Package: rush (via r-universe)

November 20, 2026

Title Rapid Asynchronous and Distributed Computing

Version 1.4.0

Description Package to tackle large-scale problems asynchronously across a distributed network. Employing a database centric model, rush enables workers to communicate tasks and their results over a shared 'Redis' database. Key features include low task overhead, efficient caching, and robust error handling. The package powers the asynchronous optimization algorithms in the 'bbotk' and 'mlr3tuning' packages.

License MIT + file LICENSE

URL https://rush.mlr-org.com, https://github.com/mlr-org/rush

BugReports https://github.com/mlr-org/rush/issues

Depends R (>= 3.6.0)

Imports checkmate, data.table, ids, jsonlite, lgr (>= 0.5.0), mirai (>= 2.5.0), mlr3misc (>= 0.20.0), processx, R6, redux, uuid

Suggests callr, knitr, lhs, quarto, ranger, rmarkdown, testthat (>= 3.0.0), xgboost

SystemRequirements Redis (>= 7.0)

Config/testthat/edition 3

Config/testthat/parallel false

Encoding UTF-8

Config/roxygen2/version 8.1.0

NeedsCompilation no

Author Marc Becker [cre, aut, cph] (ORCID: <https://orcid.org/0000-0002-8115-0400>)

Maintainer Marc Becker <marcbecker@posteo.de>

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

Date/Publication 2026-11-20 13:13:52 UTC

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

RemoteRef HEAD

RemoteSha 9fe20f5d1d92a16a4294afbb337cb779811e430a

Contents

rush-package . . . 2
AppenderRedis . . . 3
remove_rush_plan . . . 5
rsh . . . 5
Rush . . . 6
rush_available . . . 23
rush_config . . . 23
rush_plan . . . 24
RushWorker . . . 26
start_worker . . . 29
store_large_object . . . 31

Index **33**

`rush-package` *rush: Rapid Asynchronous and Distributed Computing*

Description

Package to tackle large-scale problems asynchronously across a distributed network. Employing a database centric model, rush enables workers to communicate tasks and their results over a shared 'Redis' database. Key features include low task overhead, efficient caching, and robust error handling. The package powers the asynchronous optimization algorithms in the 'bbotk' and 'mlr3tuning' packages.

Options

- `rush.max_object_size`: Maximum size in MiB of the serialized worker configuration stored in Redis. Defaults to `512`. If the configuration exceeds this limit, an error is raised unless `large_objects_path` is set in `rush_plan()`, in which case the configuration is written to disk instead.

Environment Variables

- `REDIS_URL`: Connection URL parsed by redux::redis_config to configure the connection to Redis. Used when no `config` is set via `rush_plan()`. If unset, a default configuration is used.

Author(s)

Maintainer: Marc Becker <marcbecker@posteo.de> (ORCID) [copyright holder]

Authors:

- Marc Becker <marcbecker@posteo.de> (ORCID) [copyright holder]

See Also

Useful links:

- https://rush.mlr-org.com
- https://github.com/mlr-org/rush
- Report bugs at https://github.com/mlr-org/rush/issues

AppenderRedis *Log to Redis Database*

Description

AppenderRedis writes log messages to a Redis database. This lgr::Appender is created internally by RushWorker when logger thresholds are passed via `rush_plan()`.

Value

Object of class R6::R6Class and `AppenderRedis` with methods for writing log events to Redis databases.

Super classes

`lgr::Filterable` -> `lgr::Appender` -> `lgr::AppenderMemory` -> `AppenderRedis`

Methods

Public methods:

- `AppenderRedis$new()`
- `AppenderRedis$flush()`

`AppenderRedis$new()`: Creates a new instance of this R6 class.

Usage:

```
AppenderRedis$new(
  config,
  key,
  threshold = NA_integer_,
  layout = lgr::LayoutJson$new(timestamp_fmt = "%Y-%m-%d %H:%M:%OS3",
    transform_event = function(event) {
     values = event$values

   values[intersect(names(values), c("level", "timestamp", "logger", "caller", "msg",
    "rawMsg"))]
  }),
  buffer_size = 0,
  flush_threshold = "error",
  flush_on_exit = TRUE,
```

```
  flush_on_rotate = TRUE,
  should_flush = NULL,
  filters = NULL
)
```

Arguments:

`config` (redux::redis_config)
: Redis configuration options.

`key` (`character(1)`)
: Key of the list holding the log messages in the Redis data store.

`threshold` (`integer(1)` | `character(1)`)
: Threshold for the log messages.

`layout` (lgr::Layout)
: Layout for the log messages. The default layout strips custom fields from the log events, because they might not be JSON-serializable. The stripping is scoped to this appender, so other appenders on the same logger still see the custom fields.

`buffer_size` (`integer(1)`)
: Size of the buffer.

`flush_threshold` (`character(1)`)
: Threshold for flushing the buffer.

`flush_on_exit` (`logical(1)`)
: Flush the buffer on exit.

`flush_on_rotate` (`logical(1)`)
: Flush the buffer on rotate.

`should_flush` (`function`)
: Function that determines if the buffer should be flushed.

`filters` (`list`)
: List of filters.

`AppenderRedis$flush()`**:** Sends the buffer's contents to the Redis data store, and then clears the buffer.

Usage:

```
AppenderRedis$flush()
```

Examples

```
if (redux::redis_available()) {
   config_local = redux::redis_config()

   rush_plan(
     config = config_local,
     n_workers = 2,
     lgr_thresholds = c(rush = "info"))

   rush = rsh(network_id = "test_network")
   rush
}
```

remove_rush_plan *Remove Rush Plan*

Description

Removes the rush plan that was set by `rush_plan()`.

Usage

```
remove_rush_plan()
```

Value

Invisible `TRUE`. Function called for side effects.

Examples

```
if (redux::redis_available()) {
  config_local = redux::redis_config()
  rush_plan(config = config_local, n_workers = 2)
  remove_rush_plan()
}
```

rsh *Syntactic Sugar for Rush Manager Construction*

Description

Function to construct a Rush manager.

Usage

```
rsh(network_id = NULL, config = NULL)
```

Arguments

`network_id`	(`character(1)`) Identifier of the rush network. Manager and workers must have the same id. Keys in Redis are prefixed with the instance id.
`config`	(redux::redis_config) Redis configuration options. If `NULL`, configuration set by `rush_plan()` is used. If `rush_plan()` has not been called, the `REDIS_URL` environment variable is parsed. If `REDIS_URL` is not set, a default configuration is used. See redux::redis_config for details.

Value

Rush manager.

Examples

```
if (redux::redis_available()) {
   config_local = redux::redis_config()
   rush = rsh(network_id = "test_network", config = config_local)
   rush
}
```

Rush *Rush*

Description

The Rush class manages a rush network by starting, monitoring, and stopping workers. It shares all task-related methods (e.g., fetching results, pushing tasks) with RushWorker. A Rush instance is created with the `rsh()` function which requires a network ID and a config argument to connect to the Redis database via the **redux** package.

Value

Object of class R6::R6Class and Rush.

Tasks

Tasks are the unit in which workers exchange information. The main components of a task are the key, computational state, input (`xs`), and output (`ys`). The key is a unique identifier for the task in the Redis database. The four possible computational states are `"running"`, `"finished"`, `"failed"`, and `"queued"`. The input `xs` and output `ys` are lists that can contain arbitrary data.

Methods to create a task:

- `$push_running_tasks(xss)`: Create running tasks
- `$push_finished_tasks(xss, yss)`: Create finished tasks.
- `$push_failed_tasks(xss, conditions)`: Create failed tasks.
- `$push_tasks(xss)`: Create queued tasks, optionally for a compute profile.

These methods return the key of the created tasks. The methods work on multiple tasks at once, so `xss` and `yss` are lists of inputs and outputs. When tasks are fetched, the `xss` and `yss` are unpacked so that the names of their inner elements become the columns of the returned table. For example, a `xss` stored as `list(list(x1 = 2, x2 = 3), list(x1 = 4, x2 = 5))` yields `x1` and `x2` columns, not a `xs` column. The inner element names must therefore be unique across these fields.

Methods to change the state of an existing task:

- `$finish_tasks(keys, yss)`: Save the output of tasks and mark them as finished.
- `$fail_tasks(keys, conditions)`: Mark tasks as failed and optionally save the condition objects.

- `$pop_task()`: Pop a task from the queue and mark it as running.

The methods `$pop_task()`, `$push_running_tasks(xss)`, `$finish_tasks(keys, yss)`, and `$fail_tasks(keys, condit` are only available on RushWorker.

The following methods are used to fetch tasks:

- `$fetch_tasks()`: Fetch all tasks.
- `$fetch_finished_tasks()`: Fetch finished tasks.
- `$fetch_failed_tasks()`: Fetch failed tasks.
- `$fetch_tasks_with_state()`: Fetch tasks with different states at once.
- `$fetch_new_tasks()`: Fetch new tasks and optionally block until new tasks are available.

The methods return a `data.table()` with the tasks.

Tasks have the following fields:

- `xs`: The input of the task.
- `ys`: The output of the task.
- `xs_extra`: Metadata created when creating the task.
- `ys_extra`: Metadata created when finishing the task.
- `condition`: Condition object when the task failed.
- `worker_id`: The id of the worker that created the task.

Workers

Workers are spawned with the `$start_workers()` method on `mirai` daemons. Use `mirai::daemons()` to start daemons. Workers can be started on the

- local machine,
- remote machine
- or HPC cluster using the **mirai** package.

Alternatively, workers can be started locally with the `$start_local_workers()` method via the **processx** package. Or a help script can be generated with the `$worker_script()` method that can be run anywhere. The only requirement is that the worker can connect to the Redis database.

Compute Profiles

Daemons can be started on separate compute profiles of **mirai**, e.g. one profile for CPU daemons and one profile for GPU daemons. The `profiles` argument of the `$start_workers()` method distributes the workers over these profiles.

```
mirai::daemons(2, .compute = "cpu")
mirai::daemons(2, .compute = "gpu")

rush$start_workers(worker_loop = worker_loop, profiles = c(cpu = 2, gpu = 2))
```

The profile of a worker is recorded in the `profile` column of `$worker_info` and is passed to the worker loop, see the worker loop section.

Each compute profile has its own queue. Tasks pushed with `$push_tasks(xss, profile = "gpu")` are only processed by the workers of the `"gpu"` profile. Tasks pushed without a profile are added to the shared queue and are processed by any worker. A worker takes tasks from the queue of its profile first and falls back to the shared queue. The `$n_queued_tasks_per_profile` field shows the number of tasks queued for each profile.

Worker Loop

The worker loop is the main function that is run on the workers. It is defined by the user and is passed to the `$start_workers()` method. The first argument of the worker loop is the RushWorker instance, which is passed as `rush`. If the worker loop has a `profile` argument, the name of the compute profile the worker runs on is passed to it. The profile is `NULL` when the worker runs on the default compute profile.

```
worker_loop = function(rush, profile = NULL) {
  while (!rush$terminated) {
    task = rush$pop_task()
    ...
  }
}
```

Debugging

The `mirai::mirai` objects started with `$start_workers()` are stored in `$processes_mirai`. Standard output and error of the workers can be written to log files with the `message_log` and `output_log` arguments of `$start_workers()`.

Public fields

`processes_processx` (processx::process)
: List of processes started with `$start_local_workers()`.

`processes_mirai` (mirai::mirai)
: List of mirai processes started with `$start_workers()`.

Active bindings

`network_id` (`character(1)`)
: Identifier of the rush network.

`config` (redux::redis_config)
: Redis configuration options. Assigning a new configuration does not affect the live connection. Call `$reconnect()` afterwards to connect with the new configuration.

`connector` (redux::redis_api)
: Returns a connection to Redis.

`n_workers` (`integer(1)`)
: Number of workers.

`n_running_workers` (`integer(1)`)
Number of running workers.

`n_terminated_workers` (`integer(1)`)
Number of terminated workers.

`worker_ids` (`character()`)
Ids of workers.

`running_worker_ids` (`character()`)
Ids of running workers.

`terminated_worker_ids` (`character()`)
Ids of terminated workers.

`tasks` (`character()`)
Keys of all tasks.

`queued_tasks` (`character()`)
Keys of queued tasks in the shared queue and the queues of all compute profiles.

`running_tasks` (`character()`)
Keys of running tasks.

`finished_tasks` (`character()`)
Keys of finished tasks.

`failed_tasks` (`character()`)
Keys of failed tasks.

`n_queued_tasks` (`integer(1)`)
Number of queued tasks in the shared queue and the queues of all compute profiles.

`n_queued_tasks_per_profile` (named `integer()`)
Number of queued tasks in the shared queue and the queues of all compute profiles. The number of tasks in the shared queue is named `"default"`.

`n_running_tasks` (`integer(1)`)
Number of running tasks.

`n_finished_tasks` (`integer(1)`)
Number of finished tasks.

`n_failed_tasks` (`integer(1)`)
Number of failed tasks.

`n_tasks` (`integer(1)`)
Number of all tasks.

`worker_info` (`data.table::data.table()`)
Contains information about the workers.

Methods

Public methods:

- `Rush$new()`
- `Rush$format()`
- `Rush$print()`
- `Rush$reconnect()`

- Rush$start_workers()
- Rush$start_local_workers()
- Rush$start_remote_workers()
- Rush$worker_script()
- Rush$wait_for_workers()
- Rush$stop_workers()
- Rush$detect_lost_workers()
- Rush$reset()
- Rush$read_log()
- Rush$print_log()
- Rush$push_tasks()
- Rush$push_finished_tasks()
- Rush$push_failed_tasks()
- Rush$empty_queue()
- Rush$fetch_tasks()
- Rush$fetch_queued_tasks()
- Rush$fetch_running_tasks()
- Rush$fetch_failed_tasks()
- Rush$fetch_finished_tasks()
- Rush$fetch_tasks_with_state()
- Rush$fetch_new_tasks()
- Rush$reset_cache()
- Rush$wait_for_tasks()
- Rush$write_hashes()
- Rush$read_hashes()
- Rush$read_hash()
- Rush$is_running_task()
- Rush$is_failed_task()
- Rush$tasks_with_state()
- Rush$clone()

`Rush$new()`: Creates a new instance of this R6 class.

Usage:

```
Rush$new(network_id = NULL, config = NULL)
```

Arguments:

`network_id` (`character(1)`)
: Identifier of the rush network. Manager and workers must have the same id. Keys in Redis are prefixed with the instance id.

`config` (redux::redis_config)
: Redis configuration options. If `NULL`, configuration set by `rush_plan()` is used. If `rush_plan()` has not been called, the `REDIS_URL` environment variable is parsed. If `REDIS_URL` is not set, a default configuration is used. See redux::redis_config for details.

`Rush$format()`: Helper for print outputs.

Usage:

```
Rush$format(...)
```

Arguments:

`...` (ignored).

Returns: (`character()`).

`Rush$print()`: Print method.

Usage:

```
Rush$print()
```

Returns: (`character()`).

`Rush$reconnect()`: Reconnect to Redis. The connection breaks when the Rush object is saved to disk. Call this method to reconnect after loading the object.

Usage:

```
Rush$reconnect()
```

`Rush$start_workers()`: Start workers to run the worker loop in `mirai::daemons()`. Initializes a RushWorker in each process and starts the worker loop.

Workers are started on the default compute profile unless `profiles` is given. With `profiles`, the workers are distributed over the compute profiles of **mirai**.

Usage:

```
Rush$start_workers(
  worker_loop,
  ...,
  n_workers = NULL,
  profiles = NULL,
  packages = NULL,
  lgr_thresholds = NULL,
  lgr_buffer_size = NULL,
  message_log = NULL,
  output_log = NULL,
  restart = NULL,
  launcher = NULL,
  max_restarts = NULL
)
```

Arguments:

`worker_loop` (`function`)
: Loop run on the workers.

`...` (`any`)
: Arguments passed to `worker_loop`.

`n_workers` (`integer(1)`)
: Number of workers to be started.

`profiles` (named `integer()`)
: Number of workers to be started on each `mirai` compute profile, e.g. `c(cpu = 2, gpu = 2)`. The names are the compute profiles created with `mirai::daemons()` and the values are the number of workers started on the daemons of the respective profile. Cannot be combined with `n_workers`.

`packages` (`character()`)
: Packages to be loaded by the workers.

`lgr_thresholds` (named `character()` | named `numeric()`)
: Logger threshold on the workers e.g. `c("mlr3/rush" = "debug")`.

`lgr_buffer_size` (`integer(1)`)
: By default (`lgr_buffer_size = 0`), the log messages are directly saved in the Redis data store. If `lgr_buffer_size > 0`, the log messages are buffered and saved in the Redis data store when the buffer is full. This improves the performance of the logging.

`message_log` (`character(1)`)
: Path to the message log files e.g. `/tmp/message_logs/` The message log files are named `message_<worker_id>.log`. If `NULL`, no messages, warnings or errors are stored.

`output_log` (`character(1)`)
: Path to the output log files e.g. `/tmp/output_logs/` The output log files are named `output_<worker_id>.log`. If `NULL`, no output is stored.

`restart` (`logical(1)`)
: Whether to restart lost workers started with `$start_workers()`. `$detect_lost_workers()` starts a new worker with a new worker id for each lost worker. The new worker runs on the same compute profile and stores the id of the lost worker in `restarted_from`. Default is `FALSE`.

`launcher` (`function()` | `list()`)
: Relaunches a daemon when the daemon of a lost worker has died, e.g. because its Slurm job was canceled. Either a launcher configuration of **mirai** created with `mirai::cluster_config()`, `mirai::ssh_config()`, or `mirai::remote_config()`, which is passed to `mirai::launch_remote()`, or a function with the arguments `n` and `profile` that launches `n` daemons on the compute profile `profile`. If `NULL`, the new worker waits until a daemon is available, e.g. because the scheduler requeues the job. Only used if `restart = TRUE`.

`max_restarts` (`integer(1)`)
: Maximum number of restarts of a worker and its successors. Default is `3`.

`Rush$start_local_workers()`**:** Start workers locally with `processx`. Initializes a RushWorker in each process and starts the worker loop. Use `$wait_for_workers()` to wait until the workers are registered in the network.

Usage:

```
Rush$start_local_workers(
  worker_loop,
  ...,
  n_workers = NULL,
  packages = NULL,
  lgr_thresholds = NULL,
  lgr_buffer_size = NULL,
  supervise = TRUE,
  message_log = NULL,
```

```
  output_log = NULL
)
```

Arguments:

`worker_loop` (`function`)
: Loop run on the workers.

`...` (`any`)
: Arguments passed to `worker_loop`.

`n_workers` (`integer(1)`)
: Number of workers to be started.

`packages` (`character()`)
: Packages to be loaded by the workers.

`lgr_thresholds` (named `character()` | named `numeric()`)
: Logger threshold on the workers e.g. `c("mlr3/rush" = "debug")`.

`lgr_buffer_size` (`integer(1)`)
: By default (`lgr_buffer_size = 0`), the log messages are directly saved in the Redis data store. If `lgr_buffer_size > 0`, the log messages are buffered and saved in the Redis data store when the buffer is full. This improves the performance of the logging.

`supervise` (`logical(1)`)
: Whether to kill the workers when the main R process is shut down.

`message_log` (`character(1)`)
: Path to the message log files e.g. `/tmp/message_logs/` The message log files are named `message_<worker_id>.log`. If `NULL`, no messages, warnings or errors are stored.

`output_log` (`character(1)`)
: Path to the output log files e.g. `/tmp/output_logs/` The output log files are named `output_<worker_id>.log`. If `NULL`, no output is stored.

`Rush$start_remote_workers()`: Start workers to run the worker loop in `mirai::daemons()`. Initializes a RushWorker in each process and starts the worker loop.

Usage:

```
Rush$start_remote_workers(
  worker_loop,
  ...,
  n_workers = NULL,
  packages = NULL,
  lgr_thresholds = NULL,
  lgr_buffer_size = NULL,
  message_log = NULL,
  output_log = NULL
)
```

Arguments:

`worker_loop` (`function`)
: Loop run on the workers.

`...` (`any`)
: Arguments passed to `worker_loop`.

`n_workers` (`integer(1)`)
: Number of workers to be started.

packages (character())
: Packages to be loaded by the workers.

lgr_thresholds (named character() | named numeric())
: Logger threshold on the workers e.g. c("mlr3/rush" = "debug").

lgr_buffer_size (integer(1))
: By default (lgr_buffer_size = 0), the log messages are directly saved in the Redis data store. If lgr_buffer_size > 0, the log messages are buffered and saved in the Redis data store when the buffer is full. This improves the performance of the logging.

message_log (character(1))
: Path to the message log files e.g. /tmp/message_logs/ The message log files are named message_<worker_id>.log. If NULL, no messages, warnings or errors are stored.

output_log (character(1))
: Path to the output log files e.g. /tmp/output_logs/ The output log files are named output_<worker_id>.log. If NULL, no output is stored.

Rush$worker_script(): Generate a script to start workers. Run this script n times to start n workers. The logged variant of the script redacts the Redis password. The script is quoted for POSIX shells (e.g., sh, bash, zsh).

Always set heartbeat_period when using this method. The heartbeat is the only way to manage a worker that was started from a script, because there is no process handle on the manager side. Without a heartbeat, $stop_workers(type = "kill") cannot kill the worker, and $detect_lost_workers() cannot detect its crash, so a crashed worker stays in the running state forever.

Usage:

```
Rush$worker_script(
  worker_loop,
  ...,
  packages = NULL,
  lgr_thresholds = NULL,
  lgr_buffer_size = NULL,
  heartbeat_period = NULL,
  heartbeat_expire = NULL,
  message_log = NULL,
  output_log = NULL
)
```

Arguments:

worker_loop (function)
: Loop run on the workers.

... (any)
: Arguments passed to worker_loop.

packages (character())
: Packages to be loaded by the workers.

lgr_thresholds (named character() | named numeric())
: Logger threshold on the workers e.g. c("mlr3/rush" = "debug").

lgr_buffer_size (integer(1))
: By default (lgr_buffer_size = 0), the log messages are directly saved in the Redis data

store. If `lgr_buffer_size > 0`, the log messages are buffered and saved in the Redis data store when the buffer is full. This improves the performance of the logging.

`heartbeat_period (integer(1))`
Period of the heartbeat in seconds. The heartbeat is updated every `heartbeat_period` seconds. Must be at least 1 second.

`heartbeat_expire (integer(1))`
Time to live of the heartbeat in seconds. The heartbeat key is set to expire after `heartbeat_expire` seconds. Must be at least `heartbeat_period`, otherwise a live worker is reaped as lost between two heartbeats. Set it larger than the longest pause a worker may experience, for example from garbage collection or swapping, because a live worker wrongly declared lost can leave a task in an inconsistent state.

`message_log (character(1))`
Path to the message log files e.g. `/tmp/message_logs/` The message log files are named `message_<worker_id>.log`. If `NULL`, no messages, warnings or errors are stored.

`output_log (character(1))`
Path to the output log files e.g. `/tmp/output_logs/` The output log files are named `output_<worker_id>.log`. If `NULL`, no output is stored.

Returns: `(character(1))`
Shell command to start a worker.

`Rush$wait_for_workers()`**:** Wait until workers are registered in the network. Either `n`, `worker_ids` or both must be provided.

Usage:

```
Rush$wait_for_workers(n = NULL, worker_ids = NULL, timeout = NULL)
```

Arguments:

`n (integer(1))`
Number of workers to wait for. If `NULL`, wait for all workers in `worker_ids`.

`worker_ids (character())`
Worker ids to wait for. If `NULL`, wait for any `n` workers to be registered.

`timeout (numeric(1))`
Timeout in seconds. Defaults to the `start_worker_timeout` set with `rush_plan()`, or `Inf` if none is set. A `timeout` of `0` checks once and errors immediately if the workers are not yet registered.

`Rush$stop_workers()`**:** Stop workers.

Usage:

```
Rush$stop_workers(type = "kill", worker_ids = NULL)
```

Arguments:

`type (character(1))`
Type of stopping. Either `"terminate"` or `"kill"`. If `"kill"` the workers are stopped immediately, and their running tasks are marked as failed with the condition message `"Worker was killed"`. If `"terminate"` the workers evaluate the currently running task and then terminate. The `"terminate"` option must be implemented in the worker loop. The `"kill"` option requires a process handle from `$start_workers()` or `$start_local_workers()`, or a heartbeat. Workers started from `$worker_script()` without a `heartbeat_period` are silently skipped.

worker_ids (character())
: Worker ids to be stopped. If NULL all workers are stopped. Ids that are not currently running are skipped with a warning.

Rush$detect_lost_workers(): Detect lost workers. The state of the worker is changed to "terminated".

Workers started with mirai or processx are monitored through their process handle, so a worker is only declared lost after its process has actually terminated. Workers started from $worker_script() are monitored through a heartbeat and are declared lost when the heartbeat key expires. Workers started from $worker_script() without a heartbeat_period cannot be monitored at all, so a crashed worker stays in the running state forever. Because this is a timeout, heartbeat_expire must be larger than the longest pause a worker may experience, for example from garbage collection or swapping. If a live worker is wrongly declared lost, its running and pending tasks are marked as failed, and the results of tasks the worker finishes afterwards are discarded. Set heartbeat_expire conservatively to avoid discarding results.

Usage:

```
Rush$detect_lost_workers()
```

Returns: (character())
Worker ids of detected lost workers.

Rush$reset(): Stop workers and delete data stored in redis.

Usage:

```
Rush$reset(workers = TRUE)
```

Arguments:

workers (logical(1))
: Whether to stop the workers or only delete the data. Default is TRUE.

Rush$read_log(): Read log messages written with the lgr package by the workers.

Usage:

```
Rush$read_log(worker_ids = NULL, time_difference = FALSE)
```

Arguments:

worker_ids (character())
: Worker ids to be read log messages from. Defaults to all worker ids.

time_difference (logical(1))
: Whether to calculate the time difference between log messages.

Returns: data.table()
Table with columns worker_id, level, timestamp, logger, caller and msg, and optionally time_difference.

Rush$print_log(): Print log messages written with the lgr package by the workers. Log messages are printed with the original logger.

Usage:

```
Rush$print_log()
```

Returns: (Rush)
Invisible self.

`Rush$push_tasks()`: Create tasks and add them to the queue.

Tasks pushed without a `profile` are added to the shared queue and are processed by any worker. Tasks pushed with a `profile` are added to the queue of the compute profile and are only processed by the workers running on that profile.

Usage:

```
Rush$push_tasks(xss, xss_extra = NULL, extra = NULL, profile = NULL)
```

Arguments:

`xss` (list of named `list()`)
: Lists of arguments for the function e.g. `list(list(x1 = 1, x2 = 2), list(x1 = 3, x2 = 4))`. If `xss` is empty, no tasks are created and the method returns an empty `character()`.

`xss_extra` (list of named `list()`)
: List of additional information stored along with the task e.g. `list(list(timestamp_xs = Sys.time()), list(timestamp_xs = Sys.time()))`.

`extra` (`list()`)
: Deprecated argument for additional information stored along with the task. Use `xss_extra` instead.

`profile` (`character(1)`)
: Name of the `mirai` compute profile the tasks are queued for. If `NULL`, the tasks are added to the shared queue.

Returns: (`character()`)
Keys of the tasks.

`Rush$push_finished_tasks()`: Create finished tasks. See `$finish_tasks()` for moving existing tasks from running to finished.

Usage:

```
Rush$push_finished_tasks(xss, yss, xss_extra = NULL, yss_extra = NULL)
```

Arguments:

`xss` (list of named `list()`)
: Lists of arguments for the function e.g. `list(list(x1 = 1, x2 = 2), list(x1 = 3, x2 = 4))`. If `xss` is empty, no tasks are created and the method returns an empty `character()`.

`yss` (list of named `list()`)
: Lists of results for the function e.g. `list(list(y1 = 1, y2 = 2), list(y1 = 3, y2 = 4))`.

`xss_extra` (list of named `list()`)
: List of additional information stored along with the task e.g. `list(list(timestamp_xs = Sys.time()), list(timestamp_xs = Sys.time()))`.

`yss_extra` (list of named `list()`)
: List of additional information stored along with the results e.g. `list(list(timestamp_ys = Sys.time()), list(timestamp_ys = Sys.time()))`.

Returns: (`character()`)
Keys of the tasks.

`Rush$push_failed_tasks()`: Create failed tasks. See `$fail_tasks()` for moving existing tasks from queued and running to failed.

Usage:

```
Rush$push_failed_tasks(xss, xss_extra = NULL, conditions = NULL)
```

Arguments:

`xss` (list of named `list()`)
: Lists of arguments for the function e.g. `list(list(x1 = 1, x2 = 2), list(x1 = 3, x2 = 4))`. If `xss` is empty, no tasks are created and the method returns an empty `character()`.

`xss_extra` (list of named `list()`)
: List of additional information stored along with the task e.g. `list(list(timestamp_xs = Sys.time()), list(timestamp_xs = Sys.time()))`.

`conditions` (`list()`)
: List conditions e.g. `list(simpleError("Error"), simpleError("Error"))`. Defaults to `list(message = "Task failed")`.

Returns: (`character()`)
Keys of the tasks.

`Rush$empty_queue()`**:** Remove all tasks from the queue. The state of the tasks is set to failed. The condition message is set to `"Removed from queue"`.

Usage:

```
Rush$empty_queue()
```

Returns: (`Rush`)
Invisible self.

`Rush$fetch_tasks()`**:** Fetch all tasks from the database.

Usage:

```
Rush$fetch_tasks(
  fields = c("xs", "ys", "xs_extra", "worker_id", "ys_extra", "condition")
)
```

Arguments:

`fields` (`character()`)
: Fields to be read from the hashes. Defaults to `c("xs", "ys", "xs_extra", "worker_id", "ys_extra", "condition")`.

Returns: `data.table()`
Table of all tasks.

`Rush$fetch_queued_tasks()`**:** Fetch queued tasks from the database. Tasks queued for a compute profile have a `profile` column.

Usage:

```
Rush$fetch_queued_tasks(fields = c("xs", "xs_extra", "profile"))
```

Arguments:

`fields` (`character()`)
: Fields to be read from the hashes. Defaults to `c("xs", "xs_extra", "profile")`.

Returns: `data.table()`
Table of queued tasks.

`Rush$fetch_running_tasks()`**:** Fetch running tasks from the database.

Usage:

```
Rush$fetch_running_tasks(fields = c("xs", "xs_extra", "worker_id"))
```

Arguments:

`fields (character())`
Fields to be read from the hashes. Defaults to `c("xs", "xs_extra", "worker_id")`.

Returns: `data.table()`
Table of running tasks.

`Rush$fetch_failed_tasks()`**:** Fetch failed tasks from the database.

Usage:

```
Rush$fetch_failed_tasks(fields = c("xs", "xs_extra", "worker_id", "condition"))
```

Arguments:

`fields (character())`
Fields to be read from the hashes. Defaults to `c("xs", "xs_extra", "worker_id", "condition")`.

Returns: `data.table()`
Table of failed tasks.

`Rush$fetch_finished_tasks()`**:** Fetch finished tasks from the database. Finished tasks are cached.

Usage:

```
Rush$fetch_finished_tasks(
  fields = c("worker_id", "xs", "ys", "xs_extra", "ys_extra", "condition")
)
```

Arguments:

`fields (character())`
Fields to be read from the hashes. Defaults to `c("worker_id", "xs", "ys", "xs_extra", "ys_extra", "condition")`. If the fields change between calls, fields requested only by a later call remain `NA` for the already cached tasks. Use `$reset_cache()` to reset the cache in this case.

Returns: `data.table()`
Table of finished tasks.

`Rush$fetch_tasks_with_state()`**:** Fetch tasks with different states from the database. If tasks with different states are to be queried at the same time, this function prevents tasks from appearing twice. This could be the case if a worker changes the state of a task while the tasks are being fetched. Finished tasks are cached.

Usage:

```
Rush$fetch_tasks_with_state(
  fields = c("worker_id", "xs", "ys", "xs_extra", "ys_extra", "condition"),
  states = c("queued", "running", "finished", "failed")
)
```

Arguments:

`fields` (`character()`)
Fields to be read from the hashes. Defaults to `c("worker_id", "xs", "ys", "xs_extra", "ys_extra", "condition")`. If the fields change between calls, fields requested only by a later call remain `NA` for the already cached tasks. Use `$reset_cache()` to reset the cache in this case.

`states` (`character()`)
States of the tasks to be fetched. Defaults to `c("queued", "running", "finished", "failed")`.

`Rush$fetch_new_tasks()`: Fetch new tasks that finished after the last call of this function. Updates the cache of the finished tasks. If `timeout` is set, blocks until new tasks are available or the timeout is reached.

"New" is tracked relative to previous calls of this method only. `$fetch_finished_tasks()` grows the same cache but does not advance this counter, so a task already returned by `$fetch_finished_tasks()` is returned again by the next `$fetch_new_tasks()`.

Usage:

```
Rush$fetch_new_tasks(
  fields = c("xs", "ys", "xs_extra", "worker_id", "ys_extra", "condition"),
  timeout = 0
)
```

Arguments:

`fields` (`character()`)
Fields to be read from the hashes. If the fields change between calls, fields requested only by a later call remain `NA` for the already cached tasks. Use `$reset_cache()` to reset the cache in this case.

`timeout` (`numeric(1)`)
Time to wait for new results in seconds. Defaults to `0` (no waiting).

Returns: `data.table()`
Table of latest results.

`Rush$reset_cache()`: Reset the cache of the finished tasks.

Usage:

```
Rush$reset_cache()
```

Returns: (`Rush`)
Invisible self.

`Rush$wait_for_tasks()`: Wait until tasks are finished. The function also unblocks when no worker is running or all tasks failed.

Usage:

```
Rush$wait_for_tasks(keys, detect_lost_workers = FALSE)
```

Arguments:

`keys` (`character()`)
Keys of the tasks to wait for.

`detect_lost_workers` (`logical(1)`)
Whether to detect failed tasks. Comes with an overhead.

`Rush$write_hashes()`: Writes R objects to Redis hashes. The function takes the vectors in `...` as input and writes each element as a field-value pair to a new hash. The name of the argument defines the field into which the serialized element is written. For example, `xs = list(list(x1 = 1, x2 = 2), list(x1 = 3, x2 = 4))` writes `serialize(list(x1 = 1, x2 = 2))` at field `xs` into a hash and `serialize(list(x1 = 3, x2 = 4))` at field `xs` into another hash. The function can iterate over multiple vectors simultaneously. For example, `xs = list(list(x1 = 1, x2 = 2), list(x1 = 3, x2 = 4)), ys = list(l` creates two hashes with the fields `xs` and `ys`. All value lists must either have the same length (the number of hashes) or length 1, in which case the value is broadcast across all hashes. Other length mismatches raise an error. Both lists and atomic vectors are supported. Arguments that are `NULL` are ignored.

Usage:

```
Rush$write_hashes(..., .values = list(), keys = NULL)
```

Arguments:

`...` (named `list()`)
: Lists to be written to the hashes. The names of the arguments are used as fields.

`.values` (named `list()`)
: Lists to be written to the hashes. The names of the list are used as fields.

`keys` (character())
: Keys of the hashes. If `NULL` new keys are generated.

Returns: (`character()`)
Keys of the hashes.

`Rush$read_hashes()`: Reads R Objects from Redis hashes. The function reads the field-value pairs of the hashes stored at `keys`. The values of a hash are deserialized and combined to a list. If `flatten` is `TRUE`, the values are flattened to a single list e.g. `list(xs = list(x1 = 1, x2 = 2), ys = list(y = 3))` becomes `list(x1 = 1, x2 = 2, y = 3)`. The reading functions combine the hashes to a table where the names of the inner lists are the column names. For example, `xs = list(list(x1 = 1, x2 = 2), list(x1 = 3, x2 = 4)), ys = list(list(y = 3), list(y = 7))` becomes `data.table(x1 = c(1, 3), x2 = c(2, 4), y = c(3, 7))`. Names must be unique across the flattened fields. Colliding names produce duplicate columns, of which only the first is reachable by name.

Usage:

```
Rush$read_hashes(keys, fields, flatten = TRUE)
```

Arguments:

`keys` (`character()`)
: Keys of the hashes.

`fields` (`character()`)
: Fields to be read from the hashes.

`flatten` (`logical(1)`)
: Whether to flatten the list.

Returns: (list of `list()`)
The outer list contains one element for each key. The inner list is the combination of the lists stored at the different fields.

`Rush$read_hash()`: Reads a single Redis hash and returns the values as a list named by the fields.

Usage:

```
Rush$read_hash(key, fields)
```

Arguments:

`key (character(1))`
Key of the hash.

`fields (character())`
Fields to be read from the hash.

Returns: (list of `list()`)
The outer list contains one element for each key. The inner list is the combination of the lists stored at the different fields.

`Rush$is_running_task()`: Checks whether tasks have the status `"running"`.

Usage:

```
Rush$is_running_task(keys)
```

Arguments:

`keys (character())`
Keys of the tasks.

`Rush$is_failed_task()`: Checks whether tasks have the status `"failed"`.

Usage:

```
Rush$is_failed_task(keys)
```

Arguments:

`keys (character())`
Keys of the tasks.

`Rush$tasks_with_state()`: Returns keys of requested states.

Usage:

```
Rush$tasks_with_state(states)
```

Arguments:

`states (character())`
States of the tasks.

Returns: (Named list of `character()`).

`Rush$clone()`: The objects of this class are cloneable with this method.

Usage:

```
Rush$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
if (redux::redis_available()) {
  config_local = redux::redis_config()
  rush = rsh(network_id = "test_network", config = config_local)
  rush
}
```

rush_available *Rush Available*

Description

Returns TRUE if a redis config file (redux::redis_config) has been set by `rush_plan()`.

Usage

```
rush_available()
```

Value

`logical(1)`

Examples

```
if (redux::redis_available()) {
  config_local = redux::redis_config()
  rush_plan(config = config_local, n_workers = 2)
  rush_available()
}
```

rush_config *Get Rush Config*

Description

Returns the rush config that was set by `rush_plan()`.

Usage

```
rush_config()
```

Value

`list()` with the stored configuration.

Examples

```
if (redux::redis_available()) {
  config_local = redux::redis_config()
  rush_plan(config = config_local, n_workers = 2)
  rush_config()
}
```

`rush_plan` *Create Rush Plan*

Description

Stores the number of workers and Redis configuration options (redux::redis_config) for Rush. Instead of a single number of workers, the workers can be distributed over the compute profiles of **mirai** with the `profiles` argument. The function tests the connection to Redis and throws an error if the connection fails. This function is usually used in third-party packages to setup how workers are started.

Usage

```
rush_plan(
  n_workers = NULL,
  config = NULL,
  lgr_thresholds = NULL,
  lgr_buffer_size = NULL,
  large_objects_path = NULL,
  worker_type = "mirai",
  start_worker_timeout = NULL,
  profiles = NULL,
  restart = FALSE,
  launcher = NULL,
  max_restarts = 3L
)
```

Arguments

`n_workers` (`integer(1)`)
Number of workers to be started.

`config` (redux::redis_config)
Configuration options used to connect to Redis. If `NULL`, the `REDIS_URL` environment variable is parsed. If `REDIS_URL` is not set, a default configuration is used. See redux::redis_config for details.

`lgr_thresholds` (named `character()` | named `numeric()`)
Logger threshold on the workers e.g. `c("mlr3/rush" = "debug")`.

`lgr_buffer_size`
(`integer(1)`)
By default (`lgr_buffer_size = 0`), the log messages are directly saved in the Redis data store. If `lgr_buffer_size > 0`, the log messages are buffered and saved in the Redis data store when the buffer is full. This improves the performance of the logging.

`large_objects_path`
(`character(1)`)
The path to the directory where large objects are stored. These files are not

removed automatically and the caller is responsible for cleaning them up, e.g. by pointing this at a subdirectory of `tempdir()`.

`worker_type` (`character(1)`)
The type of worker to use. Options are `"mirai"` to start with **mirai**, `"processx"` to use **processx** or `"script"` to get a script to run.

`start_worker_timeout`
(`numeric(1)`)
Default timeout in seconds used by `$wait_for_workers()` of Rush when no `timeout` is passed. If `NULL`, `$wait_for_workers()` waits indefinitely by default. A timeout of `0` checks once and errors immediately if the workers are not yet registered.

`profiles` (named `integer()`)
Number of workers to be started on each `mirai` compute profile, e.g. `c(cpu = 2, gpu = 2)`. The names are the compute profiles created with `mirai::daemons()` and the values are the number of workers started on the daemons of the respective profile. Cannot be combined with `n_workers`.

`restart` (`logical(1)`)
Whether to restart lost workers started with `$start_workers()`. `$detect_lost_workers()` starts a new worker with a new worker id for each lost worker. The new worker runs on the same compute profile and stores the id of the lost worker in `restarted_from`. Default is `FALSE`.

`launcher` (`function()` | `list()`)
Relaunches a daemon when the daemon of a lost worker has died, e.g. because its Slurm job was canceled. Either a launcher configuration of **mirai** created with `mirai::cluster_config()`, `mirai::ssh_config()`, or `mirai::remote_config()`, which is passed to `mirai::launch_remote()`, or a function with the arguments `n` and `profile` that launches `n` daemons on the compute profile `profile`. If `NULL`, the new worker waits until a daemon is available, e.g. because the scheduler requeues the job. Only used if `restart = TRUE`.

`max_restarts` (`integer(1)`)
Maximum number of restarts of a worker and its successors. Default is `3`.

Value

`list()` with the stored configuration.

Examples

```
if (redux::redis_available()) {
  config_local = redux::redis_config()
  rush_plan(config = config_local, n_workers = 2)

  rush = rsh(network_id = "test_network")
  rush
}
```

RushWorker *Rush Worker*

Description

RushWorker inherits all methods from Rush. Upon initialization, the worker registers itself in the Redis database as a running worker. This class is usually not constructed directly by the user.

In addition to the inherited methods, the worker provides methods that require a worker identity:

- `$pop_task()`: Pop a task from the queue and mark it as running.
- `$push_running_tasks(xss)`: Create running tasks evaluated by the worker.
- `$finish_tasks(keys, yss)`: Save the output of tasks and mark them as finished.
- `$fail_tasks(keys, conditions)`: Mark tasks as failed and optionally save the condition objects.
- `$n_queued_available_tasks`: Number of queued tasks the worker can pop.

Value

Object of class R6::R6Class and `RushWorker`.

Super class

`Rush` -> `RushWorker`

Public fields

`worker_id (character(1))`
: Identifier of the worker.

`profile (character(1))`
: Name of the `mirai` compute profile the worker runs on. `NULL` if the worker runs on the default compute profile.

`heartbeat (callr::r_bg)`
: Background process for the heartbeat.

Active bindings

`terminated (logical(1))`
: Whether to shutdown the worker. Used in the worker loop to determine whether to continue.

`n_queued_available_tasks (integer(1))`
: Number of queued tasks the worker can pop, i.e. the tasks in the shared queue and in the queue of the compute profile the worker runs on. Tasks queued for other compute profiles are not counted.

Methods

Public methods:

- RushWorker$new()
- RushWorker$pop_task()
- RushWorker$push_running_tasks()
- RushWorker$finish_tasks()
- RushWorker$fail_tasks()
- RushWorker$set_terminated()
- RushWorker$clone()

`RushWorker$new()`: Creates a new instance of this R6 class.

Usage:

```
RushWorker$new(
  network_id,
  config = NULL,
  worker_id = NULL,
  profile = NULL,
  heartbeat_period = NULL,
  heartbeat_expire = NULL,
  restarted_from = NULL
)
```

Arguments:

`network_id` (`character(1)`)
: Identifier of the rush network. Manager and workers must have the same id. Keys in Redis are prefixed with the instance id.

`config` (redux::redis_config)
: Redis configuration options. If `NULL`, configuration set by `rush_plan()` is used. If `rush_plan()` has not been called, the `REDIS_URL` environment variable is parsed. If `REDIS_URL` is not set, a default configuration is used. See redux::redis_config for details.

`worker_id` (`character(1)`)
: Identifier of the worker. Keys in redis specific to the worker are prefixed with the worker id.

`profile` (`character(1)`)
: Name of the `mirai` compute profile the worker runs on. If `NULL`, the worker runs on the default compute profile.

`heartbeat_period` (`integer(1)`)
: Period of the heartbeat in seconds. The heartbeat is updated every `heartbeat_period` seconds. Must be at least 1 second.

`heartbeat_expire` (`integer(1)`)
: Time to live of the heartbeat in seconds. The heartbeat key is set to expire after `heartbeat_expire` seconds. Must be at least `heartbeat_period`, otherwise a live worker is reaped as lost between two heartbeats. Set it larger than the longest pause a worker may experience, for example from garbage collection or swapping, because a live worker wrongly declared lost can leave a task in an inconsistent state.

`restarted_from` (`character(1)`)
: Id of the lost worker that this worker replaces. `NULL` if the worker is not a restart.

`RushWorker$pop_task()`: Pop a task from the queue and mark it as running. Returns `NULL` if no task is available.

A worker running on a compute profile takes tasks from the queue of its profile first and falls back to the shared queue, so that tasks pushed without a profile are processed by any worker. A worker running on the default compute profile only takes tasks from the shared queue.

Usage:

```
RushWorker$pop_task(timeout = 1, fields = "xs")
```

Arguments:

`timeout` (`numeric(1)`)
Time to wait for task in seconds.

`fields` (`character()`)
Fields to be returned.

`RushWorker$push_running_tasks()`: Create running tasks.

Usage:

```
RushWorker$push_running_tasks(xss, xss_extra = NULL, extra = NULL)
```

Arguments:

`xss` (list of named `list()`)
Lists of arguments for the function e.g. `list(list(x1 = 1, x2 = 2), list(x1 = 3, x2 = 4))`. If `xss` is empty, no tasks are created and the method returns an empty `character()`.

`xss_extra` (list of named `list()`)
List of additional information stored along with the task e.g. `list(list(timestamp_xs = Sys.time()), list(timestamp_xs = Sys.time()))`.

`extra` (`list`)
Deprecated argument for additional information stored along with the task. Use `xss_extra` instead.

Returns: (`character()`)
Keys of the tasks.

`RushWorker$finish_tasks()`: Save the output of tasks and mark them as finished.

Usage:

```
RushWorker$finish_tasks(keys, yss, yss_extra = NULL, extra = NULL)
```

Arguments:

`keys` (`character(1)`)
Keys of the associated tasks.

`yss` (list of named `list()`)
Lists of results for the function e.g. `list(list(y1 = 1, y2 = 2), list(y1 = 3, y2 = 4))`.

`yss_extra` (list of named `list()`)
List of additional information stored along with the results e.g. `list(list(timestamp_ys = Sys.time()), list(timestamp_ys = Sys.time()))`.

`extra` (named `list()`)
Deprecated argument for additional information stored along with the results. Use `yss_extra` instead.

Returns: (RushWorker)
Invisible self.

`RushWorker$fail_tasks()`: Move running tasks to failed and optionally save the condition objects.

Usage:

```
RushWorker$fail_tasks(keys, conditions = NULL)
```

Arguments:

`keys` (`character()`)
Keys of the running tasks to be moved.

`conditions` (`list()`)
List conditions e.g. `list(simpleError("Error"), simpleError("Error"))`. Defaults to `list(message = "Task failed")`.

Returns: (RushWorker)
Invisible self.

`RushWorker$set_terminated()`: Mark the worker as terminated. Last step in the worker loop before the worker terminates.

Usage:

```
RushWorker$set_terminated()
```

Returns: (RushWorker)
Invisible self.

`RushWorker$clone()`: The objects of this class are cloneable with this method.

Usage:

```
RushWorker$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

`start_worker` *Start a worker*

Description

Starts a worker. The function loads packages, initializes the RushWorker instance and invokes the worker loop. This function is called by `$start_local_workers()` or by the user after creating the worker script with `$worker_script()`.

Usage

```
start_worker(
  worker_id = NULL,
  network_id,
  config = NULL,
  profile = NULL,
  lgr_thresholds = NULL,
  lgr_buffer_size = 0,
  heartbeat_period = NULL,
  heartbeat_expire = NULL,
  message_log = NULL,
  output_log = NULL,
  restarted_from = NULL
)
```

Arguments

worker_id	(character(1)) Identifier of the worker. Keys in redis specific to the worker are prefixed with the worker id.
network_id	(character(1)) Identifier of the rush network. Manager and workers must have the same id. Keys in Redis are prefixed with the instance id.
config	(list()) Configuration for the Redis connection.
profile	(character(1)) Name of the mirai compute profile the worker runs on. If NULL, the worker runs on the default compute profile.
lgr_thresholds	(named character() | named numeric()) Logger threshold on the workers e.g. c("mlr3/rush" = "debug").
lgr_buffer_size	(integer(1)) By default (lgr_buffer_size = 0), the log messages are directly saved in the Redis data store. If lgr_buffer_size > 0, the log messages are buffered and saved in the Redis data store when the buffer is full. This improves the performance of the logging.
heartbeat_period	(integer(1)) Period of the heartbeat in seconds. The heartbeat is updated every heartbeat_period seconds. Must be at least 1 second.
heartbeat_expire	(integer(1)) Time to live of the heartbeat in seconds. The heartbeat key is set to expire after heartbeat_expire seconds. Must be at least heartbeat_period, otherwise a live worker is reaped as lost between two heartbeats. Set it larger than the longest pause a worker may experience, for example from garbage collection or

swapping, because a live worker wrongly declared lost can leave a task in an inconsistent state.

`message_log` (`character(1)`)
Path to the message log files e.g. `/tmp/message_logs/` The message log files are named `message_<worker_id>.log`. If `NULL`, no messages, warnings or errors are stored.

`output_log` (`character(1)`)
Path to the output log files e.g. `/tmp/output_logs/` The output log files are named `output_<worker_id>.log`. If `NULL`, no output is stored.

`restarted_from` (`character(1)`)
Id of the lost worker that this worker replaces. `NULL` if the worker is not a restart.

Details

The compute profile of the worker is passed to the worker loop when the worker loop has a `profile` argument.

Value

`NULL`

Examples

```
# This example is not executed since Redis must be installed
## Not run:
  rush::start_worker(
    network_id = "test-rush",
    config = list(host = "127.0.0.1", port = "6379"))

## End(Not run)
```

`store_large_object` *Store Large Objects*

Description

Store large objects to disk and return a reference to the object. The written `.rds` files are not cleaned up automatically. The caller is responsible for removing them, e.g. by storing them below `tempdir()` or by calling `unlink()` on the returned `path`.

Usage

```
store_large_object(obj, path)
```

Arguments

obj	(any) Object to store.
path	(character(1)) Path to an existing directory to store the object in.

Value

`list()` of class `"rush_large_object"` with the path of the stored object.

Examples

```
obj = list(a = 1, b = 2)
rush_large_object = store_large_object(obj, tempdir())
```

Index

AppenderRedis, 3, *3*

data.table::data.table(), *9*

lgr::Appender, *3*
lgr::AppenderMemory, *3*
lgr::Filterable, *3*
lgr::Layout, *4*

mirai::cluster_config(), *12*, *25*
mirai::daemons(), *7*, *12*, *25*
mirai::launch_remote(), *12*, *25*
mirai::mirai, *8*
mirai::remote_config(), *12*, *25*
mirai::ssh_config(), *12*, *25*

processx::process, *8*

R6, *3*, *10*, *27*
R6::R6Class, *3*, *6*, *26*
redux::redis_api, *8*
redux::redis_config, *2*, *4*, *5*, *8*, *10*, *23*, *24*, *27*
remove_rush_plan, 5
rsh, 5
rsh(), *6*
Rush, *5*, 6, *6*, *24–26*
rush *(*rush-package*)*, 2
rush-package, 2
rush_available, 23
rush_config, 23
rush_plan, 24
rush_plan(), *2*, *3*, *5*, *10*, *15*, *23*, *27*
RushWorker, *3*, *6–8*, *11–13*, 26, *26*, *29*

start_worker, 29
store_large_object, 31

tempdir(), *25*, *31*

unlink(), *31*