Scripting API Reference Guide

Scripting API Reference Guide

This guide provides reference information for LineRate Scripting.

Node.js Modules

LineRate Scripting supports the following Node.js libraries:

- Assert
- Buffer
- Console
- Crypto
- DNS
- Events
- Executing JavaScript
- File System
- Global Objects
- HTTP
- HTTPS
- Modules
- Net
- Os
- Path
- Process
- Punycode
- Query String
- Stream
- Timers
- URL
- Util
- Zlib

LineRate Modules

- ForwardProxyModule
- ManagementRest
- VirtualServerModule

Bundled Node Modules

- abbrev 1.0.4
- jshint 2.4.3
- jslint 0.2.7
- nopt 1.0.10
- redis 0.8.2

Standard Node.js Modules That Are Not Supported

- C/C++ Addons
- Domain
- TLS/SSL
- String Decoder
- UDP/Datagram
- Readline
- REPL
- VM
- Child Processes
- TTY
- Debugger
- Cluster

Legal Notices

Copyright

Trademarks

LineRate

Assert

This module is used for writing unit tests for your applications. You can access it with `require('assert')`.

assert.fail(actual, expected, message, operator)

Throws an exception that displays the values for `actual` and `expected`, separated by the provided operator.

assert(value, message), assert.ok(value, [message])

Tests if `value` is truthy, it is equivalent to `assert.equal(true, !!value, message);`

assert.equal(actual, expected, [message])

Tests shallow, coercive equality with the equal comparison operator (`==`).

assert.notEqual(actual, expected, [message])

Tests shallow, coercive non-equality with the not equal comparison operator (`!=`).

assert.deepEqual(actual, expected, [message])

Tests for deep equality.

assert.notDeepEqual(actual, expected, [message])

Tests for any deep inequality.

assert.strictEqual(actual, expected, [message])

Tests strict equality, as determined by the strict equality operator (`===`)

assert.notStrictEqual(actual, expected, [message])

Tests strict non-equality, as determined by the strict not equal operator (`!==`)

assert.throws(block, [error], [message])

Expects `block` to throw an error. `error` can be constructor, RegExp, or validation function.

Validate instanceof using constructor:

```
assert.throws(
  function() {
    throw new Error("Wrong value");
  },
  Error
);
```

Validate error message using RegExp:

```
assert.throws(
  function() {
    throw new Error("Wrong value");
  },
  /value/
);
```

Custom error validation:

```
assert.throws(
  function() {
    throw new Error("Wrong value");
  },
  function(err) {
    if ( (err instanceof Error) && /value/.test(err) ) {
      return true;
    }
  },
  "unexpected error"
);
```

assert.doesNotThrow(block, [error], [message])

Expects `block` not to throw an error. See assert.throws for details.

assert.ifError(value)

Tests if value is not a false value, and throws if it is a true value. Useful when testing the first argument, `error` in callbacks.

Buffer

Pure JavaScript is Unicode friendly but not nice to binary data. When dealing with TCP streams or the file system, it's necessary to handle octet streams. LineRate Scripting has several strategies for manipulating, creating, and consuming octet streams.

Raw data is stored in instances of the `Buffer` class. A `Buffer` is similar to an array of integers but corresponds to a raw memory allocation outside the V8 heap. A `Buffer` cannot be resized.

The `Buffer` class is a global, making it very rare that one would need to ever `require('buffer')`.

Converting between Buffers and JavaScript string objects requires an explicit encoding method. Here are the different string encodings:

- `'ascii'` - For 7 bit ASCII data only. This encoding method is very fast, and will strip the high bit if set. Note that this encoding converts a null character (`'\0'` or `'\u0000'`) into `0x20` (character code of a space). If you want to convert a null character into `0x00`, you should use `'utf8'`.

- `'utf8'` - Multibyte encoded Unicode characters. Many web pages and other document formats use UTF-8.

- `'base64'` - Base64 string encoding.

- `'binary'` - A way of encoding raw binary data into strings by using only the first 8 bits of each character. This encoding method is deprecated and should be avoided in favor of `Buffer` objects where possible. This encoding will be removed in future versions.

- `'hex'` - Encode each byte as two hexadecimal characters.

Class: Buffer

The `Buffer` class is a global type for dealing with binary data directly. It can be constructed in a variety of ways.

new Buffer(size)

Allocates a new buffer of `size` (number) octets.

new Buffer(array)

Allocates a new buffer using an `array` of octets.

new Buffer(str, [encoding])

- `str` String - string to encode.
- `encoding` String - encoding to use, optional.

Allocates a new buffer containing the given `str`. `encoding` defaults to `'utf8'`.

buf.write(string, [offset], [length], [encoding])

- `string` String - data to be written to buffer
- `offset` Number, Optional, Default: 0
- `length` Number, Optional, Default: `buffer.length - offset`
- `encoding` String, Optional, Default: 'utf8'

Writes `string` to the buffer at `offset` using the given encoding. `offset` defaults to `0`. `encoding` defaults to `'utf8'`. `length` is the number of bytes to write. Returns number of octets written. If `buffer` did not contain enough space to fit the entire string, it will write a partial amount of the string. `length` defaults to `buffer.length - offset`. The method will not write partial characters.

```
buf = new Buffer(256);
len = buf.write('\u00bd + \u00bc = \u00be', 0);
console.log(len + " bytes: " + buf.toString('utf8', 0, len));
```

The number of characters written (which may be different than the number of bytes written) is set in `Buffer._charsWritten` and will be overwritten the next time `buf.write()` is called.

buf.toString([encoding], [start], [end])

- `encoding` String, Optional, Default: 'utf8'
- `start` Number, Optional, Default: 0
- `end` Number, Optional, Default: `buffer.length`

Decodes and returns a string from buffer data encoded with `encoding` (defaults to `'utf8'`) beginning at `start` (defaults to `0`) and ending at `end` (defaults to `buffer.length`).

See `buffer.write()` example, above.

buf[index]

Get and set the octet at `index`. The values refer to individual bytes, so the legal range is between `0x00` and `0xFF` hex or `0` and `255`.

Example: copy an ASCII string into a buffer, one byte at a time:

```
str = "node.js";
buf = new Buffer(str.length);

for (var i = 0; i < str.length ; i++) {
  buf[i] = str.charCodeAt(i);
}

console.log(buf);

// node.js
```

Class Method: Buffer.isBuffer(obj)

- `obj` Object
- Return: Boolean

Tests if `obj` is a `Buffer`.

Class Method: Buffer.byteLength(string, [encoding])

- `string` String
- `encoding` String, Optional, Default: 'utf8'
- Return: Number

Gives the actual byte length of a string. `encoding` defaults to `'utf8'`. This is not the same as `String.prototype.length` since that returns the number of *characters* in a string.

Example:

```
str = '\u00bd + \u00bc = \u00be';

console.log(str + ": " + str.length + " characters, " +
  Buffer.byteLength(str, 'utf8') + " bytes");

// ½ + ¼ = ¾: 9 characters, 12 bytes
```

Class Method: Buffer.concat(list, [totalLength])

- `list` Array - List of Buffer objects to concat
- `totalLength` Number - Total length of the buffers when concatenated

Returns a buffer that is the result of concatenating all the buffers in the list together.

If the list has no items, or if the `totalLength` is 0, then it returns a zero-length buffer.

If the list has exactly one item, then the first item of the list is returned.

If the list has more than one item, then a new `Buffer` is created.

If `totalLength` is not provided, it is read from the buffers in the list. However, this adds an additional loop to the function, so it is faster to provide the length explicitly.

buf.length

The size of the buffer in bytes. Note that this is not necessarily the size of the contents. `length` refers to the amount of memory allocated for the buffer object. It does not change when the contents of the buffer are changed.

```
buf = new Buffer(1234);

console.log(buf.length);
```

```
buf.write("some string", 0, "ascii");
console.log(buf.length);

// 1234
// 1234
```

buf.copy(targetBuffer, [targetStart], [sourceStart], [sourceEnd])

- `targetBuffer` Buffer object - Buffer to copy into
- `targetStart` Number, Optional, Default: 0
- `sourceStart` Number, Optional, Default: 0
- `sourceEnd` Number, Optional, Default: `buffer.length`

Does copy between buffers. The source and target regions can be overlapped. `targetStart` and `sourceStart` default to `0`. `sourceEnd` defaults to `buffer.length`.

Example: build two Buffers, then copy `buf1` from byte 16 through byte 19 into `buf2`, starting at the 8th byte in `buf2`.

```
buf1 = new Buffer(26);
buf2 = new Buffer(26);

for (var i = 0 ; i < 26 ; i++) {
  buf1[i] = i + 97; // 97 is ASCII a
  buf2[i] = 33; // ASCII !
}

buf1.copy(buf2, 8, 16, 20);
console.log(buf2.toString('ascii', 0, 25));

// !!!!!!!!qrst!!!!!!!!!!!!!
```

buf.slice([start], [end])

- `start` Number, Optional, Default: 0
- `end` Number, Optional, Default: `buffer.length`

Returns a new buffer that references the same memory as the old, but offset and cropped by the `start` (defaults to `0`) and `end` (defaults to `buffer.length`) indexes.

Modifying the new buffer slice will modify memory in the original buffer!

Example: build a Buffer with the ASCII alphabet, take a slice, then modify one byte from the original Buffer.

```
var buf1 = new Buffer(26);

for (var i = 0 ; i < 26 ; i++) {
  buf1[i] = i + 97; // 97 is ASCII a
}

var buf2 = buf1.slice(0, 3);
console.log(buf2.toString('ascii', 0, buf2.length));
buf1[0] = 33;
```

```
console.log(buf2.toString('ascii', 0, buf2.length));

// abc
// !bc
```

buf.readUInt8(offset, [noAssert])

- `offset` Number
- `noAssert` Boolean, Optional, Default: false
- Return: Number

Reads an unsigned 8-bit integer from the buffer at the specified offset.

Set `noAssert` to true to skip validation of `offset`. This means that `offset` may be beyond the end of the buffer. Defaults to `false`.

Example:

```
var buf = new Buffer(4);

buf[0] = 0x3;
buf[1] = 0x4;
buf[2] = 0x23;
buf[3] = 0x42;

for (ii = 0; ii < buf.length; ii++) {
  console.log(buf.readUInt8(ii));
}

// 0x3
// 0x4
// 0x23
// 0x42
```

buf.readUInt16LE(offset, [noAssert]) or buf.readUInt16BE(offset, [noAssert])

- `offset` Number
- `noAssert` Boolean, Optional, Default: false
- Return: Number

Reads an unsigned 16-bit integer from the buffer at the specified offset with specified endian format.

Set `noAssert` to true to skip validation of `offset`. This means that `offset` may be beyond the end of the buffer. Defaults to `false`.

Example:

```
var buf = new Buffer(4);

buf[0] = 0x3;
buf[1] = 0x4;
buf[2] = 0x23;
buf[3] = 0x42;
```

```
console.log(buf.readUInt16BE(0));
console.log(buf.readUInt16LE(0));
console.log(buf.readUInt16BE(1));
console.log(buf.readUInt16LE(1));
console.log(buf.readUInt16BE(2));
console.log(buf.readUInt16LE(2));

// 0x0304
// 0x0403
// 0x0423
// 0x2304
// 0x2342
// 0x4223
```

buf.readUInt32LE(offset, [noAssert]) or buf.readUInt32BE(offset, [noAssert])

- `offset` Number
- `noAssert` Boolean, Optional, Default: false
- Return: Number

Reads an unsigned 32-bit integer from the buffer at the specified offset with specified endian format.

Set `noAssert` to true to skip validation of `offset`. This means that `offset` may be beyond the end of the buffer. Defaults to `false`.

Example:

```
var buf = new Buffer(4);

buf[0] = 0x3;
buf[1] = 0x4;
buf[2] = 0x23;
buf[3] = 0x42;

console.log(buf.readUInt32BE(0));
console.log(buf.readUInt32LE(0));

// 0x03042342
// 0x42230403
```

buf.readInt8(offset, [noAssert])

- `offset` Number
- `noAssert` Boolean, Optional, Default: false
- Return: Number

Reads a signed 8-bit integer from the buffer at the specified offset.

Set `noAssert` to true to skip validation of `offset`. This means that `offset` may be beyond the end of the buffer. Defaults to `false`.

Works as `buffer.readUInt8`, except buffer contents are treated as two's complement signed values.

buf.readInt16LE(offset, [noAssert]) or buf.readInt16BE(offset, [noAssert])

- `offset` Number
- `noAssert` Boolean, Optional, Default: false
- Return: Number

Reads a signed 16-bit integer from the buffer at the specified offset with specified endian format.

Set `noAssert` to true to skip validation of `offset`. This means that `offset` may be beyond the end of the buffer. Defaults to `false`.

Works as `buffer.readUInt16*`, except buffer contents are treated as two's complement signed values.

buf.readInt32LE(offset, [noAssert]) or buf.readInt32BE(offset, [noAssert])

- `offset` Number
- `noAssert` Boolean, Optional, Default: false
- Return: Number

Reads a signed 32-bit integer from the buffer at the specified offset with specified endian format.

Set `noAssert` to true to skip validation of `offset`. This means that `offset` may be beyond the end of the buffer. Defaults to `false`.

Works as `buffer.readUInt32*`, except buffer contents are treated as two's complement signed values.

buf.readFloatLE(offset, [noAssert]) or buf.readFloatBE(offset, [noAssert])

- `offset` Number
- `noAssert` Boolean, Optional, Default: false
- Return: Number

Reads a 32-bit float from the buffer at the specified offset with specified endian format.

Set `noAssert` to true to skip validation of `offset`. This means that `offset` may be beyond the end of the buffer. Defaults to `false`.

Example:

```
var buf = new Buffer(4);

buf[0] = 0x00;
buf[1] = 0x00;
buf[2] = 0x80;
buf[3] = 0x3f;
```

```
console.log(buf.readFloatLE(0));

// 0x01
```

buf.readDoubleLE(offset, [noAssert]) or buf.readDoubleBE(offset, [noAssert])

- `offset` Number
- `noAssert` Boolean, Optional, Default: false
- Return: Number

Reads a 64-bit double from the buffer at the specified offset with specified endian format.

Set `noAssert` to true to skip validation of `offset`. This means that `offset` may be beyond the end of the buffer. Defaults to `false`.

Example:

```
var buf = new Buffer(8);

buf[0] = 0x55;
buf[1] = 0x55;
buf[2] = 0x55;
buf[3] = 0x55;
buf[4] = 0x55;
buf[5] = 0x55;
buf[6] = 0xd5;
buf[7] = 0x3f;

console.log(buf.readDoubleLE(0));

// 0.3333333333333333
```

buf.writeUInt8(value, offset, [noAssert])

- `value` Number
- `offset` Number
- `noAssert` Boolean, Optional, Default: false

Writes `value` to the buffer at the specified offset. Note: `value` must be a valid unsigned 8-bit integer.

Set `noAssert` to true to skip validation of `value` and `offset`. This means that `value` may be too large for the specific function, and `offset` may be beyond the end of the buffer leading to the values being silently dropped. This should not be used unless you are certain of correctness. Defaults to `false`.

Example:

```
var buf = new Buffer(4);
buf.writeUInt8(0x3, 0);
buf.writeUInt8(0x4, 1);
buf.writeUInt8(0x23, 2);
buf.writeUInt8(0x42, 3);
```

```
console.log(buf);

// <Buffer 03 04 23 42>
```

buf.writeUInt16LE(value, offset, [noAssert]) or buf.writeUInt16BE(value, offset, [noAssert])

- `value` Number
- `offset` Number
- `noAssert` Boolean, Optional, Default: false

Writes `value` to the buffer at the specified offset with specified endian format. Note: `value` must be a valid unsigned 16-bit integer.

Set `noAssert` to true to skip validation of `value` and `offset`. This means that `value` may be too large for the specific function, and `offset` may be beyond the end of the buffer leading to the values being silently dropped. This should not be used unless you are certain of correctness. Defaults to `false`.

Example:

```
var buf = new Buffer(4);
buf.writeUInt16BE(0xdead, 0);
buf.writeUInt16BE(0xbeef, 2);

console.log(buf);

buf.writeUInt16LE(0xdead, 0);
buf.writeUInt16LE(0xbeef, 2);

console.log(buf);

// <Buffer de ad be ef>
// <Buffer ad de ef be>
```

buf.writeUInt32LE(value, offset, [noAssert]) or buf.writeUInt32BE(value, offset, [noAssert])

- `value` Number
- `offset` Number
- `noAssert` Boolean, Optional, Default: false

Writes `value` to the buffer at the specified offset with specified endian format. Note: `value` must be a valid unsigned 32-bit integer.

Set `noAssert` to true to skip validation of `value` and `offset`. This means that `value` may be too large for the specific function, and `offset` may be beyond the end of the buffer leading to the values being silently dropped. This should not be used unless you are certain of correctness. Defaults to `false`.

Example:

```
var buf = new Buffer(4);
buf.writeUInt32BE(0xfeedface, 0);

console.log(buf);

buf.writeUInt32LE(0xfeedface, 0);

console.log(buf);

// <Buffer fe ed fa ce>
// <Buffer ce fa ed fe>
```

buf.writeInt8(value, offset, [noAssert])

- `value` Number
- `offset` Number
- `noAssert` Boolean, Optional, Default: false

Writes `value` to the buffer at the specified offset. Note: `value` must be a valid signed 8-bit integer.

Set `noAssert` to true to skip validation of `value` and `offset`. This means that `value` may be too large for the specific function, and `offset` may be beyond the end of the buffer leading to the values being silently dropped. This should not be used unless you are certain of correctness. Defaults to `false`.

Works as `buffer.writeUInt8`, except value is written out as a two's complement signed integer into `buffer`.

buf.writeInt16LE(value, offset, [noAssert]) or buf.writeInt16BE(value, offset, [noAssert])

- `value` Number
- `offset` Number
- `noAssert` Boolean, Optional, Default: false

Writes `value` to the buffer at the specified offset with specified endian format. Note: `value` must be a valid signed 16-bit integer.

Set `noAssert` to true to skip validation of `value` and `offset`. This means that `value` may be too large for the specific function, and `offset` may be beyond the end of the buffer leading to the values being silently dropped. This should not be used unless you are certain of correctness. Defaults to `false`.

Works as `buffer.writeUInt16*`, except value is written out as a two's complement signed integer into `buffer`.

buf.writeInt32LE(value, offset, [noAssert]) or buf.writeInt32BE(value, offset, [noAssert])

- `value` Number
- `offset` Number

- `noAssert` Boolean, Optional, Default: false

Writes `value` to the buffer at the specified offset with specified endian format. Note: `value` must be a valid signed 32-bit integer.

Set `noAssert` to true to skip validation of `value` and `offset`. This means that `value` may be too large for the specific function, and `offset` may be beyond the end of the buffer leading to the values being silently dropped. This should not be used unless you are certain of correctness. Defaults to `false`.

Works as `buffer.writeUInt32*`, except value is written out as a two's complement signed integer into `buffer`.

buf.writeFloatLE(value, offset, [noAssert]) or buf.writeFloatBE(value, offset, [noAssert])

- `value` Number
- `offset` Number
- `noAssert` Boolean, Optional, Default: false

Writes `value` to the buffer at the specified offset with specified endian format. Note: `value` must be a valid 32-bit float.

Set `noAssert` to true to skip validation of `value` and `offset`. This means that `value` may be too large for the specific function, and `offset` may be beyond the end of the buffer leading to the values being silently dropped. This should not be used unless you are certain of correctness. Defaults to `false`.

Example:

```
var buf = new Buffer(4);
buf.writeFloatBE(0xcafebabe, 0);

console.log(buf);

buf.writeFloatLE(0xcafebabe, 0);

console.log(buf);

// <Buffer 4f 4a fe bb>
// <Buffer bb fe 4a 4f>
```

buf.writeDoubleLE(value, offset, [noAssert]) or buf.writeDoubleBE(value, offset, [noAssert])

- `value` Number
- `offset` Number
- `noAssert` Boolean, Optional, Default: false

Writes `value` to the buffer at the specified offset with specified endian format. Note: `value` must be a valid 64-bit double.

Set `noAssert` to true to skip validation of `value` and `offset`. This means that `value` may be too large for the specific function, and `offset` may be beyond the end of the buffer leading to the values being silently dropped. This should not be used unless you are certain of correctness. Defaults to `false`.

Example:

```
var buf = new Buffer(8);
buf.writeDoubleBE(0xdeadbeefcafebabe, 0);

console.log(buf);

buf.writeDoubleLE(0xdeadbeefcafebabe, 0);

console.log(buf);

// <Buffer 43 eb d5 b7 dd f9 5f d7>
// <Buffer d7 5f f9 dd b7 d5 eb 43>
```

buf.fill(value, [offset], [end])

- `value`
- `offset` Number, Optional
- `end` Number, Optional

Fills the buffer with the specified value. If the `offset` (defaults to `0`) and `end` (defaults to `buffer.length`) are not given, it will fill the entire buffer.

```
var b = new Buffer(50);
b.fill("h");
```

buffer.INSPECT_MAX_BYTES

- Number, Default: 50

How many bytes will be returned when `buffer.inspect()` is called. This can be overridden by user modules.

Note that this is a property on the buffer module returned by `require('buffer')`, not on the Buffer global or a buffer instance.

Class: SlowBuffer

This class is primarily for internal use. JavaScript programs should use Buffer instead of using SlowBuffer.

To avoid the overhead of allocating many C++ Buffer objects for small blocks of memory in the lifetime of a server, the system, allocates memory in 8Kb (8192 byte) chunks. If a buffer is smaller than this size, then it will be backed by a parent SlowBuffer object. If it is larger than this, then the systrem will allocate a SlowBuffer slab for it directly.

Console

The console is an object for printing to stdout and stderr. Similar to the console object functions provided by most web browsers.

LineRate Extension

There is no stdout/stderr in LineRate, messages sent to stdout/stderr appear in the syslog instead, in `/var/log/controller.messages`.

console.log([data], [...])

Prints to stdout with newline. This function can take multiple arguments in a `printf()`-like way. Example:

```
console.log('count: %d', count);
```

If formatting elements are not found in the first string, then `util.inspect` is used on each argument. See util.format() for more information.

console.info([data], [...])

Same as `console.log`.

console.error([data], [...])

Same as `console.log` but prints to stderr.

console.warn([data], [...])

Same as `console.error`.

console.dir(obj)

Uses `util.inspect` on `obj` and prints resulting string to stdout.

console.time(label)

Marks a time.

console.timeEnd(label)

Finishes timer and records output. Example:

```
console.time('100-elements');
for (var i = 0; i < 100; i++) {
  ;
}
console.timeEnd('100-elements');
```

console.trace(label)

Prints a stack trace to stderr of the current position.

console.assert(expression, [message])

Same as assert.ok() where if the `expression` evaluates as `false`, throws an AssertionError with `message`.

Crypto

Standard Node.js Functions That Are Not Supported

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

- crypto.pbkdf2(password, salt, iterations, keylen, callback), that is, the async version of crypto.pbkdf2()
- crypto.randomBytes(size, callback), that is, the async version of crypto.randomBytes()

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

The crypto module offers a way of encapsulating secure credentials to be used as part of a secure HTTPS net or HTTP connection.

It also offers a set of wrappers for OpenSSL's hash, hmac, cipher, decipher, sign, and verify methods.

crypto.createCredentials(details)

Creates a credentials object, with the optional details being a dictionary with keys:

- `pfx` : A string or buffer holding the PFX- or PKCS12- encoded private key, certificate and CA certificates.
- `key` : A string holding the PEM encoded private key.
- `passphrase` : A string of passphrase for the private key or pfx.
- `cert` : A string holding the PEM-encoded certificate.
- `ca` : Either a string or list of strings of PEM-encoded CA certificates to trust.
- `crl` : Either a string or list of strings of PEM-encoded CRLs (Certificate Revocation List).
- `ciphers`: A string describing the ciphers to use or exclude. Consult http://www.openssl.org/docs/apps/ciphers.html#CIPHER_LIST_FORMAT for details on the format.

If no 'ca' details are given, then node.js will use the default publicly trusted list of CAs as given in

http://mxr.mozilla.org/mozilla/source/security/nss/lib/ckfw/builtins/certdata.txt.

crypto.getCiphers()

Function added from Node.js version 0.10.18.

Returns an array with the names of the supported ciphers.

Example:

```
var ciphers = crypto.getCiphers();
console.log(ciphers); // ['AES-128-CBC', 'AES-128-CBC-HMAC-SHA1', ...]
```

crypto.getHashes()

Function added from Node.js version 0.10.18.

Returns an array with the names of the supported hash algorithms.

Example:

```
var hashes = crypto.getHashes();
console.log(hashes); // ['sha', 'sha1', 'sha1WithRSAEncryption', ...]
```

crypto.createHash(algorithm)

Creates and returns a hash object, a cryptographic hash with the given algorithm that can be used to generate hash digests.

`algorithm` is dependent on the available algorithms supported by the version of OpenSSL on the platform. Examples are `'sha1'`, `'md5'`, `'sha256'`, `'sha512'`, etc. On recent releases, `openssl list-message-digest-algorithms` will display the available digest algorithms. You can also use crypto.getHashes() to list the available digest algorithms.

Example: this example takes the sha1 sum of a file

```
var filename = process.argv[2];
var crypto = require('crypto');
var fs = require('fs');

var shasum = crypto.createHash('sha1');

var s = fs.ReadStream(filename);
s.on('data', function(d) {
  shasum.update(d);
});

s.on('end', function() {
  var d = shasum.digest('hex');
  console.log(d + '  ' + filename);
});
```

Class: Hash

The class for creating hash digests of data.

Returned by [`crypto.createHash(algorithm)`][].

hash.update(data, [input_encoding])

Updates the hash content with the given `data`, the encoding of which is given in `input_encoding` and can be `'utf8'`, `'ascii'`, or `'binary'`. If no encoding is provided, then a buffer is expected.

This can be called many times with new data as it is streamed.

hash.digest([encoding])

Calculates the digest of all of the passed data to be hashed. The `encoding` can be `'hex'`, `'binary'`, or `'base64'`. If no encoding is provided, then a buffer is returned.

Note: The `hash` object cannot be used after a `digest()` method has been called.

crypto.createHmac(algorithm, key)

Creates and returns an hmac object, a cryptographic hmac with the given algorithm and key.

`algorithm` is dependent on the available algorithms supported by OpenSSL. See [`crypto.createHash(algorithm)`][] above. `key` is the hmac key to be used.

Class: Hmac

Class for creating cryptographic hmac content.

Returned by [`crypto.createHmac`][].

hmac.update(data)

Updates the hmac content with the given `data`. This can be called many times with new data as it is streamed.

hmac.digest([encoding])

Calculates the digest of all of the passed data to the hmac. The `encoding` can be `'hex'`, `'binary'`, or `'base64'`. If no encoding is provided, then a buffer is returned.

Note: The `hmac` object can not be used after a `digest()` method has been called.

crypto.createCipher(algorithm, password)

Creates and returns a cipher object, with the given algorithm and password.

`algorithm` is dependent on OpenSSL. Examples are `'aes192'`, etc. On recent releases, `openssl list-cipher-algorithms` will display the available cipher algorithms. You can also use crypto.getCiphers() to list the available cipher algorithms.

`password` is used to derive key and iv, which must be a `'binary'`-encoded string or a buffer.

crypto.createCipheriv(algorithm, key, iv)

Creates and returns a cipher object, with the given algorithm, key, and iv.

`algorithm` is the same as the argument to `createCipher()`. `key` is the raw key used by the algorithm. `iv` is an initialization vector.

`key` and `iv` must be `'binary'`-encoded strings or buffers.

Class: Cipher

Class for encrypting data.

Returned by [`crypto.createCipher`][] and [`crypto.createCipheriv`][].

cipher.update(data, [input_encoding], [output_encoding])

Updates the cipher with `data`, the encoding of which is given in `input_encoding` and can be `'utf8'`, `'ascii'`, or `'binary'`. If no encoding is provided, then a buffer is expected.

The `output_encoding` specifies the output format of the enciphered data and can be `'binary'`, `'base64'`, or `'hex'`. If no encoding is provided, then a buffer is returned.

Returns the enciphered contents and can be called many times with new data as it is streamed.

cipher.final([output_encoding])

Returns any remaining enciphered contents, with `output_encoding` being one of: `'binary'`, `'base64'`, or `'hex'`. If no encoding is provided, then a buffer is returned.

Note: The `cipher` object can not be used after a `final()` method has been called.

cipher.setAutoPadding(auto_padding=true)

You can disable automatic padding of the input data to block size. If `auto_padding` is false, the length of the entire input data must be a multiple of the cipher's block size or `final` will fail. Useful for non-standard padding, for example, using `0x0` instead of PKCS padding.You must call this before `cipher.final`.

crypto.createDecipher(algorithm, password)

Creates and returns a decipher object, with the given algorithm and key. This is the mirror of the [crypto.createCipher()][] above.

crypto.createDecipheriv(algorithm, key, iv)

Creates and returns a decipher object, with the given algorithm, key, and iv. This is the mirror of the [crypto.createCipheriv()][] above.

Class: Decipher

Class for decrypting data.

Returned by [`crypto.createDecipher`][] and [`crypto.createDecipheriv`][].

decipher.update(data, [input_encoding], [output_encoding])

Updates the decipher with `data`, which is encoded in `'binary'`, `'base64'`, or `'hex'`. If no encoding is provided, then a buffer is expected.

The `output_decoding` specifies in what format to return the deciphered plaintext: `'binary'`, `'ascii'`, or `'utf8'`. If no encoding is provided, then a buffer is returned.

decipher.final([output_encoding])

Returns any remaining plaintext, which is deciphered, with `output_encoding` being one of: `'binary'`, `'ascii'`, or `'utf8'`. If no encoding is provided, then a buffer is returned.

Note: The `decipher` object can not be used after a `final()` method has been called.

decipher.setAutoPadding(auto_padding=true)

You can disable auto padding if the data has been encrypted without standard block padding to prevent [`decipher.final`][] from checking and removing it. Can only work if the input data's length is a multiple of the cipher's block size. You must call this before streaming data to [`crypto.decipher.update`][].

crypto.createSign(algorithm)

Creates and returns a signing object, with the given algorithm. On recent OpenSSL releases, `openssl list-public-key-algorithms` will display the available signing algorithms. Example is `'RSA-SHA256'`.

Class: Sign

Class for generating signatures.

Returned by [`crypto.createSign`][].

sign.update(data)

Updates the sign object with data. This can be called many times with new data as it is streamed.

sign.sign(private_key, [output_format])

Calculates the signature on all the updated data passed through the sign. `private_key` is a string containing the PEM-encoded private key for signing.

Returns the signature in `output_format`, which can be `'binary'`, `'hex'`, or `'base64'`. If no encoding is provided, then a buffer is returned.

Note: The `sign` object can not be used after a `sign()` method has been called.

crypto.createVerify(algorithm)

Creates and returns a verification object, with the given algorithm. This is the mirror of the signing object above.

Class: Verify

Class for verifying signatures.

Returned by [`crypto.createVerify`][].

verifier.update(data)

Updates the verifier object with data. This can be called many times with new data as it is streamed.

verifier.verify(object, signature, [signature_format])

Verifies the signed data by using the `object` and `signature`. `object` is a string containing a PEM-encoded object, which can be one of RSA public key, DSA public key, or X.509 certificate. `signature` is the previously calculated signature for the data, in the `signature_format`, which can be `'binary'`, `'hex'`, or `'base64'`. If no encoding is specified, then a buffer is expected.

Returns true or false depending on the validity of the signature for the data and public key.

Note: The `verifier` object can not be used after a `verify()` method has been called.

crypto.createDiffieHellman(prime_length)

Creates a Diffie-Hellman key exchange object and generates a prime of the given bit length. The generator used is `2`.

crypto.createDiffieHellman(prime, [encoding])

Creates a Diffie-Hellman key exchange object using the supplied prime. The generator used is `2`. Encoding can be `'binary'`, `'hex'`, or `'base64'`. If no encoding is specified, then a buffer is expected.

Class: DiffieHellman

The class for creating Diffie-Hellman key exchanges.

Returned by [`crypto.createDiffieHellman`][].

diffieHellman.generateKeys([encoding])

Generates private and public Diffie-Hellman key values and returns the public key in the specified encoding. This key should be transferred to the other party. Encoding can be `'binary'`, `'hex'`, or `'base64'`. If no encoding is provided, then a buffer is returned.

diffieHellman.computeSecret(other_public_key, [input_encoding], [output_encoding])

Computes the shared secret using `other_public_key` as the other party's public key and returns the computed shared secret. Supplied key is interpreted using specified `input_encoding`, and secret is encoded using specified `output_encoding`. Encodings can be `'binary'`, `'hex'`, or `'base64'`. If the input encoding is not provided, then a buffer is expected.

If no output encoding is given, then a buffer is returned.

diffieHellman.getPrime([encoding])

Returns the Diffie-Hellman prime in the specified encoding, which can be `'binary'`, `'hex'`, or `'base64'`. If no encoding is provided, then a buffer is returned.

diffieHellman.getGenerator([encoding])

Returns the Diffie-Hellman prime in the specified encoding, which can be `'binary'`, `'hex'`, or `'base64'`. If no encoding is provided, then a buffer is returned.

diffieHellman.getPublicKey([encoding])

Returns the Diffie-Hellman public key in the specified encoding, which can be `'binary'`, `'hex'`, or `'base64'`. If no encoding is provided, then a buffer is returned.

diffieHellman.getPrivateKey([encoding])

Returns the Diffie-Hellman private key in the specified encoding, which can be `'binary'`, `'hex'`, or `'base64'`. If no encoding is provided, then a buffer is returned.

diffieHellman.setPublicKey(public_key, [encoding])

Sets the Diffie-Hellman public key. Key encoding can be `'binary'`, `'hex'`, or `'base64'`. If no encoding is provided, then a buffer is expected.

diffieHellman.setPrivateKey(private_key, [encoding])

Sets the Diffie-Hellman private key. Key encoding can be `'binary'`, `'hex'` or `'base64'`. If no encoding is provided, then a buffer is expected.

crypto.getDiffieHellman(group_name)

Creates a predefined Diffie-Hellman key exchange object. The supported groups are: `'modp1'`, `'modp2'`, `'modp5'` (defined in RFC 2412), `'modp14'`, `'modp15'`, `'modp16'`, `'modp17'`, and `'modp18'` (defined in RFC 3526).

The returned object mimics the interface of objects created by crypto.createDiffieHellman() above, but will not allow to change the keys (with diffieHellman.setPublicKey() for example). The advantage of using this routine is that the parties don't have to generate or exchange group modulus beforehand, saving both processor and communication time.

Example (obtaining a shared secret):

```
var crypto = require('crypto');
var alice = crypto.getDiffieHellman('modp5');
var bob = crypto.getDiffieHellman('modp5');

alice.generateKeys();
bob.generateKeys();

var alice_secret = alice.computeSecret(bob.getPublicKey(), null, 'hex');
var bob_secret = bob.computeSecret(alice.getPublicKey(), null, 'hex');

/* alice_secret and bob_secret should be the same */
console.log(alice_secret == bob_secret);
```

crypto.pbkdf2Sync(password, salt, iterations, keylen)

Synchronous PBKDF2 applies pseudo-random function HMAC-SHA1 to derive a key of given length from the given password, salt, and iterations. Returns derivedKey or throws error.

crypto.randomBytes(size)

Generates cryptographically strong pseudo-random data. Usage:

```
// sync
try {
  var buf = crypto.randomBytes(256);
  console.log('Have %d bytes of random data: %s', buf.length, buf);
} catch (ex) {
  // handle error
}
```

DNS

1. dns.lookup(domain, [family], callback)

Standard Node.js Functions That Are Not Supported

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

- dns.resolve(domain, [rrtype], callback)
- dns.resolve4(domain, callback)
- dns.resolve6(domain, callback)
- dns.resolveMx(domain, callback)
- dns.resolveTxt(domain, callback)
- dns.resolveSrv(domain, callback)
- dns.resolveNs(domain, callback)
- dns.resolveCname(domain, callback)
- dns.reverse(ip, callback)
- Error codes

Use `require('dns')` to access this module. LineRate Scripting does not currently support DNS resolution. The stub that is present will allow HTTP requests by IP address to work.

dns.lookup(domain, [family], callback)

If `domain` is a string representing an IPv4 or IPv6 address, then `callback` is called with `domain` as an argument. Otherwise, an error is thrown.

This example prints "Result: 1.2.3.4":

```
var dns = require('dns');
dns.lookup('1.2.3.4', function onLookup(result) {
  console.log('Result:', result);
});
```

This method is not useful, and is provided for compatibility with modules that take domain names or IP addresses as arguments.

Events

Many objects emit events: a `net.Server` emits an event each time a peer connects to it, an `fs.readStream` emits an event when the file is opened, and so on. All objects that emit events are instances of `events.EventEmitter`. You can access this module by doing: `require("events");`

Typically, event names are represented by a camel-cased string, however, there aren't any strict restrictions on that, as any string will be accepted.

Functions can then be attached to objects, to be executed when an event is emitted. These functions are called *listeners*. When the event is emitted, the EventEmitter class invokes all listeners with the Listener Signature arguments included in the description of each event. For an example, see new.listener.

Class: events.EventEmitter

To access the EventEmitter class, `require('events').EventEmitter`.

When an `EventEmitter` instance experiences an error, the typical action is to emit an `'error'` event. Error events are treated as a special case in node. If there is no listener for it, then the default action is to trigger an uncaught exception. See uncaughtException.

All EventEmitters emit the event `'newListener'` when new listeners are added.

emitter.addListener(event, listener) or

emitter.on(event, listener)

Adds a listener to the end of the listeners array for the specified event.

```
server.on('connection', function (stream) {
  console.log('someone connected!');
});
```

emitter.once(event, listener)

Adds a **one-time** listener for the event. This listener is invoked only the next time the event is fired, after which it is removed.

```
server.once('connection', function (stream) {
  console.log('Ah, we have our first user!');
});
```

emitter.removeListener(event, listener)

Removes a listener from the listener array for the specified event. **Caution**: Changes array indices in the listener array behind the listener.

```
var callback = function(stream) {
  console.log('someone connected!');
};
server.on('connection', callback);
// ...
server.removeListener('connection', callback);
```

emitter.removeAllListeners([event])

Removes all listeners or those of the specified event.

Note that this will **invalidate** any arrays that have previously been returned by emitter.listeners(event).

emitter.setMaxListeners(n)

By default, EventEmitters will print a warning if more than 10 listeners are added for a particular event. This is a useful default that helps finding memory leaks. Obviously not all EventEmitters should be limited to 10. This function allows that to be increased. Set to zero for unlimited.

emitter.listeners(event)

Returns an array of listeners for the specified event.

```
server.on('connection', function (stream) {
  console.log('someone connected!');
});
console.log(util.inspect(server.listeners('connection'))); // [ [Function] ]
```

This array **may** be a mutable reference to the same underlying list of listeners that is used by the event subsystem. However, certain actions (specifically, removeAllListeners) will invalidate this reference.

If you would like to get a copy of the listeners at a specific point in time that is guaranteed not to change, make a copy, for example by doing `emitter.listeners(event).slice(0)`.

In a future release, this behavior **may** change to always return a copy, for consistency. In your scripts, do not rely on being able to modify the EventEmitter listeners using array methods. Always use the 'on' method to add new listeners.

emitter.emit(event, [arg1], [arg2], [...])

Executes each of the listeners in order with the supplied arguments.

Event: 'newListener'

Listener Signature: `function(event, listener) {}`

Invoked when a new listener is registered on this object. `event` is the name of the event, and `listener` is the function that was registered to listen for `event`.

This event is emitted any time someone adds a new listener. For example:

```
var util = require('util');
var EventEmitter = require('events').EventEmitter;
function MyClass() {}
util.inherits(MyClass, EventEmitter);

var myInstance = new MyClass();
myInstance.on('newListener', function newListenerRegistered(event, listener) {
 console.log('There is a new listener for event ' + event + ':', listener);
 });
myInstance.on('foo', function onEventFoo(bar) {
 console.log('Foo happened, bar:', bar);
 });
setTimeout(function () { myInstance.emit('foo', 'bar'); }, 5000);
```

In this example, the newListenerRegistered() function is listening for newListener events. When a listener is registered for the event 'foo', the newListener event fires, so newListenerRegistered() is invoked. According to the Listener Signature, the first argument is the event that is being listened for ('foo'). The second argument is the listener (the function onEventFoo()).

Then, 5 seconds later, the event foo is actually emitted from myInstance, so the listener onEventFoo() is invoked with the argument bar.

Executing JavaScript

Standard Node.js Functions That Are Not Supported

The following items from the vm module in the standard Node.js API are not supported in LineRate Scripting:

- vm.runInNewContext()
- vm.runInContext()
- vm.createContext()
- vm.createScript()
- Class: Script
- script.runInThisContext()
- script.runInNewContext()

You can access this module with:

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

JavaScript code can be compiled and run immediately.

Caveats

The `vm` module has many known issues and edge cases. If you run into issues or unexpected behavior, please consult the open issues on GitHub. Some of the biggest problems are described below.

Globals

Properties of the global object, like `Array` and `String`, have different values inside of a context. This means that common expressions like `[] instanceof Array` or `Object.getPrototypeOf([]) === Array.prototype` may not produce expected results when used inside of scripts evaluated via the `vm` module.

Some of these problems have known workarounds listed in the issues for `vm` on GitHub. For example, `Array.isArray` works around the example problem with `Array`.

vm.runInThisContext(code, [filename])

`vm.runInThisContext()` compiles `code`, runs it, and returns the result. Running code does not have access to local scope. `filename` is optional; it's used only in stack traces.

Example of using `vm.runInThisContext` and `eval` to run the same code:

```
var localVar = 123,
    usingscript, evaled,
    vm = require('vm');

usingscript = vm.runInThisContext('localVar = 1;',
  'myfile.vm');
console.log('localVar: ' + localVar + ', usingscript: ' +
  usingscript);
evaled = eval('localVar = 1;');
console.log('localVar: ' + localVar + ', evaled: ' +
  evaled);

// localVar: 123, usingscript: 1
// localVar: 1, evaled: 1
```

`vm.runInThisContext` does not have access to the local scope, so `localVar` is unchanged. `eval` does have access to the local scope, so `localVar` is changed.

In case of syntax error in `code`, `vm.runInThisContext` emits the syntax error to stderr and throws an exception.

File System

Standard Node.js Functions That Are Not Supported

Note: Currently, only the synchronous forms of file system methods are supported.

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

- fs.rename(oldPath, newPath, [callback])
- fs.truncate(fd, len, [callback])
- fs.stat(path, [callback])
- fs.lstat(path, [callback])
- fs.fstat(fd, [callback])
- fs.readlink(path, [callback])
- fs.realpath(path, [cache], callback)
- fs.unlink(path, [callback])
- fs.close(fd, [callback])
- fs.open(path, flags, [mode], [callback])
- fs.write(fd, buffer, offset, length, position, [callback])
- fs.read(fd, buffer, offset, length, position, [callback])
- fs.readFile(filename, [encoding], [callback])
- fs.writeFile(filename, data, [encoding], [callback])

- fs.appendFile(filename, data, encoding='utf8', [callback])
- fs.exists(path, [callback])

Note: The (2) designation lets you know that more information is available in section 2 of the manpage for that item. When working in LineRate, you can go into bash to view the manpage content specific to the system.

```
LROS# bash
$ man 2 <item>
```

File I/O is provided by simple wrappers around standard POSIX functions. To use this module, do `require('fs')`.

For the synchronous form, any exceptions are immediately thrown. You can use try/catch to handle exceptions or allow them to bubble up.

Below is an example of the synchronous version:

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

fs.unlinkSync('/tmp/hello')
console.log('successfully deleted /tmp/hello');
```

The synchronous methods block the entire process until they complete--halting all connections.

Relative path to filename can be used, remember however that this path will be relative to `process.cwd()`.

fs.renameSync(oldPath, newPath)

Synchronous rename(2).

fs.truncateSync(fd, len)

Synchronous ftruncate(2).

fs.statSync(path)

Synchronous stat(2). Returns an instance of [`fs.Stats`][].

fs.lstatSync(path)

Synchronous lstat(2). Returns an instance of `[`fs.Stats`][].

fs.fstatSync(fd)

Synchronous fstat(2). Returns an instance of [`fs.Stats`][].

fs.readlinkSync(path)

Synchronous readlink(2). Returns the symbolic link's string value.

fs.realpathSync(path, [cache])

Synchronous realpath(2). Returns the resolved path. May use `process.cwd` to resolve relative paths. `cache` is an object literal of mapped paths that can be used to force a specific path resolution or avoid additional `fs.stat` calls for known real paths.

Example:

```
try {
  resolvedPath = fs.realpathSync('/etc/passwd', cache);
} catch (err) {
  //Handle error
}
```

fs.unlinkSync(path)

Synchronous unlink(2).

fs.closeSync(fd)

Synchronous close(2).

fs.readdirSync(path)

Reads the contents of a directory. Synchronous readdir(3). Returns an array of filenames excluding `'.'` and `'..'`.

fs.openSync(path, flags, [mode])

Synchronous open(2). `flags` can be:

- `'r'` - Open file for reading. An exception occurs if the file does not exist.
- `'r+'` - Open file for reading and writing. An exception occurs if the file does not exist.

- `'rs'` - Open file for reading in synchronous mode. Instructs the operating system to bypass the local file system cache.

  This is primarily useful for opening files on NFS mounts as it allows you to skip the potentially stale local cache. It has a very real impact on I/O performance so don't use this mode unless you need it.

- `'rs+'` - Open file for reading and writing, telling the OS to open it synchronously. See notes for `'rs'` about using this with caution.
- `'w'` - Open file for writing. The file is created (if it does not exist) or truncated (if it exists).
- `'wx'` - Like `'w'` but opens the file in exclusive mode.
- `'w+'` - Open file for reading and writing. The file is created (if it does not exist) or truncated (if it exists).
- `'wx+'` - Like `'w+'` but opens the file in exclusive mode.
- `'a'` - Open file for appending. The file is created if it does not exist.
- `'ax'` - Like `'a'` but opens the file in exclusive mode.
- `'a+'` - Open file for reading and appending. The file is created if it does not exist.
- `'ax+'` - Like `'a+'` but opens the file in exclusive mode.

`mode` defaults to `0666`.

Exclusive mode (`O_EXCL`) ensures that `path` is newly created. `fs.openSync()` fails if a file by that name already exists. On POSIX systems, symlinks are not followed. Exclusive mode may or may not work with network file systems.

fs.writeSync(fd, buffer, offset, length, position)

Write buffer to the file specified by `fd`. Returns the number of bytes written.

`offset` and `length` determine the part of the buffer to be written.

`position` refers to the offset from the beginning of the file where this data should be written. If `position` is `null`, the data will be written at the current position. See pwrite(2).

fs.readSync(fd, buffer, offset, length, position)

Read data from the file specified by `fd`. Returns the number of `bytesRead`.

- `buffer` is the buffer that the data will be written to.
- `offset` is offset within the buffer where writing will start.
- `length` is an integer specifying the number of bytes to read.
- `position` is an integer specifying where to begin reading from in the file. If `position` is `null`, data will be read from the current file position.

fs.readFileSync(filename, [encoding])

Synchronously reads the entire contents of a file. Returns the contents of the `filename`.

Example:

```
try {
  var passwdContents = fs.readFileSync('/etc/passwd');
} catch (err) {
  //Handle error
}
```

If `encoding` is specified then this function returns a string. Otherwise it returns a buffer.

fs.writeFileSync(filename, data, [encoding])

Synchronously writes data to a file, replacing the file if it already exists. `data` can be a string or a buffer. The `encoding` argument is ignored if `data` is a buffer. It defaults to `'utf8'`.

```
try {
  fs.writeFileSync('message.txt', 'Hello Node');
} catch (err) {
  //Handle error
}
```

fs.appendFileSync(filename, data, encoding='utf8')

Synchronously append data to a file, creating the file if it does not yet exist. `data` can be a string or a buffer. The `encoding` argument is ignored if `data` is a buffer.

```
try {
  fs.appendFileSync('message.txt', 'data to append');
} catch (err) {
  //Handle error
}
```

fs.existsSync(path)

Tests whether or not the given path exists by checking with the file system.

Class: fs.Stats

Objects returned from `fs.statSync()`, `fs.lstatSync()` and `fs.fstatSync()`.

- `stats.isFile()`
- `stats.isDirectory()`
- `stats.isBlockDevice()`
- `stats.isCharacterDevice()`
- `stats.isSymbolicLink()` (only valid with `fs.lstatSync()`)
- `stats.isFIFO()`
- `stats.isSocket()`

For a regular file `util.inspect(stats)` would return a string very similar to this:

```
{ dev: 2114,
  ino: 48064969,
  mode: 33188,
  nlink: 1,
  uid: 85,
  gid: 100,
  rdev: 0,
  size: 527,
  blksize: 4096,
  blocks: 8,
  atime: Mon, 10 Oct 2011 23:24:11 GMT,
  mtime: Mon, 10 Oct 2011 23:24:11 GMT,
  ctime: Mon, 10 Oct 2011 23:24:11 GMT }
```

Please note that `atime`, `mtime` and `ctime` are instances of Date object and to compare the values of these objects you should use appropriate methods. For most general uses getTime() will return the number of milliseconds elapsed since *1 January 1970 00:00:00 UTC* and this integer should be sufficient for any comparison, however there additional methods which can be used for displaying fuzzy information. More details can be found in the MDN JavaScript Reference page.

Global Objects

These objects are available in all modules. Some of these objects aren't actually in the global scope but in the module scope. This will be noted.

global

The global namespace object.

In browsers, the top-level scope is the global scope. That means that in browsers if you're in the global scope, `var something` will define a global variable. In LineRate Scripting, this is different. The top-level scope is not the global scope; `var something` inside a scripting tool module will be local to that module. Every proxy process has an independent global namespace object for each script.

process

The `process` object. See the process object section.

console

The `console` object. Used to print to `stdout` and `stderr`. See the console section.

Class: Buffer

The `buffer` function. Used to handle binary data. See the buffer section

require()

The `require` function. Use to require modules. See the Modules section. `require` isn't actually a global but rather local to each module.

require.resolve()

Uses the internal `require()` machinery to look up the location of a module, but rather than loading the module, just return the resolved filename.

require.cache

The `require.cache` object. Modules are cached in this object when they are required. By deleting a key value from this object, the next `require` will reload the module.

require.extensions

The `require.extensions` array. Instructs `require` on how to handle certain file extensions.

Process files with the extension `.sjs` as `.js`:

```
require.extensions['.sjs'] = require.extensions['.js'];
```

Write your own extension handler:

```
require.extensions['.sjs'] = function(module, filename) {
  var content = fs.readFileSync(filename, 'utf8');
  // Parse the file content and give to module.exports
  module.exports = content;
};
```

__scriptname (LineRate Extension)

A string that is the name of the configured script. Irrespective of how the script is configured, '__scriptname' is always defined.

Example: running `script foo` with the following source

```
console.log(__scriptname);
// foo
```

__filename

A string that is filename of the script being executed. This is the resolved absolute path of this script file. The value inside a module is the path to that module file. Note: when the script source is specified inline, '__filename' is not defined.

Example: running `source file example.js`

```
console.log(__filename);
// /home/linerate/data/scripting/example.js
```

`__filename` isn't actually a global but rather local to each module.

__dirname

A string that is the name of the directory that the currently executing script resides in.

Example: running `source file example.js`

```
console.log(__dirname);
// /home/linerate/data/scripting
```

`__dirname` isn't actually a global but rather local to each module.

module

The `module` object. A reference to 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.

See the module system documentation for more information.

exports

An object that is shared between all instances of the current module and made accessible through `require()`. `exports` is the same as the `module.exports` object. `exports` isn't actually a global but rather local to each module.

See the module system documentation for more information.

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

Schedule a one-shot timer. See the set timeout section for more information.

clearTimeout(timeoutId)

Stop a timer that was previously created with `setTimeout()`. See the clear timeout section for more information.

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

Schedule a repeated timer. See the set interval section for more information.

clearInterval(intervalId)

Stop a timer that was previously created with `setInterval()`. See the clear interval section for more information.

The timer functions described above are available in global scope. See the timers section for additional information.

HTTP

Standard Node.js Functions That Are Not Supported

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

- http.createServer()
- Class: http.Server
- Class: http.Agent
- http.global.agent
- Class: http.ClientRequest Event: socket
- Class: http.ClientRequest Event: connect
- Class: http.ClientRequest Event: upgrade
- Class: http.ClientRequest request.setTimeout(timeout, [callback])
- Class: http.ClientRequest request.setNoDelay([noDelay])
- Class: http.ClientRequest request.setSocketKeepAlive([enable], [initialDelay])
- Class: http.ClientResponse response.trailers
- Class: http.ClientResponse response.setEncoding([encoding])

To use the HTTP server and client, you must `require('http')`.

The HTTP interfaces are designed to support many features of the protocol which have been traditionally difficult to use. In particular, large, possibly chunk-encoded, messages. The interface is careful to never buffer entire requests or responses--you can stream data.

HTTP message headers are represented by an object like this:

```
{ 'content-length': '123',
  'content-type': 'text/plain',
  'connection': 'keep-alive',
  'accept': '*/*' }
```

Values are not modified.

To support the full spectrum of possible HTTP applications, the HTTP API is very low-level. It deals with stream handling and message parsing only. It parses a message into headers and body, but it does not parse the actual headers or the body.

http.STATUS_CODES

This object is a collection of all the standard HTTP response status codes, and the short description of each. For example, `http.STATUS_CODES[404] === 'Not Found'`.

http.createClient([port], [host])

This function is **deprecated**. Use http.request() instead. Constructs a new HTTP client. The `port` and `host` arguments refer to the server to be connected to.

Class: http.ServerRequest

The HTTP server or proxy, not the user, creates an object of this type internally and passes the object as the first argument to a `'request'` listener. If the object is created by a proxy's request event (such as the 'VirtualServer.request' or 'ForwardProxy.request' events), then the object starts paused and will automatically unpause when the first `data` listener is registered.

The `http.ServerRequest` class is an EventEmitter with the events described in the sections that follow.

Implements: Readable Stream

The request implements the Readable Stream interface.

Event: 'data'

Listener signature: `function(chunk) {}`

The system emits the `data` event when a piece of the message body is received. If an encoding has been set with `request.setEncoding()`, the `chunk` is a string. Otherwise, it's a Buffer.

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

Event: 'end'

Listener signature: `function() {}`

The system emits the `end` event exactly once for each request. After that, no more `'data'` events are emitted on the request.

Event: 'close'

Listener signature: `function() {}`

A `close` event indicates that the underlying connection was terminated improperly.

Just like `'end'`, this event occurs only once per request, and no more `'data'` events are emitted afterwards.

Note: The system can emit a `'close'` event after an `'end'` event, but not vice versa.

Event: 'response' (LineRate Extension)

This event is a LineRate extension.

Listener signature: `function(cliResp) {}`

The system emits the `response` event when this request receives a response. The `cliResp` is a http.ClientResponse. This `cliResp` is paused, and will automatically unpause when the first `data` listener is registered.

request.method

The request method as a string. Read-only. Example: `'GET'`, `'DELETE'`.

request.url

Request URL string. This contains only the URL that is present in the actual HTTP request. If the request is:

```
GET /status?name=ryan HTTP/1.1\r\n
Accept: text/plain\r\n
\r\n
```

Then `request.url` is:

```
'/status?name=ryan'
```

If you want to parse the URL into its parts, you can use `require('url').parse(request.url)`. Example:

```
require('url').parse('/status?name=ryan')
```

will return the following object:

```
{ href: 097Release_2.4/450Scripting_API_Reference_Guide/'/status?name=ryan',
  search: '?name=ryan',
  query: 'name=ryan',
  pathname: '/status' }
```

If you want to extract the params from the query string, you can use the `require('querystring').parse` function or pass `true` as the second argument to `require('url').parse`. Example:

```
require('url').parse('/status?name=ryan', true)
```

will return the following object:

```
{ href: 097Release_2.4/450Scripting_API_Reference_Guide/'/status?name=ryan',
  search: '?name=ryan',
  query: { name: 'ryan' },
  pathname: '/status' }
```

request.headers

Read-only.

request.trailers

Read-only; HTTP trailers (if present). Only populated after the `end` event.

request.httpVersion

The HTTP protocol version as a string. Read-only. Examples: `'1.1'`, `'1.0'`. Also `request.httpVersionMajor` is the first integer and `request.httpVersionMinor` is the second.

request.setEncoding([encoding])

Sets the encoding for the request body. See stream.setEncoding() for more information.

request.pause()

Pauses request from emitting events. Useful to throttle back an upload.

request.resume()

Resumes a paused request.

request.connection

The net.Socket object associated with the connection.

request.addHeader(name, value) (LineRate Extension)

This is a LineRate extension.

Adds an HTTP header with the provided name and value to the request.

If there is an existing header with the same name, it is not altered; another HTTP header with `name: value` is added. Throws an exception when input is invalid according to RFC 2616.

request.removeHeader(name) (LineRate Extension)

This is a LineRate extension.

Removes all HTTP headers with the provided name from the request.

request.resetTimeout(timeout) (LineRate Extension)

This is a LineRate extension.

Resets the timeout for responding to this request.

When one of the following occurs, a timeout starts with a default value of 10 seconds:

- A request arrives at a 'VirtualServer.request' or 'ForwardProxy.request' event listener.
- A response arrives at a 'http.ServerRequest.response' event listener.

If the timeout is not restarted or disabled within the timeout interval, a 504 Gateway Timeout message is written to the response. Every time one of the following occurs, the timeout is restarted for `timeout` seconds from now:

- Part of the response is written via http.ServerResponse.writeHead()
- Part of the response is written via http.ServerResponse.write()
- The timeout is modified by this method.

The timer is disabled when the request is handled by:

- Calling `'next()'` to pass to the next element in the datapath (see 'VirtualServer.request' or 'ForwardProxy.request' events). If there is a listener for the 'http.ServerRequest.response' event, the timer resumes when the response arrives.

- Forwarding the request to another proxy using the VirtualServer.moveRequest() or ForwardProxy.moveRequest() methods. If there is a request listener on the target proxy, a timeout starts in the target proxy's request listener.
- Completing the response writing with http.ServerResponse.end()
- Connecting an http.ClientResponse to the ServerResponse with http.ClientResponse.fastPipe()

If `timeout` is 0, the timer is disabled for this request, and a 504 Gateway Timeout is never automatically sent to the client.

The underlying socket will inherit its idle timeout from the virtual IP's keepalive-timeout. This function can raise the value of the idle timeout or disable it (for this connection only) ensuring the script's timeout will never be shorter than the connection's idle timeout. All new connections will continue to inherit the virtual IP's keepalive-timeout until reset from a script.

request.bindHeaders(cliReq) (LineRate Extension)

This is a LineRate extension.

Efficiently causes the `cliReq` to have the same HTTP headers as the request. Modifying headers on the request causes them to also be modified on the `cliReq`. You can only call `request.bindHeaders` before `cliReq` headers are finalized (for example, using http.ClientRequest.write()).

This method is faster than copying the headers from a http.ServerRequest manually.

request.fastPipe(cliReq, [options]) (LineRate Extension)

This is a LineRate extension.

Efficiently causes the HTTP body of the request to be written out as the body of the `cliReq`. This must be called before data has been read from the request or written to `cliReq`, and the request must not have been forwarded. A request can only be fastPiped once. After calling this method, no more modifications to the request or `cliReq` can be made. The request object no longer emits any events such as `'data'` or `'event'`.

This method is faster than receiving the http.ServerRequest's `'data'` event and writing the data to the `cliReq` manually, and faster than using stream.pipe().

The optional `options` argument is an object that can have these properties:

- `response`: When the response to `cliReq` arrives, automatically http.ClientResponse.bindHeaders() and http.ClientResponse.fastPipe() it to this http.ServerResponse, without actually firing the `response` event in the script. This is faster than capturing the `response` event and using bindHeaders() and fastPipe() manually.

Class: http.ServerResponse

The HTTP server, not the user, creates an object of this type internally and passes the object as the second parameter to a `'request'` event.

The http.ServerResponse class is an EventEmitter with the events described in the sections that follow.

Implements: Writable Stream

The request implements the Writable Stream interface.

Event: 'close'

Listener signature: `function() {}`

A `close` event indicates that the underlying connection was terminated improperly.

response.writeContinue()

Sends a HTTP/1.1 100 Continue message to the client, indicating that the request body should be sent. See the 'VirtualServer.checkContinue' and 'ForwardProxy.checkContinue' events.

response.writeHead(statusCode, [reasonPhrase], [headers])

Sends a response header to the request. The status code is a 3-digit HTTP status code, like `404`. The last argument, `headers`, is the response headers. Optionally, you can add a human-readable `reasonPhrase` as the second argument.

Example:

```
var body = 'hello world';
response.writeHead(200, {
  'Content-Length': body.length,
  'Content-Type': 'text/plain' });
```

You can only call method once on a message, and you must call it before `response.end()`.

If you call `response.write()` or `response.end()` before calling this, the implicit/mutable headers are calculated and call this function for you.

Note: Content-Length is given in bytes not characters. The above example works because the string `'hello world'` contains only single byte characters. If the body contains higher coded characters, you should use `Buffer.byteLength()` to determine the number of bytes in a given encoding.

response.statusCode

When using implicit headers (not calling `response.writeHead()` explicitly), this property controls the status code that is sent to the client when the headers get flushed.

Example:

```
response.statusCode = 404;
```

After response header was sent to the client, this property indicates the status code that was sent.

response.setHeader(name, value)

Sets a single header value for implicit headers. If this header already exists in the to-be-sent headers, its value is replaced. If you need to send multiple headers with the same name, use an array of strings here. Throws an exception when input is invalid according to RFC 2616.

Example:

```
response.setHeader("Content-Type", "text/html");
```

or

```
response.setHeader("Set-Cookie", ["type=ninja", "language=javascript"]);
```

response.sendDate

When true, the Date header is automatically generated and sent in the response if it is not already present in the headers. Defaults to true.

This should only be disabled for testing; HTTP requires the Date header in responses.

response.getHeader(name)

Reads out a header that's already been queued but not sent to the client. Note that the name is case insensitive. This can only be called before headers get implicitly flushed.

Example:

```
var contentType = response.getHeader('content-type');
```

response.removeHeader(name)

Removes a header that's queued for implicit sending.

Example:

```
response.removeHeader("Content-Encoding");
```

response.write(chunk, [encoding])

If this method is called and `response.writeHead()` has not been called, it switches to implicit header mode and flush the implicit headers.

This sends a chunk of the response body. This method may be called multiple times to provide successive parts of the body.

The `chunk` can be a string or a buffer. If `chunk` is a string, the second parameter specifies how to encode it into a byte stream. By default the `encoding` is `'utf8'`.

Note: This is the raw HTTP body and has nothing to do with higher-level multi-part body encodings that may be used.

The first time `response.write()` is called, it sends the buffered header information and the first body to the client. The second time `response.write()` is called, the system assumes you're going to be streaming data, and sends that separately. That is, the response is buffered up to the first chunk of body.

response.addTrailers(headers)

This method adds HTTP trailing headers (a header but at the end of the message) to the response.

Trailers are **only** be emitted if chunked encoding is used for the response. If it is not (for example, if the request was HTTP/1.0), they are silently discarded.

Note that HTTP requires the `Trailer` header to be sent if you intend to emit trailers, with a list of the header fields in its value. For example,

```
response.writeHead(200, { 'Content-Type': 'text/plain',
                          'Trailer': 'Content-MD5' });
response.write(fileData);
response.addTrailers({'Content-MD5': "7895bf4b8828b55ceaf47747b4bca667"});
response.end();
```

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

This method signals to the server that all of the response headers and body has been sent; that server should consider this message complete. The method, `response.end()`, **must** be called on each response.

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

http.request(options, callback)

The system maintains several connections per server to make HTTP requests. This function lets you transparently issue requests.

the `options` argument can be an object or a string. If `options` is a string, it is automatically parsed with url.parse().

Options:

- `host`: IP address of the server to issue the request to. Defaults to `'localhost'`.
- `hostname`: To support `url.parse()` `hostname` is preferred over `host`.
- `port`: Port of remote server. Defaults to 80.
- `localAddress`: Local interface to bind for network connections.
- `method`: A string specifying the HTTP request method. Defaults to `'GET'`.
- `path`: Request path. Defaults to `'/'`. Should include query string if any. For example, `'/index.html?page=12'`
- `headers`: An object containing request headers.
- `auth`: Basic authentication, that is, `'user:password'` to compute an Authorization header.

The `http.request()` function returns an instance of the `http.ClientRequest` class. The `ClientRequest` instance is a writable stream. If you need to upload a file with a POST request, then write to the `ClientRequest` object.

Example:

```
var options = {
  host: '208.187.128.24',
  port: 80,
  path: '/upload',
  method: 'POST'
};

var req = http.request(options, function(res) {
  console.log('STATUS: ' + res.statusCode);
  console.log('HEADERS: ' + JSON.stringify(res.headers));
  res.setEncoding('utf8');
  res.on('data', function (chunk) {
    console.log('BODY: ' + chunk);
  });
});

req.on('error', function(e) {
  console.log('problem with request: ' + e.message);
});

// write data to request body
req.write('data\n');
req.write('data\n');
req.end();
```

Note that in the example, `req.end()` was called. With `http.request()`, you must always call `req.end()` to signify that you're done with the request, even if there is no data being written to the request body.

If any error is encountered during the request (be that with DNS resolution, TCP-level errors, or actual HTTP parse errors), an `'error'` event is emitted on the returned request object.

Note the following special headers:

- Sending a 'Connection: keep-alive' notifies the system that the connection to the server should persist until the next request.
- Sending a 'Content-length' header disables the default chunked encoding.
- Sending an 'Expect' header immediately sends the request headers. Usually, when sending 'Expect: 100-continue', you should both set a timeout and listen for the `continue` event. See RFC2616 Section 8.2.3 for more information.
- Sending an Authorization header overrides using the `auth` option to compute basic authentication.

http.get(options, callback)

Because most requests are GET requests without bodies, the system provides this convenience method. The only difference between this method and `http.request()` is that it sets the method to GET and calls `req.end()` automatically.

Example:

```
http.get("208.187.128.24", function(res) {
  console.log("Got response: " + res.statusCode);
}).on('error', function(e) {
  console.log("Got error: " + e.message);
});
```

Class: http.ClientRequest

The system creates this object internally as a return from `http.request()`. It represents an *in-progress* request whose header has already been queued. The header is still mutable using the `setHeader(name, value)`, `getHeader(name)`, `removeHeader(name)` API. The actual header is sent along with the first data chunk or when closing the connection.

To get the response, add a listener for `'response'` to the request object. The request object emits the `'response'` event when the response headers have been received. The `'response'` event is executed with one argument, which is an instance of `http.ClientResponse`.

During the `'response'` event, you can add listeners to the response object, particularly to listen for the `'data'` event. Note that the `'response'` event is called before any part of the response body is received, so you do not need to be concerned about catching the first part of the body. As long as a listener for `'data'` is added during the `'response'` event, the entire body is caught.

```
// Good
request.on('response', function (response) {
  response.on('data', function (chunk) {
    console.log('BODY: ' + chunk);
  });
});

// Bad - misses all or part of the body
request.on('response', function (response) {
  setTimeout(function () {
    response.on('data', function (chunk) {
      console.log('BODY: ' + chunk);
    });
  }, 10);
});
```

The http.ClientRequest class is an EventEmitter with the events described in the sections that follow.

Implements: Writable Stream

The request implements the Writable Stream interface.

Event 'response'

Listener signature: `function(response) {}`

The system emits the `response` event when it receives a response to this request. This event is emitted only once. The `response` argument is an instance of `http.ClientResponse`.

Options:

- `host`: IP address of the server to issue the request to.
- `port`: Port of remote server.

Event: 'continue'

`function () { }`

Emitted when the server sends a '100 Continue' HTTP response, usually because the request contained 'Expect: 100-continue'. This is an instruction that the client should send the request body.

request.addHeader(name, value) (LineRate Extension)

This is a LineRate extension.

Adds an HTTP header with the provided name and value to the request.

If there is an existing header with the same name, it is not altered; another HTTP header with `name: value` is added. Throws an exception when input is invalid according to RFC 2616.

request.removeHeader(name) (LineRate Extension)

This is a LineRate extension.

Removes all HTTP headers with the provided name from the request.

request.setHeader(name, value) (LineRate Extension)

Sets a single header value. If this header already exists in the to-be-sent headers, its value is replaced. If you need to send multiple headers with the same name, use an array of strings here. Throws an exception when input is invalid according to RFC 2616.

Example:

```
response.setHeader("Content-Type", "text/html");
```

or

```
response.setHeader("Set-Cookie", ["type=ninja", "language=javascript"]);
```

request.write(chunk, [encoding])

Sends a chunk of the body. By calling this method many times, the user can stream a request body to a server. In that case, we suggest using the `['Transfer-Encoding', 'chunked']` header line when creating the request.

The `chunk` argument should be a Buffer or a string.

The `encoding` argument is optional and only applies when `chunk` is a string. Defaults to `'utf8'`.

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

Finishes sending the request. If any parts of the body are unsent, it flushes them to the stream. If the request is chunked, this sends the terminating `'0\r\n\r\n'`.

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

request.abort()

Aborts a request.

Class: http.ProxiedClientRequest (LineRate Extension)

The system creates this object internally and passes it as the third argument to the listener for 'VirtualServer.request' or 'ForwardProxy.request' events. It represents the request that will be proxied by the script to the back-end servers.

The `http.ProxiedClientRequest` is an instance of `http.ClientRequest` with the events described in the sections that follow.

Event 'newServerSelected'

Listener signature: `function(newServerName) {}`

The system emits the `newServerSelected` event in two conditions:

- When a proxied client request is sent out to a real server different from the one specified by `selectServer()`
- When no real server was chosen for the proxied client request.

This event is guaranteed to be emitted before the `response` event, and is emitted only once.

The `newServerName` argument is a string containing the real server name on which the proxied client request was sent out on.

Note: the `newServerSelected` event is emitted on proxied client request objects triggered by requests received on virtual servers only (as the third argument to the listener for 'VirtualServer.request'). This event is not applicable to forward proxies.

request.selectServer(serverName)

Selects a real server to be used to send out the proxied client request. You can select the real server until any part of the proxied client request is written out to the back-end server.

The `serverName` argument should be a string containing the real server name.

Note: the `selectServer()` method is available on proxied client request objects triggered by requests received on virtual servers only (as the third argument to the listener for 'VirtualServer.request'). This method is not applicable to forward proxies.

request.sendHeaders()

Sends the request headers if they have not already been sent. This is useful if you have intercepted a request that has an `Expect: 100-continue` header.

See 'VirtualServer.checkContinue' or 'ForwardProxy.checkContinue' events for more information.

Class: http.ClientResponse

This object is created when making a request with `http.request()`. It is passed to the `'response'` event of the request object.

The http.ClientResponse class is an EventEmitter with the events described in the sections that follow.

Implements: Readable Stream

The request implements the Readable Stream interface.

Event: 'data'

Listener signature: `function(chunk) {}`

The system emits the `data` event when it receives a piece of the message body.

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

Event: 'end'

Listener signature: `function() {}`

The system emits the `end` event exactly once for each message. After emitting the `end` event, no other events are emitted on the response.

Event: 'close'

Listener signature: `function(err) {}`

The `close` event indicates that the underlaying connection was terminated before the `end` event was emitted. See http.ServerRequest's `'close'` event for more information.

Event: 'aborted'

Listener signature: `function() {}`

The system emits the `aborted` event in the case of an early close of the TCP socket. In the event the backend server crashes or indicates it will send more data than it actually does, the `aborted` event will emit. This event will emit before the `end` event.

Note: This event is undocumented in Node.js but exists and its semantics remain unaltered in LineRate Scripting.

response.statusCode

The 3-digit HTTP response status code. For example, `404`.

response.httpVersion

The HTTP version of the connected-to server. Probably either `'1.1'` or `'1.0'`. The `response.httpVersionMajor` object is the first integer and `response.httpVersionMinor` object is the second.

response.headers

The response headers object.

response.pause()

Pauses response from emitting events. Useful to throttle back a download.

response.resume()

Resumes a paused response.

response.bindHeaders(servResp) (LineRate Extension)

This is a LineRate extension.

Efficiently causes the `servResp` to have the same HTTP headers as the response. Modifying headers on the response causes them to also be modified on the `servResp`. You can only call `response.bindHeaders` before `servResp` headers are finalized (for example, using http.ServerResponse.writeHead()).

This method is faster than copying the headers from a http.ServerResponse manually.

response.fastPipe(servResp) (LineRate Extension)

This is a LineRate extension.

Efficiently causes the HTTP body of the response to be written out as the body of the `servResp`. This must be called before data has been read from the response or written to `servResp`. A response can only be fastPiped once. After calling this method, no more modifications to the response or `servResp` can be made. The response object no longer emits any events such as `'data'` or `'event'`.

This method is faster than receiving the http.ClientResponse's `'data'` event and writing the data to the `servResp` manually, and faster than using stream.pipe().

LineRate

HTTPS

1. https.request(options, callback)
2. https.get(options, callback)

Standard Node.js Functions That Are Not Supported

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

- https.Server
- https.createServer(options, [requestListener])
- https.Agent
- https.globalAgent

HTTPS is the HTTP protocol over TLS/SSL. In LineRate, this is implemented as a separate module.

https.request(options, callback)

Makes a request to a secure web server. All options from http.request() are valid.

Example:

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

var options = {
  host: '208.187.128.24',
  hostname: 'google.com'
  port: 443,
  path: '/',
  method: 'GET'
};

var req = https.request(options, function(res) {
  console.log("statusCode: ", res.statusCode);
  console.log("headers: ", res.headers);

  res.on('data', function(d) {
    process.stdout.write(d);
  });
});
req.end();

req.on('error', function(e) {
  console.error(e);
});
```

The options argument has the following options:

- `host`: IP address of the server to issue the request to. Defaults to `'localhost'`.

- `hostname`: To support `url.parse()`, `hostname` is preferred over `host`. For peer certificate validation, `hostname` or host `header` with the domain name is required.
- `port`: Port of remote server. Defaults to 443.
- `method`: A string specifying the HTTP request method. Defaults to `'GET'`.
- `path`: Request path. Defaults to `'/'`. Should include query string if any. For example: `'/index.html?page=12'`
- `headers`: An object containing request headers. For peer certificate validation, `hostname` or host `header` with the domain in it is required.

Note: The `agent` option is not supported. As shown in the example below, set agent to false.

The following options for secure connections are also supported:

- `pfx`: A string or buffer holding the PFX- or PKCS12-encoded private key, certificate, and CA certificates to use for SSL. Default `null`.
- `key`: A string holding the PEM-encoded private key to use for SSL. Default `null`.
- `passphrase`: A string or passphrase for the private key or pfx. Default `null`.
- `cert`: A string holding the public x509 certificate to use. Default `null`.
- `ca`: An authority certificate or array of authority certificates to check the remote host against. Use an array for multiple certificates. If no `ca` details are given, then the system will use the default, publicly trusted list of CAs.
- `ciphers`: A string describing the ciphers to use or exclude. Consult http://www.openssl.org/docs/apps/ciphers.html#CIPHER_LIST_FORMAT for details on the format. Default setting: "HIGH:!ADH:!SSLv2:!PSK:!ECDH:!kEDH:!SRP:+AES:+3DES"
- `rejectUnauthorized`: This option is ignored. Check http.ClientResponse.connection.authorized is a boolean value which indicates if the client has verified by one of the supplied certificate authorities for the server. If http.ClientResponse.connection.authorized is false, then http.ClientResponse.connection.authorizationError is set to describe how authorization failed. Unauthorized connections are always accepted.

Example:

```
var options = {
  host: '208.187.128.24',
  hostname: 'google.com'
  port: 443,
  path: '/',
  method: 'GET',
  key: fs.readFileSync('test/fixtures/keys/agent2-key.pem'),
  cert: fs.readFileSync('test/fixtures/keys/agent2-cert.pem'),
  agent: false
};

var req = https.request(options, function(res) {
  ...
}
```

https.get(options, callback)

Like `http.get()` but for HTTPS.

Example:

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

https.get({ host: '208.187.128.24', path: '/', hostname: 'google.com' }, function(res) {
  console.log("statusCode: ", res.statusCode);
  console.log("headers: ", res.headers);

  res.on('data', function(d) {
    process.stdout.write(d);
  });

}).on('error', function(e) {
  console.error(e);
});
```

Modules

LineRate Scripting has a simple module loading system. Files and modules are in one-to-one correspondence. As an example, `foo.js` loads the module `circle.js` in the same directory.

The contents of `foo.js`:

```
var circle = require('./circle.js');
console.log( 'The area of a circle of radius 4 is '
           + circle.area(4));
```

The contents of `circle.js`:

```
var PI = Math.PI;

exports.area = function (r) {
  return PI * r * r;
};

exports.circumference = function (r) {
  return 2 * PI * r;
};
```

The module `circle.js` has exported the functions `area()` and `circumference()`. To export an object, add to the special `exports` object.

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`.

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.

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.

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);
```

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.

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.

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'` occurs 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.

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()`.

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.