

Package ‘rush’

April 4, 2026

Title Rapid Asynchronous and Distributed Computing

Version 1.0.1

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 (>= 3.1.0)

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

Config/testthat/edition 3

Config/testthat/parallel false

Encoding UTF-8

RoxygenNote 7.3.3

NeedsCompilation yes

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

Maintainer Marc Becker <marcbecker@posteo.de>

Repository CRAN

Date/Publication 2026-04-04 09:30:02 UTC

Contents

rush-package	2
AppenderRedis	3
filter_custom_fields	4
remove_rush_plan	5
rsh	5
Rush	6
RushWorker	22
rush_available	23
rush_config	24
rush_plan	25
start_worker	26
store_large_object	28
Index	29

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.

Author(s)

Maintainer: 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 data base. 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 data bases.

Super classes

[lgr::Filterable](#) -> [lgr::Appender](#) -> [lgr::AppenderMemory](#) -> [AppenderRedis](#)

Methods

Public methods:

- [AppenderRedis\\$new\(\)](#)
- [AppenderRedis\\$flush\(\)](#)

Method [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"),
  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.

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.

Method 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
}
```

filter_custom_fields *Filter Custom Fields*

Description

Filters custom fields from log events.

Usage

```
filter_custom_fields(event)
```

Arguments

event [\(lgr::LogEvent\)](#)
 Log event.

remove_rush_plan	<i>Remove Rush Plan</i>
------------------	-------------------------

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	<i>Syntactic Sugar for Rush Manager Construction</i>
-----	--

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.
...	(ignored).

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.

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.

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 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.

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.

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_remote_workers()`.

Active bindings

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

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

`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.

`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.

`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$pop_task()`
- `Rush$finish_tasks()`
- `Rush$fail_tasks()`
- `Rush$push_tasks()`
- `Rush$push_running_tasks()`
- `Rush$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$tasks_with_state()`
- `Rush$push_results()`
- `Rush$push_failed()`
- `Rush$clone()`

Method `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.

Method `format()`: Helper for print outputs.

Usage:

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

Arguments:

... (ignored).

Returns: (`character()`).

Method `print()`: Print method.

Usage:

```
Rush$print()
```

Returns: (`character()`).

Method `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()
```

Method `start_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_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.

Method start_local_workers(): Start workers locally with processx. Initializes a [Rush-Worker](#) 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.

Method `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.

Method `worker_script()`: Generate a script to start workers. Run this script `n` times to start `n` workers.

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.

`heartbeat_expire` (integer(1))

Time to live of the heartbeat in seconds. The heartbeat key is set to expire after `heartbeat_expire` seconds.

`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.

Method `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 = Inf)
```

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. Default is Inf.

Method `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. If "terminate" the workers evaluate the currently running task and then terminate. The "terminate" option must be implemented in the worker loop.
`worker_ids` (character())
 Worker ids to be stopped. If NULL all workers are stopped.

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

Usage:

```
Rush$detect_lost_workers()
```

Returns: (character())

Worker ids of detected lost workers.

Method `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.

Method `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 level, timestamp, logger, caller and message, and optionally time difference.

Method `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.

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

Usage:

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

Arguments:

`timeout` (numeric(1))

Time to wait for task in seconds.

`fields` (character())

Fields to be returned.

Method `finish_tasks()`: Save output of tasks and mark them as finished.

Usage:

```
Rush$finish_tasks(keys, yss, extra = NULL)
```

Arguments:

`keys` (character(1))

Keys of the associated tasks.

`yss` (named list())

List of lists of named results.

`extra` (named list())

List of lists of additional information stored along with the results.

Returns: (Rush)

Invisible self.

Method `fail_tasks()`: Mark tasks as failed and optionally save the condition objects

Usage:

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

Arguments:

`keys` (character())

Keys of the tasks to be moved. Defaults to all queued tasks.

`conditions` (named list())

List of lists of conditions. Defaults to `list(message = "Failed")`.

Returns: (Rush)

Invisible self.

Method `push_tasks()`: Create queued tasks and add them to the queue.

Usage:

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

Arguments:

`xss` (list of named `list()`)

Lists of arguments for the function e.g. `list(list(x1, x2), list(x1, x2))`.

`extra` (`list()`)

List of additional information stored along with the task e.g. `list(list(timestamp), list(timestamp))`.

Returns: (`character()`)

Keys of the tasks.

Method `push_running_tasks()`: Create running tasks.

Usage:

`Rush$push_running_tasks(xss, extra = NULL)`

Arguments:

`xss` (list of named `list()`)

Lists of arguments for the function e.g. `list(list(x1, x2), list(x1, x2))`.

`extra` (`list`)

List of additional information stored along with the task e.g. `list(list(timestamp), list(timestamp))`.

Returns: (`character()`)

Keys of the tasks.

Method `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, x2), list(x1, x2))`.

`yss` (list of named `list()`)

Lists of results for the function e.g. `list(list(y1, y2), list(y1, y2))`.

`xss_extra` (`list`)

List of additional information stored along with the task e.g. `list(list(timestamp), list(timestamp))`.

`yss_extra` (`list`)

List of additional information stored along with the results e.g. `list(list(timestamp), list(timestamp))`.

Returns: (`character()`)

Keys of the tasks.

Method `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)`

Arguments:

`xss` (list of named `list()`)

Lists of arguments for the function e.g. `list(list(x1, x2), list(x1, x2))`.

`xss_extra` (`list`)

List of additional information stored along with the task e.g. `list(list(timestamp), list(timestamp))`.

conditions (named list())
List of lists of conditions.

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

Method empty_queue(): Remove all tasks from the queue. The state of the tasks is set to failed.

Usage:

```
Rush$empty_queue(keys = NULL, conditions = NULL)
```

Arguments:

keys (character())
Keys of the tasks to be moved. Defaults to all queued tasks.
conditions (named list())
List of lists of conditions.

Returns: (Rush)
Invisible self.

Method fetch_tasks(): Fetch all tasks from the data base.

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", "xs_extra", "worker_id", "ys",
"ys_extra", "condition").

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

Method fetch_queued_tasks(): Fetch queued tasks from the data base.

Usage:

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

Arguments:

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

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

Method fetch_running_tasks(): Fetch running tasks from the data base.

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.

Method `fetch_failed_tasks()`: Fetch failed tasks from the data base.

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.

Method `fetch_finished_tasks()`: Fetch finished tasks from the data base. 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("xs", "xs_extra", "worker_id", "ys", "ys_extra")`.

Returns: `data.table()`

Table of finished tasks.

Method `fetch_tasks_with_state()`: Fetch tasks with different states from the data base. 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")`.

`states` (`character()`)

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

Method `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.

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.

timeout (numeric(1))

Time to wait for new results in seconds. Defaults to 0 (no waiting).

Returns: data.table()

Table of latest results.

Method reset_cache(): Reset the cache of the finished tasks.

Usage:

```
Rush$reset_cache()
```

Returns: (Rush)

Invisible self.

Method 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.

Method 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 = ... creates two hashes with the fields xs and ys. The vectors are recycled to the length of the longest vector. 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.

Method 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))`.

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.

Method 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.

Method is_running_task(): Checks whether tasks have the status "running".

Usage:

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

Arguments:

keys (character())

Keys of the tasks.

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

Usage:

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

Arguments:

keys (character())

Keys of the tasks.

Method `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()).

Method `push_results()`: Deprecated method. Use `$finish_tasks()` instead.

Usage:

```
Rush$push_results(keys, yss, extra = NULL)
```

Arguments:

keys (character())

Keys of the associated tasks.

yss (named list())

List of lists of named results.

extra (named list())

List of lists of additional information stored along with the results.

Returns: (Rush)

Invisible self.

Method `push_failed()`: Deprecated method. Use `$fail_tasks()` instead.

Usage:

```
Rush$push_failed(keys, conditions)
```

Arguments:

keys (character())

Keys of the associated tasks.

conditions (list())

List of conditions.

Returns: (Rush)

Invisible self.

Method `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
}

```

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.

Value

Object of class [R6::R6Class](#) and [RushWorker](#).

Super class

[rush::Rush](#) -> [RushWorker](#)

Public fields

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

`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.

Methods**Public methods:**

- [RushWorker\\$new\(\)](#)
- [RushWorker\\$set_terminated\(\)](#)
- [RushWorker\\$clone\(\)](#)

Method `new()`: Creates a new instance of this [R6](#) class.

Usage:

```
RushWorker$new(
  network_id,
  config = NULL,
  worker_id = NULL,
  heartbeat_period = NULL,
  heartbeat_expire = 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.

`heartbeat_period` (integer(1))

Period of the heartbeat in seconds. The heartbeat is updated every `heartbeat_period` seconds.

`heartbeat_expire` (integer(1))

Time to live of the heartbeat in seconds. The heartbeat key is set to expire after `heartbeat_expire` seconds.

Method `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.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

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	<i>Create Rush Plan</i>
-----------	-------------------------

Description

Stores the number of workers and Redis configuration options ([redux::redis_config](#)) for **Rush**. 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"
)

```

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.
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.

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
}

```

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 `$start_local_workers()` or by the user after creating the worker script with `$create_worker_script()`.

Usage

```

start_worker(
  worker_id = NULL,
  network_id,
  config = NULL,
  lgr_thresholds = NULL,
  lgr_buffer_size = 0,
  heartbeat_period = NULL,
  heartbeat_expire = NULL,
  message_log = NULL,
  output_log = 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.
lgr_thresholds	(named character() named numeric()) Logger threshold on the workers e.g. <code>c("mlr3/rush" = "debug")</code> .

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.
heartbeat_expire	(integer(1)) Time to live of the heartbeat in seconds. The heartbeat key is set to expire after heartbeat_expire seconds.
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.

Value

NULL

Note

The function initializes the connection to the Redis data base. It loads the packages required by the worker loop. The function initialize the [RushWorker](#) instance and starts the worker loop.

Examples

```
# This example is not executed since Redis must be installed
## Not run:
rush::start_worker(
  network_id = 'test-rush',
  url = 'redis://127.0.0.1:6379',
  scheme = 'redis',
  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.

Usage

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

Arguments

obj	(any) Object to store.
path	(character(1)) Path to store the object.

Value

list() of class "rush_large_object" with the name and 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()`, [8](#)

`filter_custom_fields`, [4](#)

`lgr::Appender`, [3](#)
`lgr::AppenderMemory`, [3](#)
`lgr::Filterable`, [3](#)
`lgr::Layout`, [3](#)
`lgr::LogEvent`, [4](#)

`mirai::daemons()`, [7](#)
`mirai::mirai`, [7](#)

`processx::process`, [7](#)

`R6`, [3, 10, 22](#)
`R6::R6Class`, [3, 6, 22](#)
`redux::redis_api`, [8](#)
`redux::redis_config`, [3, 5, 8, 10, 23, 25](#)
`remove_rush_plan`, [5](#)
`rsh`, [5](#)
`rsh()`, [6](#)
`Rush`, [5, 6, 6, 22, 25](#)
`rush (rush-package)`, [2](#)
`rush-package`, [2](#)
`rush::Rush`, [22](#)
`rush_available`, [23](#)
`rush_config`, [24](#)
`rush_plan`, [25](#)
`rush_plan()`, [3, 5, 10, 23, 24](#)
`RushWorker`, [3, 6, 10–12, 22, 22, 26, 27](#)

`start_worker`, [26](#)
`store_large_object`, [28](#)