boing Documentation

Release 0.3.1

Paolo Olivo & Nicolas Roussel

Aug 28, 2017

Contents

Welcome! This is the documentation for Boing 0.3.1, last updated Aug 28, 2017.

Boing is a Python 3 toolkit designed to support the development of multi-touch and gesture enabled applications.

Boing enables to create pipelines for connecting different input sources to multiple target destinations (e.g. applications, logs, etc.) and eventually process the data before being dispatched. Boing provides a set of functional nodes that enable to:

- read and decode input sources (e.g. TUIO, OSC, JSON);
- encode and forward data to target outputs (e.g. TUIO, OSC, JSON);
- record and replay the data flow;
- process gesture data (calibration, smoothing filtering, debugging, etc.);
- debug and get statistics of the data flow.

Get started!

CHAPTER 1

Showcase

Boing documentation contents

Introduction

Boing enables to create pipelines for connecting different input sources to multiple target destinations (e.g. applications, logs, etc.) and eventually process the data before being dispatched.

As an example, consider the pipeline in *figure 1.1*: two tactile devices (left side) are connected to a single user application (top-right). At the same time, the contact events from both the devices are forwarded as a JSON stream to a second remote application (e.g. a contact visualiser), while an event recorder is used to log into a file the data stream provided by the second device only.

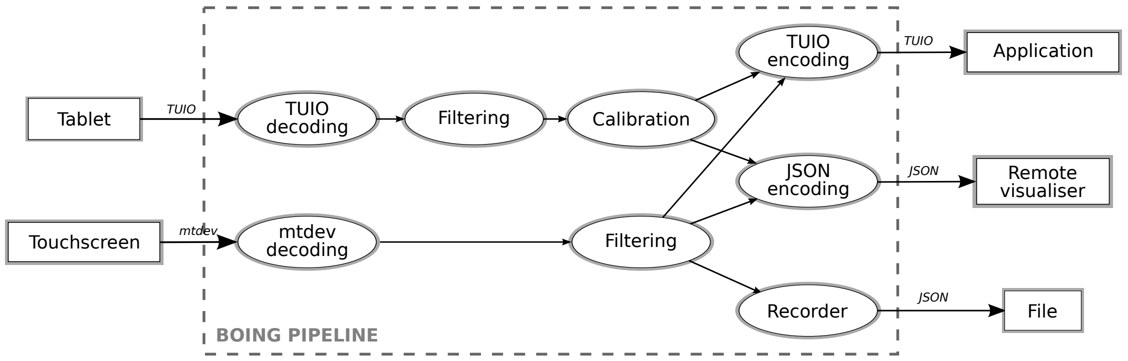

Fig. 1.1: Example of pipeline created using Boing.

Even if the tactile devices provides different data structures, Boing enables to merge them in a single data stream (in this example the TUIO and the JSON stream). Contact events are also processed before being passed to the application: Boing provides nodes to smooth or calibrate the input data (e.g. position, speed, etc.). As shown in the example, pipelines can be composed by parallel branches so that each input/output can have its own processing suite.

Boing does not impose a specific data model; instead it exploits a query path language (similar to JSONPath) for accessing the data to be processed, so that it can fit a wide range of application domains.

Installing Boing

Download

The current version is 0.3.1, released on September 18, 2012. Read the *ChangeLog*.

Download one of the following depending on your platform:

File	Type	Py Version	Size
	Source	3.2	702KB
	Source	3.2	833KB
	MS Windows MSI installer	3.2	544KB

If you are interested to the development version, you can clone the source repository. Open a terminal and type:

```
git clone https://github.com/olivopaolo/boing
```

Installation

Boing requires the Python 3.2 interpreter (or newer) and the PyQt4 package (a set of Python bindings for Nokia's Qt application framework). Moreover, it also requires the following Python packages:

- numpy
- pyparsing

Extensive installation instructions are available for the following platforms:

Boing on Ubuntu 12.04

It is possible to install both Python and PyQt directly from the standard repositories by typing in a terminal:

```
sudo apt-get install python3-pyqt4 python3-setuptools python3-numpy
```

In order to complete the installation, open a terminal and type:

```
cd <BOING-DIRECTORY>
sudo python3.2 setup.py install
```

Other Ubuntu releases

- Ubuntu 11.04

Boing on OS X 10.8 and 10.7

PyQt4 and Distribute

If you have Mac Ports, getting PyQt4, numpy and Distribute is as simple as typing:

```
sudo port install py32-pyqt4 py32-distribute py32-numpy
```

Boing

In order to complete the installation, open a terminal and type:

```
cd <BOING-DIRECTORY>
sudo python3.2 setup.py install
```

The **boing** executable may be not installed into a directory indexed by the `PATH` variable, so that it is always necessary to use the full path to launch it. To avoid this annoying behaviour, a simple solution is to set the installer target directory using the option `--install-scripts`. As an example:

```
sudo python3.2 setup.py install --install-scripts /usr/local/bin
```

Boing on Windows 7

Python 3.2

First download the Python 3.2 binary installer and run it.

After Python has been installed, it may be useful to modify the PATH environment variable, so that Windows can find Python binaries and scripts without the need of specifing all the times the entire path. As an example, the PATH variable has been set to:

```
PATH = C:\Python32;C:\Python32\Scripts
```

PyQt4

Download the PyQt4 binary installer and run it.

Distribute

The package Distribute is necessary to run the boing's installer script. Download the file distribute_setup.py and type in a terminal:

```
cd <DOWNLOAD-DIRECTORY>
python distribute_setup.py
```

numpy

Download the numpy binary installer and run it.

If you are running a Windows 7 64 bit, you may also use the unofficial numpy Windows binary installer that you can find at http://www.lfd.uci.edu/~gohlke/pythonlibs

pyparsing

Download the source code of pyparsing, extract the archive and type in a terminal:

```
cd <PYPARSING-DIRECTORY>
python setup.py install
```

Boing

If you downloaded the binary installer you just have to launch it, otherwise extract the source archive and type in the terminal:

```
cd <BOING-DIRECTORY>
python setup.py install
```

Tests

After the installation has been completed, it may be useful to run the test suite in order to verify that everything has been correctly installed. In order to do so, type in a terminal:

```
python3 setup.py test
```

It is also possible to test only a subset of the Boing's modules:

```
cd <BOING-DIRECTORY>
python3 boing/test/run.py [MODULE [MODULE ...]]
```

The available modules are: `core`, `filtering`, `gesture`, `net`, `nodes`, `utils`. If no module is specified, all the available modules will be tested.

If you are interested to check the code coverage, you may use the tool called coverage by Ned Batchelder. Once the tool has been installed, you simply have to type:

```
cd <BOING-DIRECTORY>
coverage run --source boing boing/test/run.py
coverage report -m
```

Getting started

This section contains various tutorials to easily learn to use the toolkit Boing:

First steps

This tutorial provides few simple examples of the functionality of the toolkit to help you starting to use the Boing toolkit.

Let's consider to have a multi-touch input device, like a tablet or a touch-screen. What cool things can I do with Boing? Boing enables to create a pipeline for connecting your device to different targets, like applications, frameworks, debuggers and eventually processing the gesture events before having being dispatched, like for example calibrate the contacts' position or apply a smoothing filter.

To make things easier, let's consider that your device can send the information of the contact events as a TUIO stream on the local port 3333.[1]

Showing multi-touch events

First of all, it is important to know that all the Boing's tools are invoked by using the script **boing**. Open a terminal and type:

```
boing "in.tuio://:3333 + viz:"
```

The script should have opened a window displaying a grid. Now when you touch your multi-touch device, you will be able to see the contact points appear on the window.

It's not difficult to notice that the script accepts a single argument that defines the configuration of the pipeline that is to be created. Configurations are defined by a formula where the operands define the functionality of the nodes of the pipeline, while the operators define how the nodes are connected, therefore also the structure of the pipeline.

In the previous example, the pipeline was composed by two nodes:

- `in.tuio://:3333` corresponds to a node that reads the socket, decodes the TUIO stream and provides the multi-touch events;
- `viz:` corresponds to the *Contact Visualizer*, a widget that shows the information of the contact points, such as position, track and speed.

The two nodes are joined using the + operator, which stands for connection *in series*. The structure of the pipeline is represented in *figure 3.1*.

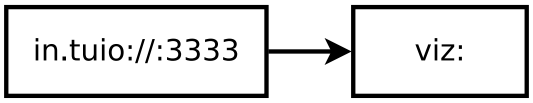

Fig. 1.2: Pipeline obtained from the configuration `in.tuio://:3333 + viz:`.

Congratulations! You have created your first Boing pipeline!

Exploring the data

Now, let's try new functionalities by adding a new node. Stop the previous pipeline by closing the visualizer widget or pressing Ctrl-C on the terminal, and type in the terminal:

[1] If you are unfamiliar with the TUIO protocol, consider having a look to the available TUIO trackers, or jumping to the *Nodes reference table*, in order to discover the different ways Boing exploits to connect to the input devices.

```
boing "in.tuio://:3333 + (viz: | dump:)"
```

As before the contact visualizer appears again, but this time, when you touch the multi-touch device, the terminal prints a lot of data! The terminal output represents all the data that the `in.tuio://:3333` node can produce and send to the connected nodes. This tutorial is not aimed to provide an exaustive description of the message structure; for the moment, simply observe that data messages are hierarchical structures mainly composed by Python built-in types, such as dictionaries, lists, strings, bytearrays, etc. Thanks to such standard structure, Boing exploits a query language, similar to JSONPath, for the indexing or the filtering of data messages. In order to understand the usefulness of such query language, stop the pipeline and type in the terminal:

```
boing "in.tuio://:3333 + (viz: | dump:?request=..contacts)"
```

Now, when you touch your multi-touch device, you can see that the terminal prints the subset of the data structures that refers only to the contact data. This is because the query `..contacts` addresses to any data named as `contacts`, searched at any level of the structure. Such query language can be very useful during development and testing phases for highlighting only the relevant information.

A more exhaustive description of the data structure and of the query language can be found in the data model section. For now, let's leave the data structure and we consider the functioning of the pipeline: it's not difficult to understand that the `|` operator (*Pipe*) is used to connect in parallel the nodes `viz:` and `dump:`, so that the products are sent to both of them. *Figure 3.2* shows the structure of the current pipeline.

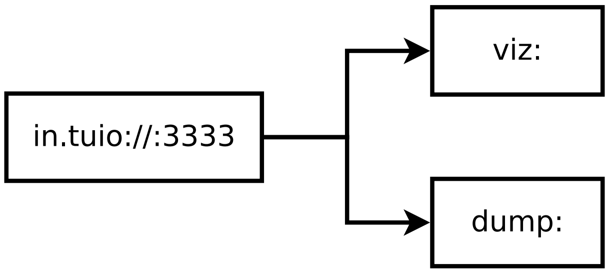

Fig. 1.3: Pipeline obtained from the configuration `in.tuio://:3333 + (viz:  | dump:)`.

Combining input sources with external applications

A key feature of Boing is the ability to provide the captured input events to external applications. This enables in most of the cases to take advantage of the toolkit's features without the need to adapt or to modify the applications, while sometimes a simple configuration may be required. As shown in *figure 3.3*, the Boing toolkit works as a semi-transparent layer placed between the input sources and the final applications.

Thanks to the many supported encodings, Boing can easily fit different combinations of devices and applications. In this basic example, let's consider to have an application listening for a TUIO stream on the local port 3335[2]. If you don't have a TUIO application, simply open a new terminal and launch a new Boing instance using the command:

```
boing "in.tuio://:3335 + viz:"
```

In the previous example you connected one input device to two output nodes. The `|` operator also enables to put in parallel different inputs, like for example a second multi-touch device enabled to send its TUIO messages to the local port 3334. Let's try a new pipeline by running the command:

[2] For more output sources, see the *Nodes reference table*.

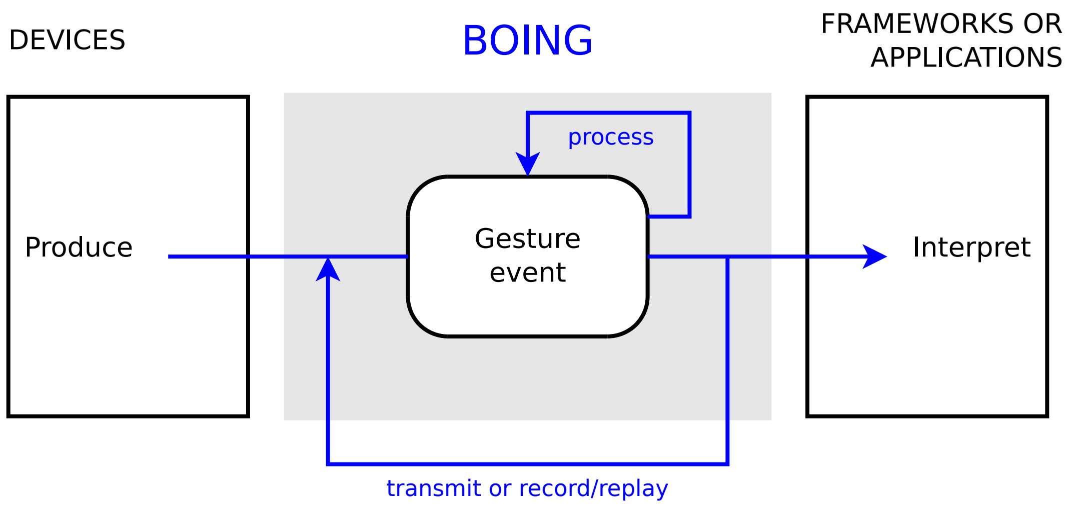

Fig. 1.4: Boing works as a semi-transparent layer placed in between the devices and the applications for processing and transmitting the input events.

```
boing "(in.tuio://:3333 | in.tuio://:3334) + (viz: | out.tuio://127.0.0.1:3335)"
```

Figure 3.4 shows the structure of the new pipeline.

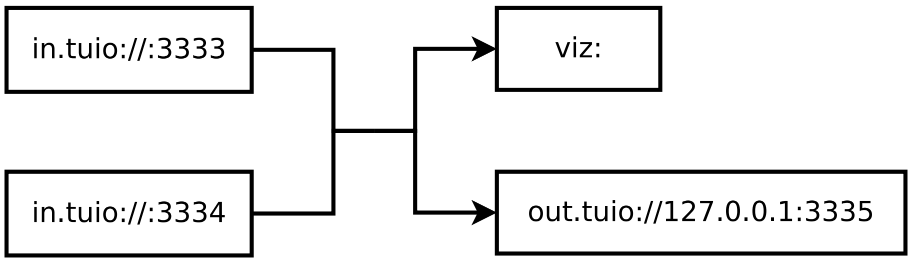

Fig. 1.5: Pipeline obtained from the configuration `(in.tuio://:3333 | in.tuio://:3334) + (viz: | out.tuio://127.0.0.1:3335)`.

As you can see, a very important feature of Boing is that you can simultaneously connect many devices to different applications. Such feature eases the usage of debugging tools and it enables multi-device and multi-user applications.

Data processing

The Boing toolkit is not only able to redirect input data to different destinations, but it also enables to process the transferred data. With regard to the multi-touch devices, recurring operations are the removal of the sensor noise and the calibration of the touch points. In order to accomplish these tasks, the toolkit provides two functional nodes that can be easily employed in our pipelines. As an example, let's run a new pipeline using the following command:

```
boing "in.tuio://:3333 + filtering: + calib:?screen=left + viz:"
```

Now, when you touch your tactile device you should still see the interactions on the visualizer widget, but now they look more smooth and they are rotated 90 degrees counterclockwise. By employing the `filtering:` node, we added the default smoothing filter, which is applied by default to the position of the contact points, while the node `calib:` performs the calibration of the touch points.

The structure of the current pipeline is shown in *figure 3.5*.

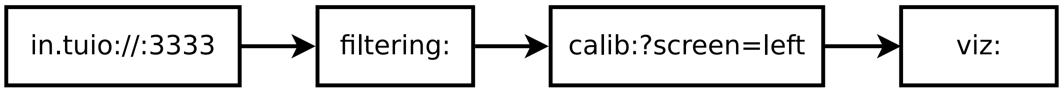

Fig. 1.6: Pipeline obtained from the configuration `in.tuio://:3333 + filtering: + calib:?screen=left + viz:`

In order to better understand the result of the processing stage, it may be useful to show at the same time the raw data and the processed one. In order to achieve such result, stop the previous pipeline and run the following command:

```
boing "in.tuio://:3333 + (filtering: + calib:?screen=left + edit:?source=filtered |␣
↪nop:) + viz:"
```

Now, when you touch your input device you can see on the visualizer widget both the raw tracks and the processed tracks, so that it is easier to note the effect of the processing stage. The structure of the modified pipeline is shown in *figure 3.6*. Note that this behaviour has been obtained by adding a parallel branch constituted only by the node `nop:`, which simply forwards the incoming data without making any modifications, and adding the node `edit:?source=filtered`, which labels the events of the processing branch so that they belong to the source *filtered* (the name is not relevant). This latter step is necessary since the data of the two parallel branches is merged into a single stream before being passed to the visualizer widget.

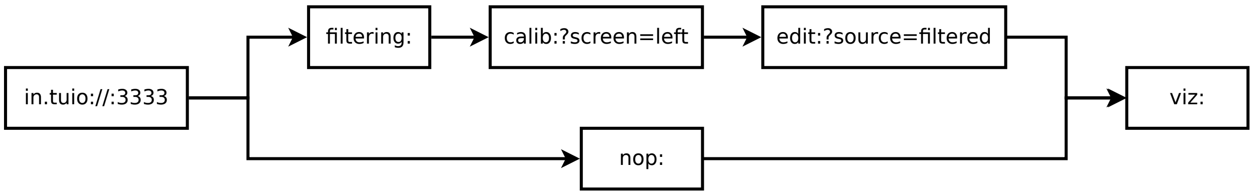

Fig. 1.7: Figure 3.6: Pipeline obtained from the configuration `in.tuio://:3333 + (filtering: + calib:?screen=left + edit:?source=filtered | nop:) + viz:`

Event recording and replaying

The Boing toolkit also provides some tools for recording input events into log files and some other tools for replaying them. These operations are often really helpful during the development and debugging of applications. The simplest way to log events into a file is to use the node `log:`. As an example, consider running the following command:

```
boing "in.tuio://:3333 + (viz: | log:./log.bz2)"
```

Now, all the gestures you make on your tactile device will be recorded and written to the file `./log.bz2`. Then, stop the pipeline by pressing Ctrl-C and let's replay the recorded gestures by executing the command:

```
boing "play:./log.bz2 + viz:"
```

Quite easy, isn't it? It is also possible to configure the player to endlessly rerun the log and set the replay speed. To do so, simply run this command:

```
boing "play:./log.bz2?loop&speed=0.2 + viz:"
```

A more powerful tool for replaying log files is the `player:` node: thanks to its GUI, it enables users to easily define a playlist of log files that the node will reproduce. As an example, run the following command:

```
boing "player: + viz:"
```

Playlists can be exported so that the `player:` tool becomes very useful during the application testing for executing the unit test.

Saving pipeline configurations

Sometimes it can be useful to store the configuration of the pipeline for later reuse. Writing long configurations in a terminal may also be quite annoying. For these reasons, Boing lets users to write the configuration of the pipeline in a text file and then to load such configuration using the special node `conf:`.

As an example, consider you have finally wrote the configuration of a pipeline for comparing the result of different smoothing filters. Now you want to save it in a file (e.g. `config-filters.txt`) and maybe you want to add some comments that will help you understanding the structure of the pipeline. The file may look like the following:

```
# Pipeline for comparing the result of different smoothing filters.
#
# Author: Me
# Date: Sept 12, 2012
# File: config-filters.txt

# ONE INPUT: a standard TUIO source
in.tuio://:3333

# PARALLEL BRANCHES
+ (
    # Raw input
    nop:

    # Default filter
    | filtering: + edit:?source=Default

    # Moving mean filter
    | filtering:/moving/average?winsize=5 + edit:?source=Mean

    # Moving median filter
    | filtering:/moving/median?winsize=5 + edit:?source=Median

    # Exponential double filter
    | filtering:/exponential/double?alpha=1&gamma=1 + edit:?source=Exponential

    # OneEuro filter
    | filtering:/oneeuro?freq=1 + edit:?source=OneEuro
    # ---
  )

# ONE OUTPUT: the visualizer
+ viz:
```

Now, in order to run the pipeline you just have to enter the command:

```
boing conf:./config-filters.txt
```

Quite easy, isn't it?

The node `conf:` is actually a composite node that contains the pipeline defined in the configuration file. For this reason, it is also possible to use the `conf:` node into another pipeline. Consider as an example that you have a multi-touch table sending contact information via the TUIO protocol, you found a good smoothing filter since the input is quite noisy and you also determined the calibration matrix to fit the touch position to the correct screen space. You are not going to change these parameters so you would like to consider all these elements as an atomic input source that does not mess up a larger configuration. Thus, first you could write the configuration of your input source into a file (e.g. `my-mt-table.txt`), which may look like the following:

```
# My multi-touch table without its ugly noise and well calibrated.
#
# Author: Me
# Date: Sept 12, 2012
# File: my-mt-table.txt

# INPUT: standard TUIO source
in.tuio://:3333

# FILTERING: OneEuro filter
+ filtering:/oneeuro?freq=60&merge

# CALIBRATION: my 4x4 matrix
+ calib:?merge&matrix=0.98,0,0,0.021,0,0.83,0,0.010,0,0,1,0,0,0,0,1

# NO OUTPUT, so I can reuse this into an external pipeline.
```

Then, you can reuse your configured input device as an atomic item in a new pipeline. As an example, let's show the contact events using the `viz:` node, and at the same time use the recorder widget and forward the contacts to an other application listening for a TUIO source on the local port 3334. The command to run is the following:

```
boing "conf:./my-mt-table.txt + (viz: | rec: | out.tuio://127.0.0.1:3334)"
```

As you can see, saving pipeline configurations into files can be quite useful in different situations. Needless to say that you can also use the `conf:` node inside a configuration written in a file, so that it is possible to arrange items in a hierarchical structure.

Boing for developers

Developers can easily deploy pipelines by invoking the *Boing's API* in their Python code. The function `boing.create()` can be used to instantiate the pipeline's nodes. This method requires as argument an URI expression that is used to specify the functionality of the nodes to be created and how they are connected. The *Nodes reference table* is the same as for the command line script. Then, the operators `+` and `|` can be used compose the pipeline.

The following code can be used as an example for creating Boing pipelines:

```
#!/usr/bin/env python3
import sys
import PyQt4
import boing

# Init application
app = PyQt4.QtGui.QApplication(sys.argv)
```

```
# Create nodes
n1 = boing.create("in.tuio://:3333")
n2 = boing.create("viz:")
n3 = boing.create("dump:?request=$..contacts")

# Compose the pipeline
graph = n1 + (n2 | n3)

# Run
sys.exit(app.exec_())
```

Todo

Describe an example of functional node.

Underlying concepts

The documentation is structured into the following sections:

The pipeline architecture

Boing pipelines are made by directed graphs, where the edge direction defines the data flow between the nodes. There are three types of nodes:

- *producers* provide the data;
- *consumers* process the incoming data;
- *workers* are both consumers and producers;

The type of a node directly influences how the node can be connected to the other nodes: producers only accept outgoing connections, while consumers accept incoming connections only. Workers are composed by both the producer and consumer interfaces, so they can have both incoming and outgoing connections. *Figure 4.1* shows an example of both valid and invalid connections.

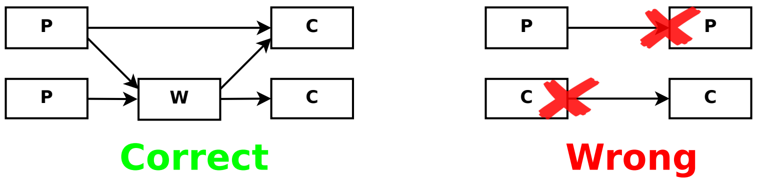

Fig. 1.8: Valid and invalid connections between producers (P), consumers (C) and workers (W).

The producer-consumer model

The core infrastructure of Boing pipelines is the producer-consumer model, which defines how the data is propagated through the pipeline. The model performs a pull technology, but it is extended by using the *Observer* pattern: consumers must subscribe to the producers in order to receive their products; for each subscribed consumer, producers keep a record containing the list of the pending products. When a producer has a new product, for each registered consumer it enqueues the product in the associated product list and it triggers the consumer, which synchronously or asynchronously can require its own pending products. Then, at the consumer's request, the producer sends all the correspondent pending products to the consumer and it cleans the correspondent buffer. The entire pipeline is run in a single thread, thus an eventloop is used to handle the asynchronous nodes. *Figure 4.2* shows the UML sequence diagram that defines the data exchange between producers and consumers.

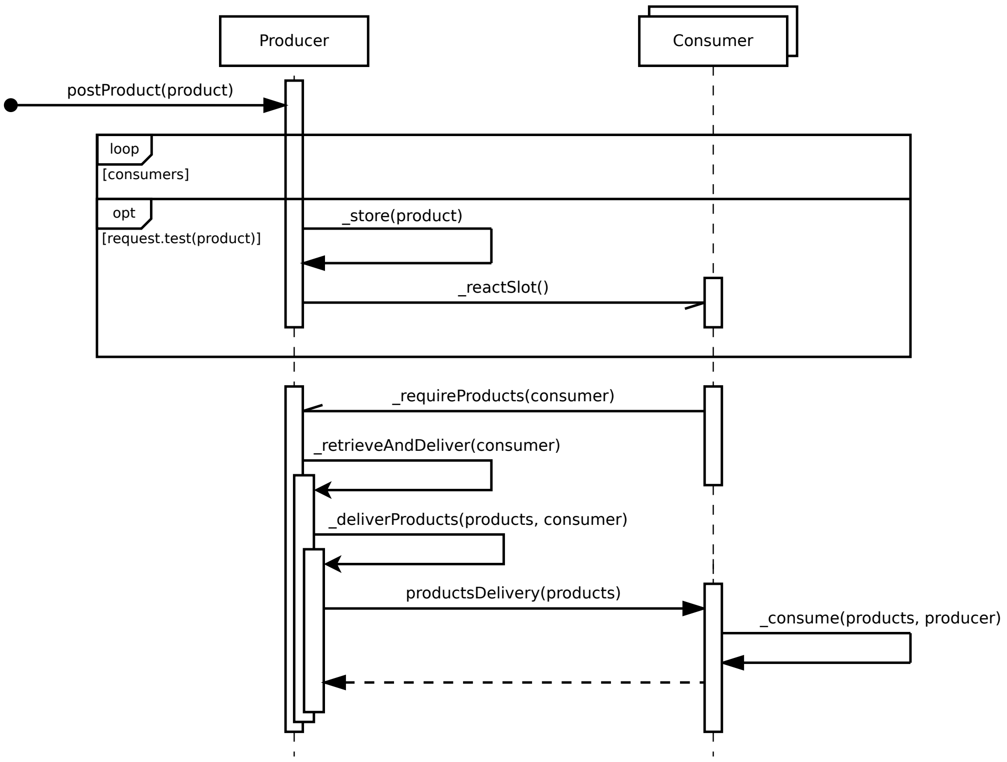
Producer
Consumer
postProduct(product)
loop
[consumers]
opt
[request.test(product)]
_store(product)
_reactSlot()
_requireProducts(consumer)
_retrieveAndDeliver(consumer)
_deliverProducts(products, consumer)
productsDelivery(products)
_consume(products, producer)

Fig. 1.9: UML sequence diagram defining the producer-consumer model

See also:

classes `boing.core.Producer` and `boing.core.Consumer`

Supply and demand

In many situations, a data source can provide a wide range of information, but consumers may not be interested in all of it. For this reason, in order to save processing time, the model permits to assign a request to each consumer. Every time a producer has a new product, it tests the request of each registered consumer and only if it matches the product, the producer notifies the consumer the new product. This behavior enables to process and transfer only the useful information, while the useless part is not processed. Requests can be added up so that a producer can easily know the entire request of all its registered consumers. The union of all the registered consumers' requests is called *aggregate demand*.

On the other side, it is good to know what a producer can supply. For this reason the model permits to assign an offer to the producers, which must be the list of templates of the products it can provide. Using its offer, a producer can say a priori whether it can meet a consumer's request. Composing the offer and the aggregate demand, it is possible to calculate the *demanded offer*, which represents the subset of the offer that is currently being demanded.

As an example, consider two producers *P1* and *P2* and two consumers *C1* and *C2* connected as shown in *figure 4.3*. It is possible to observe that the aggregate demand of *P1* is equal to the union of the requests of both *C1* and *C2*. Moreover, even if *P1* produces both *A* and *B*, only the products *A* are sent to *C1*, while both *A* and *B* products are sent to *C2*. Also note that *P2*'s *demandedOffer* is only *B*, because *P2* is only connected to *C2* and this one does not require the products *C*.

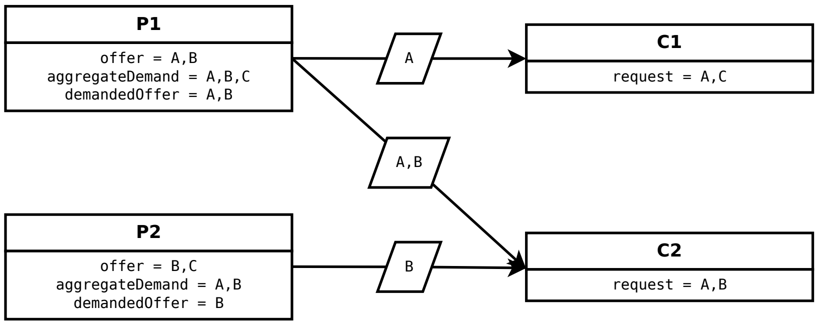

Fig. 1.10: Example of supply and request behavior.

Note: It is important to understand that a node's offer does not impose that the only products that the nodes produces are coherent with the offer and even that it is sure that the node will ever produce such products. The offer is only used to describe the node standard behavior. *It's easier said than done!*

See also:

classes `boing.core.Offer` and `boing.core.Request`

As previously seen, it is possible to create long pipelines by serializing worker nodes. In order to spread the supply and demand strategy, a worker node must be able to propagate the requests of the consumers it is connected to in addition to its own request and to propagate the offer of the producers it is connected to in addition to its own offer. In order to understand such necessity, consider the pipeline shown in *figure 4.4*: in this case the worker *W* is not propagating its neighbors' requests and offers (the variables *isPropagantingRequest* and *isPropagatingOffer* are false), so that its own request and offer, which are defined by the variables *_selfRequest* and *_selfOffer*, are actually the same of its (public) request and offer. In this case, it is possible to notice that even if the consumer *C* require the products *B*, such demand is hidden by the worker *W*, so that even if the producer *P* can provide *B* products, it can't see anyone interested to them, so they are not produced.

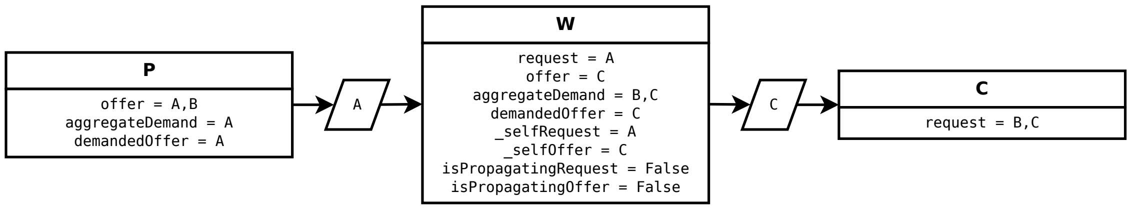

Fig. 1.11: The worker *W* is not propagating its connected consumers' requests, thus the producer *P* does not provides the products *B*.

The *figure 4.5* shows the same pipeline as before with the difference that the worker *W* is now propagating its neighbors' requests and offers. It is possible to notice that the request of *W* is equal to the union of the request of *C* and its own request, and its public offer is equal to the union of the offer of *P* and its own offer. *W* is now requiring *B* products because a subsequent node is also requiring them, thus *P* will produce and dispatch them.

Note: It is important to understand that the variables *isPropagatingRequest* and *isPropagatingOffer* do not control the output of *W*, but only the fact that its request and offer are determined by accumulating the neighbors requests and

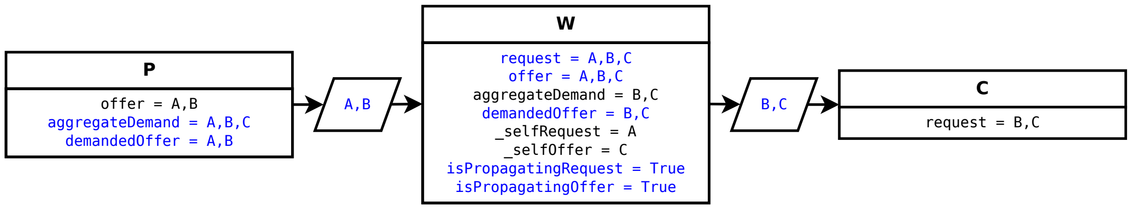

Fig. 1.12: Example of supply and request behavior.

offers. The fact that *W* forwards *B* products only depends on the specific implementation of *W*. See class `boing.core.Functor` for product forwarding cases.

The wise worker and the auto-configuration feature

As formerly described, *worker* nodes are both consumers and producers, and they can be considered as the pipeline's processing units. Workers normally calculate simple or atomic operations because they can be easily serialized in order to compose more complex processing pipelines. Boing pipelines can be modified dynamically in order to evolve and fit a flexible environment. This may entail that not all the processing units are really necessary in order to compute the expected result. In order to avoid a waste of time, the pipeline exploits a auto-configuration technique based on the nodes' supply-demand knowledge. This technique, exploited by the *Wise Workers*, can be summarized into the following two rules:

1. the worker's request is nullified if no one requires the worker's own products;
2. the worker's offer is nullified if its own request is not satisfied.

As an example consider the pipeline in *figure 4.6*: the producer *P* provides the products *A*, which are required by the consumer *C*; this one also requires the products *B*, but *P* cannot provide them. For this reason the worker *W*, which can produce *B* from *A*, has been employed. Since *B* is required by *C*, *W* is currently active. In this example the worker *W* is set to forward all the products it receives even it is not directly interested to them.

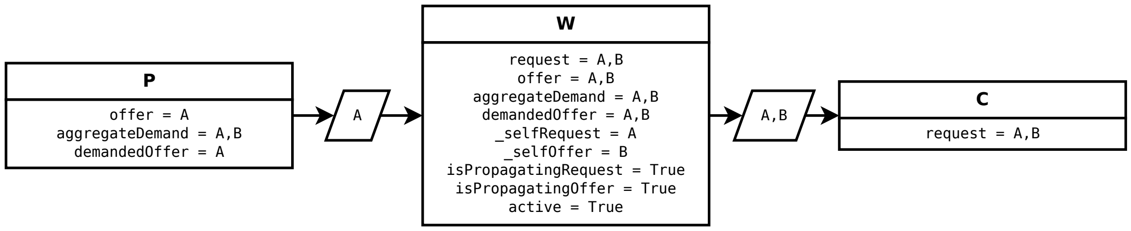

Fig. 1.13: The producer *P* provides the products *A*, while the worker *W* produces the products *B* using the products *A*. Both *A* and *B* are actually required by the consumer *C*.

Now suppose that the consumer *C* changes its own request to *A* only. In this case, nobody is interested to *B* anymore, thus, following the first rule of the *Wise Worker*, the worker stops requiring *A* for itself and it passes into an inactive state, but, since it is propagating *C*'s requests, it still requires *A* products. *Figure 4.7* shows the state of the pipeline in this case.

Considering the pipeline in *figure 4.6*, a different situation may arrive: if the producer *P* changes its offer to *D*, no one will provide the products *A*, thus, following the second rule of the *Wise Worker*, since the worker's request is not satisfied anymore, it nullifies its own offer. The resulted pipeline is shown in *figure 4.8*. In this case requests do not change, so that no more products are exchanged between the nodes.

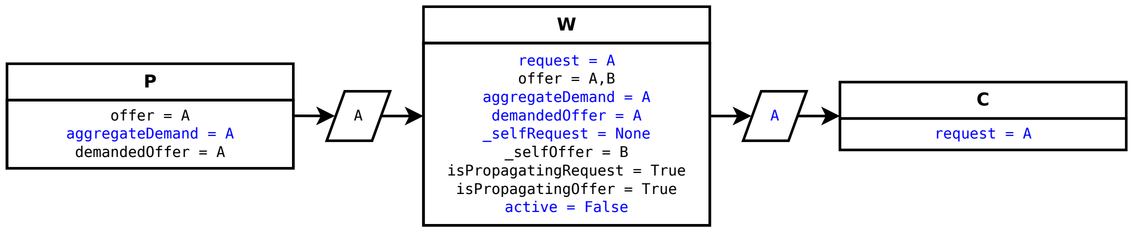

Fig. 1.14: If *C* does not require products *B* anymore, the worker *W* automatically stops producing them and requiring *A* products for itself, but since it is propagating *C*'s requests, it still requires *A* products so it can forward them to *C*.

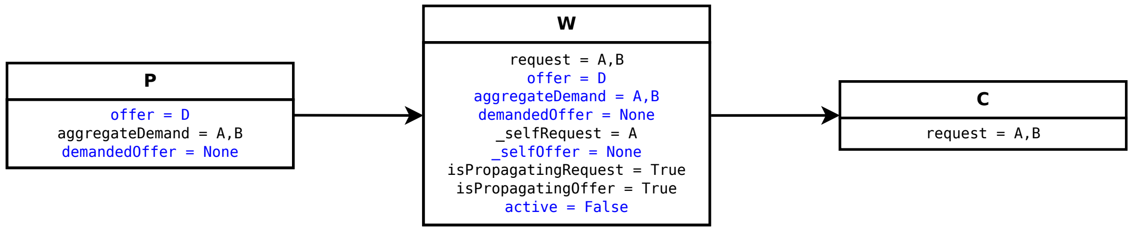

Fig. 1.15: Considering the pipeline of *figure 4.6*, if the producer *P* starts producing *D* only, the worker's request is not satisfied anymore, so it automatically nullifies its own offer.

In some cases workers do not previously know the products they provide since it only depends on the products they will receive. As an example, a worker may forward only a subset of the products it receives or it may make simple changes to the products it requires and then forward them. In those cases, it is not possible to set the offer in advance of the pipeline execution, thus the first rule of the *Wise Worker* cannot be applied. In order to handle those cases, the *Wise Workers* can use the *Tunneling* exception, that makes the first rule considering the entire propagated offer instead of the worker's own offer.

As an example consider the pipeline in *figure 4.9*: the worker *W* simply forwards the products it receives so it has not its own offer. Despite this, thanks to the tunneling exception, *W* is still active, since its global offer matches the request of *C*.

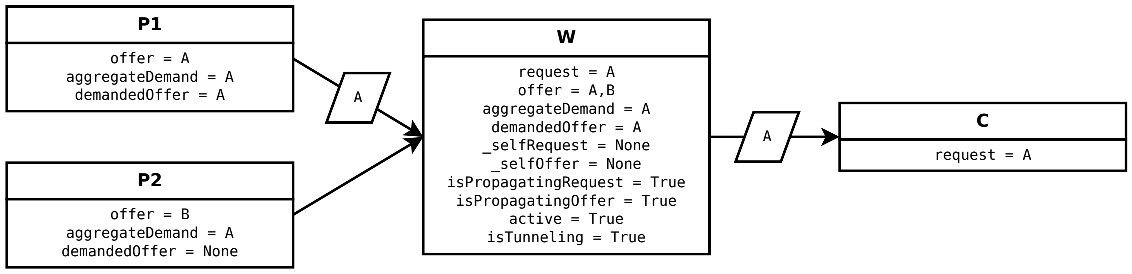

Fig. 1.16: When using the tunneling option, the propagated offer is considered to determine if the worker is active instead of its own offer only.

Concrete workers using the tunneling feature are the `Filter` and `Calibration` classes.

See also:

classes *`boing.core.WiseWorker`* and *`boing.core.Functor`*

Todo

- Describe the composite nodes and node syntax (+ and | operators).

Nodes reference table

Node URIs				
OSs	*Mode*	Value	Query keys[1]	Description Data Redirection
LWX	I			listen and decode data from an input device
LWX	O			encode and forward the data to a target destination Record/Replay
LWX	I		loop, speed, interval	replay a log file (default encoding)
LWX	O			record data to log file (default encoding)
LWX	O	rec:	request, timelimit, sizelimit, oversizecut, fps, timewarping	data recorder with GUI
LWX	I		interval, open	log files player with GUI (default encoding) Data Debug
LWX	O		request, mode, separator, src, dest, depth	dump products to an output device
LWX	O		request, fps	print products statistics to an output device
LWX	O	viz:	antialiasing, fps	display multi-touch contacts Data Processing
LWX	W	nop:		no operation node
LWX	W	edit:	merge, copy, result, **dict	apply to all the received products **dict
LWX	W	calib:	matrix, screen, attr, request, merge, copy, result	apply a 4x4 transformation matrix
LWX	W	filtering:[<filter-path>]	attr, request, merge, copy, result, <filter-attr>[2]	filter product data
LWX	W	timekeeper:	merge, copy, result	mark each received product with a timetag
LWX	W	lag:[<msec>]		add a lag to each received product Utils
LWX	[3]	conf:<filepath>		composite node containing the pipeline defined into the specified configuration file.

Encodings

Encodings[4]				
OSs	*Mode*	Value	Query keys	Description
LWX	IO	slip		bytestream from/to SLIP
LWX	I	pickle	noslip	pickle to products
LWX	O	pickle	protocol, request, noslip	Products to pickle
LWX	I	json	noslip	JSON to products
LWX	O	json	request, noslip	products to JSON
LWX	IO	osc	rt, noslip	bytestream from/to OSC
LWX	IO	tuio[.osc]	rawsource	Multi-touch events from/to TUIO

[1] The available query keys are obtained from the union of the available query keys of all the uri components. As an example, the URI `out.json://[::1]:7777` is by default translated to `out.json.udp://[::1]:7777`, so it owns the query keys of the JSON encoder (`request` and `filter`) and of the udp socket node (`writeend`).

[2] `<filter-attr>` dependes on the requested filter.

[3] The *mode* depends on the pipeline defined into the configuration file. It is important to note that pipelines may have a closed configuration, which means they do not behave neither as input nor output, nor worker. This happens when all the inputs are connected in series to the outputs.

[4] Some encodings have default input/output devices (e.g. `in.tuio:` is by default translated into `in.tuio.udp://[::]:3333`).

Input/Output devices

Input/Output devices				
OSs	*Mode*	Value	Query keys	Description
LX	I	:[stdin]		read from standard input
LWX	I	:[stdout]		write to standard output
LWX	I	[.file]:<filepath>	uncompress, postend	read from file
LWX	O	[.file]:<filepath>		write to file
LWX	I			read from UDP socket
LWX	O		writeend	write to UDP socket
LWX	IO		writeend	read/write on TCP socket

Hosts

Hosts			
OSs	*Mode*	Value	Description
LWX	I	*empty*	same as IPv4 any address
LWX	I	0.0.0.0	IPv4 any address
LWX	I	[::]	IPv6 any address
LWX	IO	127.0.0.1	IPv4 loopback
LWX	IO	[::1]	IPv6 loopback
LWX	IO	x.x.x.x	specific IPv4 address
LWX	IO	[x:x:x:x:x:x:x:x]	specific IPv6 address
LWX	IO	<hostname>	specific hostname

Modes

Modes	
Value	Description
I	Input
O	Output
W	Worker

See also:

Section *The pipeline architecture*

OS support

OS support	
Value	Description
L	Linux
W	Windows 7[5]
X	OS X

[5] On Windows, in order to define a file using the scheme `file:` it is necessary to place the character '/' (slash) before the drive letter (e.g. `file:///C:/Windows/explorer.exe`).

API documentation

This section presents the different modules that constitute the toolkit Boing. The documentation of the classes and functions of the toolkit's API is hereinafter provided.

Users API:

`boing` — Creating and managing pipelines

Developers can easily deploy Boing pipelines by invoking the toolkit's API in their Python code. The most important element is the function `boing.create()`, which is used to instantiate the nodes of the pipeline.

boing.**create**(*expr*, *parent=None*)

Return a new node created as defined in the expression *expr*, with parent object *parent*. If *expr* is composed by a single URI, the returned object will be a new node correspondent to the provided URI; if *expr* is formed by an URI expression, the returned object will be a composed node.

All the available nodes are listed and described in the *Nodes reference table*.

In order to compose the pipeline, the nodes can be attached using the Python operators + and `|`, which work the same as the operators used in the URI expressions. As an example, consider the following code:

```
n1 = boing.create("in.tuio:")
n2 = boing.create("viz:")
n3 = boing.create("dump:")
pipeline = n1 + (n2 | n3)
```

The same pipeline can be obtained using the following code:

```
pipeline = boing.create("in.tuio:+(viz:|dump:)")
```

In order to run the pipeline it is necessary to launch the Qt Application that should have been initialized before creating the pipeline. The following code can be used as an example for creating custom Boing pipelines:

```
#!/usr/bin/env python3
import sys
import PyQt4
import boing

# Init application
app = PyQt4.QtGui.QApplication(sys.argv)

# Create nodes
n1 = boing.create("in.tuio:")
n2 = boing.create("viz:")
n3 = boing.create("dump:?request=$..contacts")

# Compose the pipeline
graph = n1 + (n2 | n3)

# Run
sys.exit(app.exec_())
```

Global configuration

Any global configuration variable can be stored in the attribute `boing.config`.

`boing.`**`config`**

`dict` object used to store any global configuration variable. Boing's own variables:

- `"--no-gui"`: defines whether the GUI widgets are enabled. If set to `True`, nodes likes `viz:`, `rec:`, `player:`, etc., cannot be created.
- `"--no-raise"`: defines whether the GUI widgets should be automatically raised as they are initialized (see method `QtGui.QWidget.raise_()`).

Dynamic configuration

Todo

Describe how to configure the pipeline dinamically

`boing.`**`activateConsole`**`(`*url=""*, *locals=None*, *banner=None*`)`

Enable a Python interpreter at *url*.

The optional *locals* argument specifies the dictionary in which code will be executed; it defaults to a newly created dictionary with key "__name__" set to "__console__" and key "__doc__" set to None.

The optional *banner* argument specifies the banner to print before the first interaction; by default it prints a banner similar to the one printed by the real Python interpreter.

The pipeline architecture:

`boing.core` — The pipeline infrastructure

The module `boing.core` contains all the classes that constitute the infrastructure of Boing pipelines.

The `Producer` and `Consumer` classes are build over the Observer design pattern of the module `boing.core.observer`: the Producer is an Observable object enabled to post products; a consumer is an Observer object that can subscribe itself to many producers. When a producer has a new product, it triggers the registered consumers; the triggered consumers will immediately or at regular time interval demand the producer the new products.

A `Worker` object is both a `Producer` and a `Consumer` and it is used as base class for defining processing nodes. By connecting producers, workers and consumers it is possible to create processing pipelines.

A `WiseWorker` is a `Worker` able to automatically detect whenever it should not propose its own offer or its own request; this is done in order to save computational time.

The `Functor` class is practical base class for inheriting custom processing `Worker` objects. Instead of implementing the classic `Consumer._consume()` method, the `Functor` proposes the more powerfull method `Functor._process()`.

Multiple `Producer`, `Consumer` and `Worker` instances can be composed into `Composite` objects. There are three types of composites:

- a `CompositeProducer` works as it was a single producer;
- a `CompositeConsumer` works as it was a single consumer;
- a `CompositeWorker` works as it was a single worker;

See also:

The pipeline architecture

Producers

class `boing.core.`**`Producer`**(*offer*, *tags=None*, *store=None*, *retrieve=None*, *haspending=None*, *parent=None*)

`Producer` instances are `Observable` objects able to post products to a set of subscribed `Consumer` instances. The argument *offer* must be an instance of `Offer` and it define the products this producer will supply, while *tags* must be a dict or None. The argument *store* can be a callable object to be used as a handler for storing posted products (see `_store()` for the handler arguments) or None, while *retrieve* can be a callable object to be used as a handler for retrieving stored products (see `_retrieveAndDeliver()` for the handler arguments) or None. *parent* defines the consumer's parent.

When a producer is demanded to posts a product, for each registered consumer it tests the product with the consumer's request and only if the match is valid it triggers the consumer.

Public signals:

`demandChanged`
Signal emitted when the aggregate demand changes.

`offerChanged`
Signal emitted when its own offer changes.

`demandedOfferChanged`
Signal emitted when its own demanded offer changes.

Available methods:

`aggregateDemand`()
Return the union of all the subscribed consumers' requests.

`demandedOffer`()
Return the producer's demanded offer.

`meetsRequest`()
Return whether the product's offer meets *request*.

`offer`()
Return the producer's offer.

`postProduct`(*product*)
Post *product*. In concrete terms, it triggers the registered consumers that require *product*, then it stores the product.

class `boing.core.`**`Offer`**(**args*, *iter=None*)

An `Offer` defines the list of products that a producer advertises to be its deliverable objects.

Note: A producer's offer only estimates the products that are normally produced. There is no guarantee that such products will ever be posted, neither that products that do not match the offer won't be produced.

`Offer.UNDEFINED` can be used to define the producer's offer, when the real offer cannot be defined a priori. This avoids to have empty offers, when they cannot be predeterminated.

Consumers

class `boing.core.`**`Consumer`**(*request*, *consume=None*, *hz=None*, *parent=None*)

`Consumer` objects are `Observer` objects that can be subscribed to several `Producer` instances for receiving their products. When a producer posts a product, it triggers the registered consumers; then the consumers will immediately or at regular time interval demand to the producer the new products.

Warning: Many consumers can be subscribed to a single producer. Each new product is actually shared within the different consumers, therefore a consumer **MUST NOT** modify any received product, unless it is supposed to be the only consumer.

Consumers have a `Request`. When a producer is demanded to posts a product, it tests the product with the consumer's request and only if the match is valid it triggers the consumer.

`request`()
Return the consumer's request.

`_consume`(*products*, *producer*)
Consume the *products* posted from *producer*.

class `boing.core.`**`Request`**
The class `Request` is an abstract class used by `Consumer` objects for specifing the set of products they are insterested to. The method `test()` is used to check whether a product matches the request.

`Request.NONE` and `Request.ANY` define respectively a "no product" and "any product" requests.

`Request` objects may also indicate the internal parts of a product to which a producer may be interested. The method `items()` returns the sequence of the product's parts a producer is interested to.

The class `Request` implements the design pattern "Composite": different requests can be combined into a single request by using the sum operation (e.g. `comp = r1 + r2`). A composite request matches the union of the products that are matched by the requests whom it is composed. `Request.NONE` is the identity element of the sum operation.

`Request` objects are immutable.

`test`(*product*)
Return whether the *product* matches the request.

`items`(*product*)
Return an iterator over the *product*'s internal parts (i.e. (key, value) pairs) that match the request.

class `boing.core.`**`QRequest`**(*string*)
The `QRequest` is a `Request` defined by a `QPath`.

Workers

class `boing.core.`**`Worker`**
A `Worker` instance is both a `Producer` and a `Consumer`. The class `Worker` is by itself an abstract class. Consider using the concrete class `BaseWorker` instead.

class `boing.core.`**`BaseWorker`**(*request*, *offer*, *tags=None*, *store=None*, *retrieve=None*, *haspending=None*, *consume=None*, *hz=None*, *parent=None*)
A `BaseWorker` is the simplest concrete `Worker`. By default it does nothing with the products it receives, but it is able to propagate the offers of the producers it is subscribed to, and it is able to propagate the requests of the consumers that are subscribed to it.

class `boing.core.`**`NopWorker`**(*store=None*, *retrieve=None*, *hz=None*, *parent=None*)
`NopWorker` instances simply forwards all the products they receive.

class `boing.core.`**`WiseWorker`**(*request*, *offer*, ***kwargs*)
A `WiseWorker` instance is able to automatically detect whenever it should not propose its own offer or its own request; this is done in order to save computational time. The behaviour of `WiseWorker` objects works as follows:

1. Its own request is deactivated if there is no consumer insterested to its offer (i.e. no items in demandedoffer), which means: <<If nobody is interested to its products, why should it requires the products necessary to the processing?>>.
2. Its own offer is deactivated if there is no producer that meets its request, which means: <<If nobody can provide it the products it requires for processing, how can it propose its offer?>>.

A `WiseWorker` is active (see method `isActive()`) when its own request is not deactivated (i.e. case 1).

A `WiseWorker` can be forced to be activated or deactivated. Use the method `setForcing()` to impose a specific behaviour.

Use `WiseWorker.TUNNELING` as the worker Offer, when the worker is supposed to process and then forward a subset of the products it receives. This is necessary since sometimes it may be possible that the worker cannot have a predefined offer since it always depend from the offer of the observed producers.

isActive()
: The `WiseWorker` is active only if its offer is requested from any subscribed consumer, unless it is forced (see `forcing()`).

isTunneling()
: Return True if the worker is forwarding a subset of the products it receives.

forcing()
: Return whether the WiseWorker is being forced. Return a value within (`WiseWorker.ACTIVATED`, `WiseWorker.DEACTIVATED`, `WiseWorker.NONE`).

setForcing(*forcing*)
: Impose to the `WiseWorker` to be activated, deactivated or remove a previous forcing. *forcing* must be a value within (`WiseWorker.ACTIVATED`, `WiseWorker.DEACTIVATED`, `WiseWorker.NONE`).

ACTIVATED
: Value used to force the `WiseWorker` to be activated.

DEACTIVATED
: Value used to force the `WiseWorker` to be deactivated.

NONE
: Value used to let the `WiseWorker` decide it it should be active.

TUNNELING
: Value used to set the `WiseWorker` the tunneling mode.

class `boing.core.`**`Functor`**(*args*, *offer*, *blender*, *process=None*, ***kwargs*)
: The `Functor` class is practical base class for inheriting custom processing `Workers`. Instead of implementing the classic `Consumer._consume()` method, the `Functor` proposes the more powerfull method `_process()`. This handler method receives as argument *sequence*, an iterator over the operands, which are iterators over the couples (key, value) obtained from applying the method `Request.items()` of the request of the functor on the list of received products. This enables to access directly to the name and values of the required data without the need of reimplementing the code to get them. The method `_process()` is a generator method and it it supposed to yield the the couples (key, value) representing the result of the node processing. The yielded results are automatically considered by the functor to create a new product that will be automatically posted.

The functor uses a `Functor.Blender` object to create the new product. A set of predefined blenders are used to set the functor behaviour:

- `Functor.MERGE` — Join the original product and results of the functor.
- `Functor.MERGECOPY` — Make a deep copy of the original product and then join the results of the functor.

- `Functor.RESULTONLY` — Post as product only the result of the functor.

A *`Functor`* instance propagates the requests only if the current blender is a `MergeBlender` and it propagates the offers only if the current blender is a `MergeBlender` or if it is tunneling (see *`WiseWorker.isTunneling()`*).

`blender``()`
Return the current active `Functor.Blender`.

`_process``(`*sequence*, *producer*`)`
This handler method receives as argument *sequence*, an iterator over the operands, which are iterators over the couples (key, value) obtained from applying the method *`Request.items()`* of the request of the functor on the list of received products. This enables to access directly to the name and values of the required data without the need of reimplementing the code to get them. This is a generator method and it it supposed to yield the the couples (key, value) representing the result of the node processing.

Composites

class `boing.core.`**`Composite`**`(`**internals*, *parent=None*`)`
A *`Composite`* instance stores a set of internal objects using strong references. Use the method *`internals()`* to scroll the references objects.

`internals``()`
Return an iterator over the set of internal nodes for which a strong reference is stored.

class `boing.core.`**`CompositeProducer`**`(`**producers*, *internals=set()*, *parent=None*`)`
A *`CompositeProducer`* combines in parallel the list of *producers* so that they can be considered as a single *`Producer`*. The argument *internals* can be used to specify the object for which a strong reference should be kept. All *producers* are by default added as internals of the *`Composite`*.

`producers``()`
Return an iterator over the first level producers.

class `boing.core.`**`CompositeConsumer`**`(`**consumers*, *internals=set()*, *parent=None*`)`
A *`CompositeConsumer`* combines in parallel the list of *consumers* so that they can be considered as a single *`Consumer`*. The argument *internals* can be used to specify the object for which a strong reference should be kept. All *consumers* are by default added as internals of the *`Composite`*.

`consumers``()`
Return an iterator over the first level consumers.

class `boing.core.`**`CompositeWorker`**`(`*consumers*, *producers*, *internals=set()*, *parent=None*`)`
A *`CompositeWorker`* object combines in parallel the list of *producers* and in parallel the list of *consumers* so that they can be considered as a single *`Worker`*. The argument *internals* can be used to specify the object for which a strong reference should be kept. All *producers* and *consumers* are by default added as internals of the *`Composite`*.

`boing.core.observer` — Observables and observers

The module *`boing.core.observer`* provides an implementation of the Observer design pattern.

Rather than the standard behaviour, this implementation enables the *`Observable`* objects to trigger only a subset of all the current registered *`Observer`* objects.

Note: The notification mechanism relies on the SIGNAL-SLOT mechanism of the Qt eventloop, thus a QApplication must be running in order to ensure that notifications are processed.

class `boing.core.observer.`**`Observable`**(*parent=None*)

`Observable` objects can be subscribed by a list of *`Observer`* instances. An Observable can trigger all or only a subset of the subscribed observers by invoking the method `trigger()`. The attribute *parent* defines the parent object of the observable.

`observerAdded`

Signal emitted when a new observer is added.

`observerRemoved`

Signal emitted when a registered observer is removed.

`observers`()

Return an iterator over the subscribed observers.

`addObserver`(*observer*, *mode=QtCore.Qt.QueuedConnection*)

Subscribe *observer* as a new observer. Return whether *observer* has been correctly added.

`removeObserver`(*observer*)

Unsubscribe *observer*. Return whether *observer* has been correctly removed.

`clear`()

Unsubscribe all registered observers.

`notify`(**restrictions*)

Trigger all the subscribed observers if *restrictions* is empty, otherwise trigger only the registered observers in *restrictions*.

Note: The list of the subscribed observers is composed by weak references, so it is necessary to keep both observables and observers alive.

class `boing.core.observer.`**`Observer`**(*react=None*, *hz=None*, *parent=None*)

`Observer` objects can be subscribed to many *`Observable`* instances in order to listen to their notifications. The argument *react* can be set to the handler function (it must accept one argument) that will be called as consequence of an observable notification. If *react* is None, the member method *`_react()`* will be called. *hz* defines the rate at which the observer will react to notifications. Available values are:

- None — immediately;
- 0 — never;
- <float> — at the selected frequency (in hz).

parent defines the observers parent.

`observableAdded`

Signal emitted when the observer is subscribed to a new observable.

`observableRemoved`

Signal emitted when the observer is unsubscribed from an observable.

`observed`()

Return an iterator over the observables it is subscribed to.

`subscribeTo`(*observable*, *mode=QtCore.Qt.QueuedConnection*)

Subscribe to *observable*. Return whether *observer* has been successfully subscribed to.

`unsubscribeFrom`(*observable*)

Unsubscribe from *observable*. Return whether *observable* has been successfully found and removed.

`clear`()

Unsubscribe from all observed observables.

hz()
Return when the observer will react to the notifications. Possible values:

- None — immediately;
- 0 — never;
- <float> — at the selected frequency (in hz).

queue()
Return an iterator over the observables that have triggered without having being reacted to yet.

_react(*observable*)
Handler method invoked as a result of the notification of *observable*, but only if the `Observer` instance has not been created with a custom *react* handler.

Note: The list of the subscribed observables is composed by weak references, so it is necessary to keep both observables and observers alive.

Preconfigured nodes:

`boing.nodes` — The nodes of the pipeline

The module `boing.nodes` contains a set of generic utility nodes.

Device input/output

class `boing.nodes.`**`DataReader`**(*inputdevice*, *postend=True*, *parent=None*)
`Producer` node that anytime the device *inputdevice* send the signal `readyRead` it reads the device and it produces a message containing the data. The provided products is a dictionary `{"str": data}` if `data` is a string, otherwise the product will be a dictionary like `{"data": data}`. If the argument *postend* is set to `True`, the `DataReader` will never produce an empty product, like `{"str": ""}` or `{"data": b""}`. *parent* defines the parent of the node.

inputDevice()
Return the considered input device.

class `boing.nodes.`**`DataWriter`**(*outputdevice*, *writeend=True*, *hz=None*, *parent=None*)
`Consumer` node that anytime it receives some data, it writes the data to the device *outputdevice*. The `DataWriter` requires the products `str` if the output device is text enabled (see method `isTextModeEnabled`) otherwise it requires the product `data`. If the argument *writeend* is set to `True`, the `DataWriter` will never write an empty string; this can be useful in order to prevent a socket to close. *parent* defines the parent of the node.

outputDevice()
Return the considered output device.

Products debugging

class `boing.nodes.`**`Dump`**(*request=Request.ANY*, *mode='items'*, *separator='\n\n'*, *src=False*, *dest=False*, *depth=None*, *parent=None*)
Instances of the `Dump` class produce a string representation of the products they receive. The string is obtained using the function `boing.utils.deepDump()`.

The parameter *request* must be an instance of the class `boing.core.Request` and it is used to select the product to be dumped. The default value for request is `Request.ALL`. *mode* defines how the received products will be dumped. The available values are:

- `'keys'`, only the matched keys are written;
- `'values'`, only the values of the matched keys are written;
- `'items'`, both the keys and values are written.

separator defines the string to be written between two products. The default value for separator is `'\n\n'`. *src* defines whether the node also dumps the producer of the received products. The default for src is False. The paramenter *dest* defines whether the node adds a reference to itself when it dumps the received products; its default value is False. The parameter *depth* defines how many levels of the data hierarchy are explored and it is directly passed to the `boing.utils.deepDump()` function.

`mode`()
Return the node's mode.

`setMode`(*mode*)
Set the node's dump *mode*.

Products editing

class `boing.nodes.`**`Editor`**(*dict*, *blender*, *parent=None*)
Instances of the `Editor` class apply to the received products the (key, values) pairs of *dict*.

blender defines the output of the node (see `boing.core.Functor`). *parent* must be a `PyQt4.QtCore.QObject` and it defines the node's parent.

`get`(*key*, *default=None*)
Return the value for *key* if *key* is in the editor's dictionary, else *default*. If *default* is not given, it defaults to None.

`set`(*key*, *value*)
Set the value for *key* to *value*.

`items`()
Return a new view of the editor dictionary's items ((key, value) pairs).

class `boing.nodes.`**`DiffArgumentFunctor`**(*functorfactory*, *request*, *blender=Functor.MERGECOPY*, *parent=None*)
It takes a functorfactory and for each different argument path, it creates a new functor which is applied to the argument value. The args must be a diff-based path so that functor can be removed depending on 'diff.removed' instances.

Timing utilities

class `boing.nodes.`**`Timekeeper`**(*blender=Functor.MERGECOPY*, *parent=None*)
Instances of the `Timekeeper` class tag each received product with the timestamp when the product is received; then they forward the product.

blender defines the output of the node (see `boing.core.Functor`). *parent* must be a `PyQt4.QtCore.QObject` and it defines the node's parent.

class `boing.nodes.`**`Lag`**(*msec*, *parent=None*)
Instances of the `Lag` class forward the received products after a delay.

The parameter *msec* defines the lag in milliseconds. *parent* must be a `PyQt4.QtCore.QObject` and it defines the node's parent.

Filtering algorithms:

`boing.filtering` - Filters and noise generators

Smoothing filters and noise generators.

Todo

Comment module *`boing.filtering`*.

Gesture recognition algorithms:

`boing.gesture.rubine` — The Rubine's recognition algorithm

The module *`boing.gesture.rubine`* provides an implementation of the Rubine's gesture recognition algorithm.

`boing.gesture.utils` — Recognizers' common utilities

The boing.gesture.utils module contains common method used by different recognizers.

`boing.gesture.utils.`**`boundingBox`**(*points*)

Return the tuple (minx, miny, maxx, maxy) defining the bounding box for *points*.

`boing.gesture.utils.`**`updateBoundingBox`**(*bb1*, *bb2*)

Return the tuple (minx, miny, maxx, maxy) defining the bounding box containing the bounding boxes *bb1* and *bb2*.

Networking utilities:

`boing.net` — Networking and encoding tools

The module *`boing.net`* provides classes and methods to ease the usage of sockets and networking encodings, like for example JSON, OSC, SLIP, etc.

class `boing.net.`**`Encoder`**

Abstract base class for implementing the encoders of all the different encodings.

`encode`(*obj*)

`@abc.abstractmethod`

Return the result obtained from encoding *obj*.

`reset`()

`@abc.abstractmethod`

Reset the encoder.

class `boing.net.`**`Decoder`**

Abstract base class for implementing the decoders of all the different encodings.

The Decoder class implements the composite pattern. Many decoders can be put in sequence into a single composed decoder using the sum operator.

`decode`(*obj*)

`@abc.abstractmethod`

Return the list of objects obtained from decoding *obj*.

`reset`()

`@abc.abstractmethod`

Reset the decoder.

`boing.net.bytes` — UNICODE encoding

The *`boing.net.bytes`* module implements the adapter design pattern by providing the standard string encoding functionalities as Encoder and Decoder objects.

`boing.net.bytes.`**`encode`**(*string*, *encoding="utf-8"*, *errors="strict"*)

Return an encoded version of *string* as a bytes object. Default encoding is 'utf-8'. *errors* may be given to set a different error handling scheme. The default for errors is 'strict', meaning that encoding errors raise a UnicodeError. Other possible values are 'ignore', 'replace', 'xmlcharrefreplace', 'backslashreplace'

`boing.net.bytes.`**`decode`**(*data*, *encoding="utf-8"*, *errors="strict"*)

Return a string decoded from the given bytes. Default *encoding* is 'utf-8'. *errors* may be given to set a different error handling scheme. The default for errors is 'strict', meaning that encoding errors raise a UnicodeError. Other possible values are 'ignore', 'replace' and any other name registered via codecs.register_error().

class `boing.net.bytes.`**`Encoder`**(*encoding="utf-8"*, *errors="strict"*)

The Encoder is able to produce encoded version of string objects as byte objects.

`encode`(*string*)

Return an encoded version of *string* as a bytes object.

`reset`()

NOP method.

class `boing.net.bytes.`**`Decoder`**(*encoding="utf-8"*, *errors="strict"*)

The Decoder is able to convert byte objects into strings.

`decode`(*data*)

Return the list of strings decoded from the given bytes.

`reset`()

NOP method.

`boing.net.slip` — SLIP encoding

The module *`boing.net.slip`* provides methods and classes for supporting the SLIP protocol encoding and decoding.

`boing.net.slip.`**`encode`**(*data*)

Return a slip encoded version of *data*.

boing.net.slip.**decode**(*data*, *previous=None*)

Return the list of bytearrays obtained from the slip decoding of *data* followed by the undecoded bytes. If previous is not None, *data* is appended to previous before decoding. A typical usage would be:

```
buffer = bytearray()
decoded, buffer = decode(data, buffer)
```

class boing.net.slip.**Encoder**

The Encoder is able to produce slip encoded version of byte strings.

encode(*obj*)

Return a slip encoded version of the byte string *obj*.

reset()

NOP method.

class boing.net.slip.**Decoder**

The Decoder object is able to decode slip encoded byte strings into the their internal components.

decode(*obj*)

Return the list of bytearrays obtained from the slip decoding of *obj*.

reset()

Reset the slip internal buffer.

`boing.net.json` — JSON encoding

The module *boing.net.json* provides methods and classes for supporting JSON object serialization. It uses the python json standard module, but it provides a default solution for serializing bytestrings and datetime.datetime objects.

Encoder and Decoder classes provide a standard interface for the JSON encoding.

boing.net.json.**encode**(*obj*)

Return a string containing the json serialization of *obj*.

boing.net.json.**decode**(*string*)

Return the object obtained for decoding *string* using the JSON decoding.

class boing.net.json.**Encoder**

The Encoder is able to serialize standard data types into json strings.

encode(*obj*)

Return a string containing the json serialization of *obj*.

reset()

NOP method.

class boing.net.json.**Decoder**

The Decoder object is able to decode json strings into the corrispetive python objects.

decode(*string*)

Return the list of object obtained from the deserialization of *string*.

reset()

NOP method.

`boing.net.osc` — OSC encoding

The module *boing.net.osc* provides methods and classes for handling OSC formatted messages.

Container classes

class `boing.net.osc.`**`Packet`**

Abstract base container of OSC data.

`encode``()`

`@abc.abstractmethod`

Return the encoded representation of this packet.

`debug``(`*out*, *indent=”“*`)`

`@abc.abstractmethod`

Write to *out* a string representation of the OSC packet. The argument *indent* can be used to format the output.

class `boing.net.osc.`**`EncodedPacket`**`(`*data*`)`

`Packet` object representing the encoded object *data*.

`decode``()`

Return the decoded representation of this packet, that is an instance of the class *`Bundle`* or *`Message`*.

class `boing.net.osc.`**`Message`**`(`*address*, *typetags=”“*, **arguments*`)`

`Packet` object representing an OSC Message. The argument *address* must be a string begginning with the character `/` (forward slash). The argument *typetags* must be a string composed by sequence of characters corresponding exactly to the sequence of OSC arguments in the given message. *arguments* is the list of object contained in the OSC Message.

`address`

String beginning with the character `/` (forward slash).

`typetags`

String composed by a sequence of characters corresponding exactly to the sequence of OSC arguments in the given message.

`arguments`

List of the arguments of the message.

class `boing.net.osc.`**`Bundle`**`(`*timetag*, *elements*`)`

`Packet` object representing an OSC Bundle. The argument *timetag* must be a `datetime.datetime` instance or `None`, while *elements* should be the list of *`Packet`* objects contained in the bundle.

`timetag`

`None` or a `datetime.datetime` instance.

`elements`

List of *`Packet`* objects contained into the bundle.

Encoding and decoding

`boing.net.osc.`**`decode`**`(`*data*, *source=None*`)`

Return the *`Packet`* object decoded from the bytestring *data*. The argument *source* can be specified to set the packet source.

The classes *`Encoder`* and *`Decoder`* provide a standard interface for the OSC encoding.

class `boing.net.osc.`**`Encoder`**

Implements the `boing.net.Encoder` interface for encoding OSC packet objects into byte strings.

`encode`(*obj*)

Return the bytestring obtained from serializing the OSC packet *obj*.

`reset`()

NOP method.

class `boing.net.osc.`**`Decoder`**

Implements the `boing.net.Decoder` interface for converting valid byte string objects into OSC Packet objects.

`decode`(*obj*)

Return the list of OSC packets decoded from the bytestring *obj*.

`reset`()

NOP method.

Usage example

```
>>> import sys
>>> import boing.net.osc as osc
>>> source = osc.Message("/tuio/2Dcur", "ss", "source", "test")
>>> alive = osc.Message("/tuio/2Dcur", "ss", "alive", "1")
>>> bundle = osc.Bundle(None,
                       (source, alive,
                        osc.Message("/tuio/2Dcur", "si", "fseq", 1)))
>>> data = bundle.encode()
>>> print(data)
b'#bundle\x00\x00\x00\x00\x00\x00\x00\x00\x01\x00\x00\x00 /tuio/2Dcur\x00,
↪ss\x00source\x00\x00test\x00\x00\x00\x00\x00\x00\x00\x1c/tuio/2Dcur\x00,
↪ss\x00alive\x00\x00\x001\x00\x00\x00\x00\x00\x00\x1c/tuio/2Dcur\x00,
↪si\x00fseq\x00\x00\x00\x00\x00\x00\x00\x01'
>>> packet = osc.decode(data)
>>> print(packet)
<Bundle instance at 0x1b756d0 [@None, 3 element(s)]>
>>> packet.debug(sys.stdout)
Bundle IMMEDIATELY
 | /tuio/2Dcur ss 'source' 'test'
 | /tuio/2Dcur ss 'alive' '1'
 | /tuio/2Dcur si 'fseq' 1
```

`boing.net.pickle` — Python pickle encoding

The module *`boing.net.pickle`* provides methods and classes for supporting Python object serialization. It uses the python pickle standard module.

Encoder and Decoder classes provide a standard interface for the pickle encoding.

`boing.net.pickle.`**`encode`**(*obj*, *protocol=None*)

Return the pickled representation of *obj* as a bytes object.

The optional protocol argument tells the pickler to use the given protocol; supported protocols are 0, 1, 2, 3. The default protocol is 3; a backward-incompatible protocol designed for Python 3.0.

Specifying a negative protocol version selects the highest protocol version supported. The higher the protocol used, the more recent the version of Python needed to read the pickle produced.

`boing.net.pickle.`**`decode`**(*data*)

Read a pickled object hierarchy from the bytes object *data* and return the reconstituted object hierarchy specified therein.

The protocol version of the pickle is detected automatically, so no protocol argument is needed. Bytes past the pickled object's representation are ignored.

class `boing.net.pickle.`**`Encoder`**

The Encoder is able to serialize Python objects into pickle bytestrings.

`encode`(*obj*)

Return the pickled representation of *obj* as a bytes object.

The optional protocol argument tells the pickler to use the given protocol; supported protocols are 0, 1, 2, 3. The default protocol is 3; a backward-incompatible protocol designed for Python 3.0.

Specifying a negative protocol version selects the highest protocol version supported. The higher the protocol used, the more recent the version of Python needed to read the pickle produced.

`reset`()

NOP method.

class `boing.net.pickle.`**`Decoder`**

The Decoder object is able to decode pickle bytestrings into the corrispetive objects hierarchy.

`decode`(*obj*)

Read a pickled object hierarchy from the bytes object *data* and return the reconstituted object hierarchy specified therein.

The protocol version of the pickle is detected automatically, so no protocol argument is needed. Bytes past the pickled object's representation are ignored.

`reset`()

Reset the slip internal buffer.

`boing.net.ip` — IP utilities

The module *`boing.net.ip`* provides few functions related to IP addressing.

`boing.net.ip.`**`resolve`**(*addr*, *port*, *family=0*, *type=0*)

Return a pair (addr, port) representing the IP address associated to the host *host* for the specified port, family and socket type.

`boing.net.ip.`**`addrToString`**(*addr*)

Return a string representing the QHostAddress *addr*.

`boing.net.tcp` — TCP utilities

Todo

Improve docs for the module boing.net.tcp

`boing.net.udp` — UDP utilities

Todo

Improve docs for the module boing.net.udp

`boing.net.ntp` — NTP utilities

The module *`boing.net.ntp`* provides few functions for handling the Network Time Protocol (NTP).

`boing.net.ntp.`**`ntpEncode`**(*t*)
: Return the bytes object obtained from encoding the POSIX timestamp *t*.

`boing.net.ntp.`**`ntpDecode`**(*data*)
: Return the POSIX timestamp obtained from decoding the bytes object *data*.

`boing.net.ntp.`**`ntpFromServer`**(*server*)
: Send an ntp time query to the *server* and return the obtained POSIX timestamp. The request is sent by using an UDP connection to the port 123 of the NTP server.

`boing.net.ntp.`**`ntp2datetime`**(*t*)
: Return the `datetime.datetime` instance corresponding to the POSIX timestamp *t*.

`boing.net.ntp.`**`datetime2ntp`**(*dt*)
: Return the POSIX timestamp corresponding to the `datetime.datetime` instance *dt*.

Usage example:

```
>>> import datetime
>>> import boing.net.ntp as ntp
>>> srvtime = ntp.ntpFromServer("europe.pool.ntp.org")
>>> srvdatetime = ntp.ntp2datetime(srvtime)
>>> now = datetime.datetime.now()
>>> print("Server time:", srvdatetime)
Server time: 2012-10-16 16:39:12.332479
>>> print("Local time:", now)
Local time: 2012-10-16 16:40:23.772048
>>> print("Delta:", now - srvdatetime)
Delta: 0:01:11.439569
```

Generic utilitities:

`boing.utils` — Common utilities

The module *`boing.utils`* contains generic utility classes and functions.

`boing.utils.`**`assertIsInstance`**(*obj*, **valid*)
: Raise TypeError if *obj* is not an instance of a class in *valid*.

`boing.utils.`**`deepDump`**(*obj*, *fd=sys.stdout*, *maxdepth=None*, *indent=2*, *end="n"*, *sort=True*)
: Write to *fd* a textual representation of *obj*.

class `boing.utils.`**`StateMachine`**(*initial=None*)
: The *`StateMachine`* class defines an object that owns a state defined by a `collections.Mapping` type object. The argument *initial* can be used to define the initial state.

  `state`()
  : Return the current state.

  `setState`(*update=None*, *add=None*, *remove=None*)
  : Change the current state by applying *update*, *add* and *remove*.

applyDiff(*diff*, *feedback=False*)
Apply the provided *diff* to the current state. *diff* must be a `collections.Mapping` type containing any of the following keys:

- `'add'`: items that will be added to the current state;
- `'update'` : items that will be update or added to the current state;
- `'remove'` : items that will be removed from the current state.

If feedback is `True` the diff structure between the previous state and the current state is provided as result of the method.

class `boing.utils.`**Console**(*inputdevice*, *outputdevice*, *banner=""*, *locals=None*, *parent=None*)
Interactive Python console running along the Qt eventloop.

push(*line*)
Pass *line* to the Python interpreter.

`boing.utils.fileutils` — File utilities

The module *`boing.utils.fileutils`* provides some useful classes for having a standard interface over standard device files, Unix non-standard files and tty devices.

It has also been created because it was necessary a more extensive support to the Unix non-standard devices (i.e. named pipes and device files) rather the one provided from the Qt framework.

Input/Output devices

class `boing.utils.fileutils.`**IODevice**(*fd*, *parent=None*)
The *`IODevice`* provides a standard interface for the generic file descriptors *fd*, which it could be a file, the stdin or the stdout.

bytesWritten
Signal emitted when a write operation has been effectuated.

aboutToClose
This signal is emitted when the device is about to close. Connect this signal if you have operations that need to be performed before the device closes (e.g., if you have data in a separate buffer that needs to be written to the device).

fd()
Return the device's file descriptor.

isatty()
Return `True` if the stream is interactive (i.e., connected to a terminal/tty device)

isOpen()
Return whether the device is open.

isTextModeEnabled()
Return whether the device provides unicode string rather than bytestream using the read methods.

bytesToWrite()
Return the number of bytes that are waiting to be written.

flush()
Flush the write buffers of the stream if applicable. This does nothing for read-only and non-blocking streams.

close()
Flush and close this stream. This method has no effect if the file is already closed. Once the file is closed, any operation on the file (e.g. reading or writing) will raise a `ValueError`.

read(*size=io.DEFAULT_BUFFER_SIZE*)
Read and return *size* bytes, or if n is not given or negative, until EOF or if the read call would block in non-blocking mode.

readline(*limit=-1*)
Read and return one line from the stream. If *limit* is specified, at most *limit* bytes will be read.

The line terminator is always `b'\n'` for binary files; for text files, the *newlines* argument to `open()` can be used to select the line terminator(s) recognized.

readall()
Read and return all the bytes from the stream until EOF, using multiple calls to the stream if necessary.

seek(*offset*, *whence=io.SEEK_SET*)
Change the stream position to the given byte *offset*. *offset* is interpreted relative to the position indicated by *whence*. Values for *whence* are:

- `SEEK_SET` or `0` – start of the stream (the default); *offset* should be zero or positive
- `SEEK_CUR` or `1` – current stream position; *offset* may be negative
- `SEEK_END` or `2` – end of the stream; *offset* is usually negative

Return the new absolute position.

write(*data*)
Write the given bytes or bytearray object, b and return the number of bytes written. It also emit the signal *`bytesWritten`*.

The *`IODevice`* class also defines the following constants:

ReadOnly

WriteOnly

ReadWrite

Append

class `boing.utils.fileutils.`**CommunicationDevice**(*fd*, *parent=None*)
Specific class for devices for which the readyRead signal is usefull, like for example Unix not regular files and stdin. TcpSocket and UdpSocket do not inherit this class because they inherit specific Qt classes.

readyRead
This signal is emitted once every time new data is available for reading from the device. It will only be emitted again once a new block of data has been appended to your device.

File support

class `boing.utils.fileutils.`**File**(*url*, *mode=IODevice.ReadOnly*, *uncompress=False*, *parent=None*)
`File` instances represent a single file or directory.

class `boing.utils.fileutils.`**CommunicationFile**(*url*, *mode=IODevice.ReadOnly*, *parent=None*)
`CommunicationFile` instances are used to access to Unix non-standard files. The argument *url* defines the path to the file to represent. *mode* can be set to:

- *`IODevice.ReadOnly`*

- *IODevice.WriteOnly*
- *IODevice.ReadWrite*
- *IODevice.Append*

class `boing.utils.fileutils.`**`FileReader`**(*url*, *mode=IODevice.ReadOnly*, *uncompress=False*, *parent=None*)

The *FileReader* can be used to read regular files along the event loop. When the method *start()* is invoked, the *FileReader* will trigger the *readyRead* signal and it will repeat it every time the read method is invoked.

`readyRead`
This signal is emitted once every time new data is available for reading from the device. It will only be emitted again once a new block of data has been appended to your device.

`completed`
Signal emitted when the file has been completed.

`start`()
Start reading the file.

`boing.utils.querypath` — A query path language

The module *boing.utils.querypath* provides *Querypath*, a query language that can be used to handle hierarchical data structures, no matter if they are composed of standard containers, e.g. `list` or `dict`, or instances of standard or custom classes.

The proposed *querypath* language is a derivation of JSONPath, which was proposed by Stefan Goessner to handle JSON structures. Beyond some minor changes, the Boing's query language exploits the fact that the attributes of Python class instances are stored inside the attribute `__dict__`, by actually treating the instances of not Container classes as they were dictionaries.

The root object is indexed using the character `$`, but it can be omitted.

Querypath expressions can use the dot–notation:

```
contacts.0.x
```

or the bracket–notation:

```
['contacts'][0]['x']
```

or a mix of them:

```
contacts[0][x]
```

Querypath allows the wildcard symbol `*` for member names and array indices, the descendant operator `..` and the array slice syntax `[start:end:step]`.

Python expressions can be used as an alternative to explicit names or indices using the syntax `[(<expr>)]`, as for example:

```
contacts[(@.__len__()-1)].x
```

using the symbol `@` for the current object. Also consider that built-in functions and classes are not available. Filter expressions are supported via the syntax `[?(<boolean expr>)]` as in:

```
contacts.*[?(@.x<10)]
```

In order to access to multiple items that have the same parent, it is possible to use the operator `,`, as in:

```
props.width,height
```

while for selecting multiple items that have different parents, it is necessary to combine two *Querypaths* using the operator `|`, as in:

```
props.*|contact..x
```

Note that the `,` structure is normally quicker than the `|` structure, since in the latter case the query always restarts from the root object. Indexing all the values of the data model is possible using the path `..*`.

The module *boing.utils.querypath* provides a set of static functions for executing *Querypath* expression on user data structures. The query expression must be provided as a standard string.

`boing.utils.querypath.`**`get`**`(`*obj*, *path*`)`
Return an iterator over the *obj*'s attributes or items matched by *path*.

`boing.utils.querypath.`**`set_`**`(`*obj*, *path*, *value*, *tocopy=False*`)`
Set the value of *obj* indexed by *path* to *value*. Return *obj* if *tocopy* is `False`, otherwise the copy of *obj* where the modification is applied.

Note: This function must be used carefully since it is supposed to set the requested property to all the matched objects of the structure even if they do not own such property. A common procedure is require to set a specific property only for the objects that already own such property. As an example:

```
>>> tuple(querypath.items(table, "..x"))
(('contacts.0.x', 100), ('contacts.1.x', 500))
>>> querypath.set_(table, "..*[?(@.x)].x",10)
<test.Surface object at 0xa2732ac>
>>> tuple(querypath.items(table, "..x"))
(('contacts.0.x', 10), ('contacts.1.x', 10))
```

Note: The function *set_()* does not accepts the querypaths `"$"` and `""`.

`boing.utils.querypath.`**`paths`**`(`*obj*, *path*`)`
Return an iterator over the paths that index the *obj*'s attributes or items matched by *path*.

`boing.utils.querypath.`**`items`**`(`*obj*, *path*`)`
Return an iterator over the pairs (path, value) of the *obj*'s items that are matched by *path*.

`boing.utils.querypath.`**`test`**`(`*obj*, *path*, *wildcard=NOWILDCARD*`)`
Return whether at least one *obj*'s attributes or items is matched by *path*. The object *wildcard* matches even if *path* does not completely match an item in obj.

`boing.utils.querypath.`**`NOWILDCARD`**
Option specifing that the method *test()* should not consider any wildcard.

Usage examples

```
>>>   class Contact:
...       def __init__(self, x, y):
...           self.x = x
...           self.y = y
...         def polar(self):
...             return math.sqrt(x*x, y*y), math.atan2(y,x)
...         def __repr__(self):
...             return "Contact(%s,%s)"%(self.x, self.y)
...
>>>     class Surface:
...         def __init__(self):
...             self.contacts = []
...             self.props = {}
...
>>> table = Surface()
>>> table.props['width'] = 800
>>> table.props['height'] = 600
>>> table.props['id'] = "mytable"
>>> table.contacts.append(Contact(100,200))
>>> table.contacts.append(Contact(500,600))
>>> tuple(querypath.get(table, "contacts.0.x"))
(100,)
>>> tuple(querypath.get(table, "contacts.*.x"))
(100, 500)
>>> tuple(querypath.get(table, "props.width,height"))
(600, 800)
>>> tuple(querypath.get(table, "..y"))
(200, 600)
>>> tuple(querypath.get(table, "contacts.*[?(@.x<=100)]"))
(Contact(100,200),)
>>> tuple(querypath.get(table, "contacts.*.x,y|props.*"))
(600, 500, 800, 200, 100, 600, "mytable")
>>> querypath.set_(table, "contacts.*.x", 10)
<test.Surface object at 0x8b2606c>
>>> tuple(querypath.get(table, "contacts.*.x"))
(10, 10)
>>> tuple(querypath.paths(table, "props.*"))
('props.height', 'props.width')
>>> tuple(querypath.items(table, "contacts.*"))
(('contacts.1', Contact(100,200)), ('contacts.2', Contact(500,600)))
>>> querypath.test(table, "props.dpi")
False
>>> querypath.test(table, "contacts.*[?(@.x>100)]")
True
>>> querypath.test(table, "props.width.mm")
False
>>> querypath.test(table, "props.width.mm", wildcard=800)
True
```

The `QPath` class

Since *Querypath* strings must be pre-processed in order to be executed, supposing you are going to use the same query multiple times, it may be better to create a `QPath` instance, and then use the member methods, instead of the `boing.utils.querypath` static functions. The proposed functuality is equal, but the string does not have to be pre-processed for all the executions.

class `boing.utils.querypath.`**`QPath`**(*path*)

A compiled *Querypath* expression.

`get`(*obj*)

Return an iterator over the *obj*'s attributes or items matched by this QPath.

set (*obj*, *value*, *tocopy=False*)
Set the value of *obj* indexed by this QPath to *value*. Return *obj* if *tocopy* is `False`, otherwise the copy of *obj* where the modification is applied.

paths (*obj*)
Return an iterator over the paths that index the *obj*'s attributes or items matched by this QPath.

items (*obj*)
Return an iterator over the pairs (path, value) of the *obj*'s items that are matched by this QPath.

test (*obj*, *wildcard=NOWILDCARD*)
Return whether this QPath matches at least one *obj*'s attributes or items. The object *wildcard* matches even if *path* does not completely match an item in obj.

Usage example:

```
>>> query = querypath.QPath("contacts.*.x")
>>> tuple(query.get(table))
(100, 500)
>>> query.set(table, 10)
<test.Surface object at 0xa2732ac>
>>> tuple(query.paths(table))
('contacts.0.x', 'contacts.1.x')
>>> tuple(query.items(table))
(('contacts.0.x', 10), ('contacts.1.x', 10))
>>> query.test(table)
True
```

`QPath` instances can be combined using the + operator. This operation concatenates the operand strings using the `|` delimiter, but it also tries to optimize the result by avoiding expression duplicates, as in:

```
>>> querypath.QPath("props")+querypath.QPath("contacts")
QPath('contacts|props')
>>> querypath.QPath("props")+querypath.QPath("props")
QPath('props')
```

Still it cannot optimize more complex overlaps:

```
>>> querypath.QPath("contacts[0]")+querypath.QPath("contacts.*")
QPath('contacts[0]|contacts.*')
```

`boing.utils.url` — Uniform Resource Locator

The module *`boing.utils.url`* mainly provides the class *`URL`*, which is used to represent a Uniform Resource Locator.

Examples:

```
>>> from boing.utils.url import URL
>>> url = URL("ftp://paolo:pwd@localhost:8888/temp?key=value#frag")
>>> url.scheme
'ftp'
>>> url.site.user
'paolo'
>>> url.site.password
'pwd'
>>> url.site.host
'localhost'
```

```
>>> url.site.port
8888
>>> str(url.path)
'/temp'
>>> url.query['key']
'value'
>>> url.fragment
'frag'
```

The class `URL`

class `boing.utils.url.`**`URL`**(*string*)

An instance of the class *`URL`* represents an Uniform Resource Locator (URL). The attribute *string* of the class constuctor defines the URL that the instance will represent; at the instance initialization, *string* is parsed in order to detect the kind of URL and to separate it into the specific components: *schema*, *site*, *path*, *query*, *fragment*.

Warning: To address a Windows drive it is **necessary** to put an additional `/` (slash punctuation) before the drive letter. Example:

```
>>> URL("file:///C:/Users/")
```

Each instance owns the following read-only attributes:

`kind`

Kind of URL. It equals to one of the following:

- `URL.EMPTY` — empty URL
- `URL.OPAQUE` — URL like `<scheme>:<opaque>`
- `URL.GENERIC` — URL like `<scheme>://<site>/<path>?<query>#<fragment>`
- `URL.NETPATH` — URL like `//<site>/<path>?<query>#<fragment>`
- `URL.ABSPATH` — URL like `/<path>?<query>#<fragment>`
- `URL.RELPATH` — URL like `<path>?<query>#<fragment>`

`scheme`

URL scheme defined by a `str`.

`site`

URL site defined by an instance of the class *`URL_site`*.

`path`

URL path defined by an instance of the class *`URL_path`*.

`query`

URL query defined by an instance of the class *`URL_query`*.

`fragment`

URL fragment defined by a `str`.

`opaque`

if the URL is of kind `URL.OPAQUE` it defines the right part of the URL; otherwise it is set by default to the empty string `""`.

The string representation of an *URL* instance is normally equal to the string passed at the instance initialization, but there are few exceptions:

```
>>> str(URL("udp://:3333"))
'udp://:3333'
>>> str(URL("udp://:3333:0"))
'udp://:3333'
>>> str(URL("file:/tmp/log"))
'file:///tmp/log'
```

URL instances are equal if their string representation is the same:

```
>>> URL("udp://:3333")==URL("udp://:3333")
True
>>> URL("udp://:3333:0")==URL("udp://:3333")
True
```

URL instances can be compared to `str` objects:

```
>>> URL("udp://:3333")=="udp://:3333"
True
```

and they can be concatenated as they were `str` objects:

```
>>> url = URL("udp://:3333")
>>> "osc."+url
'osc.udp://:3333'
>>> url+"#frag"
'udp://:3333#frag'
```

Note that the result is a `str`, not an *URL* instance.

URL internal classes

class `boing.utils.url.`**`URL_site`**(*string*)

Used to store the component *site* of an URL. Each instance owns the following attributes:

`user`
User defined by a string.

`password`
Password defined by a string. It is NOT encripted.

`host`
Site host defined by a string.

`port`
Port number defined by an integer. It defaults to `0`.

Usage example:

```
>>> url = URL("ftp://paolo:pwd@localhost:8888")
>>> url.site
URL_site('paolo:pwd@localhost:8888')
>>> print(url.site)
paolo:pwd@localhost:8888
>>> url.site.user
'paolo'
```

```
>>> url.site.password
'pwd'
>>> url.site.host
'localhost'
>>> url.site.port
8888
```

Instances can be compared to `str` objects:

```
>>> url = URL("udp://localhost:3333")
>>> url.site=="localhost:3333"
True
```

and have Boolean value to `True` if anyone of the component attributes is defined:

```
>>> bool(URL("udp://localhost:3333").site)
True
>>> bool(URL("udp://").site)
False
```

Warning: Pay attention to the default case:

```
>>> bool(URL("udp://:0").site)
False
```

class `boing.utils.url.`**`URL_path`**(*string*)

Used to store the component *path* of an URL. Usage example:

```
>>> url = URL("file:///tmp/log")
>>> url.path
URL_path('/tmp/log')
>>> print(url.path)
/tmp/log
>>> url.path.isAbsolute()
True
```

`isAbsolute`()

Return wheter the path is absolute:

```
>>> URL("file:///tmp/log").path.isAbsolute()
True
>>> URL("/tmp/log").path.isAbsolute()
True
>>> URL("file").path.isAbsolute()
False
>>> URL("./file").path.isAbsolute()
False
```

Instances can be compared to `str` objects:

```
>>> url = URL("file:///tmp/log")
>>> url.path=="/tmp/log"
True
```

and have Boolean value to `True` if the URL path is not empty:

```
>>> bool(URL("file:///tmp/log").path)
True
>>> bool(URL("/").path)
True
>>> bool(URL("udp://:8888").path)
False
```

Warning: Pay attention to the default transformation:

```
>>> str(URL("file:/tmp/log"))
'file:///tmp/log'
```

class `boing.utils.url.`**`URL_query`**(*string*)

Used to store the component *query* of an URL. This class implements the `collections.MutableMapping` *ABC*. It is also able to encode the URL's *query* into a "percent-encoded" string.

Examples:

```
>>> url = URL("udp://:8888?name=Jérémie&connect")
>>> url
URL('udp://:8888?name=J%e9r%e9mie&connect')
>>> url.query
URL_query('name=J%e9r%e9mie&connect')
>>> url.query['name']
'Jérémie'
>>> dict(url.query)
{'name': 'Jérémie', 'connect': ''}
>>> URL("udp://:8888?name=Jérémie&connect")==URL("udp://:8888?name=J%e9r%e9mie&
↪connect")
True
```

It is possible to use the structure `<!...!>` in order to assign to the query key or value a string that contains the characters `&` or `=`, which is normally not permitted since it would break the URL syntax.

Example:

```
>>> url = URL("edit:?<!..contacts.*[?(@.id>=2)]!>=<!True&False!>&k2=v2")
>>> dict(url.query)
{'k2': 'v2', '..contacts.*[?(@.id>=2)]': 'True&False'}
```

Warning: The characters `#` and `?` cannot be used inside the `<!...!>` structure.

Getting help

The easiest way to get help with the project is to join the mailing list boing@librelist.com.

The archive of the mailing list can be checked out at http://librelist.com/browser/boing/ .

The other good way is to open an issue on github.

ChangeLog

v0.3.1, Sep, 18 2012 — Fixed and upgraded the installation guide.

- Added attribute `boing.config` for storing global configuration variables.
- Added argument `--no-gui` for running Boing without a display server.
- `.ui` files are automatically compiled when the package is installed.
- `boing.VERSION` changed into `boing.__version__`
- The installation guide has been updated.
- Added documentation and unit test for module *`boing.utils.url`*.

v0.3.0, Sep, 10 2012 – First public release.

v0.2.0, Apr, 1 2012 – Architetecture wide modifications.

v0.1.0, Jan, 1 2012 – Pre-release.

Authors

- Paolo Olivo (MSc, Milano Bicocca, 2009) is a contract engineer at INRIA Lille. His interests include software engineering and development for tactile and gestural interaction. The present focus of his work is on programming tools for supporting research in Human-Computer Interaction.
- Nicolas Roussel (PhD/Hab, Paris-Sud, 2000/2007) is a senior researcher at INRIA Lille. His research lies in the field of Human-Computer Interaction with current primary interests in the design, implementation and evaluation of simple yet powerful tactile and gestural interactions. Other interests include engineering of interactive systems, window systems, computer-mediated communication and groupware.

License

Todo list

Toolkit todo

- *Bugs and unittest*:
  - Class `DataWriter` should signal an error when it is demanded to write to a closed device.
  - Add `boing.test.nodes.test_loader` cases for the data processing nodes.
  - Add docs and unit test for class *`boing.utils.url.URL`* on MS Windows.
  - Improve class `QPath`: regular expression compilation, join method, add unittest. The command:

    ```
    QPath.filter(boing.nodes.encoding.TuioDecoder.getTemplate(), "diff.*.contacts.
    ↪*.rel_pos")
    ```

    raises an error if `QPath._iterProp()` returns a real iterator.
  - when `boing.create()` raises an exception, it shows the lower URI and not the original one. This may be misleading for users.
  - The `player:` ‘s playlist has some model trouble: when I drag and drop some files from a folder to the root level before the folder an Exception is raised. Sometimes files desappears.
  - Handle when a source has been closed and when to start players (e.g. if TCP socked has been disconnected, TcpServer turned off).
  - Resolve the UDP socket reuse port issue on Windows.

  - The structure `<!...!>` used in defining not standard *URL* query keys and values does not work if characters `#` or `%` are used inside the structure.

- *Pipeline architecture*:
  - The class *Producer* should also automatically know whether being active or not, like the class *WiseWorker* does. Check the 'tag' structure.
  - The class `Node` shouldn't be a QObject?
  - Improve Graphers (Graphers should draw themselves).
  - Add exclusive requests in order to optimize productivity.
- *Data model*:
  - json and pickle decoders should someway know what they produce.
  - Check the `quickdict` constructor: if an hierarchical dictionary is passed to the constructor not all the hierarchy is transformed to a quickdict.
- *Functionalities*:
  - Encoder and Decoders in module boing.nodes.encoding should inherit boing.nodes.Encoder and boing.node.Decoder.
  - Find a way so that the boing.node.loader can create nodes from external source files, so that users can add custom nodes.
  - Develop the transformation node, which transforms the data hierarchy (JSON-schema validator).
  - Develop `evdev` and `uinput` in&out bridges.
  - Enable remote node.
  - Improve Contact Viz.
  - Consider adding the module `libfilter.filtering.signal` to *boing.filtering*.
  - Develop lib tIO cython bindings.
  - When Qt4.8 will be available, add multicast support to UdpSocket.
- *Gesture Recognition*:
  - Prepare the directory with the gesture templates that the recognizer can use.
  - Fix the recognition nodes.
  - Support 1$ algorithm.
- *Docs*:
  - Check which Ubuntu packages are really necessary.
  - Improve docs for modules *boing.net.tcp* and *boing.net.udp*.
- *Other*:
  - Module *boing.utils.fileutils* should be reengineered.

Docs todo

Todo

Describe how to configure the pipeline dinamically

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/boing/checkouts/latest/doc/source/boing.rst, line 101.)

Todo

Comment module `boing.filtering`.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/boing/checkouts/latest/doc/source/boing.filtering.rst, line 29.)

Todo

Improve docs for the module boing.net.tcp

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/boing/checkouts/latest/doc/source/boing.net.tcp.rst, line 24.)

Todo

Improve docs for the module boing.net.udp

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/boing/checkouts/latest/doc/source/boing.net.udp.rst, line 24.)

Todo

Describe the data model.

(The *original entry* is located in /home/docs/checkouts/readthedocs.org/user_builds/boing/checkouts/latest/doc/source/datamodel.rst, line 24.)

Todo

Describe an example of functional node.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/boing/checkouts/latest/doc/source/developer.rst, line 57.)

Todo

Speak about the default nodes and the node uris convention. Also add a link to the *Nodes reference table*.

(The *original entry* is located in /home/docs/checkouts/readthedocs.org/user_builds/boing/checkouts/latest/doc/source/functionalities.rst line 24.)

Todo

Describe data logging and replaying (both OSC and JSON formats).

(The *original entry* is located in /home/docs/checkouts/readthedocs.org/user_builds/boing/checkouts/latest/doc/source/functionalities.rst, line 46.)

Todo

Describe the recorder tool.

(The *original entry* is located in /home/docs/checkouts/readthedocs.org/user_builds/boing/checkouts/latest/doc/source/functionalities.rst, line 52.)

Todo

Describe the Player tool.

(The *original entry* is located in /home/docs/checkouts/readthedocs.org/user_builds/boing/checkouts/latest/doc/source/functionalities.rst, line 65.)

Todo

Describe the calibration node.

(The *original entry* is located in /home/docs/checkouts/readthedocs.org/user_builds/boing/checkouts/latest/doc/source/functionalities.rst, line 81.)

Todo

Describe the filtering module.

(The *original entry* is located in /home/docs/checkouts/readthedocs.org/user_builds/boing/checkouts/latest/doc/source/functionalities.rst, line 87.)

Todo

Logging utils tutorial

(The *original entry* is located in /home/docs/checkouts/readthedocs.org/user_builds/boing/checkouts/latest/doc/source/logging.rst, line 24.)

Todo

multi-touch utilities tutorial.

(The *original entry* is located in /home/docs/checkouts/readthedocs.org/user_builds/boing/checkouts/latest/doc/source/multitouch.rst, line 24.)

Todo

Improve the OSC tutorial.

(The *original entry* is located in /home/docs/checkouts/readthedocs.org/user_builds/boing/checkouts/latest/doc/source/osc.rst, line 24.)

Todo

- Describe the composite nodes and node syntax (+ and | operators).

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/boing/checkouts/latest/doc/source/pipeline.rst, line 406.)

Todo

Data redirection tutorial

(The *original entry* is located in /home/docs/checkouts/readthedocs.org/user_builds/boing/checkouts/latest/doc/source/redirection.rst, line 24.)

Todo

Script advanced options tutorial

(The *original entry* is located in /home/docs/checkouts/readthedocs.org/user_builds/boing/checkouts/latest/doc/source/scriptadvanced.rs line 24.)

Todo

Improve the TUIO tutorial.

(The *original entry* is located in /home/docs/checkouts/readthedocs.org/user_builds/boing/checkouts/latest/doc/source/tuio.rst, line 24.)

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index