zircon Documentation

Release 0.1.0

Hayk Martirosyan

Sep 27, 2017

Contents

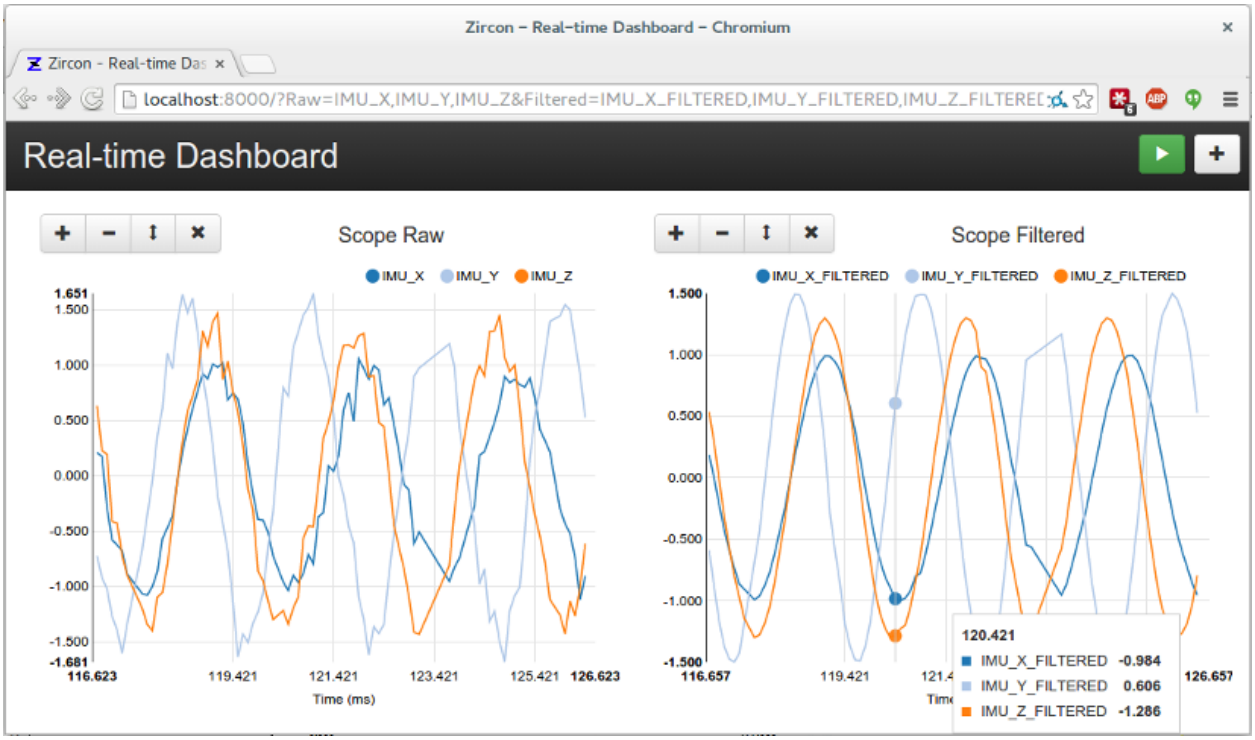
Zircon is a lightweight framework to intercept, store, analyze, and visualize high-speed signals in real-time using modern technologies. High-speed means thousands of messages per second and real-time means a latency of milliseconds to tens of milliseconds.

CHAPTER 1

Table of Contents

Introduction to Zircon

Zircon is a lightweight framework to intercept, store, analyze, and visualize high-speed signals in real-time using modern technologies. High-speed means thousands of messages per second and real-time means a latency of milliseconds to tens of milliseconds.



Purpose

Zircon is designed to capture rapid streams of data in large communication networks like CAN busses and wireless meshes. This usually means sensor and actuator signals, but can be anything that boils down to events or time series. Zircon consists of pluggable components useful individually or as a full stack for decoding and logging, structured querying and analysis, real-time visualization, and integration into custom applications. Written in Python, with a small and elegant code base.

Zircon is free and open-source, fast, platform-independent, fully extensible, and easy to integrate with your system. By default, InfluxDB is used for blazing-fast storage, ZeroMQ for local or remote reporting, and Socket.IO for real-time data streaming. It provides base classes and examples that can be extended to support any protocol or encoding.

Application Example

Steve is the lead engineer for a new hovercraft, which contains ECUs for propulsion, steering, sensing, power, and safety systems on a CAN bus. Every day, Steve's engineers test and debug their respective systems through the messages they send and receive. The team has various expensive, bulky, archaic, and/or proprietary software options to filter and log messages, but they still spend a lot of time watching the stream and manually decoding bytes. Or worse, they pull raw data from log files and hunt for something meaningful.

Steve installs Zircon on his laptop, writes a little code, and fires it up. What he instantly gets is a slick web interface to visualize all communications inside his hovercraft, in real-time. What he also gets is a powerful API to query, filter, downsample, aggregate, and export all signals for the past days or weeks. Excited, Steve sets up Zircon on a dedicated machine. His engineers start using the Zircon interface to wirelessly debug their firmware, tune the control gains, and check for voltage spikes. Soon, they create custom diagnostic dashboards, an interactive driver display, and a mobile app that can start the hovercraft by sending messages back to the bus. Productivity soars, and profit is made.

Getting Started

Zircon is a flexible tool that adapts well to individual use cases. Before you start using it, look through the *System Overview* and get a sense of what the component classes do and how they are connected together to make a complete data pipeline from source to database to client application.

Zircon's built-in components will fit many needs out of the box, but with customization can be adapted for nearly any scenario. The default datastore (InfluxDB), messaging protocol (ZeroMQ), and client-side API (Socket.IO) used by Zircon's components provide excellent speed and robustness.

Make sure to have a picture of what the following components should be for your application:

- Transceivers (interfaces to your sources of data)
- Transformers (tasks - batching, formatting, [de]compression, [de]serialization, etc.)
- Client (end goal - just visualization, or integration into some system?)

The guide below outlines how to install Zircon and use it to store some programatically-generated timeseries data at 2kHz and visualize it in real-time with the web dashboard. From there, it is a simple matter to create a Transceiver for your own data source, run complex queries, and build custom applications.

Installation

Note: Instructions are for Debian-based distributions. Many commands require root privileges.

1. Clone the repository.

```
git clone https://github.com/hmartiro/zircon.git
```

2. Install required Python packages using `pip`. I highly recommended using `virtualenv` and `virtualenvwrapper` to isolate the Python environment.

```
cd zircon
pip install -r requirements.txt
```

3. Install and start InfluxDB, the default datastore. It should start automatically from now on.

```
service influxdb start
```

4. Install `bower` for managing the frontend JavaScript libraries.

```
apt-get install nodejs npm
ln -s /usr/bin/nodejs /usr/bin/node # Quirk on ubuntu, node package was already taken
npm -g install bower
```

5. Configure the `PYTHONPATH` and `DJANGO_SETTINGS_MODULE` environment variables. If using a `virtualenv`, do this in your `activate` or `postactivate` script. Otherwise, source it from your `~/.bashrc` or similar.

```
PYTHONPATH=$PYTHONPATH:/path/to/zircon
DJANGO_SETTINGS_MODULE=zircon.frontend.settings
```

Note: You should now be able to access Zircon from Python. Make sure `import zircon` works. Also, `python -m zircon.frontend.manage` is the entry point to Zircon's built-in Django application. You can interface with it on the command line.

---

6. Initialize the real-time web dashboard

```
python -m zircon.frontend.manage syncdb
python -m zircon.frontend.manage bower install
```

That's it for installation. You are ready to dive into Zircon!

Reporter Process

Copy the file `zircon/tests/sample_reporter.py` and open it up. This script initializes and runs a *Reporter*. A Reporter is a class that collects data from a *Transceiver*, processes it using a chain of *Transformers*, and broadcasts the processed data using a *Publisher*.

To create a Reporter, we simply initialize it with a Transceiver, a list of Transformers, and a Publisher. Then, we call `run()`.

```
reporter = Reporter(
    transceiver=..,
    transformers=[.., .., ..],
    publisher=..
)
reporter.run()
```

For this demo, we are using a `DummyTransceiver`, which generates a single signal by sampling a given function. In our case, it invokes `sine_wave(t)` at the specified frequency of 1 kHz. We name this signal 'MY_SIGNAL'. The output of the Transceiver is a tuple of the form (timestamp, name, value).

```
transceiver=DummyTransceiver(
    signal_name='MY_SIGNAL',
    data_gen=sine_wave,
    dt=1.0/freq
)
```

Next, we specify three Transformers. The return value of the transceiver's `read` method is fed into each Transformer's `push` method, in a chain.

The first is a *`TimedCombiner`* , which batches up the signals for more efficient transmission and database insertion. It reads in all messages, and outputs them chunked up into a list on a given interval. Every individual point at 1kHz is saved, but we save each set of 100 points as a group at a rate of 10 Hz. Batching is not necessary, but it dramatically raises the ceiling on achievable throughput. By default, the web dashboard downsamples the data to 10 Hz, so there is no reason to transmit or insert at a faster rate. You can tweak this based on your needs - if you need 20ms of latency, set the dt of TimedCombiner to 0.02.

The next Transformers are a *`Pickler`* and a *`Compressor`*. The Pickler serializes the output of the TimedCombiner using Python's `pickle`, and the Compressor uses `zlib` to shrink the message and save on network bandwidth. If processing power is your bottleneck rather than network bandwidth, you can skip the Compressor. These classes are essentially one-liners, but having them as a Transformer interface makes them awesome to plug and play. You can use any method of serialization, but the output of the last Transformer must be a bufferable object (a string, usually).

```
transformers=[
    TimedCombiner(dt=0.1),
    Pickler(),
    Compressor()
],
```

Finally, the serialized data is broadcast by a Publisher to any processes that want to listen. Zircon's default Publisher is the `ZMQPublisher`, which writes the data to any subscribed entities, local or remote, using the ZeroMQ messaging protocol.

Okay, enough talk. Are you ready to run the reporter process? Here it is:

```
python sample_reporter.py
```

When you start it, you might see some output from the Publisher of how many messages it is sending. Nothing else is happening, because nobody is listening yet. We need to start another process, an *Injector*.

Injector Process

Copy the file `zircon/tests/sample_injector.py` and take a look at it. Like a Reporter, an Injector has three components.

First, a Subscriber receives serialized messages from a Publisher. The Subscriber should specify the network address of the Publisher (localhost is default). The default subscriber is the `ZMQSubscriber`, which connects to the ZMQPublisher.

Next, a series of Transformers are applied to the data, just like with the Reporter. Here, we simply use a Decompressor to reverse the Compressor, and an Unpickler to reverse the Pickler. The output of the Unpickler is a list of (timestamp, name, value) tuples as outputted by the TimedCombiner.

```
injector = Injector(
    subscriber=ZMQSubscriber(),
    transformers=[
        Decompressor(),
        Unpickler(),
    ],
    datastore=InfluxDatastore()
)
injector.run()
```

Finally, our data is fed to a *Datastore*. Datastores implement methods to insert and query signal data. Each message outputted by the Unpickler is fed into the `insert` method of the Datastore.

Zircon's built-in Datastore is the `InfluxDatastore`, which uses InfluxDB, an extremely fast timeseries database with powerful query capabilities. Our client applications interface with the Datatstore.

Now, start the Injector:

```
python sample_injector.py
```

Start the Reporter up as well, and the Injector should output that it is saving around 90-95 points every 0.1 seconds, whatever the DummyTransceiver actually outputs at. Your signal is now being saved into an InfluxDB database instance. If you like, you can explore it directly using InfluxDB's web UI. However, we will be focusing on Zircon's dashboard.

Real-time Dashboard

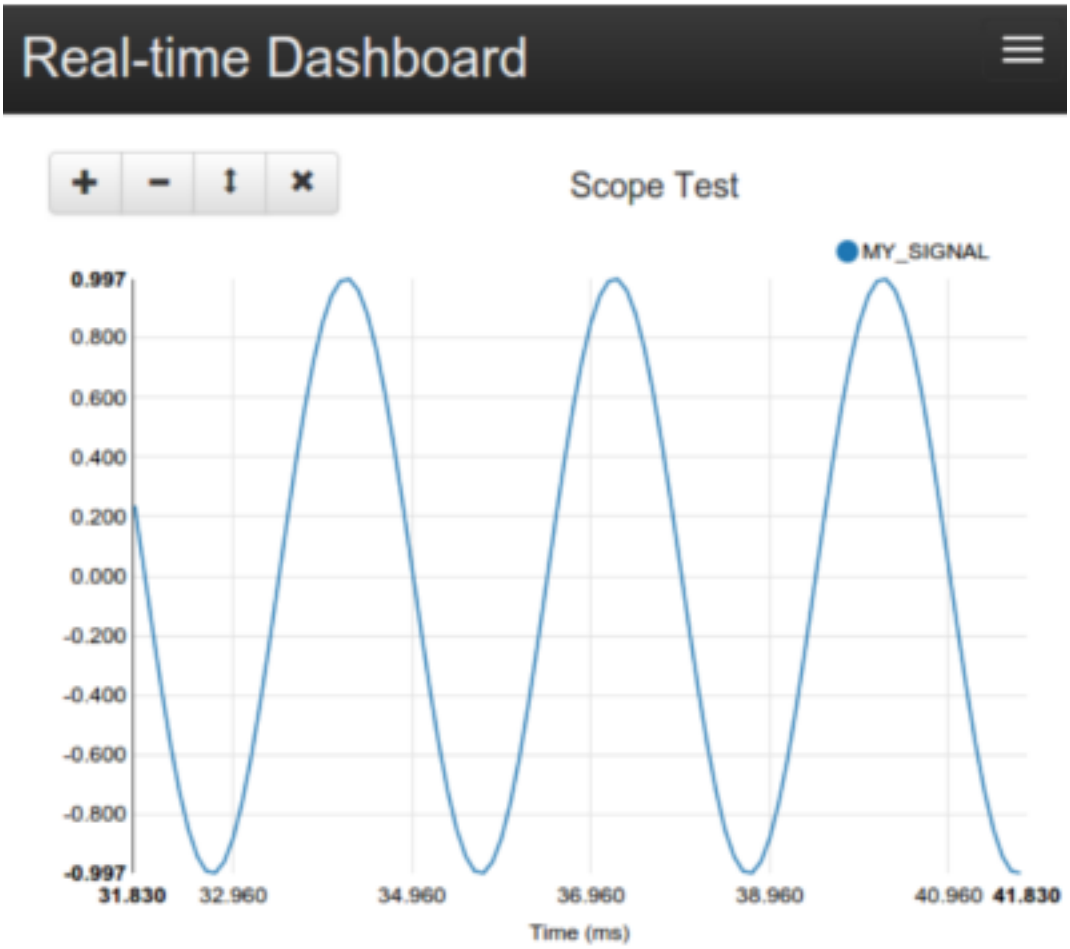
With your Reporter and Injector processes running, your sine wave is flowing into the Datastore. We can visualize this signal using Zircon's web interface. The web interface provides a general purpose solution for viewing signal data, and more importantly acts as an example for how to build custom applications using Zircon.

Start the web interface using Django:

```
python -m zircon.frontend.manage runserver_socketio 0.0.0.0:8000
```

Navigate to `http://localhost:8000/`. You should see a blank page with a header bar. Click the '+' button in the upper-right corner to add a scope, and name it 'my_scope'. Now, click the '+' in the scope to add a signal, 'MY_SIGNAL'.

You should now see your glorious sine wave! It is being dynamically sampled at 100ms by default, from the Datastore. You can play around with the plot controls, mouse-over to see the values, and pause/play the scope from the top bar. Note, the scope configuration is encoded in the URL, so you can copy and paste it to save configurations.



There are two Django apps running here - the `datasocket` and the `dashboard`. The `datasocket` provides a Socket.IO API directly to the Datastore. The `dashboard` acts as a client to the `datasocket`. The client-side JavaScript in `dashboard` opens up a connection to the Socket.IO API and requests data for the signals the user has selected.

You can browse the code at `zircon/zircon/frontend`.

Onward

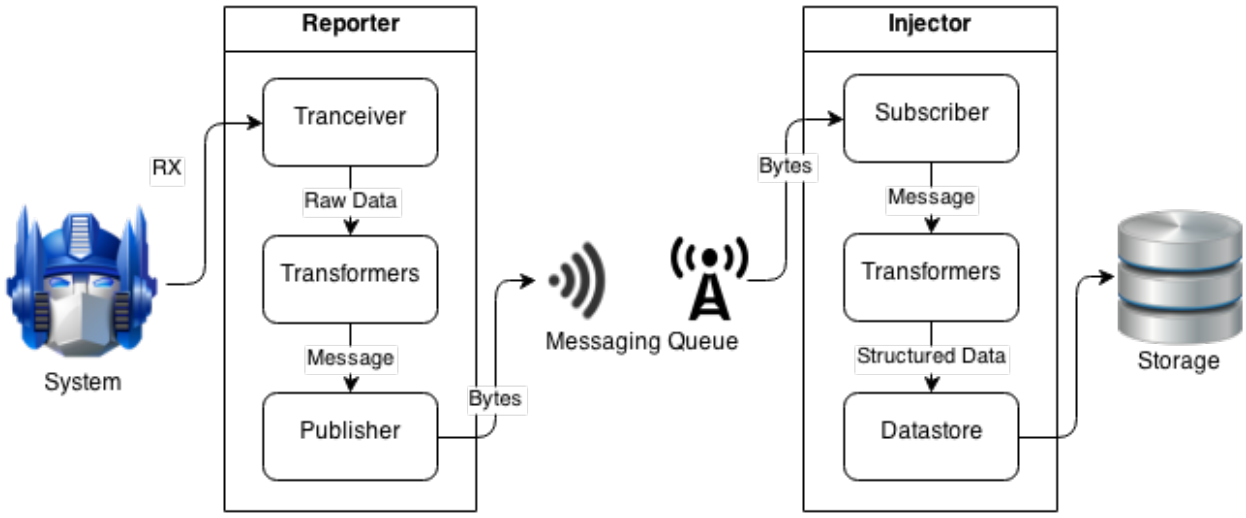
Hopefully, you now understand how the Zircon stack works! Here are some things you can try:

- Run your Reporter and Injector processes on different machines. Just specify the IP address to the ZMQSubscriber, like `ZMQSubscriber(host='192.168.1.45')`.
- Query the Datastore programatically, just by initializing an instance of `InfluxDatastore`. You can take a look at `zircon/zircon/utils/export.py` for an example. You can query for something like the mean of a signal value, for a given hour, in 1 second buckets.
- Monitor six signals at once, using `zircon/tests/noisy_imu_reporter.py`.
- Take a look at some common Transformers in `zircon/zircon/transformers/common.py`.
- See how high of a throughput you can get, or how low of a latency. You can play around with the dashboard code at `zircon/zircon/frontend/dashboard/static/dashboard/js/dash.js`. In particular, play with the `data_frametime` and `view_frametime` variables.

Create a Transceiver that reads from your sensor! Just extend BaseTransceiver and implement the `read()` method. Zircon can be useful for anything from Arduino signals to events in a distributed network.

Overview of Components

Zircon consists of pluggable components useful individually or as a full stack for decoding and logging, structured querying and analysis, real-time visualization, and integration into custom applications. Components are connected together to achieve a flexible data pipeline in centralized and distributed scenarios.



Transceivers A Transceiver reads and/or writes to some source. It is the lowest-level component in zircon, which would for example interface with a CAN bus, serial port, or XBee.

Transformers A Transformer takes in messages, applies some transformation, and spits them back out. It is a general piece of middleware in a data stream that can be used to compress/decompress, encode/decode, or split/combine messages in a data pipeline.

Publishers A Publisher broadcasts data in some form, to be picked up by one or more Subscribers. It is used by Reporters to communicate with Injectors.

Subscribers A Subscriber receives data from a Publisher. It is used by Injectors to listen to Reporters.

Datastores A Datastore is a connector to something that can store timeseries data. It provides an interface to add, remove, and access timeseries data efficiently. A single piece of information consists of a signal name, a timestamp in microseconds, and some associated data.

Reporter A Reporter continuously reads data from a Transceiver, feeds it through a row of Transformers, and broadcasts the result using a Publisher.

Injector An Injector listens for data from a Reporter, feeds it through a row of Transformers, and inserts the result into a Datastore.

Server Zircon's backend server provides a Socket.IO interface to query information from a Datastore. It allows real-time bidirectional event-based communication between a client application that receives data and the Datastore. Works for web or native applications, on any platform. Allows powerful querying of every recorded data point, by default for the past week.

Client A client is anyone who wants to access the data coming from Transceivers. It is the end goal of Zircon to provide a fast, robust, and easy way for clients to monitor the signals they are interested in. Clients can be web apps, mobile apps, native apps, or hardware systems. Zircon's default client is a web dashboard that allows

real-time visualization of arbitrary signals. It is a Django application with a JavaScript interface to the Socket.IO connection.

Transceivers

Transceiver Interface

class zircon.transceivers.base.**BaseTransceiver**
Abstract base class defining the Transceiver interface.

A Transceiver reads and/or writes to some source. It is the lowest-level component in zircon, which would for example interface with a CAN bus, serial port, or XBee.

Usage:

```
t = MyTransceiver()
t.open()

while not done:
    data = t.read()
    process(data)

t.close()
```

open()
Open the connection.

close()
Close the connection.

read()
Return data read from the connection, or None.

write(*data*)
Write data to the connection.

Transformers

Transformer Interface

class zircon.transformers.base.**BaseTransformer**
Abstract base class defining the Transformer interface.

A Transformer takes in messages, applies some transformation, and spits them back out. It is a general piece of middleware in a data stream that can be used to compress/decompress, encode/decode, or split/combine messages in a data pipeline.

Usage:

```
def process(msg):
    do_something(msg)

t = MyTransformer()
t.set_callback(process)
```

```
for msg in messages:
    t.push(msg)
```

set_callback(*callback*)
Set a function to be invoked for each outputted message.

push(*msg*)
Feed in a message.

Pass-through Transformer

class zircon.transformers.base.**Transformer**
Bases: *zircon.transformers.base.BaseTransformer*

Transformer that acts as a pass-through, invoking the callback for each message received with no alterations.

Extend this and override `push()` to implement a Transformer.

set_callback(*callback*)

output(*msg*)
If I have a callback, invoke it.

push(*msg*)
Output exactly what I receive.

Common Transformers

class zircon.transformers.common.**Combiner**(*limit*)
Bases: *zircon.transformers.base.Transformer*

Combine messages into a list, then send them out.

__init__(*limit*)

Parameters **limit** – How many messages to combine.

push(*msg*)

class zircon.transformers.common.**Doubler**
Bases: *zircon.transformers.base.Transformer*

Output each message twice.

push(*msg*)

class zircon.transformers.common.**Splitter**
Bases: *zircon.transformers.base.Transformer*

Split messages into parts by iterating through them.

push(*msg*)

class zircon.transformers.common.**Uppercaser**
Bases: *zircon.transformers.base.Transformer*

Capitalize string-like messages.

push(*msg*)

class zircon.transformers.common.**Lowercaser**
Bases: *zircon.transformers.base.Transformer*

Lowercase string-like messages.

push(*msg*)

class zircon.transformers.common.**Pickler**
Bases: *zircon.transformers.base.Transformer*

Pickle messages with the latest protocol.

push(*msg*)

class zircon.transformers.common.**Unpickler**
Bases: *zircon.transformers.base.Transformer*

Unpickle messages with the latest protocol.

push(*msg*)

class zircon.transformers.common.**Compressor**
Bases: *zircon.transformers.base.Transformer*

Compress messages using zlib.

push(*msg*)

class zircon.transformers.common.**Decompressor**
Bases: *zircon.transformers.base.Transformer*

Decompress messages using zlib.

push(*compressed_msg*)

class zircon.transformers.common.**TimedCombiner**(*dt=0.1*)
Bases: *zircon.transformers.base.Transformer*

Convert individual data points into a dictionary of signal names to time series, outputted at a regular interval.

Input: (12345, 'MYSIGNAL', -5.2), (12346, 'MYSIGNAL', 1.3), ... Output: {'MYSIGNAL': ((12345, -5.2), (12346, 1.3))}

__init__(*dt=0.1*)

push(*msg*)

class zircon.transformers.common.**Printer**(*prefix=None*)
Bases: *zircon.transformers.base.Transformer*

Prints messages and passes them on unaltered.

__init__(*prefix=None*)

push(*msg*)

class zircon.transformers.common.**Timer**
Bases: *zircon.transformers.base.Transformer*

Prints the time between messages, and passes them on unaltered.

__init__()

push(*msg*)

Publishers and Subscribers

Publisher Interface

class zircon.publishers.base.**BasePublisher**

Abstract base class defining the Publisher interface.

A Publisher broadcasts data in some form, to be picked up by one or more Subscribers. It is used by Reporters to communicate with Injectors.

Usage:

```
p = MyPublisher()
p.open()

while not done:
    msg = get_data()
    p.send(msg)

p.close()
```

open()

Open the connection.

close()

Close the connection.

send(*msg*)

Broadcast a message.

Subscriber Interface

class zircon.subscribers.base.**BaseSubscriber**

Abstract base class defining the Subscriber interface.

A Subscriber receives data from a Publisher. It is used by Injectors to listen to Reporters.

Usage:

```
s = MySubscriber()
s.open()

while not done:
    msg = s.receive()
    process(msg)

p.close()
```

open()

Open the connection.

close()

Close the connection.

receive()

Receive a message.

Datastores

Datastore Interface

class zircon.datastores.base.**BaseDatastore**

Abstract base class defining the Datastore interface.

A Datastore is a connector to something that can store timeseries data. It provides an interface to add, remove, and access timeseries data efficiently. A single piece of information consists of a signal name, a timestamp in microseconds, and some associated data.

To be efficient, a Datastore should keep information sorted by timestamp and separated by signal name. The most important ingredient is that the most recent N points for a given signal can be retrieved in constant time.

What kind of data can be stored depends on the implementation. For example, a Datastore may accept integers, floats, strings, or any combination of them.

create_database(*db_name*)

Create a database.

Returns True if successful, False otherwise.

delete_database(*db_name*)

Delete a database.

Returns True if successful, False otherwise.

switch_database(*db_name*)

Switch the current database.

Returns True if successful, False otherwise.

list_databases()

Return a list of databases.

list_signals()

Return a list of signals in this database.

```
>>> datastore.list_signals()
['SIGNAL_A', 'SIGNAL_B', 'SIGNAL_C']
```

delete_signal(*data*)

Delete this signal and all associated data.

Returns True if successful, False otherwise.

insert(*data*)

Insert data.

Parameters **data** – Dictionary mapping signal names to timeseries.

Returns True if successful, False otherwise.

Timeseries consist of an epoch timestamp in microseconds followed by some data.

```
>>> datastore.insert({
... 'SIGNAL_A': (
...                 (1409481110001000, 1.2),
...                 (1409481110002000, 1.5)
...             ),
... 'SIGNAL_B': (
...                 (1409481110001500, -2.1)
```

```
...                 )
... })
True
```

get_last_points(*signals*, *num*)
Return the last N points for the given signals.

Parameters

- **signals** – A list of signals.
- **num** – The number of points to fetch.

Returns A dictionary mapping signals to points.

```
>>> signal = datastore.get_last_points(['SIGNAL_A'], 10)
{'SIGNAL_A': [[1409481110001000, 1.2], [1409481110002000, 1.5], ...]}
```

get_timeseries(*signals*, *t0*, *t1*, *dt*, *aggregate*, *limit*)
Return a uniformly sampled time series in a given time interval. Can downsample, aggregate, and limit the result.

Aggregate functions depend on the implementation, but should at least include 'mean', 'first', 'last', 'min', and 'max'.

Parameters

- **signals** – A list of signals.
- **t0** – Start time in microseconds.
- **t1** – End time in microseconds.
- **dt** – Sample time in microseconds
- **aggregate** – Aggregate function to apply.
- **limit** – Maximum number of points per signal to return.

Returns A dictionary mapping signals to points.

Reporters and Injectors

Reporter

class zircon.reporters.base.**Reporter**(*transceiver*, *transformers=None*, *publisher=None*)
A Reporter continuously reads data from a Transceiver, feeds it through a row of Transformers, and broadcasts the result using a Publisher.

When creating a Reporter, you supply instances of a Transceiver, one or more Transformers, and a Publisher. If not specified, a pickling Transformer and the default Publisher are used.

Usage:

```
reporter = Reporter(
    transceiver=MyTransceiver(),
    transformers=[MyDecoder(), MyCompressor(), ...],
    publisher=MyPublisher()
)
```

A Reporter can be run as its own process:

```
reporter.run()
```

Or stepped through by an external engine:

```
reporter.open()
while not done:
    reporter.step()
```

open()
Initialize the Transceiver and Publisher.

step()
Read data and feed it into the first Transformer.

run()
Initialize components and start broadcasting.

Injector

class zircon.injectors.base.**Injector**(*subscriber=None*, *transformers=None*, *datastore=None*)
An Injector listens for data from a Reporter, feeds it through a row of Transformers, and inserts the result into a Datastore.

When creating an Injector, you supply instances of a Subscriber, one or more Transformers, and a Datastore. If not specified, an unpickling Transformer and the default Subscriber and Datastore are used.

Usage:

```
injector = Injector(
    subscriber=MySubscriber(),
    transformers=[MyDecompressor(), MyFormatter(), ...],
    datastore=MyDatastore()
)
```

An Injector can be run as its own process:

```
injector.run()
```

Or stepped through by an external engine:

```
injector.open()
while not done:
    injector.step()
```

open()
Initialize the Subscriber.

step()
Receive data and feed it into the first Transformer.

run()
Initialize components and start listening.

Socket.IO Server

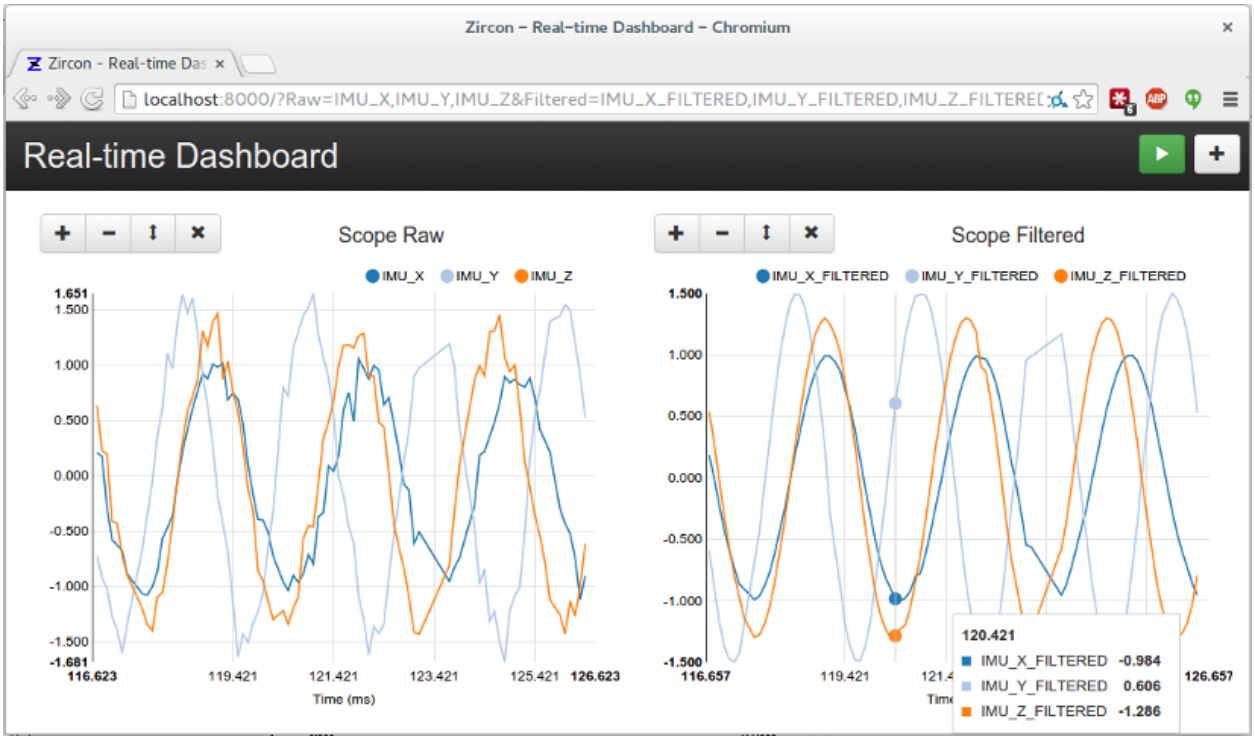
Note: This documentation is a work in progress. Browse the code in `zircon/zircon/frontend`.

Zircon's backend server provides a Socket.IO interface to query information from a Datastore. It allows real-time bidirectional event-based communication between a client application that receives data and the Datastore. Works for web or native applications, on any platform. Allows powerful querying of every recorded data point, by default for the past week.

Client Applications

Note: This documentation is a work in progress. Browse the code in `zircon/zircon/frontend`.

A client is anyone who wants to access the data coming from Transceivers. It is the end goal of Zircon to provide a fast, robust, and easy way for clients to monitor the signals they are interested in. Clients can be web apps, mobile apps, native apps, or hardware systems. Zircon's default client is a web dashboard that allows real-time visualization of arbitrary signals. It is a Django application with a JavaScript interface to the Socket.IO connection.



CHAPTER 2

Index

- genindex

Python Module Index

z

Index

Symbols

B

C

D

G

I

L

O

P

R

S

T

U

W

Z