Network Tester Documentation

Release 1.0.2

Jonathan Heathcote

August 22, 2016

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 cores with a fixed set of multicast flows of SpiNNaker packets between them.

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 cores between which multicast flows of packets exist. Each flow is sourced by a traffic generator in one core and sunk by traffic consumers in another set of cores. A single core may source and sink many flows simultaneously 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 cores and flows of network traffic between them. In this example we'll create a network with 64 cores with random flows between them. First the cores are created using *new_core()*:

```
>>> cores = [e.new_core() for _ in range(64)]
```

Next we create a single flow for each core using *new_flow()* which connects to eight randomly selected cores:

```
>>> import random
>>> flows = [e.new_flow(core, random.sample(cores, 8))
...          for core in cores]
```

By default, the cores and flows we've defined will be automatically placed and routed in the SpiNNaker machine before we run the experiment. To manually specify which chip each core is added to, this can be given as arguments to *new_core()*, for example `e.new_core(1, 2)` would create a core on chip (1, 2). For greater control over the place and route process, see *run()*.

1.1.4 Controling packet generation

Every flow has its own traffic generator on its source core. 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 *Core* and *Flow* objects returned by *new_core()* and *new_flow()* 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 flow and core) and thus can only changed experiment-wide. Other parameters, such as *probability* can be set individually for different cores or flows. As a convenience, setting these parameters on the *Experiment* object sets the 'default' value for all cores or flows. For example:

```
>>> for flow in flows:
...     flow.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 flow. 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-flow 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 flows, cores 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()`*:

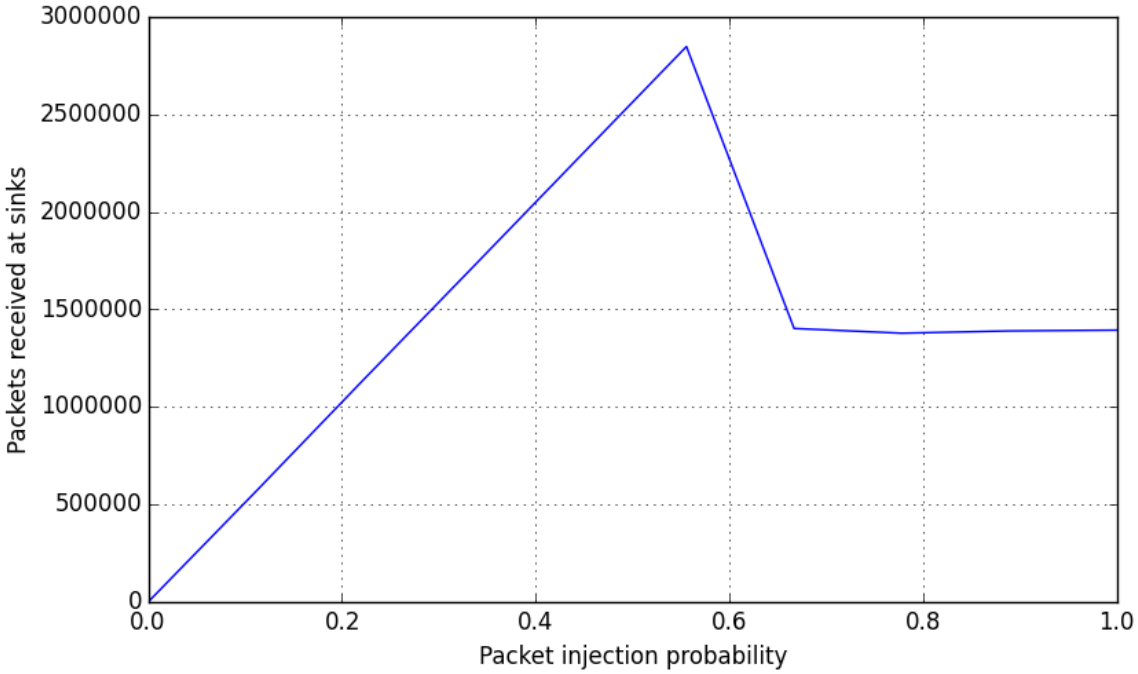
```
>>> import logging
>>> logging.basicConfig(level=logging.INFO)
```

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 defined set of SpiNNaker application cores (*`new_core()`*) between which traffic flows (*`new_flow()`*).

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 cores and the routing of traffic flows is fixed throughout an experiment, the rate and pattern of the generated traffic flows can be varied between groups allowing, for example, different traffic patterns to be tested.

When the experiment is *`run()`*, cores will be loaded with traffic generators and the routing tables initialised. When the experiment completes, recorded results are read back ready for analysis.

`__init__`(*hostname_or_machine_controller*)

Create a new network experiment on a particular SpiNNaker machine.

Typical 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 flow 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 {*`Core`*: {resource: slice}, ...} or None.

Defines the resources allocated to each core. This will include an allocation of the `Cores` resource representing the core allocated.

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

This attribute was made read-only in v0.2.0.

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

> **Warning:** This method is now deprecated, users should use *`system_info`* instead. Also, as of v0.3.0, users should *not* modify this object: all changes are now ignored.

new_core(*chip_x=None*, *chip_y=None*, *name=None*)
Create a new *`Core`*.

A SpiNNaker application core which can source and sink a number of traffic *`flows`*.

Example:

```
>>> # Create three cores, the first two are placed on chip (0, 0)
>>> # and final core will be placed automatically onto some
>>> # available core in the system.
>>> c0 = e.new_core(0, 0)
>>> c1 = e.new_core(0, 0)
>>> c2 = e.new_core()
```

The experimental parameters for each core can also be specified individually if desired:

```
>>> # Traffice flows sourced at core c2 will transmit with 50%
>>> # probability each timestep
>>> c2.probability = 0.5
```

Renamed from new_vertex in v0.2.0.

Parameters **chip_x** : int or None

chip_y : int or None

If both `chip_x` and `chip_y` are given, this core will be placed on the chip with the specified coordinates. Note: It is not possible to place more cores on a specific chip than the chip has available.

If both `chip_x` and `chip_y` are None (the default), the core will be automatically placed on some available chip (see *`network_tester.Experiment.run()`*).

Note that either both must be an integer or both must be None.

name

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

Returns *`Core`*

An object representing the core.

new_flow(*source*, *sinks*, *weight=1.0*, *name=None*)
Create a new flow.

A flow of SpiNNaker packets from one source core to many sink cores.

For example:

```
>>> # A flow with c0 as a source and c1 as a sink.
>>> f0 = e.new_flow(c0, c1)

>>> # Another flow with c0 as a source and both c1 and c2 as sinks.
>>> f1 = e.new_flow(c0, [c1, c2])
```

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

For example:

```
>>> # Flow f0 will generate a packet in 80% of timesteps
>>> f0.probability = 0.8
```

Renamed from new_net in v0.2.0.

Parameters **source** : *Core*

The source *Core* of the flow. A stream of packets will be generated by this core and sent to all sinks.

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

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

The sink *Core* or list of sink cores for the flow.

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

weight : float

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

name

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

Returns *Flow*

An object representing the flow.

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-core and per-flow parameters. Note that parameters set globally for the experiment in particular group do not take precedence over per-core or per-flow 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 c2 within the group
...     c2.probability = 0.25
```

```
...     # Overrides the probability for f0 within the group
...     f0.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.

placements

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

The placements selected for each core, set after calling *run()*.

See also rig.place_and_route.place().

This attribute was made read-only in v0.2.0.

routes

A dictionary {*Flow*: rig.place_and_route.routing_tree.RoutingTree, ...} or None.

Defines the route used for each flow.

See also rig.place_and_route.route().

This attribute was made read-only in v0.2.0.

run(*app_id=219*, *create_group_if_none_exist=True*, *ignore_deadline_errors=False*, *constraints=None*, *place=<function place>*, *place_kwargs={}*, *allocate=<function allocate>*, *allocate_kwargs={}*, *route=<function route>*, *route_kwargs={}*, *before_load=None*, *before_group=None*, *before_read_results=None*)

Run the experiment on SpiNNaker and return the results.

Before the experiment is started, any cores whose location was not specified are placed automatically on suitable chips and routes are generated between them. If fine-grained control is required the place, allocate and route arguments allow user-defined placement, allocation and routing algorithms to be used. The result of these processes can be found in the *placements*, *allocations* and *routes* respectively at any time after the before_load callback has been called.

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 are read back at the end of the experiment.

> **Warning:** Though a global synchronisation barrier is used between the execution of each group, the timers in each core 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.

The place_and_route method was removed in v0.2.0 and its functionality merged into this method.

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.

constraints : [constraint, ...]

A list of additional place-and-route constraints to apply. In additon to the constraints supplied via this argument, a `rig.place_and_route.constraints.ReserveResourceConstraint` will be added to reserve the monitor processor on every chip and a `rig.place_and_route.constraints.LocationConstraint` is also added for all cores whose chip was explicitly specified (and for other special purpose cores such as packet reinjection cores and router-register recording cores).

place : placer

A `Rig-API complaint` placement algorithm. This will be used to automatically place any cores whose location was not specified.

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. Used to route all flows in the experiment.

route_kwargs : dict

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

before_load : function or None

If not None, this function is called before the network tester application is loaded onto the machine and after place-and-route has been completed. It is called with the *`Experiment`* object as its argument. The function may block to postpone the loading process as required.

before_group : function or None

If not None, this function is called before each experimental group is run on the machine. It is called with the *`Experiment`* object and *`Group`* as its argument. The function may block to postpone the execution of each group as required.

before_read_results : function or None

If not None, this function is called after all experimental groups have run and before the results are read back from the machine. It is called with the *`Experiment`* object as its argument. The function may block to postpone the reading of results as required.

Returns *`Results`*

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

Raises **NetworkTesterError**

A `NetworkTesterError` is raised if any cores 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 cores too high in extreme circumstances. Other types of error indicate far more severe problems and are probably a bug in 'Network Tester'.

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

system_info
The `SystemInfo` object describing the SpiNNaker system under test.

This value is fetched from the machine once and cached for all future accesses. This value may be modified to influence the place-and-route processes.

2.1.2 Experimental Parameters

Global Parameters

The following experimental parameters apply globally and cannot be overridden on a core-by-core or flow-by-flow 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 core is the source of many flows 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 core on every chip will be used to set the router timeout. This may result in new cores 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*.

Flow/Traffic Generator Parameters

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

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

The default for flows sourced by a particular core can be set by setting the corresponding *Core* attribute.

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

For example:

```
>>> e = Experiment(...)
>>> c0 = e.new_core()
>>> c1 = e.new_core()
>>> c2 = e.new_core()
>>> f0 = e.new_flow(c0, c1)
>>> f1 = e.new_flow(c0, c3)
>>> f2 = e.new_flow(c2, c3)

>>> e.probability = 1.0
>>> c0.probability = 0.9
>>> f0.probability = 0.8

>>> # First group: f0, f1, f2 all have probability == 0.0
>>> with e.new_group():
...     e.probability = 0.0
...     c0.probability = 0.0
...     f0.probability = 0.0

>>> # Second group: f2 has probability == 0.1 but f0 and f1 have
>>> # probabilities 0.9 and 0.8 respectively.
>>> with e.new_group():
...     e.probability = 0.1
```

Flow.**probability**
Core.**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.

Flow.**packets_per_timestep**
Core.**packets_per_timestep**
Experiment.**packets_per_timestep**
: The number of packets to generate each timestep.

  Default value: 1 (Generate a single packet every timestep.)

  The probability of each packet being generated is independent and set to *Flow.probability*.

Flow.**num_retries**
Core.**num_retries**
Experiment.**num_retries**
: Number of times to re-try sending a packet if back-pressure from the network blocks it.

  Default value: 0 (Give up immediately when faced with back-pressure)

Flow.**burst_period**
Flow.**burst_duty**
Flow.**burst_phase**
Core.**burst_period**
Core.**burst_duty**
Core.**burst_phase**
Experiment.**burst_period**
Experiment.**burst_duty**
Experiment.**burst_phase**

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

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     ...---+     .  +-----------------------+    .  +---...
#                            .                              .
#                            .                              .
#                          time = t                  time = t + burst_period
```

burst_phase gives the initial phase of the bursting behaviour. Note that the phase is not reset at the start of each group. Values outside of the range 0.0 - 1.0 will be wrapped to within this range.

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

Flow.**use_payload**
Core.**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.

Core Parameters

The following experimental parameters apply to each core 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.

Core.**seed**
Experiment.**seed**

The seed for the random number generator in the specified core 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.

Core.**consume_packets**
Experiment.**consume_packets**
Should the specified core consume packets from the network?

Default value: True (Consume packets from the network)

If set to False, packets sent via any flow to the core 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 core on every chip will be configured accordingly ensuring router counter values are recorded for all chips in the machine. This may result in new cores 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 core on every chip will be configured accordingly ensuring router counter values are recorded for all chips in the machine. This may result in new cores 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 flow.

Experiment.**record_blocked**
Record the number of packets which could not be sent for each flow 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 flow.

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.

core_totals()
Gives the counter totals for each core giving the summed metrics of all flows sourced/sunk there.

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

If the number of sent and received packets is recorded, an additional column, 'ideal_received', is added which contains total number of packets which would be received if all sent packets arrived at every sink.

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

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

'flow' The *Flow* object associated with each result.

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

'source_core' The source *Core* object.

'sink_core' The sink *Core* 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 cores on the same chip.

A field for each recorded flow-specific metric.

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

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

'flow' The *Flow* object associated with each result.

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

A field for each recorded flow-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.

If the number of sent packets is recorded, an additional column, 'ideal_received', is added which contains total number of packets which would be received if all sent packets arrived at every sink.

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*, *Core* or *Flow* object in the table of results will be represented by its name attribute rather than the str() of the object.

2.1.5 The Core, Flow and Group Classes

class network_tester.**Core**
A core in the experiment, created by *Experiment.new_core()*.

A core represents a single core running a traffic generator/consumer and logging results.

See *core parameters* and *flow parameters* for experimental parameters associated with cores.

Renamed from Vertex in v0.2.0.

name
The human-readable name of this core.

chip
The (x, y) coordinate of the chip this core should reside on. Alternatively may be set to None if the core should be placed automatically on a suitable chip when the experiment is run.

class network_tester.**Flow**
A flow of traffic between cores, created by *Experiment.new_flow()*.

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

See *flow parameters* for experimental parameters associated with flows.

Renamed from Net in v0.2.0.

name
The human-readable name of this flow.

source
The source core of the flow.

sinks
A list of sink cores for the flow.

weight
The weight placement & routing hint for the flow.

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.

2.2 Upgrading from Network Tester v0.1.x

Between the Network Tester v0.1.x and v0.2.x series a number of API and terminology changes occurred which will break nearly all experiments written for older versions of Network Tester. This document describes how to modify your experiment scripts to work with the new network tester API and describes the terminology changes which occurred.

At a glance, the API changes are summarised in this table:

Old API (v0.1.x)	New API
`v = e.new_vertex("fred", (1, 2))`	*`c = e.new_core(1, 2, "fred")`*
`n = e.new_net(v0, v1)`	*`f = e.new_flow(c0, c1)`*
`e.placements = {v0: (1, 2), ...}`	*`c0.chip = (1, 2); ...`*
`e.place_and_route(place=hilbert_place); e.run()`	*`e.run(place=hilbert_place)`*

Note: The descision to break in API backwards-compatibility between v0.1.x and v0.2.x was not taken lightly, despite Network Tester's pre-release state. The developer believes the changes made to the API in this release will make the library substantially easier to understand, easier to use and also considerably more robust. It is unlikely that another major breaking change such as this will occur again before the 1.0 release.

To aid users in the transition to v0.2.x the deprecated functions have been replaced with stubs which raise a hepful error message linking to this documentation.

2.2.1 Terminology Changes

Previous versions of Network Tester used the term 'vertex' to refer to an application core on the machine and 'net' to refer to the flows of traffic between the cores. In v0.2.x onwards the term 'core' is used in place of vertex and the term 'flow' in place of net. These changes intend to make it much clearer what is being done by the API.

The following API name changes have been made to reflect the new terminology:

Old Name (v0.1.x)	New Name
`Experiment.new_vertex()`	*`Experiment.new_core()`*
`Experiment.new_net()`	*`Experiment.new_flow()`*
`Vertex`	*`Core()`*
`Net`	*`Flow()`*

2.2.2 Manual Placement Changes

The *`Experiment.new_core()`* method's first two arguments are now the x and y coordinates of the chip the core should be placed on. Previously, the first argument was the name of the vertex and the chip coordinates were given as a single argument.

Old Syntax (v0.1.x)	New Syntax
`e.new_vertex("fred", (1, 2))`	*`e.new_core(1, 2, "fred")`*

The *`Experiment.placements`*, *`Experiment.allocations`* and *`Experiment.routes`* properties are now strictly read-only. Manual placement should be performed by specifying the chip position of each *`Core`* when it is created or by setting the *`Core.chip`* attribute of cores. If greater flexibility is required, you should supply a rig-compilent `place()`, `allocate()` and `route()` function to *`Experiment.run()`*.

2.2.3 Place-and-Route Changes

The place-and-route process now always occurs as part of the *Experiment.run()* method, `Experiment.place_and_route()` is no longer available. The *run()* method now accepts all the arguments the `place_and_route()` method used to. This change prevents the accidental use of stale place-and-route information resulting from changes being made to the experiment between calling `place_and_route()` and *run()*. For example:

Old Syntax (v0.1.x)	New Syntax
`e.place_and_route(place=hilbert_place); e.run()`	*e.run(place=hilbert_place)*

2.2.4 Reverting to v0.1.x

If for some reason this is not possible to modify your experiment script to support the new Network Tester API, you can revert to the last v0.1.x version of Network Tester using:

```
$ pip install -I "network_tester<0.2.0"
```

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

n

Symbols

A

B

C

D

E

F

G

M

N

P

R

S

T

U

W