Net

> Stability: 2 - Stable

The `net` module provides an asynchronous network API for creating stream-based TCP or IPC servers (`net.createServer()`) and clients (`net.createConnection()`).

It can be accessed using:

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

IPC support

The `net` module supports IPC with named pipes on Windows, and Unix domain sockets on other operating systems.

Identifying paths for IPC connections

`net.connect()`, `net.createConnection()`, `server.listen()` and `socket.connect()` take a `path` parameter to identify IPC endpoints.

On Unix, the local domain is also known as the Unix domain. The path is a filesystem pathname. It gets truncated to an OS-dependent length of `sizeof(sockaddr_un.sun_path) - 1`. Typical values are 107 bytes on Linux and 103 bytes on macOS. If a Node.js API abstraction creates the Unix domain socket, it will unlink the Unix domain socket as well. For example, `net.createServer()` may create a Unix domain socket and `server.close()` will unlink it. But if a user creates the Unix domain socket outside of these abstractions, the user will need to remove it. The same applies when a Node.js API creates a Unix domain socket but the program then crashes. In short, a Unix domain socket will be visible in the filesystem and will persist until unlinked.

On Windows, the local domain is implemented using a named pipe. The path *must* refer to an entry in `\\?\pipe\` or `\\.\pipe\`. Any characters are permitted, but the latter may do some processing of pipe names, such as resolving `..` sequences. Despite how it might look, the pipe namespace is flat. Pipes will *not persist*. They are removed when the last reference to them is closed. Unlike Unix domain sockets, Windows will close and remove the pipe when the owning process exits.

JavaScript string escaping requires paths to be specified with extra backslash escaping such as:

```
net.createServer().listen(
  path.join('\\\\?\\pipe', process.cwd(), 'myctl'));
```

Class: `net.BlockList`

The `BlockList` object can be used with some network APIs to specify rules for disabling inbound or outbound access to specific IP addresses, IP ranges, or IP subnets.

`blockList.addAddress(address[, type])`

- `address` {string|net.SocketAddress} An IPv4 or IPv6 address.
- `type` {string} Either `'ipv4'` or `'ipv6'`. **Default:** `'ipv4'`.

Adds a rule to block the given IP address.

`blockList.addRange(start, end[, type])`

- `start` {string|net.SocketAddress} The starting IPv4 or IPv6 address in the range.
- `end` {string|net.SocketAddress} The ending IPv4 or IPv6 address in the range.
- `type` {string} Either `'ipv4'` or `'ipv6'`. **Default:** `'ipv4'`.

Adds a rule to block a range of IP addresses from `start` (inclusive) to `end` (inclusive).

`blockList.addSubnet(net, prefix[, type])`

- `net` {string|net.SocketAddress} The network IPv4 or IPv6 address.
- `prefix` {number} The number of CIDR prefix bits. For IPv4, this must be a value between `0` and `32`. For IPv6, this must be between `0` and `128`.
- `type` {string} Either `'ipv4'` or `'ipv6'`. **Default:** `'ipv4'`.

Adds a rule to block a range of IP addresses specified as a subnet mask.

`blockList.check(address[, type])`

- `address` {string|net.SocketAddress} The IP address to check
- `type` {string} Either `'ipv4'` or `'ipv6'`. **Default:** `'ipv4'`.
- Returns: {boolean}

Returns `true` if the given IP address matches any of the rules added to the `BlockList`.

```js
const blockList = new net.BlockList();
blockList.addAddress('123.123.123.123');
blockList.addRange('10.0.0.1', '10.0.0.10');
blockList.addSubnet('8592:757c:efae:4e45::', 64, 'ipv6');

console.log(blockList.check('123.123.123.123'));  // Prints: true
console.log(blockList.check('10.0.0.3'));  // Prints: true
console.log(blockList.check('222.111.111.222'));  // Prints: false
```

```
// IPv6 notation for IPv4 addresses works:
console.log(blockList.check('::ffff:7b7b:7b7b', 'ipv6')); // Prints: true
console.log(blockList.check('::ffff:123.123.123.123', 'ipv6')); // Prints: true
```

`blockList.rules`

- Type: {string[]}

The list of rules added to the blocklist.

Class: `net.SocketAddress`

`new net.SocketAddress([options])`

- `options` {Object}
  - `address` {string} The network address as either an IPv4 or IPv6 string. **Default**: `'127.0.0.1'` if `family` is `'ipv4'`; `'::'` if `family` is `'ipv6'`.
  - `family` {string} One of either `'ipv4'` or `'ipv6'`. **Default**: `'ipv4'`.
  - `flowlabel` {number} An IPv6 flow-label used only if `family` is `'ipv6'`.
  - `port` {number} An IP port.

`socketaddress.address`

- Type {string}

`socketaddress.family`

- Type {string} Either `'ipv4'` or `'ipv6'`.

`socketaddress.flowlabel`

- Type {number}

`socketaddress.port`

- Type {number}

Class: `net.Server`

- Extends: {EventEmitter}

This class is used to create a TCP or IPC server.

`new net.Server([options][, connectionListener])`

- `options` {Object} See `net.createServer([options][, connectionListener])`.
- `connectionListener` {Function} Automatically set as a listener for the `'connection'` event.
- Returns: {net.Server}

`net.Server` is an `EventEmitter` with the following events:

Event: `'close'`

Emitted when the server closes. If connections exist, this event is not emitted until all connections are ended.

Event: `'connection'`

- {net.Socket} The connection object

Emitted when a new connection is made. `socket` is an instance of `net.Socket`.

Event: `'error'`

- {Error}

Emitted when an error occurs. Unlike `net.Socket`, the `'close'` event will **not** be emitted directly following this event unless `server.close()` is manually called. See the example in discussion of `server.listen()`.

Event: `'listening'`

Emitted when the server has been bound after calling `server.listen()`.

`server.address()`

- Returns: {Object|string|null}

Returns the bound `address`, the address `family` name, and `port` of the server as reported by the operating system if listening on an IP socket (useful to find which port was assigned when getting an OS-assigned address): `{ port: 12346, family: 'IPv4', address: '127.0.0.1' }`.

For a server listening on a pipe or Unix domain socket, the name is returned as a string.

```js
const server = net.createServer((socket) => {
  socket.end('goodbye\n');
}).on('error', (err) => {
  // Handle errors here.
  throw err;
});
```

```
// Grab an arbitrary unused port.
server.listen(() => {
  console.log('opened server on', server.address());
});
```

`server.address()` returns `null` before the `'listening'` event has been emitted or after calling `server.close()`.

`server.close([callback])`

- `callback` {Function} Called when the server is closed.
- Returns: {net.Server}

Stops the server from accepting new connections and keeps existing connections. This function is asynchronous, the server is finally closed when all connections are ended and the server emits a `'close'` event. The optional `callback` will be called once the `'close'` event occurs. Unlike that event, it will be called with an `Error` as its only argument if the server was not open when it was closed.

`server.getConnections(callback)`

- `callback` {Function}
- Returns: {net.Server}

Asynchronously get the number of concurrent connections on the server. Works when sockets were sent to forks.

Callback should take two arguments `err` and `count`.

`server.listen()`

Start a server listening for connections. A `net.Server` can be a TCP or an IPC server depending on what it listens to.

Possible signatures:

- `server.listen(handle[, backlog][, callback])`
- `server.listen(options[, callback])`
- `server.listen(path[, backlog][, callback])` for IPC servers
- `server.listen([port[, host[, backlog]]][, callback])` for TCP servers

This function is asynchronous. When the server starts listening, the `'listening'` event will be emitted. The last parameter `callback` will be added as a listener for the `'listening'` event.

All `listen()` methods can take a `backlog` parameter to specify the maximum length of the queue of pending connections. The actual length will be determined by the OS through sysctl settings such as `tcp_max_syn_backlog` and `somaxconn` on Linux. The default value of this parameter is 511 (not 512).

All `net.Socket` are set to `SO_REUSEADDR` (see `socket(7)` for details).

The `server.listen()` method can be called again if and only if there was an error during the first `server.listen()` call or `server.close()` has been called. Otherwise, an `ERR_SERVER_ALREADY_LISTEN` error will be thrown.

One of the most common errors raised when listening is `EADDRINUSE`. This happens when another server is already listening on the requested `port`/`path`/`handle`. One way to handle this would be to retry after a certain amount of time:

```
server.on('error', (e) => {
  if (e.code === 'EADDRINUSE') {
    console.log('Address in use, retrying...');
    setTimeout(() => {
      server.close();
      server.listen(PORT, HOST);
    }, 1000);
  }
});
```

`server.listen(handle[, backlog][, callback])`

- `handle` {Object}
- `backlog` {number} Common parameter of `server.listen()` functions
- `callback` {Function}
- Returns: {net.Server}

Start a server listening for connections on a given `handle` that has already been bound to a port, a Unix domain socket, or a Windows named pipe.

The `handle` object can be either a server, a socket (anything with an underlying `_handle` member), or an object with an `fd` member that is a valid file descriptor.

Listening on a file descriptor is not supported on Windows.

`server.listen(options[, callback])`

- `options` {Object} Required. Supports the following properties:
  - `port` {number}
  - `host` {string}
  - `path` {string} Will be ignored if `port` is specified. See Identifying paths for IPC connections.
  - `backlog` {number} Common parameter of `server.listen()` functions.
  - `exclusive` {boolean} **Default:** `false`
  - `readableAll` {boolean} For IPC servers makes the pipe readable for all users. **Default:** `false`.
  - `writableAll` {boolean} For IPC servers makes the pipe writable for all users. **Default:** `false`.

  - ipv6Only {boolean} For TCP servers, setting `ipv6Only` to `true` will disable dual-stack support, i.e., binding to host `::` won't make `0.0.0.0` be bound. **Default:** `false`.
  - `signal` {AbortSignal} An AbortSignal that may be used to close a listening server.
- `callback` {Function} functions.
- Returns: {net.Server}

If `port` is specified, it behaves the same as `server.listen([port[, host[, backlog]]][, callback])`. Otherwise, if `path` is specified, it behaves the same as `server.listen(path[, backlog][, callback])`. If none of them is specified, an error will be thrown.

If `exclusive` is `false` (default), then cluster workers will use the same underlying handle, allowing connection handling duties to be shared. When `exclusive` is `true`, the handle is not shared, and attempted port sharing results in an error. An example which listens on an exclusive port is shown below.

```
server.listen({
  host: 'localhost',
  port: 80,
  exclusive: true
});
```

When `exclusive` is `true` and the underlying handle is shared, it is possible that several workers query a handle with different backlogs. In this case, the first `backlog` passed to the master process will be used.

Starting an IPC server as root may cause the server path to be inaccessible for unprivileged users. Using `readableAll` and `writableAll` will make the server accessible for all users.

If the `signal` option is enabled, calling `.abort()` on the corresponding `AbortController` is similar to calling `.close()` on the server:

```
const controller = new AbortController();
server.listen({
  host: 'localhost',
  port: 80,
  signal: controller.signal
});
// Later, when you want to close the server.
controller.abort();
```

`server.listen(path[, backlog][, callback])`

- `path` {string} Path the server should listen to. See Identifying paths for IPC connections.
- `backlog` {number} Common parameter of `server.listen()` functions.

- `callback` {Function}.
- Returns: {net.Server}

Start an IPC server listening for connections on the given `path`.

`server.listen([port[, host[, backlog]]][, callback])`

- `port` {number}
- `host` {string}
- `backlog` {number} Common parameter of `server.listen()` functions.
- `callback` {Function}.
- Returns: {net.Server}

Start a TCP server listening for connections on the given `port` and `host`.

If `port` is omitted or is 0, the operating system will assign an arbitrary unused port, which can be retrieved by using `server.address().port` after the `'listening'` event has been emitted.

If `host` is omitted, the server will accept connections on the unspecified IPv6 address (`::`) when IPv6 is available, or the unspecified IPv4 address (`0.0.0.0`) otherwise.

In most operating systems, listening to the unspecified IPv6 address (`::`) may cause the `net.Server` to also listen on the unspecified IPv4 address (`0.0.0.0`).

`server.listening`

- {boolean} Indicates whether or not the server is listening for connections.

`server.maxConnections`

- {integer}

Set this property to reject connections when the server's connection count gets high.

It is not recommended to use this option once a socket has been sent to a child with `child_process.fork()`.

`server.ref()`

- Returns: {net.Server}

Opposite of `unref()`, calling `ref()` on a previously `unref`ed server will *not* let the program exit if it's the only server left (the default behavior). If the server is `ref`ed calling `ref()` again will have no effect.

`server.unref()`

- Returns: {net.Server}

Calling `unref()` on a server will allow the program to exit if this is the only active server in the event system. If the server is already `unref`ed calling `unref()` again will have no effect.

Class: `net.Socket`

- Extends: {stream.Duplex}

This class is an abstraction of a TCP socket or a streaming IPC endpoint (uses named pipes on Windows, and Unix domain sockets otherwise). It is also an `EventEmitter`.

A `net.Socket` can be created by the user and used directly to interact with a server. For example, it is returned by `net.createConnection()`, so the user can use it to talk to the server.

It can also be created by Node.js and passed to the user when a connection is received. For example, it is passed to the listeners of a `'connection'` event emitted on a `net.Server`, so the user can use it to interact with the client.

`new net.Socket([options])`

- `options` {Object} Available options are:
  - `fd` {number} If specified, wrap around an existing socket with the given file descriptor, otherwise a new socket will be created.
  - `allowHalfOpen` {boolean} If set to `false`, then the socket will automatically end the writable side when the readable side ends. See `net.createServer()` and the `'end'` event for details. **Default:** `false`.
  - `readable` {boolean} Allow reads on the socket when an `fd` is passed, otherwise ignored. **Default:** `false`.
  - `writable` {boolean} Allow writes on the socket when an `fd` is passed, otherwise ignored. **Default:** `false`.
  - `signal` {AbortSignal} An Abort signal that may be used to destroy the socket.
- Returns: {net.Socket}

Creates a new socket object.

The newly created socket can be either a TCP socket or a streaming IPC endpoint, depending on what it `connect()` to.

Event: `'close'`

- `hadError` {boolean} `true` if the socket had a transmission error.

Emitted once the socket is fully closed. The argument `hadError` is a boolean which says if the socket was closed due to a transmission error.

Event: `'connect'`

Emitted when a socket connection is successfully established. See `net.createConnection()`.

Event: `'data'`

- {Buffer|string}

Emitted when data is received. The argument `data` will be a `Buffer` or `String`. Encoding of data is set by `socket.setEncoding()`.

The data will be lost if there is no listener when a `Socket` emits a `'data'` event.

Event: `'drain'`

Emitted when the write buffer becomes empty. Can be used to throttle uploads.

See also: the return values of `socket.write()`.

Event: `'end'`

Emitted when the other end of the socket signals the end of transmission, thus ending the readable side of the socket.

By default (`allowHalfOpen` is `false`) the socket will send an end of transmission packet back and destroy its file descriptor once it has written out its pending write queue. However, if `allowHalfOpen` is set to `true`, the socket will not automatically `end()` its writable side, allowing the user to write arbitrary amounts of data. The user must call `end()` explicitly to close the connection (i.e. sending a FIN packet back).

Event: `'error'`

- {Error}

Emitted when an error occurs. The `'close'` event will be called directly following this event.

Event: `'lookup'`

Emitted after resolving the host name but before connecting. Not applicable to Unix sockets.

- `err` {Error|null} The error object. See `dns.lookup()`.
- `address` {string} The IP address.
- `family` {string|null} The address type. See `dns.lookup()`.

- `host` {string} The host name.

Event: `'ready'`

Emitted when a socket is ready to be used.

Triggered immediately after `'connect'`.

Event: `'timeout'`

Emitted if the socket times out from inactivity. This is only to notify that the socket has been idle. The user must manually close the connection.

See also: `socket.setTimeout()`.

`socket.address()`

- Returns: {Object}

Returns the bound `address`, the address `family` name and `port` of the socket as reported by the operating system: `{ port: 12346, family: 'IPv4', address: '127.0.0.1' }`

`socket.bufferSize`

> Stability: 0 - Deprecated: Use `writable.writableLength` instead.

- {integer}

This property shows the number of characters buffered for writing. The buffer may contain strings whose length after encoding is not yet known. So this number is only an approximation of the number of bytes in the buffer.

`net.Socket` has the property that `socket.write()` always works. This is to help users get up and running quickly. The computer cannot always keep up with the amount of data that is written to a socket. The network connection simply might be too slow. Node.js will internally queue up the data written to a socket and send it out over the wire when it is possible.

The consequence of this internal buffering is that memory may grow. Users who experience large or growing `bufferSize` should attempt to "throttle" the data flows in their program with `socket.pause()` and `socket.resume()`.

`socket.bytesRead`

- {integer}

The amount of received bytes.

`socket.bytesWritten`

- {integer}

The amount of bytes sent.

`socket.connect()`

Initiate a connection on a given socket.

Possible signatures:

- `socket.connect(options[, connectListener])`
- `socket.connect(path[, connectListener])` for IPC connections.
- `socket.connect(port[, host][, connectListener])` for TCP connections.
- Returns: {net.Socket} The socket itself.

This function is asynchronous. When the connection is established, the `'connect'` event will be emitted. If there is a problem connecting, instead of a `'connect'` event, an `'error'` event will be emitted with the error passed to the `'error'` listener. The last parameter `connectListener`, if supplied, will be added as a listener for the `'connect'` event **once**.

This function should only be used for reconnecting a socket after `'close'` has been emitted or otherwise it may lead to undefined behavior.

`socket.connect(options[, connectListener])`

- `options` {Object}
- `connectListener` {Function} Common parameter of `socket.connect()` methods. Will be added as a listener for the `'connect'` event once.
- Returns: {net.Socket} The socket itself.

Initiate a connection on a given socket. Normally this method is not needed, the socket should be created and opened with `net.createConnection()`. Use this only when implementing a custom Socket.

For TCP connections, available `options` are:

- `port` {number} Required. Port the socket should connect to.
- `host` {string} Host the socket should connect to. **Default:** `'localhost'`.
- `localAddress` {string} Local address the socket should connect from.
- `localPort` {number} Local port the socket should connect from.
- `family` {number}: Version of IP stack. Must be `4`, `6`, or `0`. The value `0` indicates that both IPv4 and IPv6 addresses are allowed. **Default:** `0`.
- `hints` {number} Optional `dns.lookup()` hints.
- `lookup` {Function} Custom lookup function. **Default:** `dns.lookup()`.
- `noDelay` {boolean} If set to `true`, it disables the use of Nagle's algorithm immediately after the socket is established. **Default:** `false`.

- `keepAlive` {boolean} If set to `true`, it enables keep-alive functionality on the socket immediately after the connection is established, similarly on what is done in `socket.setKeepAlive([enable][, initialDelay])`. **Default:** `false`.
- `keepAliveInitialDelay` {number} If set to a positive number, it sets the initial delay before the first keepalive probe is sent on an idle socket.**Default:** `0`.

For IPC connections, available `options` are:

- `path` {string} Required. Path the client should connect to. See Identifying paths for IPC connections. If provided, the TCP-specific options above are ignored.

For both types, available `options` include:

- `onread` {Object} If specified, incoming data is stored in a single `buffer` and passed to the supplied `callback` when data arrives on the socket. This will cause the streaming functionality to not provide any data. The socket will emit events like `'error'`, `'end'`, and `'close'` as usual. Methods like `pause()` and `resume()` will also behave as expected.
  - `buffer` {Buffer|Uint8Array|Function} Either a reusable chunk of memory to use for storing incoming data or a function that returns such.
  - `callback` {Function} This function is called for every chunk of incoming data. Two arguments are passed to it: the number of bytes written to `buffer` and a reference to `buffer`. Return `false` from this function to implicitly `pause()` the socket. This function will be executed in the global context.

Following is an example of a client using the `onread` option:

```js
const net = require('net');
net.connect({
  port: 80,
  onread: {
    // Reuses a 4KiB Buffer for every read from the socket.
    buffer: Buffer.alloc(4 * 1024),
    callback: function(nread, buf) {
      // Received data is available in `buf` from 0 to `nread`.
      console.log(buf.toString('utf8', 0, nread));
    }
  }
});
```

`socket.connect(path[, connectListener])`

- `path` {string} Path the client should connect to. See Identifying paths for IPC connections.

- `connectListener` {Function} Common parameter of `socket.connect()` methods. Will be added as a listener for the `'connect'` event once.
- Returns: {net.Socket} The socket itself.

Initiate an IPC connection on the given socket.

Alias to `socket.connect(options[, connectListener])` called with `{ path: path }` as `options`.

`socket.connect(port[, host][, connectListener])`

- `port` {number} Port the client should connect to.
- `host` {string} Host the client should connect to.
- `connectListener` {Function} Common parameter of `socket.connect()` methods. Will be added as a listener for the `'connect'` event once.
- Returns: {net.Socket} The socket itself.

Initiate a TCP connection on the given socket.

Alias to `socket.connect(options[, connectListener])` called with `{port: port, host: host}` as `options`.

`socket.connecting`

- {boolean}

If `true`, `socket.connect(options[, connectListener])` was called and has not yet finished. It will stay `true` until the socket becomes connected, then it is set to `false` and the `'connect'` event is emitted. Note that the `socket.connect(options[, connectListener])` callback is a listener for the `'connect'` event.

`socket.destroy([error])`

- `error` {Object}
- Returns: {net.Socket}

Ensures that no more I/O activity happens on this socket. Destroys the stream and closes the connection.

See `writable.destroy()` for further details.

`socket.destroyed`

- {boolean} Indicates if the connection is destroyed or not. Once a connection is destroyed no further data can be transferred using it.

See `writable.destroyed` for further details.

`socket.end([data[, encoding]][, callback])`

- `data` {string|Buffer|Uint8Array}
- `encoding` {string} Only used when data is `string`. **Default:** `'utf8'`.
- `callback` {Function} Optional callback for when the socket is finished.
- Returns: {net.Socket} The socket itself.

Half-closes the socket. i.e., it sends a FIN packet. It is possible the server will still send some data.

See `writable.end()` for further details.

`socket.localAddress`

- {string}

The string representation of the local IP address the remote client is connecting on. For example, in a server listening on `'0.0.0.0'`, if a client connects on `'192.168.1.1'`, the value of `socket.localAddress` would be `'192.168.1.1'`.

`socket.localPort`

- {integer}

The numeric representation of the local port. For example, `80` or `21`.

`socket.pause()`

- Returns: {net.Socket} The socket itself.

Pauses the reading of data. That is, `'data'` events will not be emitted. Useful to throttle back an upload.

`socket.pending`

- {boolean}

This is `true` if the socket is not connected yet, either because `.connect()` has not yet been called or because it is still in the process of connecting (see `socket.connecting`).

`socket.ref()`

- Returns: {net.Socket} The socket itself.

Opposite of `unref()`, calling `ref()` on a previously `unref`ed socket will *not* let the program exit if it's the only socket left (the default behavior). If the socket is `ref`ed calling `ref` again will have no effect.

`socket.remoteAddress`

- {string}

The string representation of the remote IP address. For example, `'74.125.127.100'` or `'2001:4860:a005::68'`. Value may be `undefined` if the socket is destroyed (for example, if the client disconnected).

`socket.remoteFamily`

- {string}

The string representation of the remote IP family. `'IPv4'` or `'IPv6'`.

`socket.remotePort`

- {integer}

The numeric representation of the remote port. For example, `80` or `21`.

`socket.resume()`

- Returns: {net.Socket} The socket itself.

Resumes reading after a call to `socket.pause()`.

`socket.setEncoding([encoding])`

- `encoding` {string}
- Returns: {net.Socket} The socket itself.

Set the encoding for the socket as a Readable Stream. See `readable.setEncoding()` for more information.

`socket.setKeepAlive([enable][, initialDelay])`

- `enable` {boolean} **Default:** `false`
- `initialDelay` {number} **Default:** `0`
- Returns: {net.Socket} The socket itself.

Enable/disable keep-alive functionality, and optionally set the initial delay before the first keepalive probe is sent on an idle socket.

Set `initialDelay` (in milliseconds) to set the delay between the last data packet received and the first keepalive probe. Setting `0` for `initialDelay` will leave the value unchanged from the default (or previous) setting.

Enabling the keep-alive functionality will set the following socket options:

- `SO_KEEPALIVE=1`
- `TCP_KEEPIDLE=initialDelay`
- `TCP_KEEPCNT=10`

- `TCP_KEEPINTVL=1`

`socket.setNoDelay([noDelay])`

- `noDelay` {boolean} **Default:** `true`
- Returns: {net.Socket} The socket itself.

Enable/disable the use of Nagle's algorithm.

When a TCP connection is created, it will have Nagle's algorithm enabled.

Nagle's algorithm delays data before it is sent via the network. It attempts to optimize throughput at the expense of latency.

Passing `true` for `noDelay` or not passing an argument will disable Nagle's algorithm for the socket. Passing `false` for `noDelay` will enable Nagle's algorithm.

`socket.setTimeout(timeout[, callback])`

- `timeout` {number}
- `callback` {Function}
- Returns: {net.Socket} The socket itself.

Sets the socket to timeout after `timeout` milliseconds of inactivity on the socket. By default `net.Socket` do not have a timeout.

When an idle timeout is triggered the socket will receive a `'timeout'` event but the connection will not be severed. The user must manually call `socket.end()` or `socket.destroy()` to end the connection.

```
socket.setTimeout(3000);
socket.on('timeout', () => {
  console.log('socket timeout');
  socket.end();
});
```

If `timeout` is 0, then the existing idle timeout is disabled.

The optional `callback` parameter will be added as a one-time listener for the `'timeout'` event.

`socket.timeout`

- {number|undefined}

The socket timeout in milliseconds as set by `socket.setTimeout()`. It is `undefined` if a timeout has not been set.

`socket.unref()`

- Returns: {net.Socket} The socket itself.

Calling `unref()` on a socket will allow the program to exit if this is the only active socket in the event system. If the socket is already `unref`ed calling `unref()` again will have no effect.

`socket.write(data[, encoding][, callback])`

- `data` {string|Buffer|Uint8Array}
- `encoding` {string} Only used when data is `string`. **Default:** `utf8`.
- `callback` {Function}
- Returns: {boolean}

Sends data on the socket. The second parameter specifies the encoding in the case of a string. It defaults to UTF8 encoding.

Returns `true` if the entire data was flushed successfully to the kernel buffer. Returns `false` if all or part of the data was queued in user memory. `'drain'` will be emitted when the buffer is again free.

The optional `callback` parameter will be executed when the data is finally written out, which may not be immediately.

See `Writable` stream `write()` method for more information.

`socket.readyState`

- {string}

This property represents the state of the connection as a string.

- If the stream is connecting `socket.readyState` is `opening`.
- If the stream is readable and writable, it is `open`.
- If the stream is readable and not writable, it is `readOnly`.
- If the stream is not readable and writable, it is `writeOnly`.

`net.connect()`

Aliases to `net.createConnection()`.

Possible signatures:

- `net.connect(options[, connectListener])`
- `net.connect(path[, connectListener])` for IPC connections.
- `net.connect(port[, host][, connectListener])` for TCP connections.

`net.connect(options[, connectListener])`

- `options` {Object}
- `connectListener` {Function}
- Returns: {net.Socket}

Alias to `net.createConnection(options[, connectListener])`.

`net.connect(path[, connectListener])`

- `path` {string}
- `connectListener` {Function}
- Returns: {net.Socket}

Alias to `net.createConnection(path[, connectListener])`.

`net.connect(port[, host][, connectListener])`

- `port` {number}
- `host` {string}
- `connectListener` {Function}
- Returns: {net.Socket}

Alias to `net.createConnection(port[, host][, connectListener])`.

`net.createConnection()`

A factory function, which creates a new `net.Socket`, immediately initiates connection with `socket.connect()`, then returns the `net.Socket` that starts the connection.

When the connection is established, a `'connect'` event will be emitted on the returned socket. The last parameter `connectListener`, if supplied, will be added as a listener for the `'connect'` event **once**.

Possible signatures:

- `net.createConnection(options[, connectListener])`
- `net.createConnection(path[, connectListener])` for IPC connections.
- `net.createConnection(port[, host][, connectListener])` for TCP connections.

The `net.connect()` function is an alias to this function.

`net.createConnection(options[, connectListener])`

- `options` {Object} Required. Will be passed to both the `new net.Socket([options])` call and the `socket.connect(options[, connectListener])` method.
- `connectListener` {Function} Common parameter of the `net.createConnection()` functions. If supplied, will be added as a listener for the `'connect'` event on the returned socket once.
- Returns: {net.Socket} The newly created socket used to start the connection.

For available options, see `new net.Socket([options])` and `socket.connect(options[, connectListener])`.

Additional options:

- `timeout` {number} If set, will be used to call `socket.setTimeout(timeout)` after the socket is created, but before it starts the connection.

Following is an example of a client of the echo server described in the `net.createServer()` section:

```
const net = require('net');
const client = net.createConnection({ port: 8124 }, () => {
  // 'connect' listener.
  console.log('connected to server!');
  client.write('world!\r\n');
});
client.on('data', (data) => {
  console.log(data.toString());
  client.end();
});
client.on('end', () => {
  console.log('disconnected from server');
});
```

To connect on the socket `/tmp/echo.sock`:

```
const client = net.createConnection({ path: '/tmp/echo.sock' });
```

`net.createConnection(path[, connectListener])`

- `path` {string} Path the socket should connect to. Will be passed to `socket.connect(path[, connectListener])`. See Identifying paths for IPC connections.
- `connectListener` {Function} Common parameter of the `net.createConnection()` functions, an "once" listener for the `'connect'` event on the initiating socket. Will be passed to `socket.connect(path[, connectListener])`.
- Returns: {net.Socket} The newly created socket used to start the connection.

Initiates an IPC connection.

This function creates a new `net.Socket` with all options set to default, immediately initiates connection with `socket.connect(path[, connectListener])`, then returns the `net.Socket` that starts the connection.

`net.createConnection(port[, host][, connectListener])`

- `port` {number} Port the socket should connect to. Will be passed to `socket.connect(port[, host][, connectListener])`.

- `host` {string} Host the socket should connect to. Will be passed to `socket.connect(port[, host][, connectListener])`. **Default:** `'localhost'`.
- `connectListener` {Function} Common parameter of the `net.createConnection()` functions, an “once” listener for the `'connect'` event on the initiating socket. Will be passed to `socket.connect(port[, host][, connectListener])`.
- Returns: {net.Socket} The newly created socket used to start the connection.

Initiates a TCP connection.

This function creates a new `net.Socket` with all options set to default, immediately initiates connection with `socket.connect(port[, host][, connectListener])`, then returns the `net.Socket` that starts the connection.

`net.createServer([options][, connectionListener])`

- `options` {Object}
  - `allowHalfOpen` {boolean} If set to `false`, then the socket will automatically end the writable side when the readable side ends. **Default:** `false`.
  - `pauseOnConnect` {boolean} Indicates whether the socket should be paused on incoming connections. **Default:** `false`.
  - `noDelay` {boolean} If set to `true`, it disables the use of Nagle’s algorithm immediately after a new incoming connection is received. **Default:** `false`.
  - `keepAlive` {boolean} If set to `true`, it enables keep-alive functionality on the socket immediately after a new incoming connection is received, similarly on what is done in `socket.setKeepAlive([enable][, initialDelay])`. **Default:** `false`.
  - `keepAliveInitialDelay` {number} If set to a positive number, it sets the initial delay before the first keepalive probe is sent on an idle socket.**Default:** `0`.
- `connectionListener` {Function} Automatically set as a listener for the `'connection'` event.
- Returns: {net.Server}

Creates a new TCP or IPC server.

If `allowHalfOpen` is set to `true`, when the other end of the socket signals the end of transmission, the server will only send back the end of transmission when `socket.end()` is explicitly called. For example, in the context of TCP, when a FIN packed is received, a FIN packed is sent back only when `socket.end()` is explicitly called. Until then the connection is half-closed (non-readable but still writable). See `'end'` event and RFC 1122 (section 4.2.2.13) for more information.

If `pauseOnConnect` is set to `true`, then the socket associated with each incoming connection will be paused, and no data will be read from its handle. This allows connections to be passed between processes without any data being read by the original process. To begin reading data from a paused socket, call `socket.resume()`.

The server can be a TCP server or an IPC server, depending on what it `listen()` to.

Here is an example of a TCP echo server which listens for connections on port 8124:

```
const net = require('net');
const server = net.createServer((c) => {
  // 'connection' listener.
  console.log('client connected');
  c.on('end', () => {
    console.log('client disconnected');
  });
  c.write('hello\r\n');
  c.pipe(c);
});
server.on('error', (err) => {
  throw err;
});
server.listen(8124, () => {
  console.log('server bound');
});
```

Test this by using `telnet`:

```
$ telnet localhost 8124
```

To listen on the socket `/tmp/echo.sock`:

```
server.listen('/tmp/echo.sock', () => {
  console.log('server bound');
});
```

Use `nc` to connect to a Unix domain socket server:

```
$ nc -U /tmp/echo.sock
```

`net.isIP(input)`

- `input` {string}
- Returns: {integer}

Returns `6` if `input` is an IPv6 address. Returns `4` if `input` is an IPv4 address in dot-decimal notation with no leading zeroes. Otherwise, returns `0`.

```
net.isIP('::1'); // returns 6
net.isIP('127.0.0.1'); // returns 4
net.isIP('127.000.000.001'); // returns 0
net.isIP('127.0.0.1/24'); // returns 0
net.isIP('fhqwhgads'); // returns 0
```

`net.isIPv4(input)`

- `input` {string}
- Returns: {boolean}

Returns `true` if `input` is an IPv4 address in dot-decimal notation with no leading zeroes. Otherwise, returns `false`.

```
net.isIPv4('127.0.0.1'); // returns true
net.isIPv4('127.000.000.001'); // returns false
net.isIPv4('127.0.0.1/24'); // returns false
net.isIPv4('fhqwhgads'); // returns false
```

`net.isIPv6(input)`

- `input` {string}
- Returns: {boolean}

Returns `true` if `input` is an IPv6 address. Otherwise, returns `false`.

```
net.isIPv6('::1'); // returns true
net.isIPv6('fhqwhgads'); // returns false
```