|  |  |
| --- | --- |
| Gerb-BMSTU_01 | **Министерство науки и высшего образования Российской Федерации**  **Федеральное государственное бюджетное образовательное учреждение**  **высшего образования**  **«Московский государственный технический университет**  **имени Н.Э. Баумана**  **(национальный исследовательский университет)»**  **(МГТУ им. Н.Э. Баумана)** |

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

**ОТЧЕТ**

**по лабораторной работе № 3**

**по курсу: «Компьютерные сети»**

|  |  |
| --- | --- |
| **Тема Передача бинарных файлов от сервера клиенту. Http-сервер для обработки GET-запросов**  **Вариант 20 (2)**  **Студент Якуба Д. В.**  **Группа ИУ7-73Б**  **Оценка (баллы)**  **Преподаватель Рогозин Н.О.** |  |

Москва, 2021

1. Задание

1.1 Часть 1

Разработать клиент-серверное приложение с использованием сетевых сокетов и протокола TCP для передачи бинарных файлов от сервера клиентам. Сервер должен позволять подключение нескольких клиентов. Имя файла задается клиентом при подключении в виде текстовой строки. Сервер должен отобразить, используя пользовательский интерфейс, названия всех файлов в папке с исполняемым файлом сервера, предоставив клиенту выбор.

1.2 Часть 2

Разработать http сервер для обработки GET-запросов и предоставления статической информации клиенту. Разработать http-клиент для проверки данного сервера с помощью GET-запросов. Использовать системные сокеты и транспортный протокол TCP. Язык С/С++. Возможно использование ранее разработанного TCP-сервера в качестве основы.

Использование фреймворков не допускается.

Выполнить дополнительную задачу в зависимости от варианта: сохранять статистику о различных форматах файлов, к которым обращался тот или иной пользователь.

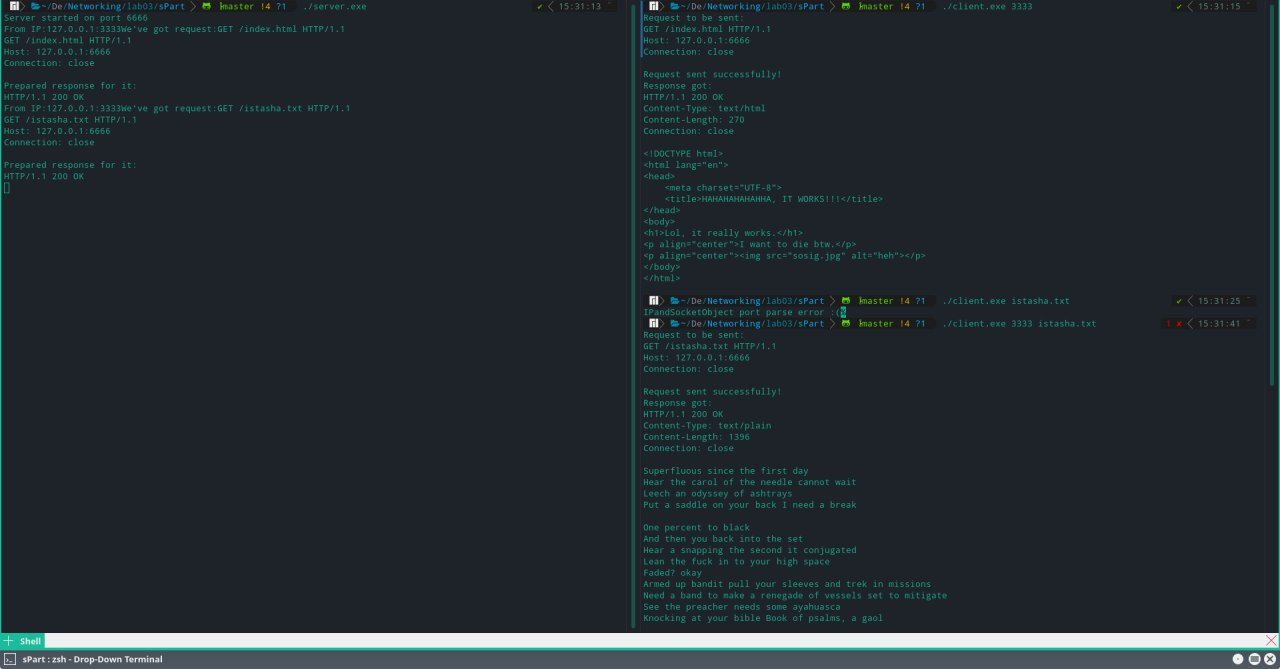
2. Примеры работы

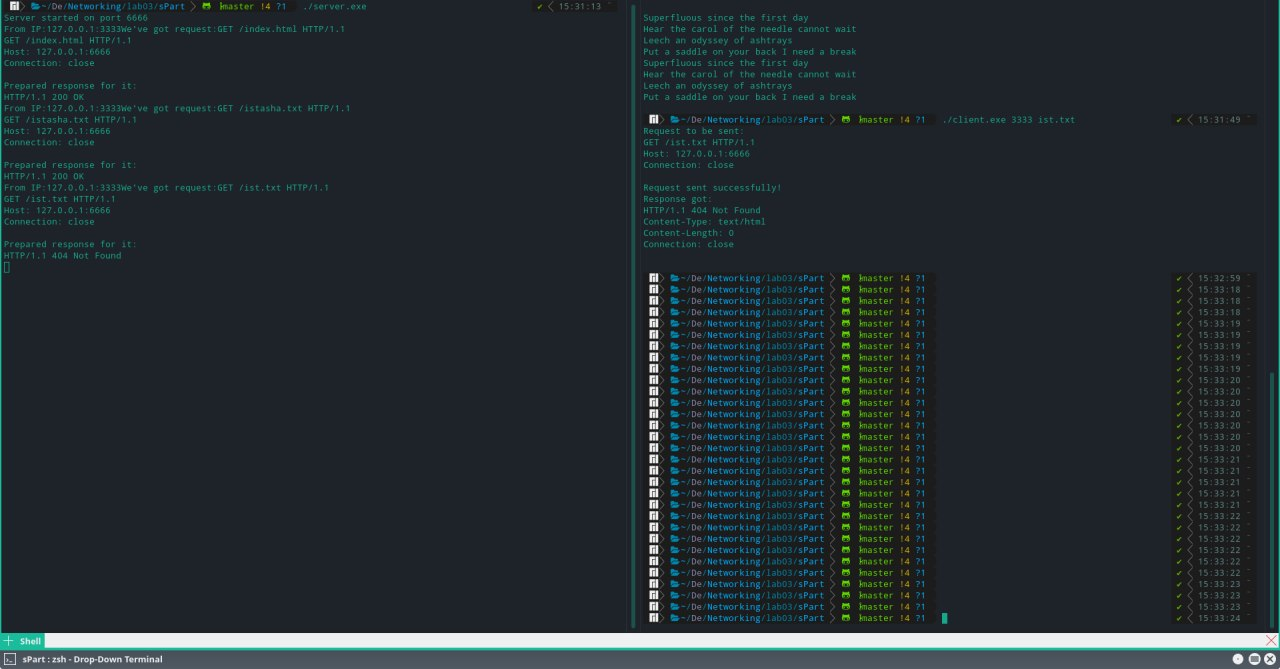
2.1 Часть 1

На рисунках ниже предоставлены примеры работы первого выполненного задания.

2.2 Часть 2

На рисунках ниже предоставлены примеры работы второго выполненного задания.





3. Описание работы

3.1 Часть 1

Работа программы строится на следующем алгоритме:

1. Направление от клиента серверу приветственного сообщения

|  |
| --- |
| if (sendto(socketDescr, "hello", strlen("hello"), 0, (*struct* sockaddr \*)&serverAddrToSend,                 sizeof(serverAddrToSend)) < 0)  {      printError("Cannot send a hello message.");      return ERROR\_SEND\_TO;  } |

|  |
| --- |
| gotInBytes = recvfrom(socketDescr, flexBuffer, BUFFER\_SIZE, 0, (*struct* sockaddr \*)&client, &len);  if (gotInBytes < 0)  {  printError("Recvfrom error");        return ERROR\_RECV;  }  flexBuffer[gotInBytes] = '\0';  if (strcmp(flexBuffer, "hello"))  {  printError("Hello step is confused because it's not a hello message.");  continue;  } |

1. Направления от сервера клиенту списка файлов, доступных для загрузки

|  |
| --- |
| if (sendto(socketDescr, filesInDirectory, strlen(filesInDirectory), 0, (*struct* sockaddr \*)&client,                     sizeof(client)) < 0)          {              printError("Cannot send a message to client");              return ERROR\_SENDTO;          } |

|  |
| --- |
| *char* flexBuffer[BUFFER\_SIZE];  *struct* sockaddr\_in server = {0};  *socklen\_t* len = sizeof(*struct* sockaddr\_in);  size\_t gotInBytes = recvfrom(socketDescr, flexBuffer, BUFFER\_SIZE, 0, (*struct* sockaddr \*)&server, &len);  if (gotInBytes < 0)  {  printError("Recvfrom error");      return ERROR\_RECV;  }  if (server.sin\_family != serverAddrToSend.sin\_family || server.sin\_port != serverAddrToSend.sin\_port)  {  printError("Got answer from another server. Kinda strange...");      return ERROR\_CATCH\_ANSWER;  }  flexBuffer[gotInBytes] = '\0';  printOkMessage("Files on server:");  printOkMessage(flexBuffer);  printOkMessage("\nChoose filename from this list:");  *char* choice[FILENAME\_MAX];  scanf("%s", choice); |

1. Отправка на сервер имени файла, который требуется получить

|  |
| --- |
| if (sendto(socketDescr, choice, strlen(choice), 0, (*struct* sockaddr \*)&serverAddrToSend, sizeof(serverAddrToSend)) < 0)  {      printError("Cannot send name of file to server.");      return ERROR\_SEND\_TO;  } |

|  |
| --- |
| gotInBytes = recvfrom(socketDescr, flexBuffer, BUFFER\_SIZE, 0, (*struct* sockaddr \*)&client, &len);  if (gotInBytes < 0)  {  printError("Recvfrom error");      return ERROR\_RECV;  }  flexBuffer[gotInBytes] = '\0'; |

1. Направление от сервера клиенту сначала размера файла, а затем уже самого файла

|  |
| --- |
| gotInBytes = recvfrom(socketDescr, flexBuffer, BUFFER\_SIZE, 0, (*struct* sockaddr \*)&client, &len);  if (gotInBytes < 0)  {  printError("Recvfrom error");     return ERROR\_RECV;  }  flexBuffer[gotInBytes] = '\0';  *char* fileToSendName[BUFFER\_SIZE] = {0};  snprintf(fileToSendName, strlen(flexBuffer) + 8, "static/%s", flexBuffer);  printf("Filename got: %s\n", fileToSendName);  *int* fileToSend = open(fileToSendName, O\_RDONLY);  if (fileToSend < 0)  {  printError("Cannot open file\n");      continue;  }  *struct* stat fileStat;  if (fstat(fileToSend, &fileStat) < 0)  {  printError("Cannot get file stats");      continue;  }  *socklen\_t* socketLen = sizeof(*struct* sockaddr\_in);  *struct* sockaddr\_in peerAddr;  *int* peerSocket = accept(socketToSendFile, (*struct* sockaddr \*)&peerAddr, &socketLen);  if (peerSocket < 0)  {  printError("Error in acception");      continue;  }  *char* fileSize[FILENAME\_MAX] = { 0 };  snprintf(fileSize, FILENAME\_MAX, "%ld", fileStat.st\_size);  if (send(peerSocket, fileSize, sizeof(fileSize), 0) < 0)  {  printError("Send error");      continue;  }  *long* offset = 0;  size\_t remainData = fileStat.st\_size;  *int* sent = 0;  while (((sent = sendfile(peerSocket, fileToSend, &offset, BUFSIZ)) > 0) && (remainData > 0))  {  remainData -= sent;  } |

|  |
| --- |
| *struct* sockaddr\_in serverAddrToGet = {          .sin\_family = AF\_INET, .sin\_port = htons(SER\_PORT\_DATA\_TRANSFER), .sin\_addr.s\_addr = INADDR\_ANY};  if (connect(socketForFileTransfer, (*struct* sockaddr\*)&serverAddrToGet, sizeof(serverAddrToGet)) < 0)  {      printError("Cannot connect to the server to get the file");  return ERROR\_CONNECT;  }  *char* sizeBuffer[BUFFER\_SIZE];  recv(socketForFileTransfer, sizeBuffer, BUFFER\_SIZE, 0);  *int* sizeOfFile = atoi(sizeBuffer);  FILE \*receivedFile = fopen(choice, "w");  if (receivedFile == NULL)  {      printError("Cannot create file");  return ERROR\_FILE;  }  for (*int* remainData = sizeOfFile, len = 0;  remainData > 0 && ((len = recv(socketForFileTransfer, flexBuffer, BUFFER\_SIZE, 0)) > 0); remainData -= len)  {  fwrite(flexBuffer, sizeof(*char*), len, receivedFile);  } |

3.2 Часть 2

В разработанном приложении отсутствует выполненное дополнительное задание, однако он является многопоточным:

|  |
| --- |
| static sig\_atomic\_t threadsAreWorking = true;  static pthread\_t poolOfThreads[NUMBER\_OF\_THREADS];  *void* threadsCreation()  {      for (*int* i = 0; i < NUMBER\_OF\_THREADS; i++)      {          pthread\_create(poolOfThreads + i, nullptr, threadFun, nullptr);      }  }  *void* threadsRemove()  {      threadsAreWorking = false;      for (*int* i = 0; i < NUMBER\_OF\_THREADS; i++)      {          pthread\_cancel(poolOfThreads[i]);      }  }  *class* IpAndSocketInfo  {  *public:*      std::string ip;  *int* socket;      IpAndSocketInfo(sockaddr\_in *clientAddr*, *int* *connectionSocket*)      {          ip = std::string(inet\_ntoa(*clientAddr*.sin\_addr)) + ":" + std::to\_string(ntohs(*clientAddr*.sin\_port));          socket = *connectionSocket*;      }  };  static pthread\_mutex\_t mutex = PTHREAD\_MUTEX\_INITIALIZER;  static pthread\_cond\_t cond = PTHREAD\_COND\_INITIALIZER;  static std::queue<IpAndSocketInfo> queue;  static *void* clientHandler(const IpAndSocketInfo &*clientInfo*)  {  *char* buff[BUFFER\_SIZE];      bzero(buff, sizeof(buff));      read(*clientInfo*.socket, buff, sizeof(buff));      std::string request = buff;  *auto* response = createResponse(*clientInfo*, request);      write(*clientInfo*.socket, response.c\_str(), response.size());  }  *void* \*threadFun(*void* \**argv*)  {      while (threadsAreWorking)      {          IpAndSocketInfo \*client = nullptr;          pthread\_mutex\_lock(&mutex);          if (queue.empty())          {              pthread\_cond\_wait(&cond, &mutex);              client = &queue.front();              queue.pop();          }          pthread\_mutex\_unlock(&mutex);          if (client)          {              clientHandler(\*client);          }      }      return nullptr;  }  *void* addClientToQueue(const sockaddr\_in &*clientAddr*, *int* *clientConnection*)  {      pthread\_mutex\_lock(&mutex);      queue.emplace(*clientAddr*, *clientConnection*);      pthread\_cond\_signal(&cond);      pthread\_mutex\_unlock(&mutex);  } |

Данные функции используются в реализации сервера:

|  |
| --- |
| *int* main()  {      threadsCreation();      if ((serverSocket = socket(AF\_INET, SOCK\_STREAM, 0)) == -1)      {          threadsRemove();          std::cout << "Cannot create socket";          return EXIT\_FAILURE;      }  …  for (;;)      {          sockaddr\_in clientAddr;  *socklen\_t* client\_size;          const *int* connectionSocket = accept(serverSocket, reinterpret\_cast<sockaddr \*>(&clientAddr), &client\_size);          if (connectionSocket == -1)          {              return exitOnServerError("Error on acception");          }          addClientToQueue(clientAddr, connectionSocket);      }  } |

Функция отправки клиентом GET-запроса:

|  |
| --- |
| *void* sendGetRequest(const std::string &*uri*)  {      std::ostringstream stringToForm;      stringToForm << "GET /" << *uri* << " HTTP/1.1\n";      stringToForm << "Host: 127.0.0.1:" << SERVER\_PORT << '\n';      stringToForm << "Connection: close\n";      const *auto* request = stringToForm.str();      std::cout << "Request to be sent:\n" << request << '\n';      write(clientSocket, request.c\_str(), request.size());  } |

Функция формирования ответа сервера:

|  |
| --- |
| static std::string createResponse(const IpAndSocketInfo &*clientInfo*, const std::string &*request*)  {      std::string response;  *int* status = 200;      std::string status\_string = "OK";      std::unordered\_map<std::string, std::string> response\_headers;      response\_headers["Connection"] = "close";      std::string body;  *auto* line\_end = *request*.find('\n');  *auto* line = *request*.substr(0, line\_end);      std::cout << "From IP:" << *clientInfo*.ip.c\_str() << "We've got request:" << line.c\_str() << std::endl;      std::cout << *request* << std::endl;      std::cout << "Prepared response for it:" << std::endl;  *auto* headers = getMapOfHeaders(*request*.substr(line\_end + 1, *request*.length() - line\_end));      status = getStatusForHost(headers);      if (status != 200)      {          response = createResponse("HTTP/1.1", status, response\_headers, body);          return response;      }  *auto* [method, path, protocol] = parseStartRequestLine(line);      if (method != "GET" || protocol != "HTTP/1.1")      {          status = 405;          response = createResponse(protocol, status, response\_headers, body);          return response;      }      std::ifstream requiredFile{path};      if (!requiredFile.good())      {          status = 404;          response = createResponse(protocol, status, response\_headers, body);          return response;      }      body = std::string{std::istreambuf\_iterator<*char*>(requiredFile), std::istreambuf\_iterator<*char*>()};      requiredFile.close();      response\_headers["Content-Type"] = getContentType(getExtension(path));      size\_t idx = response\_headers["Content-Type"].find("image");      if (idx != std::string::npos)      {          body = inHex(body);      }      response\_headers["Content-Length"] = std::to\_string(body.length());      response = createResponse(protocol, status, response\_headers, body);      return response;  } |