Package: nanonext (via r-universe)

October 30, 2026

Type Package

Title Lightweight Toolkit for Messaging, Concurrency and the Web

Version 1.10.4

Description R binding for NNG (Nanomsg Next Gen), a successor to ZeroMQ. A toolkit for messaging, concurrency and the web. High-performance socket messaging over in-process, IPC, TCP, WebSocket and secure TLS transports implements 'Scalability Protocols', a standard for common communications patterns including publish/subscribe, request/reply and survey. A threaded concurrency framework with intuitive 'aio' objects that resolve automatically upon completion of asynchronous operations, and synchronisation primitives that allow R to wait on events signalled by concurrent threads. A unified HTTP server hosting REST endpoints, WebSocket connections and streaming on a single port, with a built-in HTTP client.

License MIT + file LICENSE

URL https://nanonext.r-lib.org, https://github.com/r-lib/nanonext

BugReports https://github.com/r-lib/nanonext/issues

Depends R (>= 3.6)

Suggests later, litedown, yaml

Enhances promises

VignetteBuilder litedown

Biarch true

Config/build/compilation-database true

Config/Needs/website tidyverse/tidytemplate

Config/roxygen2/markdown TRUE

Config/roxygen2/version 8.1.0

Config/usethis/last-upkeep 2026-07-11

Encoding UTF-8

SystemRequirements 'libnng' >= 1.12 and 'libmbedtls' >= 3.0 (optional, bundled sources compiled otherwise)

NeedsCompilation yes

Author Charlie Gao [aut, cre] (ORCID: <https://orcid.org/0000-0002-0750-061X>), Posit Software, PBC [cph, fnd] (ROR: <https://ror.org/03wc8by49>), Hibiki AI Limited [cph], Staysail Systems, Inc. [cph] (NNG library), Capitar IT Group BV [cph] (NNG library), The Mbed TLS Contributors [cph] (Mbed TLS library), Pierre L'Ecuyer [cph] (RngStreams library), sakura authors [cph] (Serialization hooks), R Consortium [fnd] (ROR: <https://ror.org/01z833950>)

Maintainer Charlie Gao <charlie.gao@posit.co>

Repository https://cran.r-universe.dev

Date/Publication 2026-10-30 19:34:38 UTC

RemoteUrl https://github.com/cran/nanonext

RemoteRef HEAD

RemoteSha f213d0783dc960084c3c88f363a38e3aca9de0e1

Contents

nanonext-package 4
%~>% 6
as.promise.ncurlAio 7
as.promise.recvAio 7
call_aio 8
close.nanoContext 9
collect_aio 10
context 11
cv 12
device_aio 14
dial 15
format_sse 17
handler 19
handler_directory 20
handler_file 21
handler_inline 22
handler_redirect 22
handler_stream 23
handler_ws 25
http_server 27
ip_addr 29
is_aio 30
is_error_value 31
listen 32
mclock 33

monitor . . . 34
msleep . . . 35
nano . . . 36
ncurl . . . 37
ncurl_aio . . . 39
ncurl_session . . . 42
nng_error . . . 43
nng_version . . . 44
opt . . . 45
parse_url . . . 49
pipe_id . . . 50
pipe_notify . . . 51
protocols . . . 52
race_aio . . . 54
random . . . 54
read_stdin . . . 55
reap . . . 56
recv . . . 57
recv_aio . . . 59
reply . . . 61
request . . . 62
run_event_loop . . . 65
send . . . 65
send_aio . . . 67
serial_config . . . 68
socket . . . 69
start . . . 71
stat . . . 72
status_code . . . 74
stop_aio . . . 74
stream . . . 75
subscribe . . . 76
survey_time . . . 78
tls_config . . . 79
transports . . . 80
unresolved . . . 83
write_cert . . . 84
write_stdout . . . 85

Index **86**

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 for reliable, high-performance messaging over in-process, IPC, TCP, WebSocket and secure TLS transports. Implements 'Scalability Protocols', a standard for common communications patterns including publish/subscribe, request/reply and service discovery. As its own threaded concurrency framework, provides a toolkit for asynchronous programming and distributed computing. Intuitive 'aio' objects resolve automatically when asynchronous operations complete, and synchronisation primitives allow R to wait upon events signalled by concurrent threads.

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

Guide to the options that can be inspected and set using: opt / opt<-

Reference Manual

```
vignette("nanonext", package = "nanonext")
```

Conceptual overview

NNG presents a socket view of networking. A socket implements precisely one protocol, such as 'bus', etc.

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

NNG sockets are message-oriented, and 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 use 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 is the service address that is contacted, whereas for listeners this is where new connections will be accepted.

Links

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

Author(s)

Maintainer: Charlie Gao <charlie.gao@posit.co> (ORCID)

Authors:

- Charlie Gao <charlie.gao@posit.co> (ORCID)

Other contributors:

- Posit Software, PBC (ROR) [copyright holder, funder]
- Hibiki AI Limited [copyright holder]
- Staysail Systems, Inc. (NNG library) [copyright holder]
- Capitar IT Group BV (NNG library) [copyright holder]
- The Mbed TLS Contributors (Mbed TLS library) [copyright holder]
- Pierre L'Ecuyer (RngStreams library) [copyright holder]
- sakura authors (Serialization hooks) [copyright holder]
- R Consortium (ROR) [funder]

See Also

Useful links:

- https://nanonext.r-lib.org
- https://github.com/r-lib/nanonext
- Report bugs at https://github.com/r-lib/nanonext/issues

`%~>%` *Signal Forwarder*

Description

Forwards signals from one 'conditionVariable' to another.

Usage

```
cv %~>% cv2
```

Arguments

`cv`	a 'conditionVariable' object, from which to forward the signal.
`cv2`	a 'conditionVariable' object, to which the signal is forwarded.

Details

The condition value of `cv` is initially reset to zero when this operator returns. Only one forwarder can be active on a `cv` at any given time, and assigning a new forwarding target cancels any currently existing forwarding.

Changes in the condition value of `cv` are forwarded to `cv2`, but only on each occassion `cv` is signalled. This means that waiting on `cv` will cause a temporary divergence between the actual condition value of `cv` and that recorded at `cv2`, until the next time `cv` is signalled.

Value

Invisibly, `cv2`.

Examples

```
cva <- cv(); cvb <- cv(); cv1 <- cv(); cv2 <- cv()

cva %~>% cv1 %~>% cv2
cvb %~>% cv2

cv_signal(cva)
cv_signal(cvb)
cv_value(cv1)
cv_value(cv2)
```

as.promise.ncurlAio *Make ncurlAio Promise*

Description

Creates a 'promise' from an 'ncurlAio' object.

Usage

```
## S3 method for class 'ncurlAio'
as.promise(x)
```

Arguments

x	an object of class 'ncurlAio'.

Details

This function is an S3 method for the generic `as.promise` for class 'ncurlAio'.

Requires the **promises** package.

Allows an 'ncurlAio' to be used with functions such as `promises::then()`, which schedules a function to run upon resolution of the Aio.

The promise is resolved with a list of 'status', 'headers' and 'data' or rejected with the error translation if an NNG error is returned e.g. for an invalid address.

Value

A 'promise' object.

as.promise.recvAio *Make recvAio Promise*

Description

Creates a 'promise' from an 'recvAio' object.

Usage

```
## S3 method for class 'recvAio'
as.promise(x)
```

Arguments

x	an object of class 'recvAio'.

Details

This function is an S3 method for the generic `as.promise` for class 'recvAio'.

Requires the **promises** package.

Allows a 'recvAio' to be used with the promise pipe `%...>%`, which schedules a function to run upon resolution of the Aio.

Value

A 'promise' object.

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

Description

`call_aio` retrieves the value of an asynchronous Aio operation, waiting for the operation to complete if still in progress. For a list of Aios, waits for all asynchronous operations to complete before returning.

`call_aio_` is a variant that allows user interrupts, suitable for interactive use.

Usage

```
call_aio(x)

call_aio_(x)
```

Arguments

`x`	an Aio or list of Aios (objects of class 'sendAio', 'recvAio' or 'ncurlAio').

Details

For a 'recvAio', the received value may be retrieved 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', if an error occurred in unserialization or conversion of the message data to the specified mode, a raw vector will be returned instead to allow recovery (accompanied by a warning).

Note: this function operates silently and does not error even if `x` is not an active Aio or list of Aios, always returning invisibly the passed object.

Value

The passed object (invisibly).

Alternatively

Aio values may be accessed directly at `$result` for a 'sendAio', and `$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, Stream, Pipe, or ncurl Session.

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 'nanoServer'
close(con, ...)
```

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

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

Arguments

con	a Socket, Context, Dialer, Listener, Stream, or 'ncurlSession'.
...	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.

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

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

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.

Closing an 'ncurlSession' closes the http(s) connection.

Value

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

See Also

reap()

---

collect_aio *Collect Data of an Aio or List of Aios*

Description

collect_aio collects the data of an Aio or list of Aios, waiting for resolution if still in progress.

collect_aio_ is a variant that allows user interrupts, suitable for interactive use.

Usage

```
collect_aio(x)

collect_aio_(x)
```

Arguments

x	an Aio or list of Aios (objects of class 'sendAio', 'recvAio' or 'ncurlAio').

Details

This function will wait for the asynchronous operation(s) to complete if still in progress (blocking).

Using `x[]` on an Aio `x` is equivalent to the user-interruptible `collect_aio_(x)`.

Value

Depending on the type of `x` supplied, an object or list of objects (the same length as `x`, preserving names).

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)
collect_aio(res)

msg <- recv_aio(s2, timeout = 100)
collect_aio_(msg)

msg[]

close(s1)
close(s2)
```

`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 Context (object of class 'nanoContext' and 'nano').

See Also

`request()` and `reply()` for use with contexts.

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()
```

---

cv *Condition Variables*

Description

`cv` creates a new condition variable (protected by a mutex internal to the object).

`wait` waits on a condition being signalled by completion of an asynchronous receive or pipe event.
`wait_` is a variant that allows user interrupts, suitable for interactive use.

`until` waits until a future time on a condition being signalled by completion of an asynchronous receive or pipe event.
`until_` is a variant that allows user interrupts, suitable for interactive use.

`cv_value` inspects the internal value of a condition variable.

`cv_reset` resets the internal value and flag of a condition variable.

`cv_signal` signals a condition variable.

Usage

```
cv()

wait(cv)

wait_(cv)

until(cv, msec)

until_(cv, msec)

cv_value(cv)

cv_reset(cv)

cv_signal(cv)
```

Arguments

`cv`	a 'conditionVariable' object.
`msec`	maximum time in milliseconds to wait for the condition variable to be signalled.

Details

Pass the 'conditionVariable' to the asynchronous receive functions `recv_aio()` or `request()`. Alternatively, to be notified of a pipe event, pass it to `pipe_notify()`.

Completion of the receive or pipe event, which happens asynchronously and independently of the main R thread, will signal the condition variable by incrementing it by 1.

This will cause the R execution thread waiting on the condition variable using `wait()` or `until()` to wake and continue.

For argument `msec`, non-integer values will be coerced to integer. Non-numeric input will be ignored and return immediately.

Value

For **cv**: a 'conditionVariable' object.

For **wait**: (invisibly) logical TRUE, or else FALSE if a flag has been set.

For **until**: (invisibly) logical TRUE if signalled, or else FALSE if the timeout was reached.

For **cv_value**: integer value of the condition variable.

For **cv_reset** and **cv_signal**: zero (invisibly).

Condition

The condition internal to this 'conditionVariable' maintains a state (value). Each signal increments the value by 1. Each time `wait()` or `until()` returns (apart from due to timeout), the value is decremented by 1.

The internal condition may be inspected at any time using `cv_value()` and reset using `cv_reset()`. This affords a high degree of flexibility in designing complex concurrent applications.

Flag

The condition variable also contains a flag that certain signalling functions such as `pipe_notify()` can set. When this flag has been set, all subsequent `wait()` calls will return logical FALSE instead of TRUE.

Note that the flag is not automatically reset, but may be reset manually using `cv_reset()`.

Examples

```
cv <- cv()

## Not run:
wait(cv) # would block until the cv is signalled
wait_(cv) # would block until the cv is signalled or interrupted

## End(Not run)

until(cv, 10L)
until_(cv, 10L)

cv_value(cv)

cv_reset(cv)

cv_value(cv)
cv_signal(cv)
cv_value(cv)
```

`device_aio` *Device (Async)*

Description

Create an asynchronous device: a zero-copy forwarder that relays raw messages between two Sockets. This is the building block for brokers and proxies, such as a publisher/subscriber relay or a pair-to-pair bridge.

Usage

```
device_aio(s1, s2 = s1)
```

Arguments

s1	a Socket, in raw mode.
s2	a Socket, in raw mode, using the complementary protocol to s1. If not supplied, defaults to s1 to create a reflector.

Details

A device runs on background threads and relays messages until it is stopped or an error occurs. The returned 'sendAio' resolves only at that point: its $result is an 'unresolved' logical NA while the device is running, and otherwise the integer exit code (usually the error that caused it to stop, such as a Socket being closed).

Both Sockets must be opened in raw mode (see socket()) and use complementary protocols, that is the peer protocol of each must be the protocol of the other (for example two 'pair' Sockets, or a 'pub' and a 'sub' Socket). The same Socket may be supplied for both arguments to create a reflector.

To stop the device, use stop_aio(), or close either of the Sockets. To block and wait for the device to stop, use call_aio().

Value

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

See Also

send_aio() for the structure of the returned 'sendAio'.

Examples

```
s1 <- socket("pair", listen = "inproc://device1", raw = TRUE)
s2 <- socket("pair", listen = "inproc://device2", raw = TRUE)

dev <- device_aio(s1, s2)
dev$result

stop_aio(dev)
dev$result
close(s1)
close(s2)
```

---

dial *Dial an Address from a Socket*

Description

Creates a new Dialer and binds it to a Socket.

Usage

```
dial(
  socket,
  url = "inproc://nanonext",
  tls = NULL,
  autostart = TRUE,
  fail = c("warn", "error", "none")
)
```

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).
tls	[default NULL] for secure tls+tcp:// or wss:// connections only, provide a TLS configuration object created by `tls_config()`.
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.
fail	[default 'warn'] failure mode - a character value or integer equivalent, whether to warn (1L), error (2L), or for none (3L) just return an 'errorValue' without any corresponding warning.

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 `opt<-()` 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 = "inproc://nanodial", autostart = FALSE)
socket$dialer
start(socket$dialer[[1]])
socket$dialer
close(socket$dialer[[1]])
close(socket)

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

`format_sse` *Format Server-Sent Event*

Description

Helper function to format messages according to the Server-Sent Events (SSE) specification. Use with `handler_stream()` to create SSE endpoints.

Usage

```
format_sse(data, event = NULL, id = NULL, retry = NULL)
```

Arguments

`data`	The data payload. Will be prefixed with "data: " on each line.
`event`	[default NULL] Optional event type (e.g., "message", "error").
`id`	[default NULL] Optional event ID for client reconnection.
`retry`	[default NULL] Optional retry interval in milliseconds.

Details

Server-Sent Events is a W3C standard for server-to-client streaming over HTTP, supported natively by browsers via the `EventSource` API. SSE is commonly used for real-time updates, notifications, and LLM token streaming.

SSE messages have this format:

```
event: <event-type>
id: <event-id>
retry: <milliseconds>
data: <data-line-1>
data: <data-line-2>
```

Each message ends with two newlines. Multi-line data is split and each line prefixed with "data: ".

When using SSE with `handler_stream()`, set the appropriate headers:

- `Content-Type: text/event-stream`
- `Cache-Control: no-cache`
- `X-Accel-Buffering: no` (prevents proxy buffering)

Value

A character string formatted as an SSE message, ready to pass to `conn$send()`.

See Also

`handler_stream()` for creating streaming HTTP endpoints.

Examples

```
format_sse(data = "Hello")
#> "data: Hello\n\n"

format_sse(data = "Hello", event = "greeting")
#> "event: greeting\ndata: Hello\n\n"

format_sse(data = "Line 1\nLine 2")
#> "data: Line 1\ndata: Line 2\n\n"

# Typical SSE endpoint setup
h <- handler_stream("/events", function(conn, req) {
  conn$set_header("Content-Type", "text/event-stream")
  conn$set_header("Cache-Control", "no-cache")
  conn$set_header("X-Accel-Buffering", "no")
  conn$send(format_sse(data = "connected", id = "1"))
})
```

handler *Create HTTP Handler*

Description

Creates an HTTP route handler for use with `http_server()`.

Usage

```
handler(path, callback, method = "GET", prefix = FALSE)
```

Arguments

`path`	URI path to match (e.g., "/api/data", "/users").
`callback`	Function to handle requests. Receives a list with: • `method` - HTTP method (character) • `uri` - Request URI (character) • `headers` - Named character vector of headers • `body` - Request body (raw vector) Should return a list with: • `status` - HTTP status code (integer, default 200) • `headers` - Response headers as a named character vector, e.g. `c("Content-Type" = "application/json")` (optional) • `body` - Response body (character or raw)
`method`	[default "GET"] HTTP method to match (e.g., "GET", "POST", "PUT", "DELETE"). Use `"*"` to match any method.
`prefix`	[default FALSE] Logical, if TRUE matches path as a prefix (e.g., "/api" will match "/api/users", "/api/items", etc.).

Details

If the callback throws an error, a 500 Internal Server Error response is returned to the client.

Value

A handler object for use with `http_server()`.

See Also

`handler_ws()` for WebSocket handlers. Static handlers: `handler_file()`, `handler_directory()`, `handler_inline()`, `handler_redirect()`.

Examples

```
# Simple GET handler
h1 <- handler("/hello", function(req) {
  list(status = 200L, body = "Hello!")
})

# POST handler that echoes the request body
h2 <- handler("/echo", function(req) {
  list(status = 200L, body = req$body)
}, method = "POST")

# Catch-all handler for a path prefix
h3 <- handler("/static", function(req) {
  # Serve static files under /static/*
}, method = "*", prefix = TRUE)
```

`handler_directory` *Create Static Directory Handler*

Description

Creates an HTTP handler that serves files from a directory tree. NNG handles MIME type detection automatically.

Usage

```
handler_directory(path, directory)
```

Arguments

`path`	URI path prefix (e.g., "/static"). Requests to "/static/foo.js" will serve "directory/foo.js".
`directory`	Path to the directory to serve.

Details

Directory handlers automatically match all paths under the prefix (prefix matching is implicit). The URI path is mapped to the filesystem:

- Request to "/static/css/style.css" serves "directory/css/style.css"
- Request to "/static/" serves "directory/index.html" if it exists

Note: The trailing slash behavior depends on how clients make requests. A request to "/static" (no trailing slash) will not automatically redirect to "/static/". Consider using `handler_redirect()` if you need this behavior.

Value

A handler object for use with `http_server()`.

Examples

```
h <- handler_directory("/static", "www/assets")
```

---

handler_file *Create Static File Handler*

Description

Creates an HTTP handler that serves a single file. NNG handles MIME type detection automatically.

Usage

```
handler_file(path, file, prefix = FALSE)
```

Arguments

`path`	URI path to match (e.g., "/favicon.ico").
`file`	Path to the file to serve.
`prefix`	[default FALSE] Logical, if TRUE matches path as a prefix.

Value

A handler object for use with `http_server()`.

Examples

```
h <- handler_file("/favicon.ico", "~/favicon.ico")
```

handler_inline *Create Inline Static Content Handler*

Description

Creates an HTTP handler that serves in-memory static content. Useful for small files like robots.txt or inline JSON/HTML.

Usage

```
handler_inline(path, data, content_type = NULL, prefix = FALSE)
```

Arguments

path	URI path to match (e.g., "/robots.txt").
data	Content to serve. Character data is converted to raw bytes.
content_type	MIME type (e.g., "text/plain", "application/json"). Defaults to "application/octet-stream" if NULL.
prefix	[default FALSE] Logical, if TRUE matches path as a prefix.

Value

A handler object for use with http_server().

Examples

```
h1 <- handler_inline("/robots.txt", "User-agent: *\nDisallow:",
                     content_type = "text/plain")
h2 <- handler_inline("/health", '{"status":"ok"}',
                     content_type = "application/json")
```

handler_redirect *Create HTTP Redirect Handler*

Description

Creates an HTTP handler that returns a redirect response.

Usage

```
handler_redirect(path, location, status = 302L, prefix = FALSE)
```

Arguments

path	URI path to match (e.g., "/old-page").
location	URL to redirect to. Can be relative (e.g., "/new-page") or absolute (e.g., "https://example.com/page").
status	HTTP redirect status code. Must be one of: • 301 - Moved Permanently • 302 - Found (default) • 303 - See Other • 307 - Temporary Redirect • 308 - Permanent Redirect
prefix	[default FALSE] Logical, if TRUE matches path as a prefix.

Value

A handler object for use with `http_server()`.

Examples

```
# Permanent redirect
h1 <- handler_redirect("/old", "/new", status = 301L)

# Redirect bare path to trailing slash
h2 <- handler_redirect("/app", "/app/")
```

---

handler_stream *Create HTTP Streaming Handler*

Description

Creates an HTTP streaming handler using chunked transfer encoding. Supports any streaming HTTP protocol including Server-Sent Events (SSE), newline- delimited JSON (NDJSON), and custom streaming formats.

Usage

```
handler_stream(path, on_request, on_close = NULL, method = "*", prefix = FALSE)
```

Arguments

path	URI path to match (e.g., "/stream").
on_request	Function called when a request arrives. Signature: `function(conn, req)` where conn is the connection object and `req` is a list with `method`, `uri`, `headers`, body.
on_close	[default NULL] Function called when the connection closes. Signature: `function(conn)`
method	[default "*"] HTTP method to match (e.g., "GET", "POST"). Use `"*"` to match any method.
prefix	[default FALSE] Logical, if TRUE matches path as a prefix.

Details

HTTP streaming uses chunked transfer encoding (RFC 9112). The first `$send()` triggers writing of HTTP headers with `Transfer-Encoding: chunked`. Headers cannot be modified after the first send.

Set an appropriate `Content-Type` header for your streaming format:

- NDJSON: `application/x-ndjson`
- JSON stream: `application/stream+json`
- SSE: `text/event-stream` (see `format_sse()`)
- Plain text: `text/plain`

SSE Reconnection: When an SSE client reconnects after a disconnect, it sends a `Last-Event-ID` header containing the last event ID it received. Access this via `req$headers["Last-Event-ID"]` in `on_request` to resume the event stream from the correct position.

To broadcast to multiple clients, store connection objects in a list and iterate over them (e.g., `lapply(conns, function(c) c$send(data))`).

Value

A handler object for use with `http_server()`.

Connection Object

The `conn` object passed to callbacks has these methods:

- `conn$send(data)`: Send data chunk to client.
- `conn$close()`: Close the connection (sends terminal chunk).
- `conn$set_status(code)`: Set HTTP status code (before first send).
- `conn$set_header(name, value)`: Set response header (before first send).
- `conn$id`: Unique connection identifier.

See Also

`format_sse()` for formatting Server-Sent Events.

Examples

```
# NDJSON streaming endpoint
h <- handler_stream("/stream", function(conn, req) {
  conn$set_header("Content-Type", "application/x-ndjson")
  conn$send('{"status":"connected"}\n')
})

# SSE endpoint with reconnection support
h <- handler_stream("/events", function(conn, req) {
  conn$set_header("Content-Type", "text/event-stream")
  conn$set_header("Cache-Control", "no-cache")
  last_id <- req$headers["Last-Event-ID"]
```

```
  # Resume from last_id if client is reconnecting
  conn$send(format_sse(data = "connected", id = "1"))
})

# Long-lived streaming with broadcast triggered by POST
conns <- list()
handlers <- list(
  handler_stream("/stream",
    on_request = function(conn, req) {
      conn$set_header("Content-Type", "application/x-ndjson")
      conns[[as.character(conn$id)]] <<- conn
      conn$send('{"status":"connected"}\n')
    },
    on_close = function(conn) {
      conns[[as.character(conn$id)]] <<- NULL
    }
  ),
  # POST endpoint triggers broadcast to all streaming clients
  handler("/broadcast", function(req) {
    msg <- paste0('{"msg":"', rawToChar(req$body), '"}\n')
    lapply(conns, function(c) c$send(msg))
    list(status = 200L, body = "sent")
  }, method = "POST")
)
```

`handler_ws` *Create WebSocket Handler*

Description

Creates a WebSocket handler for use with `http_server()`.

Usage

```
handler_ws(
  path,
  on_message,
  on_open = NULL,
  on_close = NULL,
  textframes = FALSE
)
```

Arguments

`path`	URI path for WebSocket connections (e.g., "/ws").
`on_message`	Function called when a message is received. Signature: `function(ws, data)` where `ws` is the connection object and `data` is the message. Use `ws$send()` to send responses; the return value is ignored.

on_open	[default NULL] Function called when a connection opens. Signature: `function(ws, req)` where `req` is a list with `uri` (character) and `headers` (named character vector) from the HTTP upgrade request.
on_close	[default NULL] Function called when a connection closes. Signature: `function(ws)`
textframes	[default FALSE] Logical, use text frames instead of binary. When TRUE: incoming `data` is character, outgoing data should be character. When FALSE: incoming `data` is raw vector, outgoing data should be raw vector.

Value

A handler object for use with `http_server()`.

Connection Object

The `ws` object passed to callbacks has the following fields and methods:

- `ws$send(data)`: Send a message to the client. `data` can be a raw vector or character string. Returns 0 on success, or an error code on failure (e.g., if the connection is closed).
- `ws$close()`: Close the connection.
- `ws$id`: Unique integer identifier for this connection. No two connections on the same server will share an ID, even across different handlers, making IDs safe to use as keys in a shared data structure.

Examples

```
# Simple echo server
h <- handler_ws("/ws", function(ws, data) ws$send(data))

# With connection tracking
clients <- list()
h <- handler_ws(
  "/chat",
  on_message = function(ws, data) {
    # Broadcast to all
    for (client in clients) client$send(data)
  },
  on_open = function(ws, req) {
    clients[[as.character(ws$id)]] <<- ws
  },
  on_close = function(ws) {
    clients[[as.character(ws$id)]] <<- NULL
  },
  textframes = TRUE
)
```

`http_server` *Create HTTP/WebSocket Server*

Description

Creates a server that can handle HTTP requests and WebSocket connections.

Usage

```
http_server(url, handlers = list(), tls = NULL)
```

Arguments

`url`	URL to listen on (e.g., "http://127.0.0.1:8080").
`handlers`	A handler or list of handlers created with `handler()`, `handler_ws()`, etc.
`tls`	TLS configuration for HTTPS/WSS, created via `tls_config()`.

Details

This function leverages NNG's shared HTTP server architecture. When both HTTP handlers and WebSocket handlers are provided, they share the same underlying server and port. WebSocket handlers automatically handle the HTTP upgrade handshake and all WebSocket framing (RFC 6455).

WebSocket and streaming callbacks are executed on R's main thread via the **later** package. To process callbacks, you must run the event loop (e.g., using `run_event_loop()` in a loop), or use `$serve()` which handles this automatically.

Requires the **later** package.

Value

A nanoServer object with methods:

- `$start()` - Start accepting connections
- `$close()` - Stop and release all resources
- `$serve()` - Start and block, processing requests via the **later** event loop until interrupted (e.g., Ctrl+C). The server is automatically closed on exit
- `$url` - The server URL

Deployment

nanonext implements the `_server.yml` standard for R server frameworks, via the internal `launch_server()` function. This allows a deployment platform to launch a server without any knowledge of the package internals, given a `_server.yml` file at the root of the deployment:

```
engine: nanonext
constructor: server.R
options:
  host: 127.0.0.1
  port: 8080
```

constructor is an R file whose final expression evaluates to a handler or list of handlers for this function. A complete example ships with the package at system.file("examples", "server-yml", package = "nanonext").

See vignette("v04-web", package = "nanonext") for the full specification of the supported fields.

Examples

```
# Simple HTTP server
server <- http_server(
  url = "http://127.0.0.1:8080",
  handlers = list(
    handler("/", function(req) {
      list(status = 200L, body = "Hello, World!")
    }),
    handler("/api/data", function(req) {
      list(
        status = 200L,
        headers = c("Content-Type" = "application/json"),
        body = '{"value": 42}'
      )
    })
  )
)
server$start()
# Run event loop: repeat run_event_loop(1000)
server$close()

# HTTP + WebSocket server
server <- http_server(
  url = "http://127.0.0.1:8080",
  handlers = list(
    handler("/", function(req) {
      list(status = 200L, body = "<html>...</html>")
    }),
    handler_ws("/ws", function(ws, data) {
      ws$send(data)  # Echo
    }, textframes = TRUE)
  )
)

# Multiple WebSocket endpoints
server <- http_server(
  url = "http://127.0.0.1:8080",
  handlers = list(
    handler_ws("/echo", function(ws, data) ws$send(data)),
    handler_ws("/upper", function(ws, data) ws$send(toupper(data)), textframes = TRUE)
  )
)

# HTTPS server with self-signed certificate
cert <- write_cert(cn = "127.0.0.1")
```

```
cfg <- tls_config(server = cert$server)
server <- http_server(
  url = "https://127.0.0.1:8443",
  handlers = list(
    handler("/", function(req) list(status = 200L, body = "Secure!"))
  ),
  tls = cfg
)
server$start()

# Send async request and run event loop
aio <- ncurl_aio(
  "https://127.0.0.1:8443/",
  tls = tls_config(client = cert$client),
  timeout = 2000
)
while (unresolved(aio)) run_event_loop(1000)

aio$status
aio$data

server$close()
```

---

ip_addr *IP Address*

Description

Returns a character string comprising the local network IPv4 address, or vector if there are multiple addresses from multiple network adapters, or an empty character string if unavailable.

Usage

```
ip_addr()
```

Details

The IP addresses will be named by interface (adapter friendly name on Windows) e.g. 'eth0' or 'en0'.

Value

A named character string.

Examples

```
ip_addr()
```

`is_aio` *Validators*

Description

Validator functions for object types created by **nanonext**.

Usage

```
is_aio(x)

is_nano(x)

is_ncurl_session(x)
```

Arguments

x	an object.

Details

Is the object an Aio (inheriting from class 'sendAio' or 'recvAio').

Is the object an object inheriting from class 'nano' i.e. a nanoSocket, nanoContext, nanoStream, nanoListener, nanoDialer, nanoMonitor or nano Object.

Is the object an ncurlSession (object of class 'ncurlSession').

Is the object a Condition Variable (object of class 'conditionVariable').

Value

Logical value TRUE or FALSE.

Examples

```
nc <- call_aio(ncurl_aio("https://postman-echo.com/get", timeout = 1000L))
is_aio(nc)

s <- socket()
is_nano(s)
n <- nano()
is_nano(n)
close(s)
n$close()

s <- ncurl_session("https://postman-echo.com/get", timeout = 1000L)
is_ncurl_session(s)
if (is_ncurl_session(s)) close(s)
```

`is_error_value` *Error Validators*

Description

Validator functions for error value types created by **nanonext**.

Usage

```
is_error_value(x)

is_nul_byte(x)
```

Arguments

x	an object.

Details

Is the object an error value generated by the package. All non-success integer return values are classed 'errorValue' to be distinguishable from integer message values. Includes error values returned after a timeout etc.

Is the object a nul byte.

Value

Logical value TRUE or FALSE.

Examples

```
s <- socket()
r <- recv_aio(s, timeout = 10)
call_aio(r)$data
close(s)
r$data == 5L
is_error_value(r$data)
is_error_value(5L)

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",
  tls = NULL,
  autostart = TRUE,
  fail = c("warn", "error", "none")
)
```

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).
`tls`	[default NULL] for secure tls+tcp:// or wss:// connections only, provide a TLS configuration object created by `tls_config()`.
`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.
`fail`	[default 'warn'] failure mode - a character value or integer equivalent, whether to warn (1L), error (2L), or for none (3L) just return an 'errorValue' without any corresponding warning.

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 `opt<-()` 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 = "inproc://nanolisten", autostart = FALSE)
socket$listener
start(socket$listener[[1]])
socket$listener
close(socket$listener[[1]])
close(socket)

nano <- nano("bus")
nano$listen(url = "inproc://nanolisten", autostart = FALSE)
nano$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
```

`monitor` *Monitor a Socket for Pipe Changes*

Description

This function monitors pipe additions and removals from a socket.

Usage

```
monitor(sock, cv)

read_monitor(x)
```

Arguments

`sock`	a Socket.
`cv`	a 'conditionVariable'.
`x`	a Monitor.

Value

For `monitor`: a Monitor (object of class 'nanoMonitor').
For `read_monitor`: an integer vector of pipe IDs (positive if added, negative if removed), or else NULL if there were no changes since the previous read.

Examples

```
cv <- cv()
s <- socket("poly")
s1 <- socket("poly")

m <- monitor(s, cv)
m

listen(s)
dial(s1)

cv_value(cv)
read_monitor(m)
```

```
close(s)
close(s1)

read_monitor(m)
```

`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(time)
```

Arguments

`time`	integer number of milliseconds to block the caller.

Details

Non-integer values for `time` are coerced to integer. Negative, logical and other non-numeric values are ignored, causing the function to 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", "poly", "push", "pull", "pub", "sub", "req", "rep",
    "surveyor", "respondent"),
  dial = NULL,
  listen = NULL,
  tls = NULL,
  autostart = TRUE
)
```

Arguments

`protocol`	[default 'bus'] choose protocol - `"bus"`, `"pair"`, `"poly"`, `"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).
`tls`	[default NULL] for secure tls+tcp:// or wss:// connections only, provide a TLS configuration object created by `tls_config()`.
`autostart`	[default TRUE] whether to start the dialer/listener. Set to FALSE if setting 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 `$dialer_opt()` and `$listener_opt()` methods will be available once dialers/listeners are attached to the object. These methods get or apply settings for all dialers or listeners equally. To get or apply settings for individual dialers/listeners, access them directly via `$dialer[[2]]` or `$listener[[2]]` etc.

The methods `$opt()`, and also `$dialer_opt()` or `$listener_opt()` as may be applicable, will get the requested option if a single argument `name` is provided, and will set the value for the option if both arguments `name` and `value` are provided.

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$opt("send-timeout", 1500)
nano$opt("send-timeout")

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,
  convert = TRUE,
  follow = FALSE,
  method = NULL,
  headers = NULL,
  data = NULL,
  response = NULL,
  timeout = NULL,
  tls = NULL
)
```

Arguments

`url`	the URL address.
`convert`	[default TRUE] logical value whether to attempt conversion of the received raw bytes to a character vector. Set to `FALSE` if downloading non-text data.
`follow`	[default FALSE] logical value whether to automatically follow redirects (not applicable for async requests). If `FALSE`, the redirect address is returned as response header 'Location'.
`method`	(optional) the HTTP method as a character string. Defaults to 'GET' if not specified, and could also be 'POST', 'PUT' etc.
`headers`	(optional) a named character vector or named list specifying the HTTP request headers, for example: `c(Authorization = "Bearer APIKEY", "Content-Type" = "text/plain")` or `list(Authorization = "Bearer APIKEY", "Content-Type" = "text/plain")` An unnamed vector or list will be ignored.
`data`	(optional) request data to be submitted. Must be a character string or raw vector, and other objects are ignored. If a character vector, only the first element is taken. When supplying binary data, the appropriate 'Content-Type' header should be set to specify the binary format.
`response`	(optional) `TRUE` to return all response headers, or a character vector specifying the response headers to return e.g. `c("date", "server")`. These are case-insensitive and will return NULL if not present. A non-character vector will be ignored (other than `TRUE`).
`timeout`	(optional) integer value in milliseconds after which the transaction times out if not yet complete.
`tls`	(optional) applicable to secure HTTPS sites only, a client TLS Configuration object created by `tls_config()`. If missing or NULL, certificates are not validated.

Value

Named list of 3 elements:

- `$status` - integer HTTP repsonse status code (200 - OK). Use `status_code()` for a translation of the meaning.
- `$headers` - named list of response headers (all headers if `response = TRUE`, or those specified 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.
- `$data` - the response body, as a character string if `convert = TRUE` (may be further parsed as html, json, xml etc. as required), or a raw byte vector if FALSE (use `writeBin()` to save as a file).

Public Internet HTTPS

When making HTTPS requests over the public internet, you should supply a TLS configuration to validate server certificates. See `tls_config()` for details.

See Also

`ncurl_aio()` for asynchronous http requests; `ncurl_session()` for persistent connections.

Examples

```
ncurl(
  "https://postman-echo.com/get",
  response = TRUE,
  timeout = 2000L
)
ncurl(
  "https://postman-echo.com/put",
  method = "PUT",
  headers = c(Authorization = "Bearer APIKEY"),
  data = "hello world",
  timeout = 2000L
)
ncurl(
  "https://postman-echo.com/post",
  convert = FALSE,
  method = "POST",
  headers = c(`Content-Type` = "application/json"),
  data = '{"key":"value"}',
  response = c("date", "content-type"),
  timeout = 2000L
)
```

---

ncurl_aio *ncurl Async*

Description

nano cURL - a minimalist http(s) client - async edition.

Usage

```
ncurl_aio(
  url,
  convert = TRUE,
  method = NULL,
  headers = NULL,
  data = NULL,
  response = NULL,
  timeout = NULL,
  tls = NULL
)
```

Arguments

`url`	the URL address.
`convert`	[default TRUE] logical value whether to attempt conversion of the received raw bytes to a character vector. Set to `FALSE` if downloading non-text data.
`method`	(optional) the HTTP method as a character string. Defaults to 'GET' if not specified, and could also be 'POST', 'PUT' etc.
`headers`	(optional) a named character vector or named list specifying the HTTP request headers, for example: `c(Authorization = "Bearer APIKEY", "Content-Type" = "text/plain")` or `list(Authorization = "Bearer APIKEY", "Content-Type" = "text/plain")` An unnamed vector or list will be ignored.
`data`	(optional) request data to be submitted. Must be a character string or raw vector, and other objects are ignored. If a character vector, only the first element is taken. When supplying binary data, the appropriate 'Content-Type' header should be set to specify the binary format.
`response`	(optional) `TRUE` to return all response headers, or a character vector specifying the response headers to return e.g. `c("date", "server")`. These are case-insensitive and will return NULL if not present. A non-character vector will be ignored (other than `TRUE`).
`timeout`	(optional) integer value in milliseconds after which the transaction times out if not yet complete.
`tls`	(optional) applicable to secure HTTPS sites only, a client TLS Configuration object created by `tls_config()`. If missing or NULL, certificates are not validated.

Value

An 'ncurlAio' (object of class 'ncurlAio' and 'recvAio') (invisibly). The following elements may be accessed:

- `$status` - integer HTTP repsonse status code (200 - OK). Use `status_code()` for a translation of the meaning.

- $headers - named list of response headers (all headers if response = TRUE, or those specified 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.
- $data - the response body, as a character string if convert = TRUE (may be further parsed as html, json, xml etc. as required), or a raw byte vector if FALSE (use writeBin() to save as a file).

Promises

'ncurlAio' may be used anywhere that accepts a 'promise' from the **promises** package through the included as.promise method.

The promises created are completely event-driven and non-polling.

If a status code of 200 (OK) is returned then the promise is resolved with the reponse body, otherwise it is rejected with a translation of the status code or 'errorValue' as the case may be.

Public Internet HTTPS

When making HTTPS requests over the public internet, you should supply a TLS configuration to validate server certificates. See tls_config() for details.

See Also

ncurl() for synchronous http requests; ncurl_session() for persistent connections.

Examples

```
nc <- ncurl_aio(
  "https://postman-echo.com/get",
  response = c("date", "server"),
  timeout = 2000L
)
call_aio(nc)
nc$status
nc$headers
nc$data


library(promises)
p <- as.promise(nc)
print(p)

p2 <- ncurl_aio("https://postman-echo.com/get") %>%
  then(function(x) cat(x$data))
is.promise(p2)
```

`ncurl_session` *ncurl Session*

Description

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

Usage

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

transact(session)
```

Arguments

`url`	the URL address.
`convert`	[default TRUE] logical value whether to attempt conversion of the received raw bytes to a character vector. Set to `FALSE` if downloading non-text data.
`method`	(optional) the HTTP method as a character string. Defaults to 'GET' if not specified, and could also be 'POST', 'PUT' etc.
`headers`	(optional) a named character vector or named list specifying the HTTP request headers, for example: `c(Authorization = "Bearer APIKEY", "Content-Type" = "text/plain")` or `list(Authorization = "Bearer APIKEY", "Content-Type" = "text/plain")` An unnamed vector or list will be ignored.
`data`	(optional) request data to be submitted. Must be a character string or raw vector, and other objects are ignored. If a character vector, only the first element is taken. When supplying binary data, the appropriate 'Content-Type' header should be set to specify the binary format.
`response`	(optional) `TRUE` to return all response headers, or a character vector specifying the response headers to return e.g. `c("date", "server")`. These are case-insensitive and will return NULL if not present. A non-character vector will be ignored (other than `TRUE`).
`timeout`	(optional) integer value in milliseconds after which the connection and subsequent transact attempts time out.

tls	(optional) applicable to secure HTTPS sites only, a client TLS Configuration object created by tls_config(). If missing or NULL, certificates are not validated.
session	an 'ncurlSession' object.

Value

For ncurl_session: an 'ncurlSession' object if successful, or else an 'errorValue'.

For transact: a named list of 3 elements:

- $status - integer HTTP repsonse status code (200 - OK). Use status_code() for a translation of the meaning.
- $headers - named list of response headers (all headers if response = TRUE was specified for the session, those specified in response, or NULL otherwise).
- $data - the response body as a character string (if convert = TRUE was specified for the session), which may be further parsed as html, json, xml etc. as required, or else a raw byte vector, which may be saved as a file using writeBin().

Public Internet HTTPS

When making HTTPS requests over the public internet, you should supply a TLS configuration to validate server certificates. See tls_config() for details.

See Also

ncurl() for synchronous http requests; ncurl_aio() for asynchronous http requests.

Examples

```
s <- ncurl_session(
  "https://postman-echo.com/get",
  response = "date",
  timeout = 2000L
)
s
if (is_ncurl_session(s)) transact(s)
if (is_ncurl_session(s)) close(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 string comprising the error code and error message separated by '|'.

Examples

```
nng_error(1L)
```

`nng_version` *NNG Library Version*

Description

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

Usage

```
nng_version()
```

Value

A character vector of length 2.

Examples

```
nng_version()
```

opt *Get and Set Options for a Socket, Context, Stream, Listener or Dialer*

Description

Get and set the value of options for a Socket, Context, Stream, Listener or Dialer.

Usage

```
opt(object, name)

opt(object, name) <- value
```

Arguments

object	a Socket, Context, Stream, Listener or Dialer.
name	name of option, e.g. 'recv-buffer', as a character string. See below options details.
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 get or 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.

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/.

Value

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

Serialization

Apart from the NNG options documented below, there is the following special option:

- 'serial' (type list)
  For Sockets only. This accepts a configuration created by `serial_config()`. Setting a new configuration replaces any already set. To remove entirely, supply an empty list. Note: this option is write-only and can be set but not retrieved.

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').
- 'recv-fd' (type 'int')
  This is the socket receive file descriptor. For supported protocols, this will become readable when a message is available for receiving on the socket. Attempts should not be made to read or write to the returned file descriptor, but it is suitable for use with poll(), select(), or WSAPoll() on Windows, and similar functions.

- 'send-fd' (type 'int')
  This is the socket send file descriptor. Attempts should not be made to read or write to the returned file descriptor, but it is suitable for use with poll(), select(), or WSAPoll() on Windows, and similar functions.
- '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.
- 'url' (type 'string')
  This read-only option is used on a listener or dialer to obtain the URL with which it was configured.

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.

- 'tcp-bound-port' (type 'int')

  (TCP transport) Local TCP port number. This is used on a listener, and is intended to be used after starting the listener in combination with a wildcard (0) local port. This determines the actual ephemeral port that was selected and bound. The value is provided as an integer, but only the low order 16 bits will be set, and is in native byte order for convenience.

- '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.

- 'ws:request-uri' (type 'string')

  (WebSocket transport) For obtaining the URI sent by the client. This can be useful when a handler supports an entire directory tree.

Examples

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

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

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

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

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

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

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

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

`parse_url` *Parse URL*

Description

Parses a character string containing an RFC 3986 compliant URL as per NNG.

Usage

```
parse_url(url)
```

Arguments

`url`	character string containing a URL.

Value

A named character vector of length 7, comprising:

- `scheme` - the URL scheme, such as "http" or "inproc" (always lower case).
- `userinfo` - the username and password (if supplied in the URL string).

- `hostname` - the name of the host.
- `port` - the port (if not specified, the default port if defined by the scheme).
- `path` - the path, typically used with HTTP or WebSocket.
- `query` - the query info (typically following ? in the URL).
- `fragment` - used for specifying an anchor, the part after # in a URL.

Values that cannot be determined are represented by an empty string `""`.

Examples

```
parse_url("https://user:password@w3.org:8080/type/path?q=info#intro")
parse_url("tcp://192.168.0.2:5555")
```

`pipe_id` *Get the Pipe ID of a recvAio*

Description

Caution: must only be used on an already-resolved 'recvAio'. This function does not perform validation of these pre-conditions.

Usage

```
pipe_id(x)
```

Arguments

`x`	a resolved 'recvAio'.

Value

Integer pipe ID.

pipe_notify *Pipe Notify*

Description

Signals a 'conditionVariable' whenever pipes (individual connections) are added or removed at a socket.

Usage

```
pipe_notify(socket, cv, add = FALSE, remove = FALSE, flag = FALSE)
```

Arguments

socket	a Socket.
cv	a 'conditionVariable' to signal, or NULL to cancel a previously set signal.
add	[default FALSE] logical value whether to signal (or cancel signal) when a pipe is added.
remove	[default FALSE] logical value whether to signal (or cancel signal) when a pipe is removed.
flag	[default FALSE] logical value whether to also set a flag in the 'conditionVariable'. This can help distinguish between different types of signal, and causes any subsequent wait() to return FALSE instead of TRUE. If a signal from the **tools** package, e.g. tools::SIGINT, or an equivalent integer value is supplied, this sets a flag and additionally raises this signal upon the flag being set. For tools::SIGTERM, the signal is raised with a 200ms grace period to allow a process to exit normally.

Details

For add: this event occurs after the pipe is fully added to the socket. Prior to this time, it is not possible to communicate over the pipe with the socket.

For remove: this event occurs after the pipe has been removed from the socket. The underlying transport may be closed at this point, and it is not possible to communicate using this pipe.

Value

Invisibly, zero on success (will otherwise error).

Examples

```
s <- socket(listen = "inproc://nanopipe")
cv <- cv()

pipe_notify(s, cv, add = TRUE, remove = TRUE, flag = TRUE)
cv_value(cv)
```

```
s1 <- socket(dial = "inproc://nanopipe")
cv_value(cv)
reap(s1)
cv_value(cv)

pipe_notify(s, NULL, add = TRUE, remove = TRUE)
s1 <- socket(dial = "inproc://nanopipe")
cv_value(cv)
reap(s1)

(wait(cv))

close(s)
```

`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 protocol may be used to send and receive messages. Sending messages will attempt to deliver to each directly connected peer. Indirectly-connected peers 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] This is NNG's pair v0. 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 send and receive messages freely.

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

Poly (one-to-one of many)

[protocol, poly] This is NNG's pair v1 polyamorous mode. It allows a socket to communicate with multiple directly-connected peers.

If no remote peer is specified by the sender, then the protocol willselect any available connected peer.

If the peer on the given pipe is not able to receive (or the pipe is no longer available, such as if the peer has disconnected), then the message will be discarded with no notification to the sender.

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 subscriber only see the data to which they have subscribed.

[protocol, pub] The pub protocol is one half of a publisher/subscriber pattern. This protocol 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 protocol 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). 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 protocol 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 may 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 protocol 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 which the last survey was received.

---

race_aio *Race Aio*

Description

Returns the index of the first resolved Aio in a list, waiting if necessary.

Usage

```
race_aio(x, cv)
```

Arguments

x	A list of Aio objects.
cv	A condition variable. This must be the same cv supplied to `recv_aio()` or `request()` when creating the Aio objects in `x`.

Value

Integer index of the first resolved Aio, or 0L if none are resolved, the list is empty, or the cv was terminated.

---

random *Random Data Generation*

Description

Strictly not for use in statistical analysis. Non-reproducible and with unknown statistical properties. Provides an alternative source of randomness from the Mbed TLS library for purposes such as cryptographic key generation. Mbed TLS uses a block-cipher in counter mode operation, as defined in NIST SP800-90A: *Recommendation for Random Number Generation Using Deterministic Random Bit Generators*. The implementation uses AES-256 as the underlying block cipher, with a derivation function, and an entropy collector combining entropy from multiple sources including at least one strong entropy source.

Usage

```
random(n = 1L, convert = TRUE)
```

Arguments

n	[default 1L] integer random bytes to generate (from 0 to 1024), coerced to integer if required. If a vector, the first element is taken.
convert	[default TRUE] logical FALSE to return a raw vector, or TRUE to return the hex representation of the bytes as a character string.

Value

A length n raw vector, or length one vector of 2n random characters, depending on the value of convert supplied.

Note

Results obtained are independent of and do not alter the state of R's own pseudo-random number generators.

Examples

```
random()
random(8L)
random(n = 8L, convert = FALSE)
```

---

read_stdin *Read stdin*

Description

Reads stdin from a background thread, allowing the stream to be accessed as messages from an NNG 'inproc' socket. As the read is blocking, it can only be used in non-interactive sessions. Closing stdin causes the background thread to exit and the socket connection to end.

Usage

```
read_stdin()
```

Details

A 'pull' protocol socket is returned, and hence can only be used with receive functions.

Value

a Socket.

`reap` *Reap*

Description

An alternative to `close` for Sockets, Contexts, Listeners, and Dialers avoiding S3 method dispatch.

Usage

```
reap(con)
```

Arguments

`con`	a Socket, Context, Listener or Dialer.

Details

May be used on unclassed external pointers e.g. those created by `.context()`. Returns silently and does not warn or error, nor does it update the state of object attributes.

Value

An integer exit code (zero on success).

See Also

`close()`

Examples

```
s <- socket("req")
listen(s)
dial(s)
ctx <- .context(s)

reap(ctx)
reap(s[["dialer"]][[1]])
reap(s[["listener"]][[1]])
reap(s)
reap(s)
```

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", "string"),
  block = NULL
)
```

Arguments

con	a Socket, Context or Stream.
mode	[default 'serial'] character value or integer equivalent - one of "serial" (1L), "character" (2L), "complex" (3L), "double" (4L), "integer" (5L), "logical" (6L), "numeric" (7L), "raw" (8L), or "string" (9L). The default "serial" means a serialised R object; for the other modes, received bytes are converted into the respective mode. "string" is a faster option for length one character vectors. For Streams, "serial" will default to "character".
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

The received data in the mode specified.

Errors

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 an error occurred in unserialization or conversion of the message data to the specified mode, a raw vector will be returned instead to allow recovery (accompanied by a warning).

Blocking

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

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

See Also

`recv_aio()` for asynchronous receive.

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)

send(s1, c(1.1, 2.2, 3.3), mode = "raw")
res <- recv(s2, mode = "double", block = 100)
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", "string"),
  timeout = NULL,
  cv = NULL
)
```

Arguments

con	a Socket, Context or Stream.
mode	[default 'serial'] character value or integer equivalent - one of "serial" (1L), "character" (2L), "complex" (3L), "double" (4L), "integer" (5L), "logical" (6L), "numeric" (7L), "raw" (8L), or "string" (9L). The default "serial" means a serialised R object; for the other modes, received bytes are converted into the respective mode. "string" is a faster option for length one character vectors. For Streams, "serial" will default to "character".
timeout	[default NULL] integer value in milliseconds or NULL, which applies a socket-specific default, usually the same as no timeout.
cv	(optional) a 'conditionVariable' to signal when the async receive is complete.

Details

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

For a 'recvAio', the received message is available at $data. 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 checked using is_error_value().

If an error occurred in unserialization or conversion of the message data to the specified mode, a raw vector will be returned instead to allow recovery (accompanied by a warning).

Value

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

Signalling

By supplying a 'conditionVariable', when the receive is complete, the 'conditionVariable' is signalled by incrementing its value by 1. This happens asynchronously and independently of the R execution thread.

See Also

recv() for synchronous receive.

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)
msg
msg$data

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

close(s1)
close(s2)

# Signalling a condition variable

s1 <- socket("pair", listen = "inproc://cv-example")
cv <- cv()
msg <- recv_aio(s1, timeout = 100, cv = cv)
until(cv, 10L)
msg$data
close(s1)

# in another process in parallel
s2 <- socket("pair", dial = "inproc://cv-example")
res <- send_aio(s2, c(1.1, 2.2, 3.3), mode = "raw", timeout = 100)
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", "string"),
  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'] character value or integer equivalent - one of `"serial"` (1L), `"character"` (2L), `"complex"` (3L), `"double"` (4L), `"integer"` (5L), `"logical"` (6L), `"numeric"` (7L), `"raw"` (8L), or `"string"` (9L). The default `"serial"` means a serialised R object; for the other modes, received bytes are converted into the respective mode. `"string"` is a faster option for length one character vectors.
`send_mode`	[default 'serial'] character value or integer equivalent - either `"serial"` (1L) to send serialised R objects, or `"raw"` (2L) to send atomic vectors of any type as a raw byte vector.
`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).

Send Modes

The default mode `"serial"` sends serialised R objects to ensure perfect reproducibility within R. When receiving, the corresponding mode `"serial"` should be used. Custom serialization and unserialization functions for reference objects may be enabled by the function `serial_config()`.

Mode `"raw"` sends atomic vectors of any type as a raw byte vector, and must be used when interfacing with external applications or raw system sockets, where R serialization is not in use. When receiving, the mode corresponding to the vector sent should be used.

Examples

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

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 for the value when required.

Usage

```
request(
  context,
  data,
  send_mode = c("serial", "raw"),
  recv_mode = c("serial", "character", "complex", "double", "integer", "logical",
    "numeric", "raw", "string"),
  timeout = NULL,
  cv = NULL,
  id = NULL
)
```

Arguments

context	a Context.
data	an object (if send_mode = "raw", a vector).
send_mode	[default 'serial'] character value or integer equivalent - either "serial" (1L) to send serialised R objects, or "raw" (2L) to send atomic vectors of any type as a raw byte vector.
recv_mode	[default 'serial'] character value or integer equivalent - one of "serial" (1L), "character" (2L), "complex" (3L), "double" (4L), "integer" (5L), "logical" (6L), "numeric" (7L), "raw" (8L), or "string" (9L). The default "serial" means a serialised R object; for the other modes, received bytes are converted into the respective mode. "string" is a faster option for length one character vectors.
timeout	[default NULL] integer value in milliseconds or NULL, which applies a socket-specific default, usually the same as no timeout.
cv	(optional) a 'conditionVariable' to signal when the async receive is complete, or NULL.
id	NULL. For package internal use only.

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. 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.

It is recommended to use a new context for each request to ensure consistent state tracking. The integer context ID is appended as the attribute id to the returned 'recvAio'.

Value

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

Send Modes

The default mode "serial" sends serialised R objects to ensure perfect reproducibility within R. When receiving, the corresponding mode "serial" should be used. Custom serialization and unserialization functions for reference objects may be enabled by the function serial_config().

Mode "raw" sends atomic vectors of any type as a raw byte vector, and must be used when interfacing with external applications or raw system sockets, where R serialization is not in use. When receiving, the mode corresponding to the vector sent should be used.

Signalling

By supplying a 'conditionVariable', when the receive is complete, the 'conditionVariable' is signalled by incrementing its value by 1. This happens asynchronously and independently of the R execution thread.

Examples

```
## Not run:

# works if req and rep are running in parallel in different processes

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

reply(.context(rep), execute = function(x) x + 1, timeout = 50)
aio <- request(.context(req), data = 2022)
aio$data

close(req)
close(rep)

# Signalling a condition variable

req <- socket("req", listen = "tcp://127.0.0.1:6546")
ctxq <- context(req)
cv <- cv()
aio <- request(ctxq, data = 2022, cv = cv)
until(cv, 10L)
close(req)

# The following should be run in another process
rep <- socket("rep", dial = "tcp://127.0.0.1:6546")
ctxp <- context(rep)
reply(ctxp, execute = function(x) x + 1)
close(rep)


## End(Not run)
```

run_event_loop *Run the Event Loop*

Description

Runs the **later** event loop, blocking until the next event and dispatching any scheduled callbacks. Used to process a server's (see `http_server()`) or other asynchronous operations' callbacks on R's main thread.

Usage

```
run_event_loop(timeout = Inf)
```

Arguments

`timeout`	maximum time to wait in milliseconds. The default `Inf` blocks until an event occurs.

Details

Requires the **later** package.

Value

Logical `TRUE` if a callback was executed, `FALSE` otherwise.

send *Send*

Description

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

Usage

```
send(con, data, mode = c("serial", "raw"), block = NULL, pipe = 0L)
```

Arguments

`con`	a Socket, Context or Stream.
`data`	an object (a vector, if `mode = "raw"`).
`mode`	[default 'serial'] character value or integer equivalent - either `"serial"` (1L) to send serialised R objects, or `"raw"` (2L) to send atomic vectors of any type as a raw byte vector. For Streams, `"raw"` is the only option and this argument is ignored.

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.
pipe	[default 0L] only applicable to Sockets using the 'poly' protocol, an integer pipe ID if directing the send via a specific pipe.

Value

An integer exit code (zero on success).

Blocking

For Sockets and Contexts: 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 Streams: the default behaviour is blocking with block = TRUE. This will wait until the send has completed. Set a timeout 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 small positive value for block rather than FALSE.

Send Modes

The default mode "serial" sends serialised R objects to ensure perfect reproducibility within R. When receiving, the corresponding mode "serial" should be used. Custom serialization and unserialization functions for reference objects may be enabled by the function serial_config().

Mode "raw" sends atomic vectors of any type as a raw byte vector, and must be used when interfacing with external applications or raw system sockets, where R serialization is not in use. When receiving, the mode corresponding to the vector sent should be used.

See Also

send_aio() for asynchronous send.

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, Stream or Pipe).

Usage

```
send_aio(con, data, mode = c("serial", "raw"), timeout = NULL, pipe = 0L)
```

Arguments

`con`	a Socket, Context or Stream.
`data`	an object (a vector, if `mode = "raw"`).
`mode`	[default 'serial'] character value or integer equivalent - either `"serial"` (1L) to send serialised R objects, or `"raw"` (2L) to send atomic vectors of any type as a raw byte vector. For Streams, `"raw"` is the only option and this argument is ignored.
`timeout`	[default NULL] integer value in milliseconds or NULL, which applies a socket-specific default, usually the same as no timeout.
`pipe`	[default 0L] only applicable to Sockets using the 'poly' protocol, an integer pipe ID if directing the send via a specific pipe.

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).

Send Modes

The default mode "serial" sends serialised R objects to ensure perfect reproducibility within R. When receiving, the corresponding mode "serial" should be used. Custom serialization and unserialization functions for reference objects may be enabled by the function serial_config().

Mode "raw" sends atomic vectors of any type as a raw byte vector, and must be used when interfacing with external applications or raw system sockets, where R serialization is not in use. When receiving, the mode corresponding to the vector sent should be used.

See Also

send() for synchronous send.

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)
```

---

serial_config *Create Serialization Configuration*

Description

Returns a serialization configuration, which may be set on a Socket for custom serialization and unserialization of non-system reference objects, allowing these to be sent and received between different R sessions. Once set, the functions apply to all send and receive operations performed in mode 'serial' over the Socket, or Context created from the Socket.

Usage

```
serial_config(class, sfunc, ufunc)
```

Arguments

class	a character string (or vector) of the class of object custom serialization functions are applied to, e.g. 'ArrowTabular' or c('torch_tensor', 'ArrowTabular').
sfunc	a function (or list of functions) that accepts a reference object inheriting from class and returns a raw vector.
ufunc	a function (or list of functions) that accepts a raw vector and returns a reference object.

Details

This feature utilises the 'refhook' system of R native serialization.

If `sfunc` errors during serialization, the error propagates to the caller and no message is sent.

If `ufunc` errors during unserialization, the error is caught and the object instead unserializes as an 'errorValue' 1000 ('Internal error detected'), rather than the error propagating to the caller.

Value

A list comprising the configuration. This should be set on a Socket using `opt<-()` with option name `"serial"`.

Examples

```
cfg <- serial_config("test_cls", function(x) serialize(x, NULL), unserialize)
cfg

cfg <- serial_config(
  c("class_one", "class_two"),
  list(function(x) serialize(x, NULL), function(x) serialize(x, NULL)),
  list(unserialize, unserialize)
)
cfg

s <- socket()
opt(s, "serial") <- cfg

# provide an empty list to remove registered functions
opt(s, "serial") <- list()

close(s)
```

---

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", "poly", "push", "pull", "pub", "sub", "req", "rep",
    "surveyor", "respondent"),
  dial = NULL,
  listen = NULL,
  tls = NULL,
```

```
  autostart = TRUE,
  raw = FALSE
)
```

Arguments

protocol	[default 'bus'] choose protocol - "bus", "pair", "poly", "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).
tls	[default NULL] for secure tls+tcp:// or wss:// connections only, provide a TLS configuration object created by tls_config().
autostart	[default TRUE] whether to start the dialer/listener. Set to FALSE if setting 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 (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 may 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'
- Poly (one-to-one of many) - protocol: 'poly'
- 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.

Transports

The following communications transports may be used:

- Inproc (in-process) - url: 'inproc://'
- IPC (inter-process communications) - url: 'ipc://' (or 'abstract://' on Linux)
- TCP and TLS over TCP - url: 'tcp://' and 'tls+tcp://'
- WebSocket and TLS over WebSocket - url: 'ws://' and 'wss://'

Please see transports for further documentation.

Examples

```
s <- socket(protocol = "req", listen = "inproc://nanosocket")
s
s1 <- socket(protocol = "rep", dial = "inproc://nanosocket")
s1

send(s, "hello world!")
recv(s1)

close(s1)
close(s)
```

---

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.

Value

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

---

stat *Get Statistic for a Socket, Listener or Dialer*

Description

Obtain value of a statistic for a Socket, Listener or Dialer. This function exposes the stats interface of NNG.

Usage

```
stat(object, name)
```

Arguments

object	a Socket, Listener or Dialer.
name	character name of statistic to return.

Details

Note: the values of individual statistics are guaranteed to be atomic, but due to the way statistics are collected there may be discrepancies between them at times. For example, statistics counting bytes and messages received may not reflect the same number of messages, depending on when the snapshot is taken. This potential inconsistency arises as a result of optimisations to minimise the impact of statistics on actual operations.

Value

The value of the statistic (character or double depending on the type of statistic requested) if available, or else NULL.

Stats

The following stats may be requested for a Socket:

- 'id' - numeric id of the socket.
- 'name' - character socket name.
- 'protocol' - character protocol type e.g. 'bus'.
- 'pipes' - numeric number of pipes (active connections).
- 'dialers' - numeric number of listeners attached to the socket.
- 'listeners' - numeric number of dialers attached to the socket.

The following stats may be requested for a Listener / Dialer:

- 'id' - numeric id of the listener / dialer.
- 'socket' - numeric id of the socket of the listener / dialer.
- 'url' - character URL address.
- 'pipes' - numeric number of pipes (active connections).

The following additional stats may be requested for a Listener:

- 'accept' - numeric total number of connection attempts, whether successful or not.
- 'reject' - numeric total number of rejected connection attempts e.g. due to incompatible protocols.

The following additional stats may be requested for a Dialer:

- 'connect' - numeric total number of connection attempts, whether successful or not.
- 'reject' - numeric total number of rejected connection attempts e.g. due to incompatible protocols.
- 'refused' - numeric total number of refused connections e.g. when starting synchronously with no listener on the other side.

Examples

```
s <- socket("bus", listen = "inproc://stats")
stat(s, "pipes")

s1 <- socket("bus", dial = "inproc://stats")
stat(s, "pipes")

close(s1)
stat(s, "pipes")

close(s)
```

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, "Unknown HTTP Status" is returned - this 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 comprising the status code and explanation separated by '|'.

Examples

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

stop_aio *Stop Asynchronous Aio Operation*

Description

Stop an asynchronous Aio operation, or a list of Aio operations.

Usage

```
stop_aio(x)
```

Arguments

x	an Aio or list of Aios (objects of class 'sendAio', 'recvAio' or 'ncurlAio').

Details

Stops the asynchronous I/O operation associated with Aio `x` by aborting, and then waits for it to complete or to be completely aborted, and for the callback associated with the Aio to have completed executing. If successful, the Aio will resolve to an 'errorValue' 20 (Operation canceled).

Note this function operates silently and does not error even if `x` 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,
  headers = NULL,
  tls = NULL,
  buffer = 65536L
)
```

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).
`headers`	(optional) applicable to websocket connections only, a **named** character vector specifying custom request headers to send during the WebSocket upgrade handshake e.g. `c(Authorization = "Bearer token", Custom = "value")` (ignored for non-websocket transports).
`tls`	(optional) applicable to secure websockets only, a client or server TLS configuration object created by `tls_config()`. If missing or NULL, certificates are not validated.

buffer [default 65536L] applicable to non-websocket streams only, the maximum number of bytes to receive. Can be an over-estimate, but note that a buffer of this size is reserved. Not used for websocket connections, which handle framing automatically.

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.

Closing a stream renders it invalid and attempting to perform additional operations on it will error.

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

# Run in interactive sessions only as connection is not always available:
s <- tryCatch(
  stream(dial = "wss://echo.websocket.events/", textframes = TRUE),
  error = identity
)
s
if (is_nano(s)) recv(s)
if (is_nano(s)) send(s, "hello")
if (is_nano(s)) recv(s)
if (is_nano(s)) close(s)
```

---

subscribe *Subscribe / Unsubscribe Topic*

Description

For a socket or context using the sub protocol in a publisher/subscriber pattern. Set a topic to subscribe to, or remove a topic from the subscription list.

Usage

```
subscribe(con, topic = NULL)

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 subscribes to all topics / unsubscribes from all topics (if all topics were previously subscribed).

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, the passed Socket or Context.

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")
unsubscribe(sub, "examples")
send(pub, c("examples", "this example is no longer received"), mode = "raw")
recv(sub, "character")

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

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 the survey timeout in milliseconds (remains valid for all subsequent surveys). Messages received by the surveyor after the timer has ended are discarded.

Usage

```
survey_time(con, value = 1000L)
```

Arguments

con	a Socket or Context using the 'surveyor' protocol.
value	[default 1000L] integer survey timeout in milliseconds.

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, the passed Socket or Context.

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)
```

`tls_config` *Create TLS Configuration*

Description

Create a TLS configuration object to be used for secure connections. Specify `client` to create a client configuration or `server` to create a server configuration.

Usage

```
tls_config(client = NULL, server = NULL, pass = NULL, auth = is.null(server))
```

Arguments

`client`	**either** the character path to a file containing X.509 certificate(s) in PEM format, comprising the certificate authority certificate chain (and revocation list if present), used to validate certificates presented by peers, **or** a length 2 character vector comprising [i] the certificate authority certificate chain and [ii] the certificate revocation list, or empty string `""` if not applicable.
`server`	**either** the character path to a file containing the PEM-encoded TLS certificate and associated private key (may contain additional certificates leading to a validation chain, with the leaf certificate first), **or** a length 2 character vector comprising [i] the TLS certificate (optionally certificate chain) and [ii] the associated private key.
`pass`	(optional) required only if the secret key supplied to `server` is encrypted with a password. For security, consider providing through a function that returns this value, rather than directly.
`auth`	logical value whether to require authentication - by default TRUE for client and FALSE for server configurations. If TRUE, the session is only allowed to proceed if the peer has presented a certificate and it has been validated. If FALSE, authentication is optional, whereby a certificate is validated if presented by the peer, but the session allowed to proceed otherwise. If neither `client` nor `server` are supplied, then no authentication is performed and this argument has no effect.

Details

Specify one of `client` or `server` only, or neither (in which case an empty client configuration is created), as a configuration can only be of one type.

Value

A 'tlsConfig' object.

Public Internet HTTPS

When making HTTPS requests over the public internet, you should supply a TLS configuration to validate server certificates.

Root CA certificates in PEM format may be found at:

- Linux: '`/etc/ssl/certs/ca-certificates.crt`' or '`/etc/pki/tls/certs/ca-bundle.crt`'
- macOS: '`/etc/ssl/cert.pem`'
- Windows: download from the Common CA Database site run by Mozilla: `https://www.ccadb.org/resources` (select the Server Authentication SSL/TLS certificates text file). *This link is not endorsed; use at your own risk.*

Examples

```
tls <- tls_config()
tls
ncurl("https://postman-echo.com/get", timeout = 1000L, tls = tls)

# client TLS configuration for public internet HTTPS on Linux
# tls <- tls_config(client = "/etc/ssl/certs/ca-certificates.crt")
```

---

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 supported by the underlying platform.

[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 shorthand 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

TLS

The TLS transport provides communication support between peers across a TCP/IP network using TLS v1.2 on top of TCP. Both IPv4 and IPv6 are supported when supported by the underlying platform.

[URI, tls+tcp://] This transport uses URIs using the scheme tls+tcp://, followed by an IP address or hostname, followed by a colon and finally a TCP port number. For example, to contact port 4433 on the localhost either of the following URIs could be used: tls+tcp://127.0.0.1:4433 or tls+tcp://localhost:4433.

- A URI may be restricted to IPv6 using the scheme tls+tcp6://, or IPv4 using the scheme tls+tcp4://.

WebSocket

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

[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.

[URI, wss://] Secure WebSockets use the scheme wss://, and the default TCP port number of 443. Otherwise the format is the same as for regular WebSockets.

- A URI may be restricted to IPv6 using the scheme ws6:// or wss6://, or IPv4 using the scheme ws4:// or wss4://.
- 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, Aio value or list of Aios remains unresolved. Unlike `call_aio()`, this function does not wait for completion.

Usage

```
unresolved(x)
```

Arguments

`x`	an Aio or list of Aios (objects of class 'sendAio', 'recvAio' or 'ncurlAio'), or Aio value stored at `$result` or `$data` etc.

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 `x` is an unresolved Aio or Aio value or the list of Aios contains at least one unresolved Aio, 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")
  msleep(100)
}
```

```
unresolved(aio)

close(s1)
```

`write_cert` *Generate Self-Signed Certificate and Key*

Description

Generate self-signed x509 certificate and 4096 bit RSA private/public key pair for use with authenticated, encrypted TLS communications.

Usage

```
write_cert(cn = "127.0.0.1", valid = "20301231235959")
```

Arguments

`cn`	[default '127.0.0.1'] character issuer common name (CN) for the certificate. This can be either a hostname or an IP address, but must match the actual server URL as client authentication will depend on it.
`valid`	[default '20301231235959'] character 'not after' date-time in 'yyyymmddhhmmss' format. The certificate is not valid after this time.

Details

Note that it can take a second or two for the key and certificate to be generated.

Value

A list of length 2, comprising `$server` and `$client`. These may be passed directly to the relevant argument of `tls_config()`.

Examples

```
cert <- write_cert(cn = "127.0.0.1")
ser <- tls_config(server = cert$server)
cli <- tls_config(client = cert$client)

s <- socket(listen = "tls+tcp://127.0.0.1:5555", tls = ser)
s1 <- socket(dial = "tls+tcp://127.0.0.1:5555", tls = cli)

# secure TLS connection established

close(s1)
close(s)

cert
```

`write_stdout` *Write to Stdout*

Description

Performs a non-buffered write to `stdout` using the C function `writev()` or equivalent. Avoids interleaved output when writing concurrently from multiple processes.

Usage

```
write_stdout(x)
```

Arguments

`x`	character string.

Details

This function writes to the C-level `stdout` of the process and hence cannot be re-directed by `sink()`.

A newline character is automatically appended to `x`, hence there is no need to include this within the input string.

Value

Invisible NULL. As a side effect, `x` is output to `stdout`.

Examples

```
write_stdout("")
```

Index

.context(), *56*
%~>%, 6

as.promise.ncurlAio, 7
as.promise.recvAio, 7

call_aio, 8
call_aio(), *15*, *59*, *63*, *67*, *83*
call_aio_ *(*call_aio*)*, 8
close *(*close.nanoContext*)*, 9
close(), *17*, *33*, *56*
close.nanoContext, 9
collect_aio, 10
collect_aio_ *(*collect_aio*)*, 10
context, 11
context(), *70*
cv, 12
cv_reset *(*cv*)*, 12
cv_signal *(*cv*)*, 12
cv_value *(*cv*)*, 12

device_aio, 14
dial, 15
dial(), *70*

format_sse, 17
format_sse(), *24*

handler, 19
handler(), *27*
handler_directory, 20
handler_directory(), *19*
handler_file, 21
handler_file(), *19*
handler_inline, 22
handler_inline(), *19*
handler_redirect, 22
handler_redirect(), *19, 20*
handler_stream, 23
handler_stream(), *17, 18*
handler_ws, 25
handler_ws(), *19*, *27*
http_server, 27
http_server(), *19*, *21–26*, *65*

ip_addr, 29
is_aio, 30
is_error_value, 31
is_error_value(), *43*, *57*, *59*
is_nano *(*is_aio*)*, 30
is_ncurl_session *(*is_aio*)*, 30
is_nul_byte *(*is_error_value*)*, 31
is_nul_byte(), *62, 63*

listen, 32
listen(), *70*

mclock, 33
monitor, 34
msleep, 35

nano, 36
nano(), *4*, *12*
nanonext *(*nanonext-package*)*, 4
nanonext-package, 4
ncurl, 37
ncurl(), *41*, *43*
ncurl_aio, 39
ncurl_aio(), *39*, *43*
ncurl_session, 42
ncurl_session(), *39*, *41*
nng_error, 43
nng_version, 44

opt, *4*, 45
opt<-, *4*
opt<- *(*opt*)*, 45

parse_url, 49
pipe_id, 50
pipe_notify, 51
pipe_notify(), *13, 14*

protocols, *4*, *36*, 52, *70, 71*

race_aio, 54
random, 54
read_monitor *(*monitor*)*, 34
read_stdin, 55
reap, 56
reap(), *10*
recv, 57
recv(), *12*, *60*
recv_aio, 59
recv_aio(), *12, 13*, *54*, *58*
reply, 61
reply(), *12*
request, 62
request(), *12, 13*, *54*
run_event_loop, 65

send, 65
send(), *12*, *68*
send_aio, 67
send_aio(), *12*, *15*, *66*, *78*
serial_config, 68
serial_config(), *45*, *62*, *64*, *66*, *68*
sink(), *85*
socket, 69
socket(), *4*, *15*
start, 71
start(), *17*, *33*
stat, 72
status_code, 74
status_code(), *39, 40*, *43*
stop_aio, 74
stop_aio(), *15*, *59*, *67*
stream, 75
subscribe, 76
survey_time, 78

tls_config, 79
tls_config(), *16*, *27*, *32*, *36*, *38–41*, *43*, *70*, *75*, *84*
transact *(*ncurl_session*)*, 42
transports, *4*, *16*, *32*, *36*, *70, 71*, 80

unresolved, 83
unresolved(), *9*
unsubscribe *(*subscribe*)*, 76
until *(*cv*)*, 12
until_ *(*cv*)*, 12

wait *(*cv*)*, 12
wait(), *51*
wait_ *(*cv*)*, 12
write_cert, 84
write_stdout, 85
writeBin(), *39*, *41*, *43*