Package 'nanonext'

March 1, 2024

Type Package

Title NNG (Nanomsg Next Gen) Lightweight Messaging Library

Version 0.13.2

Description R binding for NNG (Nanomsg Next Gen), a successor to ZeroMQ. NNG is a socket library implementing 'Scalability Protocols', a reliable, high-performance standard for common communications patterns including publish/subscribe, request/reply and service discovery, over in-process, IPC, TCP, WebSocket and secure TLS transports. As its own threaded concurrency framework, provides a toolkit for asynchronous programming and distributed computing, with intuitive 'aio' objects which resolve automatically upon completion of asynchronous operations, and synchronisation primitives allowing R to wait upon events signalled by concurrent threads.

License GPL (>= 3)

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

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

Encoding UTF-8

SystemRequirements 'libnng' >= 1.5 and 'libmbedtls' >= 2.5, or 'cmake' to compile NNG and/or Mbed TLS included in package sources

Depends R (>= 3.5)

Suggests knitr, markdown

VignetteBuilder knitr

RoxygenNote 7.3.1

NeedsCompilation yes

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

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

Repository CRAN

Date/Publication 2024-03-01 18:00:05 UTC

R topics documented:

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

Description

R binding for NNG (Nanomsg Next Gen), a successor to ZeroMQ. NNG is a socket library implementing 'Scalability Protocols', a reliable, high-performance standard for common communications patterns including publish/subscribe, request/reply and service discovery, over in-process, IPC, TCP, WebSocket and secure TLS transports. As its own threaded concurrency framework, provides a toolkit for asynchronous programming and distributed computing, with intuitive 'aio' objects which resolve automatically upon completion of asynchronous operations, and synchronisation primitives allowing 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 website: https://nng.nanomsg.org/
Mbed TLS website: https://www.trustedfirmware.org/projects/mbed-tls/

Author(s)

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

See Also

Useful links:

- https://shikokuchuo.net/nanonext/
- https://github.com/shikokuchuo/nanonext/
- Report bugs at https://github.com/shikokuchuo/nanonext/issues

---

`.context` *Technical Utility: Open Context*

Description

Open a new Context to be used with a Socket. This function is a performance variant of `context`, designed to wrap a socket in a function argument when calling `request` or `reply`.

Usage

```
.context(socket)
```

Arguments

`socket`	a Socket.

Details

External pointers created by this function are unclassed, hence methods for contexts such as `close` will not work (use `reap` instead). Otherwise they function identically to a Context when passed to all messaging functions.

Value

An external pointer.

Examples

```
s <- socket("req", listen = "inproc://nanonext")
r <- request(.context(s), "request data")
close(s)
```

`.unresolved` *Technical Utility: Query if an Aio is Unresolved*

Description

Query whether an Aio remains unresolved. This is an experimental technical utility version of `unresolved` not intended for ordinary use. Provides a method of querying the busy status of an Aio without altering its state in any way i.e. not attempting to retrieve the result or message.

Usage

```
.unresolved(aio)
```

Arguments

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

Details

`.unresolved()` is not intended to be used for 'recvAio' returned by a signalling function, in which case `unresolved` must be used in all cases.

Value

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

base64enc *Base64 Encode / Decode*

Description

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

Usage

```
base64enc(x, convert = TRUE)

base64dec(x, convert = TRUE)
```

Arguments

x	an object.
convert	For **base64enc**: [default TRUE] logical TRUE to encode to a character string or FALSE to a raw vector. For **base64dec**: [default TRUE] logical TRUE to convert back to a character string, FALSE to convert back to a raw vector or NA to decode and then unserialize back to the original object.

Details

For encoding: a character string or raw vector (with no attributes) is encoded 'as is', whilst all other objects are first serialized (using R serialisation version 3, big-endian representation).

For decoding: the value of 'convert' should be set to TRUE, FALSE or NA to be the analogue of the above 3 cases in order to return the original object.

Value

For **base64enc**: A character string or raw vector depending on the value of 'convert'.

For **base64dec**: A character string, raw vector, or other object depending on the value of 'convert'.

Examples

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

base64enc(as.raw(c(1L, 2L, 4L)), convert = FALSE)
base64dec(base64enc(as.raw(c(1L, 2L, 4L))), convert = FALSE)

base64enc(data.frame())
base64dec(base64enc(data.frame()), convert = NA)
```

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.

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

Usage

```
call_aio(aio)

call_aio_(aio)
```

Arguments

aio	an Aio (object 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).

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

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

Value

The passed object (invisibly).

Alternatively

Aio values may be accessed directly at $result for a 'sendAio', and $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, 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 '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.

Value

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

See Also

`reap`

`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 signalling forms of the asynchronous receive functions: `recv_aio_signal` or `request_signal`. 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()

# wait(cv) # uncommenting will block until the cv is signalled
# wait_(cv) # block until the cv is signalled or interrupted

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

cv_value(cv)

cv_reset(cv)

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

`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,
  error = FALSE
)
```

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.
error	[default FALSE] behaviour on error: if FALSE, returns an integer exit code accompanied by a warning, or, if TRUE, generates an error and halts execution.

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 = "tcp://127.0.0.1:6545", autostart = FALSE)
```

```
socket$dialer
start(socket$dialer[[1]])
socket$dialer
close(socket$dialer[[1]])
close(socket)

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

`is_aio` *Validators*

Description

Validator functions for object types created by {nanonext}.

Usage

```
is_aio(x)

is_nano(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, or nano Object.

Value

Logical value TRUE or FALSE.

Examples

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

```
close(sock)

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

`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,
  error = FALSE
)
```

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.
`error`	[default FALSE] behaviour on error: if FALSE, returns an integer exit code accompanied by a warning, or, if TRUE, generates an error and halts execution.

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 = "tcp://127.0.0.1:6547", autostart = FALSE)
socket$listener
start(socket$listener[[1]])
socket$listener
close(socket$listener[[1]])
close(socket)

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

---

lock *Lock / Unlock a Socket*

Description

Prevents further pipe connections from being established at a Socket. If a socket is locked, new pipe connections are closed before they can be added to the socket.

Usage

```
lock(socket, cv = NULL)

unlock(socket)
```

Arguments

socket	a Socket.
cv	(optional) a 'conditionVariable'. If supplied, the socket is locked only whilst the condition variable is an odd value. This is designed to allow an initial connection, as well as subsequent re-connections after a connection has ended, if the conditon variable is also registered with pipe_notify for both add and remove pipe events.

Value

Invisibly, zero on success (will otherwise error).

Examples

```
s <- socket("bus", listen = "inproc://nanolock")
s1 <- socket("bus", dial = "inproc://nanolock")
lock(s)
s2 <- socket("bus", dial = "inproc://nanolock")

send(s, "test")
recv(s1)
recv(s2)

unlock(s)
s3 <- socket("bus", dial = "inproc://nanolock")
send(s, "test")
recv(s1)
recv(s3)

close(s)
close(s1)
close(s2)
close(s3)
```

---

mclock *Clock Utility*

Description

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

Usage

```
mclock()
```

Details

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

Value

A double.

Examples

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

---

messenger *Messenger*

Description

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

Usage

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

Arguments

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

Value

Invisible NULL.

Usage

Type outgoing messages and hit return to send.

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

`:q` is the command to quit.

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

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

msleep *Sleep Utility*

Description

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

Usage

```
msleep(time)
```

Arguments

time	integer number of milliseconds to block the caller.

Details

Non-integer values for 'time' are coerced to integer, and the absolute value is taken (the sign is ignored). 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", "push", "pull", "pub", "sub", "req", "rep", "surveyor",
    "respondent"),
  dial = NULL,
  listen = NULL,
  tls = NULL,
  autostart = TRUE
)
```

Arguments

protocol	[default 'bus'] choose protocol - 'bus', 'pair', 'push', 'pull', 'pub', 'sub', 'req', 'rep', 'surveyor', or 'respondent' - see protocols.
dial	(optional) a URL to dial, specifying the transport and address as a character string e.g. 'inproc://anyvalue' or 'tcp://127.0.0.1:5555' (see transports).
listen	(optional) a URL to listen at, specifying the transport and address as a character string e.g. 'inproc://anyvalue' or 'tcp://127.0.0.1:5555' (see transports).
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 (defaults to 'GET' if not specified).
headers	(optional) a named character vector specifying the HTTP request headers, for example: `c(Authorization = "Bearer APIKEY", `Content-Type` = "text/plain")` A non-character or non-named vector will be ignored.
data	(optional) character request data to be submitted.
response	(optional) 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.
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 supplied in 'response', or NULL otherwise. If the status code is within the 300 range, i.e. a redirect, the response header 'Location' is automatically appended to return the redirect address.
- `$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).

See Also

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

Examples

```
ncurl("https://postman-echo.com/get",
       convert = FALSE,
       response = c("date", "content-type"),
       timeout = 1200L)
ncurl("https://postman-echo.com/put",
      method = "PUT",
      headers = c(Authorization = "Bearer APIKEY"),
      data = "hello world",
      timeout = 1500L)
ncurl("https://postman-echo.com/post",
      method = "POST",
      headers = c(`Content-Type` = "application/json"),
      data = '{"key":"value"}',
      timeout = 1500L)
```

`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 (defaults to 'GET' if not specified).
`headers`	(optional) a named character vector specifying the HTTP request headers, for example: `c(Authorization = "Bearer APIKEY", `Content-Type` = "text/plain")` A non-character or non-named vector will be ignored.

data	(optional) character request data to be submitted.
response	(optional) 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.
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 supplied in 'response', or NULL otherwise. If the status code is within the 300 range, i.e. a redirect, the response header 'Location' is automatically appended to return the redirect address.
- $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).

See Also

ncurl_session for persistent connections.

Examples

```
nc <- ncurl_aio("https://www.r-project.org/",
                response = c("date", "server"),
                timeout = 2000L)
call_aio(nc)
nc$status
nc$headers
nc$data
```

---

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 (defaults to 'GET' if not specified).
headers	(optional) a named character vector specifying the HTTP request headers, for example: `c(Authorization = "Bearer APIKEY", `Content-Type` = "text/plain")` A non-character or non-named vector will be ignored.
data	(optional) character request data to be submitted.
response	(optional) 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.
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 (if specified in the session), or NULL otherwise. If the status code is within the 300 range, i.e. a redirect, the response header 'Location' is automatically appended to return the redirect address.
- `$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`.

See Also

ncurl_aio for asynchronous http requests.

Examples

```
s <- ncurl_session("https://www.r-project.org/", response = "date", timeout = 2000L)
s
if (!is_error_value(s)) transact(s)
if (!is_error_value(s)) close(s)
```

`next_config` *Configure Next Mode*

Description

Configures send mode 'next' by registering functions for custom serialization and unserialization of external pointer reference objects, allowing these to be sent and received between different R sessions.

Usage

```
next_config(refhook = list(), mark = FALSE)
```

Arguments

`refhook`	**either** a list or pairlist of two functions: the signature for the first must accept a list of external pointer objects and return a raw vector, e.g. `torch::torch_serialize`, and the second must accept a raw vector and return a list of external pointer objects, e.g. `torch::torch_load`, **or else** NULL to reset.
`mark`	[default FALSE] (for advanced use only) logical value, whether to mark serialized data with a special bit.

Details

Calling this function without any arguments returns the pairlist of currently-registered 'refhook' functions (and resets 'mark' to FALSE).

Value

A pairlist comprising the currently-registered 'refhook' functions.

Examples

```
g <- next_config(refhook = list(function(x) serialize(x, NULL), unserialize))
next_config()
next_config(g, mark = TRUE)

next_config(NULL)
next_config()
```

`nng_error` *Translate Error Codes*

Description

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

Usage

```
nng_error(xc)
```

Arguments

`xc`	integer exit code to translate.

Value

A character vector.

Examples

```
nng_error(1L)
```

`nng_version` *NNG Library Version*

Description

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

Usage

```
nng_version()
```

Value

A character vector of length 2.

Examples

```
nng_version()
```

`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').

Global Options

- 'reconnect-time-min' [type 'ms']

  This is the minimum amount of time (milliseconds) to wait before attempting to establish a connection after a previous attempt has failed. This can be set on a socket, but it can also be overridden on an individual dialer. The option is irrelevant for listeners.

- 'reconnect-time-max' [type 'ms']

  This is the maximum amount of time (milliseconds) to wait before attempting to establish a connection after a previous attempt has failed. If this is non-zero, then the time between successive connection attempts will start at the value of 'reconnect-time-min', and grow exponentially, until it reaches this value. If this value is zero, then no exponential back-off between connection attempts is done, and each attempt will wait the time specified by 'reconnect-time-min'. This can be set on a socket, but it can also be overridden on an individual dialer. The option is irrelevant for listeners.

- 'recv-size-max' [type 'size']

  This is the maximum message size that the will be accepted from a remote peer. If a peer attempts to send a message larger than this, then the message will be discarded. If the value of this is zero, then no limit on message sizes is enforced. This option exists to prevent certain kinds of denial-of-service attacks, where a malicious agent can claim to want to send an extraordinarily large message, without sending any data. This option can be set for the socket, but may be overridden for on a per-dialer or per-listener basis. NOTE: Applications on hostile networks should set this to a non-zero value to prevent denial-of-service attacks. NOTE: Some transports may have further message size restrictions.

- 'recv-buffer' [type 'int']

  This is the depth of the socket's receive buffer as a number of messages. Messages received by a transport may be buffered until the application has accepted them for delivery. This value must be an integer between 0 and 8192, inclusive. NOTE: Not all protocols support buffering received messages. For example req can only deal with a single reply at a time.

- 'recv-timeout' [type 'ms']

  This is the socket receive timeout in milliseconds. When no message is available for receiving at the socket for this period of time, receive operations will fail with a return value of 5L ('timed out').

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

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

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

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

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

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

- '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 10, comprising:

- `rawurl` - the unparsed URL string.
- `scheme` - the URL scheme, such as "http" or "inproc" (always lower case).
- `userinfo` - the username and password if supplied in the URL string.
- `host` - the full host part of the URL, including the port if present (separated by a colon).
- `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.
- `requri` - the full Request-URI (path[?query][#fragment]).

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_notify` *Pipe Notify*

Description

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

Usage

```
pipe_notify(socket, cv, cv2 = NULL, add = FALSE, remove = FALSE, flag = FALSE)
```

Arguments

`socket`	a Socket.
`cv`	a 'conditionVariable' to signal.
`cv2`	[default NULL] optionally, if specified, a second 'conditionVariable' to signal. Note that this cv is signalled sequentially after the first condition variable.
`add`	[default FALSE] logical value whether to signal when a pipe is added.
`remove`	[default FALSE] logical value whether to 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.

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()
cv2 <- cv()

pipe_notify(s, cv, cv2, add = TRUE, remove = TRUE, flag = TRUE)
cv_value(cv)
cv_value(cv2)

s1 <- socket(dial = "inproc://nanopipe")
cv_value(cv)
cv_value(cv2)
close(s1)
cv_value(cv)
cv_value(cv2)

(wait(cv))
(wait(cv2))

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] 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.

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.

`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).
`convert`	[default TRUE] logical FALSE to return a raw vector, or TRUE to return the hex representation of the bytes as a character string.

Details

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

Value

A length 'n' 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)
```

`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,
  n = 65536L
)
```

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' is not an option and the default is '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.
n	[default 65536L] applicable to Streams only, the maximum number of bytes to receive. Can be an over-estimate, but note that a buffer of this size is reserved.

Details

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

If 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

The received data in the 'mode' specified.

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).

A signalling version of the function takes a 'conditionVariable' as an additional argument and signals it when the async receive is complete.

Usage

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

recv_aio_signal(
  con,
  cv,
 mode = c("serial", "character", "complex", "double", "integer", "logical", "numeric",
    "raw", "string"),
  timeout = NULL,
  n = 65536L
)
```

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' is not an option and the default is 'character'.
timeout	[default NULL] integer value in milliseconds or NULL, which applies a socket-specific default, usually the same as no timeout.
n	[default 65536L] applicable to Streams only, the maximum number of bytes to receive. Can be an over-estimate, but note that a buffer of this size is reserved.
cv	**For the signalling version**: 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).

For the signalling version: when the receive is complete, the supplied 'conditionVariable' is signalled by incrementing its value by 1. This happens asynchronously and independently of the R execution thread.

Value

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

Examples

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

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

res <- send_aio(s1, c(1.1, 2.2, 3.3), mode = "raw", timeout = 100)
msg <- recv_aio(s2, mode = "double", timeout = 100)
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 = "tcp://127.0.0.1:6546")
cv <- cv()
msg <- recv_aio_signal(s1, timeout = 100, cv = cv)
until(cv, 10L)
msg$data
```

```
close(s1)

# in another process in parallel
s2 <- socket("pair", dial = "tcp://127.0.0.1:6546")
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"),
  send_mode = c("serial", "raw", "next"),
  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 - one of 'serial' (1L) to send serialised R objects, 'raw' (2L) to send atomic vectors of any type as a raw byte vector, or 'next' (3L) - see 'Send Modes' section below.
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.

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.

Mode 'next' sends serialised R objects, with native extensions enabled by `next_config`. This configures custom serialization and unserialization functions for external pointer reference objects. When receiving, mode 'serial' should be used as 'next' sends are fully compatible.

Examples

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

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

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

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

close(req)
close(rep)
```

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

Description

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

A signalling version of the function takes a 'conditionVariable' as an additional argument and signals it when the async receive is complete.

Usage

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

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

Arguments

context	a Context.
data	an object (if send_mode = 'raw', a vector).
send_mode	[default 'serial'] character value or integer equivalent - one of 'serial' (1L) to send serialised R objects, 'raw' (2L) to send atomic vectors of any type as a raw byte vector, or 'next' (3L) - see 'Send Modes' section below.
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 **For the signalling version**: a 'conditionVariable' to signal when the async receive is complete.

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. For safety, the context used for the request is closed when all references to the returned 'recvAio' are removed and the object is garbage collected.

For the signalling version: when the receive is complete, the supplied 'conditionVariable' is signalled by incrementing its value by 1. This happens asynchronously and independently of the R execution thread.

Value

A 'recvAio' (object of class '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.

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.

Mode 'next' sends serialised R objects, with native extensions enabled by `next_config`. This configures custom serialization and unserialization functions for external pointer reference objects. When receiving, mode 'serial' should be used as 'next' sends are fully compatible.

Examples

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

# works if req and rep are running in parallel in different processes
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_signal(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)
```

---

send *Send*

Description

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

Usage

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

Arguments

con	a Socket, Context or Stream.
data	an object (a vector, if mode = 'raw').
mode	[default 'serial'] character value or integer equivalent - one of 'serial' (1L) to send serialised R objects, 'raw' (2L) to send atomic vectors of any type as a raw byte vector, or 'next' (3L) - see 'Send Modes' section below. 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.

Value

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.

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.

Mode 'next' sends serialised R objects, with native extensions enabled by `next_config`. This configures custom serialization and unserialization functions for external pointer reference objects. When receiving, mode 'serial' should be used as 'next' sends are fully compatible.

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 or Stream).

Usage

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

Arguments

con	a Socket, Context or Stream.
data	an object (a vector, if mode = 'raw').
mode	[default 'serial'] character value or integer equivalent - one of 'serial' (1L) to send serialised R objects, 'raw' (2L) to send atomic vectors of any type as a raw byte vector, or 'next' (3L) - see 'Send Modes' section below. 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.

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.

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.

Mode 'next' sends serialised R objects, with native extensions enabled by `next_config`. This configures custom serialization and unserialization functions for external pointer reference objects. When receiving, mode 'serial' should be used as 'next' sends are fully compatible.

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)
```

---

sha256 *Cryptographic Hashing Using the SHA-2 Algorithms*

Description

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

Usage

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

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

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

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

Arguments

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

Details

For arguments 'x' and 'key', a scalar string or raw vector (with no attributes) is hashed 'as is'.

All other objects are first serialized using R serialization v3 XDR, with headers skipped (for portability as these contain R version and encoding information).

The result of hashing is always a byte sequence, which is converted to a character string hex representation if 'convert' is TRUE, or returned as a raw vector if 'convert' is FALSE.

Value

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

Examples

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

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

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

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

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

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

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

---

socket *Open Socket*

Description

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

Usage

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

```
  listen = NULL,
  tls = NULL,
  autostart = TRUE,
  raw = FALSE
)
```

Arguments

protocol	[default 'bus'] choose protocol - 'bus', 'pair', 'push', 'pull', 'pub', 'sub', 'req', 'rep', 'surveyor', or 'respondent' - see protocols.
dial	(optional) a URL to dial, specifying the transport and address as a character string e.g. 'inproc://anyvalue' or 'tcp://127.0.0.1:5555' (see transports).
listen	(optional) a URL to listen at, specifying the transport and address as a character string e.g. 'inproc://anyvalue' or 'tcp://127.0.0.1:5555' (see transports).
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'
- 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, 'Non-standard Response' is returned, which may be a custom code used by the server.

Usage

```
status_code(x)
```

Arguments

x	numeric HTTP status code to translate.

Value

A character vector.

Examples

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

stop_aio *Stop Asynchronous Aio Operation*

Description

Stop an asynchronous Aio operation.

Usage

```
stop_aio(aio)
```

Arguments

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

Details

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

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

Value

Invisible NULL.

---

strcat *Concatenate Strings*

Description

A fast implementation that combines two character values into a single string.

Usage

```
strcat(a, b)
```

Arguments

a	character value.
b	character value.

Details

If either 'a' or 'b' is a vector of length greater than 1, only the first element of each is concatenated.

Value

A character string.

Examples

```
strcat("hello ", "world!")
```

`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, tls = NULL)
```

Arguments

`dial`	a URL to dial, specifying the transport and address as a character string e.g. 'ipc:///tmp/anyvalue' or 'tcp://127.0.0.1:5555' (not all transports are supported).
`listen`	a URL to listen at, specifying the transport and address as a character string e.g. 'ipc:///tmp/anyvalue' or 'tcp://127.0.0.1:5555' (not all transports are supported).
`textframes`	[default FALSE] applicable to the websocket transport only, enables sending and receiving of TEXT frames (ignored otherwise).
`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.

Details

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

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

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

Value

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

Examples

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

`subscribe` *Subscribe / 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.

For creating client configurations for public internet usage, root CA ceritficates may usually be found at '/etc/ssl/certs/ca-certificates.crt' on Linux systems. Otherwise, root CA certificates in PEM format are available at 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.*

Value

A 'tlsConfig' object.

Examples

```
tls <- tls_config()
tls
ncurl("https://www.r-project.org/", timeout = 1000L, tls = tls)
```

---

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.

BSD Socket (experimental)

The socket transport provides communication support between peers across arbitrary BSD sockets, such as those created with socketpair.

[**URI, socket://**] This transport uses the URL socket://, without further qualification.

This transport only supports listeners. The socket file descriptor is passed to the listener using the 'socket:fd' option (as an integer). Setting this option (which is write-only and can be set multiple times) will cause the listener to create a pipe backed by the file descriptor.

`unresolved` *Query if an Aio is Unresolved*

Description

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

Usage

```
unresolved(aio)
```

Arguments

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

Details

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

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

Value

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

Examples

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

while (unresolved(aio)) {
  # do stuff before checking resolution again
  cat("unresolved\n")
  msleep(20)
}

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 = "localhost", valid = "20301231235959")
```

Arguments

cn	[default 'localhost'] 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

For interactive sessions only, a status message is printed at the start of key / certificate generation and also when complete.

Value

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

Examples

```
if (interactive()) {
# Only run examples in interactive R sessions

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

}
```

`%~>%` *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)
```

Index