Package 'rush'

June 20, 2024

Title Rapid Parallel and Distributed Computing

Version 0.1.0

Description Parallel computing with a network of local and remote workers. Fast exchange of results between the workers through a 'Redis' database. Key features include task queues, local caching, and sophisticated error handling.

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

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

Depends R (>= 3.1.0)

Imports checkmate, data.table, jsonlite, lgr, mlr3misc, parallel, processx, redux, uuid

Suggests callr, knitr, rmarkdown, testthat (>= 3.0.0)

Config/testthat/edition 3

Config/testthat/parallel false

Encoding UTF-8

RoxygenNote 7.3.1

License MIT + file LICENSE

NeedsCompilation no

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

Maintainer Marc Becker <marcbecker@posteo.de>

Repository CRAN

Date/Publication 2024-06-20 15:40:06 UTC

Contents

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

Description

Parallel computing with a network of local and remote workers. Fast exchange of results between the workers through a 'Redis' database. Key features include task queues, local caching, and sophisticated error handling.

Author(s)

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

See Also

Useful links:

- 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(),
  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

```
# This example is not executed since Redis must be installed

  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

```
# This example is not executed since Redis must be installed

  config_local = redux::redis_config()
  rush_plan(config = config_local, n_workers = 2)
  remove_rush_plan()
```

rsh *Synctatic Sugar for Rush Controller Construction*

Description

Function to construct a Rush controller.

Usage

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

Arguments

network_id	(character(1)) Identifier of the rush network. Controller and workers must have the same instance 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 controller.

Examples

```
# This example is not executed since Redis must be installed

  config_local = redux::redis_config()
  rush = rsh(network_id = "test_network", config = config_local)
  rush
```

Rush *Rush Controller*

Description

Rush is the controller in a centralized rush network. The controller starts and stops the workers, pushes tasks to the workers and fetches results.

Value

Object of class R6::R6Class and Rush with controller methods.

Local Workers

A local worker runs on the same machine as the controller. Local workers are spawned with the `$start_local_workers()` method via the processx‘ package.

Remote Workers

A remote worker runs on a different machine than the controller. Remote workers are started manually with the `$create_worker_script()` and `$start_remote_workers()` methods. Remote workers can be started on any system as long as the system has access to Redis and all required packages are installed. Only a heartbeat process can kill remote workers. The heartbeat process also monitors the remote workers for crashes.

Stopping Workers

Local and remote workers can be terminated with the `$stop_workers(type = "terminate")` method. The workers evaluate the currently running task and then terminate. The option `type = "kill"` stops the workers immediately. Killing a local worker is done with the `processx` package. Remote workers are killed by pushing a kill signal to the heartbeat process. Without a heartbeat process a remote worker cannot be killed (see section heartbeat).

Heartbeat

The heartbeat process periodically signals that a worker is still alive. This is implemented by setting a timeout on the heartbeat key. Furthermore, the heartbeat process can kill the worker.

Data Structure

Tasks are stored in Redis hashes. Hashes are collections of field-value pairs. The key of the hash identifies the task in Redis and `rush`.

```
key : xs | ys | xs_extra
```

The field-value pairs are written by different methods, e.g. `$push_tasks()` writes `xs` and `$push_results()` writes `ys`. The values of the fields are serialized lists or atomic values e.g. unserializing `xs` gives `list(x1 = 1, x2 = 2)` This data structure allows quick converting of a hash into a row and joining multiple hashes into a table.

```
| key | x1 | x2 | y | timestamp |
| 1.. |  3 |  4 | 7 |  12:04:11 |
| 2.. |  1 |  4 | 5 |  12:04:12 |
| 3.. |  1 |  1 | 2 |  12:04:13 |
```

When the value of a field is a named list, the field can store the cells of multiple columns of the table. When the value of a field is an atomic value, the field stores a single cell of a column named after the field. The methods `$push_tasks()` and `$push_results()` write into multiple hashes. For example, `$push_tasks(xss = list(list(x1 = 1, x2 = 2), list(x1 = 2, x2 = 2))` writes `xs` in two hashes.

Task States

A task can go through four states `"queued"`, `"running"`, `"finished"` or `"failed"`. Internally, the keys of the tasks are pushed through Redis lists and sets to keep track of their state. Queued tasks are waiting to be evaluated. A worker pops a task from the queue and changes the state to `"running"` while evaluating the task. When the task is finished, the state is changed to `"finished"` and the result is written to the

Queues

Rush uses a shared queue and a queue for each worker. The shared queue is used to push tasks to the workers. The first worker that pops a task from the shared queue evaluates the task. The worker queues are used to push tasks to specific workers.

Fetch Tasks and Results

The `$fetch_*()` methods retrieve data from the Redis database. A matching method is defined for each task state e.g. `$fetch_running_tasks()` and `$fetch_finished_tasks()`. The methods `$fetch_new_tasks()` and `$fetch_finished_tasks()` cache the already queried data. The `$wait_for_finished_tasks()` variant wait until a new result is available.

Error Handling

When evaluating tasks in a distributed system, many things can go wrong. Simple R errors in the worker loop are caught and written to the archive. The task is marked as `"failed"`. If the connection to a worker is lost, it looks like a task is `"running"` forever. The method `$detect_lost_workers()` identifies lost workers. Running this method periodically adds a small overhead.

Logging

The worker logs all messages written with the `lgr` package to the data base. The `lgr_thresholds` argument defines the logging level for each logger e.g. `c(rush = "debug")`. Saving log messages adds a small overhead but is useful for debugging. By default, no log messages are stored.

Seed

Setting a seed is important for reproducibility. The tasks can be evaluated with a specific L'Ecuyer-CMRG seed. If an initial seed is passed, the seed is used to generate L'Ecuyer-CMRG seeds for each task. Each task is then evaluated with a separate RNG stream. See parallel::nextRNGStream for more details.

Public fields

`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.

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

Active bindings

`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.

`n_killed_workers (integer(1))`
Number of killed workers.

`n_lost_workers (integer(1))`
Number of lost workers. Run `$detect_lost_workers()` to update the number of lost workers.

`n_pre_workers (integer(1))`
Number of workers that are not yet completely started.

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

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

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

`killed_worker_ids (character())`
Ids of killed workers.

`lost_worker_ids (character())`
Ids of lost workers.

`pre_worker_ids (character())`
Ids of workers that are not yet completely started.

`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_queued_priority_tasks (integer(1))`
Number of queued priority 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.

`worker_states` (`data.table::data.table()`)
Contains the states of the workers.

`all_workers_terminated` (`logical(1)`)
Whether all workers are terminated.

`all_workers_lost` (`logical(1)`)
Whether all workers are lost. Runs `$detect_lost_workers()` to detect lost workers.

`priority_info` (data.table::data.table)
Contains the number of tasks in the priority queues.

`snapshot_schedule` (`character()`)
Set a snapshot schedule to periodically save the data base on disk. For example, `c(60, 1000)` saves the data base every 60 seconds if there are at least 1000 changes. Overwrites the redis configuration file. Set to `NULL` to disable snapshots. For more details see redis.io.

`redis_info` (`list()`)
Information about the Redis server.

Methods

Public methods:

- `Rush$new()`
- `Rush$format()`
- `Rush$print()`
- `Rush$start_local_workers()`
- `Rush$restart_local_workers()`
- `Rush$create_worker_script()`
- `Rush$start_remote_workers()`
- `Rush$wait_for_workers()`
- `Rush$stop_workers()`
- `Rush$detect_lost_workers()`
- `Rush$reset()`
- `Rush$read_log()`
- `Rush$print_log()`
- `Rush$push_tasks()`
- `Rush$push_priority_tasks()`
- `Rush$push_failed()`
- `Rush$retry_tasks()`

- Rush$fetch_queued_tasks()
- Rush$fetch_priority_tasks()
- Rush$fetch_running_tasks()
- Rush$fetch_finished_tasks()
- Rush$wait_for_finished_tasks()
- Rush$fetch_new_tasks()
- Rush$wait_for_new_tasks()
- Rush$fetch_failed_tasks()
- Rush$fetch_tasks()
- Rush$fetch_tasks_with_state()
- 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()

Method `new()`**:** Creates a new instance of this R6 class.

Usage:

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

Arguments:

`network_id` (`character(1)`)
: Identifier of the rush network. Controller and workers must have the same instance 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.

`seed` (`integer()`)
: Initial seed for the random number generator. Either a L'Ecuyer-CMRG seed (`integer(7)`) or a regular RNG seed (`integer(1)`). The later is converted to a L'Ecuyer-CMRG seed. If `NULL`, no seed is used for the random number generator.

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

Usage:

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

Arguments:

`...` (ignored).

Returns: (`character()`).

Method `print()`**:** Print method.

Usage:

```
Rush$print()
```

Returns: (`character()`).

Method `start_local_workers()`**:** Start workers locally with `processx`. The processx::process are stored in `$processes`. Alternatively, use `$create_worker_script()` to create a script for starting workers on remote machines. By default, `worker_loop_default()` is used as worker loop. This function takes the arguments `fun` and optionally `constants` which are passed in `....`

Usage:

```
Rush$start_local_workers(
  n_workers = NULL,
  wait_for_workers = TRUE,
  timeout = Inf,
  globals = NULL,
  packages = NULL,
  heartbeat_period = NULL,
  heartbeat_expire = NULL,
  lgr_thresholds = NULL,
  lgr_buffer_size = 0,
  supervise = TRUE,
  worker_loop = worker_loop_default,
  ...
)
```

Arguments:

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

`wait_for_workers` (`logical(1)`)
: Whether to wait until all workers are available.

`timeout` (`numeric(1)`)
: Timeout to wait for workers in seconds.

`globals` (`character()`)
: Global variables to be loaded to the workers global environment.

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

`heartbeat_period` (`integer(1)`)
: Period of the heartbeat in seconds.

`heartbeat_expire` (`integer(1)`)
: Time to live of the heartbeat in seconds.

`lgr_thresholds` (named `character()` | named `numeric()`)
: Logger threshold on the workers e.g. `c(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.

`worker_loop` (`function`)
Loop run on the workers. Defaults to worker_loop_default which is called with `fun`. Pass `fun` in `...`. Use worker_loop_callr to run `fun` in an external callr session.

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

Method `restart_local_workers()`: Restart local workers. If the worker is is still running, it is killed and restarted.

Usage:

```
Rush$restart_local_workers(worker_ids, supervise = TRUE)
```

Arguments:

`worker_ids` (`character()`)
Worker ids to be restarted.

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

Method `create_worker_script()`: Create script to remote start workers. Run these command to pre-start a worker. The worker will wait until the start arguments are pushed with `$start_remote_workers()`.

Usage:

```
Rush$create_worker_script()
```

Method `start_remote_workers()`: Push start arguments to remote workers. Remote workers must be pre-started with `$create_worker_script()`.

Usage:

```
Rush$start_remote_workers(
  globals = NULL,
  packages = NULL,
  heartbeat_period = NULL,
  heartbeat_expire = NULL,
  lgr_thresholds = NULL,
  lgr_buffer_size = 0,
  worker_loop = worker_loop_default,
  ...
)
```

Arguments:

`globals` (`character()`)
Global variables to be loaded to the workers global environment.

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

`heartbeat_period` (`integer(1)`)
Period of the heartbeat in seconds.

`heartbeat_expire` (`integer(1)`)
Time to live of the heartbeat in seconds.

`lgr_thresholds` (named `character()` | named `numeric()`)
Logger threshold on the workers e.g. `c(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.

`worker_loop` (`function`)
: Loop run on the workers. Defaults to worker_loop_default which is called with `fun`. Pass `fun` in `....` Use worker_loop_callr to run `fun` in an external callr session.

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

Method `wait_for_workers()`**:** Wait until `n` workers are available.

Usage:

```
Rush$wait_for_workers(n, timeout = Inf)
```

Arguments:

`n` (`integer(1)`)
: Number of workers to wait for.

`timeout` (`numeric(1)`)
: Timeout in seconds. Default is `Inf`.

Method `stop_workers()`**:** Stop workers.

Usage:

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

Arguments:

`type` (`character(1)`)
: Type of stopping. Either `"terminate"` or `"kill"`. If `"terminate"` the workers evaluate the currently running task and then terminate. If `"kill"` the workers are stopped immediately.

`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 `"lost"`. Local workers without a heartbeat are checked by their process id. Checking local workers on unix systems only takes a few microseconds per worker. But checking local workers on windows might be very slow. Workers with a heartbeat process are checked with the heartbeat. Lost tasks are marked as `"lost"`.

Usage:

```
Rush$detect_lost_workers(restart_local_workers = FALSE)
```

Arguments:

`restart_local_workers` (`logical(1)`)
: Whether to restart lost workers.

Method `reset()`**:** Stop workers and delete data stored in redis.

Usage:

```
Rush$reset(type = "kill")
```

Arguments:

type (character(1))
Type of stopping. Either "terminate" or "kill". If "terminate" the workers evaluate the currently running task and then terminate. If "kill" the workers are stopped immediately.

Method read_log(): Read log messages written with the lgr package from a worker.

Usage:

```
Rush$read_log(worker_ids = NULL)
```

Arguments:

worker_ids (character(1))
Worker ids. If NULL all worker ids are used.

Method print_log(): Print log messages written with the lgr package from a worker.

Usage:

```
Rush$print_log()
```

Method push_tasks(): Pushes a task to the queue. Task is added to queued tasks.

Usage:

```
Rush$push_tasks(
  xss,
  extra = NULL,
  seeds = NULL,
  timeouts = NULL,
  max_retries = NULL,
  terminate_workers = FALSE
)
```

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))).

seeds (list())
List of L'Ecuyer-CMRG seeds for each task e.g list(list(c(104071, 490840688, 1690070564, -495119766, 503491950, 1801530932, -1629447803))). If NULL but an initial seed is set, L'Ecuyer-CMRG seeds are generated from the initial seed. If NULL and no initial seed is set, no seeds are used for the random number generator.

timeouts (integer())
Timeouts for each task in seconds e.g. c(10, 15). A single number is used as the timeout for all tasks. If NULL no timeout is set.

max_retries (integer())
Number of retries for each task. A single number is used as the number of retries for all tasks. If NULL tasks are not retried.

terminate_workers (logical(1))
Whether to stop the workers after evaluating the tasks.

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

Method `push_priority_tasks()`**:** Pushes a task to the queue of a specific worker. Task is added to queued priority tasks. A worker evaluates the tasks in the priority queue before the shared queue. If `priority` is `NA` the task is added to the shared queue. If the worker is lost or worker id is not known, the task is added to the shared queue.

Usage:

```
Rush$push_priority_tasks(xss, extra = NULL, priority = 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))`.

`priority` (`character()`)
Worker ids to which the tasks should be pushed.

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

Method `push_failed()`**:** Pushes failed tasks to the data base.

Usage:

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

Arguments:

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

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

Method `retry_tasks()`**:** Retry failed tasks.

Usage:

```
Rush$retry_tasks(keys, ignore_max_retries = FALSE, next_seed = FALSE)
```

Arguments:

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

`ignore_max_retries` (`logical(1)`)
Whether to ignore the maximum number of retries.

`next_seed` (`logical(1)`)
Whether to change the seed of the task.

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

Usage:

```
Rush$fetch_queued_tasks(
  fields = c("xs", "xs_extra"),
  data_format = "data.table"
)
```

Arguments:

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

`data_format` (`character()`)
Returned data format. Choose `"data.table"` or "list". The default is `"data.table"` but `"list"` is easier when list columns are present.

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

Method `fetch_priority_tasks()`**:** Fetch queued priority tasks from the data base.

Usage:

```
Rush$fetch_priority_tasks(
  fields = c("xs", "xs_extra"),
  data_format = "data.table"
)
```

Arguments:

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

`data_format` (`character()`)
Returned data format. Choose `"data.table"` or "list". The default is `"data.table"` but `"list"` is easier when list columns are present.

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

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

Usage:

```
Rush$fetch_running_tasks(
  fields = c("xs", "xs_extra", "worker_extra"),
  data_format = "data.table"
)
```

Arguments:

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

`data_format` (`character()`)
Returned data format. Choose `"data.table"` or "list". The default is `"data.table"` but `"list"` is easier when list columns are present.

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

Method `fetch_finished_tasks()`**:** Fetch finished tasks from the data base. Finished tasks are cached.

Usage:

```
Rush$fetch_finished_tasks(
  fields = c("xs", "ys", "xs_extra", "worker_extra", "ys_extra", "condition"),
  reset_cache = FALSE,
  data_format = "data.table"
)
```

Arguments:

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

`reset_cache` (`logical(1)`)
Whether to reset the cache.

`data_format` (`character()`)
Returned data format. Choose `"data.table"` or "list". The default is `"data.table"` but `"list"` is easier when list columns are present.

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

Method `wait_for_finished_tasks()`**:** Block process until a new finished task is available. Returns all finished tasks or `NULL` if no new task is available after `timeout` seconds.

Usage:

```
Rush$wait_for_finished_tasks(
  fields = c("xs", "ys", "xs_extra", "worker_extra", "ys_extra"),
  timeout = Inf,
  data_format = "data.table"
)
```

Arguments:

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

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

`data_format` (`character()`)
Returned data format. Choose `"data.table"` or "list". The default is `"data.table"` but `"list"` is easier when list columns are present.

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

Method `fetch_new_tasks()`**:** Fetch finished tasks from the data base that finished after the last fetch. Updates the cache of the finished tasks.

Usage:

```
Rush$fetch_new_tasks(
  fields = c("xs", "ys", "xs_extra", "worker_extra", "ys_extra", "condition"),
  data_format = "data.table"
)
```

Arguments:

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

`data_format` (`character()`)
Returned data format. Choose `"data.table"` or "list". The default is `"data.table"` but `"list"` is easier when list columns are present.

Returns: `data.table()`
Latest results.

Method `wait_for_new_tasks()`**:** Block process until a new finished task is available. Returns new tasks or `NULL` if no new task is available after `timeout` seconds.

Usage:

```
Rush$wait_for_new_tasks(
  fields = c("xs", "ys", "xs_extra", "worker_extra", "ys_extra", "condition"),
  timeout = Inf,
  data_format = "data.table"
)
```

Arguments:

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

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

`data_format` (`character()`)
Returned data format. Choose `"data.table"` or "list". The default is `"data.table"` but `"list"` is easier when list columns are present.

Returns: `data.table()` | `list()`.

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

Usage:

```
Rush$fetch_failed_tasks(
  fields = c("xs", "worker_extra", "condition"),
  data_format = "data.table"
)
```

Arguments:

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

`data_format` (`character()`)
Returned data format. Choose `"data.table"` or "list". The default is `"data.table"` but `"list"` is easier when list columns are present.

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

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

Usage:

```
Rush$fetch_tasks(
  fields = c("xs", "ys", "xs_extra", "worker_extra", "ys_extra", "condition"),
  data_format = "data.table"
)
```

Arguments:

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

`data_format` (`character()`)
Returned data format. Choose `"data.table"` or "list". The default is `"data.table"` but `"list"` is easier when list columns are present.

Returns: `data.table()`
Table of all 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("xs", "ys", "xs_extra", "worker_extra", "ys_extra", "condition"),
  states = c("queued", "running", "finished", "failed"),
  reset_cache = FALSE,
  data_format = "data.table"
)
```

Arguments:

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

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

`reset_cache` (`logical(1)`)
Whether to reset the cache of the finished tasks.

`data_format` (`character()`)
Returned data format. Choose `"data.table"` or "list". The default is `"data.table"` but `"list"` is easier when list columns are present.

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 `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

```
# This example is not executed since Redis must be installed

config_local = redux::redis_config()
rush = rsh(network_id = "test_network", config = config_local)
rush
```

RushWorker *Rush Worker*

Description

RushWorker evaluates tasks and writes results to the data base. The worker inherits from Rush.

Value

Object of class R6::R6Class and RushWorker with worker methods.

Super class

rush::Rush -> RushWorker

Public fields

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

remote (logical(1))
: Whether the worker is on a remote machine.

heartbeat ('r_process")
: 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.

terminated_on_idle (logical(1))
: Whether to shutdown the worker if no tasks are queued. Used in the worker loop to determine whether to continue.

Methods

Public methods:

- RushWorker$new()
- RushWorker$push_running_tasks()
- RushWorker$pop_task()
- RushWorker$push_results()
- RushWorker$set_terminated()
- RushWorker$clone()

Method new(): Creates a new instance of this R6 class.

Usage:

```
RushWorker$new(
  network_id,
  config = NULL,
  remote,
  worker_id = NULL,
  heartbeat_period = NULL,
  heartbeat_expire = NULL,
  lgr_thresholds = NULL,
  lgr_buffer_size = 0,
  seed = NULL
)
```

Arguments:

`network_id` (`character(1)`)
: Identifier of the rush network. Controller and workers must have the same instance 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.

`remote` (`logical(1)`)
: Whether the worker is started on a remote machine. See Rush 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.

`heartbeat_expire` (`integer(1)`)
: Time to live of the heartbeat in seconds.

`lgr_thresholds` (named `character()` | named `numeric()`)
: Logger threshold on the workers e.g. `c(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.

`seed` (`integer()`)
: Initial seed for the random number generator. Either a L'Ecuyer-CMRG seed (`integer(7)`) or a regular RNG seed (`integer(1)`). The later is converted to a L'Ecuyer-CMRG seed. If `NULL`, no seed is used for the random number generator.

Method `push_running_tasks()`**:** Push a task to running tasks without queue.

Usage:

```
RushWorker$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 pop_task(): Pop a task from the queue. Task is moved to the running tasks.

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.

Method push_results(): Pushes results to the data base.

Usage:

```
RushWorker$push_results(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.

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

Usage:

```
RushWorker$set_terminated()
```

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.

Note

The worker registers itself in the data base of the rush network.

Examples

```
# This example is not executed since Redis must be installed

  config_local = redux::redis_config()
  rush = rsh(network_id = "test_network", config = config_local)

  fun = function(x1, x2, ...) list(y = x1 + x2)
```

```
rush$start_local_workers(fun = fun)

rush$stop_workers()
```

`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

```
# This example is not executed since Redis must be installed

  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

```
# This example is not executed since Redis must be installed

  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. The function tests the connection to Redis and throws an error if the connection fails.

Usage

```
rush_plan(
  n_workers = NULL,
  config = NULL,
  lgr_thresholds = NULL,
  large_objects_path = NULL,
  start_worker_timeout = Inf
)
```

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(rush = "debug")`.
`large_objects_path`	(`character(1)`) The path to the directory where large objects are stored.
`start_worker_timeout`	(`numeric(1)`) The time in seconds to wait for a worker to start.

Value

`list()` with the stored configuration.

Examples

```
# This example is not executed since Redis must be installed

   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 the globals and 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()`. Use with caution. The global environment is changed.

Usage

```
start_worker(network_id, worker_id = NULL, remote = TRUE, ...)
```

Arguments

`network_id`	(`character(1)`) Identifier of the rush network. Controller and workers must have the same instance id. Keys in Redis are prefixed with the instance id.
`worker_id`	(`character(1)`) Identifier of the worker. Keys in redis specific to the worker are prefixed with the worker id.
`remote`	(`logical(1)`) Whether the worker is on a remote machine.
`...`	(any) Arguments passed to redux::redis_config.

Value

`NULL`

Note

The function initializes the connection to the Redis data base. It loads the packages and copies the globals to the global environment of the worker. 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',
   remote = TRUE,
   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())
```

`worker_loop_callr` *Single Task Worker Loop with callr Encapsulation*

Description

Worker loop that pops a single task from the queue, executes the function in an external callr session and pushes the results. Supports timeouts on the tasks.

Usage

```
worker_loop_callr(fun, constants = NULL, rush)
```

Arguments

`fun`	(`function`) Function to be executed.
`constants`	(`list`) List of constants passed to `fun`.
`rush`	(RushWorker) Rush worker instance.

Value

`NULL`

Examples

```
# This example is not executed since Redis must be installed

   config_local = redux::redis_config()
   rush = rsh(network_id = "test_network", config = config_local)

   fun = function(x1, x2, ...) list(y = x1 + x2)
   rush$start_local_workers(
     fun = fun,
     worker_loop = worker_loop_callr)

   rush$stop_workers()
```

`worker_loop_default` *Single Task Worker Loop*

Description

Worker loop that pops a single task from the queue, executes the function and pushes the results.

Usage

```
worker_loop_default(fun, constants = NULL, rush)
```

Arguments

`fun`	(`function`) Function to be executed.
`constants`	(`list`) List of constants passed to `fun`.
`rush`	(RushWorker) Rush worker instance.

Value

`NULL`

Examples

```
# This example is not executed since Redis must be installed

   config_local = redux::redis_config()
   rush = rsh(network_id = "test_network", config = config_local)

   fun = function(x1, x2, ...) list(y = x1 + x2)
   rush$start_local_workers(
     fun = fun,
     worker_loop = worker_loop_default)

   rush$stop_workers()
```

Index