pysig Documentation

Release 0.7.1

Mircescu Daniel-Alexandru

Dec 19, 2019

Contents

Contents:

CHAPTER 1

About

Pysig is a framework designed to manage event dispatching between two or more registered endpoints. The main philosophy behind a signaling framework, like **pysig** is, is to simplify the process by whitch a certain endpoint receives notifications or events from another, in an efficient and simple way.

One of the most interesting features of **pysig** is that it has the ability to dispatch messages over network, between different machines or different processes running on the same machine.

1.1 Feature set

To make a *very short* summary of what **pysig** can do, we could list the followings:

- **it enables subscription mechanisms that are able to**
  - register for a **specific event** triggered by a **specific sender**
  - register for **broadcast events** triggered by **any** event fired by a **specific sender**
  - register for **broadcast events** triggered by **any** event of **any sender**
  - register for **channel events** triggered by those **senders** that share same **event**
  - register listeners without depending on the sender registration
- supports firing **requests** to connected senders, for accessing data on reply
- stateful sender connection, automatically firing **connect** and **diconnect** events
- distribute events intra-process, inter-process or over a given network
- built-in server and client implementation
- built-in support for transporting messages via TCP
- permits custom transport carrier implementation for dispatching events over different communication mediums (like serial connections) or for transporting them under a different format
- permits custom data encoding (default is JSON encoding)

1.2 Terminology

Several terms are used in the context of this library and their meaning will be described here:

1.2.1 Router

Creates and manages the list of senders and listeners.

When run over a certain network, the router connects itself to a given **carrier** in order to transport messages and also comes in two flavors:

- the **server** that is responsible with managing with the flow of events for all registered listeners or senders
- the **client** that is responsible with communicating with the **server** for the purpose of registering listeners, senders or events

1.2.2 Sender

Represents the endpoint that signals the events.

It is also the endpoint that is optionaly capable of responding to **requests**.

1.2.3 Listener

Represents the endpoint that registers itself for receiving signaled events.

1.2.4 Event

Represents the object that stores the list of registered listeners.

An event is identified by its given name and is attached to a specific sender.

There are many types of events supported by pysig:

- specific events
- broadcast events
- channel events

1.2.5 Signal

Represents the object responsible with triggering the events.

The signal carries out the list of events it knows it must fire when invoked.

1.2.6 Carrier

Represents the object in charge with transporting data over a given communication medium.

pysig comes with a list of built-in carriers, described later on in this documentation.

1.3 Logging support

Before using the library, it will be good to know that is supports logging. The default 'logging' library from python, can be connected to pysig, in the following manner:

```
import logging
import sig

# prepare logger object
logging.basicConfig()
logger = logging.getLogger('sig')

# setup sig logger
sig.setup_logger(logger)
```

If by any means, the **logging** library is not acceptable, **pysig** has its own logging utility called **SimpleLogger**. This utility will use **print** to output the logs, and **inspect** module in order to trace source line number and caller function. It will be described later in detail. (**TBD**)

```
import sig

# prepare logger
logger = sig.SimpleLogger(tag= "[sig]", level= sig.LOG_LEVELS.LEVEL_INFO)

# setup sig logger
sig.setup_logger(logger)
```

A glance of how the log will look like:

```
[sig][1713][              connect][    info]-[TCP_CL] Connecting to localhost:3000
[sig][1607][     _thread_receive][    info]-[TCP_CL] Started receive thread for␣
↪localhost:3000
[sig][1653][     _thread_receive][    info]-[TCP_CL] Receive thread terminated
```

1.4 Roadmap

The current version of **pysig** is `0.7.1`.

Until **pysig** reaches it's first stable version, which will be `1.0`, the author reserves the right of changing the API if it's necessary. From version `1.0` all the versions of **pysig** sharing the same *major number* will be API compatible.

The modifications targeted for `1.0` version are:

- improve coverage of unit-tests
- improve multi-threading support
- improve **ServerRouter** and **ClientRouter** implementation
- improve **CarrierTCPClient** and **CarrierTCPServer** implementation
- improve examples
- improve overall documentation structure
- **refactor** design for direct requests
- add validation of communication protocol between **ServerRouter** and **ClientRouter**

- add SSL support to built-in TCP carriers
- add support for using multiple carriers at once for the very same server

Far fetched objectives:

- add **ACL** (Access Control List) support, for allowing access only to some listeners

1.5 Authors

Library develop and maintained by **Alex Mircescu**.

For feature requests or comments please address to `mircescu [at] gmail.com`.

CHAPTER 2

Getting started

This sections intends to be a crash course of **pysig** library, describing basic usage scenarios. For more information please refer to the rest of documentation. It will provide you all the necessary details to understand all the features supported by **pysig**.

2.1 The basics

This example will show how you can create a simple router for managing the flow of events between a listener and a sender. First, lets create a sender.

```
import sig

# create router object
router = sig.Router()

# create a sender
sender = router.addSender("my_sender", [ "my_event_1", "my_event_2"])
```

It's simple enough to understand, that we have created a sender object called **my_sender** that may trigger two events in the near future.

Let's trigger first event, to see how this is done.

```
# create a trigger for my_event_1
sig_ev1 = sender.getSignal("my_event_1")

# trigger this event
data = { "info1" : "something", "info2" : 2 }

# do trigger
sig_ev1.trigger(data)
```

For the sake of simplicity, we can write this in only one line:

```
router.getSender("my_sender").getSignal("my_event_1").trigger({"info1":"somedata",
↪"info2":2})
```

Now, what we have done, is to trigger an event with additional data attached, even though no listeners are registered to it. That is no problem for **pysig**, but is somehow useless.

Lets define our callback function.

```
# this is your callback function
def listen_to_events(info, data):
        event = info.get("event")
        sender = info.get("sender")
        print "Received event '%s' from sender '%s' having data '%s'" % (event,␣
↪sender, data)
```

Now, let's register it to the sender.

```
# let's register our listener callback
router.addListener(listen_to_events, "my_sender", "my_event_1")
```

It's done, we are listening to the first event of our sender. You may notice, we used the router object for registering the listener. We may also use the sender object with the same effect.

```
sender.addListener(listen_to_events, "my_event1")
```

The router object simplifies the registration. But be aware, when using the router, the sender and event parameter are purely optional. That is for a very good reason, the listener may register to **all** sender events or even **all** router events, using special **broadcast** events that will be described later on in this documentation.

Note!

Please note that **sender_identifier** and **event_identifier** are the exact same *objects* passed by the caller to the **Router.addSender** function and not just strings. The only restriction regarding these parameters is that they need to be JSON-serializable and *hashable*. Otherwise said, they can be **str** objects as well as **int**, **float** objects.

For example, this code is perfectly valid for **pysig**, but it also may be confusing:

```
[...]
sender = router.addSender( 20, [ 1, 2.0, "2.0"])
sig_ev1 = sender.getSignal(1)
sig_ev2 = sender.getSignal(2.0)
sig_ev3 = sender.getSignal("2.0")
[...]
```

You can also use dictionaries as long as they are encoded:

```
import json
[...]
sender_name = { "id" : 123, "source" : "http://www.something.com" }
sender_name = json.dumps(sender_name)
sender = router.addSender(senderName, [1,2,3] )
[...]
```

2.1.1 Example

Lets see how it looks, in the proper order.

```
import sig

# this is your callback function
def listen_to_events(info, data):
        event = info.get("event")
        sender = info.get("sender")
        print "Received event '%s' from sender '%s' having data '%s'" % (event,␣
↪sender, data)

# create router
router = sig.Router()

# create sender
sender = router.addSender("my_sender", ["my_event_1", "my_event_2"])
sig_ev1 = sender.getSignal("my_event_1")

# register listener
router.addListener(listen_to_events, "my_sender", "my_event_1")

# trigger first event
sig_ev1.trigger({"info1":"something", "info2":2})
```

The output will be, of course:

```
Received event 'my_event_1' from sender 'my_sender' having data '{'info1': 'something
↪', 'info2': 2}'
```

2.2 Network

The above example is useful for dispatching events *intra-process*, otherwise said, inside the same application. For distributing these events over a network, we have to use the server and client implementation of **pysig**.

2.2.1 Carrier

The communication between a **pysig** server and its client are assured by the **Carrier** class. This class defines the abstraction layer necessary for **pysig** to communicate. We can implement your own carrier, by deriving from this class.

Currently **pysig** supports only built-in **TCP Client** and **TCP Server** carriers.

2.2.2 Server

In order to receive commands remotely, you need to use instead of the **Router** class the **ServerRouter** class. The difference of this two, is that the second one requires a carrier for instantiation.

2.2.3 Client

In order to send commands remotely, you need to use the **ClientRouter** class instead of the **Router** class.

Also, instead of `addSender`, `addListener`, `removeSender`, `removeListener` methods, you are supposed to use the corresponding `addRemoteSender`, `addRemoteListener`, `removeRemoteSender` and `removeRemoteListener` methods.

The first set of methods will register senders and listeners only locally, as explained later in this documentation.

2.2.4 Examples

In the root of **pysig** repository we can find the `examples` folder. In this folder we can find several ready to use examples, that shows network functionality support. We will enumerate some of them.

generic_server.py

This example starts a basic **pysig** router server that also registers a sender which periodically fires a given event. The name of the sender and the event fired can be customized the the caller from **stdin**.

```
Log level   (default: info):
Server bind (default: 0.0.0.0):
Server port (default: 3000):
Sender      (default: timer):
Event       (default: tick):
Event data  (default: None):server data
Timer delay (default: 5):
[sig][  36][               info][    info]-[TCP_SRV] Starting to listen on 0.0.0.
↪0:3000
[sig][  79][           <module>][    info]-[timer] Triggering event tick with data
↪'server data'
```

generic_client_sender.py

This example shows how to create a router client that connects to a given server. It uses the client to register a custom sender that triggers an event periodically. The address of router, the name of the sender and the name of the event are read from **stdin**.

```
Server IP   (default: 127.0.0.1):
Server port (default: 3000):
Sender      (default: timer_client):
Event       (default: tick):
Event data  (default: None):client data
Timer delay (default: 5):
[sig][  36][               info][    info]-[TCP_CL] Connecting to 127.0.0.1:3000
[sig][  36][               info][    info]-[TCP_CL] Started receive thread for 127.0.
↪0.1:3000
[sig][  60][           <module>][    info]-[timer_client] Triggering event tick with␣
↪data 'client data'
```

generic_client_listener.py

This example shows how to create a router client that connects to a given server. It uses the client to register a listener to a specific event, a **broadcast** event or a **channel** event. It reads the address of the router from **stding** and also the name of the sender and event.

The output below, shows how we ran the example and instructed it to listen for the *tick* event issued by the server called *timer*.

```
Server IP    (default: 127.0.0.1):
Server port  (default: 3000):
Sender       (default: None):timer
Event        (default: None):tick
[sig][  36][                info][    info]-[TCP_CL] Connecting to 127.0.0.1:3000
[sig][  36][                info][    info]-[TCP_CL] Started receive thread for 127.0.
↪0.1:3000
[sig][  41][            listener][    info]-[timer] Event tick received with data '
↪"test"'
```

We can also instruct it to listen to the *tick* event issued by the `generic_client_sender.py` as long as this example runs.

```
Server IP    (default: 127.0.0.1):
Server port  (default: 3000):
Sender       (default: None):timer_client
Event        (default: None):tick
[sig][  36][                info][    info]-[TCP_CL] Connecting to 127.0.0.1:3000
[sig][  36][                info][    info]-[TCP_CL] Started receive thread for 127.0.
↪0.1:3000
[sig][  41][            listener][    info]-[timer_client] Event tick received with␣
↪data '"client data"'
```

When `generic_client_sender.py` is restarted, the example should output the following:

```
[sig][  34][    listener_connect][    info]-Sender timer_client is now disconnected
[sig][  34][    listener_connect][    info]-Sender timer_client is now connected
```

That is because the `generic_client_listener.py` automatically registers for the built-in events **sig.EVENT_CONNECT** and **sig.EVENT_DISCONNECT**, for the indicated sender as this code shows:

```
client.addRemoteListener(listener_connect, sender, sig.EVENT_CONNECT)
client.addRemoteListener(listener_connect, sender, sig.EVENT_DISCONNECT)
client.addRemoteListener(listener, sender, event)
```

Note!

> Using the examples, mentioned above, you can easily experiment the concept of **broadcast** events, **broadcast** senders and **channel** events, by simply modiifying the parameters passed from **stdin**.

alert_listener.py

This example implements a listener connected remotely to a **pysig** server, to a list of events from a plural of senders, that triggers a Desktop notification for **Linux** users, whenever an event was triggered.

web_events_logger.py

This example implements a web application that logs all the received events. You need to install `web.py` library in order to make it work.

You can install `web.py` using pip:

```
pip install web.py
```

Just like `alert_listener.py` this example implements a listener connected remotely to a **pysig** server.

From **stdin** you can configure it to listen to a list of specific events, from a list of specific senders or to the **broadcast** event of specific senders or to **channel** events. After is configured you can modifiy the list of subscriptions by using a RESTful API.

The log of events are printed out either in **HTML** or **PLAIN** text format.

The following is an example input of a possible configuration of `web_events_logger.py` example, instructing it to:

- connect to the **pysig** server at *192.168.9.20:3000*
- limit the number of events to 200 entries, after which the least-recent entry will be deleted
- listen to all events from *ifttt* sender (registering to iffttt **broadcast** event)
- listen to the specific *ir* event from *keros* sender
- listen to the *#notify* channel from **all** senders

```
Server IP   (default: 127.0.0.1):
Server port (default: 3000):
Queue limit (default: 20):200
Sender 1:ifttt
Event  1:None
Event  2:
Sender 2:keros
Event  1:ir
Event  2:
Sender 3:None
Event  1:#notify
Event  2:
Sender 4:
```

After configuring the web application via **stdin** the following output should be seen:

```
[sig][  36][                info][    info]-[TCP_CL] Connecting to 192.168.9.20:3000
[sig][  36][                info][    info]-[TCP_CL] Started receive thread for 192.
↪168.9.20:3000
[sig][  79][perform_registration][    info]-Registering to ifttt:None
[sig][  90][perform_registration][    info]-done
[sig][  79][perform_registration][    info]-Registering to keros:ir
[sig][  90][perform_registration][    info]-done
[sig][  79][perform_registration][    info]-Registering to None:#notify
[sig][  90][perform_registration][    info]-done
http://0.0.0.0:8080/
```

The RESTful API that you can use with this web application is:

/ The index page will return the recorded log entries. An optional parameter is the **fmt** parameter that can be passed as **plain** or **html**. Default is **html**.

Another set of optional arguments is the **sender** and **event** parameter, that once passed to the GET request it will filter out the log for the given sender and/or event.

Ex: `http://localhost:8080/?fmt=plain`

/regs or /registrations Returns the list of subscriptions this application is currently registered at. Same as for index / page, the optional **fmt** parameter is used to select format.

Ex: `http://localhost:8080/regs?fmt=plain`

/clear or /flush Resets the list of logged events.

Ex: `http://localhost:8080/clear`

`/config` or `/cfg` Receives the optional argument **limit** that is used to modify the limit of events that logs and then outputs a **JSON** containing the current configuration.

Ex: `http://localhost:8080/config?limit=100`

`/add` Registers to the given subscription and receives a **sender** and **event** parameter. If you omit one of them, it will be defaulted to **None**.

Ex: `http://localhost:8080/add?sender=timer&event=tick`

`/remove` or `/rem` Removes a certain registration by using the given **sender** and **event** parameters. If you omit one of them, it will be defaulted to **None**.

Ex: `http://localhost:8080/remove?sender=timer&event=tick`

CHAPTER 3

Features

3.1 Subscriptions

The above example shows how we can create a trivial sender and then register a listener to one of its events. This mechanism is called **subscription** and it has several aspects related to it.

One of this aspects is that a listener can subscribe to a sender, before the sender is connected to **pysig**. A listener is not forced to wait for a sender to connect in order to subscribe to its events. Also, the listener can be informed when a sender is connected or disconnected via a special event, defined by **sig.EVENT_CONNECT** and **sig.EVENT_DISCONNECT** attributes.

3.1.1 Types of subscriptions

A listener can connect to multiple type of events, as follows:

- a specific event, triggered by a specific sender
- any event triggered by a specific sender, called **sender broadcast** event
- any event triggered by any sender, called **router broadcast** event
- a specific event, triggered by any sender, called **channel**

Type of Event	Event	Sender	Syntax
normal	specific	specific	router.addListener(callback, “sender”, “event”)
sender broadcast	any	specific	router.addListener(callback, “sender”, None)
router broadcast	any	any	router.addListener(callback, None, None)
channel	specific	any	router.addListener(callback, None, “event”)

Each of these types of events will be described below, in this documentation.

3.1.2 Example

Let's see how a listener can subscribe before a sender is connected, in the following example:

```
import sig

# this function will be called when the sender connects or disconnects
def listen_to_connect_disconnect(info, data):
    event = info.get("event")
    print "Sender connect/disconnect event, current state: ",
    if event == sig.EVENT_CONNECT:
        print "connected"
    else:    print "disconnected"

# this is your callback function
def listen_to_events(info, data):
    event = info.get("event")
    sender = info.get("sender")
    print "Received event '%s' from sender '%s' having data '%s'" % (event, sender,␣
↪data)

def print_sender_state():
    print "Sender state: %s" % ("connected" if router.getSender("my_sender").
↪isConnected() else "disconnected")

# create router
router = sig.Router()

# register listener, even thouhg no sender is yet connected
router.addListener(listen_to_events, "my_sender", "my_event_1")

# register to connect and disconnect events
router.addListener(listen_to_connect_disconnect, "my_sender", sig.EVENT_CONNECT)
router.addListener(listen_to_connect_disconnect, "my_sender", sig.EVENT_DISCONNECT)

print_sender_state()

# create sender (connect event will be triggered)
my_events = ["my_event_1", "my_event_2"]
sender = router.addSender("my_sender", my_events)

print_sender_state()

# disconnect sender (disconnect event will be triggered)
router.removeSender(sender)

print_sender_state()

# connect sender again (connect event will be triggered)
sender = router.addSender("my_sender", my_events)

print_sender_state()

# trigger first event
sender.getSignal(my_events[0]).trigger(None)
```

This **example** will have the following output:

```
Sender state: disconnected
Sender connect/disconnect event, current state:  connected
Sender state: connected
Sender connect/disconnect event, current state:  disconnected
Sender state: disconnected
Sender connect/disconnect event, current state:  connected
Sender state: connected
Received event 'my_event_1' from sender 'my_sender' having data 'None'
```

3.2 Broadcast events

We've talked about broadcast event, now it will be good to show an example of how they work.

Just before we start, we need to know that there are two broadcast events in **pysig**:

1. the **sender** broadcast event
2. the **router** broadcast event

3.2.1 Sender broadcast event

As the name suggests, the sender broadcast event, is that special event that is sent each time a specific sender triggers a signal.

A small example of how we can register a the broadcast event of a specific sender named **my_sender**:

```
# register to all events from this sender
router.addListener(listen_to_events, "my_sender")
```

This special event will be triggered upon any event sent by **my_sender**. When registering to a broadcast event, we can differentiate between different events fired by the same sender by using the **info** dictionary parameter passed to listeners callback.

3.2.2 Router broadcast event

This is a special event that is triggered before any event from any sender is triggered. When a listener is registered to this event, it will receive all events that are managed by the router object.

```
# register to all events from this router
router.addListener(listen_to_events)
```

Of course, you can differentiate between different senders, using the same **info** parameter. The **info** parameter is constructed by the **Signal** class and contains the following information:

```
{
  "sender" : sender_identifier
  "event"  : event_identifier
}
```

3.2.3 Example

In this example we will register a single listener, connected to all router events. The listener will print out when a sender is connected or disconnected and any other event triggered by any sender.

```
import sig

# this function will be called for any signal triggered
def listen_to_any(info, data):
    event = info.get("event")
    sender = info.get("sender")

    if event == sig.EVENT_CONNECT:
        print "Sender '%s' is now connected" % (sender)
    elif event == sig.EVENT_DISCONNECT:
        print "Sender '%s' is now disconnected" % (sender)
    else:
        print "Sender '%s' triggered event '%s' with data '%s'" % (sender, event,␣
↪data)

# create router
router = sig.Router()

# register listener, even thouhg no sender is yet connected
router.addListener(listen_to_any)

# create sender (connect event will be triggered)
my_events = ["my_event_1", "my_event_2"]
sender = router.addSender("my_sender", my_events)

# disconnect sender (disconnect event will be triggered)
router.removeSender(sender)

# connect sender again (connect event will be triggered)
sender = router.addSender("my_sender", my_events)

# connect another sender
sender2 = router.addSender("my_second_sender", my_events)

# trigger some events
sender.getSignal(my_events[0]).trigger(None)
sender2.getSignal(my_events[1]).trigger(None)
```

And the output:

```
Sender 'my_sender' is now connected
Sender 'my_sender' is now disconnected
Sender 'my_sender' is now connected
Sender 'my_second_sender' is now connected
Sender 'my_sender' triggered event 'my_event_1' with data 'None'
Sender 'my_second_sender' triggered event 'my_event_2' with data 'None'
```

Needless to say, registering broadcast events can be a performance penalty if the listener is only interested in receiving only a couple of events and not all. It may still be good for debugging and tracing events when necessary.

3.3 Channel Events

A **channel** event is actually an event that shares a plural of senders. So far, the structure of events was limited to the scope of the senders that declared them upon registration.

For example, let's say you have three posible sensors, each one destined to measure temperature, humidity and light intensity. These will play the role of senders, called *temp*, *humidity* and *light*. They all register the same event, called *changed*, that is fired when the value of their measurement changes significantly.

In order to listen for the *changed* event, you have to register three times, as follows:

```
router.addRemoteListener(callback, "temp"    , "changed")
router.addRemoteListener(callback, "humidity", "changed")
router.addRemoteListener(callback, "light"   , "changed")
```

If the data published to any of the *changed* event is generic enough to determine its type, registering for each sender in particular may not be so scalable if another set of sensors will be deployed later on. We will have to modify the code to keep adding subscription to senders that trigger the same event.

For this purpose **pysig** introces the notion of **channel** events. Using **channel** events, a listener can only subscribe to one generic event and listen for events from senders that share the same channel, transparently.

Like in the case of **broadcast** events, you can differentiate between different senders using the **info** parameter, passed to the callback.

3.3.1 Senders

The design of **pysig** intends that **channel** events to be declared explicit by the senders that want to share the same event. In order to achieve this easily, we only need to prefix the events with the "#" character.

For the temperature sensor, the code would look like:

```
sender = router.addRemoteSender("temp", ["#changed", "cold", "hot"])
```

This tells **pysig** that the *changed* event is a **channel** event, while the *cold* and *hot* events are events specific to this sensors.

To use the same **channel**, the humidity sensor will register like this:

```
sender = router.addRemoteSender("humidity", ["#changed", "dry", "wet"])
```

Now, the two senders share the same *changed* event. In comparison to a regular event, the main difference for the *changed* event is that allows listeners to register to this event *without the need of knowing which senders are publishing it*.

3.3.2 Listeners

This is where **channel** events are visibly different from any others.

We can now capture all *changed* events with only one subscription:

```
router.addRemoteListener(callback, event= "#changed")
```

or, equaly correct

```
router.addRemoteListener(callback, None, "#changed")
```

As it appears, we have registered to the broadcast sender for an event called *changed*. We will now receive the *changed* event from **any** sender that uses this **channel**.

3.3.3 Good to know

Channel events offers great flexibility for listeners, but they introduce some complexity that needs to be detailed.

- The built-in events of **pysig** are by default **channels**. Therefore it may be possible to listen to any **sig.EVENT_CONNECT** event, for any sender that connects to the router, without having the need to subscribe to all published events and filter out connect messages.

  ```
  router.addRemoteListener(listen_for_connects, event= sig.EVENT_CONNECT)
  ```

- If a sender does not explicitly declare an event as being a **channel**, it is considered a regular event. Therefore the following listener will receive **nothing** from the *temp* sender, even if the *cold* event is fired by it, because the *temp* sender didn't declared the event as being a **channel**:

  ```
  router.addRemoteListener(callback, event= "cold")
  ```

- The following subscription will register for the *changed* event from the *temp* sender and not to the *changed* **channel**, therefore will receive events only from *temp* sender:

  ```
  router.addRemoteListener(callback, "temp","#changed")
  ```

3.4 Requests

A nice feature of **pysig** is that is able to *request* data from a particular sender, without having the need of waiting a particular event to achieve that. This is called a *request* and can be made by anyone that has access to the **Router** object, to any registered sender.

The way you may issue a request, is as follows:

```
import sig
router = sig.Router()
[...]
response = router.request("my_sender", "getWeather")
print "Response for 'getWether' is '%s'" % (response)
```

The limitation of the requests feature, is that it can only be issued to senders that:

1. are connected to the router
2. implements a special request handler

3.4.1 Example

The way you register a request handler for a sender is:

```
import sig

class SenderRequestHandler:
        def getWeather(self, params):
                if params.get("when","") == "now":
                        return 20.2
                else
                        return "unknown"

        def get(self, method, params):
                return "Unknown method %s" % (method)

# create router
router = sig.Router()

# create sender with support for requests
sender = router.addSender("my_sender", ["weather_change"], request_handler=␣
↪SenderRequestHandler())

# now we can request things from sender
response = sender.request("getWeather", {"when":"now"})
print "Weather now is: %s" % (response)

response = router.request("my_sender","getWether",{"when":"tomorrow"})
print "Weather tomorrow is: %s" % (response)

# request something unknown
response = router.request("my_sender", "getWeatherInformation", None)
print response
```

Notice how we can make requests directly using **Sender** instance or indirectly via the **Router** instance.

Also notice the **get** method from **SenderRequestHandler** that is invoked when a particular method is not found as being implemented by the request handler. This generic method is optional, but if not implemented a request sent with an unknown method for a particular sender, will otherwise raise an **LookupError** exception. This would have happened in the case of the last request to *getWeatherInformation*.

You can also choose just to implement this generic **get** method instead of implementing separate methods for each method invoked by a request.

As a summary, the things you want to know about a request are:

1. Works only for senders that are currently registered
2. Any time the sender is reconnected it must pass its request handler object
3. If the sender request handler doesn't implement the generic **get** method, any request with an unknown method will raise an **LookupError** exception
4. If the method implemented by the request handler raises an exception, the exception must be caught by caller
5. The request returns the reponse received by the method implemented by the request handler

CHAPTER 4

Extending functionality

The design of **pysig** is flexible so that the user can override most of it's functionality. We will show a couple of examples of how that can be useful.

4.1 The basics

Each router object can be instructed to use different classes. when constructing **Sender**, **Event** or **Signal** objects. This can be done when a new router object is created, likewise:

```
import sig

class MySender(sig.Sender): pass
class MySignal(sig.Signal): pass
class MyEvent(sig.Event): pass

router = sig.Router(class_sender= MySender, class_event= MyEvent, class_signal=␣
↪MySignal)
```

Or it can be done on the fly, like this:

```
router.class_sender = MySender
router.class_event = MyEvent
router.class_signal = MySignal
```

If you intend to modify the behavior of the **Signal** class, just for a particular **Sender**, this is how you may proceed:

```
# find my sender
sender = router.getSender("my_sender")

# all signals created by this sender, from now on, will be
# instances of MySignal class
sender.class_signal = MySignal
```

4.2 Exteding Signal class

The **Signal** class is responsible for triggering events in **pysig**. The **Signal** object is created by the **Sender** class using the method **getSignal**, just like you've seen above.

By overriding the **trigger** method of the **Signal** class, we can add or modify the **data** parameter or we may even add other types of information to the **info** parameter.

Due to the fact that the **info** parameter passed to the callback of a listener, is a dictionary object, we may add additional information if we *hook* the **Signal** class and override the **trigger** method.

The information we've decided to add to the **info** parameter in the following example, is the number of occurences of each event sent by any sender. For that matter, we've registered a listener to the router broadcast event, in order to capture all events and we've used it to print the number of occurrences of that particular event, whenever is triggered.

The number of occurrences is stored globaly and added to the **info** parameter by our new **Signal** class called **MySignal**.

The full example looks like this:

4.2.1 Example

```
import sig

# counts the number of occurrences for each key
count_dict = {}
def count_message_from(key):
        global count_dict
        value = count_dict.get(key,0)
        value += 1
        count_dict[key] = value
        return value

# my own signal class
class MySignal(sig.Signal):

        def __init__(self, objevent, broad_events):
                sig.Signal.__init__(self, objevent, broad_events)

        def trigger(self, data):
                event_name  = self.objevent.getName()
                sender_name = self.objsender.getName()

                # count each message occurrence count
                value = count_message_from("%s:%s" % (sender_name, event_name))
                self.event_info["count"] = value

                # call super
                sig.Signal.trigger(self, data)

# listener callback
def listen_to_any(info, data):
        sender = info.get("sender")
        event  = info.get("event")
        occur  = info.get("count")
        print "Sender: %17.17s Event: %12.12s Occurrence: %u times" % (sender, event,␣
↪occur)
```

(continues on next page)

(continued from previous page)

```
# print statistics
def print_statistics():
        print "\nEvent statistics:"
        for key, value in count_dict.iteritems():
                print "%30.30s was triggered %u times" % (key, value)

# create router and instruct it to use a custom  Signal class
router = sig.Router(class_signal= MySignal)

# register broadcast listener
router.addListener(listen_to_any)

# create sender (connect event will be triggered)
my_events = ["my_event_1", "my_event_2"]
sender = router.addSender("my_sender", my_events)

# disconnect sender (disconnect event will be triggered)
router.removeSender(sender)

# connect sender again (connect event will be triggered)
sender = router.addSender("my_sender", my_events)

# connect another sender
sender2 = router.addSender("my_second_sender", my_events)

# trigger some events
sender.getSignal(my_events[0]).trigger(None)
sender2.getSignal(my_events[1]).trigger(None)

# remove senders
router.removeSender(sender)
router.removeSender(sender2)

# print statistics
print_statistics()
```

And the output:

```
Sender:         my_sender Event:     #connect Occurrence: 1 times
Sender:         my_sender Event:  #disconnect Occurrence: 1 times
Sender:         my_sender Event:     #connect Occurrence: 2 times
Sender:  my_second_sender Event:     #connect Occurrence: 1 times
Sender:         my_sender Event:   my_event_1 Occurrence: 1 times
Sender:  my_second_sender Event:   my_event_2 Occurrence: 1 times
Sender:         my_sender Event:  #disconnect Occurrence: 2 times
Sender:  my_second_sender Event:  #disconnect Occurrence: 1 times

Event statistics:
     my_second_sender:#connect was triggered 1 times
         my_sender:#disconnect was triggered 2 times
  my_second_sender:#disconnect was triggered 1 times
           my_sender:#connect was triggered 2 times
   my_second_sender:my_event_2 was triggered 1 times
          my_sender:my_event_1 was triggered 1 times
```

By modifying only the **class_signal** member of a **Sender** object, we can achieve the same effect but just for a particular sender.

CHAPTER 5

Network features

Using **pysig** for intra-process communication it's useful, especially for big applications or aplications divided in modules that want to comunicate events to each other without requiring to be aware of when the module is loaded by the main application. It simply favors a lite binding between two endpoints that want to signal specific events.

For small applications though, using a centralized event dispatching framework, sounds more like over-engineering than a good decision. How would be like to have a python application with a couple of functions that signals events to each other via a centralized event dispatching framework?!

But for those apps and not only, a very useful feature of **pysig** would be to communicate their events over the network. How would be like for your python application, even if it's small and straight-forward, to communicate it's results to another application, running in the same network (or even outside the local network) in real time?!

5.1 Important

When **pysig** is running over a network, it still supports the same number of features described before. That is, the following features are still available:

- registration to specific sender events
- connect/disconnect events
- broadcast events
- channels
- requests

5.2 Use cases

We would like to list just a couple of possible applications for communicating events over a network, using a simple framework like **pysig**.

5.2.1 Distributed applications

You can create a listener that subscribes itself to several senders and several events, that simply logs the incoming data and processes it in a meaningful way. The senders can connect when they want and signal events to the router whenever their finish their jos and have new meaningful data (e.g. a python script that measures disk usage on each machine).

You can make several different python applications that run on a schedule, by unix-like cron (or Windows Scheduler) and signals their results to the router. Whenever the application that listens for this events it's up, the data is stored or logged or processed.

This creates a very flexible setup, that can be adjusted on the run with no hassle.

5.2.2 Sensors

Almoast the same scenario as [1], just that in this case the *sensor* sends the data to our **pysig** router, based on a trigger that may happen spuriously (e.g. when the light sensor is detecting day or night) and not on a regular basis, as [1] implies.

Anyone interested on the information signaled by these *sensors* will register theirselfs to the router, for a specific event or for all events triggered by a sensor.

Both the sensors and the listeners can be installed on different machines communicating via the local network or the internet and connected to the centralized **pysig** router.

5.2.3 Push-like service

We can run a push-like service for your python applications and by using the dispatching mechanism implemented in **pysig** a listener that connects to the service, will register itself just for the events that presents interest.

The design in **pysig** is flexible, so you may implement your own message carrier, focusing only on how to transmit and receive data over the network and not the entire dispatching logic, that is already assured by **pysig**.

Therefore, you can make an UDP carrier, TCP carrier or even HTTP carrier that may run on plain or encrypted channels.

5.2.4 Inter-process communication

Of course, **IPC** is a good application example of **pysig**.

If your application requires dispatching events to another application, running on the same machine, **pysig** can do the job. You can run a server on the targeted machine that handles all the message dispatching, with different processes connect to it via sockets or pipes.

5.2.5 Custom transport

You can define custom carriers for **pysig** events that can transport them over different communication environments, like serial connections.

Once you define and implement your custom carrier, you can decide how the messages are packed, encrypted or compressed over this transport medium.

5.3 Design

In **pysig** there are three classes which allows senders, listeners and routers to be inter-connected via a common data transportation channel.

5.3.1 Server Router

The first one is the **ServerRouter**, which is reponsible with receiving commands and messages from its connected clients and dispatch them accordingly. The **ServerRouter** class is declaring several RPC methods (where RPC stands for Remote Procedure Call) in order to allow a remotely connected sender or listener to register and receive events.

All the senders registered to this endpoint, whether they where registered by the python application that created the object (using direct API calls like *ServerRouter.addSender*) or they are registered remotely (via a message), are visible to any listener connected to it.

The **ServerRouter** design is *stateful* meaning it's aware of each currently connected listener and each connected sender. Whenever one of them disconnects, it is automatically removed from the dispatching framework. To exemplify this more clearly, if you register a sender on a machine that somehow looses network connection with the **ServerRouter**, the server will automatically remove sender (i.e. just like if **ServerRouter.removeSender** was called) and it will fire the **sig.EVENT_DISCONNECT** for all listeners registered to this event.

5.3.2 Client Router

As you may expect, there is a implementation for the client side too. This **ClientRouter** allows you to register remote listeners and remote senders to a **ServerRouter** via whatever transportation carrier you are using.

Using this router, you can register senders and their corresponding events to the centralized **ServerRouter** and trigger events almoast the same way you would have done using a simple, local **Router** implementation. Those events will be communicated over network by the **ClientRouter**.

Important

Please note that there is a slight distiction between the **Router** and the **ClientRouter** class. If you use the functions **addListener/addSender** respectively, you will register listeners/senders only locally visible.

If you intend to add listeners or senders connected to the **ServerRouter**, you must use the following corresponding set of functions:

- addRemoteListener / removeRemoteListener
- addRemoteSender / removeRemoteSender

This distinction is valid only for **ClientRouter** and not for the **ServerRouter** where all registered senders or listeners are visible to the connected clients.

For firing requests you must use **remoteRequest** instead of **request** function.

5.3.3 Carrier

The **Carrier** implementation is the main actor of this remote signaling feature of **pysig**. The class is expected to be inherited by the one that really implements the transportation layery.

The role of the **Carrier** is to provide an abstract API for the **ServerRouter** and **ClientRouter** for sending and receiving messages.

The **Carrier** class defines the following methos, that MAY or MUST be implemented.

Carrier.pack(self, message)

This methods packs the received message to a format that is acceptable by the carrier. It returns the object containing the packed data or **None** in case of an exception.

The default implementation uses **json** module and encodes the message in a json object, therefore the method **MAY** be overrided.

Carrier.unpack(self, data)

This method unpacks the received data and returns the python dictionary object containing the message. In case of an exception it returns **None**.

The default implementation uses **json** module to decode the data, therefore it **MAY** be overrided.

Carrier.handleRX(self, clientid, message)

This method **MUST** be invoked by the carrier implementation whenever a new message is received. The **message** passed to this function must be already *unpacked* and ready to be interpreted.

The main purpose of this method is to translate the message and run the corresponding RPC method. The RPC methods supported will be listed by the **Carrier.methods** dictionary, in the following format:

Carrier.methods = { "method_name" : method_callback, [...] }

When this function is called, it will execute **Carrier.handleRPC** function to search for methods defined by **Carrier.methods** and execute them accordingly. The function will **always** return a reply message, that must be sent back to the client, even if the method is unsupported (is not present in **Carrier.methods**) or it fails during execution. The method will not raise an exception.

The **clientid** paremeter, uniquely identifies the client from which this mesage was received and it will be used mostly by the **ServerRouter** class to distinguish between multiple connected clients. It can be in any form (e.g. **int** or **str**), as long as it is uniquely identifying the client. For **ClientRouter** implementation it can be anything, it will be ignored.

For example, for a TCP Server, the **clientid** will be unique for each client connected to the listening socket. The identifier can be the **id** of the instance that is processing the communication with the client. When the **Carrier** receives this parameter on its **Carrier.handleTX** function, it will select the proper client to send its message to.

This method **MAY NOT** be overrided.

Message format

```
{
  "id" : "23",
  "method" : "add_sender",
  "params" :
      {
        "sender" : "lorem",
        "events" : ["ipsum"]
      }
}
```

Reply format

```
{
  "id" : "23",
  "status" : 0,
  "response" : None
}
```

As you can see in this format, the method name is stored in the **method** field while it's parameter within the **params** field. Each message must contain these two fields, including the **id** field that will be described below.

Each reply however, returns back the same **id** field received within the message, a **status** field containing the error code that **Carrier.handleRPC** returned and the **response** field that stores that the RPC method responded with.

In case the RPC method raised an exception, the reply will also store a field called **exception** containing the string representation of the exception raised.

Carrier.handleTX(self, clientid, message)

This method **MUST** be implemented in order to allow sending data to a specific client. The **message** passed to this function must be packed before doing the actual send operation. The default implementation does nothing.

The method **MUST** return the reply received by the client specified by **clientid**, in a python dictionary object, threfore it must be unpacked. In **pysig** no message is sent without receiving a reply.

This method **MUST** add to the **message** an unique identifier called the **id** field. This is useful for avoiding the case where the immediate data received after sending the message is NOT the reply that actually corresponds to this message but some other sent before it. The **Carrier.handleRX** default implementation, will store the **id** field from the message and sent it back in the reply (see the example above).

Note:

There are several rules that you must respect, when implementing a carrier: * each message sent respects the format above (contains id, method, params as fields) * each reply respects the format above (contains id, status, response and may contain exception as fields) * each call to **Carrier.handleTX** will return the corresponding reply in an unpacked form * the method **Carrier.handleRX** must be called for each message (not reply) received in an unpacked form

Carrier.handle_client_connected(self, clientid)

This method **MUST** be invoked by the implementation whenever a new client is connected. It is useful for **ServerRouter** and not for a carrier used in the context of the **ClientRouter**. Upon calling this method **ServerRouter** will map the corresponding data to this client.

Returns nothing.

Carrier.handle_client_disconnected(self, clientid)

This method **MUST** be called by the implementation whenever an existing client is disconnected. It is useful only for **ServerRouter**. Upon calling this method the **ServerRouter** will detach all registered listeners or senders corresponding by this client.

Returns nothing.

Carrier.handle_all_clients_diconnected(self)

This method **MAY** be called by the implementation whenever the server looses connection with all of his clients. This method is useful for **ServerRouter** and it's an optimized version for callind **Carrier.handle_client_disconnected** for each client in particular.

Returns nothing.

5.4 Built-in carriers

Currently **pysig** suppors several ready-to-use carriers, as follows:

- **TCP Server** for using it in conjuction with **ServerRouter**
- **TCP Client** for using it in conjuction with **ClientRouter**
- **Local** carrier, for testing purposes only

5.4.1 TCP Server

pysig provides a ready-to-use TCP Server carrier for connecting it to the **ServerRouter**. The way you use it is pretty simple

```
import time
import sig
from sig.carrier.tcpserver import *

# create the tcp server
tcp_server = CarrierTCPServer()

# create the server router
router = sig.ServerRouter(tcp_server)

# add a sender
sender_timer = router.addSender("timer", ["tic"])
signal_tic = sender_timer.getSignal("tic")

# start server
tcpserver.start("localhost", 3000)

# loop
try:
  while True:
    # tic every ten seconds
    signal_tic.trigger(None)
    time.sleep(10)
except KeyboardInterrupt:
  print "Stopping server.."
  tcpserver.stop()
  print "Done."
```

Very well, we have a server router that sends a **tic** signal every ten seconds. This signal can be listened by anyone on the network that can connect to this machine to tcp port 3000.

5.4.2 TCP Client

Also, **pysig** has a ready-to-use TCP Client carrier for pairing it with **ClientRouter**. This carrier of course can communicate with the built-in TCP server presented above.

Let's see how we can listen for the tic signal sent above:

```
import sig
import time
from sig.carrier.tcpclient import *

# create the tcp client
tcpclient = carrier.CarrierTCPClient()

# create the client router
router = sig.ClientRouter(tcpclient)

# connect client to the server
tcpclient.connect("localhost", 3000)

# register for the tic signal
```

(continues on next page)

(continued from previous page)

```
def listen_for_tic(info, data):
  print "'%s' received from '%s' (data: %s)" % (info.get("event"), info.get("sender"),
↪ data)

router.addRemoteListener(listen_for_tic, "timer", "tic")
router.addRemoteListener(listen_for_tic, "another_timer", "tic")

# loop
try:
  while True: time.sleep(10)
except KeyboardInterrupt:
  print "Disconnecting client.."
  tcpclient.disconnect()
  print "Stop"
```

Great, we have our client connected to the server above. Notice how we used **addRemoteListener** and not **addListener**. The difference between these two is that the last one only registers a listener to *local* senders and not the senders registered to our **ServerRouter**. That is, you can still use **addListener** to connect to senders directly registered to **ClientRouter** and not the **ServerRouter** we've created in our first example.

Also, please notice our second listener registration, to a sender called **another_timer**. For now, this client will register itself to an unexisting sender, which is quite legit in **pysig**. Wouldn't be nice to use a client for adding senders to the entire scheme?

Let's see how we can do that in our next example.

```
import sig
import time
from sig.carrier.tcpclient import *

# create the tcp client
tcpclient = carrier.CarrierTCPClient()

# create the client router
router = sig.ClientRouter(tcpclient)

# connect client to the server
tcpclient.connect("localhost", 3000)

# add our remote sender
sender = router.addRemoteSender("another_timer", ["tic"])
signal_tic = sender.getSignal("tic")

# loop
try:
  while True:
     time.sleep(10)
     signal_tic.trigger(None)
except KeyboardInterrupt:
  print "Disconnecting client.."
  tcpclient.disconnect()
  print "Stop"
```

That's it. If we run all three examples in the same time, we will have: * a server that triggers a *tic* event in the name of *timer* as sender * a client that triggers a *tic* event in the name of *another_timer* as sender * a client that registers for listening both *tic* events

Of course, as a consequence, the client that listens for the *tic* events will receive events from *another_timer* only when

the client that registers the remote sender is running. But will always receive the tic events from the *timer* sender, that is directly registered to the **ServerRouter**.

CHAPTER 6

Indices and tables

- genindex
- modindex
- search