Vumi Documentation

Release 0.6.13

Praekelt Foundation

January 10, 2017

Contents

Contents:

CHAPTER 1

Vumi Overview

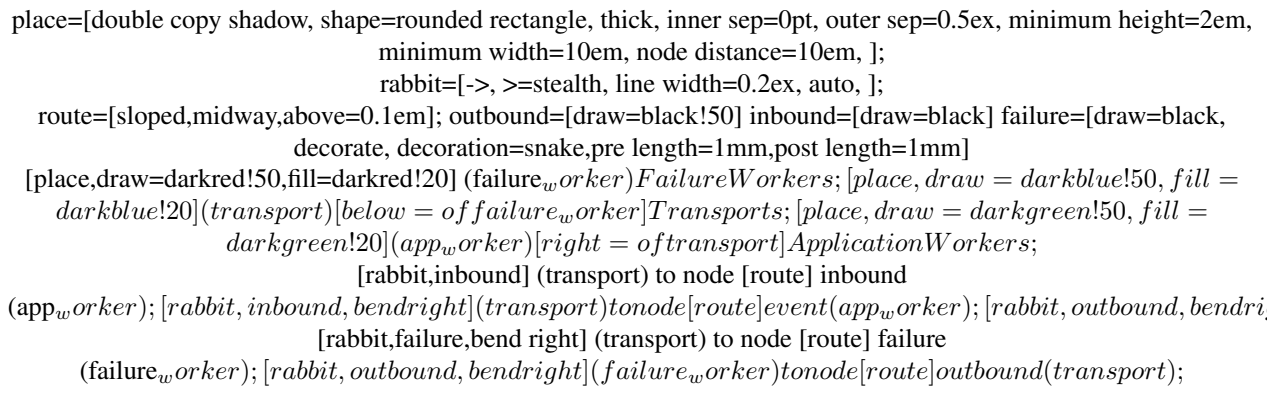


Fig. 1.1: A simple Vumi worker setup

CHAPTER 2

Vumi Tutorial

New to Vumi? Well, you came to the right place: read this material to quickly get up and running.

2.1 Writing your first Vumi app - Part 1

This is the first part in a series of tutorials demonstrating how to develop Vumi apps.

We'll assume you have a working knowledge of Python, RabbitMQ and VirtualEnv.

Where to get help

If you're having trouble at any point feel free to drop by #vumi on irc.freenode.net to chat with other Vumi users who might be able to help.

In this part of the tutorial we'll be creating and testing a simple working environment.

2.1.1 Environment Setup

Before we proceed let's create an isolated working environment using VirtualEnv.

From the command line `cd` into a directory where you'd like to store your code then run the following command:

```
$ virtualenv --no-site-packages ve
```

This will create a `ve` directory where any libraries you install will go, thus isolating your environment. Once the virtual environment has been created activate it by running `source ve/bin/activate`.

Note: For this to work VirtualEnv needs to be installed. You can tell it's installed by executing `virtualenv` from the command line. If that command runs successfully with no errors VirtualEnv is installed. If not you can install it by executing `sudo pip install virtualenv` from the command line.

Note: From this point onwards your virtual environment should always be active. The virtualenv is activated by running `source ve/bin/activate`.

Now that you created and activated the virtual environment install Vumi with the following command:

```
$ pip install -e git+git://github.com/praekelt/vumi.git@develop#egg=vumi
```

Note: This will install the development version of Vumi containing the latest-and-greatest features. Although the development branch is kept stable it is not recommended for production environments.

If this is your first Vumi application you need to take care of some initial RabbitMQ setup. Namely you need to add a `vumi` user and a `develop` virtual host and grant the required permissions. Vumi includes a script to do this for you which you can execute with the following command:

```
$ sudo ./ve/src/vumi/utils/rabbitmq.setup.sh
```

Note: Vumi workers communicate over RabbitMQ and requires it being installed and running. You can tell it's installed and its current status by executing `sudo rabbitmq-server` from the command line. If the command is not found you can install RabbitMQ by executing `sudo apt-get install rabbitmq-server` from the command line (assuming you are on a Debian based distribution).

2.1.2 Testing the Environment

Let's verify this worked. As a test you can create a Telnet worker and an *echo* application, both of which are included in Vumi.

Philosophy

A complete Vumi instance consists of a *transport worker* and an *application worker* which are managed as separate processes. A *transport worker* is responsible for sending messages to and receiving messages from some communications medium. An *application worker* processes messages received from a *transport worker* and generates replies.

Start the Telnet *transport worker* by executing the following command:

```
$ twistd -n --pidfile=transportworker.pid vumi_worker --worker-class vumi.transports.telnet.TelnetSe
```

This utilizes Twisted to start a Telnet process listening on port 9010. Specifically it uses Vumi's builtin `TelnetServerTransport` to handle communication with Telnet clients. Note that we specify `telnet` as the transport name when providing `--set-option=transport_name:telnet`. When starting the *application worker* as described next the same name should be used, thus connecting the *transport worker* with the *application worker*.

Philosophy

A *transport worker* is responsible for sending messages over and receiving messages from some communication medium. For this example we are using a very simple transport that communicates over Telnet. Other transport mechanisms Vumi supports include SMPP, XMPP, Twitter, IRC, HTTP and a variety of mobile network aggregator specific messaging protocols. In subsequent parts of this tutorial we'll be using the XMPP transport to communicate over Google Talk.

In a command line session you should now be able to connect to the *transport worker* via Telnet:

```
$ telnet localhost 9010
```

If you keep an eye on the *transport worker's* output you should see the following as clients connect:

```
2012-03-06 12:06:32+0200 [twisted.internet.protocol.ServerFactory] Registering client connected from
```

Note: At this point only the *transport worker* is running so Telnet input will not be processed yet. To process the input and generate an echo we need to start the *application worker*.

In a new command line session start the echo *application worker* by executing the following command:

```
$ twistd -n --pidfile=applicationworker.pid vumi_worker --worker-class vumi.demos.words.EchoWorker --
```

This utilizes Twisted to start a Vumi `EchoWorker` process connected to the previously created Telnet *transport worker*.

Philosophy

An *application worker* is responsible for processing messages received from a *transport worker* and generating replies - it holds the application logic. For this example we are using an *echo* worker that will simply echo messages it receives back to the *transport worker*. In subsequent parts of this tutorial we'll be utilizing A.I. to generate *seemingly intelligent* replies.

Now if you enter something in your previously created Telnet session you should immediately receive an *echo*. The *application worker's* output should reflect the activity, for example when entering `hallo world`:

```
2012-03-06 12:10:39+0200 [WorkerAMQClient,client] User message: hallo world
```

That concludes part 1 of this tutorial. In part 2 we'll be creating a Google Talk chat bot.

2.2 Writing your first Vumi app - Part 2

This is the second part in a series of tutorials demonstrating how to develop Vumi apps.

If you haven't done so already you might want to work through part 1 of this tutorial before proceeding.

In this part of the tutorial we'll be creating a simple chat bot communicating over Google Talk.

More specifically we'll be utilizing Vumi's XMPP *transport worker* to log into a Google Talk account and listen for incoming chat messages. When messages are received an Alice Bot based *application worker* will determine an appropriate response based on the incoming message. The XMPP *transport worker* will then send the response. For another Google Talk user chatting with the Vumi connected account it should *appear* as if she is conversing with another human being.

Note: Remember your virtual environment should be active. Activate it by running running `source ve/bin/activate`.

2.2.1 XMPP Transport Worker

Continuing from part 1 of this tutorial, instead of using the Telnet *transport worker* we'll be using Vumi's built-in XMPP *transport worker* to communicate over Google Talk.

In order to use the XMPP *transport worker* you first need to create a configuration file.

To do this, create a `transport.yaml` file in your current directory and edit it to look like this (replacing `"username"` and `"password"` with your specific details):

```
transport_name: xmpp_transport
username: "username"
password: "password"
status: Playing with Vumi.
host: talk.google.com
port: 5222
```

Going through that line by line:

`transport_name: xmpp_transport` - specifies the transport name. This identifies the *transport worker* for subsequent connection by *application workers*.

`username: "username"` - the Google Talk account username to which the *transport worker* will connect.

`password: "password"` - the Google Talk account password.

`status: Playing with Vumi` - causes the Google Talk account's chat status to change to `Playing with Vumi.`

`host: talk.google.com` - The XMPP host to connect to. Google Talk uses `talk.google.com`.

`port: 5222` - The XMPP port to connect to. Google Talk uses `5222`.

Note: Vumi utilizes YAML based configuration files to provide configuration parameters to workers, both transport and application. YAML is a human friendly data serialization standard that works quite well for specifying configurations.

Now start the XMPP *transport worker* with the created configuration by executing the following command:

```
$ twistd -n --pidfile=transportworker.pid vumi_worker --worker-class vumi.transports.xmpp.XMPPTranspo
```

SASLNoAcceptableMechanism Exceptions

In the event of this command raising a `twisted.words.protocols.jabber.sasl.SASLNoAcceptableMechanism` exception you should upgrade your pyOpenSSL package by executing `pip install --upgrade pyOpenSSL` from the command line.

Note: This is different from the example in part 1 of this tutorial in that we no longer set any configuration options through the command line. Instead all configuration is contained in the specified `transport.yaml` config file.

This causes a Vumi XMPP *transport worker* to connect to the configuration specified Google Talk account and listen for messages. You should now be able to start messaging the account from another Google Talk account using any Google Talk client (although no response will be generated until the *application worker* is instantiated).

2.2.2 Alice Bot Application Worker

Continuing from part 1 of this tutorial, instead of using the *echo application worker* we'll be creating our own worker to generate *seemingly intelligent* responses.

Philosophy

Remember *application workers* are responsible for processing messages received from *transport workers* and generating replies - it holds the application logic. When developing Vumi applications you'll mostly be implementing

application workers to process messages based on your use case. For the most part you'll be relying on Vumi's built-in *transport workers* to take care of the communications medium. This enables you to forget about the hairy details of the communications medium and instead focus on the fun stuff.

Before we proceed let's install our dependencies. We'll be using PyAIML to provide our bot with *knowledge*. Install it by executing the following command:

```
$ pip install http://sourceforge.net/projects/pyaiml/files/PyAIML%20%28unstable%29/0.8.6/PyAIML-0.8.6
```

We also need a *brain* for our bot. Download a precompiled brain by executing the following command:

```
$ wget https://github.com/downloads/praekelt/public-eggs/alice.brn
```

Note: For the sake of simplicity we're using an existing brain. You can however compile your own brain by downloading the free Alice AIML set and *learning* it as described in the PyAIML examples. Perhaps you rather want a Fake Captain Kirk.

Now we can move on to creating the *application worker*. Create a `workers.py` file in your current directory and edit it to look like this:

```
import aiml
from vumi.application.base import ApplicationWorker

class AliceApplicationWorker(ApplicationWorker):

    def __init__(self, *args, **kwargs):
        self.bot = aiml.Kernel()
        self.bot.bootstrap(brainFile="alice.brn")
        return super(AliceApplicationWorker, self).__init__(*args, **kwargs)

    def consume_user_message(self, message):
        message_content = message['content']
        message_user = message.user()
        response = self.bot.respond(message_content, message_user)
        self.reply_to(message, response)
```

The code is straightforward. *Application workers* are represented by a class that subclasses *`vumi.application.base.ApplicationWorker`*. In this example the `__init__` method is overridden to initialize our bot's brain. The heart of *application workers* though is the `consume_user_message` method, which is passed messages for processing as they are received by *transport workers*. The message argument contains details on the received message. In this example the content of the message is retrieved from `message['content']`, and the Google Talk user sending the message is determined by calling `message.user()`. A response is then generated for the specific user utilizing the bot by calling `self.bot.respond(message_content, message_user)`. This response is then sent as a reply to the original message by calling `self.reply_to(message, response)`. The *transport worker* then takes care of sending the response to the correct user over the communications medium.

Philosophy

The *application worker* has very little knowledge about and does not need to know the specifics of the communications medium. In this example we could just as easily have communicated over SMS or even Twitter without having to change the *application worker's* implementation.

Now start the Alice Bot *application worker* in a new command line session by executing the following command:

```
$ twistd -n --pidfile=applicationworker.pid vumi_worker --worker-class workers.AliceApplicationWorke
```

Note: Again note how the *application worker* is connected to the previously defined, already running *transport worker* by specifying `--set-option=transport_name:xmpp_transport`.

Now with both the *transport worker* and *application worker* running you should be able to send a chat message to the Google Talk account configured in `transport.yaml` and receive a *seemingly intelligent* response generated by our Alice Bot.

2.2.3 Coming soon

The tutorial ends here for the time being. Future installments of the tutorial will cover:

- Advanced applications.
- Scaling and deploying.

In the meantime, you might want to check out some other docs.

2.3 ScaleConf Workshop - General Introduction

Note: These instructions were written for the first Vumi workshop on the 21st of April 2013, right after the ScaleConf conference in Cape Town.

Spotted an error? Please feel free to contribute to the documentation.

2.3.1 What is Vumi?

Vumi is a scalable, multi channel messaging platform. It has been designed to allow large scale mobile messaging in the majority world. It is actively being developed by the Praekelt Foundation and other contributors. It is available as Open Source software under the BSD license.

2.3.2 What were the design goals?

The Praekelt Foundation has a lot of experience building mobile messaging campaigns in the areas such as mobile health, education and democracy. Unfortunately, a lot of this experience comes from having built systems that caused problems in terms of scale and/or maintenance.

Key learnings from these mistakes led to a number of guiding principles in the design of Vumi, such as:

1. The campaign application logic should be decoupled from how it communicates with the end user.
2. The campaign application and the means of communication with the end-user should each be re-usable in a different context.
3. The system should be able to scale by adding more commodity machines, i.e. it should scale horizontally.

The above mentioned guiding principles resulted in a number of core concepts that make up a Vumi application.

A Vumi Message

A Vumi Message is the means of communication inside Vumi. Esentially a Vumi Message is just a bit of JSON that contains information on where a message was received from, who it was addressed to, what the message contents were and some extra metadata to allow it to be routed from and end-user to an application and back again.

Transports

Transports provide the communication channel to end users by integrating into various services such as chat systems, mobile network operators or possibly even traditional voice phone lines.

Transports are tasked with translating an inbound request into a standardized Vumi Message and vice-versa.

A simple example would be an SMS, which when received is converted into a bit of JSON that looks something like this:

```
{
    "message_id": "message1",
    "to_addr": "1234",
    "from_addr": "27761234567",
    "content": "This is an incoming SMS!",
    "transport_name": "smpp_transport",
    "transport_type": "sms",
    "transport_metadata": {
        // this is a dictionary containing
        // transport specific data
    }
}
```

Applications

Applications are tasked with either generating messages to be sent to or acting on the messages received from end users via the transports.

As a general rule the Applications should not care about which transport the message was received from, it merely acts on the message contents and provides a suitable reply.

A reply message looks something like this:

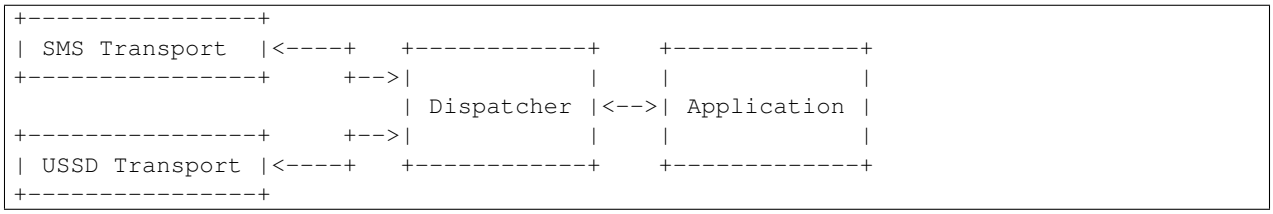
```
{
    "message_id": "message2",
    "in_reply_to": "message1",
    "to_addr": "27761234567",
    "from_addr": "1234",
    "content": "Thanks! We've received your SMS!",
    "transport_name": "smpp_transport",
    "transport_type": "sms",
    "helper_metadata": {
        // this is a dictionary containing
        // application specific data
    }
}
```

Dispatchers

Dispatchers are an optional means of connecting Transports and Applications. They allow for more complicated routing between the two.

A simple scenario is an application that receives from a USSD transport but requires the option of also replying via an SMS transport. A dispatcher would allow one to contruct this.

Dispatchers do this by inspecting the messages exchanged between the Transport and the Application and then deciding where it needs to go.



2.3.3 How does it work?

All of these different components are built using the Python programming language using Twisted, an event driven networking library.

The messages between the different components are exchanged and routed using RabbitMQ a high performance AMQP message broker.

For data storage Redis is used for data that are generally temporary but and may potentially be lost. Riak is used for things that need strong availability guarantees.

A sample use case of Redis would be to store session state whereas Riak would be used to store all messages sent and received indefinitely.

Supervisord is used to manage all the different processes and provide any easy commandline tool to start and stop them.

2.3.4 Let's get started!

As part of the workshop we will provide you with a South African USSD code and an SMS longcode. In the next section we'll help you get Vumi `running on your local machine` so you can start developing your first application!

CHAPTER 3

Forwarding SMSs from an SMPP bind to a URL

A simple use case for Vumi is to aggregate incoming SMSs and forward them via HTTP POST to a URL.

In this use case we are going to:

1. Use a SMSC simulator for local development.
2. Configure Vumi accept all incoming and outgoing messages on an SMPP bind.
3. Setup a worker that forwards all incoming messages to a URL via HTTP POST.
4. Setup Supervisord to manage all the different processes.

Note: Vumi relies for a large part on AMQP for its routing capabilities and some basic understanding is assumed. Have a look at http://blog.springsource.com/2010/06/14/understanding-amqp-the-protocol-used-by-rabbitmq/ for a more detailed explanation of AMQP.

3.1 Installing the SMSC simulator

Go to the *./utils* directory in the Vumi repository and run the bash script called *install_smpp_simulator.sh*. This will install the SMSC simulator from http://seleniumsoftware.com on your local machine. This simulator does exactly the same as a normal SMSC would do with the exception that it doesn't actually relay the messages to mobile networks.:

```
$ cd ./utils
$ ./install_smpp_simulator.sh
```

This will have installed the application in the *./utils/smppsim/SMPPSim* directory.

By default the SMPP simulator tries to open port 88 for it's HTTP console, since you often need administrative rights to open ports lower than 1024 let's change that to 8080 instead.

Line 60 of *./utils/smppsim/SMPPSim/conf/smppsim.props* says:

```
HTTP_PORT=88
```

Change this to:

```
HTTP_PORT=8080
```

Another change we need to make is on line 83:

```
ESME_TO_ESME=TRUE
```

Needs to be changed to, FALSE:

```
ESME_TO_ESME=FALSE
```

Having this set to True sometimes causes the SMSC and Vumi to bounce messages back and forth without stopping.

Note: The simulator is a Java application and we're assuming you have Java installed correctly.

3.2 Configuring Vumi

Vumi applications are made up of at least two components, the **Transport** which deals with in & outbound messages and the **Application** which acts on the messages received and potentially generates replies.

3.2.1 SMPP Transport

Vumi's SMPP Transport can be configured by a YAML file, *./config/example_smpp.yaml*. For this example, this is what our SMPP configuration looks like:

```
transport_name: smpp_transport
system_id: smppclient1  # username
password: password      # password
host: localhost         # the host to connect to
port: 2775              # the port to connect to
```

The SMPP Transport publishes inbound messages in Vumi's common message format and accepts the same format for outbound messages.

Here is a sample message:

```
{
    "message_id": "message1",
    "to_addr": "1234",
    "from_addr": "27761234567",
    "content": "This is an incoming SMS!",
    "transport_name": "smpp_transport",
    "transport_type": "sms",
    "transport_metadata": {
        // this is a dictionary containing
        // transport specific data
    }
}
```

3.2.2 HTTP Relay Application

Vumi ships with a simple application which forwards all messages it receives as JSON to a given URL with the option of using HTTP Basic Authentication when doing so. This application is also configured using the YAML file:

Setting up the webserver that responds to the HTTP request that the *HTTPRelayApplication* makes is left as an exercise for the reader. The *HTTPRelayApplication* has the ability to automatically respond to incoming messages based on the HTTP response received.

To do this:

1. The resource must return with a status of 200

2. The resource must set an HTTP Header *X-Vumi-HTTPRelay-Reply* and it must be set to *true* (case insensitive)
3. Any content that is returned in the body of the response is sent back as a message. If you want to limit this to 140 characters for use with SMS then that is the HTTP resource's responsibility.

3.3 Supervisord!

Let's use Supervisord to ensure all the different parts keep running. Here is the configuration file *supervisord.example.conf*:

```
[inet_http_server]         ; inet (TCP) server disabled by default
port=127.0.0.1:9010        ; (ip_address:port specifier, *:port for all iface)

[supervisord]
pidfile=./tmp/pids/supervisord.pid ; (supervisord pidfile;default supervisord.pid)

[rpcinterface:supervisor]
supervisor.rpcinterface_factory = supervisor.rpcinterface:make_main_rpcinterface

[supervisorctl]
serverurl=http://127.0.0.1:9010 ; use an http:// url to specify an inet socket

[program:transport]
command=twistd -n
    --pidfile=./tmp/pids/%(program_name)s.pid
    vumi_worker
    --worker-class=vumi.transports.smpp.SmppTransport
    --config=./config/example_smpp.yaml
stdout_logfile=./logs/%(program_name)s_%(process_num)s.log
stderr_logfile=./logs/%(program_name)s_%(process_num)s.err

[program:application]
command=twistd -n
    --pidfile=./tmp/pids/%(program_name)s.pid
    vumi_worker
    --worker-class=vumi.application.http_relay.HTTPRelayApplication
    --config=./config/example_http_relay.yaml
autorestart=true
stdout_logfile=./logs/%(program_name)s_%(process_num)s.log
stderr_logfile=./logs/%(program_name)s_%(process_num)s.err

[program:smsc]
command=java
    -Djava.net.preferIPv4Stack=true
    -Djava.util.logging.config.file=conf/logging.properties
    -jar smppsim.jar
    conf/smppsim.props
autorestart=true
directory=./utils/smppsim/SMPPSim/
stdout_logfile=./logs/%(program_name)s_%(process_num)s.log
stderr_logfile=./logs/%(program_name)s_%(process_num)s.err
```

Ensure you're in your python *virtualenv* and start it with the following command:

```
$ supervisord -c etc/supervisord.example.conf
```

You'll be able to see the HTTP management console at http://localhost:9010/ or at the command line with:

```
$ supervisorctl -c etc/supervisord.example.conf
```

3.4 Let's give it a try:

1. Go to http://localhost:8080 and send an SMS to Vumi via "Inject an MO message".
2. Type a message, it doesn't matter what *destination_addr* you chose, all incoming messages will be routed using the SMPP Transport's *transport_name* to the application subscribed to those messages. The HTTPRelayApplication will HTTP POST to the URL provided.

CHAPTER 4

Applications

Vumi applications implement application logic or call out to external services that implement such logic. Usually you will implement your own application workers but Vumi does provide a base application worker class and a few generic application workers.

4.1 Base class for applications

A base class you should extend when writing applications.

4.1.1 Application

class vumi.application.base.**ApplicationConfig**(*config_data*, *static=False*)
Base config definition for applications.

You should subclass this and add application-specific fields.

Configuration options:

Parameters

- **amqp_prefetch_count** (*int*) – The number of messages fetched concurrently from each AMQP queue by each worker instance.
- **transport_name** (*str*) – The name this application instance will use to create its queues.
- **send_to** (*dict*) – 'send_to' configuration dict.

class vumi.application.base.**ApplicationWorker**(*options*, *config=None*)
Base class for an application worker.

Handles `vumi.message.TransportUserMessage` and `vumi.message.TransportEvent` messages.

Application workers may send outgoing messages using `reply_to()` (for replies to incoming messages) or `send_to()` (for messages that are not replies).

`send_to()` can take either an *endpoint* parameter to specify the endpoint to send on (and optionally add additional message data from application configuration).

`ALLOWED_ENDPOINTS` lists the endpoints this application is allowed to send messages to using the `send_to()` method. If it is set to *None*, any endpoint is allowed.

Messages sent via `send_to()` pass optional additional data from configuration to the TransportUserMessage constructor, based on the endpoint parameter passed to send_to. This usually contains information useful for routing the message.

An example `send_to()` configuration might look like:

```
- send_to:
  - default:
    transport_name: sms_transport
```

NOTE: If you are using non-endpoint routing, 'transport_name' **must** be defined for each send_to section since dispatchers rely on this for routing outbound messages.

The available set of endpoints defaults to just the single endpoint named *default*. If applications wish to define their own set of available endpoints they should override `ALLOWED_ENDPOINTS`. Setting `ALLOWED_ENDPOINTS` to *None* allows the application to send on arbitrary endpoint names.

CONFIG_CLASS
alias of *ApplicationConfig*

static check_endpoint(*allowed_endpoints*, *endpoint*)
Check that endpoint is in the list of allowed endpoints.

Parameters

- **allowed_endpoints** (*list*) – List (or set) of allowed endpoints. If `allowed_endpoints` is `None`, all endpoints are allowed.
- **endpoint** (*str*) – Endpoint to check. The special value `None` is equivalent to `default`.

close_session(*message*)
Close a session.

The .reply_to() method should not be called when the session is closed.

consume_ack(*event*)
Handle an ack message.

consume_delivery_report(*event*)
Handle a delivery report.

consume_nack(*event*)
Handle a nack message

consume_user_message(*message*)
Respond to user message.

dispatch_event(*event*)
Dispatch to event_type specific handlers.

dispatch_user_message(*message*)
Dispatch user messages to handler.

new_session(*message*)
Respond to a new session.

Defaults to calling consume_user_message.

setup_application()
All application specific setup should happen in here.

Subclasses should override this method to perform extra setup.

setup_worker()
Set up basic application worker stuff.

You shouldn't have to override this in subclasses.

teardown_application()
Clean-up of setup done in setup_application should happen here.

4.2 HTTP Relay

Calls out to an external HTTP API that implements application logic and provides a similar API for application logic to call when sending messages.

4.2.1 HTTP Relay

class vumi.application.http_relay.**HTTPRelayConfig**(*config_data*, *static=False*)
Configuration options:

Parameters

- **amqp_prefetch_count** (*int*) – The number of messages fetched concurrently from each AMQP queue by each worker instance.
- **transport_name** (*str*) – The name this application instance will use to create its queues.
- **send_to** (*dict*) – 'send_to' configuration dict.
- **url** (*URL*) – URL to submit incoming message to.
- **event_url** (*URL*) – URL to submit incoming events to. (Defaults to the same as 'url').
- **http_method** (*str*) – HTTP method for submitting messages.
- **auth_method** (*str*) – HTTP authentication method.
- **username** (*str*) – Username for HTTP authentication.
- **password** (*str*) – Password for HTTP authentication.

class vumi.application.http_relay.**HTTPRelayApplication**(*options*, *config=None*)

4.3 RapidSMS Relay

Calls out to an application implemented in RapidSMS.

4.3.1 RapidSMS Relay

class vumi.application.rapidsms_relay.**RapidSMSRelayConfig**(*config_data*, *static=False*)
RapidSMS relay configuration.

Configuration options:

Parameters

- **amqp_prefetch_count** (*int*) – The number of messages fetched concurrently from each AMQP queue by each worker instance.

- **transport_name** (*str*) – The name this application instance will use to create its queues.
- **send_to** (*dict*) – 'send_to' configuration dict.
- **web_path** (*str*) – Path to listen for outbound messages from RapidSMS on.
- **web_port** (*int*) – Port to listen for outbound messages from RapidSMS on.
- **redis_manager** (*dict*) – Redis manager configuration (only required if *allow_replies* is true)
- **allow_replies** (*bool*) – Whether to support replies via the *in_reply_to* argument from RapidSMS.
- **vumi_username** (*str*) – Username required when calling *web_path* (default: no authentication)
- **vumi_password** (*str*) – Password required when calling *web_path*
- **vumi_auth_method** (*str*) – Authentication method required when calling *web_path*.The 'basic' method is currently the only available method
- **vumi_reply_timeout** (*int*) – Number of seconds to keep original messages in redis so that replies may be sent via *in_reply_to*.
- **allowed_endpoints** (*list*) – List of allowed endpoints to send from.
- **rapidsms_url** (*URL*) – URL of the rapidsms http backend.
- **rapidsms_username** (*str*) – Username to use for the *rapidsms_url* (default: no authentication)
- **rapidsms_password** (*str*) – Password to use for the *rapidsms_url*
- **rapidsms_auth_method** (*str*) – Authentication method to use with *rapidsms_url*. The 'basic' method is currently the only available method.
- **rapidsms_http_method** (*str*) – HTTP request method to use for the *rapidsms_url*

class vumi.application.rapidsms_relay.**RapidSMSRelay**(*options*, *config=None*)
Application that relays messages to RapidSMS.

4.4 Sandbox

Runs custom application logic in a sandbox.

4.4.1 Sandbox application workers

Sandbox

class vumi.application.sandbox.**SandboxConfig**(*config_data*, *static=False*)
Configuration options:

Parameters

- **amqp_prefetch_count** (*int*) – The number of messages fetched concurrently from each AMQP queue by each worker instance.
- **transport_name** (*str*) – The name this application instance will use to create its queues.

- **send_to** (*dict*) – 'send_to' configuration dict.
- **sandbox** (*dict*) – Dictionary of resources to provide to the sandbox. Keys are the names of resources (as seen inside the sandbox). Values are dictionaries which must contain a *cls* key that gives the full name of the class that provides the resource. Other keys are additional configuration for that resource.
- **executable** (*str*) – Full path to the executable to run in the sandbox.
- **args** (*list*) – List of arguments to pass to the executable (not including the path of the executable itself).
- **path** (*str*) – Current working directory to run the executable in.
- **env** (*dict*) – Custom environment variables for the sandboxed process.
- **timeout** (*int*) – Length of time the subprocess is given to process a message.
- **recv_limit** (*int*) – Maximum number of bytes that will be read from a sandboxed process' stdout and stderr combined.
- **rlimits** (*dict*) – Dictionary of resource limits to be applied to sandboxed processes. Defaults are fairly restricted. Keys maybe names or values of the RLIMIT constants in Python *resource* module. Values should be appropriate integers.
- **logging_resource** (*str*) – Name of the logging resource to use to report errors detected in sandboxed code (e.g. lines written to stderr, unexpected process termination). Set to null to disable and report these directly using Twisted logging instead.
- **sandbox_id** (*str*) – This is set based on individual messages.

class vumi.application.sandbox.**Sandbox**(*options*, *config=None*)
Sandbox application worker.

Javascript Sandbox

class vumi.application.sandbox.**JsSandboxConfig**(*config_data*, *static=False*)
JavaScript sandbox configuration.

Configuration options:

Parameters

- **amqp_prefetch_count** (*int*) – The number of messages fetched concurrently from each AMQP queue by each worker instance.
- **transport_name** (*str*) – The name this application instance will use to create its queues.
- **send_to** (*dict*) – 'send_to' configuration dict.
- **sandbox** (*dict*) – Dictionary of resources to provide to the sandbox. Keys are the names of resources (as seen inside the sandbox). Values are dictionaries which must contain a *cls* key that gives the full name of the class that provides the resource. Other keys are additional configuration for that resource.
- **executable** (*str*) – Full path to the executable to run in the sandbox.
- **args** (*list*) – List of arguments to pass to the executable (not including the path of the executable itself).
- **path** (*str*) – Current working directory to run the executable in.
- **env** (*dict*) – Custom environment variables for the sandboxed process.

- **timeout** (*int*) – Length of time the subprocess is given to process a message.
- **recv_limit** (*int*) – Maximum number of bytes that will be read from a sandboxed process' stdout and stderr combined.
- **rlimits** (*dict*) – Dictionary of resource limits to be applied to sandboxed processes. Defaults are fairly restricted. Keys maybe names or values of the RLIMIT constants in Python *resource* module. Values should be appropriate integers.
- **sandbox_id** (*str*) – This is set based on individual messages.
- **javascript** (*str*) – JavaScript code to run.
- **app_context** (*str*) – Custom context to execute JS with.
- **logging_resource** (*str*) – Name of the logging resource to use to report errors detected in sandboxed code (e.g. lines written to stderr, unexpected process termination). Set to null to disable and report these directly using Twisted logging instead.

class vumi.application.sandbox.**JsSandbox**(*options*, *config=None*)

Configuration options:

As for *Sandbox* except:

- *executable* defaults to searching for a *node.js* binary.
- *args* defaults to the JS sandbox script in the *vumi.application* module.
- An instance of *JsSandboxResource* is added to the sandbox resources under the name *js* if no *js* resource exists.
- An instance of *LoggingResource* is added to the sandbox resources under the name *log* if no *log* resource exists.
- *logging_resource* is set to *log* if it is not set.
- An extra 'javascript' parameter specifies the javascript to execute.
- An extra optional 'app_context' parameter specifying a custom context for the 'javascript' application to execute with.

Example 'javascript' that logs information via the sandbox API (provided as 'this' to 'on_inbound_message') and checks that logging was successful:

```
api.on_inbound_message = function(command) {
    this.log_info("From command: inbound-message", function (reply) {
        this.log_info("Log successful: " + reply.success);
        this.done();
    });
}
```

Example 'app_context' that makes the Node.js 'path' module available under the name 'path' in the context that the sandboxed javascript executes in:

```
{path: require('path')}
```

Javascript File Sandbox

class vumi.application.sandbox.**JsFileSandbox**(*options*, *config=None*)

class CONFIG_CLASS(*config_data*, *static=False*)

Configuration options:

Parameters

- **amqp_prefetch_count** (*int*) – The number of messages fetched concurrently from each AMQP queue by each worker instance.
- **transport_name** (*str*) – The name this application instance will use to create its queues.
- **send_to** (*dict*) – 'send_to' configuration dict.
- **sandbox** (*dict*) – Dictionary of resources to provide to the sandbox. Keys are the names of resources (as seen inside the sandbox). Values are dictionaries which must contain a *cls* key that gives the full name of the class that provides the resource. Other keys are additional configuration for that resource.
- **executable** (*str*) – Full path to the executable to run in the sandbox.
- **args** (*list*) – List of arguments to pass to the executable (not including the path of the executable itself).
- **path** (*str*) – Current working directory to run the executable in.
- **env** (*dict*) – Custom environment variables for the sandboxed process.
- **timeout** (*int*) – Length of time the subprocess is given to process a message.
- **recv_limit** (*int*) – Maximum number of bytes that will be read from a sandboxed process' stdout and stderr combined.
- **rlimits** (*dict*) – Dictionary of resource limits to be applied to sandboxed processes. Defaults are fairly restricted. Keys maybe names or values of the RLIMIT constants in Python *resource* module. Values should be appropriate integers.
- **logging_resource** (*str*) – Name of the logging resource to use to report errors detected in sandboxed code (e.g. lines written to stderr, unexpected process termination). Set to null to disable and report these directly using Twisted logging instead.
- **sandbox_id** (*str*) – This is set based on individual messages.
- **javascript_file** (*str*) – The file containting the Javascript to run
- **app_context** (*str*) – Custom context to execute JS with.

4.4.2 Sandbox resources

RedisResource

class vumi.application.sandbox.**RedisResource**(*name*, *app_worker*, *config*)

Resource that provides access to a simple key-value store.

Configuration options:

Parameters

- **redis_manager** (*dict*) – Redis manager configuration options.
- **keys_per_user_soft** (*int*) – Maximum number of keys each user may make use of in redis before usage warnings are logged. (default: 80% of hard limit).
- **keys_per_user_hard** (*int*) – Maximum number of keys each user may make use of in redis (default: 100). Falls back to keys_per_user.
- **keys_per_user** (*int*) – Synonym for *keys_per_user_hard*. Deprecated.

handle_delete(**args*, ***kwargs*)
Delete a key.

Command fields:

- `key`: The key to delete.

Reply fields:

- `success`: `true` if the operation was successful, otherwise `false`.

Example:

```
api.request(
    'kv.delete',
    {key: 'foo'},
    function(reply) {
        api.log_info('Value deleted: ' +
                     reply.success);
    }
);
```

handle_get(**args*, ***kwargs*)
Retrieve the value of a key.

Command fields:

- `key`: The key whose value should be retrieved.

Reply fields:

- `success`: `true` if the operation was successful, otherwise `false`.
- `value`: The value retrieved.

Example:

```
api.request(
    'kv.get',
    {key: 'foo'},
    function(reply) {
        api.log_info(
            'Value retrieved: ' +
            JSON.stringify(reply.value));
    }
);
```

handle_incr(**args*, ***kwargs*)
Atomically increment the value of an integer key.

The current value of the key must be an integer. If the key does not exist, it is set to zero.

Command fields:

- `key`: The key to delete.
- `amount`: The integer amount to increment the key by. Defaults to 1.

Reply fields:

- `success`: `true` if the operation was successful, otherwise `false`.
- `value`: The new value of the key.

Example:

```
        api.request(
            'kv.incr',
            {key: 'foo',
             amount: 3},
            function(reply) {
                api.log_info('New value: ' +
                             reply.value);
            }
        );
```

handle_set(**args*, ***kwargs*)
Set the value of a key.

Command fields:

- `key`: The key whose value should be set.
- `value`: The value to store. May be any JSON serializable object.
- `seconds`: Lifetime of the key in seconds. The default `null` indicates that the key should not expire.

Reply fields:

- `success`: `true` if the operation was successful, otherwise `false`.

Example:

```
        api.request(
            'kv.set',
            {key: 'foo',
             value: {x: '42'}},
            function(reply) { api.log_info('Value store: ' +
                                           reply.success); });
```

OutboundResource

class vumi.application.sandbox.**OutboundResource**(*name*, *app_worker*, *config*)
Resource that provides the ability to send outbound messages.

Includes support for replying to the sender of the current message, replying to the group the current message was from and sending messages that aren't replies.

JsSandboxResource

class vumi.application.sandbox.**JsSandboxResource**(*name*, *app_worker*, *config*)
Resource that initializes a Javascript sandbox.

Typically used alongside vumi/applicaiton/sandboxer.js which is a simple node.js based Javascript sandbox.

Requires the worker to have a *javascript_for_api* method.

LoggingResource

class vumi.application.sandbox.**LoggingResource**(*name*, *app_worker*, *config*)
Resource that allows a sandbox to log messages via Twisted's logging framework.

handle_critical(*api*, *command*)
Logs a message at the `CRITICAL` log level.

See *handle_log()* for details.

handle_debug(*api*, *command*)
Logs a message at the `DEBUG` log level.

See *handle_log()* for details.

handle_error(*api*, *command*)
Logs a message at the `ERROR` log level.

See *handle_log()* for details.

handle_info(*api*, *command*)
Logs a message at the `INFO` log level.

See *handle_log()* for details.

handle_log(**args*, ***kwargs*)
Log a message at the specified severity level.

The other log commands are identical except that `level` need not be specified. Using the log-level specific commands is preferred.

Command fields:

- `level`: The severity level to log at. Must be an integer log level. Default severity is the `INFO` log level.
- `msg`: The message to log.

Reply fields:

- `success`: `true` if the operation was successful, otherwise `false`.

Example:

```
api.request(
    'log.log',
    {level: 20,
     msg: 'Abandon ship!'},
    function(reply) {
        api.log_info('New value: ' +
                     reply.value);
    }
);
```

handle_warning(*api*, *command*)
Logs a message at the `WARNING` log level.

See *handle_log()* for details.

log(*api*, *msg*, *level*)
Logs a message via vumi.log (i.e. Twisted logging).

Sub-class should override this if they wish to log messages elsewhere. The *api* parameter is provided for use by such sub-classes.

The *log* method should always return a deferred.

HttpClientResource

class vumi.application.sandbox.**HttpClientResource**(*name*, *app_worker*, *config*)
Resource that allows making HTTP calls to outside services.

All command on this resource share a common set of command and response fields:

Command fields:

- `url`: The URL to request
- **`verify_options`: A list of options to verify when doing** an HTTPS request. Possible string values are `VERIFY_NONE`, `VERIFY_PEER`, `VERIFY_CLIENT_ONCE` and `VERIFY_FAIL_IF_NO_PEER_CERT`. Specifying multiple values results in passing along a reduced `OR` value (e.g. VERIFY_PEER | VERIFY_FAIL_IF_NO_PEER_CERT)
- **`headers`: A dictionary of keys for the header name and a list** of values to provide as header values.
- `data`: The payload to submit as part of the request.
- **`files`: A dictionary, submitted as multipart/form-data** in the request:

```
[{
    "field name": {
        "file_name": "the file name",
        "content_type": "content-type",
        "data": "data to submit, encoded as base64",
    }
}, ...]
```

The `data` field in the dictionary will be base64 decoded before the HTTP request is made.

Success reply fields:

- `success`: Set to `true`
- `body`: The response body
- `code`: The HTTP response code

Failure reply fields:

- `success`: set to `false`
- `reason`: Reason for the failure

Example:

```
api.request(
    'http.get',
    {url: 'http://foo/'},
    function(reply) { api.log_info(reply.body); });
```

`agent_class`
alias of `Agent`

`handle_delete`(*api*, *command*)
Make an HTTP DELETE request.

See `HttpResource` for details.

handle_get(*api*, *command*)
Make an HTTP GET request.

See `HttpResource` for details.

handle_head(*api*, *command*)
Make an HTTP HEAD request.

See `HttpResource` for details.

handle_patch(*api*, *command*)
Make an HTTP PATCH request.

See `HttpResource` for details.

handle_post(*api*, *command*)
Make an HTTP POST request.

See `HttpResource` for details.

handle_put(*api*, *command*)
Make an HTTP PUT request.

See `HttpResource` for details.

CHAPTER 5

Transports

Transports provide the means for Vumi to send and receive messages from people, usually via a third-party such as a mobile network operator or instant message service provider.

Vumi comes with support for numerous transports built-in. These include SMPP (SMS), SSMI (USSD), SMSSync (SMS over your Android phone), XMPP (Google Chat and Jabber), Twitter, IRC, telnet and numerous SMS and USSD transports for specific mobile network aggregators.

5.1 Transports for common protocols

5.1.1 Base class for transports

A base class you should extend when writing transports.

Transport

class vumi.transports.base.**TransportConfig**(*config_data*, *static=False*)
Base config definition for transports.

You should subclass this and add transport-specific fields.

Configuration options:

Parameters

- **amqp_prefetch_count** (*int*) – The number of messages fetched concurrently from each AMQP queue by each worker instance.
- **transport_name** (*str*) – The name this transport instance will use to create its queues.
- **publish_status** (*bool*) – Whether status messages should be published by the transport

class vumi.transports.base.**Transport**(*options*, *config=None*)
Base class for transport workers.

The following attributes are available for subclasses to control behaviour:

- start_message_consumer – Set to False if the message consumer should not be started. The subclass is responsible for starting it in this case.

CONFIG_CLASS
alias of *TransportConfig*

static generate_message_id()
Generate a message id.

handle_outbound_message(*message*)
This must be overridden to read outbound messages and do the right thing with them.

publish_ack(*user_message_id*, *sent_message_id*, ***kw*)
Helper method for publishing an `ack` event.

publish_delivery_report(*user_message_id*, *delivery_status*, ***kw*)
Helper method for publishing a `delivery_report` event.

publish_event(***kw*)
Publish a `TransportEvent` message.

Some default parameters are handled, so subclasses don't have to provide a lot of boilerplate.

publish_message(***kw*)
Publish a `TransportUserMessage` message.

Some default parameters are handled, so subclasses don't have to provide a lot of boilerplate.

publish_nack(*user_message_id*, *reason*, ***kw*)
Helper method for publishing a `nack` event.

publish_status(***kw*)
Helper method for publishing a status message.

send_failure(*message*, *exception*, *traceback*)
Send a failure report.

setup_transport()
All transport_specific setup should happen in here.

Subclasses should override this method to perform extra setup.

setup_worker()
Set up basic transport worker stuff.

You shouldn't have to override this in subclasses.

teardown_transport()
Clean-up of setup done in setup_transport should happen here.

5.1.2 SMPP

SMPP Transport

vumi.transports.smpp.**SmppTransport**
alias of `SmppTransceiverTransportWithOldConfig`

Example configuration

```
system_id: <provided by SMSC>
password: <provided by SMSC>
host: smpp.smppgateway.com
port: 2775
system_type: <provided by SMSC>

# Optional variables, some SMSCs require these to be set.
```

```
interface_version: "34"
dest_addr_ton: 1
dest_addr_npi: 1
registered_delivery: 1

TRANSPORT_NAME: smpp_transport

# Number Recognition
COUNTRY_CODE: "27"

OPERATOR_NUMBER:
    VODACOM: "<outbound MSISDN to be used for this MNO>"
    MTN: "<outbound MSISDN to be used for this MNO>"
    CELLC: "<outbound MSISDN to be used for this MNO>"
    VIRGIN: "<outbound MSISDN to be used for this MNO>"
    8TA: "<outbound MSISDN to be used for this MNO>"
    UNKNOWN: "<outbound MSISDN to be used for this MNO>"

OPERATOR_PREFIX:
    2771:
        27710: MTN
        27711: VODACOM
        27712: VODACOM
        27713: VODACOM
        27714: VODACOM
        27715: VODACOM
        27716: VODACOM
        27717: MTN
        27719: MTN

    2772: VODACOM
    2773: MTN
    2774:
        27740: CELLC
        27741: VIRGIN
        27742: CELLC
        27743: CELLC
        27744: CELLC
        27745: CELLC
        27746: CELLC
        27747: CELLC
        27748: CELLC
        27749: CELLC

    2776: VODACOM
    2778: MTN
    2779: VODACOM
    2781:
        27811: 8TA
        27812: 8TA
        27813: 8TA
        27814: 8TA

    2782: VODACOM
    2783: MTN
    2784: CELLC
```

Notes

- This transport does no MSISDN normalization
- This transport tries to guess the outbound MSISDN for any SMS sent using a operator prefix lookup.

Use of Redis in the SMPP Transport

Redis is used for all situations where temporary information must be cached where:

1. it will survive system shutdowns
2. it can be shared between workers

One use of Redis is for mapping between SMPP sequence_numbers and long term unique id's on the ESME and the SMSC.

The sequence_number parameter is a revolving set of integers used to pair outgoing async pdu's with their response, i.e. submit_sm & submit_sm_resp.

Both submit_sm and the corresponding submit_sm_resp will share a single sequence_number, however, for long term storage and future reference, it is necessary to link the id of the message stored on the SMSC (message_id in the submit_sm_resp) back to the id of the sent message. As the submit_sm_resp pdu's are received, the original id is looked up in Redis via the sequence_number and associated with the message_id in the response.

Followup pdu's from the SMSC (i.e. delivery reports) will reference the original message by the message_id held by the SMSC which was returned in the submit_sm_resp.

Status event catalogue

The SMPP transport publishes the following status events when status event publishing is enabled.

`starting`

Published when the transport is busy starting.

Fields:

- `status`: `down`
- `type`: `starting`
- `component`: `smpp`

`binding`

Published when the transport has established a connection to the SMSC, has attempted to bind, and is waiting for the SMSC's response.

Fields:

- `status`: `down`
- `type`: `binding`
- `component`: `smpp`

bound

Published when the transport has received a bind response from the SMSC and is ready to send and receive messages.

Fields:

- `status`: `ok`
- `type`: `bound`
- `component`: `smpp`

bind_timeout

Published when the transport has not bound within the interval given by the `smpp_bind_timeout` config field.

Fields:

- `status`: `down`
- `type`: `bind_timeout`
- `component`: `smpp`

unbinding

Published when the transport has attempted to unbind, and is waiting for the SMSC's response.

Fields:

- `status`: `down`
- `type`: `unbinding`
- `component`: `smpp`

connection_lost

Published when a transport loses its connection to the SMSC. This occurs in the following situations:

- after successfully unbinding
- if an unbind attempt times out
- when the connection to the SMSC is lost unexpectedly

Fields:

- `status`: `down`
- `type`: `connection_lost`
- `component`: `smpp`

throttled

Published when throttling starts for the transport and when throttling continues for a transport after rebinding. Throttling starts in two situations:

- the SMSC has replied to a message we attempted to send with an `ESME_RTHROTTLED` response

- we have reached the maximum number of transmissions per second allowed by the transport (set by the `mt_tps` config field), where a transmission is a mobile-terminating message put onto the wire by the transport.

Fields:

- `status`: `degraded`
- `type`: `throttled`
- `component`: `smpp`

throttled_end

Published when the transport is no longer throttled. This happens in two situations:

- we have retried an earlier message we attempted to send that was given a `ESME_RTHROTTLED` response, and the SMSC has responded to the retried message with a `ESME_ROK` response (that is, the retry was successful)
- the transport is no longer at the maximum number of transmissions per second

Fields:

- `status`: `ok`
- `type`: `throttled_end`
- `component`: `smpp`

5.1.3 SSMI

Truteq SSMI Transport

TruTeq USSD transport.

class vumi.transports.truteq.truteq.**TruteqTransport**(*options*, *config=None*)
Bases: *vumi.transports.base.Transport*

A transport for TruTeq.

Currently only USSD messages are supported.

CONFIG_CLASS
alias of *TruteqTransportConfig*

service_class
alias of ReconnectingClientService

class vumi.transports.truteq.truteq.**TruteqTransportConfig**(*config_data*, *static=False*)
Bases: *vumi.transports.base.TransportConfig*

Configuration options:

Parameters

- **amqp_prefetch_count** (*int*) – The number of messages fetched concurrently from each AMQP queue by each worker instance.
- **transport_name** (*str*) – The name this transport instance will use to create its queues.
- **publish_status** (*bool*) – Whether status messages should be published by the transport
- **username** (*str*) – Username of the TruTeq account to connect to.

- **password** (*str*) – Password for the TruTeq account.
- **twisted_endpoint** (*twisted_endpoint*) – The endpoint to connect to.
- **link_check_period** (*int*) – Number of seconds between link checks sent to the server.
- **ussd_session_lifetime** (*int*) – Maximum number of seconds to retain USSD session information.
- **debug** (*bool*) – Print verbose log output.
- **redis_manager** (*dict*) – How to connect to Redis.
- **host** (*str*) – *DEPRECATED* 'host' and 'port' fields may be used in place of the 'twisted_endpoint' field.
- **port** (*int*) – *DEPRECATED* 'host' and 'port' fields may be used in place of the 'twisted_endpoint' field.

5.1.4 HTTP RPC

Base class for constructing HTTP-based transports.

HTTP RPC base class

class vumi.transports.httprpc.httprpc.**HttpRpcTransport**(*options*, *config=None*)
Bases: *vumi.transports.base.Transport*

Base class for synchronous HTTP transports.

Because a reply from an application worker is needed before the HTTP response can be completed, a reply needs to be returned to the same transport worker that generated the inbound message. This means that currently there many only be one transport worker for each instance of this transport of a given name.

CONFIG_CLASS
alias of *HttpRpcTransportConfig*

add_status(***kw*)
Publishes a status if it is not a repeat of the previously published status.

get_clock()
For easier stubbing in tests

get_transport_url(*suffix=''*)
Get the URL for the HTTP resource. Requires the worker to be started.

This is mostly useful in tests, and probably shouldn't be used in non-test code, because the API might live behind a load balancer or proxy.

on_degraded_response_time(*message_id*, *time*)
Can be overridden by subclasses to do something when the response time is high enough for the transport to be considered running in a degraded state.

on_down_response_time(*message_id*, *time*)
Can be overridden by subclasses to do something when the response time is high enough for the transport to be considered non-functioning.

on_good_response_time(*message_id*, *time*)
Can be overridden by subclasses to do something when the response time is low enough for the transport to be considered running normally.

on_timeout(*message_id*, *time*)
Can be overridden by subclasses to do something when the response times out.

set_request_end(*message_id*)
Checks the saved timestamp to see the response time. If the starting timestamp for the message cannot be found, nothing is done. If the time is more than *response_time_down*, a *down* status event is sent. If the time more than *response_time_degraded*, a *degraded* status event is sent. If the time is less than *response_time_degraded*, an *ok* status event is sent.

class vumi.transports.httprpc.httprpc.**HttpRpcTransportConfig**(*config_data*, *static=False*)

Bases: *vumi.transports.base.TransportConfig*

Base config definition for transports.

You should subclass this and add transport-specific fields.

Configuration options:

Parameters

- **amqp_prefetch_count** (*int*) – The number of messages fetched concurrently from each AMQP queue by each worker instance.
- **transport_name** (*str*) – The name this transport instance will use to create its queues.
- **publish_status** (*bool*) – Whether status messages should be published by the transport
- **web_path** (*str*) – The path to listen for requests on.
- **web_port** (*int*) – The port to listen for requests on, defaults to *0*.
- **web_username** (*str*) – The username to require callers to authenticate with. If `None` then no authentication is required. Currently only HTTP Basic authentication is supported.
- **web_password** (*str*) – The password to go with `web_username`. Must be `None` if and only if `web_username` is `None`.
- **web_auth_domain** (*str*) – The name of authentication domain.
- **health_path** (*str*) – The path to listen for downstream health checks on (useful with HAProxy)
- **request_cleanup_interval** (*int*) – How often should we actively look for old connections that should manually be timed out. Anything less than *1* disables the request cleanup meaning that all request objects will be kept in memory until the server is restarted, regardless if the remote side has dropped the connection or not. Defaults to 5 seconds.
- **request_timeout** (*int*) – How long should we wait for the remote side generating the response for this synchronous operation to come back. Any connection that has waited longer than *request_timeout* seconds will manually be closed. Defaults to 4 minutes.
- **request_timeout_status_code** (*int*) – What HTTP status code should be generated when a timeout occurs. Defaults to *504 Gateway Timeout*.
- **request_timeout_body** (*str*) – What HTTP body should be returned when a timeout occurs. Defaults to ''.
- **noisy** (*bool*) – Defaults to *False* set to *True* to make this transport log verbosely.
- **validation_mode** (*str*) – The mode to operate in. Can be 'strict' or 'permissive'. If 'strict' then any parameter received that is not listed in EXPECTED_FIELDS nor in IGNORED_FIELDS will raise an error. If 'permissive' then no error is raised as long as all the EXPECTED_FIELDS are present.

- **response_time_down** (*float*) – The maximum time allowed for a response before the service is considered *down*
- **response_time_degraded** (*float*) – The maximum time allowed for a response before the service is considered *degraded*

5.1.5 Mxit

Mxit Transport

class vumi.transports.mxit.mxit.**MxitTransport**(*options*, *config=None*)
Bases: *vumi.transports.httprpc.httprpc.HttpRpcTransport*

HTTP Transport for MXit, implemented using the MXit Mobi Portal (for inbound messages and replies) and the Messaging API (for sends that aren't replies).

•Mobi Portal API specification: http://dev.mxit.com/docs/mobi-portal-api

•Message API specification: https://dev.mxit.com/docs/restapi/messaging/post-message-send

CONFIG_CLASS
alias of *MxitTransportConfig*

html_decode(*html*)
Turns 'foo' into u'foo'

class vumi.transports.mxit.mxit.**MxitTransportConfig**(*config_data*, *static=False*)
Bases: *vumi.transports.httprpc.httprpc.HttpRpcTransportConfig*

Configuration options:

Parameters

- **amqp_prefetch_count** (*int*) – The number of messages fetched concurrently from each AMQP queue by each worker instance.
- **transport_name** (*str*) – The name this transport instance will use to create its queues.
- **publish_status** (*bool*) – Whether status messages should be published by the transport
- **web_path** (*str*) – The path to listen for requests on.
- **web_port** (*int*) – The port to listen for requests on, defaults to *0*.
- **web_username** (*str*) – The username to require callers to authenticate with. If `None` then no authentication is required. Currently only HTTP Basic authentication is supported.
- **web_password** (*str*) – The password to go with `web_username`. Must be `None` if and only if `web_username` is `None`.
- **web_auth_domain** (*str*) – The name of authentication domain.
- **health_path** (*str*) – The path to listen for downstream health checks on (useful with HAProxy)
- **request_cleanup_interval** (*int*) – How often should we actively look for old connections that should manually be timed out. Anything less than *1* disables the request cleanup meaning that all request objects will be kept in memory until the server is restarted, regardless if the remote side has dropped the connection or not. Defaults to 5 seconds.

- **request_timeout** (*int*) – How long should we wait for the remote side generating the response for this synchronous operation to come back. Any connection that has waited longer than *request_timeout* seconds will manually be closed. Defaults to 4 minutes.
- **request_timeout_status_code** (*int*) – What HTTP status code should be generated when a timeout occurs. Defaults to *504 Gateway Timeout*.
- **request_timeout_body** (*str*) – What HTTP body should be returned when a timeout occurs. Defaults to ''.
- **noisy** (*bool*) – Defaults to *False* set to *True* to make this transport log verbosely.
- **validation_mode** (*str*) – The mode to operate in. Can be 'strict' or 'permissive'. If 'strict' then any parameter received that is not listed in EXPECTED_FIELDS nor in IGNORED_FIELDS will raise an error. If 'permissive' then no error is raised as long as all the EXPECTED_FIELDS are present.
- **response_time_down** (*float*) – The maximum time allowed for a response before the service is considered *down*
- **response_time_degraded** (*float*) – The maximum time allowed for a response before the service is considered *degraded*
- **client_id** (*str*) – The OAuth2 ClientID assigned to this transport.
- **client_secret** (*str*) – The OAuth2 ClientSecret assigned to this transport.
- **timeout** (*int*) – Timeout for outbound Mxit HTTP API calls.
- **redis_manager** (*dict*) – How to connect to Redis
- **api_send_url** (*str*) – The URL for the Mxit message sending API.
- **api_auth_url** (*str*) – The URL for the Mxit authentication API.
- **api_auth_scopes** (*list*) – The list of scopes to request access to.

exception vumi.transports.mxit.mxit.**MxitTransportException**
Bases: exceptions.Exception

Raised when the Mxit API returns an error

5.1.6 ParlayX

ParlayX SMS Transport

class vumi.transports.parlayx.parlayx.**ParlayXTransport**(*options*, *config=None*)
Bases: *vumi.transports.base.Transport*

ParlayX SMS transport.

ParlayX is a defunkt standard web service API for telephone networks. See http://en.wikipedia.org/wiki/Parlay_X for an overview.

> **Warning:** This transport has not been tested against another ParlayX implementation. If you use it, please provide feedback to the Vumi development team on your experiences.

CONFIG_CLASS
alias of *ParlayXTransportConfig*

handle_outbound_message(*message*)
Send a text message via the ParlayX client.

handle_outbound_message_failure(**args*, ***kwargs*)
Handle outbound message failures.

ServiceException, *PolicyException* and client-class SOAP faults result in *PermanentFailure* being raised; server-class SOAP faults instances result in *TemporaryFailure* being raised; and other failures are passed through.

handle_raw_inbound_message(*correlator*, *linkid*, *inbound_message*)
Handle incoming text messages from *SmsNotificationService* callbacks.

class vumi.transports.parlayx.parlayx.**ParlayXTransportConfig**(*config_data*, *static=False*)
Bases: *vumi.transports.base.TransportConfig*

Configuration options:

Parameters

- **amqp_prefetch_count** (*int*) – The number of messages fetched concurrently from each AMQP queue by each worker instance.
- **transport_name** (*str*) – The name this transport instance will use to create its queues.
- **publish_status** (*bool*) – Whether status messages should be published by the transport
- **web_notification_path** (*str*) – Path to listen for delivery and receipt notifications on
- **web_notification_port** (*int*) – Port to listen for delivery and receipt notifications on
- **notification_endpoint_uri** (*str*) – URI of the ParlayX SmsNotificationService in Vumi
- **short_code** (*str*) – Service activation number or short code to receive deliveries for
- **remote_send_uri** (*str*) – URI of the remote ParlayX SendSmsService
- **remote_notification_uri** (*str*) – URI of the remote ParlayX SmsNotificationService
- **start_notifications** (*bool*) – Start (and stop) the ParlayX notification service?
- **service_provider_service_id** (*str*) – Provisioned service provider service identifier
- **service_provider_id** (*str*) – Provisioned service provider identifier/username
- **service_provider_password** (*str*) – Provisioned service provider password

vumi.transports.parlayx.parlayx.**extract_message_id**(*correlator*)
Extract the Vumi message identifier from a ParlayX correlator.

vumi.transports.parlayx.parlayx.**unique_correlator**(*message_id*, *_uuid=None*)
Construct a unique message identifier from an existing message identifier.

This is necessary for the cases where a `TransportMessage` needs to be transmitted, since ParlayX wants unique identifiers for all sent messages.

5.1.7 SMSSync

SMSSync Transport

class vumi.transports.smssync.smssync.**BaseSmsSyncTransport**(*options*, *config=None*)
Bases: *vumi.transports.httprpc.httprpc.HttpRpcTransport*

Ushahidi SMSSync Transport for getting messages into vumi.

Parameters

- **web_path** (*str*) – The path relative to the host where this listens
- **web_port** (*int*) – The port this listens on
- **transport_name** (*str*) – The name this transport instance will use to create its queues
- **redis_manager** (*dict*) – Redis client configuration.
- **reply_delay** (*float*) – The amount of time to wait (in seconds) for a reply message before closing the SMSSync HTTP inbound message request. Replies received within this amount of time will be returned with the reply (default: 0.5s).

add_msginfo_metadata(*payload*, *msginfo*)
Update an outbound message's payload's transport_metadata to allow msginfo to be reconstructed from replies.

callLater(*_seconds*, *_f*, **args*, ***kw*)
See twisted.internet.interfaces.IReactorTime.callLater.

msginfo_for_message(*msg*)
Returns an *SmsSyncMsgInfo* instance for this outbound message.

May return a deferred that yields the actual result to its callback.

msginfo_for_request(*request*)
Returns an *SmsSyncMsgInfo* instance for this request.

May return a deferred that yields the actual result to its callback.

class vumi.transports.smssync.smssync.**MultiSmsSync**(*options*, *config=None*)
Bases: *vumi.transports.smssync.smssync.BaseSmsSyncTransport*

Ushahidi SMSSync Transport for a multiple phones.

Each phone accesses a URL that has the form <*web_path*>/<*account_id*>/. A blank secret should be entered into the SMSSync *secret* field.

Additional configuration options:

Parameters **country_codes** (*dict*) – Map from *account_id* to the country code to use when normalizing MSISDNs sent by SMSSync to that API URL. If an *account_id* is not in this map the default is to use an empty country code string).

class vumi.transports.smssync.smssync.**SingleSmsSync**(*options*, *config=None*)
Bases: *vumi.transports.smssync.smssync.BaseSmsSyncTransport*

Ushahidi SMSSync Transport for a single phone.

Additional configuration options:

Parameters

- **smssync_secret** (*str*) – Secret of the single phone (default: '', i.e. no secret set)

- **account_id** (*str*) – Account id for storing outbound messages under. Defaults to the *smssync_secret* which is fine unless the secret changes.
- **country_code** (*str*) – Default country code to use when normalizing MSISDNs sent by SMSSync (default is the empty string, which assumes numbers already include the international dialing prefix).

class vumi.transports.smssync.smssync.**SmsSyncMsgInfo**(*account_id*, *smssync_secret*, *country_code*)

Bases: object

Holder of attributes needed to process an SMSSync message.

Parameters

- **account_id** (*str*) – An ID for the acocunt this message is being sent to / from.
- **smssync_secret** (*str*) – The shared SMSSync secret for the account this message is being sent to / from.
- **country_code** (*str*) – The default country_code for the account this message is being sent to / from.

5.1.8 Telnet

Telnet Transport

Transport that sends and receives to telnet clients.

class vumi.transports.telnet.telnet.**TelnetServerConfig**(*config_data*, *static=False*)

Bases: *vumi.transports.base.TransportConfig*

Telnet transport configuration.

Configuration options:

Parameters

- **amqp_prefetch_count** (*int*) – The number of messages fetched concurrently from each AMQP queue by each worker instance.
- **transport_name** (*str*) – The name this transport instance will use to create its queues.
- **publish_status** (*bool*) – Whether status messages should be published by the transport
- **twisted_endpoint** (*twisted_endpoint*) – The endpoint the Telnet server will listen on.
- **to_addr** (*str*) – The to_addr to use for inbound messages. The default is to use the host:port of the telnet server.
- **transport_type** (*str*) – The transport_type to use for inbound messages.
- **telnet_host** (*str*) – *DEPRECATED* 'telnet_host' and 'telnet_port' fields may be used inplace of the 'twisted_endpoint' field.
- **telnet_port** (*int*) – *DEPRECATED* 'telnet_host' and 'telnet_port' fields may be used in place of the 'twisted_endpoint' field.

class vumi.transports.telnet.telnet.**TelnetServerTransport**(*options*, *config=None*)

Bases: *vumi.transports.base.Transport*

Telnet based transport.

This transport listens on a specified port for telnet clients and routes lines to and from connected clients.

CONFIG_CLASS
alias of *TelnetServerConfig*

protocol
alias of *TelnetTransportProtocol*

class vumi.transports.telnet.telnet.**TelnetTransportProtocol**(*vumi_transport*)
Bases: twisted.conch.telnet.TelnetProtocol

Extends Twisted's TelnetProtocol for the Telnet transport.

5.1.9 Twitter

Twitter Transport

class vumi.transports.twitter.twitter.**TwitterTransport**(*options*, *config=None*)
Bases: *vumi.transports.base.Transport*

Twitter transport.

CONFIG_CLASS
alias of *TwitterTransportConfig*

class vumi.transports.twitter.twitter.**TwitterTransportConfig**(*config_data*, *static=False*)
Bases: *vumi.transports.base.TransportConfig*

Configuration options:

Parameters

- **amqp_prefetch_count** (*int*) – The number of messages fetched concurrently from each AMQP queue by each worker instance.
- **transport_name** (*str*) – The name this transport instance will use to create its queues.
- **publish_status** (*bool*) – Whether status messages should be published by the transport
- **screen_name** (*str*) – The screen name for the twitter account
- **consumer_key** (*str*) – The OAuth consumer key for the twitter account
- **consumer_secret** (*str*) – The OAuth consumer secret for the twitter account
- **access_token** (*str*) – The OAuth access token for the twitter account
- **access_token_secret** (*str*) – The OAuth access token secret for the twitter account
- **endpoints** (*twitter_endpoints*) – Which endpoints to use for dms and tweets
- **terms** (*list*) – A list of terms to be tracked by the transport
- **autofollow** (*bool*) – Determines whether the transport will follow users that follow the transport's user

5.1.10 Vumi Go bridge

Vumi Bridge Transport

class vumi.transports.vumi_bridge.vumi_bridge.**VumiBridgeTransportConfig**(*config_data*, *static=False*)

Bases: *vumi.transports.base.TransportConfig*

Configuration options:

Parameters

- **amqp_prefetch_count** (*int*) – The number of messages fetched concurrently from each AMQP queue by each worker instance.
- **transport_name** (*str*) – The name this transport instance will use to create its queues.
- **publish_status** (*bool*) – Whether status messages should be published by the transport
- **account_key** (*str*) – The account key to connect with.
- **conversation_key** (*str*) – The conversation key to use.
- **access_token** (*str*) – The access token for the conversation key.
- **base_url** (*str*) – The base URL for the API
- **message_life_time** (*int*) – How long to keep message_ids around for.
- **redis_manager** (*dict*) – Redis client configuration.
- **max_reconnect_delay** (*int*) – Maximum number of seconds between connection attempts
- **max_retries** (*int*) – Maximum number of consecutive unsuccessful connection attempts after which no further connection attempts will be made. If this is not explicitly set, no maximum is applied
- **initial_delay** (*float*) – Initial delay for first reconnection attempt
- **factor** (*float*) – A multiplicitive factor by which the delay grows
- **jitter** (*float*) – Percentage of randomness to introduce into the delay lengthto prevent stampeding.
- **web_port** (*int*) – The port to listen for requests on, defaults to *0*.
- **web_path** (*str*) – The path to listen for inbound requests on.
- **health_path** (*str*) – The path to listen for downstream health checks on (useful with HAProxy)

5.1.11 XMPP

XMPP Transport

class vumi.transports.xmpp.xmpp.**TransportPresenceClientProtocol**(*initialized_callback*, **args*, ***kwargs*)

Bases: wokkel.xmppim.PresenceClientProtocol

A custom presence protocol to automatically accept any subscription attempt.

class vumi.transports.xmpp.xmpp.**XMPPTransport**(*options*, *config=None*)
Bases: *vumi.transports.base.Transport*

XMPP transport.

Configuration parameters:

Parameters

- **host** (*str*) – The host of the XMPP server to connect to.
- **port** (*int*) – The port on the XMPP host to connect to.
- **debug** (*bool*) – Whether or not to show all the XMPP traffic. Defaults to False.
- **username** (*str*) – The XMPP account username
- **password** (*str*) – The XMPP account password
- **status** (*str*) – The XMPP status 'away', 'xa', 'chat' or 'dnd'
- **status_message** (*str*) – The natural language status message for this XMPP transport.
- **presence_interval** (*int*) – How often (in seconds) to send a presence update to the roster.
- **ping_interval** (*int*) – How often (in seconds) to send a keep-alive ping to the XMPP server to keep the connection alive. Defaults to 60 seconds.

5.1.12 IRC

IRC Transport

IRC transport.

class vumi.transports.irc.irc.**IrcConfig**(*config_data*, *static=False*)
Bases: *vumi.transports.base.TransportConfig*

IRC transport config.

Configuration options:

Parameters

- **amqp_prefetch_count** (*int*) – The number of messages fetched concurrently from each AMQP queue by each worker instance.
- **transport_name** (*str*) – The name this transport instance will use to create its queues.
- **publish_status** (*bool*) – Whether status messages should be published by the transport
- **twisted_endpoint** (*twisted_endpoint*) – Endpoint to connect to the IRC server on.
- **nickname** (*str*) – IRC nickname for the transport IRC client to use.
- **channels** (*list*) – List of channels to join.
- **network** (*str*) – *DEPRECATED* 'network' and 'port' fields may be used in place of the 'twisted_endpoint' field.
- **port** (*int*) – *DEPRECATED* 'network' and 'port' fields may be used in place of the 'twisted_endpoint' field.

class vumi.transports.irc.irc.**IrcMessage**(*sender*, *command*, *recipient*, *content*, *nickname=None*)

Bases: object

Container for details of a message to or from an IRC user.

Parameters

- **sender** (*str*) – Who sent the message (usually user!ident@hostmask).
- **recipient** (*str*) – User or channel recieving the message.
- **content** (*str*) – Contents of message.
- **nickname** (*str*) – Nickname used by the client that received the message. Optional.
- **command** (*str*) – IRC command that produced the message.

static canonicalize_recipient(*recipient*)

Convert a generic IRC address (with possible server parts) to a simple lowercase username or channel.

channel()

Return the channel if the recipient is a channel.

Otherwise return None.

class vumi.transports.irc.irc.**IrcTransport**(*options*, *config=None*)

Bases: *vumi.transports.base.Transport*

IRC based transport.

CONFIG_CLASS

alias of *IrcConfig*

class vumi.transports.irc.irc.**VumiBotFactory**(*vumibot_args*)

Bases: twisted.internet.protocol.ClientFactory

A factory for VumiBotClient instances.

A new protocol instance will be created each time we connect to the server.

protocol

alias of *VumiBotProtocol*

class vumi.transports.irc.irc.**VumiBotProtocol**(*nickname*, *channels*, *irc_transport*)

Bases: twisted.words.protocols.irc.IRCClient

An IRC bot that bridges IRC to Vumi.

action(*sender*, *recipient*, *message*)

This will get called when the bot sees someone do an action.

alterCollidedNick(*nickname*)

Generate an altered version of a nickname that caused a collision in an effort to create an unused related name for subsequent registration.

irc_NICK(*prefix*, *params*)

Called when an IRC user changes their nickname.

joined(*channel*)

This will get called when the bot joins the channel.

noticed(*sender*, *recipient*, *message*)

This will get called when the bot receives a notice.

privmsg(*sender*, *recipient*, *message*)

This will get called when the bot receives a message.

signedOn()
Called when bot has succesfully signed on to server.

5.1.13 Dev Null

Dev Null Transport

class vumi.transports.devnull.devnull.**DevNullTransport**(*options*, *config=None*)
Bases: *vumi.transports.base.Transport*

DevNullTransport for messages that need fake delivery to networks. Useful for testing.

Configuration parameters:

Parameters

- **transport_type** (*str*) – The transport type to emulate, defaults to sms.
- **ack_rate** (*float*) – How many messages should be ack'd. The remainder will be nacked. The *failure_rate* and *reply_rate* treat the *ack_rate* as 100%.
- **failure_rate** (*float*) – How many messages should be treated as failures. Float value between 0.0 and 1.0.
- **reply_rate** (*float*) – For how many messages should we generate a reply? Float value between 0.0 and 1.0.
- **reply_copy** (*str*) – What copy should be sent as the reply, defaults to echo-ing the content of the outbound message.

5.1.14 Vumi HTTP API Transport

HTTP API Transport

class vumi.transports.api.api.**HttpApiConfig**(*config_data*, *static=False*)
Bases: *vumi.transports.httprpc.httprpc.HttpRpcTransportConfig*

HTTP API configuration.

Configuration options:

Parameters

- **amqp_prefetch_count** (*int*) – The number of messages fetched concurrently from each AMQP queue by each worker instance.
- **transport_name** (*str*) – The name this transport instance will use to create its queues.
- **publish_status** (*bool*) – Whether status messages should be published by the transport
- **web_path** (*str*) – The path to listen for requests on.
- **web_port** (*int*) – The port to listen for requests on, defaults to *0*.
- **web_username** (*str*) – The username to require callers to authenticate with. If `None` then no authentication is required. Currently only HTTP Basic authentication is supported.
- **web_password** (*str*) – The password to go with `web_username`. Must be `None` if and only if `web_username` is `None`.

- **web_auth_domain** (*str*) – The name of authentication domain.
- **health_path** (*str*) – The path to listen for downstream health checks on (useful with HAProxy)
- **request_cleanup_interval** (*int*) – How often should we actively look for old connections that should manually be timed out. Anything less than *1* disables the request cleanup meaning that all request objects will be kept in memory until the server is restarted, regardless if the remote side has dropped the connection or not. Defaults to 5 seconds.
- **request_timeout** (*int*) – How long should we wait for the remote side generating the response for this synchronous operation to come back. Any connection that has waited longer than *request_timeout* seconds will manually be closed. Defaults to 4 minutes.
- **request_timeout_status_code** (*int*) – What HTTP status code should be generated when a timeout occurs. Defaults to *504 Gateway Timeout*.
- **request_timeout_body** (*str*) – What HTTP body should be returned when a timeout occurs. Defaults to ''.
- **noisy** (*bool*) – Defaults to *False* set to *True* to make this transport log verbosely.
- **validation_mode** (*str*) – The mode to operate in. Can be 'strict' or 'permissive'. If 'strict' then any parameter received that is not listed in EXPECTED_FIELDS nor in IGNORED_FIELDS will raise an error. If 'permissive' then no error is raised as long as all the EXPECTED_FIELDS are present.
- **response_time_down** (*float*) – The maximum time allowed for a response before the service is considered *down*
- **response_time_degraded** (*float*) – The maximum time allowed for a response before the service is considered *degraded*
- **reply_expected** (*bool*) – True if a reply message is expected.
- **allowed_fields** (*list*) – The list of fields a request is allowed to contain. Defaults to the DEFAULT_ALLOWED_FIELDS class attribute.
- **field_defaults** (*dict*) – Default values for fields not sent by the client.

class vumi.transports.api.api.**HttpApiTransport**(*options*, *config=None*)

Bases: *vumi.transports.httprpc.httprpc.HttpRpcTransport*

Native HTTP API for getting messages into vumi.

NOTE: This has no security. Put it behind a firewall or something.

If reply_expected is True, the transport will wait for a reply message and will return the reply's content as the HTTP response body. If False, the message_id of the dispatched incoming message will be returned.

CONFIG_CLASS

alias of *HttpApiConfig*

5.1.15 Old Vumi HTTP Transport

A deprecated simple API for submitting Vumi messages into Vumi.

Old HTTP Transports

class vumi.transports.api.oldapi.**OldSimpleHttpTransport**(*options*, *config=None*)

Maintains the API used by the old Django based method of loading SMS's into VUMI over HTTP

Configuration options:

web_path [str] The path relative to the host where this listens

web_port [int] The port this listens on

transport_name [str] The name this transport instance will use to create it's queues

identities [dictionary] user : str password : str default_transport : str

class vumi.transports.api.oldapi.**OldTemplateHttpTransport**(*options*, *config=None*)

Notes

Default allowed keys:

- content
- to_addr
- from_addr

Others can be allowed by specifying the *allowed_fields* in the configuration file.

There is no limit on the length of the content so if you are publishing to a length constrained transport such as SMS then you are responsible for limiting the length appropriately.

If you expect a reply from the *Application* that is dealing with these requests then set the *reply_expected* boolean to *true* in the config file. That will keep the HTTP connection open until a response is returned. The *content* of the reply message is used as the HTTP response body.

Example configuration

```
transport_name: http_transport
web_path: /a/path/
web_port: 8123
reply_expected: false
allowed_fields:
    - content
    - to_addr
    - from_addr
    - provider
field_defaults:
    transport_type: http
```

5.2 Transports for specific aggregators

5.2.1 Airtel

Airtel USSD Transport

class vumi.transports.airtel.airtel.**AirtelUSSDTransport**(*options*, *config=None*)

Bases: *vumi.transports.httprpc.httprpc.HttpRpcTransport*

Client implementation for the Comviva Flares HTTP Pull API. Based on Flares 1.5.0, document version 1.2.0

CONFIG_CLASS
alias of *AirtelUSSDTransportConfig*

class vumi.transports.airtel.airtel.**AirtelUSSDTransportConfig**(*config_data*, *static=False*)

Bases: *vumi.transports.httprpc.httprpc.HttpRpcTransportConfig*

Configuration options:

Parameters

- **amqp_prefetch_count** (*int*) – The number of messages fetched concurrently from each AMQP queue by each worker instance.
- **transport_name** (*str*) – The name this transport instance will use to create its queues.
- **publish_status** (*bool*) – Whether status messages should be published by the transport
- **web_path** (*str*) – The path to listen for requests on.
- **web_port** (*int*) – The port to listen for requests on, defaults to *0*.
- **web_username** (*str*) – The username to require callers to authenticate with. If `None` then no authentication is required. Currently only HTTP Basic authentication is supported.
- **web_password** (*str*) – The password to go with `web_username`. Must be `None` if and only if `web_username` is `None`.
- **web_auth_domain** (*str*) – The name of authentication domain.
- **health_path** (*str*) – The path to listen for downstream health checks on (useful with HAProxy)
- **request_cleanup_interval** (*int*) – How often should we actively look for old connections that should manually be timed out. Anything less than *1* disables the request cleanup meaning that all request objects will be kept in memory until the server is restarted, regardless if the remote side has dropped the connection or not. Defaults to 5 seconds.
- **request_timeout** (*int*) – How long should we wait for the remote side generating the response for this synchronous operation to come back. Any connection that has waited longer than *request_timeout* seconds will manually be closed. Defaults to 4 minutes.
- **request_timeout_status_code** (*int*) – What HTTP status code should be generated when a timeout occurs. Defaults to *504 Gateway Timeout*.
- **request_timeout_body** (*str*) – What HTTP body should be returned when a timeout occurs. Defaults to ''.
- **noisy** (*bool*) – Defaults to *False* set to *True* to make this transport log verbosely.
- **validation_mode** (*str*) – The mode to operate in. Can be 'strict' or 'permissive'. If 'strict' then any parameter received that is not listed in EXPECTED_FIELDS nor in IGNORED_FIELDS will raise an error. If 'permissive' then no error is raised as long as all the EXPECTED_FIELDS are present.
- **response_time_down** (*float*) – The maximum time allowed for a response before the service is considered *down*
- **response_time_degraded** (*float*) – The maximum time allowed for a response before the service is considered *degraded*
- **airtel_username** (*str*) – The username for this transport
- **airtel_password** (*str*) – The password for this transport

- **airtel_charge** (*bool*) – Whether or not to charge for the responses sent.
- **airtel_charge_amount** (*int*) – How much to charge
- **redis_manager** (*dict*) – Parameters to connect to Redis with.
- **session_key_prefix** (*str*) – The prefix to use for session key management. Specify thisif you are using more than 1 worker in a load-balancedfashion.
- **ussd_session_timeout** (*int*) – Max length of a USSD session
- **to_addr_pattern** (*str*) – A regular expression that to_addr values in messages that start a new USSD session must match. Initial messages with invalid to_addr values are rejected.

5.2.2 Apposit

Apposit Transport

class vumi.transports.apposit.apposit.**AppositTransport**(*options*, *config=None*)

Bases: *vumi.transports.httprpc.httprpc.HttpRpcTransport*

HTTP transport for Apposit's interconnection services.

CONFIG_CLASS

alias of *AppositTransportConfig*

class vumi.transports.apposit.apposit.**AppositTransportConfig**(*config_data*, *static=False*)

Bases: *vumi.transports.httprpc.httprpc.HttpRpcTransportConfig*

Apposit transport config.

Configuration options:

Parameters

- **amqp_prefetch_count** (*int*) – The number of messages fetched concurrently from each AMQP queue by each worker instance.
- **transport_name** (*str*) – The name this transport instance will use to create its queues.
- **publish_status** (*bool*) – Whether status messages should be published by the transport
- **web_path** (*str*) – The path to listen for requests on.
- **web_port** (*int*) – The port to listen for requests on, defaults to *0*.
- **web_username** (*str*) – The username to require callers to authenticate with. If `None` then no authentication is required. Currently only HTTP Basic authentication is supported.
- **web_password** (*str*) – The password to go with `web_username`. Must be `None` if and only if `web_username` is `None`.
- **web_auth_domain** (*str*) – The name of authentication domain.
- **health_path** (*str*) – The path to listen for downstream health checks on (useful with HAProxy)
- **request_cleanup_interval** (*int*) – How often should we actively look for old connections that should manually be timed out. Anything less than *1* disables the request cleanup meaning that all request objects will be kept in memory until the server is restarted, regardless if the remote side has dropped the connection or not. Defaults to 5 seconds.

- **request_timeout** (*int*) – How long should we wait for the remote side generating the response for this synchronous operation to come back. Any connection that has waited longer than *request_timeout* seconds will manually be closed. Defaults to 4 minutes.
- **request_timeout_status_code** (*int*) – What HTTP status code should be generated when a timeout occurs. Defaults to *504 Gateway Timeout*.
- **request_timeout_body** (*str*) – What HTTP body should be returned when a timeout occurs. Defaults to ''.
- **noisy** (*bool*) – Defaults to *False* set to *True* to make this transport log verbosely.
- **validation_mode** (*str*) – The mode to operate in. Can be 'strict' or 'permissive'. If 'strict' then any parameter received that is not listed in EXPECTED_FIELDS nor in IGNORED_FIELDS will raise an error. If 'permissive' then no error is raised as long as all the EXPECTED_FIELDS are present.
- **response_time_down** (*float*) – The maximum time allowed for a response before the service is considered *down*
- **response_time_degraded** (*float*) – The maximum time allowed for a response before the service is considered *degraded*
- **credentials** (*dict*) – A dictionary where the *from_addr* is used for the key lookup and the returned value should be a dictionary containing the corresponding username, password and service id.
- **outbound_url** (*str*) – The URL to send outbound messages to.

5.2.3 Cellulant

Cellulant Transport

exception vumi.transports.cellulant.cellulant.**CellulantError**
Bases: vumi.errors.VumiError

Used to log errors specific to the Cellulant transport.

class vumi.transports.cellulant.cellulant.**CellulantTransport**(*options*, *config=None*)
Bases: *vumi.transports.httprpc.httprpc.HttpRpcTransport*

Cellulant USSD (via HTTP) transport.

Cellulant USSD Transport

class vumi.transports.cellulant.cellulant_sms.**CellulantSmsTransport**(*options*, *config=None*)
Bases: *vumi.transports.httprpc.httprpc.HttpRpcTransport*

HTTP transport for Cellulant SMS.

CONFIG_CLASS
alias of *CellulantSmsTransportConfig*

class vumi.transports.cellulant.cellulant_sms.**CellulantSmsTransportConfig**(*config_data*, *static=False*)
Bases: *vumi.transports.httprpc.httprpc.HttpRpcTransportConfig*

Cellulant transport config.

Configuration options:

Parameters

- **amqp_prefetch_count** (*int*) – The number of messages fetched concurrently from each AMQP queue by each worker instance.
- **transport_name** (*str*) – The name this transport instance will use to create its queues.
- **publish_status** (*bool*) – Whether status messages should be published by the transport
- **web_path** (*str*) – The path to listen for requests on.
- **web_port** (*int*) – The port to listen for requests on, defaults to *0*.
- **web_username** (*str*) – The username to require callers to authenticate with. If `None` then no authentication is required. Currently only HTTP Basic authentication is supported.
- **web_password** (*str*) – The password to go with `web_username`. Must be `None` if and only if `web_username` is `None`.
- **web_auth_domain** (*str*) – The name of authentication domain.
- **health_path** (*str*) – The path to listen for downstream health checks on (useful with HAProxy)
- **request_cleanup_interval** (*int*) – How often should we actively look for old connections that should manually be timed out. Anything less than *1* disables the request cleanup meaning that all request objects will be kept in memory until the server is restarted, regardless if the remote side has dropped the connection or not. Defaults to 5 seconds.
- **request_timeout** (*int*) – How long should we wait for the remote side generating the response for this synchronous operation to come back. Any connection that has waited longer than *request_timeout* seconds will manually be closed. Defaults to 4 minutes.
- **request_timeout_status_code** (*int*) – What HTTP status code should be generated when a timeout occurs. Defaults to *504 Gateway Timeout*.
- **request_timeout_body** (*str*) – What HTTP body should be returned when a timeout occurs. Defaults to ''.
- **noisy** (*bool*) – Defaults to *False* set to *True* to make this transport log verbosely.
- **validation_mode** (*str*) – The mode to operate in. Can be 'strict' or 'permissive'. If 'strict' then any parameter received that is not listed in EXPECTED_FIELDS nor in IGNORED_FIELDS will raise an error. If 'permissive' then no error is raised as long as all the EXPECTED_FIELDS are present.
- **response_time_down** (*float*) – The maximum time allowed for a response before the service is considered *down*
- **response_time_degraded** (*float*) – The maximum time allowed for a response before the service is considered *degraded*
- **credentials** (*dict*) – A dictionary where the *from_addr* is used for the key lookup and the returned value should be a dictionary containing the username and password.
- **outbound_url** (*str*) – The URL to send outbound messages to.

5.2.4 IMImobile Transport

IMIMobile Transport

class vumi.transports.imimobile.imimobile_ussd.**ImiMobileUssdTransport**(*options*, *config=None*)

Bases: *vumi.transports.httprpc.httprpc.HttpRpcTransport*

HTTP transport for USSD with IMImobile in India.

Configuration parameters:

Parameters

- **transport_name** (*str*) – The name this transport instance will use to create its queues
- **web_path** (*str*) – The HTTP path to listen on.
- **web_port** (*int*) – The HTTP port to listen on.
- **suffix_to_addrs** (*dict*) – Mappings between url suffixes and to addresses.
- **user_terminated_session_message** (*str*) – A regex used to identify user terminated session messages. Default is '^Map Dialog User Abort User Reason'.
- **user_terminated_session_response** (*str*) – Response given back to the user if the user terminated the session. Default is 'Session Ended'.
- **redis_manager** (*dict*) – The configuration parameters for connecting to Redis.
- **ussd_session_timeout** (*int*) – Number of seconds before USSD session information stored in Redis expires. Default is 600s.

get_to_addr(*request*)

Extracts the request url path's suffix and uses it to obtain the tag associated with the suffix. Returns a tuple consisting of the tag and a dict of errors encountered.

classmethod ist_to_utc(*timestamp*)

Accepts a timestamp in the format *[M]M/[D]D/YYYY HH:MM:SS (am|pm)* and in India Standard Time, and returns a datetime object normalized to UTC time.

5.2.5 Infobip

Infobip Transport

Infobip USSD transport.

exception vumi.transports.infobip.infobip.**InfobipError**

Bases: vumi.errors.VumiError

Used to log errors specific to the Infobip transport.

class vumi.transports.infobip.infobip.**InfobipTransport**(*options*, *config=None*)

Bases: *vumi.transports.httprpc.httprpc.HttpRpcTransport*

Infobip transport.

Currently only supports the Infobip USSD interface.

Configuration parameters:

Parameters **ussd_session_timeout** (*int*) – Number of seconds before USSD session information stored in Redis expires. Default is 600s.

Excerpt from *INFOBIP USSD Gateway to Third-party Application HTTP/REST/JSON Web Service API*:

Third party application provides four methods for session management. Their parameters are as follows:

•sessionActive (type Boolean) – true if the session is active, false otherwise. The parameter is mandatory.

•sessionId (type String) – is generated for each started session and. The parameter is mandatory. exitCode (type Integer) – defined the status of the session that is complete. All the exit codes can be found in Table 1. The parameter is mandatory.

•reason (type String) – in case a third-party applications releases the session before its completion it will contain the reason for the release. The parameter is used for logging purposes and is mandatory. msisdn (type String) – of the user that sent the response to the menu request. The parameter is mandatory.

•imsi (type String) – of the user that sent the response to the menu request. The parameter is optional.

•text (type String) – text the user entered in the response. The parameter is mandatory. shortCode – Short code entered in the mobile initiated session or by calling start method. The parameter is optional.

•language (type String)– defines which language will be used for message text. Used in applications that support internationalization. The parameter should be set to null if not used. The parameter is optional.

•optional (type String)– left for future usage scenarios. The parameter is optional. ussdGwId (type String)– id of the USSD GW calling the third-party application. This parameter is optional.

Responses to requests sent from the third-party-applications have the following parameters:

•ussdMenu (type String)– menu to send as text to the user. The parameter is mandatory.

•shouldClose (type boolean)– set to true if this is the last message in this session sent to the user, false if there will be more. The parameter is mandatory. Please note that the first message in the session will always be sent as a menu (i.e. shouldClose will be set to false).

•thirdPartyId (type String)– Id of the third-party application or server. This parameter is optional.

•responseExitCode (type Integer) – request processing exit code. Parameter is mandatory.

5.2.6 Integrat

Integrat Transport

class vumi.transports.integrat.integrat.**IntegratTransport**(*options*, *config=None*)
Bases: *vumi.transports.base.Transport*

Integrat USSD transport over HTTP.

setup_transport(**args*, ***kwargs*)
All transport_specific setup should happen in here.

validate_config()
Transport-specific config validation happens in here.

5.2.7 MediaEdgeGSM

MediaEdgeGSM Transport

class vumi.transports.mediaedgegsm.mediaedgegsm.**MediaEdgeGSMTransport**(*options*, *config=None*)

Bases: *vumi.transports.httprpc.httprpc.HttpRpcTransport*

HTTP transport for MediaEdgeGSM in Ghana.

Parameters

- **web_path** (*str*) – The HTTP path to listen on.
- **web_port** (*int*) – The HTTP port
- **transport_name** (*str*) – The name this transport instance will use to create its queues
- **username** (*str*) – MediaEdgeGSM account username.
- **password** (*str*) – MediaEdgeGSM account password.
- **outbound_url** (*str*) – The URL to hit for outbound messages that aren't replies.
- **outbound_username** (*str*) – The username for outbound non-reply messages.
- **outbound_password** (*str*) – The username for outbound non-reply messages.
- **operator_mappings** (*dict*) – A nested dictionary mapping MSISDN prefixes to operator names

5.2.8 Mediafone Cameroun

Mediafone Cameroun Transport

class vumi.transports.mediafonemc.mediafonemc.**MediafoneTransport**(*options*, *config=None*)

Bases: *vumi.transports.httprpc.httprpc.HttpRpcTransport*

HTTP transport for Mediafone Cameroun.

Parameters

- **web_path** (*str*) – The HTTP path to listen on.
- **web_port** (*int*) – The HTTP port
- **transport_name** (*str*) – The name this transport instance will use to create its queues
- **username** (*str*) – Mediafone account username.
- **password** (*str*) – Mediafone account password.
- **outbound_url** (*str*) – The URL to send outbound messages to.

5.2.9 MTECH USSD

MTECH USSD Transport

class vumi.transports.mtech_ussd.mtech_ussd.**MtechUssdTransport**(*options*, *config=None*)

Bases: *vumi.transports.httprpc.httprpc.HttpRpcTransport*

MTECH USSD transport.

Configuration parameters:

Parameters

- **transport_name** (*str*) – The name this transport instance will use to create its queues
- **ussd_session_timeout** (*int*) – Number of seconds before USSD session information stored in Redis expires. Default is 600s.
- **web_path** (*str*) – The HTTP path to listen on.
- **web_port** (*int*) – The HTTP port to listen on.

NOTE: We currently only support free-text USSD, not menus. At the time of writing, vumi has no suitable message format for specifying USSD menus. This may change in the future.

5.2.10 Safaricom

Safaricom Transport

class vumi.transports.safaricom.safaricom.**SafaricomTransport**(*options*, *config=None*)

Bases: *vumi.transports.httprpc.httprpc.HttpRpcTransport*

HTTP transport for USSD with Safaricom in Kenya.

Parameters

- **web_path** (*str*) – The HTTP path to listen on.
- **web_port** (*int*) – The HTTP port
- **transport_name** (*str*) – The name this transport instance will use to create its queues
- **redis** (*dict*) – The configuration parameters for connecting to Redis.
- **ussd_session_timeout** (*int*) – The number of seconds after which a timeout is forced on a transport level.

5.2.11 Opera

Opera Transport

exception vumi.transports.opera.opera.**BadRequestError**

Bases: exceptions.Exception

An exception we can throw while parsing a request to return a 400 response.

class vumi.transports.opera.opera.**OperaTransport**(*options*, *config=None*)
Bases: *vumi.transports.base.Transport*

Opera transport.

See https://dragon.sa.operatelecom.com:1089/ for documentation on the Opera XML-RPC interface.

Configuration options:

Parameters

- **message_id_lifetime** (*int*) – Seconds message ids should be kept for before expiring. Once an id expires, delivery reports can no longer be associated with the original message id. Default is one week.
- **web_receipt_path** (*str*) – Path part of JSON reply URL (should match value given to Opera). E.g. /api/v1/sms/opera/receipt.json
- **web_receive_path** (*str*) – Path part of XML reply URL (should match value given to Opera). E.g. /api/v1/sms/opera/receive.xml
- **web_port** (*int*) – Port the transport listens to for responses from Opera. Affects both web_receipt_path and web_receive_path.
- **url** (*str*) – Opera XML-RPC gateway. E.g. https://dragon.sa.operatelecom.com:1089/Gateway
- **channel** (*str*) – Opera channel number.
- **password** (*str*) – Opera password.
- **service** (*str*) – Opera service number.
- **max_segments** (*int*) – Maximum number of segments to allow messages to be broken into. Default is 9. Minimum is 1. Maximum is 9. Note: Opera's own default is 1. This transport defaults to 9 to minimise the possibility of message sends failing.

get_message_id_for_identifier(*identifier*)
Get an internal message id for a given identifier

Parameters **identifier** (*str*) – The message id we originally got from Opera when the message was accepted for delivery.

get_transport_url(*suffix=''*)
Get the URL for the HTTP resource. Requires the worker to be started.

This is mostly useful in tests, and probably shouldn't be used in non-test code, because the API might live behind a load balancer or proxy.

handle_outbound_message_failure(**args*, ***kwargs*)
Decide what to do on certain failure cases.

set_message_id_for_identifier(*identifier*, *message_id*)
Link an external message id, the identifier, to an internal message id for *MAX_ID_LIFETIME* amount of seconds

Parameters

- **identifier** (*str*) – The message id we get back from Opera
- **message_id** (*str*) – The internal message id that was used when the message was sent.

validate_config()
Transport-specific config validation happens in here.

5.2.12 Vas2Nets

A WASP providing connectivity in Nigeria via an HTTP API.

Vas2nets Transport

class vumi.transports.vas2nets.**Vas2NetsTransport**(*options*, *config=None*)

Notes

Valid single byte characters:

```
string.ascii_lowercase,      # a-z
string.ascii_uppercase,      # A-Z
'0123456789',
'äöüÄÖÜàùòìèé§Ññ£$@',
' ',
'/?!#%&()*+,-:;<=>.',
'\n\r'
```

Valid double byte characters, will limit SMS to max length of 70 instead of 160 if used:

```
'|{}[]€\~^'
```

If any characters are published that aren't in this list the transport will raise a *Vas2NetsEncodingError*. If characters are published that are in the double byte set the transport will print warnings in the log.

Example configuration

```
transport_name: vas2nets
web_receive_path: /api/v1/sms/vas2nets/receive/
web_receipt_path: /api/v1/sms/vas2nets/receipt/
web_port: 8123

url: <provided by vas2nets>
username: <provided by vas2nets>
password: <provided by vas2nets>
owner: <provided by vas2nets>
service: <provided by vas2nets>
subservice: <provided by vas2nets>
```

5.2.13 Vodacom Messaging

Vodacom Messaging Transport

class vumi.transports.vodacom_messaging.vodacom_messaging.**VodacomMessagingTransport**(*options*, *config=None*)

Bases: *vumi.transports.httprpc.httprpc.HttpRpcTransport*

Vodacom Messaging USSD over HTTP transport.

5.2.14 MTN Nigeria

MTN Nigeria USSD Transport

class vumi.transports.mtn_nigeria.mtn_nigeria_ussd.**MtnNigeriaUssdTransport**(*options*, *config=None*)

Bases: *vumi.transports.base.Transport*

USSD transport for MTN Nigeria.

This transport connects as a TCP client and sends messages using a custom protocol whose packets consist of binary headers plus an XML body.

CONFIG_CLASS
alias of *MtnNigeriaUssdTransportConfig*

class vumi.transports.mtn_nigeria.mtn_nigeria_ussd.**MtnNigeriaUssdTransportConfig**(*config_data*, *static=False*)

Bases: *vumi.transports.base.TransportConfig*

MTN Nigeria USSD transport configuration.

Configuration options:

Parameters

- **amqp_prefetch_count** (*int*) – The number of messages fetched concurrently from each AMQP queue by each worker instance.
- **transport_name** (*str*) – The name this transport instance will use to create its queues.
- **publish_status** (*bool*) – Whether status messages should be published by the transport
- **server_hostname** (*str*) – Hostname of the server the transport's client should connect to.
- **server_port** (*int*) – Port that the server is listening on.
- **username** (*str*) – The username for this transport.
- **password** (*str*) – The password for this transport.
- **application_id** (*str*) – An application ID required by MTN Nigeria for client authentication.
- **enquire_link_interval** (*int*) – The interval (in seconds) between enquire links sent to the server to check whether the connection is alive and well.
- **timeout_period** (*int*) – How long (in seconds) after sending an enquire link request the client should wait for a response before timing out. NOTE: The timeout period should not be longer than the enquire link interval
- **user_termination_response** (*str*) – Response given back to the user if the user terminated the session.
- **redis_manager** (*dict*) – Parameters to connect to Redis with
- **session_timeout_period** (*int*) – Max length (in seconds) of a USSD session

CHAPTER 6

Dispatchers

Dispatchers are vumi workers that route messages between sets of transports and sets of application workers.

Vumi transports and application workers both have a single *endpoint* on which messages are sent and received (the name of the endpoint is given by the *transport_name* configuration option). Connecting sets of transports and applications requires a kind of worker with *multiple* endpoints. This class of workers is the dispatcher.

Examples of use cases for dispatchers:

- A single application that sends and receives both SMSes and XMPP messages.
- A single application that sends and receives SMSes in multiple countries using a different transport in each.
- A single SMPP transport that sends and receives SMSes on behalf of multiple applications.
- Multiple applications that all send and receive SMSes in multiple countries using a shared set of SMPP transports.

Vumi provides a pluggable dispatch worker `BaseDispatchWorker` that may be extended by much simpler *routing classes* that implement only the logic for routing messages (see *Routers*). The pluggable dispatcher handles setting up endpoints for all the transports and application workers the dispatcher communicates with. A simple

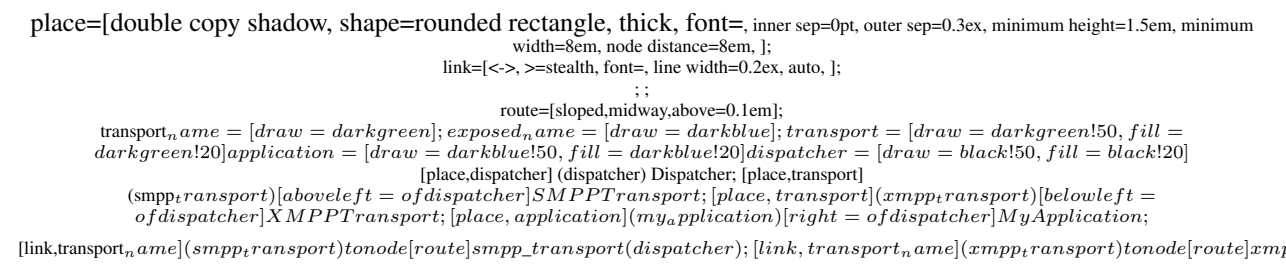


Fig. 6.1: A simple dispatcher configuration. Boxes represent workers. Edges are routing links between workers. Edges are labelled with endpoint names (i.e. transport_names).

`BaseDispatchWorker` YAML configuration file for the example above might look like:

```
# dispatcher config

router_class: vumi.dispatchers.SimpleDispatchRouter

transport_names:
  - smpp_transport
  - xmpp_transport

exposed_names:
```

```
  - my_application

# router config

route_mappings:
  smpp_transport: my_application
  xmpp_transport: my_application
```

The *router_class*, *transport_names* and *exposed_names* sections are all configuration for the `BaseDispatchWorker` itself and will be present in all dispatcher configurations:

- *router_class* gives the full Python path to the class implementing the routing logic.
- *transport_names* is the list of transport endpoints the dispatcher should receive and publish messages on.
- *exposed_names* is the list of application endpoints the dispatcher should receive and publish messages on.

The last section, *routing_mappings*, is specific to the router class used (i.e. *vumi.dispatchers.SimpleDispatchRouter*). It lists the application endpoint that messages and events from each transport should be sent to. In this simple example message from both transports are sent to the same application worker.

Other router classes will have different router configuration options. These are described in Builtin routers.

6.1 Routers

Router classes implement dispatching of inbound and outbound messages and events. Inbound messages and events come from transports and are typically dispatched to an application. Outbound messages come from applications and are typically dispatched to a transport.

Many routers follow a simple pattern:

- *inbound* messages are routed using custom routing logic.
- *events* are routed towards the same application the associated message was routed to.
- *outbound* messages that are replies are routed towards the transport that the original message came from.
- *outbound* messages that are not replies are routed based on additional information provided by the application (in simple setups its common for the application to simply provide the name of the transport the message should be routed to).

You can read more about the routers Vumi provides and about how to write your own router class in the following sections:

6.1.1 Builtin routers

Vumi ships with a small set of generically useful routers:

Vumi routers

- *SimpleDispatchRouter*
- *TransportToTransportRouter*
- *ToAddrRouter*
- *FromAddrMultiplexRouter*
- *UserGroupingRouter*
- *ContentKeywordRouter*

SimpleDispatchRouter

class vumi.dispatchers.**SimpleDispatchRouter**(*dispatcher*, *config*)
Simple dispatch router that maps transports to apps.

Configuration options:

Parameters

- **route_mappings** (*dict*) – A map of *transport_names* to *exposed_names*. Inbound messages and events received from a given transport are dispatched to the application attached to the corresponding exposed name.
- **transport_mappings** (*dict*) – An optional re-mapping of *transport_names* to *transport_names*. By default, outbound messages are dispatched to the transport attached to the *endpoint* with the same name as the transport name given in the message. If a transport name is present in this dictionary, the message is instead dispatched to the new transport name given by the re-mapping.

TransportToTransportRouter

class vumi.dispatchers.**TransportToTransportRouter**(*dispatcher*, *config*)
Simple dispatch router that connects transports to other transports.

Note: Connecting transports to one results in event messages being discarded since transports cannot receive events. Outbound messages never need to be dispatched because transports only send inbound messages.

Configuration options:

Parameters **route_mappings** (*dict*) – A map of *transport_names* to *transport_names*. Inbound messages received from a transport are sent as outbound messages to the associated transport.

ToAddrRouter

class vumi.dispatchers.**ToAddrRouter**(*dispatcher*, *config*)
Router that dispatches based on msg to_addr.

Parameters **toaddr_mappings** (*dict*) – Mapping from application transport names to regular expressions. If a message's to_addr matches the given regular expression the message is sent to the applications listening on the given transport name.

FromAddrMultiplexRouter

class vumi.dispatchers.**FromAddrMultiplexRouter**(*dispatcher*, *config*)
Router that multiplexes multiple transports based on msg from_addr.

This router is intended to be used to multiplex a pool of transports that each only supports a single external address, and present them to applications (or downstream dispatchers) as a single transport that supports multiple external addresses. This is useful for multiplexing vumi.transports.xmpp.XMPPTransport instances, for example.

Note: This router rewrites *transport_name* in both directions. Also, only one exposed name is supported.

Configuration options:

Parameters **fromaddr_mappings** (*dict*) – Mapping from message *from_addr* to *transport_name*.

UserGroupingRouter

class vumi.dispatchers.**UserGroupingRouter**(*dispatcher*, *config*)
Router that dispatches based on msg *from_addr*. Each unique *from_addr* is round-robin assigned to one of the defined groups in *group_mappings*. All messages from that *from_addr* are then routed to the *app* assigned to that group.

Useful for A/B testing.

Configuration options:

Parameters

- **group_mappings** (*dict*) – Mapping of group names to transport_names. If a user is assigned to a given group the message is sent to the application listening on the given transport_name.
- **dispatcher_name** (*str*) – The name of the dispatcher, used internally as the prefix for Redis keys.

ContentKeywordRouter

class vumi.dispatchers.**ContentKeywordRouter**(*dispatcher*, *config*)
Router that dispatches based on the first word of the message content. In the context of SMSes the first word is sometimes called the 'keyword'.

Parameters

- **keyword_mappings** (*dict*) – Mapping from application transport names to simple keywords. This is purely a convenience for constructing simple routing rules. The rules generated from this option are appened to the of rules supplied via the *rules* option.
- **rules** (*list*) – A list of routing rules. A routing rule is a dictionary. It must have *app* and *keyword* keys and may contain *to_addr* and *prefix* keys. If a message's first word matches a given keyword, the message is sent to the application listening on the transport name given by the value of *app*. If a 'to_addr' key is supplied, the message *to_addr* must also match the value of the 'to_addr' key. If a 'prefix' is supplied, the message *from_addr* must *start with* the value of the 'prefix' key.

- **fallback_application** (*str*) – Optional application transport name to forward inbound messages that match no rule to. If omitted, unrouted inbound messages are just logged.
- **transport_mappings** (*dict*) – Mapping from message *from_addr* values to transports names. If a message's from_addr matches a given from_addr, the message is sent to the associated transport.
- **expire_routing_memory** (*int*) – Time in seconds before outbound message's ids are expired from the redis routing store. Outbound message ids are stored along with the transport_name the message came in on and are used to route events such as acknowledgements and delivery reports back to the application that sent the outgoing message. Default is seven days.

6.1.2 Implementing your own router

A routing class publishes message on behalf of a dispatch worker. To do so it must provide three dispatch functions – one for inbound user messages, one for outbound user messages and one for events (e.g. delivery reports and acknowledgements). Failure messages are not routed via dispatchers and are typically sent directly to a failure worker. The receiving of messages and events is handled by the dispatcher itself.

A dispatcher provides three dictionaires of publishers as attributes:

- *exposed_publisher* – publishers for sending inbound user messages to applications attached to the dispatcher.
- *exposed_event_publisher* – publishers for sending events to applications.
- ***transport_publisher* – publishers for sending outbound user** messages to transports attached to the dispatcher.

Each of these dictionaries is keyed by *endpoint* name. The keys for *exposed_publisher* and *exposed_event_publisher* are the endpoints listed in the *exposed_names* configuration option passed to the dispatcher. The keys for *transport_publisher* are the endpoints listed in the *transport_names* configuration option. Routing classes publish messages by calling the `publish_message()` method on one of the publishers in these three dictionaries.

Routers are required to have the same interface as the `BaseDipatcherRouter` class which is described below.

class `vumi.dispatchers.`**BaseDispatchRouter**(*dispatcher*, *config*)

Base class for dispatch routing logic.

This is a convenient definition of and set of common functionality for router classes. You need not subclass this and should not instantiate this directly.

The `__init__()` method should take exactly the following options so that your class can be instantiated from configuration in a standard way:

Parameters

- **dispatcher** (*vumi.dispatchers.BaseDispatchWorker*) – The dispatcher this routing class is part of.
- **config** (*dict*) – The configuration options passed to the dispatcher.

If you are subclassing this class, you should not override `__init__()`. Custom setup should be done in *setup_routing()* instead.

setup_routing()

Perform setup required for router.

Return type Deferred or None

Returns May return a Deferred that is called when setup is complete

teardown_routing()
Perform teardown required for router.

Return type Deferred or None

Returns May return a Deferred that is called when teardown is complete

dispatch_inbound_message(*msg*)
Dispatch an inbound user message to a publisher.

Parameters **msg** (*vumi.message.TransportUserMessage*) – Message to dispatch.

dispatch_inbound_event(*msg*)
Dispatch an event to a publisher.

Parameters **msg** (*vumi.message.TransportEvent*) – Message to dispatch.

dispatch_outbound_message(*msg*)
Dispatch an outbound user message to a publisher.

Parameters **msg** (*vumi.message.TransportUserMessage*) – Message to dispatch.

Example of a simple router implementation from `vumi.dispatcher.base`:

```
class SimpleDispatchRouter(BaseDispatchRouter):
    """Simple dispatch router that maps transports to apps.

    Configuration options:

    :param dict route_mappings:
        A map of *transport_names* to *exposed_names*. Inbound
        messages and events received from a given transport are
        dispatched to the application attached to the corresponding
        exposed name.

    :param dict transport_mappings: An optional re-mapping of
        *transport_names* to *transport_names*.  By default, outbound
        messages are dispatched to the transport attached to the
        *endpoint* with the same name as the transport name given in
        the message. If a transport name is present in this
        dictionary, the message is instead dispatched to the new
        transport name given by the re-mapping.
    """

    def dispatch_inbound_message(self, msg):
        names = self.config['route_mappings'][msg['transport_name']]
        for name in names:
            # copy message so that the middleware doesn't see a particular
            # message instance multiple times
            self.dispatcher.publish_inbound_message(name, msg.copy())

    def dispatch_inbound_event(self, msg):
        names = self.config['route_mappings'][msg['transport_name']]
        for name in names:
            # copy message so that the middleware doesn't see a particular
            # message instance multiple times
            self.dispatcher.publish_inbound_event(name, msg.copy())

    def dispatch_outbound_message(self, msg):
        name = msg['transport_name']
        name = self.config.get('transport_mappings', {}).get(name, name)
        if name in self.dispatcher.transport_publisher:
```

```
            self.dispatcher.publish_outbound_message(name, msg)
        else:
            log.error(DispatcherError(
                'Unknown transport_name: %s, discarding %r' % (
                    name, msg.payload)))
```

CHAPTER 7

Middleware

Middleware provides additional functionality that can be attached to any existing transport, application or dispatcher worker. For example, middleware could log inbound and outbound messages, store delivery reports in a database or modify a message.

Attaching middleware to your worker is fairly straight forward. Just extend your YAML configuration file with lines like:

```
middleware:
    - mw1: vumi.middleware.LoggingMiddleware

mw1:
    log_level: info
```

The *middleware* section contains a list of middleware items. Each item consists of a *name* (e.g. *mw1*) for that middleware instance and a *class* (e.g. `vumi.middleware.LoggingMiddleware`) which is the full Python path to the class implementing the middleware. A *name* can be any string that doesn't clash with another top-level configuration option – it's just used to look up the configuration for the middleware itself.

If a middleware class doesn't require any additional parameters, the configuration section (i.e. the *mw1: debug_level* ... in the example above) may simply be omitted.

Multiple layers of middleware may be specified as follows:

```
middleware:
    - mw1: vumi.middleware.LoggingMiddleware
    - mw2: mypackage.CustomMiddleware
```

You can think of the layers of middleware sitting on top of the underlying transport or application worker. Messages being consumed by the worker enter from the top and are processed by the middleware in the order you have defined them and eventually reach the worker at the bottom. Messages published by the worker start at the bottom and travel up through the layers of middleware before finally exiting the middleware at the top.

Further reading:

7.1 Builtin middleware

Vumi ships with a small set of generically useful middleware:

Vumi middleware

7.1.1 AddressTranslationMiddleware

Overwrites *to_addr* and *from_addr* values based on a simple mapping. Useful for debugging and testing.

class vumi.middleware.address_translator.**AddressTranslationMiddleware**(*name*, *config*, *worker*)

Address translation middleware.

Used for mapping a set of *to_addr* values in outbound messages to new values. Inbound messages have the inverse mapping applied to their *from_addr* values.. This is useful during debugging, testing and development.

For example, you might want to make your Gmail address look like an MSISDN to an application to test SMS address handling, for instance. Or you might want to have an outgoing SMS end up at your Gmail account.

Configuration options:

Parameters **outbound_map** (*dict*) – Mapping of old *to_addr* values to new *to_addr* values for outbound messages. Inbound messages have the inverse mapping applied to *from_addr* values. Addresses not in this dictionary are not affected.

7.1.2 LoggingMiddleware

Logs messages, events and failures as they enter or leave a transport.

class vumi.middleware.logging.**LoggingMiddleware**(*name*, *config*, *worker*)

Middleware for logging messages published and consumed by transports and applications.

Optional configuration:

Parameters

- **log_level** (*string*) – Log level from vumi.log to log inbound and outbound messages and events at. Default is *info*.
- **failure_log_level** (*string*) – Log level from vumi.log to log failure messages at. Default is *error*.

7.1.3 TaggingMiddleware

class vumi.middleware.tagger.**TaggingMiddleware**(*name*, *config*, *worker*)

Transport middleware for adding tag names to inbound messages and for adding additional parameters to outbound messages based on their tag.

Transports that wish to eventually have incoming messages associated with an existing message batch by vumi.application.MessageStore or via vumi.middleware.StoringMiddleware need to ensure that incoming messages are provided with a tag by this or some other middleware.

Configuration options:

Parameters

- **incoming** (*dict*) –
  - **addr_pattern** (*string*): Regular expression matching the to_addr of incoming messages. Incoming messages with to_addr values that don't match the pattern are not modified.
  - **tagpool_template** (*string*): Template for producing tag pool from successful matches of *addr_pattern*. The string is expanded using *match.expand(tagpool_template)*.
  - **tagname_template** (*string*): Template for producing tag name from successful matches of *addr_pattern*. The string is expanded using *match.expand(tagname_template)*.
- **outgoing** (*dict*) –
  - **tagname_pattern** (*string*): Regular expression matching the tag name of outgoing messages. Outgoing messages with tag names that don't match the pattern are not modified. Note: The tag pool the tag belongs to is not examined.
  - **msg_template** (*dict*): A dictionary of additional key-value pairs to add to the outgoing message payloads whose tag matches *tag_pattern*. Values which are strings are expanded using *match.expand(value)*. Values which are dicts are recursed into. Values which are neither are left as is.

7.1.4 StoringMiddleware

class `vumi.middleware.message_storing.`**`StoringMiddleware`**(*name*, *config*, *worker*)

Middleware for storing inbound and outbound messages and events.

Failures are not stored currently because these are typically stored by `vumi.transports.FailureWorker` instances.

Messages are always stored. However, in order for messages to be associated with a particular batch_id (see `vumi.application.MessageStore`) a batch needs to be created in the message store (typically by an application worker that initiates sending outbound messages) and messages need to be tagged with a tag associated with the batch (typically by an application worker or middleware such as `vumi.middleware.TaggingMiddleware`).

Configuration options:

Parameters

- **store_prefix** (*string*) – Prefix for message store keys in key-value store. Default is 'message_store'.
- **redis_manager** (*dict*) – Redis configuration parameters.
- **riak_manager** (*dict*) – Riak configuration parameters. Must contain at least a bucket_prefix key.
- **store_on_consume** (*bool*) – `True` to store consumed messages as well as published ones, `False` to store only published messages. Default is `True`.

7.2 Implementing your own middleware

A middleware class provides four handler functions, one for processing each of the four kinds of messages transports, applications and dispatchers typically send and receive (i.e. inbound user messages, outbound user messages, event messages and failure messages).

Although transport and application middleware potentially both provide the same sets of handlers, the two make use of them in slightly different ways. Inbound messages and events are published by transports but consumed by applications while outbound messages are opposite. Failure messages are not seen by applications at all and are allowed only so that certain middleware may be used on both transports and applications. Dispatchers both consume and publish all kinds of messages except failure messages.

Middleware is required to have the same interface as the *BaseMiddleware* class which is described below. Two subclasses, `TransportMiddleware` and `ApplicationMiddleware`, are provided but subclassing from these is just a hint as to whether a piece of middleware is intended for use on transports or applications (middleware for use on both or for dispatchers may inherit from *BaseMiddleware*). The two subclasses provide identical interfaces and no extra functionality.

class `vumi.middleware.`**BaseMiddleware**(*name*, *config*, *worker*)
Common middleware base class.

This is a convenient definition of and set of common functionality for middleware classes. You need not subclass this and should not instantiate this directly.

The `__init__()` method should take exactly the following options so that your class can be instantiated from configuration in a standard way:

Parameters

- **name** (*string*) – Name of the middleware.
- **config** (*dict*) – Dictionary of configuraiton items.
- **worker** (*vumi.service.Worker*) – Reference to the transport or application being wrapped by this middleware.

If you are subclassing this class, you should not override `__init__()`. Custom setup should be done in *setup_middleware()* instead. The config class can be overidden by replacing the `config_class` class variable.

CONFIG_CLASS
alias of `BaseMiddlewareConfig`

setup_middleware()
Any custom setup may be done here.

Return type Deferred or None

Returns May return a deferred that is called when setup is complete.

teardown_middleware()
"Any custom teardown may be done here

Return type Deferred or None

Returns May return a Deferred that is called when teardown is complete

handle_consume_inbound(*message*, *connector_name*)
Called when an inbound transport user message is consumed.

The other methods listed below all function in the same way. Only the kind and direction of the message being processed differs.

- *handle_publish_inbound()*
- *handle_consume_outbound()*
- *handle_publish_outbound()*
- *handle_consume_event()*

- *handle_publish_event()*
- *handle_failure()*

By default, the `handle_consume_*` and `handle_publish_*` methods call their `handle_*` equivalents.

Parameters

- **message** (*vumi.message.TransportUserMessage*) – Inbound message to process.
- **connector_name** (*string*) – The name of the connector the message is being received on or sent to.

Return type vumi.message.TransportUserMessage

Returns The processed message.

handle_publish_inbound(*message*, *connector_name*)
Called when an inbound transport user message is published.

See *handle_consume_inbound()*.

handle_inbound(*message*, *connector_name*)
Default handler for published and consumed inbound messages.

See *handle_consume_inbound()*.

handle_consume_outbound(*message*, *connector_name*)
Called when an outbound transport user message is consumed.

See *handle_consume_inbound()*.

handle_publish_outbound(*message*, *connector_name*)
Called when an outbound transport user message is published.

See *handle_consume_inbound()*.

handle_outbound(*message*, *connector_name*)
Default handler for published and consumed outbound messages.

See *handle_consume_inbound()*.

handle_consume_event(*event*, *connector_name*)
Called when a transport event is consumed.

See *handle_consume_inbound()*.

handle_publish_event(*event*, *connector_name*)
Called when a transport event is published.

See *handle_consume_inbound()*.

handle_event(*event*, *connector_name*)
Default handler for published and consumed events.

See *handle_consume_inbound()*.

handle_consume_failure(*failure*, *connector_name*)
Called when a failure message is consumed.

See *handle_consume_inbound()*.

handle_publish_failure(*failure*, *connector_name*)
Called when a failure message is published.

See *handle_consume_inbound()*.

handle_failure (*failure*, *connector_name*)
Called to process a failure message (`vumi.transports.failures.FailureMessage`).

See *handle_consume_inbound()*.

Example of a simple middleware implementation from `vumi.middleware.logging`:

```
class LoggingMiddleware(BaseMiddleware):
    """Middleware for logging messages published and consumed by
    transports and applications.

    Optional configuration:

    :param string log_level:
        Log level from :mod:`vumi.log` to log inbound and outbound
        messages and events at. Default is `info`.
    :param string failure_log_level:
        Log level from :mod:`vumi.log` to log failure messages at.
        Default is `error`.
    """
    CONFIG_CLASS = LoggingMiddlewareConfig

    def setup_middleware(self):
        log_level = self.config.log_level
        self.message_logger = getattr(log, log_level)
        failure_log_level = self.config.failure_log_level
        self.failure_logger = getattr(log, failure_log_level)

    def _log(self, direction, logger, msg, connector_name):
        logger("Processed %s message for %s: %s" % (
                direction, connector_name, msg.to_json()))
        return msg

    def handle_inbound(self, message, connector_name):
        return self._log(
            "inbound", self.message_logger, message, connector_name)

    def handle_outbound(self, message, connector_name):
        return self._log(
            "outbound", self.message_logger, message, connector_name)

    def handle_event(self, event, connector_name):
        return self._log("event", self.message_logger, event, connector_name)

    def handle_failure(self, failure, connector_name):
        return self._log(
            "failure", self.failure_logger, failure, connector_name)
```

7.2.1 How your middleware is used inside Vumi

While writing complex middleware, it may help to understand how a middleware class is used by Vumi transports and applications.

When a transport or application is started a list of middleware to load is read from the configuration. An instance of each piece of middleware is created and then `setup_middleware()` is called on each middleware object in order. If any call to `setup_middleware()` returns a `Deferred`, setup will continue after the deferred has completed.

Once the middleware has been setup it is combined into a `MiddlewareStack`. A middleware stack has two important methods `apply_consume()` and `apply_publish()` The former is used when a message is being consumed and applies the appropriate handlers in the order listed in the configuration file. The latter is used when a message is being published and applies the handlers in the *reverse* order.

CHAPTER 8

Metrics

Metrics are a means for workers to publish statistics about their operations for real-time plotting and later analysis. Vumi provides built-in support for publishing metric values to Carbon (the storage engine used by Graphite).

8.1 Using metrics from a worker

The set of metrics a worker wishes to plublish are managed via a *MetricManager* instance. The manager acts both as a container for the set of metrics and the publisher that pushes metric values out via AMQP.

Example:

```
class MyWorker(Worker):

    def startWorker(self, config):
        self.metrics = yield self.start_publisher(MetricManager,
                                                  "myworker.")
        self.metrics.register(Metric("a.value"))
        self.metrics.register(Count("a.count"))
```

In the example above a *MetricManager* publisher is started. All its metric names with be prefixed with *myworker.*. Two metrics are registered – *a.value* whose values will be averaged and *a.count* whose values will be summed. Later, the worker may set the metric values like so:

```
self.metrics["a.value"].set(1.23)
self.metrics["a.count"].inc()
```

class vumi.blinkenlights.metrics.**MetricManager**(*prefix*, *publish_interval=5*, *on_publish=None*, *publisher=None*)

Utility for creating and monitoring a set of metrics.

Parameters

- **prefix** (*str*) – Prefix for the name of all metrics registered with this manager.
- **publish_interval** (*int in seconds*) – How often to publish the set of metrics.
- **on_publish** (*f(metric_manager)*) – Function to call immediately after metrics after published.

oneshot(*metric*, *value*)

Publish a single value for the given metric.

Parameters

- **metric** (*Metric*) – Metric object to register. Will have the manager's prefix added to its name.
- **value** (*float*) – The value to publish for the metric.

publish_metrics()
Publish all waiting metrics.

register(*metric*)
Register a new metric object to be managed by this metric set.

A metric can be registered with only one metric set.

Parameters **metric** (*Metric*) – Metric object to register. The metric will have its *.manage()* method called with this manager as the manager.

Return type For convenience, returns the metric passed in.

start(*channel*)
Start publishing metrics in a loop.

start_polling()
Start the metric polling and publishing task.

stop()
Stop publishing metrics.

stop_polling()
Stop the metric polling and publishing task.

8.2 Metrics

A *Metric* object publishes floating point values under a metric *name*. The name is created by combining the *prefix* from a metric manager with the *suffix* provided when the metric is constructed. A metric may only be registered with a single *MetricManager*.

When a metric value is *set* the value is stored in an internal list until the *MetricManager* polls the metric for values and publishes them.

A metric includes a list of aggregation functions to request that the metric aggregation workers apply (see later sections). Each metric class has a default list of aggregators but this may be overridden when a metric is created.

class vumi.blinkenlights.metrics.**Metric**(*name*, *aggregators=None*)
Simple metric.

Values set are collected and polled periodically by the metric manager.

Parameters

- **name** (*str*) – Name of this metric. Will be appened to the *MetricManager* prefix when this metric is published.
- **aggregators** (*list of aggregators, optional*) – List of aggregation functions to request eventually be applied to this metric. The default is to average the value.

Examples:

```
>>> mm = MetricManager('vumi.worker0.')
>>> my_val = mm.register(Metric('my.value'))
>>> my_val.set(1.5)
>>> my_val.name
'my.value'
```

DEFAULT_AGGREGATORS = [<vumi.blinkenlights.metrics.Aggregator object at 0x7f7c496abfd0>]
Default aggregators are [AVG]

manage(*manager*)
Called by *MetricManager* when this metric is registered.

poll()
Called periodically by the *MetricManager*.

set(*value*)
Append a value for later polling.

class vumi.blinkenlights.metrics.**Count**(*name*, *aggregators=None*)
Bases: *vumi.blinkenlights.metrics.Metric*

A simple counter.

Examples:

```
>>> mm = MetricManager('vumi.worker0.')
>>> my_count = mm.register(Count('my.count'))
>>> my_count.inc()
```

DEFAULT_AGGREGATORS = [<vumi.blinkenlights.metrics.Aggregator object at 0x7f7c496abf10>]
Default aggregators are [SUM]

inc()
Increment the count by 1.

class vumi.blinkenlights.metrics.**Timer**(**args*, ***kws*)
Bases: *vumi.blinkenlights.metrics.Metric*

A metric that records time spent on operations.

Examples:

```
>>> mm = MetricManager('vumi.worker0.')
>>> my_timer = mm.register(Timer('hard.work'))
```

Using the timer as a context manager:

```
>>> with my_timer.timeit():
>>>     process_data()
```

Using the timer without a context manager:

```
>>> event_timer = my_timer.timeit()
>>> event_timer.start()
>>> d = process_other_data()
>>> d.addCallback(lambda r: event_timer.stop())
```

Note that timers returned by *timeit* may only have *start* and *stop* called on them once (and only in that order).

Note: Using `.start()` or `.stop()` directly or via using the *Timer* instance itself as a context manager is deprecated because they are not re-entrant and it's easy to accidentally overlap multiple calls to `.start()` and `.stop()` on the same *Timer* instance (e.g. by letting the reactor run in between).

All applications should be updated to use `.timeit()`.

Deprecated use of `.start()` and `.stop()`:

```
>>> my_timer.start()
>>> try:
>>>     process_other_data()
>>> finally:
>>>     my_timer.stop()
```

Deprecated use of `.start()` and `.stop()` via using the *Timer* itself as a context manager:

```
>>> with my_timer:
>>>     process_more_data()
```

DEFAULT_AGGREGATORS = [<vumi.blinkenlights.metrics.Aggregator object at 0x7f7c496abfd0>]
Default aggregators are [AVG]

8.3 Aggregation functions

Metrics declare which aggregation functions they wish to have applied but the actual aggregation is performed by aggregation workers. All values sent during an aggregation interval are aggregated into a single new value.

Aggregation fulfils two primary purposes:

- To combine metrics from multiple workers into a single aggregated value (e.g. to determine the average time taken or total number of requests processed across multiple works).
- To produce metric values at fixed time intervals (as is commonly required by metric storage backends such as Graphite and RRD Tool).

The aggregation functions currently available are:

- `SUM` – returns the sum of the supplied values.
- `AVG` – returns the arithmetic mean of the supplied values.
- `MIN` – returns the minimum value.
- `MAX` – returns the maximum value.

All aggregation functions return the value 0.0 if there are no values to aggregate.

New aggregators may be created by instantiating the *Aggregator* class.

Note: The aggregator must be instantiated in both the process that generates the metric (usually a worker) and the process that performs the aggregation (usually an aggregation worker).

---

class vumi.blinkenlights.metrics.**Aggregator**(*name*, *func*)
Registry of aggregate functions for metrics.

Parameters

- **name** (*str*) – Short name for the aggregator.
- **func** (*f(list of values) -> float*) – The aggregation function. Should return a default value if the list of values is empty (usually this default is 0.0).

8.4 Metrics aggregation system

The metric aggregation system consists of `MetricTimeBucket` and `MetricAggregator` workers.

The `MetricTimeBucket` workers pull metrics messages from the *vumi.metrics* exchange and publish them on the *vumi.metrics.buckets* exchange under a routing key specific to the `MetricAggregator` which should process them. Once sufficient time has passed for all metrics for a specific time period (a.k.a. time bucket) to have arrived at the aggregator, the requested aggregation functions are applied and the resulting aggregated metrics are published to the *vumi.metrics.aggregates* exchange.

A typical metric aggregation setup might consist of the following workers: * 2 `MetricTimeBucket` workers * 3 `MetricAggregator` workers * a final metric collector, e.g. `GraphiteMetricsCollector`.

A shell script to start-up such a setup might be:

```
#!/bin/bash
BUCKET_OPTS="--worker_class=vumi.blinkenlights.MetricTimeBucket \
--set-option=buckets:3 --set-option=bucket_size:5"

AGGREGATOR_OPTS="--worker_class=vumi.blinkenlights.MetricAggregator \
--set-option=bucket_size:5"

GRAPHITE_OPTS="--worker_class=vumi.blinkenlights.GraphiteMetricsCollector"

twistd -n vumi_worker $BUCKET_OPTS &
twistd -n vumi_worker $BUCKET_OPTS &

twistd -n vumi_worker $AGGREGATOR_OPTS --set-option=bucket:0 &
twistd -n vumi_worker $AGGREGATOR_OPTS --set-option=bucket:1 &
twistd -n vumi_worker $AGGREGATOR_OPTS --set-option=bucket:2 &

twistd -n vumi_worker $GRAPHITE_OPTS &
```

8.5 Publishing to Graphite

The `GraphiteMetricsCollector` collects aggregate metrics (produced by the metrics aggregators) and publishes them to Carbon (Graphite's metric collection package) over AMQP.

You can read about installing a configuring Graphite at http://graphite.wikidot.com but at the very least you will have to enable AMQP support by setting:

```
[cache]
ENABLE_AMQP = True
AMQP_METRIC_NAME_IN_BODY = False
```

in Carbon's configuration file.

If you have the metric aggregation system configured as in the section above you can start Carbon cache using:

```
carbon-cache.py --config <config file> --debug start
```

CHAPTER 9

Vumi Roadmap

The roadmap outlines features intended for upcoming releases of Vumi. Information on older releases can be found in Release Notes.

9.1 Version 0.5

Projected date end of April 2012

- add ability to identify a single user across multiple transports as per Identity Datastore.
- associate messages with billing accounts. See Accounting.
- support custom application logic in Javascript. See Custom Application Logic.
- support dynamic addition and removal of workers. See Dynamic Workers.
- add Riak storage support. See Datastore Access.

9.2 Future

Future plans that have not yet been scheduled for a specific milestone are outlined in the following sections. Parts of these features may already have been implemented or have been included in the detailed roadmap above:

9.2.1 Blinkenlights

Failure is guaranteed, what will define our success when things fail is how we respond. We can only respond as good as we can gauge the performance of the individual components that make up Vumi. Blinkenlights is a technical management module for Vumi that gives us that insight. It will give us accurate and realtime data on the general health and well being of all of the different moving parts.

Implementation Details

Blinkenlights connects to a dedicated exchange on our message broker. All messages broadcast to this exchange are meant for Blinkenlights to consume. Every component connected to our message broker has a dedicated channel for broadcasting status updates to Blinkenlights. Blinkenlights will consume these messages and make them available for viewing in a browser.

Note: Blinkenlights will probably remain intentionally ugly as we do not want people to mistake this for a dashboard.

Typical Blinkenlights Message Payload

The messages that Blinkenlights are JSON encoded dictionaries. An example Blinkenlights message only requires three keys:

```
{
    "name": "SMPP Transport 1",
    "uuid": "0f148162-a25b-11e0-ba57-0017f2d90f78",
    "timestamp": [2011, 6, 29, 15, 3, 23]
}
```

name The name of the component connected to AMQP. Preferably unique.

uuid An identifier for this component, must be unique.

timestamp A UTC timestamp as a list in the following format: [YYYY, MM, DD, HH, MM, SS]. We use a list as Javascript doesn't have a built-in date notation for JSON.

The components should publish a status update in the form of a JSON dictionary every minute. If an update hasn't been received for two minutes then the component will be flagged as being in an error state.

Any other keys and values can be added to the dictionary, they'll be published in a tabular format. Each transport is free to add whatever relevant key/value pairs. For example, for SMPP a relevant extra key/value pair could be messages per second processed.

9.2.2 Dynamic Workers

This has been completely rethought since the last version of this document. (This is still very much a work in progress, so please correct, update or argue as necessary.)

In the old system, we have a separate `twistd` process for each worker, managed by supervisord. In the Brave New Dyanmic Workers World, we will be able to start and stop arbitrary workers in a `twistd` process by sending a Blinkenlights message to a supervisor worker in that process.

Advantages:

- We can manage Vumi workers separately from OS processes, which gives us more flexibility.
- We can segregate workers for different projects/campaigns into different processes, which can make accounting easier.

Disadvantages:

- We have to manage Vumi workers separately from OS processes, which requires more work and higher system complexity. (This is the basic cost of the feature, though, and it's worth it for the flexibility.)
- A badly-behaved worker can take down a bunch of other workers if it manages to kill/block the process.

Supervisor workers

Note: I have assumed that the supervisor will be a worker rather than a static component of the process. I don't have any really compelling reasons either way, but making it a worker lets us coexist easily with the current one-worker-one-process model.

A supervisor worker is nothing more than a standard worker that manages other workers within its process. Its responsibilites have not yet been completely defined, but will likely the following:

- Monitoring and reportng process-level metrics.
- Starting and stopping workers as required.

Monitoring will use the usual Blinkenlights mechanisms, and will work the same way as any other worker's monitoring. The supervisor will also provide a queryable status API to allow interrogation via Blinkenlights. (Format to be decided.)

Starting and stopping workers will be done via Blinkenlights messages with a payload format similar to the following:

```
{
    "operation": "vumi_worker",
    "worker_name": "SMPP Transport for account1",
    "worker_class": "vumi.workers.smpp.transport.SMPPTransport",
    "worker_config": {
        "host": "smpp.host.com",
        "port": "2773",
        "username": "account1",
        "password": "password",
        "system_id": "abc",
    },
}
```

We could potentially even have a hierarchy of supervisors, workers and hybrid workers:

```
process
 +- supervisor
     +- worker
     +- worker
     +- hybrid supervisor/worker
     |   +- worker
     |   +- worker
     +- worker
```

9.2.3 Identity Datastore

To be confirmed.

9.2.4 Conversation Datastore

We are currently using PostgreSQL as our main datastore and are using Django's ORM as our means of interacting with it. **This however is going to change.**

What we are going towards:

1. HBase as our conversation store.
2. Interface with it via HBase's Stargate REST APIs.

9.2.5 Custom Application Logic

Javascript is the DSL of the web. Vumi will allow developers used to front-end development technologies to build and host frontend and backend applications using Javascript as the main language.

Pros:

- Javascript lends itself well to event based programming, ideal for messaging.
- Javascript is well known to the target audience.
- Javascript is currently experiencing major development in terms of performance improvements by Google, Apple, Opera & Mozilla.
- Javascript has AMQP libraries available.

Cons:

- We would need to sandbox it (but we'd need to do that regardless, Node.js has some capabilities for this but I'd want the sandbox to restrict any file system access).
- We're introducing a different environment next to Python.
- Data access could be more difficult than Python.

How would it work?

Application developers could bundle (zip) their applications as follows:

- application/index.html is the HTML5 application that we'll host.
- application/assets/ is the Javascript, CSS and images needed by the frontend application.
- workers/worker.js has the workers that we'd fire up to run the applications workers for specific campaigns. These listen to messages arriving over AMQP as 'events' trigger specific pieces of logic for that campaign.

The HTML5 application would have direct access to the Vumi JSON APIs, zero middleware would be needed.

This application could then be uploaded to Vumi and we'd make it available in their account and link their logic to a specific SMS short/long code, twitter handle or USSD code.

Python would still be driving all of the main pieces (SMPP, Twitter, our JSON API's etc...) only the hosted applications would be javascript based. Nothing is stopping us from allowing Python as a worker language at a later stage as well.

9.2.6 Accounting

Note: Accounting at this stage is the responsibility of the campaign specific logic, this however will change over time.

Initially Vumi takes a deliberately simple approach to accounting.

What Vumi should do now:

1. An account can be active or inactive.
2. Messaging only takes place for active accounts, messages submitted for inactive accounts are discarded and unrecoverable.
3. Every message sent or received is linked to an account.
4. Every message sent or received is timestamped.
5. All messages sent or received can be queried and exported by date per account.

What Vumi will do in the future:

1. Send and receive messages against a limited amount of message credits.
2. Payment mechanisms in order to purchase more credits.

9.2.7 Datastore Access

Currently all datastore access is via Django's ORM with the database being PostgreSQL. This is going to change.

We will continue to use PostgreSQL for data that isn't going to be very write heavy. These include:

1. User accounts
2. Groups
3. Accounting related data (related to user accounts and groups)

The change we are planning for is to be using HBase for the following data:

1. Conversation
2. Messages that are part of a conversation

CHAPTER 10

Release Notes

10.1 Version 0.6

NOTE: Version 0.6.x is backward-compatible with 0.5.x for the most part, with some caveats. The first few releases will be removing a bunch of obsolete and deprecated code and replacing some of the internals of the base worker. While this will almost certainly not break the majority of things built on vumi, old code or code that relies too heavily on the details of worker setup may need to be fixed.

Version 0.6.13

Date released 10 January 2017

- Ensure that the keys for the data coding mapping of the SMPP transport get converted to ints, as it's not possible with Junebug's JSON channel config to represent dictionary keys as integers.
- Update Message Sender tests to work with the new Riak client.

Version 0.6.12

Date released 23 September 2016

- Update the HttpRPCTransport to use the new vumi logging to be compatible with Junebug log collection.

Version 0.6.11

Date released 12 August 2016

- Change logging level from warning to info for SMPP disconnections, to reduce sentry entries when being throttled.

Version 0.6.10

Date released 27 July 2016

- Update Dmark transport to send null content at the start of a USSD session rather than sending the USSD code.

Version 0.6.9

Date released 27 July 2016

- Apply numerous cleanups to the Dockerfile.
- Use only decimal digits for session identifiers in the MTN Nigeria USSD XML over TCP transport.
- Add the ability to configure the PDU field the dialed USSD code is taken from in the 6D SMPP processor.
- Update tests to pass with Twisted 16.3.

Version 0.6.8

Date released 12 May 2016

- Allow disabling of delivery report handling as sometimes these cause more noise than signal.
- Embed the original SMPP transports delivery report status into the message transport metadata. This is useful information that applications may chose to act on.

Version 0.6.7

Date released 19 April 2016

- Re-fix the bug in the Vumi Bridge transport that prevents it making outbound requests.

Version 0.6.6

Date released 18 April 2016

- Fix bug in Vumi Bridge transport that prevented it making outbound requests.

Version 0.6.5

Date released 15 April 2016

- Update the Vumi Bridge transport to perform teardown more carefully (including tearing down the Redis manager and successfully tearing down even if start up failed halfway).
- Add support for older SSL CA certificates when using the Vumi Bridge transport to connect to Vumi Go.

Version 0.6.4

Date released 8 April 2016

- Fix object leak caused by creating lots of Redis submanagers.
- Remove deprecated manhole middleware.
- Update fake_connections wrapping of abortConnection to work with Twisted 16.1.

Version 0.6.3

Date released 31 March 2016

- Refactor and update the Vumi Bridge non-streaming HTTP API client, including adding status events and a web_path configuration option for use with Junebug.
- Remove the deprecated Vumi Bridge streaming HTTP API client.
- Add a Dockerfile entrypoint script.
- Rename the TWISTD_APPLICATION Dockerfile variable to TWISTD_COMMAND.
- Pin the version of Vumi installed in the Dockerfile.
- Update manhole middleware so that tests pass with Twisted 16.0.

Version 0.6.2

Date released 3 March 2016

- Add support for uniformly handling Redis ResponseErrors across different Redis implementations.

Version 0.6.1

Date released 2 March 2016

- Removed support for Python 2.6.
- Publish status messages from WeChat transport (for use with Junebug).
- A support for the rename command to FakeRedis.

- Add Dockerfile for running Vumi.
- Fixed typo in "How we do releases" documentation.

  Version 0.6.0

  Date released 7 Dec 2015
- Removed various obsolete test helper code in preparation for AMQP client changes.
- Started writing release notes again.

10.2 Version 0.5

No release notes for three and a half years. Sorry. :-(

10.3 Version 0.4

Version 0.4.0

Date released 16 Apr 2012

- added support for once-off scheduling of messages.
- added MultiWorker.
- added support for grouped messages.
- added support for middleware for transports and applicatons.
- added middleware for storing of all transport messages.
- added support for tag pools.
- added Mediafone transport.
- added support for setting global vumi worker options via a YAML configuration file.
- added a keyword-based message dispatcher.
- added a grouping dispatcher that assists with A/B testing.
- added support for sending outbound messages that aren't replies to application workers.
- extended set of message parameters supported by the http_relay worker.
- fixed twittytwister installation error.
- fixed bug in Integrat transport that caused it to send two new session messages.
- ported the TruTeq transport to the new message format.
- added support for longer messages to the Opera transport.
- wrote a tutorial.
- documented middleware and dispatchers.
- cleaned up of SMPP transport.
- removed UglyModel.
- removed Django-based vumi.webapp.
- added support for running vumi tests using tox.

10.4 Version 0.3

Version 0.3.1

Date released 12 Jan 2012

- Use yaml.safe_load everywhere YAML config files are loaded. This fixes a potential security issue which allowed those with write access to Vumi configuration files to run arbitrary Python code as the user running Vumi.
- Fix bug in metrics manager that unintentionally allowed two metrics with the same name to be registered.

Version 0.3.0

Date released 4 Jan 2012

- defined common message format.
- added user session management.
- added transport worker base class.
- added application worker base class.
- made workers into Twisted services.
- re-organized example application workers into a separate package and updated all examples to use common message format
- deprecated Django-based vumi.webapp
- added and deprecated UglyModel
- re-organized transports into a separate package and updated all transports except TruTeq to use common message (TruTeq will be migrated in 0.4 or a 0.3 point release).
- added satisfactory HTTP API(s)
- removed SMPP transport's dependency on Django

10.5 Version 0.2

Version 0.2.0

Date released 19 September 2011

- System metrics as per Blinkenlights.
- Realtime dashboarding via Geckoboard.

10.6 Version 0.1

Version 0.1.0

Date released 4 August 2011

- SMPP Transport (version 3.4 in transceiver mode)
  - Send & receive SMS messages.
  - Send & receive USSD messages over SMPP.

  - Supports SAR (segmentation and reassembly, allowing receiving of SMS messages larger than 160 characters).
  - Graceful reconnecting of a failed SMPP bind.
  - Delivery reports of SMS messages.
- XMPP Transport
  - Providing connectivity to Gtalk, Jabber and any other XMPP based service.
- IRC Transport
  - Currently used to log conversations going on in various IRC channels.
- GSM Transport (currently uses pygsm, looking at gammu as a replacement)
  - Interval based polling of new SMS messages that a GSM modem has received.
  - Immediate sending of outbound SMS messages.
- Twitter Transport
  - Live tracking of any combination of keywords or hashtags on twitter.
- USSD Transports for various aggregators covering 12 African countries.
- HTTP API for SMS messaging:
  - Sending SMS messages via a given transport.
  - Receiving SMS messages via an HTTP callback.
  - Receiving SMS delivery reports via an HTTP callback.
  - Querying received SMS messages.
  - Querying the delivery status of sent SMS messages.

Note: Looking for documentation for writing Javascript applications for the hosted Vumi Go environment? Visit http://vumi-go.readthedocs.org for documentation on the hosted platform and http://vumi-jssandbox-toolkit.readthedocs.org for documentation on the Javascript sandbox.

Getting Started:

- `installation`
- `getting-started`
- Writing your first Vumi app - Part 1 | Part 2
- ScaleConf workshop instructions

For developers:

CHAPTER 11

Routing Naming Conventions

11.1 Transports

Transports use the following routing key convention:

- <*transport_name*>.*inbound* for sending messages from users (to vumi applications).
- <*transport_name*>.*outbound* for receiving messages to send to users (from vumi applications).
- <*transport_name*>.*event* for sending message-related events (e.g. acknowledgements, delivery reports) to vumi applications.
- <*transport_name*>.*failures* for sending failed messages to failure workers.

Transports use the *vumi* exchange (which is a *direct* exchange).

11.2 Metrics

The routing keys used by metrics workers are detailed in the table below. Exchanges are *direct* unless otherwise specified.

Table 11.1: Routing Naming Conventions

Component	Consumer / Producer	Exchange	Exch. Type	Queue Name	Routing Key	Notes
MetricTime-Bucket						
	Consumer	vumi.metrics		vumi.metrics	vumi.metrics	
	Publisher	vumi.metrics.buckets		bucket.<number>	bucket.<number>	
MetricAggrega-tor						
	Consumer	vumi.metrics.buckets		bucket.<number>	bucket.<number>	
	Publisher	vumi.metrics.aggregates		vumi.metrics.aggregates	vumi.metrics.aggregates	
GraphiteMetric-sCollector						
	Consumer	vumi.metrics.aggregates		vumi.metrics.aggregates	vumi.metrics.aggregates	
	Publisher	graphite	topic	n/a	<metric name>	

CHAPTER 12

How we do releases

12.1 Update the release notes and roadmap

Update the Vumi Roadmap and Release Notes as necessary.

12.2 Create a release branch

Select a release series number and initial version number:

```
$ SERIES=0.1.x
$ VER=0.1.0a
```

Start by creating the release branch (usually from develop but you can also specify a commit to start from):

```
$ git flow release start $SERIES [<start point>]
```

Set the version in the release branch:

```
$ ./utils/bump-version.sh $VER
$ git add setup.py docs/conf.py vumi/__init__.py
$ git commit -m "Set initial version for series $SERIES"
```

Set the version number in the develop branch *if necessary*.

Push your changes to Github:

```
$ git push origin release/$SERIES
```

12.3 Tag the release

Select a series to release from and version number:

```
$ SERIES=0.1.x
$ VER=0.1.0
$ NEXTVER=0.1.1a
```

Bump version immediately prior to release and tag the commit:

```
$ git checkout release/$SERIES
$ ./utils/bump-version.sh $VER
$ git add setup.py docs/conf.py vumi/__init__.py
$ git commit -m "Version $VER"
$ git tag vumi-$VER
```

Bump version number on release branch:

```
$ ./utils/bump-version.sh $NEXTVER
$ git add setup.py docs/conf.py vumi/__init__.py
$ git commit -m "Bump release series version."
```

Merge to master *if this is a tag off the latest release series*:

```
$ git checkout master
$ git merge vumi-$VER
```

Push your changes to Github (don't forget to push the new tag):

```
$ git push
$ git push origin refs/tags/vumi-$VER
```

12.4 Release to PyPI

Select the version number:

```
$ VER=0.1.0
$ git checkout vumi-$VER
```

Register the release with PyPI:

```
$ python setup.py register
```

Build the source distribution package:

```
$ python setup.py sdist
```

Upload the release to PyPI:

```
$ twine-upload dist/vumi-$VER.tar.gz
```

Declare victory.

CHAPTER 13

Coding Guidelines

Code contributions to Vumi should:

- Adhere to the **PEP 8** coding standard.
- Come with unittests.
- Come with docstrings.

13.1 Vumi docstring format

- For classes, __init__ should be documented in the class docstring.
- Function docstrings should look like:

```
    def format_exception(etype, value, tb, limit=None):
        """Format the exception with a traceback.

        :type etype: exception class
        :param etype: exception type
        :param value: exception value
        :param tb: traceback object
        :param limit: maximum number of stack frames to show
        :type limit: integer or None
        :rtype: list of strings
        """
```

13.2 Unit tests

`test-helper-api`

CHAPTER 14

Indices and tables

- genindex
- modindex
- search

V

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V

X