Липецкий государственный технический университет

Факультет автоматизации и информатики

Кафедра автоматизированных систем управления

ЛАБОРАТОРНАЯ РАБОТА №5

по дисциплине «Компьютерные сети»

Прикладные протоколы стека TCP/IP

Вариант 19

Студент

Группа АС-19 Цыганов Н.А.

Стюфляев А.Р.

.

Руководитель Самсонов А.Н.

Липецк 2022 г.

Цель работы

Изучить прикладные протоколы стека TCP/IP, принципы реализации сетевого взаимодействия приложений с использованием сокетов.

Задание кафедры

Реализовать клиентское приложение заданной сетевой службы на основе сокетов. Приложение должно выполнять заданные функции сетевой службы.

Протокол HTTP - Hyper Text Transfer Protocol (RFC 7231), реализовать синхронизацию времени с сервером (получение времени от сервера).

Ход работы

1 Теоретическая информация

Основная информация

HTTP — широко распространённый протокол передачи данных, изначально предназначенный для передачи гипертекстовых документов (то есть документов, которые могут содержать ссылки, позволяющие организовать переход к другим документам).

Протокол HTTP предполагает использование клиент-серверной структуры передачи данных. Клиентское приложение формирует запрос и отправляет его на сервер, после чего серверное программное обеспечение обрабатывает данный запрос, формирует ответ и передаёт его обратно клиенту. После этого клиентское приложение может продолжить отправлять другие запросы, которые будут обработаны аналогичным образом.

Задача, которая традиционно решается с помощью протокола HTTP — обмен данными между пользовательским приложением, осуществляющим доступ к веб-ресурсам (обычно это веб-браузер) и веб-сервером. На данный момент именно благодаря протоколу HTTP обеспечивается работа Всемирной паутины.

Также HTTP часто используется как протокол передачи информации для других протоколов прикладного уровня, таких как SOAP, XML-RPC и WebDAV. В таком случае говорят, что протокол HTTP используется как «транспорт».

API многих программных продуктов также подразумевает использование HTTP для передачи данных — сами данные при этом могут иметь любой формат, например, XML или JSON.

Как правило, передача данных по протоколу HTTP осуществляется через TCP/IP-соединения. Серверное программное обеспечение при этом обычно использует TCP-порт 80 (и, если порт не указан явно, то обычно клиентское программное обеспечение по умолчанию использует именно 80-й порт для открываемых HTTP-соединений), хотя может использовать и любой другой.

Безопасность

Сам по себе протокол HTTP не предполагает использование шифрования для передачи информации. Тем не менее, для HTTP есть распространённое расширение, которое реализует упаковку передаваемых данных в криптографический протокол SSL или TLS.

Название этого расширения — HTTPS (HyperText Transfer Protocol Secure). Для HTTPS-соединений обычно используется TCP-порт 443. HTTPS широко используется для защиты информации от перехвата, а также, как правило, обеспечивает защиту от атак вида man-in-the-middle — в том случае, если сертификат проверяется на клиенте, и при этом приватный ключ сертификата не был скомпрометирован, пользователь не подтверждал использование неподписанного сертификата, и на компьютере пользователя не были внедрены сертификаты центра сертификации злоумышленника.

На данный момент HTTPS поддерживается всеми популярными веб-браузерами.

Структура запроса HTTP

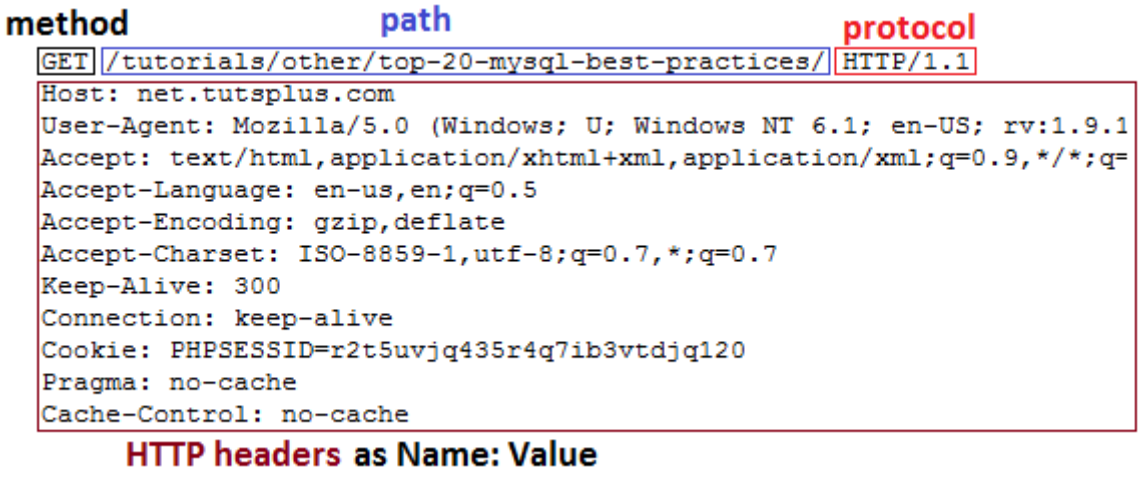

Рисунок 1 – Запрос HTTP

Первая строка HTTP-запроса называется линией запроса и состоит из трёх частей:

- “method” указывает, какой это запрос. Наиболее распространённые методы GET, POST и HEAD.

GET: получение документа. Это основной метод, используемый для извлечения html, изображений, JavaScript, CSS и т. д. С использованием этого метода запрошено большинство данных, загружаемых в браузер.

POST: отправка данных на сервер. Запросы POST чаще всего отправляются веб-формами.

HEAD: получение информации заголовка. HEAD идентичен GET, за исключением того, что сервер не возвращает содержимое HTTP-ответа. Когда вы отправляете запрос HEAD, это означает, что вас интересуют только код ответа и HTTP headers, а не сам документ.

- "path", как правило, является частью URL-адреса, который идёт после host (домена).
- Часть "protocol" содержит "HTTP" и версию, которая обычно 1.1 в современных браузерах.

Остальная часть запроса содержит HTTP headers как пары "Name: Value" в каждой строке. Они содержат различную информацию о HTTP-запросе и вашем браузере. Например, строка "User-Agent" предоставляет информацию о версии браузера и операционной системе, которую вы используете. “Accept-Encoding” сообщает серверу, может ли браузер принимать сжатый output, например, gzip.

Структура ответа HTTP

После того, как браузер отправляет HTTP-запрос, сервер отвечает HTTP-ответом.

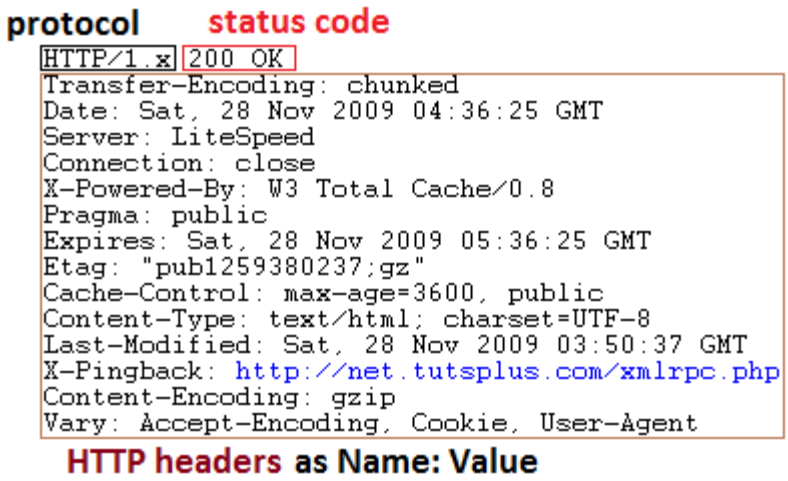

Рисунок 2 – Ответ HTTP

Первой порцией данных является протокол. Обычно это снова HTTP/1.x или HTTP/1.1 на современных серверах.

Следующая часть - это код состояния, за которым следует короткое сообщение. Код 200 означает, что наш запрос GET был успешным и сервер вернёт содержимое запрошенного документа сразу после headers.

Код 404. Это число фактически приходит из части кода состояния HTTP-ответа. Если запрос GET будет создан для path, который сервер не может найти, он ответил бы 404, а не 200.

Остальная часть ответа содержит headers так же, как HTTP-запрос. Эти значения могут содержать информацию о софте сервера при последнем изменении страницы/файла, типе mime и т.п.

Коды статуса HTTP

- 200 используются для успешных запросов.
- 300 для перенаправления.
- 400 используются, если возникла проблема с запросом.
- 500 используются, если возникла проблема с сервером.

2 Реализация программы

Для реализации программы был использован язык программирования C#. Запустим разработанную программу, рисунок 3.

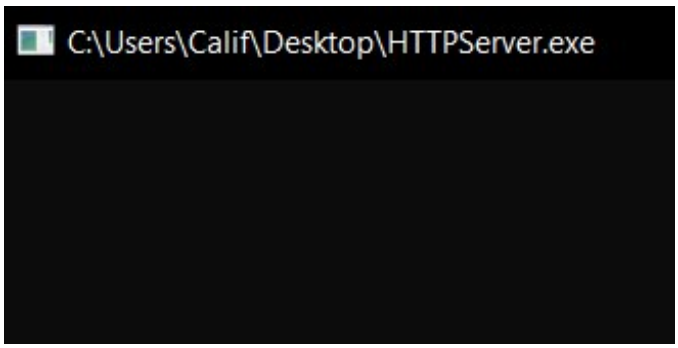

Рисунок 3 – Запуск сервера

Продемонстрируем пример работы программы, рисунок 4.

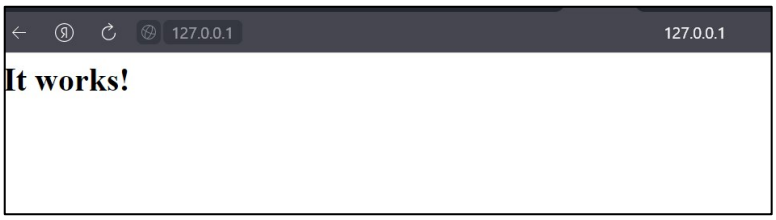

Рисунок 4 – Пример работы сервера

Представим пример работы HTTP клиента, рисунок 5.

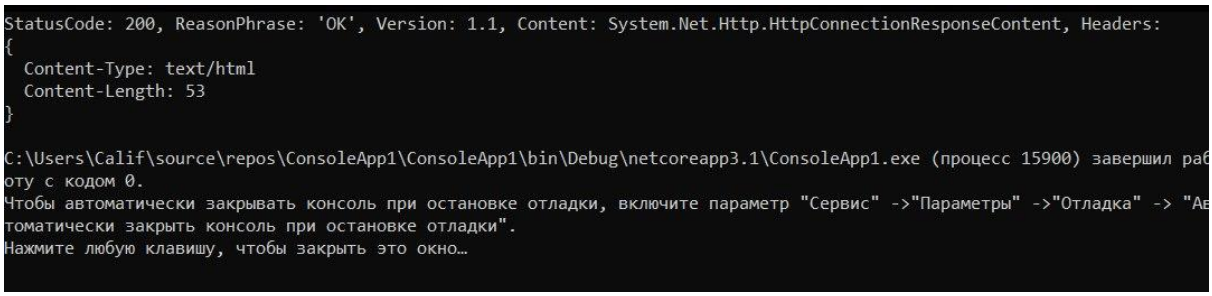

Рисунок 5 – Консоль клиента

Вывод

В результате выполнения данной лабораторной работы был изучен прикладной протокол HTTP - Hyper Text Transfer Protocol (RFC 7231) и реализована его часть.

ПРИЛОЖЕНИЕ А

Исходный код программы HTTP сервера

```
 1:
 2:  using System;
 3:  using System.Collections.Generic;
 4:  using System.Text;
 5:  using System.Net.Sockets;
 6:  using System.Net;
 7:  using System.Threading;
 8:  using System.Text.RegularExpressions;
 9:  using System.IO;
10:
11:  namespace HTTPServer
12:  {
13:      // Класс-обработчик клиента
14:      class Client
15:      {
16:          // Отправка страницы с ошибкой
17:          private void SendError(TcpClient Client, int Code)
18:          {
19:              // Получаем строку вида "200 OK"
20:              // HttpStatusCode хранит в себе все статус-коды HTTP/1.1
21:                                  string  CodeStr  =  Code.ToString()  +  " "  +
((HttpStatusCode)Code).ToString();
22:              // Код простой HTML-странички
23:              string Html = "<html><body><h1>" + CodeStr + "</h1></body></html>";
24:
// Необходимые заголовки: ответ сервера, тип и длина содержимого. После двух пустых строк - са
мо содержимое
25:           string Str = "HTTP/1.1 " + CodeStr + "\nContent-type: text/html\nContent-
Length:" + Html.Length.ToString() + "\n\n" + Html;
26:              // Приведем строку к виду массива байт
27:              byte[] Buffer = Encoding.ASCII.GetBytes(Str);
28:              // Отправим его клиенту
29:              Client.GetStream().Write(Buffer, 0, Buffer.Length);
30:              // Закроем соединение
31:              Client.Close();
32:          }
33:
34:
// Конструктор класса. Ему нужно передавать принятого клиента от TcpListener
35:          public Client(TcpClient Client)
36:          {
37:              // Объявим строку, в которой будет хранится запрос клиента
38:              string Request = "";
39:              // Буфер для хранения принятых от клиента данных
40:              byte[] Buffer = new byte[1024];
41:              // Переменная для хранения количества байт, принятых от клиента
42:              int Count;
43:             // Читаем из потока клиента до тех пор, пока от него поступают данные
44:              while ((Count = Client.GetStream().Read(Buffer, 0, Buffer.Length)) >
0)
45:              {
46:              // Преобразуем эти данные в строку и добавим ее к переменной Request
47:                  Request += Encoding.ASCII.GetString(Buffer, 0, Count);
48:                  // Запрос должен обрываться последовательностью \r\n\r\n
49:
// Либо обрываем прием данных сами, если длина строки Request превышает 4 килобайта
50:                                              // Нам не нужно получать данные из POST-
запроса (и т. п.), а обычный запрос
51:                  // по идее не должен быть больше 4 килобайт
52:                  if (Request.IndexOf("\r\n\r\n") >= 0 || Request.Length > 4096)
53:                  {
54:                      break;
55:                  }
```

```
            }

            // Парсим строку запроса с использованием регулярных выражений
            // При этом отсекаем все переменные GET-запроса
                                        Match   ReqMatch   =   Regex.Match(Request,
@"^\w+\s+([^\s\?]+)[^\s]*\s+HTTP/.*|");

            // Если запрос не удался
            if (ReqMatch == Match.Empty)
            {
                // Передаем клиенту ошибку 400 - неверный запрос
                SendError(Client, 400);
                return;
            }

            // Получаем строку запроса
            string RequestUri = ReqMatch.Groups[1].Value;

           // Приводим ее к изначальному виду, преобразуя экранированные символы
            // Например, "%20" -> " "
            RequestUri = Uri.UnescapeDataString(RequestUri);

            // Если в строке содержится двоеточие, передадим ошибку 400
           // Это нужно для защиты от URL типа http://example.com/../../file.txt
            if (RequestUri.IndexOf("..") >= 0)
            {
                SendError(Client, 400);
                return;
            }

          // Если строка запроса оканчивается на "/", то добавим к ней index.html
            if (RequestUri.EndsWith("/"))
            {
                RequestUri += "index.html";
            }

            string FilePath = "www/" + RequestUri;

            // Если в папке www не существует данного файла, посылаем ошибку 404
            if (!File.Exists(FilePath))
            {
                SendError(Client, 404);
                return;
            }

            // Получаем расширение файла из строки запроса
           string Extension = RequestUri.Substring(RequestUri.LastIndexOf('.'));

            // Тип содержимого
            string ContentType = "";

            // Пытаемся определить тип содержимого по расширению файла
            switch (Extension)
            {
                case ".htm":
                case ".html":
                    ContentType = "text/html";
                    break;
                case ".css":
                    ContentType = "text/stylesheet";
                    break;
                case ".js":
                    ContentType = "text/javascript";
                    break;
                case ".jpg":
                    ContentType = "image/jpeg";
                    break;
                case ".jpeg":
```

```
123:                 case ".png":
124:                 case ".gif":
125:                     ContentType = "image/" + Extension.Substring(1);
126:                     break;
127:                 default:
128:                     if (Extension.Length > 1)
129:                     {
130:                         ContentType = "application/" + Extension.Substring(1);
131:                     }
132:                     else
133:                     {
134:                         ContentType = "application/unknown";
135:                     }
136:                     break;
137:             }
138:
139:             // Открываем файл, страхуясь на случай ошибки
140:             FileStream FS;
141:             try
142:             {
143:                   FS = new FileStream(FilePath, FileMode.Open, FileAccess.Read,
FileShare.Read);
144:             }
145:             catch (Exception)
146:             {
147:                 // Если случилась ошибка, посылаем клиенту ошибку 500
148:                 SendError(Client, 500);
149:                 return;
150:             }
151:
152:             // Посылаем заголовки
153:               string Headers = "HTTP/1.1 200 OK\nContent-Type: " + ContentType +
"\nContent-Length: " + FS.Length + "\n\n";
154:             byte[] HeadersBuffer = Encoding.ASCII.GetBytes(Headers);
155:             Client.GetStream().Write(HeadersBuffer, 0, HeadersBuffer.Length);
156:
157:             // Пока не достигнут конец файла
158:             while (FS.Position < FS.Length)
159:             {
160:                 // Читаем данные из файла
161:                 Count = FS.Read(Buffer, 0, Buffer.Length);
162:                 // И передаем их клиенту
163:                 Client.GetStream().Write(Buffer, 0, Count);
164:             }
165:
166:             // Закроем файл и соединение
167:             FS.Close();
168:             Client.Close();
169:         }
170:     }
171:
172:     class Server
173:     {
174:         TcpListener Listener; // Объект, принимающий TCP-клиентов
175:
176:         // Запуск сервера
177:         public Server(int Port)
178:         {
179:                               Listener  =  new  TcpListener(IPAddress.Any,  Port);
// Создаем "слушателя" для указанного порта
180:             Listener.Start(); // Запускаем его
181:
182:             // В бесконечном цикле
183:             while (true)
184:             {
185:                 // Принимаем нового клиента
186:                 TcpClient Client = Listener.AcceptTcpClient();
187:                 // Создаем поток
```

```
188:                Thread Thread = new Thread(new ParameterizedThreadStart(ClientThread));
189:                    // И запускаем этот поток, передавая ему принятого клиента
190:                    Thread.Start(Client);
191:                }
192:            }
193:
194:            static void ClientThread(Object StateInfo)
195:            {
196:
// Просто создаем новый экземпляр класса Client и передаем ему приведенный к классу TcpClient
объект StateInfo
197:                new Client((TcpClient)StateInfo);
198:            }
199:
200:            // Остановка сервера
201:            ~Server()
202:            {
203:                // Если "слушатель" был создан
204:                if (Listener != null)
205:                {
206:                    // Остановим его
207:                    Listener.Stop();
208:                }
209:            }
210:
211:            static void Main(string[] args)
212:            {
213:                // Создадим новый сервер на порту 80
214:                new Server(80);
215:            }
216:        }
217:    }
218:
```

ПРИЛОЖЕНИЕ Б

Исходный код HTTP клиента

```
using System;
using System.Net.Http;
using System.Threading.Tasks;

namespace HttpClientHead
{
    class Program
    {
        static async Task Main(string[] args)
        {
            var url = "http://127.0.0.1/";
            using var client = new HttpClient();

            var result = await client.SendAsync(new HttpRequestMessage(HttpMethod.Head, url));

            Console.WriteLine(result);
        }
    }
}
```