snoop Documentation

Release 0.7

Andy Mastbaum

May 15, 2012

CONTENTS

snoop is an online data processing and monitoring tool developed for the SNO+ experiment. It reads event data from a file or avalanche dispatcher and calculates statistics. It is used to monitor the quality of the data being taken.

CHAPTER

ONE

OVERVIEW

snoop is based on "processor model," not unlike RAT. Processors are called for each event, and may operate on and store that data as needed. Processors are "sampled" with a user-defined period. When sampled, a processor creates a document, which is subsequently written out (probably to a database).

With this model, there are two main classes of processors:

1. Processors that look at every event, gathering data, and when sampled compute and returns statistics related to that data
2. Processors that ignore events, and perform some action when sampled, like query a server to get its disk usage

In either case, a document is produced that represents a snapshot of the parameters the processor cares about.

snoop can only get the samples to a database; the user interface layer is handled by separate software, e.g.:

```
data stream --Reader-> snoop --Writer-> database --REST API-> woodstock -> client
```

woodstock was developed as a front-end.

CHAPTER

TWO

INSTALLATION

snoop is packaged with `setuptools` for easy installation. To install directly from the repository, run:

```
$ pip install -e git+git://github.com/mastbaum/snoop.git#egg=snoop
```

CHAPTER

THREE

USAGE

snoop is intended to be run as a daemon, but additionally can be run in the foreground or used from Python.

You can communicate with snoop processes using signals, most importantly SIGUSR1 (10), which will reload processors from the configured processor path. If processors define a `load` function, this is used to copy state from the old processor instances to the new ones upon reloading.

3.1 Configuration

snoop is configured using a Python module as a configuration file. The following must be defined:

- `sample_period`: The time between samples, in seconds
- `processor_paths`: List of Python paths to processors, as (path, fromlist) tuples
- `writer`: A `Writer` subclass instance with which to handle output
- `reader`: A `Reader` subclass instance, from which events will be read

`processor_kwargs`, a `{'name': dict}` dictionary may also be defined to supply keyword arguments to processors.

The default configuration file path is `./config.py`.

3.2 Daemon

To run snoop as a daemon:

```
$ snoop -d [/path/to/config.py]
```

3.3 CLI

To run snoop on the command line:

```
$ snoop [/path/to/config.py]
```

3.4 Python module

To use snoop from Python:

```
>>> import snoop
```

All of snoop's functionality is available in Python modules – see API documentation for detail details.

CHAPTER

FOUR

HISTORY

This software is completely distinct from the old snoop used in SNO, sharing only the name and purpose.

CHAPTER

FIVE

DOCUMENTATION

5.1 snoop Python API

5.1.1 Processors

class snoop.processor.**Processor**

A Processor represents a chunk of analysis code that can be placed into the event loop by the user.

event(*event*)

Called once per event.

load(*rhs*)

Load data from another instance. Called when instances are replaced during a module reload.

sample()

Called periodically, "sampling" the processor state.

exception snoop.processor.**ProcessorAbort**(*value*)

Processors should throw *ProcessorAbort* to indicate that a problem happened, but that the processor should remain in the event loop.

class snoop.processor.**ProcessorBlock**(*processors*, *kwargs={}*)

A processor block is analagous to a block of Python statements. It consists of a list of processors, calling each of their *event* methods when *event* is called, and each of their *sample* methods when *sample* is called.

There are two ways to create a block of processors: either provide a list of *Processor* instances or a list of *(module_name, fromlist)* tuples. In the latter case, an instance of every *Processor* found in the given modules will be added.

event(*ev*)

Call *Processor.event(ev)* for each processor

load_processors(*paths*, *kwargs={}*, *preserve=True*)

(Re)load all processors found in the provided paths and add them to this *ProcesorBlock*. If *preserve* is true, load the new instances with the data from the old via *Processor.load*, providing some rudimentary "schema evolution."

sample()

Call *Processor.sample()* for each processor.

This is done asynchronously with a thread pool farm sampling out to multiple cores and prevent I/O-heavy sample methods from blocking processing.

Returns a *multiprocessing.pool.AsyncResult*; *get()* the results out when it is *ready()*.

5.1.2 Readers

class snoop.reader.**DispatchReader**(*address*)
Read records from the dispatcher stream at *address*.

add_db(*host*, *dbname*, *mapper*)
Attach a database changes feed to this reader. See *avalanche* documentation for more details.

add_dispatcher(*address*)
Attach an additional dispatch stream to this reader. See *avalanche* documentation for more details.

read()
Generator of events from the dispatcher.

class snoop.reader.**ROOTFileReader**(*filenames*, *tree_name*, *branch_name*, *obj*)
Read entries from a ROOT tree in a file.

load_tree()
Load the ROOT tree for reading. This is deferred until the first read to prevent file descriptors from getting lost if running as a daemon.

read()
Generator of entries on the requested branch. Raises *StopIteration* when no more entries are available.

5.1.3 Writers

class snoop.writer.**CouchDBWriter**(*host*, *dbname*, *username=None*, *password=None*)
Output handler that writes to a CouchDB database.

class snoop.writer.**PrintWriter**
Output handler that just prints.

class snoop.writer.**Writer**
Output handler (base class).

5.2 snoop Design

This document outlines the design requirements for a SNOOP replacement, covering features found in both snoop and woodstock.

5.2.1 Design Criteria

Logic Layer

- Polls the same data sources as old snoop
  - Dispatcher
  - Alarms
  - DAQ/Computer status
  - Data flow
- Runs forever
  - Cannot crash

  - Cannot be restarted

Data Layer

- Query over time ranges
- Query fields selecting on other fields
- Caching of large/frequent requests
- Persistent
- Automatic fault recovery or replication

Presentation Layer

- Sensible defaults, highly configurable
- Programmable by non-experts (ASCII templating)
- Alarms immediate but not intrusive
- Plotting of arbitrary data
  - Histograms
  - Time series
  - Scatter plots

5.2.2 Design Choices

Logic Layer

- Python daemon
  - Communication through signals
  - start, stop, restart, reload operations
  - Dynamic module reloading while running
- Processor model
  - Arbitrary processor code
    - Aggregated event data
    - Polling external sources
  - Called per event
  - State sampled at regular interval
    - Asynchronous
    - Output handled by writer
      - Push to database
      - Log, email, alarm, ...

Data Layer

- CouchBase server
  - JSON key/value store
  - High performance
    - Incremental view indexing
    - In-memory caching
    - Clustered
- Python/WSGI interface
  - Implements REST API for client queries
    - Date ranges, SELECT-like operations

Presentation Layer

- snoop Web Interface
  - Templates written in ReST/Markdown with special tags
  - Framework renders templates as HTML + JS (Backbone?)

CHAPTER

SIX

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX

S