Gofer Documentation

Release 0.76

Jeff Ortel

Jun 29, 2018

Contents

Contents:

CHAPTER 1

Release Notes

1.1 gofer 2.12

Notes:

- Support python 2.7+ and 3.2+
- Python < 2.7 no longer supported.

Fixes:

- [2.12.1] Reload plugin when queue not-found or no-route condition is detected.

Deprecated:

1.2 gofer 2.11

Notes:

- Exit handler terminate threads.

Fixes:

- Fix compatibility python-amqp 2.1.4 Channel.wait().

Deprecated:

1.3 gofer 2.10

Notes:

- Added support for `soft` plugin shutdown. Mainly internal API enhancement but improves behavior of plugin `unload` and `reload`. Both operations now do a `soft` shutdown by default.
- The thread-pool design improved.

Fixes:

- The `hard` plugin/thread-pool shutdown aborted threads which caused reply messages to silently never be sent. Only affected `unload` and `reload` operations.

Deprecated:

1.4 gofer 2.9

Notes:

- Added `direct` and `fork` plugin decorators used to specify the RMI invocation model. Using one of these decorators is preferred to using the `model=` parameter to the `remote` decorator.
- Added memory profiler to metrics.
- Added context manager to Timer and associated decorator.

Fixes:

Deprecated:

1.5 gofer 2.8

Notes:

- Added support for RMI invocation models. The `direct` model is the default and invokes the remote method within the `goferd` process. This is the model used by <= 2.7. The new `fork` model spawns a child process for each method invocation. Invoking the method in a separate process provides isolation and better cancellation behavior. The isolation protects `goferd` against memory leaks and corruption potentially introduced by plugins (or code used by plugins). When using the `fork` model, RMI cancellation is implemented by killing the child process. As a result cancellation is certain and immediate regardless of whether cancellation is implemented by the method. See: `direct` and `fork` decorators.

Fixes:

- Proton message sending reliability regression introduced in 2.7.

Deprecated:

1.6 gofer 2.7

Notes:

- Add `gofer` command for interaction with goferd. See: `man gofer` for details. Packaged in gofer-tools. See newly added *[management]* section of */etc/pulp/agent.conf*.
- Plugin monitoring removed. Use gofer.agent.PluginContainer.load() and gofer.agent.PluginContainer.unload() instead.
- Added `@load` and `@unload` decorators. Plugins can participate in plugin loading and unloading.
- The *package* plugin has been rewritten to shell out instead of using the yum library. Much simpler.
- The gofer.rmi.shell module added. This can be used by plugins to easily and consistently provide functionality when using external commands is needed. Supports cancellation, progress reporting and returns stdout and stderr. The *system* and *package* plugins converted to use this.

- Improved debug logging in messaging adaptor reliability packages. This helps with troubleshooting AMQP issues.
- Added *latency* property to the *[main]* section of the plugin descriptor. Adding latency can be used for throttling and widening the request cancellation window.
- Canceled RMI requests discarded just prior to execution. Plugin still responsible for canceling requests already in progress.
- Reference plugins no longer packaged. The *test* plugin renamed to *demo* and not enabled by default.
- Dynamic plugin loading, reloading and unloading improved.
- As with every release, better unit test coverage.

Fixes:

- Minor memory leak fixed. The leak was ~384 bytes per request.
- Fixes issue whereby locally stored requests are routed to a plugin that no longer specifies a URL. The requests are discarded.
- AMQP connections used by plugin thread pool workers closed between requests. These connections can be idle/unused for long periods. Closing them reduces the number of open network connections.

Deprecated:

- The `uuid` in the [messaging] section of the plugin descriptor has been deprecated. Use [model] `queue` instead.
- The `@initializer` decorator has been deprecated. Use `@load` instead.
- Authorization has been support. It will continue to support authentication. This includes:
  - Shared secret. The *secret* option in the @remote decorator.
  - The @pam decorator.
  - The @user decorator.
  - The *pam* property in the message.

1.7 gofer 2.6

Notes:

- Fixed recursion issue in proton adapter reconnect logic.
- Add support for dynamic plugin loading, reloading and unloading.
- Add plugin monitoring. When enabled in agent.conf, the agent container will monitor the /etc/gofer/plugins directory for changes to plugin descriptors. When a descriptor has changed, the plugin is reloaded. When a *new* descriptor is found, the plugin is loaded. When a plugin descriptor is deleted, the plugin is unloaded. See [main] *monitor* property in agent.conf.
- Decentralized RMI scheduling. Each plugin has its own scheduler.
- Add support for RMI request forwarding to other plugins. Requests can be forwarded to other plugins when they cannot be satisfied by the target plugin. See [main] *accept* and *forward* properties for details.
- Much better AMQP connection management. When plugins are unloaded, all associated AMQP connections are closed.
- Add services API to the *system* plugin. The *Service* class supports *start*, *restart*, *stop* and *status* operations on services.

- The python-gofer-qpid package *Requires:* python-ssl. Needed so that python-qpid will support SSL.

Deprecated:

- The *maintenance window* feature and associated properties.

1.8 gofer 2.5

Notes:

- Added the python-gofer-proton messaging adapter. The adapter supports AMQP 1.0 and use the Apache Qpid `proton` library.
- The gofer.messaging.Exchange and gofer.messaging.Queue now support an additional `url` parameter which is used when `url` is not passed to specific method.
- NotFound raised when an AMQP node (queue) does not exist. See messaging.adapter.model for details on affected methods.

Deprecated:

- Using gofer.proxy.agent() has been deprecated.

1.9 gofer 2.4

Notes:

- AMQP Message durability fixed in python-amqp adapter.
- Added support for plugin descriptor properties that specifies the level to which the agent manages the broker model. Specifically, how the agent manages its request queue. The `[messaging]` *exchange* property was replace by support in the new [model] section documented below. See: descriptor documentation for details.
- Thread pool distribution fixed so that idle worker threads are selected when available.
- The python-amqplib AMQP library is no longer supported. It was redundant to support for python-amqp which is better maintained and widely available. This means that the python-gofer-amqplib package is no longer provided. Further that, AMQP-0-8 is no longer supported. This functionality can be resurrected on community request.
- The *amqp* adapter (python-amqp) updated to use EPOLL and basic_consume() instead of using dynamic polling and basic_get().
- By default, the proxy (caller) will no longer declare the agent queue. Since the *address* really specifies AMQP routing (exchange/queue), gofer cannot assume the queue name or properties. The agent declaration and binding is the responsibility of the agent or the (caller) application.
- The *qpid* adapter enables qpid heartbeat option on connections.

Added `[model]` section with the following properties:

- *managed* - Defines level of broker model management.
- *queue* - The name of the request queue.
- *exchange* - An (optional) exchange. The exchange is not declared/deleted.

1.10 gofer 2.3

Notes:

- Support for custom AMQP exchanges added. This includes an additional *exchange* option passed by callers to indicate the exchange to be used for temporary queues used for synchronous replies. For plugins, the descriptor was augmented to support an *exchange* property in the [messaging] section.

1.11 gofer 2.2

Not Released.

1.12 gofer 2.1

Not Released.

1.13 gofer 2.0

The 2.0 major release and contains API changes, minor message format changes and the removal of deprecated functionality. The goal of this release was to overhaul and streamline may major component and flows. This release also contains hundreds of new unit integration and unit tests as part of a major effort to reach 100% test coverage.

Overhauled:

- The agent thread pool was replaced with *Queue* based approach.
- Support for multiple messaging libraries. Standard messaging adapter model that uses delegation pattern instead of python meta-classes. Much better.

1.13.1 Concept changes

- The *transport* concept was replaced with *messaging adapters*. Each *adapter* implements an interface defined in the adapter model and provides integration with 3rd part AMQP messaging libraries. The *transport* option and descriptor property replaced with rich protocol handler support in the URL. See documented URL.
- All options are only supported when creating the agent proxy. They are no longer supported when constructing the stub. This semantic is not reserved for passing arguments to the remote object (class) constructor.
- The agent *uuid* is being phased out. RMI calls are routed to the agent based on the queue on which it was received. This term is being replaced by more AMQP related terms and concepts. An address has the format of: *exchange/queue* or *queue*.
- Support for agent broadcast was removed. This feature was deemed as not useful since most applications do not track requests using the serial number. Also, this can be easily implemented by the caller. Removed to make code paths and the API simpler.

1.13.2 API changes

There are API changes that affect both RMI calling (proxy) and the Plugin object exposed to agent plugins. Proxy changes pertain to the options passed to the *Agent* class and the *Stubs* created.

The *Agent* constructor changed from: Agent(uuid, **options) to: Agent(url, address, **options).

Example (adapter = qpid):

```
url = qpid+amqp://localhost
```

Option changes:

- *async* - Removed.
- *wait* - Added and indicates how long the caller is blocked on calls.
- *timeout* - Replaced by *ttl*.
- *ttl* - Added and replaces *timeout*. Strictly applies to request (and message) TTL.
- *ctag* - Replaced by *reply*.
- *reply* - Replaces *ctag* and is an AMQP address that specifies where RMI replies are sent.
- *any* - Removed and replaced by *data*.
- *data* - User defined data that is round-tripped back to the caller. Replaces *any*.
- *transport* - Replaced with rich protocol handlers supported by the URL.

1.13.3 Plugin (class) changes

All accessor methods replaced with @*property* and appear as attributes.

Here are a few major methods affected:

- enabled()
- get_uuid()
- get_url()
- get_cfg()

1.14 gofer 1.4

Here is a summary of 1.0 changes:

- Support for multiple *transports* was added.
- Message authentication added.
- The *accepted* status reply was added.
- The *watchdog* as removed.
- An ISO 8601 timestamp is included in all reply messages.

CHAPTER 2

Overview

Gofer provides an extensible, light weight, universal python agent. It has no relation to the Gopher protocol. The gofer core agent is a python daemon (service) that provides infrastructure for exposing a remote API and for running Recurring Actions. The APIs contributed by plugins are accessible by Remote Method Invocation (RMI). The transport for RMI is AMQP using the QPID message broker. Actions are also provided by plugins and are executed at the specified interval.

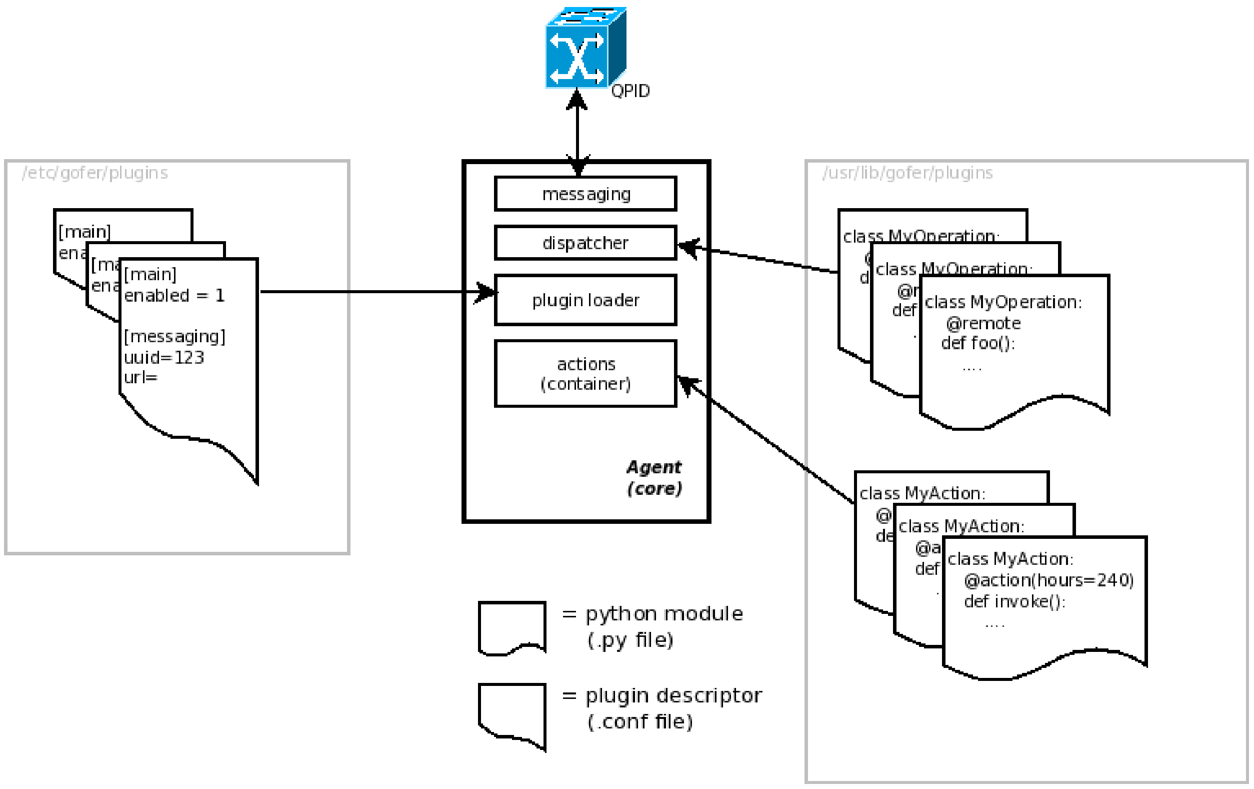
License: LGPLv2

Gofer provides:

- An agent (daemon)
- Plugin Container
- Remote access to API provided by plugins
- Action scheduling

Plugins provide:

- Remote API.
- Recurring (scheduled) actions
- Agent identity (optional)


QPID
/etc/gofer/plugins
[main]
enabled = 1
[messaging]
uuid=123
url=
messaging
dispatcher
plugin loader
actions
(container)
Agent
(core)
/usr/lib/gofer/plugins
class MyOperation:
@remote
def foo():
....
class MyAction:
@action(hours=240)
def invoke():
....
= python module
(.py file)
= plugin descriptor
(.conf file)

CHAPTER 3

Installation

3.1 Packages

Gofer is packaged into RPMs for Linux. These packages are as follows:

- **gofer** - The gofer agent (goferd).
- **python-gofer** - The common library.
- **python-gofer-qpid** - The python-qpid messaging adapter.
- **python-gofer-amqp** - The python-amqp messaging adapter.
- **python-gofer-proton** - The python-proton (AMQP 1.0) messaging adapter.

Depending on system capabilities, the *gofer* package registers goferd with systemd or upstart service managers.

3.1.1 Daemon

The daemon (service) is goferd. The `/etc/sysconfig/goferd` file defines configuration and contains the following properties:

- **PYTHON** - The python interpreter.
- **PYTHONOPTIMIZE** - Python interpreter optimization (0=disabled, 1=enabled).

3.2 Development

The gofer project is hosted by Github. To install from source, you must first clone the git repository. The python library can be installed using something like pip. Once installed, the goferd daemon can be installed.

Cloning the repository:

```
$ git clone https://github.com/jortel/gofer.git
```

In the examples below, <*git*> is the directory containing the cloned repository.

Files can be link or copied.

3.2.1 goferd

To install goferd:

```
# cp <git>/gofer/bin/goferd /usr/bin
```

3.2.2 systemd

To register goferd with systemd:

```
# cp <git>/gofer/usr/lib/systemd/system/goferd.service /usr/lib/systemd/system
```

3.2.3 upstart

To register goferd with upstart:

```
# cp <git>/gofer/etc/init.d/goferd /etc/init.d
# chkconfig --add goferd
```

CHAPTER 4

Getting Started

4.1 Installation

First, install and start QPID (qpidd)

Then,

1. Install gofer.

```
yum install gofer python-gofer-qpid
```

2. Edit the /etc/gofer/plugins/demo.conf and set the url to point at your broker. Then, set queue=123. Or, look in /var/log/messages to find the auto-assigned UUID for your system.

```
[main]
```

enabled = 1

[messaging] url=qpid+amqp://localhost

[model] queue=123

3. Start the goferd service.

```
service goferd start
```

4. Now, invoke the remote operations provided by the demo plugin:

4.1.1 Python

```
>>> from gofer.proxy import Agent
>>>
>>> url = 'amqp://localhost'
>>> agent = Agent(url, '123')
```

(continues on next page)

(continued from previous page)

```
>>> admin = agent.Admin()
>>> print admin.help()
    Plugins:
      demo
    Actions:
      demo.TestAction.hello() 0:10:00
    Methods:
     Admin.hello()
     Admin.help()
     Shell.run()
   Functions:
     demo.echo()
```

4.2 Writing A Plugin

The installing plugins is done in 6 easy steps.

1. Write your plugin descriptor.
2. Write your plugin module.
3. Copy (or symlink) the plugin descriptor (myplugin.conf) to /etc/gofer/plugins/
4. Copy (or symlink) the plugin module (myplugin.py) to /usr/lib/gofer/plugins/
5. Restart *goferd*
6. Add server side code to invoke remote methods

Let's create a plugin named *myplugin*.

4.2.1 Step 1

Create your plugin descriptor (myplugin.conf) as follows:

```
[main]
enabled = 1

[messaging]
url=qpid+amqp://localhost

[model]
queue=123
```

4.2.2 Step 2

Write your plugin. It will be defined in a module named *myplugin.py* and would look something like this:

```
from gofer.decorators import  *

class MyClass:

   @remote
```

(continues on next page)

(continued from previous page)

```
    def hello(self):
        return 'MyPlugin says, "hello".'
```

Stand alone (plain) functions may be decorated as *remote*.

Your class may have constructor arguments.

```
from gofer.decorators import *

@remote
def hello(self):
    return 'MyPlugin says, "hello".'
```

4.2.3 Step 3

Install or update your plugin descriptor.

```
cp myplugin.conf /etc/gofer/plugins
```

4.2.4 Step 4

Install or update your plugin.

```
cp myplugin.py /usr/lib/gofer/plugins
```

4.2.5 Step 5

Restart the gofer daemon.

```
sudo /etc/sbin/service goferd restart
```

4.2.6 Step 6

Add *server-side* code to invoke methods on your plugin.

This is done by instantiating a *proxy* for the agent. You need to specifying the *uuid* of the agent (plugin).

```
...
# your server code
from gofer.proxy import Agent

url = 'amqp://localhost'
uuid = '123'
agent = Agent(url, uuid)
myclass = agent.MyClass()
myclass.hello()
```

Invoke the stand alone function. Instead of instantiating the remote class, the function is invoked directly using the plugin module's namespace:

```
...
# your server code
from gofer.proxy import Agent

url = 'amqp://localhost'
uuid = '123'
agent = Agent(url, uuid)
agent.myplugin.hello()
```

4.3 Interactive Testing

After adding classes or methods in myplugin.py, you'll want to test them. First, ensure the plugin is still loading properly. The easiest way to do this is by examining the gofer log file at: /var/log/gofer/agent. At start up, you should see something like:

```
2010-11-08 08:49:04,909 [INFO][MainThread] __import() @ plugin.py:103 - plugin
↪"myplugin", imported as: "myplugin"
```

The gofer log (/var/log/messages) may be examined to verify that *Actions* are running as expected. Also, RMI requests (massages) are logged upon receipt in the gofer agent log.

Testing added *remote methods*, can be done easily using an interactive python (shell). Be sure your changes to *your* plugin have been picked up by *Gofer* by **restarting goferd**. Let's say you added a new class named "Foo" that has a remote method named … you guessed it: "bar". You can test your new stuff as follows:

```
[jortel@localhost pulp]$ python
Python 2.6.2 (r262:71600, Jun  4 2010, 18:28:04)
[GCC 4.4.3 20100127 (Red Hat 4.4.3-4)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> from gofer.proxy import Agent
>>>
>>> url = 'amqp://localhost'
>>> uuid = '123'
>>> agent = Agent(url, uuid)
>>> myclass = agent.MyClass()
>>> print myclass.hello()
MyPlugin says, "hello".
```

Another useful tool, it invoke *Admin.help()* from within interactive python as follows:

```
[jortel@localhost pulp]$ python
Python 2.6.2 (r262:71600, Jun  4 2010, 18:28:04)
[GCC 4.4.3 20100127 (Red Hat 4.4.3-4)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> from gofer.proxy import Agent
>>>
>>> url = 'amqp://localhost'
>>> uuid = '123'
>>> agent = Agent(url, uuid)
>>> admin = agent.Admin()
>>> print admin.help()

Plugins:
  demo
  myplugin
```

(continues on next page)

(continued from previous page)

```
Actions:
  demo.TestAction 0:10:00
Methods:
  myplugin.MyClass.hello()
  demo.AgentAdmin.hello()
  demo.AgentAdmin.help()
  demo.Shell.run()
Functions:
  demo.echo()
>>>
```

4.3.1 Security

The @remote decorator and gofer infrastructure supports (1) option:

- secret (default=None): A shared secret used for authentication. The value may be:
  - str
  - [str,..]
  - (str,..)
  - *callable*

In this example, MyClass.hello() must provide the *secret* to be invoked.

```
c = agent.MyClass(secret='mycathas9lives')
c.hello()
```

```
from gofer.decorators import *

class MyClass:

   @remote(secret='mycathas9lives')
   def hello(self):
       return 'MyPlugin says, "hello".'
```

The decorator also support the *secret* being a callable that returns the secret matched to the request.

Example:

```
from gofer.decorators import *

def getsecret():
   ...
  return secret

class MyClass:

   @remote(secret=getsecret)
   def hello(self):
       return 'MyPlugin says, "hello".'
```

CHAPTER 5

Tools

The gofer project includes the follow tools.

5.1 Command Line Interface

The `gofer` CLI provides both management (MGT) of *goferd* and remote method invocation (RMI). The management tool may be used to get the status of *goferd* and to dynamically load, reload and unload plugins. The management tool connects to *goferd* on the management port as defined in `/etc/gofer/agent.conf`. Management must be explicitly enabled.

```
[management]
enabled=1
host=localhost
port=5650
```

The RMI tool may be used to remotely invoke methods provided by plugins. It does not need management to be enabled.

Note: The CLI is new in gofer 2.7

5.1.1 Examples

The following are example of what can be done using `gofer`. It's assumed that management has been enabled on the default port and the host name is `localhost`. When configured with these defaults, the `(-h|--host) and (-p|--port)` are not necessary but shown in the examples for for better illustration.

See: `man gofer` for complete details.

Show the status of goferd

```
$ gofer mgt -h localhost -p 5650 -s
   Plugins:

     <plugin> package
       Classes:
         <class> Package
           methods:
             install(name)
             remove(name)
             update(name)
       Functions:

     <plugin> virt
       Classes:
         <class> Virt
           methods:
             getDomainID(name)
             isAlive(id)
             listDomains()
             shutdown(id)
             start(id)
       Functions:

     <plugin> __builtin__
       Classes:
         <class> Admin
           methods:
             cancel(sn, criteria)
             echo(text)
             hello()
             help()
       Functions:

     <plugin> system
       Classes:
         <class> Shell
           methods:
             run(cmd)
         <class> System
           methods:
             cancel()
             halt(when)
             reboot(when)
         <class> Service
           methods:
             restart()
             start()
             status()
             stop()
         <class> Script
           methods:
             run(user, password, *options)
       Functions:

     <plugin> demo
```

(continues on next page)

(continued from previous page)

```
        Classes:
          <class> Demo
            methods:
              demo()
              echo(something)
              hello()
        Functions:

    Actions:
```

Load a plugin

Plugins can be dynamically loaded using the path to its descriptor.

```
$ gofer mgt -h localhost -p 5650 -s
   Plugins:
   Actions:

$ gofer mgt -h localhost -p 5650 -l /opt/gofer/plugins/package.conf
$ gofer mgt -h localhost -p 5650 -s
   Plugins:

     <plugin> package
       Classes:
         <class> Package
           methods:
             install(name)
             remove(name)
             update(name)
       Functions:

   Actions:
```

Reload a plugin

Plugins can be dynamically reloaded by name or path to its descriptor.

```
$ gofer mgt -h localhost -p 5650 -r package
```

Unload a plugin

Plugins can be dynamically unloaded by name or using the path to its descriptor.

```
$ gofer mgt -h localhost -p 5650 -s
   Plugins:

     <plugin> package
       Classes:
         <class> Package
           methods:
             install(name)
             remove(name)
```

(continues on next page)

(continued from previous page)

```
             update(name)
       Functions:

   Actions:

$ gofer mgt -h localhost -p 5650 -u package
$ gofer mgt -h localhost -p 5650 -s
   Plugins:
   Actions:
```

5.1.2 Remote Method Invocation

The following examples assume a plugin is loaded in *goferd* at the URL of `qpid+amqp://localhost` and subscribed to the *demo* queue. So `-a demo` will be the *address* used. Further, it's assumed that the plugin provides the following API.

```
class Dog(object):

    @remote
    def bark(self, words):
        return 'Yes master.  I will bark because that is what dogs do. "%s"' % words

   @remote
   def wag(self, n):
       for i in range(0, n):
           print 'wag'
       return 'Yes master.  I will wag my tail because that is what dogs do.'
```

Synchronous RMI

```
$ gofer rmi -u qpid+amqp://localhost -a demo -t Dog.bark howdy

  Yes master.  I will bark because that is what dogs do. "howdy"

$ gofer rmi -u qpid+amqp://localhost -a demo -t Dog.wag 3

  Yes master.  I will wag my tail because that is what dogs do.
```

Asynchronous RMI

The following uses the `-r <address` option to specify that the reply is to be sent to the *replies* AMQP address (queue).

```
$ gofer rmi -u qpid+amqp://localhost -a demo -r replies -t Dog.bark howdy

  719d234f-480d-4035-9c2b-b08d17d77f13
```

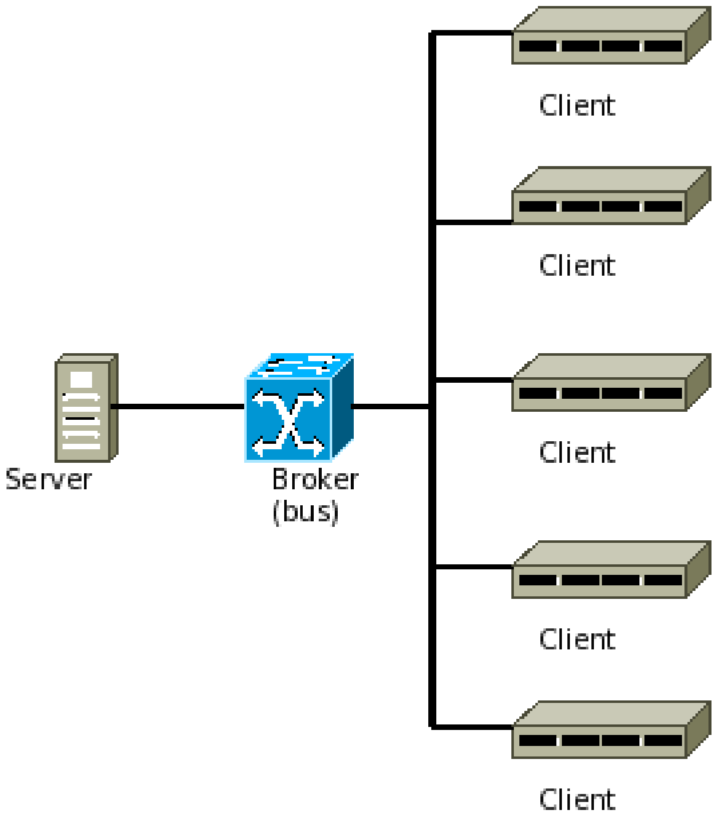
CHAPTER 6

Design

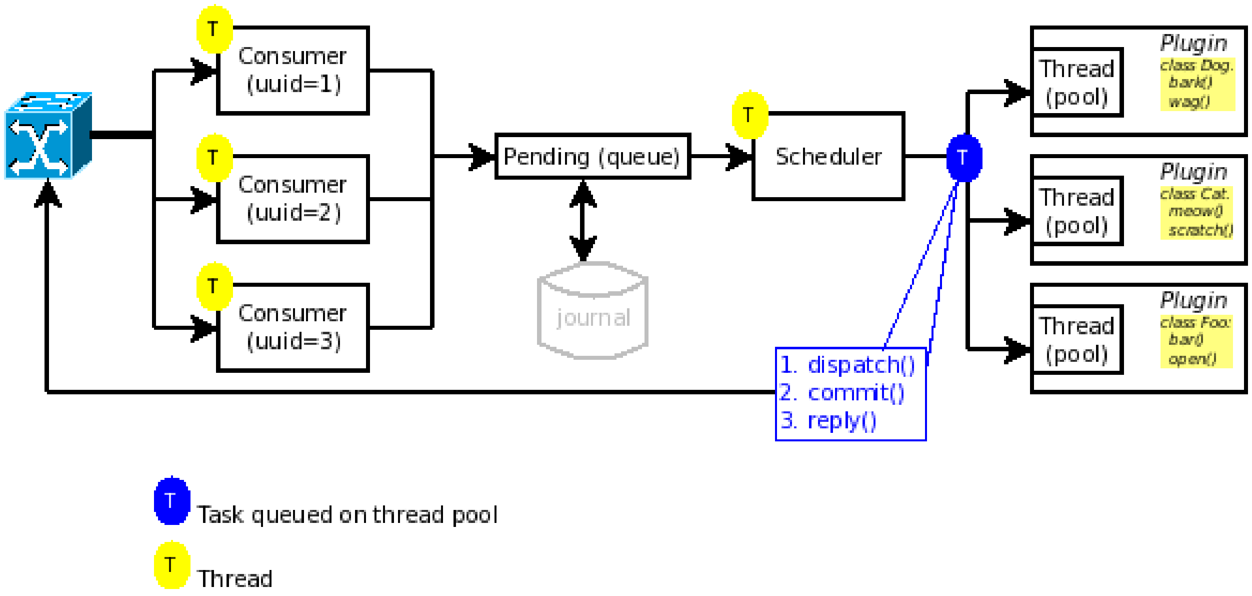
6.1 Approach

The preferred approach is to leverage a Message Bus and possibly a Messaging Framework that uses the message bus for transport. The advantages over home grown and/or point-to-point solutions are as follows:

- Hub and Spoke topology. Each node knows the address of the broken but not each other.
- Key-based routing. Nodes are associated with properties instead of IP addresses.
- Reliable message delivery.
- Message queueing.
- Automatic reconnect behaviour.
- And probably others . . .



Dispatch Architecture:



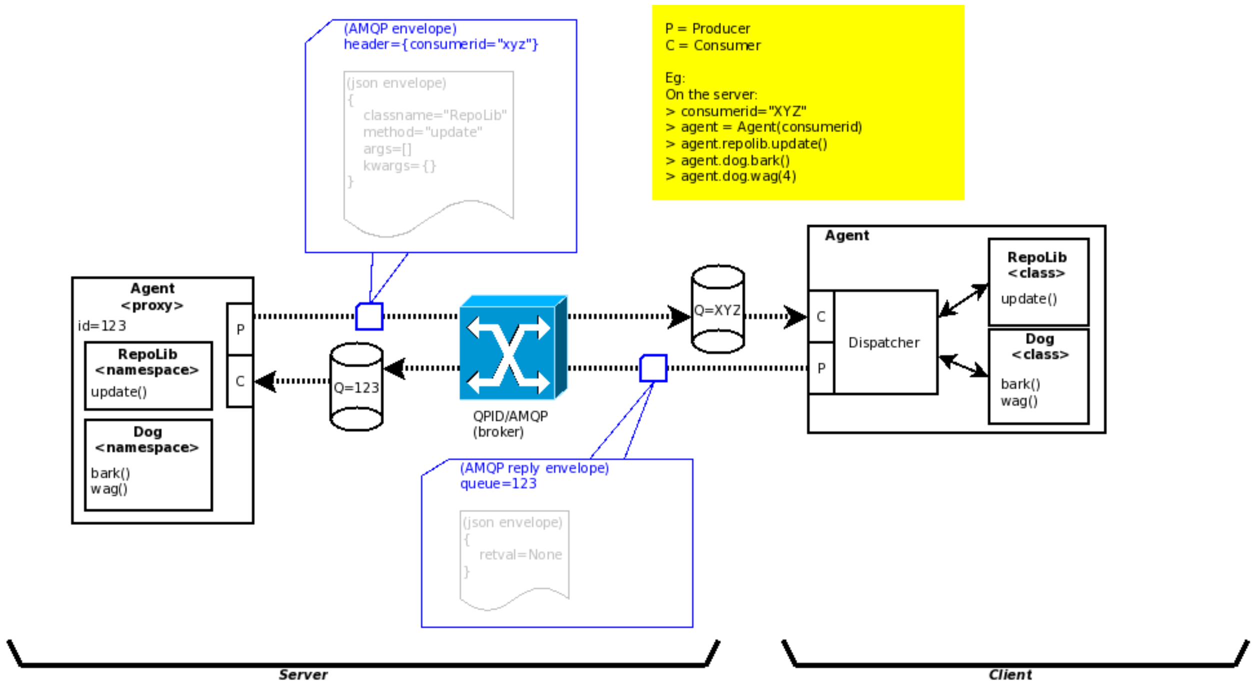
6.2 Messaging

A Messaging Framework provides RMI (Remote Method Invocation) & Event semantics on top of messaging. This gives application developers an easy to use abstraction and hides some of the complexities of exchange and dispatching. Especially in OO applications, invoking a method remotely on an agent without regard for message exchange and routing enhances reliability and productivity.

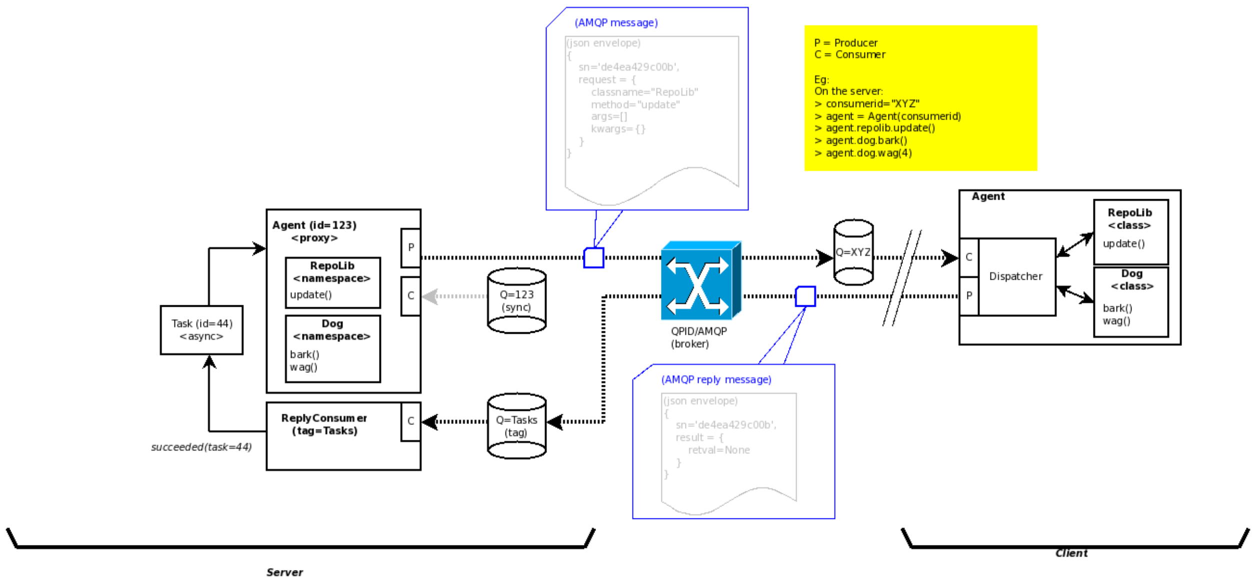
Requirements Summary:

- Key-based routing based on consumer ID.
- Synchronous RMI.
- Asynchronous RMI.
- Fire and Forget
- Callbacks
- Returned values.
- Exception propagation.
- Easy to use.
- Easy to extend classes/method exposed for RMI.
- Events
- Support multiple API versions.

6.2.1 Synchronous RMI:



6.2.2 Asynchronous RMI:



```
(json envelope)
{
   sn='de4ea429c00b',
   request = {
      classname="RepoLib"
      method="update"
      args=[]
      kwargs={}
   }
}
```

```
P = Producer
C = Consumer

Eg:
On the server:
> consumerid="XYZ"
> agent = Agent(consumerid)
> agent.repolib.update()
> agent.dog.bark()
> agent.dog.wag(4)
```

```
(json envelope)
{
   sn='de4ea429c00b',
   result = {
      retval=None
   }
}
```

6.2.3 Messages

The message format is json:

- **Security-Wrapper:**
  - **signature** - A base64 encoded signature.
  - **message** - A json message with stricture of: (Request | Result | Exception)
- **Envelope:**
  - **sn** - Serial Number (uuid).
  - **version** - The API version.
  - **routing** - A tuple containing the amqp (sender, destination).
  - **secret** - The (optional) shared secret used for request authentication. **DEPRECATED** in 2.7.
  - **pam** - The (optional) PAM authentication credentials. **DEPRECATED** in 2.7.
  - **replyto** - The reply amqp address (optional).
  - **one of**
    - **request** - An RMI request. See: Request.
    - **result** - An RMI result. Has value of: (Result | Exception).
    - **status** - An RMI request status report. See: Status.
  - **timestamp** - An ISO-8601 reply timestamp (UTC).
  - **data** - User defined data.
- **Request(Envelope):**
  - **classname** - The target class name.
  - **cntr** - The (optional) remote class constructor arguments. format: ([],{}).

  - **method** - The target instance method name.
  - **args[]** - The list of parameters passed to method
  - **kws{}** - The named keyword arguments passed to method.

- **Status(Envelope):**
  - **status - A request status with value of**
    - *accepted* - Accepted by the agent and queued.
    - *rejected* - Rejected by the agent.
    - *started* - The request has started execution.
    - *progress* - Progress is begin reported. See: Progress.
- **Progress(Status):**
  - **total** - The total number of items to be completed.
  - **completed** - The number of items completed.
  - **details** - Reported details. Can be anything.
- **Result(Envelope):**
  - **retval** - The returned data. Can be anything.
- **Exception(Envelope)**
  - **exval** - The formatted exception (including trace).
  - **xmodule** - The exception module name.
  - **xclass** - The exception class.
  - **xstate** - The exception state. Contains the exception __dict__.
  - **xargs** - The exception *args* attribute when subclass of *Exception*.

Example RMI request message:

```
{
   "sn": "e7e91fb6-611b-4284-a9ed-ac1636b2c709",
   "routing": [
       "cfa806a4-919a-495f-b1dd-3fc11be9a8d0" ,
       "19802a28-a18c-4ae3-ac57-b7a2e78a427a"
   ],
   "replyto": "cfa806a4-919a-495f-b1dd-3fc11be9a8d0",
   "version": "0.2"
   "request": {
       "classname": "Dog",
       "method": "bark"
       "args": ["hello"],
       "kws": {}
   }
}
```

Example reply:

```
{
   "sn": "e7e91fb6-611b-4284-a9ed-ac1636b2c709",
   "version": "0.2",
   "result": {
```

(continues on next page)

(continued from previous page)

```
        "retval": "Yes master.  I will bark because that is what dogs do."
    }
}
```

Example status reply:

```
{
   "origin": "123",
   "status": "accepted",
   "version": "0.2",
   "sn": "985cb165-d291-47de-ab34-ecb20895384e",
   "data": "group 2"
}
```

CHAPTER 7

Messaging Adapters

Each adapter is a standardized integration with an external messaging library. They are a specialized plugin that provides communication with message brokers supported by the library.

7.1 Supported

7.1.1 python-qpid

This adapter uses the `qpid.messaging` library.

- *AMQP* - 0-10
- *package* - gofer.messaging.adapter.qpid
- ***provides***:
    - amqp-0-10
    - qpid.messaging
    - qpid

7.1.2 proton

This adapter uses the `proton` library.

- *AMQP* - 1.0
- *package* - gofer.messaging.adapter.proton
- ***provides***:
    - amqp-1-0
    - proton

   - qpid

7.1.3 python-amqp

This adapter uses the `amqp` library.

- *AMQP* - 0-9-1
- *package* - gofer.messaging.adapter.amqp
- ***provides***:
   - amqp-0-9-1
   - rabbitmq
   - rabbit

CHAPTER 8

Configuration

The gofer agent and plugins are configured using *ini* style configuration files located in `/etc/gofer`.

8.1 Agent Configuration

The agent configuration is specified in: `/etc/gofer/agent.conf` and through files located in `/etc/gofer/conf.d`. During startup, gofer first reads `agent.conf`. Then, reads and merges in values found in the `conf.d` files.

All configuration files support the following sections and properties:

8.1.1 [management]

Defines agent management properties.

- **enabled** - Management is (1=enabled|0=disabled).
- **host** - The host (interface) the manager listens on. Defaults to: *localhost*.
- **port** - The port the manager listens on. Defaults to: *5650*.

8.1.2 [logging]

This section sets logging properties. Currently, the logging level can be set for each gofer package as follows:

```
<package>=<level>
```

The special *root* package may be used to set the logging level for all packages.

Levels (may be lower case):

- CRITICAL
- DEBUG

- ERROR
- FATAL
- INFO
- WARN
- WARNING

Gofer packages:

The list of common packages. Set to DEBUG for more information.

- **root** - The root logger for the python process. Applies to all packages.
- **gofer** - All of gofer.
- **gofer.agent** - The agent which includes plugin management and request dispatching.
- **gofer.messaging** - All AMQP messaging.
- **gofer.rmi** - The RMI request/response protocol and processing.

Examples:

```
[logging]
gofer.agent=DEBUG
gofer.messaging=DEBUG
```

```
[logging]
root=DEBUG
```

8.1.3 [pam]

- **service** - The (optional) service to be used for PAM authentication.

8.2 Plugin Descriptors

Each plugin has a configuration located in `/etc/gofer/plugins`. Plugin descriptors are *ini* style configuration files that require the following sections and properties:

8.2.1 [main]

Defines basic plugin properties.

- **name** - The (optional) plugin name. The basename of the descriptor is used when not specified.
- **plugin** - The (optional) fully qualified path to the module to be loaded from the PYTHON path. When *plugin* is not specified, the plugin is loaded by searching the following directories for a module with the same name as the plugin:
  - /usr/share/gofer/plugins
  - /usr/lib/gofer/plugins
  - /usr/lib64/gofer/plugins
  - /opt/gofer/plugins

- **enabled** - The plugin is (1=enabled|=0disabled).
- **threads** - The (optional) number of threads for the RMI dispatcher.
- **latency** - The (optional) latency (seconds) to be introduced into RMI execution.
- **accept** - Accept forwarding list. Comma ‘,’ separated list of plugin names.
- **forward** - Forwarding list. Comma ‘,’ separated list of plugin names.

The *latency* property is intended to be used to create a cancellation window or provide throttling. Adding *latency*, increases the opportunity for an RMI request to be canceled prior to being started.

8.2.2 [messaging]

- **authenticator** - The (optional) fully qualified path to a message *Authenticator* to be loaded from the PYTHON path.
- **uuid** - The agent identity. This value also specifies the queue name.
- **‘url** - The (optional) broker connection URL. No value indicates the plugin should **not** connect to broker. *format*: `<adapter>+<protocol>://<user>:<password>@<host>:<port>/<virtual-host>`, protocol is one of:
  - **tcp**: non-SSL protocol
  - **amqp**: non-SSL protocol
  - **ssl**: SSL protocol
  - **amqps**: SSL protocol

  The <adapter>, <user>:<password> and /<virtual-host> are optional. See: *Messaging Adapters* for list of supported adapters.

  The <port> is optional and defaults based on the protocol when not specified:
  - (amqp|tcp) port:5672
  - (amqps|ssl) port:5671
- **cacert** - The (optional) SSL CA certificate used to validate the server certificate.
- **clientkey** - The (optional) SSL client private key.
- **clientcert** - The (optional) SSL client certificate. A (PEM) file may contain **both** the private key and certificate.
- **host_validation** - The (optional) flag indicates SSL host validation should be performed. Default to (1) when not specified.
- **heartbeat** - The (optional) AMQP heartbeat in seconds. (default:10).

File extensions just be (.conf|.json).

8.2.3 [model]

- **managed** - The model is manged. Default:2
  - 0: Not managed.
  - 1: The queue is declared on *attach* and bound the the exchange as needed.
  - 2: The queue is declared on *attach* and bound the the exchange as needed and drained and deleted on explicit *detach*.

- **queue** - The (optional) AMQP queue name. This has precedent over uuid. Format: <exchange>/<queue> where *exchange* is optional.
- **expiration** - The (optional) auto-deleted queue expiration (seconds).

8.3 Examples

This example enables messaging and defines the uuid:

```
[main]
enabled=1

[messaging]
url=qpid+amqp://localhost

[model]
queue=123
```

This example enables messaging and does **not** define the uuid. It is expected that the plugin defines an @load decorated method/function that provides the url and queue:

```
[main]
enabled=1
accept=*
```

This example does **not** enable messaging for this plugin. This would be done when the plugin does not need to specify an additional identity. This example also specifies a user defined sections to be used by the plugin:

```
[main]
enabled=1

[messaging]
url=qpid+amqp://localhost

[model]
queue=123

[foobar]
timeout=100
```

However, additional user defined sections and properties are supported and made available to the plugin(s) as follows:

```
from gofer.agent.plugin import Plugin
...
class MyPlugin:
  ...
  def mymethod(self):
      cfg = Plugin.find(__name__).cfg()
      timeout = cfg.foobar.timeout
      ...
```

CHAPTER 9

Options

The RMI options can be passed to the proxy.agent() function and the Agent or Stub constructor. When options passed to either the proxy.agent() or Agent constructor, they apply to all RMI calls unless they are overridden in the Stub constructor.

These options are as follows:

9.1 Summary

reply The asynchronous RMI reply address. Eg: amq.direct/test-queue

trigger Specifies trigger used for RMI calls. (0=auto <default>, 1=manual)

secret The shared secret (security)

ttl The TTL (seconds) for the agent to accept the RMI request.

wait The time (seconds) to wait (block) for a result.

progress A progress callback specified for synchronous RMI. Must have signature: fn(report).

user A user (name), used for PAM authenticated access to remote methods.

password A password, used for PAM authenticated access to remote methods.

authenticator A subclass of pulp.messaging.auth.Authenticator that provides message authentication.

data User defined data associated with the RMI request and is round-tripped.

9.2 Details

9.2.1 reply

The **reply** option specifies the reply address. When specified, it implies all requests are asynchronous and that all replies are sent to the AMQP address.

Example: Assume a reply listener on the topic or queue named: “foo”:

Passed to Agent() and apply to all RMI calls.

```
from gofer.proxy import Agent

agent = Agent(url, uuid, reply='foo')
```

9.2.2 trigger

The **trigger** option specifies the *trigger* used for asynchronous RMI. When the trigger is specified as *manual*, the RMI calls return a *trigger* object instead of the request serial number. Each trigger contains a **sn** (serial number) property that can be used for reply correlation. The trigger is *pulled* by calling the trigger as: *trigger()*.

Trigger values:

- **0** = Automatic *(default)*
- **1** = Manual

Passed to Agent() and apply to all RMI calls.

```
from gofer.proxy import Agent

agent = Agent(uuid, trigger=1)
dog = agent.Dog()
trigger = dog.bark('delayed!')
print trigger.sn       # do something with serial number
trigger()              # pull the trigger
```

9.2.3 secret

The **secret** option is used to provide *shared secret* credentials to each RMI call. This option is only used for agent plugin RMI methods where a *secret* is specified as required.

Examples: Assume the agent has a plugin with methods decorated with a secret=’foobar’

Passed to Agent() and apply to all RMI calls.

```
from gofer.proxy import Agent

agent = Agent(url, uuid, secret='foobar')
```

9.2.4 ttl and wait

The **ttl** option is used to specify the RMI call lifespan. The *ttl* is the time in seconds for the agent to *accept* the request. The message TTL (time-to-live) is set to the *ttl* for both synchronous and asynchronous RMI calls. Additionally, for synchronous RMI, the caller is blocked for the number of seconds specified in the *wait* option. The default *timeout* is 10 seconds and the default *wait* for synchronous RMI is 90 seconds. A *wait=0* indicates that the stub should not block and wait for a reply.

The *timeout* and *wait* can be a string and supports a suffix to define the unit of time. The supported units are as follows:

- **s** : seconds
- **m** : minutes

- **h** : hours
- **d** : days

Passed to Agent() and apply to all RMI calls.

```
from gofer.proxy import Agent

# TTL 5 seconds
agent = Agent(url, uuid, ttl=5)

# TTL 5 minutes
agent = Agent(url, uuid, ttl=5m)

# TTL 30 seconds, wait for 5 seconds
agent = Agent(url, uuid, ttl=30, wait=5)
```

9.2.5 user/password

The **user** and **password** options are used to provide PAM authentication credentials to each RMI call. This option is only used for agent plugin RMI methods decorated with @pam or @user. This is really just a short-hand for the **pam** option.

Examples: Assume the agent has a plugin with methods decorated with @pam(user='root')

Passed to Agent() and apply to all RMI calls.

```
from gofer.proxy import Agent

agent = Agent(url, uuid, user='root', password='xxx')
```

CHAPTER 10

Plugin Decorators

Decorators for remote methods/functions provided by plugins. All decorators may be stacked.

10.1 @action

The *action* decorator is used to designate a function or class method to treated as a recurring action.

Options:

- **interval (one of):**
    - days
    - seconds
    - minutes
    - hours
    - weeks
- required: Yes
- type: int
- default: n/a

10.2 @remote

The *remote* decorator is used to designate a function or class method as being remotely accessible.

Options:

- **secret - used to specify a *shared* secret that must be passed for authorization.**
    - required: No

    - type: str|callable
    - default: None
    - note: **DEPRECATED** in 2.7

- **model** - the RMI execution model (direct|fork). The *fork* model spawns a child process for each method invocation.
  - required: No
  - type: str
  - default: direct
  - note: Added in 2.8

10.3 @direct

The *direct* decorator is used to designate a function to use the *direct* invocation model. With this model, the function is invoked within the goferd process.

Added: 2.8

10.4 @fork

The *fork* decorator is used to designate a function to use the *fork* invocation model. With this model, the function is invoked in a newly spawned child process. This model may be used to insulate the goferd process from unwanted side effects such as memory and filedes leaks, global configuration changes and core dumps.

Added: 2.8

10.5 @pam

The *pam* decorator is used to specify PAM authentication criteria for access to a function or class method. This additional authentication may be used in conjunction with shared secrets.

DEPRECATED in 2.7

Options:

- **user - specified user name.**
    - required: Yes
    - type: str
    - default: n/a
- **service - used to specify the PAM service to be used for the authentication.**
    - required: No
    - type: str
    - default: passwd

10.6 @user

The *user* decorator is used to specify PAM authentication criteria for access to a function or class method. This additional authentication may be used in conjunction with shared secrets. This is an alias for the @pam decorator.

DEPRECATED in 2.7

Options:

- **name - specified user name.**
  - required: Yes
  - type: str
  - default: n/a

CHAPTER 11

Plugin Extension

As gofer plugins are written and shared throughout the open source community, it seems likely that rather than writing your plugin from scratch, it would be useful to be able to extend one that already exists. Plugins have an API for extending their *remote* API with *remote* objects provided by other plugins. Plugin extension can also be specified in the plugin *descriptor* using the *extends* property

11.1 Extending

What can be imported:

- class object
- function object

Using the descriptor property:

Example:

plugin: **dog** can extend the **animals** plugin. Using this method will add all of the *dog* API to the *animals* API.

```
[main]
enabled=1
extends=animals
```

Using the API:

Example:

plugin: **animals**

```
class Dog:
   @remote
   def bark(self):
       pass
```

plugin: **myplugin**

```
from gofer.agent.plugin import Plugin

animals = Plugin.find('animals')
plugin = Plugin.find(__name__)

# just import Dog
plugin += animals['Dog']

# import everything
plugin += animals
```

11.2 Inheritance

Imported class objects may be used as if imported using the standard *import* directive. When used as a *superclass*, the inherited methods will be exposed (@remote) as decorated in the superclass.

Examples:

```
from gofer.agent.plugin import Plugin

# import Dog from animals plugin

animals = Plugin.find('animals')
Dog = animals['Dog']

class Retriever(Dog):
   @remote(secret='wagf')
   def fetch(self):
       pass
```

Results in the following *remote* API:

Retriever:

- **bark()**
    - auth: *None*
- **fetch()**
    - auth: *shared secret*

However, notice that the *auth* on the inherited bark() is different than fetch(). To change this, simply override the method and re-decorate as needed:

```
from gofer.agent.plugin import Plugin

# import Dog from animals plugin

animals = Plugin.find('animals')
Dog = animals['Dog']

class Retriever(Dog):

   @remote(secret='wag')
   def bark(self):
       Dog.bark(self)
```

(continues on next page)

(continued from previous page)

```
    @remote(secret='wag')
    def fetch(self):
        pass
```

11.3 Delegation

In many cases, plugins may choose to leverage imported objects by delegation rather than inheritance.

In this example, the Old McDonald toy does not extend *Dog* but rather delegates the functionality of a dog bark() to an instance of Dog:

```
from gofer.agent.plugin import Plugin

# import Dog from animals plugin

animals = Plugin.find('animals')
Dog = animals['Dog']

# Old McDonald toy
class Toy:
    @remote
    def theDog(self):
        dog = Dog()
        dog.bark()
```

CHAPTER 12

Python Examples

12.1 Server-side

Sample server-side code:

```
from gofer.proxy import Agent

url = 'amqp://localhost'
address = 'test'
agent = Agent(url, address)
```

12.2 Define Agent-side

Sample agent-side code. This module is placed in `/user/share/gofer/plugins/` along with a plugin descriptor in `/etc/gofer/plugins/`

Plugin descriptor: `/etc/gofer/plugins/plugin.conf`

```
[main]
enabled=1

[messaging]
url=amqp://localhost
uuid=test
```

Code: `/user/share/gofer/plugins/plugin.py`

```
from gofer.decorators import remote
from gofer.rmi.model.FORK
from gofer.agent.plugin import Plugin

# (optional) access to the plugin descriptor
```

(continues on next page)

(continued from previous page)

```
# which you can use to define custom sections/properties

plugin = Plugin.find(__name__)

class Dog:

   @remote
   def bark(self, words):
       woof = cfg.dog.bark_noise
       print '%s %s' % (woof, words)
       return 'Yes master.  I will bark because that is what dogs do.'

   @remote(model=FORK)
   def wag(self, n):
       for i in range(0, n):
           print 'wag'
       return 'Yes master.  I will wag my tail because that is what dogs do.'
```

The plugin may be loaded from the PYTHON path by specifying the *plugin* property in descriptor as follows:

```
[main]
enabled=1
plugin=application.agent.plugin.py

[messaging]
url=amqp://localhost
uuid=zoo
```

12.3 Synchronous Invocation

Sample of server code invoking synchronously methods (remotely) on the agent. This is the default behaviour and the timeout is 90 seconds by default.

12.3.1 Python

```
from gofer.proxy import Agent

agent = Agent('amqp://localhost', 'test')

# invoke methods on the agent (remotely)

dog = agent.Dog()
print dog.bark('hello')
print dog.wag(3)
print dog.bark('hello')

# methods that raise exceptions

try:
   print dog.sit()
except Exception, e:
   print repr(e)
```

(continues on next page)

(continued from previous page)

```python
try:
   print dog.not_permitted()
except Exception, e:
   print repr(e)
```

12.3.2 Using the CLI

```
$ gofer rmi -u 'amqp://localhost' -a test -t Dog.bark hello
$ gofer rmi -u 'amqp://localhost' -a test -t Dog.wag 3
$ gofer rmi -u 'amqp://localhost' -a test -t Dog.bark hello
```

12.4 Synchronous Invocation (specify timeout)

Sample of server code invoking synchronously methods (remotely) on the agent with a timeout of 180 seconds.

12.4.1 Python

```python
from gofer.proxy import Agent

amqp://localhost
agent = Agent('amqp://localhost', 'test', wait=180)  # specify timeout

# invoke methods on the agent (remotely)
dog = agent.Dog()
dog.bark('hello')
dog.wag(3)
dog.bark('hello')
```

12.4.2 Using the CLI

```
$ gofer rmi -u 'amqp://localhost' -a test -w 180 -t Dog.bark hello
$ gofer rmi -u 'amqp://localhost' -a test -w 180 -t Dog.wag 3
$ gofer rmi -u 'amqp://localhost' -a test -w 180 -t Dog.bark hello
```

12.5 Asynchronous (fire & forget) Invocation

Sample of server code invoking synchronously methods (remotely) on the agent. This works the same for asynchronous *fire-and-forget* where not reply is wanted. Asynchronous invocation returns the serial number of the request.

12.5.1 Python

```python
from gofer.proxy import Agent

# create an agent where user data = 'task_id'
agent = Agent('amqp://localhost', 'test', wait=0)

# invoke methods on the agent (remotely)
dog = agent.Dog()
dog.bark('hello')
dog.wag(3)
print dog.bark('hello')
   'e688f50b-3108-43dd-9a57-813f434749a8'

# methods that raise exceptions
try:
   print dog.sit()
except Exception, e:
   print repr(e)

try:
   print dog.not_permitted()
except Exception, e:
   print repr(e)
```

12.5.2 Using the CLI

```
$ gofer rmi -u 'amqp://localhost' -a test -w 0 -t Dog.bark hello
e688f50b-3108-43dd-9a57-813f434749a8
```

12.6 Asynchronous (callback) Invocation

Sample of server code invoking asynchronously methods (remotely) on the agent. The is the *callback* form of asynchronous invocation. This example uses a *Listener* class. But, the *listener* can also be any *callable*. Asynchronous invocation returns the serial number of the request to be used by the caller to further correlate request & response.

12.6.1 Python

```python
from gofer.proxy import Agent
from gofer.messaging.async import ReplyConsumer

# specify a reply address to be used for asynchronous responses.

reply_to = 'tasks'

# create my listener class

class Listener:
   """
   An asynchronous operation callback listener.
   """

   def succeeded(self, reply):
```

(continues on next page)

(continued from previous page)

```
        """
        Async request succeeded.
        :param reply: The reply data.
        :type reply: Succeeded.
        """
        pass

    def failed(self, reply):
        """
        Async request failed (raised an exception).
        :param reply: The reply data.
        :type reply: Failed.
        """
        pass

    def accepted(self, reply):
        """
        Async request has been accepted.
        :param reply: The request.
        :type reply: Accepted.
        """
        pass

    def rejected(self, reply):
        """
        Async request has been rejected.
        :param reply: The request.
        :type reply: Accepted.
        """
        pass

    def started(self, reply):
        """
        Async request has started.
        :param reply: The request.
        :type reply: Started.
        """
        pass

    def progress(self, reply):
        """
        Async progress report.
        :param reply: The request.
        :type reply: Progress.
        """
        pass


# create my reply consumer using the reply to and my listener

reader = ReplyConsumer(reply_to)
reader.start(Listener())

# create an agent where user data is {'task_id': 1234} and
# setup for asynchronous invocation with my reply address.

agent = Agent('amqp://localhost', 'test', reply=reply_to)
```

(continues on next page)

(continued from previous page)

```
# invoke methods on the agent (remotely)
dog = agent.Dog()
dog.bark('hello')
dog.wag(3)
print dog.bark('hello')
   'e688f50b-3108-43dd-9a57-813f434749a8'
```

Same asynchronous example except specify a *callable* as the listener. Also, it uses the *throw()* method on reply.

```
# specify a reply address to be used for responses.

reply_to = 'tasks'

# create my listener

def callback(reply):
   try:
       reply.throw()
       ...
       print reply.retval # succeeded, do something with return value.
       ...
   except Exception, ex:
       # handle general exception
       pass

# create my reply consumer using the reply address and my callback

reader = ReplyConsumer(reply_to)
reader.start(callback)
...
```

12.6.2 Using the CLI

Note: `-r tasks`

```
$ gofer rmi -u 'amqp://localhost' -a test -w 0 -r tasks -t Dog.bark hello
e688f50b-3108-43dd-9a57-813f434749a8
```

12.7 Class Constructor Arguments

Classes defined in the agent can have constructor arguments. Though, remember, an instance is constructed for each request so remote objects are stateless. The *stub* provides for passing __init__() arguments by calling the *stub*.

Examples:

In the plugin:

```
class Dog:

 def __init__(self, name, age=1):
   self.name = name
   self.age = age
```

(continues on next page)

(continued from previous page)

```
  @remote
  def bark(self):
    pass

  @remote
  def wag():
    pass
```

Calling:

```
...
dog = agent.Dog()       # stub constructor, pass gofer options here.
dog('rover', age=10)    # constructor arguments set here.
dog.bark('hello')
dog.wag()

# change the constructor arguments and call something else.

dog('max', age=5)    # changing constructor arguments.
dog.bark('howdy')
```

Subsequent calls simply update the constructor arguments.

This:

```
dog('rover', age=10)
```

equals this (in the agent):

```
dog = Dog('rover', age=10)
```

12.8 Security

When *remote* methods or functions are decorated to require a shared secret for request authentication, it must be passed as an option.

Example:

```
from gofer.proxy import Agent
from gofer.messaging.dispatcher import NotAuthorized

agent = Agent('amqp://localhost', 'test', secret='mycathas9lives')
# invoke methods on the agent (remotely)
dog = agent.Dog()
try:
   dog.bark('secure hello')
except NotAuthorized:
   log.error('wrong secret')
```

12.9 Progress Reporting

In gofer 0.72+ remote method progress can be reported by plugins. In the case of synchronous RMI, the caller can specify a *callback* for progress reporting by specifying the *progress* option. The *callback* must take a single (dict) parameter (report).

The report has the following keys:

- **sn** - *serial number*
- **any** - *user data*
- **total** - *the number total units*
- **completed** - *the number of completed units*
- **details** - *arbitrary details*

For asynchronous RMI, the *listener* is called with progress reports.

Examples:

```
from gofer.proxy import Agent

def progress_reported(report)
 pass

agent = Agent('amqp://localhost', 'test')
dog = agent.Dog(progress=progress_reported)
dog.bark('howdy')
```

On the agent, plugins report progress from with a method by using the *Progress* object defined within the current call *Context*.

Example:

```
from gofer.agent.rmi import Context
from gofer.decorators import remote

class MyClass:

   @remote
   def foo(self):
       """
       Do something reports progress
       """
       total = 10
       # get the call context
       ctx = Context.current()
       ctx.progress.total = total
       # demo reporting progress for 10 units
       for n in range(0, total):
           ctx.progress.completed += 1
           sleep(1)

   @remote
   def bar(self):
       """
       Do something reports progress with details.
       """
```

(continues on next page)

(continued from previous page)

```
        total = 10
        # get the call context
        ctx = Context.current()
        ctx.progress.total = total
        # demo reporting progress for 10 units
        for n in range(0, total):
            ctx.progress.completed += 1
            ctx.progress.details='for: %d' % n)
            sleep(1)
```

12.10 Testing

12.10.1 Logs

After adding/updating classes or methods in myplugin.py, you'll want to test them. First, ensure the plugin is still loading properly. The easiest way to do this is by examining the gofer log file at: `/var/log/gofer/agent`. At start up, you should see something like:

```
2010-11-08 08:49:04,491 [WARNING][MainThread] __mangled() @ plugin.py:122 - "pulp"␣
↪found in python-path
2010-11-08 08:49:04,503 [INFO][MainThread] __mangled() @ plugin.py:123 - "pulp"␣
↪mangled to avoid collisions
2010-11-08 08:49:04,909 [INFO][MainThread] __import() @ plugin.py:103 - plugin "pulp",
↪ imported as: "pulp_plugin"
```

Either the gofer log or the pulp client.log may be examined to verify that *Actions* are running as expected.

12.10.2 Interactive Shell

Testing added/updated *remote methods*, can be done easily using an interactive python (shell). Be sure your changes to the pulp plugin have been picked up by *Gofer* by **restarting goferd**. Let's say you added a new class named "Foo" that has a remote method named . . . you guessed it: "bar".

You can test your new stuff as follows:

```
[jortel@localhost pulp]$ python
Python 2.6.2 (r262:71600, Jun  4 2010, 18:28:04)
[GCC 4.4.3 20100127 (Red Hat 4.4.3-4)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> from gofer.proxy import Agent
>>> uuid = <your consumer ID>
>>> agent = Agent('amqp://localhost', uuid)
>>> foo = agent.Foo()
>>> print foo.bar()
```

Or, using the proxy module API:

```
[jortel@localhost pulp]$ python
Python 2.6.2 (r262:71600, Jun  4 2010, 18:28:04)
[GCC 4.4.3 20100127 (Red Hat 4.4.3-4)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> from gofer import proxy
```

(continues on next page)

(continued from previous page)

```
>>> uuid = <your consumer ID>
>>> agent = proxy.agent('amqp://localhost', uuid)
>>> foo = agent.Foo()
>>> print foo.bar()
```

12.10.3 Admin.help()

Another useful tool, it invoke *Admin.help()* from within interactive python as follows:

```
[jortel@localhost pulp]$ python
Python 2.6.2 (r262:71600, Jun  4 2010, 18:28:04)
[GCC 4.4.3 20100127 (Red Hat 4.4.3-4)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> from pulp.server.agent import Agent
>>> uuid = <your consumer ID>
>>> agent = Agent('amqp://localhost', uuid)
>>> admin = agent.Admin()
>>> print admin.help()

Plugins:
  demo
  pulp [pulp_admin]
Actions:
  demo.TestAction 0:10:00
Methods:
  custom.Dog.bark()
  custom.Dog.wag()
  demo.Admin.hello()
  demo.Admin.help()
  demo.Shell.run()
Functions:
  demo.echo()
>>>
```

12.10.4 Test Main

The `test/main.py` module provides a good testing entry point that does not require the process owner to be root.

12.10.5 Mocks

The gofer *mock* feature provides better testability. Essentially, it allows uses to test the server-side code that uses the gofer proxy. Instead of calling through to the remote agent, RMI calls can be mocked-up.

Added 0.33.

The *mock* module provides an API for registering custom *stub* mocks.

Items that can be registered with *mock*.register():

- instance (object)
- class
- module

Example:

```
from gofer.messaging import mock
mock.install()
from gofer.proxy import Agent

agent = Agent('xyz')

# define mock impl for testing
class Dog:
   def bark(self, msg):
       return 'mock Dog, called with: [%s]' % msg

# register our mock class
mock.register(Dog=Dog)

# call bark()

dog = agent.Dog()

print dog.bark('hello')
  'mock Dog, called with: [hello]'

print dog.bark('world')
  'mock Dog, called with: [world]'

#
# now, let look at the call history
#

h =  dog.bark.history()
print h
 '[("hello",),{}), ("world",),{})]'

# get last call
last = h[-1]

# look at the passed args
print last.args[0]
 'world'

# look at the keyword args
print last.kwargs
 '{}'
```

It's very important to note the difference between registering a class (as a stub) and an instance (as a stub). In short, nstances are shared across all *mock* agents and classes are associated to the instance of the *mock* agent that created them. That way, call history is scoped to *mock* agent as well.

In some cases, it's useful to have a stub method raise an exception. Here's how it's done:

```
from gofer.messaging import mock
mock.install()
from gofer.proxy import Agent

agent = Agent('amqp://localhost', 'xyz')

# define mock impl for testing
```

(continues on next page)

(continued from previous page)

```
class Dog:

   def bark(self, msg):
       return 'mock Dog, called with: [%s]' % msg

# register our mock class
mock.register(Dog=Dog)

dog = agent.Dog()

# call bark() normally
print dog.bark('hello')

# now, let's have it raise an exception

dog.bark.push(Exception('no more barking'))
try:
   dog.bark('hello')
except Exception, e:
  print e
  '"no more barking'"
```

CHAPTER 13

QPID Configuration

This page describes qpidd SSL configuration.

13.1 LINKS

Here are some helpful links:

- http://www.mail-archive.com/qpid-commits@incubator.apache.org/msg06212.html
- http://www.mozilla.org/projects/security/pki/nss/tools/certutil.html
- http://rajith.2rlabs.com/2010/03/01/apache-qpid-securing-connections-with-ssl/

13.2 RPMS

The SSL configuration for QPID is based on NSS. So, the *certutil* tool needs to be installed to manage the NSS certificate databases. Also, the qpidd-ssl package needs to be installed to enable SSL on the qpid broker.

Fedora:

- nss-tools - contains certutil used to manage NSS database for SSL.
- qpidd-ssl - contains *ssl.so* which enables SSL.

13.3 Certificates

The easiest way to create the NSS DB and SSL certificates needed, is to run the `nss-db-gen` in <gofer.git>/tools.

```
[jortel@~]$ cd git/gofer/tools
[jortel@localhost tools]$ nss-db-gen
bash: nss-db-gen: command not found
[jortel@localhost tools]$ ./nss-db-gen

Working in: /tmp/tmp20823


Please specify a directory into which the created NSS database
and associated certificates will be installed.

Enter a directory [/tmp/redhat/qpid]:
/tmp/redhat/qpid

Enter NSS database password:

Please specify a CA.  Generated if not specified.

Enter a path:

Password file created.

Database created.

Creating CA certificate:


Generating key.  This may take a few moments...

CA created

Creating BROKER certificate:


Generating key.  This may take a few moments...

Broker certificate created.

Creating CLIENT certificate:


Generating key.  This may take a few moments...

Client certificate created.
Enter Password or Pin for "NSS Certificate DB":
Enter Password or Pin for "NSS Certificate DB":
Enter password for PKCS12 file:
Re-enter password:
pk12util: PKCS12 EXPORT SUCCESSFUL
Enter Import Password:
MAC verified OK
Client key & certificate exported

Artifacts copied to: /tmp/redhat/qpid.

Please update /etc/qpidd.conf as follows:
```

(continues on next page)

(continued from previous page)

```
....
auth=no
....
# SSL
require-encryption=yes
ssl-require-client-authentication=yes
ssl-cert-db=/tmp/redhat/qpid/nss
ssl-cert-password-file=/tmp/redhat/qpid/nss/password
ssl-cert-name=broker
ssl-port=5674
...


Please configure gofer as follows:

...
[messaging]
url=ssl://<host>:5674
cacert=/tmp/redhat/qpid/ca.crt
clientcert=/tmp/redhat/qpid/client.crt
```

Files generated by the script:

```
redhat/
redhat/qpid
redhat/qpid/broker.crt
redhat/qpid/client.crt
redhat/qpid/nss
redhat/qpid/nss/secmod.db
redhat/qpid/nss/password
redhat/qpid/nss/key3.db
redhat/qpid/nss/cert8.db
redhat/qpid/ca.crt
```

Notes:

- The “Enter a directory [/tmp/redhat/qpid]:” can be defined as any directory.
- The passwords can be anything.

13.4 Configuration

13.4.1 QPID

Edit `/etc/qpidd.conf`:

auth Require authentication. (value: no)

require-encryption Require all connections to use SSL. (value: yes)

ssl-require-client-authentication Require client SSL certificates for all SSL connections. (value: yes)

ssl-cert-db The fully qualified path to the NSS DB. (example: /tmp/redhat/qpid/nss)

ssl-cert-password-file The fully qualified path to the password file used to access the NSS DB. (example: /tmp/redhat/qpid/nss/password)

ssl-cert-name The *name* of the certificate in the NSS DB to be used by the qpid broker. (example: broker)

ssl-port The port to be use for SSL connections. (example: 5671)

13.4.2 Gofer Agent

Edit `/etc/gofer/plugins/<yourplugin>.conf` and under the [messaging] section:

url The URL to the qpid broker. Protocol choices: tcp=plain, ssl=SSL. (example: ssl://<host>:5671)

cacert The fully qualified path to the CA certificate used to validate the broker. (example: /tmp/redhat/qpid/ca.crt)

clientcert The fully qualified path a file containing both the *client* private key and certificate. (example: /tmp/redhat/qpid/client.crt)

CHAPTER 14

Indices and tables

- genindex
- modindex
- search