Package: mirai (via r-universe)

November 2, 2026

Type Package

Title Minimalist Async Evaluation Framework for R

Version 2.7.4.9000

Description Evaluates R expressions asynchronously and in parallel, locally or distributed across networks. An official parallel cluster type for R. Built on 'nanonext' and 'NNG', its non-polling, event-driven architecture scales from a laptop to thousands of processes across high-performance computing clusters and cloud platforms. Features FIFO scheduling with task cancellation and bounded queues, promises for reactive programming, 'OpenTelemetry' distributed tracing, and custom serialization for cross-language data types.

License MIT + file LICENSE

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

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

Depends R (>= 3.6)

Imports nanonext (>= 1.10.4)

Suggests cli, litedown, mori, otel, otelsdk, secretbase

Enhances parallel, promises

VignetteBuilder litedown

Config/Needs/coverage rlang

Config/Needs/website tidyverse/tidytemplate

Config/roxygen2/markdown TRUE

Config/roxygen2/version 8.1.1

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

Encoding UTF-8

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

Date/Publication 2026-11-02 11:21:50 UTC

RemoteUrl https://github.com/r-lib/mirai

RemoteRef HEAD

RemoteSha 59f21d9c10f83b49656967b891aabcffe7efbe3a

Contents

mirai-package . . . 2
as.promise.mirai . . . 4
as.promise.mirai_map . . . 5
call_mirai . . . 6
cluster_config . . . 7
collect_mirai . . . 9
daemon . . . 10
daemons . . . 12
daemons_set . . . 16
everywhere . . . 17
host_url . . . 19
http_config . . . 20
info . . . 22
is_mirai . . . 23
is_mirai_error . . . 24
launch_local . . . 25
make_cluster . . . 26
mirai . . . 28
mirai_map . . . 31
on_daemon . . . 34
race_mirai . . . 35
register_serial . . . 36
remote_config . . . 37
require_daemons . . . 38
serial_config . . . 39
ssh_config . . . 40
status . . . 42
stop_mirai . . . 43
unresolved . . . 44
with.miraiDaemons . . . 44
with_daemons . . . 45

Index **47**

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

Description

Evaluates R expressions asynchronously and in parallel, locally or distributed across networks. An official parallel cluster type for R. Built on 'nanonext' and 'NNG', its non-polling, event-driven architecture scales from a laptop to thousands of processes across high-performance computing clusters and cloud platforms. Features FIFO scheduling with task cancellation and bounded queues, promises for reactive programming, 'OpenTelemetry' distributed tracing, and custom serialization for cross-language data types.

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 overridden by specifying 'url' in `daemons()` and launching daemons using `launch_local()`.

OpenTelemetry

mirai provides comprehensive OpenTelemetry tracing support for observing asynchronous operations and distributed computation. Please refer to the OpenTelemetry vignette for further details: `vignette("v05-opentelemetry", package = "mirai")`

Reference Manual

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

Author(s)

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

Authors:

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

Other contributors:

- Joe Cheng <joe@posit.co> [contributor]
- Posit Software, PBC (ROR) [copyright holder, funder]
- Hibiki AI Limited [copyright holder]

See Also

Useful links:

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

`as.promise.mirai` *Make mirai Promise*

Description

Creates a 'promise' from a 'mirai'. S3 method for `promises::as.promise()`.

Usage

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

Arguments

`x`	(mirai) object to convert to promise.

Details

Allows a 'mirai' to be used with the promise pipe `%...>%`, scheduling a function to run upon resolution.

Requires the **promises** package.

Value

A 'promise' object.

Examples

```
library(promises)

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

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

`as.promise.mirai_map` *Make mirai_map Promise*

Description

Creates a 'promise' from a 'mirai_map'. S3 method for `promises::as.promise()`.

Usage

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

Arguments

`x`	(mirai_map) object to convert to promise.

Details

Allows a 'mirai_map' to be used with the promise pipe `%...>%`, scheduling a function to run upon resolution of all mirai.

Uses `promises::promise_all()` internally: resolves to a list of values if all succeed, or rejects with the first error encountered.

Requires the **promises** package.

Value

A 'promise' object.

Examples

```
library(promises)

with(daemons(1), {
  mp <- mirai_map(1:3, function(x) { Sys.sleep(1); x })
  p <- as.promise(mp)
  print(p)
  p %...>% print
  mp[.flat]
})
```

`call_mirai` *mirai (Call Value)*

Description

Waits for the 'mirai' to resolve if still in progress (blocking but user-interruptible), stores the value at `$data`, and returns the 'mirai' object.

Usage

```
call_mirai(x)
```

Arguments

`x`	(mirai | list) a 'mirai' object or list of 'mirai' objects.

Details

Accepts a list of 'mirai' objects, such as those returned by `mirai_map()`, as well as individual 'mirai'.

`x[]` may also be used to wait for and return the value of a mirai `x`, and is the equivalent of `call_mirai(x)$data`.

Value

The passed object (invisibly). For a 'mirai', 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 it has yet to resolve and FALSE otherwise. This is suitable for use in control flow statements such as `while` or `if`.

Errors

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. The elements of the original condition are accessible via `$` on the error object. A stack trace comprising a list of calls is also available at `$stack.trace`, and the original condition classes at `$condition.class`.

If a daemon crashes or terminates unexpectedly during evaluation, an 'errorValue' 19 (Connection reset) is returned.

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

See Also

collect_mirai() to return the value directly rather than the 'mirai' object. race_mirai() to wait for the first of many.

Examples

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

---

cluster_config *Cluster Remote Launch Configuration*

Description

Generates a remote configuration for launching daemons using a high- performance computing (HPC) cluster resource manager such as Slurm sbatch, SGE and Torque/PBS qsub or LSF bsub.

Usage

```
cluster_config(command = "sbatch", options = "", rscript = "Rscript")
```

Arguments

command	(character) cluster manager executable: "sbatch" (Slurm), "qsub" (SGE/Torque/PBS), or "bsub" (LSF).
options	(character) script options for command (e.g. "#SBATCH –mem=16G"), newline-separated. May include shell commands such as "module load R/4.5.0". Shebang line such as "#!/bin/bash" not required.
rscript	(character) Rscript executable. Use full path if needed, or "Rscript.exe" on Windows.

Value

A list in the required format to be supplied to the `remote` argument of `daemons()` or `launch_remote()`.

See Also

`ssh_config()`, `http_config()` and `remote_config()` for other types of remote configuration.

Examples

```
# Slurm Config:
cluster_config(
  command = "sbatch",
  options = "#SBATCH --job-name=mirai
             #SBATCH --mem=16G
             #SBATCH --output=job.out
             module load R/4.5.0",
  rscript = file.path(R.home("bin"), "Rscript")
)

# SGE Config:
cluster_config(
  command = "qsub",
  options = "#$ -N mirai
             #$ -l mem_free=16G
             #$ -o job.out
             module load R/4.5.0",
  rscript = file.path(R.home("bin"), "Rscript")
)

# Torque/PBS Config:
cluster_config(
  command = "qsub",
  options = "#PBS -N mirai
             #PBS -l mem=16gb
             #PBS -o job.out
             module load R/4.5.0",
  rscript = file.path(R.home("bin"), "Rscript")
)

# LSF Config:
cluster_config(
  command = "bsub",
  options = "#BSUB -J mirai
             #BSUB -M 16000
             #BSUB -o job.out
             module load R/4.5.0",
  rscript = file.path(R.home("bin"), "Rscript")
)

## Not run:

# Launch 2 daemons using the Slurm sbatch defaults:
```

```
daemons(n = 2, url = host_url(), remote = cluster_config())

## End(Not run)
```

`collect_mirai` *mirai (Collect Value)*

Description

Waits for the 'mirai' to resolve if still in progress (blocking but interruptible) and returns its value directly. Equivalent to `call_mirai(x)$data`.

Usage

```
collect_mirai(x, options = NULL)
```

Arguments

`x`	(mirai | list) a 'mirai' object or list of 'mirai' objects.
`options`	(character) collection options for list input, e.g. `".flat"` or `c(".progress", ".stop")`. See Options section.

Details

`x[]` is an equivalent way to wait for and return the value of a mirai `x`.

Value

An object (the return value of the 'mirai'), or a list of such objects (the same length as `x`, preserving names).

Options

A named list may also be supplied instead of a character vector, where the names are the collection options. The value for `.progress` is passed to the cli progress bar: a character value sets the bar name, and a list is passed as named parameters to `cli::cli_progress_bar`. List elements not matching the arguments of `cli::cli_progress_bar` are ignored, as are `total`, `current`, `auto_terminate` and `.envir`, which are set by mirai. Examples: `c(.stop = TRUE, .progress = "bar name")` or `list(.stop = TRUE, .progress = list(name = "bar", type = "tasks"))`

Supplying an integer for `.progress` instead attaches to an existing cli progress bar: its name supplies the bar id, and its value the amount to increment per result collected, e.g. `list(.progress = setNames(1L, id))` where `id` is the value returned by `cli::cli_progress_bar()`. The bar is neither created nor closed by mirai: its lifecycle remains entirely with the caller. This allows a single progress bar to span multiple map operations.

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 it has yet to resolve and FALSE otherwise. This is suitable for use in control flow statements such as `while` or `if`.

Errors

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. The elements of the original condition are accessible via `$` on the error object. A stack trace comprising a list of calls is also available at `$stack.trace`, and the original condition classes at `$condition.class`.

If a daemon crashes or terminates unexpectedly during evaluation, an 'errorValue' 19 (Connection reset) is returned.

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

See Also

`call_mirai()` to return the 'mirai' object (with the value at `$data`) rather than the value directly.

Examples

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

# using x[]
m[]

# mirai_map with collection options
daemons(1, dispatcher = FALSE)
m <- mirai_map(1:3, rnorm)
collect_mirai(m, c(".flat", ".progress"))
daemons(0)
```

---

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,
  dispatcher = TRUE,
  ...,
  asyncdial = FALSE,
  autoexit = TRUE,
  cleanup = TRUE,
  output = FALSE,
  idletime = Inf,
  walltime = Inf,
  maxtasks = Inf,
  tlscert = NULL,
  rs = NULL
)
```

Arguments

`url`	(character) host or dispatcher URL to dial into, e.g. 'tcp://hostname:5555' or 'tls+tcp://10.75.32.70:5555'.
`dispatcher`	(logical) whether dialing into dispatcher or directly to host.
`...`	reserved for future use.
`asyncdial`	(logical) whether to dial asynchronously. `FALSE` errors if connection fails immediately. `TRUE` retries indefinitely (more resilient but can mask connection issues).
`autoexit`	(logical) whether to exit when the socket connection ends. `TRUE` exits immediately, `NA` completes current task first, `FALSE` persists indefinitely. See Persistence section.
`cleanup`	(logical) whether to restore global environment, packages, and options to initial state after each evaluation.
`output`	(logical) whether to output stdout/stderr. For local daemons via `daemons()` or `launch_local()`, redirects output to host process.
`idletime`	(integer) milliseconds to wait idle before exiting.
`walltime`	(integer) milliseconds of real time before exiting (soft limit).
`maxtasks`	(integer) maximum tasks to execute before exiting.
`tlscert`	(character) for secure TLS connections. Either a file path to PEM-encoded certificate authority certificate chain (starting with the TLS certificate and ending with the CA certificate), or a length-2 vector of (certificate chain, empty string `""`).
`rs`	(integer vector) initial `.Random.seed` value. Set automatically by host process; do not supply manually.

Details

Daemons dial into the host or dispatcher, which listens at `url`. This allows network resources to be added or removed dynamically, with the host or dispatcher automatically distributing tasks to all connected daemons.

Value

Invisibly, an integer exit code: 0L for normal termination, and a positive value if a self-imposed limit was reached: 1L (idletime), 2L (walltime), 3L (maxtasks).

Persistence

The `autoexit` argument governs persistence settings for the daemon. The default `TRUE` ensures that it exits as soon as its socket connection with the host process drops. A 200ms grace period allows the daemon process to exit normally, after which it will be forcefully terminated.

Supplying `NA` ensures that a daemon always exits cleanly after its socket connection with the host drops. This means that it can temporarily outlive this connection, but only to complete any task that is currently in progress. This can be useful if the daemon is performing a side effect such as writing files to disk, with the result not being required back in the host process.

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 dialed in. Daemons must be terminated with `daemons(NULL)` in this case instead of `daemons(0)`. This sends explicit exit signals to all connected daemons.

`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` to receive connections from remote daemons (for distributed computing across the network). Specify `remote` to optionally launch remote daemons via a remote configuration. Dispatcher (enabled by default) ensures optimal scheduling.

Usage

```
daemons(
  n,
  url = NULL,
  remote = NULL,
  dispatcher = TRUE,
  ...,
  sync = FALSE,
  seed = NULL,
  memory = NULL,
  serial = NULL,
  tls = NULL,
  pass = NULL,
  .compute = NULL
)
```

Arguments

n	(integer) number of daemons to launch.
url	(character) URL at which to listen for daemon connections, e.g. 'tcp://hostname:5555'. Use scheme 'tls+tcp://' for secure TLS connections (see Distributed Computing section). host_url() may be used to construct a valid URL.
remote	(configuration) for launching remote daemons, generated by ssh_config(), cluster_config(), or remote_config().
dispatcher	(logical) whether to use dispatcher for optimal first-in-first-out (FIFO) scheduling. See Dispatcher section below.
...	(daemon arguments) passed to daemon() when launching daemons. Includes asyncdial, autoexit, cleanup, output, maxtasks, idletime, walltime, and tlscert.
sync	(logical) whether to evaluate mirai synchronously in the current process for testing and debugging. When TRUE, other arguments except seed and .compute are disregarded.
seed	(integer) for reproducible random number generation (RNG). NULL (default) initializes L'Ecuyer-CMRG RNG streams per daemon (statistically sound, non-reproducible). An integer value instead initializes a stream per mirai, allowing reproducible results independent of which daemon evaluates it.
memory	(numeric) memory capacity in MB (metric) for queued task payloads at dispatcher. New tasks block until queued bytes drop below this threshold, providing memory-based backpressure to prevent the host process from running out of memory. NULL (default) is unbounded. Requires dispatcher.
serial	(configuration) for custom serialization of reference objects (e.g. Arrow Tables, torch tensors), created by serial_config(). Requires dispatcher. NULL applies any configurations from register_serial().
tls	(character) for secure TLS connections. Either a file path to PEM-encoded TLS certificate (possibly followed by other certificates in a validation chain) and private key, or a length-2 vector of (certificate, private key). NULL auto-generates single-use credentials.
pass	(function) returning the password for an encrypted tls private key. Use a function rather than a direct value for security.
.compute	(character) name of the compute profile. Each profile has its own independent set of daemons. NULL (default) uses the 'default' profile.

Details

Use daemons(0) to reset daemon connections:

- All connected daemons and/or dispatchers exit automatically.
- Any as yet unresolved 'mirai' will return an 'errorValue' 19 (Connection reset).
- mirai() reverts to the default behaviour of creating a new background process for each request.

If the host session ends, all connected dispatcher and daemon processes automatically exit as soon as their connections are dropped.

Calling daemons() implicitly resets any existing daemons for the compute profile with daemons(0). Instead, launch_local() or launch_remote() may be used to add daemons at any time without resetting daemons.

Value

Invisibly, logical TRUE when creating daemons and FALSE when resetting.

Local Daemons

Setting daemons, or persistent background processes, is typically more efficient as it removes the need for, and overhead of, creating new processes for each mirai evaluation. It also provides control over the total number of processes at any one time.

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

Dispatcher

By default dispatcher = TRUE enables optimal FIFO scheduling, queuing tasks and sending to daemons as they become available. The memory argument caps the approximate total memory (MB, metric — 1 MB = 1,000,000 bytes) of queued task payloads at dispatcher. New tasks block until existing ones are dispatched, providing memory-based backpressure to prevent the host process from running out of memory. Current usage is surfaced under the memory field of status(). Dispatcher also enables (i) mirai cancellation using stop_mirai() or a .timeout argument to mirai(), and (ii) custom serialization configurations.

With dispatcher = FALSE, daemons connect directly to the host and tasks are distributed round-robin, with tasks queued at each daemon. Optimal scheduling is not guaranteed, as tasks can queue at one daemon while others remain idle. However, this is the most lightweight option, suited to similar-length tasks or when concurrent tasks do not exceed available daemons.

Distributed Computing

Specify url as a character string to allow tasks to be distributed across the network (n is only required in this case if also providing a launch configuration to remote).

The host / dispatcher listens at this URL, utilising a single port, and daemon() processes dial in to this URL. Host / dispatcher automatically adjusts to the number of daemons actually connected, allowing dynamic upscaling / downscaling.

The URL should have a 'tcp://' scheme, such as 'tcp://10.75.32.70:5555'. Switching the URL scheme to 'tls+tcp://' automatically upgrades the connection to use TLS. The auxiliary function host_url() may be used to construct a valid host URL based on the computer's IP address.

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' will automatically assign a free ephemeral port. Use `status()` to inspect the actual assigned port at any time.

Specify `remote` with a call to `ssh_config()`, `cluster_config()` or `remote_config()` to launch (programatically deploy) daemons on remote machines, from where they dial back to `url`. If not launching daemons, `launch_remote()` may be used to generate the shell commands for manual deployment.

Compute Profiles

If `NULL`, 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 operates independently. Some usage examples follow:

local / remote daemons may be set by specifying a host URL and `.compute` as `"remote"`, creating 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 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.

See Also

`with_daemons()` and `local_daemons()` for managing the compute profile used locally.

Examples

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

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

# Set up dispatcher accepting TLS over TCP connections
daemons(url = host_url(tls = TRUE))
info()
# Reset to zero
daemons(0)

# Set host URL for remote daemons to dial into
daemons(url = host_url(), dispatcher = FALSE)
```

```
info()
# Reset to zero
daemons(0)

# Use with() to evaluate with daemons for the duration of the expression
with(
  daemons(2),
  {
    m1 <- mirai(Sys.getpid())
    m2 <- mirai(Sys.getpid())
    cat(m1[], m2[], "\n")
  }
)

## Not run:

# Launch 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'))
)

# Launch 4 daemons on the remote machine 10.75.32.90 using SSH tunnelling:
daemons(
  n = 4,
  url = local_url(tcp = TRUE),
  remote = ssh_config('ssh://10.75.32.90', tunnel = TRUE)
)


## End(Not run)

# Synchronous mode
# mirai are run in the current process - useful for testing and debugging
daemons(sync = TRUE)
m <- mirai(Sys.getpid())
daemons(0)
m[]

# Synchronous mode restricted to a specific compute profile
daemons(sync = TRUE, .compute = "sync")
with_daemons("sync", {
  m <- mirai(Sys.getpid())
})
daemons(0, .compute = "sync")
m[]
```

---

daemons_set *Query if Daemons are Set*

Description

Returns a logical value, whether or not daemons have been set for a given compute profile.

Usage

```
daemons_set(.compute = NULL)
```

Arguments

.compute	(character) name of the compute profile. Each profile has its own independent set of daemons. NULL (default) uses the 'default' profile.

Value

Logical TRUE or FALSE.

Examples

```
daemons_set()
daemons(sync = TRUE)
daemons_set()
daemons(0)
```

---

everywhere *Evaluate Everywhere*

Description

Evaluate an expression 'everywhere' on all connected daemons for the specified compute profile. Daemons must be set prior to calling this function. Performs operations across daemons such as loading packages or exporting common data. Resultant changes to the global environment, loaded packages and options are persisted regardless of a daemon's cleanup setting.

Usage

```
everywhere(.expr, ..., .args = list(), .min = 1L, .compute = NULL)
```

Arguments

.expr	(expression) code to evaluate asynchronously, or a language object. Wrap multi-line expressions in {}.
...	(named arguments | environment) objects required by .expr, assigned to the daemon's global environment. See 'evaluation' section below.
.args	(named list | environment) objects required by .expr, kept local to the evaluation environment (unlike ...). See 'evaluation' section below.

.min	(integer) minimum daemons to evaluate on (dispatcher only). Creates a synchronization point, useful for remote daemons that take time to connect.
.compute	(character) name of the compute profile. Each profile has its own independent set of daemons. NULL (default) uses the 'default' profile.

Details

If using dispatcher, this function forces a synchronization point: the everywhere() call must complete on all daemons before subsequent mirai evaluations proceed.

Calling everywhere() does not affect the random number generator (RNG) stream for mirai calls when using a reproducible seed value at daemons(). This allows the seed associated with each mirai call to be the same, regardless of the number of daemons used. However, code evaluated in an everywhere() call is itself non-reproducible if it involves random numbers.

Value

A 'mirai_map' (list of 'mirai' objects).

Evaluation

The expression .expr will be evaluated in a separate R process in a clean environment (not the global environment), consisting only of the objects supplied to .args, with the objects passed as ... assigned to the global environment of that process.

As evaluation occurs in a clean environment, all undefined objects must be supplied through ... and/or .args, including self-defined functions. Functions from a package should use namespaced calls such as mirai::mirai(), or else the package should be loaded beforehand as part of .expr.

Supply objects to ... rather than .args for evaluation to occur *as if* in your global environment. This is needed for non-local variables or helper functions required by other functions, which scoping rules may otherwise prevent from being found.

Examples

```
daemons(sync = TRUE)

# export common data by a super-assignment expression:
everywhere(y <<- 3)
mirai(y)[]

# '...' variables are assigned to the global environment
# '.expr' may be specified as an empty {} in such cases:
everywhere({}, a = 1, b = 2)
mirai(a + b - y == 0L)[]

# everywhere() returns a mirai_map object:
mp <- everywhere("just a normal operation")
mp
mp[.flat]
mp <- everywhere(stop("everywhere"))
collect_mirai(mp)
daemons(0)
```

```
# loading a package on all daemons
daemons(sync = TRUE)
everywhere(library(parallel))
m <- mirai("package:parallel" %in% search())
m[]
daemons(0)
```

---

host_url *URL Constructors*

Description

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

local_url() constructs a URL suitable for local daemons, or for use with SSH tunnelling. This may be supplied directly to the url argument of daemons().

Usage

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

local_url(tcp = FALSE, port = 0)
```

Arguments

tls	(logical) whether to use TLS (scheme 'tls+tcp://').
port	(integer) port number. 0 assigns a free ephemeral port. For host_url(), must be open to daemon connections. For local_url(), only used when tcp = TRUE.
tcp	(logical) whether to use TCP. Required for SSH tunnelling.

Details

host_url() will return a vector of URLs if multiple network adapters are in use, and each will be named by the interface name (adapter friendly name on Windows). If this entire vector is passed to the url argument of functions such as daemons(), the first URL is used. If no suitable IP addresses are detected, the computer's hostname will be used as a fallback.

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 vector (comprising a valid URL or URLs), named for host_url().

Examples

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

local_url()
local_url(tcp = TRUE)
local_url(tcp = TRUE, port = 5555)
```

`http_config` *HTTP Remote Launch Configuration*

Description

Generates a remote configuration for launching daemons via HTTP API, as may be used by Kubernetes or other such platforms. By default, automatically configures for Posit Workbench using environment variables.

Usage

```
http_config(
  url = posit_workbench_url,
  method = "POST",
  headers = posit_workbench_headers,
  data = posit_workbench_data,
  ...,
  cookie = NULL,
  token = NULL
)
```

Arguments

`url`	(character or function) URL endpoint for the launch API. May be a function returning the URL value.
`method`	(character) HTTP method, typically `"POST"`.
`headers`	(named character vector or function) HTTP headers sent with the request, supplying any required authentication (e.g. session cookie, bearer token, API key) as well as other API metadata. May be a function returning a named character vector.
`data`	(character or function) JSON or formatted request body containing the daemon launch command. May be a function returning the data value. Should include a placeholder `"%s"` where the `mirai::daemon()` call will be inserted at launch time.
`...`	additional arguments passed to `data` when it is a function. See the Posit Workbench Options section for those accepted by the default value of `data`.

`cookie` (character or function) convenience argument that, if non-NULL, appends a `Cookie: <value>` entry to `headers`. May be a function returning the cookie value.

`token` (character or function) convenience argument that, if non-NULL, appends an `Authorization: Bearer <value>` entry to `headers`. May be a function returning the token value.

Details

Arguments accepting either a value or a function (`url`, `headers`, `data`, `cookie`, `token`) may be supplied as a function to defer evaluation: a plain value is captured when the configuration is created, whereas a function is evaluated at the time each daemon is launched. This is the recommended way to supply credentials such as session cookies or API tokens, as the same configuration object may be stored and reused (for example to scale up later in a session), with a fresh credential fetched at each launch.

At launch time, the `"%s"` placeholder in `data` is replaced by a `mirai::daemon()` call, e.g. `mirai::daemon("tcp://10.0.0.`
(when using TLS, the certificate is also inlined in the call). The receiving platform only has to run this expression using `Rscript -e` to start a daemon, which then dials back to the host.

Value

A list in the required format to be supplied to the `remote` argument of `daemons()` or `launch_remote()`.

Posit Workbench Options

The default values of `url`, `headers` and `data` configure the launch automatically on Posit Workbench. The following arguments may additionally be supplied via `...` to customise the launched job:

- `rscript` (character) Rscript executable path. Default `"Rscript"`.
- `job_name` (character) base name for launched jobs. Default `"mirai_daemon"`.
- `cluster` (character) name of the cluster to use. Default uses the first available cluster.
- `resource_profile` (character) named resource profile (e.g. `"rstudio"`). Default uses the first profile available on the chosen cluster.
- `cpus` (integer) number of CPUs for custom resource allocation. Specify together with or instead of `memory` to override `resource_profile`.
- `memory` (integer) memory in MB for custom resource allocation. Specify together with or instead of `cpus` to override `resource_profile`.

See Also

`ssh_config()`, `cluster_config()` and `remote_config()` for other types of remote configuration.

Examples

```
tryCatch(http_config(), error = identity)

# Custom HTTP configuration example:
http_config(
  url = "https://api.example.com/launch",
  method = "POST",
  headers = function() c(
    Authorization = sprintf("Bearer %s", Sys.getenv("MY_API_KEY")),
    `X-API-Version` = "2"
  ),
  data = '{"command": "%s"}'
)

## Not run:
# Launch 2 daemons using http config default (for Posit Workbench):
daemons(n = 2, url = host_url(), remote = http_config())

# Customise the default Posit Workbench launch (named cluster and profile):
daemons(
  n = 2,
  url = host_url(),
  remote = http_config(cluster = "Kubernetes", resource_profile = "rstudio")
)

# Or specify custom resources (4 CPUs, 8 GB memory):
daemons(
  n = 2,
  url = host_url(),
  remote = http_config(cluster = "Kubernetes", cpus = 4, memory = 8192)
)

## End(Not run)
```

`info` *Information Statistics*

Description

Retrieve statistics for the specified compute profile.

Usage

```
info(.compute = NULL)
```

Arguments

`.compute`	(character) name of the compute profile. Each profile has its own independent set of daemons. `NULL` (default) uses the 'default' profile.

Details

The returned statistics are:

- Connections: active daemon connections.
- Cumulative: total daemons that have ever connected.
- Awaiting: mirai tasks currently queued for execution at dispatcher.
- Executing: mirai tasks currently being evaluated on a daemon.
- Completed: mirai tasks that have been completed or cancelled.

For non-dispatcher daemons: only 'connections' will be available and the other values will be NA.

Value

Named integer vector or else NULL if the compute profile is yet to be set up.

Examples

```
info()
daemons(sync = TRUE)
info()
daemons(0)
```

---

is_mirai *Is mirai / mirai_map*

Description

Is the object a 'mirai' or 'mirai_map'.

Usage

```
is_mirai(x)

is_mirai_map(x)
```

Arguments

x	(object) to test.

Value

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

Examples

```
daemons(1, dispatcher = FALSE)
df <- data.frame()
m <- mirai(as.matrix(df), df = df)
is_mirai(m)
is_mirai(df)

mp <- mirai_map(1:3, runif)
is_mirai_map(mp)
is_mirai_map(mp[])
daemons(0)
```

---

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	(object) to test.

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'. The elements of the original condition are accessible via $ on the error object. A stack trace is also available at `$stack.trace`.

Is the object a 'miraiInterrupt'. When an ongoing 'mirai' is sent a user interrupt, it 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

```
m <- mirai(stop())
call_mirai(m)
is_mirai_error(m$data)
is_mirai_interrupt(m$data)
is_error_value(m$data)
m$data$stack.trace

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 Daemons*

Description

`launch_local()` launches daemons on the local machine as background R processes that connect back to the host.

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

Usage

```
launch_local(n = 1L, ..., .compute = NULL)

launch_remote(n = 1L, remote = remote_config(), ..., .compute = NULL)
```

Arguments

`n`	(integer) number of daemons to launch. For `launch_remote()`, may also be a 'miraiCluster' or 'miraiNode'.
`...`	(daemon arguments) passed to `daemon()`, including `asyncdial`, `autoexit`, `cleanup`, `output`, `maxtasks`, `idletime`, and `walltime`. Overrides arguments from `daemons()` if supplied.
`.compute`	(character) name of the compute profile. Each profile has its own independent set of daemons. `NULL` (default) uses the 'default' profile.
`remote`	(configuration) for launching daemons, generated by `ssh_config()`, `cluster_config()`, `http_config()`, or `remote_config()`. An empty `remote_config()` returns shell commands for manual deployment without launching.

Details

Daemons must already be set for launchers to work.

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

For non-dispatcher daemons using the default seed strategy, the generated command contains 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**: Integer number of daemons launched.

For **launch_remote**: A character vector of daemon launch commands, classed as 'miraiLaunchCmd'. The printed output may be copy / pasted directly to the remote machine. For the `http_config()` launcher, a list of server response data, returned invisibly.

Examples

```
daemons(url = host_url(), dispatcher = FALSE)
info()
launch_local(1L, cleanup = FALSE)
launch_remote(1L, cleanup = FALSE)
Sys.sleep(1)
info()
daemons(0)

daemons(url = host_url(tls = TRUE))
info()
launch_local(2L, output = TRUE)
Sys.sleep(1)
info()
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 `parallel::clusterApply()` or `parallel::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. Launched locally unless url is supplied.
url	(character) host URL for remote nodes to dial into, e.g. 'tcp://10.75.37.40:5555'. Use 'tls+tcp://' for secure TLS.
remote	(configuration) for launching remote nodes, generated by ssh_config(), cluster_config(), or remote_config().
...	(daemons arguments) passed to daemons().
cl	(miraiCluster) cluster to stop.

Details

For R version 4.5 or newer, parallel::makeCluster() specifying type = "MIRAI" is equivalent to this function.

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.

Errors

Errors are thrown by the **parallel** package mechanism if one or more nodes failed (quit unexpectedly). The returned 'errorValue' is 19 (Connection reset). Other types of error, e.g. in evaluation, 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 to daemons() for additional customisation, and not all combinations may be supported by **parallel** functions.

Examples

```
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 = NULL)

try_mirai(.expr, ..., .args = list(), .timeout = NULL, .compute = NULL)
```

Arguments

`.expr`	(expression) code to evaluate asynchronously, or a language object. Wrap multi-line expressions in `{}`.
`...`	(named arguments | environment) objects required by `.expr`, assigned to the daemon's global environment. See 'evaluation' section below.
`.args`	(named list | environment) objects required by .expr, kept local to the evaluation environment (unlike `...`). See 'evaluation' section below.
`.timeout`	(integer) timeout in milliseconds. The mirai resolves to an 'errorValue' 5 (timed out) if evaluation exceeds this limit. `NULL` (default) for no timeout.
`.compute`	(character) name of the compute profile. Each profile has its own independent set of daemons. `NULL` (default) uses the 'default' profile.

Details

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. Each mirai has an attribute `id`, which is a monotonically increasing integer identifier in each session.

`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.

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

Value

For `mirai()`: a 'mirai' object.

For `try_mirai()`: a 'mirai' object, or `NULL` (invisibly) if the dispatcher's `memory` capacity is exhausted at the time of submission.

Evaluation

The expression `.expr` will be evaluated in a separate R process in a clean environment (not the global environment), consisting only of the objects supplied to `.args`, with the objects passed as `...` assigned to the global environment of that process.

As evaluation occurs in a clean environment, all undefined objects must be supplied through `...` and/or `.args`, including self-defined functions. Functions from a package should use namespaced calls such as `mirai::mirai()`, or else the package should be loaded beforehand as part of `.expr`.

Supply objects to `...` rather than `.args` for evaluation to occur *as if* in your global environment. This is needed for non-local variables or helper functions required by other functions, which scoping rules may otherwise prevent from being found.

Timeouts

Specifying the `.timeout` argument ensures that the mirai always resolves. When using dispatcher, the mirai will be cancelled after it times out (as if `stop_mirai()` had been called). However, cancellation is not guaranteed – for example, compiled code may not be interruptible. When not using dispatcher, the mirai task continues to completion in the daemon process, even if it times out in the host process.

Errors

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. The elements of the original condition are accessible via `$` on the error object. A stack trace comprising a list of calls is also available at `$stack.trace`, and the original condition classes at `$condition.class`.

If a daemon crashes or terminates unexpectedly during evaluation, an 'errorValue' 19 (Connection reset) is returned.

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

Memory

The `memory` argument to `daemons()` caps the queued task payload at dispatcher (in MB), preventing host out-of-memory. `mirai()` blocks the calling R thread on submission until queued bytes drop below this capacity.

`try_mirai()` is a non-blocking variant for event-loop contexts (Shiny, promises) where the host R thread cannot afford to wait. It returns `NULL` (invisibly) immediately if the queue is at the memory

limit, instead of blocking. With no dispatcher, or `memory` unset, `try_mirai()` always returns a 'mirai'. Respond to a `NULL` return value by dropping the task, retrying later, or propagating back-pressure upstream.

Use `status()` to inspect current and peak queue usage under its `memory` field.

Examples

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

# passing the calling environment to '...'
df1 <- data.frame(a = 1, b = 2)
df2 <- data.frame(a = 3, b = 1)
df_matrix <- function(x, y) {
  mirai(as.matrix(rbind(x, y)), environment(), .timeout = 1000)
}
m <- df_matrix(df1, df2)
m[]

# 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() in '.expr'
n <- 10L
file <- tempfile()
cat("r <- rnorm(n)", file = file)
m <- mirai({source(file); r}, file = file, n = n)
call_mirai(m)$data
unlink(file)

# use source(local = TRUE) when passing in local variables via '.args'
n <- 10L
file <- tempfile()
cat("r <- rnorm(n)", file = file)
m <- mirai({source(file, local = TRUE); r}, .args = list(file = file, n = n))
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)
collect_mirai(m)


# non-blocking submission - caller handles backpressure
daemons(1, memory = 1)
m <- try_mirai(1 + 1)
if (is.null(m)) {
  # queue at memory limit - drop, retry, signal upstream, etc.
} else {
  m[]
}
daemons(0)
```

`mirai_map` *mirai Map*

Description

Asynchronous parallel map of a function over a list or vector using **mirai**, with optional **promises** integration. For matrix or dataframe inputs, maps over rows.

Usage

```
mirai_map(.x, .f, ..., .args = list(), .promise = NULL, .compute = NULL)
```

Arguments

`.x`	(list | vector | matrix | data.frame) input to map over. For matrix or dataframe, maps over rows (see Multiple Map section).
`.f`	(function) applied to each element of `.x`, or each row of a matrix / dataframe.
`...`	(named arguments) objects referenced but not defined in `.f`.
`.args`	(list) constant arguments passed to `.f`.
`.promise`	(function | list) registers a promise against each mirai. Either an `onFulfilled` function, or a list of (`onFulfilled`, `onRejected`) functions for `promises::then()`. Requires the **promises** package.
`.compute`	(character) name of the compute profile. Each profile has its own independent set of daemons. `NULL` (default) uses the 'default' profile.

Details

Sends each application of function `.f` on an element of `.x` (or row of `.x`) for computation in a separate `mirai()` call. If `.x` is named, names are preserved.

Takes advantage of **mirai** scheduling to minimise overall execution time.

Facilitates recovery from partial failure by returning all 'miraiError' / 'errorValue' as the case may be, thus allowing only failures to be re-run.

This function requires daemons to have previously been set, and will error otherwise.

Value

A 'mirai_map' (list of 'mirai' objects).

Collection Options

`x[]` collects the results of a 'mirai_map' `x` and returns a list. This will wait for all asynchronous operations to complete if still in progress, blocking but user-interruptible.

`x[.flat]` collects and flattens map results to a vector, checking that they are of the same type to avoid coercion. Note: errors if an 'errorValue' has been returned or results are of differing type.

`x[.progress]` collects map results whilst showing a progress bar from the **cli** package, if installed, with completion percentage and ETA, or else a simple text progress indicator. Note: if the map operation completes too quickly then the progress bar may not show at all.

To update an existing **cli** progress bar instead of creating one, e.g. to span a single bar over multiple map operations, supply the bar id via the `options` argument of `collect_mirai()`.

`x[.stop]` collects map results applying early stopping, which stops at the first failure and cancels remaining operations.

The options above may be combined in the manner of:
`x[.stop, .progress]` which applies early stopping together with a progress indicator.

Multiple Map

If `.x` is a matrix or dataframe (or other object with 'dim' attributes), *multiple* map is performed over its **rows**. Character row names are preserved as names of the output.

This allows map over 2 or more arguments, and `.f` should accept at least as many arguments as there are columns. If the dataframe has column names, or the matrix has column dimnames, arguments are passed to `.f` by name.

To map over **columns** instead, first wrap a dataframe in `as.list()`, or transpose a matrix using `t()`.

Nested Maps

To run maps within maps, the function provided to the outer map must include a call to `daemons()` to set daemons for the inner map. To guard against inadvertently spawning an excessive number of daemons on the same machine, attempting to launch local daemons within a map using `daemons(n)` will error.

When the outer daemons run on remote machines and you want local daemons on each, use 2 separate calls instead of `daemons(n)`: `daemons(url = local_url()); launch_local(n)`. This is equivalent, and is permitted from within a map.

Examples

```
daemons(4)

# perform and collect mirai map
mm <- mirai_map(c(a = 1, b = 2, c = 3), rnorm)
mm
mm[]

# map with constant args specified via '.args'
mirai_map(1:3, rnorm, .args = list(n = 5, sd = 2))[]

# flatmap with helper function passed via '...'
mirai_map(
  10^(0:9),
  function(x) rnorm(1L, valid(x)),
  valid = function(x) min(max(x, 0L), 100L)
)[.flat]

# unnamed matrix multiple map: arguments passed to function by position
(mat <- matrix(1:4, nrow = 2L))
mirai_map(mat, function(x = 10, y = 0, z = 0) x + y + z)[.flat]

# named matrix multiple map: arguments passed to function by name
(mat <- matrix(1:4, nrow = 2L, dimnames = list(c("a", "b"), c("y", "z"))))
mirai_map(mat, function(x = 10, y = 0, z = 0) x + y + z)[.flat]

# dataframe multiple map: using a function taking '...' arguments
df <- data.frame(a = c("Aa", "Bb"), b = c(1L, 4L))
mirai_map(df, function(...) sprintf("%s: %d", ...))[.flat]

# indexed map over a vector (using a dataframe)
v <- c("egg", "got", "ten", "nap", "pie")
mirai_map(
  data.frame(1:length(v), v),
  sprintf,
  .args = list(fmt = "%d_%s")
)[.flat]

# return a 'mirai_map' object, check for resolution, collect later
mp <- mirai_map(2:4, function(x) runif(1L, x, x + 1))
unresolved(mp)
mp
mp[.flat]
unresolved(mp)

# progress indicator counts up from 0 to 4 seconds
res <- mirai_map(1:4, Sys.sleep)[.progress]
```

```
# stops early when second element returns an error
tryCatch(mirai_map(list(1, "a", 3), sum)[.stop], error = identity)

daemons(0)


# promises example that outputs the results, including errors, to the console
daemons(1, dispatcher = FALSE)
ml <- mirai_map(
  1:30,
  function(i) {Sys.sleep(0.1); if (i == 30) stop(i) else i},
  .promise = list(
    function(x) cat(paste(x, "")),
    function(x) { cat(conditionMessage(x), "\n"); daemons(0) }
  )
)
```

---

on_daemon *On Daemon*

Description

Returns a logical value, whether or not evaluation is taking place within a mirai call on a daemon.

Usage

```
on_daemon()
```

Value

Logical TRUE or FALSE.

Examples

```
on_daemon()
mirai(mirai::on_daemon())[]
```

`race_mirai` *mirai (Race)*

Description

Accepts a list of 'mirai' objects, such as those returned by `mirai_map()`. Returns the index of the first resolved 'mirai'. If any mirai is already resolved, returns immediately. Otherwise waits for at least one to resolve, blocking but user-interruptible.

Usage

```
race_mirai(x, .compute = NULL)
```

Arguments

`x`	(list) of 'mirai' objects.
`.compute`	(character) name of the compute profile. Each profile has its own independent set of daemons. `NULL` (default) uses the 'default' profile.

Details

All of the 'mirai' objects supplied must belong to the same compute profile.

When called on a list where some mirais are already resolved, returns the index of the first resolved mirai immediately without waiting. When all mirais are unresolved, blocks until at least one resolves. If multiple mirais resolve during the same wait iteration, returns the index of the first resolved in list order.

This enables an efficient "process as completed" pattern:

```
remaining <- list(m1, m2, m3)
while (length(remaining) > 0) {
  idx <- race_mirai(remaining)
  process(remaining[[idx]]$data)
  remaining <- remaining[-idx]
}
```

Value

Integer index of the first resolved 'mirai' (invisibly), or `0L` if the list is empty.

See Also

`call_mirai()`

Examples

```
daemons(2)
m1 <- mirai({ Sys.sleep(0.2); "one" })
m2 <- mirai({ Sys.sleep(0.1); "two" })
m3 <- mirai({ Sys.sleep(0.3); "three" })
remaining <- list(m1, m2, m3)
while (length(remaining) > 0) {
  idx <- race_mirai(remaining)
  print(remaining[[idx]]$data)
  remaining <- remaining[-idx]
}
daemons(0)
```

`register_serial` *Register Serialization Configuration*

Description

Registers a serialization configuration, which may be set to perform custom serialization and unserialization of normally non-exportable reference objects, allowing these to be used seamlessly between different R sessions. Once registered, the functions apply to all `daemons()` calls where the `serial` argument is `NULL`.

Usage

```
register_serial(class, sfunc, ufunc)
```

Arguments

`class`	(character) class name(s) for custom serialization, e.g. `'ArrowTabular'` or `c('torch_tensor', 'ArrowTabular')`.
`sfunc`	(function | list) serialization function(s) accepting a reference object and returning a raw vector.
`ufunc`	(function | list) unserialization function(s) accepting a raw vector and returning a reference object.

Value

Invisible NULL.

remote_config *Generic Remote Launch Configuration*

Description

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

Usage

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

Arguments

command	(character) shell command for launching daemons (e.g. "ssh"). NULL returns shell commands for manual deployment without launching.
args	(character vector) arguments to command, must include "." as placeholder for the daemon launch command. May be a list of vectors for multiple launches.
rscript	(character) Rscript executable. Use full path if needed, or "Rscript.exe" on Windows.
quote	(logical) whether to quote the daemon launch command. Required for "sbatch" and "ssh", not for "srun".

Value

A list in the required format to be supplied to the remote argument of daemons() or launch_remote().

See Also

ssh_config(), cluster_config() and http_config() for other types of remote configuration.

Examples

```
# Slurm srun example
remote_config(
  command = "srun",
  args = c("--mem 512", "-n 1", "."),
  rscript = file.path(R.home("bin"), "Rscript")
)

# SSH requires 'quote = TRUE'
remote_config(
```

```
  command = "/usr/bin/ssh",
  args = c("-fTp 22 10.75.32.90", "."),
  quote = TRUE
)

# can be used to start local daemons with special configurations
remote_config(
  command = "Rscript",
  rscript = "--default-packages=NULL --vanilla"
)
```

`require_daemons` *Require Daemons*

Description

Returns `TRUE` invisibly only if daemons are set, otherwise produces an informative error for the user to set daemons, with a clickable function link if the **cli** package is available.

Usage

```
require_daemons(.compute = NULL, call = environment())
```

Arguments

`.compute`	(character) name of the compute profile. Each profile has its own independent set of daemons. `NULL` (default) uses the 'default' profile.
`call`	(environment) execution environment for error attribution, e.g. `environment()`. Used by **cli** for error messages.

Value

Invisibly, logical `TRUE`, or else errors.

Examples

```
daemons(sync = TRUE)
(require_daemons())
daemons(0)
```

`serial_config` *Create Serialization Configuration*

Description

Returns a serialization configuration, which may be set to perform custom serialization and unserialization of normally non-exportable reference objects, allowing these to be used seamlessly between different R sessions. Once set by passing to the `serial` argument of `daemons()`, the functions apply to all mirai requests for that compute profile.

Usage

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

Arguments

`class`	(character) class name(s) for custom serialization, e.g. `'ArrowTabular'` or `c('torch_tensor', 'ArrowTabular')`.
`sfunc`	(function | list) serialization function(s) accepting a reference object and returning a raw vector.
`ufunc`	(function | list) unserialization function(s) accepting a raw vector and returning a reference object.

Details

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

An error in `sfunc` (serialization) is raised directly in the calling process, e.g. the `mirai()` call itself errors and the task is not dispatched.

An error in `ufunc` (unserialization) is caught, and the object instead unserializes as an 'errorValue' 1000 ('Internal error detected'). This prevents a faulty hook from crashing the daemon.

Value

A list comprising the configuration. This should be passed to the `serial` argument of `daemons()`.

Examples

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

cfg2 <- serial_config(
  c("class_one", "class_two"),
  list(function(x) serialize(x, NULL), function(x) serialize(x, NULL)),
  list(unserialize, unserialize)
)
cfg2
```

ssh_config *SSH Remote Launch Configuration*

Description

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

Usage

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

Arguments

remotes	(character) URL(s) to SSH into using scheme 'ssh://', e.g. 'ssh://10.75.32.90:22' or 'ssh://nodename'. Port defaults to 22. Optionally include a username if it differs on the remote system, e.g. 'ssh://user@10.75.32.90'.
tunnel	(logical) whether to use SSH tunnelling. Requires url hostname '127.0.0.1' (use local_url() with tcp = TRUE). See SSH Tunnelling section.
timeout	(integer) maximum seconds for connection setup.
command	(character) shell command for launching daemons (e.g. "ssh"). NULL returns shell commands for manual deployment without launching.
rscript	(character) Rscript executable. Use full path if needed, or "Rscript.exe" on Windows.

Value

A list in the required format to be supplied to the remote argument of daemons() or launch_remote().

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.

SSH key-based authentication must already be in place. The relevant port on the host must be open to inbound connections from the remote machine. This approach is suited to trusted networks.

SSH Tunnelling

SSH tunnelling launches remote daemons without requiring the remote machine to access the host directly. Often firewall configurations or security policies may prevent opening a port to accept outside connections.

A tunnel is created once the initial SSH connection is made. For simplicity, this SSH tunnelling implementation uses the same port on both host and daemon. SSH key-based authentication must already be in place, but no other configuration is required.

To use tunnelling, set the hostname of the `daemons()` `url` argument to be '127.0.0.1'. Using `local_url()` with `tcp = TRUE` also does this for you. Specifying a specific port to use is optional, with a random ephemeral port assigned otherwise. For example, specifying 'tcp://127.0.0.1:5555' uses the local port '5555' to create the tunnel on each machine. The host listens to '127.0.0.1:5555' on its machine and the remotes each dial into '127.0.0.1:5555' on their own respective machines.

Daemons can be launched on any machine accessible via SSH, whether on the local network or in the cloud.

See Also

`cluster_config()`, `http_config()` and `remote_config()` for other types of remote configuration.

Examples

```
# direct SSH example (with username)
ssh_config(c("ssh://user@10.75.32.90:222", "ssh://nodename"), timeout = 5)

# SSH tunnelling example
ssh_config(c("ssh://10.75.32.90:222", "ssh://nodename"), tunnel = TRUE)

## Not run:

# launch 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(
  n = 1,
  url = host_url(tls = TRUE),
  remote = ssh_config(c("ssh://10.75.32.90:222", "ssh://10.75.32.91:222"))
)

# launch 2 daemons on the remote machine 10.75.32.90 using SSH tunnelling:
daemons(
  n = 2,
  url = local_url(tcp = TRUE),
  remote = ssh_config("ssh://10.75.32.90", tunnel = TRUE)
)

## End(Not run)
```

`status` *Status Information*

Description

Retrieve status information for the specified compute profile, comprising current connections, daemons status, and (when using dispatcher) queue depth and memory pressure.

Usage

```
status(.compute = NULL)
```

Arguments

`.compute`	(character | miraiCluster) compute profile name, or `NULL` for 'default'. Also accepts a 'miraiCluster'.

Value

A named list comprising:

- **connections** - integer number of active daemon connections.
- **daemons** - character URL at which host / dispatcher is listening, or else `0L` if daemons have not yet been set.
- **mirai** (present only if using dispatcher) - a named integer vector comprising: **awaiting** - number of tasks queued for execution at dispatcher, **executing** - number of tasks sent to a daemon for execution, and **completed** - number of tasks for which the result has been received (either completed or cancelled).
- **memory** (present only if using dispatcher) - a named numeric vector in MB (metric, 1 MB = 1,000,000 bytes) comprising: **used** - current and **peak** - high-watermark queued task payloads at dispatcher, and **capacity** - the value set as the `memory` argument to `daemons()` (`NA_real_` if unset/unbounded).

See Also

`info()` for more succinct information statistics.

Examples

```
status()
daemons(sync = TRUE)
status()
daemons(0)
```

`stop_mirai` *mirai (Stop)*

Description

Stops a 'mirai' if still in progress, causing it to resolve immediately to an 'errorValue' 20 (Operation canceled).

Usage

```
stop_mirai(x)
```

Arguments

`x`	(mirai | list) a 'mirai' object or list of 'mirai' objects.

Details

Cancellation requires dispatcher. If the 'mirai' is awaiting execution, it is discarded from the queue and never evaluated. If already executing, an interrupt is sent.

A cancellation request does not guarantee the task stops: it may have already completed before the interrupt is received, and compiled code is not always interruptible. Take care if the code performs side effects such as writing to files.

Value

Logical TRUE if the cancellation request was successful (was awaiting execution or in execution), or else FALSE (if already completed or previously cancelled). Will always return FALSE if not using dispatcher.

Or a vector of logical values if supplying a list of 'mirai', such as those returned by `mirai_map()`.

Examples

```
m <- mirai(Sys.sleep(n), n = 5)
stop_mirai(m)
m$data
```

unresolved *Query if a mirai is Unresolved*

Description

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

Usage

```
unresolved(x)
```

Arguments

`x`	(mirai | list | mirai value) a 'mirai', list of 'mirai' objects, or value from `$data`.

Details

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

Value

Logical TRUE if `x` is an unresolved 'mirai' or 'mirai' value or the list contains at least one unresolved 'mirai', or FALSE otherwise.

Examples

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

with.miraiDaemons *With Mirai Daemons*

Description

Evaluate an expression with daemons that last for the duration of the expression. Ensure each mirai within the statement is explicitly called (or their values collected) so that daemons are not reset before they have all completed.

Usage

```
## S3 method for class 'miraiDaemons'
with(data, expr, ...)
```

Arguments

data	(miraiDaemons) return value from daemons().
expr	(expression) to evaluate with daemons active.
...	unused.

Details

This function is an S3 method for the generic with() for class 'miraiDaemons'.

Value

The return value of expr.

Examples

```
with(
  daemons(2, dispatcher = FALSE),
  {
    m1 <- mirai(Sys.getpid())
    m2 <- mirai(Sys.getpid())
    cat(m1[], m2[], "\n")
  }
)

Sys.getpid()
```

---

with_daemons *With Daemons*

Description

Evaluate an expression using a specific compute profile.

Usage

```
with_daemons(.compute, expr)

local_daemons(.compute, frame = parent.frame())
```

Arguments

.compute	(character) name of the compute profile. Each profile has its own independent set of daemons. NULL (default) uses the 'default' profile.
expr	(expression) to evaluate using the compute profile.
frame	(environment) scope for the compute profile setting.

Details

Will error if the specified compute profile is not yet set up.

Value

For **with_daemons**: the return value of `expr`.
For **local_daemons**: invisible NULL.

Examples

```
daemons(1, dispatcher = FALSE, .compute = "cpu")
daemons(1, dispatcher = FALSE, .compute = "gpu")

with_daemons("cpu", {
  m1 <- mirai(Sys.getpid())
})

with_daemons("gpu", {
  m2 <- mirai(Sys.getpid())
  m3 <- mirai(Sys.getpid(), .compute = "cpu")
  local_daemons("cpu")
  m4 <- mirai(Sys.getpid())
})

m1[]

m2[] # different to m1

m3[] # same as m1
m4[] # same as m1

with_daemons("cpu", daemons(0))
with_daemons("gpu", daemons(0))
```

Index

as.list(), *32*
as.promise.mirai, 4
as.promise.mirai_map, 5

call_mirai, 6
call_mirai(), *10*, *29*, *35*, *44*
cluster_config, 7
cluster_config(), *13*, *15*, *21*, *25*, *27*, *37*, *41*
collect_mirai, 9
collect_mirai(), *7*, *32*

daemon, 10
daemon(), *13, 14*, *25*
daemons, 12
daemons(), *3*, *8*, *10, 11*, *14*, *18, 19*, *21*, *25*, *27*, *29*, *32*, *36, 37*, *39–42*, *45*
daemons_set, 16

everywhere, 17
everywhere(), *18*

host_url, 19
host_url(), *13, 14*, *19*
http_config, 20
http_config(), *8*, *25, 26*, *37*, *41*

info, 22
info(), *42*
is_error_value *(*is_mirai_error*)*, 24
is_error_value(), *6*, *10*, *29*
is_mirai, 23
is_mirai_error, 24
is_mirai_error(), *6*, *10*, *29*
is_mirai_interrupt *(*is_mirai_error*)*, 24
is_mirai_map *(*is_mirai*)*, 23

launch_local, 25
launch_local(), *3*, *11*, *14*
launch_remote *(*launch_local*)*, 25
launch_remote(), *8*, *14, 15*, *21*, *27*, *37*, *40*
local_daemons *(*with_daemons*)*, 45
local_daemons(), *15*
local_url *(*host_url*)*, 19
local_url(), *19*, *40, 41*

make_cluster, 26
mirai, 28
mirai(), *10*, *12–15*, *29*, *32*, *39*
mirai-package, 2
mirai_map, 31
mirai_map(), *6*, *35*, *43*

on_daemon, 34

parallel::clusterApply(), *26*
parallel::makeCluster(), *27*
parallel::parLapply(), *26*
promises::then(), *31*

race_mirai, 35
race_mirai(), *7*
register_serial, 36
register_serial(), *13*
remote_config, 37
remote_config(), *8*, *13*, *15*, *21*, *25*, *27*, *41*
require_daemons, 38

serial_config, 39
serial_config(), *13*
ssh_config, 40
ssh_config(), *8*, *13*, *15*, *21*, *25*, *27*, *37*
status, 42
status(), *14, 15*, *30*
stop_cluster *(*make_cluster*)*, 26
stop_mirai, 43
stop_mirai(), *14*, *29*

t(), *32*
try_mirai *(*mirai*)*, 28
try_mirai(), *29, 30*

unresolved, 44

unresolved(), *6*, *10*, *28*

with(), *45*
with.miraiDaemons, 44
with_daemons, 45
with_daemons(), *15*