

ПРОГРАММИРОВАНИЕ СЕРВЕРНЫХ КРОССПЛАТФОРМЕННЫХ ПРИЛОЖЕНИЙ

WEBSOCKET

WebSocket =

протокол прикладного уровня для **полнодуплексной** связи поверх TCP-соединения, предназначенный для обмена сообщениями между браузером и веб-сервером через постоянное соединение в режиме реального времени.

Данные передаются по нему **в обоих направлениях** в виде «датафреймов», без разрыва соединения и дополнительных HTTP-запросов.

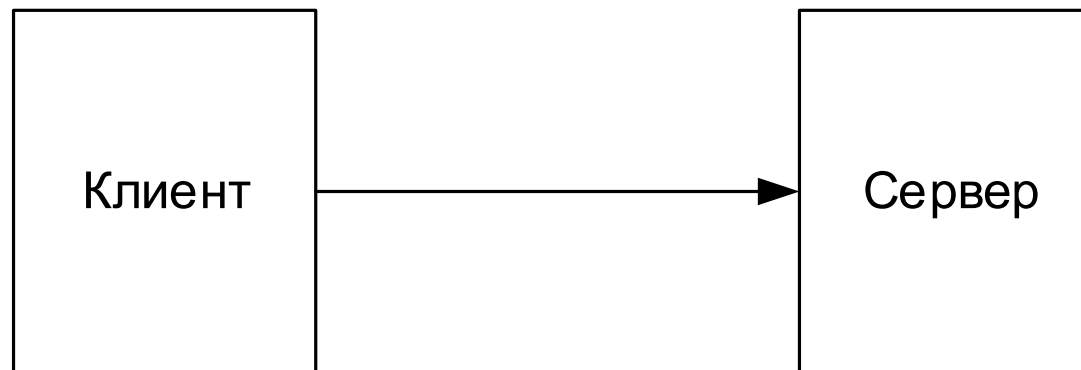


стандарт

Websocket **МОЖНО ИСПОЛЬЗОВАТЬ** в:

- приложениях реального времени;
- чат-приложениях;
- IoT-приложениях;
- приложениях для бирж и аукционов;
- многопользовательских играх;
- и другое.

Симплекс

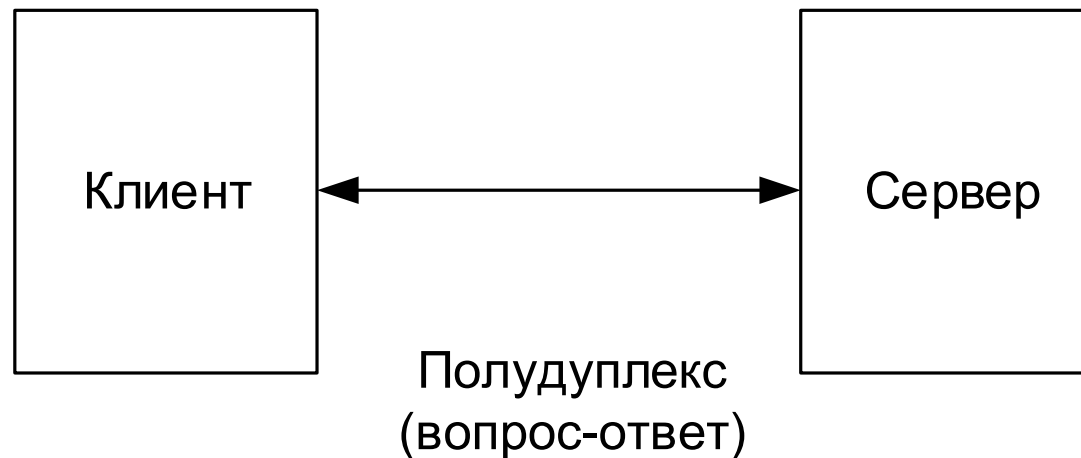


Симплекс
(однонаправленный)

это **односторонний** канал, данные по нему могут **передаваться только в одном направлении**. Первый узел способен отсылать сообщения, второй может только принимать их, но не может подтвердить получение или ответить.

Пример: радио, Mailslot.

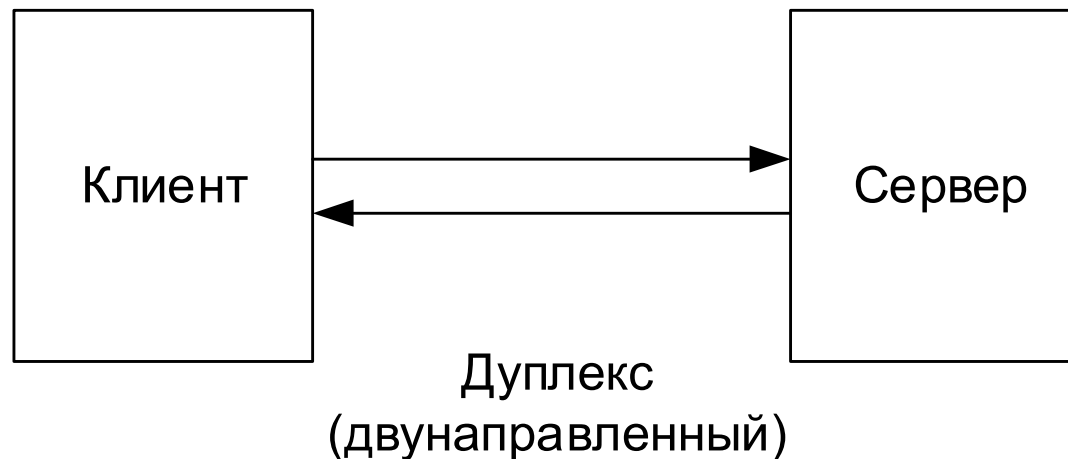
Полудуплекс



двусторонний канал связи, данные по нему могут **передаваться в одном направлении в один момент времени**. Оба абонента имеют возможность принимать и передавать сообщения. Поток сообщений может идти в обоих направлениях, но не одновременно.

Пример: рация, HTTP.

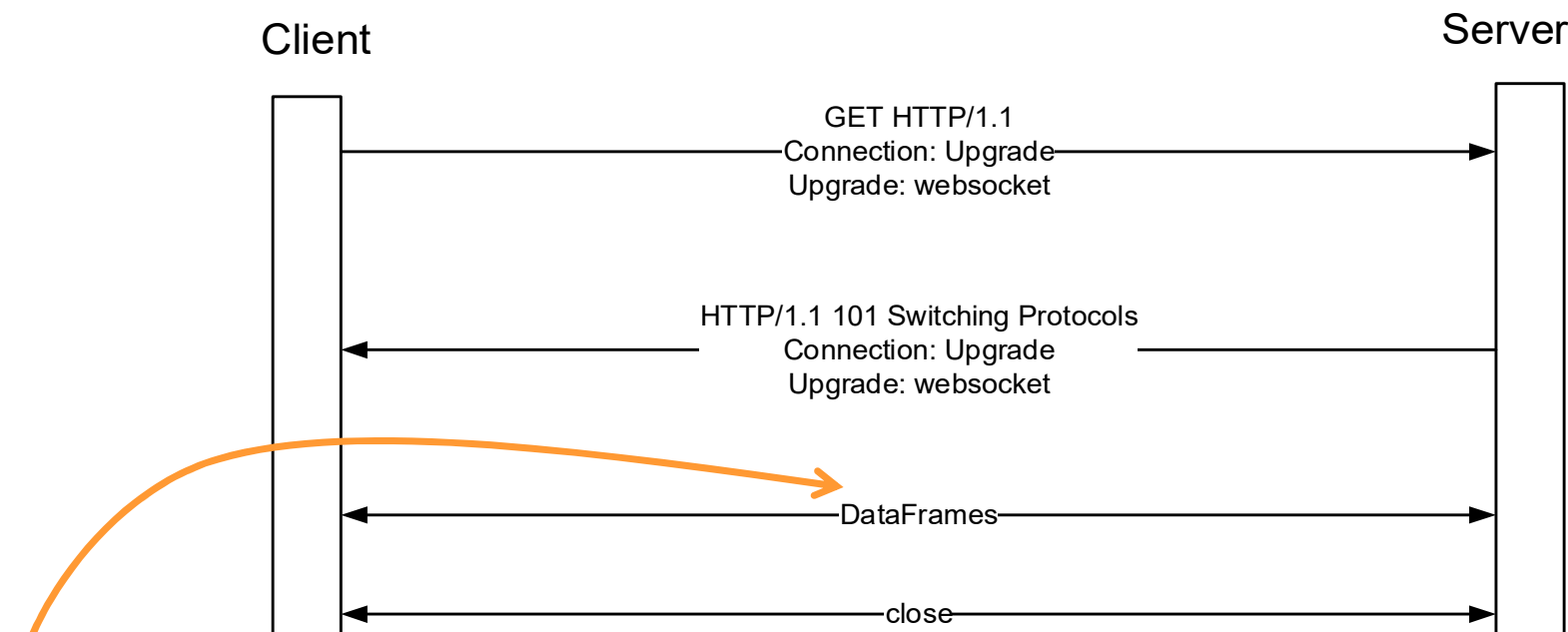
Дуплекс



двусторонний канал связи, данные могут **передаваться в обе стороны одновременно**. Дуплекс имеет два физически отдельных канала связи, один из которых предназначен для движения данных в одном направлении, а другой – в противоположном направлении.

Пример: телефон, WebSocket.

WebSocket-handshake



Датафреймы: текстовые, бинарные, ping-pong, close...

Простейший websocket-сервер

```
const httpserver = require('http').createServer((req, res)=>{
  if (req.method == 'GET' && req.url == '/start' ){
    res.writeHead(200, {'Content-Type': 'text/html; charset=utf-8'});
    res.end(require('fs').readFileSync('./09-01.html'));
  }
});
httpserver.listen(3000)
console.log ('ws server: 3000');

let k = 0;
const WebSocket = require('ws') // npm install ws
const wsserver = new WebSocket.Server({ port: 4000, host:'localhost', path:'/wsserver'})
wsserver.on('connection', (ws) => {
  ws.on('message', message => {
    console.log(`Received message => ${message}`)
  })
  setInterval(()=>{ws.send(`server: ${++k}`)}, 1500);
})
wsserver.on('error',(e)=>{console.log('ws server error', e)});
console.log(`ws server: host:${wsserver.options.host}, port:${wsserver.options.port}, path:${wsserver.options.path}`);
```

Экземпляр класса
WebSocketServer

Экземпляр класса
WebSocket

Событие **connection** генерируется после завершения рукопожатия

Генерируется при получении сообщения.

Класс WebSocketServer

События

- **close** генерируется при закрытии сервера. Это событие зависит от события закрытия HTTP-сервера только в том случае, если он создается внутри.
- **connection** генерируется после завершения рукопожатия. Первым параметром передается сокет (экземпляр класса `WebSocket`), вторым параметром `request` (экземпляр класса `http.IncomingMessage`) – это HTTP-запрос GET, отправленный клиентом.
- **error** генерируется при возникновении ошибки.
- **headers** генерируется перед записью заголовков ответа в сокет в рамках рукопожатия. Первым параметром передается массив заголовком, а вторым – `request` (экземпляр класса `http.IncomingMessage`).
- **listening** генерируется, когда базовый сервер привязан.

Класс WebSocketServer

Свойства

- **clients** – набор, в котором хранятся все подключенные клиенты.

Методы

- **.address()** возвращает объект со свойствами port, family и address, сообщаемые операционной системой при прослушивании сокета.
- **.close([callback])** запрещает серверу принимать новые соединения и закрывает HTTP-сервер, если он создан внутри. Если внешний HTTP-сервер используется через параметры конструктора server или noServer, его необходимо закрыть вручную. Необязательный callback вызывается при возникновении события закрытия и получает ошибку, если сервер уже закрыт.
- **.handleUpgrade(request, socket, head, callback)** обработка запроса на обновление HTTP. Чаще всего вызывается автоматически. При работе в режиме «noServer» этот метод необходимо вызывать вручную.

Простейший websocket-клиент (браузер)

```
<body>
<h1>Lec 09</h1>
<script>
  let k = 0;
  function startWS(){
    let socket = new WebSocket('ws://localhost:4000/wsserver');

    socket.onopen = ()=>{ console.log('socket.onopen');
      setInterval(()=>{socket.send(++k);}, 1000);
    };

    socket.onclose = (e)=>{ console.log('socket.onclose', e);};

    socket.onmessage =(e)=>{console.log('socket.onmessage', e.data)};

    socket.onerror = function(error) { alert("Ошибка " + error.message); };

  };
</script>
<button onclick="startWS()">startWS</button>
</body>
```

Этот класс представляет WebSocket. Он наследует EventEmitter.

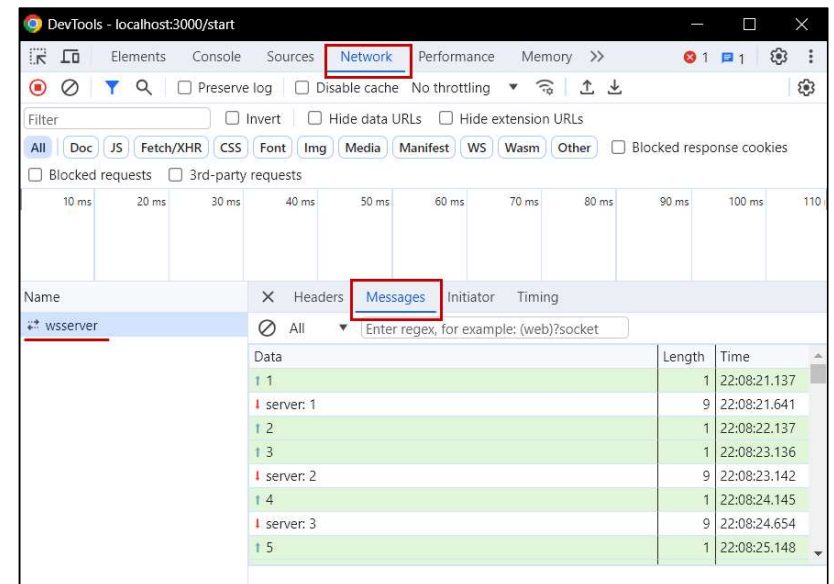
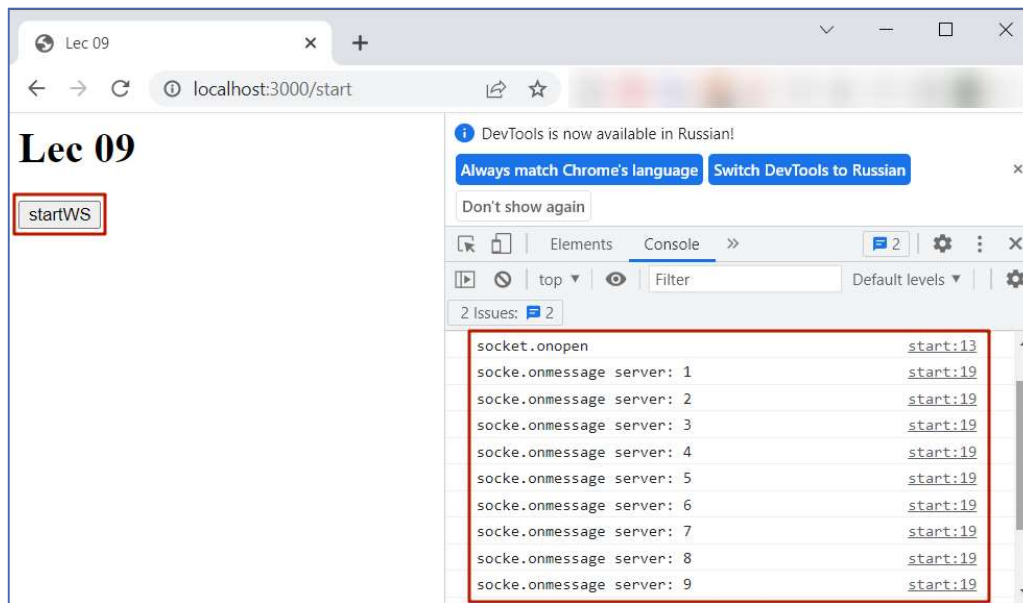
Для установки соединения в скрипте создаем экземпляр **класса WebSocket**, в конструктор которого передаем параметр address (со специальным протоколом **ws** (нешифрованное соединение) или **wss** (зашифрованное соединение)).

У объекта WebSocket можно слушать его события, например:

- **open** – соединение установлено;
- **message** – получены данные;
- **error** – ошибка;
- **close** – соединение закрыто.

```
PS D:\NodeJS\samples\cwp_08> node
08-08
ws server: 3000
ws server: host:localhost, port:40
00, path:/wsserver
Received message => 1
Received message => 2
Received message => 3
Received message => 4
Received message => 5
Received message => 6
```

Демонстрация работы



Класс WebSocket

События

- **close** генерируется при закрытии соединения. Первым параметром передается code – числовое значение, указывающее код состояния, объясняющий, почему соединение было закрыто, вторым reason – это буфер, содержащий строку, объясняющую, почему соединение было закрыто.
- **error** генерируется при возникновении ошибки.
- **message** генерируется при получении сообщения. Первым параметром передается data – это содержимое сообщения, вторым isBinary – указывает, является ли сообщение двоичным или нет.
- **open** генерируется при установлении соединения.
- **ping** генерируется при получении ping.
- **pong** генерируется при получении pong.
- **upgrade** генерируется, когда заголовки ответа получены от сервера как часть рукопожатия.

Класс WebSocket

Методы

для браузерного клиента

- `.addEventListener(type, listener[, options])` добавить слушатель `listener` на событие `type`.
- `.removeEventListener(type, listener)` удалить слушатель `listener` с события `type`.
- `.send(data[, options][, callback])` отправляет данные. В `options` можно настроить в бинарном ли виде должны быть отправлены данные(`binary`), надо ли их сжимать(`compress`), последний ли это фрагмент сообщения(`fin`) и надо ли маскировать передаваемые данные(`mask`).
- `.close([code[, reason]])` инициирует заключительное рукопожатие.
- `.ping([data[, mask]][, callback])` отправляет фрейм `ping`. В `data` можно передать данные для отправки во фрейме.
- `.pong([data[, mask]][, callback])` отправляет фрейм `pong`. В `data` можно передать данные для отправки во фрейме.
- `.terminate()` принудительно закрывает соединение. Внутри это вызывает метод `socket.destroy()`.

Класс WebSocket

Свойства

для браузерного клиента

- **binaryType** указывает тип двоичных данных, передаваемых по соединению.
- **bufferedAmount** указывает количество байтов данных, которые были поставлены в очередь с помощью вызовов `send()`, но еще не переданы в сеть.
- **onclose** позволяет добавить слушатель событий, который будет вызываться при закрытии соединения
- **onerror** позволяет добавить слушатель событий, который будет вызываться при возникновении ошибки.
- **onmessage** позволяет добавить слушатель событий, который будет вызываться при получении сообщения.
- **onopen** позволяет добавить слушатель событий, который будет вызываться при установке соединения.
- **readyState** содержит текущее состояние соединения
- **url** содержит URL-адрес сервера WebSocket. Серверные клиенты не имеют этого атрибута.

Простейший websocket-клиент (серверный)

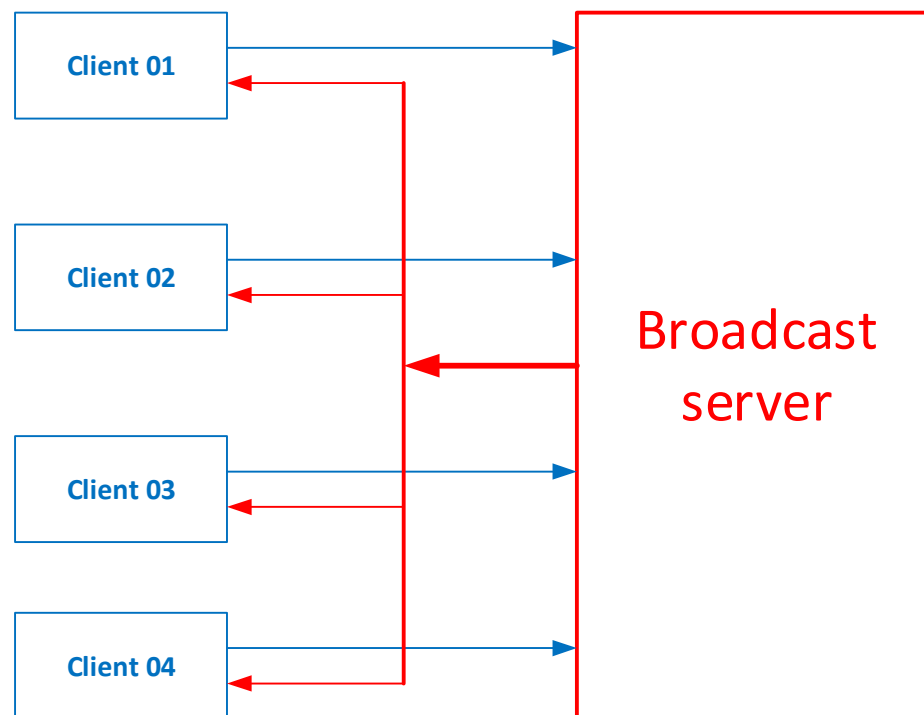
```
const WebSocket = require('ws');  
  
const ws = new WebSocket('ws://localhost:4000/wsserver');  
  
ws.on('open', () => {  
  ws.send('message 1'); // отправить сообщение серверу  
  ws.send('message 2'); // отправить сообщение серверу  
  ws.on('message', message => {  
    console.log(`Received message => ${message}`)  
  })  
  setTimeout(() => {ws.close()}, 5000); // остановить через 5 секунд  
});
```

С помощью метода `close()` иницилируем завершающее рукопожатие и закрываем соединение.

```
PS D:\NodeJS\samples\cwp_08> node .\08-00.js  
ws server: 3000  
ws server: host:localhost, port:4000, path:/wsserver  
Received message => message 1  
Received message => message 2  
□
```

```
PS D:\NodeJS\samples\cwp_08> node .\08-00b.js  
Received message => server: 1  
Received message => server: 2  
Received message => server: 3  
PS D:\NodeJS\samples\cwp_08> □
```


Широковещательный websocket-сервер



Такой сервер позволяет отправлять сообщения, которые доставляются
всем узлам сети.

Широковещательный websocket-сервер

```
const WebSocket = require('ws');  
  
const wss = new WebSocket.Server({ port: 5000, host: 'localhost', path: '/broadcast' });  
  
wss.on('connection', (ws) => {  
  ws.on('message', (message) => {  
    wss.clients.forEach((client) => {  
      if (client.readyState === WebSocket.OPEN) client.send('server: ' + message);  
    });  
  })  
  
  ws.on('close', () => { console.log('wss socket closed'); })  
});
```

Свойство `clients` хранит набор со всеми подключенными клиентами.

Свойство `readyState` содержит текущее состояние соединения (CONNECTING, OPEN, CLOSING, CLOSED).

Клиент для широковещательного сервера с параметрами командной строки

```
const WebSocket = require('ws');

let parm0 = process.argv[0]; // path node
let parm1 = process.argv[1]; // path application
let parm2 = process.argv[2]; // first parameter

console.log('parm2 = ', parm2);

let prfx = typeof parm2 == 'undefined'? 'A': parm2;
const ws = new WebSocket('ws://localhost:5000/broadcast');

ws.on('open', () =>{

  let k = 0;
  setInterval(()=>{
    ws.send(`client: ${prfx}-${++k}`); // отправить сообщение серверу
  }, 1000);

  ws.on('message', message => {
    console.log(`Received message => ${message}`)
  })

  setTimeout(()=>{ws.close();}, 25000); // остановить через 25 секунд
});
```

При запуске приложения из терминала/командной строки мы можем передавать процессу **параметры**. Для получения параметров в коде приложения применяется массив **process.argv**.

```
PS D:\NodeJS\samples\cwp_08> node .\08-01.js
ws socket closed
ws socket closed
```

```
PS D:\NodeJS\samples\cwp_08> node .\08-01a.js M
param2 = M
Received message => server: client: M-1
Received message => server: client: M-2
Received message => server: client: K-1
Received message => server: client: M-3
Received message => server: client: K-2
Received message => server: client: M-4
Received message => server: client: K-3
Received message => server: client: M-5
Received message => server: client: K-4
Received message => server: client: M-6
```

```
PS D:\NodeJS\samples\cwp_08> node .\08-01a.js K
param2 = K
Received message => server: client: M-2
Received message => server: client: K-1
Received message => server: client: M-3
Received message => server: client: K-2
Received message => server: client: M-4
Received message => server: client: K-3
Received message => server: client: M-5
Received message => server: client: K-4
Received message => server: client: M-6
Received message => server: client: K-5
```

Дуплексный поток данных

```
const WebSocket = require('ws');  
  
const ws = new WebSocket('ws://localhost:5000/broadcast');  
  
const duplex = WebSocket.createWebSocketStream(ws, { encoding: 'utf8' });  
  
duplex.pipe(process.stdout); // сообщения от сервера --> stdout  
  
process.stdin.pipe(duplex); // stdin --> сообщение серверу
```

```
PS D:\NodeJS\samples\cwp_08> node .\08-01.js  
wss socket closed  
█
```

```
PS D:\NodeJS\samples\cwp_08> node .\08-01b.js  
Hello from M  
server: Hello from M  
server: Hello from K  
PS D:\NodeJS\samples\cwp_08> █
```

```
PS D:\NodeJS\samples\cwp_08> node .\08-01b.js  
server: Hello from M  
Hello from K  
server: Hello from K  
█
```

`createWebSocketStream(websocket[, options])` возвращает **дуплексный поток**, который позволяет использовать API потоков Node.js поверх заданного WebSocket.

Механизм ping/pong (сервер => клиент)

сервер

```
const WebSocket = require('ws');
const wss = new WebSocket.Server({ port: 5000, host: 'localhost' });
wss.on('connection', (ws) => {
  ws.on('pong', (data) => {
    console.log('on pong: ', data.toString());
  });
  ws.on('message', (data) => {
    console.log('on message: ', data.toString());
    ws.send(data);
  });
  setInterval(() => { console.log('server: ping'); ws.ping('server: ping') }, 5000);
});
```

В протокол WebSocket встроена проверка связи при помощи **управляющих фреймов типа PING и PONG**.

Тот, кто хочет проверить соединение, отправляет фрейм PING с произвольным телом. Его получатель должен в разумное время ответить фреймом PONG с тем же телом.

клиент

```
const WebSocket = require('ws');

const ws = new WebSocket('ws://localhost:5000');

const duplex = WebSocket.createWebSocketStream(ws, { encoding: 'utf8' });

duplex.pipe(process.stdout); // сообщения от сервера --> stdout

process.stdin.pipe(duplex); // stdin --> сообщение серверу

// ws.on('ping', (data) => { // можно ловить
//   console.log('on ping: ', data.toString());
// });
```

PS D:\NodeJS\samples\cwp_08> node .\08-02.js	PS D:\NodeJS\samples\cwp_08> node .\08-02a.js
server: ping	
on pong: server: ping	Hi!
server: ping	Hi!
on pong: server: ping	
on message: Hi!	
server: ping	
on pong: server: ping	
server: ping	
on pong: server: ping	
server: ping	
on pong: server: ping	
□	

результат

Механизм ping/pong

(клиент => сервер)

сервер

```
const WebSocket = require('ws');
const wss = new WebSocket.Server({ port: 5000, host: 'localhost' });
wss.on('connection', (ws) => {
  ws.on('message', (data) => {
    console.log('on message: ', data.toString());
    ws.send(data);
  });
});
wss.on('error', (e) => { console.log('wss server error', e); });
```

клиент

```
const WebSocket = require('ws');

const ws = new WebSocket('ws://localhost:5000');

const duplex = WebSocket.createWebSocketStream(ws, { encoding: 'utf8' });

duplex.pipe(process.stdout); // сообщения от сервера --> stdout

process.stdin.pipe(duplex); // stdin --> сообщение серверу

ws.on('pong', (data) => { // можно ловить
  console.log('on pong: ', data.toString());
});

setInterval(() => { console.log('server: ping'); ws.ping('client: ping'); }, 5000);
```

Событие ping возникает, когда получен фрейм PING.

Событие pong возникает, когда получен фрейм PONG.

Метод ping() отправляет PING, в параметрах можно передать какую-то информацию.

Метод pong() отправляет PONG. Однако сообщения PONG чаще всего автоматически отправляются в ответ на сообщения PING, как того требует спецификация.

```
PS D:\NodeJS\samples\cwp_08> node .\08-03.js
on message: HI!

client: ping
on pong: client: ping
client: ping
on pong: client: ping
HI!
client: ping
on pong: client: ping
client: ping
on pong: client: ping
client: ping
```

результат

Отправка JSON

сервер

```
const WebSocket = require('ws');
const wss = new WebSocket.Server({ port: 5000, host: 'localhost' });
wss.on('connection', (ws) => {

  ws.on('message', (data) => { console.log('on message: ', JSON.parse(data)); });

  let k = 0;
  setInterval(() => { ws.send(JSON.stringify({k:++k, sender: 'Server', date: new Date().toISOString()})); }, 5000);

});
```

КЛИЕНТ

```
const WebSocket = require('ws');
const ws = new WebSocket('ws://localhost:5000');
ws.on('open', () => {
  ws.on('message', data => { console.log('on message: ', JSON.parse(data)); })
  let k = 0;
  setInterval(() => { ws.send(JSON.stringify({k:++k, sender: 'Client', date: new Date().toISOString()})); }, 3000);
});
```

```
PS D:\NodeJS\samples\cwp_08> node .\08-04.js
on message: { k: 1, sender: 'Client', date:
'2022-10-28T22:30:05.136Z' }
on message: { k: 2, sender: 'Client', date:
'2022-10-28T22:30:10.148Z' }
on message: { k: 3, sender: 'Client', date:
'2022-10-28T22:30:15.162Z' }
█
```

```
PS D:\NodeJS\samples\cwp_08> node .\08-04a.js
on message: { k: 1, sender: 'Server', date:
'2022-10-28T22:30:05.136Z' }
on message: { k: 2, sender: 'Server', date:
'2022-10-28T22:30:10.148Z' }
on message: { k: 3, sender: 'Server', date:
'2022-10-28T22:30:15.162Z' }
█
```

Upload file

сервер

```
const fs = require('fs');
const WebSocket = require('ws');

const wss = new WebSocket.Server({ port: 5000, host: 'localhost' });
let k = 0;
wss.on('connection', (ws) => {
  const duplex = WebSocket.createWebSocketStream(ws, { encoding: 'utf8' });
  let wfile = fs.createWriteStream(`./file${++k}.txt`);
  duplex.pipe(wfile);
});
```

1. получаем дуплексный поток
2. открываем поток записи в файл
3. информацию, поступающую из дуплексного потока, записываем в поток записи файла

клиент

```
const fs = require('fs');
const WebSocket = require('ws');
const ws = new WebSocket('ws://localhost:5000');
ws.on('open', () => {
  const duplex = WebSocket.createWebSocketStream(ws, { encoding: 'utf8' });
  let rfile = fs.createReadStream(`./MyFile.txt`);
  rfile.pipe(duplex);
});
```

1. получаем дуплексный поток
2. открываем поток чтения из файла
3. Информацию из потока чтения перенаправляем в дуплексный поток

Download file

сервер

```
const fs = require('fs');
const WebSocket = require('ws');

const wss = new WebSocket.Server({ port: 5000, host: 'localhost' });
wss.on('connection', (ws) => {
  const duplex = WebSocket.createWebSocketStream(ws, { encoding: 'utf8' });
  let rfile = fs.createReadStream('./MyFile.txt');
  rfile.pipe(duplex);
});
```

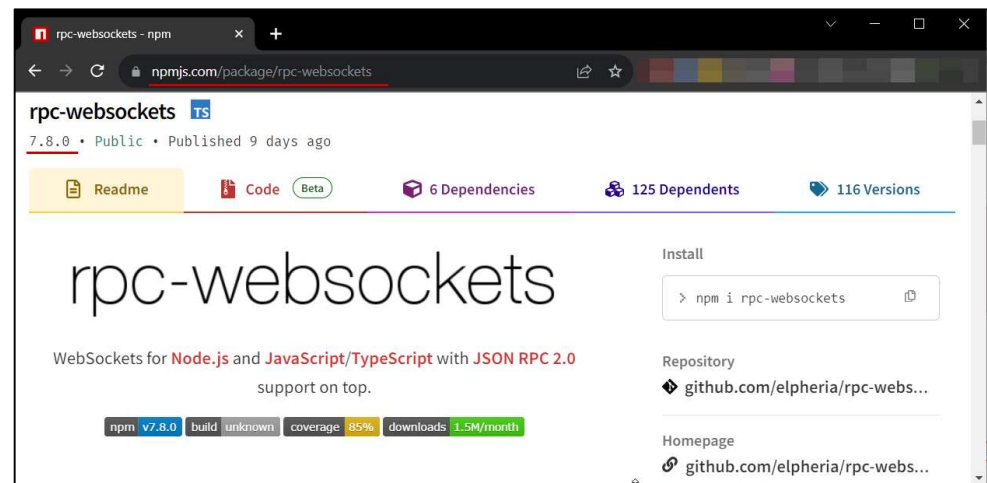
То же самое, но наоборот.

КЛИЕНТ

```
const fs = require('fs');
const WebSocket = require('ws');
const ws = new WebSocket('ws://localhost:5000');
let k = 0;
ws.on('open', () => {
  const duplex = WebSocket.createWebSocketStream(ws, { encoding: 'utf8' });
  let wfile = fs.createWriteStream(`./MyFile${++k}.txt`);
  duplex.pipe(wfile);
});
```

rpc-websockets=

пакет, позволяющий создавать WebSocket-сервер и клиента с поддержкой протокола JSON-RPC 2.0.



JSON-RPC =

протокол удаленного вызова процедур, использующий формат JSON для передачи сообщений. Используется для создания API в стиле RPC.



Создание удаленных процедур

```
const rpcWSS = require('rpc-websockets').Server;

let server = new rpcWSS({port: 5000, host: 'localhost'});

server.register('sum', (params) =>{ return params[0] + params[1]}).public();
server.register('sub', (params) =>{ return params[0] - params[1]}).public();
server.register('mul', (params) =>{ return params[0] * params[1]}).public();
server.register('div', (params) =>{ return params[0] / params[1]}).public();
server.register('get', (params) =>{
    // сходить в БД за данными
    return {id:params[0], name:'Иванов И.И.', bday: '2000-12-02'};
}).public();
```

Создаем экземпляр класса Server, указываем порт и хост.

Метод `.public()` помечает метод RPC как общедоступный

Метод `.register(method, handler[, namespace])` регистрирует метод RPC и возвращает объект `RPCMethod` для управления разрешениями метода. Первым параметром передаем название метода RPC, вторым – функцию, которая будет запущена после вызова метода (принимает только один параметр(массив, одиночное значение или др.)).

Вызов удаленных процедур

```
const rpcWSC = WebSocket = require('rpc-websockets').Client
let ws = new rpcWSC('ws://localhost:5000');

ws.on('open', ()=>{
  ws.call('sum', [5, 3]).then((r)=>{console.log('sum = ', r)});
  ws.call('sub', [5, 3]).then((r)=>{console.log('sub = ', r)});
  ws.call('mul', [5, 3]).then((r)=>{console.log('mul = ', r)});
  ws.call('div', [5, 3]).then((r)=>{console.log('div = ', r)});
  ws.call('get', [123], 3000).catch((e)=>{return e}).then((r)=>{console.log('get_student = ', r)});
});
ws.on('error', (e)=>{console.log('error = ', e)});
```

Создаем экземпляр класса Client. В параметрах передаем адрес сервера.

С помощью метода `.call(method[, params[, timeout[, ws_options]])` происходит вызов зарегистрированного метода RPC на сервере, метод возвращает Promise.

```
PS D:\NodeJS\samples\cwp_08> node .\08-07.js
sum = 8
sub = -8
mul = 15
div = 0.06666666666666667
get_student = { id: 123, name: 'Иванов И.И.', bday: '2000-12-02' }
```

Создание защищенных удаленных процедур

```
const rpcWSS = require('rpc-websockets').Server;

let server = new rpcWSS({port: 5000, host: 'localhost'});

server.setAuth((l) => { return (l.login == 'smw' && l.password == '777') });

server.register('sum', (params) => { return params[0] + params[1] }).public();
server.register('sub', (params) => { return params[0] - params[1] }).public();
server.register('mul', (params) => { return params[0] * params[1] }).public();
server.register('div', (params) => { return params[0] / params[1] }).public();
server.register('get', (params) => { return {id: params[0], name: 'Иванов И.И.', bday: '2000-12-02'}; }).public();

server.register('mod', (params) => { return params[0] % params[1] }).protected();
server.register('abs', (params) => { return params[0] >= 0 ? params[0] : -params[0] }).protected();
```

Метод `.protected()` помечает метод RPC как защищенный. Метод будет доступен только в том случае, если клиент успешно прошел аутентификацию.

Метод `.setAuth(handler[, namespace])` устанавливает определяемый пользователем метод аутентификации. Функция обработчика должна возвращать логическое значение.

Вызов защищенных процедур

```
const rpcWSC = WebSocket = require('rpc-websockets').Client;
let ws = new rpcWSC('ws://localhost:5000/');

ws.on('open', () => {
  ws.call('sum', [5, 3]).then((r) => { console.log('sum = ', r); });
  ws.call('sub', [5, 3]).then((r) => { console.log('sub = ', r); });
  ws.call('mul', [5, 3]).then((r) => { console.log('mul = ', r); });
  ws.call('div', [5, 3]).then((r) => { console.log('div = ', r); });
  ws.call('get', [123])
    .then((r) => { console.log('get_student = ', r); })
    .catch((e) => { return e; });

  ws.login({ login: 'smw', password: '777' })
    .then((login) => {
      if (login) {
        ws.call('mod', [33, 5]).then((r) => { console.log('mod = ', r); });
        ws.call('abs', [-33])
          .then((r) => { console.log('abs = ', r); })
          .catch((e) => { console.log('catch abs: ', e); });
      }
      else console.log('login error');
    })
    .catch((e) => { console.log('login failed: ', e.message); });

  //ws.close(); // асинхронность! вызывает ошибку
});

ws.on('error', (e) => console.log('error = ', e));
```

Метод **.login** используется для того, чтобы аутентифицироваться с другой стороны подключения. Метод возвращает Promise.

```
PS D:\NodeJS\samples\cwp_08> node .\08-08a.js
sum = 8
sub = -8
mul = 15
div = 0.06666666666666667
get_student = { id: 123, name: 'Иванов И.И.', bday: '2000-12-02' }
mod = 3
abs = 33
login failed: authentication failed
```

```
PS D:\NodeJS\samples\cwp_08> node .\08-08a.js
sum = 8
sub = -8
mul = 15
div = 0.06666666666666667
get_student = { id: 123, name: 'Иванов И.И.', bday: '2000-12-02' }
mod = 3
abs = 33
```


Асинхронный параллельный вызов процедур

```
const async = require('async');
const rpcWSC = WebSocket = require('rpc-websockets').Client;

let ws = new rpcWSC('ws://localhost:5000');
let h = (x=ws)=>async.parallel({
  sum: (cb)=>{ ws.call('sum', [5, 3]).catch((e)=>cb(e,null)).then((r)=>cb(null,r));},
  sub: (cb)=>{ ws.call('sub', [5, 3]).catch((e)=>cb(e,null)).then((r)=>cb(null,r));},
  mul: (cb)=>{ ws.call('mul', [5, 3]).catch((e)=>cb(e,null)).then((r)=>cb(null,r));},
  div: (cb)=>{ ws.call('div', [5, 3]).catch((e)=>cb(e,null)).then((r)=>cb(null,r));},
  get: (cb)=>{ ws.call('get', [123],3000).catch((e)=>cb(e,null)).then((r)=>cb(null,r));},
  mod: (cb)=>{
    ws.login({login: 'smw', password: '777'})
    .then((login)=>{
      if (login) ws.call('mod', [33, 5]).catch((e)=>cb(e,null)).then((r)=>cb(null,r));
      else cb({message1: 'login error'}, null);
    })
  },
  abs: (cb)=>{
    ws.login({login: 'smw', password: '777'})
    .then((login)=>{
      if (login) ws.call('abs', [-33]).catch((e)=>cb(e,null)).then((r)=>cb(null,r));
      else cb({message2: 'login error'}, null);
    })
  }
},
(e, r)=>{
  if(e) console.log('e =',e);
  else console.log('r =',r);
  ws.close();
});
ws.on('open', h);
```

Метод .close класса Client закрывает Websocket соединение

Метод `async.parallel()` используется для параллельного выполнения нескольких асинхронных операций.

```
PS D:\NodeJS\samples\cwp_08> node .\08-08b.js
r = {
  sum: 8,
  sub: -8,
  mul: 15,
  div: 0.06666666666666667,
  get: { id: 123, name: 'Иванов И.И.', bday: '2000-12-02' },
  mod: 3,
  abs: 33
}
PS D:\NodeJS\samples\cwp_08>
```


Асинхронный последовательный вызов процедур

```
const async = require('async');
const rpcWSC = WebSocket = require('rpc-websockets').Client;
// |(((8+3)-2)*(-4)/2)%4|
let ws = new rpcWSC('ws://localhost:5000');
let h = ()=>async.waterfall([
  (cb)=>{ ws.call('sum', [8, 3]).catch((e)=>cb(e,null)).then((r)=>cb(null,r));},
  (p,cb)=>{ ws.call('sub', [p, 2]).catch((e)=>cb(e,null)).then((r)=>cb(null,r));},
  (p,cb)=>{ ws.call('mul', [p, -4]).catch((e)=>cb(e,null)).then((r)=>cb(null,r));},
  (p,cb)=>{ ws.call('div', [p, 2]).catch((e)=>cb(e,null)).then((r)=>cb(null,r));},
  (p,cb)=>{
    ws.login({login: 'smw', password: '777'})
    .then((login)=>{
      if (login) ws.call('mod', [p, 4]).catch((e)=>cb(e,null)).then((r)=>cb(null,r));
      else cb({message1: 'login error'}, null);
    })
  },
  (p,cb)=>{
    ws.login({login: 'smw', password: '777'})
    .then((login)=>{
      if (login) ws.call('abs', [-p]).catch((e)=>cb(e,null)).then((r)=>cb(null,r));
      else cb({message2: 'login error'}, null);
    })
  }
],
  (e, r)=>{
    if(e) console.log('e =',e);
    else console.log('r =',r);
    ws.close();
  }
);
ws.on('open', h);
```

Выполнение нескольких асинхронных операций последовательно, когда каждая операция зависит от результатов предыдущих операций, осуществляется **методом** `async.waterfall()`.

```
PS D:\NodeJS\samples\cwp_08> node .\08-08c.js
r = 0.009615384615384616
PS D:\NodeJS\samples\cwp_08> 
```

События

сервер

```
const rpcWSS = require('rpc-websockets').Server;

let k = 0;

let server = new rpcWSS({port: 5000, host: 'localhost'});
```

```
server.event('event01');
server.event('event02');
server.event('event03');
```

Метод `server.event(name[,namespace])` создает новое событие и возвращает объект `RPCMethod`.

```
setInterval(()=>server.emit('event01', {n:++k, x:1, y:2}),1000);
setInterval(()=>server.emit('event02', {n:++k, s:'hello', d:'2019-09-09'}),2000);
setInterval(()=>server.emit('event03', {n:++k}), 3000);
```

Метод `server.emit(name[, ...params])` генерирует созданное событие. После первого параметра можно передать параметры, которые будут переданы клиентам.

КЛИЕНТ

```
const rpcWSC = WebSocket = require('rpc-websockets').Client;
let ws = new rpcWSC('ws://localhost:5000');
```

```
ws.on('open', ()=>{
```

```
ws.subscribe('event01');
ws.subscribe('event02');
ws.subscribe('event03');
```

Метод `ws.subscribe(event)` позволяет подписаться на определенное событие.

```
ws.on('event01', (p)=>{console.log('event01:', p)});
ws.on('event02', (p)=>console.log('event02:', p));
ws.on('event03', (p)=>console.log('event03:', p));
```

```
});
```

Метод `ws.on(...)` позволяет указать обработчик события, на которое подписался клиент.

```
PS D:\NodeJS\samples\cwp_08> node .\08-09.js
```

```
█
```

```
PS D:\NodeJS\samples\cwp_08> node .\08-09a.js
```

```
event02: { n: 1, s: 'hello', d: '2019-09-09' }
event01: { n: 2, x: 1, y: 2 }
event03: { n: 3 }
event01: { n: 4, x: 1, y: 2 }
event02: { n: 5, s: 'hello', d: '2019-09-09' }
event01: { n: 6, x: 1, y: 2 }
event01: { n: 7, x: 1, y: 2 }
```

Уведомления

похоже на запрос без id, ответ на него не будет возвращен.

сервер

```
const rpcWSS = require('rpc-websockets').Server;

let server = new rpcWSS({port: 5000, host: 'localhost'});

server.register('notify1', (params) =>{ console.log('notify1', params)}).public();
server.register('notify2', (params) =>{ console.log('notify2', params)}).public();
```

клиент

```
const rpcWSC = WebSocket = require('rpc-websockets').Client;
let ws = new rpcWSC('ws://localhost:5000');

let k = 0;
ws.on('open', ()=>{

  setInterval(()=>ws.notify('notify1', {n:++k, x:1, y:2}),1000);
  setInterval(()=>ws.notify('notify2', {n:++k, x:1, y:2}),5000);

});
```

Метод `ws.notify()` отправляет на сервер уведомление JSON-RPC 2.0. Можно к запросу добавить параметры.

```
PS D:\NodeJS\samples\cwp_08> node .\08-10.js
notify1 { n: 0, x: 1, y: 2 }
notify1 { n: 1, x: 1, y: 2 }
notify1 { n: 2, x: 1, y: 2 }
notify1 { n: 3, x: 1, y: 2 }
notify2 { n: 4, x: 1, y: 2 }
notify1 { n: 5, x: 1, y: 2 }
notify1 { n: 6, x: 1, y: 2 }
notify1 { n: 7, x: 1, y: 2 }
█
```

```
PS D:\NodeJS\samples\cwp_08> node .\08-10a.js
█
```