Кроссплатформенный многопоточный TCP/IP сервер на C++

**16 мин**

**97K**

<https://habr.com/ru/articles/503432/>

[C++\*](https://habr.com/ru/hubs/cpp/)[Разработка под Linux\*](https://habr.com/ru/hubs/linux_dev/)[Разработка под Windows\*](https://habr.com/ru/hubs/win_dev/)

[Из песочницы](https://habr.com/ru/sandbox/)

Решил задаться целью написать простой в использовании и при этом быстрый многопоточного TCP/IP сервера на C++ и при этом кроссплатформенный — как минимум чтобы работал на платформах Windows и Linux без требования как-либо изменять код за пределами самописной библиотеки. Ранее, на чистом C++ без библиотек вроде Qt, сетевым программировнием не занимался, и предвещал себе долгое время мучений с платформо-зависимостью. Но как оказалось всё гораздо проще чем казалось на первый взгляд, ведь в основном интерфейсы сокетов обоих систем похожи как две капли воды и различаются лишь в мелких деталях.

Для начала определим общий для клиента и сервера заголовок:

**general.h**

#ifndef GENERAL\_H  
#define GENERAL\_H  
  
#ifdef \_WIN32  
#else  
#define SD\_BOTH 0  
#endif  
  
#include <cstdint>  
#include <cstring>  
#include <cinttypes>  
#include <malloc.h>   
  
// IP 127.0.0.1  
**uint32\_t** LOCALHOST\_IP = 0x0100007f;  
  
// Код состояния сокета  
**enum** **class** SocketStatus : **uint8\_t** {  
 connected = 0,  
 err\_socket\_init = 1,  
 err\_socket\_bind = 2,  
 err\_socket\_connect = 3,  
 disconnected = 4  
};  
  
// Буффер данных куда у нас будет приниматься данные от другой стороны  
**struct** DataBuffer {  
 **int** size = 0;  
 **void\*** data\_ptr = nullptr;  
  
 DataBuffer() = **default**;  
 DataBuffer(**int** size, **void\*** data\_ptr) : size(size), data\_ptr(data\_ptr) {}  
 DataBuffer(**const** DataBuffer& other) : size(other.size), data\_ptr(malloc(size)) {memcpy(data\_ptr, other.data\_ptr, size);}  
 DataBuffer(DataBuffer&& other) : size(other.size), data\_ptr(other.data\_ptr) {other.data\_ptr = nullptr;}  
 ~DataBuffer() {**if**(data\_ptr) free(data\_ptr); data\_ptr = nullptr;}  
  
 **bool** isEmpty() {**return** !data\_ptr || !size;}  
 **operator** **bool**() {**return** data\_ptr && size;}  
};  
// в последней версии библиотеки typedef от std::vector<uint8\_t>  
  
// Тип сокета  
**enum** **class** SocketType : **uint8\_t** {  
 client\_socket = 0,  
 server\_socket = 1  
};  
  
// Базовый класс TCP клиента  
**class** TcpClientBase {  
**public**:  
 **typedef** SocketStatus status;  
 **virtual** ~TcpClientBase() {};  
 **virtual** status disconnect() = 0;  
 **virtual** status getStatus() **const** = 0;  
 **virtual** **bool** sendData(**const** **void\*** buffer, **const** **size\_t** size) **const** = 0;  
 **virtual** DataBuffer loadData() = 0;  
 **virtual** **uint32\_t** getHost() **const** = 0;  
 **virtual** **uint16\_t** getPort() **const** = 0;  
 **virtual** SocketType getType() **const** = 0;  
};  
  
#endif // GENERAL\_H

Итак интерфейсы классов сервера и клиента(со стороны сервера) выглядят следующим образом:

**TcpServer.h**

#ifndef TCPSERVER\_H  
#define TCPSERVER\_H  
  
#include <functional>  
#include <list>  
  
#include <thread>  
#include <mutex>  
#include <shared\_mutex>  
  
#ifdef \_WIN32 // Windows NT  
  
#include <WinSock2.h>  
#include <mstcpip.h>  
  
#else // \*nix  
  
#include <sys/socket.h>  
#include <netinet/in.h>  
#include <netinet/tcp.h>  
#include <unistd.h>  
#include <stdio.h>  
#include <stdlib.h>  
  
#endif  
  
#include "general.h"  
  
#ifdef \_WIN32 // Windows NT  
**typedef** **int** **SockLen\_t**;  
**typedef** SOCKADDR\_IN SocketAddr\_in;  
**typedef** SOCKET Socket;  
**typedef** u\_long **ka\_prop\_t**;  
#else // POSIX  
**typedef** **socklen\_t** **SockLen\_t**;  
**typedef** **struct** sockaddr\_in SocketAddr\_in;  
**typedef** **int** Socket;  
**typedef** **int** **ka\_prop\_t**;  
#endif  
  
// Конфигурация Keep-Alive соединения  
**struct** KeepAliveConfig{  
 **ka\_prop\_t** ka\_idle = 120;  
 **ka\_prop\_t** ka\_intvl = 3;  
 **ka\_prop\_t** ka\_cnt = 5;  
};  
  
// Класс Tcp сервера  
**struct** TcpServer {  
 // Класс клиента сервера (реализация определена ниже)  
 **struct** Client;  
 // Тип обработчик данных клиента  
 **typedef** std::function<**void**(DataBuffer, Client&)> **handler\_function\_t**;  
 // Тип обработчика подключения/отсоединения клиента  
 **typedef** std::function<**void**(Client&)> **con\_handler\_function\_t**;  
  
 // Коды статуса сервера  
 **enum** **class** status : **uint8\_t** {  
 up = 0,  
 err\_socket\_init = 1,  
 err\_socket\_bind = 2,  
 err\_scoket\_keep\_alive = 3,  
 err\_socket\_listening = 4,  
 close = 5  
 };  
  
**private**:  
 // Сокет сервера  
 Socket serv\_socket;  
 // Порт сервера  
 **uint16\_t** port;  
 // Код статуса  
 status \_status = status::close;  
 // Обработчик данных от клиента  
 **handler\_function\_t** handler;  
 // Обработчик подключения клиента  
 **con\_handler\_function\_t** connect\_hndl = [](Client&){};  
 // Обработчик отсоединения клиента  
 **con\_handler\_function\_t** disconnect\_hndl = [](Client&){};  
 // Поток-обработчик подключений  
 std::thread accept\_handler\_thread;  
 // Поток ожидания данных  
 std::thread data\_waiter\_thread;  
 // Тип итератора клиента  
 **typedef** std::list<std::unique\_ptr<Client>>::iterator ClientIterator;  
  
 // Keep-Alive конфигурация  
 KeepAliveConfig ka\_conf;  
  
 // Список клиентов  
 std::list<std::unique\_ptr<Client>> client\_list;  
 // Мьютекс для синзронизации потоков подключения и ожидания данных  
 std::mutex client\_mutex;   
  
 // Для систем Windows так же требуется  
 // структура определяющая версию WinSocket  
#ifdef \_WIN32 // Windows NT  
 WSAData w\_data;  
#endif  
  
 // Включить Keep-Alive для сокета  
 **bool** enableKeepAlive(Socket socket);  
 // Метод обработчика подключений  
 **void** handlingAcceptLoop();  
 // Метод ожидания данных  
 **void** waitingDataLoop();  
  
**public**:  
 // Упрощённый конструктор с указанием:  
 // \* порта  
 // \* обработчика данных  
 // \* конфигурации Keep-Alive  
 TcpServer(**const** **uint16\_t** port,  
 **handler\_function\_t** handler,  
 KeepAliveConfig ka\_conf = {});  
 // Конструктор с указанием:  
 // \* порта  
 // \* обработчика данных  
 // \* обработчика подключений  
 // \* обработчика отключений  
 // \* конфигурации Keep-Alive  
 TcpServer(**const** **uint16\_t** port,  
 **handler\_function\_t** handler,  
 **con\_handler\_function\_t** connect\_hndl,  
 **con\_handler\_function\_t** disconnect\_hndl,  
 KeepAliveConfig ka\_conf = {});  
  
 // Деструктор  
 ~TcpServer();  
  
 // Заменить обработчик данных  
 **void** setHandler(**handler\_function\_t** handler);  
 // Getter порта  
 **uint16\_t** getPort() **const**;  
 // Setter порта  
 **uint16\_t** setPort(**const** **uint16\_t** port);  
 // Getter кода статуса сервера  
 status getStatus() **const** {**return** \_status;}  
 // Метод запуска сервера  
 status start();  
 // Метод остановки сервера  
 **void** stop();  
 // Метод для входа присоединения циклических потоков сервера  
 **void** joinLoop();  
  
 // Исходящее подключение от сервера к другому серверу  
 **bool** connectTo(**uint32\_t** host, **uint16\_t** port, **con\_handler\_function\_t** connect\_hndl);  
  
 // Отправить данные всем клиентам сервера  
 **void** sendData(**const** **void\*** buffer, **const** **size\_t** size);  
 // Отправить данные клиенту по порту и хосту  
 **bool** sendDataBy(**uint32\_t** host, **uint16\_t** port, **const** **void\*** buffer, **const** **size\_t** size);  
 // Отключить клиента по порту и хосту  
 **bool** disconnectBy(**uint32\_t** host, **uint16\_t** port);  
 // Отключить всех клиентов  
 **void** disconnectAll();  
};  
  
// Класс клиента (со стороны сервера)  
**struct** TcpServer::Client : **public** TcpClientBase {  
 **friend** **struct** TcpServer;  
  
 // Мьютекс для синхронизации обработки данныз  
 std::mutex access\_mtx;  
 // Адрес клиента  
 SocketAddr\_in address;  
 // Сокет слиента  
 Socket socket;  
 // Код статуса клиента  
 status \_status = status::connected;  
  
**public**:  
 // Конструктор с указанием:  
 // \* сокета клиента  
 // \* адреса клиента  
 Client(Socket socket, SocketAddr\_in address);  
 // Деструктор  
 **virtual** ~Client() **override**;  
 // Getter хоста  
 **virtual** **uint32\_t** getHost() **const** **override**;  
 // Getter порта  
 **virtual** **uint16\_t** getPort() **const** **override**;  
 // Getter кода статуса подключения  
 **virtual** status getStatus() **const** **override** {**return** \_status;}  
 // Отключить клиента  
 **virtual** status disconnect() **override**;  
 // Получить данные от клиента  
 **virtual** DataBuffer loadData() **override**;  
 // Отправить данные клиенту  
 **virtual** **bool** sendData(**const** **void\*** buffer, **const** **size\_t** size) **const** **override**;  
 // Определить "сторону" клиента  
 **virtual** SocketType getType() **const** **override** {**return** SocketType::server\_socket;}  
};  
  
#endif // TCPSERVER\_H

Как можно заметить на данном этапи зависимости операционных систем без особых проблем решаются при помощи макросов и псевдонимов типов данных. Так же в Windows части TcpServer-хедера присутствует структура для обозначения используемой версии WinSocket — WSAData w\_data;(см. [WSAData](https://docs.microsoft.com/en-us/windows/win32/api/winsock/ns-winsock-wsadata))

Перейдём к реализации сервера:

**TcpServer.cpp**

#include "../include/TcpServer.h"  
#include <chrono>  
#include <cstring>  
#include <mutex>  
  
#ifdef \_WIN32  
// Макросы для выражений зависимых от OS  
#define WIN(exp) exp  
#define NIX(exp)  
  
// Конвертировать WinSocket код ошибки в Posix код ошибки  
**inline** **int** convertError() {  
 **switch** (WSAGetLastError()) {  
 **case** 0:  
 **return** 0;  
 **case** WSAEINTR:  
 **return** EINTR;  
 **case** WSAEINVAL:  
 **return** EINVAL;  
 **case** WSA\_INVALID\_HANDLE:  
 **return** EBADF;  
 **case** WSA\_NOT\_ENOUGH\_MEMORY:  
 **return** ENOMEM;  
 **case** WSA\_INVALID\_PARAMETER:  
 **return** EINVAL;  
 **case** WSAENAMETOOLONG:  
 **return** ENAMETOOLONG;  
 **case** WSAENOTEMPTY:  
 **return** ENOTEMPTY;  
 **case** WSAEWOULDBLOCK:  
 **return** EAGAIN;  
 **case** WSAEINPROGRESS:  
 **return** EINPROGRESS;  
 **case** WSAEALREADY:  
 **return** EALREADY;  
 **case** WSAENOTSOCK:  
 **return** ENOTSOCK;  
 **case** WSAEDESTADDRREQ:  
 **return** EDESTADDRREQ;  
 **case** WSAEMSGSIZE:  
 **return** EMSGSIZE;  
 **case** WSAEPROTOTYPE:  
 **return** EPROTOTYPE;  
 **case** WSAENOPROTOOPT:  
 **return** ENOPROTOOPT;  
 **case** WSAEPROTONOSUPPORT:  
 **return** EPROTONOSUPPORT;  
 **case** WSAEOPNOTSUPP:  
 **return** EOPNOTSUPP;  
 **case** WSAEAFNOSUPPORT:  
 **return** EAFNOSUPPORT;  
 **case** WSAEADDRINUSE:  
 **return** EADDRINUSE;  
 **case** WSAEADDRNOTAVAIL:  
 **return** EADDRNOTAVAIL;  
 **case** WSAENETDOWN:  
 **return** ENETDOWN;  
 **case** WSAENETUNREACH:  
 **return** ENETUNREACH;  
 **case** WSAENETRESET:  
 **return** ENETRESET;  
 **case** WSAECONNABORTED:  
 **return** ECONNABORTED;  
 **case** WSAECONNRESET:  
 **return** ECONNRESET;  
 **case** WSAENOBUFS:  
 **return** ENOBUFS;  
 **case** WSAEISCONN:  
 **return** EISCONN;  
 **case** WSAENOTCONN:  
 **return** ENOTCONN;  
 **case** WSAETIMEDOUT:  
 **return** ETIMEDOUT;  
 **case** WSAECONNREFUSED:  
 **return** ECONNREFUSED;  
 **case** WSAELOOP:  
 **return** ELOOP;  
 **case** WSAEHOSTUNREACH:  
 **return** EHOSTUNREACH;  
 **default**:  
 **return** EIO;  
 }  
}  
  
#else  
// Макросы для выражений зависимых от OS  
#define WIN(exp)  
#define NIX(exp) exp  
#endif  
  
// Реализация конструктора сервера с указанием  
// \* порта  
// \* обработчика данных  
// \* Keep-Alive конфигурации  
TcpServer::TcpServer(**const** **uint16\_t** port,  
 **handler\_function\_t** handler,  
 KeepAliveConfig ka\_conf)  
 : TcpServer(port, handler, [](Client&){}, [](Client&){}, ka\_conf) {}  
  
// Реализация конструктора сервера с указанием  
// \* порта  
// \* обработчика данных  
// \* обработчика подключения  
// \* обработчика отключения  
// \* Keep-Alive конфигурации  
TcpServer::TcpServer(**const** **uint16\_t** port,  
 **handler\_function\_t** handler,  
 **con\_handler\_function\_t** connect\_hndl,  
 **con\_handler\_function\_t** disconnect\_hndl,  
 KeepAliveConfig ka\_conf)  
 : port(port), handler(handler), connect\_hndl(connect\_hndl), disconnect\_hndl(disconnect\_hndl), ka\_conf(ka\_conf) {}  
  
// Деструктор сервера  
// автоматически закрывает сокет сервера   
TcpServer::~TcpServer() {  
 **if**(\_status == status::up)  
 stop();  
 WIN(WSACleanup());  
}  
  
// Setter обработчика данных  
**void** TcpServer::setHandler(**TcpServer::handler\_function\_t** handler) {**this**->handler = handler;}  
  
// Getter порта  
**uint16\_t** TcpServer::getPort() **const** {**return** port;}  
// Setter порта  
**uint16\_t** TcpServer::setPort( **const** **uint16\_t** port) {  
 **this**->port = port;  
 start();  
 **return** port;  
}  
  
// Реализация запуска сервера  
TcpServer::status TcpServer::start() {  
 **int** flag;  
 // Если сервер запущен, то отключаем его  
 **if**(\_status == status::up) stop();  
  
 // Для Windows указываем версию WinSocket  
 WIN(**if**(WSAStartup(MAKEWORD(2, 2), &w\_data) == 0) {})  
  
 // Задаём адрес сервера  
 SocketAddr\_in address;  
 // INADDR\_ANY - любой IP адрес  
 address.sin\_addr  
 WIN(.S\_un.S\_addr)NIX(.s\_addr) = INADDR\_ANY;  
 // Задаём порт сервера  
 address.sin\_port = htons(port);  
 // Семейство сети AF\_INET - IPv4 (AF\_INET6 - IPv6)  
 address.sin\_family = AF\_INET;  
  
 // Создаём TCP сокет  
 **if**((serv\_socket = socket(AF\_INET, SOCK\_STREAM, 0)) WIN(== INVALID\_SOCKET)NIX(== -1))  
 **return** \_status = status::err\_socket\_init;  
  
 flag = true;  
 // Устанавливаем параметр сокета SO\_REUSEADDR в true (подробнее https://it.wikireading.ru/7093)  
 **if**((setsockopt(serv\_socket, SOL\_SOCKET, SO\_REUSEADDR, WIN((**char\***))&flag, **sizeof**(flag)) == -1) ||  
 // Привязываем к сокету адрес и порт  
 (bind(serv\_socket, (**struct** sockaddr\*)&address, **sizeof**(address)) WIN(== SOCKET\_ERROR)NIX(< 0)))  
 **return** \_status = status::err\_socket\_bind;  
  
 // Активируем ожидание фходящих соединений  
 **if**(listen(serv\_socket, SOMAXCONN) WIN(== SOCKET\_ERROR)NIX(< 0))  
 **return** \_status = status::err\_socket\_listening;  
  
 \_status = status::up;  
 // Запускаем поток ожидания соединений  
 accept\_handler\_thread = std::thread([**this**]{handlingAcceptLoop();});  
 // Запускаем поток ожидания данных  
 data\_waiter\_thread = std::thread([**this**]{waitingDataLoop();});  
 **return** \_status;  
}  
  
// Реализация остановки сервера  
**void** TcpServer::stop() {  
 \_status = status::close;  
 // Закрываем сокет  
 WIN(closesocket)NIX(close)(serv\_socket);  
 // Ожидаем завершения потоков  
 joinLoop();  
 // Вычищаем список клиентов  
 client\_list.clear();  
}  
  
// "Вхождение" в потоки ожидания  
**void** TcpServer::joinLoop() {accept\_handler\_thread.join(); data\_waiter\_thread.join();}  
  
// Создание подключение со стороны сервера  
// (подключение аналогично клиентоскому, но обрабатывается  
// тем же обработчиком, что и входящие соединения)  
**bool** TcpServer::connectTo(**uint32\_t** host, **uint16\_t** port, **con\_handler\_function\_t** connect\_hndl) {  
 Socket client\_socket;  
 SocketAddr\_in address;  
 // Создание TCP сокета  
 **if**((client\_socket = socket(AF\_INET, SOCK\_STREAM, IPPROTO\_IP)) WIN(== INVALID\_SOCKET) NIX(< 0)) **return** false;  
  
 **new**(&address) SocketAddr\_in;  
 address.sin\_family = AF\_INET;  
 address.sin\_addr.s\_addr = host;  
 WIN(address.sin\_addr.S\_un.S\_addr = host;)  
 NIX(address.sin\_addr.s\_addr = host;)  
  
 address.sin\_port = htons(port);  
  
 // Установка соединения  
 **if**(connect(client\_socket, (sockaddr \*)&address, **sizeof**(address))  
 WIN(== SOCKET\_ERROR)NIX(!= 0)  
 ) {  
 WIN(closesocket(client\_socket);)NIX(close(client\_socket);)  
 **return** false;  
 }  
  
 // Активация Keep-Alive  
 **if**(!enableKeepAlive(client\_socket)) {  
 shutdown(client\_socket, 0);  
 WIN(closesocket)NIX(close)(client\_socket)  
 }  
  
 std::unique\_ptr<Client> client(**new** Client(client\_socket, address));  
 // Запуск обработчика подключения  
 connect\_hndl(\*client);  
 // Добавление клиента в список клиентов  
 client\_mutex.lock();  
 client\_list.emplace\_back(std::move(client));  
 client\_mutex.unlock();  
 **return** true;  
}  
  
// Отправка данных всем клиентам  
**void** TcpServer::sendData(**const** **void\*** buffer, **const** **size\_t** size) {  
 **for**(std::unique\_ptr<Client>& client : client\_list)  
 client->sendData(buffer, size);  
}  
  
// Отправка данных по конкретному хосту и порту  
**bool** TcpServer::sendDataBy(**uint32\_t** host, **uint16\_t** port, **const** **void\*** buffer, **const** **size\_t** size) {  
 **bool** data\_is\_sended = false;  
 **for**(std::unique\_ptr<Client>& client : client\_list)  
 **if**(client->getHost() == host &&  
 client->getPort() == port) {  
 client->sendData(buffer, size);  
 data\_is\_sended = true;  
 }  
 **return** data\_is\_sended;  
}  
  
// Отключение клиента по конкретному хосту и порту  
**bool** TcpServer::disconnectBy(**uint32\_t** host, **uint16\_t** port) {  
 **bool** client\_is\_disconnected = false;  
 **for**(std::unique\_ptr<Client>& client : client\_list)  
 **if**(client->getHost() == host &&  
 client->getPort() == port) {  
 client->disconnect();  
 client\_is\_disconnected = true;  
 }  
 **return** client\_is\_disconnected;  
}  
  
// Отключение всех клиентов  
**void** TcpServer::disconnectAll() {  
 **for**(std::unique\_ptr<Client>& client : client\_list)  
 client->disconnect();  
}  
  
// Цикл обработки входящих подключений  
// (исполняется в отдельном потоке)  
**void** TcpServer::handlingAcceptLoop() {  
 **SockLen\_t** addrlen = **sizeof**(SocketAddr\_in);  
 // Пока сервер запущен  
 **while** (\_status == status::up) {  
 SocketAddr\_in client\_addr;  
 // Принятеи новго подключения (блокирующи вызов)  
 **if** (Socket client\_socket = accept(serv\_socket, (**struct** sockaddr\*)&client\_addr, &addrlen);  
 client\_socket WIN(!= 0)NIX(>= 0) && \_status == status::up) {  
 // Если получен сокет с ошибкой продолжить ожидание  
 **if**(client\_socket == WIN(INVALID\_SOCKET)NIX(-1)) **continue**;  
  
 // Активировать Keep-Alive для клиента  
 **if**(!enableKeepAlive(client\_socket)) {  
 shutdown(client\_socket, 0);  
 WIN(closesocket)NIX(close)(client\_socket);  
 }  
  
 std::unique\_ptr<Client> client(**new** Client(client\_socket, client\_addr));  
 // Запустить обработчик подключений  
 connect\_hndl(\*client);  
 // Добавить клиента в список клиентов  
 client\_mutex.lock();  
 client\_list.emplace\_back(std::move(client));  
 client\_mutex.unlock();  
 }  
 }  
  
}  
  
// Цикл ожидания данных  
**void** TcpServer::waitingDataLoop() {  
 **using** **namespace** std::chrono\_literals;  
 **while** (true) {  
 client\_mutex.lock();  
 // Перебрать всех клиентов  
 **for**(**auto** it = client\_list.begin(), end = client\_list.end(); it != end; ++it) {  
 **auto**& client = \*it;  
 // Если unique\_ptr содержит объект клиента  
 **if**(client){  
 **if**(DataBuffer data = client->loadData(); data.size) {  
 // При наличии данных запустить обработку входящих данных в отдельном потоке  
 std::thread([**this**, \_data = std::move(data), &client]{  
 client->access\_mtx.lock();  
 handler(\_data, \*client);  
 client->access\_mtx.unlock();  
 }).detach();  
 } **else** **if**(client->\_status == SocketStatus::disconnected) {  
 // При отключении клиента запустить обработку в отдельном потоке  
 std::thread([**this**, &client, it]{  
 // Извлечь объект клиента из unique\_ptr в списке  
 client->access\_mtx.lock();  
 Client\* pointer = client.release();  
 client = nullptr;  
 pointer->access\_mtx.unlock();  
 // Запуск обработчика отключения  
 disconnect\_hndl(\*pointer);  
 // Удалить элемент клиента из списка  
 client\_list.erase(it);  
 // Удалить объект клиента  
 **delete** pointer;  
 }).detach();  
 }  
 }  
 }  
 client\_mutex.unlock();  
 // Ожидание 50 млисекунд так как в данном потоке  
 // не содержится блокирующих вызовов и данный  
 // цикл сильно повышает загруженность CPU  
 std::this\_thread::sleep\_for(50ms);  
 }  
}  
  
// Функция запуска и конфигурации Keep-Alive для сокета  
**bool** TcpServer::enableKeepAlive(Socket socket) {  
 **int** flag = 1;  
#ifdef \_WIN32  
 tcp\_keepalive ka {1, ka\_conf.ka\_idle \* 1000, ka\_conf.ka\_intvl \* 1000};  
 **if** (setsockopt (socket, SOL\_SOCKET, SO\_KEEPALIVE, (**const** **char** **\***) &flag, **sizeof**(flag)) != 0) **return** false;  
 **unsigned** **long** numBytesReturned = 0;  
 **if**(WSAIoctl(socket, SIO\_KEEPALIVE\_VALS, &ka, **sizeof** (ka), nullptr, 0, &numBytesReturned, 0, nullptr) != 0) **return** false;  
#else //POSIX  
 **if**(setsockopt(socket, SOL\_SOCKET, SO\_KEEPALIVE, &flag, **sizeof**(flag)) == -1) **return** false;  
 **if**(setsockopt(socket, IPPROTO\_TCP, TCP\_KEEPIDLE, &ka\_conf.ka\_idle, **sizeof**(ka\_conf.ka\_idle)) == -1) **return** false;  
 **if**(setsockopt(socket, IPPROTO\_TCP, TCP\_KEEPINTVL, &ka\_conf.ka\_intvl, **sizeof**(ka\_conf.ka\_intvl)) == -1) **return** false;  
 **if**(setsockopt(socket, IPPROTO\_TCP, TCP\_KEEPCNT, &ka\_conf.ka\_cnt, **sizeof**(ka\_conf.ka\_cnt)) == -1) **return** false;  
#endif  
 **return** true;  
}

Реализация для Linux и Windows практически идентична за исключением некотрых мест который без проблем обкладываются макросами. Теперь же мы перейдём непосредственно к реализации класса клиента:

**TcpServerClient.cpp**

#include "../include/TcpServer.h"  
  
#ifdef \_WIN32  
// Макросы для выражений зависимых от OS  
#define WIN(exp) exp  
#define NIX(exp)  
  
// Конвертировать WinSocket код ошибки в Posix код ошибки  
**inline** **int** convertError() {  
 **switch** (WSAGetLastError()) {  
 **case** 0:  
 **return** 0;  
 **case** WSAEINTR:  
 **return** EINTR;  
 **case** WSAEINVAL:  
 **return** EINVAL;  
 **case** WSA\_INVALID\_HANDLE:  
 **return** EBADF;  
 **case** WSA\_NOT\_ENOUGH\_MEMORY:  
 **return** ENOMEM;  
 **case** WSA\_INVALID\_PARAMETER:  
 **return** EINVAL;  
 **case** WSAENAMETOOLONG:  
 **return** ENAMETOOLONG;  
 **case** WSAENOTEMPTY:  
 **return** ENOTEMPTY;  
 **case** WSAEWOULDBLOCK:  
 **return** EAGAIN;  
 **case** WSAEINPROGRESS:  
 **return** EINPROGRESS;  
 **case** WSAEALREADY:  
 **return** EALREADY;  
 **case** WSAENOTSOCK:  
 **return** ENOTSOCK;  
 **case** WSAEDESTADDRREQ:  
 **return** EDESTADDRREQ;  
 **case** WSAEMSGSIZE:  
 **return** EMSGSIZE;  
 **case** WSAEPROTOTYPE:  
 **return** EPROTOTYPE;  
 **case** WSAENOPROTOOPT:  
 **return** ENOPROTOOPT;  
 **case** WSAEPROTONOSUPPORT:  
 **return** EPROTONOSUPPORT;  
 **case** WSAEOPNOTSUPP:  
 **return** EOPNOTSUPP;  
 **case** WSAEAFNOSUPPORT:  
 **return** EAFNOSUPPORT;  
 **case** WSAEADDRINUSE:  
 **return** EADDRINUSE;  
 **case** WSAEADDRNOTAVAIL:  
 **return** EADDRNOTAVAIL;  
 **case** WSAENETDOWN:  
 **return** ENETDOWN;  
 **case** WSAENETUNREACH:  
 **return** ENETUNREACH;  
 **case** WSAENETRESET:  
 **return** ENETRESET;  
 **case** WSAECONNABORTED:  
 **return** ECONNABORTED;  
 **case** WSAECONNRESET:  
 **return** ECONNRESET;  
 **case** WSAENOBUFS:  
 **return** ENOBUFS;  
 **case** WSAEISCONN:  
 **return** EISCONN;  
 **case** WSAENOTCONN:  
 **return** ENOTCONN;  
 **case** WSAETIMEDOUT:  
 **return** ETIMEDOUT;  
 **case** WSAECONNREFUSED:  
 **return** ECONNREFUSED;  
 **case** WSAELOOP:  
 **return** ELOOP;  
 **case** WSAEHOSTUNREACH:  
 **return** EHOSTUNREACH;  
 **default**:  
 **return** EIO;  
 }  
}  
  
#else  
// Макросы для выражений зависимых от OS  
#define WIN(exp)  
#define NIX(exp) exp  
#endif  
  
#include <iostream>  
  
// Реализация загрузки данных  
DataBuffer TcpServer::Client::loadData() {  
 // Если клиент не подключён вернуть пустой буффер  
 **if**(\_status != SocketStatus::connected) **return** DataBuffer();  
 **using** **namespace** std::chrono\_literals;  
 DataBuffer buffer;  
 **int** err;  
  
 // Чтение длинный данных в неблокирующем режиме  
 // MSG\_DONTWAIT - Unix флаг неблокирующего режима для recv  
 // FIONBIO - Windows-флаг неблокирующего режима для ioctlsocket  
 WIN(**if**(u\_long t = true; SOCKET\_ERROR == ioctlsocket(socket, FIONBIO, &t)) **return** DataBuffer();)  
 **int** answ = recv(socket, (**char\***)&buffer.size, **sizeof** (buffer.size), NIX(MSG\_DONTWAIT)WIN(0));  
  
 // Обработка отключения  
 **if**(!answ) {  
 disconnect();  
 **return** DataBuffer();  
 } **else** **if**(answ == -1) {  
 // Чтение кода ошибки  
 WIN(  
 err = convertError();  
 **if**(!err) {  
 **SockLen\_t** len = **sizeof** (err);  
 getsockopt (socket, SOL\_SOCKET, SO\_ERROR, WIN((**char\***))&err, &len);  
 }  
 )NIX(  
 **SockLen\_t** len = **sizeof** (err);  
 getsockopt (socket, SOL\_SOCKET, SO\_ERROR, WIN((**char\***))&err, &len);  
 **if**(!err) err = errno;  
 )  
  
 // Отключение неблокирующего режима для Windows  
 WIN(**if**(u\_long t = false; SOCKET\_ERROR == ioctlsocket(socket, FIONBIO, &t)) **return** DataBuffer();)   
  
 // Обработка ошибки при наличии  
 **switch** (err) {  
 **case** 0: **break**;  
 // Keep alive timeout  
 **case** ETIMEDOUT:  
 **case** ECONNRESET:  
 **case** EPIPE:  
 disconnect();  
 [[fallthrough]];  
 // No data  
 **case** EAGAIN: **return** DataBuffer();  
 **default**:  
 disconnect();  
 std::cerr << "Unhandled error!\n"  
 << "Code: " << err << " Err: " << std::strerror(err) << '\n';  
 **return** DataBuffer();  
 }  
 }  
  
 // Если прочитанный размер нулевой, то вернуть пустой буффер  
 **if**(!buffer.size) **return** DataBuffer();  
 // Если размер не нулевой выделить буффер в куче для чтения данных  
 buffer.data\_ptr = (**char\***)malloc(buffer.size);  
 // Чтение данных в блокирующем режиме  
 recv(socket, (**char\***)buffer.data\_ptr, buffer.size, 0);  
 // Возврат буффера с прочитанными данными  
 **return** buffer;  
}  
  
// Обработка отключения клиента  
TcpClientBase::status TcpServer::Client::disconnect() {  
 \_status = status::disconnected;  
 // Если сокет не валидный прекратить обработку  
 **if**(socket == WIN(INVALID\_SOCKET)NIX(-1)) **return** \_status;  
 // Отключение сокета  
 shutdown(socket, SD\_BOTH)  
 // Закрытие сокета  
 WIN(closesocket)NIX(close)(socket);  
 // Установление в сокета не валидного значения  
 socket = WIN(INVALID\_SOCKET)NIX(-1);  
 **return** \_status;  
}  
  
// Отправка данных  
**bool** TcpServer::Client::sendData(**const** **void\*** buffer, **const** **size\_t** size) **const** {  
 // Если сокет закрыт вернуть false  
 **if**(\_status != SocketStatus::connected) **return** false;  
 // Сформировать сообщение с длинной в начале  
 **void\*** send\_buffer = malloc(size + **sizeof** (**int**));  
 memcpy(**reinterpret\_cast**<**char\***>(send\_buffer) + **sizeof**(**int**), buffer, size);  
 \***reinterpret\_cast**<**int\***>(send\_buffer) = size;  
  
 // Отправить сообщение  
 **if**(send(socket, **reinterpret\_cast**<**char\***>(send\_buffer), size + **sizeof** (**int**), 0) < 0) **return** false;  
 // Вычистить буффер сообщения  
 free(send\_buffer);  
 **return** true;  
}  
  
// Конструктор клиента  
TcpServer::Client::Client(Socket socket, SocketAddr\_in address)  
 : address(address), socket(socket) {}  
  
// Деструктор клиента с закрытием сокета  
TcpServer::Client::~Client() {  
 **if**(socket == WIN(INVALID\_SOCKET)NIX(-1)) **return**;  
 shutdown(socket, SD\_BOTH);  
 WIN(closesocket(socket);)  
 NIX(close(socket);)  
}  
  
// Получить хост клиента  
**uint32\_t** TcpServer::Client::getHost() **const** {**return** NIX(address.sin\_addr.s\_addr) WIN(address.sin\_addr.S\_un.S\_addr);}  
// Получить порт клиента  
**uint16\_t** TcpServer::Client::getPort() **const** {**return** address.sin\_port;}

Пример использования:

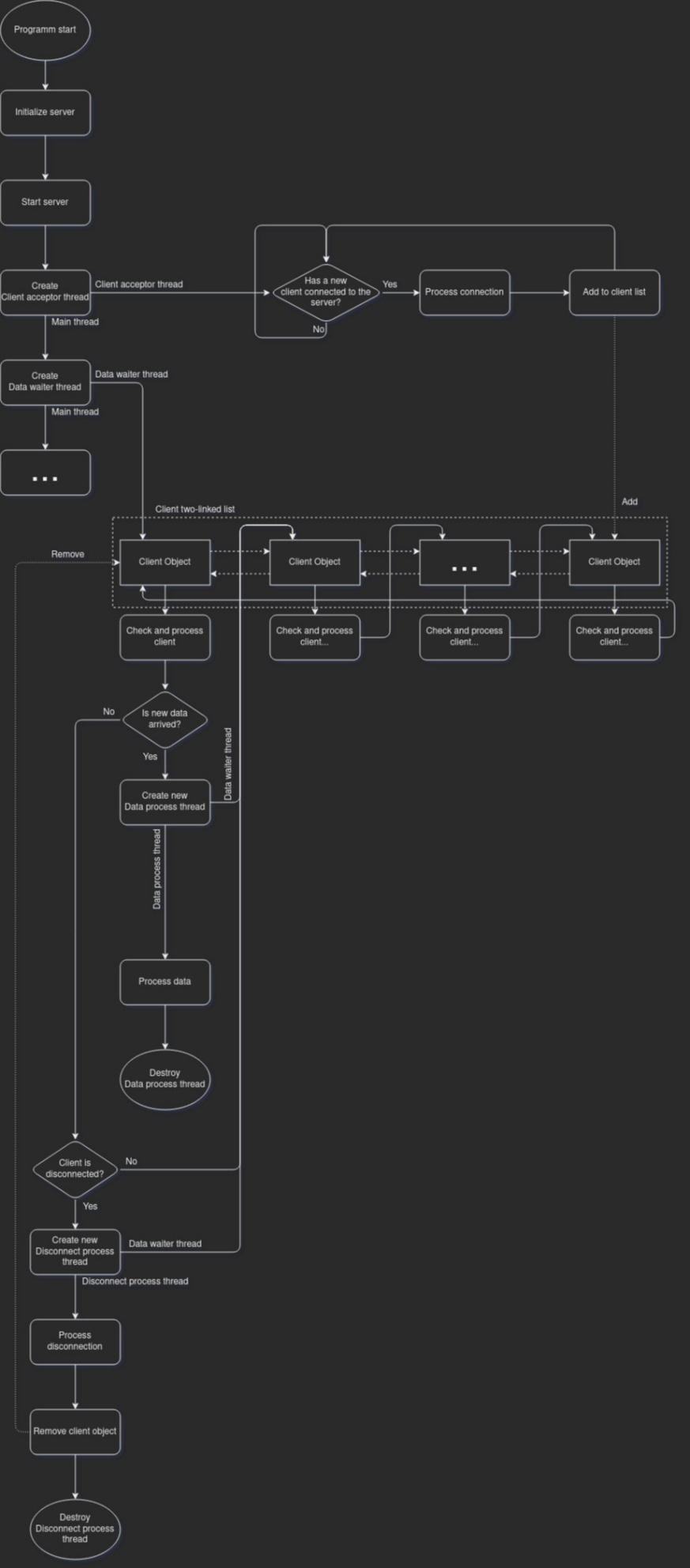
**main.cpp**

#include <iostream>  
  
// Парсинр IPv4-адреса и порта в std::string  
std::string getHostStr(**const** TcpServer::Client& client) {  
**uint32\_t** ip = client.getHost ();  
**return** std::string() + std::to\_string(**int**(**reinterpret\_cast**<**char\***>(&ip)[0])) + '.' +  
 std::to\_string(**int**(**reinterpret\_cast**<**char\***>(&ip)[1])) + '.' +  
 std::to\_string(**int**(**reinterpret\_cast**<**char\***>(&ip)[2])) + '.' +  
 std::to\_string(**int**(**reinterpret\_cast**<**char\***>(&ip)[3])) + ':' +  
 std::to\_string( client.getPort ());  
}  
  
**int** main() {  
 // Создание экземпляра сервера  
 TcpServer server( 8080, [](DataBuffer data, TcpServer::Client& client){  
 std::cout << "("<<getHostStr(client)<<")[ " << data.size << " bytes ]: " << (**char\***)data.data\_ptr << '\n';  
 client.sendData("Hello, client!", **sizeof**("Hello, client!"));  
 }, {1, 1, 1}); // Keep alive{ожидание:1s, интервал: 1s, кол-во пакетов: 1};  
  
 // Запуск сервера  
 **if**(server.start() == TcpServer::status::up) {  
 std::cout << "Server is up!" << std::endl;  
 server.joinLoop(); //Joing to the client handling loop  
 } **else** {  
 std::cout << "Server start error! Error code:" << **int**(server.getStatus()) << std::endl;  
 **return** -1;  
 }  
  
}

Источники:

* [Программирование сокетов в Linux](https://www.rsdn.org/article/unix/sockets.xml)
* [Простой и быстрый сервер на C/C++...](https://habr.com/ru/post/327574/)
* [Winsock Server Application](https://docs.microsoft.com/en-us/windows/win32/winsock/winsock-server-application)
* [Проект OpenNet: MAN socket (7) Макропакеты и соглашения (FreeBSD и Linux)](https://www.opennet.ru/man.shtml?topic=socket&category=7&russian=0)

**UPDATE:** Реализация многопоточного TCP-сервера представленная в данной статье имеет такой недостаток как "потокове голодание" при слишком большом колличестве подключённых к серверу клиентов, по скольку для обработки каждого клиента создаётся отдельный поток, который находится в состоянии активного ожидания данных от клиента. Для уменьшения влияния данного недостатка было принято решение изменить модель организации многопоточности приложения. Новая модель представлена на следующей схеме:



Как можно заметить, при данной модели организации многопоточности имеется всего один поток-обходчик, который проходит все клиенты и проверяет наличие пришедших от них данных. При условии найденых данных поток-обходчик переходит в состояние обработки данных клиента предварительно создав новый поток-обходчик, который идёт дальше. После обработки данных поток-обработчик переносит объект клиента в конец очереди ожидания и самоуничтожается.

Реализацию с применением данного исправления можно увидеть [в данном репозитории GitHub](https://github.com/gbytegear/TcpServer). Данный сервер рассчитан только для отправки бинарных данных по протоколу TCP, поскольку отправляет вместе с данными заголовок с их размером в байтах, так что для реализации HTTP-сервера данная реализация не подходит. Реализованно это именно так по причине того что TCP/IP — потоковый протокол, то есть данные идущие через TCP/IP никак не терминируются, что осложняет работу именно в случаях работы с сырыми данными, когда поток нельзя терминировать "завершающим символом", как это принято в текстовых потоках.

**UPDATE 2:** На текущий момент, реализация представленная в репозитории GitHub лишена недостатка с "потоковым голоданием". Операции чтения префикса размера бинарного пакета при получении его от клиента, а так же приём новых соединений переведены в неблокирующий режим, а многопоточность реализована с помощью простой реализации пула потоков.