Desenvolvimento de um modelo base em c++ de servidor http

Relatório de desenvolvimento

LORENA BASSANI[1]

[1]Universidade Federal do Espirito Santo

Resumo

Este relatório se dedica a detalhar o processo de desenvolvimento de um servidor http escrito em c++ para a coleção de códigos base da Equipe de Robótica da UFES.

Palavras-chave: *http, servidor, c++, socket, comunicação*

1. INTRODUÇÃO

Escrever um servidor http em c++ não se mostrou um desafio simples, em especial pela falta de documentação e tutoriais sobre o assunto de forma prática de acessar. A ideia de escrever um servidor http começou com o processo de inciar um novo paradigma de implementação de projetos de robótica complexos na ERUS: programação orientada a serviços. A programação de serviços se baseia em escrever diversos programas que oferecem serviços, ou seja, programas servidores, para que um programa principal que controle o sistema robótico utilize estes serviços. Isso propõe uma série de vantagens, como: facilidade na reutilização de serviços já programados, alta modularização, melhora na utilização dos recursos disponíveis em máquinas com mais poder de computação e possibilidade de utilizar diferentes linguagens para diferentes serviços de forma que toda a comunicação entre eles seja feita através do protocolo http.
Com isso em mente, deixamos disponível [1] uma base simples de um servidor escrito em c++, apenas para deixar um exemplo de como ficaria um programa servidor escrito em c++.
Na seção 2 veremos uma breve explicação dos principais conceitos necessários para implementar este servidor. Já no tópico 3 veremos alguns detalhes interessantes sobre o processo de implementar o servidor http em c++, e concluímos em 4. Após a conclusão você pode ver os códigos utilizados nos Anexos e Apêndices.

2. CONCEITOS E PROTOCOLOS DE COMUNICAÇÃO

Neste tópico teremos uma breve explicação dos conceitos principais necessários para compreender o funcionamento do servidor HTTP que desenvolvemos neste projeto.
No tópico 2.1 veremos alguns conceitos básicos sobre o protocolo HTTP, seguido dos métodos HTTP em 2.1.1 e do cabeçalho em 2.1.2. Após isso veremos sobre Sockets e Modelo Servidor-Cliente nos tópicos 2.2 e 2.3, respectivamente.

2.1. Protocolo HTTP

O protocolo HTTP está em todos os lugares, principalmente na internet. O HTTP é o protocolo de requisições utilizado pelos mais diversos tipos de serviços e sites disponíveis online [2], sendo um protocolo muito interessante como escolha para basear as requisições e respostas de um servidor, pois isso faz com que ele esteja no padrão dos serviços online, obedecendo protocolos muito conhecidos, bem documentados e amplamente implementados nas mais diversas linguagens de programação.
Este protocolo pode ser utilizado como base em arquiteturas de web-services, como REST e RPC. E a melhor parte deste

protocolo é sua simplicidade: seu formato é bem simples e pode ser implementado em qualquer linguagem de proposito geral.
O HTTP foi proposto na RFC 2616 [3], que foi atualizada pelas RFCs a seguir: RFC 7230 [4], RFC 7231 [5], RFC 7232 [6], RFC 7233 [7], RFC 7234 [8] e RFC 7235 [9]. Essas RFCs descrevem as diretrizes de implementação do protocolo HTTP/1.1, que é o que usaremos no nosso servidor.

2.1.1. Métodos HTTP

O protocolo HTTP oferece diversos métodos para trabalharmos nossas mensagens. Estes métodos podem ser utilizados, por exemplo, para implementar operações CRUD em uma API Rest [10]. Alguns desse métodos são:

- GET
- POST
- PUT
- PATCH
- DELETE

No nosso servidor, vamos nos concentrar em dois métodos: GET e POST. Os métodos em si não possuem uma funcionalidade fixa e você pode utilizá-los de forma criativa para os mais amplos tipos de comunicação cliente-servidor através do protocolo HTTP. Vamos nos concentrar nesses dois métodos pois representam dois tipos bem distintos: um com parâmetros e outro, teoricamente, sem.

2.1.2. Cabeçalho HTTP

O cabeçalho HTTP é o que define as vantagens de utilizar este protocolo. Existe uma grande diversidade de palavras-chave e informações que podemos conseguir a partir do cabeçalho que vão nos permitir manipular nossas informações de formas muito eficientes.
Para esse trabalho, vamos nos concentrar em algumas partes específicas dos cabeçalhos de requisição e resposta:

- Requisição
  - primeira linha de cabeçalho: método, url e versão do protocolo.
  - User-Agent: define o tipo de aplicação que realizou a requisição
  - Content-Length: tamanho do conteúdo de dados que iremos trabalhar
  - Content-Type: tipo do conteúdo de dados que iremos trabalhar
- Resposta
  - primeira linha de cabeçalho: versão do protocolo, status e mensagem de status
  - Date: data e horário de envio da resposta
  - Server: nome do nosso servidor
  - Access-Control-Allow-Origin: quem pode receber nossas respostas
  - Content-Length: tamanho do conteúdo da resposta que estamos enviando
  - Content-Type: tipo do conteúdo da resposta que estamos enviando
  - Conection: se a conexão será encerrada ou permanecerá aberta

Essas informações nos permitem realizar uma série de ações de forma muito eficiente como: tratar de formas diferentes requisições de clientes de natureza diferentes (navegadores de um interpretador python, por exemplo); tratar de formas diferentes dados recebidos de acordo com o seu tipo (Json, dados de formulários, páginas html inteiras ou texto puro); ou até controlar quem pode receber as informações que estamos enviando.

2.2. Socket

Sockets são pontos de comunicação entre hosts. Eles identificam um endereço e uma porta de comunicação, e se encontram comunicando aplicação com aplicação (nível acima de comunicação de máquina com máquina por exemplo). As aplicações podem se comunicar mesmo em máquinas diferentes, através da internet, mas nós vamos nos concentrar em comunicar aplicações de uma mesma máquina através dos sockets.
Quando abrimos um socket, estamos abrindo a comunicação a partir de um endereço (no nosso caso, local) e uma porta. Ao abrirmos o socket do nosso servidor em uma porta, passamos a escutar requisições de nossos serviços que chegarem através dessa porta.
Sockets em c++ são tecnologias muito ligadas ao sistema operacional, utilizando amplamente chamadas de sistema. Nosso servidor foi construído para rodar em sistema linux.

2.3. Modelo Servidor-Cliente

O modelo servidor-cliente é um modelo de computação distribuída, ou seja, onde temos a computação sendo feita em diversas plataformas diferentes. Nele, os servidores oferecem um serviço, que os cliente requisitam e utilizam sua resposta para sua própria computação.
Geralmente servidores e clientes são hosts que se comunicam através de uma rede de computadores, mas no nosso caso serão duas aplicações em um mesmo host que se comunicarão através de uma comunicação local utilizando sockets 2.2 que também são utilizados para redes de computadores.

3. Implementação do Servidor

Neste tópico vamos apresentar alguns detalhes e particularidades da implementação de um servidor HTTP em c++, detalhando as dificuldades encontradas no tópico 3.1 e os resultados obtidos em 3.2. Após isso, temos uma breve reflexão sobre os trabalhos futuros em cima do nosso servidor base no tópico 3.3.

3.1. Dificuldades

A implementação do servidor HTTP se mostrou muito desafiadora por utilizar tantas chamadas de sistema, mesmo que todas encapsuladas na classe Socket. As chamadas de sistema se tornam uma parte sensível do código e suscetíveis a diversos erros.
Outro problema encontrado foi a diferença entre mensagens enviadas de fontes diferentes. Neste trabalho, testamos quatro fontes: três navegadores e um interpretador python. Enquanto dois dos navegadores enviavam suas mensagens de forma que elas chegavam inteiras, um dos navegadores e o interpretador python enviavam de uma forma que esta chegava dividida em cabeçalho e dados. O tratamento dessas questões leva a apreciar ainda mais a riqueza de informações que podemos ter no cabeçalho para que possamos tratar este tipo de diferença, mas ainda faz pensar sobre como é complicado debugar essas mensagens e ainda mais em identificar quais fontes causam quais erros.

3.2. Testes e Resultados

Para o nosso servidor, resolvemos atender as requisições respondendo páginas html, dessa forma poderíamos testar facilmente através dos navegadores.
Os testes do método GET são bem simples: basta digitar o endereço para qual a requisição será enviada no navegador (http://localhost:porta/endpoint) e a página de resposta é exibida. Mas, para realizar os testes POST, resolvemos criar uma página de testes. A página de testes possui interface intuitiva, de forma que ela te mostra os endpoint (/hello e /bye) e as requisições possíveis (POST e GET), te permite selecionar a porta e ainda enviar uma mensagem através do método POST. Ela também mostra o resultado em um campo de resposta, de forma que não é necessário retornar a ela após a resposta, já podendo realizar um novo teste após o anterior obter resposta. Aproveitamos ainda a oportunidade para deixar um breve resumo das informações mais importantes nesta mesma página, tornando-a bem intuitiva e agradável para utilização. Na figura 1 é possível ver um exemplo do uso da página de testes.

Figura 1: *Página de testes do servidor*

Os resultados foram satisfatórios. As requisições das quatro fontes testadas foram satisfeitas em todos os testes, mesmo que uma por vez, o que configura um bom servidor para ser utilizado na implementação de sistemas de robótica da ERUS que rodem em plataformas com certo nível poder computacional ou com acesso a rede, como o Very Small Size Soccer ou o robô do Desafio Open.
O servidor se mostrou muito fácil de lidar com as mensagens de diferentes tipos também, como JSON e respostas de formulários, podendo ser útil para implementar certas aplicações utilizadas pela organização da equipe.

3.3. Tabalhos Futuros

Para um trabalho futuro, a equipe gostaria de trazer o servidor para dentro de uma classe, limpando o código do arquivo main das preocupações com as rotinas de serviço e se preocupando apenas com a montagem da resposta. Outro desejo do time é fazer um controle de serviços que permita que mais de um cliente seja atendido paralelamente, dessa forma poderíamos aproveitar ainda mais de poder computacional para realizarmos diversos serviços em um mesmo servidor.

4. Conclusão

O Protocolo HTTP se mostra um protocolo rico em recursos que podemos utilizar para trabalharmos com robótica, oferecendo serviços e programando robôs de forma distribuída. Com este código, a equipe espera guardar uma base simples e exemplo de como programar um servidor em c++, uma das principais linguagens empregadas na robótica, e deixar disponível para as gerações futuras de membros da equipe e para os alunos que tiverem interesse em aprender mais sobre o assunto.

Anexos

A. Código da Classe Socket

Listing 1: *Cabeçalho da Classe Socket escrito no arquivo Socket.h*

```
1  /* http cpp server base
2   * Base de um servidor http simples em c++ para escrever
3   * serviços para projetos de robótica.
4   * Definition of the Socket class.
5   *
6   * @author : Rob Tougher (mail to: rtougher@yahoo.com)
7   * @source : http://tldp.org/LDP/LG/issue74/tougher.html
8   * @date : 2002
9   */
10
11 #ifndef Socket_class
12 #define Socket_class
13
14
15 #include <sys/types.h>
16 #include <sys/socket.h>
17 #include <netinet/in.h>
18 #include <netdb.h>
19 #include <unistd.h>
20 #include <string>
21 #include <arpa/inet.h>
22
23
24 const int MAXHOSTNAME = 200;
25 const int MAXCONNECTIONS = 5;
26 const int MAXRECV = 500;
27
28 class Socket
29 {
30  public:
31   Socket();
32   virtual ~Socket();
33
34   // Server initialization
35   bool create();
36   bool bind ( const int port );
37   bool listen() const;
38   bool accept ( Socket& ) const;
39
40   // Client initialization
41   bool connect ( const std::string host, const int port );
42
43   // Data Transimission
44   bool send ( const std::string ) const;
45   int recv ( std::string& ) const;
46
47
```

```
48    void set_non_blocking ( const bool );
49
50    bool is_valid() const { return m_sock != -1; }
51
52   private:
53
54    int m_sock;
55    sockaddr_in m_addr;
56
57
58  };
59
60
61  #endif
```

Listing 2: *Implementação da Classe Socket escrito no arquivo Socket.cpp*

```
1   /* http cpp server base
2    * Base de um servidor http simples em c++ para escrever
3    * serviços para projetos de robótica.
4    * Implementation of the Socket class.
5    *
6    * @author : Rob Tougher (mail to: rtougher@yahoo.com)
7    * @source : http://tldp.org/LDP/LG/issue74/tougher.html
8    * @date : 2002
9    */
10
11
12  #include "Socket.h"
13  #include "string.h"
14  #include <string.h>
15  #include <errno.h>
16  #include <fcntl.h>
17  #include <iostream>
18
19
20
21  Socket::Socket() :
22    m_sock ( -1 )
23  {
24
25    memset ( &m_addr,
26             0,
27             sizeof ( m_addr ) );
28
29  }
30
31  Socket::~Socket()
32  {
33    if ( is_valid() )
34      ::close ( m_sock );
35  }
36
37  bool Socket::create()
```

```
38 {
39   m_sock = socket ( AF_INET,
40                     SOCK_STREAM,
41                     0 );
42
43   if ( ! is_valid() )
44     return false;
45
46
47   // TIME_WAIT    argh
48   int on = 1;
49   if ( setsockopt ( m_sock, SOL_SOCKET, SO_REUSEADDR, ( const char* ) &on, sizeof ( on ) ) == -1 )
50     return false;
51
52
53   return true;
54
55 }
56
57
58
59 bool Socket::bind ( const int port )
60 {
61
62   if ( ! is_valid() )
63     {
64       return false;
65     }
66
67
68
69   m_addr.sin_family = AF_INET;
70   m_addr.sin_addr.s_addr = INADDR_ANY;
71   m_addr.sin_port = htons ( port );
72
73   int bind_return = ::bind ( m_sock,
74                              ( struct sockaddr * ) &m_addr,
75                              sizeof ( m_addr ) );
76
77
78   if ( bind_return == -1 )
79     {
80       return false;
81     }
82
83   return true;
84 }
85
86
87 bool Socket::listen() const
88 {
89   if ( ! is_valid() )
90     {
```

```
91        return false;
92      }
93
94    int listen_return = ::listen ( m_sock, MAXCONNECTIONS );
95
96
97    if ( listen_return == -1 )
98      {
99        return false;
100     }
101
102   return true;
103 }
104
105
106 bool Socket::accept ( Socket& new_socket ) const
107 {
108   int addr_length = sizeof ( m_addr );
109   new_socket.m_sock = ::accept ( m_sock, ( sockaddr * ) &m_addr, ( socklen_t * ) &addr_length );
110
111   if ( new_socket.m_sock <= 0 ){
112     std::cout << strerror(errno) << " " << errno << std::endl;
113     return false;
114   }else
115     return true;
116 }
117
118
119 bool Socket::send ( const std::string s ) const
120 {
121   int status = ::send ( m_sock, s.c_str(), s.size(), MSG_NOSIGNAL );
122   if ( status == -1 )
123     {
124       return false;
125     }
126   else
127     {
128       return true;
129     }
130 }
131
132
133 int Socket::recv ( std::string& s ) const
134 {
135   char buf [ MAXRECV + 1 ];
136
137   s = "";
138
139   memset ( buf, 0, MAXRECV + 1 );
140
141   int status = ::recv ( m_sock, buf, MAXRECV, 0 );
142
143   if ( status == -1 )
```

```
144       {
145         std::cout << "status == 1   errno == " << errno << "  in Socket::recv\n";
146         return 0;
147       }
148     else if ( status == 0 )
149       {
150         return 0;
151       }
152     else
153       {
154         s = buf;
155         return status;
156       }
157   }
158
159
160
161   bool Socket::connect ( const std::string host, const int port )
162   {
163     if ( ! is_valid() ) return false;
164
165     m_addr.sin_family = AF_INET;
166     m_addr.sin_port = htons ( port );
167
168     int status = inet_pton ( AF_INET, host.c_str(), &m_addr.sin_addr );
169
170     if ( errno == EAFNOSUPPORT ) return false;
171
172     status = ::connect ( m_sock, ( sockaddr * ) &m_addr, sizeof ( m_addr ) );
173
174     if ( status == 0 )
175       return true;
176     else
177       return false;
178   }
179
180   void Socket::set_non_blocking ( const bool b )
181   {
182
183     int opts;
184
185     opts = fcntl ( m_sock,
186                    F_GETFL );
187
188     if ( opts < 0 )
189       {
190         return;
191       }
192
193     if ( b )
194       opts = ( opts | O_NONBLOCK );
195     else
196       opts = ( opts & ~O_NONBLOCK );
```

```
197
198     fcntl ( m_sock,
199             F_SETFL, opts );
200
201 }
```

B. Código da classe ServerSocket

Listing 3: *Cabeçalho da Classe ServerSocket escrito no arquivo ServerSocket.h*

```
1  /* http cpp server base
2   * Base de um servidor http simples em c++ para escrever
3   * serviços para projetos de robótica.
4   * Definition of the ServerSocket class
5   *
6   * @author : Rob Tougher (mail to: rtougher@yahoo.com)
7   * @source : http://tldp.org/LDP/LG/issue74/tougher.html
8   * @date : 2002
9   */
10
11 #ifndef ServerSocket_class
12 #define ServerSocket_class
13
14 #include "Socket.h"
15
16
17 class ServerSocket : private Socket
18 {
19  public:
20
21    ServerSocket ( int port );
22    ServerSocket (){};
23    virtual ~ServerSocket();
24
25    const ServerSocket& operator << ( const std::string& ) const;
26    const ServerSocket& operator >> ( std::string& ) const;
27
28    void accept ( ServerSocket& );
29
30 };
31
32
33 #endif
```

Listing 4: *Implementação da Classe ServerSocket escrito no arquivo ServerSocket.cpp*

```
1  /* http cpp server base
2   * Base de um servidor http simples em c++ para escrever
3   * serviços para projetos de robótica.
4   * Implementation of the ServerSocket class
5   *
6   * @author : Rob Tougher (mail to: rtougher@yahoo.com)
7   * @source : http://tldp.org/LDP/LG/issue74/tougher.html
8   * @date : 2002
9   */
```

```
10
11 #include "ServerSocket.h"
12 #include "SocketException.h"
13
14
15 ServerSocket::ServerSocket ( int port )
16 {
17   if ( ! Socket::create() )
18     {
19       throw SocketException ( "Could␣not␣create␣server␣socket." );
20     }
21
22   if ( ! Socket::bind ( port ) )
23     {
24       throw SocketException ( "Could␣not␣bind␣to␣port." );
25     }
26
27   if ( ! Socket::listen() )
28     {
29       throw SocketException ( "Could␣not␣listen␣to␣socket." );
30     }
31
32 }
33
34 ServerSocket::~ServerSocket()
35 {
36 }
37
38
39 const ServerSocket& ServerSocket::operator << ( const std::string& s ) const
40 {
41   if ( ! Socket::send ( s ) )
42     {
43       throw SocketException ( "Could␣not␣write␣to␣socket." );
44     }
45
46   return *this;
47
48 }
49
50
51 const ServerSocket& ServerSocket::operator >> ( std::string& s ) const
52 {
53   if ( ! Socket::recv ( s ) )
54     {
55       throw SocketException ( "Could␣not␣read␣from␣socket." );
56     }
57
58   return *this;
59 }
60
61 void ServerSocket::accept ( ServerSocket& sock )
62 {
```

```
63    if ( ! Socket::accept ( sock ) )
64      {
65        throw SocketException ( "Could not accept socket." );
66      }
67  }
```

C. Código da classe SocketException

Listing 5: *Cabeçalho da Classe SocketException escrito no arquivo SocketException.h*

```
1  /* http cpp server base
2   * Base de um servidor http simples em c++ para escrever
3   * serviços para projetos de robótica.
4   * SocketException class
5   *
6   * @author : Rob Tougher (mail to: rtougher@yahoo.com)
7   * @source : http://tldp.org/LDP/LG/issue74/tougher.html
8   * @date : 2002
9   */
10
11 #ifndef SocketException_class
12 #define SocketException_class
13
14 #include <string>
15
16 class SocketException
17 {
18  public:
19    SocketException ( std::string s ) : m_s ( s ) {};
20    ~SocketException (){};
21
22    std::string description() { return m_s; }
23
24  private:
25
26    std::string m_s;
27
28 };
29
30 #endif
```

Apêndices

D. Código do servidor

Listing 6: *Implementação da lógica do servidor http escrita no arquivo main.cpp*

```
1  /* http cpp server base
2   * Base de um servidor http simples em c++ para escrever
3   * serviços para projetos de robótica.
4   *
5   * @author : Lorena "Ino" Bassani (https://github.com/LBBassani)
6   * @source : https://github.com/LBBassani/http_cpp_server_base
7   * @license : GPLv3
8   * @date : 2020
9   */
10
11 #include <iostream>
12 #include <thread>
13 #include <vector>
14 #include <string>
15 #include <cstring>
16 #include <sstream>
17
18 #include "httpServer/ServerSocket.h"
19 #include "httpServer/SocketException.h"
20
21 std::string constructResponse(std::vector<std::string>, std::string);
22 void runServer(int);
23
24 int main(int argc, char ** argv){
25
26     int port = (argc > 1) ? (atoi(argv[1])) : 30000;
27     runServer(port);
28
29     return 0;
30
31 }
32
33 void runServer(int port){
34
35     try{
36         ServerSocket server(port);
37
38         std::cout << "running..." << std::endl;
39
40         while ( true ){
41             ServerSocket new_sock;
42             server.accept(new_sock);
43
44             try{
45                 while( true ){
46
47                     // Recebe dados pela porta
```

```
48                      std::string header, data;
49                      new_sock >> header;
50
51                      // Quebra o header para ter as informações
52                      std::vector<std::string> headerwords;
53                      std::istringstream iss (header) ;
54                      for(std::string s; iss >> s; ){
55                          headerwords.push_back(s);
56                      }
57
58                      // Se o método for post, patch ou put, recebe corpo
59                      if (!headerwords[0].compare("POST") || !headerwords[0].compare("PUT") || !header
60                          int contentlen = 0;
61                          for (int i = 0; i < (int)headerwords.size(); i++){
62                              if(headerwords[i].find("Content Length:") != std::string::npos ){ conten
63                          }
64                          if (contentlen) {
65                              bool continueReading = false;
66                              for (int i = 0; i < (int)headerwords.size(); i++){
67                                  if(headerwords[i].find("User Agent:") != std::string::npos ){
68                                      for (int j = i + 1; j < (int)headerwords.size(); j++){
69                                          if (headerwords[j].find(":") != std::string::npos ) break;
70                                          if (headerwords[j].find("OPR/") != std::string::npos || head
71                                              continueReading = true;
72                                              break;
73                                          }
74                                      }
75                                      break;
76                                  }
77                              }
78                              if (continueReading) new_sock >> data;
79                              else data = header.substr(header.length()   contentlen);
80
81                          }
82                      }
83
84                      // envia a resposta
85                      std::string response = constructResponse(headerwords, data);
86                      new_sock << response;
87
88                      // Imprime que respondeu
89                      // declaring argument of time()
90                      time_t my_time = time(NULL);
91
92                      // ctime() used to give the present time
93                      std::cout << "Response␣sent␣" << ctime(&my_time);
94
95                  }
96              }catch ( SocketException&) { }
97          }
98
99      }catch ( SocketException& e ){
100         std::cout << "Exception␣was␣caught:" << e.description() << "\nExiting.\n";
```

```
101     }
102
103 }
104
105 std::string constructResponse(std::vector<std::string> header, std::string data){
106
107     std::string response;
108     std::string responseheader("HTTP/1.1 ");
109     // Responde requisição GET
110     if(!header[0].compare("GET")){
111         // Responde saudação
112         if (!header[1].compare("/hello")){
113             response = "<html> <body> <h1>Hello there!</h1> </body> </html>";
114         }
115
116         // Responde despedida
117         else if (!header[1].compare("/bye")){
118             response = "<html> <body> <h1>Bye!</h1> </body> </html>";
119         }
120
121         if (response.length() > 0){
122             responseheader += "200 OK\n";
123         } else{
124             responseheader += "404 Not Found\n";
125         }
126     }
127
128     // Responde requisição POST, PATCH e PUT
129     else if(!header[0].compare("POST") || !header[0].compare("PATCH") || !header[0].compare("PUT")){
130         if (!header[1].compare("/hello")){
131             response = "<html> <body> <h1>Hello there!</h1> <p> Your message was " + data + " and the
132         }
133
134         else if (!header[1].compare("/bye")){
135             response = "<html> <body> <h1>Bye!</h1> <p>Your message was " + data + " and the method
136         }
137
138         if (response.length() > 0){
139             responseheader += "200 OK\n";
140         } else{
141             responseheader += "404 Not Found\n";
142         }
143     }
144
145     // Monta a resposta:
146
147     // Se não passou pelos métodos que tem, manda mensagem de metodo não permitido
148     if (!responseheader.compare("HTTP/1.1 ")){
149         responseheader += "405 Method Not Allowed\n";
150     }
151
152     // Informações de cabeçalho :
153     // Informações de data e hora :
```

```
154         char savedlocale[256];
155         strcpy(savedlocale, setlocale(LC_ALL, NULL));
156         setlocale(LC_ALL, "C");
157         time_t now = time(0);
158         struct tm tm = *gmtime(&now);
159         char buf[30];
160         strftime(buf, sizeof(buf), "%a, %d %b %Y %H:%M:%S %Z", &tm);
161         setlocale(LC_ALL, savedlocale);
162         responseheader += "Date: " + std::string(buf) + "\n";
163 
164         // Informações do servidor :
165         responseheader += "Server: Sample Server @ERUS\n";
166         responseheader += "Access Control Allow Origin: *\n";
167 
168         // Informações do conteúdo :
169         responseheader += "Content Type: text/html; charset=UTF 8\n";
170         responseheader += "Content Length: " + std::to_string(response.length()) + "\n";
171         responseheader += "Connection: close\n";
172 
173         // Retorna a mensagem :
174         return (responseheader + "\n" + response);
175 
176 };
```

REFERÊNCIAS

[1] Lorena Bassani. Repositório do projeto http-cpp-server-base, 2020. Disponível em: https://github.com/LBBassani/http-cpp-server-base – acesso em 14 jul. 2020.

[2] Skrew Everything. Http server: Everything you need to know to build a simple http server from scratch, 2018. Disponível em: https://medium.com/from-the-scratch/http-server-what-do-you-need-to-know-to-build-a-simple-http-server-from-scratch-d1ef8945e4fa – acesso em 14 jul. 2020.

[3] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee. Rfc 2616. Disponível em: https://www.rfc-editor.org/info/rfc2616 – acesso em 15 jul. 2020.

[4] R. Fielding, Ed., J. Reschke, Ed. Rfc 7230. Disponível em: https://www.rfc-editor.org/info/rfc7230 – acesso em 15 jul. 2020.

[5] R. Fielding, Ed., J. Reschke, Ed. Rfc 7231. Disponível em: https://www.rfc-editor.org/info/rfc7231 – acesso em 15 jul. 2020.

[6] R. Fielding, Ed., J. Reschke, Ed. Rfc 7232. Disponível em: https://www.rfc-editor.org/info/rfc7232 – acesso em 15 jul. 2020.

[7] R. Fielding, Ed., Y. Lafon, Ed., J. Reschke, Ed. Rfc 7233. Disponível em: https://www.rfc-editor.org/info/rfc7233 – acesso em 15 jul. 2020.

[8] R. Fielding, Ed., M. Nottingham, Ed., J. Reschke, Ed. Rfc 7234. Disponível em: https://www.rfc-editor.org/info/rfc7234 – acesso em 15 jul. 2020.

[9] R. Fielding, Ed., J. Reschke, Ed. Rfc 7235. Disponível em: https://www.rfc-editor.org/info/rfc7235 – acesso em 15 jul. 2020.

[10] RestApiTutorial.com. Using http methods for restful services. Disponível em: https://www.restapitutorial.com/lessons/httpmethods.html – acesso em 14 jul. 2020.