Vumi Documentation

Release 0.4.0

Praekelt Foundation

April 16, 2012

CONTENTS

Contents:

CHAPTER

ONE

VUMI OVERVIEW

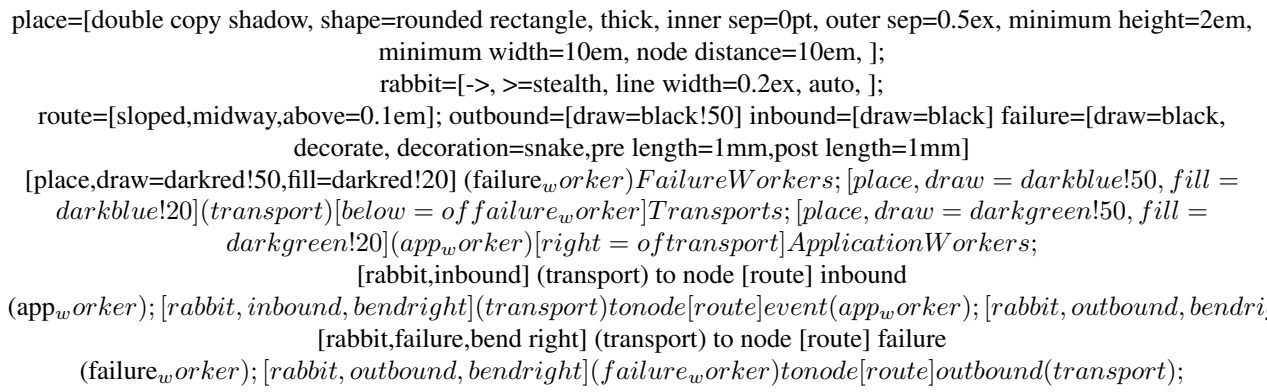

Figure 1.1: A simple Vumi worker setup

CHAPTER

TWO

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.

2.1 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.

2.2 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.

2.2.1 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:

```
{
    "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 dataT
    }
}
```

2.2.2 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:

```
transport_name: smpp_transport
url: http://127.0.0.1:8001/
```

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.

2.3 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
    start_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
    start_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
```

2.4 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

THREE

VUMI TRANSPORTS

Vumi supports various transports, the internal specifics are documented for each separately below.

Contents

3.1 HTTP Transport

A simple API for submitting Vumi messages into Vumi.

3.1.1 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.

3.1.2 Configuration parameters

```
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
```

The API implementation can be found here https://github.com/praekelt/vumi/blob/develop/vumi/transports/api/api.py

3.2 SMPP

An SMPP transport for version 3.4 of the protocol, operating in Transceiver mode.

3.2.1 Notes

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

3.2.2 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.

3.2.3 Configuration parameters

```
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
```

3.3 Vas2Nets

A WASP providing connectivity in Nigeria via an HTTP API.

3.3.1 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.

3.3.2 Configuration parameters

```
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>
```

CHAPTER

FOUR

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

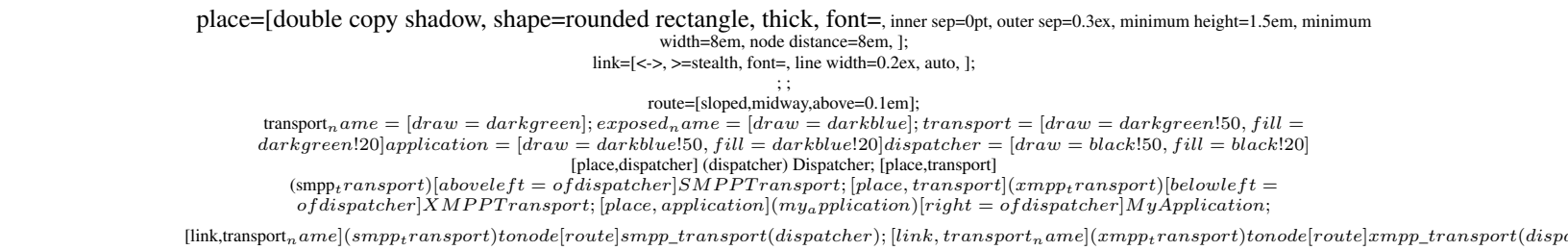

Figure 4.1: A simple dispatcher configuration. Boxes represent workers. Edges are routing links between workers. Edges are labelled with endpoint names (i.e. transport_names).

`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*.

4.1 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:

4.1.1 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'es 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.

4.1.2 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 routing messages.

`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:
            self.dispatcher.exposed_publisher[name].publish_message(msg)

    def dispatch_inbound_event(self, msg):
        names = self.config['route_mappings'][msg['transport_name']]
        for name in names:
            self.dispatcher.exposed_event_publisher[name].publish_message(msg)

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

CHAPTER

FIVE

MIDDLEWARE

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

Attaching middleware to your transport or application 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:

5.1 Builtin middleware

Vumi ships with a small set of generically useful middleware:

Vumi middleware

- LoggingMiddleware
- TaggingMiddleware
- StoringMiddleware

5.1.1 LoggingMiddleware

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

class vumi.middleware.**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*.

5.1.2 TaggingMiddleware

class vumi.middleware.**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.

5.1.3 StoringMiddleware

class vumi.middleware.**StoringMiddleware**(*name*, *config*, *worker*)
Middleware for storing inbound and outbound messages and events.

Failures are not stored currently because these are typically stored by :class:'vumi.transports.FailureWorker's.

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** (*dict*) – Redis configuration parameters.

5.2 Implementing your own middleware

A middleware class provides four handler functions, one for processing each of the four kinds of messages transports and applications typical sent 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.

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. 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.

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.

handle_inbound(*message*, *endpoint*)
Called when an inbound transport user message is published or consumed.

The other methods – `handle_outbound()`, `handle_event()`, `handle_failure()` – all function in the same way. Only the kind of message being processed differs.

Parameters

- **message** (*vumi.message.TransportUserMessage*) – Inbound message to process.
- **endpoint** (*string*) – The *transport_name* of the endpoint the message is being received on or send to.

Return type vumi.message.TransportUserMessage

Returns The processed message.

handle_outbound(*message*, *endpoint*)
Called to process an outbound transport user message. See `handle_inbound()`.

handle_event(*event*, *endpoint*)
Called to process an event message (`vumi.message.TransportEvent`). See `handle_inbound()`.

handle_failure(*failure*, *endpoint*)
Called to process a failure message (`vumi.transports.failures.FailureMessage`). See `handle_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'.
    """

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

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

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

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

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

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

5.2.1 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

SIX

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).

6.1 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*)

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.

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. Will have the manager's prefix added to its name.

Return type For convenience, returns the metric passed in.

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

stop()
Stop publishing metrics.

6.2 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**(*suffix*, *aggregators=None*)
Simple metric.

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

Parameters

- **suffix** (*str*) – Suffix to append to the MetricManager prefix to create the metric name.
- **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
'vumi.worker0.my.value'
```

DEFAULT_AGGREGATORS = [<vumi.blinkenlights.metrics.Aggregator object at 0x33eaa90>]
Default aggregators are [AVG]

manage(*prefix*)
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**(*suffix*, *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 0x33eafd0>]
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:
>>>     process_data()
```

Or equivalently using .start() and stop() directly:

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

DEFAULT_AGGREGATORS = [<vumi.blinkenlights.metrics.Aggregator object at 0x33eaa90>]
Default aggregators are [AVG]

6.3 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).

6.4 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.workers.blinkenlights.metrics.\
MetricTimeBucket --set-option=buckets:3 --set-option=bucket_size:5"

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

GRAPHITE_OPTS="--worker_class=vumi.workers.blinkenlights.metrics.\
GraphiteMetricsCollector"

twistd -n start_worker $BUCKET_OPTS &
twistd -n start_worker $BUCKET_OPTS &

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

twistd -n start_worker $GRAPHITE_OPTS &
```

6.5 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 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

SEVEN

VUMI ROADMAP

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

7.1 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*.

7.2 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:

7.2.1 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.

7.2.2 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": "start_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
```

7.2.3 Identity Datastore

To be confirmed.

7.2.4 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.

7.2.5 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.

7.2.6 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.

7.2.7 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

EIGHT

RELEASE NOTES

8.1 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.

8.2 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

8.3 Version 0.2

Version 0.2.0

Date released 19 September 2011

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

8.4 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.

Getting Started:

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

For developers:

CHAPTER

NINE

ROUTING NAMING CONVENTIONS

9.1 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).

9.2 Metrics

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

Table 9.1: 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

TEN

HOW WE DO RELEASES

10.1 Update the release notes and roadmap

Update the *Vumi Roadmap* and *Release Notes* as necessary.

10.2 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
$ 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
```

10.3 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
$ 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
$ 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
```

Declare victory.

CHAPTER

ELEVEN

CODING GUIDELINES

Code contributions to Vumi should:

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

11.1 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
    """
```

CHAPTER

TWELVE

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX

V