SpiNNaker Partitioning Server

Release 1!7.0.0-a5

unknown

Apr 26, 2023

CONTENTS

The SpiNNaker Partitioning Server is a Python program built on Rig designed to facilitate the sharing and partitioning of large SpiNNaker machines.

This server enables users to request SpiNNaker machines of various shapes and sizes. These requests are queued and allocated to available SpiNNaker machines, partitioning large SpiNNaker machines into smaller ones if possible. When faced with the numerous research problems of optimal packing and scheduling of allocations, this implementation uses the 'simplest mechanism that could possibly work'. With this said, the server intends to be 'production ready' in the sense that it is intended to be robust against real end users and routine maintenance requirements.

CHAPTER

ONE

SERVER MANAGEMENT AND CONFIGURATION

Essential reading for system administrators. The following documentation describes how to configure and manage a SpiNNaker Partitioning Server instance.

1.1 Server Management and Configuration

1.1.1 Installation and Requirements

The server and its dependencies can be installed from PyPI like so:

```
$ pip install spalloc_server
```

The server is currently only compatible with Linux due to its use of the `poll()` system call.

1.1.2 Operation

The SpiNNaker partitioning server is started like so:

```
$ spalloc_server configfile.cfg
```

The server is configured by a configuration file whose name is supplied as the first command-line argument. See the *section below* for an overview of the config file format. You can trigger the server to reread its config file by sending a `SIGHUP` signal to the server process. When you do this (and where it is possible), running jobs will continue to execute without interruption and queued jobs will automatically be enqueued on any newly added machines.

1.1.3 Command-line usage

There is one mandatory argument, the name of the file holding the configuration definitions, and a number of options that may be specified.

spalloc_server [OPTIONS] CONFIG_FILE [OPTIONS]

The options that may be given are:

--version Print the version of `spalloc_server` and quit.

--quiet Hide non-error output.

--cold-start Force a cold start, discarding any existing saved state.

--port PORT The network port that the service should listen on. Defaults to 22244. (The older style of setting the port through the configuration file is deprecated.)

1.1.4 Stopping the server

The server runs until it is terminated by `SIGINT`, i.e. pressing ctrl+c. When terminated the server attempts to gracefully shut-down, completing any outstanding board power-management commands and saving its state to disk. When the server is subsequently restarted, the saved state is restored and operation may continue as if the server had never been shut-down. Alternatively a cold-start may be enforced using the `--cold-start` argument when starting the server.

When the server is terminated, machines allocated to running jobs are left powered on meaning that user's jobs are not interrupted by the partitioning server being restarted. If it is necessary to perform major maintenance and fully shut-down the server and all controlled SpiNNaker machines, the recommended approach is to add the following line to the bottom of the server's config file:

```
configuration = Configuration()
```

This causes the server to believe all machines have been removed resulting in all queued and running jobs being cancelled and all previously allocated SpiNNaker boards being powered-down. To maximise the chances of all clients realising their jobs have been cancelled, the server should then be left running for a few minutes before being finally shut down.

1.1.5 Configuration file format

A configuration file is used to describe the machines which are to be managed. Configuration files are Python scripts which define a global `configuration` variable which is an instance of the *Configuration* class.

Note: Everything in *spalloc_server.configuration* and *spalloc_server.coordinates* modules is implicitly imported into the namespace of the config file.

A minimal (though useless) configuration file would look like so:

```
configuration = Configuration()
```

This causes the server to listen on all interfaces on the default port but does not define any machines for the server to manage. As a result, this server will cancel all jobs sent to it for lack of a suitable machine. See the *Configuration* class for a description of all available configuration options and default values.

Machines are defined using *Machine* objects. These specify the dimensions, broken boards and links, physical layout and IP addresses of a SpiNNaker machine. All machines are presumed to be interconnected in a valid hexagonal torus topology constructed from a rectangular array of triads of boards. (See also *spalloc_server.coordinates* for details of the coordinate systems used when referring to boards.)

Defining Machines

Since defining machines completely by hand can be quite verbose (see example below), a some convenience functions are provided to deal with the common case of machines constructed in the standard manner.

To define an isolated single-board machine, the *Machine.single_board()* constructor may be used as follows:

```
m = Machine.single_board("my-board",
                         bmp_ip="spinn-board-bmp",
                         spinnaker_ip="spinn-board")
configuration = Configuration(machines=[m])
```

Most multi-board systems follow a standardised IP addressing scheme and have their physical layout defined by SpiNNer. The *board_locations_from_spinner()* function reads CSV files produced by spinner-ethernet-chips describing machine layouts and the *Machine.with_standard_ips()* constructor produces a *Machine* with IP addresses based on the standard IP addressing scheme. These may be used together like so:

```
# spinner-ethernet-chips -n 1200 > ethernet_chips.csv
m = Machine.with_standard_ips(
    "million-core-machine",
    board_locations=board_locations_from_spinner("ethernet_chips.csv"),
    base_ip="10.2.0.0",
)
configuration = Configuration(machines=[m])
```

If neither of the above convenience functions apply to your machine, you can also explicitly define your machine's parameters. (Be sure to read about the *coordinates* used when referring to boards.) For example, a desktop 3-board machine may look something like:

```
m = Machine(name="my-three-board-machine",
            board_locations={
                #X  Y  Z    C  F  B
                (0, 0, 0): (0, 0, 0),
                (0, 0, 1): (0, 0, 2),
                (0, 0, 2): (0, 0, 5),
            },
            # Just one BMP
            bmp_ips={
                #C  F
                (0, 0): "192.168.240.0",
            },
            # Each SpiNNaker board has an IP
            spinnaker_ips={
                #X  Y  Z
                (0, 0, 0): "192.168.240.1",
                (0, 0, 1): "192.168.240.17",
                (0, 0, 2): "192.168.240.41",
            })
configuration = Configuration(machines=[m])
```

Remember, since the configuration file is just a normal Python file, you can use any code you like to pragmatically specify machines, etc. which you use.

Configuration File API Reference

class Configuration(*machines=None*, *port=22244*, *ip=''*, *timeout_check_interval=5.0*, *max_retired_jobs=1200*, *seconds_before_free=30*)

Defines the configuration of a server.

Parameters

machines
[[*Machine*, ...]] The list of machines, highest priority first, the server is to manage. (Default: [])

port
[int] The port number the server should listen on. Note that this is now deprecated; the port should be specified by the `--port` option on the spalloc_server command line. (Default: 22244)

ip
[str] The IP the server should listen on. (Default: "", i.e. all interfaces)

timeout_check_interval
[float] The number of seconds between the server's checks for job timeouts. (Default: 5.0)

max_retired_jobs
[int] The number of retired jobs to keep records of. (Default: 1200)

static __new__(*cls*, *machines=None*, *port=22244*, *ip=''*, *timeout_check_interval=5.0*, *max_retired_jobs=1200*, *seconds_before_free=30*)

Create new instance of Configuration(machines, port, ip, timeout_check_interval, max_retired_jobs, seconds_before_free)

class Machine(*name*, *tags=frozenset({'default'})*, *width=None*, *height=None*, *dead_boards=frozenset({})*, *dead_links=frozenset({})*, *board_locations=None*, *bmp_ips=None*, *spinnaker_ips=None*)

Defines a SpiNNaker machine.

Parameters

name
[str] The name of the machine.

tags
[set([str, ...])] A set of tags which jobs may use to filter machines by. Note that by default jobs are assigned the 'default' tag and thus machines probably ought have this tag too.

width, height
[int] The dimensions of the machine in triads of boards. If omitted, these are inferred from the boards defined in board_locations and dead_boards.

dead_boards
[set([(x, y, z), ...])] The board coordinates of all dead boards in the machine.

dead_links
[set([(x, y, z, `Links`), ...])] The board coordinates of all dead links in the machine. Links to dead boards are implicitly dead and may or may not be included in this set.

board_locations
[{(x, y, z): (c, f, b), ...}] Lookup from board coordinate to its physical in a SpiNNaker machine in terms of cabinet, frame and board position. Must give the coordinates of *all* working boards.

bmp_ips
[{(c, f): hostname, …}] The IP address of a BMP in every frame of the machine which contains working boards.

spinnaker_ips
[{(x, y, z): hostname, …}] For every working board gives the IP address of the SpiNNaker board's Ethernet connected chip.

static __new__(*cls*, *name*, *tags=frozenset({'default'})*, *width=None*, *height=None*, *dead_boards=frozenset({})*, *dead_links=frozenset({})*, *board_locations=None*, *bmp_ips=None*, *spinnaker_ips=None*)

Create new instance of Machine(name, tags, width, height, dead_boards, dead_links, board_locations, bmp_ips, spinnaker_ips)

classmethod single_board(*name*, *tags=frozenset({'default'})*, *bmp_ip=None*, *spinnaker_ip=None*)

Convenience constructor. Construct a `Machine` representing a single SpiNNaker board.

Parameters

name
[str] The name of the machine

tags
[set([tag, …])] The tags to assign to the machine.

bmp_ip
[str] The hostname of the BMP controlling the board.

spinnaker_ip
[str] The hostname of the SpiNNaker board.

classmethod with_standard_ips(*name*, *tags=frozenset({'default'})*, *width=None*, *height=None*, *dead_boards=frozenset({})*, *dead_links=frozenset({})*, *board_locations=None*, *base_ip='192.168.0.0'*, *cabinet_stride='0.0.5.0'*, *frame_stride='0.0.1.0'*, *board_stride='0.0.0.8'*, *bmp_offset='0.0.0.0'*, *spinnaker_offset='0.0.0.1'*)

Convenience constructor. Construct a `Machine` which infers IP addresses of the form conventionally used by SpiNNaker installations.

In standard SpiNNaker installations, IP addresses are allocated in a regular fashion as described below.

IP addresses for a particular machine are allocated within an address range, e.g. 192.168.0.0 - 192.168.255.255.

This address range is then subdivided into address ranges for each frame, for example:

- Cabinet 0, Frame 0: 192.168.0.0 - 192.168.0.255
- Cabinet 0, Frame 1: 192.168.1.0 - 192.168.1.255
- Cabinet 0, Frame 2: 192.168.2.0 - 192.168.2.255
- Cabinet 0, Frame 3: 192.168.3.0 - 192.168.3.255
- Cabinet 0, Frame 4: 192.168.4.0 - 192.168.4.255
- Cabinet 1, Frame 0: 192.168.5.0 - 192.168.5.255
- …

Boards within a frame are each allocated their own range of IPs, for example:

- Cabinet 0, Frame 0, Board 0: 192.168.0.0 - 192.168.0.7

- Cabinet 0, Frame 0, Board 1: 192.168.0.8 - 192.168.0.15
- Cabinet 0, Frame 0, Board 2: 192.168.0.16 - 192.168.0.23
- …

Finally, the IP address of the BMP and Ethernet-connected SpiNNaker chip of each board is at some fixed offset within this range, for example:

- Cabinet 0, Frame 0, Board 0, BMP: 192.168.0.0
- Cabinet 0, Frame 0, Board 0, SpiNNaker: 192.168.0.1
- Cabinet 0, Frame 0, Board 1, BMP: 192.168.0.8
- Cabinet 0, Frame 0, Board 1, SpiNNaker: 192.168.0.9

Finally, we assume that board 0's BMP is to be used as the BMP for controlling all boards in its frame.

Parameters

name
[str] The name of the machine.

tags
[iterable([str, …])] A set of tags which jobs may use to filter machines by. Note that by default jobs are assigned the 'default' tag and thus machines probably ought have this tag too.

width, height
[int] The dimensions of the machine in triads of boards. If omitted, these are inferred from the boards defined in board_locations and dead_boards.

dead_boards
[iterable([(x, y, z), …])] The board coordinates of all dead boards in the machine.

dead_links
[iterable([(x, y, z, `Links`), …])] The board coordinates of all dead links in the machine. Links to dead boards are implicitly dead and may or may not be included in this set.

board_locations
[{(x, y, z): (c, f, b), …}] Lookup from board coordinate to its physical in a SpiNNaker machine in terms of cabinet, frame and board position. Must give the coordinates of *all* working boards.

base_ip
[str] The IPv4 address from which the IP address range assigned to the machine starts.

cabinet_stride
[str] The stride in IP addresses between individual cabinets, expressed as an IPv4 address.

frame_stride
[str] The stride in IP addresses between individual frames within a cabinet, expressed as an IPv4 address.

board_stride
[str] The stride in IP addresses between individual boards within a frame, expressed as an IPv4 address.

bmp_offset
[str] The offset of a board's BMP IP from the start of a board's IP address range, expressed as an IPv4 address.

spinnaker_offset
[str] The offset of a board's Ethernet-connected SpiNNaker chip IP from the start of a board's IP address range, expressed as an IPv4 address.

`board_locations_from_spinner`(*filename*)

Utility function which converts a CSV file produced by the spinner-ethernet-chips utility into a `board_locations` dictionary suitable for defining *`Machine`* objects.

Parameters

filename
[str] The name of a CSV file produced by spinner-ethernet-chips defining the relationship between Ethernet connected chip coordinates and physical board locations.

This file is expected to have five columns (named in the first line of the CSV) named 'board', 'cabinet', 'frame', 'x', and 'y'.

Returns

{(x, y, z): (c, f, b), ...}
The mapping from board coordinates to physical locations.

1.1.6 Coordinate systems

Utilities for working with board/triad coordinates.

This software assumes that all machines provided to it are interconnected in a valid (subset of) a hexagonal torus topology. Boards locations are expressed in one of several forms depending on circumstance.

- **Board coordinates** `(x, y, z)` giving the logical location of the board within a hexagonal torus configuration. This is the most frequently used coordinate system used by this software.
- **Physical coordinates** `(cabinet, frame, board)` giving the physical location of a board in a cabinet. Generally only used when dealing with board power management.
- **Ethernet chip coordinates** `(x, y)` giving the *chip* coordinates of the Ethernet connected chip at the bottom-left coordinate of a SpiNNaker board. Generally only used when relating information to a client.

To deal with these coordinate systems a selection of utility functions are provided in the *`spalloc_server.coordinates`*.

Board coordinates

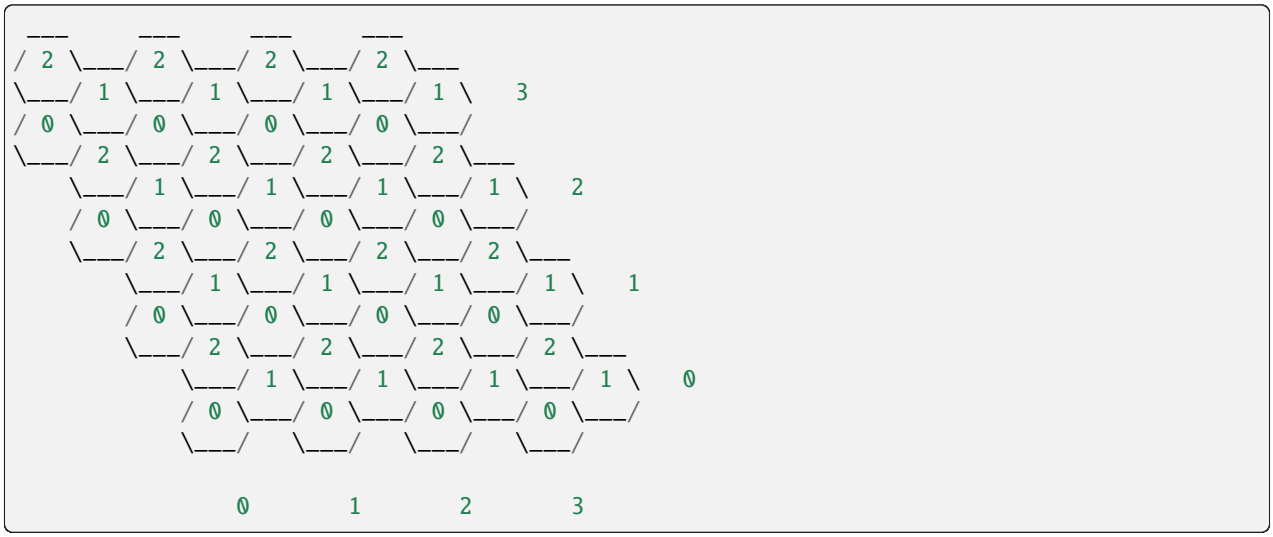
Board coordinates are given as a tuple `(x, y, z)`.

Systems of SpiNNaker boards are defined in terms of 'triads' of boards. The figure below shows a single triad. The 'z' part of a board coordinate comes from the index of the board within its triad and are numbered as follows:

```
 ___
/ 2 \___
\___/ 1 \
/ 0 \___/
\___/
```

Larger systems are defined by replicating this pattern of triads. Triads are indexed along the X axis as follows:

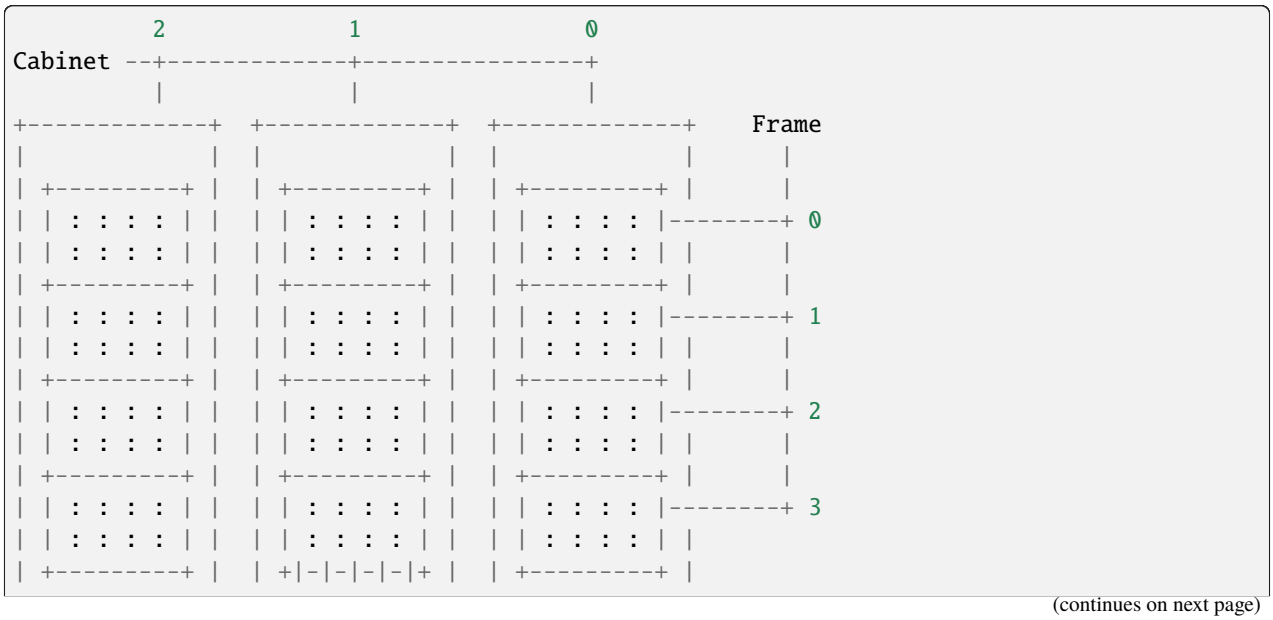
And then along the Y axis thus:



Physical coordinates

Physical coordinates are given as a tuple `(cabinet, frame, board)`.

Physical coordinates give the positions of boards within a set of cabinets containing several frames containing several boards. These are indexed as illustrated below, *starting from the top-right corner*:



(continues on next page)

(continued from previous page)

```
|             |  |  | | | | | |  |  |             |
+-------------+  +--|-|-|-|-|--+  +-------------+
                    | | | | |
          Board -----+-+-+-+-+
                     4 3 2 1 0
```

A mapping from board coordinates to physical coordinates is supplied by the user and is unique to the machine being built. A tool such as SpiNNer may be used to generate such mappings.

Ethernet chip coordinates

Ethernet chip coordinates are given as a tuple `(x, y)`.

Ethernet chip coordinates give the chip coordinates of Ethernet connected chips at the bottom-left corner of SpiNNaker boards.

Utilities

The following utilities are provided for working with the above coordinate systems.

link_to_vector = {(0, <Links.east: 0>): (0, -1, 2), (0, <Links.north_east: 1>): (0, 0, 1), (0, <Links.north: 2>): (0, 0, 2), (0, <Links.west: 3>): (-1, 0, 1), (0, <Links.south_west: 4>): (-1, -1, 2), (0, <Links.south: 5>): (-1, -1, 1), (1, <Links.east: 0>): (1, 0, -1), (1, <Links.north_east: 1>): (1, 0, 1), (1, <Links.north: 2>): (1, 1, -1), (1, <Links.west: 3>): (0, 0, 1), (1, <Links.south_west: 4>): (0, 0, -1), (1, <Links.south: 5>): (0, -1, 1), (2, <Links.east: 0>): (0, 0, -1), (2, <Links.north_east: 1>): (1, 1, -2), (2, <Links.north: 2>): (0, 1, -1), (2, <Links.west: 3>): (0, 1, -2), (2, <Links.south_west: 4>): (-1, 0, -1), (2, <Links.south: 5>): (0, 0, -2)}

A lookup from (z, `spalloc_server.links.Links`) to (dx, dy, dz).

board_down_link(*x1*, *y1*, *z1*, *link*, *width*, *height*)

Get the coordinates of the board down the specified link.

Parameters

x1, y1, z1
[int] The board coordinates from which a link will be traversed.

link
[`spalloc_server.links.Link`] The link to follow.

width, height
[int] The dimensions of the system in triads.

Returns

x, y, z
[int] The coordinates of the board down the specified link.

wrapped
[*WrapAround*] In what way did we wrap-around when following that link?

board_to_chip(*x*, *y*, *z*)

Convert a board coordinate into a chip coordinate.

Assumes a regular torus composed of SpiNN-5 boards.

Parameters

x, y, z
[int] Board coordinates.

Returns

x, y
[int] Chip coordinates.

`chip_to_board`(*x*, *y*, *w*, *h*)

Convert a chip coordinate into a board coordinate.

Assumes a regular torus composed of SpiNN-5 boards.

Parameters

x, y
[int] Chip coordinates.

w, h
[int] Dimensions of the system, in chips.

Returns

x, y, z
[int] Board coordinates.

`triad_dimensions_to_chips`(*w*, *h*, *torus*)

Convert the dimensions of a system from numbers of triads to numbers of chips in the underlying network.

Assumes a regular torus composed of SpiNN-5 boards.

Parameters

w, h
[int] Dimensions of the system in triads.

torus
[*WrapAround*] What wrap-around connections are present?

Returns

w, h
[int] Dimensions of the SpiNNaker chip network in the specified machine, e.g. for booting.

`class WrapAround`(*value*)

Defines what type of wrap-around links a torus has, if any.

Values chosen have the following useful properties:

```
>>> # Can be meaningfully cast to bool
>>> assert bool(WrapAround.none) is False
>>> assert bool(WrapAround.x) is True
>>> assert bool(WrapAround.y) is True
>>> assert bool(WrapAround.both) is True

>>> # Bit-operations make sense
>>> assert bool(WrapAround.both & WrapAround.x) is True
>>> assert bool(WrapAround.both & WrapAround.y) is True
>>> assert bool(WrapAround.x & WrapAround.x) is True
>>> assert bool(WrapAround.x & WrapAround.y) is False
```

`none = 0`

No wrap-around links.

`x = 1`

Has wrap around links around X-axis.

`y = 2`

Has wrap around links around Y-axis.

`both = 3`

Has wrap around links on X and Y axes.

CHAPTER

TWO

COMMUNICATION PROTOCOL

The following documentation describes the network protocol which clients must use to communicate with the server.

2.1 Communication Protocol

The SpiNNaker partition server uses a simple JSON-based protocol over TCP to communicate with clients. The protocol has no security features what-so-ever, just like SpiNNaker hardware, and it is assumed that the server is operated within the same trusted network as the boards it manages.

By default, the server listens on TCP port number 22244. The client and server communicate by sending and receiving newline (`\n`) delimited JSON objects (i.e. lines of the form `{...}`). Clients may send *commands* to the server to which *return values* are sent by the server. The server may also asynchronously send *notifications* to the client, if requested by the client.

As soon as a client connects to the server it should send a *version()* command and ensure the server version is compatible with the client.

2.1.1 Sending Commands

Commands map exactly to Python function calls. A command has a name and arguments and returns a value. To send a command, a client must sent a JSON object the following keys, followed by a newline:

"command"
: A string. The name of the command to be executed.

"args"
: An array. A list of positional arguments for the command.

"kwargs"
: An object. A list of keyword arguments for the command.

For example, if a client sent the following to the server:

```
{"command": "create_job", "args": [4, 2], "kwargs": {"owner": "me"}}\n
```

This would be interpreted as a function call like:

```
create_job(4, 2, owner="me")
```

Note: In all examples, `\n` means a newline character (ASCII 10), **not** the `\` and `n` characters.

The server will then respond with a JSON object with a single key, "return", whose value is the value returned by the command. For example:

```
{"return": 42}\n
```

Commands are processed and return values sent in FIFO order. No blocking commands are implemented by the server and the server will make a best-effort attempt to respond to all commands as quickly as possible. If any command is malformed or causes an error for any reason, the client is immediately disconnected.

2.1.2 Receiving Asynchronous Notifications

If the client requests to be notified of certain events (using a command, as described above), the server may send a JSON object to the client which does not contain the key "return". Notifications may be sent at any time, once requested, including between a function being called and the return value being sent. The exact format of the notification depends on its type. An example notification may look like the following:

```
{"jobs_changed": [42, 10, 3]}\n
```

2.1.3 Available Commands

The commands supported by the server are enumerated below and expressed in the form of Python functions.

version()

Return type
str

Returns
The server's version number.

create_job(**args*, ***kwargs*)

Create a new job (i.e. allocation of boards).

This function should be called in one of the following styles:

```
# Any single (SpiNN-5) board
job_id = create_job(owner="me")
job_id = create_job(1, owner="me")

# Board x=3, y=2, z=1 on the machine named "m"
job_id = create_job(3, 2, 1, machine="m", owner="me")

# Any machine with at least 4 boards
job_id = create_job(4, owner="me")

# Any 7-or-more board machine with an aspect ratio at least as
# square as 1:2
job_id = create_job(7, min_ratio=0.5, owner="me")

# Any 4x5 triad segment of a machine (may or may-not be a
# torus/full machine)
job_id = create_job(4, 5, owner="me")
```

(continues on next page)

(continued from previous page)

```
# Any torus-connected (full machine) 4x2 machine
job_id = create_job(4, 2, require_torus=True, owner="me")
```

The 'other parameters' enumerated below may be used to further restrict what machines the job may be allocated onto.

Jobs for which no suitable machines are available are immediately destroyed (and the reason given).

Once a job has been created, it must be 'kept alive' by a simple watchdog mechanism. Jobs may be kept alive by periodically calling the *job_keepalive()* command or by calling any other job-specific command. Jobs are culled if no keep alive message is received for `keepalive` seconds. If absolutely necessary, a job's keepalive value may be set to None, disabling the keepalive mechanism.

Once a job has been allocated some boards, these boards will be automatically powered on and left unbooted ready for use.

Parameters

- **owner** (*str*) – **Required.** The name of the owner of this job.
- **keepalive** (*float or None*) – The maximum number of seconds which may elapse between a query on this job before it is automatically destroyed. If None, no timeout is used. (Default: 60.0)
- **machine** (*str or None*) – Specify the name of a machine which this job must be executed on. If None, the first suitable machine available will be used, according to the tags selected below. Must be None when tags are given. (Default: None)
- **tags** (*list(str) or None*) – The set of tags which any machine running this job must have. If None is supplied, only machines with the "default" tag will be used. If machine is given, this argument must be None. (Default: None)
- **min_ratio** (*float*) – The aspect ratio (h/w) which the allocated region must be 'at least as square as'. Set to 0.0 for any allowable shape, 1.0 to be exactly square. Ignored when allocating single boards or specific rectangles of triads.
- **max_dead_boards** (*int or None*) – The maximum number of broken or unreachable boards to allow in the allocated region. If None, any number of dead boards is permitted, as long as the board on the bottom-left corner is alive (Default: None).
- **max_dead_links** (*int or None*) – The maximum number of broken links allow in the allocated region. When require_torus is True this includes wrap-around links, otherwise peripheral links are not counted. If None, any number of broken links is allowed. (Default: None).
- **require_torus** (*bool*) – If True, only allocate blocks with torus connectivity. In general this will only succeed for requests to allocate an entire machine (when the machine is otherwise not in use!). Must be False when allocating boards. (Default: False)

Returns

The job ID given to the newly allocated job.

Return type

int

`job_keepalive`(*job_id*)

Reset the keepalive timer for the specified job.

Note: All other job-specific commands implicitly do this.

Parameters
job_id (*int*) – A job ID to be kept alive.

`get_job_state`(*job_id*)

Poll the state of a running job.

Parameters
job_id (*int*) – A job ID to get the state of.

Return type
dict(str, ...)

Returns

A dictionary with the following keys:

`state`
[*JobState*] The current state of the queried job.

`power`
[bool or None] If job is in the ready or power states, indicates whether the boards are power{ed,ing} on (`True`), or power{ed,ing} off (`False`). In other states, this value is `None`.

`keepalive`
[float or None] The Job's keepalive value: the number of seconds between queries about the job before it is automatically destroyed. `None` if no timeout is active (or when the job has been destroyed).

`reason`
[str or None] If the job has been destroyed, this may be a string describing the reason the job was terminated.

`start_time`
[float or None] For queued and allocated jobs, gives the Unix time (UTC) at which the job was created (or None otherwise).

`get_job_machine_info`(*job_id*)

Get the list of Ethernet connections to the allocated machine.

Parameters
job_id (*int*) – A job ID to get the machine info for.

Return type
dict(str, ...)

Returns

A dictionary with the following keys:

`width, height`
[int or None] The dimensions of the machine in chips, e.g. for booting. `None` if no boards are allocated to the job.

`connections`
[[[[x, y], hostname], ...] or None] A list giving Ethernet-connected chip coordinates in the machine to hostname. `None` if no boards are allocated to the job.

machine_name
[str or None] The name of the machine the job is allocated on. `None` if no boards are allocated to the job.

boards
[[[x, y, z], ...] or None] All the boards allocated to the job or `None` if no boards allocated.

power_on_job_boards(*job_id*)

Power on (or reset if already on) boards associated with a job.

Once called, the job will enter the 'power' state until the power state change is complete, this may take some time.

Parameters
job_id (*int*) – A job ID to turn boards on for.

power_off_job_boards(*job_id*)

Power off boards associated with a job.

Once called, the job will enter the 'power' state until the power state change is complete, this may take some time.

Parameters
job_id (*int*) – A job ID to turn boards off for.

destroy_job(*job_id*, *reason=None*)

Destroy a job.

Call when the job is finished, or to terminate it early, this function releases any resources consumed by the job and removes it from any queues.

Parameters

- **job_id** (*int*) – A job ID to destroy.
- **reason** (*str*) – An optional human-readable description of the reason for the job's destruction.

notify_job(*job_id=None*)

Register to be notified about changes to a specific job ID.

Once registered, a client will be asynchronously be sent notifications form `{"jobs_changed": [job_id, ...]}\n` enumerating job IDs which have changed. Notifications are sent when a job changes state, for example when created, queued, powering on/off, powered on and destroyed. The specific nature of the change is not reflected in the notification.

Parameters
job_id (*int or None*) – A job ID to be notified of or None if all job state changes should be reported. Defaults to None (i.e., all jobs).

See also:

no_notify_job
Stop being notified about a job.

notify_machine
Register to be notified about changes to machines.

no_notify_job(*job_id=None*)

Stop being notified about a specific job ID.

Once this command returns, no further notifications for the specified ID will be received.

Parameters

job_id (*int or None*) – A job ID to no longer be notified of or None to not be notified of any jobs. Note that if all job IDs were registered for notification, this command only has an effect if the specified job_id is None. Defaults to None (i.e., all jobs).

See also:

notify_job
Register to be notified about changes to a specific job.

notify_machine(*machine_name=None*)

Register to be notified about a specific machine name.

Once registered, a client will be asynchronously be sent notifications of the form `{"machines_changed": [machine_name, ...]}\n` enumerating machine names which have changed. Notifications are sent when a machine changes state, for example when created, change, removed, allocated a job or an allocated job is destroyed.

Parameters

machine_name (*str or None*) – A machine name to be notified of or None if all machine state changes should be reported. Defaults to None (i.e., all machines).

See also:

no_notify_machine
Stop being notified about a machine.

notify_job
Register to be notified about changes to jobs.

no_notify_machine(*machine_name=None*)

Unregister to be notified about a specific machine name.

Once this command returns, no further notifications for the specified ID will be received.

Parameters

machine_name (*str or None*) – A machine name to no longer be notified of or None to not be notified of any machines. Note that if all machines were registered for notification, this command only has an effect if the specified machine_name is None. Defaults to None (i.e., all machines).

See also:

notify_machine
Register to be notified about changes to a machine.

list_jobs()

Enumerate all non-destroyed jobs.

Return type

list(dict(str, ...))

Returns

A list of allocated/queued jobs in order of creation from oldest (first) to newest (last). Each job is described by a dictionary with the following keys:

job_id
the ID of the job.

owner
the string giving the name of the Job's owner.

start_time
the time the job was created (Unix time, UTC).

keepalive
the maximum time allowed between queries for this job before it is automatically destroyed (or None if the job can remain allocated indefinitely).

state
the current *JobState* of the job.

power
indicates whether the boards are powered on or not. If job is in the ready or power states, indicates whether the boards are power{ed,ing} on (True), or power{ed,ing} off (False). In other states, this value is None.

args and kwargs
the arguments to the alloc function which specifies the type/size of allocation requested and the restrictions on dead boards, links and torus connectivity.

allocated_machine_name
the name of the machine the job has been allocated to run on (or None if not allocated yet).

boards
a list [(x, y, z), ...] of boards allocated to the job.

keepalivehost
the IP address of the host reckoned to be keeping this job alive (i.e., the host that did a request most recently that updated the internal keep-alive timeout).

list_machines()

Enumerates all machines known to the system.

Return type
list(dict(str, ...))

Returns

The list of machines known to the system in order of priority from highest (first) to lowest (last). Each machine is described by a dictionary with the following keys:

name
the name of the machine.

tags
the list ['tag', ...] of tags the machine has.

width and height
the dimensions of the machine in triads.

dead_boards
a list([(x, y, z), ...]) giving the coordinates of known-dead boards.

dead_links
a list([(x, y, z, link), ...]) giving the locations of known-dead links from the perspective of the sender. Links to dead boards may or may not be included in this list.

get_board_position(*machine_name*, *x*, *y*, *z*)

Get the physical location of a specified board.

Parameters

- **machine_name** (*str*) – The name of the machine containing the board.
- **x** (*int*) – Logical address within machine: first coordinate.
- **y** (*int*) – Logical address within machine: second coordinate.
- **z** (*int*) – Logical address within machine: third coordinate.

Returns
The physical location of the board (cabinet, frame, board) at the specified location or None if the machine/board are not recognised.

Return type
list(int) or None

get_board_at_position(*machine_name*, *x*, *y*, *z*)

Get the logical location of a board at the specified physical location.

Parameters

- **machine_name** (*str*) – The name of the machine containing the board.
- **x** (*int*) – Physical address within machine: cabinet ID.
- **y** (*int*) – Physical address within machine: frame ID.
- **z** (*int*) – Physical address within machine: board ID.

Returns
The logical location of the board (a triple) at the specified location or None if the machine/board are not recognised.

Return type
list(int) or None

where_is(***kwargs*)

Find out where a SpiNNaker board or chip is located, logically and physically.

May be called in one of the following styles:

```
>>> # Query by logical board coordinate within a machine.
>>> where_is(machine=..., x=..., y=..., z=...)

>>> # Query by physical board location within a machine.
>>> where_is(machine=..., cabinet=..., frame=..., board=...)

>>> # Query by chip coordinate (as if the machine were booted as
>>> # one large machine).
>>> where_is(machine=..., chip_x=..., chip_y=...)

>>> # Query by chip coordinate, within the boards allocated to a
>>> # job.
>>> where_is(job_id=..., chip_x=..., chip_y=...)
```

Only these patterns of use are supported; all keyword arguments listed above are mandatory when used for a particular query.

Return type
dict(str, ...) or None

Returns

If a board exists at the supplied location, a dictionary giving the location of the board/chip, supplied in a number of alternative forms. If the supplied coordinates do not specify a specific chip, the chip coordinates given are those of the Ethernet connected chip on that board.

If no board exists at the supplied position, `None` is returned instead.

The dictionary will have the following keys:

machine
: the name of the machine containing the board.

logical
: the logical board coordinate, (x, y, z) within the machine.

physical
: the physical board location, (cabinet, frame, board), within the machine.

chip
: the coordinates of the chip, (x, y), if the whole machine were booted as a single machine.

board_chip
: the coordinates of the chip, (x, y), within its board.

job_id
: the job ID of the job currently allocated to the board identified or `None` if the board is not allocated to a job.

job_chip
: the coordinates of the chip, (x, y), within its job, if a job is allocated to the board, or `None` otherwise.

class JobState

A job may be in any of the following (numbered) states.

Number	State
0	*unknown*
1	*queued*
2	*power*
3	*ready*
4	*destroyed*

CHAPTER

THREE

INTERNALS OVERVIEW

Essential reading for developers and maintainers. The following documentation describes the internal architecture of the SpiNNaker Partitioning Server and should serve as a guide to those wishing to extend its functionality.

3.1 Internals Guide

This guide attempts (in some logical order) to explain the structure of the SpiNNaker Partitioning Server code base and development process.

This guide starts with a brief development installation guide and then proceeds to introduce the structure and internals of the SpiNNaker Partitioning Server. We then introduce the coordinate systems used throughout the code base and descend the hierarchy working down from the server logic through job scheduling/queueing logic and finishing with the underlying board allocation logic.

The 'meat' of the server can be broken down into the following principle modules:

spalloc_server.server
: The top-level server implementation. Contains all server logic, command handling etc. Internally uses a *Controller* for scheduling and allocating jobs and managing SpiNNaker machines.

spalloc_server.controller
: Schedules and allocates jobs to machines (using *JobQueue*) and accordingly configures SpiNNaker hardware (using *AsyncBMPController*).

spalloc_server.job_queue
: Schedules and allocates jobs to machines. Attempts to allocate jobs to machines (using *Allocator*s) and queues up and schedules jobs for which no machines are available.

spalloc_server.allocator
: Provides an 'alloc' like mechanism for allocating boards in a single SpiNNaker machine. Allocations occur at the granularity of individual SpiNNaker boards and high-level constraints on the allocations may be made (e.g. requiring a specific proportion of an allocation's links or boards to be functional). Internally the *PackTree* 2D packing algorithm is used pack allocations at the coarser granularity of triads of boards.

spalloc_server.pack_tree
: A simple 'online' 2D packing algorithm/data structure. This is used to manage the allocation and freeing of rectangular regions of SpiNNaker machine at the granularity of triads of boards.

spalloc_server.async_bmp_controller
: An object which allows power and link configuration commands to be asynchronously queued, coalesced and executed on SpiNNaker board BMPs.

If you enjoy the MVC design pattern, the major pieces above can be categorised approximately as:

- *Model*: *job_queue*, *allocator* and *pack_tree*.

- *View*: *server*
- *Controller*: *controller* and *async_bmp_controller*.

A number of other smaller modules also exist as follows:

spalloc_server.coordinates
: Utility functions for working with the coordinate systems used to describe the locations of boards in large machines.

spalloc_server.area_to_rect
: Utility functions for working out sensible sizes of machine to allocate given a minimum number of boards and worst-case aspect ratio.

spalloc_server.configuration
: Objects used to define a configuration of the server, constructed by the user's config file.

The documentation below is presented in a recommended skimming/reading order for new developers who wish to understand the code base.

3.1.1 Development Installation

A development install is produced from the checked out Git repository as usual:

```
$ # Setup virtual env...
$ python setup.py develop
```

Tests are run using pytest:

```
$ pip install -r requirements-test.txt
$ py.test tests
```

Documentation is built using Sphinx:

```
$ pip install -r requirements-docs.txt
$ cd docs
$ make html
```

3.1.2 Server logic (server)

A TCP server which exposes a public interface for allocating boards.

This module is essentially the 'top level' module for the functionality of theSpiNNaker Partitioning Server, containing the function <.main>`which is mapped to the ``spalloc-server`() command-line tool.

_COMMANDS = {'create_job': <function SpallocServer.create_job>, 'destroy_job': <function SpallocServer.destroy_job>, 'get_board_at_position': <function SpallocServer.get_board_at_position>, 'get_board_position': <function SpallocServer.get_board_position>, 'get_job_machine_info': <function SpallocServer.get_job_machine_info>, 'get_job_state': <function SpallocServer.get_job_state>, 'job_keepalive': <function SpallocServer.job_keepalive>, 'list_jobs': <function SpallocServer.list_jobs>, 'list_machines': <function SpallocServer.list_machines>, 'no_notify_job': <function SpallocServer.no_notify_job>, 'no_notify_machine': <function SpallocServer.no_notify_machine>, 'notify_job': <function SpallocServer.notify_job>, 'notify_machine': <function SpallocServer.notify_machine>, 'power_off_job_boards': <function SpallocServer.power_off_job_boards>, 'power_on_job_boards': <function SpallocServer.power_on_job_boards>, 'version': <function SpallocServer.version>, 'where_is': <function SpallocServer.where_is>}

A dictionary from command names to (unbound) methods of the *Server* class.

spalloc_command(*f*)

Decorator which marks a class function of *Server* as a command which may be called by a client.

exception _CriticalStop

__weakref__

list of weak references to the object (if defined)

class Server(*config_filename*, *cold_start=False*, *port=22244*)

A TCP server which manages, power, partitioning and scheduling of jobs on SpiNNaker machines.

Once constructed the server starts a background thread (`_server_thread`, *_run()*) which implements the main server logic and handles communication with clients, monitoring of asynchronous board control events (e.g. board power-on completion) and watches for selected signals. All members of this object are assumed to be accessed only from this thread while it is running. The thread is stopped, and its completion awaited by calling *stop_and_join()*, stopping the server.

The server uses a `Controller` object to implement scheduling, allocation and machine management functionality. This object is `pickled` when the server shuts down in order to preserve the state of all managed machines (e.g. allocated jobs etc.).

To allow the interruption of the server thread on asynchronous events from the Controller a `socketpair()` (`_notify_send` and `_notify_send`) is used which monitored along with client connections and SIGHUP signals (used to trigger config file reloading).

A number of callable commands are implemented by the server in the form of a subset of the *Server*'s methods indicated by the *spalloc_command()* decorator. These may be called by a client by sending a line `{"command": "...", "args": [...], "kwargs": {...}}`. If the function throws an exception, the client is disconnected. If the function returns, it is packed as a JSON line `{"return": ...}`, and if the function raises an exception, it is packed as a JSON line `{"exception": "the message"}`.

__init__(*config_filename*, *cold_start=False*, *port=22244*)

Parameters

- **config_filename** (*str*) – The filename of the config file for the server which describes the machines to be controlled.
- **cold_start** (*bool*) – If False (the default), the server will attempt to restore its previous state, if True, the server will start from scratch.
- **port** (*int*) – Which port to listen on. Defaults to 22244.

_get_state_filename(*cfg*)

How to get the name of the state file from the name of another file (expected to be the config file). Assumes that the config file is in a directory that can be written to.

Parameters
cfg (*str*) – The name of the file to use as a base.

Returns
The full filename

_parse_config(*config_file_contents*)

How to parse the contents of the configuration file. Note that this is intended to be a basic parse, not an extended semantic check for high-level validity.

Parameters
config_file_contents (*str*) – the contents of the file

Returns
Some parsed description of the configuration. Will be passed to validator method.

Raises
Exception – if the parse fails

_validate_config(*parsed_config*)

How to check the parsed contents of the configuration for high-level semantic validity.

Parameters
parsed_config – Whatever was produced by the parser method.

Returns
The validated configuration object, or *None* if the configuration was invalid.

_load_valid_config(*validated_config*)

How to install the validated configuration. Not expected to have a failure mode under normal circumstances.

Parameters
validated_config – Whatever was produced by the validator method.

_close()

Close all server sockets and disconnect all client connections.

Returns
Whether the server socket itself was closed.

_disconnect_client(*client*)

Disconnect a client.

Parameters
client (socket.Socket) – The client to disconnect

_accept_client()

Accept a new client.

_msg_client(*client*, *message*)

Low-level way to send a message to a client.

Parameters

- **client** (socket.Socket) – The client that we are sending a message to.
- **message** – The object or array to send. Will be converted to JSON.

_handle_command(*client*, *line*)

Dispatch a single command.

Parameters

- **client** (socket.Socket) – The client that made the request, used to provide a session context where relevant.
- **line** (*str*) – The line parsed from the socket. Should be a complete JSON object with at least a 'command' key.

Returns
Dictionary describing the result of the operation.

Return type
dict

Raises
IOError – If something really bad goes wrong with message decoding. The expected response at this point would be simply closing the socket.

_handle_commands(*client*)

Handle incoming commands from a client.

Parameters
client (socket.Socket) – The client that made the request, used to provide a session context where relevant.

_send_notifications(*label*, *changes*, *watches*)

How to actually send requested notifications.

_run()

The main server thread.

This 'infinite' loop runs in a background thread and waits for and processes events such as the PollingServerCore.wake() method being called, the config file changing, clients sending commands or new clients connecting. It also periodically calls *controller.Controller.destroy_timed_out_jobs()* on the controller.

is_alive()

Is the server running?

join()

Wait for the server to completely shut down.

stop_and_join()

Stop the server and wait for it to shut down completely.

static _name(*client*)

Get the 'name' of the client. This is the string form of the IP address, or None if the concept isn't well defined.

Returns
The client's name (for logging).

class SpallocServer(*config_filename*, *cold_start=False*, *port=22244*)

__init__(*config_filename*, *cold_start=False*, *port=22244*)

Parameters

- **config_filename** (*str*) – The filename of the config file for the server which describes the machines to be controlled.
- **cold_start** (*bool*) – If False (the default), the server will attempt to restore its previous state, if True, the server will start from scratch.
- **port** (*int*) – Which port to listen on. Defaults to 22244.

_send_change_notifications()

Send any registered change notifications to clients.

Sends notifications of the forms `{"jobs_changed": [job_id, ...]}` and `{"machines_changed": [machine_name, ...]}` to clients who have subscribed to be notified of changes to jobs or machines.

version(*_client*)

Return type
str

Returns
The server's version number.

create_job(*client*, **args*, ***kwargs*)

Create a new job (i.e. allocation of boards).

This function should be called in one of the following styles:

```
# Any single (SpiNN-5) board
job_id = create_job(owner="me")
job_id = create_job(1, owner="me")

# Board x=3, y=2, z=1 on the machine named "m"
job_id = create_job(3, 2, 1, machine="m", owner="me")

# Any machine with at least 4 boards
job_id = create_job(4, owner="me")

# Any 7-or-more board machine with an aspect ratio at least as
# square as 1:2
job_id = create_job(7, min_ratio=0.5, owner="me")

# Any 4x5 triad segment of a machine (may or may-not be a
# torus/full machine)
job_id = create_job(4, 5, owner="me")

# Any torus-connected (full machine) 4x2 machine
job_id = create_job(4, 2, require_torus=True, owner="me")
```

The 'other parameters' enumerated below may be used to further restrict what machines the job may be allocated onto.

Jobs for which no suitable machines are available are immediately destroyed (and the reason given).

Once a job has been created, it must be 'kept alive' by a simple watchdog mechanism. Jobs may be kept alive by periodically calling the *job_keepalive()* command or by calling any other job-specific command. Jobs are culled if no keep alive message is received for `keepalive` seconds. If absolutely necessary, a job's keepalive value may be set to None, disabling the keepalive mechanism.

Once a job has been allocated some boards, these boards will be automatically powered on and left unbooted ready for use.

Parameters

- **owner** (*str*) – **Required.** The name of the owner of this job.
- **keepalive** (*float or None*) – The maximum number of seconds which may elapse between a query on this job before it is automatically destroyed. If None, no timeout is used. (Default: 60.0)
- **machine** (*str or None*) – Specify the name of a machine which this job must be executed on. If None, the first suitable machine available will be used, according to the tags selected below. Must be None when tags are given. (Default: None)
- **tags** (*list(str) or None*) – The set of tags which any machine running this job must have. If None is supplied, only machines with the "default" tag will be used. If machine is given, this argument must be None. (Default: None)
- **min_ratio** (*float*) – The aspect ratio (h/w) which the allocated region must be 'at least as square as'. Set to 0.0 for any allowable shape, 1.0 to be exactly square. Ignored when allocating single boards or specific rectangles of triads.
- **max_dead_boards** (*int or None*) – The maximum number of broken or unreachable boards to allow in the allocated region. If None, any number of dead boards is permitted, as long as the board on the bottom-left corner is alive (Default: None).
- **max_dead_links** (*int or None*) – The maximum number of broken links allow in the allocated region. When require_torus is True this includes wrap-around links, otherwise peripheral links are not counted. If None, any number of broken links is allowed. (Default: None).
- **require_torus** (*bool*) – If True, only allocate blocks with torus connectivity. In general this will only succeed for requests to allocate an entire machine (when the machine is otherwise not in use!). Must be False when allocating boards. (Default: False)

Returns
The job ID given to the newly allocated job.

Return type
int

job_keepalive(*client*, *job_id*)

Reset the keepalive timer for the specified job.

Note: All other job-specific commands implicitly do this.

Parameters
job_id (*int*) – A job ID to be kept alive.

get_job_state(*client*, *job_id*)

Poll the state of a running job.

Parameters
job_id (*int*) – A job ID to get the state of.

Return type
dict(str, ...)

Returns

A dictionary with the following keys:

state
[*JobState*] The current state of the queried job.

power
[bool or None] If job is in the ready or power states, indicates whether the boards are power{ed,ing} on (`True`), or power{ed,ing} off (`False`). In other states, this value is `None`.

keepalive
[float or None] The Job's keepalive value: the number of seconds between queries about the job before it is automatically destroyed. `None` if no timeout is active (or when the job has been destroyed).

reason
[str or None] If the job has been destroyed, this may be a string describing the reason the job was terminated.

start_time
[float or None] For queued and allocated jobs, gives the Unix time (UTC) at which the job was created (or None otherwise).

get_job_machine_info(*client*, *job_id*)

Get the list of Ethernet connections to the allocated machine.

Parameters
job_id (*int*) – A job ID to get the machine info for.

Return type
dict(str, ...)

Returns

A dictionary with the following keys:

width, height
[int or None] The dimensions of the machine in chips, e.g. for booting. `None` if no boards are allocated to the job.

connections
[[[[x, y], hostname], ...] or None] A list giving Ethernet-connected chip coordinates in the machine to hostname. `None` if no boards are allocated to the job.

machine_name
[str or None] The name of the machine the job is allocated on. `None` if no boards are allocated to the job.

boards
[[[x, y, z], ...] or None] All the boards allocated to the job or `None` if no boards allocated.

power_on_job_boards(*client*, *job_id*)

Power on (or reset if already on) boards associated with a job.

Once called, the job will enter the 'power' state until the power state change is complete, this may take some time.

Parameters
job_id (*int*) – A job ID to turn boards on for.

power_off_job_boards(*client*, *job_id*)

Power off boards associated with a job.

Once called, the job will enter the 'power' state until the power state change is complete, this may take some time.

Parameters
job_id (*int*) – A job ID to turn boards off for.

destroy_job(*client*, *job_id*, *reason=None*)

Destroy a job.

Call when the job is finished, or to terminate it early, this function releases any resources consumed by the job and removes it from any queues.

Parameters

- **job_id** (*int*) – A job ID to destroy.
- **reason** (*str*) – An optional human-readable description of the reason for the job's destruction.

_register_for_notifications(*client*, *watchset*, *_id*)

Helper method that handles the protocol for registration for notifications.

_unregister_for_notifications(*client*, *watchset*, *_id*)

Helper method that handles the protocol for unregistration for notifications.

notify_job(*client*, *job_id=None*)

Register to be notified about changes to a specific job ID.

Once registered, a client will be asynchronously be sent notifications form `{"jobs_changed": [job_id, ...]}\n` enumerating job IDs which have changed. Notifications are sent when a job changes state, for example when created, queued, powering on/off, powered on and destroyed. The specific nature of the change is not reflected in the notification.

Parameters
job_id (*int or None*) – A job ID to be notified of or None if all job state changes should be reported. Defaults to None (i.e., all jobs).

See also:

no_notify_job
Stop being notified about a job.

notify_machine
Register to be notified about changes to machines.

no_notify_job(*client*, *job_id=None*)

Stop being notified about a specific job ID.

Once this command returns, no further notifications for the specified ID will be received.

Parameters
job_id (*int or None*) – A job ID to no longer be notified of or None to not be notified of any jobs. Note that if all job IDs were registered for notification, this command only has an effect if the specified job_id is None. Defaults to None (i.e., all jobs).

See also:

notify_job
Register to be notified about changes to a specific job.

notify_machine(*client*, *machine_name=None*)

Register to be notified about a specific machine name.

Once registered, a client will be asynchronously be sent notifications of the form `{"machines_changed": [machine_name, ...]}\n` enumerating machine names which have changed. Notifications are sent when a machine changes state, for example when created, change, removed, allocated a job or an allocated job is destroyed.

Parameters

machine_name (*str or None*) – A machine name to be notified of or None if all machine state changes should be reported. Defaults to None (i.e., all machines).

See also:

no_notify_machine
Stop being notified about a machine.

notify_job
Register to be notified about changes to jobs.

no_notify_machine(*client*, *machine_name=None*)

Unregister to be notified about a specific machine name.

Once this command returns, no further notifications for the specified ID will be received.

Parameters

machine_name (*str or None*) – A machine name to no longer be notified of or None to not be notified of any machines. Note that if all machines were registered for notification, this command only has an effect if the specified machine_name is None. Defaults to None (i.e., all machines).

See also:

notify_machine
Register to be notified about changes to a machine.

list_jobs(*_client*)

Enumerate all non-destroyed jobs.

Return type

list(dict(str, ...))

Returns

A list of allocated/queued jobs in order of creation from oldest (first) to newest (last). Each job is described by a dictionary with the following keys:

job_id
the ID of the job.

owner
the string giving the name of the Job's owner.

start_time
the time the job was created (Unix time, UTC).

keepalive
the maximum time allowed between queries for this job before it is automatically destroyed (or `None` if the job can remain allocated indefinitely).

state
the current *JobState* of the job.

power
indicates whether the boards are powered on or not. If job is in the ready or power states, indicates whether the boards are power{ed,ing} on (`True`), or power{ed,ing} off (`False`). In other states, this value is `None`.

args and kwargs
the arguments to the alloc function which specifies the type/size of allocation requested and the restrictions on dead boards, links and torus connectivity.

allocated_machine_name
the name of the machine the job has been allocated to run on (or `None` if not allocated yet).

boards
a list [(x, y, z), ...] of boards allocated to the job.

keepalivehost
the IP address of the host reckoned to be keeping this job alive (i.e., the host that did a request most recently that updated the internal keep-alive timeout).

list_machines(*_client*)

Enumerates all machines known to the system.

Return type
list(dict(str, ...))

Returns

The list of machines known to the system in order of priority from highest (first) to lowest (last). Each machine is described by a dictionary with the following keys:

name
the name of the machine.

tags
the list ['tag', ...] of tags the machine has.

width and height
the dimensions of the machine in triads.

dead_boards
a list([(x, y, z), ...]) giving the coordinates of known-dead boards.

dead_links
a list([(x, y, z, link), ...]) giving the locations of known-dead links from the perspective of the sender. Links to dead boards may or may not be included in this list.

get_board_position(*_client*, *machine_name*, *x*, *y*, *z*)

Get the physical location of a specified board.

Parameters

- **machine_name** (*str*) – The name of the machine containing the board.
- **x** (*int*) – Logical address within machine: first coordinate.
- **y** (*int*) – Logical address within machine: second coordinate.
- **z** (*int*) – Logical address within machine: third coordinate.

Returns
The physical location of the board (cabinet, frame, board) at the specified location or None if the machine/board are not recognised.

Return type
list(int) or None

get_board_at_position(*_client*, *machine_name*, *x*, *y*, *z*)

Get the logical location of a board at the specified physical location.

Parameters

- **machine_name** (*str*) – The name of the machine containing the board.
- **x** (*int*) – Physical address within machine: cabinet ID.
- **y** (*int*) – Physical address within machine: frame ID.
- **z** (*int*) – Physical address within machine: board ID.

Returns
The logical location of the board (a triple) at the specified location or None if the machine/board are not recognised.

Return type
list(int) or None

where_is(*_client*, ***kwargs*)

Find out where a SpiNNaker board or chip is located, logically and physically.

May be called in one of the following styles:

```
>>> # Query by logical board coordinate within a machine.
>>> where_is(machine=..., x=..., y=..., z=...)

>>> # Query by physical board location within a machine.
>>> where_is(machine=..., cabinet=..., frame=..., board=...)

>>> # Query by chip coordinate (as if the machine were booted as
>>> # one large machine).
>>> where_is(machine=..., chip_x=..., chip_y=...)

>>> # Query by chip coordinate, within the boards allocated to a
>>> # job.
>>> where_is(job_id=..., chip_x=..., chip_y=...)
```

Only these patterns of use are supported; all keyword arguments listed above are mandatory when used for a particular query.

Return type
dict(str, ...) or None

Returns

If a board exists at the supplied location, a dictionary giving the location of the board/chip, supplied in a number of alternative forms. If the supplied coordinates do not specify a specific chip, the chip coordinates given are those of the Ethernet connected chip on that board.

If no board exists at the supplied position, `None` is returned instead.

The dictionary will have the following keys:

machine
the name of the machine containing the board.

logical
the logical board coordinate, (x, y, z) within the machine.

physical
the physical board location, (cabinet, frame, board), within the machine.

chip
the coordinates of the chip, (x, y), if the whole machine were booted as a single machine.

board_chip
the coordinates of the chip, (x, y), within its board.

job_id
the job ID of the job currently allocated to the board identified or `None` if the board is not allocated to a job.

job_chip
the coordinates of the chip, (x, y), within its job, if a job is allocated to the board, or `None` otherwise.

main(*args=None*)

Command-line launcher for the server.

The server may be cleanly terminated using a keyboard interrupt (e.g., `Ctrl+c`), and may be told to reload its configuration via *SIGHUP*.

Parameters
args – The command-line arguments passed to the program.

3.1.3 Top-level scheduling, allocation and hardware control logic (`controller`)

A high-level control interface for scheduling and allocating jobs and managing hardware in a collection of SpiNNaker machines.

class Controller(*next_id=1*, *max_retired_jobs=1200*, *on_background_state_change=None*, *seconds_before_free=30*)

An object which allocates jobs to machines and manages said machines' hardware.

This object is intended to form the core of a server which manages the queueing and execution of jobs on several SpiNNaker machines at once using a *JobQueue* and interacts with the hardware of said machines using *AsyncBMPController*.

'Jobs' may be created using the *create_job()* and are allocated a unique ID. Jobs are then queued, allocated and destroyed according to machine availability and user intervention. The state of a job may be queried using methods such as *get_job_state()*. When a job changes state it is added to the `changed_jobs` set. If a job's state is changed due to a background process (rather than in response to calling a *Controller* method), `on_background_state_change` is called.

JobQueue calls callbacks in this object when queued jobs are allocated to machines (*_job_queue_on_allocate()*), allocations are freed (*_job_queue_on_free()*) or cancelled without being allocated (*_job_queue_on_cancel()*). These callback functions implement the bulk of the functionality of this object by recording state changes in jobs and triggering the sending of power/link commands to SpiNNaker machines.

Machines may be added, modified and removed at any time by modifying the `machines` attribute. If a machine is removed or changes significantly, jobs running on the machine are cancelled, otherwise existing jobs should continue to execute or be scheduled on any new machines as appropriate.

Finally, once the controller is shut down (and outstanding BMP commands are flushed) using *stop()* and *join()* methods, it may be pickled and later unpickled to resume operation of the controller from where it left off before it was shut down.

Users should, at a regular interval call *destroy_timed_out_jobs()* in order to destroy any queued or running jobs which have not been kept alive recently enough.

Unless otherwise indicated, all methods are thread safe.

Attributes

max_retired_jobs
[int] Maximum number of retired jobs to retain the state of.

machines
[{name: *Machine*, ...} or similar OrderedDict] Defines the machines now available to the controller.

changed_jobs
[set([job_id, ...])] The set of job_ids whose state has changed since the last time this set was accessed. Reading this value clears it.

changed_machines
[set([machine_name, ...])] The set of machine names whose state has changed since the last time this set was accessed. Reading this value clears it. For example, machines are marked as changed if their tags are changed, if they are added or removed or if a job is allocated or freed on them.

on_background_state_change
[function() or None] A function which is called (from any thread) when any state changes occur in a background process and not as a direct result of calling a method of the controller.

The callback function *must not* call any methods of the controller object.

Note that this attribute is not pickled and unpicking a controller sets this attribute to None.

__init__(*next_id=1*, *max_retired_jobs=1200*, *on_background_state_change=None*, *seconds_before_free=30*)

Parameters

next_id
[int, optional] The next Job ID to assign

max_retired_jobs
[int, optional] See attribute of same name.

on_background_state_change
[function, optional] See attribute of same name.

seconds_before_free
[int] The number of seconds between a board being freed and it becoming available again

__getstate__()

Called when pickling this object.

This object may only be pickled once *stop()* and *join()* have returned.

__setstate__(*state*)

Called when unpickling this object.

Note that though the object must be pickled when stopped, the unpickled object will start running immediately.

stop()

Request that all background threads stop.

This will cause all outstanding BMP commands to be flushed.

> **Warning:** Apart from *join()*, no methods of this controller object may be called once this method has been called.

See also:

join
to wait for the threads to actually terminate.

join()

Block until all background threads have halted and all queued BMP commands completed.

create_job(*ownerhost*, **args*, ***kwargs*)

Create a new job (i.e. allocation of boards).

This function is a wrapper around *JobQueue.create_job()* which automatically selects (and returns) a new job_id. As such, the following *additional* (keyword) arguments are accepted:

Parameters

owner
[str] **Required.** The name of the owner of this job.

keepalive
[float or None] *Optional.* The maximum number of seconds which may elapse between a query on this job before it is automatically destroyed. If None, no timeout is used. (Default: 60.0)

Returns

job_id
[int] The Job ID assigned to the job.

job_keepalive(*clienthost*, *job_id*)

Reset the keepalive timer for the specified job.

Note all other job-specific functions implicitly call this method.

get_job_state(*clienthost*, *job_id*)

Poll the state of a running job.

Returns

JobStateTuple

get_job_machine_info(*clienthost*, *job_id*)

Get information about the machine the job has been allocated.

Returns

JobMachineInfoTuple

power_on_job_boards(*clienthost*, *job_id*)

Power on (or reset if already on) boards associated with a job.

`power_off_job_boards`(*clienthost*, *job_id*)

Power off boards associated with a job.

`destroy_job`(*clienthost*, *job_id*, *reason=None*)

Destroy a job.

When the job is finished, or to terminate it early, this function releases any resources consumed by the job and removes it from any queues.

Parameters

reason
[str or None, optional] A human-readable string describing the reason for the job's destruction.

`list_jobs`()

Enumerate all current jobs.

Returns

jobs
[[*JobTuple*, ...]] A list of allocated/queued jobs in order of creation from oldest (first) to newest (last).

`list_machines`()

Enumerates all machines known to the system.

Returns

machines
[[*MachineTuple*, ...]] The list of machines known to the system in order of priority from highest (first) to lowest (last).

`get_board_position`(*machine_name*, *x*, *y*, *z*)

Get the physical location of a specified board.

Parameters

machine_name
[str] The name of the machine containing the board.

x, y, z
[int] The logical board location within the machine.

Returns

(cabinet, frame, board) or None
The physical location of the board at the specified location or None if the machine/board are not recognised.

`get_board_at_position`(*machine_name*, *cabinet*, *frame*, *board*)

Get the logical location of a board at the specified physical location.

Parameters

machine_name
[str] The name of the machine containing the board.

cabinet, frame, board
[int] The physical board location within the machine.

Returns

(x, y, z) or None
The logical location of the board at the specified location or None if the machine/board are not recognised.

`_job_for_location`(*machine*, *x*, *y*, *z*)

Determine what job is running on the given board.

`_where_is_by_logical_triple`(*machine_name*, *x*, *y*, *z*)

Helper for *where_is()*

`_where_is_by_physical_triple`(*machine_name*, *cabinet*, *frame*, *board*)

Helper for *where_is()*

`_where_is_by_chip_coordinate`(*machine_name*, *chip_x*, *chip_y*)

Helper for *where_is()*

`_where_is_by_job_chip_coordinate`(*job_id*, *chip_x*, *chip_y*)

Helper for *where_is()*

`where_is`(***kwargs*)

Find out where a SpiNNaker board or chip is located, logically and physically.

May be called in one of the following styles:

```
>>> # Query by logical board coordinate within a machine.
>>> where_is(machine=..., x=..., y=..., z=...)

>>> # Query by physical board location within a machine.
>>> where_is(machine=..., cabinet=..., frame=..., board=...)

>>> # Query by chip coordinate (as if the machine were booted as
>>> # one large machine).
>>> where_is(machine=..., chip_x=..., chip_y=...)

>>> # Query by chip coordinate, within the boards allocated to a
>>> # job.
>>> where_is(job_id=..., chip_x=..., chip_y=...)
```

Returns

{"machine": ..., "logical": ..., "physical": ..., "chip": ..., "board_chip": ..., "job_chip": ..., "job_id": ...} or None
If a board exists at the supplied location, a dictionary giving the location of the board/chip, supplied in a number of alternative forms. If the supplied coordinates do not specify a specific chip, the chip coordinates given are those of the Ethernet connected chip on that board.

If no board exists at the supplied position, None is returned instead.

`machine` gives the name of the machine containing the board.

`logical` the logical board coordinate, (x, y, z) within the machine.

`physical` the physical board location, (cabinet, frame, board), within the machine.

`chip` the coordinates of the chip, (x, y), if the whole machine were booted as a single machine.

`board_chip` the coordinates of the chip, (x, y), within its board.

`job_id` is the job ID of the job currently allocated to the board identified or None if the board is not allocated to a job.

`job_chip` the coordinates of the chip, (x, y), within its job, if a job is allocated to the board or None otherwise.

`destroy_timed_out_jobs()`

Destroy any jobs which have timed out.

`check_free()`

Check for freed machines that are now available

`_bmp_on_request_complete(job, success, reason=None)`

Callback function called by an AsyncBMPController when it completes a previously issued request.

This function sets the specified Job's state to JobState.ready when this function has been called job.bmp_requests_until_ready times.

This function should be passed partially-called with the job the callback is associated it.

Parameters

job
[*_Job*] The job whose state should be set. (To be defined by wrapping this method in a partial).

success
[bool] Command success indicator provided by the AsyncBMPController.

`_set_job_power_and_links(job, power, link_enable=None)`

Power on/off and configure links for the boards associated with a specific job.

Parameters

job
[*_Job*] The job whose boards should be controlled.

power
[bool] The power state to apply to the boards. True = on, False = off.

link_enable
[bool or None, optional] Whether to enable (True) or disable (False) peripheral links or leave them unchanged (None).

`_job_queue_on_allocate(job_id, machine_name, boards, periphery, torus)`

Called when a job is successfully allocated to a machine.

`_job_queue_on_free(job_id, reason)`

Called when a job is freed.

`_job_queue_on_cancel(job_id, reason)`

Called when a job is cancelled before having been allocated.

`_teardown_job(job_id, reason)`

Called once job has been removed from the JobQueue.

Powers down any hardware in use and finally removes the job from _jobs.

`_create_machine_bmp_controllers(machine, on_thread_start=None)`

Create BMP controllers for a machine.

`_init_dynamic_state()`

Initialise all dynamic (non-pickleable) state.

Specifically:

- Creates the global controller lock
- Creates connections to BMPs.
- Reset keepalive_until on all existing jobs (e.g. allowing remote devices a chance to reconnect before terminating their jobs).

`__weakref__`

list of weak references to the object (if defined)

`class JobState(value)`

All the possible states that a job may be in.

`unknown = 0`

The job ID requested was not recognised.

`queued = 1`

The job is waiting in a queue for a suitable machine.

`power = 2`

The boards allocated to the job are currently being powered on or powered off.

`ready = 3`

The job has been allocated boards and the boards are not currently powering on or powering off.

`destroyed = 4`

The job has been destroyed.

`class JobStateTuple(state, power, keepalive, reason, start_time, keepalivehost)`

Tuple describing the state of a particular job, returned by *Controller.get_job_state()*.

Parameters

state
[*JobState*] The current state of the queried job.

power
[bool or None] If job is in the ready or power states, indicates whether the boards are power{ed,ing} on (True), or power{ed,ing} off (False). In other states, this value is None.

keepalive
[float or None] The Job's keepalive value: the number of seconds between queries about the job before it is automatically destroyed. None if no timeout is active (or when the job has been destroyed).

reason
[str or None] If the job has been destroyed, this may be a string describing the reason the job was terminated.

start_time
[float or None] The Unix time (UTC) at which the job was created.

keepalivehost
[str or None] The IP address of the last system to take an action that caused the job to be kept alive.

class JobMachineInfoTuple(*width*, *height*, *connections*, *machine_name*, *boards*)

Tuple describing the machine alloated to a job, returned by *Controller.get_job_machine_info()*.

Parameters

width, height
[int or None] The dimensions of the machine in *chips* or None if no machine allocated.

connections
[{(x, y): hostname, ...} or None] A dictionary mapping from SpiNNaker Ethernet-connected chip coordinates in the machine to hostname or None if no machine allocated.

machine_name
[str or None] The name of the machine the job is allocated on or None if no machine allocated.

boards
[set([(x, y, z), ...]) or None] The boards allocated to the job.

class JobTuple(*job_id*, *owner*, *start_time*, *keepalive*, *state*, *power*, *args*, *kwargs*, *allocated_machine_name*, *boards*, *keepalivehost*)

Tuple describing a job in the list of jobs returned by *Controller.list_jobs()*.

Parameters

job_id
[int] The ID of the job.

owner
[str] The string giving the name of the Job's owner.

start_time
[float] The time the job was created (Unix time, UTC)

keepalive
[float or None] The maximum time allowed between queries for this job before it is automatically destroyed (or None if the job can remain allocated indefinitely).

machine
[str or None] The name of the machine the job was specified to run on (or None if not specified).

state
[*JobState*] The current state of the job.

power
[bool or None] If job is in the ready or power states, indicates whether the boards are power{ed,ing} on (True), or power{ed,ing} off (False). In other states, this value is None.

args, kwargs
The arguments to the alloc function which specifies the type/size of allocation requested and the restrictions on dead boards, links and torus connectivity.

allocated_machine_name
[str or None] The name of the machine the job has been allocated to run on (or None if not allocated yet).

boards
[set([(x, y, z), ...])] The boards allocated to the job.

keepalivehost
[str] The name of the host that is reckoned to be keeping this job alive. Will be the empty string if no known host is doing so (a possible state after a service restart).

`class MachineTuple`(*name*, *tags*, *width*, *height*, *dead_boards*, *dead_links*)

Tuple describing a machine in the list of machines returned by *Controller.list_machines()*.

Parameters

name
[str] The name of the machine.

tags
[set(['tag', ...])] The tags the machine has.

width, height
[int] The dimensions of the machine in triads.

dead_boards
[set([(x, y, z), ...])] The coordinates of known-dead boards.

dead_links
[set([(x, y, z, `spalloc_server.links.Links`), ...])] The locations of known-dead links from the perspective of the sender. Links to dead boards may or may not be included in this list.

`class _Job`(*_id*, *owner*, *start_time=None*, *keepalive=60.0*, *lasthost=None*, *state=JobState.queued*, *power=None*, *args=()*, *kwargs=None*, *allocated_machine=None*, *boards=None*, *periphery=None*, *torus=None*, *width=None*, *height=None*, *connections=None*, *bmp_requests_until_ready=0*)

The metadata, used internally, associated with a non-destroyed job.

Attributes

id
[int] The ID of the job.

owner
[str] The job's owner.

start_time
[float] The time the job was created (Unix time, UTC)

keepalive
[float or None] The maximum time allowed between queries for this job before it is automatically destroyed (or None if the job can remain allocated indefinitely).

keepalive_until
[float or None] The time at which this job will become timed out (or None if no timeout required).

state
[*JobState*] The current state of the job.

power
[bool or None] If job is in the ready or power states, indicates whether the boards are power{ed,ing} on (True), or power{ed,ing} off (False). In other states, this value is None.

args, kwargs
The arguments to the alloc function which specifies the type/size of allocation requested and the restrictions on dead boards, links and torus connectivity.

allocated_machine
[*spalloc_server.configuration.Machine* or None] The machine the job has been allocated to run on (or None if not allocated yet).

boards
[set([(x, y, z), ...]) or None] The boards allocated to the job or None if not allocated.

periphery
[set([(x, y, z, `spalloc_server.links.Links`), ...]) or None] The links around the periphery of the job or None if not allocated.

torus
[*spalloc_server.coordinates.WrapAround* or None] Does the allocated set of boards have wrap-around links? None if not allocated.

width, height
[int or None] The dimensions of the SpiNNaker network in the allocated boards or None if not allocated any boards.

connections
[{(x, y): hostname, ...} or None] If boards are allocated, gives the mapping from chip coordinate to Ethernet connection hostname.

bmp_requests_until_ready
[int] A counter incremented whenever a BMP command is started and decremented when the command completes. When this counter reaches zero, the user sets the state of the job to `JobState.ready`.

__init__(*_id*, *owner*, *start_time=None*, *keepalive=60.0*, *lasthost=None*, *state=JobState.queued*, *power=None*, *args=()*, *kwargs=None*, *allocated_machine=None*, *boards=None*, *periphery=None*, *torus=None*, *width=None*, *height=None*, *connections=None*, *bmp_requests_until_ready=0*)

__weakref__
list of weak references to the object (if defined)

3.1.4 Job scheduling and allocation logic (`job_queue`)

A multi-machine and job queueing and allocation mechanism.

class JobQueue(*on_allocate*, *on_free*, *on_cancel*, *seconds_before_free=30*)

A mechanism for matching incoming allocation requests (jobs) to a set of available machines.

For every *_Machine* being managed this object contains a queue of outstanding jobs and an *spalloc_server.allocator.Allocator* which manages allocation of jobs onto that machine. A simplistic scheduling mechanism is used (see *_enqueue_job()*) which simultaneously enqueues all jobs to every candidate machine the job could possibly fit on. The first machine to accept the job is allocated the job (all other machines which subsequently encounter the job in their queues must skip it).

Though this object is entirely single threaded (and not thread safe!), callbacks (`on_allocate`, `on_free` and `on_cancel`) are used to indicate when a job is queued, allocated or cancelled.

A new job may be considered 'queued' the moment the *create_job()* method is called. At some point in the future, possibly before *create_job()* returns, the `on_allocate` or `on_cancel` callbacks will be called to indicate the job has either successfully been allocated to a machine or been cancelled (e.g. due to unavailability of a suitable machine or *destroy_job()* being called).

Once the `on_allocate()` callback has been called for a job, a job may be considered to be allocated to the specific machine indicated in the callback. This remains true until the `on_free()` callback is produced (as a result of calling `destroy_job`).

__init__(*on_allocate*, *on_free*, *on_cancel*, *seconds_before_free=30*)

Create a new (empty) job queue with no machines.

Parameters

on_allocate
[f(job_id, machine_name, boards, periphery, torus)] A callback function which is called when a job is successfully allocated resources on a machine.

job_id
[int] The job's unique identifier.

machine_name
[str] The name of the machine the job was allocated on.

boards
[set([(x, y, z), ...])] Enumerates the boards in the allocation.

periphery
[set([(x, y, z, `spalloc_server.links.Links`), ...])] Enumerates the links which leave the set of allocated boards.

torus
[*spalloc_server.coordinates.WrapAround*] Specifies which types of wrap-around links may be present.

on_free
[f(job_id, reason)] A callback called when a job which was previously allocated resources has had those resources withdrawn or freed them.

job_id
[int] The job's unique identifier.

reason
[str or None] A human readable string explaining why the job was freed or None if no reason was given.

on_cancel
[f(job_id, reason)] A callback called when a job which was previously queued is removed from the queue. This may be due to a lack of a suitable machine or due to user action.

job_id
[int] The job's unique identifier.

reason
[str or None] A human readable string explaining why the job was cancelled or None if no reason was given.

seconds_before_free
[int] The number of seconds between a board being freed and it becoming available again

`__enter__()`

This context manager will cause all changes to machines to be made atomically, without regenerating queues between each change.

This is useful when modifying multiple machines at once or destroying multiple jobs at once since it prevents jobs being allocated and then immediately destroyed.

Usage example:

```
>> q = JobQueue(...)
>> with q:
..     q.remove_machine("m")
..     q.add_machine(...)
```

Note: This context manager should be used sparingly as it causes all job queues to be regenerated from scratch when it exits.

`__getstate__`()

Called when pickling this object.

In Python 2, references to methods cannot be pickled. Since this object maintains a number of function pointers as callbacks (which may often be methods) we must remove these before pickling. When unpickling, these pointers should be recreated externally.

`_try_job`(*job*, *machine*)

Attempt to allocate a job on a given machine.

Returns

job_allocated
[bool] Was the job successfully allocated? If True, the internal metadata associated with the job is updated and the `on_allocate` callback is called.

`_enqueue_job`(*job*)

Either allocate or enqueue a new job.

Given a new job which is not yet running or enqueued use a simple scheduling mechanism to decided what to do with it:

- Attempt to allocate the job on each machine matching the job's requirements in turn.
- If no machine can allocate the job immediately, add the job to the queues of all machines which meet the job's requirements.

Parameters

job
[*_Job*] The job to attempt to enqueue or allocate.

`_process_queue`()

Try and process any queued jobs.

`_regenerate_queues`()

Regenerate all queues to account for any significant changes to the machines available.

This function clears all machine queues and then reinserts the jobs using `_enqueue_job`, potentially allocating the job to a running machine. Since the jobs are re-enqueued in the order they were supplied, the queueing priorities are unaffected.

`add_machine`(*name*, *width*, *height*, *tags=None*, *dead_boards=frozenset({})*, *dead_links=frozenset({})*)

Add a new machine for processing jobs.

Jobs are offered for allocation on machines in the order the machines are inserted to this list.

Parameters

name
[str] The name which identifies the machine.

width, height
[int] The dimensions of the machine in triads.

tags
[set([str, ...])] The set of tags jobs may select to indicate they wish to use this machine. Note that the tag default is given to any job whose tags are not otherwise specified. Defaults to set(["default"])

dead_boards
[set([(x, y, z), ...])] The boards in the machine which do not work.

dead_links
[set([(x, y, z, `spalloc_server.links.Links`), ...])] The board-to-board links in the machine which do not work.

See also:

`__enter__`
A context manager to allow atomic changes to be made to machines.

`move_machine_to_end`
Modify machine priorities

`modify_machine`
Modify certain machine parameters without removing and then re-adding it.

`remove_machine`
Remove a machine.

`move_machine_to_end`(*name*)

Move the specified machine to the end of the OrderedDict of machines.

Parameters

name
[str] The name of the machine to move.

`modify_machine`(*name*, *tags=None*, *dead_boards=None*, *dead_links=None*)

Make minor modifications to the description of an existing machine.

Note that any changes made will not impact already allocated jobs but may alter queued jobs.

Parameters

name
[str] The name of the machine to change.

tags
[set([str, ...]) or None] If not None, change the Machine's tags to match the supplied set.

dead_boards
[set([(x, y, z), ...])] If not None, change the set of dead boards in the machine.

dead_links
[set([(x, y, z, `spalloc_server.links.Links`), ...])] If not None, change the set of dead links in the machine.

`remove_machine`(*name*)

Remove a machine from the available set.

All jobs allocated on that machine will be freed and then the machine will be removed.

`create_job`(**args*, ***kwargs*)

Attempt to create a new job.

If no machine is immediately available to allocate the job the job is placed in the queues of all machines into which it can fit. The first machine to be able to allocate the job will get the job. This means that identical jobs are handled in a FIFO fashion but jobs which can only be executed on certain machines may be 'overtaken' by jobs which can run on machines the overtaken job cannot.

Note that during this method call the job may be allocated or cancelled and the associated callbacks called. Callers should be prepared for this eventuality. If no callbacks are produced the job has been queued.

This function is a thin wrapper around *spalloc_server.allocator.Allocator.alloc()* and thus accepts all arguments it accepts plus those named below.

Parameters

job_id
[int] **Mandatory.** A unique identifier for the job, supplied by the caller.

machine
[None or str] If not None, require that the job is executed on the specified machine. Not valid when tags are given.

tags
[None or set(['tag', ...])] The set of tags which any machine running this job must have. Not valid when machine is given. If None is supplied, only machines with the "default" tag will be used (unless machine is specified).

`destroy_job`(*job_id*, *reason=None*)

Destroy a queued or allocated job.

If the job is already allocated, this frees the job resulting in the `on_free` callback being called. If the job is queued, this removes the job from all queues and the `on_cancel` callback being called.

Parameters

job_id
[int] The ID of the job to destroy.

reason
[str or None] *Optional.* A human-readable reason that the job was destroyed.

`check_free`()

Check for freed machines that are now available

`__weakref__`

list of weak references to the object (if defined)

`class _Job`(*_id*, *pending=True*, *machine_name=None*, *tags=frozenset({})*, *args=()*, *kwargs=None*, *machine=None*, *allocation_id=None*)

The internal state representing a job.

Attributes

id
[int] A unique ID assigned to the job.

pending
[bool] If True, the job is currently queued for execution, if False the job has been allocated.

machine_name
[str or None] The machine this job must be executed on or None if any machine with matching tags is sufficient.

tags
[set([str, ...]) or None] The set of tags required of any machine the job is to be executed on. If None, only machines with the "default" tag will be used.

args, kwargs
[tuple, dict] The arguments to the alloc function for this job.

machine
[*_Machine* or None] The machine the job has been allocated on.

allocation_id
[int or None] The allocation ID for the Job's allocation.

`__init__`(*_id*, *pending=True*, *machine_name=None*, *tags=frozenset({})*, *args=()*, *kwargs=None*, *machine=None*, *allocation_id=None*)

`__repr__`()
Return repr(self).

`__weakref__`
list of weak references to the object (if defined)

`class _Machine`(*name*, *tags*, *allocator*, *queue=None*)
Internal data which maintains state information about machine on which jobs may run.

Attributes

name
[str] The name of the machine.

tags
[set([str, ...])] The set of tags the machine has. For a job to be allocated on a machine all of its tags must also be tags of the machine.

allocator
[*spalloc_server.allocator.Allocator*] An allocator for boards in this machine.

queue
[deque([*_Job*, ...])] A queue for jobs tentatively scheduled for this machine. Note that a job may be present in many queues at once. The first machine to accept the job is the only one which may process it.

`__init__`(*name*, *tags*, *allocator*, *queue=None*)

`__repr__`()
Return repr(self).

`__weakref__`
list of weak references to the object (if defined)

3.1.5 Machine resource allocation at the board-granularity (`allocator`)

Algorithm/data structure for allocating boards in SpiNNaker machines at the granularity of individual SpiNNaker boards and with awareness of the functionality of a machine.

class Allocator(*width*, *height*, *dead_boards=None*, *dead_links=None*, *next_id=1*, *seconds_before_free=30*)

Performs high-level allocation of SpiNNaker boards from a larger, possibly faulty, toroidal machine.

Internally this object uses a *spalloc_server.pack_tree.PackTree* to allocate rectangular blocks of triads in a machine. A *_CandidateFilter* to restrict the allocations made by *PackTree* to those which match the needs of the user (e.g. specific connectivity requirements).

The allocator can allocate either rectangular blocks of triads or individual boards. When allocating individual boards, the allocator allocates a 1x1 triad block from the *PackTree* and returns one of the boards from that block. Subsequent single-board allocations will use up spare boards left in triads allocated for single-board allocations before allocating new 1x1 triads.

__init__(*width*, *height*, *dead_boards=None*, *dead_links=None*, *next_id=1*, *seconds_before_free=30*)

Parameters

width, height
[int] Dimensions of the machine in triads.

dead_boards
[set([(x, y, z), ...])] The set of boards which are dead and which must not be allocated.

dead_links :set([(x, y, z, :py:class:`spalloc_server.links.Links`), ...])
The set of links leaving boards which are known not to be working. Connections to boards in the set dead_boards are assumed to be dead and need not be included in this list. Note that both link directions must be flagged as dead (if the link is bidirectionally down).

next_id
[int] The ID of the next allocation to be made.

seconds_before_free
[int] The number of seconds between a board being freed and it becoming available again

__getstate__()

Called when pickling this object.

This object may only be pickled once *stop()* and *join()* have returned.

__setstate__(*state*)

Called when unpickling this object.

Note that though the object must be pickled when stopped, the unpickled object will start running immediately.

_alloc_triads_possible(*width*, *height*, *max_dead_boards=None*, *max_dead_links=None*, *require_torus=False*, *min_ratio=0.0*)

Is it guaranteed that the specified allocation *could* succeed if enough of the machine is free?

This function may be conservative. If the specified request would fail when no resources have been allocated, we return False, even if some circumstances the allocation may succeed. For example, if one board in each of the four corners of the machine is dead, no allocation with max_dead_boards==0 can succeed when the machine is empty but may succeed if some other allocation has taken place.

Parameters

width, height
[int] The size of the block to allocate, in triads.

max_dead_boards
[int or None] The maximum number of broken or unreachable boards to allow in the allocated region. If None, any number of dead boards is permitted, as long as the board on the bottom-left corner is alive (Default: None).

max_dead_links
[int or None] The maximum number of broken links allow in the allocated region. When require_torus is True this includes wrap-around links, otherwise peripheral links are not counted. If None, any number of broken links is allowed. (Default: None).

require_torus
[bool] If True, only allocate blocks with torus connectivity. In general this will only succeed for requests to allocate an entire machine (when the machine is otherwise not in use!). (Default: False)

min_ratio
[float] Ignored.

Returns

bool

See also:

alloc_possible
The (public) wrapper which also supports checking triad allocations.

`_alloc_triads`(*width*, *height*, *max_dead_boards=None*, *max_dead_links=None*, *require_torus=False*, *min_ratio=0.0*)

Allocate a rectangular block of triads of interconnected boards.

Parameters

width, height
[int] The size of the block to allocate, in triads.

max_dead_boards
[int or None] The maximum number of broken or unreachable boards to allow in the allocated region. If None, any number of dead boards is permitted, as long as the board on the bottom-left corner is alive (Default: None).

max_dead_links
[int or None] The maximum number of broken links allow in the allocated region. When require_torus is True this includes wrap-around links, otherwise peripheral links are not counted. If None, any number of broken links is allowed. (Default: None).

require_torus
[bool] If True, only allocate blocks with torus connectivity. In general this will only succeed for requests to allocate an entire machine (when the machine is otherwise not in use!). (Default: False)

min_ratio
[float] Ignored.

Returns

(allocation_id, boards, periphery, torus) or None
If the allocation was successful a four-tuple is returned. If the allocation was not successful None is returned.

The `allocation_id` is an integer which should be used to free the allocation with the *free()* method. `boards` is a set of (x, y, z) tuples giving the locations of the (working) boards in the allocation. `periphery` is a set of (x, y, z, link) tuples giving the links which leave the allocated region. `torus` is True iff at least one torus link is working.

See also:

alloc
The (public) wrapper which also supports allocating individual

boards.

_alloc_boards_possible(*boards*, *min_ratio=0.0*, *max_dead_boards=None*, *max_dead_links=None*, *require_torus=False*)

Is it guaranteed that the specified allocation *could* succeed if enough of the machine is free?

This function may be conservative. If the specified request would fail when no resources have been allocated, we return False, even if some circumstances the allocation may succeed. For example, if one board in each of the four corners of the machine is dead, no allocation with max_dead_boards==0 can succeed when the machine is empty but may succeed if some other allocation has taken place.

Parameters

boards
[int] The *minimum* number of boards, must be at least 1. Note that if only 1 board is required, `_alloc_board` would be a more appropriate function since this function may return as many as three boards when only a single one is requested.

min_ratio
[float] The aspect ratio which the allocated region must be 'at least as square as'. Set to 0.0 for any allowable shape.

max_dead_boards
[int or None] The maximum number of broken or unreachable boards to allow in the allocated region. If None, any number of dead boards is permitted, as long as the board on the bottom-left corner is alive (Default: None).

max_dead_links
[int or None] The maximum number of broken links allow in the allocated region. When require_torus is True this includes wrap-around links, otherwise peripheral links are not counted. If None, any number of broken links is allowed. (Default: None).

require_torus
[bool] If True, only allocate blocks with torus connectivity. In general this will only succeed for requests to allocate an entire machine (when the machine is otherwise not in use!). (Default: False)

Returns

bool

See also:

alloc_possible
The (public) wrapper which also supports checking triad allocations.

_alloc_boards(*boards*, *min_ratio=0.0*, *max_dead_boards=None*, *max_dead_links=None*, *require_torus=False*)

Allocate a rectangular block of triads with at least the specified number of boards which is 'at least as square' as the specified aspect ratio.

Parameters

boards
[int] The *minimum* number of boards, must be at least 1. Note that if only 1 board is required, `_alloc_board` would be a more appropriate function since this function may return as many as three boards when only a single one is requested.

min_ratio
[float] The aspect ratio which the allocated region must be 'at least as square as'. Set to 0.0 for any allowable shape.

max_dead_boards
[int or None] The maximum number of broken or unreachable boards to allow in the allocated region. If None, any number of dead boards is permitted, as long as the board on the bottom-left corner is alive (Default: None).

max_dead_links
[int or None] The maximum number of broken links allow in the allocated region. When require_torus is True this includes wrap-around links, otherwise peripheral links are not counted. If None, any number of broken links is allowed. (Default: None).

require_torus
[bool] If True, only allocate blocks with torus connectivity. In general this will only succeed for requests to allocate an entire machine (when the machine is otherwise not in use!). (Default: False)

Returns

(allocation_id, boards, periphery, torus) or None
If the allocation was successful a four-tuple is returned. If the allocation was not successful None is returned.

The `allocation_id` is an integer which should be used to free the allocation with the *free()* method. `boards` is a set of (x, y, z) tuples giving the locations of the (working) boards in the allocation. `periphery` is a set of (x, y, z, link) tuples giving the links which leave the allocated region. `torus` is a *WrapAround* value indicating torus connectivity when at least one torus may exist.

See also:

alloc
The (public) wrapper which also supports allocating individual

`boards.`

`_alloc_board_possible`(*x=None*, *y=None*, *z=None*, *max_dead_boards=None*, *max_dead_links=None*, *require_torus=False*, *min_ratio=0.0*)

Is it guaranteed that the specified allocation *could* succeed if enough of the machine is free?

Parameters

x, y, z
[ints or None] If specified, requests a specific board.

max_dead_boards
[int or None] Ignored.

max_dead_links
[int or None] Ignored.

require_torus
[bool] Must be False.

min_ratio
[float] Ignored.

Returns

bool

See also:

alloc_possible
The (public) wrapper which also supports checking board allocations.

_alloc_board(*x=None*, *y=None*, *z=None*, *max_dead_boards=None*, *max_dead_links=None*, *require_torus=False*, *min_ratio=0.0*)

Allocate a single board, optionally specifying a specific board to allocate.

Parameters

x, y, z
[ints or None] If None, an arbitrary free board will be returned if possible. If all are defined, attempts to allocate the specific board requested if available and working.

max_dead_boards
[int or None] Ignored.

max_dead_links
[int or None] Ignored.

require_torus
[bool] Must be False.

min_ratio
[float] Ignored.

Returns

(allocation_id, boards, periphery, torus) or None
If the allocation was successful a four-tuple is returned. If the allocation was not successful None is returned.

The `allocation_id` is an integer which should be used to free the allocation with the *free()* method. `boards` is a set of (x, y, z) tuples giving the location of to allocated board. `periphery` is a set of (x, y, z, link) tuples giving the links which leave the board. `torus` is always *WrapAround.none* for single boards.

See also:

alloc
The (public) wrapper which also supports allocating triads.

_alloc_type(*x_or_num_or_width=None*, *y_or_height=None*, *z=None*, *max_dead_boards=None*, *max_dead_links=None*, *require_torus=False*, *min_ratio=0.0*)

Returns the type of allocation the user is attempting to make (and fails if it is invalid.

Usage:

```
a.alloc()  # Allocate any single board
a.alloc(1)  # Allocate any single board
a.alloc(3, 2, 1)  # Allocate the specific board (3, 2, 1)
a.alloc(4)  # Allocate at least 4 boards
a.alloc(2, 3, **kwargs)  # Allocate a 2x3 triad machine
```

Parameters

<nothing> OR num OR x, y, z OR width, height
If nothing, allocate a single board.

If num, allocate at least that number of boards. Special case: if 1, allocate exactly 1 board.

If x, y and z, allocate a specific board.

If width and height, allocate a block of this size, in triads.

max_dead_boards
[int or None] The maximum number of broken or unreachable boards to allow in the allocated region. If None, any number of dead boards is permitted, as long as the board on the bottom-left corner is alive (Default: None).

max_dead_links
[int or None] The maximum number of broken links allow in the allocated region. When require_torus is True this includes wrap-around links, otherwise peripheral links are not counted. If None, any number of broken links is allowed. (Default: None).

require_torus
[bool] If True, only allocate blocks with torus connectivity. In general this will only succeed for requests to allocate an entire machine (when the machine is otherwise not in use!). Must be False when allocating boards. (Default: False)

Returns

_AllocationType

`alloc_possible`(**args*, ***kwargs*)

Is the specified allocation actually possible on this machine?

Usage:

```
a.alloc_possible()  # Can allocate a single board?
a.alloc_possible(1)  # Can allocate a single board?
a.alloc_possible(4)  # Can allocate at least 4 boards?
a.alloc_possible(3, 2, 1)  # Can allocate a board (3, 2, 1)?
a.alloc_possible(2, 3, **kwargs)  # Can allocate 2x3 triads?
```

Parameters

<nothing> OR num OR x, y, z OR width, height
If nothing, allocate a single board.

If num, allocate at least that number of boards. Special case: if 1, allocate exactly 1 board.

If x, y and z, allocate a specific board.

If width and height, allocate a block of this size, in triads.

min_ratio
[float] The aspect ratio which the allocated region must be 'at least as square as'. Set to 0.0 for any allowable shape. Ignored when allocating single boards or specific rectangles of triads.

max_dead_boards
[int or None] The maximum number of broken or unreachable boards to allow in the allocated region. If None, any number of dead boards is permitted, as long as the board on the bottom-left corner is alive (Default: None).

max_dead_links
[int or None] The maximum number of broken links allow in the allocated region. When require_torus is True this includes wrap-around links, otherwise peripheral links are not counted. If None, any number of broken links is allowed. (Default: None).

require_torus
[bool] If True, only allocate blocks with torus connectivity. In general this will only succeed for requests to allocate an entire machine (when the machine is otherwise not in use!). Must be False when allocating boards. (Default: False)

Returns

bool

`alloc`(**args*, ***kwargs*)

Attempt to allocate a board or rectangular region of triads of boards.

Usage:

```
a.alloc()  # Allocate a single board
a.alloc(1)  # Allocate a single board
a.alloc(4)  # Allocate at least 4 boards
a.alloc(3, 2, 1)  # Allocate a specific board (3, 2, 1)
a.alloc(2, 3, **kwargs)  # Allocate a 2x3 triad machine
```

Parameters

<nothing> OR num OR x, y, z OR width, height
If all None, allocate a single board.

If num, allocate at least that number of boards. Special case: if 1, allocate exactly 1 board.

If x, y and z, allocate a specific board.

If width and height, allocate a block of this size, in triads.

min_ratio
[float] The aspect ratio which the allocated region must be 'at least as square as'. Set to 0.0 for any allowable shape. Ignored when allocating single boards or specific rectangles of triads.

max_dead_boards
[int or None] The maximum number of broken or unreachable boards to allow in the allocated region. If None, any number of dead boards is permitted, as long as the board on the bottom-left corner is alive (Default: None).

max_dead_links
[int or None] The maximum number of broken links allow in the allocated region. When require_torus is True this includes wrap-around links, otherwise peripheral links are not counted. If None, any number of broken links is allowed. (Default: None).

require_torus
[bool] If True, only allocate blocks with torus connectivity. In general this will only succeed for requests to allocate an entire machine (when the machine is otherwise not in use!). Must be False when allocating boards. (Default: False)

Returns

(allocation_id, boards, periphery, torus) or None
If the allocation was successful a four-tuple is returned. If the allocation was not successful None is returned.

The `allocation_id` is an integer which should be used to free the allocation with the *free()* method. `boards` is a set of (x, y, z) tuples giving the locations of to allocated boards. `periphery` is a set of (x, y, z, link) tuples giving the links which leave the allocated set of boards. `torus` is a *WrapAround* value indicating torus connectivity when at least one torus may exist.

free(*allocation_id*)

Free the resources consumed by the specified allocation.

Parameters

allocation_id
[int] The ID of the allocation to free.

check_free()

Free any of the items on the "to free" list that have expired

_free_next()

Free the next item on the "to_free" list

__weakref__

list of weak references to the object (if defined)

class _AllocationType(*value*)

Type identifiers for allocations.

triads = 0

A rectangular block of triads.

board = 1

A single board.

boards = 2

Two or more boards, to be allocated as triads.

This type only returned by *Allocator._alloc_type()* and is never used as an allocation type.

class _CandidateFilter(*width*, *height*, *dead_boards*, *dead_links*, *max_dead_boards*, *max_dead_links*, *require_torus*, *expected_boards=None*)

A filter which, given a rectangular region of triads, will check to see if it meets some set of criteria.

If any candidate is accepted the following attributes are set according to the last accepted candidate.

Attributes

boards
[set([(x, y, z), ...])] The working boards present in the accepted candidate. None if no candidate has been accepted.

periphery
[set([(x, y, z, spalloc_server.links.Links), ...])] The links around the periphery of the selection of boards which should be disabled to isolate the system. None if no candidate has been accepted.

torus
[*WrapAround*] Describes the types of wrap-around links the candidate has.

__init__(*width*, *height*, *dead_boards*, *dead_links*, *max_dead_boards*, *max_dead_links*, *require_torus*, *expected_boards=None*)

Create a new candidate filter.

Parameters

width, height
[int] Dimensions (in triads) of the system within which candidates are being chosen.

dead_boards
[set([(x, y, z), ...])] The set of boards which are dead and which must not be allocated.

dead_links
[set([(x, y, z, spalloc_server.links.Links), ...])] The set of links leaving boards which are known not to be working. Connections to boards in the set dead_boards are assumed to be dead and need not be included in this list. Note that both link directions must be flagged as dead (if the link is bidirectionally down).

max_dead_boards
[int or None] The maximum number of broken or unreachable boards to allow in the allocated region. If None, any number of dead boards is permitted, as long as the board on the bottom-left corner is alive (Default: None).

max_dead_links
[int or None] The maximum number of broken links allow in the allocated region. When require_torus is True this includes wrap-around links, otherwise peripheral links are not counted. If None, any number of broken links is allowed. (Default: None).

require_torus
[bool] If True, only allocate blocks with torus connectivity. In general this will only succeed for requests to allocate an entire machine (when the machine is otherwise not in use!). (Default: False)

expected_boards
[int or None] If given, specifies the number of boards which are expected to be in a candidate. This ensures that when an over-allocation is made, the max_dead_boards figure is offset by any over-allocation.

If None, assumes the candidate width * candidate height * 3.

__weakref__

list of weak references to the object (if defined)

_enumerate_boards(*x*, *y*, *width*, *height*)

Starting from board (x, y, 0), enumerate as many reachable and working boards as possible within the rectangle width x height triads.

Returns

set([(x, y, z), ...])

`_classify_links`(*boards*)

Get a list of links of various classes connected to the supplied set of boards.

Parameters

boards
[set([(x, y, z), ...])] A set of fully-connected, alive boards.

Returns

alive
[set([(x, y, z, `links.Links`), ...])] Links which are working and connect one board in the set to another.

wrap
[set([(x, y, z, `links.Links`), ...])] Working links between working boards in the set which wrap-around the toroid.

dead
[set([(x, y, z, `links.Links`), ...])] Links which are not working and connect one board in the set to another.

dead_wrap
[set([(x, y, z, `links.Links`), ...])] Dead links between working boards in the set which wrap-around the toroid.

periphery
[set([(x, y, z, `links.Links`), ...])] Links are those which connect from one board in the set to a board outside the set. These links may be dead or alive.

wrap_around_type
[*WrapAround*] What types of wrap-around links are present (making no distinction between dead and alive links)?

`__call__`(*x*, *y*, *width*, *height*)

Test whether the region specified meets the stated requirements.

If True is returned, the set of boards in the region is stored in self.boards and the set of links on the periphery are stored in self.periphery.

3.1.6 Machine resource allocation at the triad-granularity (`pack_tree`)

An algorithm/datastructure for allocating/packing rectangles into a fixed 2D space.

This algorithm is used to allocate triads of boards in SpiNNaker systems butis otherwise a relatively generic 2D packing algorithm.

`class PackTree`(*x*, *y*, *width*, *height*)

A tree-based datastructure for allocating/packing rectangles into a fixed 2D space.

This tree structure is used to allocate/pack rectangular subregions of SpiNNaker machine in a fashion similar to this lightmap packing algorithm. It is certainly not the most efficient or flexible packing algorithm available but due to time constraints it is ideal due to its simplicity.

`__init__`(*x*, *y*, *width*, *height*)

Defines a region of which may be allocated and/or divided in two.

Parameters

x, y
[int] The (absolute) location of the bottom left corner of the region.

width, height
[int] The dimensions of the region.

`__contains__`(*xy*)

Test whether a coordinate is inside this region.

`hsplit`(*y*)

Split this node along the X axis.

The bottom half of split will be just before the "y" position.

```
   +-----------+
   |           |
---+-----------+---
   |           |
   +-----------+
```

`vsplit`(*x*)

Split this node along the Y axis.

The left half of split will be just before the "x" position.

```
      |
+-----+-----+
|     |     |
|     |     |
|     |     |
+-----+-----+
      |
```

`alloc`(*width*, *height*, *candidate_filter=None*)

Attempt to allocate a rectangular region of a specified size.

Parameters

width, height
[int] The dimensions of the region to attempt to allocate. Must be strictly 1x1 or greater.

candidate_filter
[None or function(x, y, w, h) -> bool] A function which will be called with candidate allocations. If the function returns False, the allocation is rejected and the allocator will attempt to find another. If the function returns True, the allocator will then create the allocation. This function may, for example, check that the suggested region is fully connected or does not have too many faults.

If this argument is None (the default) the first candidate allocation found will be returned.

Returns

allocation
[(x, y) or None] If the allocation request was met, a tuple giving the position of the bottom-left corner of the allocation.

If the request could not be met, None is returned and no allocation is made.

`alloc_area`(*area*, *min_ratio=0.0*, *candidate_filter=None*)

Attempt to allocate a rectangular region with at least the specified area which is 'at least as square' as the specified aspect ratio.

Parameters

area
[int] The *minimum* area to allocate, must be at least 1.

min_ratio
[float] The aspect ratio which the allocated region must be 'at least as square as'. Set to 0.0 for any allowable shape.

candidate_filter
[None or function(x, y, w, h) -> bool] A function which will be called with candidate allocations. If the function returns False, the allocation is rejected and the allocator will attempt to find another. If the function returns True, the allocator will then create the allocation. This function may, for example, check that the suggested region is fully connected or does not have too many faults.

If this argument is None (the default) the first candidate allocation found will be returned.

Returns

allocation
[(x, y, w, h) or None] If the allocation request was met, a tuple giving the position of the bottom-left corner, width and height of the allocation is returned.

If the request could not be met, None is returned and no allocation is made.

`request`(*x*, *y*)

Request the allocation of a specific 1x1 block.

This function may be useful when, e.g., specific boards are required for testing.

Returns

allocation
[(x, y) or None] If the request request was met, the coordinates passed in are returned.

If the request could not be met, None is returned and no allocation is made.

`free`(*x*, *y*)

Free a previous allocation, allowing the space to be reused.

Parameters

x, y
[int] The bottom-left corner of the allocation.

`__weakref__`

list of weak references to the object (if defined)

`exception FreeError`

Thrown when attempting to free a region fails.

`__weakref__`

list of weak references to the object (if defined)

3.1.7 Number of baords to rectangle conversion (area_to_rect)

Utility functions for working out sensible sizes of machine to allocate given a minimum number of boards and worst-case aspect ratio.

area_to_rect(*area*, *bound_width*, *bound_height*, *min_ratio=0.0*)

Given an area requirement, select a rectangular subregion of the given width and height bounds whose aspect ratio (when rotated into a landscape orientation) is at least that specified.

Parameters

area
[int] Lower-bound on area requirement.

bound_width
[int] Upper-bound on the width of the rectangle.

bound_height
[int] Upper-bound on the height of the rectangle.

min_ratio
[float] Require that the area of the selected rectangle be 'at least as square' as the supplied aspect ratio (h/w). By convention this is specified in the range 0.0 <= min_ratio <= 1.0 but values > 1.0 are also accepted (and are internally flipped to the conventional range).

Returns

(width, height) or None
The dimensions of a rectangle meeting the request or None if no such rectangle is possible. When sensible and possible, the rectangle allocated will have the same width and/or height as the bounds.

squarest_rectangle(*area*)

Given a specific area, calculate the squarest possible rectangle with exactly that area, preferring landscape rectangles.

Returns

(w, h)
Width and height of a rectangle with the area specified where w and h are as similar as possible and w >= h.

rectangle_with_aspect_ratio(*area*, *ratio*)

Return a (landscape) rectangle with at least the specified area and aspect ratio.

Note that the returned rectangles are not at all guaranteed to be the 'squarest' possible, just greater than the specified ratio. This is intended to generate alternatives to the very wide rectangles produced by squarest_rectangle when prime (or 'nearly prime') numbers of boards are requested.

Parameters

area
[int] Minimum area the rectangle must cover.

ratio
[float] The minimum aspect ratio (h/w) the rectangle should have. Must be in the range 0.0 < ratio <= 1.0.

Returns

(width, height)
The dimensions of the rectangle selected. Rectangles are always square or landscape (i.e. w >= h).

3.1.8 Asynchronous BMP Controller (async_bmp_controller)

Provide (basic) asynchronous control over a BMP responsible for controlling a whole rack.

class AsyncBMPController(*hostname*, *on_thread_start=None*)

An object which provides an asynchronous interface to a power and link control commands of a SpiNNaker BMP.

Since BMP commands, particularly power-on commands, take some time to complete, it is desirable for them to be executed asynchronously. This object uses a SpiNNMan `Transceiver` object to communicate with a BMP controlling a single frame of boards.

Power and link configuration commands are queued and executed in a background thread. When a command completes, a user-supplied callback is called.

Sequential power commands of the same type (on/off) are coalesced into a single power on command. When a power command is sent, all previous link configuration commands queued for that board are skipped. Additionally, all power commands are completed before link configuration commands are carried out.

__init__(*hostname*, *on_thread_start=None*)

Start a new asynchronous BMP Controller

Parameters

hostname
[str] The hostname/IP of the BMP to connect to.

on_thread_start
[function() or None] *Optional.* A function to be called by the controller's background thread before it starts. This can be used to ensure proper sequencing/handing-over between two AsyncBMPControllers connected to the same machine.

__enter__()

When used as a context manager, make requests 'atomic'.

add_requests(*atomic_requests*)

Add an atomic request to be performed

stop()

Stop the background thread, as soon as possible after completing all queued actions.

join()

Wait for the thread to actually stop.

_set_link_state(*link*, *enable*, *board*)

Set the power state of a link.

Parameters

- **link** (*value in Links enum*) – The link (direction) to set the enable-state of.
- **state** (*bool*) – What to set the state to. True for on, False for off.
- **board** (*int or iterable*) – Which board or boards to set the link enable-state of.

_run()

The background thread for interacting with the BMP.

__weakref__

list of weak references to the object (if defined)

class AtomicRequests(*on_done*)

A list of requests that need to be done atomically; these are carried out in order of power on, link enable or disable, power off

__init__(*on_done*)

class LinkRequest(*board*, *link*, *enable*)

Requests that a specific board should have its power state set to a particular value.

Parameters

board
[int] Board whose link should be blocked/unblocked

link
[spalloc_server.links.Link] The link whose state should be changed

enable
[bool] State of the link: Enabled (True), disabled (False).

CHAPTER

FOUR

INDICIES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

C

S

INDEX

Symbols

A

B

C

D

F

G

H

I

J

L

M

N

P

Q

R

S

T

U

V

W

X

Y