|  |  |
| --- | --- |
| ДИСЦИПЛИНА | Технологии разработки серверных приложений |
| ИНСТИТУТ | ИПТИП |
| КАФЕДРА | Индустриального программирования |
| ВИД УЧЕБНОГО МАТЕРИАЛА | Методические указания к практическим занятиям по дисциплине |
| ПРЕПОДАВАТЕЛЬ | Макиевский Станислав Евгеньевич |
| СЕМЕСТР | 4 семестр, 2024/2025 уч. год |

# Практическая работа №7

## Сетевое взаимодействие между программами

В настоящее время трудно представить себе такую программу, которая работала бы полностью в формате «оффлайн», т.е. вообще не требовала бы подключения к интернету для выполнения каких-либо функций.

Даже если весь функционал программы доступен без необходимости подключения к интернету, то, как минимум, для активации лицензии, необходимо единоразово подключиться к интернету.

Также, стоит обратить внимание, что многие программы или сервисы имеют между собой интеграцию, например, для регистрации в сервисе пользователь может использовать аккаунт Яндекс, ВКонтакте и т.п.

Другой особенностью многих современных программ и сервисов является разделение функционала на несколько отдельных сервисов – «микросервисов» – суть заключается в разделении всего функционала системы на небольшие сервисы, взаимодействующие между собой, которые можно масштабировать при необходимости или добавлять, удалять или заменять.

Что объединяет все эти особенности? Все они используют интернет для взаимодействия – запрос на сервер для проверки лицензии, запрос на сервис Яндекса, ВКонтакте и т.п. для авторизации, запрос от одного микросервиса к другому и т.д. – все перечисленные методы выполняют запросы через интернет для получения или отправки информации. Но как они это делают?

Для понимания, как передаётся информация в интернете, необходимо обратиться к модели OSI:

Модель OSI (Open Systems Interconnection) — это эталонная модель, разработанная для описания функций телекоммуникационных или вычислительных систем, необходимых для сетевого взаимодействия. При этом сама по себе эталонная модель – не стандарт интернета, – её можно сравнить с фреймворками в языках программирования – в OSI «из коробки» доступны разные веб-стандарты — UDP, HTTP, FTP, Telnet и другие.

Сама модель OSI разделяет процесс сетевого взаимодействия на семь взаимосвязанных уровней, каждый из которых выполняет определённую функцию, например, передать данные или представить их в понятном для человека виде и т.п.

Представить модель OSI можно следующим способом:

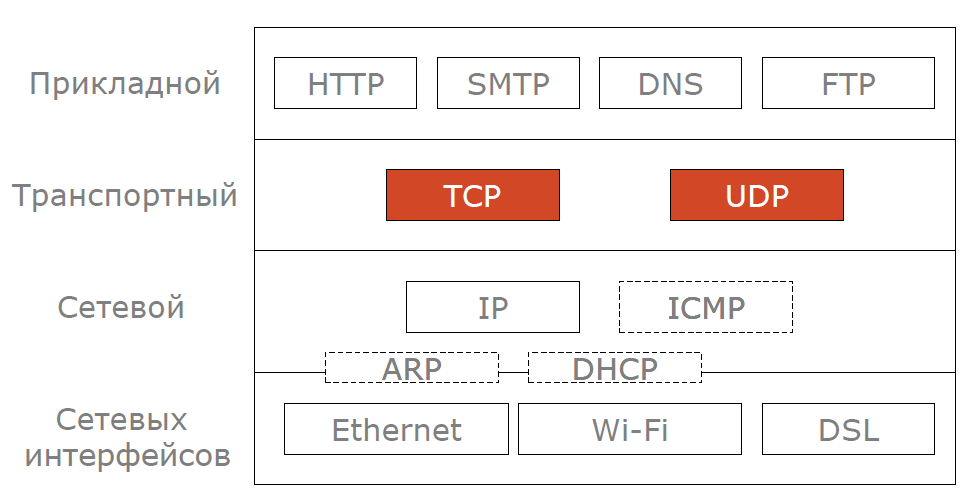


Семислойная модель OSI

Рассмотрим уровни модели OSI и их функции:

1. Физический уровень (L1, physical layer). На самом нижнем уровне модели OSI данные представляют собой физические объекты — ток, свет или радиоволны. Они передаются по проводам или с помощью беспроводных сигналов.
2. Канальный уровень (L2, data link layer). Здесь данные должны преобразоваться в унифицированный формат, чтобы их можно было передавать на разные устройства и операционные системы.
3. Сетевой уровень (L3, Network Layer). На этом уровне происходит передача информации между различными локальными сетями с помощью таких устройств, как маршрутизатор.
4. Транспортный уровень (L4, transport layer). Обеспечивает надёжную передачу данных, контроль ошибок, сегментацию и повторную сборку данных. Гарантирует, что данные передаются без потерь и дубликатов.
5. Сеансовый уровень (L5, session layer). Устанавливает и завершает сеансы связи между устройствами, обеспечивает синхронизацию, управление длительностью соединения и восстановление после сбоев.
6. Уровень представления данных (L6, presentation layer). Преобразует информацию в стандартизированные форматы, чтобы гарантировать совместимость между отдельными системами.
7. Прикладной уровень (L7, application layer). Осуществляется взаимодействие пользователя с приложениями, обеспечивая веб-сёрфинг, электронную почту и передачу файлов.

На 4 уровне модели OSI, т.е. на транспортном уровне, реализуются протоколы TCP, UDP и другие.



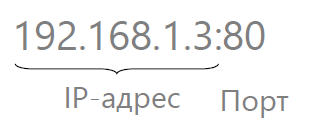
Протоколы уровней.

Основными задачами транспортного уровня являются:

* Передача данных между процессами на хостах;
* Адресация запросов;
* Предоставление нужного уровня надёжности передачи данных, независимо от надёжности сети.

Каждое сетевое приложение на хосте имеет свой порт – число от 1 до 65535. Если одно приложение заняло конкретный порт, то для других приложений порт становится недоступным. При попытке занятия уже занятого порта происходит ошибка.

Формат записи адреса выглядит следующим образом:

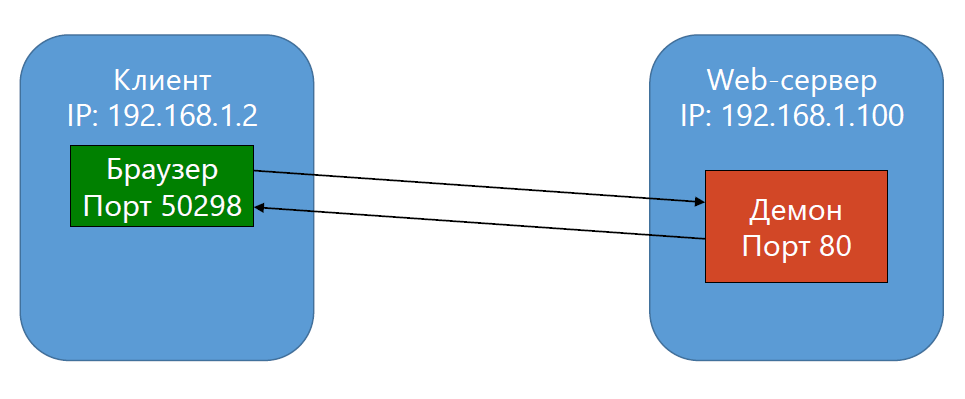


Некоторые порты забронированы под определённые протоколы, например:

* 80 – HTTP;
* 25 – SMTP (электронная почта);
* 67, 68 – DHCP
* 21 – FTP;
* 53 – DNS;
* Другие порты в диапазоне 1-1024.

Обычно данные забронированные порты используют сервера, чтобы стандартизировать процесс подключения.

Для клиентских приложений, порты обычно назначаются системой в диапазоне 49151 – 65535. Например, вот так:



Взаимодействие браузера с HTTP сервером.

Благодаря этим протоколам 2 и более приложений могут взаимодействовать между собой:



Точки сетевого взаимодействия в системе называются сокетами.

Сокет в транспортном уровне – это программный интерфейс, используемый для связи между компьютерами, с использованием стека протоколов TCP/IP.

Каждый сокет определяется парой адреса и порта, где адрес может быть 32-битной структурой для протокола IPv4 или 128-битной для IPv6, а номер порта – целым числом в диапазоне от 0 до 65535. Взаимодействие осуществляется через два сокета - сокет отправителя и сокет получателя.

Так, например, открывая веб-сайт, мы делаем HTTP запрос в браузере к определённому сайту. Например, отправляя запрос <http://www.mysite.com> мы видим страницу сайта, однако на самом деле браузер отправляет запрос на сервер по его IP адресу и порту 80, а сервер в ответ на данный запрос отправляет HTML страницу на Ваш IP адрес и тот порт, на котором он открыт.

В таком случае взаимодействие происходит по принципу клиент-сервер:

Клиент — это устройство или программа, которая запрашивает услуги или данные у сервера. Клиент может быть веб-браузером, мобильным приложением или любой другой программой, которая взаимодействует с сервером.

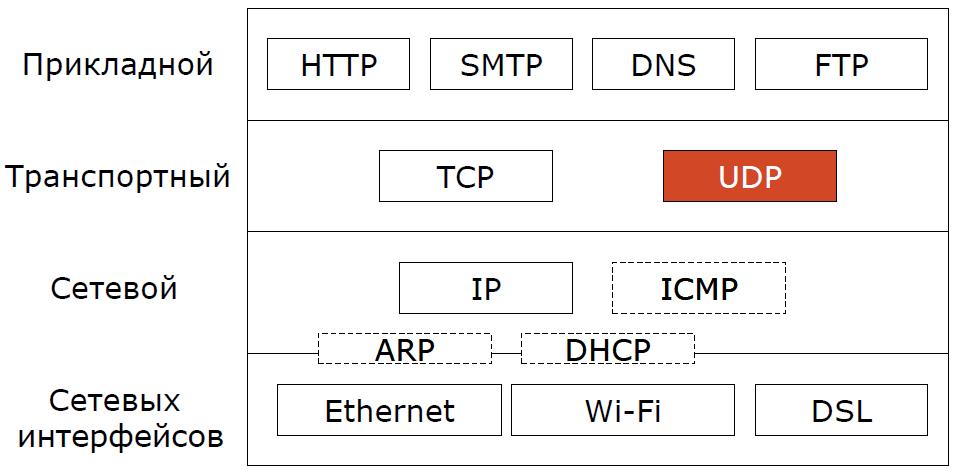
Сервер — это мощное устройство или программа, которая предоставляет услуги или данные клиентам. Серверы могут обрабатывать множество запросов одновременно.

В данном случае, сторона фронтэнда является клиентом, а сторона бэкенда – сервером. Однако, возможна и ситуация, когда бэкенд становится клиентом, например, если необходимо выполнить авторизацию с помощью аккаунта Яндекс, в таком случае бэкенд становится клиентом, отправляющим запрос на сервер Яндекса.

Таким образом, клиент – это сторона, которая запрашивает данные, а сервер – сторона, которая эти данные отдаёт.

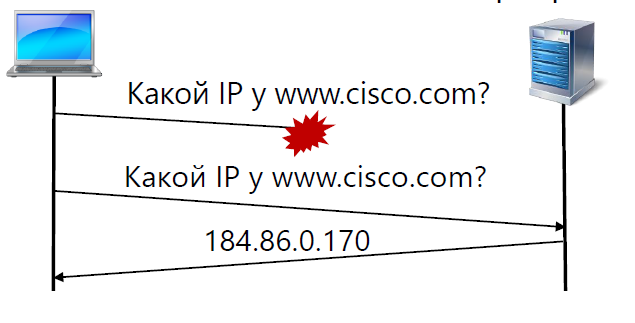
## Протокол UDP

Протокол UDP – User Datagram Protocol (UDP) – протокол дейтаграмм (сообщений) пользователя. Он предназначен для передачи данных с минимальной задержкой, избегая ряда сложностей и издержек, связанных с проверкой целостности и правильностью передачи данных.



Протокол UDP не требует создания соединения перед отправкой данных. Это значит, что в данном протоколе отсутствуют такие понятия «клиент» и «сервер». Пакеты, в данном случае, отправляются независимо, что позволяет минимизировать задержки.

Также протокол UDP не гарантирует, что пакеты будут доставлены, не будут потеряны в процессе доставки, не продублируются или придут в правильном порядке – протокол не гарантирует данных возможностей и не предоставляет механизмы для исправления таких ошибок. Так, если пакет не дошёл, то отправителю необходимо самостоятельно отправить пакет повторно:



Однако, за счет отсутствия таких особенностей, как контроль ошибок и установление соединения, протокол UDP обеспечивает более быструю передачу данных по сравнению с TCP.

Скорости протокола также способствуют низкие накладные расходы – пакеты UDP содержат минимальный заголовок в 8 байт, что оставляет больше пространства для собственно данных.

Также, при работе с протоколом UDP стоит учитывать его технические особенности:

1. Отправитель не знает, достигли ли данные получателя в целости.
2. Из-за отсутствия механизмов повторной отправки пропавших пакетов протокол не подходит для критически важных приложений, требующих гарантированной доставки.
3. Ограничение длины пакета: Размер датаграммы UDP ограничен 65,507 байт для IPv4 и больше для IPv6.

Исходя из этого можно обозначить основные области применения протокола UDP:

1. Потоковое мультимедиа – передача аудио и видео, где небольшие потери данных скорее всего незаметны и более быстрая доставка предпочтительна (например, если в видео будет пропущен 1-2 кадра, зритель, скорее всего, этого даже не заметит).
2. В онлайн играх, где требуется минимальная задержка, а небольшие потери пакетов могут быть некритичными (например, передача местоположения игрока – можно допустить потерю одного пакета в пользу более высокой скорости обновления позиции).
3. Сетевые протоколы, такие, как DNS и DHCP используют UDP, поскольку они требуют быстрой передачи и не всегда чувствительны к потерям данных.

Таким образом, протокол UDP стоит применять в тех случаях, когда скорость важнее надежности.

**Примеры реализация протокола UDP**

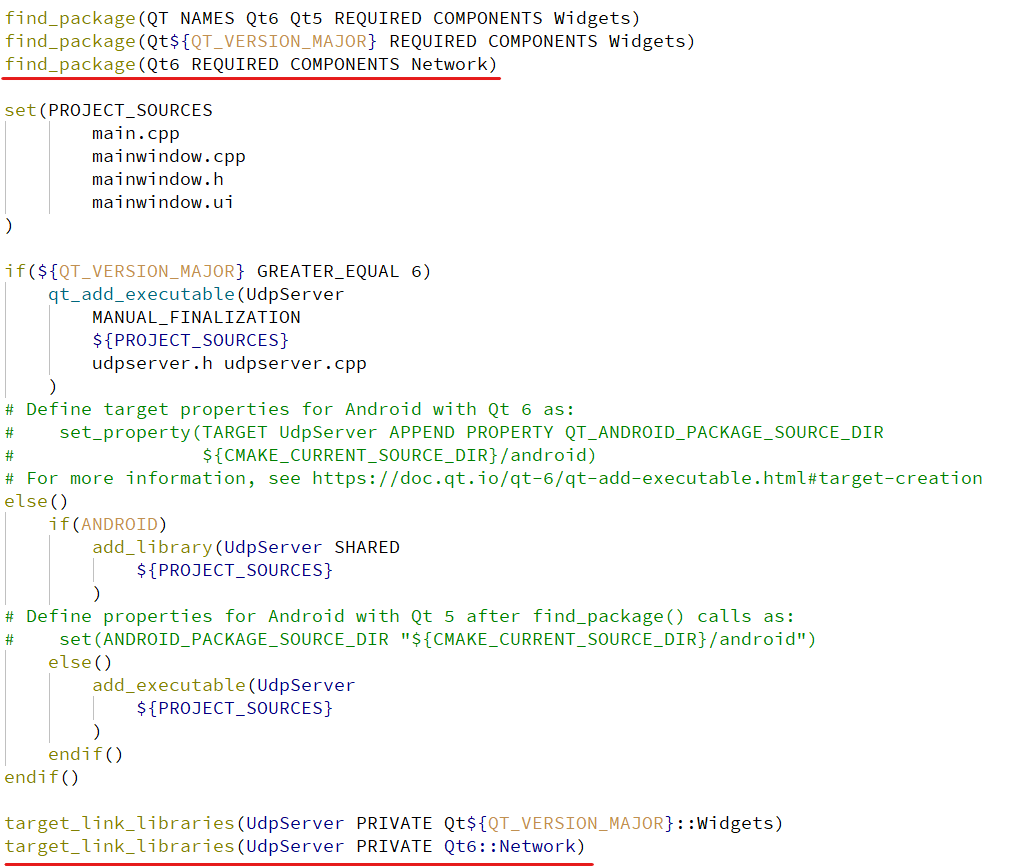
Рассмотрим примеры на разных языках программирования:

Для реализации «сервера» на языке C++ потребуется применение библиотеки Qt. Для этого, после создания проекта в CMakeLists.txt необходимо добавить строки:

find\_package(Qt6 REQUIRED COMPONENTS Network)

target\_link\_libraries(UdpServer PRIVATE Qt6::Network)

Где UdpServer нужно заменить на название вашего проекта:



CmakeLists.txt

В случае использования qmake необходимо добавить QT += network в файл \*.pro

Далее создадим класс для сервера со следующим кодом:



#ifndef UDPSERVER\_H

#define UDPSERVER\_H

#include <QUdpSocket>

#include <QNetworkDatagram>

class UdpServer : public QObject {

Q\_OBJECT

public:

UdpServer() {

socket = new QUdpSocket(this);

socket->bind(QHostAddress::Any, 1234);

connect(socket, &QUdpSocket::readyRead, this, &UdpServer::onReadyRead);

}

private slots:

void onReadyRead() {

while (socket->hasPendingDatagrams()) {

QNetworkDatagram datagram = socket->receiveDatagram();

qDebug() << "Received:" << datagram.data();

}

}

private:

QUdpSocket \*socket;

};

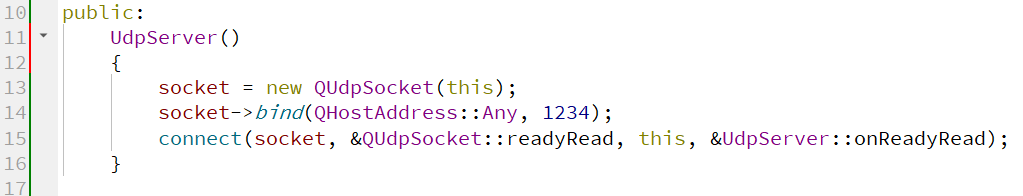
#endif // UDPSERVER\_H

В данном коде мы создаём объект UdpSocket:

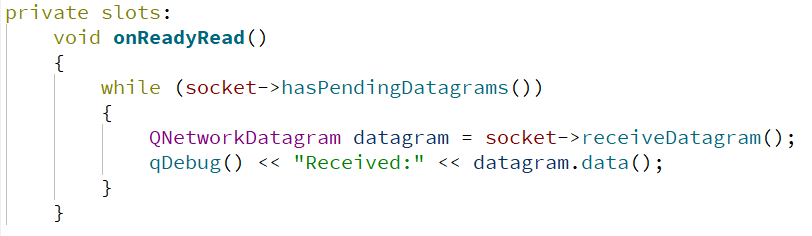
private:

QUdpSocket \*socket;

Инициализируем прослушивание сокетом порта 1234 нашей системы:



После чего обрабатываем все входящие сообщения, выводя их текст на экран:



Таким образом можно реализовать простой «сервер» для протокола UDP в языке С++.

Клиент в свою очередь создаётся аналогичным образом. Аналогичным образом, после создания проекта, нужно добавить необходимые строки в CMakeLists.txt:

find\_package(Qt6 REQUIRED COMPONENTS Network)

target\_link\_libraries(UdpServer PRIVATE Qt6::Network)

А в случае использования qmake необходимо добавить QT += network в файл \*.pro.

Далее создадим «клиента». Для этого создадим класс UdpClient:



#ifndef UDPCLIENT\_H

#define UDPCLIENT\_H

#include <QUdpSocket>

#include <QTimer>

#include <QHostAddress>

class UdpClient : public QObject

{

Q\_OBJECT

public:

UdpClient()

{

socket = new QUdpSocket(this);

QTimer \*timer = new QTimer(this);

connect(timer, &QTimer::timeout, this, &UdpClient::sendDatagram);

timer->start(1000); // отправка каждые 1000 мс

}

private slots:

void sendDatagram()

{

QByteArray data = "Hello, Server!";

socket->writeDatagram(data, QHostAddress::LocalHost, 1234);

}

private:

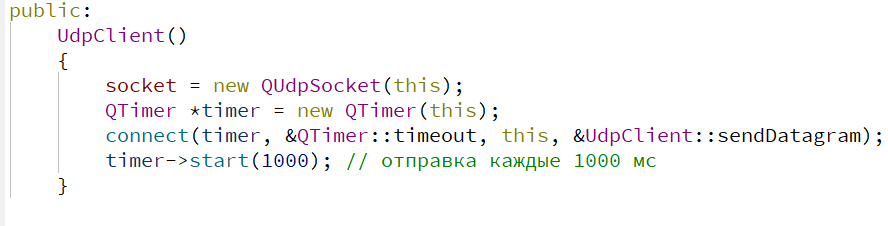
QUdpSocket \*socket;

};

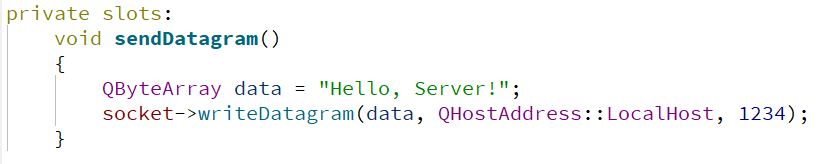
#endif // UDPCLIENT\_H

В данном коде мы также создаём объект QUdpSocket, т.к. в UDP нет понятий клиента и сервера, а есть только сокеты.

В качестве инициализации запросов на сервер, наш клиент просто вызывает каждую секунду функцию отправки на сервер сообщения:



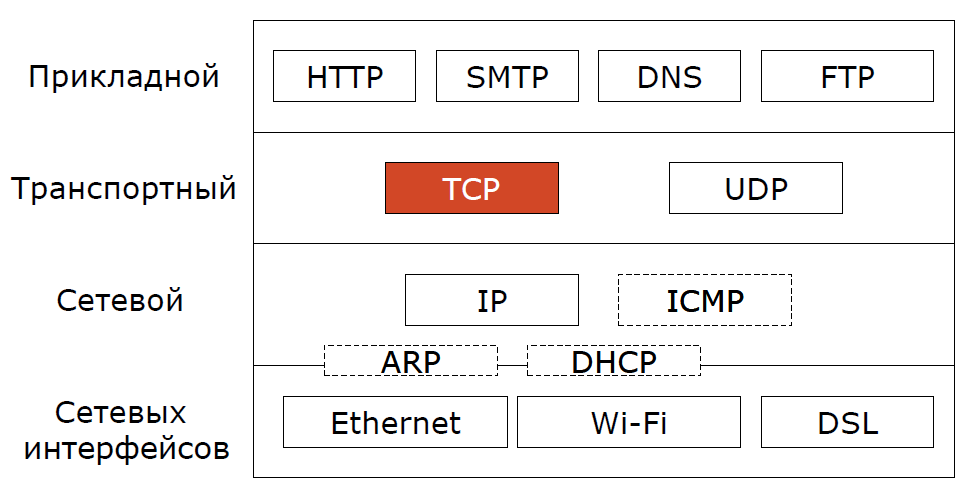
Которое просто пишет «Hello server!»:



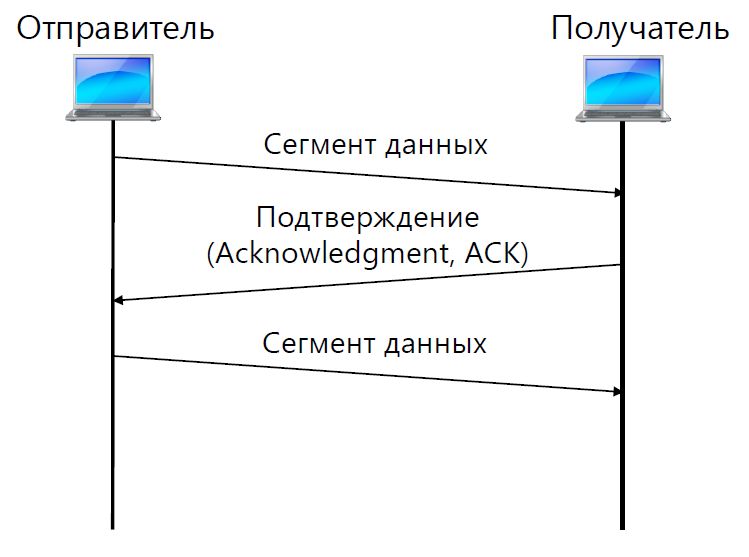
Таким образом реализуется взаимодействие между программами по протоколу UDP.

## Протокол TCP

Протокол TCP – Transmission Control Protocol (TCP) – это протокол управления передачей, который обеспечивает надежную доставку данных. В отличие от UDP, TCP гарантирует, что данные будут доставлены получателю в правильном порядке и без потерь.

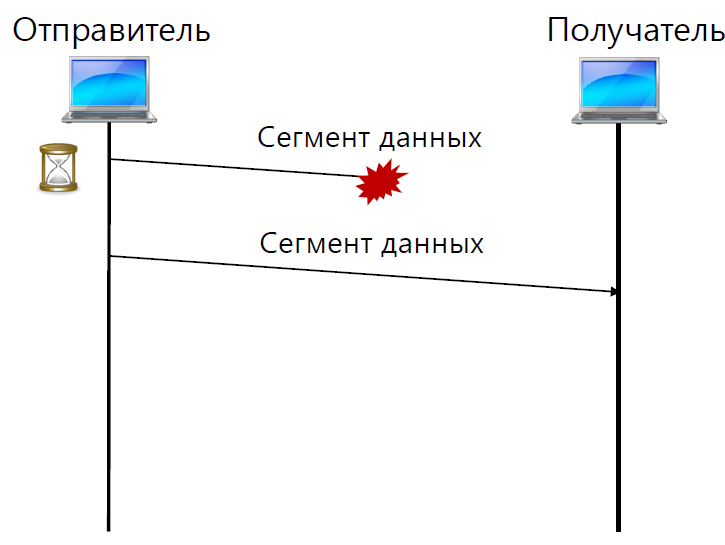


TCP требует установления соединения перед началом передачи данных. Это означает, что между отправителем и получателем создается виртуальное соединение, что позволяет гарантировать целостность и корректность доставки данных, за счёт передачи данных исключительно последовательно, с подтверждением получения текущего пакета, перед отправкой следующего:

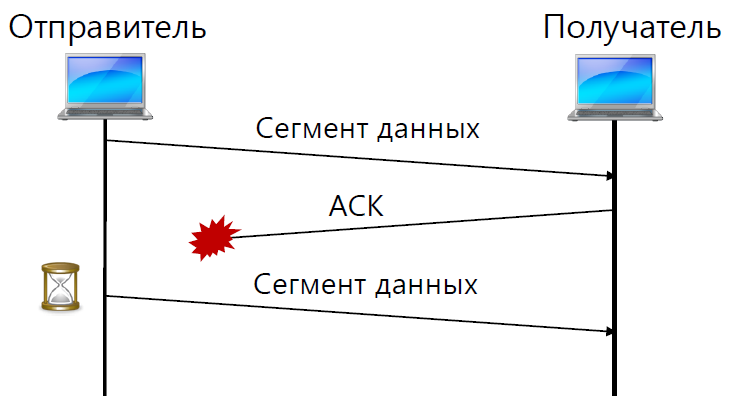


В TCP используются понятия «клиент» и «сервер», где клиент инициирует соединение, а сервер его принимает.

Когда клиент или сервер отправляют друг другу данные, они ожидают, что будет получен ответ об успешной доставки сообщения, а если ответ не пришёл в течение некоторого времени, пакет будет отправлен повторно:



При этом данные будут отправлены повторно, даже если ошибка произошла не в процессе доставки пакета, а в ответе от получателя:



Таким образом, протокол TCP гарантирует доставку данных.

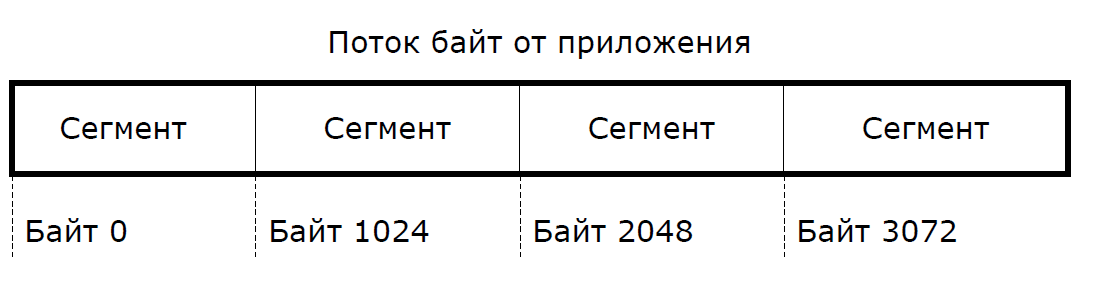
Протокол TCP обеспечивает надежность передачи данных за счет следующих механизмов:

1. Контроль ошибок: TCP использует контрольные суммы для проверки целостности данных.

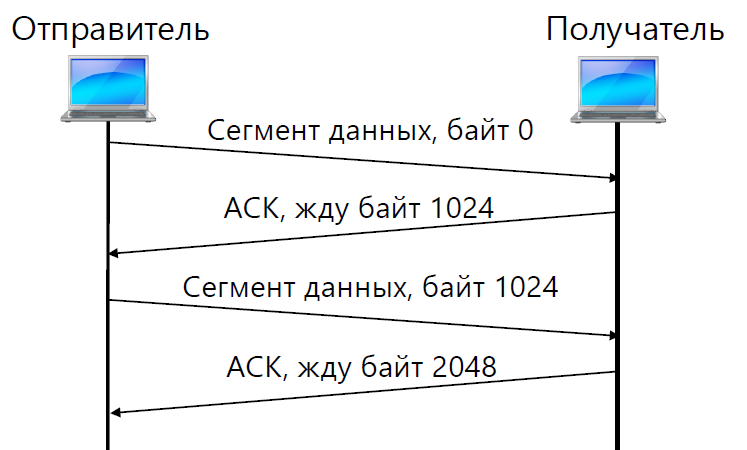
2. Управление потоком: TCP регулирует скорость передачи данных, чтобы избежать перегрузки сети.

3. Повторная отправка: в случае потери пакетов TCP автоматически отправляет их заново.

Сохранять порядок сообщений позволяет механизм нумерации следования сообщений:



Благодаря тому, что сообщения нумеруются и делятся на сегменты, получатель может запрашивать у отправителя конкретные сегменты, указывая, какие байты ему нужны:



Таким образом, если будет потерян один сегмент, отправителю не нужно начинать всю передачу повторно, а достаточно отправить потерявшиеся байты:



Эти механизмы делают протокол TCP более надежным, но увеличивают задержку и накладные расходы, по сравнению с UDP.

Технические особенности протокола TCP:

1. Гарантия доставки: Данные доставляются в правильном порядке и без потерь.

2. Установление соединения: Передача данных начинается только после установления соединения между клиентом и сервером.

3. Сложность реализации: TCP сложнее в реализации из-за необходимости управления соединением и контроля ошибок.

Основные области применения протокола TCP:

1. Веб-серфинг: HTTP и HTTPS используют TCP для обеспечения надежной передачи веб-страниц.

2. Электронная почта: Протоколы SMTP, POP3 и IMAP используют TCP для надежной доставки сообщений.

3. Передача файлов: Протоколы FTP и SFTP используют TCP для гарантии целостности передаваемых файлов.

Таким образом, протокол TCP стоит использовать в тех случаях, когда надежность важнее скорости.

**Примеры реализация протокола TCP**

Рассмотрим примеры на разных языках программирования:

#### Реализация на языке C++

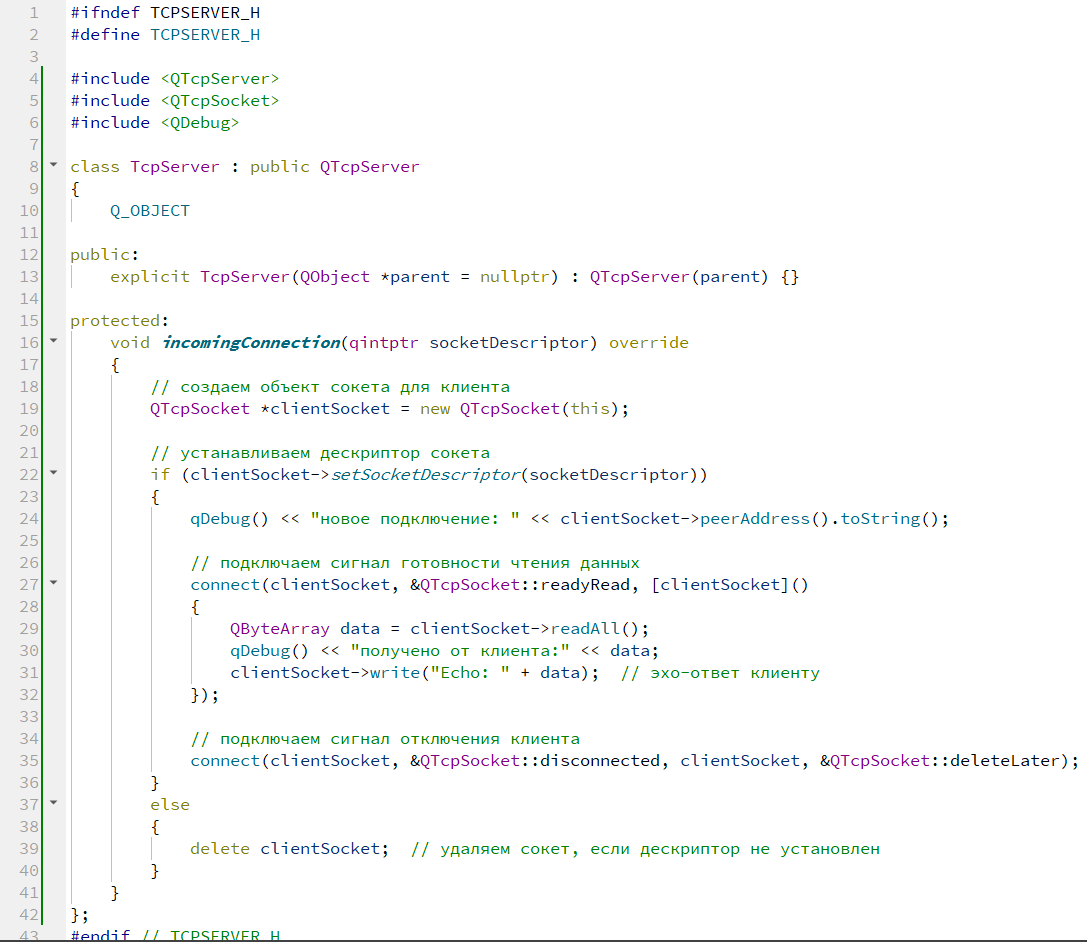
Для реализации «сервера» на языке C++ потребуется применение библиотеки Qt. Для этого, аналогично примеру UDP, после создания проекта в CMakeLists.txt необходимо добавить строки:

find\_package(Qt6 REQUIRED COMPONENTS Network)

target\_link\_libraries(TcpServer PRIVATE Qt6::Network)

Где TcpServer нужно заменить на название вашего проекта, а в случае использования qmake необходимо добавить QT += network в файл \*.pro

Далее создадим класс для сервера со следующим кодом:



#ifndef TCPSERVER\_H

#define TCPSERVER\_H

#include <QTcpServer>

#include <QTcpSocket>

#include <QDebug>

class TcpServer : public QTcpServer

{

Q\_OBJECT

public:

explicit TcpServer(QObject \*parent = nullptr) : QTcpServer(parent) {}

protected:

void incomingConnection(qintptr socketDescriptor) override

{

// создаем объект сокета для клиента

QTcpSocket \*clientSocket = new QTcpSocket(this);

// устанавливаем дескриптор сокета

if (clientSocket->setSocketDescriptor(socketDescriptor))

{

qDebug() << "новое подключение: " << clientSocket->peerAddress().toString();

// подключаем сигнал готовности чтения данных

connect(clientSocket, &QTcpSocket::readyRead, [clientSocket]()

{

QByteArray data = clientSocket->readAll();

qDebug() << "получено от клиента:" << data;

clientSocket->write("Echo: " + data); // эхо-ответ клиенту

});

// подключаем сигнал отключения клиента

connect(clientSocket, &QTcpSocket::disconnected, clientSocket, &QTcpSocket::deleteLater);

}

else

{

delete clientSocket; // удаляем сокет, если дескриптор не установлен

}

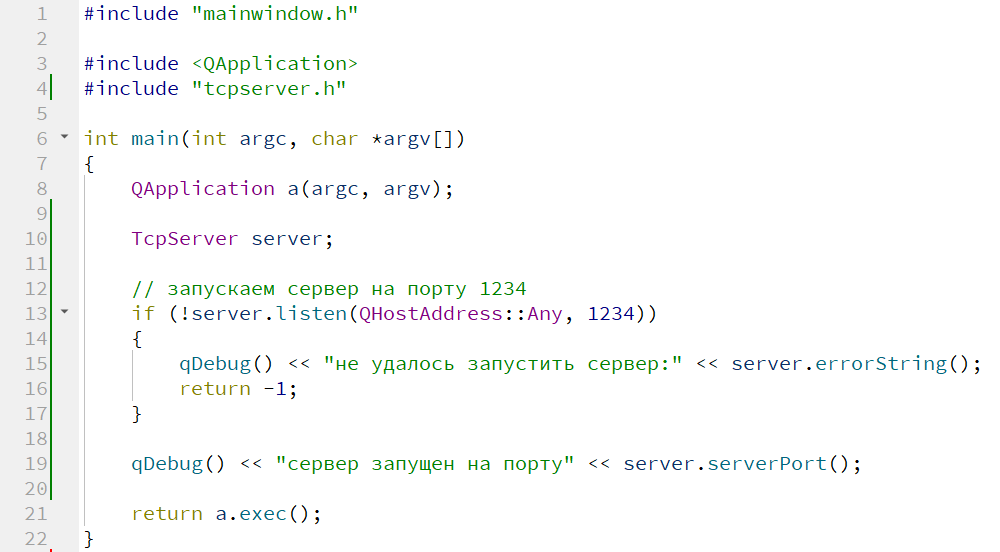
}

};

#endif // TCPSERVER\_H

В данном примере наш класс сервера наследуется от базового QTcpServer, переопределяя метод void incomingConnection(qintptr socketDescriptor), который вызывается при каждой установке нового соединения, в котором для каждого клиента добавляется функция-обработчик, отправляя клиенту ответ на каждый запрос в виде полученного текста от клиента.

Для запуска сервера добавляем в main.cpp следующий код:



TcpServer server;

// запускаем сервер на порту 1234

if (!server.listen(QHostAddress::Any, 1234))

{

qDebug() << "не удалось запустить сервер:" << server.errorString();

return -1;

}

qDebug() << "сервер запущен на порту" << server.serverPort();

Реализация клиента TCP реализуется похожим образом. Сначала также добавляем:

find\_package(Qt6 REQUIRED COMPONENTS Network)

target\_link\_libraries(TcpServer PRIVATE Qt6::Network)

В файл CMakeLists.txt, где TcpServer нужно заменить на название вашего проекта, а в случае использования qmake необходимо добавить QT += network в файл \*.pro.

Далее создадим класс TcpClient:





#ifndef TCPCLIENT\_H

#define TCPCLIENT\_H

#include <QTcpSocket>

#include <QString>

#include <QDebug>

class TcpClient : public QObject

{

Q\_OBJECT

public:

TcpClient(const QString &host, quint16 port, QObject \*parent = nullptr)

: QObject(parent), socket(new QTcpSocket(this)), host(host), port(port)

{

connect(socket, &QTcpSocket::connected, this, &TcpClient::onConnected);

connect(socket, &QTcpSocket::readyRead, this, &TcpClient::onReadyRead);

connect(socket, &QTcpSocket::disconnected, this, &TcpClient::onDisconnected);

connect(socket, QOverload<QAbstractSocket::SocketError>::of(&QAbstractSocket::errorOccurred),

this, &TcpClient::onError);

}

void connectToServer()

{

qDebug() << "Connecting to server...";

socket->connectToHost(host, port);

}

void sendMessage(const QString &message)

{

if (socket->state() == QAbstractSocket::ConnectedState) {

socket->write(message.toUtf8());

} else {

qDebug() << "Not connected to server.";

}

}

private slots:

void onConnected()

{

qDebug() << "Connected to server.";

}

void onReadyRead()

{

QByteArray data = socket->readAll();

qDebug() << "Received from server:" << data;

}

void onDisconnected()

{

qDebug() << "Disconnected from server.";

}

void onError(QAbstractSocket::SocketError socketError)

{

qDebug() << "Socket error:" << socket->errorString();

}

private:

QTcpSocket \*socket;

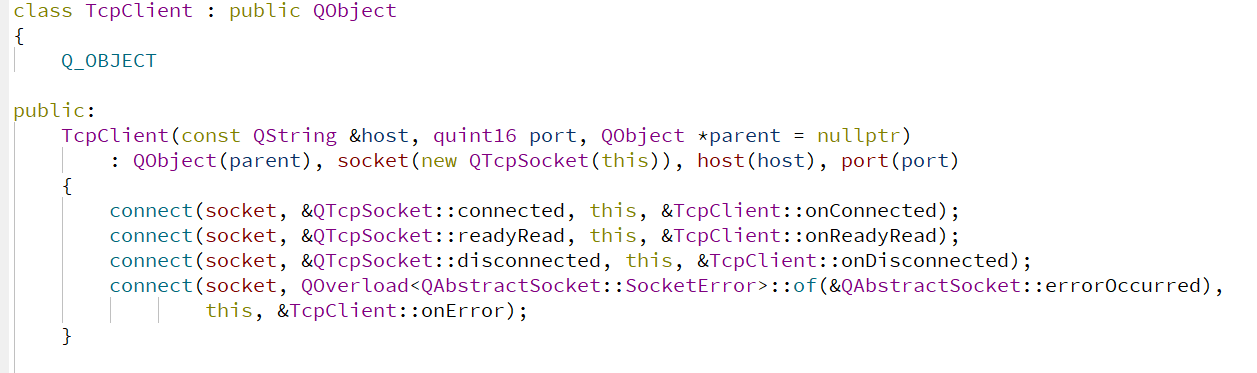
QString host;

quint16 port;

};

#endif // TCPCLIENT\_H

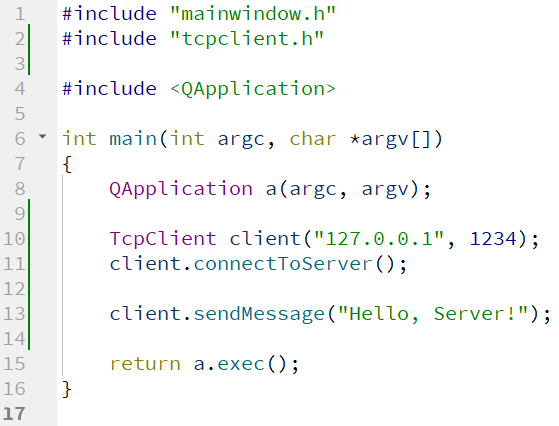
В данном коде мы соединяем события успешного подключения к серверу (connected), получения сообщения от сервера (readyRead), отключения от сервера (disconnected) и обработку ошибок (onError):



В данном примере все данные события просто выводятся в консоль:



Для запуска клиента необходимо добавить в main.cpp следующий код:



TcpClient client("127.0.0.1", 1234);

client.connectToServer();

client.sendMessage("Hello, Server!");

После запуска клиент подключится к серверу и отправить сообщение “Hello, server!”.

#### Реализация на языке C#

Описание реализации протоколов UDP и TCP на языке C# и реализация, соответствующих клиента и сервера подробно расписаны в дополнительном материале: <https://gitverse.ru/stasnorman/csharp-tutorian/content/master/sytem-prog/3.md>

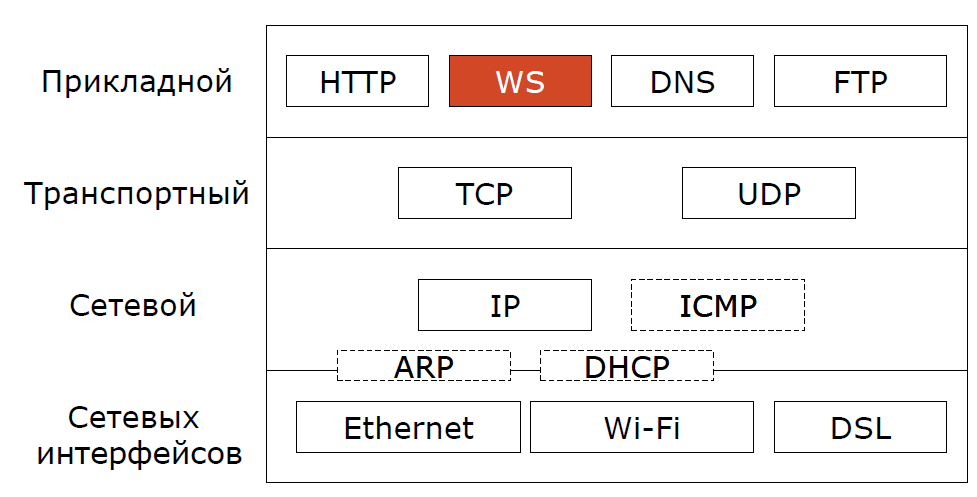
Пример применения UDP и TCP протоколов в языке C# можно рассмотреть на примере данного проекта:

* Сервер: <https://gitverse.ru/stasnorman/server>
* Клиент: <https://gitverse.ru/stasnorman/keyMasterClient>

Данную реализацию можно взять за основу своего решения.

## Протокол WebSocket

Протокол WebSocket – это современный протокол для двусторонней коммуникации между клиентом и сервером, реализованный поверх TCP-соединения, и предназначенный для установления постоянного соединения между клиентом и сервером.



В отличие от HTTP, который работает по принципу «запрос-ответ», WebSocket предоставляет двунаправленную коммуникацию, позволяя передавать данные от клиента к серверу и обратно без необходимости повторного установления соединения.

Это делает его особенно полезным для приложений, требующих низкой задержки и высокой производительности, например, чатов, онлайн-игр, финансовых приложений и потоковой передачи данных в реальном времени.

Основные особенности протокола WebSocket:

* Двусторонняя связь: Клиент и сервер могут обмениваться данными в обоих направлениях в рамках одного соединения.
* Постоянное соединение: WebSocket соединение остается открытым до тех пор, пока его не закроет одна из сторон.
* Снижение накладных расходов: WebSocket не требует необходимости пересылать заголовки HTTP при каждом сообщении, что экономит ресурсы.
* Поддержка стандартных портов: WebSocket использует стандартные порты TCP (80 для ws и 443 для wss), что делает его совместимым с большинством сетей и прокси-серверов.
* Поддержка текстовых и бинарных данных: WebSocket поддерживает как текстовые сообщения, так и бинарные данные (например, изображения или аудио).
* Событийно-ориентированная архитектура: Работа с WebSocket строится на событиях, таких как подключение, полученные данные, разрыв соединения, ошибки и т.д.

Технические особенности:

* Надежность: Как и TCP, WebSocket гарантирует доставку данных в правильном порядке.
* Совместимость с HTTP: Использует стандартные порты 80 (HTTP) и 443 (HTTPS), что упрощает развертывание в существующей инфраструктуре.
* Асинхронность: Сервер может инициировать отправку данных клиенту без ожидания запроса.

Области применения WebSocket:

* Онлайн-чаты и мессенджеры: Мгновенная доставка сообщений между пользователями.
* Многопользовательские онлайн-игры: Синхронизация действий игроков в реальном времени.
* Финансовые платформы: Передача биржевых котировок и обновлений.
* Push-уведомления: Рассылка уведомлений от сервера к клиенту (например, в веб-приложениях).

**Примеры реализация протокола WebSocket**

Рассмотрим примеры на разных языках программирования:

Для работы с WebSocket в C# .NET можно использовать пространство имен System.Net.WebSockets, без установки дополнительных библиотек.

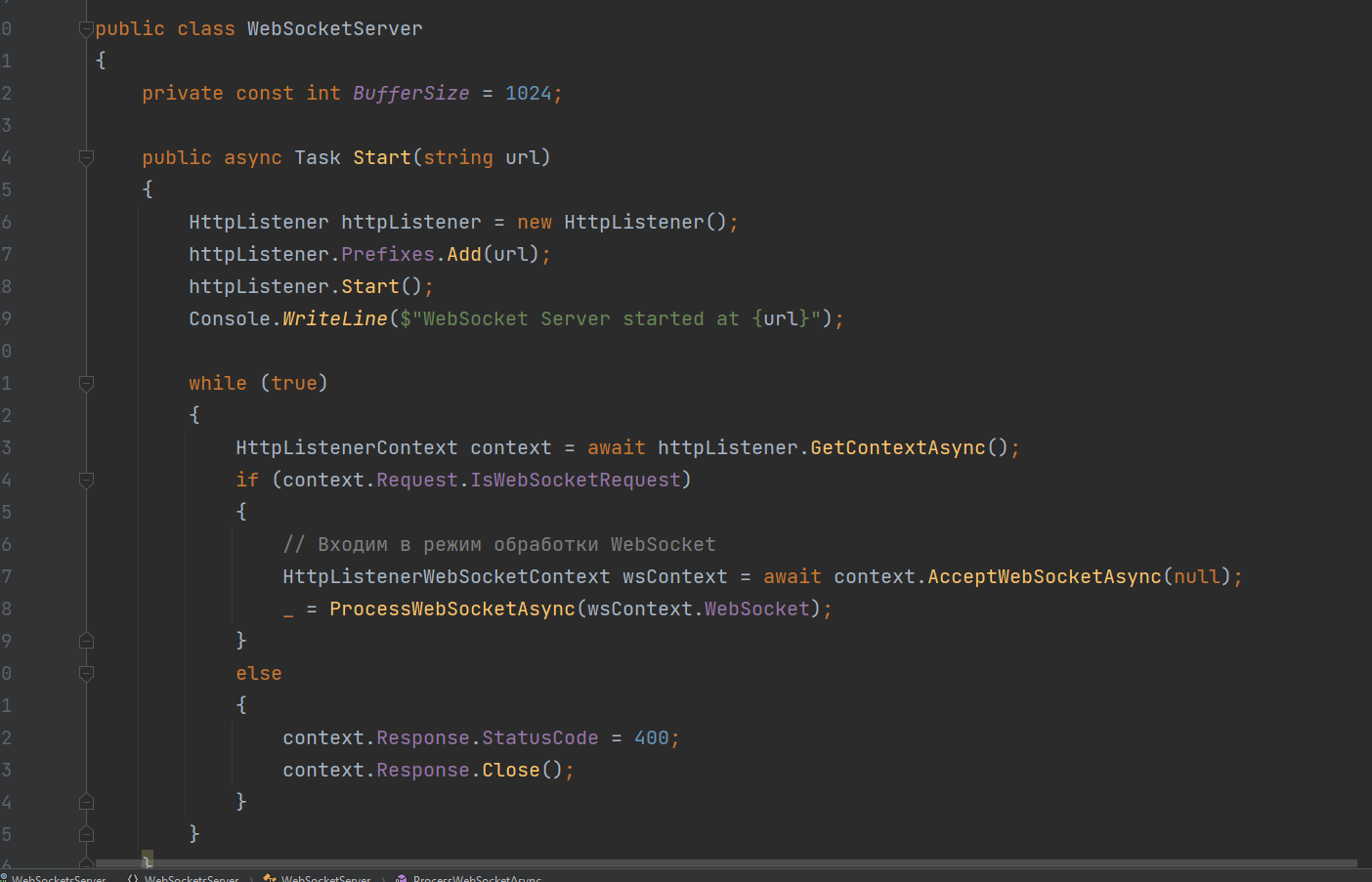
Начнём с создания WebSocket сервера. Создадим сервер, который будет ожидать подключения клиентов, принимать сообщения от них и отправлять ответы. В данном примере мы создадим WebSocket сервер, который отвечает клиенту его-же сообщением. В данном примере будет использоваться асинхронный подход.

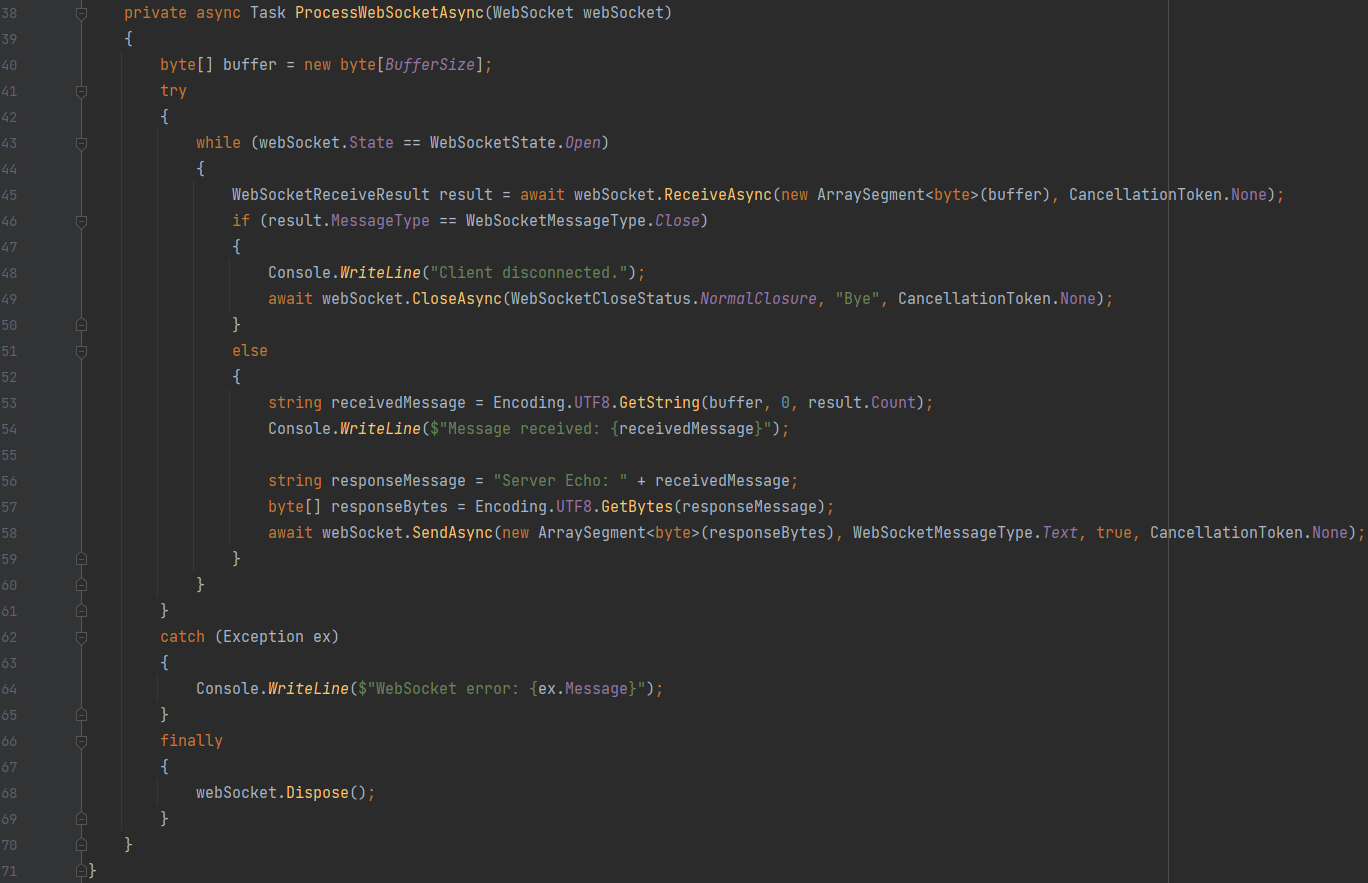
Для начала создадим класс WebSocketServer с двумя методами:

* public async Task Start(string url)
* private async Task ProcessWebSocketAsync(WebSocket webSocket)

Первый метод отвечает за запуск сервера, а второй за обработку входящих событий.

Код данных методов, следующий:





namespace WebSocketsServer;

using System;

using System.Net;

using System.Net.WebSockets;

using System.Text;

using System.Threading;

using System.Threading.Tasks;

public class WebSocketServer

{

private const int BufferSize = 1024;

public async Task Start(string url)

{

HttpListener httpListener = new HttpListener();

httpListener.Prefixes.Add(url);

httpListener.Start();

Console.WriteLine($"WebSocket Server started at {url}");

while (true)

{

HttpListenerContext context = await httpListener.GetContextAsync();

if (context.Request.IsWebSocketRequest)

{

// Входим в режим обработки WebSocket

HttpListenerWebSocketContext wsContext = await context.AcceptWebSocketAsync(null);

\_ = ProcessWebSocketAsync(wsContext.WebSocket);

}

else

{

context.Response.StatusCode = 400;

context.Response.Close();

}

}

}

private async Task ProcessWebSocketAsync(WebSocket webSocket)

{

byte[] buffer = new byte[BufferSize];

try

{

while (webSocket.State == WebSocketState.Open)

{

WebSocketReceiveResult result = await webSocket.ReceiveAsync(new ArraySegment<byte>(buffer), CancellationToken.None);

if (result.MessageType == WebSocketMessageType.Close)

{

Console.WriteLine("Client disconnected.");

await webSocket.CloseAsync(WebSocketCloseStatus.NormalClosure, "Bye", CancellationToken.None);

}

else

{

string receivedMessage = Encoding.UTF8.GetString(buffer, 0, result.Count);

Console.WriteLine($"Message received: {receivedMessage}");

string responseMessage = "Server Echo: " + receivedMessage;

byte[] responseBytes = Encoding.UTF8.GetBytes(responseMessage);

await webSocket.SendAsync(new ArraySegment<byte>(responseBytes), WebSocketMessageType.Text, true, CancellationToken.None);

}

}

}

catch (Exception ex)

{

Console.WriteLine($"WebSocket error: {ex.Message}");

}

finally

{

webSocket.Dispose();

}

}

}

В данном примере сервер использует HttpListener для ожидающих WebSocket-соединений, и, если клиент отправляет запрос WebSocket, сервер получает сообщение от клиента, отправляет ему ответное сообщение в формате «эхо». При получении команды на завершение от клиента, сервер закрывает соединение.

Запуск сервера происходит путём создания объекта WebSocketServer и указанием адреса и порта:

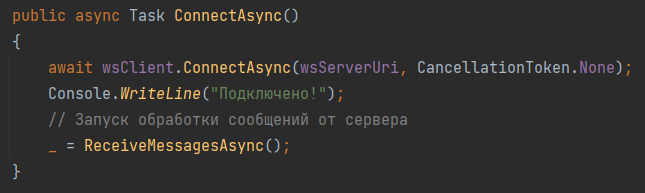


Теперь создадим клиента. Создадим класс WebSocketClient с полями:

private readonly ClientWebSocket wsClient;

private readonly Uri wsServerUri;

Добавим метод для подключения к серверу:

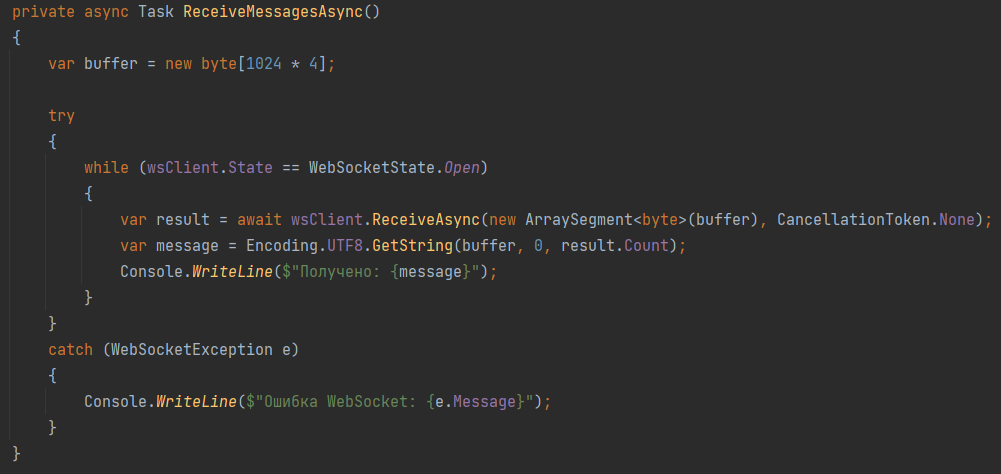


Тут wsServerUri – адрес WebSocket сервера.

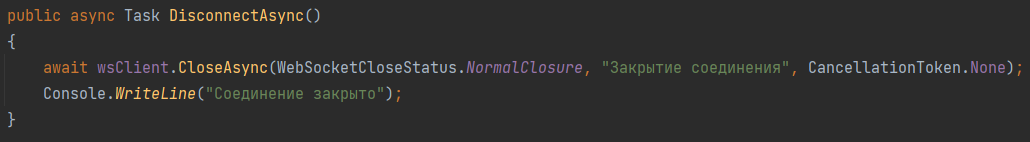
Добавим метод для отправки сообщения:



Добавим метод обработки входящих сообщений:



И метод закрытия соединения:



Полный код:

namespace WebSocketsClient;

using System;

using System.Net.WebSockets;

using System.Text;

using System.Threading;

using System.Threading.Tasks;

class WebSocketClient

{

private readonly ClientWebSocket wsClient;

private readonly Uri wsServerUri;

public WebSocketClient(string uri)

{

wsClient = new ClientWebSocket();

wsServerUri = new Uri(uri);

}

public async Task ConnectAsync()

{

await wsClient.ConnectAsync(wsServerUri, CancellationToken.None);

Console.WriteLine("Подключено!");

// Запуск обработки сообщений от сервера

\_ = ReceiveMessagesAsync();

}

public async Task SendMessageAsync(string? message)

{

var messageBytes = Encoding.UTF8.GetBytes(message);

var buffer = new ArraySegment<byte>(messageBytes);

await wsClient.SendAsync(buffer, WebSocketMessageType.Text, true, CancellationToken.None);

Console.WriteLine($"Отправлено: {message}");

}

private async Task ReceiveMessagesAsync()

{

var buffer = new byte[1024 \* 4];

try

{

while (wsClient.State == WebSocketState.Open)

{

var result = await wsClient.ReceiveAsync(new ArraySegment<byte>(buffer), CancellationToken.None);

var message = Encoding.UTF8.GetString(buffer, 0, result.Count);

Console.WriteLine($"Получено: {message}");

}

}

catch (WebSocketException e)

{

Console.WriteLine($"Ошибка WebSocket: {e.Message}");

}

}

public async Task DisconnectAsync()

{

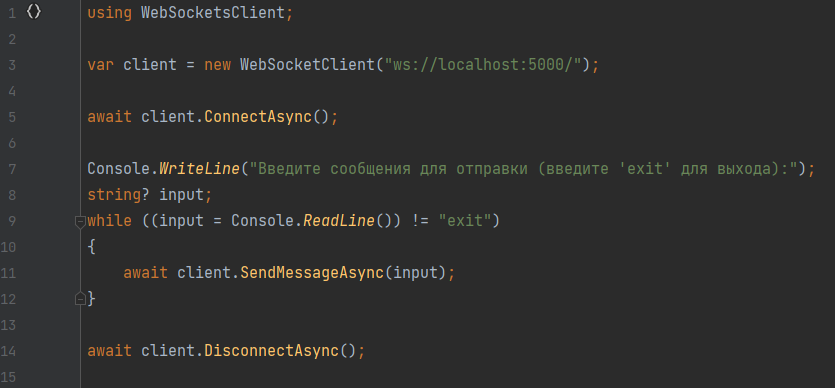
await wsClient.CloseAsync(WebSocketCloseStatus.NormalClosure, "Закрытие соединения", CancellationToken.None);

Console.WriteLine("Соединение закрыто");

}

}

Запустить клиент можно, добавив в Program.cs следующий код:



using WebSocketsClient;

var client = new WebSocketClient("ws://localhost:5000/");

await client.ConnectAsync();

Console.WriteLine("Введите сообщения для отправки (введите 'exit' для выхода):");

string? input;

while ((input = Console.ReadLine()) != "exit")

{

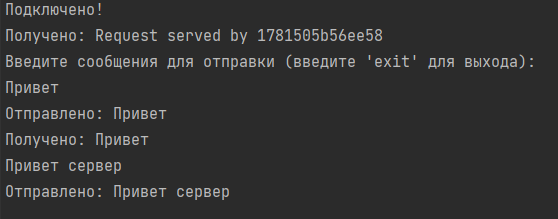
await client.SendMessageAsync(input);

}

await client.DisconnectAsync();

В данном примере мы подключаемся к WebSocket серверу, запущенному на локальном компьютере, отправляя ему сообщения, вводимые пользователем в консоль.

В итоге работа клиента и сервера будет выглядеть таким образом:



# Задания для самостоятельной работы

На основе изученного теоретического материала и материалов лекций необходимо выполнить один из вариантов творческого задания.

**Требования и темы вариантов для реализации проекта:**

1. **При реализации клиента и сервера можно использовать свободный стек технологий;**
2. **В задании необходимо применить все 3 протокола: UDP, TCP и WebSocket.**
3. **Клиенты подключаются к серверу через TCP/IP, используя уникальный идентификатор сессии. На клиенте должна быть возможность добавлять и удалять несколько серверов (IP адрес и порт вводятся пользователем) и иметься возможность подключения к любому из доступных.**
4. **Сервер с помощью WebSocket, сервер генерирует данные, которые меняются каждые 1-5 минут, в зависимости от темы задания.**
5. **Данные о перемещениях в реальном времени передаются по протоколу UDP.**
6. **Клиент должен хранить историю последних событий в локальном кэше.**
7. **Должна быть реализована событийная система – сервер уведомляет всех игроков о «событиях дня», например, выпадении уникального артефакта.**
8. **Нельзя игнорировать ни один из ранее указанных пунктов.**

## Темы заданий

1. **Космическая биржа.** Торговая платформа для покупки/продажи виртуальных ресурсов между космическими станциями. TCP - регистрация кораблей, WebSocket - аукционные события, UDP - потоковая передача цен на ресурсы.
2. **ЭкоМонитор.** Система контроля загрязнения воздуха. TCP - подключение датчиков, WebSocket - экстренные оповещения, UDP - поток показаний сенсоров.
3. **Шахматы.** Мультиплеер с пользовательским чатом. TCP - ходы, WebSocket - уведомления о ходе, UDP - позиции аватаров на шахматной доске.
4. **Киберспортивный букмекер.** Ставки на матчи в реальном времени. TCP - управление балансом, WebSocket - live-обновления коэффициентов, UDP - секундомер матча.
5. **Цифровой аукцион искусства.** TCP - регистрация участников, WebSocket - уведомления о ставках, UDP - анимация предпросмотра работ.
6. **Умная ферма.** Управление IoT-устройствами теплицы. TCP - команды полива, WebSocket - прогноз погоды, UDP - поток данных с датчиков влажности.
7. **Киберпанк-детектив.** Поиск улик в open-world. TCP - диалоги с NPC, WebSocket - появление новых квестов, UDP - позиции других детективов.
8. **AlgoTrader.** Платформа для алгоритмической торговли. TCP - исполнение заказов, WebSocket - рыночные новости, UDP - тиковые данные.
9. **Голосовой ассистент для слепых.** TCP - настройка профиля, WebSocket - обновления ПО, UDP - поток локационных подсказок через bone conduction.
10. **Интерактивный музей.** Экскурсии с дополненной реальностью. TCP - загрузка экспонатов, WebSocket - квестовые события, UDP - позиция посетителей в зале.
11. **Кибер-госпиталь.** Симулятор экстренной помощи. TCP - история пациентов, WebSocket - массовые ЧП, UDP - витальные показатели в реальном времени.
12. **Крипто-квест.** Поиск спрятанных кошельков с NFT. TCP - авторизация, WebSocket - подсказки сообщества, UDP - геолокационные метки.
13. **Дрон-рейсинг.** Соревнования БПЛА. TCP - регистрация трасс, WebSocket - изменение погодных условий, UDP - телеметрия дронов.
14. **Нейрочат.** Общение через нейроинтерфейсы. TCP - авторизация, WebSocket - модерация контента, UDP - поток эмоциональных паттернов.
15. **Цифровой двойник завода.** IoT-мониторинг оборудования. TCP - аутентификация инженеров, WebSocket - аварийные события, UDP - вибрация станков.
16. **Кибер-Паркур.** ММО с физикой движения. TCP - инвентарь, WebSocket - турнирные события, UDP - позиции игроков в 3D-пространстве.
17. **Биржа талантов.** Трекер для рекрутеров. TCP - резюме, WebSocket - push-уведомления о кандидатах, UDP - live-статистика рынка труда.
18. **Автомобильный навигатор.** Маршруты с препятствиями. TCP - карты, WebSocket - обновления инфраструктуры, UDP - позиция пользователя + ориентация.
19. **Квантовый симулятор.** Облачные вычисления. TCP - задачи, WebSocket - готовность результатов, UDP - промежуточные состояния кубитов.
20. **Цифровой тренер.** Фитнес-платформа. TCP - программы тренировок, WebSocket - челленджи, UDP - показатели пульса/скорости.
21. **Кибер-Почтальон.** Доставка писем в метавселенной. TCP - инвентарь посылок, WebSocket - новые заказы, UDP - координаты дронов-курьеров.
22. **НейроСпорт.** Соревнования по концентрации. TCP - профиль игрока, WebSocket - старт/финиш, UDP - EEG-сигналы в реальном времени.
23. **Цифровой зоопарк.** Уход за виртуальными питомцами. TCP - покупка корма, WebSocket - эпидемии, UDP - активность животных.
24. **Космический логист.** Оптимизация грузоперевозок. TCP - маршруты, WebSocket - метеоритные дожди, UDP - позиции кораблей.
25. **Совместный 3Д конструктор.** Совместное 3D-моделирование. TCP - сохранение проектов, WebSocket - комментарии, UDP – положение объектов участников.
26. **Кибер-Судья.** Платформа для онлайн-дебатов. TCP - регистрация, WebSocket - тайминг раундов, UDP - анализ тона речи.
27. **Цифровой садовод.** Выращивание редких растений. TCP - покупка семян, WebSocket - сезонные события, UDP - параметры почвы.
28. **Кибер-Сыщик.** Расследования киберпреступлений. TCP - отчеты, WebSocket - новые улики, UDP - отслеживание IP-адресов.
29. **Кибер-Спектр.** Поиск призраков в AR. TCP - журнал находок, WebSocket - аномалии, UDP - данные EMF-датчиков.
30. **Цифровой алхимик.** Создание виртуальных элементов. TCP - рецепты, WebSocket - редкие катастрофы, UDP - реакции частиц.
31. **Кибер-Шеф.** Управление рестораном. TCP - меню, WebSocket – рейтинги, события клиентов, UDP - статус заказов.
32. **AR Палеонтолог.** Раскопки динозавров. TCP - каталог находок, WebSocket - новые локации, UDP - позиция инструментов.
33. **Кибер-Геолог.** Поиск месторождений. TCP - карты, WebSocket - землетрясения, UDP - данные сейсмодатчиков.
34. **Нейро-Танцы.** Танцевальный симулятор. TCP - треки, WebSocket - баттлы, UDP - мышечные сигналы.
35. **Цифровой архитектор.** Строительство городов. TCP - чертежи, WebSocket - стихийные бедствия, UDP - трафик транспорта.
36. **Кибер-Агроном.** Управление умной фермой. TCP - планы посевов, WebSocket – изменение погоды, UDP - данные дронов, прогнозы.
37. **Кибер-Врач.** Диагностика заболеваний. TCP - истории болезней, WebSocket – процессы лечения, течения болезни, UDP - данные с медоборудования.
38. **Цифровой океанолог.** Исследование глубин. TCP - карты, WebSocket – штормы, UDP - показатели батискафа.
39. **Кибер-Астроном.** Наблюдение за космосом. TCP - каталоги, WebSocket - новые объекты, UDP - данные телескопов.
40. **Цифровой метеоролог.** Прогнозирование катастроф. TCP - погода, WebSocket – тревоги, образование погодных явлений, UDP - данные радаров.
41. **Игра Археолог.** Поиск древних артефактов. TCP - артефакты, WebSocket - квесты, UDP - геолокация.
42. **Кибер-Пират.** Морские сражения. TCP - корабли, WebSocket – штормы, боевая система, UDP - координаты судов.
43. **Децентрализованная энергобиржа.** TCP — смарт-контракты, WebSocket — аукционы энергии, UDP — поток с IoT-счётчиков.
44. **Глобальный логистический хаб.** TCP — мультимодальные цепочки, WebSocket — геополитические риски, UDP — телеметрия контейнеров.
45. **Квантовая биржа активов**. TCP — HFT-стратегии, WebSocket — аномалии рынка, UDP — синхронизация времени.
46. **CBDC-расчётная система.** TCP — межбанковские транзакции, WebSocket — санкционные списки, UDP — аналитика ликвидности.
47. **Умный мегаполис.** TCP — управление инфраструктурой, WebSocket — прогнозы ЧС, UDP — данные сенсоров города.
48. **Кибербезопасность АЭС.** TCP — управление реакторами, WebSocket — аварийные сценарии, UDP — радиационные данные.
49. **Межпланетная сеть связи.** TCP — передача научных данных, WebSocket — маршрутизация спутников, UDP — телеметрия с задержками.
50. **Макроэкономический симулятор.** TCP — экономические модели, WebSocket — кризисные события, UDP — биржевые данные.
51. **Промышленный IoT-щит.** TCP — патч-менеджмент, WebSocket — zero-day угрозы, UDP — сетевые аномалии.
52. **Роевые БПЛА.** TCP — координация агентов, WebSocket — перераспределение задач, UDP — обмен данными роя.
53. **Геномная аналитика.** TCP — секвенирование ДНК, WebSocket — мутационные алерты, UDP — поток геномных данных.
54. **Квантовый ЦОД.** TCP — оркестрация кубитов, WebSocket — ошибки декогеренции, UDP — квантовая телеметрия.
55. **Киберполигон для ЧС.** TCP — сценарии атак, WebSocket — инциденты в реальном времени, UDP — трафик DDoS-атак.
56. **Галактический блокчейн.** TCP — межзвёздные транзакции, WebSocket — консенсусные события, UDP — метрики задержек.
57. **Федеративный ИИ-хаб.** TCP — распределённое обучение, WebSocket — обновления моделей, UDP — edge-данные.
58. **Постквантовая криптобиржа.** TCP — транзакции NIST-алгоритмов, WebSocket — угрозы взлома, UDP — хеш-рейт.
59. **Конфиденциальные вычисления.** TCP — SGX-сессии, WebSocket — утечки данных, UDP — телеметрия TEE.
60. **Автономный флот такси.** TCP — маршрутизация, WebSocket — дорожные инциденты, UDP — LiDAR-поток.
61. **Смарт-агрокомплекс.** TCP — управление роботами, WebSocket — прогнозы урожайности, UDP — данные дронов.
62. **Умная энергосеть.** TCP: балансировка, WS: аварии, UDP: smart meter data.
63. **Киберзащита умных городов.** TCP — управление системами безопасности, WebSocket — атаки на инфраструктуру, UDP — поток данных с камер наблюдения.
64. **Цифровая клиника будущего.** TCP — истории болезней, WebSocket — экстренные диагнозы ИИ, UDP — данные с биосенсоров пациентов.
65. **Платформа квантового майнинга.** TCP — распределение задач, WebSocket — обнаружение коллизий, UDP — хеш-потоки.
66. **Глобальная система мониторинга океана.** TCP — анализ загрязнений, WebSocket — утечки нефти, UDP — данные с подводных дронов.
67. **Биржа космических ресурсов.** TCP — контракты на астероиды, WebSocket — аукционы редких металлов, UDP — координаты спутников.
68. **Сеть доверенных транзакций.** TCP — верификация участников, WebSocket — фрод-алерты, UDP — поток подписей.
69. **Автономная сельхозтехника.** TCP — маршруты полей, WebSocket — прогнозы урожая, UDP — телеметрия тракторов.
70. **Децентрализованный суд.** TCP — подача исков, WebSocket — голосование присяжных, UDP — анонимные свидетельства.
71. **Система контроля климата.** TCP — настройки климатических зон, WebSocket — экстренные изменения, UDP — данные с датчиков CO₂.
72. **Роботизированная доставка органов.** TCP — логистика трансплантатов, WebSocket — изменения приоритетов, UDP — GPS медицинских дронов.
73. **Цифровой двойник планеты.** TCP — загрузка климатических моделей, WebSocket — катаклизмы, UDP — данные спутников.
74. **Платформа для синтетической биологии.** TCP — проектирование ДНК, WebSocket — мутационные риски, UDP — поток биосенсоров.
75. **Киберзащита геномов.** TCP — шифрование данных ДНК, WebSocket — утечки персональной генетики, UDP — аномалии секвенирования.
76. **Платформа для ситиферм.** TCP — управление роботами-агрономами, WebSocket — прогнозы роста, UDP — микроклимат теплиц.
77. **Система мониторинга вулканов.** TCP — модели извержений, WebSocket — экстренные эвакуации, UDP — данные с сейсмодатчиков.
78. **Глобальная сеть оповещений о ЧС.** TCP — оповещения от МЧС, WebSocket — стихийные бедствия, UDP — данные с дронов.
79. **Автономные грузовые поезда.** TCP — расписания, WebSocket — аварии на путях, UDP — данные с локомотивов.
80. **Система управления квантовыми сенсорами.** TCP — калибровка, WebSocket — аномалии, UDP — поток измерений.
81. **Автономные умные порты.** TCP — логистика грузов, WebSocket — кибератаки на инфраструктуру, UDP — координаты кранов.
82. **Система контроля космического мусора.** TCP — расчет орбит, WebSocket — угрозы столкновений, UDP — данные телескопов.
83. **Роевые подводные дроны.** TCP — исследование океана, WebSocket — обнаружение аномалий, UDP — гидролокационные данные.
84. **Платформа для симуляции климата.** TCP — климатические модели, WebSocket — экстремальные сценарии, UDP — спутниковые данные.
85. **Система предиктивного ремонта инфраструктуры.** TCP — графики ТО, WebSocket — аварии, UDP — вибрации мостов/дорог.
86. **Глобальная сеть умных дорог.** TCP — управление светофорами, WebSocket — ДТП, UDP — данные с датчиков покрытия.
87. **Система контроля пищевых цепочек.** TCP — отслеживание поставок, WebSocket — загрязнения, UDP — температура грузов.
88. **Киберполигон для беспилотников.** TCP — сценарии тестов, WebSocket — аварийные ситуации, UDP — координаты дронов.
89. **Цифровой двойник экономики.** TCP — макроэкономические модели, WebSocket — кризисы, UDP — рыночные индексы.
90. **Система прогнозирования землетрясений.** TCP — геологические модели, WebSocket — предупреждения, UDP — данные сейсмографов.
91. **Платформа для управления космическими миссиями.** TCP — планирование полетов, WebSocket — аварии, UDP — телеметрия зондов.
92. **Платформа для симуляции пандемий.** TCP — загрузка эпидемиологических моделей, WebSocket — вспышки заболеваний, UDP — данные мобильности населения.
93. **Автономные системы опреснения воды.** TCP — управление станциями, WebSocket — аварии, UDP — данные о солености и расходе.
94. **Платформа для управления космическим туризмом.** TCP — бронирование полетов, WebSocket — изменения орбит, UDP — телеметрия кораблей.
95. **Глобальная сеть контроля радиоактивных отходов.** TCP — инвентаризация, WebSocket — утечки, UDP — данные с дозиметров.
96. **Платформа для симуляции финансовых кризисов.** TCP — макроэкономические параметры, WebSocket — банковские паники, UDP — биржевые данные.
97. **Автономные системы рекультивации земель.** TCP — карты загрязнений, WebSocket — прогресс очистки, UDP — данные с дронов-экологов.
98. **Квантовая сеть для межпланетной торговли.** TCP — межгалактические контракты, WebSocket — аукционы ресурсов, UDP — координаты грузовых кораблей.
99. **ИИ-куратор образования.** TCP – учебные планы, WebSocket – адаптация программ, UDP – успеваемость.
100. **Глобальная сеть квантовых сенсоров.** TCP – управление, WebSocket – угрозы, UDP – метрики пространства-времени.