Variables local to the module will be private. In this example, the variable `PI` is private to `circle.js`.

The module system is implemented in the `require("module")` module.

Cycles

When there are circular `require()` calls, a module might not be done being executed when it is returned.

Consider this situation:

`a.js`:

```
console.log('a starting');
exports.done = false;
var b = require('./b.js');
console.log('in a, b.done = %j', b.done);
exports.done = true;
console.log('a done');
```

`b.js`:

```
console.log('b starting');
exports.done = false;
var a = require('./a.js');
console.log('in b, a.done = %j', a.done);
exports.done = true;
console.log('b done');
```

`main.js`:

```
console.log('main starting');
var a = require('./a.js');
var b = require('./b.js');
console.log('in main, a.done=%j, b.done=%j', a.done, b.done);
```

When `main.js` loads `a.js`, then `a.js` in turn loads `b.js`. At that point, `b.js` tries to load `a.js`. In order to prevent an infinite loop an **unfinished copy** of the `a.js` exports object is returned to the `b.js` module. `b.js` then finishes loading, and its exports object is provided to the `a.js` module.

By the time `main.js` has loaded both modules, they're both finished. The output of this program would thus be:

```
main starting
a starting
b starting
in b, a.done = false
b done
in a, b.done = true
a done
in main, a.done=true, b.done=true
```

If you have cyclic module dependencies in your program, make sure to plan accordingly.

Core Modules

The scripting tool has several core modules compiled into the binary. These modules are described in greater detail elsewhere in this documentation.

Core modules are always preferentially loaded if their identifier is passed to `require()`. For instance, `require('http')` will always return the built in HTTP module, even if there is a file by that name.

File Modules

If the exact filename is not found, then LineRate Scripting will attempt to load the required filename with the added extension of `.js`, and then `.json`.

`.js` files are interpreted as JavaScript text files, and `.json` files are parsed as JSON text files.

A module prefixed with `'/'` is an absolute path to the file. For example, `require('/home/marco/foo.js')` will load the file at `/home/marco/foo.js`.

A module prefixed with `'./'` is relative to the file calling `require()`. That is, `circle.js` must be in the same directory as `foo.js` for `require('./circle')` to find it.

Without a leading '/' or './' to indicate a file, the module is either a "core module" or is loaded from a `node_modules` folder.

If the given path does not exist, `require()` will throw an Error with its `code` property set to `'MODULE_NOT_FOUND'`.

Loading from `node_modules` Folders

If the module identifier passed to `require()` is not a native module, and does not begin with `'/'`, `'../'`, or `'./'`, then LineRate Scripting starts at the parent directory of the current module, and adds `/node_modules`, and attempts to load the module from that location.

If it is not found there, then it moves to the parent directory, and so on, until the root of the tree is reached.

For example, if the file at `'/home/linerate/data/scripting/foo.js'` called `require('bar.js')`, then LineRate Scripting would look in the following locations, in this order:

1. `/home/linerate/data/scripting/proxy/node_modules/bar.js`
2. `/home/linerate/data/scripting/node_modules/bar.js`
3. `/home/linerate/data/node_modules/` (not recommended)
4. `/home/linerate/node_modules` (not recommended)

5. `/home/linerate/data/scripting/lib/node_modules`

This allows programs to localize their dependencies, so that they do not clash.

Folders as Modules

It is convenient to organize programs and libraries into self-contained directories, and then provide a single entry point to that library. There are three ways in which a folder may be passed to `require()` as an argument.

The first is to create a `package.json` file in the root of the folder, which specifies a `main` module. An example package.json file might look like this:

```
{ "name" : "some-library",
  "main" : "./lib/some-library.js" }
```

If this was in a folder at `./some-library`, then `require('./some-library')` would attempt to load `./some-library/lib/some-library.js`.

This is the extent of LineRate Scripting's awareness of package.json files.

If there is no package.json file present in the directory, then LineRate Scripting will attempt to load an `index.js` or `index.node` file out of that directory. For example, if there was no package.json file in the above example, then `require('./some-library')` would attempt to load:

- `./some-library/index.js`
- `./some-library/index.node`

Caching

Modules are cached after the first time they are loaded. This means (among other things) that every call to `require('foo')` will get exactly the same object returned, if it would resolve to the same file.

Multiple calls to `require('foo')` may not cause the module code to be executed multiple times. This is an important feature. With it, "partially done" objects can be returned, thus allowing transitive dependencies to be loaded even when they would cause cycles.

If you want to have a module execute code multiple times, then export a function, and call that function.

Module Caching Caveats

Modules are cached based on their resolved filename. Since modules may resolve to a different filename based on the location of the calling module (loading from `node_modules` folders), it is not a *guarantee* that `require('foo')` will always return the exact same object, if it would resolve to different files.

The `module` Object

In each module, the `module` free variable is a reference to the object representing the current module. In particular, `module.exports` is the same as the `exports` object. `module` isn't actually a global but rather local to each module.

module.exports

The `exports` object is created by the Module system. Sometimes this is not acceptable. Many want their module to be an instance of some class. To do this, assign the desired export object to `module.exports`. For example, suppose we were making a module called `a.js`

```
var EventEmitter = require('events').EventEmitter;

module.exports = new EventEmitter();

// Do some work, and after some time emit
// the 'ready' event from the module itself.
setTimeout(function() {
  module.exports.emit('ready');
}, 1000);
```

Then in another file we could do:

```
var a = require('./a');
a.on('ready', function() {
  console.log('module a is ready');
});
```

Note that assignment to `module.exports` must be done immediately. It cannot be done in any callbacks. This does *not* work:

x.js:

```
setTimeout(function() {
  module.exports = { a: "hello" };
}, 0);
```

y.js:

```
var x = require('./x');
console.log(x.a);
```

module.require(id)

The `id` argument is a string. The `require` object returns `exports` from the resolved module.

The `module.require` method provides a way to load a module as if `require()` was called from the original module.

Note that to do this, you must get a reference to the `module` object. Since `require()` returns the `exports`, and the `module` is typically *only* available within a specific module's code, it must be explicitly exported to be used.

module.id

The identifier (string) for the module. Typically this is the fully resolved filename.

module.filename

The fully resolved filename (string) to the module.

module.loaded

Whether or not the module is done loading or is in the process of loading (Boolean).

module.parent

The module object that required this one.

module.children

The module objects (array) required by this one.

Putting It All Together...

To get the exact filename that will be loaded when `require()` is called, use the `require.resolve()` function.

Putting together all of the above, here is the high-level algorithm in pseudocode of what require.resolve does:

```
require(X) from module at path Y
1. If X is a core module,
   a. return the core module
   b. STOP
2. If X begins with './' or '/' or '../'
   a. LOAD_AS_FILE(Y + X)
   b. LOAD_AS_DIRECTORY(Y + X)
3. LOAD_NODE_MODULES(X, dirname(Y))
4. THROW "not found"

LOAD_AS_FILE(X)
1. If X is a file, load X as JavaScript text.  STOP
2. If X.js is a file, load X.js as JavaScript text.  STOP
3. If X.node is a file, load X.node as binary addon.  STOP

LOAD_AS_DIRECTORY(X)
1. If X/package.json is a file,
   a. Parse X/package.json, and look for "main" field.
   b. let M = X + (json main field)
   c. LOAD_AS_FILE(M)
```

```
2. If X/index.js is a file, load X/index.js as JavaScript text.  STOP
3. If X/index.node is a file, load X/index.node as binary addon.  STOP

LOAD_NODE_MODULES(X, START)
1. let DIRS=NODE_MODULES_PATHS(START)
2. for each DIR in DIRS:
   a. LOAD_AS_FILE(DIR/X)
   b. LOAD_AS_DIRECTORY(DIR/X)

NODE_MODULES_PATHS(START)
1. let PARTS = path split(START)
2. let ROOT = index of first instance of "node_modules" in PARTS, or 0
3. let I = count of PARTS - 1
4. let DIRS = []
5. while I > ROOT,
   a. if PARTS[I] = "node_modules" CONTINUE
   c. DIR = path join(PARTS[0 .. I] + "node_modules")
   b. DIRS = DIRS + DIR
   c. let I = I - 1
6. return DIRS
```

Loading from the global folders

If the `NODE_PATH` environment variable is set to a colon-delimited list of absolute paths, then LineRate Scripting will search those paths for modules if they are not found elsewhere.

Additionally, the system will search in the following locations:

1. `/home/linerate/data/scripting/proxy/.node_modules`
2. `/home/linerate/data/scripting/proxy/.node_libraries`

These are mostly for historic reasons. You are highly encouraged to place your dependencies locally in `node_modules` folders. They will be loaded faster and more reliably.

Addenda: Package Manager Tips

The semantics the `require()` function were designed to be general enough to support a number of sane directory structures. Package manager programs such as `dpkg`, `rpm`, and `npm` will hopefully find it possible to build native packages from scripting tool modules without modification.

Below we give a suggested directory structure that could work:

Let's say that we wanted to have the folder at `/home/linerate/data/scripting/proxy/node_modules/<some-package>/<some-version>` hold the contents of a specific version of a package.

Packages can depend on one another. To install package `foo`, you may have to install a specific version of package `bar`. The `bar` package may itself have dependencies, and in some cases, these dependencies may even collide or form cycles. Since scripting tool looks up the `realpath` of any modules it loads (that is, resolves symlinks), and then looks for their dependencies in the `node_modules` folders as described above, this situation is very simple to resolve with the following architecture:

- `/home/linerate/data/scripting/proxy/node_modules/foo/1.2.3/` - Contents of the `foo` package, version 1.2.3.
- `/home/linerate/data/scripting/proxy/node_modules/bar/4.3.2/` - Contents of the `bar` package that `foo` depends on.
- `/home/linerate/data/scripting/proxy/node_modules/foo/1.2.3/node_modules/bar` - Symbolic link to `/home/linerate/data/scripting/proxy/node_modules/bar/4.3.2/`.
- `/home/linerate/data/scripting/proxy/node_modules/bar/4.3.2/node_modules/*` - Symbolic links to the packages that `bar` depends on.

Thus, even if a cycle is encountered, or if there are dependency conflicts, every module will be able to get a version of its dependency that it can use.

When the code in the `foo` package does `require('bar')`, it will get the version that is symlinked into `/home/linerate/data/scripting/proxy/node_modules/foo/1.2.3/node_modules/bar`. Then, when the code in the `bar` package calls `require('quux')`, it'll get the version that is symlinked into `/home/linerate/data/scripting/proxy/node_modules/bar/4.3.2/node_modules/quux`.

LineRate

Net

Standard Node.js Functions That Are Not Supported

The following items in the standard Node.js API are not supported in LineRate Scripting.

Creating or connecting to UNIX domain sockets is not supported in LineRate Scripting. These variants of functions that attempt UNIX domain connections will fail:

- net.connect(path, [connectListener])
- net.createConnection(path, [connectListener])
- server.listen(path, [listeningListener])
- server.listen(handle, [listeningListener])
- socket.connect(path, [connectListener])

The `net` module provides you with an asynchronous network wrapper. It contains methods for creating both servers and clients (called streams). You can include this module with `require('net');`. The `net` module creates Streams that can be read from, written to, or both. These streams are an interface to TCP sockets from the operating system.

TCP streams are used for communicating with other services, either on the local system or on a remote system. A remote system could be in the same network or across the Internet.

TCP Servers in LineRate

LineRate Scripting runs independent processes on multiple cores, so special approaches are required when creating listening or server sockets. See Understanding Script Execution for an overview of the architecture.

A script that tries to listen on a TCP socket via net.createServer() or server.listen() cannot listen on the same port simultaneously in multiple processes. Attempts to listen on the same port may succeed in one process, but will fail and throw an error in other processes. The error will have the `error.message` property "listen Address already in use".

There are two common approaches for this architecture.

Approach 1: Open ephemeral listen sockets in each process

If the listen socket is for a high-load application that needs to be load-balanced, or if it needs to run in each process, then each process can create its own server socket. However, the port for the server socket will not be known until after it is opened, so an external method to communicate that to the clients is required. When server.listen() is called with `0` for `port`, the operating system will ensure each process gets a unique port. Once the port has been assigned, each process can discover which port it was assigned by accessing the `port` property of the server.address(). Here is an example where server.listen() is called with port `0`:

```
var net = require('net');
var pidServer = undefined;
function sayPidAndPort(connection) {
  connection.end('Script running in process ' + process.pid +
                 ' listening on port ' + pidServer.address().port);
}
pidServer = net.createServer(sayPidAndPort);
pidServer.listen(0, function() {
  console.log('Script running in process ' + process.pid +
              ' listening on port ' + pidServer.address().port);
});
```

When this script runs, the following appears in syslog (with different PIDs and ports):

```
LROS: Script running in process 862 listening on port 5293
LROS: Script running in process 1468 listening on port 5294
LROS: Script running in process 1469 listening on port 5296
LROS: Script running in process 1467 listening on port 5295
LROS: Script running in process 3979 listening on port 5297
```

Approach 2: Only open the listen socket in one process

If the listen socket is for simple and low-load uses, it could be opened in only one process. Here's an example where a TCP server is set up that responds with the version of the scripting engine that is currently in use:

```
var net = require('net');
var versionServer = undefined;
function sayVersion(connection) {
  connection.end('Scripting engine version: ' +
                 process.versions['linerate'] + '\n');
}
function startServerIfMaster() {
  if (process.isMaster() === false) {
    console.log('Process ' + process.pid + ' is not master');
    if (versionServer) {
      versionServer.close();
    }
    return;
  }
  versionServer = net.createServer(sayVersion);
  var listenErrorListener = function(err) {
    console.log('Error listening: ' + err.message);
    setTimeout(startServerIfMaster, 10000);
  };
  versionServer.on('error', listenErrorListener);
  versionServer.listen(8124, function() {
```

```
    versionServer.removeListener('error', listenErrorListener);
    console.log('Listening on port ' + versionServer.address().port +
                ' in master PID ' + process.pid);
  });
};

process.on('masterChanged', function() {
  startServerIfMaster();
});
```

This script ensures that there is one process listening on port 8124. That process is identified as the master process. The call process.isMaster() evaluates to `true` in only one of the processes, and `false` in all others. Every time the master process changes, the 'masterChanged' event will fire and all scripts will have an opportunity to reconfigure; in this case, whichever process is master has the responsibility to keep a server listening on port 8124.

When this script runs, the following appears in syslog (with different PIDs):

```
LROS: Listening on port 8124 in master PID 856
LROS: Process 862 is not master
LROS: Process 1467 is not master
LROS: Process 1468 is not master
LROS: Process 1469 is not master
```

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

Creates a new TCP server. The `connectionListener` argument is automatically set as a listener for the 'connection' event.

`options` is an object with the following defaults:

```
{ allowHalfOpen: false
}
```

If `allowHalfOpen` is `true`, then the socket won't automatically send a FIN packet when the other end of the socket sends a FIN packet. The socket becomes non-readable, but still writable. You should call the `end()` method explicitly. See 'end' event for more information.

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

```
var net = require('net');
var server = undefined;
function startServerIfMaster() {
  if (process.isMaster() === false) {
    console.log('Process ' + process.pid + ' is not master');
    if (server) {
      server.close();
    }
    return;
  }
  server = net.createServer(function(c) { //'connection' listener
    console.log('server connected');
    c.on('end', function() {
      console.log('server disconnected');
    });
```

```
    c.write('hello\r\n');
    c.pipe(c);
  });
  var listenErrorListener = function(err) {
    console.log('Error listening: ' + err.message);
    setTimeout(startServerIfMaster, 10000);
  };
  server.on('error', listenErrorListener);
  server.listen(8124, function() {
    server.removeListener('error', listenErrorListener);
    console.log('Listening on port ' + server.address().port +
                ' in master PID ' + process.pid);
  });
}
process.on('masterChanged', startServerIfMaster);
```

Test this by using `telnet`:

```
telnet localhost 8124
```

net.connect(options, [connectionListener]) or net.createConnection(options, [connectionListener])

Constructs a new socket object and opens the socket to the given location. When the socket is established, the 'connect' event will be emitted.

For TCP sockets, `options` argument should be an object which specifies:

- `port`: Port the client should connect to (Required).
- `host`: Host the client should connect to. Defaults to `'localhost'`.
- `localAddress`: Local interface to bind to for network connections.

Common options are:

- `allowHalfOpen`: if `true`, the socket won't automatically send a FIN packet when the other end of the socket sends a FIN packet. Defaults to `false`. See 'end' event for more information.

The `connectListener` parameter will be added as an listener for the 'connect' event.

Here is an example of a client of echo server as described previously:

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

net.connect(port, [host], [connectListener]) or net.createConnection(port, [host], [connectListener])

Similar to [net.connect(options, connectionListener)][] and net.createConnection(options, connectionListener), creates a TCP connection to `port` on `host`. If `host` is omitted, `'localhost'` will be assumed. The `connectListener` parameter will be added as an listener for the 'connect' event.

Class: net.Server

This class is used to create a TCP server. A server is a `net.Socket` that can listen for new incoming connections.

server.listen(port, [host], [backlog], [listeningListener])

Begin accepting connections on the specified `port` and `host`. If the `host` is omitted, the server will accept connections directed to any IPv4 address (`INADDR_ANY`). A port value of zero results in an ephemeral assigned by the OS.

This function is asynchronous. When the server has been bound, 'listening' event will be emitted. The last parameter `listeningListener` will be added as an listener for the 'listening' event.

Backlog is the maximum length of the queue of pending connections. The default value of this parameter is 511 (not 512). Requesting `backlog` greater than the sysctl `kern.ipc.somaxconn` will succeed but the actual socket backlog will be `kern.ipc.somaxconn`.

The `kern.ipc.somaxconn` sysctl can be read:

```
lros# bash "sysctl kern.ipc.somaxconn"
kern.ipc.somaxconn: 4096
```

The `kern.ipc.somaxconn` sysctl applies to all newly-created listening sockets on the system, including virtual IP sockets. It is usually not necessary to adjust this sysctl. It should not be decreased, but it can be increased:

```
lros# bash "sudo sysctl kern.ipc.somaxconn=8192"
kern.ipc.somaxconn: 4096 -> 8192
```

The backlog of sockets can be inspected with sockstat, in the `Listen` column's `maxqlen` portion:

```
lros# bash "netstat -aL -f inet"
Current listen queue sizes (qlen/incqlen/maxqlen)
Proto Listen         Local Address
tcp4  0/0/511        *.8005
tcp4  0/0/2048       *.8004
tcp4  0/0/500        *.8003
tcp4  0/0/100        *.8002
tcp4  0/0/10         *.8001
tcp4  0/0/128        *.ssh
```

```
tcp4  0/0/128        localhost.3001
tcp4  0/0/4096       *.8443
tcp4  0/0/511        localhost.6379
tcp4  0/0/128        localhost.krb524
tcp4  0/0/128        localhost.dectalk
tcp4  0/0/128        localhost.conf
tcp4  0/0/128        localhost.dec-notes
tcp4  0/0/128        localhost.2100
```

One issue you may run into is getting `EADDRINUSE` errors. This means that another server is already running on the requested port. The other server could be created by another instance of the same script in a different process (see TCP Servers in LineRate). Or, it could be a different server that will be closed, so the script should wait a second and then try again. This can be done with:

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

Note: All sockets in scripting tool set `SO_REUSEADDR` already.

server.close([cb])

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. Optionally, you can pass a callback to listen for the `'close'` event.

server.address()

Returns the bound address, the address family name, and port of the server, as reported by the operating system. Useful to find which port was assigned when getting an OS-assigned address. Returns an object with three properties, for example, `{ port: 12346, family: 'IPv4', address: '127.0.0.1' }`

Example:

```
var server = net.createServer(function (socket) {
  socket.end("goodbye\n");
});

// grab an ephemeral port.
server.listen(function() {
  address = server.address();
  console.log("Process " + process.pid + " opened server on %j", address);
});
```

The syslog will have log messages like the following, with different process IDs and ports. Each process has opened its own ephemeral port (see TCP Servers in LineRate):

```
LROS: Process 3417 opened server on {"family":"IPv4","port":4608,"address":"0.0.0.0"}
LROS: Process 3414 opened server on {"family":"IPv4","port":4609,"address":"0.0.0.0"}
LROS: Process 3415 opened server on {"family":"IPv4","port":4610,"address":"0.0.0.0"}
LROS: Process 3416 opened server on {"family":"IPv4","port":4611,"address":"0.0.0.0"}
```

Don't call `server.address()` until the `'listening'` event has been emitted.

server.maxConnections

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

server.connections

The number of concurrent connections on the server.

Implements: EventEmitter

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

Event: 'listening'

Listener signature: `function() {}`

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

Event: 'connection'

Listener signature: `function(socket) {}`

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

Event: 'close'

Listener signature: `function() {}`

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

Event: 'error'

Listener signature: `function(error) {}`

Emitted when an error occurs. The `close` event is emitted directly following this event. See example in discussion of server.listen().

Class: net.Socket

This object is an abstraction of a TCP socket. `net.Socket` instances implement a duplex Stream interface. They can be created by the user and used as a client (with `connect()`) or they can be created by Node and passed to the user through the `'connection'` event of a server.

new net.Socket([options])

Construct a new socket object.

`options` is an object with the following defaults:

```
{ fd: null
  type: null
  allowHalfOpen: false
}
```

`fd` allows you to specify the existing file descriptor of socket. `type` specified underlying protocol. It can be `'tcp4'` or `'tcp6'`. About `allowHalfOpen`, refer to `createServer()` and `'end'` event.

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

Opens the connection for a given socket. If `port` and `host` are given, then the socket will be opened as a TCP socket, if `host` is omitted, `localhost` will be assumed.

Normally this method is not needed, as `net.createConnection` opens the socket. Use this only if you are implementing a custom Socket or if a Socket is closed and you want to reuse it to connect to another server.

This function is asynchronous. When the 'connect' event is emitted the socket is established. If there is a problem connecting, the `'connect'` event will not be emitted, the `'error'` event will be emitted with the exception.

The `connectListener` parameter will be added as an listener for the 'connect' event.

socket.bufferSize

`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 will internally queue up the data written to a socket and send it out over the wire when it is possible. (Internally it is polling on the socket's file descriptor for being writable).

The consequence of this internal buffering is that memory may grow. This property shows the number of characters currently buffered to be written. (Number of characters is approximately equal to the number

of bytes to be written, but the buffer may contain strings, and the strings are lazily encoded, so the exact number of bytes is not known.)

Users who experience large or growing `bufferSize` should attempt to "throttle" the data flows in their program with `pause()` and `resume()`.

socket.setEncoding([encoding])

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

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

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 - this may not be immediately.

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

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

If `data` is specified, it is equivalent to calling `socket.write(data, encoding)` followed by `socket.end()`.

socket.destroy()

Ensures that no more I/O activity happens on this socket. Only necessary in case of errors (parse error or so).

socket.pause()

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

socket.resume()

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

socket.setTimeout(timeout, [callback])

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 `end()` or `destroy()` the socket.

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.setNoDelay([noDelay])

Disables the Nagle algorithm. By default TCP connections use the Nagle algorithm, they buffer data before sending it off. Setting `true` for `noDelay` will immediately fire off data each time `socket.write()` is called. `noDelay` defaults to `true`.

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

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

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. Defaults to `0`.

socket.address()

Returns the bound address, the address family name and port of the socket as reported by the operating system. Returns an object with three properties, e.g. `{ port: 12346, family: 'IPv4', address: '127.0.0.1' }`

socket.remoteAddress

The string representation of the remote IP address. For example, `'74.125.127.100'` or `'2001:4860:a005::68'`.

socket.remotePort

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

socket.bytesRead

The amount of received bytes.

socket.bytesWritten

The amount of bytes sent.

Implements: EventEmitter

The `net.Socket` class is an EventEmitter with the following events:

Event: 'connect'

Listener signature: `function() {}`

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

Event: 'data'

Listener signature: `function(buffer) {}`

Emitted when data is received. The argument `data` will be a `Buffer` or `String`. Encoding of data is set by `socket.setEncoding()`. (See the Readable Stream section for more information.)

Note that the **data will be lost** if there is no listener when a `Socket` emits a `'data'` event.

Event: 'end'

Listener signature: `function() {}`

Emitted when the other end of the socket sends a FIN packet.

By default (`allowHalfOpen == false`) the socket will destroy its file descriptor once it has written out its pending write queue. However, by setting `allowHalfOpen == true` the socket will not automatically `end()` its side allowing the user to write arbitrary amounts of data, with the caveat that the user is required to `end()` their side now.

Event: 'timeout'

Listener signature: `function() {}`

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()`

Event: 'drain'

Listener signature: `function() {}`

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

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

Event: 'error'

Listener signature: `function(error) {}`

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

Event: 'close'

Listener signature: `function(had_error) {}`

Emitted once the socket is fully closed. The argument `had_error` is a Boolean that says if the socket was closed due to a transmission error.

net.isIP(input)

Tests if input is an IP address. Returns 0 for invalid strings, returns 4 for IP version 4 addresses, and returns 6 for IP version 6 addresses.

net.isIPv4(input)

Returns true if input is a version 4 IP address, otherwise returns false.

net.isIPv6(input)

Returns true if input is a version 6 IP address, otherwise returns false.

LineRate

Os

Provides a few basic operating-system related utility functions.

Use `require('os')` to access this module.

os.tmpDir()

Returns the operating system's default directory for temp files.

os.hostname()

Returns the hostname of the operating system.

os.type()

Returns the operating system name.

os.platform()

Returns the operating system platform.

os.arch()

Returns the operating system CPU architecture.

os.release()

Returns the operating system release.

os.uptime()

Returns the system uptime in seconds.

os.loadavg()

Returns an array containing the 1, 5, and 15 minute load averages.

os.totalmem()

Returns the total amount of system memory in bytes.

os.freemem()

Returns the amount of free system memory in bytes.

os.cpus()

Returns an array of objects containing information about each CPU/core installed: model, speed (in MHz), and times (an object containing the number of CPU ticks spent in: user, nice, sys, idle, and irq).

Example inspection of os.cpus:

```
[ { model: 'Intel(R) Core(TM) i7 CPU         860  @ 2.80GHz',
    speed: 2926,
    times:
     { user: 252020,
       nice: 0,
       sys: 30340,
       idle: 1070356870,
       irq: 0 } },
  { model: 'Intel(R) Core(TM) i7 CPU         860  @ 2.80GHz',
    speed: 2926,
    times:
     { user: 306960,
       nice: 0,
       sys: 26980,
       idle: 1071569080,
       irq: 0 } },
```

```
{ model: 'Intel(R) Core(TM) i7 CPU         860  @ 2.80GHz',
  speed: 2926,
  times:
   { user: 248450,
     nice: 0,
     sys: 21750,
     idle: 1070919370,
     irq: 0 } },
{ model: 'Intel(R) Core(TM) i7 CPU         860  @ 2.80GHz',
  speed: 2926,
  times:
   { user: 256880,
     nice: 0,
     sys: 19430,
     idle: 1070905480,
     irq: 20 } },
{ model: 'Intel(R) Core(TM) i7 CPU         860  @ 2.80GHz',
  speed: 2926,
  times:
   { user: 511580,
     nice: 20,
     sys: 40900,
     idle: 1070842510,
     irq: 0 } },
{ model: 'Intel(R) Core(TM) i7 CPU         860  @ 2.80GHz',
  speed: 2926,
  times:
   { user: 291660,
     nice: 0,
     sys: 34360,
     idle: 1070888000,
     irq: 10 } },
{ model: 'Intel(R) Core(TM) i7 CPU         860  @ 2.80GHz',
  speed: 2926,
  times:
   { user: 308260,
     nice: 0,
     sys: 55410,
     idle: 1071129970,
     irq: 880 } },
{ model: 'Intel(R) Core(TM) i7 CPU         860  @ 2.80GHz',
  speed: 2926,
  times:
   { user: 266450,
     nice: 1480,
     sys: 34920,
     idle: 1072572010,
     irq: 30 } } ]
```

os.networkInterfaces()

Get a list of network interfaces:

```
{ lo0:
   [ { address: '::1', family: 'IPv6', internal: true },
     { address: 'fe80::1', family: 'IPv6', internal: true },
     { address: '127.0.0.1', family: 'IPv4', internal: true } ],
  en1:
   [ { address: 'fe80::cabc:c8ff:feef:f996', family: 'IPv6',
       internal: false },
     { address: '10.0.1.123', family: 'IPv4', internal: false } ],
  vmnet1: [ { address: '10.99.99.254', family: 'IPv4', internal: false } ],
```

```
vmnet8: [ { address: '10.88.88.1', family: 'IPv4', internal: false } ],
ppp0: [ { address: '10.2.0.231', family: 'IPv4', internal: false } ] }
```

os.EOL

A constant defining the appropriate End-of-line marker for the operating system.

LineRate

Path

This module contains utilities for handling and transforming file paths. Almost all these methods perform only string transformations. The file system is not consulted to check whether paths are valid.

Use `require('path')` to use this module. The following methods are provided:

path.normalize(p)

Normalize a string path, taking care of `'..'` and `'.'` parts.

When multiple slashes are found, they're replaced by a single one; when the path contains a trailing slash, it is preserved.

Example:

```
path.normalize('/foo/bar//baz/asdf/quux/..')
// returns
'/foo/bar/baz/asdf'
```

path.join([path1], [path2], [...])

Join all arguments together and normalize the resulting path. Non-string arguments are ignored.

Example:

```
path.join('/foo', 'bar', 'baz/asdf', 'quux', '..')
// returns
'/foo/bar/baz/asdf'

path.join('foo', {}, 'bar')
// returns
'foo/bar'
```

path.resolve([from ...], to)

Resolves `to` to an absolute path.

If `to` isn't already absolute, `from` arguments are prepended in right to left order, until an absolute path is found. If, after using all `from` paths, still no absolute path is found, the current working directory is used as well. The resulting path is normalized, and trailing slashes are removed unless the path gets resolved to the root directory. Non-string arguments are ignored.

Another way to think of it is as a sequence of `cd` commands in a shell.

```
path.resolve('foo/bar', '/tmp/file/', '..', 'a/../subfile')
```

Is similar to:

```
cd foo/bar
cd /tmp/file/
cd ..
cd a/../subfile
pwd
```

The difference is that the different paths don't need to exist and may also be files.

Examples:

```
path.resolve('/foo/bar', './baz')
// returns
'/foo/bar/baz'

path.resolve('/foo/bar', '/tmp/file/')
// returns
'/tmp/file'

path.resolve('wwwroot', 'static_files/png/', '../gif/image.gif')
// if currently in /home/myself/node, it returns
'/home/myself/node/wwwroot/static_files/gif/image.gif'
```

path.relative(from, to)

Solves the relative path from `from` to `to`.

At times, we have two absolute paths, and we need to derive the relative path from one to the other. This is actually the reverse transform of `path.resolve`, which means we see that:

```
path.resolve(from, path.relative(from, to)) == path.resolve(to)
```

Examples:

```
path.relative('C:\\orandea\\test\\aaa', 'C:\\orandea\\impl\\bbb')
// returns
'..\\..\\impl\\bbb'

path.relative('/data/orandea/test/aaa', '/data/orandea/impl/bbb')
```

```
// returns
'../../impl/bbb'
```

path.dirname(p)

Returns the directory name of a path. Similar to the UNIX `dirname` command.

Example:

```
path.dirname('/foo/bar/baz/asdf/quux')
// returns
'/foo/bar/baz/asdf'
```

path.basename(p, [ext])

Returns the last portion of a path. Similar to the UNIX `basename` command.

Example:

```
path.basename('/foo/bar/baz/asdf/quux.html')
// returns
'quux.html'

path.basename('/foo/bar/baz/asdf/quux.html', '.html')
// returns
'quux'
```

path.extname(p)

Returns the extension of the path, from the last '.' to end of string in the last portion of the path. If there is no '.' in the last portion of the path or the first character of it is '.', then it returns an empty string. Examples:

```
path.extname('index.html')
// returns
'.html'

path.extname('index.')
// returns
'.'

path.extname('index')
// returns
''
```

path.sep

The platform-specific file separator. `'\\'` or `'/'`.

An example on Linux:

```
'foo/bar/baz'.split(path.sep)
// returns
['foo', 'bar', 'baz']
```

An example on Windows:

```
'foo\\bar\\baz'.split(path.sep)
// returns
['foo', 'bar', 'baz']
```

Process

Standard Node.js Functions That Are Not Supported

The following item in the standard Node.js API is not supported in LineRate Scripting:

- process.stdin

The `process` object is a global object and can be accessed from anywhere.

Implements: EventEmitter

The `process` object is an event emitter EventEmitter with the following events:

Event: 'exit'

Listener Signature: `function() {}`

Emitted in each LineRate process when the script is about to exit. Scripts can exit because:

- A script's admin status is set to "offline" using the CLI or REST API.
- A script is removed from the configuration using the CLI or REST API.
- A script's source (inline or file) is reconfigured. *Note:* This will not automatically happen when the file contents are changed.
- A script throws an uncaught exception, and the uncaught exception is not handled.

This is a good hook to perform constant time checks of the module's state (like for unit tests). The main event loop will no longer be run after the 'exit' callback finishes, so timers may not be scheduled.

Example of listening for `exit`:

```
process.on('exit', function() {
  process.nextTick(function() {
   console.log('This will not run');
  });
  console.log('About to exit.');
});
```

Event: 'uncaughtException'

Listener Signature: `function(err) {}`

Emitted when an exception bubbles all the way back to the event loop. If a listener is added for this exception, the default action will not occur.

The default action when an uncaught exception is encountered is to:

1. Send HTTP 502 Bad Gateway errors for all HTTP requests that have been emitted from a Virtual Server or Forward Proxy, but have no response data (headers or body) written.
2. Close early all HTTP requests that have had a partial response written.
3. Close all http.ClientRequests initiated by the script.
4. Close all net.Sockets created by the script.
5. Responses connected to other responses with http.ClientResponse.pipe() will continue to pipe; they will not be closed early or have HTTP 502 errors written to them.
6. Cause the script to be halted in all other datapath processes, even if this script did not encounter errors in other datapath processes.
7. Record a backtrace of where the error occurred, and save it in the REST API node '/status/script/<scriptName>/lastError/message'
8. Record the time that the error occurred, and save it in the REST API node '/status/script/<scriptName>/lastError/timestamp'
9. If a script has the auto-restart timeout configured, it will be restarted once that timeout fires.

This behavior is designed to provide consistent behavior across processes. The script is intentionally halted in all processes even if only one encounters an error. Otherwise, in the clustered environment of LineRate, it would be nondeterministic whether a particular request would be processed by a script or not.

Example of listening for `uncaughtException`:

```
process.on('uncaughtException', function(err) {
  console.log('Caught exception: ' + err);
  throw err;  // Best practice: rethrow exception
});

// Intentionally cause an exception, but don't catch it.
nonexistentFunc();
console.log('This will not run.');
```

Note that if the uncaught exception handler throws, then any pending events will not fire. Since console.log() and console.error() use synchronous streams, log and error messages from the uncaught exception handler will be recorded.

For more information about handling uncaught exceptions, see Uncaught Exceptions in the Scripting Developer's Guide.

Event: 'masterChanged' (LineRate Extension)

This event is a LineRate extension.

Listener signature: `function() {}`

Emitted when the master process changes and once at script startup. The script can check if it is running in the new master process by calling process.isMaster().

process.stdout (Implements Writable Stream)

The `process.stdout` implements the Writable Stream interface and is writable stream to `stdout`. `stdout` appears in the logs. For more details refer to Logging with console.log.

Example: the definition of `console.log`

```
console.log = function (d) {
  process.stdout.write(d + '\n');
};
```

`process.stderr` and `process.stdout` are unlike other streams in that writes to them are usually blocking.

process.stderr (Implements Writable Stream)

The `process.stderr` implements the Writable Stream interface and is a writable stream to `stderr`.

`process.stderr` and `process.stdout` are unlike other streams in that *writes to them are blocking*. They are blocking when they refer to regular files or TTY file descriptors. When refer to pipes, they are non-blocking like other streams.

process.cwd()

Returns the current working directory of the process.

```
console.log('Current directory: ' + process.cwd());
```

process.env

An object containing the user environment. See environ(7).

process.pid

The PID of the process.

```
console.log('This process is pid ' + process.pid);
```

process.arch

What processor architecture you're running on: `'arm'`, `'ia32'`, or `'x64'`.

```
console.log('This processor architecture is ' + process.arch);
```

process.platform

What platform you're running on: `'darwin'`, `'freebsd'`, `'linux'`, `'solaris'` or `'win32'`

```
console.log('This platform is ' + process.platform);
```

process.nextTick(callback)

On the next loop around, the event loop calls this callback. This is *not* a simple alias to `setTimeout(fn, 0)`, it's much more efficient.

```
process.nextTick(function () {
  console.log('nextTick callback');
});
```

process.isMaster() (LineRate Extension)

This method is a LineRate extension.

Returns `true` if the current process is the master process. A script executes independently in multiple LineRate processes. See Understanding Script Execution for more details. After the 'masterChanged' event is emitted, this will evaluate to `true` in exactly one process, and `false` in all others, until the 'masterChanged' event is emitted again.

The master process can change due to configuration changes or system problems; scripts should use this method and the 'masterChanged' event to execute code in only one process.

```
var wasMaster = undefined;
process.on('masterChanged', function () {
  if (wasMaster === undefined) {
    if (process.isMaster()) {
      console.log('Process ' + process.pid + ' is the master');
    } else {
      console.log('Process ' + process.pid + ' is not master');
    }
  } else if (wasMaster === true) {
    if (process.isMaster() === true) {
      console.log('Process ' + process.pid + ' is still master');
    } else {
      console.log('Process ' + process.pid + ' is not master any more');
    }
  } else if (process.isMaster()) {
    console.log('Process ' + process.pid + ' is the new master');
  } else {
    console.log('Process ' + process.pid + ' still is not master');
  }
  wasMaster = process.isMaster();
});
```

For forward compatibility, scripts should not rely on particular triggers or behavior for electing a master. However, during testing, the master process can be changed by killing processes or adjusting the number of proxy processes:

```
lros# bash
[admin@lros ~]$ sudo kill <PID of master>
[admin@lros ~]$ exit
lros# conf
lros(config)# proxy processes 10
lros(config)# proxy processes 1
lros(config)# proxy processes auto
```

process.busyTimeout (LineRate Extension)

This property is a LineRate extension.

Specifies how long the script can run (in milliseconds) before being automatically terminated. Valid values are from 1 to 3000 ms (3 seconds). The default is 2000 ms (2 seconds).

The system is designed to gracefully diagnose and recover from two simultaneous script hangs. The system MAY take more aggressive actions to restore system functionality, including intentionally reloading the system, if more scripts hang simultaneously.

The system checks every 100 ms to see if the script is running longer than permitted by `process.busyTimeout`. If the script is running too long, the system terminates the script immediately. A script can run just less than 200 ms more than the allowed time.

The example below sets the busy timeout to 1/2 second and logs it:

```
process.busyTimeout = 500;
console.log('process.busyTimeout: ' + process.busyTimeout);
// 'process.busyTimeout: 500' appears in the logged output
```

process.hrtime()

Returns the current high-resolution real time in a `[seconds, nanoseconds]` tuple Array. It is relative to an arbitrary time in the past. It is not related to the time of day and therefore not subject to clock drift. The primary use is for measuring performance between intervals.

You may pass in the result of a previous call to `process.hrtime()` to get a diff reading, useful for benchmarks and measuring intervals:

```
var time = process.hrtime();
// [ 1800216, 25 ]

setTimeout(function() {
  var diff = process.hrtime(time);
  // [ 1, 552 ]

  console.log('benchmark took %d nanoseconds', diff[0] * 1e9 + diff[1]);
  // benchmark took 1000000527 nanoseconds
}, 1000);
```

LineRate

Punycode

Punycode.js is bundled with LineRate Scripting. Use `require('punycode')` to access it.

punycode.decode(string)

Converts a Punycode string of ASCII code points to a string of Unicode code points.

```
// decode domain name parts
punycode.decode('maana-pta'); // 'mañana'
punycode.decode('--dqo34k'); // '?-?'
```

punycode.encode(string)

Converts a string of Unicode code points to a Punycode string of ASCII code points.

```
// encode domain name parts
punycode.encode('mañana'); // 'maana-pta'
punycode.encode('?-?'); // '--dqo34k'
```

punycode.toUnicode(domain)

Converts a Punycode string representing a domain name to Unicode. Only the Punycoded parts of the domain name will be converted, that is, it doesn't matter if you call it on a string that has already been converted to Unicode.

```
// decode domain names
punycode.toUnicode('xn--maana-pta.com'); // 'mañana.com'
punycode.toUnicode('xn----dqo34k.com'); // '?-?.com'
```

punycode.toASCII(domain)

Converts a Unicode string representing a domain name to Punycode. Only the non- ASCII parts of the domain name will be converted, that is, it doesn't matter if you call it with a domain that's already in ASCII.

```
// encode domain names
punycode.toASCII('mañana.com'); // 'xn--maana-pta.com'
punycode.toASCII('?-?.com'); // 'xn----dqo34k.com'
```

punycode.ucs2

punycode.ucs2.decode(string)

Creates an array containing the decimal code points of each Unicode character in the string. While JavaScript uses UCS-2 internally, this function will convert a pair of surrogate halves (each of which UCS-2 exposes as separate characters) into a single code point, matching UTF-16.

```
punycode.ucs2.decode('abc'); // [97, 98, 99]
// surrogate pair for U+1D306 tetragram for centre:
punycode.ucs2.decode('\uD834\uDF06'); // [0x1D306]
```

punycode.ucs2.encode(codePoints)

Creates a string based on an array of decimal code points.

```
punycode.ucs2.encode([97, 98, 99]); // 'abc'
punycode.ucs2.encode([0x1D306]); // '\uD834\uDF06'
```

punycode.version

A string representing the current Punycode.js version number.

LineRate

Query String

This module provides utilities for dealing with query strings. It provides the following methods:

querystring.stringify(obj, [sep], [eq])

Serializes an object to a query string. Optionally overrides the default separator (`'&'`) and assignment (`'='`) characters.

Example:

```
querystring.stringify({ foo: 'bar', baz: ['qux', 'quux'], corge: '' })
// returns
'foo=bar&baz=qux&baz=quux&corge='

querystring.stringify({foo: 'bar', baz: 'qux'}, ';', ':')
// returns
'foo:bar;baz:qux'
```

querystring.parse(str, [sep], [eq], [options])

Deserializes a query string to an object. Optionally overrides the default separator (`'&'`) and assignment (`'='`) characters.

Options object may contain `maxKeys` property (equal to 1000 by default). It will be used to limit processed keys. Set it to 0 to remove key count limitation.

Example:

```
querystring.parse('foo=bar&baz=qux&baz=quux&corge')
// returns
{ foo: 'bar', baz: ['qux', 'quux'], corge: '' }
```

querystring.escape

The escape function used by `querystring.stringify`, provided so that it could be overridden if necessary.

querystring.unescape

The unescape function used by `querystring.parse`, provided so that it could be overridden if necessary.

LineRate

Stream

A stream is an abstract interface implemented by various objects in LineRate Scripting. For example, a request to an HTTP server is a stream, as is stdout. Streams are readable, writable, or both.

You can access the stream base class by doing `require('stream')`.

Implements: EventEmitter

All streams are instances of EventEmitter with the events as described in the sections below.

Readable Stream

A `Readable Stream` has the following methods, members, and events:

Event: 'data'

Listener Signature: `function(data) {}`

The `'data'` event emits either a `Buffer` (by default) or a string if `setEncoding()` was used.

Note that the **data will be lost** if there is no listener when a `Readable Stream` emits a `'data'` event.

Event: 'end'

Listener Signature: `function() {}`

Emitted when the stream has received an EOF (FIN in TCP terminology). Indicates that no more `'data'` events will happen. If the stream is also writable, it may be possible to continue writing.

Event: 'error'

Listener Signature: `function(exception) {}`

Emitted if there was an error receiving data.

Event: 'close'

Listener Signature: `function() {}`

Emitted when the underlying resource (for example, the backing file descriptor) has been closed. Not all streams will emit this.

stream.readable

A boolean that is `true` by default, but turns `false` after an `'error'` occurs, the stream comes to an `'end'`, or `destroy()` is called.

stream.setEncoding([encoding])

Makes the `'data'` event emit a string instead of a `Buffer`. `encoding` can be `'utf8'`, `'utf16le'` (`'ucs2'`), `'ascii'`, or `'hex'`. Defaults to `'utf8'`.

stream.pause()

Issues an advisory signal to the underlying communication layer, requesting that no further data be sent until `resume()` is called.

Note that, due to the advisory nature, certain streams will not be paused immediately, so `'data'` events may be emitted for some indeterminate period of time even after `pause()` is called. You may want to buffer such `'data'` events.

stream.resume()

Resumes the incoming `'data'` events after a `pause()`.

stream.destroy()

Closes the underlying file descriptor. Stream is no longer `writable` nor `readable`. The stream will not emit any more `'data'` or `'end'` events. Any queued write data will not be sent. The stream should emit a `'close'` event after its resources have been disposed of.

stream.pipe(destination, [options])

This is a `Stream.prototype` method available on all `Stream`s.

Connects this read stream to `destination` WriteStream. Incoming data on this stream gets written to `destination`. The destination and source streams are kept in sync by pausing and resuming as necessary.

This function returns the `destination` stream.

Emulating the UNIX `cat` command:

```
process.stdin.resume(); process.stdin.pipe(process.stdout);
```

By default, `end()` is called on the destination when the source stream emits `end`, so that `destination` is no longer writable. Pass `{ end: false }` as `options` to keep the destination stream open.

This keeps `process.stdout` open so that "Goodbye" can be written at the end.

```
process.stdin.resume();

process.stdin.pipe(process.stdout, { end: false });

process.stdin.on("end", function() {
process.stdout.write("Goodbye\n"); });
```

Writable Stream

A `Writable Stream` has the following methods, members, and events.

Event: 'drain'

Listener Signature: `function() {}`

After a `write()` method returns `false`, this event is emitted to indicate that it is safe to write again.

Event: 'error'

Listener Signature: `function(exception) {}`

Emitted on error with the exception `exception`.

Event: 'close'

Listener Signature: `function() {}`

Emitted when the underlying file descriptor has been closed.

Event: 'pipe'

Listener Signature: `function(src) {}`

Emitted when the stream is passed to a readable stream's pipe method.

stream.writable

A boolean that is `true` by default, but turns `false` after an `'error'` occurrs or `end()` / `destroy()` is called.

stream.write(string, [encoding], [fd])

Writes `string` with the given `encoding` to the stream. Returns `true` if the string has been flushed to the kernel buffer. Returns `false` to indicate that the kernel buffer is full, and the data will be sent out in the future. The `'drain'` event will indicate when the kernel buffer is empty again. The `encoding` defaults to `'utf8'`.

If the optional `fd` parameter is specified, it is interpreted as an integral file descriptor to be sent over the stream. This is only supported for UNIX streams, and is silently ignored otherwise. When writing a file descriptor in this manner, closing the descriptor before the stream drains risks sending an invalid (closed) FD.

stream.write(buffer)

Same as the above except with a raw buffer.

stream.end()

Terminates the stream with EOF or FIN. This call will allow queued write data to be sent before closing the stream.

stream.end(string, encoding)

Sends `string` with the given `encoding` and terminates the stream with EOF or FIN. This is useful to reduce the number of packets sent.

stream.end(buffer)

Same as above but with a `buffer`.

stream.destroy()

Closes the underlying file descriptor. Stream is no longer `writable` nor `readable`. The stream will not emit any more `'data'` or `'end'` events. Any queued write data will not be sent. The stream should emit a `'close'` event after its resources have been disposed of.

stream.destroySoon()

After the write queue is drained, closes the file descriptor. `destroySoon()` can still destroy straight away, as long as there is no data left in the queue for writes.

LineRate

Timers

All of the timer functions are globals. You do not need to `require()` this module in order to use them.

setTimeout(callback, delay, [arg], [...])

Schedules execution of a one-time `callback` after `delay` milliseconds. Returns a `timeoutId` for possible use with `clearTimeout()`. Optionally you can also pass arguments to the callback.

The timeout `delay` must be in the range of 1 - 2,147,483,647 inclusive. If the value is outside this range, it is changed to 1 millisecond. Broadly speaking, a timer cannot span more than 24.8 days.

Note: Your callback will probably not be called in exactly `delay` milliseconds. The system makes no guarantees about the exact timing of when the callback will fire, nor of the order things will fire in. The callback will be called as close as possible to the time specified.

clearTimeout(timeoutId)

Stops a timer that was previously created with `setTimeout()`. The `timeoutId` argument represents the timer identifier that was returned by `setTimeout()`.

setInterval(callback, delay, [arg], [...])

Schedules the repeated execution of `callback` every `delay` milliseconds. Returns a `intervalId` for possible use with `clearInterval()`. Optionally you can also pass arguments to the callback.

The timeout `delay` must be in the range of 1 - 2,147,483,647 inclusive. If the value is outside this range, it is changed to 1 millisecond. Broadly speaking, a timer cannot span more than 24.8 days.

Note: Your callback will probably not be called in exactly `delay` milliseconds. The system makes no guarantees about the exact timing of when the callback will fire, nor of the order things will fire in. The callback will be called as close as possible to the time specified.

clearInterval(intervalId)

Stops a interval timer that was previously created with `setInterval()`. The `timeoutId` argument represents the timer identifier that was returned by `setInterval()`.

LineRate

URL

1. url.parse(urlStr, [parseQueryString], [slashesDenoteHost])
2. url.format(urlObj)
3. url.resolve(from, to)

This module has utilities for URL resolution and parsing. Call `require('url')` to use it.

Parsed URL objects have some or all of the following fields, depending on whether or not they exist in the URL string. Any parts that are not in the URL string will not be in the parsed object. Examples are shown for the URL

`'http://user:pass@host.com:8080/p/a/t...ry=string#hash'`

- `href The full URL that was originally parsed. Both the protocol and host are lowercased.`

  Example: `'http://user:pass@host.com:8080/p/a/t...ry=string#hash'`

- `protocol`: The request protocol, lowercased.

  Example: `'http:'`

- `host`: The full lowercased host portion of the URL, including port information.

  Example: `'host.com:8080'`

- `auth`: The authentication information portion of a URL.

  Example: `'user:pass'`

- `hostname`: Just the lowercased hostname portion of the host.

  Example: `'host.com'`

- `port`: The port number portion of the host.

  Example: `'8080'`

- `pathname`: The path section of the URL, that comes after the host and before the query, including the initial slash if present.

Example: `'/p/a/t/h'`

- `search`: The 'query string' portion of the URL, including the leading question mark.

  Example: `'?query=string'`

- `path`: Concatenation of `pathname` and `search`.

  Example: `'/p/a/t/h?query=string'`

- `query`: Either the 'params' portion of the query string or a querystring-parsed object.

  Example: `'query=string'` or `{'query':'string'}`

- `hash`: The 'fragment' portion of the URL including the pound-sign.

  Example: `'#hash'`

The following methods are provided by the URL module:

url.parse(urlStr, [parseQueryString], [slashesDenoteHost])

Takes a URL string and return an object.

Passes `true` as the second argument to also parse the query string using the `querystring` module. Defaults to `false`.

Passes `true` as the third argument to treat `//foo/bar` as `{ host: 'foo', pathname: '/bar' }` rather than `{ pathname: '//foo/bar' }`. Defaults to `false`.

url.format(urlObj)

Takes a parsed URL object and return a formatted URL string.

- `hrefwill be ignored.`
- `protocol`is treated the same with or without the trailing `:` (colon).
  - The protocols `http`, `https`, `ftp`, `gopher`, and `file` will be postfixed with `://` (colon-slash-slash).
  - All other protocols `mailto`, `xmpp`, `aim`, `sftp`, `foo`, etc will be postfixed with `:` (colon).
- `auth` will be used if present.

- `hostname` will only be used if `host` is absent.
- `port` will only be used if `host` is absent.
- `host` will be used in place of `hostname` and `port`.
- `pathname` is treated the same with or without the leading `/` (slash).
- `search` will be used in place of `query`.
- `query` (object; see [querystring]) will only be used if `search` is absent.
- `search` is treated the same with or without the leading `?` (question mark).
- `hash` is treated the same with or without the leading `#` (pound sign, anchor).

url.resolve(from, to)

Takes a base URL and an href URL, and resolves them as a browser would for an anchor tag.

LineRate

Util

These functions are in the module `'util'`. Use `require('util')` to access them.

util.format(format, [...])

Returns a formatted string using the first argument as a `printf`-like format.

The first argument is a string that contains zero or more *placeholders*. Each placeholder is replaced with the converted value from its corresponding argument. Supported placeholders are:

- `%s` - String.
- `%d` - Number (both integer and float).
- `%j` - JSON.
- `%%` - single percent sign (`'%'`). This does not consume an argument.

If the placeholder does not have a corresponding argument, the placeholder is not replaced.

```
util.format('%s:%s', 'foo'); // 'foo:%s'
```

If there are more arguments than placeholders, the extra arguments are converted to strings with `util.inspect()` and these strings are concatenated, delimited by a space.

```
util.format('%s:%s', 'foo', 'bar', 'baz'); // 'foo:bar baz'
```

If the first argument is not a format string, then `util.format()` returns a string that is the concatenation of all its arguments separated by spaces. Each argument is converted to a string with `util.inspect()`.

```
util.format(1, 2, 3); // '1 2 3'
```

util.debug(string)

A synchronous output function. Will block the process and output `string` immediately to `stderr`.

```
require('util').debug('message on stderr');
```

util.error([...])

Same as `util.debug()`, except this will output all arguments immediately to `stderr`.

util.puts([...])

A synchronous output function. Will block the process and output all arguments to [stdout][] with newlines after each argument.

util.print([...])

A synchronous output function. Will block the process, cast each argument to a string, then output to `stdout`. Does not place newlines after each argument.

util.log(string)

Output with timestamp on `stdout`.

```
require('util').log('Timestamped message.');
```

util.inspect(object, [showHidden], [depth], [colors])

Return a string representation of `object`, which is useful for debugging.

If `showHidden` is `true`, then the object's non-enumerable properties will be shown too. Defaults to `false`.

If `depth` is provided, it tells `inspect` how many times to recurse while formatting the object. This is useful for inspecting large complicated objects.

The default is to only recurse twice. To make it recurse indefinitely, pass in `null` for `depth`.

If `colors` is `true`, the output will be styled with ANSI color codes. Defaults to `false`.

Example of inspecting all properties of the `util` object:

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

console.log(util.inspect(util, true, null));
```

util.isArray(object)

Returns `true` if the given "object" is an `Array`, `false` otherwise.

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

util.isArray([])
  // true
util.isArray(new Array)
  // true
util.isArray({})
  // false
```

util.isRegExp(object)

Returns `true` if the given "object" is a `RegExp`, `false` otherwise.

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

util.isRegExp(/some regexp/)
  // true
util.isRegExp(new RegExp('another regexp'))
  // true
util.isRegExp({})
  // false
```

util.isDate(object)

Returns `true` if the given "object" is a `Date`, `false` otherwise.

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

util.isDate(new Date())
  // true
util.isDate(Date())
  // false (without 'new' returns a String)
util.isDate({})
  // false
```

util.isError(object)

Returns `true` if the given "object" is an `Error`, `false` otherwise.

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

util.isError(new Error())
```

```
  // true
util.isError(new TypeError())
  // true
util.isError({ name: 'Error', message: 'an error occurred' })
  // false
```

util.pump(readableStream, writableStream, [callback])

Experimental

Reads the data from `readableStream` and sends it to the `writableStream`. When `writableStream.write(data)` returns `false`, `readableStream` will be paused until the `drain` event occurs on the `writableStream`. `callback` gets an error as its only argument and is called when `writableStream` is closed or when an error occurs.

util.inherits(constructor, superConstructor)

Inherits the prototype methods from one constructor into another. The prototype of `constructor` will be set to a new object created from `superConstructor`.

As an additional convenience, `superConstructor` will be accessible through the `constructor.super_` property.

```
var util = require("util");
var events = require("events");

function MyStream() {
    events.EventEmitter.call(this);
}

util.inherits(MyStream, events.EventEmitter);

MyStream.prototype.write = function(data) {
    this.emit("data", data);
}

var stream = new MyStream();

console.log(stream instanceof events.EventEmitter); // true
console.log(MyStream.super_ === events.EventEmitter); // true

stream.on("data", function(data) {
    console.log('Received data: "' + data + '"');
})
stream.write("It works!"); // Received data: "It works!"

[stdout]:process.html#process_process_stdout
```

Zlib

You can access this module with:

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

This provides bindings to Gzip/Gunzip, Deflate/Inflate, and DeflateRaw/InflateRaw classes. Each class takes the same options, and is a readable/writable Stream.

Implements Readable and Writable Streams

Each class implements the ReadableStream and WritableStream interfaces.

Examples

Compressing or decompressing a file can be done by piping an fs.ReadStream into a zlib stream, then into an fs.WriteStream.

```
var gzip = zlib.createGzip();
var fs = require('fs');
var inp = fs.createReadStream('input.txt');
var out = fs.createWriteStream('input.txt.gz');

inp.pipe(gzip).pipe(out);
```

Compressing or decompressing data in one step can be done by using the convenience methods.

```
var input = '.................................';
zlib.deflate(input, function(err, buffer) {
  if (!err) {
    console.log(buffer.toString('base64'));
  }
});

var buffer = new Buffer('eJzT0yMAAGTvBe8=', 'base64');
zlib.unzip(buffer, function(err, buffer) {
  if (!err) {
    console.log(buffer.toString());
  }
});
```

To use this module in an HTTP client or server, use the accept-encoding on requests, and the content-encoding header on responses.

Note: These examples are drastically simplified to show the basic concept. Zlib encoding can be expensive, and the results ought to be cached. See Memory Usage Tuning below for more information on the speed/memory/compression tradeoffs involved in zlib usage.

```
// client request example
var zlib = require('zlib');
var http = require('http');
var fs = require('fs');
var request = http.get({ host: 'izs.me',
                         path: '/',
                         port: 80,
                         headers: { 'accept-encoding': 'gzip,deflate' } });
request.on('response', function(response) {
  var output = fs.createWriteStream('izs.me_index.html');

  switch (response.headers['content-encoding']) {
    // or, just use zlib.createUnzip() to handle both cases
    case 'gzip':
      response.pipe(zlib.createGunzip()).pipe(output);
      break;
    case 'deflate':
      response.pipe(zlib.createInflate()).pipe(output);
      break;
    default:
      response.pipe(output);
      break;
  }
});
```

```
// server example
// Running a gzip operation on every request is quite expensive.
// It would be much more efficient to cache the compressed buffer.
var zlib = require('zlib');
var http = require('http');
var fs = require('fs');
http.createServer(function(request, response) {
  var raw = fs.createReadStream('index.html');
  var acceptEncoding = request.headers['accept-encoding'];
  if (!acceptEncoding) {
    acceptEncoding = '';
  }

  // Note: this is not a conformant accept-encoding parser.
  // See http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.3
  if (acceptEncoding.match(/\bdeflate\b/)) {
    response.writeHead(200, { 'content-encoding': 'deflate' });
    raw.pipe(zlib.createDeflate()).pipe(response);
  } else if (acceptEncoding.match(/\bgzip\b/)) {
    response.writeHead(200, { 'content-encoding': 'gzip' });
    raw.pipe(zlib.createGzip()).pipe(response);
  } else {
    response.writeHead(200, {});
    raw.pipe(response);
  }
}).listen(1337);
```

zlib.createGzip([options])

Returns a new Gzip object with an options object.

zlib.createGunzip([options])

Returns a new Gunzip object with an options object.

zlib.createDeflate([options])

Returns a new Deflate object with an options object.

zlib.createInflate([options])

Returns a new Inflate object with an options object.

zlib.createDeflateRaw([options])

Returns a new DeflateRaw object with an options object.

zlib.createInflateRaw([options])

Returns a new InflateRaw object with an options object.

zlib.createUnzip([options])

Returns a new Unzip object with an options object.

Class: zlib.Gzip

Compress data using gzip.

Class: zlib.Gunzip

Decompress a gzip stream.

Class: zlib.Deflate

Compress data using deflate.

Class: zlib.Inflate

Decompress a deflate stream.

Class: zlib.DeflateRaw

Compress data using deflate, and do not append a zlib header.

Class: zlib.InflateRaw

Decompress a raw deflate stream.

Class: zlib.Unzip

Decompress either a Gzip- or Deflate-compressed stream by auto-detecting the header.

Convenience Methods

All of these take a string or Buffer as the first argument, and call the supplied callback with `callback(error, result)`. The compression/decompression engine is created using the default settings in all convenience methods. To supply different options, use the zlib classes directly.

zlib.deflate(buf, callback)

Compress a string with Deflate.

zlib.deflateRaw(buf, callback)

Compress a string with DeflateRaw.

zlib.gzip(buf, callback)

Compress a string with Gzip.

zlib.gunzip(buf, callback)

Decompress a raw Buffer with Gunzip.

zlib.inflate(buf, callback)

Decompress a raw Buffer with Inflate.

zlib.inflateRaw(buf, callback)

Decompress a raw Buffer with InflateRaw.

zlib.unzip(buf, callback)

Decompress a raw Buffer with Unzip.

Options

Each class takes an options object. All options are optional. (The convenience methods use the default settings for all options.)

Note that some options are only relevant when compressing, and are ignored by the decompression classes.

- chunkSize (default: 16*1024)
- windowBits
- level (compression only)
- memLevel (compression only)
- strategy (compression only)
- dictionary (deflate/inflate only, empty dictionary by default)

See the description of `deflateInit2` and `inflateInit2` at http://zlib.net/manual.html#Advanced for more information on these.

Memory Usage Tuning

From `zlib/zconf.h`, modified to node's usage:

The memory requirements for deflate are (in bytes):

```
(1 << (windowBits+2)) +  (1 << (memLevel+9))
```

that is: 128K for windowBits=15 + 128K for memLevel = 8 (default values) plus a few kilobytes for small objects.

For example, if you want to reduce the default memory requirements from 256K to 128K, set the options to:

```
{ windowBits: 14, memLevel: 7 }
```

Of course this will generally degrade compression (there's no free lunch).

The memory requirements for inflate are (in bytes)

```
1 << windowBits
```

that is, 32K for windowBits=15 (default value) plus a few kilobytes for small objects.

This is in addition to a single internal output slab buffer of size `chunkSize`, which defaults to 16K.

The speed of zlib compression is affected most dramatically by the `level` setting. A higher level will result in better compression, but will take longer to complete. A lower level will result in less compression, but will be much faster.

In general, greater memory usage options will mean that node has to make fewer calls to zlib, since it'll be able to process more data in a single `write` operation. So, this is another factor that affects the speed, at the cost of memory usage.

Constants

All of the constants defined in zlib.h are also defined on `require('zlib')`. In the normal course of operations, you will not need to ever set any of these. They are documented here so that their presence is not surprising. This section is taken almost directly from the zlib documentation. See

http://zlib.net/manual.html#Constants for more details.

Allowed flush values.

- `zlib.Z_NO_FLUSH`
- `zlib.Z_PARTIAL_FLUSH`

- `zlib.Z_SYNC_FLUSH`
- `zlib.Z_FULL_FLUSH`
- `zlib.Z_FINISH`
- `zlib.Z_BLOCK`
- `zlib.Z_TREES`

Return codes for the compression/decompression functions. Negative values are errors. Positive values are used for special but normal events.

- `zlib.Z_OK`
- `zlib.Z_STREAM_END`
- `zlib.Z_NEED_DICT`
- `zlib.Z_ERRNO`
- `zlib.Z_STREAM_ERROR`
- `zlib.Z_DATA_ERROR`
- `zlib.Z_MEM_ERROR`
- `zlib.Z_BUF_ERROR`
- `zlib.Z_VERSION_ERROR`

Compression levels.

- `zlib.Z_NO_COMPRESSION`
- `zlib.Z_BEST_SPEED`
- `zlib.Z_BEST_COMPRESSION`
- `zlib.Z_DEFAULT_COMPRESSION`

Compression strategy.

- `zlib.Z_FILTERED`
- `zlib.Z_HUFFMAN_ONLY`

- `zlib.Z_RLE`
- `zlib.Z_FIXED`
- `zlib.Z_DEFAULT_STRATEGY`

Possible values of the data_type field.

- `zlib.Z_BINARY`
- `zlib.Z_TEXT`
- `zlib.Z_ASCII`
- `zlib.Z_UNKNOWN`

The deflate compression method (the only one supported in this version).

- `zlib.Z_DEFLATED`

For initializing zalloc, zfree, opaque.

- `zlib.Z_NULL`

ForwardProxyModule

Use `require('lrs/forwardProxyModule')` to access this module.

The forward proxy module lets your script listen for and act on events for existing or newly created forward proxies.

Implements: EventEmitter

The forwardProxyModule module is an EventEmitter with the following events:

Event: 'create'

Listener signature: `function(forwardProxyObject) {}`

The system emits the `create` event when a new forward proxy is created, then invokes the specifed callback. The event is emitted with the newly created ForwardProxyModule.ForwardProxy object, which we have called `forwardProxyObject` in the function example above.

Use `create` when you want your script to act on any forward proxy. The `create` event only detects the forward proxy object when it is created after the script is online. The system does not invoke the callback for forward proxies that already exist when the script registers the callback.

In the example below, the creation of a forward proxy invokes the `createCallback` function. That function logs the name of the forward proxy using the `.id` property.

Example of listening for `create`:

```
var fpm = require('lrs/forwardProxyModule');
var createCallback = function(forwardProxyObject) {
  console.log('Forward Proxy ' + forwardProxyObject.id + ' created.');
}
fpm.on('create', createCallback);
```

Event: 'exist'

Listener signature: `function(forwardProxyObject) {}`

The system emits the `exist` event immediately if the forward proxy already exists, or whenever the forward proxy is created. The event is emitted with the ForwardProxyModule.ForwardProxy object.

Use `exist` when you want your script to act on only a specific forward proxy that you specify by name. The system invokes the callback for the named forward proxy object, whether it exists when you put the script online or it is created later.

In the example below, the system is listening only for the existence of the forward proxy called myForwardProxy. When the system detects the existence of myForwardProxy, it invokes the `createCallback` function. That function logs the name of myForwardProxy using the `.id` property.

Example of listening for `exist`:

```
var assert = require('assert');
var fpm = require('lrs/forwardProxyModule');
var createCallback = function(forwardProxyObject) {
  assert.equal(forwardProxyObject.id, 'myForwardProxy');
  console.log('Forward Proxy ' + forwardProxyObject.id + ' exists.');
}
fpm.on('exist', 'myForwardProxy', createCallback);
```

ForwardProxyModule.find(name)

Returns a ForwardProxyModule.ForwardProxy object for the forward proxy specified by `name`. If the forward proxy is not found, `find` returns `undefined`.

Use `find` when you know a specific forward proxy already exists and some other event triggers the use of that specific forward proxy. One use of `find` is to support redirecting a request. For example, if the request is for a video, the script can look for the forward proxy of the video optimizer.

In the example below, the system tries to find the forward proxy called myForwardProxy. If it does not find myForwardProxy, the the system returns `undefined`.

Example:

```
var fpm = require('lrs/forwardProxyModule');
var forwardProxyObject = fpm.find('myForwardProxy');
if(forwardProxyObject != undefined) {
```

```
  // do something interesting with the forward proxy object
}
```

Class: ForwardProxyModule.ForwardProxy

This class lets your script listen for and act on request and response events on a forward proxy.

Implements: EventEmitter

The forwardProxyModule.ForwardProxy class is an EventEmitter with the following event:

Event: 'request'

Listener signature: `function(servReq, servResp, cliReq) {}`

The system emits the `request` event when a request is received on the forward proxy. The event is emitted with the `servReq`, `servResp`, and `cliReq` objects.

The `servReq` object is a http.ServerRequest. It is the request from the client of the forward proxy. It is a Readable Stream that starts paused, and will automatically unpause when the first `data` listener is registered.

The `servResp` object is a http.ServerResponse. It is the response that will be returned to the client of the forward proxy. It is a Writable Stream.

The `cliReq` object is a http.ProxiedClientRequest. It is the request that will be sent to the server that is assigned to handle the request. It is a Writable Stream. To make simple use cases concise, it is also a function that connects the objects in a manner equivalent to the following pseudocode (the actual implementation may be different):

```
function cliReq() {
  // Assume that servReq and servResp are in scope.
  if (servReq.forwarded) {
    // Request has been forwarded via moveRequest, do nothing.
    return;
  }
  if (cliReq.res) {
    // Response to client request has already arrived, do nothing.
    return;
  }
  if (servResp.dataHasBeenWritten) {
    // Script has written the servResp; clean up request.
    servReq.destroy();
    cliReq.abort();
    return;
  }
  if (cliReq.dataHasBeenWritten) {
    // Script has written to the backend request, do nothing.
    return;
  }
  // None of the above cases apply; do the default action:
  // Connect the servReq to the cliReq, and when the response arrives,
```

```
  // send it back to the client.
  servReq.bindHeaders(cliReq);
  servReq.fastPipe(cliReq, { 'response' : servResp });
}
```

In the example below, the system is listening for any request on any forward proxy, then adding an "X-Proxy-PID" header, then passing the request back into the original data path.

Example of listening for `request`:

```
var fpm = require('lrs/forwardProxyModule');
var insertExampleHeader = function(servReq, servResp, cliReq) {
  // Add an X-Proxy-PID header (replace with your logic)
  servReq.addHeader('X-Proxy-PID', process.pid);

  // Hand request/response off to the back-end server
  cliReq();
}
var registerForRequest = function(vs) {
  vs.on('request', insertExampleHeader);
}
vsm.on('create', registerForRequest);
```

Event: 'checkContinue'

Listener Signature: `function (servReq, servResp, cliReq) { }`

Emitted each time a request with an http Expect: 100-continue is received.

The `servReq`, `servResp`, and `cliReq` objects are the same as in the request event.

If there is no listener for this event, LineRate **will not** automatically respond with a 100 Continue. This behavior is different than the node.js behavior in HTTP.Server. The behavior in LineRate is easy to use in proxy applications where the decision to accept an expectation is made by the real server.

While the automatic behavior is different, the script can still handle the expectation in any way: it can accept it (sending `100 Continue`), reject it (sending `417 Expectation Failed` for example), or it can pass it along without accepting or rejecting it (calling `cliReq()`).

If the script knows that the real server will accept the request, it should remove the Expect header, and call `servResp.writeContinue`. If the script is accepting the request and generating the response itself, it should call `servResp.writeContinue`. If the script is rejecting the request, it should generate an appropriate HTTP response to prevent the client from sending the request body. Otherwise it should proxy the request to the real server, which allows the real server to process the Expect header and generate an appropriate response.

Note that when this event is emitted and there is an event listener, the `request` event will not be emitted.

In node.js, if the `checkContinue` event is not listened for, `100 Continue` responses are automatically generated. To do the same in LineRate, invoke the following function on the proxy:

```
function applyContinueShim(fp) {
  fp.on('checkContinue', function (servReq, servResp, cliReq) {
    servResp.writeContinue();
    fp.emit('request', servReq, servResp, cliReq);
  });
}
```

ForwardProxy.id

This property is the name of the forward proxy.

ForwardProxy.moveRequest(servReq, servResp)

This method moves requests from the receiving proxy to another proxy. The proxies involved can either be a virtual-server or a forward-proxy. The function needs to be invoked with the `servReq` (http.ServerRequest), and `servResp` (http.ServerResponse) arguments from the receiving proxy.

Once the method completes, `servReq` and `servResp` are dead: they will no longer emit any of the documented events, they will appear as closed streams, and header modification will fail. If there is a script listening for requests on the callee proxy, it will get a `'request'` event with new objects.

Example: This example moves requests receieved on forward proxy `'fp1'` to the virtual server `'vs1'`.

```
var vsm = require('lrs/virtualServerModule');
var fpm = require('lrs/forwardProxyModule');
var vs;

var onVsRequest = function(servReq, servResp, cliReq) {
  // do something interesting with the request or response object
  cliReq();
}

var onFpRequest = function(servReq, servResp, cliReq) {
  console.log('Request received on ' + this.id);
  // This request should be handled by vs1, so move request to vs1.
  vs.moveRequest(servReq, servResp);
  // Now, servReq and servResp are dead.
}

var onFpCreate = function(fp) {
  console.log('Forward proxy ' + fp.id + ' created');
  vs = vsm.find('vs1');
  if(!vs) {
    throw new Error('could not find virtual server vs1');
  }
  vs.on('request', onVsRequest);
  fp.on('request', onFpRequest);
}

fpm.on('exist', 'fp1', onFpCreate);
```

ForwardProxy.newRequest(servReq, servResp)

This function is **deprecated**; please use ForwardProxy.moveRequest() instead.

ManagementRest

Use `require('lrs/managementRest')` to access this module.

The LRS Management REST module accesses a LineRate management REST API. You can change system configuration and read system status via this API. This page documents clients of the management REST API inside scripts. The management REST API itself is documented in the REST API Reference Guide.

Class Client

This class connects as a client to the management REST API.

Event: 'login'

Listener signature: `function() {}`

Emitted when the call to client.logIn() completes successfully.

Event: 'loginFailure'

Listener signature: `function(loginResponse, body) {}`

Emitted when the call to client.logIn() fails because the request was denied, that is, the request was successfully made to the REST API but the REST API rejected it, perhaps due to invalid credentials.

`loginResponse` is an http.ClientResponse from which the body has been entirely read and stored in `body`. `loginResponse` and `body` are from the REST API's response to the login request and can be inspected by the listener to determine why the login failed.

```
var lrsRest = require('lrs/managementRest');
var url = require('url');
var restClient = new lrsRest.Client();
restClient.on('loginFailure', function(loginResponse, body) {
  if (loginResponse.statusCode == 302 &&
      url.parse(loginResponse.headers.Location, true).query.login == 1) {
    console.log('REST login failed, invalid password');
  } else {
    console.log('Unknown failure logging in', loginResponse, body);
  }
});
restClient.logIn({ username: 'admin', password: 'theWrongPassword' });
```

Event: 'loginRequestFailure'

Listener signature: `function(error) {}`

Emitted when the call to client.logIn() fails, because there was an error making the request. `error` is any error that can be emitted by the http.ClientRequest class.

Event: 'logout'

Listener signature: `function() {}`

Emitted when the call to client.logOut() completes successfully. After this event is emitted, the session cookie in client.sid is no longer valid for access to the REST API.

Event: 'logoutRequestFailure'

Listener signature: `function(error) {}`

Emitted when the call to client.logIn() fails because there was an error making the request. `error` is any error that can be emitted by the http.ClientRequest class.

If this event is emitted instead of the event `logout`, the cookie in client.sid may still be valid.

client.apiPrefix

The prefix to the version of the management REST API that this client will use. For example, if it is `'/lrs/api/v1.0'`, then a call like this:

```
client.getJSON({ path: '/system/status/uptime' }, gotUptimeCallback);
```

will query the path `http://127.0.0.1:3001/lrs/api/v1.0/s.../status/uptime`

Defaults to the path to the latest supported REST API version on the system.

client.loggedIn

This is set to true after client.logIn() completes successfully.

client.host

This is set to the `host` option passed to client.logIn(). Defaults to `127.0.0.1`.

client.port

This is set to the `port` option passed to client.logIn(). Defaults to `3001`.

client.sid

This is set to the session ID cookie returned by the management REST API when client.logIn() completes. Anyone with this cookie can access the management API for the lifetime of the session. The cookie is used automatically by this class for future REST operations.

client.logIn(options)

Connects to the REST API and logs in. `options` is an object that can have these properties:

- `username`: A valid username for the LineRate system.
- `password`: The password for the username.
- `host`: The host to log in to. Defaults to `'127.0.0.1'`.
- `port`: The port for access to the REST API. Defaults to `3001`, the internal-only non-SSL REST API interface.
- `path`: The path to the REST API login service. Defaults to `'/login'`.

Usually it is only necessary to specify the `username` and `password` options:

```
var lrsRest = require('lrs/managementRest');
var restClient = new lrsRest.Client();
restClient.on('login', function() { console.log('logged in'); });
restClient.logIn({ username: 'admin', password: 'changeme'});
```

client.logOut(options)

Logs out of the REST API, causing the session ID cookie to expire. `options` is an object that can have this property:

- `path`: The path to the REST API logout service. Defaults to `'/logout'`.

Usually it is not necessary to specify any options.

client.getJSON(options, callback)

Gets a JSON object describing the current value of a management REST API path. The interface is similar to http.get().

`options` is an object that can have this property:

- `path`: The management REST API path to get, for example, `'/status/system/uptime'`.

`callback` is a callback that will be called with the response from the management REST server: `function callback(response) {}`. The `response` argument is an instance of the http.ClientResponse class. If `callback` is not specified, it defaults to printResponse().

A simpler form of invocation is allowed. The `path` can be passed as the first argument instead. For example, these two calls are equivalent:

```
client.getJSON('/status/system/uptime', myCallback);
client.getJSON({path: '/status/system/uptime'}, myCallback);
```

Returns an instance of the http.ClientRequest class that calls `req.end()` automatically.

client.putJSON(options, callback)

Puts a JSON object to a management REST API path using the HTTP PUT method. The interface is similar to http.request().

`options` is an object that can have these properties:

- `path`: The management REST API path to get, for example, `'/status/system/uptime'`.
- `body`: The object to use as the body of the HTTP PUT; either an object (that will be serialized with JSON.stringify) or a string that is valid JSON.

`callback` is a callback that will be called with the response from the management REST server: `function callback(response) {}`. The `response` argument is an instance of the http.ClientResponse class. If `callback` is not specified, it defaults to printResponse().

A simpler form of invocation is allowed. The `path` and `body` can be passed as the first and second arguments instead. For example, these two calls are equivalent:

```
var jsonToPut = { "type": "uint32", "data": "15", "default": false };
client.putJSON('/config/app/proxy/processes', jsonToPut, myCallback);
client.putJSON({path: '/config/app/proxy/processes', body: jsonToPut },
               myCallback);
```

Returns an instance of the http.ClientRequest class that calls `req.end()` automatically.

client.postJSON(options, callback)

Posts a JSON object to a management REST API path using the HTTP POST method. The interface is similar to http.request().

`options` is an object that can have these properties:

- `path`: The management REST API path to get, for example, `'/status/system/uptime'`.
- `body`: The object to use as the body of the HTTP POST; either an object (that will be serialized with JSON.stringify) or a string that is valid JSON.

`callback` is a callback that will be called with the response from the management REST server: `function callback(response) {}`. The `response` argument is an instance of the http.ClientResponse class. If `callback` is not specified, it defaults to printResponse().

A simpler form of invocation is allowed. The `path` and `body` can be passed as the first and second arguments instead. For example, these two calls are equivalent:

```
var jsonToPost = { "type": "string", "data": "vs1", "default": false };
client.postJSON('/config/app/proxy/virtualServer/vs1',
               jsonToPost,
               myCallback);
client.postJSON({path: '/config/app/proxy/virtualServer/vs1',
                 body: jsonToPost },
               myCallback);
```

Returns an instance of the http.ClientRequest class that calls `req.end()` automatically.

client.deleteJSON(options, callback)

Does an HTTP DELETE to a management REST API path. The interface is similar to http.request().

`options` is an object that can have this property:

- `path`: The management REST API path to get, for example, `'/status/system/uptime'`.

`callback` is a callback that will be called with the response from the management REST server: `function callback(response) {}`. The `response` argument is an instance of the http.ClientResponse class. If `callback` is not specified, it defaults to printResponse().

A simpler form of invocation is allowed. The `path` can be passed as the first argument instead. For example, these two calls are equivalent:

```
client.deleteJSON('/config/app/proxy/virtualServer/vs1', myCallback);
client.deleteJSON({path: '/config/app/proxy/virtualServer/vs1'},
                  myCallback);
```

Returns an instance of the http.ClientRequest class that calls `req.end()` automatically.

printResponse(response)

Prints the status code, headers, and body of an http.ClientResponse using console.log(). If a callback argument isn't specified when invoking one of the request methods on the Client class, this function will be used as the default callback.

For example, this request will default to using `printResponse(response)` to handle the response:

```
client.getJSON({path:'/status/system/uptime'});
```

The syslog would contain something similar to:

```
Jun 26 17:29:34 hostname LROS: Management REST xaction STATUS: 200
Jun 26 17:29:34 hostname LROS: Management REST xaction BODY: {"/status/system/
uptime":{"default":false,"type":"uint64","data":942,"numChildren":0,"defaultAllowed":false,"deleteAllowed":false},"h
system/uptime","recurse":false}
```

VirtualServerModule

Use `require('lrs/virtualServerModule')` to access this module.

The virtual server module lets your script listen for and act on events for existing or newly created virtual servers.

Implements: EventEmitter

The virtualServerModule module is an EventEmitter with the following events:

Event: 'create'

Listener signature: `function(virtualServerObject) {}`

The system emits the `create` event when a new virtual server is created, then invokes the specifed callback. The event is emitted with the newly created VirtualServerModule.VirtualServer object, which we have called `virtualServerObject` in the function example above.

Use `create` when you want your script to act on any virtual server. The `create` event only detects the virtual server object when it is created after the script is online. The system does not invoke the callback for virtual servers that already exist when the script registers the callback.

In the example below, the creation of a virtual server invokes the `createCallback` function. That function logs the name of the virtual server using the `.id` property.

Example of listening for `create`:

```
var vsm = require('lrs/virtualServerModule');
var createCallback = function(virtualServerObject) {
  console.log('Virtual Server ' + virtualServerObject.id + ' created.');
};
vsm.on('create', createCallback);
```

Event: 'exist'

Listener signature: `function(virtualServerObject) {}`

The system emits the `exist` event immediately if the virtual server already exists, or whenever the virtual server is created. The event is emitted with the VirtualServerModule.VirtualServer object.

Use `exist` when you want your script to act on only a specific virtual server that you specify by name. The system invokes the callback for the named virtual server object, whether it exists when you put the script online or it is created later.

In the example below, the system is listening only for the existence of the virtual server called `myVirtualServer`. When the system detects the existence of `myVirtualServer`, it invokes the `createCallback` function. That function logs the name of `myVirtualServer` using the `.id` property.

Example of listening for `exist`:

```
var assert = require('assert');
var vsm = require('lrs/virtualServerModule');
var existCallback = function(virtualServerObject) {
  assert.equal(virtualServerObject.id, 'myVirtualServer');
  console.log('Virtual Server ' + virtualServerObject.id + ' exists.');
};
vsm.on('exist', 'myVirtualServer', existCallback);
```

VirtualServerModule.find(name)

Returns a VirtualServerModule.VirtualServer object for the virtual server specified by `name`. If the virtual server is not found, `find` returns `undefined`.

Use `find` when you know a specific virtual server already exists and some other event triggers the use of that specific virtual server. One use of `find` is to support redirecting a request. For example, if the request is for a video, the script can look for the virtual server of the video optimizer.

In the example below, the system tries to find the virtual server called `myVirtualServer`. If it does not find `myVirtualServer`, the the system returns `undefined`.

Example:

```
var vsm = require('lrs/virtualServerModule');
var virtualServerObject = vsm.find('myVirtualServer');
if(virtualServerObject != undefined) {
```

```
  // do something interesting with the virtual server object
}
```

Class: VirtualServerModule.VirtualServer

This class lets your script listen for and act on request and response events on a virtual server.

Implements: EventEmitter

The VirtualServerModule.VirtualServer class is an EventEmitter with the following event:

Event: 'request'

Listener signature: `function(servReq, servResp, cliReq) {}`

The system emits the `request` event when a request is received on the virtual server. The event is emitted with the `servReq`, `servResp`, and `cliReq` objects.

The `servReq` object is a http.ServerRequest. It is the request from the client of the virtual server. It is a Readable Stream that starts paused, and will automatically unpause when the first `data` listener is registered.

The `servResp` object is a http.ServerResponse. It is the response that will be returned to the client of the virtual server. It is a Writable Stream.

The `cliReq` object is a http.ProxiedClientRequest. It is the request that will be sent to the real server that is assigned to handle the request. It is a Writable Stream. To make simple use cases concise, it is also a function that connects the objects in a manner equivalent to the following pseudocode (the actual implementation may be different):

```
function cliReq() {
  // Assume that servReq and servResp are in scope.
  if (servReq.forwarded) {
    // Request has been forwarded via moveRequest, do nothing.
    return;
  }
  if (cliReq.res) {
    // Response to client request has already arrived, do nothing.
    return;
  }
  if (servResp.dataHasBeenWritten) {
    // Script has written the servResp; clean up request.
    servReq.destroy();
    cliReq.abort();
    return;
  }
  if (cliReq.dataHasBeenWritten) {
    // Script has written to the backend request, do nothing.
    return;
  }
  // None of the above cases apply; do the default action:
  // Connect the servReq to the cliReq, and when the response arrives,
```

```
  // send it back to the client.
  servReq.bindHeaders(cliReq);
  servReq.fastPipe(cliReq, { 'response' : servResp });
}
```

In the example below, the system is listening for any request on any virtual server, then adding an "X-Proxy-PID" header, then passing the request back into the original data path.

Example of listening for `request`:

```
var vsm = require('lrs/virtualServerModule');
var insertExampleHeader = function(servReq, servResp, cliReq) {
  // Add an X-Proxy-PID header (replace with your logic)
  servReq.addHeader('X-Proxy-PID', process.pid);

  // Hand request/response off to the back-end server
  cliReq();
};
var registerForRequest = function(vs) {
  vs.on('request', insertExampleHeader);
};
vsm.on('create', registerForRequest);
```

Event: 'checkContinue'

Listener Signature: `function (servReq, servResp, cliReq) { }`

Emitted each time a request with an http Expect: 100-continue is received.

The `servReq`, `servResp`, and `cliReq` objects are the same as in the request event.

If there is no listener for this event, LineRate **will not** automatically respond with a 100 Continue. This behavior is different than the node.js behavior in HTTP.Server. The behavior in LineRate is easy to use in proxy applications where the decision to accept an expectation is made by the real server.

While the automatic behavior is different, the script can still handle the expectation in any way: it can accept it (sending `100 Continue`), reject it (sending `417 Expectation Failed` for example), or it can pass it along without accepting or rejecting it (calling `cliReq()`).

If the script knows that the real server will accept the request, it should remove the Expect header, and call `servResp.writeContinue`. If the script is accepting the request and generating the response itself, it should call `servResp.writeContinue`. If the script is rejecting the request, it should generate an appropriate HTTP response to prevent the client from sending the request body. Otherwise it should proxy the request to the real server, which allows the real server to process the Expect header and generate an appropriate response.

The simplest way to proxy the request is to call the `cliReq()` argument as a function, which provides the same behavior as in the 'VirtualServer.request' event. However, if the script is going to pass the data itself (in order to inspect or transform the data), then it cannot call `cliReq()` because the data is not visible to the script. In this case, it must signal that the proxied request headers should be sent to the

real server without waiting for any data by calling http.ProxiedClientRequest.sendHeaders(), like in the following example:

```
vs.on('checkContinue', function (servReq, servRes, cliReq) {
  servReq.bindHeaders(cliReq);
  servReq.on('data', function (d) {
    // inspect data here
  });
  cliReq.on('continue', function () {
    // Real server says continue; tell the client to continue.
    servRes.writeContinue();
  });
  cliReq.on('response', function (cliResp) {
    // Handle the response here.
  });

  // This causes all body data from servReq to be proxied.  But if the
  // client sent an "Expect: 100-continue" header then it will not send
  // any data until it gets the 100 Continue response or times out.
  servReq.pipe(cliReq);

  // Send the headers now, so that the real server can respond to the
  // "Expect: 100-continue"
  cliReq.sendHeaders();
});
```

Note that when this event is emitted and there is an event listener, the `request` event will not be emitted.

In node.js, if the `checkContinue` event is not listened for, `100 Continue` responses are automatically generated. To do the same in LineRate, invoke the following function on the proxy:

```
function applyContinueShim(vs) {
  vs.on('checkContinue', function (servReq, servResp, cliReq) {
    servResp.writeContinue();
    vs.emit('request', servReq, servResp, cliReq);
  });
}
```

VirtualServer.id

This property is the name of the virtual server.

VirtualServer.moveRequest(servReq, servResp)

This method moves requests from the receiving proxy to another proxy. The proxies involved can either be a virtual server or a forward proxy. The function needs to be invoked with the `servReq` (http.ServerRequest), and `servResp` (http.ServerResponse) arguments from the receiving proxy.

Once the method completes, `servReq` and `servResp` in the caller are dead: they will no longer emit any of the documented events, they will appear as closed streams, and header modification will fail. If there is a script listening for requests on the callee proxy, it will get a `'request'` event with new objects.

Example: This example moves requests receieved on virtual server `'vsA'` to the virtual server `'vsB'`, based on some custom logic for determining A vs. B.

```
var vsm = require('lrs/virtualServerModule');
var vsB = undefined;


var isUserB = function(netSocket) {
  // Replace with custom logic for determining A/B
  return true;
}


var onVsARequest = function(servReq, servResp, cliReq) {
  if (vsB && isUserB(servReq.connection)) {
    // This request should be handled by vsB, so move request to vsB.
    vsB.moveRequest(servReq, servResp);
    // Now, servReq and servResp are dead.
    return;
  }
  // This request should be handled by vsA, so send it along the datapath.
  cliReq();
}


var onVsAExist = function(vs) {
  console.log('virtual server ' + vs.id + ' exists');
  vs.on('request', onVsARequest);
}
vsm.on('exist', 'vsA', onVsAExist);
var onVsBExist = function(vs) {
  console.log('virtual server ' + vs.id + ' exists');
  vsB = vs;
}
vsm.on('exist', 'vsB', onVsBExist);
```

VirtualServer.newRequest(servReq, servResp)

This function is **deprecated**; please use VirtualServer.moveRequest() instead.