Network Tester Documentation

Release 0.1.5

Jonathan Heathcote

July 09, 2015

Contents

'Network Tester' is a library designed to enable experimenters to quickly and easily describe and run experiments on SpiNNaker's interconnection network. In particular, network tester is designed to make recreating traffic loads similar to typical neural software straight-forward. Such network loads feature a fixed set of vertices (cores) which produce SpiNNaker packets which are then multicast to a fixed set of vertices.

The following is a (non-exhaustive) list of the kinds of experiments which can be performed with 'Network Tester':

- Determining how a network copes with different rates and patterns of packet generation. For example to determining the maximum speed at which a particular neural simulation may run on SpiNNaker without dropping packets.
- Determining the effectiveness of place and route algorithms by finding 'hot-spots' in the network.
- Characterising the behaviour of the network in the presence of locally and globally synchronised bursting traffic.

CHAPTER 1

Getting started

1.1 Getting started with Network Tester

This guide aims to introduce the basic concepts required to start working with Network Tester. Complete detailed documentation can be found in the *API documentation*.

This guide introduces the key concepts and terminology used by Network Tester and walks through the creation of a simple experiment. In this experiment we measure how the SpiNNaker network handles the load produced by a small random network of packet generators.

1.1.1 Introduction & Terminology

Network Tester is a Python library and native SpiNNaker application which generates artificial network traffic in SpiNNaker while recording network performance metrics. In particular, network tester is designed to recreate traffic loads similar to neural applications running on SpiNNaker. This means that the connectivity of a network loads remains fixed throughout experiments but the rate and pattern of injected packets can be varied.

A Network Tester 'experiment' consists of a network description along with a series of experimental 'groups' during which different traffic patterns or network parameters are applied in sequence.

A network is described as a set of 'vertices', representing SpiNNaker application cores, connected by a set of multicast 'nets' which represent streams of packets sourced by a traffic generator in one vertex and sunk by traffic consumers in another set of vertices. A single vertex (core) may be associated with many nets and thus may contain multiple traffic generators and traffic consumers.

Each experimental group consists of a period of traffic generation and consumption according to a particular set of parameters. A typical experiment may consist of several groups with a single parameter being changed between groups. Various metrics (including packet counts and router diagnostic counters) can be recorded individually for each group and then collected after the experiment into easily manipulated Numpy arrays.

Once an experiment has been defined, Network Tester will automatically load and configure traffic generation software onto a target SpiNNaker machine and execute each experimental group in sequence before collecting results.

1.1.2 Installation

The latest stable version of Network Tester library may be installed from PyPI using:

```
$ pip install network_tester
```

The standard installation includes precompiled SpiNNaker binaries and should be ready to use 'out of the box'.

1.1.3 Defining a network

First we must create a new *Experiment* object which takes a SpiNNaker IP address or hostname as its argument:

```
>>> from network_tester import Experiment
>>> e = Experiment("192.168.240.253")
```

The first task when defining an experiment is to define a set of vertices and nets between them. In this example we'll create a network with 64 randomly connected vertices. First the vertices are created using *new_vertex()*:

```
>>> vertices = [e.new_vertex() for _ in range(64)]
```

Next we create a single net for each vertex using *new_vertex()* which connects to eight randomly selected vertices:

```
>>> import random
>>> nets = [e.new_net(vertex, random.sample(vertices, 8))
...         for vertex in vertices]
```

By default, the vertices and nets we've defined will be automatically placed and routed in the SpiNNaker machine before we run the experiment. For greater control over this process, see *place_and_route()*.

1.1.4 Controling packet generation

Every net has its own traffic generator on its source vertex. These traffic generators can be configured to produce a range of different traffic patterns but in this example we'll configure the traffic generators to produce a simple Bernoulli traffic pattern. In a Bernoulli distribution, each traffic generator will produce a single packet (or not) with a specific probability at a regular interval (the 'timestep'). By varying the probability of a packet being generated we can change the load our simple example exerts on the SpiNNaker network.

The timestep and packet generation probability are examples of some of the *experimental parameters* which can be controlled and varied during an experiment. These parameters can be controlled by setting attributes of the *Experiment* object or *Vertex* and *Net* objects returned by *new_vertex()* and *new_vertex()* respectively.

In our example we'll set the *timestep* to 10 microseconds meaning the packet generators in the experiment *may* generate a packet every 10 microseconds:

```
>>> e.timestep = 1e-5  # 10 microseconds (in seconds)
```

In our example experiment we'll change the probability of a packet being generated (thus changing the network load) and see how the network behaves. To do this we'll create a number of experimental groups with different probabilities:

```
>>> num_steps = 10
>>> for step in range(num_steps):
...     with e.new_group() as group:
...         e.probability = step / float(num_steps - 1)
...         group.add_label("probability", e.probability)
```

The *new_group()* method creates a new experimental *Group* object. When a *Group* object is used with a `with` statement it causes any parameters changed inside the `with` block to apply only to that experimental group. In this example we set the *probability* parameter to a different value for each group.

The *Group.add_label()* call is optional but adds a custom extra column to the results collected by Network Tester. In this case we add a "probability" column which we set to the probability used in that group. Though the results are automatically broken up into groups, this extra column makes it much easier to plot data straight out of the tool.

Note: Some parameters such as *timestep* are 'global' (i.e. they're the same for every net and vertex) and thus can only changed experiment-wide. Other parameters, such as *probability* can be set individually for different

vertices or nets. As a convenience, setting these parameters on the *Experiment* object sets the 'default' value for all vertices or nets. For example:

```
>>> for net in nets:
...     net.probability = 0.5
```

Is equivilent to:

```
>>> e.probability = 0.5
```

One last detail is to specify how long to run the traffic generators for each group using *duration*:

```
>>> e.duration = 0.1  # Run each group for 1/10th of a second
```

In experiments with highly static network loads it is important to 'warm up' the network to allow it to reach a stable state before recording results for each group. Such a warmup can be added using *warmup*:

```
>>> e.warmup = 0.05  # Warm up without recording results for 1/20th of a second
```

Finally, Network Tester does not attempt to maintain clock synchronisation in long experiments in large SpiNNaker machines. As a result, some traffic generators may finish before others causing artefacts in the results. To help alleviate this a 'cool down' period can be added after each group using the *cooldown* parameter. During the cool down period the traffic generators continue to run but no further results are recorded.

```
>>> e.cooldown = 0.01  # Cool down without recording results for 1/100th of a second
```

A complete list of the available parameters is *available in the API documentation*.

1.1.5 Recording results

Various metrics may be recorded during an experiment. In our example we'll simply record the number of packets received by the sinks of each net. Attributes of the *Experiment* object whose names start with `record_` are used to select what metrics are recorded, in this case we enable *record_received*:

```
>>> e.record_received = True
```

The full set of recordable metrics is *enumerated in the API documentation* and includes per-net packet counts, router diagnostic counters and packet reinjection statistics.

By default, the recorded metrics are sampled once at the end of each experimental group's execution but they can alternatively be sampled at a regular interval (see the *record_interval* parameter).

Note: Unlike the experimental parameters, the set of recorded metrics is fixed for the whole experiment and cannot be changed within groups. Further, individual nets, vertices or router's metrics cannot be enabled and disabled individually. Note, however, that *record_interval* is an experimental parameter and thus *can* be set independently for each group.

1.1.6 Running the experiment and plotting results

Once everything has been defined, the experiment is started using *run()*:

```
>>> results = e.run(ignore_deadline_errors=True)
```

Note that the `ignore_deadline_errors` option is enabled for this experiment. This is necessary since when the injected load is very high the load on the traffic sinks causes the Network Tester to miss its realtime deadlines. In experiments where the network is not expected to saturate this option should *not* be used.

Note: Running an experiment can take some time. To see informational messages indicating progress you can enable INFO messages in the Python `logging` module before calling *`run()`*:

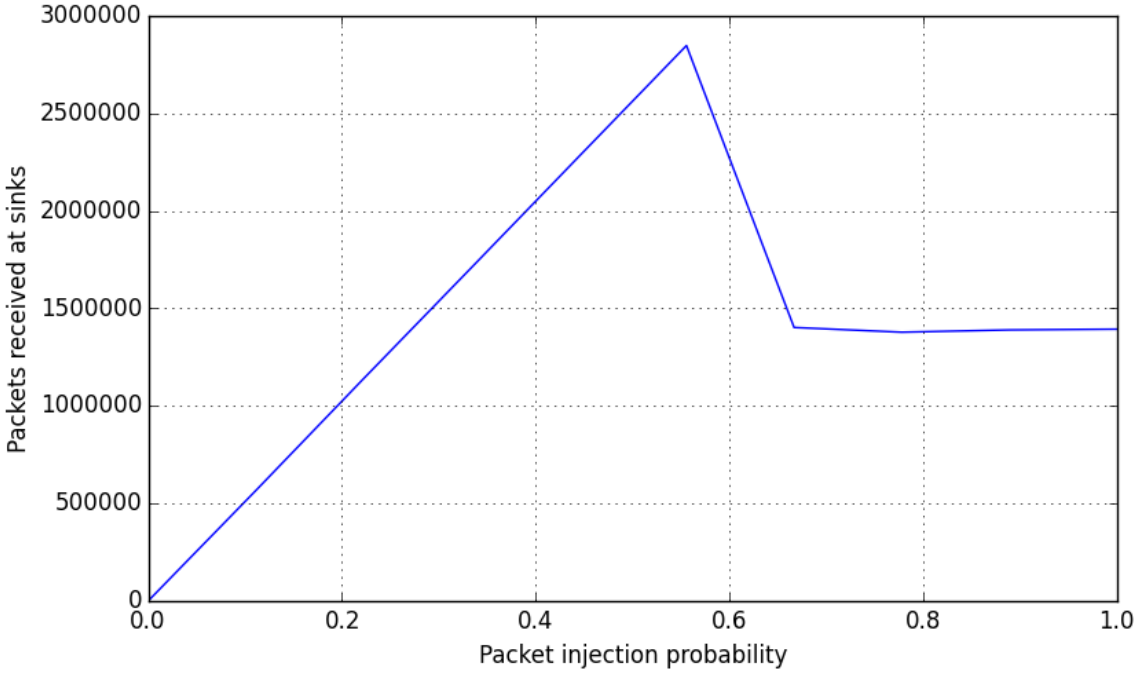
```
>>> import logging
>>> logging.basicConfig(level=logging.INFO)
```

The returned *`Results`* object provides a number of methods which present the recorded data in useful ways. In this case we're just interested in the overall behaviour of the network so we'll grab the *`totals()`*:

```
>>> totals = results.totals()
>>> totals.dtype.names
('probability', 'group', 'time', 'received')
>>> totals
[(0.0, <Group 0>, 0.1, 0.0)
 (0.1111111111111111, <Group 1>, 0.1, 566026.0)
 (0.2222222222222222, <Group 2>, 0.1, 1138960.0)
 (0.3333333333333333, <Group 3>, 0.1, 1707350.0)
 (0.4444444444444444, <Group 4>, 0.1, 2277734.0)
 (0.5555555555555556, <Group 5>, 0.1, 2847388.0)
 (0.6666666666666666, <Group 6>, 0.1, 1401762.0)
 (0.7777777777777778, <Group 7>, 0.1, 1377632.0)
 (0.8888888888888888, <Group 8>, 0.1, 1389261.0)
 (1.0, <Group 9>, 0.1, 1393182.0)]
```

We can then plot this data using `pyplot`:

```
>>> import matplotlib.pyplot as plt
>>> plt.plot(totals["probability"], totals["received"])
>>> plt.xlabel("Packet injection probability")
>>> plt.ylabel("Packets received at sinks")
>>> plt.show()
```



Alternatively, we can export the data as a CSV suitable for processing or plotting with another tool, for example R,

using the included *network_tester.to_csv()* function:

```
>>> from network_tester import to_csv
>>> print(to_csv(totals))
probability,group,time,received
0.0,0,0.1,0.0
0.1111111111111111,1,0.1,566026.0
0.2222222222222222,2,0.1,1138960.0
0.3333333333333333,3,0.1,1707350.0
0.4444444444444444,4,0.1,2277734.0
0.5555555555555556,5,0.1,2847388.0
0.6666666666666666,6,0.1,1401762.0
0.7777777777777778,7,0.1,1377632.0
0.8888888888888888,8,0.1,1389261.0
1.0,9,0.1,1393182.0
```

Note: Unlike the Numpy built-in `numpy.savetxt()` function, *to_csv()* automatically adds headers and correctly formats missing elements.

CHAPTER 2

API Reference

2.1 The Network Tester API

2.1.1 The `Experiment` Class

class network_tester.**Experiment**

Defines a network experiment to be run on a SpiNNaker machine.

An experiment consists of a fixed set of 'vertices' (*new_vertex()*) connected together by 'nets' (*new_net()*). Vertices correspond with SpiNNaker application cores running artificial traffic generators and the nets correspond with traffic flows between cores.

An experiment is broken up into 'groups' (*new_group()*), during which the traffic generators produce packets according to a specified traffic pattern. Within each group, metrics, such as packet counts, may be recorded. Though the placement of vertices and the routing of nets is fixed throughout an experiment, the rate and pattern with which which packets are produced can be varied between groups allowing, for example, different traffic patterns to be tested.

When the experiment is *run()*, appropriately-configured traffic generator applications will be loaded onto SpiNNaker and, after the experiment completes, the results are read back ready for analysis.

__init__(*hostname_or_machine_controller*)

Create a new network experiment on a particular SpiNNaker machine.

Example usage:

```
>>> import sys
>>> from network_tester import Experiment
>>> e = Experiment(sys.argv[1])  # Takes hostname as a CLI argument
```

The experimental parameters can be set by setting attributes of the *Experiment* instance like so:

```
>>> e = Experiment(...)
>>> # Set the probability of a packet being generated at the source
>>> # of each net every timestep
>>> e.probability = 1.0
```

Parameters **hostname_or_machine_controller** : str or rig.machine_control.MachineController

The hostname or MachineController of a SpiNNaker machine to run the experiment on.

allocations
A dictionary {*Vertex*: {resource: slice}, ...} or None.

Defines the resources allocated to each vertex. This must include exactly 1 unit of the `Cores` resource. Note that the allocation must define the resource allocation of *every* vertex. If None, calling *run()* or *place_and_route()* will cause all vertices to have their resources allocated automatically.

Setting this attribute will also set *routes* to None.

Any allocation must be valid for the `Machine` specified by the *machine* attribute. Core 0 must always be reserved for the monitor processor and, if packet reinjection is used or recorded (see *Experiment.reinject_packets*), core 1 must also be reserved for the packet reinjection application.

See also `rig.place_and_route.allocate()`.

machine
The `Machine` object describing the SpiNNaker system under test.

This property caches the machine description read from the machine to avoid repeatedly polling the SpiNNaker system.

new_group(*name=None*)
Define a new experimental group.

The experiment can be divided up into groups where the traffic pattern generated (but not the structure of connectivity) varies for each group. Results are recorded separately for each group and the network is drained of packets between groups.

The returned *Group* object can be used as a context manager within which experimental parameters specific to that group may be set, including per-vertex and per-net parameters. Note that parameters set globally for the experiment in particular group do not take precedence over per-vertex or per-net parameter settings.

For example:

```
>>> with e.new_group():
...     # Overrides default probability of sending a packet within
...     # the group.
...     e.probability = 0.5
...     # Overrides the probability for v2 within the group
...     v2.probability = 0.25
...     # Overrides the probability for n0 within the group
...     n0.probability = 0.4
```

Parameters **name**

Optional. A name for the group. If not specified the group will be given a number as its name. This name will be used in results tables.

Returns *Group*

An object representing the group.

new_net(*source*, *sinks*, *weight=1.0*, *name=None*)
Create a new net.

A net represents a flow of SpiNNaker packets from one source vertex to many sink vertices.

For example:

```
>>> # A net with v0 as a source and v1 as a sink.
>>> n0 = e.new_net(v0, v1)

>>> # Another net with v0 as a source and both v1 and v2 as sinks.
>>> n1 = e.new_net(v0, [v1, v2])
```

The experimental parameters for each net can also be overridden individually if desired. This will take precedence over any overridden values set for the source vertex of the net.

For example:

```
>>> # Net n0 will generate a packet in 80% of timesteps
>>> n0.probability = 0.8
```

Parameters **source** : *Vertex*

The source *Vertex* of the net. A stream of packets will be generated by this vertex and sent to all sinks.

Only *Vertex* objects created by this *Experiment* may be used.

sinks : *Vertex* or [*Vertex*, ...]

The sink *Vertex* or list of sink vertices for the net.

Only *Vertex* objects created by this *Experiment* may be used.

weight : float

Optional. A hint for place and route tools indicating the relative amount of traffic that may flow through this net. This number is not used by the traffic generator.

name

Optional. A name for the net. If not specified the net will be given a number as its name. This name will be used in results tables.

Returns *Net*

An object representing the net.

new_vertex(*name=None*)

Create a new *Vertex*.

A vertex corresponds with a SpiNNaker application core and can produce or consume SpiNNaker packets.

Example:

```
>>> # Create three vertices
>>> v0 = e.new_vertex()
>>> v1 = e.new_vertex()
>>> v2 = e.new_vertex()
```

The experimental parameters for each vertex can also be overridden individually if desired:

```
>>> # Nets sourced at vertex v2 will transmit with 50% probability
>>> # each timestep
>>> v2.probability = 0.5
```

Parameters **name**

Optional. A name for the vertex. If not specified the vertex will be given a number as its name. This name will be used in results tables.

Returns *Vertex*

An object representing the vertex.

place_and_route(*constraints=None*, *place=<function place>*, *place_kwargs={}*, *allocate=<function allocate>*, *allocate_kwargs={}*, *route=<function route>*, *route_kwargs={}*)

Place and route the vertices and nets in the current experiment, if required.

If extra control is required over placement and routing of vertices and nets in an experiment, this method allows additional constraints and custom placement, allocation and routing options and algorithms to be used.

The result of placement, allocation and routing can be found in *placements*, *allocations* and *routes* respectively.

If even greater control is required, *placements*, *allocations* and *routes* may be set explicitly. Once these attributes have been set, this method will not alter them.

Since many applications will not care strongly about placement, allocation and routing, this method is called implicitly by *run()*.

Parameters **constraints** : [constraint, ...]

A list of additional constraints to apply. A rig.place_and_route.constraints.ReserveResourceConstraint will be applied to reserve the monitor processor on top of this constraint.

place : placer

A Rig-API complaint placement algorithm.

place_kwargs : dict

Additional algorithm-specific keyword arguments to supply to the placer.

allocate : allocator

A Rig-API complaint allocation algorithm.

allocate_kwargs : dict

Additional algorithm-specific keyword arguments to supply to the allocator.

route : router

A Rig-API complaint route algorithm.

route_kwargs : dict

Additional algorithm-specific keyword arguments to supply to the router.

placements

A dictionary {*Vertex*: (x, y), ...}, or None.

Defines the chip on which each vertex will be placed during the experiment. Note that the placement must define the position of *every* vertex. If None, calling *run()* or *place_and_route()* will cause all vertices to be placed automatically.

Setting this attribute will also set *allocations* and *routes* to None.

Any placement must be valid for the Machine specified by the *machine* attribute. Core 0 must always be reserved for the monitor processor and, if packet reinjection is used or recorded (see *Experiment.reinject_packets*), core 1 must also be reserved for the packet reinjection application.

See also `rig.place_and_route.place()`.

routes
A dictionary {*Net*: `rig.place_and_route.routing_tree.RoutingTree`, ...} or None.

Defines the route used for each net. Note that the route must be defined for *every* net. If None, calling *`run()`* or *`place_and_route()`* will cause all nets to be routed automatically.

See also `rig.place_and_route.route()`.

run(*app_id=66*, *create_group_if_none_exist=True*, *ignore_deadline_errors=False*)
Run the experiment on SpiNNaker and return the results.

If placements, allocations or routes have not been provided, the vertices and nets will be automatically placed, allocated and routed using the default algorithms in Rig.

Following placement, the experimental parameters are loaded onto the machine and each experimental group is executed in turn. Results are recorded by the machine and at the end of the experiment are read back.

> **Warning:** Though a global synchronisation barrier is used between the execution of each group, the timers in each vertex may drift out of sync during each group's execution. Further, the barrier synchronisation does not give any guarantees about how closely-synchronised the timers will be at the start of each run.

Parameters **app_id** : int

Optional. The SpiNNaker application ID to use for the experiment.

create_group_if_none_exist : bool

Optional. If True (the default), a single group will be automatically created if none have been defined with *`new_group()`*. This is the most sensible behaviour for most applications.

If you *really* want to run an experiment with no experimental groups (where no traffic will ever be generated and no results recorded), you can set this option to False.

ignore_deadline_errors : bool

If True, any realtime deadline-missed errors will no longer cause this method to raise an exception. Other errors will still cause an exception to be raised.

This option is useful when running experiments which involve over-saturating packet sinks or the network in some experimental groups.

Returns *`Results`*

If no vertices reported errors, the experimental results are returned. See the *`Results`* object for details.

Raises **NetworkTesterError**

A `NetworkTesterError` is raised if any vertices reported an error. The most common error is likely to be a 'deadline missed' error as a result of the experimental timestep being too short or the load on some vertices too high in extreme circumstances. Other types of error indicate far more severe problems.

Any results recorded during the run will be included in the `results` attribute of the exception. See the *`Results`* object for details.

2.1.2 Experimental Parameters

Global Parameters

The following experimental parameters apply globally and cannot be overridden on a vertex-by-vertex or net-by-net basis. They can be changed between groups.

`Experiment.`**`timestep`**
The timestep (in seconds) with which packets are generated by any packet generators in the experiment.

Default value: 0.001 (i.e. 1 ms)

All timing related parameters (e.g. *`duration`* and *`record_interval`*) are internally converted into multiples of this timestep and rounded to the nearest number of steps

> **Warning:** Setting this value too small will result in the traffic generator software being unable to meet timing deadlines and thus the experiment failing. In practice, 1.5 us is the smallest this can be set but larger values are required if any vertex is the source of many nets or if large numbers of metrics are being recorded.

`Experiment.`**`duration`**
The duration (in seconds) to run the traffic generators and record results for each experimental group.

Default value: 1.0

See also: *`warmup`*, *`cooldown`* and *`Experiment.flush_time`*.

`Experiment.`**`warmup`**
The duration (in seconds) to run the traffic generators without recording results before beginning result recording for each experimental group.

Default value: 0.0

Many experiments require the network to be in a stable state for their results to be meaningful. To achieve this, a warmup period can be specified during which time the network is allowed to settle into a stable state before any results are recorded.

See also: *`duration`*, *`cooldown`* and *`flush_time`*.

`Experiment.`**`cooldown`**
The duration (in seconds) to run the traffic generators without recording results after result recording is completed for each experimental group.

Default value: 0.0

Since the clocks in individual SpiNNaker cores are not perfectly in sync (and can drift significantly during long-duration experimental groups), it may be necessary for traffic generators to continue producing traffic for a short time after their local timer indicates the end of the experiment in order to maintain consistent network load for any traffic generators which are still running.

See also: *`duration`*, *`warmup`* and *`flush_time`*.

`Experiment.`**`flush_time`**
The pause (in seconds) which is added between experimental groups to allow any packets which remain in the network to be drained.

Default value: 0.01

This is especially important when *`consume_packets`* has been set to False.

See also: *`duration`*, *`warmup`* and *`cooldown`*.

`Experiment.`**`record_interval`**
: The interval (in seconds) at which metrics are recorded during an experiment.

  Default value: 0.0

  Special case: If zero, metrics will be recorded once at the end each group's execution.

  See *Metric Recording Selection* for the set of metrics which can be recorded.

`Experiment.`**`router_timeout`**
: Sets the router timeout (in router clock cycles at (usually) 133MHz).

  Default value: None (do not change)

  If set to an integer, this sets the number of clock cycles before a packet times out and is dropped. *`Experiment.run()`* will throw a `ValueError` if the value presented is not a valid router timeout. Emergency routing will be disabled.

  If set to a tuple (wait1, wait2), wait1 sets the number of clock cycles before emergency routing is tried and wait2 gives the number of additional cycles before the packet is dropped. *`Experiment.run()`* will throw a `ValueError` if the values presented are not valid router timeouts.

  If None, the timeout will be left as the default when the system was booted.

  If this field is set to anything other than None at any point during the experiment, a single vertex on every chip will be used to set the router timeout. This may result in new vertices being created internally and placed on otherwise unused chips.

`Experiment.`**`reinject_packets`**
: Enable dropped packet reinjection.

  Default value: False (do not use dropped packet reinjection).

  Enabling this feature at any point during the experiment will cause core 1 to be reserved on *all* chips to perform packet reinjection.

  See also: *`Experiment.record_reinjected`*, *`Experiment.record_reinject_overflow`* and *`Experiment.record_reinject_missed`*.

Net/Traffic Generator Parameters

The following experimental parameters apply to each net (and thus each traffic generator) in the experiment and control the pattern of packets generated and sent down each net.

The global default for each value can be set by setting the corresponding *`Experiment`* attribute.

The default for nets sourced by a particular vertex can be set by setting the corresponding *`Vertex`* attribute.

These values may also be overridden on a group-by-group basis.

For example:

```
>>> e = Experiment(...)
>>> v0 = e.new_vertex()
>>> v1 = e.new_vertex()
>>> v2 = e.new_vertex()
>>> n0 = e.new_net(v0, v1)
>>> n1 = e.new_net(v0, v3)
>>> n2 = e.new_net(v2, v3)

>>> e.probability = 1.0
>>> v0.probability = 0.9
>>> n0.probability = 0.8
```

```
>>> # First group: n0, n1, n2 all have probability == 0.0
>>> with e.new_group():
...     e.probability = 0.0
...     v0.probability = 0.0
...     n0.probability = 0.0

>>> # Second group: n2 has probability == 0.1 but n0 and n1 have
>>> # probabilities 0.9 and 0.8 respectively.
>>> with e.new_group():
...     e.probability = 0.1
```

Net.**probability**
Vertex.**probability**
Experiment.**probability**

The probability, between 0.0 and 1.0, of a packet being generated in a given timestep.

Default value: 1.0 (Generate a packet every timestep.)

Packet generation is also subject to the burst parameters *burst_period*, *burst_duty*, and *burst_phase*. By default packets are only generated according to *probability* since bursting behaviour is not enabled.

Net.**burst_period**
Net.**burst_duty**
Net.**burst_phase**
Vertex.**burst_period**
Vertex.**burst_duty**
Vertex.**burst_phase**
Experiment.**burst_period**
Experiment.**burst_duty**
Experiment.**burst_phase**

Set the bursting behaviour of the traffic generated for a net.

Default values: *burst_period* = 1.0, *burst_duty* = 0.0 and *burst_phase* = 0.0. (Not bursting).

If *burst_period* is 0.0, packets will be generated each timestep with probability *probability*.

If *burst_period* is set to a non-zero number of seconds, packets may only be generated the proportion of the time specified by *burst_duty*, every *burst_period* seconds.

```
    #
    #                            burst_period seconds
    #                       |-----------------------------|
    #                (burst_duty * burst_period) seconds
    #                            |-------|
    #         (burst_phase * burst_period) seconds
    #                       |----|
    #                       .                             .
    # maybe send            .    +-------+                .
    #                       .    |       |                .
    #                       .    |       |                .
    # never send    ...---.----+       +----------------.---...
    #                       .                             .
    #                       .                             .
```

burst_phase gives the initial phase of the bursting behaviour. Note that the phase is not reset at the start of each group.

Special case: If *burst_phase* is set to None, the phase of the bursting behaviour will be chosen randomly.

Net.**use_payload**
Vertex.**use_payload**
Experiment.**use_payload**
Should a payload field be added to each generated packet?

Default value: False (Generate 40-bit short packets)

If True, 72-bit 'long' multicast packets are generated. If False, 40-bit 'short' packets are generated.

Vertex Parameters

The following experimental parameters apply to each vertex in the experiment.

The global default for each value can be set by setting the corresponding *Experiment* attribute.

These values may also be overridden on a group-by-group basis.

Vertex.**seed**
Experiment.**seed**
The seed for the random number generator in the specified vertex or None to select a random seed automatically.

Default value: None (Seed randomly automatically).

If set to a non-None value, the seed is (re)set at the start of each group.

Vertex.**consume_packets**
Experiment.**consume_packets**
Should the specified vertex consume packets from the network?

Default value: True (Consume packets from the network)

If set to False, packets sent via any net to the vertex will not be consumed from the network. This will cause the packets to back-up in the network and eventually cause routers to drop packets.

See also *Experiment.flush_time*.

2.1.3 Metric Recording Selection

The following boolean attributes control what metrics are recorded during an experiment. Note that the set of recorded metrics may not be changed during an experiment (though the recording interval can, see *Experiment.record_interval*.

By default, no metrics are recorded.

Experiment.**record_local_multicast**
Experiment.**record_external_multicast**
Experiment.**record_local_p2p**
Experiment.**record_external_p2p**
Experiment.**record_local_nearest_neighbour**
Experiment.**record_external_nearest_neighbour**
Experiment.**record_local_fixed_route**
Experiment.**record_external_fixed_route**
Experiment.**record_dropped_multicast**
Experiment.**record_dropped_p2p**
Experiment.**record_dropped_nearest_neighbour**
Experiment.**record_dropped_fixed_route**
Experiment.**record_counter12**
Experiment.**record_counter13**
Experiment.**record_counter14**

Experiment.**record_counter15**
Record changes in each of the SpiNNaker router counter registers.

If any of these metrics are recorded, a single vertex on every chip will be configured accordingly ensuring router counter values are recorded for all chips in the machine. This may result in new vertices being created internally and placed on core 2 of otherwise unused chips.

Experiment.**record_reinjected**
Experiment.**record_reinject_overflow**
Experiment.**record_reinject_missed**
Records droppped packet reinjection metrics.

Experiment.record_reinjected The number of dropped packets which have been reinjected.

Experiment.record_reinject_overflow The number of dropped packets which were not reinjected due to the packet reinjector's buffer being full.

Experiment.record_reinject_missed A lower bound on the number of dropped packets which were not reinjected due to the packet reinjector being unable to collect them from the router in time.

If any of these metrics are recorded, a single vertex on every chip will be configured accordingly ensuring router counter values are recorded for all chips in the machine. This may result in new vertices being created internally and placed on core 2 of otherwise unused chips.

Additionally, recording any of these values will cause a further core to be reserved on *all* chips to perform packet reinjection, even if it is not enabled by *Experiment.reinject_packets* at any point in the experiment.

See also: *Experiment.reinject_packets*.

Experiment.**record_sent**
Record the number of packets successfully sent for each net.

Experiment.**record_blocked**
Record the number of packets which could not be sent for each net due to back-pressure from the network. Note that blocked packets are not resent.

Experiment.**record_received**
Record the number of packets received at each sink of each net.

2.1.4 The `Results` Class

class network_tester.**Results**
The results of an experiment, returned by *Experiment.run()*.

The experimental results may be accessed via one of the methods of this class. These methods produce Numpy `ndarray` in the form of a structured array. The exact set of fields of this array depend on the method used, however, a number of standard fields are universally present:

Any fields added using the *Group.add_label()* method If a group does not have an associated value for a particular label, the value will be set to None.

'group' The *Group* object that result is associated with.

'time' The time that the value was recorded. Given in seconds since the start of the group's execution (not including any warmup time).

A utility function, *to_csv()*, is also provided which can produce R-compatible CSV files from the output of methods in this class.

net_counters()
Gives the complete counter values for every net in the system, listing the counts for every source/sink pair individually.

In addition to the standard fields, the output of this method has:

'net' The *Net* object associated with each result.

'fan_out' The fan-out of the associated net (i.e. number of sinks).

'source_vertex' The source *Vertex* object.

'sink_vertex' The sink *Vertex* object.

'num_hops' The number of chip-to-chip hops in the route from source to sink. Note that this is 0 for a pair of vertices on the same chip.

A field for each recorded net-specific metric.

net_totals()
Gives the counter totals for each net, summing source and sink specific metrics.

In addition to the standard fields, the output of this method has:

'net' The *Net* object associated with each result.

'fan_out' The fan-out of the associated net (i.e. number of sinks).

A field for each recorded net-specific metric.

router_counters()
Gives the router and reinjector counter values for every chip in the system.

In addition to the standard fields, the output of this method has:

'x' The X-coordinate of the chip.

'y' The Y-coordinate of the chip.

A field for each recorded router- or reinjector-specific metric.

totals()
Gives the total counts for all recorded metrics.

The output of this method has a field for each recorded metric in addition to the standard fields.

vertex_totals()
Gives the counter totals for each vertex giving the summed metrics of all nets sourced/sunk there.

In addition to the standard fields, the output of this method has a 'vertex' field containing the *Vertex* object associated with each result along with a field for each recorded net-specific metric.

network_tester.**to_csv**(*data*, *header=True*, *col_sep=','*, *row_sep='\n'*, *none='NA'*, *objects_as_name=True*)
Render a structured array produced *Results* as a CSV complete with headings.

Parameters **data** : np.ndarray

A structured array produced by *Results*.

header : bool

If True, column headings are included in the output. If False, they are omitted.

col_sep : str

The separator between columns in the output. (Default: ',')

row_sep : str

The separator between rows in the output. (Default: 'n')

none : str

The string to use to represent `None`. (Default: 'NA')

objects_as_name : bool

If True, any *Group*, *Vertex* or *Net* object in the table of results will be represented by its name attribute rather than the str() of the object.

2.1.5 The `Vertex`, `Net` and `Group` Classes

class network_tester.**Vertex**

A vertex in the experiment, created by *Experiment.new_vertex()*.

A vertex represents a single core running a traffic generator/consumer.

See *vertex parameters* and *net parameters* for experimental parameters associated with vertices.

name

The human-readable name of this vertex.

class network_tester.**Net**

A connection between vertices, created by *Experiment.new_net()*.

This object inherits its attributes from `rig.netlist.Net`.

See *net parameters* for experimental parameters associated with nets.

name

The human-readable name of this net.

source

The source vertex of the net.

sinks

A list of sink vertices for the net.

weight

The weight placement & routing hint for the net.

class network_tester.**Group**

An experimental group, created by *Experiment.new_group()*.

name

The human-readable name of this group.

add_label(*name*, *value*)

Set the value of a label results column for this group.

Label columns can be used to give more meaning to each experimental group. For example:

```
>>> for probability in [0.0, 0.5, 1.0]:
...     with e.new_group() as g:
...         g.add_label("probability", probability)
...         e.probability = probability
```

In the example above, all results generated would feature a 'probability' field with the corresponding value for each group making it much easier to plat experimental results.

Parameters **name** : str

The name of the field.

value

The value of the field for this group.

num_samples
The number of metric recordings which will be made during the execution of this group.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

n

Symbols

A

B

C

D

E

F

G

M

N

P

R

S

T

U

V

W