Package 'nanonext'

January 22, 2023

Type Package

Title NNG (Nanomsg Next Gen) Lightweight Messaging Library

Version 0.7.3

Description R binding for NNG (Nanomsg Next Gen), a successor to ZeroMQ. NNG is a socket library providing high-performance scalability protocols, implementing a cross-platform standard for messaging and communications. Serves as a concurrency framework for building distributed applications, utilising 'aio' objects which resolve automatically upon completion of asynchronous operations.

License GPL (>= 3)

BugReports https://github.com/shikokuchuo/nanonext/issues

URL https://shikokuchuo.net/nanonext/, https://github.com/shikokuchuo/nanonext/

Encoding UTF-8

SystemRequirements 'libnng' >= 1.6.0 and 'libmbedtls' >= 2, or 'cmake' to compile NNG and/or Mbed TLS included in package sources

Depends R (>= 2.5)

RoxygenNote 7.2.3

NeedsCompilation yes

Author Charlie Gao [aut, cre] (<https://orcid.org/0000-0002-0750-061X>), Hibiki AI Limited [cph]

Maintainer Charlie Gao <charlie.gao@shikokuchuo.net>

Repository CRAN

Date/Publication 2023-01-22 14:30:02 UTC

R topics documented:

nanonext-package *nanonext: NNG (Nanomsg Next Gen) Lightweight Messaging Library*

Description

R binding for NNG (Nanomsg Next Gen), a successor to ZeroMQ. NNG is a socket library providing high-performance scalability protocols, implementing a cross-platform standard for messaging and communications. Serves as a concurrency framework for building distributed applications, utilising 'aio' objects which resolve automatically upon completion of asynchronous operations.

Usage notes

{nanonext} offers 2 equivalent interfaces: a functional interface, and an object-oriented interface.

The primary object in the functional interface is the Socket. Use `socket` to create a socket and dial or listen at an address. The socket is then passed as the first argument of subsequent actions such as `send()` or `recv()`.

The primary object in the object-oriented interface is the nano object. Use `nano` to create a nano object which encapsulates a Socket and Dialer/Listener. Methods such as `$send()` or `$recv()` can then be accessed directly from the object.

Documentation

Guide to the implemented protocols for sockets: protocols

Guide to the supported transports for dialers and listeners: transports

Options that can be set using `setopt()`: opts

Conceptual overview

NNG presents a socket view of networking. The sockets are constructed using protocol-specific functions, as a given socket implements precisely one protocol.

Each socket can be used to send and receive messages (if the protocol supports it, and implements the appropriate protocol semantics). For example, sub sockets automatically filter incoming messages to discard those for topics that have not been subscribed.

NNG sockets are message oriented, so that messages are either delivered wholly, or not at all. Partial delivery is not possible. Furthermore, NNG does not provide any other delivery or ordering guarantees; messages may be dropped or reordered (some protocols, such as req may offer stronger guarantees by performing their own retry and validation schemes).

Each socket can have zero, one, or many endpoints, which are either listeners or dialers (a given socket may freely choose whether it uses listeners, dialers, or both). These endpoints provide access to underlying transports, such as TCP, etc.

Each endpoint is associated with a URL, which is a service address. For dialers, this will be the service address that will be contacted, whereas for listeners this is where the listener will accept new connections.

Links

nanonext website: https://shikokuchuo.net/nanonext/
nanonext on CRAN: https://cran.r-project.org/package=nanonext

NNG website: https://nng.nanomsg.org/
Mbed TLS website: https://www.trustedfirmware.org/projects/mbed-tls/

Author(s)

Charlie Gao <charlie.gao@shikokuchuo.net> (ORCID)

`base64enc` *Base64 Encode / Decode*

Description

Encodes / decodes a character string or arbitrary R object to base64 encoding.

Usage

```
base64enc(x, convert = TRUE)

base64dec(x, convert = TRUE)
```

Arguments

`x`	an object.
`convert`	[default TRUE] logical value whether to convert the output to a character string or keep as a raw vector. Supplying a non-logical value will error.

Details

For encoding: a raw vector is encoded directly, a scalar character string is translated to raw before encoding, whilst all other objects are serialised first.

The result of encoding or decoding is always a raw vector, which is translated to a character string if 'convert' is TRUE, or returned directly if 'convert' is FALSE.

Set 'convert' to FALSE when decoding a raw vector or serialised object, which may be further passed to `unserialize`.

Value

A raw vector or character string depending on 'convert'.

Examples

```
base64enc("hello world!")
base64dec(base64enc("hello world!"))

base64enc("hello world!", convert = FALSE)
base64dec(base64enc("hello world!", convert = FALSE))

base64enc(data.frame())
unserialize(base64dec(base64enc(data.frame()), convert = FALSE))
```

`call_aio` *Call the Value of an Asynchronous Aio Operation*

Description

Retrieve the value of an asynchronous Aio operation, waiting for the operation to complete if still in progress.

Usage

```
call_aio(aio)
```

Arguments

`aio`	an Aio (object of class 'sendAio' or 'recvAio').

Details

For a 'recvAio', the received raw vector may be retrieved at `$raw` (unless 'keep.raw' was set to FALSE when receiving), and the converted R object at `$data`.

For a 'sendAio', the send result may be retrieved at `$result`. This will be zero on success, or else an integer error code.

To access the values directly, use for example on a 'recvAio' `x`: `call_aio(x)$data`.

For a 'recvAio', in case of an error in unserialisation or data conversion (for example if the incorrect mode was specified), the received raw vector will be stored at `$data` to allow for the data to be recovered.

Once the value has been successfully retrieved, the Aio is deallocated and only the value is stored in the Aio object.

Note this function operates silently and does not error even if 'aio' is not an active Aio, always returning invisibly the passed object.

Value

The passed object (invisibly).

Alternatively

Aio values may be accessed directly at `$result` for a 'sendAio', and `$raw` or `$data` for a 'recvAio'. If the Aio operation is yet to complete, an 'unresolved' logical NA will be returned. Once complete, the resolved value will be returned instead.

`unresolved` may also be used, which returns TRUE only if an Aio or Aio value has yet to resolve and FALSE otherwise. This is suitable for use in control flow statements such as `while` or `if`.

Examples

```
s1 <- socket("pair", listen = "inproc://nanonext")
s2 <- socket("pair", dial = "inproc://nanonext")

res <- send_aio(s1, data.frame(a = 1, b = 2), timeout = 100)
res
call_aio(res)
res$result

msg <- recv_aio(s2, timeout = 100)
msg
call_aio(msg)$data

close(s1)
close(s2)
```

`close.nanoContext` *Close Connection*

Description

Close Connection on a Socket, Context, Dialer, Listener or Stream.

Usage

```
## S3 method for class 'nanoContext'
close(con, ...)

## S3 method for class 'nanoDialer'
close(con, ...)

## S3 method for class 'nanoListener'
close(con, ...)

## S3 method for class 'ncurlSession'
close(con, ...)

## S3 method for class 'nanoSocket'
close(con, ...)
```

```
## S3 method for class 'nanoStream'
close(con, ...)
```

Arguments

con	a Socket, Context, Dialer, Listener or Stream.
...	not used.

Details

Closing an object explicitly frees its resources. An object can also be removed directly in which case its resources are freed when the object is garbage collected.

Dialers and Listeners are implicitly closed when the Socket they are associated with is closed.

Closing a Socket associated with a Context also closes the Context.

Closing a Socket or a Context: messages that have been submitted for sending may be flushed or delivered, depending upon the transport. Closing the Socket while data is in transmission will likely lead to loss of that data. There is no automatic linger or flush to ensure that the Socket send buffers have completely transmitted.

Closing a Stream: if any send or receive operations are pending, they will be terminated and any new operations will fail after the connection is closed.

Value

Invisibly, an integer exit code (zero on success).

---

context *Open Context*

Description

Open a new Context to be used with a Socket. The purpose of a Context is to permit applications to share a single socket, with its underlying dialers and listeners, while still benefiting from separate state tracking.

Usage

```
context(socket)
```

Arguments

socket	a Socket.

Details

Contexts allow the independent and concurrent use of stateful operations using the same socket. For example, two different contexts created on a rep socket can each receive requests, and send replies to them, without any regard to or interference with each other.

Only the following protocols support creation of contexts: req, rep, sub (in a pub/sub pattern), surveyor, respondent.

To send and receive over a context use `send` and `recv` or their async counterparts `send_aio` and `recv_aio`.

For nano objects, use the `$context_open()` method, which will attach a new context at `$context`. See `nano`.

Value

A new Context (object of class 'nanoContext' and 'nano').

Examples

```
s <- socket("req", listen = "inproc://nanonext")
ctx <- context(s)
ctx
close(ctx)
close(s)

n <- nano("req", listen = "inproc://nanonext")
n$context_open()
n$context
n$context_open()
n$context
n$context_close()
n$close()
```

---

device *Create Device*

Description

Creates a device which is a socket forwarder or proxy. Provides for improved horizontal scalability, reliability, and isolation.

Usage

```
device(s1, s2)
```

Arguments

s1	a raw mode Socket.
s2	a raw mode Socket.

Details

Only raw mode sockets may be used with this function. Sockets s1 and s2 must be compatible with each other, i.e. be opposite halves of a two protocol pattern, or both the same protocol for a single protocol pattern.

Value

NULL. If the device was successfully created, this function does not return.

Usage

Warning: this function is designed to be called in an isolated process with the two sockets. Once called, it will block with no ability to interrupt. To terminate the device, the process must be killed (in interactive sessions this may be done by sending SIGQUIT e.g. ctrl + \).

`dial` *Dial an Address from a Socket*

Description

Creates a new Dialer and binds it to a Socket.

Usage

```
dial(socket, url = "inproc://nanonext", autostart = TRUE)
```

Arguments

`socket`	a Socket.
`url`	[default 'inproc://nanonext'] a URL to dial, specifying the transport and address as a character string e.g. 'inproc://anyvalue' or 'tcp://127.0.0.1:5555' (see transports).
`autostart`	[default TRUE] whether to start the dialer (by default asynchronously). Set to NA to start synchronously - this is less resilient if a connection is not immediately possible, but avoids subtle errors from attempting to use the socket before an asynchronous dial has completed. Set to FALSE if setting configuration options on the dialer as it is not generally possible to change these once started. Supplying a non-logical value will error.

Details

To view all Dialers bound to a socket use `$dialer` on the socket, which returns a list of Dialer objects. To access any individual Dialer (e.g. to set options on it), index into the list e.g. `$dialer[[1]]` to return the first Dialer.

A Dialer is an external pointer to a dialer object, which creates a single outgoing connection at a time. If the connection is broken, or fails, the dialer object will automatically attempt to reconnect, and will keep doing so until the dialer or socket is destroyed.

Value

Invisibly, an integer exit code (zero on success). A new Dialer (object of class 'nanoDialer' and 'nano') is created and bound to the Socket if successful.

Further details

Dialers and Listeners are always associated with a single socket. A given socket may have multiple Listeners and/or multiple Dialers.

The client/server relationship described by dialer/listener is completely orthogonal to any similar relationship in the protocols. For example, a rep socket may use a dialer to connect to a listener on an req socket. This orthogonality can lead to innovative solutions to otherwise challenging communications problems.

Any configuration options on the dialer/listener should be set by `setopt` before starting the dialer/listener with `start`.

Dialers/Listeners may be destroyed by `close`. They are also closed when their associated socket is closed.

Examples

```
socket <- socket("rep")
dial(socket, url = "tcp://127.0.0.1:6545", autostart = FALSE)
socket$dialer
start(socket$dialer[[1]])
socket$dialer
close(socket$dialer[[1]])
close(socket)

nano <- nano("bus")
nano$dial(url = "tcp://127.0.0.1:6546", autostart = FALSE)
nano$dialer
nano$dialer_start()
nano$dialer
close(nano$dialer[[1]])
nano$close()
```

---

getopt *Get Option for a Socket, Context, Stream, Listener or Dialer*

Description

Get value of opts for a Socket, Context, Stream, Listener or Dialer.

Usage

```
getopt(object, opt)
```

Arguments

object	a Socket, Context, Stream, Listener or Dialer.
opt	name of option, e.g. 'reconnect-time-min', as a character string. See opts.

Details

To get options for a Listener or Dialer attached to a Socket or nano object, pass in the objects directly via for example `$listener[[1]]` for the first Listener.

Value

The value of the option (logical for type 'bool', integer for 'int', 'duration' and 'size', character for 'string' and double for 'uint64').

Examples

```
s <- socket("pair")
getopt(s, "send-buffer")
close(s)

s <- socket("req")
ctx <- context(s)
getopt(ctx, "send-timeout")
close(ctx)
close(s)

s <- socket("pair", dial = "inproc://nanonext", autostart = FALSE)
getopt(s$dialer[[1]], "reconnect-time-min")
close(s)

s <- socket("pair", listen = "inproc://nanonext", autostart = FALSE)
getopt(s$listener[[1]], "recv-size-max")
close(s)
```

---

is_aio *Is Aio*

Description

Is the object an Aio (sendAio or recvAio).

Usage

```
is_aio(x)
```

Arguments

x	an object.

Value

Logical value TRUE if 'x' is of class 'recvAio' or 'sendAio', FALSE otherwise.

Examples

```
sock <- socket(listen = "inproc://isaio")
r <- recv_aio(sock)
s <- send_aio(sock, "test")

is_aio(r)
is_aio(s)

close(sock)
```

---

is_error_value *Is Error Value*

Description

Is the object an error value generated by NNG. All returned integer error codes are classed as 'errorValue' to be distinguishable from integer message values. Includes user-specified errors such as 'aio' timeouts.

Usage

```
is_error_value(x)
```

Arguments

x	an object.

Value

Logical value TRUE if 'x' is of class 'errorValue', FALSE otherwise.

Examples

```
is_error_value(1L)
```

is_nano *Is Nano*

Description

Is the object an object created by {nanonext} i.e. a nanoSocket, nanoContext, nanoStream, nanoListener, nanoDialer or nano Object.

Usage

```
is_nano(x)
```

Arguments

x	an object.

Details

Note: does not include Aio objects, for which there is a separate function is_aio.

Value

Logical value TRUE or FALSE.

Examples

```
s <- socket()
is_nano(s)
n <- nano()
is_nano(n)

close(s)
n$close()
```

is_nul_byte *Is Nul Byte*

Description

Is the object a nul byte.

Usage

```
is_nul_byte(x)
```

Arguments

x	an object.

Value

Logical value TRUE or FALSE.

Examples

```
is_nul_byte(as.raw(0L))
is_nul_byte(raw(length = 1L))
is_nul_byte(writeBin("", con = raw()))

is_nul_byte(0L)
is_nul_byte(NULL)
is_nul_byte(NA)
```

---

listen *Listen to an Address from a Socket*

Description

Creates a new Listener and binds it to a Socket.

Usage

```
listen(socket, url = "inproc://nanonext", autostart = TRUE)
```

Arguments

socket	a Socket.
url	[default 'inproc://nanonext'] a URL to dial or listen at, specifying the transport and address as a character string e.g. 'inproc://anyvalue' or 'tcp://127.0.0.1:5555' (see transports).
autostart	[default TRUE] whether to start the listener. Set to FALSE if setting configuration options on the listener as it is not generally possible to change these once started. Supplying a non-logical value will error.

Details

To view all Listeners bound to a socket use `$listener` on the socket, which returns a list of Listener objects. To access any individual Listener (e.g. to set options on it), index into the list e.g. `$listener[[1]]` to return the first Listener.

A listener is an external pointer to a listener object, which accepts incoming connections. A given listener object may have many connections at the same time, much like an HTTP server can have many connections to multiple clients simultaneously.

Value

Invisibly, an integer exit code (zero on success). A new Listener (object of class 'nanoListener' and 'nano') is created and bound to the Socket if successful.

Further details

Dialers and Listeners are always associated with a single socket. A given socket may have multiple Listeners and/or multiple Dialers.

The client/server relationship described by dialer/listener is completely orthogonal to any similar relationship in the protocols. For example, a rep socket may use a dialer to connect to a listener on an req socket. This orthogonality can lead to innovative solutions to otherwise challenging communications problems.

Any configuration options on the dialer/listener should be set by `setopt` before starting the dialer/listener with `start`.

Dialers/Listeners may be destroyed by `close`. They are also closed when their associated socket is closed.

Examples

```
socket <- socket("req")
listen(socket, url = "tcp://127.0.0.1:6547", autostart = FALSE)
socket$listener
start(socket$listener[[1]])
socket$listener
close(socket$listener[[1]])
close(socket)

nano <- nano("bus")
nano$listen(url = "tcp://127.0.0.1:6548", autostart = FALSE)
nano$listener
nano$listener_start()
nano$listener
close(nano$listener[[1]])
nano$close()
```

---

mclock *Clock Utility*

Description

Provides the number of elapsed milliseconds since an arbitrary reference time in the past. The reference time will be the same for a given session, but may differ between sessions.

Usage

```
mclock()
```

Details

A convenience function for building concurrent applications. The resolution of the clock depends on the underlying system timing facilities and may not be particularly fine-grained. This utility should however be faster than using `Sys.time()`.

Value

A double.

Examples

```
time <- mclock(); msleep(100); mclock() - time
```

`messenger` *Messenger*

Description

Multi-threaded, console-based, 2-way instant messaging system with authentication, based on NNG scalability protocols.

Usage

```
messenger(url, auth = NULL)
```

Arguments

`url`	a URL to connect to, specifying the transport and address as a character string e.g. 'tcp://127.0.0.1:5555' (see transports).
`auth`	[default NULL] an R object (possessed by both parties) which serves as a pre-shared key on which to authenticate the communication. Note: the object is never sent, only a random subset of its SHA-512 hash.

Value

Invisible NULL.

Usage

Type outgoing messages and hit return to send.

The timestamps of outgoing messages are prefixed by > and that of incoming messages by <.

`:q` is the command to quit.

Both parties must supply the same argument for 'auth', otherwise the party trying to connect will receive an 'authentication error' and be disconnected immediately.

NOTE: This is currently a proof of concept with an experimental authentication protocol and should not be used for critical applications.

msleep *Sleep Utility*

Description

Sleep function. May block for longer than requested, with the actual wait time determined by the capabilities of the underlying system.

Usage

```
msleep(msec)
```

Arguments

msec	integer number of milliseconds to block the caller.

Details

If 'msec' is non-integer, it will be coerced to integer. Non-numeric input will be ignored and return immediately.

Note that unlike Sys.sleep, this function is not user-interruptible by sending SIGINT e.g. with ctrl + c.

Value

Invisible NULL.

Examples

```
time <- mclock(); msleep(100); mclock() - time
```

nano *Create Nano Object*

Description

Create a nano object, encapsulating a Socket, Dialers/Listeners and associated methods.

Usage

```
nano(
 protocol = c("bus", "pair", "push", "pull", "pub", "sub", "req", "rep", "surveyor",
    "respondent"),
  dial = NULL,
  listen = NULL,
  autostart = TRUE
)
```

Arguments

`protocol`	[default 'bus'] choose protocol - 'bus', 'pair', 'push', 'pull', 'pub', 'sub', 'req', 'rep', 'surveyor', or 'respondent' - see protocols.
`dial`	(optional) a URL to dial, specifying the transport and address as a character string e.g. 'inproc://anyvalue' or 'tcp://127.0.0.1:5555' (see transports).
`listen`	(optional) a URL to listen at, specifying the transport and address as a character string e.g. 'inproc://anyvalue' or 'tcp://127.0.0.1:5555' (see transports).
`autostart`	[default TRUE] whether to start the dialer/listener. Set to FALSE if you wish to set configuration options on the dialer/listener as it is not generally possible to change these once started. For dialers only: set to NA to start synchronously - this is less resilient if a connection is not immediately possible, but avoids subtle errors from attempting to use the socket before an asynchronous dial has completed.

Details

This function encapsulates a Socket, Dialer and/or Listener, and its associated methods.

The Socket may be accessed by `$socket`, and the Dialer or Listener by `$dialer[[1]]` or `$listener[[1]]` respectively.

The object's methods may be accessed by `$` e.g. `$send()` or `$recv()`. These methods mirror their functional equivalents, with the same arguments and defaults, apart from that the first argument of the functional equivalent is mapped to the object's encapsulated socket (or context, if active) and does not need to be supplied.

More complex network topologies may be created by binding further dialers or listeners using the object's `$dial()` and `$listen()` methods. The new dialer/listener will be attached to the object e.g. if the object already has a dialer, then at `$dialer[[2]]` etc.

Note that `$dialer_setopt()` and `$listener_setopt()` methods will be available once dialers/listeners are attached to the object. These methods apply settings to all dialers or listeners equally. To apply settings to individual dialers/listeners, access them directly via `$dialer[[2]]` or `$listener[[2]]` etc.

For Dialers or Listeners not automatically started, the `$dialer_start()` or `$listener_start()` methods will be available. These act on the most recently created Dialer or Listener respectively.

For applicable protocols, new contexts may be created by using the `$context_open()` method. This will attach a new context at `$context` as well as a `$context_close()` method. While a context is active, all object methods use the context rather than the socket. A new context may be created by calling `$context_open()`, which will replace any existing context. It is only necessary to use `$context_close()` to close the existing context and revert to using the socket.

Value

A nano object of class 'nanoObject'.

Examples

```
nano <- nano("bus", listen = "inproc://nanonext")
nano
```

```
nano$socket
nano$listener[[1]]

nano$setopt("send-timeout", 1000)

nano$listen(url = "inproc://nanonextgen")
nano$listener

nano1 <- nano("bus", dial = "inproc://nanonext")
nano$send("example test", mode = "raw")
nano1$recv("character")

nano$close()
nano1$close()
```

---

ncurl *ncurl*

Description

nano cURL - a minimalist http(s) client.

Usage

```
ncurl(
  url,
  async = FALSE,
  convert = TRUE,
  follow = FALSE,
  method = NULL,
  headers = NULL,
  data = NULL,
  response = NULL,
  pem = NULL
)
```

Arguments

url	the URL address.
async	[default FALSE] logical value whether to perform an async request, in which case an 'ncurlAio' is returned instead of a list.
convert	[default TRUE] logical value whether to attempt conversion of the received raw bytes to a character vector. Supplying a non-logical value will error.
follow	[default FALSE] logical value whether to automatically follow redirects (not applicable for async requests). If FALSE (or async), the redirect address is returned as response header 'Location'. Supplying a non-logical value will error.

method	(optional) the HTTP method (defaults to 'GET' if not specified).
headers	(optional) a named list or character vector specifying the HTTP request headers e.g. list(`Content-Type` = "text/plain") or c(Authorization = "Bearer APIKEY"). Supplying a non-named list or vector will error.
data	(optional) the request data to be submitted.
response	(optional) a character vector or list specifying the response headers to return e.g. c("date", "server") or list("Date", "Server"). These are case-insensitive and will return NULL if not present.
pem	(optional) applicable to secure HTTPS sites only. The path to a file containing X.509 certificate(s) in PEM format, comprising the certificate authority certificate chain (and revocation list if present). If missing or NULL, certificates are not validated.

Value

Named list of 4 elements:

- $status - integer HTTP repsonse status code (200 - OK). Use status_code for a translation of the meaning.
- $headers - named list of response headers supplied in 'response', or NULL otherwise. If the status code is within the 300 range, i.e. a redirect, the response header 'Location' is automatically appended to return the redirect address.
- $raw - raw vector of the received resource (use writeBin to save to a file).
- $data - converted character string (if 'convert' = TRUE and content is a recognised text format), or NULL otherwise. This may be further parsed as html, json, xml etc. if required.

Or else, if async = TRUE, an 'ncurlAio' (object of class 'ncurlAio' and 'recvAio') (invisibly).

Examples

```
ncurl("https://httpbin.org/get", response = c("date", "server"))
ncurl("http://httpbin.org/put",,,,"PUT", list(Authorization = "Bearer APIKEY"), "hello world")
ncurl("http://httpbin.org/post",,,,"POST", c(`Content-Type` = "application/json"),'{"k":"v"}')
```

---

ncurl_session *ncurl Session*

Description

nano cURL - a minimalist http(s) client. A session encapsulates a connection, along with all related parameters. A session may be used multiple times to return data by repeatedly calling transact.

Usage

```
ncurl_session(
  url,
  convert = TRUE,
  method = NULL,
  headers = NULL,
  data = NULL,
  response = NULL,
  pem = NULL
)
```

Arguments

url	the URL address.
convert	[default TRUE] logical value whether to attempt conversion of the received raw bytes to a character vector. Supplying a non-logical value will error.
method	(optional) the HTTP method (defaults to 'GET' if not specified).
headers	(optional) a named list or character vector specifying the HTTP request headers e.g. `list(`Content-Type` = "text/plain")` or `c(Authorization = "Bearer APIKEY")`. Supplying a non-named list or vector will error.
data	(optional) the request data to be submitted.
response	(optional) a character vector or list specifying the response headers to return e.g. `c("date", "server")` or `list("Date", "Server")`. These are case-insensitive and will return NULL if not present.
pem	(optional) applicable to secure HTTPS sites only. The path to a file containing X.509 certificate(s) in PEM format, comprising the certificate authority certificate chain (and revocation list if present). If missing or NULL, certificates are not validated.

Value

An 'ncurlSession' object.

Examples

```
s <- tryCatch(ncurl_session("https://httpbin.org/get", response = c("date")), error = identity)
s
```

---

nng_error *Translate Error Codes*

Description

Translate integer exit codes generated by the NNG library. All package functions return an integer exit code on error rather than the expected return value. These are classed 'errorValue' and may be checked by `is_error_value`.

Usage

```
nng_error(xc)
```

Arguments

xc	integer exit code to translate.

Value

A character vector.

Examples

```
nng_error(1L)
```

`nng_version` *NNG Library Version*

Description

Returns the versions of the 'libnng' and 'libmbedtls' libraries used.

Usage

```
nng_version()
```

Value

A character vector of length 2.

Examples

```
nng_version()
```

`opts` *Opts [Documentation]*

Description

Options that can be set on Sockets, Contexts, Streams, Dialers or Listeners.

Use `setopt` to set options and `getopt` to get the value of options.

Some options are only meaningful or supported in certain contexts; for example there is no single meaningful address for a socket, since sockets can have multiple dialers and endpoints associated with them.

For an authoritative guide please refer to the online documentation for the NNG library at https://nng.nanomsg.org/man/.

Global Options

- 'reconnect-time-min' [type 'ms']
  This is the minimum amount of time (milliseconds) to wait before attempting to establish a connection after a previous attempt has failed. This can be set on a socket, but it can also be overridden on an individual dialer. The option is irrelevant for listeners.
- 'reconnect-time-max' [type 'ms']
  This is the maximum amount of time (milliseconds) to wait before attempting to establish a connection after a previous attempt has failed. If this is non-zero, then the time between successive connection attempts will start at the value of 'reconnect-time-min', and grow exponentially, until it reaches this value. If this value is zero, then no exponential back-off between connection attempts is done, and each attempt will wait the time specified by 'reconnect-time-min'. This can be set on a socket, but it can also be overridden on an individual dialer. The option is irrelevant for listeners.
- 'recv-size-max' [type 'size']
  This is the maximum message size that the will be accepted from a remote peer. If a peer attempts to send a message larger than this, then the message will be discarded. If the value of this is zero, then no limit on message sizes is enforced. This option exists to prevent certain kinds of denial-of-service attacks, where a malicious agent can claim to want to send an extraordinarily large message, without sending any data. This option can be set for the socket, but may be overridden for on a per-dialer or per-listener basis. NOTE: Applications on hostile networks should set this to a non-zero value to prevent denial-of-service attacks. NOTE: Some transports may have further message size restrictions.
- 'recv-buffer' [type 'int']
  This is the depth of the socket's receive buffer as a number of messages. Messages received by a transport may be buffered until the application has accepted them for delivery. This value must be an integer between 0 and 8192, inclusive. NOTE: Not all protocols support buffering received messages. For example req can only deal with a single reply at a time.
- 'recv-timeout' [type 'ms']
  This is the socket receive timeout in milliseconds. When no message is available for receiving at the socket for this period of time, receive operations will fail with a return value of 5L ('timed out').

- 'send-buffer' [type 'int']

  This is the depth of the socket send buffer as a number of messages. Messages sent by an application may be buffered by the socket until a transport is ready to accept them for delivery. This value must be an integer between 0 and 8192, inclusive. NOTE: Not all protocols support buffering sent messages; generally multicast protocols like pub will simply discard messages when they cannot be delivered immediately.

- 'send-timeout' [type 'ms']

  This is the socket send timeout in milliseconds. When a message cannot be queued for delivery by the socket for this period of time (such as if send buffers are full), the operation will fail with a return value of 5L ('timed out').

- 'socket-name' [type 'string']

  This is the socket name. By default this is a string corresponding to the value of the socket. The string must fit within 64-bytes, including the terminating NUL byte. The value is intended for application use, and is not used for anything in the library itself.

Protocol-specific Options

- 'req:resend-time' [type 'ms']

  (Request protocol) When a new request is started, a timer of this duration is also started. If no reply is received before this timer expires, then the request will be resent. (Requests are also automatically resent if the peer to whom the original request was sent disconnects, or if a peer becomes available while the requester is waiting for an available peer.)

- 'sub:subscribe' [type 'string']

  (Subscribe protocol) This option registers a topic that the subscriber is interested in. Each incoming message is checked against the list of subscribed topics. If the body begins with the entire set of bytes in the topic, then the message is accepted. If no topic matches, then the message is discarded. To receive all messages, set the topic to NULL.

- 'sub:unsubscribe' [type 'string']

  (Subscribe protocol) This option removes a topic from the subscription list. Note that if the topic was not previously subscribed to with 'sub:subscribe' then an 'entry not found' error will result.

- 'sub:prefnew' [type 'bool']

  (Subscribe protocol) This option specifies the behavior of the subscriber when the queue is full. When TRUE (the default), the subscriber will make room in the queue by removing the oldest message. When FALSE, the subscriber will reject messages if the message queue does not have room.

- 'surveyor:survey-time' [type 'ms']

  (Surveyor protocol) Duration of surveys. When a new survey is started, a timer of this duration is also started. Any responses arriving after this time will be discarded. Attempts to receive after the timer expires with no other surveys started will result in an 'incorrect state' error. Attempts to receive when this timer expires will result in a 'timed out' error.

Transport-specific Options

- 'ipc:permissions' [type 'int']

(IPC transport) This option may be applied to a listener to configure the permissions that are used on the UNIX domain socket created by that listener. This property is only supported on POSIX systems. The value is of type int, representing the normal permission bits on a file, such as 0600 (typically meaning read-write to the owner, and no permissions for anyone else.) The default is system-specific, most often 0644.

- 'tcp-nodelay' [type 'bool']

  (TCP transport) This option is used to disable (or enable) the use of Nagle's algorithm for TCP connections. When TRUE (the default), messages are sent immediately by the underlying TCP stream without waiting to gather more data. When FALSE, Nagle's algorithm is enabled, and the TCP stream may wait briefly in an attempt to coalesce messages. Nagle's algorithm is useful on low-bandwidth connections to reduce overhead, but it comes at a cost to latency. When used on a dialer or a listener, the value affects how newly created connections will be configured.

- 'tcp-keepalive' [type 'bool']

  (TCP transport) This option is used to enable the sending of keep-alive messages on the underlying TCP stream. This option is FALSE by default. When enabled, if no messages are seen for a period of time, then a zero length TCP message is sent with the ACK flag set in an attempt to tickle some traffic from the peer. If none is still seen (after some platform-specific number of retries and timeouts), then the remote peer is presumed dead, and the connection is closed. When used on a dialer or a listener, the value affects how newly created connections will be configured. This option has two purposes. First, it can be used to detect dead peers on an otherwise quiescent network. Second, it can be used to keep connection table entries in NAT and other middleware from expiring due to lack of activity.

- 'ws:request-headers' [type 'string']

  (WebSocket transport) Concatenation of multiple lines terminated by CRLF sequences, that can be used to add further headers to the HTTP request sent when connecting. This option can be set on dialers, and must be done before the transport is started.

- 'ws:response-headers' [type 'string']

  (WebSocket transport) Concatenation of multiple lines terminated by CRLF sequences, that can be used to add further headers to the HTTP response sent when connecting. This option can be set on listeners, and must be done before the transport is started.

`protocols` *Protocols [Documentation]*

Description

Protocols implemented by {nanonext}.

For an authoritative guide please refer to the online documentation for the NNG library at https://nng.nanomsg.org/man/.

Bus (mesh networks)

[protocol, bus] The bus protocol is useful for routing applications or for building mesh networks where every peer is connected to every other peer. In this protocol, each message sent by a node is sent to every one of its directly connected peers. This socket may be used to send and receive messages. Sending messages will attempt to deliver to each directly connected peer.

Messages are only sent to directly connected peers. This means that in the event that a peer is connected indirectly, it will not receive messages. When using this protocol to build mesh networks, it is therefore important that a fully-connected mesh network be constructed.

All message delivery in this pattern is best-effort, which means that peers may not receive messages. Furthermore, delivery may occur to some, all, or none of the directly connected peers (messages are not delivered when peer nodes are unable to receive). Hence, send operations will never block; instead if the message cannot be delivered for any reason it is discarded.

Pair (two-way radio)

[protocol, pair] The pair protocol implements a peer-to-peer pattern, where relationships between peers are one-to-one. Only one peer may be connected to another peer at a time, but both may speak freely.

Normally, this pattern will block when attempting to send a message if no peer is able to receive the message.

Push/Pull (one-way pipeline)

In the pipeline pattern, pushers distribute messages to pullers, hence useful for solving producer/consumer problems.

If multiple peers are connected, the pattern attempts to distribute fairly. Each message sent by a pusher will be sent to one of its peer pullers, chosen in a round-robin fashion. This property makes this pattern useful in load-balancing scenarios.

[protocol, push] The push protocol is one half of a pipeline pattern. The other side is the pull protocol.

[protocol, pull] The pull protocol is one half of a pipeline pattern. The other half is the push protocol.

Publisher/Subscriber (topics & broadcast)

In a publisher/subscriber pattern, a publisher sends data, which is broadcast to all subscribers. The subscribing applications only see the data to which they have subscribed.

[protocol, pub] The pub protocol is one half of a publisher/subscriber pattern. This socket may be used to send messages, but is unable to receive them.

[protocol, sub] The sub protocol is one half of a publisher/subscriber pattern. This socket may be used to receive messages, but is unable to send them.

Request/Reply (RPC)

In a request/reply pattern, a requester sends a message to one replier, who is expected to reply with a single answer. This is used for synchronous communications, for example remote procedure calls (RPCs).

The request is resent automatically if no reply arrives, until a reply is received or the request times out.

[protocol, req] The req protocol is one half of a request/reply pattern. This socket may be used to send messages (requests), and then to receive replies. Generally a reply can only be received after sending a request.

[protocol, rep] The rep protocol is one half of a request/reply pattern. This socket may be used to receive messages (requests), and then to send replies. Generally a reply can only be sent after receiving a request.

Surveyor/Respondent (voting & service discovery)

In a survey pattern, a surveyor sends a survey, which is broadcast to all peer respondents. The respondents then have a chance to reply (but are not obliged to reply). The survey itself is a timed event, so that responses received after the survey has finished are discarded.

[protocol, surveyor] The surveyor protocol is one half of a survey pattern. This socket may be used to send messages (surveys), and then to receive replies. A reply can only be received after sending a survey. A surveyor can normally expect to receive at most one reply from each responder. (Messages can be duplicated in some topologies, so there is no guarantee of this.)

[protocol, respondent] The respondent protocol is one half of a survey pattern. This socket may be used to receive messages, and then to send replies. A reply can only be sent after receiving a survey, and generally the reply will be sent to the surveyor from whom the last survey was received.

`random` *NNG Random Number Generator*

Description

Strictly not for statistical analysis. Not reproducible. No ability to set a seed value. Provides random numbers suitable for system functions such as cryptographic key generation. Random values are obtained using platform-specific strong cryptographic random number facilities where available.

Usage

```
random(n = 1L)
```

Arguments

`n`	[default 1L] length of vector to return.

Details

If 'n' is non-integer, it will be coerced to integer; if a vector, only the first element will be used.

Value

A length 'n' vector of random positive doubles.

Examples

```
random()
random(n = 3L)
```

---

recv *Receive*

Description

Receive data over a connection (Socket, Context or Stream).

Usage

```
recv(
  con,
  mode = c("serial", "character", "complex", "double", "integer", "logical", "numeric",
    "raw"),
  block = NULL,
  keep.raw = FALSE,
  n = 65536L
)
```

Arguments

con	a Socket, Context or Stream.
mode	[default 'serial'] mode of vector to be received - one of 'serial', 'character', 'complex', 'double', 'integer', 'logical', 'numeric', or 'raw'. The default 'serial' means a serialised R object, for the other modes, the raw vector received will be converted into the respective mode. For Streams, 'serial' is not an option and the default is 'character'. Alternatively, for performance, specify an integer position in the vector of choices e.g. 1L for 'serial', 2L for 'character' etc.
block	[default NULL] which applies the connection default (see section 'Blocking' below). Specify logical TRUE to block until successful or FALSE to return immediately even if unsuccessful (e.g. if no connection is available), or else an integer value specifying the maximum time to block in milliseconds, after which the operation will time out.
keep.raw	[default FALSE] logical flag whether to keep and return the received raw vector along with the converted data. Supplying a non-logical value will error.
n	[default 65536L] applicable to Streams only, the maximum number of bytes to receive. Can be an over-estimate, but note that a buffer of this size is reserved.

Details

In case of an error, an integer 'errorValue' is returned (to be distiguishable from an integer message value). This can be verified using `is_error_value`.

If the raw message was successfully received but an error occurred in unserialisation or data conversion (for example if the incorrect mode was specified), the received raw vector will always be returned to allow for the data to be recovered.

Value

Depending on the value of 'keep.raw': if TRUE, a named list of 2 elements - `$raw` containing the received raw vector and `$data` containing the converted data, or if FALSE, the converted data.

Blocking

For Sockets: the default behaviour is non-blocking with `block = FALSE`. This will return immediately with an error if no messages are available.

For Contexts and Streams: the default behaviour is blocking with `block = TRUE`. This will wait until a message is received. Set a timeout in this case to ensure that the function returns under all scenarios. As the underlying implementation uses an asynchronous send with a wait, it is recommended to set a positive integer value for `block` rather than FALSE.

Examples

```
s1 <- socket("pair", listen = "inproc://nanonext")
s2 <- socket("pair", dial = "inproc://nanonext")

send(s1, data.frame(a = 1, b = 2))
res <- recv(s2)
res
send(s1, data.frame(a = 1, b = 2))
recv(s2, keep.raw = TRUE)

send(s1, c(1.1, 2.2, 3.3), mode = "raw")
res <- recv(s2, mode = "double", block = 100, keep.raw = TRUE)
res
send(s1, "example message", mode = "raw")
recv(s2, mode = "character")

close(s1)
close(s2)

req <- socket("req", listen = "inproc://nanonext")
rep <- socket("rep", dial = "inproc://nanonext")

ctxq <- context(req)
ctxp <- context(rep)
send(ctxq, data.frame(a = 1, b = 2), block = 100)
recv(ctxp, block = 100)

send(ctxq, c(1.1, 2.2, 3.3), mode = "raw", block = 100)
```

```
recv(ctxp, mode = "double", block = 100)

close(req)
close(rep)
```

---

recv_aio *Receive Async*

Description

Receive data asynchronously over a connection (Socket, Context or Stream).

Usage

```
recv_aio(
  con,
 mode = c("serial", "character", "complex", "double", "integer", "logical", "numeric",
    "raw"),
  timeout = NULL,
  keep.raw = FALSE,
  n = 65536L
)
```

Arguments

con	a Socket, Context or Stream.
mode	[default 'serial'] mode of vector to be received - one of 'serial', 'character', 'complex', 'double', 'integer', 'logical', 'numeric', or 'raw'. The default 'serial' means a serialised R object, for the other modes, the raw vector received will be converted into the respective mode. For Streams, 'serial' is not an option and the default is 'character'. Alternatively, for performance, specify an integer position in the vector of choices e.g. 1L for 'serial', 2L for 'character' etc.
timeout	[default NULL] integer value in milliseconds or NULL, which applies a socket-specific default, usually the same as no timeout.
keep.raw	[default FALSE] logical flag whether to keep and return the received raw vector along with the converted data. Supplying a non-logical value will error.
n	[default 65536L] applicable to Streams only, the maximum number of bytes to receive. Can be an over-estimate, but note that a buffer of this size is reserved.

Details

Async receive is always non-blocking and returns a 'recvAio' immediately.

For a 'recvAio', the received message is available at `$data`, and the raw message at `$raw` (if kept). An 'unresolved' logical NA is returned if the async operation is yet to complete.

To wait for the async operation to complete and retrieve the received message, use call_aio on the returned 'recvAio' object.

Alternatively, to stop the async operation, use stop_aio.

In case of an error, an integer 'errorValue' is returned (to be distiguishable from an integer message value). This can be verified using is_error_value.

If the raw message was successfully received but an error occurred in unserialisation or data conversion (for example if the incorrect mode was specified), the received raw vector will be stored at $data to allow for the data to be recovered.

Value

A 'recvAio' (object of class 'recvAio') (invisibly).

Examples

```
s1 <- socket("pair", listen = "inproc://nanonext")
s2 <- socket("pair", dial = "inproc://nanonext")

res <- send_aio(s1, data.frame(a = 1, b = 2), timeout = 100)
msg <- recv_aio(s2, timeout = 100)
msg
msg$data

res <- send_aio(s1, c(1.1, 2.2, 3.3), mode = "raw", timeout = 100)
msg <- recv_aio(s2, mode = "double", timeout = 100, keep.raw = TRUE)
msg
msg$raw
msg$data

res <- send_aio(s1, "example message", mode = "raw", timeout = 100)
msg <- recv_aio(s2, mode = "character", timeout = 100, keep.raw = TRUE)
call_aio(msg)
msg$raw
msg$data

close(s1)
close(s2)
```

---

reply *Reply over Context (RPC Server for Req/Rep Protocol)*

Description

Implements an executor/server for the rep node of the req/rep protocol. Awaits data, applies an arbitrary specified function, and returns the result to the caller/client.

Usage

```
reply(
  context,
  execute,
  recv_mode = c("serial", "character", "complex", "double", "integer", "logical",
    "numeric", "raw"),
  send_mode = c("serial", "raw"),
  timeout = NULL,
  ...
)
```

Arguments

context	a Context.
execute	a function which takes the received (converted) data as its first argument. Can be an anonymous function of the form `function(x) do(x)`. Additional arguments can also be passed in through '...'.
recv_mode	[default 'serial'] mode of vector to be received - one of 'serial', 'character', 'complex', 'double', 'integer', 'logical', 'numeric', or 'raw'. The default 'serial' means a serialised R object, for the other modes, the raw vector received will be converted into the respective mode. Alternatively, for performance, specify an integer position in the vector of choices e.g. 1L for 'serial', 2L for 'character' etc.
send_mode	[default 'serial'] whether data will be sent serialized or as a raw vector. Use 'serial' for sending and receiving within R to ensure perfect reproducibility. Use 'raw' for sending vectors of any type (will be converted to a raw byte vector for sending) - essential when interfacing with external applications. Alternatively, for performance, specify an integer position in the vector of choices i.e. 1L for 'serial' or 2L for 'raw'.
timeout	[default NULL] integer value in milliseconds or NULL, which applies a socket-specific default, usually the same as no timeout. Note that this applies to receiving the request. The total elapsed time would also include performing 'execute' on the received data. The timeout then also applies to sending the result (in the event that the requestor has become unavailable since sending the request).
...	additional arguments passed to the function specified by 'execute'.

Details

Receive will block while awaiting a message to arrive and is usually the desired behaviour. Set a timeout to allow the function to return if no data is forthcoming.

In the event of an error in either processing the messages or in evaluation of the function with respect to the data, a nul byte `00` (or serialized nul byte) will be sent in reply to the client to signal an error. This is to be distinguishable from a possible return value. `is_nul_byte` can be used to test for a nul byte.

Value

Integer exit code (zero on success).

Examples

```
req <- socket("req", listen = "tcp://127.0.0.1:6546")
rep <- socket("rep", dial = "tcp://127.0.0.1:6546")

ctxq <- context(req)
ctxp <- context(rep)

send(ctxq, 2022, block = 100)
reply(ctxp, execute = function(x) x + 1, send_mode = "raw", timeout = 100)
recv(ctxq, mode = "double", block = 100)

send(ctxq, 100, mode = "raw", block = 100)
reply(ctxp, recv_mode = "double", execute = log, base = 10, timeout = 100)
recv(ctxq, block = 100)

close(req)
close(rep)
```

`request` *Request over Context (RPC Client for Req/Rep Protocol)*

Description

Implements a caller/client for the req node of the req/rep protocol. Sends data to the rep node (executor/server) and returns an Aio, which can be called when the result is required.

Usage

```
request(
  context,
  data,
  send_mode = c("serial", "raw"),
  recv_mode = c("serial", "character", "complex", "double", "integer", "logical",
    "numeric", "raw"),
  timeout = NULL,
  keep.raw = FALSE
)
```

Arguments

`context`	a Context.
`data`	an object (if send_mode = 'raw', a vector).
`send_mode`	[default 'serial'] whether data will be sent serialized or as a raw vector. Use 'serial' for sending and receiving within R to ensure perfect reproducibility. Use 'raw' for sending vectors of any type (will be converted to a raw byte vector for sending) - essential when interfacing with external applications. Alternatively, for performance, specify an integer position in the vector of choices i.e. 1L for 'serial' or 2L for 'raw'.

recv_mode	[default 'serial'] mode of vector to be received - one of 'serial', 'character', 'complex', 'double', 'integer', 'logical', 'numeric', or 'raw'. The default 'serial' means a serialised R object, for the other modes, the raw vector received will be converted into the respective mode. Alternatively, for performance, specify an integer position in the vector of choices e.g. 1L for 'serial', 2L for 'character' etc.
timeout	[default NULL] integer value in milliseconds or NULL, which applies a socket-specific default, usually the same as no timeout. Note that this applies to receiving the result.
keep.raw	[default FALSE] logical flag whether to keep and return the received raw vector along with the converted data. Supplying a non-logical value will error.

Details

Sending the request and receiving the result are both performed async, hence the function will return immediately with a 'recvAio' object. Access the return value at `$data`.

This is designed so that the process on the server can run concurrently without blocking the client.

Optionally use `call_aio` on the 'recvAio' to call (and wait for) the result.

If an error occured in the server process, a nul byte `00` will be received (as `$data` if 'recv_mode' = 'serial', as `$raw` otherwise). This allows an error to be easily distinguished from a NULL return value. `is_nul_byte` can be used to test for a nul byte.

Value

A 'recvAio' (object of class 'recvAio') (invisibly).

Examples

```
req <- socket("req", listen = "tcp://127.0.0.1:6546")
rep <- socket("rep", dial = "tcp://127.0.0.1:6546")

ctxq <- context(req)
ctxp <- context(rep)

# works if req and rep are running in parallel in different processes
reply(ctxp, execute = function(x) x + 1, timeout = 10)
aio <- request(ctxq, data = 2022, timeout = 10)
call_aio(aio)$data

close(req)
close(rep)
```

send *Send*

Description

Send data over a connection (Socket, Context or Stream).

Usage

```
send(con, data, mode = c("serial", "raw"), block = NULL)
```

Arguments

con	a Socket, Context or Stream.
data	an object (a vector, if mode = 'raw').
mode	[default 'serial'] for sending serialised R objects, or 'raw' for sending vectors of any type (converted to a raw byte vector for sending). For Streams, 'raw' is the only option and this argument is ignored. Use 'serial' for perfect reproducibility within R, although 'raw' must be used when interfacing with external applications that do not understand R serialisation. Alternatively, for performance, specify an integer position in the vector of choices i.e. 1L for 'serial' or 2L for 'raw'.
block	[default NULL] which applies the connection default (see section 'Blocking' below). Specify logical TRUE to block until successful or FALSE to return immediately even if unsuccessful (e.g. if no connection is available), or else an integer value specifying the maximum time to block in milliseconds, after which the operation will time out.

Value

Integer exit code (zero on success).

Blocking

For Sockets: the default behaviour is non-blocking with `block = FALSE`. This will return immediately with an error if the message could not be queued for sending. Certain protocol / transport combinations may limit the number of messages that can be queued if they have yet to be received.

For Contexts and Streams: the default behaviour is blocking with `block = TRUE`. This will wait until the send has completed. Set a timeout in this case to ensure that the function returns under all scenarios. As the underlying implementation uses an asynchronous send with a wait, it is recommended to set a positive integer value for `block` rather than FALSE.

Examples

```
pub <- socket("pub", dial = "inproc://nanonext")

send(pub, data.frame(a = 1, b = 2))
send(pub, c(10.1, 20.2, 30.3), mode = "raw", block = 100)

close(pub)

req <- socket("req", listen = "inproc://nanonext")
rep <- socket("rep", dial = "inproc://nanonext")

ctx <- context(req)
send(ctx, data.frame(a = 1, b = 2), block = 100)

msg <- recv_aio(rep, timeout = 100)
send(ctx, c(1.1, 2.2, 3.3), mode = "raw", block = 100)

close(req)
close(rep)
```

---

send_aio *Send Async*

Description

Send data asynchronously over a connection (Socket, Context or Stream).

Usage

```
send_aio(con, data, mode = c("serial", "raw"), timeout = NULL)
```

Arguments

con	a Socket, Context or Stream.
data	an object (a vector, if mode = 'raw').
mode	[default 'serial'] for sending serialised R objects, or 'raw' for sending vectors of any type (converted to a raw byte vector for sending). For Streams, 'raw' is the only option and this argument is ignored. Use 'serial' for perfect reproducibility within R, although 'raw' must be used when interfacing with external applications that do not understand R serialisation. Alternatively, for performance, specify an integer position in the vector of choices i.e. 1L for 'serial' or 2L for 'raw'.
timeout	[default NULL] integer value in milliseconds or NULL, which applies a socket-specific default, usually the same as no timeout.

Details

Async send is always non-blocking and returns a 'sendAio' immediately.

For a 'sendAio', the send result is available at `$result`. An 'unresolved' logical NA is returned if the async operation is yet to complete, The resolved value will be zero on success, or else an integer error code.

To wait for and check the result of the send operation, use `call_aio` on the returned 'sendAio' object.

Alternatively, to stop the async operation, use `stop_aio`.

Value

A 'sendAio' (object of class 'sendAio') (invisibly).

Examples

```
pub <- socket("pub", dial = "inproc://nanonext")

res <- send_aio(pub, data.frame(a = 1, b = 2), timeout = 100)
res
res$result

res <- send_aio(pub, "example message", mode = "raw", timeout = 100)
call_aio(res)$result

close(pub)
```

---

setopt *Set Option on Socket, Context, Stream, Listener or Dialer*

Description

Set opts on a Socket, Context, Stream, Listener or Dialer.

Usage

```
setopt(object, opt, value)
```

Arguments

`object`	a Socket, Context, Stream, Listener or Dialer.
`opt`	name of option, e.g. 'reconnect-time-min', as a character string. See opts.
`value`	value of option. Supply character type for 'string' options, integer or double for 'int', 'duration', 'size' and 'uint64', and logical for 'bool'.

Details

Note: once a dialer or listener has started, it is not generally possible to change its configuration. Hence create the dialer or listener with 'autostart = FALSE' if configuration needs to be set.

To set options on a Listener or Dialer attached to a Socket or nano object, pass in the objects directly via for example `$listener[[1]]` for the first Listener.

Value

Invisibly, an integer exit code (zero on success).

Examples

```
s <- socket("pair")
setopt(s, "recv-timeout", 2000)
close(s)

s <- socket("req")
ctx <- context(s)
setopt(ctx, "send-timeout", 2000)
close(ctx)
close(s)

s <- socket("pair", dial = "inproc://nanonext", autostart = FALSE)
setopt(s$dialer[[1]], "reconnect-time-min", 2000)
start(s$dialer[[1]])
close(s)

s <- socket("pair", listen = "inproc://nanonext", autostart = FALSE)
setopt(s$listener[[1]], "recv-size-max", 1024)
start(s$listener[[1]])
close(s)
```

---

sha256 *Cryptographic Hashing Using the SHA-2 Algorithms*

Description

Returns a SHA-256, SHA-224, SHA-384, or SHA-512 hash or HMAC of the supplied R object. Uses the optimised implementation from the Mbed TLS library.

Usage

```
sha256(x, key = NULL, convert = TRUE)

sha224(x, key = NULL, convert = TRUE)

sha384(x, key = NULL, convert = TRUE)

sha512(x, key = NULL, convert = TRUE)
```

Arguments

x	an object.
key	(optional) supply a secret key to generate an HMAC. If missing or NULL, the SHA-256/224/384/512 hash of 'x' is returned.
convert	[default TRUE] logical value whether to convert the output to a character string or keep as a raw vector. Supplying a non-logical value will error.

Details

For arguments 'x' and 'key', a raw vector is hashed directly, a scalar character string is translated to raw before hashing, whilst all other objects are serialised first.

The result of hashing is always a raw vector, which is translated to a character string if 'convert' is TRUE, or returned directly if 'convert' is FALSE.

Value

A raw vector or character string depending on 'convert', of byte length 32 for SHA-256, 28 for SHA-224, 48 for SHA-384, and 64 for SHA-512.

Examples

```
# SHA-256 hash as character string:
sha256("hello world!")

# SHA-256 hash as raw vector:
sha256("hello world!", convert = FALSE)

# Obtain HMAC:
sha256("hello world!", "SECRET_KEY")

# Hashing a file:
tempfile <- tempfile()
cat(rep(letters, 256), file = tempfile)
con <- file(tempfile, open = "rb")
vec <- NULL
while (length(upd <- readBin(con, raw(), 8192))) vec <- c(vec, upd)
sha256(vec)
close(con)
unlink(tempfile)

# SHA-224 hash:
sha224("hello world!")

# SHA-384 hash:
sha384("hello world!")

# SHA-512 hash:
sha512("hello world!")
```

`socket` *Open Socket*

Description

Open a Socket implementing 'protocol', and optionally dial (establish an outgoing connection) or listen (accept an incoming connection) at an address.

Usage

```
socket(
 protocol = c("bus", "pair", "push", "pull", "pub", "sub", "req", "rep", "surveyor",
    "respondent"),
  dial = NULL,
  listen = NULL,
  autostart = TRUE,
  raw = FALSE
)
```

Arguments

`protocol`	[default 'bus'] choose protocol - 'bus', 'pair', 'push', 'pull', 'pub', 'sub', 'req', 'rep', 'surveyor', or 'respondent' - see protocols.
`dial`	(optional) a URL to dial, specifying the transport and address as a character string e.g. 'inproc://anyvalue' or 'tcp://127.0.0.1:5555' (see transports).
`listen`	(optional) a URL to listen at, specifying the transport and address as a character string e.g. 'inproc://anyvalue' or 'tcp://127.0.0.1:5555' (see transports).
`autostart`	[default TRUE] whether to start the dialer/listener. Set to FALSE if you wish to set configuration options on the dialer/listener as it is not generally possible to change these once started. For dialers only: set to NA to start synchronously - this is less resilient if a connection is not immediately possible, but avoids subtle errors from attempting to use the socket before an asynchronous dial has completed.
`raw`	[default FALSE] whether to open raw mode sockets. Note: not for general use - do not enable unless you have a specific need, such as for use with `device` (refer to NNG documentation).

Details

NNG presents a socket view of networking. The sockets are constructed using protocol-specific functions, as a given socket implements precisely one protocol.

Each socket may be used to send and receive messages (if the protocol supports it, and implements the appropriate protocol semantics). For example, sub sockets automatically filter incoming messages to discard those for topics that have not been subscribed.

This function (optionally) binds a single Dialer and/or Listener to a Socket. More complex network topologies may be created by binding further Dialers/Listeners to the Socket as required using `dial` and `listen`. New contexts can also be created using `context` if the protocol supports it.

Value

A Socket (object of class 'nanoSocket' and 'nano').

Protocols

The following Scalability Protocols (communication patterns) are implemented:

- Bus (mesh networks) - protocol: 'bus'
- Pair (two-way radio) - protocol: 'pair'
- Pipeline (one-way pipe) - protocol: 'push', 'pull'
- Publisher/Subscriber (topics & broadcast) - protocol: 'pub', 'sub'
- Request/Reply (RPC) - protocol: 'req', 'rep'
- Survey (voting & service discovery) - protocol: 'surveyor', 'respondent'

Please see protocols for further documentation.

Examples

```
socket <- socket("pair")
socket
close(socket)
```

---

start *Start Listener/Dialer*

Description

Start a Listener/Dialer.

Usage

```
## S3 method for class 'nanoListener'
start(x, ...)

## S3 method for class 'nanoDialer'
start(x, async = TRUE, ...)
```

Arguments

x	a Listener or Dialer.
...	not used.
async	[default TRUE] (applicable to Dialers only) logical flag whether the connection attempt, including any name resolution, is to be made asynchronously. This behaviour is more resilient, but also generally makes diagnosing failures somewhat more difficult. If FALSE, failure, such as if the connection is refused, will be returned immediately, and no further action will be taken. Supplying a non-logical value will error.

Value

Invisibly, an integer exit code (zero on success).

`status_code` *Translate HTTP Status Codes*

Description

Provides an explanation for HTTP response status codes (in the range 100 to 599). If the status code is not defined as per RFC 9110, 'Non-standard Response' is returned, which may be a custom code used by the server.

Usage

```
status_code(x)
```

Arguments

`x`	numeric HTTP status code to translate.

Value

A character vector.

Examples

```
status_code(200)
status_code(404)
```

`stop_aio` *Stop Asynchronous Aio Operation*

Description

Stop an asynchronous Aio operation.

Usage

```
stop_aio(aio)
```

Arguments

`aio`	an Aio (object of class 'sendAio' or 'recvAio').

Details

Stops the asynchronous I/O operation associated with 'aio' by aborting, and then waits for it to complete or to be completely aborted. The Aio is then deallocated and no further operations may be performed on it.

Note this function operates silently and does not error even if 'aio' is not an active Aio, always returning invisible NULL.

Value

Invisible NULL.

---

stream *Open Stream*

Description

Open a Stream by either dialing (establishing an outgoing connection) or listening (accepting an incoming connection) at an address. This is a low-level interface intended for communicating with non-NNG endpoints.

Usage

```
stream(dial = NULL, listen = NULL, textframes = FALSE, pem = NULL)
```

Arguments

`dial`	a URL to dial, specifying the transport and address as a character string e.g. 'ipc:///tmp/anyvalue' or 'tcp://127.0.0.1:5555' (not all transports are supported).
`listen`	a URL to listen at, specifying the transport and address as a character string e.g. 'ipc:///tmp/anyvalue' or 'tcp://127.0.0.1:5555' (not all transports are supported).
`textframes`	[default FALSE] applicable to the websocket transport only, enables sending and receiving of TEXT frames (ignored otherwise). Supplying a non-logical value will error.
`pem`	(optional) applicable to secure websockets only. The path to a file containing X.509 certificate(s) in PEM format, comprising the certificate authority certificate chain (and revocation list if present). If missing or NULL, certificates are not validated.

Details

A Stream is used for raw byte stream connections. Byte streams are reliable in that data will not be delivered out of order, or with portions missing.

Can be used to dial a (secure) websocket address starting 'ws://' or 'wss://'. It is often the case that 'textframes' needs to be set to TRUE.

Specify only one of 'dial' or 'listen'. If both are specified, 'listen' will be ignored.

Value

A Stream (object of class 'nanoStream' and 'nano').

Examples

```
# will succeed only if there is an open connection at the address:
s <- tryCatch(stream(dial = "tcp://127.0.0.1:5555"), error = identity)
s
```

`subscribe` *Subscribe Topic*

Description

For a socket or context using the sub protocol in a publisher/subscriber pattern. Set a topic to subscribe to.

Usage

```
subscribe(con, topic = NULL)
```

Arguments

`con`	a Socket or Context using the 'sub' protocol.
`topic`	[default NULL] an atomic type or NULL. The default NULL subscribes to all topics.

Details

To use pub/sub the publisher must:

- specify `mode = 'raw'` when sending.
- ensure the sent vector starts with the topic.

The subscriber should then receive specifying the correct mode.

Value

Invisibly, an integer exit code (zero on success).

Examples

```
pub <- socket("pub", listen = "inproc://nanonext")
sub <- socket("sub", dial = "inproc://nanonext")

subscribe(sub, "examples")

send(pub, c("examples", "this is an example"), mode = "raw")
recv(sub, "character")
send(pub, "examples will also be received", mode = "raw")
recv(sub, "character")
send(pub, c("other", "this other topic will not be received"), mode = "raw")
recv(sub, "character")

subscribe(sub, 2)
send(pub, c(2, 10, 10, 20), mode = "raw")
recv(sub, "double", keep.raw = FALSE)

close(pub)
close(sub)
```

---

survey_time *Set Survey Time*

Description

For a socket or context using the surveyor protocol in a surveyor/respondent pattern. Set a survey timeout in ms (remains valid for all subsequent surveys). Messages received by the surveyor after the timer has ended are discarded.

Usage

```
survey_time(con, time)
```

Arguments

con	a Socket or Context using the 'surveyor' protocol.
time	the survey timeout in ms.

Details

After using this function, to start a new survey, the surveyor must:

- send a message.
- switch to receiving responses.

To respond to a survey, the respondent must:

- receive the survey message.
- send a reply using send_aio before the survey has timed out (a reply can only be sent after receiving a survey).

Value

Invisibly, an integer exit code (zero on success).

Examples

```
sur <- socket("surveyor", listen = "inproc://nanonext")
res <- socket("respondent", dial = "inproc://nanonext")

survey_time(sur, 1000)
send(sur, "reply to this survey")
aio <- recv_aio(sur)

recv(res)
s <- send_aio(res, "replied")

call_aio(aio)$data

close(sur)
close(res)
```

`transact` *ncurl Transact*

Description

nano cURL - a minimalist http(s) client. Transact once over the connection and stored parameters in an ncurl Session created with `ncurl_session`.

Usage

```
transact(session)
```

Arguments

`session`	an 'ncurlSession' object.

Value

Named list of 4 elements:

- `$status` - integer HTTP repsonse status code (200 - OK). Use `status_code` for a translation of the meaning.
- `$headers` - named list of response headers (if specified in the session), or NULL otherwise. If the status code is within the 300 range, i.e. a redirect, the response header 'Location' is automatically appended to return the redirect address.
- `$raw` - raw vector of the received resource (use `writeBin` to save to a file).
- `$data` - converted character string (if specified in the session), or NULL otherwise. This may be further parsed this as html, json, xml etc. if required.

Examples

```
s <- tryCatch(ncurl_session("https://httpbin.org/get"), error = identity)
if (!inherits(s, "error")) transact(s)
```

`transports` *Transports [Documentation]*

Description

Transports supported by {nanonext}.

For an authoritative guide please refer to the online documentation for the NNG library at https://nng.nanomsg.org/man/.

Inproc

The inproc transport provides communication support between sockets within the same process. This may be used as an alternative to slower transports when data must be moved within the same process. This transport tries hard to avoid copying data, and thus is very light-weight.

[**URI, inproc://**] This transport uses URIs using the scheme inproc://, followed by an arbitrary string of text, terminated by a NUL byte. inproc://nanonext is a valid example URL.

- Multiple URIs can be used within the same application, and they will not interfere with one another.
- Two applications may also use the same URI without interfering with each other, and they will be unable to communicate with each other using that URI.

IPC

The ipc transport provides communication support between sockets within different processes on the same host. For POSIX platforms, this is implemented using UNIX domain sockets. For Windows, this is implemented using Windows Named Pipes. Other platforms may have different implementation strategies.

Traditional Names

[**URI, ipc://**] This transport uses URIs using the scheme ipc://, followed by a path name in the file system where the socket or named pipe should be created.

- On POSIX platforms, the path is taken literally, and is relative to the current directory, unless it begins with /, in which case it is relative to the root directory. For example, ipc://nanonext refers to the name nanonext in the current directory, whereas ipc:///tmp/nanonext refers to nanonext located in /tmp.
- On Windows, all names are prefixed by \.\ pipe\ and do not reside in the normal file system - the required prefix is added automatically by NNG, so a URL of the form ipc://nanonext is fine.

UNIX Aliases

[**URI, unix://**] The unix:// scheme is an alias for ipc:// and can be used inter-changeably, but only on POSIX systems. The purpose of this scheme is to support a future transport making use of AF_UNIX on Windows systems, at which time it will be necessary to discriminate between the Named Pipes and the AF_UNIX based transports.

Abstract Names

[**URI, abstract://**] On Linux, this transport also can support abstract sockets. Abstract sockets use a URI-encoded name after the scheme, which allows arbitrary values to be conveyed in the path, including embedded NUL bytes. abstract://nanonext is a valid example URL.

- Abstract sockets do not have any representation in the file system, and are automatically freed by the system when no longer in use. Abstract sockets ignore socket permissions, but it is still possible to determine the credentials of the peer.

TCP/IP

The tcp transport provides communication support between sockets across a TCP/IP network. Both IPv4 and IPv6 are supported when the underlying platform also supports it.

[**URI, tcp://**] This transport uses URIs using the scheme tcp://, followed by an IP address or hostname, followed by a colon and finally a TCP port number. For example, to contact port 80 on the localhost either of the following URIs could be used: tcp://127.0.0.1:80 or tcp://localhost:80.

- A URI may be restricted to IPv6 using the scheme tcp6://, and may be restricted to IPv4 using the scheme tcp4://
- Note: Specifying tcp6:// may not prevent IPv4 hosts from being used with IPv4-in-IPv6 addresses, particularly when using a wildcard hostname with listeners. The details of this varies across operating systems.
- Note: both tcp6:// and tcp4:// are specific to NNG, and might not be understood by other implementations.
- It is recommended to use either numeric IP addresses, or names that are specific to either IPv4 or IPv6 to prevent confusion and surprises.
- When specifying IPv6 addresses, the address must be enclosed in square brackets ([]) to avoid confusion with the final colon separating the port. For example, the same port 80 on the IPv6 loopback address (::1) would be specified as tcp://[::1]:80.
- The special value of 0 (INADDR_ANY) can be used for a listener to indicate that it should listen on all interfaces on the host. A short-hand for this form is to either omit the address, or specify the asterisk (*) character. For example, the following three URIs are all equivalent, and could be used to listen to port 9999 on the host: (1) tcp://0.0.0.0:9999 (2) tcp://*:9999 (3) tcp://:9999

WebSocket

The ws transport provides communication support between peers across a TCP/IP network using WebSockets. Both IPv4 and IPv6 are supported when the underlying platform also supports it.

[**URI, ws://**] This transport uses URIs using the scheme ws://, followed by an IP address or hostname, optionally followed by a colon and a TCP port number, optionally followed by a path. (If

no port number is specified then port 80 is assumed. If no path is specified then a path of / is assumed.) For example, the URI ws://localhost/app/pubsub would use port 80 on localhost, with the path /app/pubsub.

- When specifying IPv6 addresses, the address must be enclosed in square brackets ([]) to avoid confusion with the final colon separating the port. For example, the same path and port on the IPv6 loopback address (::1) would be specified as ws://[::1]/app/pubsub.
- Note: The value specified as the host, if any, will also be used in the Host: HTTP header during HTTP negotiation.
- To listen to all ports on the system, the host name may be elided from the URL on the listener. This will wind up listening to all interfaces on the system, with possible caveats for IPv4 and IPv6 depending on what the underlying system supports. (On most modern systems it will map to the special IPv6 address ::, and both IPv4 and IPv6 connections will be permitted, with IPv4 addresses mapped to IPv6 addresses.)
- This transport makes use of shared HTTP server instances, permitting multiple sockets or listeners to be configured with the same hostname and port. When creating a new listener, it is registered with an existing HTTP server instance if one can be found. Note that the matching algorithm is somewhat simple, using only a string based hostname or IP address and port to match. Therefore it is recommended to use only IP addresses or the empty string as the hostname in listener URLs.
- All sharing of server instances is only typically possible within the same process.
- The server may also be used by other things (for example to serve static content), in the same process.

`unresolved` *Query if an Aio is Unresolved*

Description

Query whether an Aio or Aio value remains unresolved. Unlike `call_aio`, this function does not wait for completion.

Usage

```
unresolved(aio)
```

Arguments

`aio`	an Aio (object of class 'sendAio' or 'recvAio'), or Aio value stored in `$result`, `$raw` or `$data` as the case may be.

Details

Suitable for use in control flow statements such as `while` or `if`.

Note: querying resolution may cause a previously unresolved Aio to resolve.

Value

Logical TRUE if 'aio' is an unresolved Aio or Aio value, or FALSE otherwise.

Examples

```
s1 <- socket("pair", listen = "inproc://nanonext")
aio <- send_aio(s1, "test", timeout = 100)

while (unresolved(aio)) {
  # do stuff before checking resolution again
  cat("unresolved\n")
  s2 <- socket("pair", dial = "inproc://nanonext")
  Sys.sleep(0.01)
}

unresolved(aio)

close(s1)
close(s2)
```

`unsubscribe` *Unsubscribe Topic*

Description

For a socket or context using the sub protocol in a publisher/subscriber pattern. Remove a topic from the subscription list.

Usage

```
unsubscribe(con, topic = NULL)
```

Arguments

`con`	a Socket or Context using the 'sub' protocol.
`topic`	[default NULL] an atomic type or NULL. The default NULL unsubscribes from all topics (if all topics were previously subscribed).

Details

Note that if the topic was not previously subscribed to then an 'entry not found' error will result.

To use pub/sub the publisher must:

- specify `mode = 'raw'` when sending.
- ensure the sent vector starts with the topic.

The subscriber should then receive specifying the correct mode.

Value

Invisibly, an integer exit code (zero on success).

Examples

```
pub <- socket("pub", listen = "inproc://nanonext")
sub <- socket("sub", dial = "inproc://nanonext")

subscribe(sub, "examples")
send(pub, c("examples", "this is an example"), mode = "raw")
recv(sub, "character")
unsubscribe(sub, "examples")
send(pub, c("examples", "this example will not be received"), mode = "raw")
recv(sub, "character")

subscribe(sub, 2)
send(pub, c(2, 10, 10, 20), mode = "raw")
recv(sub, "double", keep.raw = FALSE)
unsubscribe(sub, 2)
send(pub, c(2, 10, 10, 20), mode = "raw")
recv(sub, "double", keep.raw = FALSE)

close(pub)
close(sub)
```

Index