UDP/datagram sockets

Stability: 2 - Stable

Source Code: lib/dgram.js

The `node:dgram` module provides an implementation of UDP datagram sockets.

```mjs
import dgram from 'node:dgram';

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

server.on('error', (err) => {
  console.error(`server error:\n${err.stack}`);
  server.close();
});

server.on('message', (msg, rinfo) => {
  console.log(`server got: ${msg} from ${rinfo.address}:${rinfo.port}`);
});

server.on('listening', () => {
  const address = server.address();
  console.log(`server listening ${address.address}:${address.port}`);
});

server.bind(41234);
// Prints: server listening 0.0.0.0:41234
```

```cjs
const dgram = require('node:dgram');
const server = dgram.createSocket('udp4');

server.on('error', (err) => {
  console.error(`server error:\n${err.stack}`);
  server.close();
});

server.on('message', (msg, rinfo) => {
  console.log(`server got: ${msg} from ${rinfo.address}:${rinfo.port}`);
});

server.on('listening', () => {
  const address = server.address();
  console.log(`server listening ${address.address}:${address.port}`);
});
```

```
server.bind(41234);
// Prints: server listening 0.0.0.0:41234
```

COPY

Class: dgram.Socket

- Extends: <EventEmitter>

Encapsulates the datagram functionality.

New instances of `dgram.Socket` are created using `dgram.createSocket()`. The `new` keyword is not to be used to create `dgram.Socket` instances.

Event: 'close'

The `'close'` event is emitted after a socket is closed with `close()`. Once triggered, no new `'message'` events will be emitted on this socket.

Event: 'connect'

The `'connect'` event is emitted after a socket is associated to a remote address as a result of a successful `connect()` call.

Event: 'error'

- `exception` <Error>

The `'error'` event is emitted whenever any error occurs. The event handler function is passed a single `Error` object.

Event: 'listening'

The `'listening'` event is emitted once the `dgram.Socket` is addressable and can receive data. This happens either explicitly with `socket.bind()` or implicitly the first time data is sent using `socket.send()`. Until the `dgram.Socket` is listening, the underlying system resources do not exist and calls such as `socket.address()` and `socket.setTTL()` will fail.

Event: 'message'

The `'message'` event is emitted when a new datagram is available on a socket. The event handler function is passed two arguments: `msg` and `rinfo`.

- `msg` <Buffer> The message.
- `rinfo` <Object> Remote address information.
  - `address` <string> The sender address.
  - `family` <string> The address family (`'IPv4'` or `'IPv6'`).
  - `port` <number> The sender port.
  - `size` <number> The message size.

If the source address of the incoming packet is an IPv6 link-local address, the interface name is added to the `address`. For example, a packet received on the `en0` interface might have the address field set to `'fe80::2618:1234:ab11:3b9c%en0'`, where `'%en0'` is the interface name as a zone ID suffix.

socket.addMembership(multicastAddress[, multicastInterface])

- `multicastAddress` <string>
- `multicastInterface` <string>

Tells the kernel to join a multicast group at the given `multicastAddress` and `multicastInterface` using the `IP_ADD_MEMBERSHIP` socket option. If the `multicastInterface` argument is not specified, the operating system will choose one interface and will add membership to it. To add membership to every available interface, call `addMembership` multiple times, once per interface.

When called on an unbound socket, this method will implicitly bind to a random port, listening on all interfaces.

When sharing a UDP socket across multiple `cluster` workers, the `socket.addMembership()` function must be called only once or an `EADDRINUSE` error will occur:

```
import cluster from 'node:cluster';
import dgram from 'node:dgram';

if (cluster.isPrimary) {
  cluster.fork(); // Works ok.
  cluster.fork(); // Fails with EADDRINUSE.
} else {
  const s = dgram.createSocket('udp4');
  s.bind(1234, () => {
    s.addMembership('224.0.0.114');
  });
}
```

```
const cluster = require('node:cluster');
const dgram = require('node:dgram');

if (cluster.isPrimary) {
  cluster.fork(); // Works ok.
  cluster.fork(); // Fails with EADDRINUSE.
} else {
  const s = dgram.createSocket('udp4');
  s.bind(1234, () => {
    s.addMembership('224.0.0.114');
  });
}
```

COPY

socket.addSourceSpecificMembership(sourceAddress, groupAddress[, multicastInterface])

- `sourceAddress` `<string>`
- `groupAddress` `<string>`
- `multicastInterface` `<string>`

Tells the kernel to join a source-specific multicast channel at the given `sourceAddress` and `groupAddress`, using the `multicastInterface` with the `IP_ADD_SOURCE_MEMBERSHIP` socket option. If the `multicastInterface` argument is not specified, the operating system will choose one interface and will add membership to it. To add membership to every available interface, call `socket.addSourceSpecificMembership()` multiple times, once per interface.

When called on an unbound socket, this method will implicitly bind to a random port, listening on all interfaces.

socket.address()

- Returns: <Object>

Returns an object containing the address information for a socket. For UDP sockets, this object will contain `address`, `family`, and `port` properties.

This method throws `EBADF` if called on an unbound socket.

socket.bind([port][, address][, callback])

- `port` <integer>
- `address` <string>
- `callback` <Function> with no parameters. Called when binding is complete.

For UDP sockets, causes the `dgram.Socket` to listen for datagram messages on a named `port` and optional `address`. If `port` is not specified or is `0`, the operating system will attempt to bind to a random port. If `address` is not specified, the operating system will attempt to listen on all addresses. Once binding is complete, a `'listening'` event is emitted and the optional `callback` function is called.

Specifying both a `'listening'` event listener and passing a `callback` to the `socket.bind()` method is not harmful but not very useful.

A bound datagram socket keeps the Node.js process running to receive datagram messages.

If binding fails, an `'error'` event is generated. In rare case (e.g. attempting to bind with a closed socket), an `Error` may be thrown.

Example of a UDP server listening on port 41234:

```mjs
import dgram from 'node:dgram';

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

server.on('error', (err) => {
  console.error(`server error:\n${err.stack}`);
  server.close();
});

server.on('message', (msg, rinfo) => {
  console.log(`server got: ${msg} from ${rinfo.address}:${rinfo.port}`);
});

server.on('listening', () => {
  const address = server.address();
  console.log(`server listening ${address.address}:${address.port}`);
});

server.bind(41234);
// Prints: server listening 0.0.0.0:41234
```

```cjs
const dgram = require('node:dgram');
const server = dgram.createSocket('udp4');

server.on('error', (err) => {
  console.error(`server error:\n${err.stack}`);
  server.close();
});
```

```
server.on('message', (msg, rinfo) => {
  console.log(`server got: ${msg} from ${rinfo.address}:${rinfo.port}`);
});

server.on('listening', () => {
  const address = server.address();
  console.log(`server listening ${address.address}:${address.port}`);
});

server.bind(41234);
// Prints: server listening 0.0.0.0:41234
```

COPY

socket.bind(options[, callback])

- `options` <Object> Required. Supports the following properties:
  - `port` <integer>
  - `address` <string>
  - `exclusive` <boolean>
  - `fd` <integer>
- `callback` <Function>

For UDP sockets, causes the `dgram.Socket` to listen for datagram messages on a named `port` and optional `address` that are passed as properties of an `options` object passed as the first argument. If `port` is not specified or is `0`, the operating system will attempt to bind to a random port. If `address` is not specified, the operating system will attempt to listen on all addresses. Once binding is complete, a `'listening'` event is emitted and the optional `callback` function is called.

The `options` object may contain a `fd` property. When a `fd` greater than `0` is set, it will wrap around an existing socket with the given file descriptor. In this case, the properties of `port` and `address` will be ignored.

Specifying both a `'listening'` event listener and passing a `callback` to the `socket.bind()` method is not harmful but not very useful.

The `options` object may contain an additional `exclusive` property that is used when using `dgram.Socket` objects with the `cluster` module. When `exclusive` is set to `false` (the default), cluster workers will use the same underlying socket handle allowing connection handling duties to be shared. When `exclusive` is `true`, however, the handle is not shared and attempted port sharing results in an error.

A bound datagram socket keeps the Node.js process running to receive datagram messages.

If binding fails, an `'error'` event is generated. In rare case (e.g. attempting to bind with a closed socket), an `Error` may be thrown.

An example socket listening on an exclusive port is shown below.

```
socket.bind({
  address: 'localhost',
  port: 8000,
  exclusive: true,
});
```

COPY

socket.close([callback])

- `callback` <Function> Called when the socket has been closed.

Close the underlying socket and stop listening for data on it. If a callback is provided, it is added as a listener for the `'close'` event.

socket[Symbol.asyncDispose]()

Stability: 1 - Experimental

Calls `socket.close()` and returns a promise that fulfills when the socket has closed.

socket.connect(port[, address][, callback])

- `port` `<integer>`
- `address` `<string>`
- `callback` `<Function>` Called when the connection is completed or on error.

Associates the `dgram.Socket` to a remote address and port. Every message sent by this handle is automatically sent to that destination. Also, the socket will only receive messages from that remote peer. Trying to call `connect()` on an already connected socket will result in an `ERR_SOCKET_DGRAM_IS_CONNECTED` exception. If `address` is not provided, `'127.0.0.1'` (for `udp4` sockets) or `'::1'` (for `udp6` sockets) will be used by default. Once the connection is complete, a `'connect'` event is emitted and the optional `callback` function is called. In case of failure, the `callback` is called or, failing this, an `'error'` event is emitted.

socket.disconnect()

A synchronous function that disassociates a connected `dgram.Socket` from its remote address. Trying to call `disconnect()` on an unbound or already disconnected socket will result in an `ERR_SOCKET_DGRAM_NOT_CONNECTED` exception.

socket.dropMembership(multicastAddress[, multicastInterface])

- `multicastAddress` `<string>`
- `multicastInterface` `<string>`

Instructs the kernel to leave a multicast group at `multicastAddress` using the `IP_DROP_MEMBERSHIP` socket option. This method is automatically called by the kernel when the socket is closed or the process terminates, so most apps will never have reason to call this.

If `multicastInterface` is not specified, the operating system will attempt to drop membership on all valid interfaces.

socket.dropSourceSpecificMembership(sourceAddress, groupAddress[, multicastInterface])

- `sourceAddress` `<string>`
- `groupAddress` `<string>`
- `multicastInterface` `<string>`

Instructs the kernel to leave a source-specific multicast channel at the given `sourceAddress` and `groupAddress` using the `IP_DROP_SOURCE_MEMBERSHIP` socket option. This method is automatically called by the kernel when the socket is closed or the process terminates, so most apps will never have reason to call this.

If `multicastInterface` is not specified, the operating system will attempt to drop membership on all valid interfaces.

socket.getRecvBufferSize()

- Returns: `<number>` the `SO_RCVBUF` socket receive buffer size in bytes.

This method throws `ERR_SOCKET_BUFFER_SIZE` if called on an unbound socket.

socket.getSendBufferSize()

- Returns: `<number>` the `SO_SNDBUF` socket send buffer size in bytes.

This method throws `ERR_SOCKET_BUFFER_SIZE` if called on an unbound socket.

socket.getSendQueueSize()

- Returns: `<number>` Number of bytes queued for sending.

socket.getSendQueueCount()

- Returns: `<number>` Number of send requests currently in the queue awaiting to be processed.

socket.ref()

- Returns: `<dgram.Socket>`

By default, binding a socket will cause it to block the Node.js process from exiting as long as the socket is open. The `socket.unref()` method can be used to exclude the socket from the reference counting that keeps the Node.js process active. The `socket.ref()` method adds the socket back to the reference counting and restores the default behavior.

Calling `socket.ref()` multiples times will have no additional effect.

The `socket.ref()` method returns a reference to the socket so calls can be chained.

socket.remoteAddress()

- Returns: `<Object>`

Returns an object containing the `address`, `family`, and `port` of the remote endpoint. This method throws an `ERR_SOCKET_DGRAM_NOT_CONNECTED` exception if the socket is not connected.

socket.send(msg[, offset, length][, port][, address][, callback])

- `msg` `<Buffer>` | `<TypedArray>` | `<DataView>` | `<string>` | `<Array>` Message to be sent.
- `offset` `<integer>` Offset in the buffer where the message starts.
- `length` `<integer>` Number of bytes in the message.
- `port` `<integer>` Destination port.
- `address` `<string>` Destination host name or IP address.
- `callback` `<Function>` Called when the message has been sent.

Broadcasts a datagram on the socket. For connectionless sockets, the destination `port` and `address` must be specified. Connected sockets, on the other hand, will use their associated remote endpoint, so the `port` and `address` arguments must not be set.

The `msg` argument contains the message to be sent. Depending on its type, different behavior can apply. If `msg` is a `Buffer`, any `TypedArray` or a `DataView`, the `offset` and `length` specify the offset within the `Buffer` where the message begins and the number of bytes in the message, respectively. If `msg` is a `String`, then it is automatically converted to a `Buffer` with `'utf8'` encoding. With messages that contain multi-byte characters, `offset` and `length` will be calculated with respect to byte length and not the character position. If `msg` is an array, `offset` and `length` must not be specified.

The `address` argument is a string. If the value of `address` is a host name, DNS will be used to resolve the address of the host. If `address` is not provided or otherwise nullish, `'127.0.0.1'` (for `udp4` sockets) or `'::1'` (for `udp6` sockets) will be used by default.

If the socket has not been previously bound with a call to `bind`, the socket is assigned a random port number and is bound to the "all interfaces" address (`'0.0.0.0'` for `udp4` sockets, `'::0'` for `udp6` sockets.)

An optional `callback` function may be specified to as a way of reporting DNS errors or for determining when it is safe to reuse the `buf` object. DNS lookups delay the time to send for at least one tick of the Node.js event loop.

The only way to know for sure that the datagram has been sent is by using a `callback`. If an error occurs and a `callback` is given, the error will be passed as the first argument to the `callback`. If a `callback` is not given, the error is emitted as an `'error'` event on the `socket` object.

Offset and length are optional but both *must* be set if either are used. They are supported only when the first argument is a `Buffer`, a `TypedArray`, or a `DataView`.

This method throws `ERR_SOCKET_BAD_PORT` if called on an unbound socket.

Example of sending a UDP packet to a port on `localhost`;

```mjs
import dgram from 'node:dgram';
import { Buffer } from 'node:buffer';

const message = Buffer.from('Some bytes');
const client = dgram.createSocket('udp4');
client.send(message, 41234, 'localhost', (err) => {
  client.close();
});
```

```cjs
const dgram = require('node:dgram');
const { Buffer } = require('node:buffer');

const message = Buffer.from('Some bytes');
const client = dgram.createSocket('udp4');
client.send(message, 41234, 'localhost', (err) => {
  client.close();
});
```

COPY

Example of sending a UDP packet composed of multiple buffers to a port on `127.0.0.1`;

```mjs
import dgram from 'node:dgram';
import { Buffer } from 'node:buffer';

const buf1 = Buffer.from('Some ');
const buf2 = Buffer.from('bytes');
const client = dgram.createSocket('udp4');
client.send([buf1, buf2], 41234, (err) => {
  client.close();
});
```

```cjs
const dgram = require('node:dgram');
const { Buffer } = require('node:buffer');

const buf1 = Buffer.from('Some ');
const buf2 = Buffer.from('bytes');
const client = dgram.createSocket('udp4');
client.send([buf1, buf2], 41234, (err) => {
  client.close();
});
```

COPY

Sending multiple buffers might be faster or slower depending on the application and operating system. Run benchmarks to determine the optimal strategy on a case-by-case basis. Generally speaking, however, sending multiple buffers is faster.

Example of sending a UDP packet using a socket connected to a port on `localhost`:

```mjs
import dgram from 'node:dgram';
import { Buffer } from 'node:buffer';

const message = Buffer.from('Some bytes');
const client = dgram.createSocket('udp4');
client.connect(41234, 'localhost', (err) => {
  client.send(message, (err) => {
    client.close();
  });
});
```

```cjs
const dgram = require('node:dgram');
const { Buffer } = require('node:buffer');

const message = Buffer.from('Some bytes');
const client = dgram.createSocket('udp4');
client.connect(41234, 'localhost', (err) => {
  client.send(message, (err) => {
    client.close();
  });
});
```

COPY

Note about UDP datagram size

The maximum size of an IPv4/v6 datagram depends on the `MTU` (Maximum Transmission Unit) and on the `Payload Length` field size.

- The `Payload Length` field is 16 bits wide, which means that a normal payload cannot exceed 64K octets including the internet header and data (65,507 bytes = 65,535 − 8 bytes UDP header − 20 bytes IP header); this is generally true for loopback interfaces, but such long datagram messages are impractical for most hosts and networks.
- The `MTU` is the largest size a given link layer technology can support for datagram messages. For any link, IPv4 mandates a minimum `MTU` of 68 octets, while the recommended `MTU` for IPv4 is 576 (typically recommended as the `MTU` for dial-up type applications), whether they arrive whole or in fragments.

  For IPv6, the minimum `MTU` is 1280 octets. However, the mandatory minimum fragment reassembly buffer size is 1500 octets. The value of 68 octets is very small, since most current link layer technologies, like Ethernet, have a minimum `MTU` of 1500.

It is impossible to know in advance the MTU of each link through which a packet might travel. Sending a datagram greater than the receiver `MTU` will not work because the packet will get silently dropped without informing the source that the data did not reach its intended recipient.

socket.setBroadcast(flag)

- `flag` <boolean>

Sets or clears the `SO_BROADCAST` socket option. When set to `true`, UDP packets may be sent to a local interface's broadcast address.

This method throws `EBADF` if called on an unbound socket.

socket.setMulticastInterface(multicastInterface)

- `multicastInterface` `<string>`

All references to scope in this section are referring to IPv6 Zone Indexes, which are defined by RFC 4007. In string form, an IP with a scope index is written as `'IP%scope'` where scope is an interface name or interface number.

Sets the default outgoing multicast interface of the socket to a chosen interface or back to system interface selection. The `multicastInterface` must be a valid string representation of an IP from the socket's family.

For IPv4 sockets, this should be the IP configured for the desired physical interface. All packets sent to multicast on the socket will be sent on the interface determined by the most recent successful use of this call.

For IPv6 sockets, `multicastInterface` should include a scope to indicate the interface as in the examples that follow. In IPv6, individual `send` calls can also use explicit scope in addresses, so only packets sent to a multicast address without specifying an explicit scope are affected by the most recent successful use of this call.

This method throws `EBADF` if called on an unbound socket.

Example: IPv6 outgoing multicast interface

On most systems, where scope format uses the interface name:

```
const socket = dgram.createSocket('udp6');

socket.bind(1234, () => {
  socket.setMulticastInterface('::%eth1');
});
```

COPY

On Windows, where scope format uses an interface number:

```
const socket = dgram.createSocket('udp6');

socket.bind(1234, () => {
  socket.setMulticastInterface('::%2');
});
```

COPY

Example: IPv4 outgoing multicast interface

All systems use an IP of the host on the desired physical interface:

```
const socket = dgram.createSocket('udp4');

socket.bind(1234, () => {
  socket.setMulticastInterface('10.0.0.2');
});
```

COPY

Call results

A call on a socket that is not ready to send or no longer open may throw a *Not running* `Error`.

If `multicastInterface` can not be parsed into an IP then an *EINVAL* `System Error` is thrown.

On IPv4, if `multicastInterface` is a valid address but does not match any interface, or if the address does not match the family then a `System Error` such as `EADDRNOTAVAIL` or `EPROTONOSUP` is thrown.

On IPv6, most errors with specifying or omitting scope will result in the socket continuing to use (or returning to) the system's default interface selection.

A socket's address family's ANY address (IPv4 `'0.0.0.0'` or IPv6 `'::'`) can be used to return control of the sockets default outgoing interface to the system for future multicast packets.

socket.setMulticastLoopback(flag)

- `flag` `<boolean>`

Sets or clears the `IP_MULTICAST_LOOP` socket option. When set to `true`, multicast packets will also be received on the local interface.

This method throws `EBADF` if called on an unbound socket.

socket.setMulticastTTL(ttl)

- `ttl` `<integer>`

Sets the `IP_MULTICAST_TTL` socket option. While TTL generally stands for "Time to Live", in this context it specifies the number of IP hops that a packet is allowed to travel through, specifically for multicast traffic. Each router or gateway that forwards a packet decrements the TTL. If the TTL is decremented to 0 by a router, it will not be forwarded.

The `ttl` argument may be between 0 and 255. The default on most systems is `1`.

This method throws `EBADF` if called on an unbound socket.

socket.setRecvBufferSize(size)

- `size` `<integer>`

Sets the `SO_RCVBUF` socket option. Sets the maximum socket receive buffer in bytes.

This method throws `ERR_SOCKET_BUFFER_SIZE` if called on an unbound socket.

socket.setSendBufferSize(size)

- `size` `<integer>`

Sets the `SO_SNDBUF` socket option. Sets the maximum socket send buffer in bytes.

This method throws `ERR_SOCKET_BUFFER_SIZE` if called on an unbound socket.

socket.setTTL(ttl)

- `ttl` `<integer>`

Sets the `IP_TTL` socket option. While TTL generally stands for "Time to Live", in this context it specifies the number of IP hops that a packet is allowed to travel through. Each router or gateway that forwards a packet decrements the TTL. If the TTL is decremented to 0 by a router, it will not be forwarded. Changing TTL values is typically done for network probes or when multicasting.

The `ttl` argument may be between 1 and 255. The default on most systems is 64.

This method throws `EBADF` if called on an unbound socket.

socket.unref()

- Returns: `<dgram.Socket>`

By default, binding a socket will cause it to block the Node.js process from exiting as long as the socket is open. The `socket.unref()` method can be used to exclude the socket from the reference counting that keeps the Node.js process active, allowing the process to exit even if the socket is still listening.

Calling `socket.unref()` multiple times will have no additional effect.

The `socket.unref()` method returns a reference to the socket so calls can be chained.

node:dgram module functions

dgram.createSocket(options[, callback])

- `options` <Object> Available options are:
  - `type` <string> The family of socket. Must be either `'udp4'` or `'udp6'`. Required.
  - `reuseAddr` <boolean> When `true` `socket.bind()` will reuse the address, even if another process has already bound a socket on it. **Default:** `false`.
  - `ipv6Only` <boolean> Setting `ipv6Only` to `true` will disable dual-stack support, i.e., binding to address `::` won't make `0.0.0.0` be bound. **Default:** `false`.
  - `recvBufferSize` <number> Sets the `SO_RCVBUF` socket value.
  - `sendBufferSize` <number> Sets the `SO_SNDBUF` socket value.
  - `lookup` <Function> Custom lookup function. **Default:** `dns.lookup()`.
  - `signal` <AbortSignal> An AbortSignal that may be used to close a socket.
- `callback` <Function> Attached as a listener for `'message'` events. Optional.
- Returns: <dgram.Socket>

Creates a `dgram.Socket` object. Once the socket is created, calling `socket.bind()` will instruct the socket to begin listening for datagram messages. When `address` and `port` are not passed to `socket.bind()` the method will bind the socket to the "all interfaces" address on a random port (it does the right thing for both `udp4` and `udp6` sockets). The bound address and port can be retrieved using `socket.address().address` and `socket.address().port`.

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

```
const controller = new AbortController();
const { signal } = controller;
const server = dgram.createSocket({ type: 'udp4', signal });
server.on('message', (msg, rinfo) => {
  console.log(`server got: ${msg} from ${rinfo.address}:${rinfo.port}`);
});
// Later, when you want to close the server.
controller.abort();
```

COPY

dgram.createSocket(type[, callback])

- `type` <string> Either `'udp4'` or `'udp6'`.
- `callback` <Function> Attached as a listener to `'message'` events.
- Returns: <dgram.Socket>

Creates a `dgram.Socket` object of the specified `type`.

Once the socket is created, calling `socket.bind()` will instruct the socket to begin listening for datagram messages. When `address` and `port` are not passed to `socket.bind()` the method will bind the socket to the "all interfaces" address on a random port (it does the right thing for

both `udp4` and `udp6` sockets). The bound address and port can be retrieved using `socket.address().address` and `socket.address().port`.