Jasmin Documentation

Release 0.9.26

Jookies LTD

Dec 03, 2017

Contents

Jasmin is an open-source SMS Gateway with many enterprise-class features, Jasmin is built to be easily customized to meet the specific needs of messaging exchange growing business.

Based on strong message routing algorithms, Jasmin provides flexibility to define rule based routing based on various criteria: sender ID, source, destination and many combinations. Auto reconnection and re-routing mechanism managing peak hours or link failover for high availability services.

Jasmin is written in Python and Twisted framework for serving highly scalable applications, SMS message delivery can be done through HTTP and SMPP protocols, intelligent routing can be configured in real-time through an API, cli interface or a web backend[1].

[1] Web backend is provided under a commercial license, c.f. *Support*

CHAPTER 1

Features

- SMPP Client / Server
- HTTP Client / Server
- Based on AMQP broker for store&forward mechanisms
- Advanced message routing : Simple & static, *Roundrobin*, *Failover*, *Leastcost* ..
- Standard message filtering: *TransparentFilter*, *ConnectorFilter*, *UserFilter* ..
- Advanced message filtering: *EvalPyFilter*
- Flexible billing support
- Supports Unicode (UTF-8 / 16) for sending out multilingual SMS
- Supports easy creation and sending of specialized/binary SMS like mono Ringtones, WAP Push, Vcards
- Supports concatenated (multipart) SMS contents (long SMS)

Jasmin is designed for **performance**, **high traffic loads** and **full in-memory execution**.

CHAPTER 2

Getting started

- *Installation* – Install and run Jasmin SMS Gateway
- *Receiving SMS* – Basic push/pull SMS application via HTTP
- *RESTful API* – RESTful API technical specification
- *SMPP Server API* – SMPP Server API technical specification
- *Routing* – Running basic SMS and routing scenarios
- *User FAQ* – Frequently asked questions

CHAPTER 3

Full contents

3.1 Architecture overview

Jasmin is composed of several components with scoped responsibilities:

1. **jCli**: Telnet management console, refer to *Management CLI overview* for more details,
2. **SMPP Client Manager PB**: A PerspectBroker providing facilities to manage (add, remove, list, start, stop ...) SMPP client connectors,
3. **Router**: A PerspectBroker providing facilities to manage message routes, groups, users, http connectors and filters,
4. **DLR Thrower**: A service for delivering acknowledgement receipts back to third party applications through HTTP, refer to *HTTP API* for more details,
5. **DeliverSM Thrower**: A service for delivering MO SMS (Mobile originated) to third party applications through HTTP, refer to *HTTP API* for more details,
6. **Restful API**: A Restful API to be used by third party application to send MT SMS (Mobile Terminated) and launch batches, refer to *RESTful API* for more details.
7. **HTTP API**: A HTTP Server to be used by third party application to send MT SMS (Mobile Terminated), refer to *HTTP API* for more details.
8. **SMPP Server API**: A SMPP Server to be used by third party application to send and receive SMS through a stateful tcp protocol, refer to *SMPP Server API* for more details.

Jasmin core and its external connectors (used for AMQP, Redis, SMPP, HTTP, Telnet ...) are written in Python and are mainly based on Twisted matrix, a event-driven networking engine.

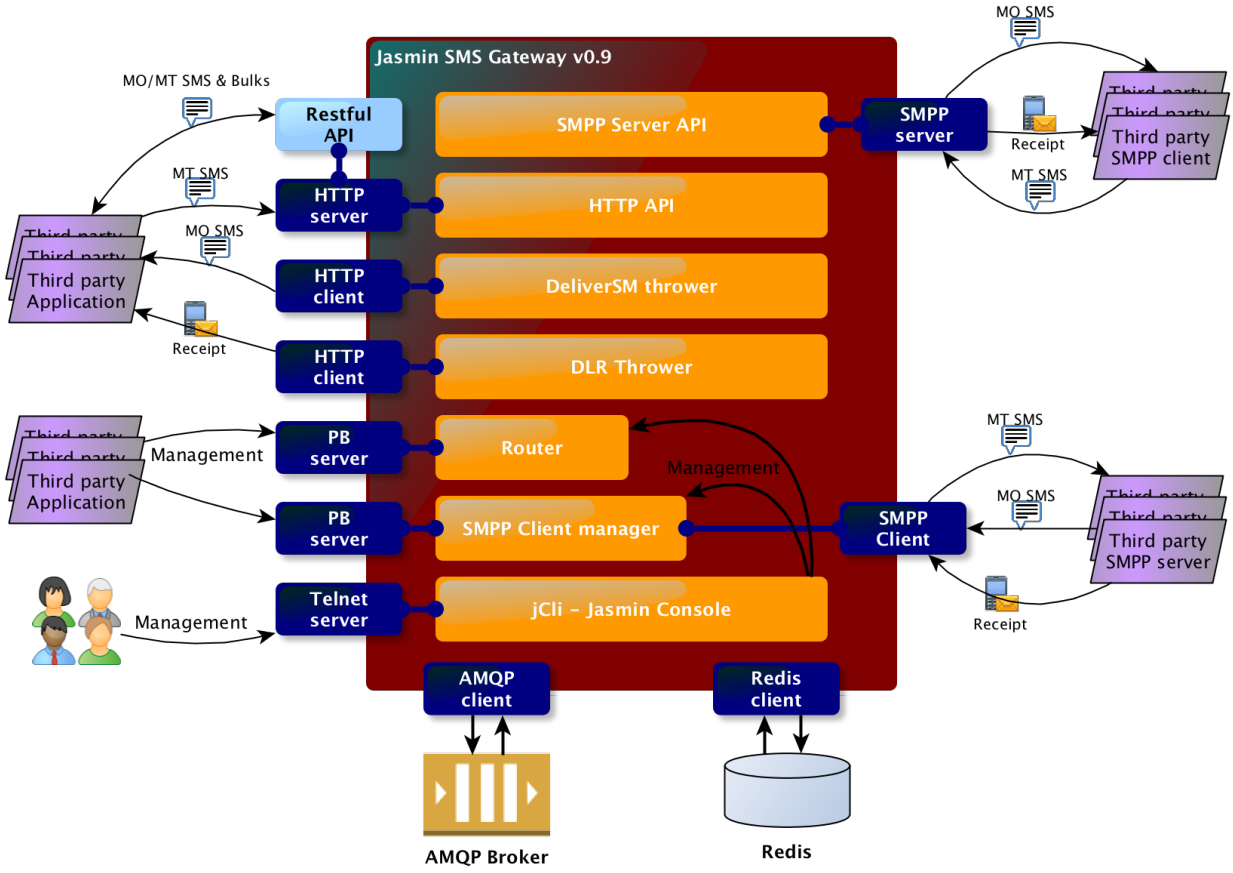


Fig. 3.1: Jasmin SMS Gateway high level design

3.2 Support

3.2.1 Getting Help

The easiest way to get help with the project is to open an issue on Github.

The forum is also available for support.

3.2.2 Commercial Support

We offer commercial support for Jasmin, commercial solution hosting, as well as remote and on-site consulting and engineering.

You can contact us at support@jasminsms.com to learn more, or visit us at http://jookies.net.

3.3 Installation

The Installation section is intended to get you up and running quickly with a simple SMS sending scenario through *HTTP API* or *SMPP Server API*.

Jasmin installation is provided as rpm & deb Linux packages, docker image and pypi package.

Important: Jasmin needs a working **RabbitMQ** and **Redis** servers, more info in *Prerequisites & Dependencies* below.

3.3.1 Prerequisites & Dependencies

Jasmin requires Python 2.7 or newer (but not Python 3) with a functioning pip module.

Hint: Latest pip module installation: # **curl https://bootstrap.pypa.io/get-pip.py | python**

Depending on the Linux distribution you are using, you may need to install the following dependencies:

- RabbitMQ Server, Ubuntu package name: **rabbitmq-server**. RabbitMQ is used heavily by Jasmin as its core AMQP.
- Redis Server, Ubuntu package name: **redis-server**. Redis is used mainly for mapping message ID's when receiving delivery receipts.
- header files and a static library for Python, Ubuntu package name: **python-dev**
- Foreign Function Interface library (development files), Ubuntu package name: **libffi-dev**
- Secure Sockets Layer toolkit - development files, Ubuntu package name: **libssl-dev**
- Twisted Matrix, Python Event-driven networking engine, Ubuntu package name: **python-twisted**

3.3.2 Ubuntu

Jasmin can be installed through **DEB** packages hosted on Packagecloud:

```
wget -qO - http://bit.ly/jasmin-deb-repo | sudo bash
sudo apt-get install python-jasmin
```

Note: Ubuntu 15.04 and higher versions are supported.

Once Jasmin installed, you may simply start the **jasmind** service:

```
sudo systemctl enable jasmind
sudo systemctl start jasmind
```

Note: redis and rabbitmq must be started with jasmin.

3.3.3 RHEL & CentOS

Jasmin can be installed through **RPM** packages hosted on Packagecloud:

```
wget -qO - http://bit.ly/jasmin-rpm-repo | sudo bash
sudo yum install python-jasmin
```

Note: Red Hat Enterprise Linux 7 & CentOS 7 are supported.

You may get the following error if **RabbitMQ** or **Redis** server are not installed:

```
No package redis available.
No package rabbitmq-server available.
```

These requirements are available from the EPEL repository, you'll need to enable it before installing Jasmin:

```
## RHEL/CentOS 7 64-Bit ##
yum -y install https://dl.fedoraproject.org/pub/epel/epel-release-latest-7.noarch.rpm
```

Once Jasmin installed, you may simply start the **jasmind** service:

```
sudo systemctl enable jasmind
sudo systemctl start jasmind
```

Note: redis and rabbitmq must be started with jasmin.

3.3.4 Pypi

Having another OS not covered by package installations described above ? using the Python package installer will be possible, you may have to follow these instructions:

System user

Jasmin system service is running under the *jasmin* system user, you will have to create this user under *jasmin* group:

```
sudo useradd jasmin
```

System folders

In order to run as a POSIX system service, Jasmin requires the creation of the following folders before installation:

```
/etc/jasmin
/etc/jasmin/resource
/etc/jasmin/store       #> Must be owned by jasmin user
/var/log/jasmin         #> Must be owned by jasmin user
```

Installation

The last step is to install jasmin through pip:

```
sudo pip install jasmin
```

systemd scripts must be downloaded from *here <https://github.com/jookies/jasmin/tree/master/misc/config/systemd>* and manually installed into your system, once placed in **/lib/systemd/system** jasmind shall be enabled and started:

```
sudo systemctl enable jasmind
sudo systemctl start jasmind
```

Note: redis and rabbitmq must be started with jasmin.

3.3.5 Docker

You probably have heard of Docker, it is a container technology with a ton of momentum. But if you haven't, you can think of containers as easily-configured, lightweight VMs that start up fast, often in under one second. Containers are ideal for microservice architectures and for environments that scale rapidly or release often, Here's more from Docker's website.

Installing Docker

Before we get into containers, we'll need to get Docker running locally. You can do this by installing the package for your system (tip: you can find yours here). Running a Mac? You'll need to install the boot2docker application before using Docker. Once that's set up, you're ready to start using Jasmin container !

Pulling Jasmin image

This command will pull latest jasmin docker image to your computer:

```
docker pull jookies/jasmin
```

You should have Jasmin image listed in your local docker images:

```
# docker images
REPOSITORY          TAG                 IMAGE ID            CREATED
↪VIRTUAL SIZE
jasmin              latest              0e4cf8879899        36 minutes ago      478.6
↪MB
```

Note: The Jasmin docker image is a self-contained/standalone box including Jasmin+Redis+RabbitMQ.

Starting Jasmin in a container

This command will create a new docker container with name *jasmin_01* which run as a demon:

```
docker run -d -p 1401:1401 -p 2775:2775 -p 8990:8990 --name jasmin_01 jookies/
↪jasmin:latest
```

Note that we used the parameter **-p** three times, it defines port forwarding from host computer to the container, typing **-p 2775:2775** will map the container's 2775 port to your host 2775 port; this can be useful in case you'll be running multiple containers of Jasmin where you keep a port offset of 10 between each, example:

```
docker run -d -p 1411:1401 -p 2785:2775 -p 8990:8990 --name jasmin_02 jookies/
↪jasmin:latest
docker run -d -p 1421:1401 -p 2795:2775 -p 9000:8990 --name jasmin_03 jookies/
↪jasmin:latest
docker run -d -p 1431:1401 -p 2805:2775 -p 9010:8990 --name jasmin_04 jookies/
↪jasmin:latest
```

You should have the container running by typing the following:

```
# docker ps
CONTAINER ID  IMAGE                    COMMAND                CREATED         STATUS
↪       PORTS                                                                  NAMES
0a2fafbe60d0  jookies/jasmin:latest    "/docker-entrypoint.   43 minutes ago  Up 41
↪minutes  0.0.0.0:1401->1401/tcp, 0.0.0.0:2775->2775/tcp, 0.0.0.0:8990->8990/tcp
↪jasmin_01
```

And in order to control the container **jasmin_01**, use:

```
docker stop jasmin_01
docker start jasmin_01
```

It's possible to access log files located in **/var/log/jasmin** inside the container by mounting it as a shared folder:

```
docker run -d -v /home/user/jasmin_logs:/var/log/jasmin --name jasmin_100 jookies/
↪jasmin:latest
```

3.3.6 Sending your first SMS

For the really impatient, if you want to give Jasmin a whirl right now and send your first SMS, you'll have to connect to *Management CLI overview* and setup a connection to your SMS-C, let's **assume** you have the following SMPP connection parameters as provided from your partner:

Table 3.1: Basic SMPP connection parameters

Paramater	Description	Value
Host	Host of remote SMS-C	172.16.10.67
Port	SMPP port on remote SMS-C	2775
Username	Authentication username	smppclient1
Password	Authentication password	password
Throughput	Maximum sent SMS/second	110

Note: In the next sections we'll be heavily using jCli console, if you feel lost, please refer to *Management CLI overview* for detailed information.

1. Adding SMPP connection

Connect to jCli console through telnet (**telnet 127.0.0.1 8990**) using **jcliadmin/jclipwd** default authentication parameters and add a new connector with an *CID=DEMO_CONNECTOR*:

```
Authentication required.

Username: jcliadmin
Password:
Welcome to Jasmin console
Type help or ? to list commands.

Session ref: 2
jcli : smppccm -a
> cid DEMO_CONNECTOR
> host 172.16.10.67
> port 2775
> username smppclient1
> password password
> submit_throughput 110
> ok
Successfully added connector [DEMO_CONNECTOR]
```

2. Starting the connector

Let's start the newly added connector:

```
jcli : smppccm -1 DEMO_CONNECTOR
Successfully started connector id:DEMO_CONNECTOR
```

You can check if the connector is bound to your provider by checking its log file (default to /var/log/jasmin/default-DEMO_CONNECTOR.log) or through jCli console:

```
jcli : smppccm --list
#Connector id                    Service Session          Starts Stops
#DEMO_CONNECTOR                  started BOUND_TRX        1      0
Total connectors: 1
```

3. Configure simple route

We'll configure a default route to send all SMS through our newly created DEMO_CONNECTOR:

```
jcli : mtrouter -a
Adding a new MT Route: (ok: save, ko: exit)
> type defaultroute
jasmin.routing.Routes.DefaultRoute arguments:
connector
> connector smppc(DEMO_CONNECTOR)
> rate 0.00
> ok
Successfully added MTRoute [DefaultRoute] with order:0
```

4. Create a user

In order to use Jasmin's HTTP API to send SMS messages, you have to get a valid user account, that's what we're going to do below.

First we have to create a group to put the new user in:

```
jcli : group -a
    Adding a new Group: (ok: save, ko: exit)
    > gid foogroup
    > ok
    Successfully added Group [foogroup]
```

And then create the new user:

```
jcli : user -a
Adding a new User: (ok: save, ko: exit)
> username foo
> password bar
> gid foogroup
> uid foo
> ok
Successfully added User [foo] to Group [foogroup]
```

5. Send SMS

Sending outbound SMS (MT) is simply done through Jasmin's HTTP API (refer to *HTTP API* for detailed information about sending and receiving SMS and receipts):

```
http://127.0.0.1:1401/send?username=foo&password=bar&to=06222172&content=hello
```

Calling the above url from any brower will send an SMS to **06222172** with **hello** content, if you receive a response like the below example it means your SMS is accepted for delivery:

```
Success "9ab2867c-96ce-4405-b890-8d35d52c8e01"
```

For more troubleshooting about message delivery, you can check details in related log files in **/var/log/jasmin**:

Table 3.2: Messaging related log files

Log filename	Description
messages.log	Information about queued, rejected, received and sent messages
default-DEMO_CONNECTOR.log	The SMPP connector log file

3.4 RESTful API

The RESTful API allows developers to expand and build their apps on Jasmin. The API makes it easy to send messages to one or many destinations, check balance and routing, as well as *enabling bulk messaging*.

This API is built on the Falcon web framework and relying on a standard WSGI architecture, this makes it simple and scalable.

If you need to use a stateful tcp protocol (**SMPP v3.4**), please refer to *SMPP Server API*.

SMS Messages can be transmitted using the RESTful api, the following requirements must be met to enable the service:

- You need a Jasmin user account
- You need sufficient credit on your Jasmin user account

3.4.1 Installation

The RESTful API's made available starting from **v0.9rc16**, it can be launched as a system service, so simply start it by typing:

```
sudo systemctl start jasmin-restapi
```

Note: The RESTful API works on Ubuntu16.04 and CentOS/RHEL 7.x out of the box, some requirements may be installed manually if you are using older Ubuntu distributions.

If you are not using rpm/deb packages to install Jasmin then that systemd service may not be installed on your system, you still can launch the RESTful API manually:

```
celery -A jasmin.protocols.rest.tasks worker -l info -c 4 --autoscale=10,3
twistd -n --pidfile=/tmp/twistd-web-restapi.pid web --wsgi=jasmin.protocols.rest.api
```

Configuration file for Celery and the Web server can be found in **/etc/jasmin/rest-api.py.conf**.

Note: You may also use any other WSGI server for better performance, eg: gunicorn with parallel workers ...

Services

The Services resource represents all web services currently available via Jasmin's RESTful API.

Table 3.3: RESTful services

Method	Service	Description / Notes
POST	*/secure/send*	Send a single message to one destination address.
POST	*/secure/sendbatch*	Send multiple messages to one or more destination addresses.
GET	*/secure/balance*	Get user account's balance and quota.
GET	*/secure/rate*	Check a route and it's rate.
GET	*/ping*	A simple check to ensure this is a Jasmin API.

3.4.2 Authentication

Services having the **/secure/** path (such as *Send a single message* and *Route check*) require authentication using Basic Auth which transmits Jasmin account credentials as username/password pairs, encoded using base64.

Example:

```
curl -X GET -H 'Authorization: Basic Zm9vOmJhcg==' http://127.0.0.1:8080/secure/
↪balance
```

We have passed the base64 encoded credentials through the **Authorization** header, '*Zm9vOmJhcg==*' is the encoded username:password pair ('*foo:bar*'), you can use any tool to base64 encode/decode.

If wrong or no authentication credentials are provided, a **401 Unauthorized** error will be returned.

3.4.3 Send a single message

Send a single message to one destination address.

Definition:

```
http://<jasmin host>:<rest api port>/secure/send
```

Parameters are the same as *the old http api*.

Examples:

```
curl -X POST -H 'Authorization: Basic Zm9vOmJhcg==' -d '{
  "to": 19012233451,
  "from": "Jookies",
  "content": "Hello",
  "dlr": "yes",
  "dlr-url": "http://192.168.202.54/dlr_receiver.php",
  "dlr-level": 3
}' http://127.0.0.1:8080/secure/send
```

Note: Do not include **username** and **password** in the parameters, they are already provided through the *Authorization header*.

Result Format:

```
{"data": "Success \"c723d42a-c3ee-452c-940b-3d8e8b944868"}
```

If successful, response header HTTP status code will be **200 OK** and and the message will be sent, the *message id* will be returned in **data**.

3.4.4 Send multiple messages

Send multiple messages to one or more destination addresses.

Definition:

```
http://<jasmin host>:<rest api port>/secure/sendbatch
```

Example of sending same message to multiple destinations:

```
curl -X POST -H 'Authorization: Basic Zm9vOmJhcg==' -d '{
  "messages": [
    {
      "to": [
        "33333331",
        "33333332",
        "33333333"
      ],
      "content": "Same content goes to 3 numbers"
    }
  ]
}' http://127.0.0.1:8080/secure/sendbatch
```

Result Format:

```
{"data": {"batchId": "af268b6b-1ace-4413-b9d2-529f4942fd9e", "messageCount": 3}}
```

If successful, response header HTTP status code will be **200 OK** and and the messages will be sent, the *batch id* and total *message count* will be returned in **data**.

Table 3.4: POST /secure/sendbatch json parameters

Parameter	Example(s)	Presence	Description / Notes
messages	[{"to": 1, "content": "hi"}, {"to": 2, "content": "hello"}]	Mandatory	A Json list of messages, every message contains the */secure/send* parameters
globals	{"from": "Jookies"}	Optional	May contain any global message parameter, c.f. *examples*
batch_config	{"callback_url": "http://127.0.0.1:7877", "schedule_at": "2017-11-15 09:00:00"}	Optional	May contain the following parameters: callback_url or/and errback_url (used for batch tracking in real time c.f. *examples*), schedule_at (used for scheduling sendouts c.f. *examples*).

Note: The Rest API server has an advanced QoS control to throttle pushing messages back to Jasmin, you may fine-tune it through the **http_throughput_per_worker** and **smart_qos** parameters.

3.4.5 Send binary messages

Sending binary messages can be done using *single* or *batch* messaging APIs.

It's made possible by replacing the **content** parameter by the **hex_content**, the latter shall contain your binary data hex value.

Example of sending a message with coding=8:

```
curl -X POST -H 'Authorization: Basic Zm9vOmJhcg==' -d '{
  "to": 19012233451,
  "from": "Jookies",
  "coding": 8,
  "hex_content": "0623063106460628"
}' http://127.0.0.1:8080/secure/send
```

The **hex_content** used in the above example is the UTF16BE encoding of arabic word “” (‘x06x23x06x31x06x46x06x28’).

Same goes for sending batches with binary data:

```
curl -X POST -H 'Authorization: Basic Zm9vOmJhcg==' -d '{
  "messages": [
    {
      "to": [
        "33333331",
        "33333332",
        "33333333"
      ],
      "hex_content": "0623063106460628"
    }
  ]
}' http://127.0.0.1:8080/secure/sendbatch
```

Usage examples:

The ref:*parameter <restapi-POST_sendbatch_params>* listed above can be used in many ways to setup a sendout batch, we’re going to list some use cases to show the flexibility of these parameters:

Example 1, send different messages to different numbers::

```
{
  "messages": [
    {
      "from": "Brand1",
      "to": [
        "55555551",
        "55555552",
        "55555553"
      ],
      "content": "Message 1 goes to 3 numbers"
    },
    {
      "from": "Brand2",
      "to": [
        "33333331",
        "33333332",
        "33333333"
      ],
      "content": "Message 2 goes to 3 numbers"
    },
    {
      "from": "Brand2",
      "to": "7777771",
      "content": "Message 3 goes to 1 number"
    }
```

```
  ]
}
```

Example 2, using global vars:

From the previous Example (#1) we used the same "from" address for two different messages (**"from": "Brand2"**), in the below example we're going to make the "from" a global variable, and we are asking for level3 dlr for all sendouts:

```
{
  "globals" : {
    "from": "Brand2",
    "dlr-level": 3,
    "dlr": "yes",
    "dlr-url": "http://some.fancy/url"
  }
  "messages": [
    {
      "from": "Brand1",
      "to": [
        "55555551",
        "55555552",
        "55555553"
      ],
      "content": "Message 1 goes to 3 numbers"
    },
    {
      "to": [
        "33333331",
        "33333332",
        "33333333"
      ],
      "content": "Message 2 goes to 3 numbers"
    },
    {
      "to": "7777771",
      "content": "Message 3 goes to 1 number"
    }
  ]
}
```

So, **globals** are vars to be inherited in **messages**, we still can force a *local* value in some messages like the **"from": "Brand1"** in the above example.

Example 3, using callbacks:

As *explained*, Jasmin is enqueuing a sendout batch everytime you call **/secure/sendbatch**, the batch job will run and call Jasmin's http api to deliver the messages, since this is running in background you can ask for success or/and error callbacks to follow the batch progress.

```
{
  "batch_config": {
    "callback_url": "http://127.0.0.1:7877/successful_batch",
    "errback_url": "http://127.0.0.1:7877/errored_batch"
      },
  "messages": [
    {
      "to": [
        "55555551",
        "55555552",
```

```
        "55555553"
      ],
      "content": "Hello world !"
    },
    {
      "to": "7777771",
      "content": "Holà !"
    }
  ]
}
```

About callbacks:

The RESTful api is a wrapper around Jasmin's http api, it relies on Celery task queue to process long running batches.

When you launch a batch, the api will enqueue the sendouts through Celery and return a **batchId**, that's the Celery task id.

Since the batch will be executed in background, the API provides a convenient way to follow its progression through two different callbacks passed inside the batch parameters:

```
{
  "batch_config": {
    "callback_url": "http://127.0.0.1:7877/successful_batch",
    "errback_url": "http://127.0.0.1:7877/errored_batch"
      },
  "messages": [
    {
      "to": "7777771",
      "content": "Holà !"
    }
  ]
}
```

The **callback_url** will be called (GET) everytime a message is successfuly sent, otherwise the **errback_url** is called.

In both callbacks the following parameters are passed:

Table 3.5: Batch callbacks parameters

Parameter	Example(s)	Description / Notes
batchId	50a4581a-6e46-48a4-b617-bbefe7faa3dc	The batch id
to	1234567890	The **to** parameter identifying the destination number
status	1	1 or 0, indicates the status of a message sendout
statusText	Success "07033084-5cfd-4812-90a4-e4d24ffb6e3d"	Extra text for the **status**

About batch scheduling:

It is possible to schedule the launch of a batch, the api will enqueue the sendouts through Celery and return a **batchId** while deferring message deliveries to the scheduled date & time.

```
{
  "batch_config": {
    "schedule_at": "2017-11-15 09:00:00"
      },
  "messages": [
    {
      "to": "7777771",
      "content": "Good morning !"
    }
  ]
}
```

The above batch will be scheduled for the 15th of November 2017 at 9am, the Rest API will consider it's local server time to make the delivery, so please make sure it's accurate to whatever timezone you're in.

It's possible to use another **schedule_at** format:

```
{
  "batch_config": {
    "schedule_at": "86400s"
      },
  "messages": [
    {
      "to": "7777771",
      "content": "Good morning !"
    }
  ]
}
```

The above batch will be scheduled for delivery in 1 day from now (86400 seconds = 1 day).

3.4.6 Balance check

Get user account's balance and quota.

Definition:

```
http://<jasmin host>:<rest api port>/secure/balance
```

Parameters are the same as *the old http api*.

Examples:

```
curl -X GET -H 'Authorization: Basic Zm9vOmJhcg==' http://127.0.0.1:8080/secure/
↪balance
```

Note: Do not include **username** and **password** in the parameters, they are already provided through the *Authorization header*.

Result Format:

```
{"data": {"balance": "10.23", "sms_count": "ND"}}
```

If successful, response header HTTP status code will be **200 OK**, the *balance* and the *sms count* will be returned in **data**.

3.4.7 Route check

Check a route and it's rate.

Definition:

```
http://<jasmin host>:<rest api port>/secure/rate
```

Parameters are the same as *the old http api*.

Examples:

```
curl -X GET -H 'Authorization: Basic Zm9vOmJhcg==' http://127.0.0.1:8080/secure/rate?
↪to=19012233451
```

Note: Do not include **username** and **password** in the parameters, they are already provided through the *Authorization header*.

Result Format:

```
{"data": {"submit_sm_count": 1, "unit_rate": 0.02}}
```

If successful, response header HTTP status code will be **200 OK**, the *message rate* and "pdu count" will be returned in **data**.

3.4.8 Ping

A simple check to ensure this is a responsive Jasmin API, it is used by third party apps like Web campaigners, cluster service checks, etc ..

Definition:

```
http://<jasmin host>:<rest api port>/ping
```

Examples:

```
curl -X GET http://127.0.0.1:8080/ping
```

Result Format:

```
{"data": "Jasmin/PONG"}
```

If successful, response header HTTP status code will be **200 OK** and a static "Jasmin/PONG" value in **data**.

3.5 HTTP API

This document is targeted at software designers/programmers wishing to integrate SMS messaging as a function into their applications using HTTP protocol, e.g. in connection with WEB-server, unified messaging, information services etc..

If you need to use a stateful tcp protocol (**SMPP v3.4**), please refer to *SMPP Server API*.

SMS Messages can be transmitted using HTTP protocol, the following requirements must be met to enable the service:

- You need a Jasmin user account

- You need sufficient credit on your Jasmin user account[1]

Note: The ABCs:

- **MT** is referred to Mobile Terminated, a SMS-MT is an SMS sent to mobile
- **MO** is referred to Mobile Originated, a SMS-MO is an SMS sent from mobile

---

3.5.1 Features

The ja-http API allows you to:

- Send and receive SMS through Jasmin's connectors,
- Receive http callbacks for delivery notification (*receipts*) when SMS-MT is received (or not) on mobile station,
- Send and receive long (more than 160 characters) SMS, unicode/binary content and receive http callbacks when a mobile station send you a SMS-MO.
- Check your balance status,
- Check a message rate price before sending it.

3.5.2 Sending SMS-MT

In order to deliver **SMS-MT** messages, Data is transferred using **HTTP GET/POST** requests. The Jasmin gateway accepts requests at the following URL:

http://127.0.0.1:1401/send

Note: Host `127.0.0.1` and port `1401` are default values and configurable in `/etc/jasmin/jasmin.cfg`, see *jasmin.cfg / http-api*.

This guide will help understand how the API works and provide *Examples* for sending SMS-MT.

HTTP request parameters

When calling Jasmin's URL from an application, the below parameters must be passed (at least mandatory ones), the api will return a message id on success, see *HTTP response*.

[1] *Billing*

Table 3.6: ja-http sending SMS parameters

Parameter	Value / Pattern	Example(s)	Presence	Description / Notes
to	Destination address	20203050	Mandatory	Destination address, only one address is supported per request
from	Originating address	20203050, Jasmin	Optional	Originating address, In case rewriting of the sender's address is supported or permitted by the SMS-C used to transmit the message, this number is transmitted as the originating address
coding	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 13 or 14	1	Optional	Sets the Data Coding Scheme bits, default is 0, accepts values all allowed values in SMPP protocol[1]
username	Text (30 char. max)	jasmin_user	Mandatory	Username for Jasmin user account.
password	Text (30 char. max)	jasmin_pass	Mandatory	Password for Jasmin user account.
priority	0, 1, 2 or 3	2	Optional	Default is 0 (lowest priority)
sdt	String	000000000100000R (send in 1 minute)	Optional	Specifies the scheduled delivery time at which the message delivery should be first attempted, default is value is None (message will take SMSC's default). Supports Absolute and Relative Times per SMPP v3.4 Issue 1.2
validity-period	Integer	1440	Optional	Message validity (minutes) to be passed to SMSC, default is value is None (message will take SMSC's default)
dlr	yes or no	yes	Optional	Default is no (no DLR will be tracked)
dlr-url	HTTP(s) URL	http://host/dlr.php	Mandatory *if dlr*	If a DLR is requested (dlr = 'yes'), dlr-url MUST be set, if not, dlr value is reconsidered as 'no'
dlr-level	1, 2 or 3	2	Mandatory *if dlr*	1: SMS-C level, 2: Terminal level, 3: Both
dlr-method	GET or POST	GET	Mandatory *if dlr*	DLR is transmitted through http to a third party application using GET or POST method.
tags	Text	1,702,9901	Optional	Will tag the routable to help interceptor or router enable specific business logics.
content	Text	Hello world !	Mandatory if **hex-content** not defined	Content to be sent
hex-content	Binary hex value	06230631064600[illegible]28	Mandatory if **content** not defined	Binary to be sent

HTTP response

When the request is validated, a SubmitSM PDU is set up with the provided request parameters and sent to the routed connector through a AMQP queue, a queued message-id is returned:

```
Success "07033084-5cfd-4812-90a4-e4d24ffb6e3d"
```

Otherwise, an error is returned:

```
Error "No route found"
```

Table 3.7: HTTP response code details

HTTP Code	HTTP Body	Meaning
200	Success “07033084-5cfd-4812-90a4-e4d24ffb6e3d”	Message is successfully queued, messaged-id is returned
400	Error “Mandatory arguments not found, please refer to the HTTPAPI specifications.”	Request parameters validation error
400	Error “Argument _ is unknown.”	Request parameters validation error
400	Error “Argument _ has an invalid value: _.”	Request parameters validation error
400	Error “Mandatory argument _ is not found.”	Request parameters validation error
400	*dynamic messages*	Credentials validation error, c.f. *User credentials*
403	Error “Authentication failure for username:_”	Authentication error
403	Error “Authorization failed for username:_”	Credentials validation error, c.f. *User credentials*
403	Error “Cannot charge submit_sm, check RouterPB log file for details”	User charging error
412	Error “No route found”	Message routing error
500	Error “Cannot send submit_sm, check SMPPClientManagerPB log file for details”	Fallback error, checking log file will provide better details

Examples

Here is an example of how to send simple GSM 03.38 messages:

```
# Python example
# http://jasminsms.com
import urllib2
import urllib

baseParams = {'username':'foo', 'password':'bar', 'to':'+336222172', 'content':'Hello
↪'}

# Send an SMS-MT with minimal parameters
urllib2.urlopen("http://127.0.0.1:1401/send?%s" % urllib.urlencode(baseParams)).read()

# Send an SMS-MT with defined originating address
baseParams['from'] = 'Jasmin GW'
urllib2.urlopen("http://127.0.0.1:1401/send?%s" % urllib.urlencode(baseParams)).read()
```

Here is an example of how to request acknowledgement when sending a SMS:

```
# Python example
# http://jasminsms.com
import urllib2
import urllib

# Send an SMS-MT and request terminal level acknowledgement callback to http://
↪myserver/acknowledgement
params = {'username':'foo', 'password':'bar', 'to':'+336222172', 'content':'Hello',
```

```
          'dlr-url':'http://myserver/acknowledgement', 'dlr-level':2}
urllib2.urlopen("http://127.0.0.1:1401/send?%s" % urllib.urlencode(params)).read()
```

And more use cases for sending long, UCS2 (UTF16) and binary messages:

```
# Python example
# http://jasminsms.com
import urllib2
import urllib

baseParams = {'username':'foo', 'password':'bar', 'to':'+336222172', 'content':'Hello
↪'}

# Sending long content (more than 160 chars):
baseParams['content'] = 'Very long message .....................................
↪.....................................................................................
↪.................................................'
urllib2.urlopen("http://127.0.0.1:1401/send?%s" % urllib.urlencode(baseParams)).read()

# Sending UCS2 (UTF-16) arabic content
baseParams['content'] = '\x06\x23\x06\x31\x06\x46\x06\x28'
baseParams['coding'] = 8
urllib2.urlopen("http://127.0.0.1:1401/send?%s" % urllib.urlencode(baseParams)).read()

# Sending UCS2 (UTF-16) arabic binary content
baseParams['hex-content'] = '0623063106460628'
baseParams['coding'] = 8
urllib2.urlopen("http://127.0.0.1:1401/send?%s" % urllib.urlencode(baseParams)).read()
```

In PHP:

```
<?php
// Sending simple message using PHP
// http://jasminsms.com

$baseurl = 'http://127.0.0.1:1401/send'

$params = '?username=foo'
$params.= '&password=bar'
$params.= '&to='.urlencode('+336222172')
$params.= '&content='.urlencode('Hello world !')

$response = file_get_contents($baseurl.$params);
?>
```

In Ruby:

```
# Sending simple message using Ruby
# http://jasminsms.com

require 'net/http'

uri = URI('http://127.0.0.1:1401/send')
params = { :username => 'foo', :password => 'bar',
           :to => '+336222172', :content => 'Hello world' }
uri.query = URI.encode_www_form(params)

response = Net::HTTP.get_response(uri)
```

jasmin.cfg / http-api

The **jasmin.cfg** file *(INI format, located in /etc/jasmin)* contain a section called **http-api** where all ja-http API related config elements are:

```
[http-api]
bind          = 0.0.0.0
port          = 1401

long_content_max_parts = 5
# Splitting long content can be made through SAR options or UDH
# Possible values are: sar and udh
long_content_split = udh

access_log          = /var/log/jasmin/http-access.log
log_level           = INFO
log_file            = /var/log/jasmin/http-api.log
log_format          = %(asctime)s %(levelname)-8s %(process)d %(message)s
log_date_format     = %Y-%m-%d %H:%M:%S
```

Table 3.8: [http-api] configuration section

Element	Default	Description
bind	0.0.0.0	The HTTP API listener will only bind to this specified address, given 0.0.0.0 the listener will bind on all interfaces.
port	1401	The binding TCP port.
long_content_max_parts	5	If the message to be sent is to be split into several parts. This is the maximum number of individual SMS-MT messages that can be used.
long_content_split	udh	Splitting method: 'udh': Will split using 6-byte long User Data Header, 'sar': Will split using sar_total_segments, sar_segment_seqnum, and sar_msg_ref_num options.
access_log	/var/log/jasmin/http-access.log	Where to log all http requests (and errors).
log_*		Python's logging module configuration.

3.5.3 Receiving DLR

When requested through dlr-* fields when *Sending SMS-MT*, a delivery receipt (**DLR**) will be sent back to the application url (set in **dlr-url**) through **HTTP GET/POST** depending on **dlr-method**.

The receiving end point must reply back using a "**200 OK**" status header **and** a body containing an **acknowledgement** of receiving the DLR, if one or both of these conditions are not met, the *DLRThrower service* will consider reshipment of the same message if **config/dlr-thrower/max_retries** is not reached (see *jasmin.cfg / dlr-thrower*).

In order to acknowledge DLR receipt, the receiving end point must reply back with **exactly** the following html body content:

```
ACK/Jasmin
```

Note: It is very important to acknowledge back each received DLR, this will prevent to receive the same message many times, c.f. *Processing* for details

Note: Reshipment of a message will be delayed for **config/dlr-thrower/retry_delay** seconds (see *jasmin.cfg / dlr-thrower*).

HTTP Parameters for a level 1 DLR

The following parameters are sent to the receiving end point (at dlr-url) when the DLR's dlr-level is set to 1 (SMS-C level only)

Table 3.9: ja-http parameters for a level 1 DLR

Parameter	Value / Pattern	Example(s)	Presence	Description / Notes
id	Universally Unique IDentifier (UUID)	16fd2706-8baf-433b-82eb-8c7fada847da	Always	Internal Jasmin's gateway message id used for tracking messages
message_status	ESME_* SMPP Command status	ESME_ROK, ESME_RINVNUMDESTS	Always	The delivery status
level	1	1	Always	This is a static value indicating the dlr-level originally requested

HTTP Parameters for a level 2 or 3 DLR

The following parameters are sent to the receiving end point (at dlr-url) when DLR's dlr-level is set to 2 or 3 (Terminal level or all levels)

Table 3.10: ja-http parameters for a level 2 or 3 DLR

Parameter	Value / Pattern	Example(s)	Presence	Description / Notes
id	Universally Unique IDentifier (UUID)	16fd2706-8baf-433b-82eb-8c7fada847da	Always	Internal Jasmin's gateway message id used for tracking messages
id_smsc	Integer	2567	Always	Message id returned from the SMS-C
message_status	ESME_* SMPP Command status	ESME_ROK, ESME_RINVNUMDESTS	Always	The delivery status
level	1	1	Always	This is a static value indicating the dlr-level originally requested
sub_date	Date & time format: YYMMDDhhmm	1311022338	Optional	The time and date at which the short message was submitted
done_date	Date & time format: YYMMDDhhmm	1311022338	Optional	The time and date at which the short message reached it's final state
sub	Integer	1	Optional	Number of short messages originally submitted. This is only relevant when the original message was submitted to a distribution list.The value is padded with leading zeros if necessary
dlvrd	Integer	1	Optional	Number of short messages delivered. This is only relevant where the original message was submitted to a distribution list.The value is padded with leading zeros if necessary
err	Integer	0	Optional	Where appropriate this may hold a Network specific error code or an SMSC error code for the attempted delivery of the message
text	Text (20 char. max)	Hello foo bar	Optional	The first 20 characters of the short message

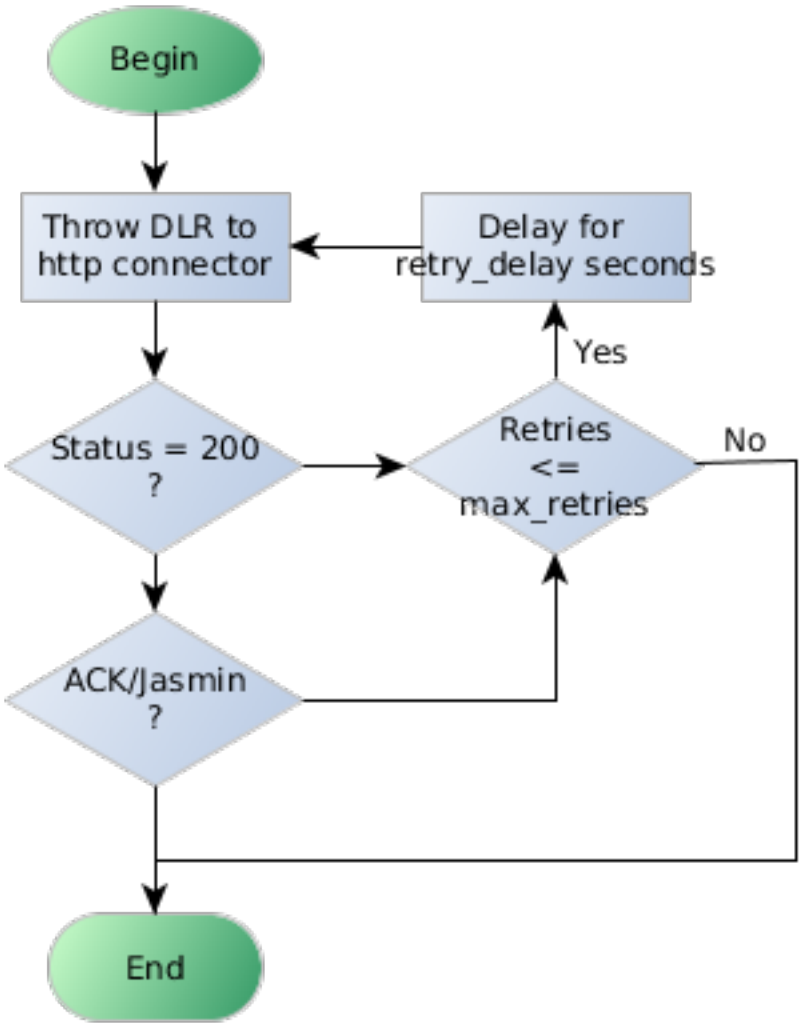
Processing

The flowchart below describes how dlr delivery and retrying policy is done inside DLRThrower service:

jasmin.cfg / dlr-thrower

The **jasmin.cfg** file *(INI format, located in /etc/jasmin)* contain a section called **deliversm-thrower** where all DLRThrower service related config elements are:

```
[dlr-thrower]
http_timeout        = 30
retry_delay         = 30
max_retries         = 3
log_level           = INFO
```


Begin
Throw DLR to http connector
Delay for retry_delay seconds
Yes
Status = 200 ?
Retries <= max_retries
No
ACK/Jasmin ?
End

```
log_file            = /var/log/jasmin/dlr-thrower.log
log_format          = %(asctime)s %(levelname)-8s %(process)d %(message)s
log_date_format     = %Y-%m-%d %H:%M:%S
```

Table 3.11: [http-api] configuration section

Element	Default	Description
http_timeout	30	Sets socket timeout in seconds for outgoing client http connections.
retry_delay	30	Define how many seconds should pass within the queuing system for retrying a failed throw.
max_retries	3	Define how many retries should be performed for failing throws of DLR.
log_*		Python's logging module configuration.

3.5.4 Receiving SMS-MO

SMS-MO incoming messages (**M**obile **O**riginated) are forwarded by Jasmin to defined URLs using simple **HTTP GET/POST**, the forwarding is made by *deliverSmHttpThrower* service, and the URL of the receiving endpoint is selected through a route checking process (c.f. *The message router*).

Receiving endpoint is a third party application which acts on the messages received and potentially generates replies, (*HTTP Client connector manager* for more details about HTTP Client connector management).

The parameters below are transmitted for each SMS-MO, the receiving end point must provide an url (set in **jasminApi.HttpConnector.baseurl**) and parse the below parameters using GET or POST method (depends on **jasminApi.HttpConnector.method**).

The receiving end point must reply back using a "**200 OK**" status header **and** a body containing an **acknowledgement** of receiving the SMS-MO, if one or both of these conditions are not met, the *deliverSmHttpThrower service* will consider reshipment of the same message if **config/deliversm-thrower/max_retries** is not reached, (see *jasmin.cfg / deliversm-thrower*).

In order to acknowledge SMS-MO receipt, the receiving end point must reply back with **exactly** the following html body content:

```
ACK/Jasmin
```

Note: It is very important to acknowledge back each received SMS-MO, this will prevent to receive the same message many times, c.f. *Processing* for details

Note: Reshipment of a message will be delayed for **config/deliversm-thrower/retry_delay** seconds (see *jasmin.cfg / deliversm-thrower*).

HTTP Parameters

When receiving an URL call from Jasmin's *deliverSmHttpThrower service*, the below parameters are delivered (at least *Always* present ones).

Table 3.12: ja-http receiving SMS parameters

Parameter	Value / Pattern	Example(s)	Presence	Description / Notes
id	Universally Unique IDentifier (UUID)	16fd2706-8baf-433b-82eb-8c7fada847da	Always	Internal Jasmin's gateway message id
from	Originating address	+21620203060, 20203060, Jasmin	Always	Originating address
to	Destination address	+21620203060, 20203060, Jasmin	Always	Destination address, only one address is supported per request
origin-connector	Alphanumeric id	23, bcd, MTN, clickatell, beepsend	Always	Jasmin http connector id
priority	1, 2 or 3	2	Optional	Default is 1 (lowest priority)
coding	Numeric	8	Optional	Default is 0, accepts values all allowed values in SMPP protocol[2]
validity	YYYY-MM-DD hh:mm:ss	2013-07-16 00-46:54	Optional	The validity period parameter indicates the Jasmin GW expiration time, after which the message should be discarded if not delivered to the destination
content	Text	Hello world !	Always	Content of the message
binary	Hexlified binary content	062A063062A	Always	Content of the message in binary hexlified form

Note: When receiving multiple parts of a long SMS-MO, *deliverSmHttpThrower service* will concatenate the content of all the parts and then throw one http call with concatenated *content*.

Processing

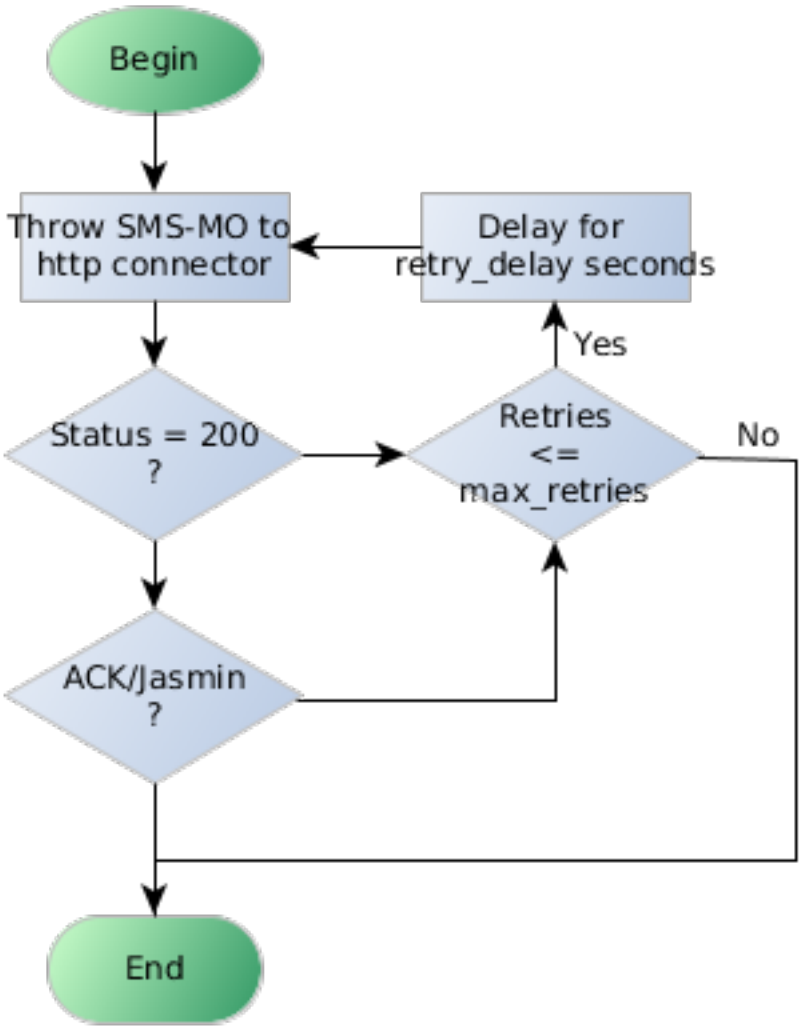
The flowchart below describes how message delivery and retrying policy are done inside *deliverSmHttpThrower* service:

jasmin.cfg / deliversm-thrower

The **jasmin.cfg** file *(INI format, located in /etc/jasmin)* contain a section called **deliversm-thrower** where all deliverSmHttpThrower service related config elements are:

```
[deliversm-thrower]
http_timeout       = 30
retry_delay        = 30
max_retries        = 3
log_level          = INFO
log_file           = /var/log/jasmin/deliversm-thrower.log
```

[2]


Begin
Throw SMS-MO to http connector
Delay for retry_delay seconds
Yes
Status = 200 ?
Retries <= max_retries
No
ACK/Jasmin ?
End

```
log_format          = %(asctime)s %(levelname)-8s %(process)d %(message)s
log_date_format     = %Y-%m-%d %H:%M:%S
```

Table 3.13: [http-api] configuration section

Element	Default	Description
http_timeout	30	Sets socket timeout in seconds for outgoing client http connections.
retry_delay	30	Define how many seconds should pass within the queuing system for retrying a failed throw.
max_retries	3	Define how many retries should be performed for failing throws of SMS-MO.
log_*		Python's logging module configuration.

3.5.5 Checking account balance

In order to check user account balance and quotas, user may request a **HTTP GET/POST** from the following URL:

http://127.0.0.1:1401/balance

Note: Host `127.0.0.1` and port `1401` are default values and configurable in `/etc/jasmin/jasmin.cfg`, see *jasmin.cfg / http-api*.

HTTP request parameters

Table 3.14: ja-http balance request parameters

Parameter	Value / Pattern	Example(s)	Presence	Description / Notes
username	Text (30 char. max)	jasmin_user	Mandatory	Username for Jasmin user account.
password	Text (30 char. max)	jasmin_pass	Mandatory	Password for Jasmin user account.

HTTP response

Successful response:

```
{"balance": 100.0, "sms_count": "ND"}
```

Otherwise, an error is returned.

Examples

Here is an example of how to check balance:

```
# Python example
# http://jasminsms.com
import urllib2
import urllib
import json

# Check user balance
```

```
params = {'username':'foo', 'password':'bar'}
response = urllib2.urlopen("http://127.0.0.1:1401/balance?%s" % urllib.
↪urlencode(params)).read()
response = json.loads(response)

print 'Balance:', response['balance']
print 'SMS Count:', response['sms_count']

#Balance: 100.0
#SMS Count: ND
```

3.5.6 Checking rate price

It is possible to ask Jasmin's HTTPAPI for a message rate price before sending it, the request will lookup the route to be considered for the message and will provide the rate price if defined.

Request is done through **HTTP GET/POST** to the following URL:

http://127.0.0.1:1401/rate

Note: Host `127.0.0.1` and port `1401` are default values and configurable in `/etc/jasmin/jasmin.cfg`, see *jasmin.cfg / http-api*.

HTTP request parameters

Table 3.15: ja-http rate request parameters

Parameter	Value / Pattern	Example(s)	Presence	Description / Notes
to	Destination address	20203050	Mandatory	Destination address, only one address is supported per request
from	Originating address	20203050,Jasmin	Optional	Originating address, In case rewriting of the sender's address is supported or permitted by the SMS-C used to transmit the message, this number is transmitted as the originating address
coding	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 13 or 14	1	Optional	Sets the Data Coding Scheme bits, default is 0, accepts values all allowed values in SMPP protocol[1]
username	Text (30 char. max)	jasmin_user	Mandatory	Username for Jasmin user account.
password	Text (30 char. max)	jasmin_pass	Mandatory	Password for Jasmin user account.
content	Text	Hello world !	Optional	Content to be sent

HTTP response

Successful response:

```
{"submit_sm_count": 2, "unit_rate": 2.8}
```

Where **submit_sm_count** is the number of message units if the **content** is longer than 160 characters, **content** parameter is optional for requesting rate price.

Otherwise, an error is returned.

Otherwise, an error is returned:

```
Error "No route found"
```

Examples

Here is an example of how to check rate price:

```
# Python example
# http://jasminsms.com
import urllib2
import urllib
import json

# Check message rate price
params = {'username':'foo', 'password':'bar', 'to': '06222172'}
response = urllib2.urlopen("http://127.0.0.1:1401/rate?%s" % urllib.
↪urlencode(params)).read()
response = json.loads(response)

print 'Unit rate price:', response['unit_rate']
print 'Units:', response['submit_sm_count']

#Unit rate price: 2.8
#Units: 1
```

Table 3.16: Data coding schemes

Bitmask	**Value**	Meaning
0 0 0 0 0 0 0 0	**0**	SMSC Default Alphabet
0 0 0 0 0 0 0 1	**1**	IA5 (CCITT T.50)/ASCII (ANSI X3.4)
0 0 0 0 0 0 1 0	**2**	Octet unspecified (8-bit binary)
0 0 0 0 0 0 1 1	**3**	Latin 1 (ISO-8859-1)
0 0 0 0 0 1 0 0	**4**	Octet unspecified (8-bit binary)
0 0 0 0 0 1 0 1	**5**	JIS (X 0208-1990)
0 0 0 0 0 1 1 0	**6**	Cyrllic (ISO-8859-5)
0 0 0 0 0 1 1 1	**7**	Latin/Hebrew (ISO-8859-8)
0 0 0 0 1 0 0 0	**8**	UCS2 (ISO/IEC-10646)
0 0 0 0 1 0 0 1	**9**	Pictogram Encoding
0 0 0 0 1 0 1 0	**10**	ISO-2022-JP (Music Codes)
0 0 0 0 1 1 0 1	**13**	Extended Kanji JIS(X 0212-1990)
0 0 0 0 1 1 1 0	**14**	KS C 5601

3.6 SMPP Server API

This document is targeted at software designers/programmers wishing to integrate SMS messaging through a stateful tcp protocol **SMPP v3.4**, if you feel this does not fit your needs and that you are more "web-service-guy" then you still can try *HTTP API*.

SMS Messages can be transmitted using SMPP protocol, the following requirements must be met to enable the service :

- You need a Jasmin user account
- You need sufficient credit on your Jasmin user account[1]

Note: The ABCs:

- **MT** is referred to Mobile Terminated, a SMS-MT is an SMS sent to mobile
- **MO** is referred to Mobile Originated, a SMS-MO is an SMS sent from mobile

---

3.6.1 Features

The SMPP Server API allows you to send and receive SMS and delivery receipts (DLR) through Jasmin's connectors, send and receive long (more than 160 characters) SMS and unicode/binary content.

jasmin.cfg / smpp-server

The **jasmin.cfg** file *(INI format, located in /etc/jasmin)* contain a section called **smpp-server** where all SMPP Server API related config elements are:

```
[smpp-server]
id              = "smpps_01"
bind            = 0.0.0.0
port            = 2775

sessionInitTimerSecs = 30
enquireLinkTimerSecs = 30
inactivityTimerSecs  = 300
responseTimerSecs    = 60
pduReadTimerSecs     = 30

log_level           = INFO
log_file            = /var/log/jasmin/default-smpps_01.log
log_format          = %(asctime)s %(levelname)-8s %(process)d %(message)s
log_date_format     = %Y-%m-%d %H:%M:%S
```

[1] *Billing*

Table 3.17: [smpp-server] configuration section

Element	Default	Description
id	smpps_01	The SMPP Server id, used to identify the instance in case you use multiple servers per Jasmin process.
bind	0.0.0.0	The SMPP Server API listener will only bind to this specified address, given 0.0.0.0 the listener will bind on all interfaces.
port	2775	The binding TCP port.
sessionInitTimerSecs	30	Protocol tuning parameter: timeout for a bind request.
enquireLinkTimerSecs	30	Protocol tuning parameter: timeout for an enquire_link request.
inactivityTimerSecs	300	Protocol tuning parameter: inactivity timeout.
responseTimerSecs	60	Protocol tuning parameter: global request timeout.
pduReadTimerSecs	30	Protocol tuning parameter: binary pdu ready timeout.
log_*		Python's logging module configuration.

Binding to SMPP Server

Using a proper SMPP Client application (or a Jasmin SMPP Client), the following parameters must be considered:

Table 3.18: SMPP Server binding parameters

Parameter	Value / Pattern	Example(s)	Presence	Description / Notes
system_id	Text (30 char. max)	jasmin_user	Mandatory	Username for Jasmin user account.
password	Text (30 char. max)	jasmin_pass	Mandatory	Password for Jasmin user account.

Supported SMPP PDUs

Jamsin's SMPP Server is supporting the following PDUs:

- bind_transmitter
- bind_transceiver
- bind_receiver
- unbind
- submit_sm
- deliver_sm
- enquire_link

3.7 The message router

The message router is Jasmin's decision making component for routing every type of exchanged message through the gateway:

1. MO Messages (deliver_sm)
2. MT Messages (submit_sm)

The router is provisioned through:

- Perspective broker interface (python programmatic API)
- jCli modules: *MO router manager* and *MT router manager*

Each time a message is requiring a route decision the following process is executed:

3.7.1 Process flow

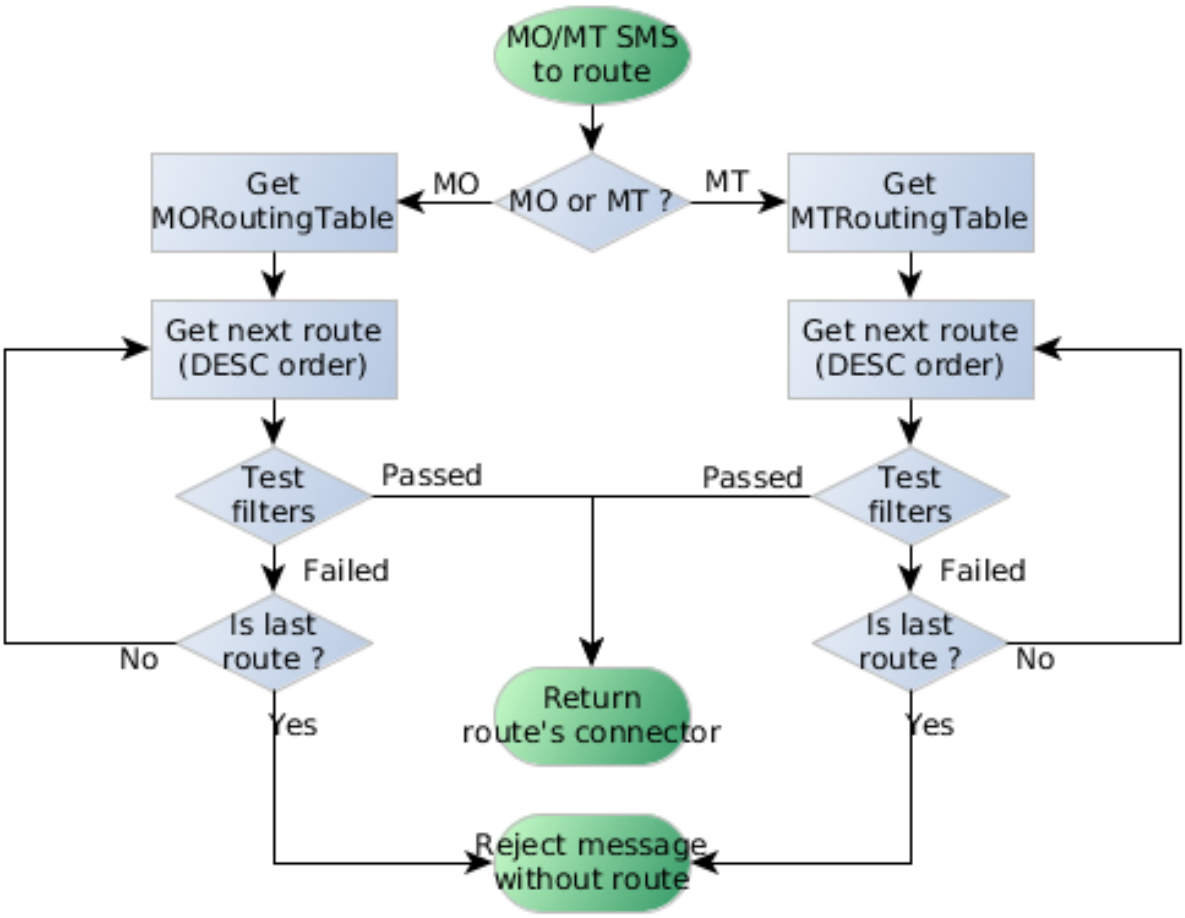


Fig. 3.2: Routing process flow

There's one **MORoutingTable** and one **MTRoutingTable** objects holding respective routes for each direction (MT or MO), these are **Route** objects that hold one or many **Filter** (s) objects and one destination **Connector** (or many connectors in some specific cases, c.f. *Multiple connectors*).

As explained by the above routing process flow figure, for each message and depending on its direction, a routing table is loaded and an iterative testing is run in order to select a final destination connector or to reject (returning no connector) it, routes are selected in descendant order, and their respective filter objects are tested against the **Routable** object (It is an extension of the low-level SMPP PDU object representing a message, more information in *Routable*).

Examples

MO Routing

Having the below MO routing table set through a jCli console session:

```
jcli : morouter -l
#MO Route order   Type                    Connector ID(s)                  Filter(s)
#30               StaticMORoute           http_3
↪<DestinationAddrFilter (dst_addr=^\+33\d+)>
#20               RandomRoundrobinMORoute http_1, http_2
↪<DateIntervalFilter (2015-06-01,2015-08-31)>, <TimeIntervalFilter (08:00:00,
↪18:00:00)>
#0                DefaultRoute            http_def
Total MO Routes: 3
```

The following routing cases are considered:

- MO message is routed to http_3 **if**:
  - Its destination address matches the regular expression "^+33d+"
- MO message is routed to http_1 **OR** http_2 **if**:
  - Its received in summer months (June, July and August) of year 2015 and in working hours interval (8pm to 6am)
- MO message is routed to http_def **if**:
  - None of the above routes are matched (fallback / default route)

MT Routing

Having the below MT routing table set through a jCli console session:

```
jcli : mtrouter -l
#MT Route order   Type                    Rate    Connector ID(s)                ␣
↪Filter(s)
#100              RandomRoundrobinMTRoute 0.00    smpp_1, smpp_2
↪<DestinationAddrFilter (dst_addr=^\+33\d+)>
#91               StaticMTRoute           0.00    smpp_4
↪<GroupFilter (gid=G2)>, <TimeIntervalFilter (08:00:00,18:00:00)>
#90               StaticMTRoute           0.00    smpp_3
↪<GroupFilter (gid=G2)>
Total MT Routes: 3
```

The following routing cases are considered:

- MT message is routed to smpp_1 **OR** smpp_2 **if**:
  - Its destination address matches the regular expression "^+33d+"
- MT message is routed to smpp_4 **if**:
  - Its sent by a user in group G2 and in working hours interval (8pm to 6am)
- MT message is routed to smpp_3 **if**:
  - Its sent by a user in group G2

Note: The route order is very important: if we swap last both routes (#90 and #91) we will run into a shadowing route where all MT messages sent by a user in group G2 will be routed to smpp_3, no matter what time of the day it is.

Note: In this example, there's no DefaultRoute, this will lead to message rejection if none of the configured routes are matched.

Note: Route's **rate** are discussed in *Billing*.

3.7.2 Router components

The router components are mainly python objects having the unique responsibility of routing messages to Jasmin connectors.

Routable

The **Routable** class is extended by child classes to hold necessary information about the message to be *routed*.

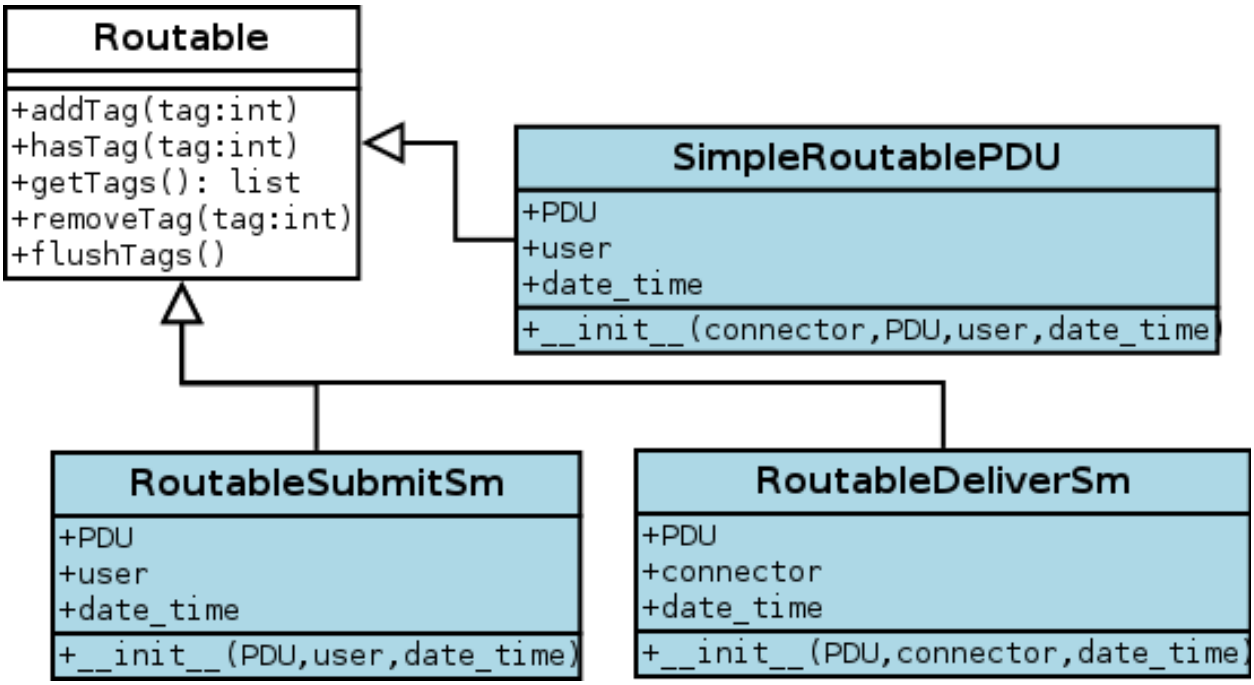


Fig. 3.3: jasmin.routing.Routables.*

The **SimpleRoutablePDU** is only used for Jasmin unit testing, **RoutableSubmitSm** and **RoutableDeliverSm** are used depending on the message direction:

- MO: **RoutableDeliverSm**
- MT: **RoutableSubmitSm**

All routables provide a tagging api through the *addTag()*, *hasTag()*, *getTags()*, *removeTag()*, *flushTags()* methods, this feature is mainly used in the *interceptor*, there's a concrete example of such usage *here*.

Table 3.19: **RoutableSubmitSm** attributes

Attribute	Type	Description
PDU	jasmin.vendor.smpp.pdu.pdu_types.PDURequest	The SMPP submit_sm PDU
user	jasmin.routing.jasminApi.User	Jasmin user sending the message
date_time	datetime.datetime	Date & time of message send request

Table 3.20: **RoutableDeliverSm** attributes

Attribute	Type	Description
PDU	jasmin.vendor.smpp.pdu.pdu_types.PDURequest	The SMPP deliver_sm PDU
connector	jasmin.routing.jasminApi.Connector	Jasmin origin connector of the message
date_time	datetime.datetime	Date & time of message reception

Connector

The **Connector** class is extended by child classes to represent concrete HTTP or SMPP Client connectors.

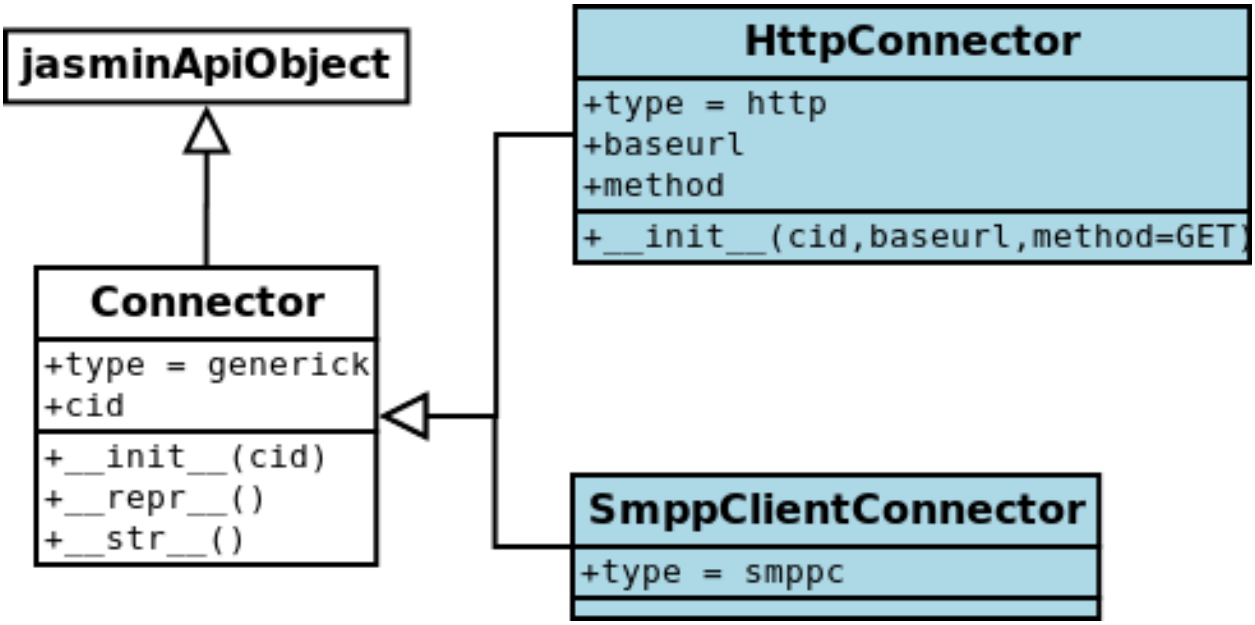


Fig. 3.4: jasmin.routing.jasminApi.Connector and childs

Filter

The **Filter** class is extended by child classes to define specific filters which are run by Jasmin router to match a desired **Routable**, every filter have a public **match(routable)** method returning a boolean value (*True* if the filter matches the given **Routable**).

As explained, filters provide an advanced and customizable method to match for routables and decide which route to consider, the figure below shows the **Filter** implementations provided by Jasmin, you can extend the **Filter** class and build a new filter of your own.

The **usedFor** attribute indicates the filter-route compatibility, as some filters are not suitable for both MO and MT routes like the examples below:

- **UserFilter** and **GroupFilter**: MO Messages are not identified by a user or a group, they are received through a connector
- **ConnectorFilter**: MT Messages are not coming from a connector, they are sent by a known user/group.

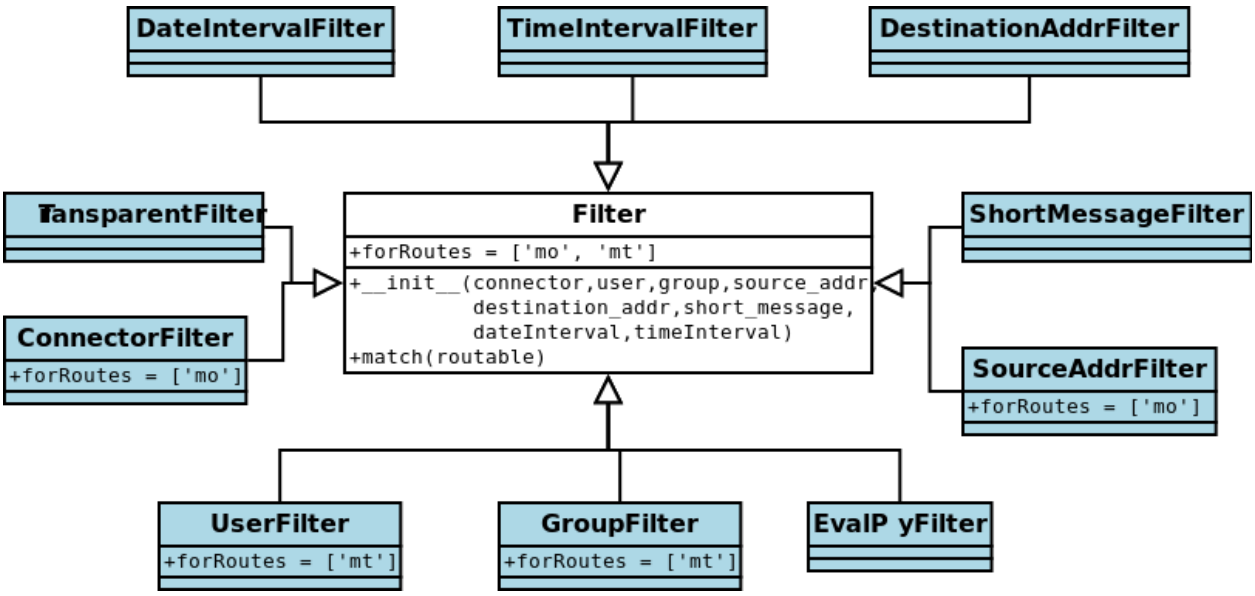


Fig. 3.5: jasmin.routing.Filters.*

Route

A **Route** class holds one or many filters, the **matchFilters(routable)** method is called to match the given routable against every filter of the **Route** (using *AND* operation when there's many filters), if the matching succeed, the Jamsin router will ask for the **Connector** to consider by calling **getConnector()** method which will return back the **Route** 's connector.

Static and default routes are the simplest implemented routes, the difference between them is:

- **DefaultRoute** 's **matchFilter()** method will always return True, it is usually a fallback route matching any **Routable**
- **StaticMORoute** and **StaticMTRoute** will return one **Connector** after matching the filters with **matchFilters(routable)** method

There's a lot of things you can do by extending the **Route** class, here's a bunch of possibilities:

- *Best quality routing*: Implement a connector scoring system to always return the best quality route for a given message

Multiple connectors

When extending **Route** class, it is possible to customize the behavior of the route and that's what **RoundrobinMORoute** and **RoundrobinMTRoute** do, they are initially provisioned with a set of connectors, and the **getConnector()** method is overloaded to return a random connector from it; this can be a basic usage of a load balancer route.

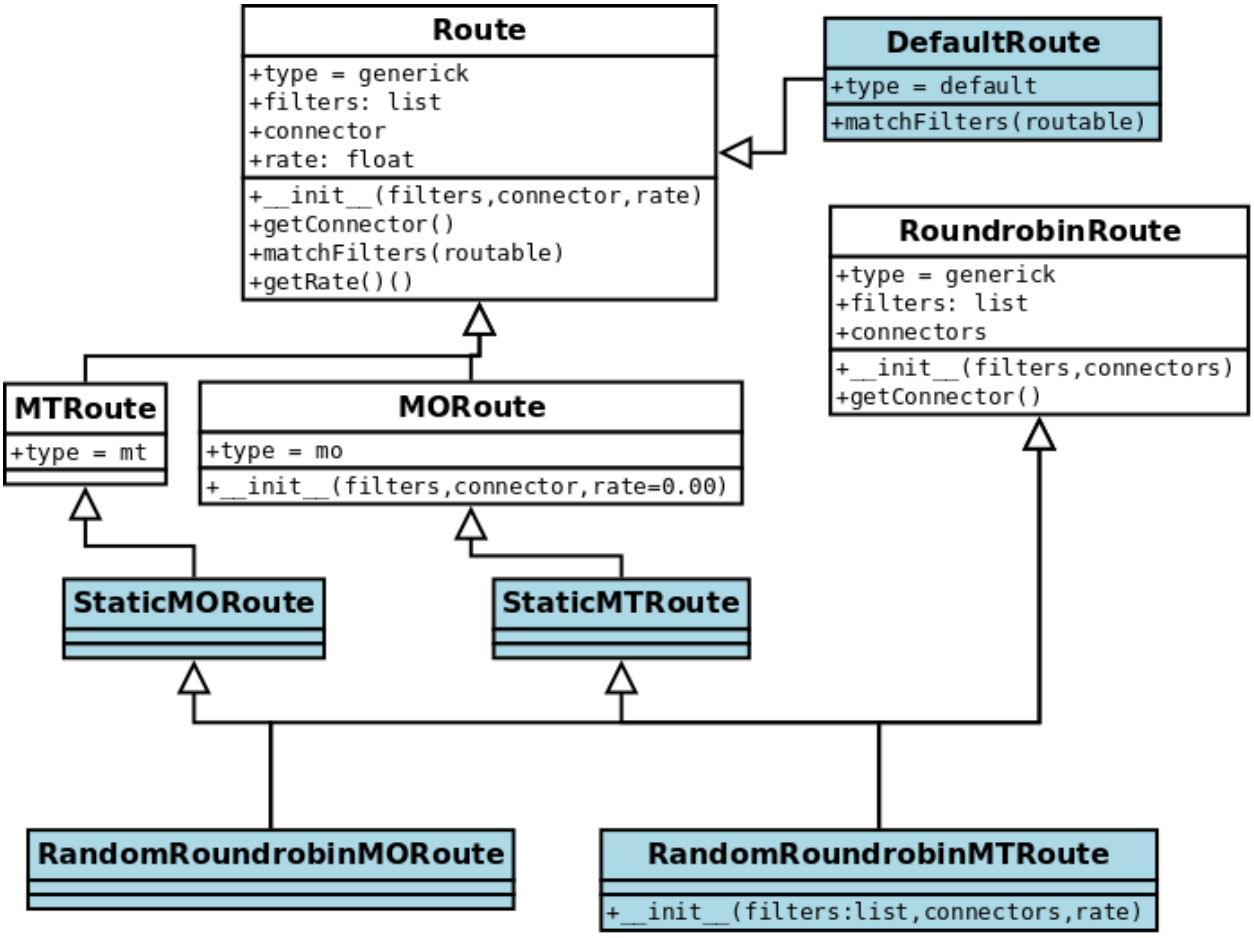


Fig. 3.6: jasmin.routing.Routes.*

The newly added (Jasmin 0.9b10+) has new **FailoverMORoute** and **FailoverMTRoute** routes, they are also extending the **Route** class to provide failover on top of multiple connectors.

RoutingTable

The **RoutingTable** class is extended by destination-specific child classes (MO or MT), each class provide a **Route** provisioning api:

- **add(route, order)**: Will add a new route at a given order, will replace an older route having the same order
- **remove(order)**: Will remove the route at the given order
- **getAll()**: Will return all the provisioned routes
- **flush()**: Will remove all provisioned routes

The **getRouteFor(routable)** will get the right route to consider for a given routable, this method will iterate through all the provisioned routes in descendant order to call their respective **matchFilters(routable)** method.

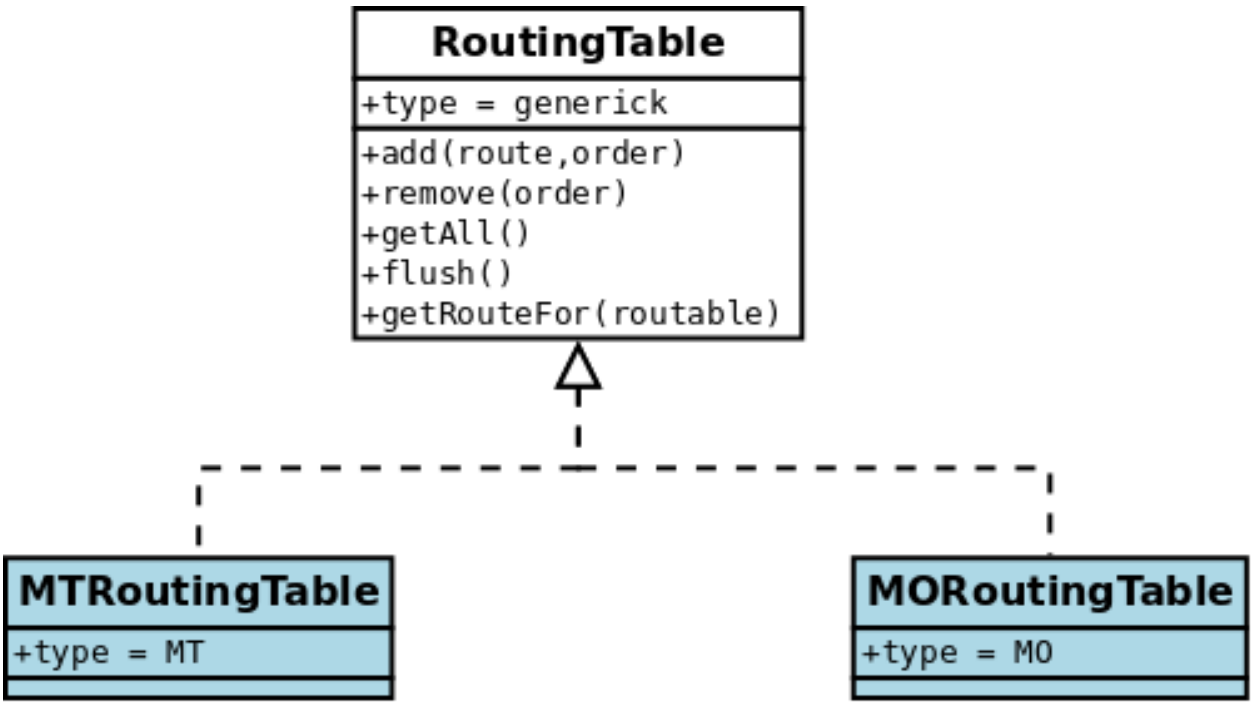


Fig. 3.7: jasmin.routing.RoutingTables.*

3.8 Interception

Starting from 0.7.0, Jasmin provides a convenient way for users to hook third party logics on **intercepted** messages (submit_sm or deliver_sm) before proceding to *The message router*.

Interception of message is based on filter matching, just like the router; every intercepted message will be handed to a user-script written in Python.

This feature permits users to implement custom behaviors on top of Jasmin router, here's some possible scenarios:

- Billing & charging of MO messages,

- Implement HLR lookup for a better SMS MT routing,
- Change a pdu content: fix npi/ton, prefixing/suffixing numbers, etc ...
- Modify Jasmin's response for the message: send back a *ESME_RINVDSTADR* instead of *ESME_ROK* for example.
- etc ..

3.8.1 Enabling interceptor

Jasmin's interceptor is a system service that run separately from Jasmin, it can be hosted on remote server as well; **interceptord** is a system service just like **jasmind**, so simply start it by typing:

```
sudo systemctl start jasmin-interceptord
```

Note: After starting the **interceptord** service, you may check */var/log/jasmin/interceptor.log* to ensure everything is okay.

Then you need to enable communication between **jasmind** and **interceptord** services by editing **jasmind** start script (locate the **jasmind.service** file in */etc/systemd*) and replacing the following line:

```
ExecStart=/usr/bin/jasmind.py --username jcliadmin --password jclipwd
```

by:

```
ExecStart=/usr/bin/jasmind.py --username jcliadmin --password jclipwd --enable-
↪interceptor-client
```

The last step is to restart **jasmind** and check */var/log/jasmin/interceptor.log* to ensure connection has been successfully established by finding the following line:

```
INFO     XXXX Authenticated Avatar: iadmin
```

3.8.2 Intercepting a message

As stated earlier, interceptor is behaving similarly to *The message router*, here's an example of setting up a MO message (deliver_sm) interception rule through *jcli management console*:

```
jcli : mointerceptor -a
Adding a new MO Interceptor: (ok: save, ko: exit)
> type DefaultInterceptor
<class 'jasmin.routing.Interceptors.DefaultInterceptor'> arguments:
script
> script python2(/opt/jasmin-scripts/interception/mo-interceptor.py)
> ok
Successfully added MOInterceptor [DefaultInterceptor] with order:0
```

Same thing apply to setting up a MT message (submit_sm) interception rule, here's another example using a filtered rule instead of a default one:

```
jcli : mtinterceptor -a
Adding a new MT Interceptor: (ok: save, ko: exit)
> type StaticMTInterceptor
```

```
<class 'jasmin.routing.Interceptors.DefaultInterceptor'> arguments:
filters, script
> script python2(/opt/jasmin-scripts/interception/mt-interceptor.py)
> filters U-foo;DA-33
> order 100
> ok
Successfully added MTInterceptor [StaticMTInterceptor] with order:100
```

As show in the above examples, the interception rules are straightforward, any matched message will be handed to the script you set through the **script python2(<path_to_pyfile>)** instruction.

When your python script is called it will get the following global variables set:

- **routable**: one of the *jasmin.routing.Routables.Routable* inheriters (*Routable* for more details)
- **smpp_status**: (default to *0*) it is the smpp response that Jasmin must return for the message, more details in *Controlling response*
- **http_status**: (default to *0*) it is the http response that Jasmin must return for the message, more details in *Controlling response*

The script can:

- Override **routable** parameters like setting destination or source addresses, short message, etc ...
- Tag the **routable** to help the router matching a desired rule (useful for HRL lookup routing)
- Control Jasmin response by setting **smpp_status** and/or **http_status**.

Some practical examples are given *below*.

3.8.3 Controlling response

The interceptor script can reject message before it goes to the router, this can be useful for implementing third party controls like:

- Billing and charging authorization: reject message if user has no credits,
- Reject some illegal message content,
- Enable anti-spam to protect destination users from getting flooded,
- etc ...

In order to reject a message, depending on the source of message (httpapi ? smpp server ? smpp client ?) the script must set **smpp_status** and/or **http_status** accordingly to the error to be returned back, here's an error mapping table for smpp:

Table 3.21: **smpp_status** Error mapping

Value	SMPP Status	Description
0	ESME_ROK	No error
1	ESME_RINVMSGLEN	Message Length is invalid
2	ESME_RINVCMDLEN	Command Length is invalid
3	ESME_RINVCMDID	Invalid Command ID
4	ESME_RINVBNDSTS	Invalid BIND Status for given command
5	ESME_RALYBND	ESME Already in Bound State
6	ESME_RINVPRTFLG	Invalid Priority Flag
7	ESME_RINVREGDLVFLG	Invalid Registered Delivery Flag

Continued on next page

Table 3.21 – continued from previous page

Value	SMPP Status	Description
8	ESME_RSYSERR	System Error
265	ESME_RINVBCASTAREAFMT	Broadcast Area Format is invalid
10	ESME_RINVSRCADR	Invalid Source Address
11	ESME_RINVDSTADR	Invalid Dest Addr
12	ESME_RINVMSGID	Message ID is invalid
13	ESME_RBINDFAIL	Bind Failed
14	ESME_RINVPASWD	Invalid Password
15	ESME_RINVSYSID	Invalid System ID
272	ESME_RINVBCAST_REP	Number of Repeated Broadcasts is invalid
17	ESME_RCANCELFAIL	Cancel SM Failed
274	ESME_RINVBCASTCHANIND	Broadcast Channel Indicator is invalid
19	ESME_RREPLACEFAIL	Replace SM Failed
20	ESME_RMSGQFUL	Message Queue Full
21	ESME_RINVSERTYP	Invalid Service Type
196	ESME_RINVOPTPARAMVAL	Invalid Optional Parameter Value
260	ESME_RINVDCS	Invalid Data Coding Scheme
261	ESME_RINVSRCADDRSUBUNIT	Source Address Sub unit is Invalid
262	ESME_RINVDSTADDRSUBUNIT	Destination Address Sub unit is Invalid
263	ESME_RINVBCASTFREQINT	Broadcast Frequency Interval is invalid
257	ESME_RPROHIBITED	ESME Prohibited from using specified operation
273	ESME_RINVBCASTSRVGRP	Broadcast Service Group is invalid
264	ESME_RINVBCASTALIAS_NAME	Broadcast Alias Name is invalid
270	ESME_RBCASTQUERYFAIL	query_broadcast_sm operation failed
51	ESME_RINVNUMDESTS	Invalid number of destinations
52	ESME_RINVDLNAME	Invalid Distribution List Name
267	ESME_RINVBCASTCNTTYPE	Broadcast Content Type is invalid
266	ESME_RINVNUMBCAST_AREAS	Number of Broadcast Areas is invalid
192	ESME_RINVOPTPARSTREAM	Error in the optional part of the PDU Body
64	ESME_RINVDESTFLAG	Destination flag is invalid (submit_multi)
193	ESME_ROPTPARNOTALLWD	Optional Parameter not allowed
66	ESME_RINVSUBREP	Invalid submit with replace request (i.e. submit_sm with replace_if_present_flag set
67	ESME_RINVESMCLASS	Invalid esm_class field data
68	ESME_RCNTSUBDL	Cannot Submit to Distribution List
69	ESME_RSUBMITFAIL	submit_sm or submit_multi failed
256	ESME_RSERTYPUNAUTH	ESME Not authorised to use specified service_type
72	ESME_RINVSRCTON	Invalid Source address TON
73	ESME_RINVSRCNPI	Invalid Source address NPI
258	ESME_RSERTYPUNAVAIL	Specified service_type is unavailable
269	ESME_RBCASTFAIL	broadcast_sm operation failed
80	ESME_RINVDSTTON	Invalid Destination address TON
81	ESME_RINVDSTNPI	Invalid Destination address NPI
83	ESME_RINVSYSTYP	Invalid system_type field
84	ESME_RINVREPFLAG	Invalid replace_if_present flag
85	ESME_RINVNUMMSGS	Invalid number of messages
88	ESME_RTHROTTLED	Throttling error (ESME has exceeded allowed message limits
271	ESME_RBCASTCANCELFAIL	cancel_broadcast_sm operation failed
97	ESME_RINVSCHED	Invalid Scheduled Delivery Time
98	ESME_RINVEXPIRY	Invalid message validity period (Expiry time)
99	ESME_RINVDFTMSGID	Predefined Message Invalid or Not Found

Continued on next page

Table 3.21 – continued from previous page

Value	SMPP Status	Description
100	ESME_RX_T_APPN	ESME Receiver Temporary App Error Code
101	ESME_RX_P_APPN	ESME Receiver Permanent App Error Code
102	ESME_RX_R_APPN	ESME Receiver Reject Message Error Code
103	ESME_RQUERYFAIL	query_sm request failed
259	ESME_RSERTYPDENIED	Specified service_type is denied
194	ESME_RINVPARLEN	Invalid Parameter Length
268	ESME_RINVBCASTMSGCLASS	Broadcast Message Class is invalid
255	ESME_RUNKNOWNERR	Unknown Error
254	ESME_RDELIVERYFAILURE	Delivery Failure (used for data_sm_resp)
195	ESME_RMISSINGOPTPARAM	Expected Optional Parameter missing

As for http errors, the value you set in **http_status** will be the http error code to return.

Note: When setting **http_status** to some value different from 0, the **smpp_status** value will be automatically set to **255** (ESME_RUNKNOWNERR).

Note: When setting **smpp_status** to some value different from 0, the **http_status** value will be automatically set to **520** (Unknown error).

Note: When setting **smpp_status** to 0, the routing process will be bypassed and an ESME_ROK status is returned.

Checkout the *MO Charging* example to see how's rejection is done.

3.8.4 Scripting examples

You'll find below some helping examples of scripts used to intercept MO and/or MT messages.

HLR Lookup routing

The following script will help the router decide where to send the MT message, let's say we have some HLR lookup webservice to call in order to know to which network the destination number belong, and then tag the routable for later filtering in router:

```
"This script will call HLR lookup api to get the MCC/MNC of the destination number"

import requests, json

hlr_lookup_url = "https://api.some-provider.com/hlr/lookup"
data = json.dumps({'number': routable.pdu.params['destination_addr']})
r = requests.post(hlr_lookup_url, data, auth=('user', '*****'))

if r.json['mnc'] == '214':
    # Spain
    if r.json['mcc'] == '01':
        # Vodaphone
        routable.addTag(21401)
```

```
    elif r.json['mcc'] == '03':
        # Orange
        routable.addTag(21403)
    elif r.json['mcc'] == '25':
        # Lyca mobile
        routable.addTag(21425)
```

The script is tagging the routable if destination is Vodaphone, Orange or Lyca mobile; that's because we need to route message to different connector based on destination network, let's say:

- *Vodaphone* needs to be routed through **connectorA**
- *Orange* needs to be routed through **connectorB**
- *Lyca mobile* needs to be routed through **connectorC**
- All the rest needs to be routed through **connectorD**

Here's the routing table to execute the above example:

```
jcli : mtrouter -l
#Order Type            Rate    Connector ID(s)     Filter(s)
#102   StaticMTRoute   0 (!)   smppc(connectorA)   <TG (tag=21401)>
#101   StaticMTRoute   0 (!)   smppc(connectorB)   <TG (tag=21403)>
#100   StaticMTRoute   0 (!)   smppc(connectorC)   <TG (tag=21425)>
#0     DefaultRoute    0 (!)   smppc(connectorD)
Total MT Routes: 4
```

MO Charging

In this case, the script is calling CGRateS charging system to check if user has sufficient balance to send sms, based on the following script, Jasmin will return a **ESME_ROK** if user balance, or **ESME_RDELIVERYFAILURE** if not:

```
"""This script will receive Mobile-Originated messages and
ask CGRateS for authorization through ApierV2.GetMaxUsage call.
"""
import json, socket
from datetime import datetime

CGR_HOST="172.20.20.140"
CGR_PORT=3422

def call(sck, name, params):
    # Build the request
    request = dict(id=1,
                   params=list(params),
                   method=name)
    sck.sendall(json.dumps(request).encode())
    # This must loop if resp is bigger than 4K
    buffer = ''
    data = True
    while data:
        data = sck.recv(4096)
        buffer += data
        if len(data) < 4096:
            break
    response = json.loads(buffer.decode())
    if response.get('id') != request.get('id'):
```

```
        raise Exception("expected id=%s, received id=%s: %s"
                        %(request.get('id'), response.get('id'),
                                response.get('error')))

    if response.get('error') is not None:
        raise Exception(response.get('error'))

    return response.get('result')

sck = None
globals()['sck'] = sck
globals()['json'] = json
try:
    sck = socket.create_connection((CGR_HOST, CGR_PORT))

    # Prepare for RPC call
    name = "ApierV2.GetMaxUsage"
    params = [{
        "Category": "sms-mt",
        "Usage": "1",
        "Direction": "*outbound",
        "ReqType": "*subscribers",
        "TOR": "*sms-mt",
        "ExtraFields": {"Cli": routable.pdu.params['source_addr']},
        "Destination": routable.pdu.params['destination_addr'],
        "Account": "*subscribers",
        "Tenant": "*subscribers",
        "SetupTime": datetime.utcnow().isoformat() + 'Z'}]

    result = call(sck, name, params)
except Exception, e:
    # We got an error when calling for charging
    # Return ESME_RDELIVERYFAILURE
    smpp_status = 254
else:
    # CGRateS has returned a value

    if type(result) == int and result >= 1:
        # Return ESME_ROK
        smpp_status = 0
    else:
        # Return ESME_RDELIVERYFAILURE
        smpp_status = 254
finally:
    if sck is not None:
        sck.close()
```

Overriding source address

There's some cases where you need to override sender-id due to some MNO policies, in the following example all intercepted messages will have their sender-id set to **123456789**:

```
"This script will override sender-id"

routable.pdu.params['source_addr'] = '123456789'
```

Note: Some pdu parameters require locking to protect them from being updated by Jasmin, more on this.

Activate logging

The following is an example of activating log inside a script:

```
"This is how logging is done inside interception script"

import logging

# Set logger
logger = logging.getLogger('logging-example')
if len(logger.handlers) != 1:
    hdlr = logging.FileHandler('/var/log/jasmin/some_file.log')
    formatter = logging.Formatter('%(asctime)s %(levelname)s %(message)s')
    hdlr.setFormatter(formatter)
    logger.addHandler(hdlr)
    logger.setLevel(logging.DEBUG)

logger.info('Got pdu: %s' % routable.pdu)
```

Enforcing DLR

Ask for DLR for all submit_sm pdus, no matter the downstream user choice, can be used for route qualification and scoring purposes.

```
"This script will enforce sending message while asking for DLR"

from jasmin.vendor.smpp.pdu.pdu_types import RegisteredDeliveryReceipt,␣
↪RegisteredDelivery

routable.pdu.params['registered_delivery'] = RegisteredDelivery(
    RegisteredDeliveryReceipt.SMSC_DELIVERY_RECEIPT_REQUESTED)
```

3.9 Programming examples

Subsequent chapters present how to send and receive messages through Jasmin *HTTP API* and some more advanced use cases, such as manipulating receipts and complex routings, will look like.

It is assumed the reader has already installed Jasmin and at least read the *HTTP API* and *The message router* chapters and knows enough about Jasmin's architecture/design concepts.

3.9.1 Sending SMS

Sending a SMS is done through the *HTTP API*:

```
# Python example
# http://jasminsms.com
import urllib2
import urllib
```

```
baseParams = {'username':'foo', 'password':'bar', 'to':'+336222172', 'content':'Hello
↪'}

# Send an SMS-MT with minimal parameters
urllib2.urlopen("http://127.0.0.1:1401/send?%s" % urllib.urlencode(baseParams)).read()

# Send an SMS-MT with defined originating address
baseParams['from'] = 'Jasmin GW'
urllib2.urlopen("http://127.0.0.1:1401/send?%s" % urllib.urlencode(baseParams)).read()
```

In PHP:

```
<?php
// Sending simple message using PHP
// http://jasminsms.com

$baseurl = 'http://127.0.0.1:1401/send'

$params = '?username=foo'
$params.= '&password=bar'
$params.= '&to='.urlencode('+336222172')
$params.= '&content='.urlencode('Hello world !')

$response = file_get_contents($baseurl.$params);
?>
```

In Ruby:

```
# Sending simple message using Ruby
# http://jasminsms.com

require 'net/http'

uri = URI('http://127.0.0.1:1401/send')
params = { :username => 'foo', :password => 'bar',
           :to => '+336222172', :content => 'Hello world' }
uri.query = URI.encode_www_form(params)

response = Net::HTTP.get_response(uri)
```

c.f. *HTTP API* for more details about sending SMS with receipt enquiry, long content etc ...

3.9.2 Receiving SMS

Receiving a SMS is done through the *HTTP API*, this a PHP script pointed by Jasmin for every received SMS (using routing):

```
<?php
// Receiving simple message using PHP through HTTP Post
// This example will store every received SMS to a SQL table
// http://jasminsms.com


$MO_SMS = $_POST;
```

```
$db = pg_connect('host=127.0.0.1 port=5432 dbname=sms_demo user=jasmin␣
↪password=jajapwd');
if (!$db)
    // We'll not ACK the message, Jasmin will resend it later
    die("Error connecting to DB");

$QUERY = "INSERT INTO sms_mo(id, from, to, cid, priority, coding, validity, content)
↪";
$QUERY.= "VALUES ('%s', '%s', '%s', '%s', '%s', '%s', '%s', '%s');";

$Q = sprintf($QUERY, pg_escape_string($MO_SMS['id']),
                     pg_escape_string($MO_SMS['from']),
                     pg_escape_string($MO_SMS['to']),
                     pg_escape_string($MO_SMS['origin-connector']),
                     pg_escape_string($MO_SMS['priority']),
                     pg_escape_string($MO_SMS['coding']),
                     pg_escape_string($MO_SMS['validity']),
                     pg_escape_string($MO_SMS['content'])
                     );
pg_query($Q);
pg_close($db);

// Acking back Jasmin is mandatory
echo "ACK/Jasmin";
```

In the above example, there's an error handling where the message is not ACKed if there's a database connection problem, if it occurs, the script will return "**Error connecting to DB**" when Jasmin HTTP thrower is waiting for a "**ACL/Jasmin**", this will lead to a message re-queue and later re-delivery to the same script, this behaviour is explained in *Processing*.

Another example of an interactive SMS application:

```
<?php
// Will filter received messages, if the syntax is correct (weather <city name>)
// it will provide a `fake` weather forecast back to the user.
// http://jasminsms.com

$MO_SMS = $_POST;

// Acking back Jasmin is mandatory
echo "ACK/Jasmin";

// Syntax check
if (!preg_match('/^(weather) (.*)/', $MO_SMS['content'], $matches))
    $RESPONSE = "SMS Syntax error, please type 'weather city' to get a fresh weather␣
↪forecast";
else
    $RESPONSE = $martches[2]." forecast: Sunny 21°C, 13Knots NW light wind";

// Send $RESPONSE back to the user ($MO_SMS['from'])
$baseurl = 'http://127.0.0.1:1401/send'
$params = '?username=foo'
$params.= '&password=bar'
$params.= '&to='.urlencode($MO_SMS['from'])
$params.= '&content='.urlencode($RESPONSE)

$response = file_get_contents($baseurl.$params);
```

```
// Note:
// If you need to check if the message is really delivered (or at least, taken by␣
↪Jasmin for delivery)
// you must test for $response value, it must begin with "Success", c.f. HTTP API doc␣
↪for more details
```

c.f. *HTTP API* for more details.

3.9.3 Routing

c.f. *MO router manager* and *MT router manager* for routing scenarios. c.f. *The message router* for details about routing.

3.10 Management CLI overview

jCli is Jasmin's CLI interface, it is an advanced console to manage and configure everything needed to start messaging through Jasmin, from users to connectors and message routing management.

jCli is multi-profile configurator where it is possible to create a testing, staging and production profiles to hold different sets of configurations depending on the desired execution environment.

In order to connect to jCli and start managing Jasmin, the following requirements must be met:

- You need a jCli admin account
- You need to have a connection to jCli's tcp port

Jasmin management through jCli is done using different modules (users, groups, filters, smpp connectors, http connectors ...), these are detailed in *Management CLI Modules*, before going to this part, you have to understand how to:

- *Configure* jCli to change it's binding host and port, authentication and logging parameters,
- *Authenticate* to jCli and discover basic commands to navigate through the console,
- *Know how* to persist to disk the current configuration before restarting or load a specific configuration profile to run test scenarios for example

3.10.1 Architecture

The Jasmin CLI interface is designed to be a user interactive interface on front of the Perspective brokers provided by Jasmin.

In the above figure, every Jasmin CLI module (blue boxes) is connected to its perspective broker, and below you find more details on the Perspective brokers used and the actions they are exposing:

- **SMPPClientManagerPB** which provides the following actions:
  1. **persist**: Persist current configuration to disk
  2. **load**: Load configuration from disk
  3. **is_persisted**: Used to check if the current configuration is persisted or not
  4. **connector_add**: Add a SMPP Client connector
  5. **connector_remove**: Remove a SMPP Client connector

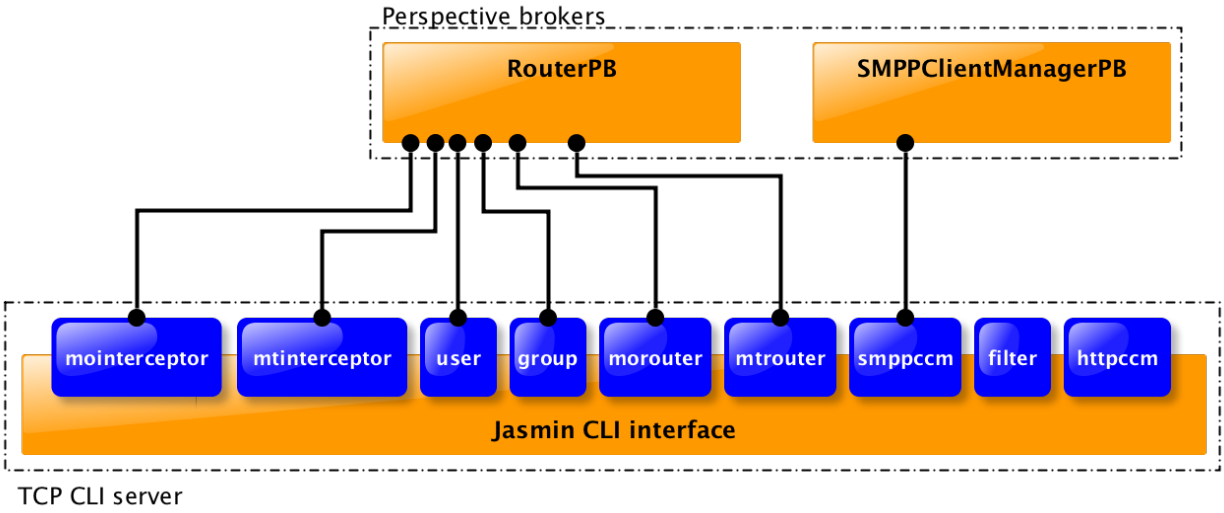


Fig. 3.8: Jasmin CLI architecture

6. **connector_list**: List all SMPP Client connectors
7. **connector_start**: Start a SMPP Client connector
8. **connector_stop**: Stop a SMPP Client connector
9. **connector_stopall**: Stop all SMPP Client connectors
10. **service_status**: Return a SMPP Client connector service status (running or not)
11. **session_state**: Return a SMPP Client connector session state (SMPP binding status)
12. **connector_details**: Get all details for a gived SMPP Client connector
13. **connector_config**: Returns a SMPP Client connector configuration
14. **submit_sm**: Send a submit_sm *

- **RouterPB** which provides the following actions:
  1. **persist**: Persist current configuration to disk
  2. **load**: Load configuration from disk
  3. **is_persisted**: Used to check if the current configuration is persisted or not
  4. **user_add**: Add a new user
  5. **user_authenticate**: Authenticate username/password with the existent users *
  6. **user_remove**: Remove a user
  7. **user_remove_all**: Remove all users
  8. **user_get_all**: Get all users
  9. **user_update_quota**: Update a user quota
  10. **group_add**: Add a group
  11. **group_remove**: Remove a group
  12. **group_remove_all**: Remove all groups

13. **group_get_all**: Get all groups
14. **mtroute_add**: Add a new MT route
15. **moroute_add**: Add a new MO route
16. **mtroute_remove**: Remove a MT route
17. **moroute_remove**: Remove a MO route
18. **mtroute_flush**: Flush MT routes
19. **moroute_flush**: Flush MO routes
20. **mtroute_get_all**: Get all MT routes
21. **moroute_get_all**: Get all MO routes
22. **mtinterceptor_add**: Add a new MT interceptor
23. **mointerceptor_add**: Add a new MO interceptor
24. **mtinterceptor_remove**: Remove a MT interceptor
25. **mointerceptor_remove**: Remove a MO interceptor
26. **mtinterceptor_flush**: Flush MT interceptor
27. **mointerceptor_flush**: Flush MO interceptor
28. **mtinterceptor_get_all**: Get all MT interceptor
29. **mointerceptor_get_all**: Get all MO interceptor

Note: (*): These actions are not exposed through jCli

Hint: **SMPPClientManagerPB** and **RouterPB** are available for third party applications to implement specific business processes, there's a *FAQ subject including an example* of how an external application can use these Perspective Brokers.

3.10.2 Configuration

The **jasmin.cfg** file *(INI format, located in /etc/jasmin)* contains a **jcli** section where all JCli interface related config elements are:

```
[jcli]
bind             = 127.0.0.1
port             = 8990
authentication   = True
admin_username   = jcliadmin
# MD5 password digest hex encoded
admin_password   = 79e9b0aa3f3e7c53e916f7ac47439bcb

log_level        = INFO
log_file         = /var/log/jasmin/jcli.log
log_format       = %(asctime)s %(levelname)-8s %(process)d %(message)s
log_date_format  = %Y-%m-%d %H:%M:%S
```

Table 3.22: [jcli] configuration section

Element	Default	Description
bind	127.0.0.1	jCli will only bind to this specified address.
port	8990	The binding TCP port.
authentication	True	If set to **False**, anonymous user can connect to jCli and admin user account is no more needed
admin_username	jcliadmin	The admin username
admin_password	jclipwd	The admin MD5 crypted password
log_*		Python's logging module configuration.

Warning: Don't set **authentication** to False if you're not sure about what you are doing

3.10.3 First connection & authentication

In order to connect to jCli, initiate a telnet session with the hostname/ip and port of jCli as set in *Configuration*:

```
telnet 127.0.0.1 8990
```

And depending on whether **authentication** is set to True or False, you may have to authenticate using the **admin_username** and **admin_password**, here's an example of an authenticated connection:

```
Authentication required.

Username: jcliadmin
Password:
Welcome to Jasmin console
Type help or ? to list commands.

Session ref: 2
jcli :
```

Once successfully connected, you'll get a welcome message, your session id (Session ref) and a prompt (jcli :) where you can start typing your commands and use *Management CLI Modules*.

Available commands:

Using tabulation will help you discover the available commands:

```
jcli : [TABULATION]
persist load user group filter mointerceptor mtinterceptor morouter mtrouter smppccm␣
↪httpccm quit help
```

Or type **help** and you'll get detailed listing of the available commands with comprehensive descriptions:

```
jcli : help
Available commands:
===================
persist             Persist current configuration profile to disk in PROFILE
load                Load configuration PROFILE profile from disk
```

```
user                 User management
group                Group management
filter               Filter management
mointerceptor        MO Interceptor management
mtinterceptor        MT Interceptor management
morouter             MO Router management
mtrouter             MT Router management
smppccm              SMPP connector management
httpccm              HTTP client connector management

Control commands:
=================
quit                 Disconnect from console
help                 List available commands with "help" or detailed help with "help
↪cmd".
```

More detailed help for a specific command can be obtained running **help cmd** where **cmd** is the command you need help for:

```
jcli : help user
User management
Usage: user [options]

Options:
  -l, --list            List all users or a group users when provided with GID
  -a, --add             Add user
  -u UID, --update=UID  Update user using it's UID
  -r UID, --remove=UID  Remove user using it's UID
  -s UID, --show=UID    Show user using it's UID
```

Interactivity:

When running a command you may enter an interactive session, for example, adding a user with **user -a** will start an interactive session where you have to indicate the user parameters, the prompt will be changed from **jcli :** to **>** indicating you are in an interactive session:

```
jcli : user -a
Adding a new User: (ok: save, ko: exit)
> username foo
> password bar
> uid u1
> gid g1
> ok
Successfully added User [u1] to Group [g1]
```

In the above example, user parameters were **username**, **password**, **uid** and **gid**, note that there's no order in entering these parameters, and you may use a simple TABULATION to get the parameters you have to enter:

```
...
> [TABULATION]
username password gid uid
...
```

3.10.4 Profiles and persistence

Everything done using the Jasmin console will be set in runtime memory, and it will remain there until Jasmin is stopped, that's where persistence is needed to keep the same configuration when restarting.

Persist

Typing **persist** command below will persist runtime configuration to disk using the default profile set in *Configuration*:

```
jcli : persist
mtrouter configuration persisted (profile:jcli-prod)
smppcc configuration persisted (profile:jcli-prod)
group configuration persisted (profile:jcli-prod)
user configuration persisted (profile:jcli-prod)
httpcc configuration persisted (profile:jcli-prod)
mointerceptor configuration persisted (profile:jcli-prod)
filter configuration persisted (profile:jcli-prod)
mtinterceptor configuration persisted (profile:jcli-prod)
morouter configuration persisted (profile:jcli-prod)
```

It is possible to persist to a defined profile:

```
jcli : persist -p testing
```

Important: On Jasmin startup, **jcli-prod** profile is automatically loaded, any other profile can only be manually loaded through **load -p AnyProfile**.

Load

Like **persist** command, there's a **load** command which will loaded a configuration profile from disk, typing **load** command below will load the default profil set in *Configuration* from disk:

```
jcli : load
mtrouter configuration loaded (profile:jcli-prod)
smppcc configuration loaded (profile:jcli-prod)
group configuration loaded (profile:jcli-prod)
user configuration loaded (profile:jcli-prod)
httpcc configuration loaded (profile:jcli-prod)
mointerceptor configuration loaded (profile:jcli-prod)
filter configuration loaded (profile:jcli-prod)
mtinterceptor configuration loaded (profile:jcli-prod)
morouter configuration loaded (profile:jcli-prod)
```

It is possible to load to a defined profile:

```
jcli : load -p testing
```

Note: When loading a profile, any defined current runtime configuration will lost and replaced by this profile configuration

3.11 Management CLI Modules

As shown in the architecture figure *Architecture*, jCli is mainly composed of management modules interfacing two Perspective brokers (**SMPPClientManagerPB** and **RouterPB**), each module is identified as a manager of a defined scope:

- User management
- Group management
- etc ..

Note: **filter** and **httpccm** modules are not interfacing any Perspective broker, they are facilitating the reuse of created filters and HTTP Client connectors in MO and MT routers, e.g. a HTTP Client connector may be created once and used many times in MO Routes.

3.11.1 User manager

The User manager module is accessible through the **user** command and is providing the following features:

Table 3.23: **user** command line options

Command	Description
-l, –list	List all users or a group users when provided with GID
-a, –add	Add user
-e, –enable	Enable user
-d, –disable	Disable user
-u UID, –update=UID	Update user using it's UID
-r UID, –remove=UID	Remove user using it's UID
-s UID, –show=UID	Show user using it's UID
–smpp-unbind=UID	Unbind user from smpp server using it's UID
–smpp-ban=UID	Unbind and ban user from smpp server using it's UID

A User object is required for:

- *SMPP Server API* authentication to send a SMS (c.f. *Sending SMS-MT*)
- *HTTP API* authentication to send a SMS (c.f. *Sending SMS-MT*)
- Creating a **UserFilter** using the **filter** manager (c.f. *Filter manager*)

Every User **must** be a member of a Group, so before adding a new User, there must be at least one Group available, Groups are identified by *GID* (Group ID).

When adding a User, the following parameters are required:

- **username**: A unique username used for authentication
- **password**
- **uid**: A unique identifier, can be same as **username**
- **gid**: Group Identfier
- **mt_messaging_cred** (*optional*): MT Messaging credentials (c.f. *User credentials*)

Here's an example of adding a new User to the **marketing** group:

```
jcli : user -a
Adding a new User: (ok: save, ko: exit)
> username foo
> password bar
> gid marketing
> uid foo
> ok
Successfully added User [foo] to Group [marketing]
```

All the above parameters can be displayed after User creation, except the password:

```
jcli : user -s foo
username foo
mt_messaging_cred defaultvalue src_addr None
mt_messaging_cred quota http_throughput ND
mt_messaging_cred quota balance ND
mt_messaging_cred quota smpps_throughput ND
mt_messaging_cred quota sms_count ND
mt_messaging_cred quota early_percent ND
mt_messaging_cred valuefilter priority ^[0-3]$
mt_messaging_cred valuefilter content .*
mt_messaging_cred valuefilter src_addr .*
mt_messaging_cred valuefilter dst_addr .*
mt_messaging_cred valuefilter validity_period ^\d+$
mt_messaging_cred authorization http_send True
mt_messaging_cred authorization http_dlr_method True
mt_messaging_cred authorization http_balance True
mt_messaging_cred authorization smpps_send True
mt_messaging_cred authorization priority True
mt_messaging_cred authorization http_long_content True
mt_messaging_cred authorization src_addr True
mt_messaging_cred authorization dlr_level True
mt_messaging_cred authorization http_rate True
mt_messaging_cred authorization validity_period True
mt_messaging_cred authorization http_bulk False
mt_messaging_cred authorization hex_content True
uid foo
smpps_cred quota max_bindings ND
smpps_cred authorization bind True
gid marketing
```

Listing Users will show currently added Users with their UID, GID and Username:

```
jcli : user -l
#User id          Group id         Username         Balance MT SMS Throughput
#foo              1                foo              ND      ND     ND/ND
Total Users: 1
```

Note: When listing a *disabled* user, his User id will be prefixed by **!**, same thing apply to group.

User credentials

MT Messaging section

As seen above, User have an optional **mt_messaging_cred** parameter which define a set of sections:

- **Authorizations**: Privileges to send messages and set some defined parameters,
- **Value filters**: Restrictions on some parameter values (such as source address),
- **Default values**: Default parameter values to be set by Jasmin when not manually set by User,
- **Quotas**: Everything about *Billing*,

For each section of the above, there's keys to be defined when adding/updating a user, the example below show how to set a source address **value filter**, a balance of **44.2**, **unlimited** sms_count and limit SMS throughput in smpp server to 2 messages per second:

```
jcli : user -a
Adding a new User: (ok: save, ko: exit)
> username foo
> password bar
> gid marketing
> uid foo
> mt_messaging_cred valuefilter src_addr ^JASMIN$
> mt_messaging_cred quota balance 44.2
> mt_messaging_cred quota sms_count none
> mt_messaging_cred quota smpps_throughput 2
> ok
Successfully added User [foo] to Group [marketing]
```

Note: Setting *none* value to a user quota will set it as *unlimited* quota.

In the below tables, you can find exhaustive list of keys for each **mt_messaging_cred** section:

Table 3.24: **authorization** section keys

Key	Default	Description
http_send	True	Privilege to send SMS through *Sending SMS-MT* (default is True)
http_balance	True	Privilege to check balance through *Checking account balance* (default is True)
http_rate	True	Privilege to check a message rate through *Checking rate price* (default is True)
http_bulk	False	Privilege to send bulks through http api *(Deprecated and will be removed)*
smpps_send	True	Privilege to send SMS through *SMPP Server API* (default is True)
http_long_content	True	Privilege to send long content SMS through *Sending SMS-MT* (default is True)
dlr_level	True	Privilege to set **dlr-level** parameter (default is True)
http_dlr_method	True	Privilege to set **dlr-method** HTTP parameter (default is True)
src_addr	True	Privilege to define source address of SMS-MT (default is True)
priority	True	Privilege to define priority of SMS-MT (default is True)
validity_period	True	Privilege to define validity_period of SMS-MT (default is True)
hex_content	True	Privilege to send binary message using the *hex-content* parameter (default is NOT SET)

Note: Authorizations keys prefixed by **http_** or **smpps_** are only applicable for their respective channels.

Table 3.25: **valuefilter** section keys

Key	Default	Description
src_addr	.*	Regex pattern to validate source address of SMS-MT
dst_addr	.*	Regex pattern to validate destination address of SMS-MT
content	.*	Regex pattern to validate content of SMS-MT
priority	^[0-3]$	Regex pattern to validate priority of SMS-MT
validity_period	^d+$	Regex pattern to validate validity_period of SMS-MT

Table 3.26: **defaultvalue** section keys

Key	Default	Description
src_addr	*None*	Default source address of SMS-MT

Table 3.27: **quota** section keys

Key	Default	Description
balance	ND	c.f. *1. Balance quota*
sms_count	ND	c.f. *2. sms_count quota*
early_percent	ND	c.f. *Asynchronous billing*
http_throughput	ND	Max. number of messages per second to accept through HTTP API
smpps_throughput	ND	Max. number of messages per second to accept through SMPP Server

Note: It is possible to increment a quota by indicating a sign, ex: *+10* will increment a quota value by 10, *-22.4* will decrease a quota value by 22.4.

SMPP Server section

User have an other optional **smpps_cred** parameter which define a specialized set of sections for defining his credentials for using the *SMPP Server API*:

- **Authorizations**: Privileges to bind,
- **Quotas**: Maximum bound connections at a time (multi binding),

For each section of the above, there's keys to be defined when adding/updating a user, the example below show how to **authorize** binding and set max_bindings to **2**:

```
jcli : user -a
Adding a new User: (ok: save, ko: exit)
> username foo
> password bar
> gid marketing
> uid foo
> smpps_cred authorization bind yes
> smpps_cred quota max_bindings 2
> ok
Successfully added User [foo] to Group [marketing]
```

In the below tables, you can find exhaustive list of keys for each **smpps_cred** section:

Table 3.28: **authorization** section keys

Key	Default	Description
bind	True	Privilege to bind to SMPP Server API

Table 3.29: **quota** section keys

Key	Default	Description
max_bindings	ND	Maximum bound connections at a time (multi binding)

Note: It is possible to increment a quota by indicating a sign, ex: *+10* will increment a quota value by 10, *-2* will decrease a quota value by 2.

3.11.2 Group manager

The Group manager module is accessible through the **group** command and is providing the following features:

Table 3.30: **group** command line options

Command	Description
-l, –list	List groups
-a, –add	Add group
-e, –enable	Enable group
-d, –disable	Disable group
-r GID, –remove=GID	Remove group using it's GID

A Group object is required for:

- Creating a **User** using the **user** manager (c.f. *User manager*)
- Creating a **GroupFilter** using the **filter** manager (c.f. *Filter manager*)

When adding a Group, only one parameter is required:

- **gid**: Group Identfier

Here's an example of adding a new Group:

```
jcli : group -a
Adding a new Group: (ok: save, ko: exit)
> gid marketing
> ok
Successfully added Group [marketing]
```

Listing Groups will show currently added Groups with their GID:

```
jcli : group  -l
#Group id
#marketing
Total Groups: 1
```

Note: When listing a *disabled* group, its group id will be prefixed by **!**.

3.11.3 MO router manager

The MO Router manager module is accessible through the **morouter** command and is providing the following features:

Table 3.31: **morouter** command line options

Command	Description
-l, –list	List MO routes
-a, –add	Add a new MO route
-r ORDER, –remove=ORDER	Remove MO route using it's ORDER
-s ORDER, –show=ORDER	Show MO route using it's ORDER
-f, –flush	Flush MO routing table

Note: MO Route is used to route inbound messages (SMS MO) through two possible channels: http and smpps (SMPP Server).

MO Router helps managing Jasmin's MORoutingTable, which is responsible of providing routes to received SMS MO, here are the basics of Jasmin MO routing mechanism:

1. **MORoutingTable** holds ordered **MORoute** objects (each MORoute has a unique order)
2. A **MORoute** is composed of:
   - **Filters**: One or many filters (c.f. *Filter manager*)
   - **Connector**: One connector (can be *many* in some situations)
3. There's many objects inheriting **MORoute** to provide flexible ways to route messages:
   - **DefaultRoute**: A route without a filter, this one can only set with the lowest order to be a default/fallback route
   - **StaticMORoute**: A basic route with **Filters** and one **Connector**
   - **RandomRoundrobinMORoute**: A route with **Filters** and many **Connectors**, will return a random **Connector** if its **Filters** are matched, can be used as a load balancer route
   - **FailoverMORoute**: A route with **Filters** and many **Connectors**, will return an available (connected) **Connector** if its **Filters** are matched
4. When a SMS MO is received, Jasmin will ask for the right **MORoute** to consider, all routes are checked in descendant order for their respective **Filters** (when a **MORoute** have many filters, they are checked with an **AND** boolean operator)
5. When a **MORoute** is considered (its **Filters** are matching a received SMS MO), Jasmin will use its **Connector** to send the SMS MO.

Check *The message router* for more details about Jasmin's routing.

When adding a MO Route, the following parameters are required:

- **type**: One of the supported MO Routes: DefaultRoute, StaticMORoute, RandomRoundrobinMORoute
- **order**: MO Route order

When choosing the MO Route **type**, additional parameters may be added to the above required parameters.

Here's an example of adding a **DefaultRoute** to a HTTP Client Connector (http_default):

```
jcli : morouter -a
Adding a new MO Route: (ok: save, ko: exit)
> type DefaultRoute
jasmin.routing.Routes.DefaultRoute arguments:
connector
> connector http(http_default)
```

```
> ok
Successfully added MORoute [DefaultRoute] with order:0
```

Note: You don't have to set **order** parameter when the MO Route type is **DefaultRoute**, it will be automatically set to 0

Here's an example of adding a **StaticMORoute** to a HTTP Client Connector (http_1):

```
jcli : morouter -a
Adding a new MO Route: (ok: save, ko: exit)
> type StaticMORoute
jasmin.routing.Routes.StaticMORoute arguments:
filters, connector
> order 10
> filters filter_1
> connector http(http_1)
> ok
Successfully added MORoute [StaticMORoute] with order:10
```

Here's an example of adding a **StaticMORoute** to a SMPP Server user (user_1):

```
jcli : morouter -a
Adding a new MO Route: (ok: save, ko: exit)
> type StaticMORoute
jasmin.routing.Routes.StaticMORoute arguments:
filters, connector
> order 15
> filters filter_2
> connector smpps(user_1)
> ok
Successfully added MORoute [StaticMORoute] with order:15
```

Note: When routing to a smpps connector like the above example the **user_1** designates the **username** of the concerned user, if he's already bound to Jasmin's *SMPP Server API* routed messages will be delivered to him, if not, queuing will take care of delivery.

Here's an example of adding a **RandomRoundrobinMORoute** to two HTTP Client Connectors (http_2 and http_3):

```
jcli : morouter -a
Adding a new MO Route: (ok: save, ko: exit)
> type RandomRoundrobinMORoute
jasmin.routing.Routes.RandomRoundrobinMORoute arguments:
filters, connectors
> filters filter_3;filter_1
> connectors http(http_2);http(http_3)
> order 20
> ok
Successfully added MORoute [RandomRoundrobinMORoute] with order:20
```

Note: It is possible to use a **RoundRobinMORoute** with a mix of connectors, example: **connectors smpps(user_1);http(http_1);http(http_3)**.

Here's an example of adding a **FailoverMORoute** to two HTTP Client Connectors (http_4 and http_5):

```
jcli : morouter -a
Adding a new MO Route: (ok: save, ko: exit)
> type FailoverMORoute
jasmin.routing.Routes.FailoverMORoute arguments:
filters, connectors
> filters filter_4
> connectors http(http_4);http(http_5)
> order 30
> ok
Successfully added MORoute [FailoverMORoute] with order:20
```

Note: It is **not possible** to use a **FailoverMORoute** with a mix of connectors, example: **connectors smpps(user_1);http(http_1);http(http_3)**.

Once the above MO Routes are added to **MORoutingTable**, it is possible to list these routes:

```
jcli : morouter -l
#Order Type                  Connector ID(s)                 Filter(s)
#30    FailoverMORoute       http(http_4), http(http_5)      <T>, <T>
#20    RandomRoundrobinMORoute http(http_2), http(http_3)    <T>, <T>
#15    StaticMORoute         smpps(user_1)                   <T>
#10    StaticMORoute         http(http_1)                    <T>
#0     DefaultRoute          http(http_default)
Total MO Routes: 3
```

Note: Filters and Connectors were created before creating these routes, please check *Filter manager* and *HTTP Client connector manager* for further details

It is possible to obtain more information of a defined route by typing **moroute -s <order>**:

```
jcli : morouter -s 20
RandomRoundrobinMORoute to 2 connectors:
   - http(http_2)
   - http(http_3)

jcli : morouter -s 10
StaticMORoute to http(http_1)

jcli : morouter -s 0
DefaultRoute to http(http_default)
```

More control commands:

- **morouter -r <order>**: Remove route at defined *order*
- **morouter -f**: Flush MORoutingTable (unrecoverable)

3.11.4 MT router manager

The MT Router manager module is accessible through the **mtrouter** command and is providing the following features:

Table 3.32: **mtrouter** command line options

Command	Description
-l, –list	List MT routes
-a, –add	Add a new MT route
-r ORDER, –remove=ORDER	Remove MT route using it's ORDER
-s ORDER, –show=ORDER	Show MT route using it's ORDER
-f, –flush	Flush MT routing table

Note: MT Route is used to route outbound messages (SMS MT) through one channel: smppc (SMPP Client).

MT Router helps managing Jasmin's MTRoutingTable, which is responsible of providing routes to outgoing SMS MT, here are the basics of Jasmin MT routing mechanism:

1. **MTRoutingTable** holds ordered **MTRoute** objects (each MTRoute has a unique order)
2. A **MTRoute** is composed of:
   - **Filters**: One or many filters (c.f. *Filter manager*)
   - **Connector**: One connector (can be *many* in some situations)
   - **Rate**: For billing purpose, the rate of sending one message through this route; it can be zero to mark the route as FREE (NOT RATED) (c.f. *Billing*)
3. There's many objects inheriting **MTRoute** to provide flexible ways to route messages:
   - **DefaultRoute**: A route without a filter, this one can only set with the lowest order to be a default/fallback route
   - **StaticMTRoute**: A basic route with **Filters** and one **Connector**
   - **RandomRoundrobinMTRoute**: A route with **Filters** and many **Connectors**, will return a random **Connector** if its **Filters** are matching, can be used as a load balancer route
   - **FailoverMTRoute**: A route with **Filters** and many **Connectors**, will return an available (connected) **Connector** if its **Filters** are matched
4. When a SMS MT is to be sent, Jasmin will ask for the right **MTRoute** to consider, all routes are checked in descendant order for their respective **Filters** (when a **MTRoute** have many filters, they are checked with an **AND** boolean operator)
5. When a **MTRoute** is considered (its **Filters** are matching an outgoing SMS MT), Jasmin will use its **Connector** to send the SMS MT.

Check *The message router* for more details about Jasmin's routing.

When adding a MT Route, the following parameters are required:

- **type**: One of the supported MT Routes: DefaultRoute, StaticMTRoute, RandomRoundrobinMTRoute
- **order**: MO Route order
- **rate**: The route rate, can be zero

When choosing the MT Route **type**, additional parameters may be added to the above required parameters.

Here's an example of adding a **DefaultRoute** to a SMPP Client Connector (smppcc_default):

```
jcli : mtrouter -a
Adding a new MT Route: (ok: save, ko: exit)
> type DefaultRoute
jasmin.routing.Routes.DefaultRoute arguments:
```

```
connector
> connector smppc(smppcc_default)
> rate 0.0
> ok
Successfully added MTRoute [DefaultRoute] with order:0
```

Note: You don't have to set **order** parameter when the MT Route type is **DefaultRoute**, it will be automatically set to 0

Here's an example of adding a **StaticMTRoute** to a SMPP Client Connector (smppcc_1):

```
jcli : mtrouter -a
Adding a new MT Route: (ok: save, ko: exit)
> type StaticMTRoute
jasmin.routing.Routes.StaticMTRoute arguments:
filters, connector
> filters filter_1;filter_2
> order 10
> connector smppc(smppcc_1)
> rate 0.0
> ok
Successfully added MTRoute [StaticMTRoute] with order:10
```

Here's an example of adding a **RandomRoundrobinMTRoute** to two SMPP Client Connectors (smppcc_2 and smppcc_3):

```
jcli : mtrouter -a
Adding a new MT Route: (ok: save, ko: exit)
> order 20
> type RandomRoundrobinMTRoute
jasmin.routing.Routes.RandomRoundrobinMTRoute arguments:
filters, connectors
> filters filter_3
> connectors smppc(smppcc_2);smppc(smppcc_3)
> rate 0.0
> ok
Successfully added MTRoute [RandomRoundrobinMTRoute] with order:20
```

Here's an example of adding a **FailoverMTRoute** to two SMPP Client Connectors (smppcc_4 and smppcc_5):

```
jcli : mtrouter -a
Adding a new MT Route: (ok: save, ko: exit)
> order 30
> type FailoverMTRoute
jasmin.routing.Routes.FailoverMTRoute arguments:
filters, connectors
> filters filter_4
> connectors smppc(smppcc_4);smppc(smppcc_5)
> rate 0.0
> ok
Successfully added MTRoute [FailoverMTRoute] with order:20
```

Once the above MT Routes are added to **MTRoutingTable**, it is possible to list these routes:

```
jcli : mtrouter -l
#Order Type                    Rate    Connector ID(s)                    Filter(s)
```

```
#20    FailoverMTRoute          0 (!)   smppc(smppcc_3), smppc(smppcc_4)     <T>
#20    RandomRoundrobinMTRoute  0 (!)   smppc(smppcc_2), smppc(smppcc_3)     <T>
#10    StaticMTRoute            0 (!)   smppc(smppcc_1)                      <T>, <T>
#0     DefaultRoute             0 (!)   smppc(smppcc_default)
Total MT Routes: 3
```

Note: Filters and Connectors were created before creating these routes, please check *Filter manager* and *HTTP Client connector manager* for further details

It is possible to obtain more information of a defined route by typing **mtroute -s <order>**:

```
jcli : mtrouter -s 20
RandomRoundrobinMTRoute to 2 connectors:
   - smppc(smppcc_2)
   - smppc(smppcc_3)
NOT RATED

jcli : mtrouter -s 10
StaticMTRoute to smppc(smppcc_1) NOT RATED

jcli : mtrouter -s 0
DefaultRoute to smppc(smppcc_default) NOT RATED
```

More control commands:

- **mtrouter -r <order>**: Remove route at defined *order*
- **mtrouter -f**: Flush MTRoutingTable (unrecoverable)

3.11.5 MO interceptor manager

The MO Interceptor manager module is accessible through the **mointerceptor** command and is providing the following features:

Table 3.33: **mointerceptor** command line options

Command	Description
-l, –list	List MO interceptors
-a, –add	Add a new MO interceptors
-r ORDER, –remove=ORDER	Remove MO interceptor using it's ORDER
-s ORDER, –show=ORDER	Show MO interceptor using it's ORDER
-f, –flush	Flush MO interception table

Note: MO Interceptor is used to hand inbound messages (SMS MO) to a user defined script, check *Interception* for more details.

MO Interceptor helps managing Jasmin's MOInterceptionTable, which is responsible of intercepting SMS MO before routing is made, here are the basics of Jasmin MO interception mechanism:

1. **MOInterceptionTable** holds ordered **MOInterceptor** objects (each MOInterceptor has a unique order)
2. A **MOInterceptor** is composed of:
   - **Filters**: One or many filters (c.f. *Filter manager*)

- **Script**: Path to python script

3. There's many objects inheriting **MOInterceptor** to provide flexible ways to route messages:
   - **DefaultInterceptor**: An interceptor without a filter, this one can only set with the lowest order to be a default/fallback interceptor
   - **StaticMOInterceptor**: A basic interceptor with **Filters** and one **Script**
4. When a SMS MO is received, Jasmin will ask for the right **MOInterceptor** to consider, all interceptors are checked in descendant order for their respective **Filters** (when a **MOInterceptor** have many filters, they are checked with an **AND** boolean operator)
5. When a **MOInterceptor** is considered (its **Filters** are matching a received SMS MO), Jasmin will call its **Script** with the **Routable** argument.

Check *Interception* for more details about Jasmin's interceptor.

When adding a MO Interceptor, the following parameters are required:

- **type**: One of the supported MO Interceptors: DefaultInterceptor, StaticMOInterceptor
- **order**: MO Interceptor order

When choosing the MO Interceptor **type**, additional parameters may be added to the above required parameters.

Here's an example of adding a **DefaultInterceptor** to a python script:

```
jcli : mointerceptor -a
Adding a new MO Interceptor: (ok: save, ko: exit)
> type DefaultInterceptor
<class 'jasmin.routing.Interceptors.DefaultInterceptor'> arguments:
script
> script python2(/opt/jasmin-scripts/interception/mo-interceptor.py)
> ok
Successfully added MOInterceptor [DefaultInterceptor] with order:0
```

Note: As of now, only **python2** script is permitted.

Note: Pay attention that the given script is copied to Jasmin core, do not expect Jasmin to refresh the script code when you update it, you'll need to redefine the *mointerceptor* rule again so Jasmin will refresh the script.

Note: You don't have to set **order** parameter when the MO Interceptor type is **DefaultInterceptor**, it will be automatically set to 0

Here's an example of adding a **StaticMOInterceptor** to a python script:

```
jcli : mointerceptor -a
Adding a new MO Interceptor: (ok: save, ko: exit)
> type StaticMOInterceptor
<class 'jasmin.routing.Interceptors.StaticMOInterceptor'> arguments:
filters, script
> order 10
> filters filter_1
> script python2(/opt/jasmin-scripts/interception/mo-interceptor.py)
> ok
Successfully added MOInterceptor [StaticMOInterceptor] with order:10
```

Once the above MO Interceptors are added to **MOInterceptionTable**, it is possible to list these interceptors:

```
jcli : mointerceptor -l
#Order    Type                    Script                    Filter(s)
#10       StaticMOInterceptor     <MOIS (pyCode= ..)>       <T>
#0        DefaultInterceptor      <MOIS (pyCode= ..)>
Total MO Interceptors: 2
```

Note: Filters were created before creating these interceptors, please check *Filter manager* for further details

It is possible to obtain more information of a defined interceptor by typing **mointerceptor -s <order>**:

```
jcli : mointerceptor -s 10
StaticMOInterceptor/<MOIS (pyCode= ..)>

jcli : mointerceptor -s 0
DefaultInterceptor/<MOIS (pyCode= ..)>
```

More control commands:

- **mointerceptor -r <order>**: Remove interceptor at defined *order*
- **mointerceptor -f**: Flush MOInterceptionTable (unrecoverable)

3.11.6 MT interceptor manager

The MT Interceptor manager module is accessible through the **mtinterceptor** command and is providing the following features:

Table 3.34: **mtinterceptor** command line options

Command	Description
-l, –list	List MT interceptors
-a, –add	Add a new MT interceptors
-r ORDER, –remove=ORDER	Remove MT interceptor using it's ORDER
-s ORDER, –show=ORDER	Show MT interceptor using it's ORDER
-f, –flush	Flush MT interception table

Note: MT Interceptor is used to hand outbound messages (SMS MT) to a user defined script, check *Interception* for more details.

MT Interceptor helps managing Jasmin's MTInterceptionTable, which is responsible of intercepting SMS MT before routing is made, here are the basics of Jasmin MT interception mechanism:

1. **MTInterceptionTable** holds ordered **MTInterceptor** objects (each MTInterceptor has a unique order)
2. A **MTInterceptor** is composed of:
   - **Filters**: One or many filters (c.f. *Filter manager*)
   - **Script**: Path to python script
3. There's many objects inheriting **MTInterceptor** to provide flexible ways to route messages:

- **DefaultInterceptor**: An interceptor without a filter, this one can only set with the lowest order to be a default/fallback interceptor
- **StaticMTInterceptor**: A basic interceptor with **Filters** and one **Script**

4. When a SMS MT is received, Jasmin will ask for the right **MTInterceptor** to consider, all interceptors are checked in descendant order for their respective **Filters** (when a **MTInterceptor** have many filters, they are checked with an **AND** boolean operator)
5. When a **MTInterceptor** is considered (its **Filters** are matching a received SMS MT), Jasmin will call its **Script** with the **Routable** argument.

Check *Interception* for more details about Jasmin's interceptor.

When adding a MT Interceptor, the following parameters are required:

- **type**: One of the supported MT Interceptors: DefaultInterceptor, StaticMTInterceptor
- **order**: MT Interceptor order

When choosing the MT Interceptor **type**, additional parameters may be added to the above required parameters.

Here's an example of adding a **DefaultInterceptor** to a python script:

```
jcli : mtinterceptor -a
Adding a new MT Interceptor: (ok: save, ko: exit)
> type DefaultInterceptor
<class 'jasmin.routing.Interceptors.DefaultInterceptor'> arguments:
script
> script python2(/opt/jasmin-scripts/interception/mt-interceptor.py)
> ok
Successfully added MTInterceptor [DefaultInterceptor] with order:0
```

Note: As of now, only **python2** script is permitted.

Note: Pay attention that the given script is copied to Jasmin core, do not expect Jasmin to refresh the script code when you update it, you'll need to redefine the *mtinterceptor* rule again so Jasmin will refresh the script.

Note: You don't have to set **order** parameter when the MT Interceptor type is **DefaultInterceptor**, it will be automatically set to 0

Here's an example of adding a **StaticMTInterceptor** to a python script:

```
jcli : mtinterceptor -a
Adding a new MT Interceptor: (ok: save, ko: exit)
> type StaticMTInterceptor
<class 'jasmin.routing.Interceptors.StaticMTInterceptor'> arguments:
filters, script
> order 10
> filters filter_1
> script python2(/opt/jasmin-scripts/interception/mt-interceptor.py)
> ok
Successfully added MTInterceptor [StaticMTInterceptor] with order:10
```

Once the above MT Interceptors are added to **MTInterceptionTable**, it is possible to list these interceptors:

```
jcli : mtinterceptor -l
#Order    Type                    Script                    Filter(s)
#10       StaticMTInterceptor     <MTIS (pyCode= ..)>       <T>
#0        DefaultInterceptor      <MTIS (pyCode= ..)>
Total MT Interceptors: 2
```

Note: Filters were created before creating these interceptors, please check *Filter manager* for further details

It is possible to obtain more information of a defined interceptor by typing **mtinterceptor -s <order>**:

```
jcli : mtinterceptor -s 10
StaticMTInterceptor/<MTIS (pyCode= ..)>

jcli : mtinterceptor -s 0
DefaultInterceptor/<MTIS (pyCode= ..)>
```

More control commands:

- **mtinterceptor -r <order>**: Remove interceptor at defined *order*
- **mtinterceptor -f**: Flush MTInterceptionTable (unrecoverable)

3.11.7 SMPP Client connector manager

The SMPP Client connector manager module is accessible through the **smppccm** command and is providing the following features:

Table 3.35: **smppccm** command line options

Command	Description
-l, –list	List SMPP connectors
-a, –add	Add SMPP connector
-u CID, –update=CID	Update SMPP connector configuration using it's CID
-r CID, –remove=CID	Remove SMPP connector using it's CID
-s CID, –show=CID	Show SMPP connector using it's CID
-1 CID, –start=CID	Start SMPP connector using it's CID
-0 CID, –stop=CID	Stop SMPP connector using it's CID

A SMPP Client connector is used to send/receive SMS through SMPP v3.4 protocol, it is directly connected to MO and MT routers to provide end-to-end message delivery.

Adding a new SMPP Client connector requires knowledge of the parameters detailed in the listing below:

Parameter	Description
cid	Connector ID (must be unique)
logfile	
logrotate	When to rotate the log file, possible values: S=Seconds, M=Minutes, H=Hours, D=Days, W0-W6=Weekday (
loglevel	Logging numeric level: 10=DEBUG, 20=INFO, 30=WARNING, 40=ERROR, 50=CRITICCAL
host	Server that runs SMSC
port	The port number for the connection to the SMSC.

Parameter	Description
ssl	Activate ssl connection
username	
password	
bind	Bind type: transceiver, receiver or transmitter
bind_to	Timeout for response to bind request
trx_to	Maximum time lapse allowed between transactions, after which, the connection is considered as inactive and v
res_to	Timeout for responses to any request PDU
pdu_red_to	Timeout for reading a single PDU, this is the maximum lapse of time between receiving PDU's header and its
con_loss_retry	Reconnect on connection loss ? (yes, no)
con_loss_delay	Reconnect delay on connection loss (seconds)
con_fail_retry	Reconnect on connection failure ? (yes, no)
con_fail_delay	Reconnect delay on connection failure (seconds)
src_addr	Default source adress of each SMS-MT if not set while sending it, can be numeric or alphanumeric, when not
src_ton	Source address TON setting for the link: 0=Unknown, 1=International, 2=National, 3=Network specific, 4=Su
src_npi	Source address NPI setting for the link: 0=Unknown, 1=ISDN, 3=Data, 4=Telex, 6=Land mobile, 8=National,
dst_ton	Destination address TON setting for the link: 0=Unknown, 1=International, 2=National, 3=Network specific, 4
dst_npi	Destination address NPI setting for the link: 0=Unknown, 1=ISDN, 3=Data, 4=Telex, 6=Land mobile, 8=Nati
bind_ton	Bind address TON setting for the link: 0=Unknown, 1=International, 2=National, 3=Network specific, 4=Subs
bind_npi	Bind address NPI setting for the link: 0=Unknown, 1=ISDN, 3=Data, 4=Telex, 6=Land mobile, 8=National, 9
validity	Default validity period of each SMS-MT if not set while sending it, when not defined it will take SMSC defau
priority	SMS-MT default priority if not set while sending it: 0, 1, 2 or 3
requeue_delay	Delay to be considered when requeuing a rejected message
addr_range	Indicates which MS's can send messages to this connector, seems to be an informative value
systype	The system_type parameter is used to categorize the type of ESME that is binding to the SMSC. Examples inc
dlr_expiry	When a SMS-MT is not acked, it will remain waiting in memory for *dlr_expiry* seconds, after this period, any
submit_throughput	Active SMS-MT throttling in MPS (Messages per second), set to 0 (zero) for unlimited throughput
proto_id	Used to indicate protocol id in SMS-MT and SMS-MO
coding	Default coding of each SMS-MT if not set while sending it: 0=SMSC Default, 1=IA5 ASCII, 2=Octet unspeci
elink_interval	Enquire link interval (seconds)
def_msg_id	Specifies the SMSC index of a pre-defined ('canned') message.
ripf	Replace if present flag: 0=Do not replace, 1=Replace
dlr_msgid	Indicates how to read msg id when receiving a receipt: 0=msg id is identical in submit_sm_resp and deliver_s

Note: When adding a SMPP Client connector, only it's **cid** is required, all the other parameters will be set to their respective defaults.

Note: Connector restart is required only when changing the following parameters: **host**, **port**, **username**, **password**, **systemType**, **logfile**, **loglevel**; any other change is applied without requiring connector to be restarted.

Here's an example of adding a new **transmitter** SMPP Client connector with **cid=Demo**:

```
jcli : smppccm -a
Adding a new connector: (ok: save, ko: exit)
> cid Demo
> bind transmitter
> ok
Successfully added connector [Demo]
```

All the above parameters can be displayed after connector creation:

```
jcli : smppccm -s Demo
ripf 0
con_fail_delay 10
dlr_expiry 86400
coding 0
submit_throughput 1
elink_interval 10
bind_to 30
port 2775
con_fail_retry yes
password password
src_addr None
bind_npi 1
addr_range None
dst_ton 1
res_to 60
def_msg_id 0
priority 0
con_loss_retry yes
username smppclient
dst_npi 1
validity None
requeue_delay 120
host 127.0.0.1
src_npi 1
trx_to 300
logfile /var/log/jasmin/default-Demo.log
systype
cid Demo
loglevel 20
bind transmitter
proto_id None
con_loss_delay 10
bind_ton 0
pdu_red_to 10
src_ton 2
```

Note: From the example above, you can see that showing a connector details will return all it's parameters even those you did not enter while creating/updating the connector, they will take their respective default values as explained in *SMPP Client connector parameters*

Listing connectors will show currently added SMPP Client connectors with their CID, Service/Session state and start/stop counters:

```
jcli : smppccm -l
#Connector id                          Service Session          Starts Stops
#888                                   stopped None             0      0
#Demo                                  stopped None             0      0
Total connectors: 2
```

Updating an existent connector is the same as creating a new one, simply type **smppccm -u <cid>** where **cid** is the connector id you want to update, you'll run into a new interactive session to enter the parameters you want to update (c.f. *SMPP Client connector parameters*).

Here's an example of updating SMPP Client connector's host:

```
jcli : smppccm -u Demo
Updating connector id [Demo]: (ok: save, ko: exit)
> host 10.10.1.2
> ok
Successfully updated connector [Demo]
```

More control commands:

- **smppccm -1 <cid>**: Start connector and try to connect
- **smppccm -0 <cid>**: Stop connector and disconnect
- **smppccm -r <cid>**: Remove connector (unrecoverable)

3.11.8 Filter manager

The Filter manager module is accessible through the **filter** command and is providing the following features:

Table 3.37: **filter** command line options

Command	Description
-l, –list	List filters
-a, –add	Add filter
-r FID, –remove=FID	Remove filter using it's FID
-s FID, –show=FID	Show filter using it's FID

Filters are used by MO/MT routers to help decide on which route a message must be delivered, the following flowchart provides details of the routing process:

Jasmin provides many Filters offering advanced flexibilities to message routing:

Table 3.38: Jasmin Filters

Name	Routes	Description
TransparentFilter	All	This filter will always match any message criteria
ConnectorFilter	MO	Will match the source connector of a message
UserFilter	MT	Will match the owner of a MT message
GroupFilter	MT	Will match the owner's group of a MT message
SourceAddrFilter	All	Will match the source address of a MO message
DestinationAddrFilter	All	Will match the source address of a message
ShortMessageFilter	All	Will match the content of a message
DateIntervalFilter	All	Will match the date of a message
TimeIntervalFilter	All	Will match the time of a message
TagFilter	All	Will check if message has a defined tag
EvalPyFilter	All	Will pass the message to a third party python script for user-defined filtering

Check *The message router* for more details about Jasmin's routing.

When adding a Filter, the following parameters are required:

- **type**: One of the supported Filters: TransparentFilter, ConnectorFilter, UserFilter, GroupFilter, SourceAddrFilter, DestinationAddrFilter, ShortMessageFilter, DateIntervalFilter, TimeIntervalFilter, TagFilter, EvalPyFilter
- **fid**: Filter id (must be unique)

When choosing the Filter **type**, additional parameters may be added to the above required parameters:

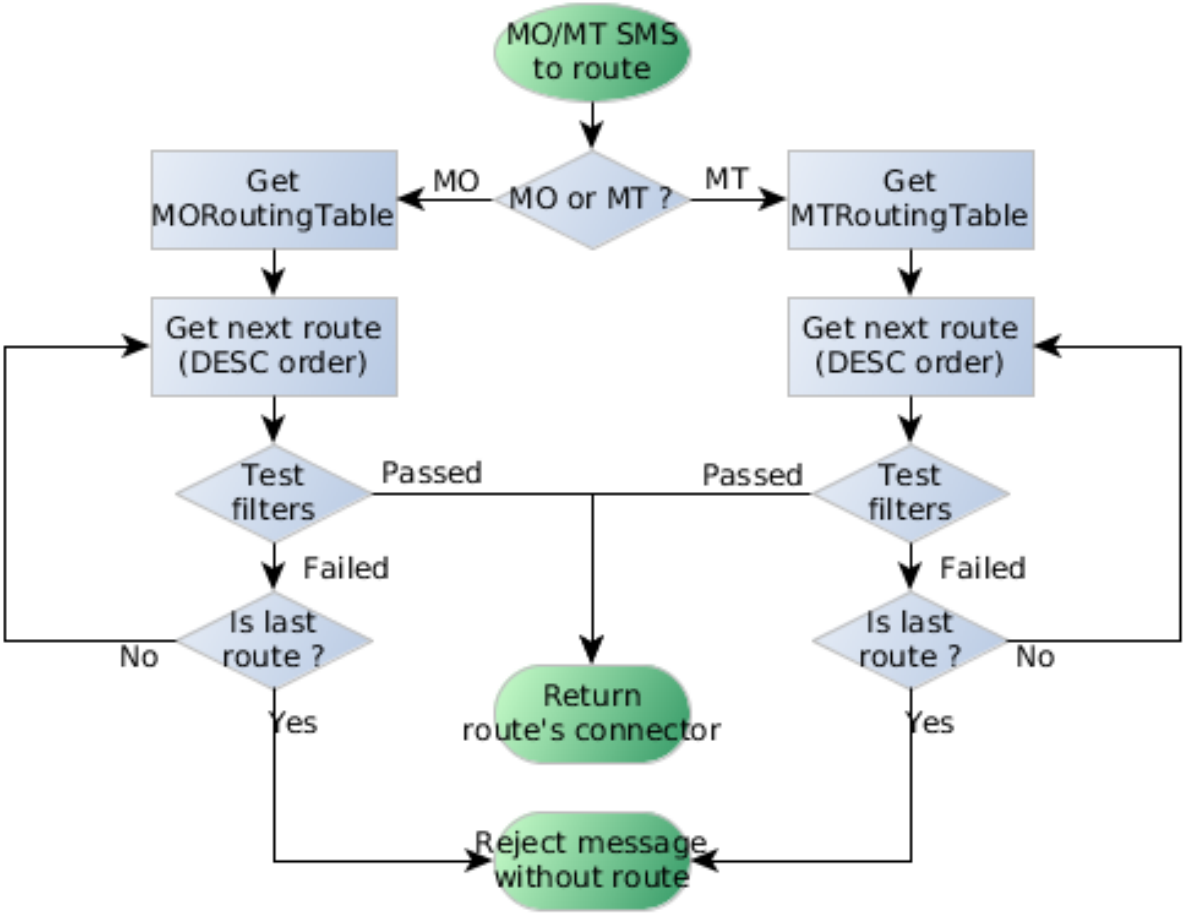


Fig. 3.9: Routing process flow

Table 3.39: Filters parameters

Name	Example	Parameters
TransparentFilter		No parameters are required
ConnectorFilter	smpp-01	**cid** of the connector to match
UserFilter	bobo	**uid** of the user to match
GroupFilter	partners	**gid** of the group to match
SourceAddrFilter	^20d+	**source_addr**: Regular expression to match source address
DestinationAddrFilter	^85111$	**destination_addr**: Regular expression to match destination address
ShortMessageFilter	^hello.*$	**short_message**: Regular expression to match message content
DateIntervalFilter	2014-09-18;2014-09-28	**dateInterval**: Two dates separated by ; (date format is YYYY-MM-DD)
TimeIntervalFilter	08:00:00;18:00:00	**timeInterval**: Two timestamps separated by ; (timestamp format is HH:MM:SS)
TagFilter	32401	**tag**: *numeric* tag to match in message
EvalPyFilter	/root/thirdparty.py	**pyCode**: Path to a python script, (*External business logic* for more details)

Here's an example of adding a **TransparentFilter**

```
jcli : filter -a
Adding a new Filter: (ok: save, ko: exit)
type fid
> type transparentfilter
> fid TF
> ok
Successfully added Filter [TransparentFilter] with fid:TF
```

Here's an example of adding a **SourceAddrFilter**

```
jcli : filter -a
Adding a new Filter: (ok: save, ko: exit)
> type sourceaddrfilter
jasmin.routing.Filters.SourceAddrFilter arguments:
source_addr
> source_addr ^20\d+
> ok
You must set these options before saving: type, fid, source_addr
> fid From20*
> ok
Successfully added Filter [SourceAddrFilter] with fid:From20*
```

Here's an example of adding a **TimeIntervalFilter**

```
jcli : filter -a
Adding a new Filter: (ok: save, ko: exit)
> fid WorkingHours
> type timeintervalfilter
jasmin.routing.Filters.TimeIntervalFilter arguments:
timeInterval
> timeInterval 08:00:00;18:00:00
> ok
Successfully added Filter [TimeIntervalFilter] with fid:WorkingHours
```

It is possible to list filters with:

```
jcli : filter -l
#Filter id         Type                    Routes Description
#StartWithHello    ShortMessageFilter      MO MT  <ShortMessageFilter (msg=^hello.*$)>
#ExternalPy        EvalPyFilter            MO MT  <EvalPyFilter (pyCode= ..)>
#To85111           DestinationAddrFilter   MO MT  <DestinationAddrFilter (dst_addr=^
↪85111$)>
#September2014     DateIntervalFilter      MO MT  <DateIntervalFilter (2014-09-01,2014-
↪09-30)>
#WorkingHours      TimeIntervalFilter      MO MT  <TimeIntervalFilter (08:00:00,
↪18:00:00)>
#TF                TransparentFilter       MO MT  <TransparentFilter>
#TG-Spain-Vodacom TagFilter                MO MT  <TG (tag=21401)>
#From20*           SourceAddrFilter        MO     <SourceAddrFilter (src_addr=^20\d+)>
Total Filters: 7
```

It is possible to obtain more information of a specific filter by typing **filter -s <fid>**:

```
jcli : filter -s September2014
DateIntervalFilter:
Left border = 2014-09-01
Right border = 2014-09-30
```

More control commands:

- **filter -r <fid>**: Remove filter

External business logic

In addition to predefined filters listed above (*Filter manager*), it is possible to extend filtering with external scripts written in Python using the **EvalPyFilter**.

Here's a very simple example where an **EvalPyFilter** is matching the connector **cid** of a message:

First, write an external python script:

```
# File @ /opt/jasmin-scripts/routing/abc-connector.py
if routable.connector.cid == 'abc':
    result = True
else:
    result = False
```

Second, create an EvalPyFilter with the python script:

```
jcli : filter -a
Adding a new Filter: (ok: save, ko: exit)
> type EvalPyFilter
jasmin.routing.Filters.EvalPyFilter arguments:
pyCode
> pyCode /opt/jasmin-scripts/routing/abc-connector.py
> fid SimpleThirdParty
> ok
Successfully added Filter [EvalPyFilter] with fid:SimpleThirdParty
```

This example will provide an **EvalPyFilter** (SimpleThirdParty) that will match any message coming from the connector with **cid** = abc.

Using **EvalPyFilter** is as simple as the shown example, when the python script is called it will get the following global variables set:

- **routable**: one of the *jasmin.routing.Routables.Routable* inheriters (*Routable* for more details)
- **result**: (default to *False*) It will be read by Jasmin router at the end of the script execution to check if the filter is matching the message passed through the routable variable, matched=True / unmatched=False

Note: It is possible to check for any parameter of the SMPP PDU: TON, NPI, PROTOCOL_ID ... since it is provided through the **routable** object.

Note: Using **EvalPyFilter** offers the possibility to call external webservices, databases ... for powerfull routing or even for logging, rating & billing through external third party systems.

Hint: More examples in the this FAQ's question: *Can you provide an example of how to use EvalPyFilter ?*

3.11.9 HTTP Client connector manager

The HTTP Client connector manager module is accessible through the **httpccm** command and is providing the following features:

Table 3.40: **httpccm** command line options

Command	Description
-l, –list	List HTTP client connectors
-a, –add	Add a new HTTP client connector
-r FID, –remove=FID	Remove HTTP client connector using it's CID
-s FID, –show=FID	Show HTTP client connector using it's CID

A HTTP Client connector is used in SMS-MO routing, it is called with the message parameters when it is returned by a matched MO Route (*Receiving SMS-MO* for more details).

When adding a HTTP Client connector, the following parameters are required:

- **cid**: Connector id (must be unique)
- **url**: URL to be called with message parameters
- **method**: Calling method (GET or POST)

Here's an example of adding a new HTTP Client connector:

```
jcli : httpccm -a
Adding a new Httpcc: (ok: save, ko: exit)
> url http://10.10.20.125/receive-sms/mo.php
> method GET
> cid HTTP-01
> ok
Successfully added Httpcc [HttpConnector] with cid:HTTP-01
```

All the above parameters can be displayed after Connector creation:

```
jcli : httpccm -s HTTP-01
HttpConnector:
cid = HTTP-01
```

```
baseurl = http://10.10.20.125/receive-sms/mo.php
method = GET
```

Listing Connectors will show currently added Connectors with their CID, Type, Method and Url:

```
jcli : httpccm -l
#Httpcc id          Type                    Method URL
#HTTP-01            HttpConnector           GET    http://10.10.20.125/receive-sms/mo.php
Total Httpccs: 1
```

3.11.10 Stats manager

The Stats manager module is responsible for showing real time statistics, aggregated counters and values such as current bound connections of a User, number of http requests, number of sent messages through a Route, Filter, Connector ...

Note: All values are collected during Jasmin's uptime and they are lost when Jasmin goes off, Stats manager shall be used for monitoring activities but not for advanced business reports.

The Stats manager module is accessible through the **stats** command and is providing the following features:

Table 3.41: **stats** command line options

Command	Description
–user=UID	Show user stats using it's UID
–users	Show all users stats
–smppc=CID	Show smpp connector stats using it's CID
–smppcs	Show all smpp connectors stats
–smppsapi	Show SMPP Server API stats

The Stats manager covers different sections, this includes Users, SMPP Client connectors, Routes (MO and MT), APIs (HTTP and SMPP).

User statistics

The Stats manager exposes an overall view of all existent users as well as a per-user information view:

- **stats –users**: Will show an overall view of all existent users
- **stats –user foo**: Will show detailed information for **foo**

Here's an example of showing an overall view where users **sandra** and **foo** are actually having 2 and 6 SMPP bound connections, user **bar** is using the HTTP Api only and **sandra** is using both APIs:

```
jcli : stats --users
#User id  SMPP Bound connections  SMPP L.A.            HTTP requests counter  HTTP L.
↪A.
#sandra   2                      2019-06-02 15:35:01  20                     2019-06-
↪01 12:12:33
#foo      6                      2019-06-02 15:35:10  0                      ND
#bar      0                      ND                   1289                   2019-06-
↪02 15:39:12
Total users: 3
```

The columns shown for each user are explained in the following table:

Table 3.42: Columns of the overall statistics for users

Column	Description
SMPP Bound connections	Number of current bound SMPP connections
SMPP L.A.	SMPP Server Last Activity date & time
HTTP requests counter	Counter of all http requests done by the user
HTTP L.A.	HTTP Api Last Activity date & time

Here's an example of showing **sandra**'s detailed statistics:

```
jcli : stats --user sandra
#Item                      Type         Value
#bind_count                SMPP Server  26
#submit_sm_count           SMPP Server  1500
#submit_sm_request_count   SMPP Server  1506
#unbind_count              SMPP Server  24
#data_sm_count             SMPP Server  0
#last_activity_at          SMPP Server  2019-06-02 15:35:01
#other_submit_error_count  SMPP Server  4
#throttling_error_count    SMPP Server  2
#bound_connections_count   SMPP Server  {'bind_transmitter': 1, 'bind_receiver': 1,
↪'bind_transceiver': 0}
#elink_count               SMPP Server  16
#qos_last_submit_sm_at     SMPP Server  2019-06-02 12:31:23
#deliver_sm_count          SMPP Server  1430
#connects_count            HTTP Api     156
#last_activity_at          HTTP Api     2019-06-01 12:12:33
#rate_request_count        HTTP Api     20
#submit_sm_request_count   HTTP Api     102
#qos_last_submit_sm_at     HTTP Api     2019-05-22 15:56:02
#balance_request_count     HTTP Api     16
```

This is clearly a more detailed view for user **sandra**, the following table explains the items shown for **sandra**:

Table 3.43: Details user statistics view items

Item	Type	Description
last_activity_at	SMPP Server	Date & time of last received PDU from user
bind_count	SMPP Server	Binds counter value
bound_connections_count	SMPP Server	Currently bound connections
submit_sm_request_count	SMPP Server	Number of requested SubmitSM (MT messages)
submit_sm_count	SMPP Server	Number of SubmitSM (MT messages) *really* sent by user
throttling_error_count	SMPP Server	Throttling errors received by user
other_submit_error_count	SMPP Server	Any other error received in response of SubmitSM requests
elink_count	SMPP Server	Number of enquire_link PDUs sent by user
deliver_sm_count	SMPP Server	Number of DeliverSM (MO messages or receipts) received
data_sm_count	SMPP Server	Number of DataSM (MO messages or receipts) received
qos_last_submit_sm_at	SMPP Server	Date & time of last SubmitSM (MT Message) sent
unbind_count	SMPP Server	Unbinds counter value
qos_last_submit_sm_at	HTTP Api	Date & time of last SubmitSM (MT Message sent)
connects_count	HTTP Api	HTTP request counter value
last_activity_at	HTTP Api	Date & time of last HTTP request
submit_sm_request_count	HTTP Api	Number of SubmitSM (MT messages) sent
rate_request_count	HTTP Api	Number of rate requests
balance_request_count	HTTP Api	Number of balance requests

SMPP Client connectors statistics

The Stats manager exposes an overall view of all existent smppc connectors as well as a per-smppc information view:

- **stats –smppcs**: Will show an overall view of all existent smppc connectors
- **stats –smppc foo**: Will show detailed information for **foo**

Here's an example of showing an overall view where smppc connectors **MTN** and **ORANGE** are actives, connector **SFONE** made no activity at all:

```
jcli : stats --smppcs
#Connector id  Connected at Bound at             Disconnected at     Submits Delivers␣
↪QoS errs Other errs
#MTN           6             2019-06-02 15:35:01  2019-06-02 15:35:01 12/10    9/10     ␣
↪2         0
#Orange        1             2019-06-02 15:35:01  2019-06-02 15:35:01 0/0      12022/0  ␣
↪0         0
#SFONE         0             ND                   ND                  0/0      0/0      ␣
↪0         0
Total connectors: 3
```

The columns shown for each user are explained in the following table:

Table 3.44: Columns of the overall statistics for smppcs

Column	Description
Bound count	Binds counter value
Connected at	Last connection date & time
Bound at	Last successful bind date & time
Disconnected at	Last disconnection date & time
Submits	Number of requested SubmitSM PDUs / Sent SubmitSM PDUs
Delivers	Number of received DeliverSM PDUs / Number of received DataSM PDUs
QoS errs	Number of rejected SubmitSM PDUs due to throttling limitation
Other errs	Number of all other rejections of SubmitSM PDUs

Here's an example of showing **MTN**'s detailed statistics:

```
jcli : stats --smppc MTN
#Item                       Value
#bound_at                   2019-06-02 15:35:01
#disconnected_count         2
#other_submit_error_count   0
#submit_sm_count            2300
#created_at                 2019-06-01 12:29:42
#bound_count                3
#last_received_elink_at     2019-06-02 15:32:28
#elink_count                34
#throttling_error_count     44
#last_sent_elink_at         2019-06-02 15:34:57
#connected_count            3
#connected_at               2019-06-02 15:35:01
#deliver_sm_count           1302
#data_sm_count              0
#submit_sm_request_count    2344
#last_seqNum                1733
#last_seqNum_at             2019-06-02 15:35:57
#last_sent_pdu_at           2019-06-02 15:35:59
#disconnected_at            2019-06-01 10:18:21
#last_received_pdu_at       2019-06-02 15:36:01
#interceptor_count          0
#interceptor_error_count    0
```

This is clearly a more detailed view for connector **MTN**, the following table explains the items shown for **MTN**:

Table 3.45: Details of smppc statistics view items

Item	Description
created_at	Connector creation date & time
last_received_pdu_at	Date & time of last received PDU
last_sent_pdu_at	Date & time of last sent PDU
last_received_elink_at	Date & time of last received enquire_link PDU
last_sent_elink_at	Date & time of last sent enquire_link PDU
last_seqNum_at	Date & time of last sequence_number claim
last_seqNum	Value of last claimed sequence_number
connected_at	Last connection date & time
bound_at	Last successful bind date & time
disconnected_at	Last disconnection date & time
connected_count	Last connection date & time
bound_count	Binds counter value
disconnected_count	Last disconnection date & time
submit_sm_request_count	Number of requested SubmitSM (MT messages)
submit_sm_count	Number of SubmitSM (MT messages) *really* sent (having **ESME_ROK** response)
throttling_error_count	Throttling errors received
other_submit_error_count	Any other error received in response of SubmitSM requests
elink_count	Number of enquire_link PDUs sent
deliver_sm_count	Number of DeliverSM (MO messages or receipts) received
data_sm_count	Number of DataSM (MO messages or receipts) received
interceptor_count	Number of successfully intercepted messages (MO)
interceptor_error_count	Number of failures when intercepting messages (MO)

SMPP Server API statistics

The Stats manager exposes collected statistics in SMPP Server API through the following *jCli* command:

- **stats –smppsapi**

Here's an example of showing the statistics:

```
jcli : stats --smppsapi
#Item                       Value
#disconnect_count           2
#bound_rx_count             1
#bound_tx_count             0
#other_submit_error_count   0
#bind_rx_count              0
#bind_trx_count             0
#created_at                 2019-06-04 02:22:17
#last_received_elink_at     ND
#elink_count                89
#throttling_error_count     1
#submit_sm_count            199
#connected_count            2
#connect_count              16
#bound_trx_count            1
#data_sm_count              2
#submit_sm_request_count    200
#deliver_sm_count           145
#last_sent_pdu_at           2019-06-05 12:12:13
#unbind_count               6
```

```
#last_received_pdu_at      2019-06-05 12:16:21
#bind_tx_count             6
#interceptor_count         0
#interceptor_error_count   0
```

The following table explains the items shown in the above example:

Table 3.46: Details of smppsapi statistics view items

Item	Description
created_at	Connector creation date & time
last_received_pdu_at	Date & time of last received PDU
last_sent_pdu_at	Date & time of last sent PDU
last_received_elink_at	Date & time of last received enquire_link PDU
connected_count	Last connection date & time
connect_count	TCP Connection request count
disconnect_count	Disconnection count
bind_trx_count	Transceiver bind request count
bound_trx_count	Actually bound transceiver connections count
bind_rx_count	Receiver bind request count
bound_rx_count	Actually bound receiver connections count
bind_tx_count	Transmitter bind request count
bound_tx_count	Actually bound transmitter connections count
submit_sm_request_count	Number of requested SubmitSM (MT messages)
submit_sm_count	Number of SubmitSM (MT messages) accepted (returned a **ESME_ROK** response)
deliver_sm_count	Number of DeliverSM (MO messages or receipts) sent
data_sm_count	Number of DataSM (MO messages or receipts) sent
elink_count	Number of enquire_link PDUs received
throttling_error_count	Throttling errors returned
other_submit_error_count	Any other error returned in response of SubmitSM requests
interceptor_count	Number of successfully intercepted messages (MT)
interceptor_error_count	Number of failures when intercepting messages (MT)

HTTP API statistics

The Stats manager exposes collected statistics in HTTP API through the following *jCli* command:

- **stats –httpapi**

Here's an example of showing the statistics:

```
jcli : stats --httpapi
#Item                    Value
#server_error_count      120
#last_request_at         ND
#throughput_error_count  4
#success_count           14332
#route_error_count       156
#request_count           20126
#auth_error_count        78
#created_at              2019-06-04 02:22:17
#last_success_at         2019-06-05 18:20:29
#charging_error_count    178
```

```
#interceptor_count          0
#interceptor_error_count    0
```

The following table explains the items shown in the above example:

Table 3.47: Details of httpapi statistics view items

Item	Description
created_at	Connector creation date & time
last_request_at	Date & time of last http request
last_success_at	Date & time of last successful http request (SMS is accepted for sending)
request_count	HTTP request count
success_count	Successful HTTP request count (SMS is accepted for sending)
auth_error_count	Authentication errors count
route_error_count	Route not found errors count
throughput_error_count	Throughput exceeded errors count
charging_error_count	Charging/Billing errors count
server_error_count	Unknown server errors count
interceptor_count	Number of successfully intercepted messages (MT)
interceptor_error_count	Number of failures when intercepting messages (MT)

3.12 Billing

Jasmin comes with a user billing feature that lets you apply rates on message routes, every time a user sends a SMS through a rated route he'll get charged, once he runs out of credit no more sending will be permitted.

Important: New routes created through *MT router manager* are not rated by default, you must *define* the rate of each route in order to enable billing.

Note: Billing is applied on all channels (SMPP Server and HTTP API) the same way.

3.12.1 Billing quotas

A user can be charged through 2 types of quotas (balance and/or sms_count), if he reaches the limit of one of these quotas no more sending will be permitted, no matter the used channel (SMPP Server or HTTP API).

1. Balance quota

The route rate will be charged on the user balance, let's get into these use cases for better comprehension:

- When sending one SMS through a route rated **1.2**, user's balance will get decreased by **1.2**
- When sending five SMS through a route rated **0.2**, user's balance will get decreased by **1**

Important: New users created through *User manager* will have unlimited balance by default, assuming you'll apply postpaid billing (*or no billing at all*), user's balance must be *defined* in order to enable billing.

Rate unit

You can see that the rates have no *unit* or *currency*, this will offer better flexibility for different business cases, you can consider the rates as:

- Local Jasmin currency and keep a rate for converting to real-life currency.
- Real-life currency
- etc ..

In all cases, Jasmin will never manage the rate *unit* (or *currency*), all it does is to ensure users are correctly charged by the rates you define.

Asynchronous billing

As explained *later*, it is important to know that whatever the used protocol, SMS is always sent **asynchronously**, this means there's always an acknowledgment to be received for every sent SMS; Jasmin provides an *optional* adapted billing *algorithm* which is able to charge the user **asynchronously**:

1. A defined percentage of the route rate is charged when the user submits the SMS for sending.
2. The rest is charged when the SMS is acknowledged by the next relay, in SMPP protocol, this means receiving **SUBMIT_SM_RESP** PDU, more details *here*.

Asynchronous billing is automatically enabled when the user have **early_decrement_balance_percent** *defined* (undefined by default), let's get back to examples for better comprehension, assuming user have **early_decrement_balance_percent = 25**:

- When sending one SMS through a route rated **1.2**:
- When sending, user's balance is decreased by **0.3** (*1.2* x **25%**)
- When acknowledged, user's balance is decreased by **0.9** (the rest)
- When sending **five** SMS through a route rated **0.2**:
- When sending, user's balance is decreased by **0.25** (**5** x *0.2* x **25%**)
- For each acknowledged SMS, user's balance is decreased by **0.15**
- When all **five** sent messages are acknowledged, the final charged amount is **0.75** (the rest)

Using asynchronous billing can be helpful in many use cases:

- Charge only when the SMS is acknowledged
- If SMS is not acknowledged for some reason, user can not fill Jasmin's queues by SMS requests indefinitely, he'll get out of credits
- etc ..

2. sms_count quota

Simpler than *Balance* management, *sms_count* is a counter to be decreased whenever the user submits the SMS for sending, let's get into these use cases for better comprehension:

- When sending one SMS through a route, user's *sms_count* will get decreased by **1**
- When sending five SMS through a route, user's *sms_count* will get decreased by **5**

Note: When defined, *sms_count* is always decreased no matter the route is rated or not.

Important: New users created through *User manager* will have unlimited sms_count by default, assuming you'll apply postpaid billing (*or no billing at all*), user's *sms_count* must be *defined* in order to enable billing (or limit).

3.12.2 Process flow

The following process flow shows how billing is done through HTTP Api (same process is applied on SMPP Server), it is including all types of billing:

- balance quota billing (*ref*) including asynchronous billing (*ref*)
- sms_count quota billing (*ref*)

Asynchronous billing call flow

When enabled, *Asynchronous billing* algorithm can charge user every time an acknowledgment is received for each SMS he sent earlier, the following call flow explain the asynchronous billing algorithm:

In the above figure, user is charged early before submitting SMS to SMSC, and the charged later when the SMSC acknowledge back reception of the message, as detailed *earlier*, the charged amount in early stage is defined by **early_decrement_balance_percent** *set in user profile*.

Note: The route rate is expressed on a per-SUBMIT_SM basis, submitting a long SMS will be splitted into multiple **submit_sm SMPP PDUs**, each one will be charged on user.

The below figure explain how asynchronous billing is handling long content messages, assuming a user is submitting a message containing 400 characters, which will imply sending 3 **submit_sm SMPP PDUs**:

Asynchronous billing is mainly relying on AMQP broker (like *messaging*), The AMQP broker is providing a queuing mechanism, through the following illustration you can see how asynchronous billing is done:

When receiving a **SUBMIT_SM_RESP** PDU, *submit_sm_resp_event()* method is called (*more details here*), it will check if there's a remaining bill to charge on user and publish it on **bill_request.submit_sm_resp.UID** (using *billing* exchange) where UID is the concerned User ID.

RouterPB's *bill_request_submit_sm_resp_callback()* is listening on the same topic and it will be fired whenever it consumes a new bill request, as the Router is holding User objects in memory, it will simply update their balances with the bill amount.

Jasmin is doing everything in-memory for performance reasons, including User charging where the balance must be persisted to disk for later synchronization whenever Jasmin is restarted, this is why RouterPB is automatically persisting Users and Groups to disk every **persistence_timer_secs** seconds as defined in jasmin.cfg file (INI format, located in /etc/jasmin).

Important: Set **persistence_timer_secs** to a reasonable value, keep in mind that every disk-access operation will cost you few performance points, and don't set it too high as you can loose Users balance data updates.

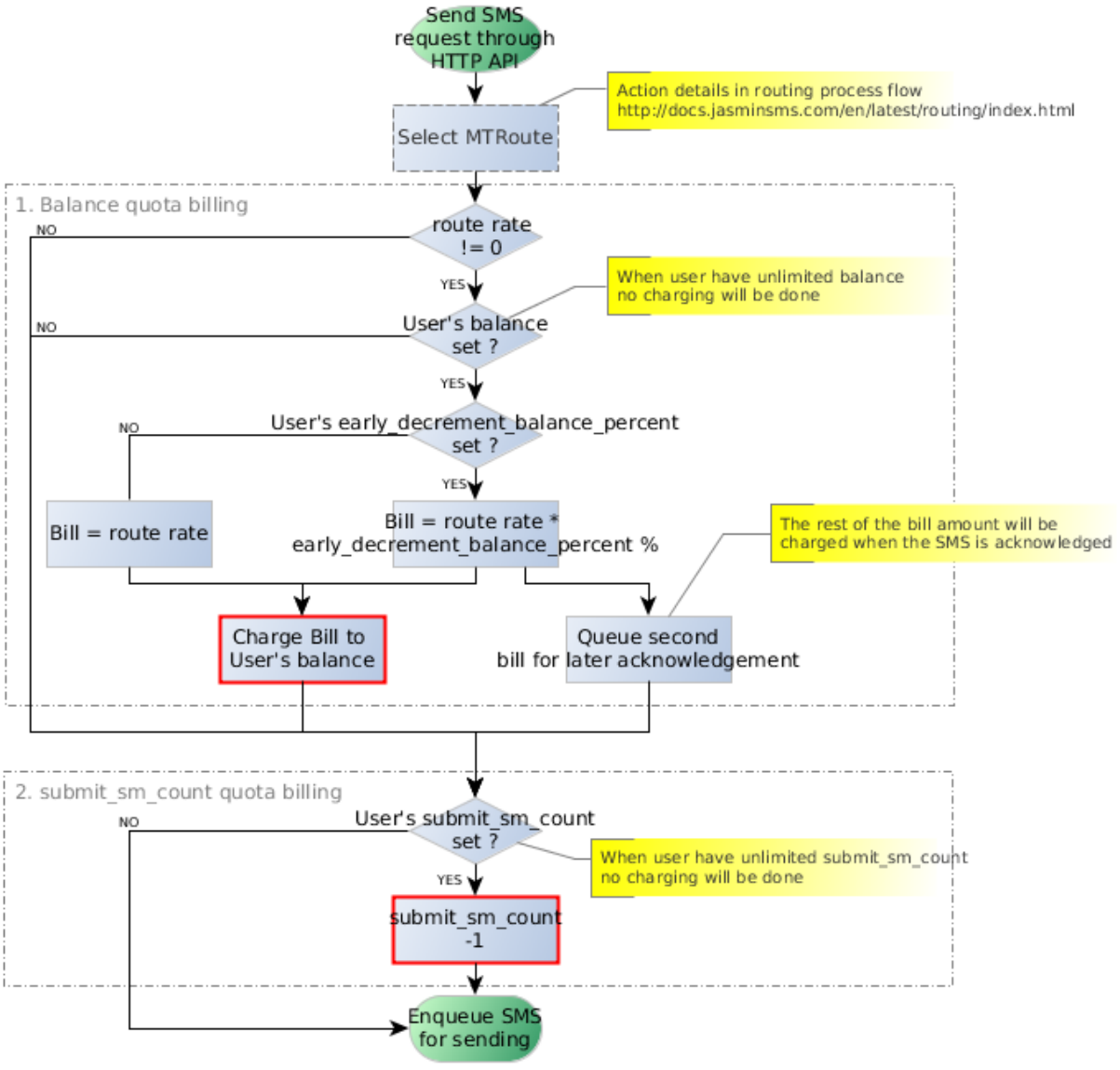


Fig. 3.10: Billing process flow

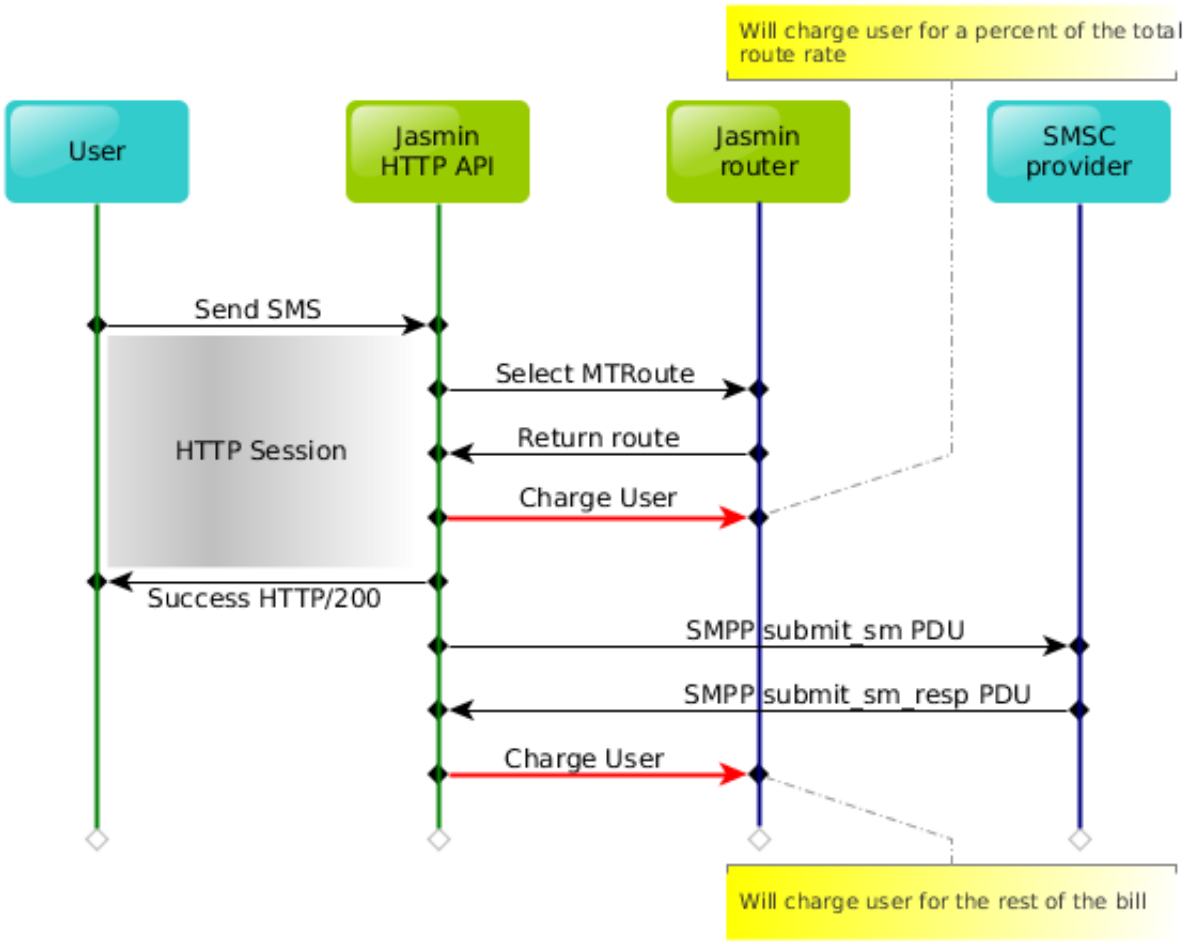


Fig. 3.11: Asynchronous billing call flow

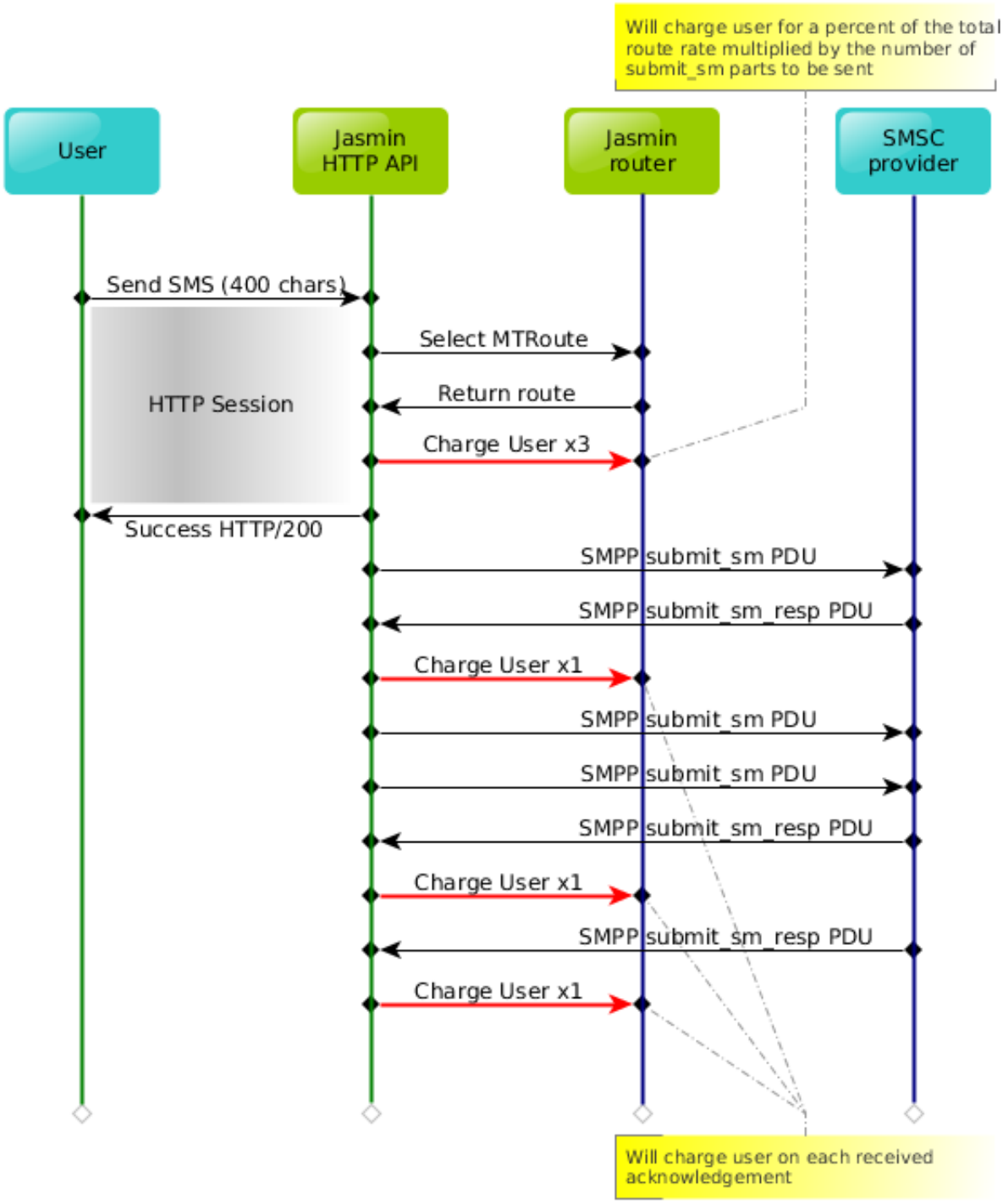


Fig. 3.12: Asynchronous billing call flow for long content messages

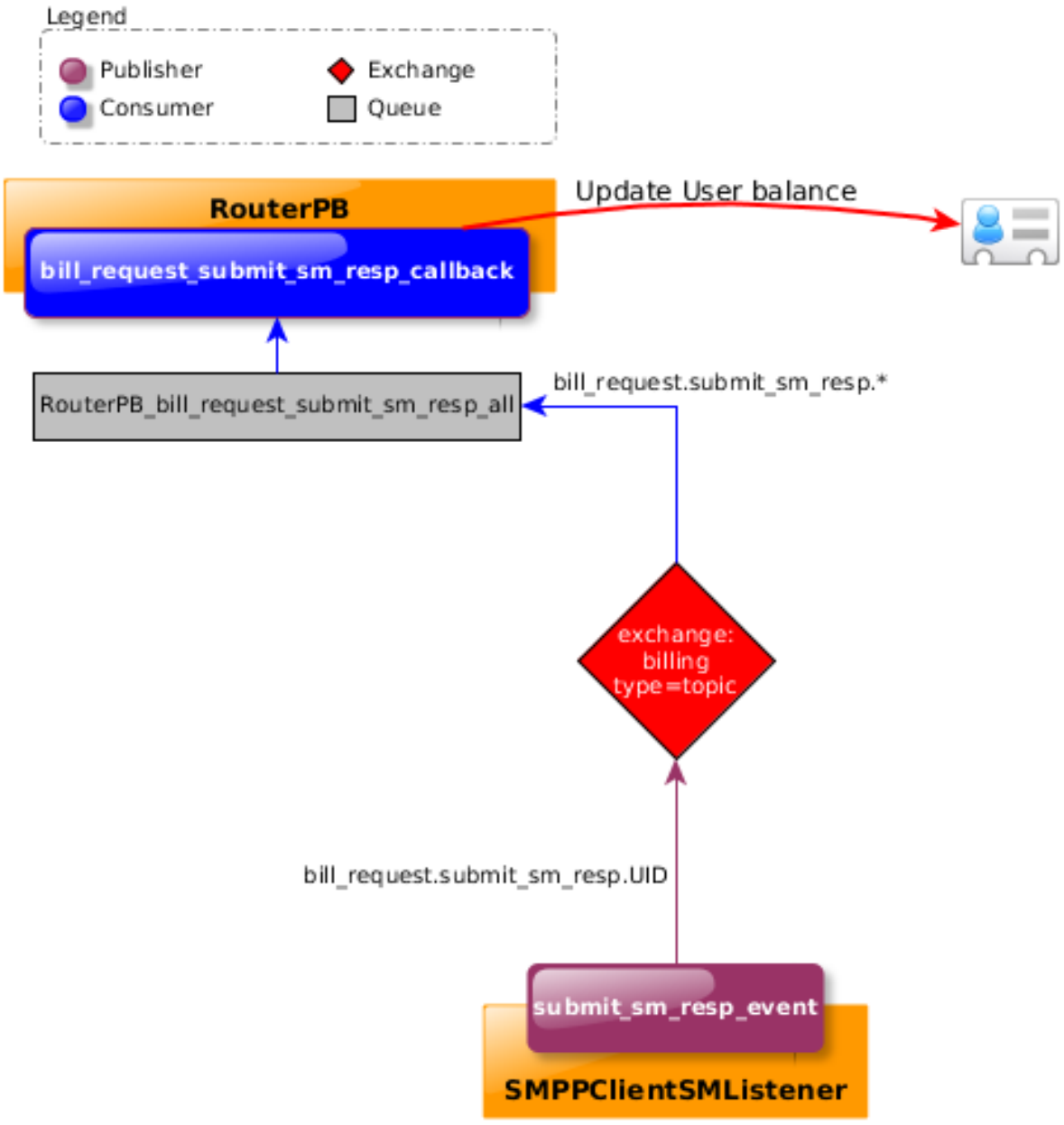


Fig. 3.13: Asynchronous billing AMQP message exchange

3.13 Messaging flows

Messaging is heavily relying on an AMQP broker using topics to queue messages for routing, delivering and acking back.

The AMQP broker is providing a strong store & forward queuing mechanism, through the following illustration you can see how every messaging component is asynchronously connected to the broker.

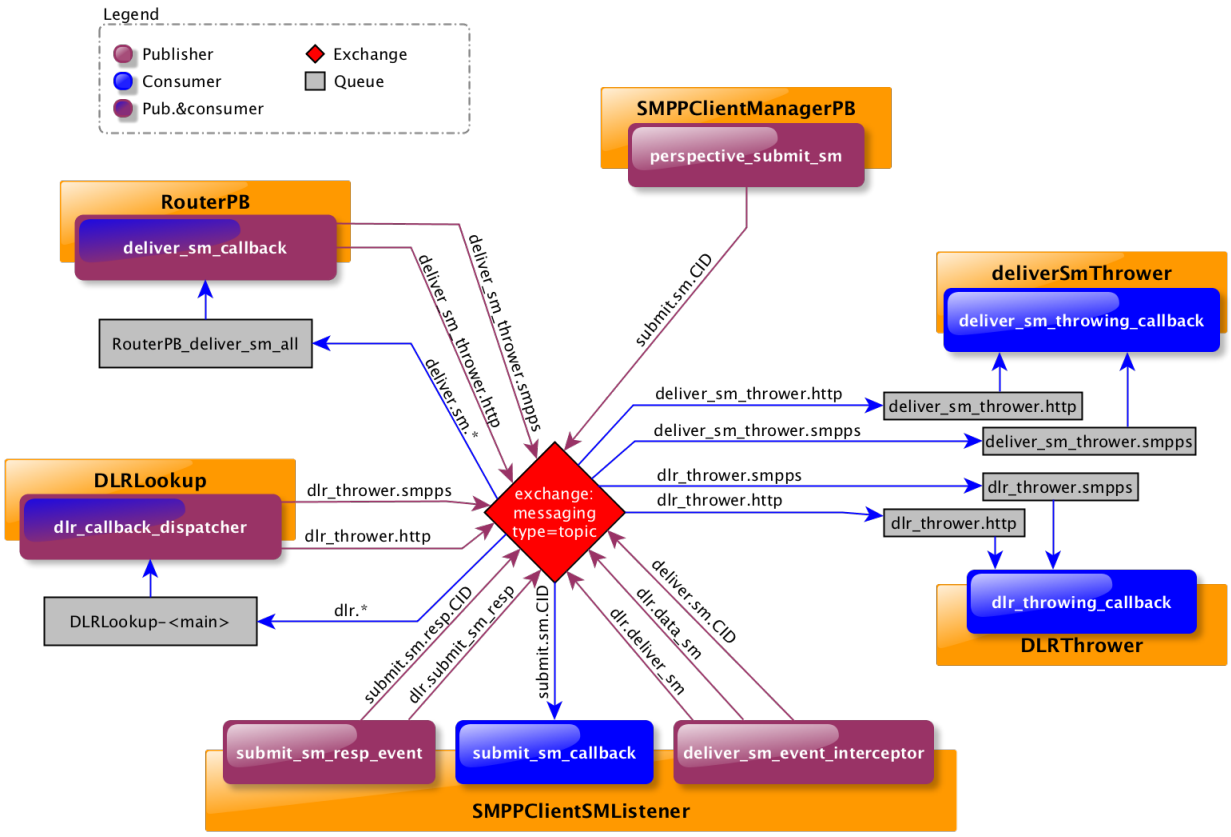


Fig. 3.14: AMQP Messaging flows

Six main actors are messaging through the "*messaging*" topic, their business logic are explained in the below paragraphs.

3.13.1 SMPPClientManagerPB

This is a *PerspectiveBroker* (PB) responsible of managing SMPP Client connectors (list, add, remove, start, stop, send SMS, etc ...), we'll be only covering the latter (Send SMS).

When the **perspective_submit_sm()** is called with a SubmitSm PDU and destination connector ID, it will build an AMQP Content message and publish it to a queue named **submit.sm.CID** where *CID* is the destination connector ID.

Note: **perspective_submit_sm()** is called from HTTP API and SMPP Server API after they check with RouterPB for the right connector to send a SubmitSM to.

Every SMPP Connector have a consumer waiting for these messages, once published as explained above, it will be consumed by the destination connector's **submit_sm_callback()** method (c.f. *SMPPClientSMListener*).

3.13.2 DLRLookup

This is a consumer on the **dlr.*** AMQP route, added in *v0.9*, it's main role is DLR map fetching from Redis database and publishing the dlr to the right thrower (http or smpp).

3.13.3 RouterPB

This is another *PerspectiveBroker* (PB) responsible of routing DeliverSm messages, these are received through the SMPP client connector's *deliver_sm_event_interceptor()* method (c.f. *SMPPClientSMListener*) which publish to **deliver.sm.CID**, the RouterPB main role is to decide whether to route DeliverSm messages to:

- **deliver_sm_thrower.smpps**: if the message is to be delivered through SMPP Server API.
- **deliver_sm_thrower.http**: if the message is to be delivered through a HTTP connector.

3.13.4 SMPPClientSMListener

Every SMPP Client connector have one attached *SMPPClientSMListener* instance, it is responsible for handling messages exchanged through the SMPP Client connector using the following event catchers:

deliver_sm_event_interceptor

Every received DeliverSm PDU is published directly to the broker with the following assumptions:

- If it's a SMS-MO message it will get published as an AMQP Content message to **deliver_sm.CID** where *CID* is the source connector ID, this message will be handled by the *RouterPB*.
- If it's a delivery receipt and if it were requested when sending the SubmitSm, it will get published as an AMQP Content message to **dlr_thrower.http** or **dlr_thrower.smpps** (depends on the used channel for sending initial SubmitSM) for later delivery by DLRThrower's **dlr_throwing_callback()** method.

Note: **deliver_sm_event_interceptor()** will check for interception rules before proceding to routing, c.f. *Interception* for more details.

---

submit_sm_callback

It is a simple consumer of **submit.sm.CID** where *CID* is its connector ID, it will send every message received through SMPP connection.

submit_sm_resp_event

It is called for every received SubmitSmResp PDU, will check if the related SubmitSm was requiring a delivery receipt and will publish it (or not) to **dlr_thrower.http** or **dlr_thrower.smpps** (depends on the used channel for sending initial SubmitSM).

Note: There's no actual reason why messages are published to **submit.sm.resp.CID**, this may change in future.

3.13.5 deliverSmThrower

This is will through any received message from **deliver_sm_thrower.http** to its final http connector, c.f. *Receiving SMS-MO* for details and from **deliver_sm_thrower.smpps** to its final SMPP Server binding.

3.13.6 DLRThrower

This is will through any received delivery receipt from **dlr_thrower.http** to its final http connector, c.f. *Receiving DLR* for details and from **dlr_thrower.smpps** to its final SMPP Server binding.

3.14 User FAQ

3.14.1 Could not find a version that satisfies the requirement jasmin

Installing Jasmin using **pip** will through this error:

```
$ sudo pip install python-jasmin
[sudo] password for richard:
Downloading/unpacking jasmin
    Could not find a version that satisfies the requirement jasmin (from versions: 0.
↪6b1, 0.6b10, 0.6b11, 0.6b12, 0.6b13, 0.6b14, 0.6b2, 0.6b3, 0.6b4, 0.6b5, 0.6b6, 0.
↪6b7, 0.6b8, 0.6b9)
Cleaning up...
No distributions matching the version for jasmin
Storing debug log for failure in /home/richard/.pip/pip.log
```

This is common question, since Jasmin is still tagged as a 'Beta' version, pip installation must be done with the **–pre** parameter:

```
$ sudo pip install --pre python-jasmin
...
```

Hint: This is clearly documented in *Installation* installation steps.

3.14.2 Cannot connect to telnet console after starting Jasmin

According to the installation guide, Jasmin requires running RabbitMQ and Redis servers, when starting it will wait for these servers to go up.

If you already have these requirements, please check jcli and redis-client logs:

- /var/log/jasmin/redis-client.log
- /var/log/jasmin/jcli.log

Hint: Please check *Prerequisites & Dependencies* before installing.

3.14.3 Should i expose my SMPP Server & HTTP API to the public internet for remote users ?

As a security best practice, place *Jasmin* instance(s) behind a firewall and apply whitelisting rules to only accept users you already know, a better solution is to get VPN tunnels with your users.

If for some reasons you cannot consider these practices, here's a simple iptables configuration that can help to prevent Denial-of-service attacks:

```
iptables -I INPUT -p tcp --dport 2775 -m state --state NEW -m recent --set --name␣
↪SMPP_CONNECT
iptables -N RULE_SMPP
iptables -I INPUT -p tcp --dport 2775 -m state --state NEW -m recent --update --
↪seconds 60 --hitcount 3 --name SMPP_CONNECT -j RULE_SMPP
iptables -A RULE_SMPP -j LOG --log-prefix 'DROPPED SMPP CONNECT ' --log-level 7
iptables -A RULE_SMPP -j DROP
```

This will drop any SMPP Connection request coming from the same source IP with more than 3 times per minute ...

3.14.4 Does Jasmin persist its configuration to disk ?

Since everything in Jasmin runs fully in-memory, what will happen if i restart Jasmin or if it crashes for some reason ? how can i ensure my configuration (Connectors, Users, Routes, Filters ...) will be reloaded with the same state they were in before Jasmin goes off ?

Jasmin is doing everything in-memory for performance reasons, and is automatically persisting newly updated configurations every **persistence_timer_secs** seconds as defined in jasmin.cfg file.

Important: Set **persistence_timer_secs** to a reasonable value, keep in mind that every disk-access operation will cost you few performance points, and don't set it too high as you can loose critical updates such as User balance updates.

3.14.5 When receiving a DLR: Got a DLR for an unknown message id

The following error may appear in **messages.log** while receiving a receipt (DLR):

```
WARNING  4403 Got a DLR for an unknown message id: 788821
```

This issue can be caused by one of these:

- The receipt is received and it indicates a message id that did not get sent by Jasmin,
- The receipt is received for a message sent by Jasmin, but message id is not recognize, if it's the case then find below what you can do.

What's happening:

When sending a message (**submit_sm**) the upstream connector will reply back with a first receipt (**submit_sm_resp**) where it indicates the message id for further tracking, then it will send back another receipt (**deliver_sm** or **data_sm**) with the same message it and different delivery state. The problem occurs when the upstream connector returns the same message id but in different encodings.

Solution:

Use the **dlr_msgid** parameter as shown in *SMPP Client connector manager* to indicate the encoding strategy of the upstream partner/connector.

3.15 Developer FAQ

3.15.1 How to 'log' messages in a third party database ?

Jasmin runs without a database, everything is in-memory and messages are exchanged through AMQP broker (RabbitMQ), if you need to get these messages you have to consume from the right queues as described in *Messaging flows*.

Here's an example:

Thanks to Pedro's contribution:

```
Here is the PySQLPool mod to @farirat ´s gist
https://gist.github.com/pguillem/5750e8db352f001138f2

Here is the code to launch the consumer as a system Daemon in Debian/Ubuntu
https://gist.github.com/pguillem/19693defb3feb0c02fe7

1) create jasmind_consumer file in /etc/init.d/
2) chmod a+x
3) Modify the path and script name of your consumer in jasmind_consumer
4) Remember to exec "update-rc.d jasmind_consumer defaults" in order to start at boot

Cheers
Pedro
```

More on this:

```
# Gist from https://gist.github.com/farirat/5701d71bf6e404d17cb4
import cPickle as pickle
from twisted.internet.defer import inlineCallbacks
from twisted.internet import reactor
from twisted.internet.protocol import ClientCreator
from twisted.python import log

from txamqp.protocol import AMQClient
from txamqp.client import TwistedDelegate

import txamqp.spec

@inlineCallbacks
def gotConnection(conn, username, password):
    print "Connected to broker."
    yield conn.authenticate(username, password)

    print "Authenticated. Ready to receive messages"
    chan = yield conn.channel(1)
    yield chan.channel_open()

    yield chan.queue_declare(queue="someQueueName")

    # Bind to submit.sm.* and submit.sm.resp.* routes
    yield chan.queue_bind(queue="someQueueName", exchange="messaging", routing_key=
↪'submit.sm.*')
    yield chan.queue_bind(queue="someQueueName", exchange="messaging", routing_key=
↪'submit.sm.resp.*')

    yield chan.basic_consume(queue='someQueueName', no_ack=True, consumer_tag="someTag
↪")
```

```
    queue = yield conn.queue("someTag")

    # Wait for messages
    # This can be done through a callback ...
    while True:
        msg = yield queue.get()
        props = msg.content.properties
        pdu = pickle.loads(msg.content.body)

            if msg.routing_key[:15] == 'submit.sm.resp.':
                    print 'SubmitSMResp: status: %s, msgid: %s' % (pdu.status,
                            props['message-id'])
        elif msg.routing_key[:10] == 'submit.sm.':
                print 'SubmitSM: from %s to %s, content: %s, msgid: %s' % (pdu.params[
↪'source_addr'],
                        pdu.params['destination_addr'],
                        pdu.params['short_message'],
                        props['message-id'])
            else:
                    print 'unknown route'

    # A clean way to tear down and stop
    yield chan.basic_cancel("someTag")
    yield chan.channel_close()
    chan0 = yield conn.channel(0)
    yield chan0.connection_close()

    reactor.stop()


if __name__ == "__main__":
    """
    This example will connect to RabbitMQ broker and consume from two route keys:
      - submit.sm.*: All messages sent through SMPP Connectors
      - submit.sm.resp.*: More relevant than SubmitSM because it contains the sending␣
↪status

    Note:
      - Messages consumed from submit.sm.resp.* are not verbose enough, they contain␣
↪only message-id and status
      - Message content can be obtained from submit.sm.*, the message-id will be the␣
↪same when consuming from submit.sm.resp.*,
        it is used for mapping.
      - Billing information is contained in messages consumed from submit.sm.*
      - This is a proof of concept, saying anyone can consume from any topic in Jasmin
↪'s exchange hack a
        third party business, more information here: http://docs.jasminsms.com/en/
↪latest/messaging/index.html
    """

    host = '127.0.0.1'
    port = 5672
    vhost = '/'
    username = 'guest'
    password = 'guest'
    spec_file = '/etc/jasmin/resource/amqp0-9-1.xml'

    spec = txamqp.spec.load(spec_file)
```

```
    # Connect and authenticate
    d = ClientCreator(reactor,
            AMQClient,
            delegate=TwistedDelegate(),
            vhost=vhost,
        spec=spec).connectTCP(host, port)
    d.addCallback(gotConnection, username, password)

    def whoops(err):
        if reactor.running:
            log.err(err)
            reactor.stop()

    d.addErrback(whoops)

    reactor.run()
```

3.15.2 How to directly access the Perspective Broker API ?

Management tasks can be done directly when accessing PerspectiveBroker API, it will be possible to:

- Manage SMPP Client connectors,
- Check status of all connectors,
- Send SMS,
- Manage Users & Groups,
- Manage Routes (MO / MT),
- Access statistics,
- ...

Here's an example:

```
# Gist from https://gist.github.com/farirat/922e1cb2c4782660c257
"""
An example of scenario with the following actions:
 1. Add and start a SMPP Client connector
 2. Provision a DefaultRoute to that connector
 3. Provision a User

This is a demonstration of using PB (PerspectiveBroker) API to gain control Jasmin.

The jasmin SMS gateway shall be already running and having
a pb listening on 8989.
"""

import cPickle as pickle
from twisted.internet import reactor, defer
from jasmin.managers.proxies import SMPPClientManagerPBProxy
from jasmin.routing.proxies import RouterPBProxy
from jasmin.routing.Routes import DefaultRoute
from jasmin.routing.jasminApi import User, Group
from jasmin.protocols.smpp.configs import SMPPClientConfig
from jasmin.protocols.cli.smppccm import JCliSMPPClientConfig as SmppClientConnector
```

```
from twisted.web.client import getPage

@defer.inlineCallbacks
def runScenario():
    try:
        ## First part, SMPP Client connector management
        ##############################################
        # Connect to SMPP Client management PB proxy
        proxy_smpp = SMPPClientManagerPBProxy()
        yield proxy_smpp.connect('127.0.0.1', 8989, 'cmadmin', 'cmpwd')

        # Provision SMPPClientManagerPBProxy with a connector and start it
        connector1 = {'id':'abc', 'username':'smppclient1',
↪'reconnectOnConnectionFailure':True}
        config1 = SMPPClientConfig(**connector1)
        yield proxy_smpp.add(config1)
        yield proxy_smpp.start('abc')

        ## Second part, User and Routing management
        ##########################################
        # Connect to Router PB proxy
        proxy_router = RouterPBProxy()
        yield proxy_router.connect('127.0.0.1', 8988, 'radmin', 'rpwd')

        # Provision RouterPBProxy with MT routes
        yield proxy_router.mtroute_add(DefaultRoute(SmppClientConnector('abc')), 0)
        routes = yield proxy_router.mtroute_get_all()
        print "Configured routes: \n\t%s" % pickle.loads(routes)

        # Provisiong router with users
        g1 = Group(1)
        u1 = User(uid = 1, group = g1, username = 'foo', password = 'bar')
        yield proxy_router.group_add(g1)
        yield proxy_router.user_add(u1)
        users = yield proxy_router.user_get_all()
        print "Users: \n\t%s" % pickle.loads(users)

        ## Last, tear down
        ##################
        # Stop connector
        yield proxy_smpp.stop('abc')
    except Exception, e:
        print "ERROR RUNNING SCENARIO: %s" % str(e)
    finally:
        reactor.stop()

runScenario()
reactor.run()
```

3.15.3 Can you provide an example of how to use EvalPyFilter ?

Let's say you need your filter to pass only messages from username **foo**:

```
if routable.user.username == 'foo':
    result = False
else:
```

```
    result = True
```

Note: Although **UserFilter** is already there to provide this feature, this is just a simple example of using EvalPyFilter.

So your python script will have a **routable** global variable, it is an instance of **RoutableDeliverSm** if you're playing with a MO Route and it will be an instance of **RoutableSubmitSm** if you're considering it with a MT Route.

In order to implement your specific filter, you have to know all the attributes these objects are providing,

Now let's make an advanced example, the below filter will:

- Connect to a database
- Check if the message *destination_address* is in *blacklisted_numbers* table
- Pass only if the *destination_address* is not blacklisted

```
"""This is an example of using EvalPyFilter with a database interrogation, it is␣
↪written
for demonstration purpose only.
"""
import MySQLdb as mdb

destination_addr = routable.pdu.params['destination_addr']

try:
        con = mdb.connect('localhost', 'jasmin', 'somepassword', 'jasmin_faq');

        cur = con.cursor()
        cur.execute("SELECT COUNT(msisdn) FROM blacklisted_numbers WHERE msisdn = %s"
↪% destination_addr)
        count = cur.fetchone()

        if count[0] == 0:
                # It is not blacklisted, filter will pass
                result = True
except mdb.Error, e:
        # A DB error, filter will block
        # Error can be logged as well ...
        result = False
finally:
        # Filter will block for any other exception / reason
        result = False
```

3.15.4 How to log events inside an EvalPyFilter ?

It is a usual method to get the filter logging directly to the Router's log file (default is **router.log**), here's a very simple example of doing it:

```
import logging

log = logging.getLogger("jasmin-router")

log.debug('Inside evalpy-test.py')
if routable.user.username == 'Evalpyusr2':
    log.info("Routable's username is Evalpyusr2 !")
```

```
    result = False
else:
    log.info("Routable's username is not Evalpyusr2: %s" % routable.user.username)
    result = True
```

Note: More on python logging: here.

3.15.5 How to set an EvalPyFilter for a MT Route ?

I have written my *EvalPyFilter*, how can i use it to filter MT messages ?

Using jCli:

First, create your filter:

```
jcli : filter -a
Adding a new Filter: (ok: save, ko: exit)
> type evalpyfilter
> pyCode /some/path/advanced_evalpyfilter.py
> fid blacklist_check
> ok
Successfully added Filter [EvalPyFilter] with fid:blacklist_check
```

Second, create a MT Route:

```
jcli : mtrouter -a
Adding a new MT Route: (ok: save, ko: exit)
> type StaticMTRoute
jasmin.routing.Routes.StaticMTRoute arguments:
filters, connector, rate
> filters blacklist_check
> connector smppc(SOME-SMSC)
> rate 0.0
> order 10
> ok
Successfully added MTRoute [StaticMTRoute] with order:10
```

And you're done ! test your filter by sending a SMS through Jasmin's APIs.

3.15.6 PDU params keep resetting to connector defaults even after interception ?

When sending MT messages through httpapi, some pdu parameters will be reset to connector defaults even if they were manually updated inside an interceptor script, how can Jasmin avoid updatingmy pdu params ?

After updating a pdu parameter, it must be locked so Jasmin will not re-update it again, here's an example:

```
# Set pdu param:
routable.pdu.params['sm_default_msg_id'] = 10
# Lock it:
routable.lockPduParam('sm_default_msg_id')
```

Note: Locking pdu parameters is only needed when message is pushed from httpapi.

CHAPTER 4

Links

- Jasmin SMS Gateway home page
- Documentation
- Source code
- Travis CI

CHAPTER 5

License

Jasmin is released under the terms of the [Apache License Version 2]. See **'LICENSE'** file for details.