who sent the message on the next line. If getting the command fails for any reason, the text ’Command Failed!’ should be displayed. It is suggested that the function prototype for the ’Get Message’ function be the following:

```
    void getMessage(const char* sessionID, const char* encryptionKey);
```