Package 'mirai'

July 15, 2025

Type Package

Title Minimalist Async Evaluation Framework for R

Version 2.4.1

Description Designed for simplicity, a 'mirai' evaluates an R expression asynchronously in a parallel process, locally or distributed over the network. Modern networking and concurrency, built on 'nanonext' and 'NNG', ensures reliable scheduling over fast inter-process communications or TCP/IP secured by TLS. Launch remote resources via SSH or cluster managers for distributed computing. Scales efficiently to millions of tasks over thousands of connections, requiring no storage on the file system due to its inherently queued architecture. Innovative features include event-driven promises, asynchronous parallel map, and seamless serialization of otherwise non-exportable reference objects.

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 (>= 1.6.2)

Suggests cli, litedown

Enhances parallel, promises

VignetteBuilder litedown

Config/Needs/website tidyverse/tidytemplate

Config/usethis/last-upkeep 2025-04-23

Encoding UTF-8

RoxygenNote 7.3.2

NeedsCompilation no

Author Charlie Gao [aut, cre] (ORCID: <https://orcid.org/0000-0002-0750-061X>),
Joe Cheng [ctb],
Posit Software, PBC [cph, fnd] (ROR: <https://ror.org/03wc8by49>),
Hibiki AI Limited [cph]

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

Repository CRAN

Date/Publication 2025-07-15 08:10:02 UTC

Contents

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

Description

Designed for simplicity, a 'mirai' evaluates an R expression asynchronously in a parallel process, locally or distributed over the network. Modern networking and concurrency, built on 'nanonext' and 'NNG', ensures reliable scheduling over fast inter-process communications or TCP/IP secured by TLS. Launch remote resources via SSH or cluster managers for distributed computing. Scales efficiently to millions of tasks over thousands of connections, requiring no storage on the file system due to its inherently queued architecture. Innovative features include event-driven promises, asynchronous parallel map, and seamless serialization of otherwise non-exportable reference objects.

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)

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

Other contributors:

- Joe Cheng <joe@posit.co> [contributor]
- Posit Software, PBC (03wc8by49) [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'.

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

```
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'.

Usage

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

Arguments

`x`	an object of class 'mirai_map'.

Details

This function is an S3 method for the generic `as.promise()` for class 'mirai_map'.

Requires the **promises** package.

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

The implementation internally uses `promises::promise_all()`. If all of the promises were successful, the returned promise will resolve to a list of the promise values; if any promise fails, the first error to be encountered will be used to reject the returned promise.

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, stores the value at `$data`, and returns the 'mirai' object.

Usage

```
call_mirai(x)
```

Arguments

`x`	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'.

Waits for the asynchronous operation(s) to complete if still in progress, blocking but user-interruptible.

`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`.

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.

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 an 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`	[default "sbatch"] for Slurm. Replace with "qsub" for SGE / Torque / PBS, or "bsub" for LSF. See examples below.
`options`	[default ""] options as would be supplied inside a script file passed to `command`, e.g. "#SBATCH –mem=10G", each separated by a new line. See examples below. Other shell commands e.g. to change working directory may also be included. For certain setups, "module load R" as a final line is required, or for example "module load R/4.5.0" for a specific R version. For the avoidance of doubt, the initial shebang line such as "#!/bin/bash" is not required.
`rscript`	[default "Rscript"] assumes the R executable is on the search path. Replace with the full path of the Rscript executable on the remote machine if necessary. If launching on Windows, `"Rscript"` should be replaced with `"Rscript.exe"`.

Value

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

See Also

`ssh_config()` for SSH launch configurations, or `remote_config()` for generic configurations.

Examples

```
# Slurm Config:
cluster_config(
  command = "sbatch",
  options = "#SBATCH --job-name=mirai
             #SBATCH --mem=10G
             #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=10G
             #$ -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=10gb
             #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 10000
             #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, and returns its value directly. It is a more efficient version of and equivalent to `call_mirai(x)$data`.

Usage

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

Arguments

`x`	a 'mirai' object, or list of 'mirai' objects.
`options`	(if `x` is a list of mirai) a character vector comprising any combination of collection options for `mirai_map()`, such as `".flat"` or `c(".progress", ".stop")`.

Details

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

`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).

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`.

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.

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,
  id = NULL,
  tls = NULL,
  rs = NULL
)
```

Arguments

url	the character host or dispatcher URL to dial into, including the port to connect to, e.g. 'tcp://hostname:5555' or 'tls+tcp://10.75.32.70:5555'.
dispatcher	[default TRUE] logical value, which should be set to TRUE if using dispatcher and FALSE otherwise.
...	reserved but not currently used.
asyncdial	[default FALSE] whether to perform dials asynchronously. The default FALSE will error if a connection is not immediately possible (for instance if 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.
autoexit	[default TRUE] logical value, whether the daemon should exit automatically when its socket connection ends. By default, the process ends immediately when the host process ends. Supply NA to have a daemon complete any tasks in progress before exiting (see 'Persistence' section below).
cleanup	[default TRUE] logical value, whether to perform cleanup of the global environment and restore attached packages and options to an initial state after each evaluation.
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).
idletime	[default Inf] integer milliseconds maximum time to wait for a task (idle time) before exiting.
walltime	[default Inf] integer milliseconds soft walltime (time limit) i.e. the minimum amount of real time elapsed before exiting.
maxtasks	[default Inf] integer maximum number of tasks to execute (task limit) before exiting.
id	[default NULL] (optional) integer daemon ID provided to dispatcher to track connection status. Causes status() to report this ID under $events when the daemon connects and disconnects.
tls	[default NULL] required for secure TLS connections over 'tls+tcp://'. **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 string "".

`rs` [default NULL] the initial value of .Random.seed. This is set automatically using L'Ecuyer-CMRG RNG streams generated by the host process if applicable, 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

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 will exit as soon as its socket connection with the host process drops.

Supplying `NA` will allow a daemon to exit cleanly once its socket connection with the host process drops, as soon as it has finished any task that is currently in progress. This may be useful if the daemon is performing some side effect such as writing files to disk, and the result is not required 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 dialled in. Daemons must be terminated with `daemons(NULL)` in this case, which 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,
  ...,
  seed = NULL,
```

```
  serial = NULL,
  tls = NULL,
  pass = NULL,
  .compute = NULL
)
```

Arguments

n	integer number of daemons to launch.
url	[default NULL] if specified, a character string comprising a URL at which to listen for remote daemons, including a port accepting incoming connections, e.g. 'tcp://hostname:5555' or 'tcp://10.75.32.70:5555'. Specify a URL with scheme 'tls+tcp://' to use secure TLS connections (for details see Distributed Computing section below). 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 runs in a separate process to ensure optimal scheduling, and should normally be kept on (for details see Dispatcher section below).
...	(optional) additional arguments passed through to `daemon()` if launching daemons. These include `asyncdial`, `autoexit`, `cleanup`, `output`, `maxtasks`, `idletime` and `walltime`.
seed	[default NULL] (optional) The default of `NULL` initializes L'Ecuyer-CMRG RNG streams for each daemon, the same as base R's parallel package. Results are statistically-sound, although generally non-reproducible, as which tasks are sent to which daemons may be non-deterministic, and also depends on the number of daemons. (experimental) supply an integer value to instead initialize a L'Ecuyer-CMRG RNG stream for the compute profile. This is advanced for each mirai evaluation, hence allowing for reproducible results, as the random seed is always associated with a given mirai, independently of where it is evaluated.
serial	[default NULL] (optional, requires dispatcher) a configuration created by `serial_config()` to register serialization and unserialization functions for normally non-exportable reference objects, such as Arrow Tables or torch tensors. If `NULL`, configurations registered with `register_serial()` are automatically applied.
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 NULL] character value for the compute profile to use (each has its own independent set of daemons), or NULL to use the 'default' profile.

Details

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

- 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' 19 (Connection reset) after a reset.
- Daemons must be reset before calling `daemons()` with revised settings for a compute profile. Daemons may be added at any time by using `launch_local()` or `launch_remote()` without needing to revise daemons settings.

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`).

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

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

Value

The integer number of daemons launched locally (zero if specifying `url` or using a remote launcher).

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` launches a background process running `dispatcher()`. Dispatcher connects to daemons on behalf of the host, queues tasks, and ensures optimal FIFO scheduling. Dispatcher also enables (i) mirai cancellation using `stop_mirai()` or when using a `.timeout` argument to `mirai()`, and (ii) the use of custom serialization configurations.

Specifying `dispatcher = FALSE`, daemons connect directly to the host and tasks are distributed in a round-robin fashion, with tasks queued at each daemon. Optimal scheduling is not guaranteed as, depending on the duration of tasks, they can be queued at one daemon while others remain idle. However, this solution is the most resource-light, and suited to similar-length tasks, or where concurrent tasks typically 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 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 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

```
# 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)

# Set up dispatcher accepting TLS over TCP connections
daemons(url = host_url(tls = 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)

# 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)
```

---

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	[default NULL] character value for the compute profile to query, or NULL to query the 'default' profile. **or** a 'miraiCluster' to obtain its status.

Value

Logical TRUE or FALSE.

Examples

```
daemons_set()
daemons(1)
daemons_set()
daemons(0)
```

---

dispatcher *Dispatcher*

Description

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

Usage

```
dispatcher(host, url = NULL, n = NULL, ...)
```

Arguments

host	the character URL dispatcher should dial in to, typically an IPC address.
url	(optional) the character URL dispatcher should listen at (and daemons should dial in to), including the port to connect to e.g. 'tcp://hostname:5555' or 'tcp://10.75.32.70:5555'. Specify 'tls+tcp://' to use secure TLS connections.
n	(optional) if specified, the integer number of daemons to launch. In this case, a local url is automatically generated.
...	(optional) additional arguments passed through to daemon(). These include asyncdial, autoexit, and cleanup.

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 - this 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(), .compute = NULL)
```

Arguments

`.expr`	an expression to evaluate asynchronously (of arbitrary length, wrapped in { } where necessary), **or else** a pre-constructed language object.
`...`	(optional) **either** named arguments (name = value pairs) specifying objects referenced, but not defined, in `.expr`, **or** an environment containing such objects. See 'evaluation' section below.
`.args`	(optional) **either** a named list specifying objects referenced, but not defined, in `.expr`, **or** an environment containing such objects. These objects will remain local to the evaluation environment as opposed to those supplied in `...` above - see 'evaluation' section below.
`.compute`	[default NULL] character value for the compute profile to use (each has its own independent set of daemons), or NULL to use the 'default' profile.

Details

If using dispatcher, this function forces a synchronization point at dispatcher, whereby the `everywhere()` call must have been evaluated on all daemons prior to subsequent evaluations taking place. It is an error to call `everywhere()` successively without at least one `mirai()` call in between, as an ordinary mirai call is required to exit each synchronization point.

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`.

For evaluation to occur *as if* in your global environment, supply objects to `...` rather than `.args`, e.g. for non-local variables or helper functions required by other functions, as scoping rules may otherwise prevent them from being found.

Examples

```
daemons(1)
# export common data by a super-assignment expression:
everywhere(y <<- 3)
# '...' variables are assigned to the global environment
# '.expr' may be specified as an empty {} in such cases:
everywhere({}, a = 1, b = 2)
m <- mirai(a + b - y == 0L)
m[]
# everywhere() returns a list of mirai which may be waited for and inspected
mlist <- everywhere("just a normal operation")
collect_mirai(mlist)
mlist <- everywhere(stop("error"))
collect_mirai(mlist)
daemons(0)

# loading a package on all daemons
daemons(1, dispatcher = FALSE)
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	[default FALSE] logical value whether to use TLS in which case the scheme used will be 'tls+tcp://'.
port	[default 0] numeric port to use. 0 is a wildcard value that automatically assigns a free ephemeral port. For host_url, this port should be open to connections from the network addresses the daemons are connecting from. For local_url, is only taken into account if tcp = TRUE.
tcp	[default FALSE] logical value whether to use a TCP connection. This must be used 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)
```

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	an object.

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	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'. 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 Daemon*

Description

Launching a daemon is very much akin to launching a satellite. They are a way to deploy a daemon (in our case) on the desired machine. Once it executes, it connects back to the host process using its own communications.

`launch_local` deploys a daemon on the local machine in a new background `Rscript` process.

`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, ..., tls = NULL, .compute = NULL)

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

Arguments

n	integer number of daemons. **or** for launch_remote only, a 'miraiCluster' or 'miraiNode'.
...	(optional) arguments passed through to daemon(). These include autoexit, cleanup, output, maxtasks, idletime and walltime. Only supply to override arguments originally provided to daemons(), otherwise those will be used instead.
tls	[default NULL] required for secure TLS connections over 'tls+tcp://'. 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 string "".
.compute	[default NULL] character value for the compute profile to use (each has its own independent set of daemons), or NULL to use the 'default' profile.
remote	required only for launching remote daemons, a configuration generated by ssh_config(), cluster_config(), or remote_config(). An empty remote_config() does not perform any launches but returns the shell commands for deploying manually on remote machines.

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.

The generated command for non-dispatcher daemons 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. For dispatcher daemons, the equivalent random seed is obtained automatically from dispatcher, and hence rs is not specified in this case.

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.

Examples

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

daemons(url = host_url(tls = TRUE))
status()
launch_local(2L, output = TRUE)
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 `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 (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'.

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.

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** package 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, 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 always be supported.

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)
```

Arguments

`.expr`	an expression to evaluate asynchronously (of arbitrary length, wrapped in { } where necessary), **or else** a pre-constructed language object.
`...`	(optional) **either** named arguments (name = value pairs) specifying objects referenced, but not defined, in `.expr`, **or** an environment containing such objects. See 'evaluation' section below.
`.args`	(optional) **either** a named list specifying objects referenced, but not defined, in `.expr`, **or** an environment containing such objects. These objects will remain local to the evaluation environment as opposed to those supplied in `...` above - see 'evaluation' section below.
`.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 NULL] character value for the compute profile to use (each has its own independent set of daemons), or NULL to use the 'default' profile.

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.

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

Value

A 'mirai' object.

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`.

For evaluation to occur *as if* in your global environment, supply objects to `...` rather than `.args`, e.g. for non-local variables or helper functions required by other functions, as scoping rules may otherwise prevent them 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). As in that case, there is no guarantee that any cancellation will be successful, if the code cannot be interrupted for instance. When not using dispatcher, the mirai task will continue 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`.

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.

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)
```

`mirai_map` *mirai Map*

Description

Asynchronous parallel map of a function over a list or vector using **mirai**, with optional **promises** integration. Performs multiple map over the rows of a dataframe or matrix.

Usage

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

Arguments

.x	a list or atomic vector. Also accepts a matrix or dataframe, in which case multiple map is performed over its rows.
.f	a function to be applied to each element of .x, or row of .x as the case may be.
...	(optional) named arguments (name = value pairs) specifying objects referenced, but not defined, in .f.
.args	(optional) further constant arguments to .f, provided as a list.
.promise	(optional) if supplied, registers a promise against each mirai. Either a function, supplied to the onFulfilled argument of promises::then() or a list of 2 functions, supplied respectively to onFulfilled and onRejected of promises::then(). Using this argument requires the **promises** package.
.compute	[default NULL] character value for the compute profile to use (each has its own independent set of daemons), or NULL to use 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.

This simple and transparent behaviour is designed to make full use 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.

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 names, or the matrix column dimnames, named arguments are provided to `.f`.

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

Nested Maps

At times you way wish to run maps within maps. To do this, the function provided to the outer map needs to include a call to `daemons()` to set daemons used by 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.

A legitimate use of this pattern however is when the outer daemons are launched on remote machines, and you then wish to launch daemons locally on each of those machines. In this case, use the following solution: instead of a single call to `daemons(n)` make 2 separate calls to `daemons(url = local_url()); launch_`
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())[]
```

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

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	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 carry out 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	[default "Rscript"] assumes the R executable is on the search path. Replace with the full path of the Rscript executable on the remote machine if necessary. If launching on Windows, "Rscript" should be replaced with "Rscript.exe".
quote	[default FALSE] logical value whether or not to quote the daemon launch command (not required for Slurm "srun" for example, but required for Slurm "sbatch" or "ssh").

Value

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

See Also

ssh_config() for SSH launch configurations, or cluster_config() for cluster resource manager launch configurations.

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 dameons with special configurations
remote_config(
  command = "Rscript",
  rscript = "--default-packages=NULL --vanilla"
)
```

`require_daemons` *Require Daemons*

Description

Returns TRUE 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(call = environment(), .compute = NULL)
```

Arguments

`call`	(only used if the **cli** package is installed) the execution environment of a currently running function, e.g. `environment()`. The function will be mentioned in error messages as the source of the error.
`.compute`	[default NULL] character value for the compute profile to query, or NULL to query the 'default' profile. **or** a 'miraiCluster' to obtain its status.

Value

Logical `TRUE`, or else errors.

Examples

```
daemons(1)
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`	a character string (or vector) of the class of object custom serialization functions are applied to, e.g. `'ArrowTabular'` or `c('torch_tensor', 'ArrowTabular')`.
`sfunc`	a function (or list of functions) that accepts a reference object inheriting from `class` and returns a raw vector.
`ufunc`	a function (or list of functions) that accepts a raw vector and returns a reference object.

Details

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

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	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'.
tunnel	[default FALSE] logical value, whether to use SSH tunnelling. If TRUE, requires the daemons() url hostname to be '127.0.0.1'. See the 'SSH Tunnelling' section below for further details.
timeout	[default 10] maximum time allowed for connection setup in seconds.
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 carry out any launches, but causes launch_remote() to return the shell commands for manual deployment on remote machines.
rscript	[default "Rscript"] assumes the R executable is on the search path. Replace with the full path of the Rscript executable on the remote machine if necessary. If launching on Windows, "Rscript" should be replaced with "Rscript.exe".

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.

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, and is hence suitable for use within trusted networks.

SSH Tunnelling

Use of SSH tunnelling provides a convenient way to launch remote daemons 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 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.

This provides a means of launching daemons on any machine you are able to access via SSH, be it on the local network or the cloud.

See Also

`cluster_config()` for cluster resource manager launch configurations, or `remote_config()` for generic configurations.

Examples

```
# direct SSH example
ssh_config(c("ssh://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 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(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 and daemons status.

Usage

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

Arguments

`.compute`	[default NULL] character value for the compute profile to query, or NULL to query the 'default' profile. **or** a 'miraiCluster' to obtain its status.

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).

Events

If dispatcher is used combined with daemon IDs, an additional element **events** will report the positive integer ID when the daemon connects and the negative value when it disconnects. Only the events since the previous status query are returned.

Examples

```
status()
daemons(url = "tcp://[::1]:0")
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`	a 'mirai' object, or list of 'mirai' objects.

Details

Using dispatcher allows cancellation of 'mirai'. In the case that the 'mirai' is awaiting execution, it is discarded from the queue and never evaluated. In the case it is already in execution, an interrupt will be sent.

A successful cancellation request does not guarantee successful cancellation: the task, or a portion of it, may have already completed before the interrupt is received. Even then, compiled code is not always interruptible. This should be noted, particularly if the code carries out side effects during execution, such as writing to files, etc.

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`	a 'mirai' object or list of 'mirai' objects, or a '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 `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	a call to daemons().
expr	an expression to evaluate.
...	not used.

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")
  }
)

status()
```

Index