Docs » Api » Dgram

UDP / Datagram Sockets

```
Stability: 3 - Stable
```

Datagram sockets are available through `require('dgram')`.

Important note: the behavior of `dgram.Socket#bind()` has changed in v0.10 and is always asynchronous now. If you have code that looks like this:

```
var s = dgram.createSocket('udp4');
s.bind(1234);
s.addMembership('224.0.0.114');
```

You have to change it to this:

```
var s = dgram.createSocket('udp4');
s.bind(1234, function() {
  s.addMembership('224.0.0.114');
});
```

dgram.createSocket(type[, callback])

- `type` String. Either 'udp4' or 'udp6'
- `callback` Function. Attached as a listener to `message` events. Optional
- Returns: Socket object

Creates a datagram Socket of the specified types. Valid types are `udp4` and `udp6`.

Takes an optional callback which is added as a listener for `message` events.

Call `socket.bind()` if you want to receive datagrams. `socket.bind()` will bind to the "all interfaces" address on a random port (it does the right thing for both `udp4` and `udp6` sockets). You can then retrieve the address and port with `socket.address().address` and `socket.address().port`.

dgram.createSocket(options[, callback])

- `options` Object
- `callback` Function. Attached as a listener to `message` events.
- Returns: Socket object

The `options` object should contain a `type` field of either `udp4` or `udp6` and an optional boolean `reuseAddr` field.

When `reuseAddr` is true `socket.bind()` will reuse the address, even if another process has already bound a socket on it. `reuseAddr` defaults to `false`.

Takes an optional callback which is added as a listener for `message` events.

Call `socket.bind()` if you want to receive datagrams. `socket.bind()` will bind to the "all interfaces" address on a random port (it does the right thing for both `udp4` and `udp6` sockets). You can then retrieve the address and port with `socket.address().address` and `socket.address().port`.

Class: dgram.Socket

The dgram Socket class encapsulates the datagram functionality. It should be created via `dgram.createSocket(...)`

Event: 'message'

- `msg` Buffer object. The message
- `rinfo` Object. Remote address information

Emitted when a new datagram is available on a socket. `msg` is a `Buffer` and `rinfo` is an object with the sender's address information:

```
socket.on('message', function(msg, rinfo) {
  console.log('Received %d bytes from %s:%d\n',
              msg.length, rinfo.address, rinfo.port);
});
```

Event: 'listening'

Emitted when a socket starts listening for datagrams. This happens as soon as UDP sockets are created.

Event: 'close'

Emitted when a socket is closed with `close()`. No new `message` events will be emitted on this socket.

Event: 'error'

- `exception` Error object

Emitted when an error occurs.

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

- `buf` Buffer object or string. 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 hostname or IP address.
- `callback` Function. Called when the message has been sent. Optional.

For UDP sockets, the destination port and address must be specified. A string may be supplied for the `address` parameter, and it will be resolved with DNS.

If the address is omitted or is an empty string, `'0.0.0.0'` or `'::0'` is used instead. Depending on the network configuration, those defaults may or may not work; it's best to be explicit about the destination address.

If the socket has not been previously bound with a call to `bind`, it gets 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 may be specified to detect DNS errors or for determining when it's safe to reuse the `buf` object. Note that DNS lookups delay the time to send for at least one tick. The only way to know for sure that the datagram has been sent is by using a callback.

With consideration for multi-byte characters, `offset` and `length` will be calculated with respect to byte length and not the character position.

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

```
var dgram = require('dgram');
var message = new Buffer("Some bytes");
var client = dgram.createSocket("udp4");
client.send(message, 0, message.length, 41234, "localhost", function(err) {
  client.close();
});
```

A 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 be larger than 64K octets including 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 datagrams are impractical for most hosts and networks.
- The `MTU` is the largest size a given link layer technology can support for datagrams. 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 have a minimum `MTU` of `1500` (like Ethernet).

Note that it's impossible to know in advance the MTU of each link through which a packet might travel, and that generally sending a datagram greater than the (receiver) `MTU` won't work (the packet gets silently dropped, without informing the source that the data did not reach its intended recipient).

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

- `port` Integer
- `address` String, Optional
- `callback` Function with no parameters, Optional. Callback when binding is done.

For UDP sockets, listen for datagrams on a named `port` and optional `address`. If `address` is not specified, the OS will try to listen on all addresses. After binding is done, a "listening" event is emitted and the `callback`(if specified) is called. Specifying both a "listening" event listener and `callback` is not harmful but not very useful.

A bound datagram socket keeps the node process running to receive datagrams.

If binding fails, an "error" event is generated. In rare case (e.g. binding a closed socket), an `Error` may be thrown by this method.

Example of a UDP server listening on port 41234:

```
var dgram = require("dgram");

var server = dgram.createSocket("udp4");

server.on("error", function (err) {
  console.log("server error:\n" + err.stack);
  server.close();
});

server.on("message", function (msg, rinfo) {
  console.log("server got: " + msg + " from " +
    rinfo.address + ":" + rinfo.port);
});

server.on("listening", function () {
  var address = server.address();
  console.log("server listening " +
      address.address + ":" + address.port);
});

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

socket.bind(options[, callback])

- `options` {Object} - Required. Supports the following properties:
- `port` {Number} - Required.
- `address` {String} - Optional.
- `exclusive` {Boolean} - Optional.
- `callback` {Function} - Optional.

The `port` and `address` properties of `options`, as well as the optional callback function, behave as they do on a call to socket.bind(port, [address], [callback]) .

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.

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

socket.close()

Close the underlying socket and stop listening for data on it.

socket.address()

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

socket.setBroadcast(flag)

- `flag` Boolean

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

socket.setTTL(ttl)

- `ttl` Integer

Sets the `IP_TTL` socket option. TTL stands for "Time to Live," but in this context it specifies the number of IP hops that a packet is allowed to go 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 argument to `setTTL()` is a number of hops between 1 and 255. The default on most systems is 64.

socket.setMulticastTTL(ttl)

- `ttl` Integer

Sets the `IP_MULTICAST_TTL` socket option. TTL stands for "Time to Live," but in this context it specifies the number of IP hops that a packet is allowed to go 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 argument to `setMulticastTTL()` is a number of hops between 0 and 255. The default on most systems is 1.

socket.setMulticastLoopback(flag)

- `flag` Boolean

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

socket.addMembership(multicastAddress[, multicastInterface])

- `multicastAddress` String
- `multicastInterface` String, Optional

Tells the kernel to join a multicast group with `IP_ADD_MEMBERSHIP` socket option.

If `multicastInterface` is not specified, the OS will try to add membership to all valid interfaces.

socket.dropMembership(multicastAddress[, multicastInterface])

- `multicastAddress` String
- `multicastInterface` String, Optional

Opposite of `addMembership` - tells the kernel to leave a multicast group with `IP_DROP_MEMBERSHIP` socket option. This is automatically called by the kernel when the socket is closed or process terminates, so most apps will never need to call this.

If `multicastInterface` is not specified, the OS will try to drop membership to all valid interfaces.

socket.unref()

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`d calling `unref` again will have no effect.

socket.ref()

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