negotiator

Release 0.8.4

April 07, 2016

Contents

The Python packages negotiator-host, negotiator-guest and negotiator-common together implement a scriptable KVM/QEMU guest agent infrastructure in Python. This infrastructure supports realtime bidirectional communication between Linux hosts and guests which allows the hosts and guests to invoke user defined commands on 'the other side'.

Because the user defines the commands that hosts and guests can execute, the user controls the amount of influence that hosts and guests have over each other (there are several built-in commands, these are all read only).

Contents

CHAPTER 1

Status

Some points to consider:

- The Negotiator project does what I expect from it: realtime bidirectional communication between Linux based KVM/QEMU hosts and guests.
- The project doesn't have an automated test suite, although its functionality has been extensively tested during development and is being used in a production environment on more than 100 virtual machines (for non-critical tasks).
- The project has not been peer reviewed with regards to security. My primary use case is KVM/QEMU hosts and guests that trust each other to some extent (think private clouds, not shared hosting :-).

CHAPTER 2

Installation

The *negotiator* packages and their dependencies are compatible with Python 2.6 and newer and are all pure Python. This means you don't need a compiler toolchain to install the *negotiator* packages. This is a design decision and so won't be changed.

2.1 On KVM/QEMU hosts

Here's how to install the negotiator-host package on your host(s):

```
$ sudo pip install negotiator-host
```

If you prefer you can install the Python package in a virtual environment:

```
$ sudo apt-get install --yes python-virtualenv
$ virtualenv /tmp/negotiator-host
$ source /tmp/negotiator-host/bin/activate
$ pip install negotiator-host
```

After installation the `negotiator-host` program is available. The usage message will help you get started, try the `--help` option. Now you need to find a way to run the `negotiator-host` command as a daemon. I have good experiences with supervisord, here's how to set that up:

```
$ sudo apt-get install --yes supervisor
$ sudo tee /etc/supervisor/conf.d/negotiator-host.conf >/dev/null << EOF
[program:negotiator-host]
command = /usr/local/bin/negotiator-host --daemon
autostart = True
redirect_stderr = True
stdout_logfile = /var/log/negotiator-host.log
EOF
$ sudo supervisorctl update negotiator-host
```

2.2 On KVM/QEMU guests

Install the negotiator-guest package on your guest(s):

```
$ sudo pip install negotiator-guest
```

If you prefer you can install the Python package in a virtual environment:

```
$ sudo apt-get install --yes python-virtualenv
$ virtualenv /tmp/negotiator-guest
$ source /tmp/negotiator-guest/bin/activate
$ pip install negotiator-guest
```

After installation you need to find a way to run the `negotiator-guest` command as a daemon. I have good experiences with supervisord, here's how to set that up:

```
$ sudo apt-get install --yes supervisor
$ sudo tee /etc/supervisor/conf.d/negotiator-guest.conf >/dev/null << EOF
[program:negotiator-guest]
command = /usr/local/bin/negotiator-guest --daemon
autostart = True
redirect_stderr = True
stdout_logfile = /var/log/negotiator-guest.log
EOF
$ sudo supervisorctl update negotiator-guest
```

CHAPTER 3

Getting started

If the instructions below are not enough to get you started, take a look at the *Debugging* section below for hints about what to do when things don't work as expected.

1. First you have to add two virtual devices to your QEMU guest. You can do so by editing the guest's XML definition file. On Ubuntu Linux KVM/QEMU hosts these files are found in the directory `/etc/libvirt/qemu`. Open the file in your favorite text editor (Vim? :-) and add the the following XML snippet inside the `<devices>` section:

```
<channel type='unix'>
   <source mode='bind' path='/var/lib/libvirt/qemu/channel/target/GUEST_NAME.negotiator-host-to-
   <target type='virtio' name='negotiator-host-to-guest.0' />
</channel>

<channel type='unix'>
   <source mode='bind' path='/var/lib/libvirt/qemu/channel/target/GUEST_NAME.negotiator-guest-to
   <target type='virtio' name='negotiator-guest-to-host.0' />
</channel>
```

   Replace `GUEST_NAME` with the name of your guest in both places. If you use libvirt 1.0.6 or newer (you can check with `virsh --version`) you can omit the `path='...'` attribute because libvirt will fill it in automatically when it reloads the guest's XML definition file (in step 2).

2. After adding the configuration snippet you have to activate it:

```
$ sudo virsh define /etc/libvirt/qemu/GUEST_NAME.xml
```

3. Now you need to shut down the guest and then start it again:

```
$ sudo virsh shutdown --mode acpi GUEST_NAME
$ sudo virsh start GUEST_NAME
```

   Note that just rebooting the guest will not add the new virtual devices, you have to actually stop the guest and then start it again!

4. Now go and create some scripts in `/usr/lib/negotiator/commands` and try to execute them from the other side! Once you start writing your own commands it's useful to know that commands on the KVM/QEMU host side have access to some environment variables.

CHAPTER 4

Debugging

This section contains hints about what to do when things don't work as expected.

4.1 Broken channels on KVM/QEMU hosts

Whether you want to get the official QEMU guest agent or the Negotiator project running, you will need a working bidirectional channel. I'm testing Negotiator on an Ubuntu 14.04 KVM/QEMU host and I needed several changes to get things working properly:

```
$ CHANNELS_DIRECTORY=/var/lib/libvirt/qemu/channel/target
$ sudo mkdir -p $CHANNELS_DIRECTORY
$ sudo chown libvirt-qemu:kvm $CHANNELS_DIRECTORY
```

The above should be done by the KVM/QEMU system packages if you ask me, but anyway. On top of this if you are running Ubuntu with AppArmor enabled (the default) you may need to apply the following patch:

```
$ diff -u /etc/apparmor.d/abstractions/libvirt-qemu.orig /etc/apparmor.d/abstractions/libvirt-qemu
--- /etc/apparmor.d/abstractions/libvirt-qemu.orig      2015-09-19 12:46:54.316593334 +0200
+++ /etc/apparmor.d/abstractions/libvirt-qemu   2015-09-24 14:43:43.642064576 +0200
@@ -49,6 +49,9 @@
   /run/shm/ r,
   owner /run/shm/spice.* rw,

+  # Local modification to enable the QEMU guest agent.
+  owner /var/lib/libvirt/qemu/channel/target/* rw,
+
   # 'kill' is not required for sound and is a security risk. Do not enable
   # unless you absolutely need it.
   deny capability kill,
```

Again this should just be part of the KVM/QEMU system packages, but whatever. The Negotiator project is playing with new-ish functionality so I pretty much know to expect sharp edges :-)

4.2 Character device detection fails

When the `negotiator-guest` program fails to detect the correct character devices it will complain loudly and point you here. Here are some of things I've run into that can cause this:

- The virtual channel(s) have not been correctly configured or the correct configuration hasn't been applied yet. Please carefully follow the instructions in the *Getting started* section above.
- The kernel module `virtio_console` is not loaded because it is not available in your kernel. You can check by using the `lsmod` command. If the module is not loaded you'll need to install and boot to a kernel that does have the module.

CHAPTER 5

Why another guest agent?

The QEMU project provides an official guest agent and this agent is very useful to increase integration between QEMU hosts and guests. However the official QEMU guest agent has two notable shortcomings (for me at least):

Extensibility The official QEMU guest agent has some generic mechanisms like being able to write files inside guests, but this is a far cry from a generic, extensible architecture. Ideally given the host and guest's permission we should be able to transfer arbitrary data and execute user defined logic on both sides.

Platform support Despite considerable effort I haven't been able to get a recent version of the QEMU guest agent running on older Linux distributions (e.g. Ubuntu Linux 10.04). Older versions of the guest agent can be succesfully compiled for such distributions but don't support the features I require. By creating my own guest agent I have more control over platform support (given the primitives required for communication).

Note that my project in no way tries to replace the official QEMU guest agent. For example I have no intention of implementing freezing and thawing of file systems because the official agent already does that just fine :-). In other words the two projects share a lot of ideas but have very different goals.

CHAPTER 6

How does it work?

The scriptable guest agent infrastructure uses the same mechanism that the official QEMU guest agent does:

- Inside the guest special character devices are created that allow reading and writing. These character devices are `/dev/vport[0-9]p[0-9]`.
- On the host UNIX domain sockets are created that are connected to the character devices inside the guest. On Ubuntu Linux KVM/QEMU hosts, these UNIX domain sockets are created in the directory `/var/lib/libvirt/qemu/channel/target`.

CHAPTER 7

Contact

The latest version of *negotiator* is available on PyPI and GitHub. You can find the documentation on Read The Docs. For bug reports please create an issue on GitHub. If you have questions, suggestions, etc. feel free to send me an e-mail at peter@peterodding.com.

CHAPTER 8

License

CHAPTER 9

Function reference

The following documentation is based on the source code of version 0.8.4 of the negotiator-host package.

9.1 `negotiator_host` - Channel for communication with guests

This module implements the *GuestChannel* class which provides the host side of the channel between QEMU hosts and guests. Channel objects can be used to query and command running guests.

class negotiator_host.**AutomaticGuestChannel**(*guest_name*, *unix_socket*)
Thin wrapper for *GuestChannel* that puts it in a separate process.

Uses multiprocessing.Process to isolate guest channels in separate processes.

__init__(*guest_name*, *unix_socket*)
Initialize a *GuestChannel* in a separate process.

Parameters

- **guest_name** – The name of the guest to connect to (a string).
- **unix_socket** – The absolute pathname of the UNIX socket that we should connect to (a string).

run()
Start the main loop of the common negotiator interface.

class negotiator_host.**GuestChannel**(*guest_name*, *unix_socket=None*)
The host side of the channel connecting KVM/QEMU hosts and guests.

See also *AutomaticGuestChannel* which wraps *GuestChannel* and puts it in its own process.

__init__(*guest_name*, *unix_socket=None*)
Initialize a negotiator host agent.

Parameters

- **guest_name** – The name of the guest to connect to (a string).
- **unix_socket** – The absolute pathname of the UNIX socket that we should connect to (a string, optional).

prepare_environment()
Prepare environment variables for command execution on KVM/QEMU hosts.

The following environment variables are currently exposed to commands:

$NEGOTIATOR_GUEST The name of the KVM/QEMU guest that invoked the command.

exception negotiator_host.**GuestChannelInitializationError**
Exception raised by *GuestChannel* when socket initialization fails.

class negotiator_host.**HostDaemon**(*channel_directory='/var/lib/libvirt/qemu/channel/target'*)
The host daemon automatically manages a group of processes that handle "guest to host" calls.

__init__(*channel_directory='/var/lib/libvirt/qemu/channel/target'*)
Initialize the host daemon.

Parameters **channel_directory** – The pathname of the directory containing UNIX sockets connected to guests (a string).

enter_main_loop()
Create and maintain active channels for all running guests.

update_active_channels()
Automatically spawn subprocesses (workers) to maintain connections to all guests.

negotiator_host.**find_available_channels**(*directory*, *name*)
Find available channels by checking for available UNIX sockets.

This uses *find_running_guests()* to ignore UNIX sockets that are not connected to a running guest (since these sockets are useless until they become connected to a running guest).

Parameters

- **directory** – The pathname of the directory to search (a string).
- **name** – The name of the channel to search for (a string).

Returns A dictionary with KVM/QEMU guest names (strings) as keys and pathnames of UNIX sockets as values.

negotiator_host.**find_running_guests**()
Find the names of the guests running on the current host.

This function parses the output of the `virsh list` command instead of using the libvirt API because of two reasons:

1. I'm under the impression that the libvirt API is still very much in flux and large changes are still being made, so it's not the most stable foundation for Negotiator to find running guests.
2. The Python libvirt API needs to match the version of the libvirt API on the host system and there is AFAIK no obvious way to express this in the `setup.py` script of Negotiator.

Returns A generator of strings with guest names.

9.2 negotiator_guest - The guest agent daemon

This module implements the guest agent, the Python daemon process that's always running inside KVM/QEMU guests.

class negotiator_guest.**GuestAgent**(*character_device*)
Implementation of the daemon running inside KVM/QEMU guests.

__init__(*character_device*)
Initialize a negotiator guest agent.

Parameters **character_device** – The absolute pathname of the character device that we should use to connect to the host (a string).

raw_readline()
Read a newline terminated string from the remote side.

This method overrides the *raw_readline()* method of the *NegotiatorInterface()* class to implement blocking reads based on os.O_ASYNC and signal.SIGIO (see also *WaitForRead*).

Returns The data read from the remote side (a string).

class negotiator_guest.**WaitForRead**(*group=None*, *target=None*, *name=None*, *args=()*, *kwargs={}*)
Used by *GuestAgent.raw_readline()* to implement blocking reads.

run()
Endless loop that waits for one or more SIGIO signals to arrive.

signal_handler(*signal_number*, *frame*)
Signal handler for SIGIO signals that immediately exits the process.

negotiator_guest.**find_character_device**(*port_name*)
Find the character device for the given port name.

Parameters **port_name** – The name of the virtio port (a string).

Returns The absolute pathname of a character device (a string).

Raises Exception when the character device cannot be found.

9.3 negotiator_common - Common shared functionality

This Python module contains the functionality that is shared between the negotiator-host and negotiator-guest packages. By moving all of the shared functionality to a separate Python package and using Python package dependencies to pull in the negotiator-common package we stimulate code reuse while avoiding code duplication.

class negotiator_common.**NegotiatorInterface**(*handle*, *label*)
Common logic shared between the host/guest components.

This class defines the protocol that's used to communicate between the Python programs running on the hosts and guests.

__init__(*handle*, *label*)
Initialize a negotiator host or guest agent.

Parameters

- **handle** – A file like object connected to the other side.
- **label** – A string describing the file like object (used in logging).

This constructor is intended to be called by sub classes to provide the base class with the context it needs to set up bidirectional communication between the host and guest agents.

call_remote_method(*method*, **args*, ***kw*)
Call a method on the remote object.

Parameters

- **method** – The name of the method to call (a string).
- **args** – The positional arguments for the method.
- **kw** – The keyword arguments for the method.

Returns The return value of the remote method.

enter_main_loop()
Wait for requests from the other side.

The communication protocol for remote procedure calls is as follows:

- Every request is a dictionary containing at least a `command` key with a string value (the name of the method to invoke).
- The value of the optional `arguments` key gives a list of positional arguments to pass to the method.
- The value of the optional `keyword-arguments` key gives a dictionary of keyword arguments to pass to the method.

Responses are structured as follows:

- Every response is a dictionary containing at least a `success` key with a boolean value.
- If `success=True` the key `result` gives the return value of the method.
- If `success=False` the key `error` gives a string explaining what went wrong.

Raises *ProtocolError* when the remote side violates the defined protocol.

execute(**command*, ***options*)
Execute a user defined or built-in command.

Parameters

- **command** – The command name and any arguments (one or more strings).
- **input** – The input to feed to the command on its standard input stream (a string or `None`).

Returns The output of the command (a string) or `None` if the command exited with a nonzero exit code.

list_commands()
Find the names of the user defined commands.

Returns A list of executable names (strings).

prepare_environment()
Prepare environment variables for command execution.

This method can be overridden by sub classes to prepare environment variables for external command execution.

raw_read(*num_bytes*)
Read the given number of bytes from the remote side.

Parameters **num_bytes** – The number of bytes to read (an integer).

Returns The data read from the remote side (a string).

raw_readline()
Read a newline terminated string from the remote side.

Returns The data read from the remote side (a string).

raw_write(*data*)
Write a string of data to the remote side.

Parameters **data** – The data to write tot the remote side (a string).

read()
Wait for a JSON encoded message from the remote side.

The basic communication protocol is really simple:

1. First an ASCII encoded integer number is received, terminated by a newline.
2. Second the number of bytes given by step 1 is read and interpreted as a JSON encoded value. This step is not terminated by a newline.

That's it :-).

Returns The JSON value decoded to a Python value.

Raises *ProtocolError* when the remote side violates the defined protocol.

write(*value*)
Send a Python value to the other side.

Parameters **value** – Any Python value that can be encoded as JSON.

exception negotiator_common.**ProtocolError**
Exception that is raised when the communication protocol is violated.

exception negotiator_common.**RemoteMethodFailed**
Exception that is raised when a remote method call failed.

n

Symbols

A

C

E

F

G

H

L

N

P

R

S

U

W