

# ПРОГРАММИРОВАНИЕ В INTERNET

---

TCP- И UDP-СЕРВЕР

TCP  
(Transmission Control  
Protocol)

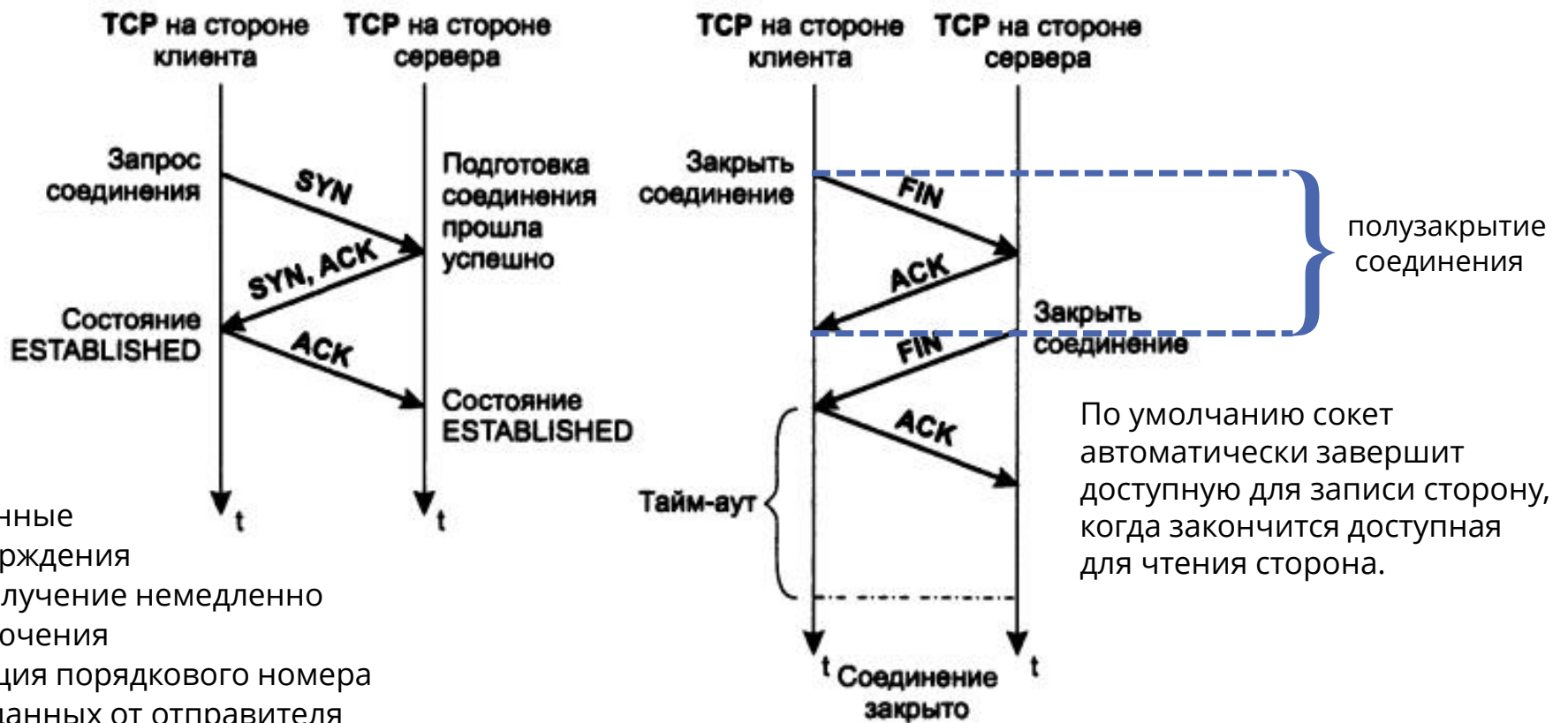
=

протокол **транспортного  
уровня** для передачи  
информации.

# Основные характеристики ТСР

- обмен осуществляется пакетами;
- надежный:
  - 1) установка соединения;
  - 2) подтверждение получения пакета;
  - 3) правильный порядок отправки;
  - 4) подсчет контрольных сумм для проверки целостности пакетов;
- низкая скорость передачи.

# Установка и закрытие (полузакрытие) соединения



URG — срочные данные

ACK — поле подтверждения

PSH — отправка/получение немедленно

RST — сброс подключения

SYN — синхронизация порядкового номера

FIN — больше нет данных от отправителя

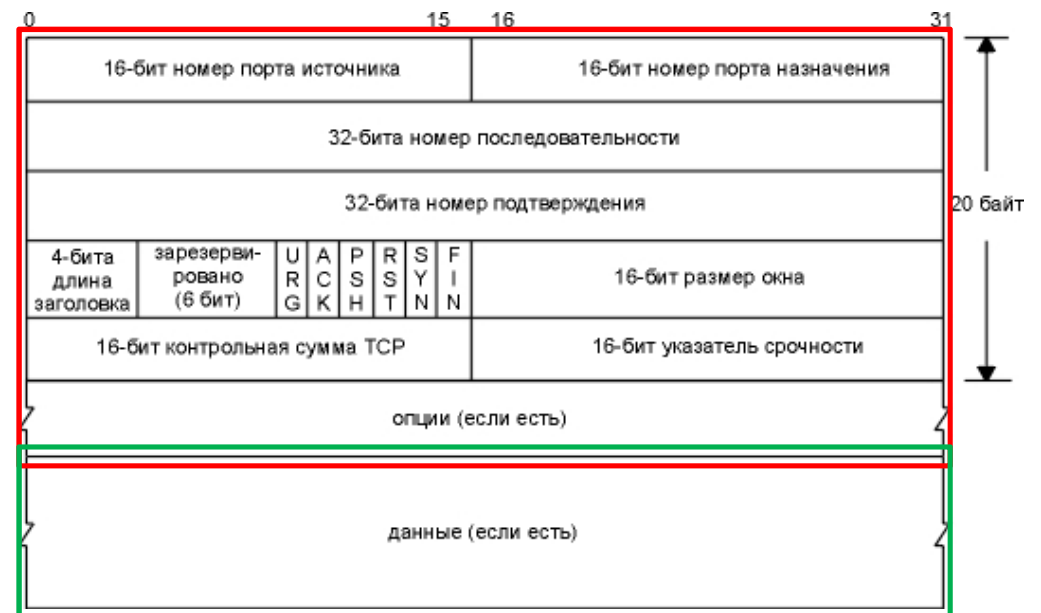
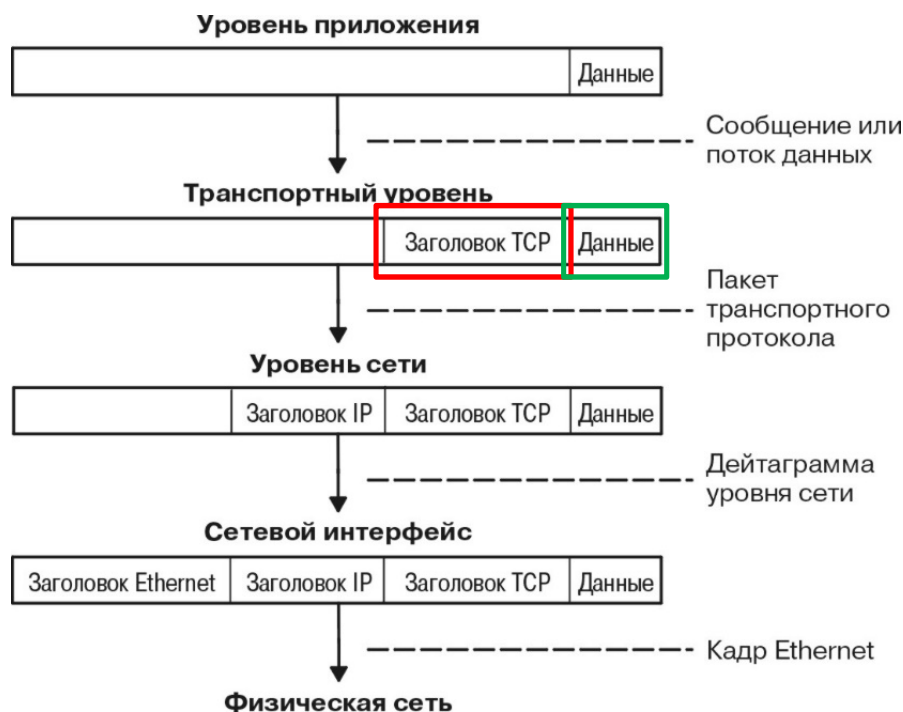
## Пример установки соединения (handshake)

TCP A		TCP B	
1. CLOSED		LISTEN	
2. SYN-SENT	→ <SEQ=100><CTL=SYN>	→ SYN-RECEIVED	
3. ESTABLISHED	← <SEQ=300><ACK=101><CTL=SYN,ACK>	← SYN-RECEIVED	
4. ESTABLISHED	→ <SEQ=101><ACK=301><CTL=ACK>	→ ESTABLISHED	
5. ESTABLISHED	← <SEQ=301><ACK=101><CTL=ACK>	← ESTABLISHED	

# Установка соединения и обмен данными

Time	192.168.1.2	174.143.213.18	Comment
0.000	(54841) →	SYN → (80)	Seq = 0 Ack = 94856056
0.047	(54841) ←	SYN, ACK → (80)	Seq = 0 Ack = 1
0.047	(54841) →	ACK → (80)	Seq = 1 Ack = 1
0.047	(54841) →	PSH, ACK - Len: 725 → (80)	Seq = 1 Ack = 1
0.097	(54841) ←	ACK → (80)	Seq = 1 Ack = <u>726</u>
0.100	(54841) →	ACK - Len: 1448 → (80)	Seq = 1 Ack = 726
0.100	(54841) →	ACK → (80)	Seq = 726 Ack = <u>1449</u>
0.100	(54841) →	ACK - Len: 1448 → (80)	Seq = 1449 Ack = 726
0.100	(54841) →	ACK → (80)	Seq = 726 Ack = <u>2897</u>
0.100	(54841) →	ACK - Len: 1448 → (80)	Seq = 2897 Ack = 726
0.100	(54841) →	ACK → (80)	Seq = 726 Ack = 4345
0.150	(54841) →	ACK - Len: 1448 → (80)	Seq = 4345 Ack = 726
0.150	(54841) →	ACK → (80)	Seq = 726 Ack = 5793
0.152	(54841) →	ACK - Len: 1448 → (80)	Seq = 5793 Ack = 726
0.152	(54841) →	ACK → (80)	Seq = 726 Ack = 7241
0.152	(54841) →	ACK - Len: 1448 → (80)	Seq = 7241 Ack = 726
0.152	(54841) →	ACK → (80)	Seq = 726 Ack = 8689

# Структура пакета



net.Server

=

класс, который используется  
для создания сервера TCP или  
IPC. Наследует EventEmitter.



# Способы создания экземпляров net.Server

- 1) С помощью метода `net.createServer([options][, connectionListener])`.
- 2) Путем явного создания экземпляра класса `new net.Server([options][, connectionListener])`.

Некоторые из опций, которые можно настроить:

- *allowHalfOpen* – если установлено значение `false`, то использование полужакрытых сокетов не будет разрешено. По умолчанию: `false`.
- *keepAlive* – если установлено значение `true`, активизируется функция поддержания активности на соquete сразу после получения нового входящего соединения. По умолчанию: `false`.

## Класс `net.Server` (события)

- **close** генерируется при закрытии сервера. Если соединения существуют, это событие не генерируется до тех пор, пока не будут завершены все соединения.
- **connection** генерируется при установке нового соединения. В обработчик передается `socket` (экземпляром `net.Socket`).
- **error** генерируется при возникновении ошибки. В отличие от `net.Socket`, событие `close` не будет сгенерировано непосредственно после этого события, если только `server.close()` не будет вызван вручную. С
- **listening** генерируется после вызова `server.listen()`.

# Класс net.Server (методы)

- `server.address()` возвращает адрес, имя семейства адресов и порт сервера.
- `server.close([callback])` запрещает серверу принимать новые соединения и сохраняет существующие соединения. Этот метод является асинхронным, сервер окончательно закрывается, когда все соединения завершены, и сервер генерирует событие `close`.
- `server.getConnections(callback)` позволяет асинхронно получить количество одновременных подключений к серверу. Обратный вызов принимает два аргумента `err` и `count`.
- `server.listen([port[, host[, backlog]]], [callback])` или `server.listen(options[, callback])` запускает сервер. Этот метод является асинхронным.
- `server.ref()/unref()` отмечает события, генерируемые сервером, второстепенной задачей.

net.Socket

=

класс, который является абстракцией  
TCP-сокета или потоковой конечной  
точки IPC. Наследует EventEmitter, а  
также stream.Duplex.

# Способы использования net.Socket

- 1) Можно создать net.Socket и использовать его непосредственно для взаимодействия с сервером:
  - `net.createConnection(options[, connectListener])` (или `net.connect(port[, host][, connectListener])`, или `socket.connect(port[, host][, connectListener])`) – фабричная функция, которая создает новый net.Socket, немедленно инициирует соединение с помощью `socket.connect()`, а затем возвращает net.Socket. Когда соединение установлено, в возвращаемом сокете будет сгенерировано событие `connect`. Последний параметр `connectListener`, если он указан, будет добавлен в качестве слушателя для события `connect` один раз.
  - `new net.Socket([options])` – создает новый объект сокета. Вновь созданный сокет может быть либо TCP-сокетом, либо потоковой конечной точкой IPC, в зависимости от того, к чему он подключается.
- 2) Он также может быть создан Node.js и передан при получении соединения. Например, он передается слушателям события `connection`, генерируемого net.Server, поэтому можно использовать его для взаимодействия с клиентом.

# Класс net.Socket (события)

- **close** генерируется после полного закрытия сокета.
- **connect** генерируется при успешном установлении соединения через сокет.
- **data** генерируется при получении данных. Данные аргумента будут буфером или строкой.
- **end** генерируется, когда другой конец сокета сигнализирует об окончании передачи, тем самым заканчивая доступную для чтения сторону сокета, т.е. при полужакрытии. По умолчанию (опция `allowHalfOpen = false`) сокет отправит обратно пакет конца передачи и уничтожит свой файловый дескриптор.

# Класс `net.Socket` (события)

- **error** генерируется при возникновении ошибки. Сразу после генерируется событие `close`.
- **ready** генерируется, когда сокет готов к использованию. Запускается сразу после `connect`.
- **timeout** генерируется, если время ожидания сокета истекло из-за бездействия. Только уведомляет, что сокет был бездействующим. Пользователь должен вручную закрыть соединение.

# Класс net.Socket (свойства)

- `socket.bytesRead` – количество полученных байтов;
- `socket.bytesWritten` – количество отправленных байтов;
- `socket.connecting` – если true, то `socket.connect()` был вызван и еще не завершен. Он будет оставаться истинным до тех пор, пока сокет не будет подключен, затем он будет установлен в значение false и будет сгенерировано событие connect;
- `socket.destroyed` – равняется true после вызова метода `destroy()`;
- `socket.pending` – true, если сокет еще не подключен, либо `connect()` еще не был вызван, либо он все еще находится в процессе подключения;
- `socket.readyState` – представляет состояние соединения в виде строки (opening, open, readOnly, writeOnly).



# Класс net.Socket (свойства)

- `socket.localAddress` – строковое представление локального IP-адреса, к которому подключается удаленный клиент;
- `socket.localPort` – числовое представление локального порта;
- `socket.localFamily` – строковое представление семейства локального IP-адреса;
- `socket.remoteAddress` – строковое представление удаленного IP-адреса;
- `socket.remotePort` – числовое представление удаленного порта;
- `socket.remoteFamily` – строковое представление семейства удаленного IP-адреса.
- `socket.timeout` – тайм-аут сокета в мс, установленный `socket.setTimeout()` (по умолчанию undefined).

# Класс net.Socket (методы)

- `socket.address()` возвращает связанный адрес, имя семейства адресов и порт сокета.
- `socket.connect(port[, host][, connectListener])` устанавливает соединение, а потом генерирует событие connect. Если возникает проблема с подключением, вместо события connect будет сгенерировано событие error. Последний параметр connectListener, если он указан, будет добавлен в качестве слушателя для события connect один раз. Этот метод является асинхронным.
- `socket.destroy([error])` гарантирует, что в этом сокете больше не будет операций ввода-вывода. Уничтожает поток и закрывает соединение.
- `socket.end([data[, encoding]][, callback])` полузакрывает сокет. т. е. отправляет пакет FIN. Возможно, что сервер будет продолжать отправлять данные (если при его создании параметр allowHalfOpen установлен в true).

# Класс net.Socket (методы)

- `socket.unref()` позволяет процессу завершиться, если это единственный активный сокет в системе событий. А метод `socket.ref()` не позволяет процессу завершиться, если это единственный оставшийся сокет.
- `socket.setEncoding([encoding])` устанавливает кодировку для сокета (для Readable Stream).
- `socket.setKeepAlive([enable][, initialDelay])` включает/отключает функции проверки активности и, при необходимости, устанавливает начальную задержку.
- `socket.setTimeout(timeout[, callback])` устанавливает для сокета тайм-аут (в мс) бездействия сокета. По умолчанию net.Socket не имеет тайм-аута (undefined).
- `socket.write(data[, encoding][, callback])` отправляет данные на сокет. Второй параметр указывает кодировку. Необязательный callback будет выполнен, когда данные будут окончательно записаны, что может произойти не сразу.

# Простейший TCP-сервер

```
const net = require('net');

let HOST = '0.0.0.0';
let PORT = 40000;

net.createServer((sock)=>{

  console.log('Server CONNECTED: ' + sock.remoteAddress + ':' + sock.remotePort);

  sock.on('data', (data)=>{
    console.log('Server DATA:', sock.remoteAddress + ': ' + data);
    sock.write('Сервер получил: ' + data );
  });

  sock.on('close', ( )=>{console.log('Ser CLOSED: ', sock.remoteAddress + ' ' + sock.remotePort);});

}).listen(PORT, HOST);

console.log('TCP-сервер ' + HOST + ':' + PORT);
```

# Простейший TSP-клиент

```
var net = require('net');

let HOST = '127.0.0.1';
let PORT = 40000;

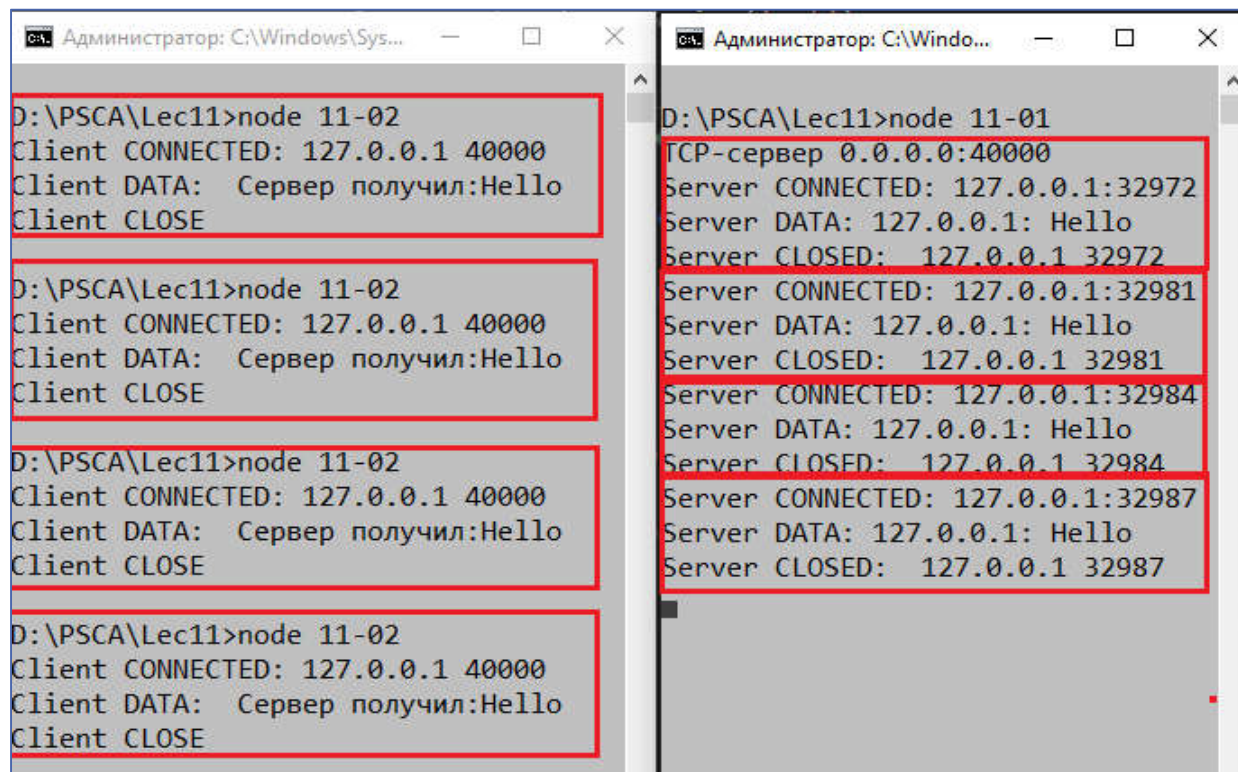
var client = new net.Socket();
client.connect(PORT, HOST, () => {
  console.log('Client CONNECTED: ', client.remoteAddress + ' ' + client.remotePort);
  client.write('Hello');
});

client.on('data', (data) => {
  console.log('Client DATA:' + data.toString());
  client.destroy();
});

client.on('close', () => { console.log('Client CLOSE'); });

client.on('error', (e) => { console.log('Client ERROR'); });
```

# Результат работы



```
Администратор: C:\Windows\Sys...
D:\PSCA\Lec11>node 11-02
Client CONNECTED: 127.0.0.1 40000
Client DATA: Сервер получил:Hello
Client CLOSE

D:\PSCA\Lec11>node 11-02
Client CONNECTED: 127.0.0.1 40000
Client DATA: Сервер получил:Hello
Client CLOSE

D:\PSCA\Lec11>node 11-02
Client CONNECTED: 127.0.0.1 40000
Client DATA: Сервер получил:Hello
Client CLOSE

D:\PSCA\Lec11>node 11-02
Client CONNECTED: 127.0.0.1 40000
Client DATA: Сервер получил:Hello
Client CLOSE

Администратор: C:\Windo...
D:\PSCA\Lec11>node 11-01
TCP-сервер 0.0.0.0:40000
Server CONNECTED: 127.0.0.1:32972
Server DATA: 127.0.0.1: Hello
Server CLOSED: 127.0.0.1 32972

Server CONNECTED: 127.0.0.1:32981
Server DATA: 127.0.0.1: Hello
Server CLOSED: 127.0.0.1 32981

Server CONNECTED: 127.0.0.1:32984
Server DATA: 127.0.0.1: Hello
Server CLOSED: 127.0.0.1 32984

Server CONNECTED: 127.0.0.1:32987
Server DATA: 127.0.0.1: Hello
Server CLOSED: 127.0.0.1 32987
```

# Отправка буфера (TCP-сервер)

```
const net = require('net');

let HOST = '0.0.0.0';
let PORT = 40000;

let sum = 0;

let server = net.createServer();
server.on('connection', (sock)=>{

    console.log('Server CONNECTED: ' + sock.remoteAddress + ':' + sock.remotePort);

    sock.on('data', (data)=>{
        console.log('Server DATA:', data, sum);
        sum += data.readInt32LE();
    });

    let buf = Buffer.alloc(4);
    setInterval(()=>{buf.writeInt32LE(sum,0); sock.write(buf)}, 3000);

    sock.on('close', (data)=>{console.log('Server CLOSED: ', sock.remoteAddress + ' ' + sock.remotePort)});

    sock.on('error', (e)=>{console.log('Server error: ', sock.remoteAddress + ' ' + sock.remotePort)});

})
server.on('listening', ()=>{ console.log('TCP-server ', HOST + ':' + PORT);} );
server.on('error', (e)=>{ console.log('TCP-server error ', e);} );
server.listen(PORT, HOST);
```

# Отправка буфера (TCP-клиент)

```
var net = require('net');

let HOST = '127.0.0.1';
let PORT = 40000;

var client = new net.Socket();
var buf = new Buffer.alloc(4);
let timerId = null;
client.connect(PORT, HOST, () => {
  console.log('Client CONNECTED: ', client.remoteAddress + ' ' + client.remotePort);

  let k = 0;
  timerId = setInterval(() => {
    buf.writeInt32LE(k++, 0);
    client.write(buf);
  }, 1000);

  setTimeout(() => {
    clearInterval(timerId);
    client.end();
  }, 30000);
})

client.on('data', (data) => { console.log('Client DATA: ' + data.readInt32LE()); });
client.on('close', () => { console.log('Client CLOSE'); });
client.on('error', (e) => { console.log('Client ERROR', e); });
```



# Результат работы

```
D:\PSCA\Lec11>node 11-03a
Client CONNECTED: 127.0.0.1 40000
Client DATA: 1
Client DATA: 10
Client DATA: 28
Client DATA: 55
Client DATA: 91
Client DATA: 136
Client DATA: 190
Client DATA: 253
Client DATA: 325
Client CLOSE
```

```
D:\PSCA\Lec11>
```

```
D:\PSCA\Lec11>■
```

```
D:\PSCA\Lec11>node 11-03
TCP-server 0.0.0.0:40000
Server CONNECTED: 127.0.0.1:48050
Server DATA: <Buffer 00 00 00 00> 0
Server DATA: <Buffer 01 00 00 00> 0
Server DATA: <Buffer 02 00 00 00> 1
Server DATA: <Buffer 03 00 00 00> 3
Server DATA: <Buffer 04 00 00 00> 6
Server DATA: <Buffer 05 00 00 00> 10
Server DATA: <Buffer 06 00 00 00> 15
Server DATA: <Buffer 07 00 00 00> 21
Server DATA: <Buffer 08 00 00 00> 28
Server DATA: <Buffer 09 00 00 00> 36
Server DATA: <Buffer 0a 00 00 00> 45
Server DATA: <Buffer 0b 00 00 00> 55
Server DATA: <Buffer 0c 00 00 00> 66
Server DATA: <Buffer 0d 00 00 00> 78
Server DATA: <Buffer 0e 00 00 00> 91
Server DATA: <Buffer 0f 00 00 00> 105
Server DATA: <Buffer 10 00 00 00> 120
Server DATA: <Buffer 11 00 00 00> 136
Server DATA: <Buffer 12 00 00 00> 153
Server DATA: <Buffer 13 00 00 00> 171
Server DATA: <Buffer 14 00 00 00> 190
Server DATA: <Buffer 15 00 00 00> 210
Server DATA: <Buffer 16 00 00 00> 231
Server DATA: <Buffer 17 00 00 00> 253
Server DATA: <Buffer 18 00 00 00> 276
Server DATA: <Buffer 19 00 00 00> 300
```

# ТСР-сервер (использование pipe)

```
const net = require('net');

let HOST = '0.0.0.0';
let PORT = 40000;

let sum = 0;

let ws = require('fs').createWriteStream('./Files/File04.data', {flags: 'a'});

let server = net.createServer();
server.on('connection', (sock)=>{

    console.log('Server CONNECTED: ' + sock.remoteAddress + ':' + sock.remotePort);

    sock.pipe(ws);

    sock.on('close', (data)=>{console.log('Server CLOSED: ', sock.remoteAddress + ' ' + sock.remotePort)});
    sock.on('error', (e)=>{console.log('Server error: ', sock.remoteAddress + ' ' + sock.remotePort)});
})
server.on('listening', ()=>{ console.log('TCP-server ', HOST + ':' + PORT);} );
server.on('error', (e)=>{ console.log('TCP-server error ', e);} );
server.listen(PORT, HOST);
```

net.Socket – это **дуплексный**  
**ПОТОК**

# ТСР-клиент (буфер)

```
var net = require('net');

let HOST = '127.0.0.1';
let PORT = 40000;

var client = new net.Socket();
var buf = new Buffer.alloc(4);
let timerId = null;
client.connect(PORT, HOST, () => {
  console.log('Client CONNECTED: ', client.remoteAddress + ' ' + client.remotePort);

  let k = 0;
  timerId = setInterval(() => { buf.writeInt32LE(k++, 0); client.write(buf); }, 1000);
  setTimeout(() => { clearInterval(timerId); client.end(); }, 30000);
});

client.on('data', (data) => { console.log('Client DATA:' + data.readInt32LE()); });

client.on('close', () => { console.log('Client CLOSE'); });

client.on('error', (e) => { console.log('Client ERROR', e); });
```

[illegible]

# ТСР-клиент (использование pipe)

```
const net = require('net');
```

```
let HOST = '127.0.0.1';
```

```
let PORT = 40000;
```

```
let rs = require('fs').createReadStream('./Files/File04.data');
```

```
let client = new net.Socket();
```

```
client.connect(PORT, HOST, ())=>{
```

```
console.log('Client CONNECTED:', client.remoteAddress + ' ' + client.remotePort);
```

```
rs.pipe(client);
```

```
});
```

```
client.on('data', (data)=>{console.log('Client DATA: ', data.readInt32LE())});
```

```
client.on('close', ()=>{console.log('Client CLOSE');});
```

```
client.on('error', (e)=>{console.log('Client ERROR', e);});
```

[illegible]

# Сервер, прослушивающий два порта

```
const net = require('net');

let HOST = '127.0.0.1';
let PORT = 40000;

let rs = require('fs').createReadStream('./Files/File04.data');

let client = new net.Socket();

client.connect(PORT, HOST, ()=>{
    console.log('Client CONNECTED:', client.remoteAddress + ' ' + client.remotePort);
    rs.pipe(client);
});

client.on('data', (data)=>{console.log('Client DATA: ', data.readInt32LE());});
client.on('close', ()=>{console.log('Client CLOSE');});
client.on('error', (e)=>{console.log('Client ERROR', e);});
```

# ТСР-клиент (номер порта берется из аргументов командной строки)

```
const net = require('net');

const HOST = '127.0.0.1';
const PORT = process.argv[2]? process.argv[2]:40000;

let client = new net.Socket();

client.connect(PORT, HOST, ()=>{

    console.log('Client CONNECTED:', client.remoteAddress + ' ' + client.remotePort);

    let k = 0;
    timerId = setInterval(()=>{client.write(`client: ${k++}`)}, 1000);
    setTimeout(()=>{clearInterval(timerId); client.end();}, 30000);

});

client.on('data', (data)=>{console.log('Client DATA: ', data)})
    .on('close', ()=>{console.log('Client CLOSE');})
    .on('error', (e)=>{console.log('Client ERROR', e);});
```



## Результат работы

```
D:\PSCA\Lec11>node 11-06a 50000
Client CONNECTED: 127.0.0.1 50000
Client DATA: <Buffer 45 43 48 4f 35 30 30 3
0 30 3a 63 6c 69 65 6e 74 3a 20 30>
Client DATA: <Buffer 45 43 48 4f 35 30 30 3
0 30 3a 63 6c 69 65 6e 74 3a 20 31>
Client DATA: <Buffer 45 43 48 4f 35 30 30 3
0 30 3a 63 6c 69 65 6e 74 3a 20 32>
Client DATA: <Buffer 45 43 48 4f 35 30 30 3
0 30 3a 63 6c 69 65 6e 74 3a 20 33>
Client DATA: <Buffer 45 43 48 4f 35 30 30 3
0 30 3a 63 6c 69 65 6e 74 3a 20 34>
Client DATA: <Buffer 45 43 48 4f 35 30 30 3
0 30 3a 63 6c 69 65 6e 74 3a 20 35>
^C
D:\PSCA\Lec11>
```

UDP  
(User Datagram Protocol) =

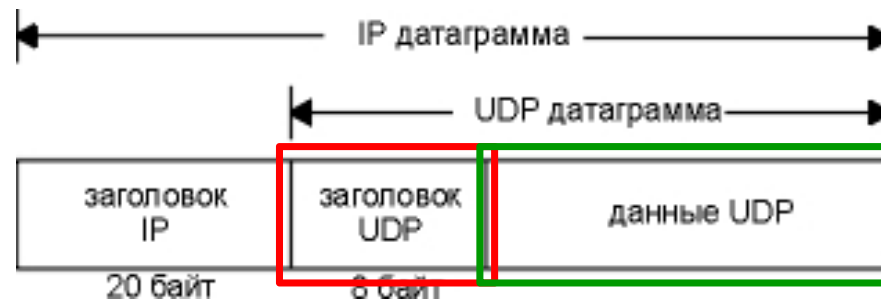
протокол транспортного  
уровня для передачи  
информации.



# Основные характеристики UDP

- обмен осуществляется дейтаграммами;
- **ненадежный**:
  - 1) не устанавливает соединение;
  - 2) упорядоченность не соблюдается;
  - 3) доставка не гарантируется;
- более **высокая** скорость передачи.

# Структура UDP-дейтаграммы



dgram.Socket

=

инкапсулирует функциональность  
дейтаграммы. Наследует EventEmitter.

Создать экземпляр `dgram.Socket` можно с помощью метода `dgram.createSocket(type[, callback])`.

! Ключевое слово `new` не должно использоваться для создания экземпляров `dgram.Socket`.

# Класс `dgram.Socket` (события)

- **close** генерируется после закрытия сокета с помощью функции `close()`. После срабатывания в этом сокете не будут создаваться новые события `message`.
- **connect** генерируется после того, как сокет связан с удаленным адресом в результате успешного вызова `connect()`.
- **error** генерируется всякий раз, когда возникает какая-либо ошибка. В функцию обработчика событий передается один объект `Error`.
- **listening** генерируется, когда `dgram.Socket` становится адресуемым и может получать данные. Это происходит либо явно с помощью `socket.bind()`, либо неявно при первой отправке данных с помощью `socket.send()`.
- **message** генерируется, когда в сокете доступна новая дейтаграмма. В функцию обработчика событий передаются два аргумента: `msg` (само сообщение) и `rinfo` (информация об удаленном адресе, отправителе сообщения).

# Класс `dgram.Socket` (методы)

- `socket.address()` возвращает объект, содержащий информацию об адресе сокета (`address`, `family` и `port`).
- `socket.bind([port][, address][, callback])` или `socket.bind(options[, callback])` прослушивает дейтаграммы на указанном порте и адресе. Если порт не указан, ОС попытается выполнить привязку к произвольному порту. Если адрес не указан, ОС попытается прослушивать все адреса. После завершения привязки генерируется событие `listening` и вызывается `callback`.
- `socket.close([callback])` закрывает сокет и перестает прослушивать данные на нем. Если предоставлен `callback`, то он добавляется в качестве слушателя для события `close`.
- `socket.connect(port[, address][, callback])` связывает сокет с удаленным адресом и портом. Каждое сообщение автоматически отправляется этому адресату. Кроме того, сокет будет получать сообщения только от этого удаленного узла. Как только связь установлена, генерируется событие `connect` и вызывается `callback`.

# Класс `dgram.Socket` (методы)

- `socket.disconnect()` отвязывает подключенный сокет от его удаленного адреса. Этот метод является синхронным.
- `socket.unref()` позволяет процессу завершиться, если это единственный активный сокет в системе событий. А метод `socket.ref()` не позволяет процессу завершиться, если это единственный оставшийся сокет.
- `socket.remoteAddress()` возвращает объект, содержащий адрес, семейство и порт удаленной конечной точки.
- `socket.send(msg[, offset, length][, port][, address][, callback])` отправляет данные на сокет. Для несвязанных сокетов необходимо указать порт назначения и адрес, иначе будет использоваться связанная с сокетом удаленная конечная точка. Единственный способ узнать наверняка, что дейтаграмма была отправлена, — это использовать `callback`.

# Простейший UDP-сервер

```
// https://gist.github.com/sid24rane/6e6698e93360f2694e310dd347a2e2eb

const udp = require('dgram');
const PORT = 3000;

let server = udp.createSocket('udp4');

server.on('error', (err) => { console.log('Ошибка: ' + err); server.close(); });

server.on('message', (msg, info) => {
  console.log('Server: от клиента получено ' + msg.toString());
  console.log('Server: получено %d байтов от %s:%d\n', msg.length, info.address, info.port);

  server.send(msg, info.port, info.address, (err) => {
    if (err) { server.close(); }
    else { console.log('Server: данные отправлены клиенту'); }
  });
});

server.on('listening', () => {
  console.log('Server: слушает порт ' + server.address().port);
  console.log('Server: ip сервера ' + server.address().address);
  console.log('Server: семейство(IP4/IP6) ' + server.address().family);
});

server.on('close', () => { console.log('Server: сокет закрыт'); });

server.bind(PORT);

//setTimeout(() => { server.close(); }, 8000);
```



# Простейший UDP-клиент

```
const buffer = require('buffer');
const udp = require('dgram');
const client = udp.createSocket('udp4');
const PORT = 3000;

client.on('message', (msg, info) => {
  console.log('Client: от сервера получено ' + msg.toString());
  console.log('Client: получено %d байтов от %s:%d\n', msg.length, info.address, info.port);
});

let data = Buffer.from('Client: сообщение 01');
client.send(data, PORT, 'localhost', (err) => {
  if(err) client.close();
  else console.log('Client: Сообщение отправлено серверу');
});

let data1 = Buffer.from('Привет ');
let data2 = Buffer.from('Мир');

client.send([data1, data2], PORT, 'localhost', (err) => {
  if(err) client.close();
  else console.log('Client: Сообщение отправлено серверу');
});
```

```
const dgram = require('dgram');

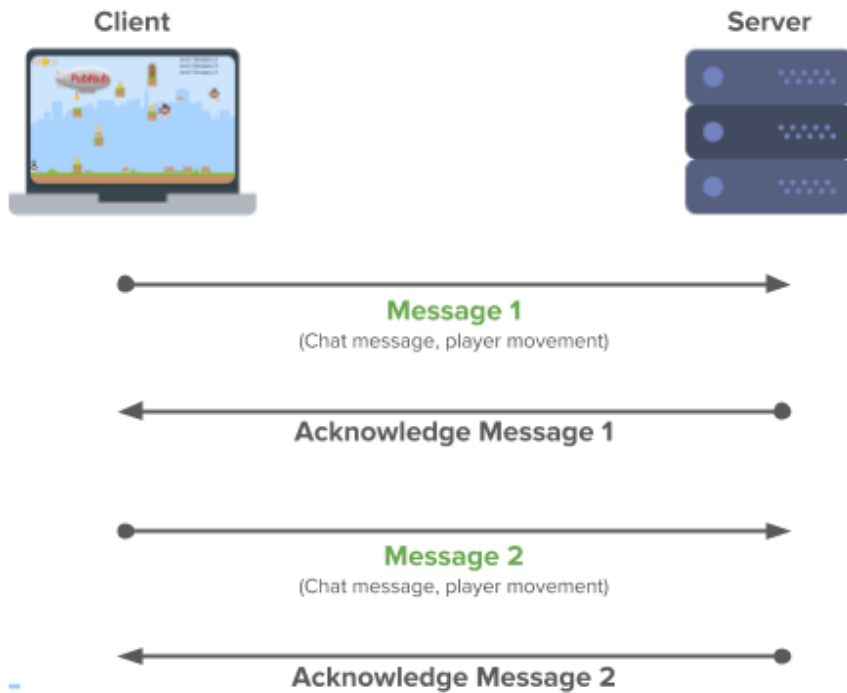
const client = dgram.createSocket('udp4');

const message = Buffer.from('Client: сообщение 01');
client.connect(3000, 'localhost', (err) => {
  client.send(message, (err) => {
    client.close();
  });
});
```

Критерии сравнения	TCP	UDP
Соединение	Требуется установленное соединение для передачи данных	Протокол без соединения
Гарантия доставки	Может гарантировать доставку данных получателю	Не гарантирует доставку данных получателю
Повторная передача данных	Повторная передача нескольких пакетов случае потери одного из них	Отсутствие повторной передачи потерянных пакетов
Проверка ошибок	Полная проверка ошибок	Базовый механизм проверки ошибок. Использует вышестоящие протоколы для проверки целостности
Упорядоченность	Гарантирует правильный порядок	Порядок не соблюдается
Скорость	Низкая	Высокая
Сферы применения	Используется для передачи сообщений электронной почты, HTML-страниц браузеров	Видеоконференции, потоковое вещание, DNS, VoIP, IPTV

# TCP vs UDP

## TCP



## UDP

