Package 'mirai'

February 2, 2024

Type Package

Title Minimalist Async Evaluation Framework for R

Version 0.12.1

Description Lightweight parallel code execution and distributed computing. Designed for simplicity, a 'mirai' evaluates an R expression asynchronously, on local or network resources, resolving automatically upon completion. Efficient scheduling over fast inter-process communications or secure TLS connections over TCP/IP, built on 'nanonext' and 'NNG' (Nanomsg Next Gen).

License GPL (>= 3)

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

URL https://shikokuchuo.net/mirai/,

https://github.com/shikokuchuo/mirai/

Encoding UTF-8

Depends R (>= 3.5)

Imports nanonext (>= 0.12.0)

Enhances parallel, promises

Suggests knitr, markdown

VignetteBuilder knitr

RoxygenNote 7.3.1

NeedsCompilation no

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

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

Repository CRAN

Date/Publication 2024-02-02 08:30:02 UTC

R topics documented:

`mirai-package` *mirai: Minimalist Async Evaluation Framework for R*

Description

Lightweight parallel code execution and distributed computing. Designed for simplicity, a 'mirai' evaluates an R expression asynchronously, on local or network resources, resolving automatically upon completion. Efficient scheduling over fast inter-process communications or secure TLS connections over TCP/IP, built on 'nanonext' and 'NNG' (Nanomsg Next Gen).

Notes

For local mirai requests, the default transport for inter-process communications is platform-dependent: abstract Unix domain sockets on Linux, Unix domain sockets on MacOS, Solaris and other POSIX platforms, and named pipes on Windows.

This may be overriden, if desired, by specifying 'url' in the `daemons` interface and launching daemons using `launch_local`.

Reference Manual

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

Author(s)

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

See Also

Useful links:

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

---

as.promise.mirai *Make Mirai Promise*

Description

Creates a 'promise' from a 'mirai'.

Usage

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

Arguments

x	an object of class 'mirai'.

Details

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

Requires the **promises** package.

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

Value

A 'promise' object.

Examples

```
if (interactive() && requireNamespace("promises", quietly = TRUE)) {

library(promises)

p <- as.promise(mirai("example"))
print(p)
is.promise(p)

p2 <- mirai("completed") %...>% identity()
p2$then(cat)
is.promise(p2)

}
```

`call_mirai` *mirai (Call Value)*

Description

`call_mirai` retrieves the value of a mirai, waiting for the the asynchronous operation to resolve if it is still in progress.

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

Usage

```
call_mirai(aio)

call_mirai_(aio)
```

Arguments

`aio`	a 'mirai' object.

Details

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

If an error occurs in evaluation, the error message is returned as a character string of class 'miraiError' and 'errorValue'. `is_mirai_error` may be used to test for this.

`is_error_value` tests for all error conditions including mirai errors, interrupts, and timeouts.

The mirai updates itself in place, so to access the value of a mirai `x` directly, use `call_mirai(x)$data`.

Value

The passed mirai (invisibly). The retrieved value is stored at `$data`.

Alternatively

The value of a mirai may be accessed at any time at `$data`, and if yet to resolve, an 'unresolved' logical NA will be returned instead.

Using `unresolved` on a mirai returns TRUE only if a mirai has yet to resolve and FALSE otherwise. This is suitable for use in control flow statements such as `while` or `if`.

Examples

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

# using call_mirai()
df1 <- data.frame(a = 1, b = 2)
df2 <- data.frame(a = 3, b = 1)
m <- mirai(as.matrix(rbind(df1, df2)), .args = list(df1, df2), .timeout = 1000)
call_mirai(m)$data

# using unresolved()
m <- mirai({
  res <- rnorm(n)
  res / rev(res)
  },
  n = 1e6)
while (unresolved(m)) {
  cat("unresolved\n")
  Sys.sleep(0.1)
}
str(m$data)

}
```

---

daemon *Daemon Instance*

Description

Starts up an execution daemon to receive `mirai` requests. Awaits data, evaluates an expression in an environment containing the supplied data, and returns the value to the host caller. Daemon settings may be controlled by `daemons` and this function should not need to be invoked directly, unless deploying manually on remote resources.

Usage

```
daemon(
  url,
  autoexit = TRUE,
  cleanup = TRUE,
```

```
  output = FALSE,
  maxtasks = Inf,
  idletime = Inf,
  walltime = Inf,
  timerstart = 0L,
  ...,
  tls = NULL,
  rs = NULL
)
```

Arguments

`url`	the character host or dispatcher URL to dial into, including the port to connect to (and optionally for websockets, a path), e.g. 'tcp://hostname:5555' or 'ws://10.75.32.70:5555/path'.
`autoexit`	[default TRUE] logical value, whether the daemon should exit automatically when its socket connection ends. If a signal from the **tools** package, e.g. `tools::SIGINT`, or an equivalent integer value is supplied, this signal is additionally raised (see 'Persistence' section below).
`cleanup`	[default TRUE] logical value, whether to perform cleanup of the global environment and restore loaded packages and options to an initial state after each evaluation. For more granular control, also accepts an integer value (see 'Cleanup Options' section below).
`output`	[default FALSE] logical value, to output generated stdout / stderr if TRUE, or else discard if FALSE. Specify as TRUE in the '...' argument to `daemons` or `launch_local` to provide redirection of output to the host process (applicable only for local daemons).
`maxtasks`	[default Inf] the maximum number of tasks to execute (task limit) before exiting.
`idletime`	[default Inf] maximum idle time, since completion of the last task (in milliseconds) before exiting.
`walltime`	[default Inf] soft walltime, or the minimum amount of real time taken (in milliseconds) before exiting.
`timerstart`	[default 0L] number of completed tasks after which to start the timer for 'idletime' and 'walltime'. 0L implies timers are started upon launch.
`...`	reserved but not currently used.
`tls`	[default NULL] required for secure TLS connections over 'tls+tcp://' or 'wss://'. **Either** the character path to a file containing X.509 certificate(s) in PEM format, comprising the certificate authority certificate chain starting with the TLS certificate and ending with the CA certificate, **or** a length 2 character vector comprising [i] the certificate authority certificate chain and [ii] the empty character `''`.
`rs`	[default NULL] the initial value of .Random.seed. This is set automatically using L'Ecuyer-CMRG RNG streams generated by the host process and should not be independently supplied.

Details

The network topology is such that daemons dial into the host or dispatcher, which listens at the 'url' address. In this way, network resources may be added or removed dynamically and the host or dispatcher automatically distributes tasks to all available daemons.

Value

Invisible NULL.

Persistence

The 'autoexit' argument governs persistence settings for the daemon. The default TRUE ensures that it will exit cleanly once its socket connection has ended.

Instead of TRUE, supplying a signal from the **tools** package, e.g. `tools::SIGINT`, or an equivalent integer value, sets the signal to be raised when the socket connection ends. As an example, supplying SIGINT allows a potentially more immediate exit by interrupting any ongoing evaluation rather than letting it complete.

Setting to FALSE allows the daemon to persist indefinitely even when there is no longer a socket connection. This allows a host session to end and a new session to connect at the URL where the daemon is dialled in. Daemons must be terminated with `daemons(NULL)` in this case, which sends explicit exit instructions to all connected daemons.

Persistence also implies that dials are performed asynchronously, which means retries are attempted (indefinitely) if not immediately successful. This is resilient behaviour but can mask potential connection issues.

Cleanup Options

The 'cleanup' argument also accepts an integer value, which operates an additive bitmask: perform cleanup of the global environment (1L), reset loaded packages to an initial state (2L), restore options to an initial state (4L), and perform garbage collection (8L).

As an example, to perform cleanup of the global environment and garbage collection, specify 9L (1L + 8L). The default argument value of TRUE performs all actions apart from garbage collection and is equivalent to a value of 7L.

Caution: do not reset options but not loaded packages if packages set options on load.

`daemons` *Daemons (Set Persistent Processes)*

Description

Set 'daemons' or persistent background processes to receive `mirai` requests. Specify 'n' to create daemons on the local machine. Specify 'url' for receiving connections from remote daemons (for distributed computing across the network). Specify 'remote' to optionally launch remote daemons via a remote configuration. By default, dispatcher ensures optimal scheduling.

Usage

```
daemons(
  n,
  url = NULL,
  remote = NULL,
  dispatcher = TRUE,
  ...,
  resilience = TRUE,
  seed = NULL,
  tls = NULL,
  pass = NULL,
  .compute = "default"
)
```

Arguments

n	integer number of daemons to set.
url	[default NULL] if specified, the character URL or vector of URLs on the host for remote daemons to dial into, including a port accepting incoming connections (and optionally for websockets, a path), e.g. 'tcp://hostname:5555' or 'ws://10.75.32.70:5555/path'. Specify a URL starting 'tls+tcp://' or 'wss://' to use secure TLS connections. Auxiliary function host_url may be used to construct a valid host URL.
remote	[default NULL] required only for launching remote daemons, a configuration generated by remote_config or ssh_config.
dispatcher	[default TRUE] logical value whether to use dispatcher. Dispatcher is a local background process that connects to daemons on behalf of the host and ensures FIFO scheduling (see Dispatcher section below).
...	(optional) additional arguments passed through to dispatcher if using dispatcher and/or daemon if launching daemons. These include 'token' at dispatcher and 'autoexit, 'cleanup', 'output', 'maxtasks', 'idletime', 'walltime' and 'timerstart' at daemon.
resilience	[default TRUE] (applicable when not using dispatcher) logical value whether to retry failed tasks on other daemons. If FALSE, an appropriate 'errorValue' will be returned in such cases.
seed	[default NULL] (optional) supply a random seed (single value, interpreted as an integer). This is used to inititalise the L'Ecuyer-CMRG RNG streams sent to each daemon. Note that reproducible results can be expected only for 'dispatcher = FALSE', as the unpredictable timing of task completions would otherwise influence the tasks sent to each daemon. Even for 'dispatcher = FALSE', reproducibility is not guaranteed if the order in which tasks are sent is not deterministic.
tls	[default NULL] (optional for secure TLS connections) if not supplied, zero-configuration single-use keys and certificates are automatically generated. If supplied, **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 TLS certificate first), **or** a length 2 character vector comprising [i] the TLS certificate (optionally certificate chain) and [ii] the associated private key.

`pass` [default NULL] (required only if the private key supplied to 'tls' is encrypted with a password) For security, should be provided through a function that returns this value, rather than directly.

`.compute` [default 'default'] character value for the compute profile to use (each compute profile has its own independent set of daemons).

Details

Use `daemons(0)` to reset daemon connections:

- A reset is required before revising settings for the same compute profile, otherwise changes are not registered.
- All connected daemons and/or dispatchers exit automatically.
- **mirai** reverts to the default behaviour of creating a new background process for each request.
- Any unresolved 'mirai' will return an 'errorValue' 7 (Object closed) after a reset.

If the host session ends, all connected dispatcher and daemon processes automatically exit as soon as their connections are dropped (unless the daemons were started with `autoexit = FALSE`). If a daemon is processing a task, it will exit as soon as the task is complete.

To reset persistent daemons started with `autoexit = FALSE`, use `daemons(NULL)` instead, which also sends exit instructions to all connected daemons prior to resetting.

For historical reasons, `daemons()` with no arguments returns the value of `status`.

Value

Depending on the arguments supplied:

- using dispatcher: integer number of daemons set.
- or else launching local daemons: integer number of daemons launched.
- otherwise: the character host URL.

Local Daemons

Daemons provide a potentially more efficient solution for asynchronous operations as new processes no longer need to be created on an *ad hoc* basis.

Supply the argument 'n' to set the number of daemons. New background `daemon` processes are automatically created on the local machine connecting back to the host process, either directly or via dispatcher.

Dispatcher

By default `dispatcher = TRUE`. This launches a background process running `dispatcher`. Dispatcher connects to daemons on behalf of the host and ensures FIFO scheduling of tasks. Dispatcher uses synchronisation primitives from **nanonext**, waiting rather than polling for tasks, which is both efficient (no resource usage) and fully event-driven (having no latency).

By specifying `dispatcher = FALSE`, daemons connect to the host directly rather than through dispatcher. The host sends tasks to connected daemons immediately in an evenly-distributed fashion. However, optimal scheduling is not guaranteed as the duration of tasks cannot be known *a priori*, such that tasks can be queued at one daemon while other daemons remain idle. Nevertheless, this provides a resource-light approach suited to working with similar-length tasks, or where concurrent tasks typically do not exceed available daemons.

Distributed Computing

Specifying 'url' allows tasks to be distributed across the network. This should be a character string such as: 'tcp://10.75.32.70:5555' at which daemon processes should connect to. Switching the URL scheme to 'tls+tcp://' or 'wss://' automatically upgrades the connection to use TLS. The auxiliary function `host_url` may be used to automatically construct a valid host URL based on the computer's hostname.

Specify 'remote' with a call to `remote_config` or `ssh_config` to launch daemons on remote machines. Otherwise, `launch_remote` may be used to generate the shell commands to deploy daemons manually on remote resources.

IPv6 addresses are also supported and must be enclosed in square brackets [ ] to avoid confusion with the final colon separating the port. For example, port 5555 on the IPv6 loopback address ::1 would be specified as 'tcp://[::1]:5555'.

Specifying the wildcard value zero for the port number e.g. 'tcp://[::1]:0' or 'ws://[::1]:0' will automatically assign a free ephemeral port. Use `status` to inspect the actual assigned port at any time.

With Dispatcher

When using dispatcher, it is recommended to use a websocket URL rather than TCP, as this requires only one port to connect to all daemons: a websocket URL supports a path after the port number, which can be made unique for each daemon.

Specifying a single host URL such as 'ws://10.75.32.70:5555' with `n = 6` will automatically append a sequence to the path, listening to the URLs 'ws://10.75.32.70:5555/1' through 'ws://110.75.32.70:5555/6'.

Alternatively, specify a vector of URLs to listen to arbitrary port numbers / paths. In this case it is optional to supply 'n' as this can be inferred by the length of vector supplied.

Individual daemons then dial in to each of these host URLs. At most one daemon can be dialled into each URL at any given time.

Dispatcher automatically adjusts to the number of daemons actually connected. Hence it is possible to dynamically scale up or down the number of daemons as required, subject to the maximum number initially specified.

Alternatively, supplying a single TCP URL will listen at a block of URLs with ports starting from the supplied port number and incrementing by one for 'n' specified e.g. the host URL 'tcp://10.75.32.70:5555' with `n = 6` listens to the contiguous block of ports 5555 through 5560.

Without Dispatcher

A TCP URL may be used in this case as the host listens at only one address, utilising a single port.

The network topology is such that daemons (started with `daemon`) or indeed dispatchers (started with `dispatcher`) dial into the same host URL.

'n' is not required in this case, and disregarded if supplied, as network resources may be added or removed at any time. The host automatically distributes tasks to all connected daemons and dispatchers in a round-robin fashion.

Compute Profiles

By default, the 'default' compute profile is used. Providing a character value for '.compute' creates a new compute profile with the name specified. Each compute profile retains its own daemons settings, and may be operated independently of each other. Some usage examples follow:

local / remote daemons may be set with a host URL and specifying '.compute' as 'remote', which creates a new compute profile. Subsequent mirai calls may then be sent for local computation by not specifying the '.compute' argument, or for remote computation to connected daemons by specifying the '.compute' argument as 'remote'.

cpu / gpu some tasks may require access to different types of daemon, such as those with GPUs. In this case, `daemons()` may be called twice to set up host URLs for CPU-only daemons and for those with GPUs, specifying the '.compute' argument as 'cpu' and 'gpu' respectively. By supplying the '.compute' argument to subsequent mirai calls, tasks may be sent to either 'cpu' or 'gpu' daemons as appropriate.

Note: further actions such as resetting daemons via `daemons(0)` should be carried out with the desired '.compute' argument specified.

Examples

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

# Create 2 local daemons (using dispatcher)
daemons(2)
status()
# Reset to zero
daemons(0)

# Create 2 local daemons (not using dispatcher)
daemons(2, dispatcher = FALSE)
status()
# Reset to zero
daemons(0)

# 2 remote daemons via dispatcher using WebSockets
daemons(2, url = host_url(ws = TRUE))
status()
# Reset to zero
daemons(0)

# Set host URL for remote daemons to dial into
```

```
daemons(url = host_url(), dispatcher = FALSE)
status()
# Reset to zero
daemons(0)

}

## Not run:
# Launch 2 daemons on remotes 'nodeone' and 'nodetwo' using SSH
# connecting back directly to the host URL over a TLS connection:

daemons(url = host_url(tls = TRUE),
        remote = ssh_config(c('ssh://nodeone', 'ssh://nodetwo')),
        dispatcher = FALSE)

# Launch 4 daemons on the remote machine 10.75.32.90 using SSH tunnelling
# over port 5555 ('url' hostname must be 'localhost' or '127.0.0.1'):

daemons(n = 4,
        url = 'ws://localhost:5555',
        remote = ssh_config('ssh://10.75.32.90', tunnel = TRUE))


## End(Not run)
```

`dispatcher` *Dispatcher*

Description

Dispatches tasks from a host to daemons for processing, using FIFO scheduling, queuing tasks as required. Daemon / dispatcher settings may be controlled by `daemons` and this function should not need to be invoked directly.

Usage

```
dispatcher(
  host,
  url = NULL,
  n = NULL,
  ...,
  asyncdial = FALSE,
  token = FALSE,
  tls = NULL,
  pass = NULL,
  rs = NULL,
  monitor = NULL
)
```

Arguments

host	the character host URL to dial (where tasks are sent from), including the port to connect to (and optionally for websockets, a path), e.g. 'tcp://hostname:5555' or 'ws://10.75.32.70:5555/path'.
url	(optional) the character URL or vector of URLs dispatcher should listen at, including the port to connect to (and optionally for websockets, a path), e.g. 'tcp://hostname:5555' or 'ws://10.75.32.70:5555/path'. Specify 'tls+tcp://' or 'wss://' to use secure TLS connections. Tasks are sent to daemons dialled into these URLs. If not supplied, 'n' local inter-process URLs will be assigned automatically.
n	(optional) if specified, the integer number of daemons to listen for. Otherwise 'n' will be inferred from the number of URLs supplied in 'url'. Where a single URL is supplied and 'n' > 1, 'n' unique URLs will be automatically assigned for daemons to dial into.
...	(optional) additional arguments passed through to daemon. These include 'autoexit', 'cleanup', 'maxtasks', 'idletime', 'walltime' and 'timerstart'.
asyncdial	[default FALSE] whether to perform dials asynchronously. The default FALSE will error if a connection is not immediately possible (e.g. daemons has yet to be called on the host, or the specified port is not open etc.). Specifying TRUE continues retrying (indefinitely) if not immediately successful, which is more resilient but can mask potential connection issues.
token	[default FALSE] if TRUE, appends a unique 24-character token to each URL path the dispatcher listens at (not applicable for TCP URLs which do not accept a path).
tls	[default NULL] (required for secure TLS connections) **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 TLS certificate first), **or** a length 2 character vector comprising [i] the TLS certificate (optionally certificate chain) and [ii] the associated private key.
pass	[default NULL] (required only if the private key supplied to 'tls' is encrypted with a password) For security, should be provided through a function that returns this value, rather than directly.
rs	[default NULL] the initial value of .Random.seed. This is set automatically using L'Ecuyer-CMRG RNG streams generated by the host process and should not be independently supplied.
monitor	(for package internal use only) do not set this parameter.

Details

The network topology is such that a dispatcher acts as a gateway between the host and daemons, ensuring that tasks received from the host are dispatched on a FIFO basis for processing. Tasks are queued at the dispatcher to ensure tasks are only sent to daemons that can begin immediate execution of the task.

Value

Invisible NULL.

`everywhere` *Evaluate Everywhere*

Description

Evaluate an expression 'everywhere' on all connected daemons for the specified compute profile. Designed for performing setup operations across daemons or exporting common data, resultant changes to the global environment, loaded packages or options are persisted regardless of a daemon's 'cleanup' setting.

Usage

```
everywhere(.expr, ..., .args = list(), .compute = "default")
```

Arguments

`.expr`	an expression to evaluate asynchronously (of arbitrary length, wrapped in { } where necessary), **or** a language object passed by name.
`...`	(optional) named arguments (name = value pairs) specifying objects referenced in '.expr'. Used in addition to, and taking precedence over, any arguments specified via '.args'.
`.args`	(optional) **either** a list of objects passed by name (found in the current scope), **or else** a list of name = value pairs, as in '...'.
`.compute`	[default 'default'] character value for the compute profile to use (each compute profile has its own independent set of daemons).

Value

Invisible NULL.

Examples

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

daemons(1)
# export common data by super-assignment:
everywhere(y <<- 3)
# assign explicitly to global environment:
everywhere(list2env(x, envir = .GlobalEnv), x = list(a = 1, b = 2))
m <- mirai(a + b - y == 0L)
call_mirai(m)$data
daemons(0)
```

```
daemons(1, dispatcher = FALSE)
everywhere(library(parallel))
m <- mirai("package:parallel" %in% search())
call_mirai(m)$data
daemons(0)

}
```

---

host_url *URL Constructors*

Description

host_url constructs a valid host URL (at which daemons may connect) based on the computer's hostname. This may be supplied directly to the 'url' argument of daemons.

local_url constructs a random URL suitable for local daemons.

Usage

```
host_url(ws = FALSE, tls = FALSE, port = 0)

local_url()
```

Arguments

ws	[default FALSE] logical value whether to use a WebSockets 'ws://' or else TCP 'tcp://' scheme.
tls	[default FALSE] logical value whether to use TLS in which case the scheme used will be either 'wss://' or 'tls+tcp://' accordingly.
port	[default 0] numeric port to use. This should be open to connections from the network addresses the daemons are connecting from. '0' is a wildcard value that automatically assigns a free ephemeral port.

Details

host_url relies on using the host name of the computer rather than an IP address and typically works on local networks, although this is not always guaranteed. If unsuccessful, substitute an IPv4 or IPv6 address in place of the hostname.

local_url generates a random URL for the platform's default inter-process communications transport: abstract Unix domain sockets on Linux, Unix domain sockets on MacOS, Solaris and other POSIX platforms, and named pipes on Windows.

Value

A character string comprising a valid URL.

Examples

```
host_url()
host_url(ws = TRUE)
host_url(tls = TRUE)
host_url(ws = TRUE, tls = TRUE, port = 5555)

local_url()
```

`is_mirai` *Is mirai*

Description

Is the object a 'mirai'.

Usage

```
is_mirai(x)
```

Arguments

`x`	an object.

Value

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

Examples

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

df <- data.frame()
m <- mirai(as.matrix(df), .args = list(df))
is_mirai(m)
is_mirai(df)

}
```

`is_mirai_error` *Error Validators*

Description

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

Usage

```
is_mirai_error(x)

is_mirai_interrupt(x)

is_error_value(x)
```

Arguments

`x`	an object.

Details

Is the object a 'miraiError'. When execution in a mirai process fails, the error message is returned as a character string of class 'miraiError' and 'errorValue'.

Is the object a 'miraiInterrupt'. When an ongoing mirai is sent a user interrupt, the mirai will resolve to an empty character string classed as 'miraiInterrupt' and 'errorValue'.

Is the object an 'errorValue', such as a mirai timeout, a 'miraiError' or a 'miraiInterrupt'. This is a catch-all condition that includes all returned error values.

Value

Logical value TRUE or FALSE.

Examples

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

m <- mirai(stop())
call_mirai(m)
is_mirai_error(m$data)
is_mirai_interrupt(m$data)
is_error_value(m$data)

m2 <- mirai(Sys.sleep(1L), .timeout = 100)
call_mirai(m2)
is_mirai_error(m2$data)
is_mirai_interrupt(m2$data)
is_error_value(m2$data)
```

```
}
```

`launch_local` *Launch Daemon*

Description

`launch_local` spawns a new background `Rscript` process calling `daemon` with the specified arguments.

`launch_remote` returns the shell command for deploying daemons as a character vector. If a configuration generated by `remote_config` or `ssh_config` is supplied then this is used to launch the daemon on the remote machine.

Usage

```
launch_local(url, ..., tls = NULL, .compute = "default")

launch_remote(
  url,
  remote = remote_config(),
  ...,
  tls = NULL,
  .compute = "default"
)
```

Arguments

`url`	the character host URL or vector of host URLs, including the port to connect to (and optionally for websockets, a path), e.g. tcp://hostname:5555' or 'ws://10.75.32.70:5555/path' **or** integer index value, or vector of index values, of the dispatcher URLs, or 1L for the host URL (when not using dispatcher). **or** for `launch_remote` only, a 'miraiCluster' or 'miraiNode'.
`...`	(optional) additional arguments passed through to `daemon`. These include 'autoexit', 'cleanup', 'output', 'maxtasks', 'idletime', 'walltime' and 'timerstart'.
`tls`	[default NULL] required for secure TLS connections over tls+tcp or wss. Zero-configuration TLS certificates generated by `daemons` are automatically passed to the daemon, without requiring to be specified here. Otherwise, supply **either** the character path to a file containing X.509 certificate(s) in PEM format, comprising the certificate authority certificate chain, **or** a length 2 character vector comprising [i] the certificate authority certificate chain and [ii] the empty character ''.
`.compute`	[default 'default'] character value for the compute profile to use (each compute profile has its own independent set of daemons).

`remote` required only for launching remote daemons, a configuration generated by `remote_config` or `ssh_config`. An empty `remote_config` does not effect any daemon launches but returns the shell commands for deploying manually on remote machines.

Details

These functions may be used to re-launch daemons that have exited after reaching time or task limits.

If daemons have been set, the generated command will automatically contain the argument 'rs' specifying the length 7 L'Ecuyer-CMRG random seed supplied to the daemon. The values will be different each time the function is called.

Value

For **launch_local**: Invisible NULL.

For **launch_remote**: A character vector of daemon launch commands, classed as 'miraiLaunchCmd'. The printed output may be directly copy / pasted to the remote machine.

Examples

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

daemons(url = host_url(ws = TRUE), dispatcher = FALSE)
status()
launch_local(status()$daemons, maxtasks = 10L)
launch_remote(1L, maxtasks = 10L)
Sys.sleep(1)
status()
daemons(0)

daemons(n = 2L, url = host_url(tls = TRUE))
status()
launch_local(1:2, idletime = 60000L, timerstart = 1L)
launch_remote(1:2, idletime = 60000L, timerstart = 1L)
Sys.sleep(1)
status()
daemons(0)

}
```

---

make_cluster *Make Mirai Cluster*

Description

`make_cluster` creates a cluster of type 'miraiCluster', which may be used as a cluster object for any function in the **parallel** base package such as `clusterApply` or `parLapply`.

`stop_cluster` stops a cluster created by `make_cluster`.

Usage

```
make_cluster(n, url = NULL, remote = NULL, ...)

stop_cluster(cl)
```

Arguments

n	integer number of nodes (automatically launched on the local machine unless 'url' is supplied).
url	[default NULL] (specify for remote nodes) the character URL on the host for remote nodes to dial into, including a port accepting incoming connections, e.g. 'tcp://10.75.37.40:5555'. Specify a URL with the scheme 'tls+tcp://' to use secure TLS connections.
remote	[default NULL] (specify to launch remote nodes) a remote launch configuration generated by remote_config or ssh_config. If not supplied, nodes may be deployed manually on remote resources.
...	additional arguments passed onto daemons.
cl	a 'miraiCluster'.

Value

For **make_cluster**: An object of class 'miraiCluster' and 'cluster'. Each 'miraiCluster' has an automatically assigned ID and 'n' nodes of class 'miraiNode'. If 'url' is supplied but not 'remote', the shell commands for deployment of nodes on remote resources are printed to the console.

For **stop_cluster**: invisible NULL.

Remote Nodes

Specify 'url' and 'n' to set up a host connection for remote nodes to dial into. 'n' defaults to one if not specified.

Also specify 'remote' to launch the nodes using a configuration generated by remote_config or ssh_config. In this case, the number of nodes is inferred from the configuration provided and 'n' is disregarded.

If 'remote' is not supplied, the shell commands for deploying nodes manually on remote resources are automatically printed to the console.

launch_remote may be called at any time on a 'miraiCluster' to return the shell commands for deployment of all nodes, or on a 'miraiNode' to return the command for a single node.

Status

Call status on a 'miraiCluster' to check the number of currently active connections as well as the host URL.

Errors

Errors are thrown by the 'parallel' mechanism if one or more nodes failed (quit unexpectedly). The resulting 'errorValue' returned is 19 (Connection reset). Other types of error, e.g. in evaluation, should result in the usual 'miraiError' being returned.

Note

The default behaviour of clusters created by this function is designed to map as closely as possible to clusters created by the **parallel** package. However, '...' arguments are passed onto daemons for additional customisation if desired, although resultant behaviour may not be supported.

Examples

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

cl <- make_cluster(2)
cl
cl[[1L]]

Sys.sleep(0.5)
status(cl)

stop_cluster(cl)

}
```

`mirai` *mirai (Evaluate Async)*

Description

Evaluate an expression asynchronously in a new background R process or persistent daemon (local or remote). This function will return immediately with a 'mirai', which will resolve to the evaluated result once complete.

Usage

```
mirai(.expr, ..., .args = list(), .timeout = NULL, .compute = "default")
```

Arguments

`.expr`	an expression to evaluate asynchronously (of arbitrary length, wrapped in { } where necessary), **or** a language object passed by name.
`...`	(optional) named arguments (name = value pairs) specifying objects referenced in '.expr'. Used in addition to, and taking precedence over, any arguments specified via '.args'.
`.args`	(optional) **either** a list of objects passed by name (found in the current scope), **or else** a list of name = value pairs, as in '...'.
`.timeout`	[default NULL] for no timeout, or an integer value in milliseconds. A mirai will resolve to an 'errorValue' 5 (timed out) if evaluation exceeds this limit.
`.compute`	[default 'default'] character value for the compute profile to use (each compute profile has its own independent set of daemons).

Details

This function will return a 'mirai' object immediately.

The value of a mirai may be accessed at any time at `$data`, and if yet to resolve, an 'unresolved' logical NA will be returned instead.

`unresolved` may be used on a mirai, returning TRUE if a 'mirai' has yet to resolve and FALSE otherwise. This is suitable for use in control flow statements such as `while` or `if`.

Alternatively, to call (and wait for) the result, use `call_mirai` on the returned mirai. This will block until the result is returned.

The expression '.expr' will be evaluated in a separate R process in a clean environment, which is not the global environment, consisting only of the named objects passed as '...' and/or the list supplied to '.args'.

If an error occurs in evaluation, the error message is returned as a character string of class 'miraiError' and 'errorValue'. `is_mirai_error` may be used to test for this.

`is_error_value` tests for all error conditions including 'mirai' errors, interrupts, and timeouts.

Specify '.compute' to send the mirai using a specific compute profile (if previously created by `daemons`), otherwise leave as 'default'.

Value

A 'mirai' object.

Examples

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

# specifying objects via '...'
n <- 3
m <- mirai(x + y + 2, x = 2, y = n)
m
m$data
Sys.sleep(0.2)
m$data

# passing existing objects by name via '.args'
df1 <- data.frame(a = 1, b = 2)
df2 <- data.frame(a = 3, b = 1)
m <- mirai(as.matrix(rbind(df1, df2)), .args = list(df1, df2), .timeout = 1000)
call_mirai(m)$data

# using unresolved()
m <- mirai(
  {
    res <- rnorm(n)
    res / rev(res)
  },
  n = 1e6
)
```

```
while (unresolved(m)) {
  cat("unresolved\n")
  Sys.sleep(0.1)
}
str(m$data)

# evaluating scripts using source(local = TRUE) in '.expr'
n <- 10L
file <- tempfile()
cat("r <- rnorm(n)", file = file)
m <- mirai({source(file, local = TRUE); r}, .args = list(file, n))
call_mirai(m)[["data"]]
unlink(file)

# specifying global variables using list2env(envir = .GlobalEnv) in '.expr'
n <- 10L
file <- tempfile()
cat("r <- rnorm(n)", file = file)
globals <- list(file = file, n = n)
m <- mirai(
  {
    list2env(globals, envir = .GlobalEnv)
    source(file)
    r
  },
  globals = globals
)
call_mirai(m)[["data"]]
unlink(file)

# passing a language object to '.expr' and a named list to '.args'
expr <- quote(a + b + 2)
args <- list(a = 2, b = 3)
m <- mirai(.expr = expr, .args = args)
call_mirai(m)$data

}
```

`nextstream` *Next » Developer Interface*

Description

`nextstream` retrieves the currently stored L'Ecuyer-CMRG RNG stream for the specified compute profile and advances it to the next stream.

`nextget` retrieves the specified item from the specified compute profile.

Usage

```
nextstream(.compute = "default")

nextget(x, .compute = "default")
```

Arguments

.compute	[default 'default'] character value for the compute profile to use (each compute profile has its own independent set of daemons).
x	character value of item to retrieve. One of 'pid' (dispatcher process ID), 'urls' (URLs dispatcher is listening at) or 'tls' (the stored client TLS configuration for use by daemons).

Details

These functions are exported for use by packages extending **mirai** with alternative launchers of daemon processes.

For nextstream: This function should be called for its return value when required. The function also has the side effect of automatically advancing the stream stored within the compute profile. This ensures that the next recursive stream is returned when the function is called again.

Value

For nextstream: a length 7 integer vector, as given by .Random.seed when the L'Ecuyer-CMRG RNG is in use (may be passed directly to the 'rs' argument of daemon), or else NULL if a stream has not yet been created.

For nextget: the requested item, or else NULL if not present.

Examples

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

daemons(1L)
nextstream()
nextstream()

nextget("pid")
nextget("urls")

daemons(0)

}
```

remote_config *Generic and SSH Remote Launch Configuration*

Description

remote_config provides a flexible generic framework for generating the shell commands to deploy daemons remotely.

ssh_config generates a remote configuration for launching daemons over SSH, with the option of SSH tunnelling.

Usage

```
remote_config(command = NULL, args = c("", "."), rscript = "Rscript")

ssh_config(
  remotes,
  timeout = 10,
  tunnel = FALSE,
  command = "ssh",
  rscript = "Rscript"
)
```

Arguments

command	the command used to effect the daemon launch on the remote machine as a character string (e.g. 'ssh'). Defaults to 'ssh' for ssh_config, although may be substituted for the full path to a specific SSH application. The default NULL for remote_config does not effect any launches, but causes launch_remote to return the shell commands for manual deployment on remote machines.
args	(optional) arguments passed to 'command', as a character vector that must include "." as an element, which will be substituted for the daemon launch command. Alternatively, a list of such character vectors to effect multiple launches (one for each list element).
rscript	(optional) name / path of the Rscript executable on the remote machine. The default assumes 'Rscript' is on the executable search path. Prepend the full path if necessary. If launching on Windows, 'Rscript' should be replaced with 'Rscript.exe'.
remotes	the character URL or vector of URLs to SSH into, using the 'ssh://' scheme and including the port open for SSH connections (defaults to 22 if not specified), e.g. 'ssh://10.75.32.90:22' or 'ssh://nodename'.
timeout	[default 10] maximum time allowed for connection setup in seconds.
tunnel	[default FALSE] logical value whether to use SSH reverse tunnelling. If TRUE, a tunnel is created between the same ports (as specified in 'url') on the local and remote machines. Setting to TRUE requires access to 'url' in the evaluation context and will error if not called from a relevant function.

Value

A list in the required format to be supplied to the 'remote' argument of launch_remote, daemons, or make_cluster.

SSH Direct Connections

The simplest use of SSH is to execute the daemon launch command on a remote machine, for it to dial back to the host / dispatcher URL.

It is assumed that SSH key-based authentication is already in place. The relevant port on the host must also be open to inbound connections from the remote machine.

SSH Tunnelling

Use of SSH tunnelling provides a convenient way to launch remote nodes without requiring the remote machine to be able to access the host. Often firewall configurations or security policies may prevent opening a port to accept outside connections.

In these cases SSH tunnelling offers a solution by creating a tunnel once the initial SSH connection is made. For simplicity, this SSH tunnelling implementation uses the same port on both the side of the host and that of the corresponding node. SSH key-based authentication must also already be in place.

Tunnelling requires the hostname for 'url' specified when setting up daemons to be either '127.0.0.1' or 'localhost'. This is as the tunnel is created between `127.0.0.1:port` or equivalently `localhost:port` on each machine. The host listens to `port` on its machine and the remotes each dial into `port` on their own respective machines.

Examples

```
remote_config(command = "ssh", args = c("-fTp 22 10.75.32.90", "."))

ssh_config(remotes = c("ssh://10.75.32.90:222", "ssh://nodename"), timeout = 5)

## Not run:

# launch 2 daemons on the remote machines 10.75.32.90 and 10.75.32.91 using
# SSH, connecting back directly to the host URL over a TLS connection:

daemons(
  url = host_url(tls = TRUE),
  remote = ssh_config(
    remotes = c("ssh://10.75.32.90:222", "ssh://10.75.32.91:222"),
    timeout = 1
  )
)

# launch 2 nodes on the remote machine 10.75.32.90 using SSH tunnelling over
# port 5555 ('url' hostname must be 'localhost' or '127.0.0.1'):

cl <- make_cluster(
  url = "tcp://localhost:5555",
  remote = ssh_config(
```

```
    remotes = c("ssh://10.75.32.90", "ssh://10.75.32.90"),
    timeout = 1,
    tunnel = TRUE
  )
)

## End(Not run)
```

---

saisei *Saisei (Regenerate Token)*

Description

When using daemons with dispatcher, regenerates the token for the URL a dispatcher socket listens at.

Usage

```
saisei(i, force = FALSE, .compute = "default")
```

Arguments

i	integer index number URL to regenerate at dispatcher.
force	[default FALSE] logical value whether to regenerate the URL even when there is an existing active connection.
.compute	[default 'default'] character value for the compute profile to use (each compute profile has its own independent set of daemons).

Details

When a URL is regenerated, the listener at the specified socket is closed and replaced immediately, hence this function will only be successful if there are no existing connections at the socket (i.e. 'online' status shows 0), unless the argument 'force' is specified as TRUE.

If 'force' is specified as TRUE, the socket is immediately closed and regenerated. If this happens while a mirai is still ongoing, it will be returned as an errorValue 7 'Object closed'. This may be used to cancel a task that consistently hangs or crashes to prevent it from failing repeatedly when new daemons connect.

Value

The regenerated character URL upon success, or else NULL.

Timeouts

Specifying the '.timeout' argument to `mirai` ensures that the 'mirai' always resolves. However, the task may not have completed and still be ongoing in the daemon process. In such situations, dispatcher ensures that queued tasks are not assigned to the busy process, however overall performance may still be degraded if they remain in use.

If a process hangs and cannot be restarted otherwise, `saisei` specifying `force = TRUE` may be used to cancel the task and regenerate any particular URL for a new `daemon` to connect to.

Examples

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

daemons(1L)
Sys.sleep(1L)
status()
saisei(i = 1L, force = TRUE)
status()

daemons(0)

}
```

`serialization` *Custom Serialization Functions*

Description

Registers custom serialization and unserialization functions for sending and receiving external pointer reference objects.

Usage

```
serialization(refhook = list())
```

Arguments

`refhook`	**either** a list or pairlist of two functions: the signature for the first must accept a list of external pointer type 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 type objects, e.g. `torch::torch_load`, **or else** NULL to reset.

Details

Calling without any arguments returns the pairlist of currently-registered 'refhook' functions.

This function may be called prior to or after setting daemons, with the registered functions applying across all compute profiles.

Value

Invisibly, the pairlist of currently-registered 'refhook' functions. A message is printed to the console when functions are successfully registered or reset.

Examples

```
r <- serialization(list(function(x) serialize(x, NULL), unserialize))
print(serialization())
serialization(r)

serialization(NULL)
print(serialization())
```

---

status *Status Information*

Description

Retrieve status information for the specified compute profile, comprising current connections and daemons status.

Usage

```
status(.compute = "default")
```

Arguments

.compute	[default 'default'] character compute profile (each compute profile has its own set of daemons for connecting to different resources). **or** a 'miraiCluster' to obtain its status.

Value

A named list comprising:

- **connections** - integer number of active connections.
  Using dispatcher: Always 1L as there is a single connection to dispatcher, which connects to the daemons in turn.
- **daemons** - of variable type.
  Using dispatcher: a status matrix (see Status Matrix section below), or else an integer 'errorValue' if communication with dispatcher failed.
  Not using dispatcher: the character host URL.
  Not set: 0L.

Status Matrix

When using dispatcher, `$daemons` comprises an integer matrix with the following columns:

- **i** - integer index number.
- **online** - shows as 1 when there is an active connection, or else 0 if a daemon has yet to connect or has disconnected.
- **instance** - increments by 1 every time there is a new connection at a URL. This counter is designed to track new daemon instances connecting after previous ones have ended (due to time-outs etc.). The count becomes negative immediately after a URL is regenerated by `saisei`, but increments again once a new daemon connects.
- **assigned** - shows the cumulative number of tasks assigned to the daemon.
- **complete** - shows the cumulative number of tasks completed by the daemon.

The dispatcher URLs are stored as row names to the matrix.

Examples

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

status()
daemons(n = 2L, url = "wss://[::1]:0")
status()
daemons(0)

}
```

`stop_mirai` *mirai (Stop Evaluation)*

Description

Stop evaluation of a mirai that is in progress.

Usage

```
stop_mirai(aio)
```

Arguments

`aio`	a 'mirai' object.

Details

Stops the asynchronous operation associated with the mirai by aborting, and then waits for it to complete or to be completely aborted. The mirai is then deallocated and attempting to access the value at `$data` will result in an error.

Value

Invisible NULL.

Examples

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

s <- mirai(Sys.sleep(n), n = 5)
stop_mirai(s)

}
```

`unresolved` *Query if a mirai is Unresolved*

Description

Query whether a mirai or mirai value remains unresolved. Unlike `call_mirai`, this function does not wait for completion.

Usage

```
unresolved(aio)
```

Arguments

`aio`	a 'mirai' object or 'mirai' value stored at `$data`.

Details

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

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

Value

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

Examples

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

m <- mirai(Sys.sleep(0.1))
unresolved(m)
Sys.sleep(0.3)
unresolved(m)
```

```
}
```

Index