LuaSocket 2.0
A Short Reference

Acknowledgments

LuaSocket, written by Diego Nehab, provides a comprehensive range of communications services. Full details available from http://www.cs.princeton.edu/~diego/professional/luasocket/home.html

This Short Reference update was done as a means of becoming familiar with LuaSocket, so has been edited from the perspective of a new-comer to Lua and LuaSocket.

Graham Henstridge

LuaSocket 2.0 Short Reference

Introduction

LuaSocket , written by Diego Nehab, provides a comprehensive range of communications services. Full details available from http://www.cs.princeton.edu/~diego/professional/luasocket/home.html

FTP Library

ftp.get (url)

Simple form of **get** has fixed functionality: it downloads the contents of a URL and returns it as a string or **nil** and error string

ftp.get { host = string, **sink = LTN12 sink,**
argument | path = string, [**user = string**,]
[**password = string**] [**command = string**,]
[**port = number**,] [**type = string**,]
[**step = LTN12 pump step**,] [**create = function**] **}**

The generic form allows more control:

- **host** is the server to connect to.
- **sink** is the simple LTN12 sink that will receive the downloaded data.
- **argument** or **path** give the target path to the resource in the server.
- **user**, **password** for authentication. Default "ftp:anonymous@anonymous.org".
- **command** ftp command to get data. Default "retr"
- **port** The port (default 21) for connection.
- **type** Transfer mode. Can take values "**i**" or "**a**". Defaults to server's default.
- **step** LTN12 pump step function to pass data from server to sink. Default LTN12 pump.step ().
- **create** An optional function to be used instead of socket.tcp when the communications socket is created.

Returns 1 or **nil** and an error string.

HTTP Library

http.request (url [, **body**] **)**

Downloads a **url** (a string). If a **body** string is provided, performs a POST method in the **url**. Otherwise, performs a GET in the **url.** Returns response body string, followed by the response status code, the response headers and the response status line, or **nil** and an error message.

http.request { url = string, [**sink = LTN12 sink**,]
[**method = string**,] [**headers = header table**,]
[**source = LTN12 source**],[**step = LTN12 pump step**,]
[**proxy = string**,] [**redirect = boolean**,]
[**create = function**] }

Performs any HTTP method and is LTN12 based.
If the first argument is instead a table, the most important fields are the url and the simple **LTN12 sink** that will receive the downloaded content. Any part of the url can be overridden by including the appropriate field in the request table. If authentication information is provided, the function uses the Basic Authentication Scheme (see note) to retrieve the document. If sink is nil, the function discards the downloaded data. The optional parameters are the following:

- **method** The HTTP request method. Defaults to "GET";
- **headers** Any additional HTTP headers to send with the request
- **source** simple LTN12 source to provide the request body. If there is a body, you need to provide an appropriate "content-length" request header field, or the function will attempt to send the body as "chunked" (something few servers support). Defaults to the empty source;
- **step** LTN12 pump step function used to move data. Default LTN12 pump.step().
- **proxy** The URL of a proxy server to use. Defaults to no proxy.
- **redirect** Set to false to prevent the function from automatically following 301 or 302 server redirect messages.
- **create** A function instead of socket.tcp when communications socket created.

Returns a **1**, followed by the response status code, the response headers and the response status line, or **nil** and error string.

SMTP Library

smtp.message (mesgt)

Returns a simple LTN12 source that sends an SMTP message body, possibly multipart (arbitrarily deep). where **mesgt** has form (notice recursive structure):

mesgt = {
headers = header-table,
body = LTN12 source | string | multipart-mesgt }
multipart-mesgt = {
[**preamble = string**,]
[**1 =**] **mesgt**, [**2 =**] **mesgt**, ... [n **=**] **mesgt**,
[**epilogue = string**,] }

smtp.send { from = string, rcpt = string | string-table,
source = LTN12 source, [**user = string**,]
[**password = string**,] [**server = string**,]
[**port = number**,] [**domain = string**,]
[**step = LTN12 pump step**,] [**create = function**] }

Socket Library

socket.bind (address, **port** [, **backlog**] **)**

Creates and returns a TCP server object bound to a local **address** and **port**, ready to accept client connections. Optionally, user can also specify the **backlog** argument to the listen method (defaults to 32). Note: The server object returned will have the option "**reuseaddr**" set to **true**.

socket.connect (address, **port** [, **locaddr**, **locport**] **)**

Creates and returns a TCP client object connected to a remote host (**address**) at a given **port**. Optionally local address (**locaddr**) and port to bind (**locport**).

socket._DEBUG

This constant is set to **true** if the library was compiled with debug support.

socket.newtry (finalizer)

Creates and returns a clean try function that allows for cleanup before the exception is raised. **Finalizer** is a function that will be called before try throws the exception. It will be called in protected mode. The function returns your customized try function.
Note: This idea saved a lot of work with the implementation of protocols in LuaSocket:

socket.protect (func)

Converts a function **func** that throws exceptions into a safe function. This function only catches exceptions thrown by the **socket.try** and **socket.newtry** functions. It does not catch normal Lua errors. **Func** calls **socket.try** (or **socket.assert**, or error) to throw exceptions.
Returns an equivalent function that instead of throwing exceptions, returns **nil** followed by an error string.
Note: Beware that if **func** performs illegal operation that raises an error, the function will catch the error and return it as a string.

socket.select (recvt, sendt [, timeout])
Waits for a number of sockets to change status. **Recvt** is table of sockets to test for characters available for reading. Sockets in the **sendt** table are watched to see if it is OK to immediately write. **Timeout** is the max. time (seconds) to wait. A **nil**, negative or omitted timeout value allows function to block indefinitely. **Recvt** and **sendt** can also be empty tables or **nil**. Non-socket values (or values with non-numeric indices) in the arrays will be silently ignored.
Returns a table with sockets ready for reading, a table with the sockets ready for writing and an error message. The error message is "**timeout**" if a timeout condition was met and **nil** otherwise.
Notes: 1. bug in WinSock causes select to fail on non-blocking TCP sockets. May return a socket as writable even though the socket is not ready for sending.
2. Calling select with a server socket in the receive parameter before a call to accept does not guarantee accept will return immediately. Use the **settimeout** method or accept might block forever.
3. A closed socket passed to select will be ignored.

socket.sink (mode, socket)
Creates an LTN12 sink from a stream socket object. **Mode** defines the behavior of the sink. The following options are available:
"**http-chunked**"
sends data through socket after applying the chunked transfer coding, closing socket when done.
"**close-when-done**"
sends all received data through the socket, closing the socket when done;
"**keep-open**"
sends all received data through the socket, leaving it open when done.
Socket is the stream socket object used to send the data. The function returns a sink with the appropriate behavior.

socket.skip (d [, ret1, ret2 ... retN])
Drops first **d** arguments and returns the remaining (**retd +1** to **retN**) of **ret1** to **retN** arguments.

socket.sleep (time)
Freezes program execution for **time** in integer seconds.

socket.source (mode, socket [, length])
Creates an LTN12 source from a stream socket object. **Mode** defines the behavior of the source. The following options are available:
"**http-chunked**"
receives data from socket and removes chunked transfer coding before returning data
"**by-length**"
receives a fixed number of bytes from the socket. This mode requires the extra argument length;
"**until-closed**"
receives data from a socket until the other side closes the connection.
Socket is the stream socket object used to receive the data. Returns a source with the appropriate behavior.

socket.gettime ( )
Returns the time in seconds, relative to epoch.

socket.try (ret1 [, ret2 ... retN])
Throws an exception on error. The exception can only be caught by the protect function. **Ret1** to **retN** can be arbitrary arguments, usually the return values of a function call nested with **socket.try**.
Returns **ret1** to **retN** if **ret1** is not **nil**. Otherwise, it calls error passing **ret2**.

socket._VERSION
Constant string with LuaSocket version.

DSN Library

socket.dns.gethostname ( )
Returns standard host name for machine as a string.

socket.dns.tohostname (address)
Converts from IP address to host name. **Address** can be an IP address or host name. Returns host name and resolver table.

socket.dns.toip (address)
Converts from host name to IP address. **Address** can be an IP address or host name. Returns first IP address found for address and resolver table.

The name resolution functions return all information obtained from the resolver in a table of the form:

```
resolved = {
    name = canonic-name,
    alias = alias-list,          -- can be empty
    ip = ip-address-list  }
```

TCP Library

socket.tcp ( )
Creates and returns a TCP master object. A master object can be transformed into a server object with the method **listen** (after a call to bind) or into a client object with the method **connect**. The only other method supported by a master object is the **close** method.
Returns new master object or **nil** and an error message.

server:accept ( )
Waits for a remote connection on the server object and returns a client object representing that connection.
If a connection is successfully initiated, a client object is returned. If a timeout condition is met, the method returns nil followed by the error string 'timeout'. Other errors are reported by nil followed by a message describing the error.
Note: calling **socket.select** with a server object in **recvt** parameter before a call to accept does not guarantee accept will return immediately. Use **settimeout** method or accept might block until another client shows up.

master:bind (address, port)
Binds a master object to **address** and **port** on the local host. **Address** can be an IP address or a host name. **Port** must be an integer number. If address is '*', the system binds to all local interfaces using the INADDR_ANY constant. If port is **0**, the system automatically chooses an ephemeral port. Returns 1or nil and error message.
Note: The function **socket.bind** is a shortcut for creation of server sockets.

master:close ( )
client:close ( )
server:close ( )
Closes internal socket and local address to which object was bound is freed.

master:connect (address, port)
Attempts to connect a master object to a remote host, transforming it into a client object. Client objects support methods **send**, **receive**, **getsockname**, **getpeername**, **settimeout**, and **close**. **Address** can be an IP address or a host name. **Port** must be an integer number.
Returns 1 or **nil** and error string.
Notes: 1. The function **socket.connect** is a shortcut for creation of client sockets.
2. S**ettimeout** method affects behavior of connect, causing it to return an error on timeout. If that happens, you can still call **socket.select** with the socket in the **sendt** table. The socket will be writable when connection is established.

client:getpeername ( )
Returns information about the remote side of a connected client object. Returns a string with the IP address of the peer, followed by the port number that peer is using for the connection. On error, returns **nil**.

master:getsockname ( )
client:getsockname ( )
server:getsockname ( )
Returns the local address information associated to the object. The method returns a string with local IP address and a number with the port. In case of error, the method returns **nil**.
master:getstats ( )
client:getstats ( )
server:getstats ( )
Returns information on socket, useful for throttling of bandwidth. Returns number of bytes received, number sent, and age of socket object in seconds.
master:listen (backlog)
Specifies socket is willing to receive connections, transforming the object into a server object. Server objects support **accept**, **getsockname**, **setoption**, **settimeout**, and **close** methods. The parameter **backlog** specifies the number of client connections that can be queued waiting for service. If queue is full and another client attempts connection, the connection is refused. Returns **1** or **nil** and error message.
client:receive ([pattern [, prefix]])
Reads data from a client object, according to specified read **pattern**. Patterns follow Lua file I/O format. **Pattern** can be any of the following:
'***a**' Reads from socket until connection is closed. No end-of-line translation performed;
'***l**' Reads a line of text from socket. Line is terminated by LF character, optionally preceded by a CR character. The CR and LF characters are not included in the returned line. This is the default pattern.
number Causes the method to read a specified number of bytes from the socket.
Prefix is an optional string to be concatenated to the beginning of any received data before return.
Returns the received pattern or **nil,** an error message ('**closed**' or '**timeout**') and partial result of transmission.
client:send (data [, i [, j]])
Sends **data** through client object. Optional arguments **i** and **j** exactly like standard **string.sub**. Output is not buffered. For small strings, better to concatenate them in Lua. Returns the index of the last byte within **[i, j]** sent. Notice that, if **i** is 1 or absent, this is effectively the total number of bytes sent. On error, returns **nil**, and error message ('**closed**' or '**timeout**'), followed by the index of the last byte within [**i**, **j**] that has been sent.
client:setoption (option [, value])
server:setoption (option [, value])
Sets options for the TCP object. Options are only needed by low-level or time-critical applications.
Option is a string with the option name, and value depends on the option being set:
'**keepalive**': Set **true**, enables periodic transmission of messages on a connected socket. Should connected party fail to respond to these messages, the connection is considered broken and processes using socket are notified.
'**linger**': Controls action taken when unsent data are queued on a socket and **close** is performed. The **value** is a table with a boolean entry '**on**' and a numeric entry for the time interval 'timeout' in seconds. If the '**on**' field is set to **true**, the system will block the process on the close attempt until it is able to transmit the data or until '**timeout**' has passed. If '**on**' is **false** and a close is issued, the system will process the close in a manner that allows the process to continue as quickly as possible. Best set to zero;
'**reuseaddr**' Setting this option indicates that the rules used in validating addresses supplied in a call to bind should allow reuse of local addresses;
'**tcp-nodelay**' Setting this option to true disables the Nagle's algorithm for the connection.
Returns **1** on success, or **nil**.
master:setstats (received, sent, age)
client:setstats (received, sent, age)
server:setstats (received, sent, age)
Resets accounting information on the socket. **Received** is a number of bytes received. **Sent** is number of bytes sent. **Age** in seconds. Returns 1 on success or nil.
master:settimeout (value [, mode])
client:settimeout (value [, mode])
server:settimeout (value [, mode])
Changes the timeout values for the object. By default, all I/O operations are blocking - any call to the methods **send**, **receive**, and **accept** will block indefinitely. When a timeout is set and the specified time has elapsed, the affected methods give up and fail with an error code.
The amount of time to wait is specified as the **value** parameter, in seconds. There are two timeout **mode**s and can be used together:
'**b**' Block timeout. Specifies the upper limit on the amount of time LuaSocket can be blocked by the operating system while waiting for completion of any single I/O operation. This is the default mode;
'**t**' Total timeout. Specifies the upper limit on the amount of time a Lua script can block.
A **nil** and negative timeout value allows operations to block indefinitely.
Note: although timeout values have millisecond precision in LuaSocket, large blocks can cause I/O functions not to respect timeout values due to the time the library takes to transfer blocks to and from the OS and to and from the Lua interpreter. Also, function that accept host names and perform automatic name resolution might be blocked by the resolver for longer than the specified timeout value.
client:shutdown (mode)
Shuts down part of a full-duplex connection. **Mode** tells which way of the connection should be shut down:
"**both**" Disallow further sends and receives (default).
"**send**" Disallow further sends.
"**receive**" Disallow further receives.
Returns **1**.

UDP Library

socket.udp ( )
Creates and returns an unconnected UDP object. Unconnected objects support the sendto, receive, receivefrom, getsockname, setoption, settimeout, setpeername, setsockname, and close. The setpeername is used to connect the object.
In case of success, a new unconnected UDP object returned. In case of error, nil is returned, followed by an error message.
connected:close()
unconnected:close()
Closes a UDP object. The internal socket used by the object is closed and the local address to which the object was bound is made available to other applications. No further operations (except for further calls to the close method) are allowed on a closed socket.
Note: It is important to close all used sockets once they are not needed, since, in many systems, each socket uses a file descriptor, which are limited system

resources. Garbage-collected objects are automatically closed before destruction, though.

connected:getpeername ( )

Returns the IP address and port number of the peer associated with a connected UDP object.

connected:getsockname ( )

unconnected:getsockname ( )

Returns a string with local IP address and port number, or on error, **nil**. UDP sockets are not bound to any address until **setsockname** or **sendto** method is called for first time (in which case it is bound to an ephemeral port and the wild-card address).

connected:receive ([size])

unconnected:receive ([size])

Returns a received datagram from the UDP object or **nil** and an error string ("**timeout**"). If the UDP object is connected, only datagrams coming from the peer are accepted. Otherwise, returned datagram can come from any host. The optional **size** (max and default 8192 bytes) specifies maximum size of datagram retrieved.

unconnected:receivefrom ([size])

Same as receive method, except it returns the IP address and port as extra return values.

connected:send (datagram)

Sends a **datagram** string to the UDP peer of a connected object. Non- blocking and can fail only if underlying transport layer refuses to send. The max. **datagram** size for UDP is 64K minus IP layer overhead. Datagrams larger than the link layer packet size will be fragmented, which may deteriorate performance. Returns **1** or **nil** and error message.

unconnected:sendto (datagram, ip, port)

Similar to he send method, sends a **datagram** string to the specified recipient's **IP** address and **port** number **Ip** is the IP address of the recipient. Host names are not allowed for performance reasons.

connected:setpeername ('*')

unconnected:setpeername (address, port)

Changes the peer of a UDP object. This method turns an unconnected UDP object into a connected UDP object or vice versa. For connected objects, outgoing datagrams will be sent to the specified peer, and datagrams received from other peers will be discarded by the OS. Connected UDP objects must use the **send** and **receive** methods instead of **sendto** and **receivefrom**. **Address** can be an IP address or a host name and **port** is the port number. If **address** is '*' and the object is connected, the peer association is removed and the object becomes an unconnected object. In that case, the port argument is ignored. Returns 1 or **nil** and error message.

unconnected:setsockname (address, port)

Binds the UDP object to a local address. **Address** can be an IP address or a host name. If address is '*' the system binds to all local interfaces using the constant INADDR_ANY. If **port** is **0**, the system chooses an ephemeral port. Returns **1** or **nil** and error message.

Note: This method can only be called before any datagram is sent through the UDP object, and only once. Otherwise, the system automatically binds the object to all local interfaces and chooses an ephemeral port as soon as the first datagram is sent. After the local **address** is set, either automatically by the system or explicitly by **setsockname**, it cannot be changed.

connected:setoption (option [, value])

unconnected:setoption (option [, value])

Sets **option**s for the UDP object. Options are only needed by low-level or time-critical applications. Option is a string with the option name, and value depends on the option being set:

'**dontroute**' Setting this option to **true** indicates that outgoing messages should bypass the standard routing facilities;

'**broadcast**' Setting this option to **true** requests permission to send broadcast datagrams on the socket.

Returns 1 or **nil** and error message.

connected:settimeout (value)

unconnected:settimeout(value)

Changes the timeout values for the object. By default, the **receive** and **receivefrom** operations are blocking. When a timeout is set (**value** in seconds) and the specified amount of time has elapsed, the affected methods give up and fail with an error code. The **nil** or negative **value** timeout value allows operations to block indefinitely.

MIME Library

mime.normalize ([marker])

Converts most common end-of-line markers to a specific given **marker**.

mime.decode (encoding)

Returns a filter that decodes data from a given transfer content **encoding** (= **"base64"** or **"quoted-printable")**

mime.encode (encoding [, mode])

Returns a filter that encodes data according to a given transfer content **encoding** (= **"base64"** or **"quoted-printable")**. For quoted-printable, **mode** can be "**text**" (default) or "**binary**".

mime.stuff ( )

Creates and returns a filter that performs stuffing of smtp messages. smtp.send() applies automatically.

mime.wrap(encoding [, length]

Returns a filter that sensibly breaks data into lines. **Encoding** can be **"text"**, **base64"** or **"quoted-printable"**. The **length** arg only applied to **"text"**, others use default of 76.

mime.b64 (s1 [, s2])

Low-level filter to perform base64 encoding. Returns encoded version of the largest prefix of **s1**..**s2** that can be encoded unambiguously (if **s2** is **nil**, padded with encoding of the remaining bytes of **s1**), AND remaining bytes of **s1**..**s2**, before encoding.

mime.dot (m [, s])

Low-level filter for smtp stuffing to enable transmission of sequence "CRLF.CRLF". Used by **smtp.send( )**. Returns stuffed version of **s** and number of CRLF.CRLF

mime.eol (last [, data, marker])

Performs end-of-line marker translation. Returns 2 values - translated version of **data** and the ASCII value of the last character of the chunk (if it was a candidate for line break) or 0 otherwise (which then becomes the **last** value in the next call). **Marker** gives the new end-of-line marker (default: CRLF).

mime.qp (d1 [, d2, marker])

Performs Quoted-Printable encoding on concatenation of **d1** and **d2**. Returns 2 values, encoded version of largest prefix of **d1**..**d2** that can be encoded unambiguously (if **d2** is **nil**, return is padded with the encoding of the remaining bytes of **d1**), and the remaining bytes of **d1**..**d2**, before encoding. Occurrences of CRLF are replaced by **marker** (default: CRLF).

mime.qpwrp (n [, B, length])

Breaks Quoted-Printable text into lines. Returns 2 values - B broken into lines of at most **length** bytes (default: 76) and the number of bytes left in the last line (that becomes the **n** in the next call).

mime.unb64 (d1 [, d2])

Performs Base64 decoding on concatenation of **d1** and **d2**. Returns 2 values - decoded version of largest prefix of **d1**..**d2** that can be decoded unambiguously (or if **d2**

is **nil**, an empty string) and whatever couldn't be decoded.

mime.unqp (d1 [, d2])
Removes the Quoted-Printable transfer content encoding from concatenation of **d1** and **d2**. Returns 2 values - the decoded version of the largest prefix of **d1**..**d2** that can be decoded unambiguously (if **d2** is **nil**, augmented with the encoding of the remaining bytes of **d1**) and the remaining bytes of **d1**..**d2**, before decoding.

mime.wrp (n [, d1, length])
Break normalized text into lines with CRLF marker. Returns 2 values - a copy of **d1**, broken into lines of at most **length** bytes (default: 76) and the number of bytes left in last line that becomes the **n** on next call.

URL Library

An URL is defined by the following grammar:

<url> ::= [<scheme>:][//<authority>][/<path>][;<params>]
[?<query>][#<fragment>]
<authority> ::= [<userinfo>@]<host>[:<port>]
<userinfo> ::= <user>[:<password>]
<path> ::= {/}

url.absolute (base, relative)
Builds an absolute URL from a **base** URL string or a parsed URL table and a **relative** URL string. Returns a string with the absolute URL. See RFC 2396 for details.

url.build (parsed_url)
Rebuilds an URL from its parts. **Parsed_url** is a table with same components returned by **parse**. Lower level components, if specified, take precedence over high level components of the URL grammar. Returns a string with the built URL.

url.build_path (segments, unsafe)
Returns a string **<path>** component from a list of **** parts. Before composition, any reserved characters found in a segment are escaped into their protected form, so that the resulting path is a valid URL path component. **Segments** is a list of strings with the **** parts. If **unsafe** is anything but **nil**, reserved characters are left untouched.

url.escape (content)
Returns the URL escape content coding of **content** string. Each byte is encoded as **%xx** where **xx** is two byte hexadecimal representation of its integer value.

url.parse (url, default)
Parses **url** string into a table with its components. If the **default** table is present, it is used to store the parsed fields. Only fields present in the **url** are overwritten. Returns a table with all the **url** components (strings):

parsed_url = **{**
url, **scheme**, **authority**, **path**,
params, **query**, **fragment**, **userinfo**,
host, **port**, **user**, **password** **}**

url.parse_path (path)
Breaks **path** string (a **<path>** URL component) into all its **** parts. Removes escaping of reserved characters. Returns a table of parsed segments.

url.unescape (content)
Removes the URL escaping content coding from **content** string. Returns the decoded string.

LTN12 Library

The ltn12 library implements ideas described in LTN012, Filters Sources and Sinks. Please refer to the LTN for a deeper explanation of the functionality provided by this module.

Filters

ltn12.filter.chain (filter1, filter2 [, ... filterN])
Returns a filter that passes all data it receives through each of a series of given filters. **Filter1** to **filterN** are simple filters. Returns the chained filter.

ltn12.filter.cycle (low [, ctx, extra])
Returns a high-level filter that cycles though a low-level filter by passing it each chunk and updating a context between calls. **Low** is low-level filter to be cycled, **ctx** is initial context and **extra** is any extra argument the low-level filter might take. Returns the high-level filter.

Pumps

ltn12.pump.all (source, sink)
Pumps all data from a **source** to a **sink**.
If successful, the function returns **true**. In case of error, the function returns a **false** and an error message.

ltn12.pump.step (source, sink)
Pumps one chunk of data from a **source** to a **sink**.
If successful, the function returns a **true**. In case of error, the function returns a **false** and an error message.

Sinks

ltn12.sink.chain (filter, sink)
Creates and returns a new sink that passes data through **filter** before sending it to a given **sink**.

ltn12.sink.error (message)
Creates and returns a sink that aborts transmission with the error **message**.

ltn12.sink.file (handle, message)
Creates a sink that sends data to a file. If file **handle** is **nil**, **message** should give the reason for failure.
Returns a sink that sends all data to **handle** and closes file when done, or a sink that aborts the transmission with the error **message**.

ltn12.sink.null ( )
Returns a sink that ignores all data it receives.

ltn12.sink.simplify (sink)
Creates and returns a simple sink given a fancy **sink**.

ltn12.sink.table ([table])
Creates a sink that stores all chunks in a table. **Table** is used to hold the chunks. If **nil**, the function creates its own table. Returns the sink and the **table** of chunks.

Sources

ltn12.source.cat (source1 [, source2, ..., sourceN])
Creates a new source that produces the concatenation of the data produced by a number of sources.
Source1 to **sourceN** are the original sources.
The function returns the new source.

ltn12.source.chain (source, filter)
Creates a new **source** that passes data through a **filter** before returning it. Returns the new source.

ltn12.source.empty ( )
Returns an empty source.

ltn12.source.error (message)
Creates and returns a source that aborts transmission with the error **message**.

ltn12.source.file (handle, message)
Creates a source that produces the contents of a file. Handle is a file handle. If file **handle** is **nil**, **message** should give the reason for failure.
Returns a source that reads chunks of data from **handle**, closing the file when done, or a source that aborts the transmission with the error message

ltn12.source.simplify (source)
Returns a simple source given a fancy **source**.

ltn12.source.string (string)
Returns a source that produces the contents of a **string**, chunk by chunk.