High Availability for OPNFV

Support for HA Guest APIs

Server Group Messaging

Release D

November 2016

[Specification of Guest Server Group State-Change-Notification & Broadcast Messaging Module – Rel. D.]

High Availability for OPNFV

Table of Contents

High Availability for OPNFV

Introduction

This module defines a Host-to-Guest Server Group Messaging API and the OpenStack-Host Design to provide a simple low-bandwidth datagram messaging and notification service for servers that are part of the same server group. This messaging channel is available regardless of whether IP networking is functional within the server, and it requires no knowledge within the server about the other members of the group. This document contains the detailed specification for this messaging-based API and describes the Design of the OpenStack-Host changes in support of this functionality.

Also included in this document is an overview of a Linux-based reference implementation of the Guest-side software for implementing this Messaging-based API in the Guest. This Guest-side reference implementation, in this module, provides source code and make/build instructions which can be used strictly as reference or built and included 'as is' in your Guest image. Full build, install and usage instructions can be found in the README files included in the Guest-side implementation. This document simply provides an overview of the reference implementation.

High Availability for OPNFV

Host – Guest Server Group Messaging API

This module implements a simple Host-to-Guest Server Group Messaging API to provide a simple low-bandwidth datagram messaging and notification service for servers that are part of the same server group. This messaging channel is available regardless of whether IP networking is functional within the server, and it requires no knowledge within the server about the other members of the group.

The Host-to-Guest Server Group Messaging API is a message-based API using a JSON-formatted application messaging layer on top of a 'virtio serial device' between QEMU on the OpenStack Host and the Guest VM. JSON formatting provides a simple, humanly readable messaging format which can be easily parsed and formatted using any high level programming language being used in the Guest VM (e.g. C, Python, Java, etc.). Use of the 'virtio serial device' provides a simple, direct communication channel between host and guest which is independent of the Guest's L2/L3 networking.

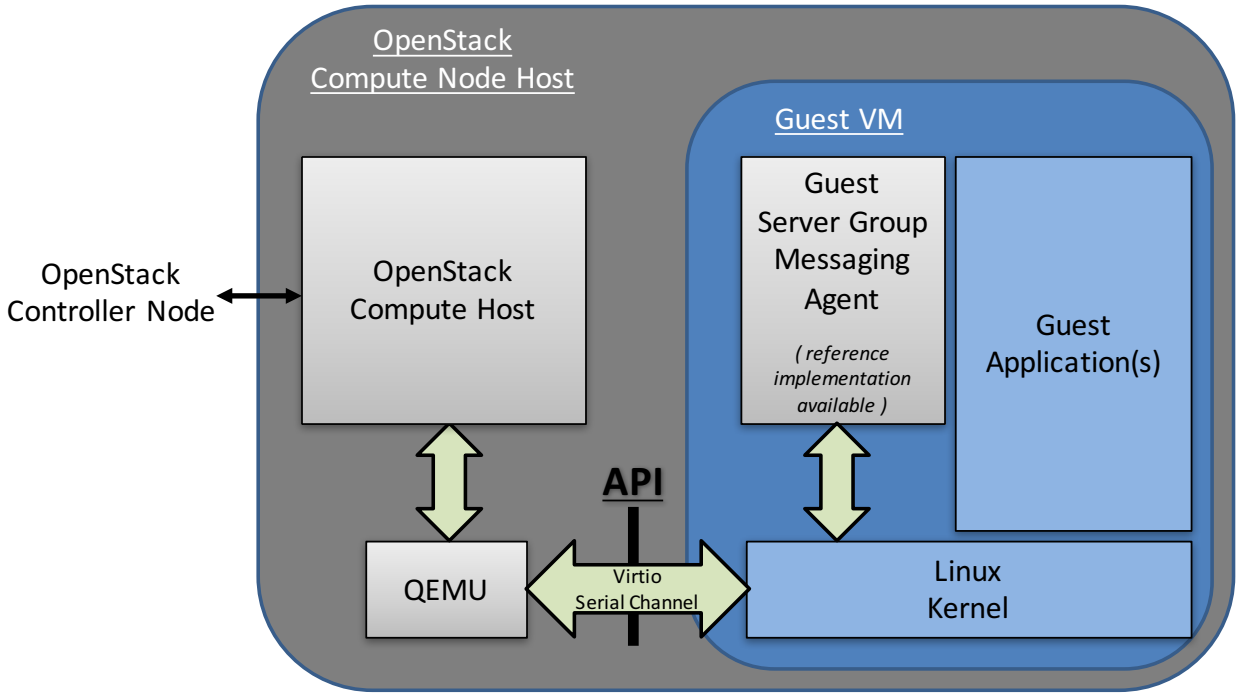

Figure 1 – Host-Guest Server Group Messaging API

Message Types and Semantics

For the Server Group Messaging API, there are four message types; Server Status Query/Response Messages, Asynchronous Server Status Change Notifications, Server Broadcast Messages and a Nack Message.

- Status Query (Guest → Host)
    - This allows a server (Guest) to query the current state of all servers within its server group, including itself,
- Status Response and Status Response Done (Host → Guest)
    - This is the Status Response from the OpenStack Host containing the current state of all servers within the Guest's server group, including this Guest,
    - This is a multiple message response with each response containing the status of a single server, followed by a final response (status response done) with no data,
    - Each message of the multiple message response has the transaction number (seq) of the status query request that it is related to,
- Notification Message (Host → Guest)
    - This asynchronous message provides the server (Guest) with information about changes to the state of other servers within the server group,
    - Each notification message contains the status of a single server,
- Broadcast Message (Guest → Host) → (Host → Guest)
    - This allows a server (Guest) to send a datagram (thru the Host, with a size of up to 3050 bytes) to all other servers (Guests) within its server group,
        - the payload portion of the message, 'data', can be formatted as desired by the Guest, however it must be a null-terminated string without embedded newlines,
        - the source field of the message is a unique, although opaque, address string representing the server (Guest) that sent the message.

- Nack (Host → Guest)
    - This is a message sent from the Host to the Guest when the Host receives a message with incorrect syntax,
    - It contains the message type of the original (incorrect) message and a log_msg describing the error,
    - This allows the Guest Application developer to debug issues when developing the Guest-side API code.

This service is not intended for high bandwidth or low-latency operations. It is best-effort, not reliable. Applications should do end-to-end acks and retries if they care about reliability.

High Availability for OPNFV

Virtio Serial Device

The transport layer of the Host-Guest Server Group Messaging API is a 'virtio serial device' (also known as a 'vmchannel') between QEMU (on the host) and the Guest VM. Device emulation in QEMU presents a virtio-pci device to the Guest, and a Guest Driver presents a char device interface to Guest userspace applications. This provides a simple transport mechanism for communication between the host userspace and the guest userspace. I.e. it is completely independent of the networking stack of the Guest, and is available very early in the boot sequence of the Guest.

This is a standard Linux QEMU/KVM feature. The Guest API for interfacing with the 'virtio serial device' can be found at http://www.linux-kvm.org/page/Virtio-serial_API . Examples of Guest code for opening, reading, writing, etc. from/to a 'virtio serial device' can also be found in the source code of the OPNFV High Availability 'Server Group Messaging' module. This module provides a Linux-based reference implementation of the Guest-side software for implementing the Guest Server Group Messaging API. Generally communicating with a 'virtio serial device' is very similar to communicating via a pipe, or a SOCK_STREAM socket.

There are however a few additional considerations to be aware of when using 'virtio serial devices':

- only one process at a time can open the device in the Guest,
- read() returns 0, if the Host is not connected to the device,
- write() blocks or returns -1 with error set to EAGAIN, if the Host is not connected,
- poll() will always set POLLHUP in revents when the Host connection is down.
    - This means that the only way to get event-driven notification of connection is to register for SIGIO. However, then a SIGIO event will occur every time the device becomes readable. The work-around is to selectively block SIGIO as long as the link is up is thought to be up, then unblock it on connection loss so a notification occurs when the link comes back.
- If the Host disconnects the Guest should still process any buffered messages from the device,
- Message boundaries are **not preserved**, the Guest needs to handle message fragment reassembly. Multiple messages can be returned in one read() call, as well as buffers beginning and ending with partial messages. This is hard to get perfect; one can study the `host_guest_msg.c` code in the OPNFV High Availability 'Guest Server Group Messaging' Module for ideas on how this can be handled.

High Availability for OPNFV

The QEMU/KVM created by OpenStack in order to host a Guest VM is created with a ‘virtio serial device’ named:

```
/dev/virtio-port/cgcs.messaging
```

for general OpenStack Host – to – Guest VM messaging (e.g. Host-Guest Server Group Messaging as well as other Host-Guest Messaging discussed in other OPNFV High Availability modules / documents).

JSON Message Syntax

The upper layer messaging format being used is 'Line Delimited JSON Format'. I.e. a '\n' character is used to identify message boundaries in the stream of data to/from the virtio serial device; specifically, a '\n' character is inserted at the start and end of the JSON Object representing a Message.

```
\n{key:value,key:value,…}\n
```

Note that key and values must NOT contain '\n' characters.

The upper layer messaging format is actually structured as a hierarchical JSON format containing a Base JSON Message Layer and an Application JSON Message Layer:

- the Base Layer provides the ability to multiplex different groups of message types on top of a single 'virtio serial device'
  e.g.
  - resource scaling,
  - server group messaging,
  - etc.

  and
- the Application Layer provides the specific message types and fields of a particular group of message types.

Base JSON Message Layer – Syntax

Again, the Base Layer provides the ability to multiplex different groups of message types on top of a single 'virtio serial device', e.g. resource scaling versus server group messaging etc.

Host – to – Guest Messages

Key	Value	Optionality*	Example value *(for Server Group Messaging)*	Description
"version"	integer	M	1	Version of the Base Layer Messaging
"source_addr"	string	M		Opaque string representing the host-side address of the message.
"dest_addr"	string	M	"cgcs.server_grp"	The Guest-side addressing of the message; specifically the Message Group Type
"data"	JSON Formatted String	M	See the following section on Application Layer JSON Message Layer – Syntax for Server Group Messaging.	Application layer JSON message whose schema is dependent on the particular Message Group Type

- M: Mandatory; O: Optional

Guest – to – Host Messages

Guest – to – Host Messages, from a Base Layer perspective, are identical to Host – to – Guest Messages except for swapped semantics of source_addr and dest_addr.

Application JSON Message Layer – Syntax

Again the Application Layer provides the specific message types and fields of a particular group of message types; in this case the messages of Server Group Messaging.

Guest – to – Host Messages

Status Query

Key	Value	Optionality*	Example value	Description
"version"	integer	M	1	Version of the interface.
"msg_type"	string	M	"status_query"	Type of the message.
"seq"	integer	M		Transaction number for the query; corresponding status_response and status_response_done messages will have a matching transaction number. This should be incremented on each status_query sent by Guest.

- M: Mandatory; O: Optional; (Condition)

Broadcast Message

Key	Value	Optionality*	Example value	Description
"version"	integer	M	1	Version of the interface
"msg_type"	string	M	"broadcast"	Type of the message.
"data"	string	M		Message content; can be formatted as desired by the Guest, however it must be a null-terminated string without embedded newlines.

- M: Mandatory; O: Optional; (Condition)

Host – to – Guest Messages

Status Response

Key	Value	Optionality*	Example value	Description
“version”	integer	M	1	Version of the interface.
“msg_type”	string	M	“status_response"	Type of the message.
“seq”	integer	M		Transaction number that the response belongs to.
“data”	string	M	see following info following table	The JSON formatted field containing the same contents as the normal notification that gets sent out by OpenStack’s notification service; see example below.

- M: Mandatory; O: Optional; (Condition)

Example contents of ‘data’ field containing status of a particular server:
(the same contents as the normal notification that gets sent out by OpenStack’s notification service)


```
{
      "state_description":"",
      "availability_zone":null,
      "terminated_at":"",
      "ephemeral_gb":0,
      "instance_type_id":10,
      "deleted_at":"",
      "reservation_id":"r-ed4i0c72",
      "instance_id":"4c074ce9-cbde-4040-9fdb-84b36168916b",
      "display_name":"jd_af_vm1",
      "hostname":"jd-af-vm1",
      "state":"active",
      "progress":"",
      "launched_at":"2015-11-26T14:33:03.000000",
      "metadata":{

      },
      "node":"compute-0",
      "ramdisk_id":"",
      "access_ip_v6":null,
      "disk_gb":1,
      "access_ip_v4":null,
      "kernel_id":"",
      "host":"compute-0",
      "user_id":"369b0103310d4a6bbf43ed389aac211d",
      "image_ref_url":"http:\/\/127.0.0.1:9292\/images\/32b386e1-5a21-47c4-
a04a-57910e7b0fc8",
      "cell_name":"",
      "root_gb":1,
```

```
      "tenant_id":"98b5838aa73c40728341336852b07772",
      "created_at":"2015-11-26 14:32:51.431455+00:00",
      "memory_mb":512,
      "instance_type":"jd1cpu",
      "vcpus":1,
      "image_meta":{
         "min_disk":"1",
         "container_format":"bare",
         "min_ram":"0",
         "disk_format":"qcow2",
         "base_image_ref":"32b386e1-5a21-47c4-a04a-57910e7b0fc8"
      },
      "architecture":null,
      "os_type":null,
      "instance_flavor_id":"101"
   }
```


Status Response Done

Key	Value	Optionality*	Example value	Description
“version”	integer	M	1	Version of the interface.
“msg_type”	string	M	“status_response_done"	Type of the message.
“seq”	integer	M		Transaction number that the response belongs to.

- M: Mandatory; O: Optional; (Condition)

Notification Message

Key	Value	Optionality*	Example value	Description
“version”	integer	M	1	Version of the interface.
“msg_type”	string	M	“notification"	Type of the message.
“data”	string	M	see contents of ‘data’ field documented for status_response	The JSON formatted output of the response to the Compute API GET /<version>/<tenant_id>/servers/<server_id> (see Compute API documentation for exact contents of response)

- M: Mandatory; O: Optional; (Condition)

Broadcast Message

Key	Value	Optionality*	Example value	Description
“version”	integer	M	1	Version of the interface
“msg_type”	string	M	“broadcast"	Type of the message.
“source_instance”	string	M		The unique, although opaque, address string representing the server (Guest) that sent the message.
“data”	string	M		Message content; can be formatted as desired by the Guest, however it must be a null-terminated string without embedded newlines.

- M: Mandatory; O: Optional; (Condition)

Nack

Key	Value	Optionality*	Example value	Description
“version”	integer	M	2	Version of the interface
“msg_type”	“nack”	M	“nack”	The type of message.
“orig_msg_type”	string	M	“broadcast”	The type of message that host previous received from guest.
“log_msg”	string	M	“failed to parse version”	Error message

- M: Mandatory; O: Optional; (Condition)

Examples

Examples of 'full' Server Group Messaging JSON messages, containing the Application JSON Message Layer encapsulated inside the Base JSON Messaging Layer.

Status Query:

Guest sends a query to OpenStack Host for status of all servers in Guest's Server Group:


```
\n{"version":1,"source_addr":"cgcs.server_grp",,"dest_addr":"cgcs.server_grp",
"data":{"version":1,"msg_type":"status_query",:"seq":1}}\n
```


OpenStack Host responds with the status of a server in the Guest's Server Group; one or more messages, each containing the status of one server in the Guest's Server Group:


```
\n{"version":1,"source_addr":"cgcs.server_grp",,"dest_addr":"cgcs.server
_grp",,"data":{"version":1,"msg_type":"status_response",,"seq":1,,"data":{
"state_description": "", "availability_zone": null, "terminated_at":
"", "ephemeral_gb": 0, "instance_type_id": 10, "deleted_at": "",
"reservation_id": "r-ed4i0c72", "instance_id": "4c074ce9-cbde-4040-
9fdb-84b36168916b", "display_name": "jd_af_vm1", "hostname": "jd-af-
vm1", "state": "active", "progress": "", "launched_at": "2015-11-
26T14:33:03.000000", "metadata": { }, "node": "compute-0",
"ramdisk_id": "", "access_ip_v6": null, "disk_gb": 1, "access_ip_v4":
null, "kernel_id": "", "host": "compute-0", "user_id":
"369b0103310d4a6bbf43ed389aac211d", "image_ref_url":
"http:\/\/127.0.0.1:9292\/images\/32b386e1-5a21-47c4-a04a-
57910e7b0fc8", "cell_name": "", "root_gb": 1, "tenant_id":
"98b5838aa73c40728341336852b07772", "created_at": "2015-11-26
14:32:51.431455+00:00", "memory_mb": 512, "instance_type": "jd1cpu",
"vcpus": 1, "image_meta": { "min_disk": "1", "container_format":
"bare", "min_ram": "0", "disk_format": "qcow2", "base_image_ref":
"32b386e1-5a21-47c4-a04a-57910e7b0fc8" }, "architecture": null,
"os_type": null, "instance_flavor_id": "101" }}}\n
```


OpenStack Host responds with response done for the current outstanding query request; with no data:


```
\n{"version":1,"source_addr":"cgcs.server_grp",,"dest_addr":"cgcs.server
_grp",,"data":{"version":1,"msg_type":"status_response_done",,"seq":1}}\n
```

Notification:

A notification of a server state change from OpenStack Host:


```
\n{"version":1,"source_addr":"cgcs.server_grp”,"dest_addr":"cgcs.server_grp”,
"data":{"version":1,"msg_type":"notification","data”:{ "state_description":
"", "availability_zone": null, "terminated_at": "", "ephemeral_gb": 0,
"instance_type_id": 10, "deleted_at": "", "reservation_id": "r-ed4i0c72",
"instance_id": "4c074ce9-cbde-4040-9fdb-84b36168916b", "display_name":
"jd_af_vm1", "hostname": "jd-af-vm1", "state": "active", "progress": "",
"launched_at": "2015-11-26T14:33:03.000000", "metadata": { }, "node":
"compute-0", "ramdisk_id": "", "access_ip_v6": null, "disk_gb": 1,
"access_ip_v4": null, "kernel_id": "", "host": "compute-0", "user_id":
"369b0103310d4a6bbf43ed389aac211d", "image_ref_url":
"http:\/\/127.0.0.1:9292\/images\/32b386e1-5a21-47c4-a04a-57910e7b0fc8",
"cell_name": "", "root_gb": 1, "tenant_id":
"98b5838aa73c40728341336852b07772", "created_at": "2015-11-26
14:32:51.431455+00:00", "memory_mb": 512, "instance_type": "jd1cpu", "vcpus":
1, "image_meta": { "min_disk": "1", "container_format": "bare", "min_ram":
"0", "disk_format": "qcow2", "base_image_ref": "32b386e1-5a21-47c4-a04a-
57910e7b0fc8" }, "architecture": null, "os_type": null, "instance_flavor_id":
"101" }}}\n
```


Broadcast:

A broadcast message to/from another server:


```
\n{"version":1,"source_addr":"cgcs.server_grp”,"dest_addr":"cgcs.server_grp”,
"data":{"version":1,"msg_type":"broadcast","source_instance”:”instance-
00000001”,“data”:”Hello World”}}\n
```


Nack:

A Nack from OpenStack Host for an invalid broadcast message sent from Guest.


```
\n{"version":1,"source_addr":"cgcs.server_grp”,"dest_addr":"cgcs.server_grp”,
"data":{"version":1,"msg_type":"nack","orig_msg_type":"broadcast","log_msg":"
failed to parse version"}}\n
```

High Availability for OPNFV

Design of OpenStack-Host Server Group Messaging

This section provides an overview of the design for supporting Host-to-Guest Server Group Messaging in OpenStack.

The implementation of the OpenStack Host design can be found in the OPNFV High Availability 'Server Group Messaging' Module. This Module provides Nova source code patches for OpenStack Newton, source code for a new Host-to-Guest Host Agent Process and README for building and installing the Host Agent Process and Nova patches. This section simply provides an overview of the design.

The diagram below provides the architecture diagram of the design for supporting Host-to-Guest Server Group Messaging in OpenStack:

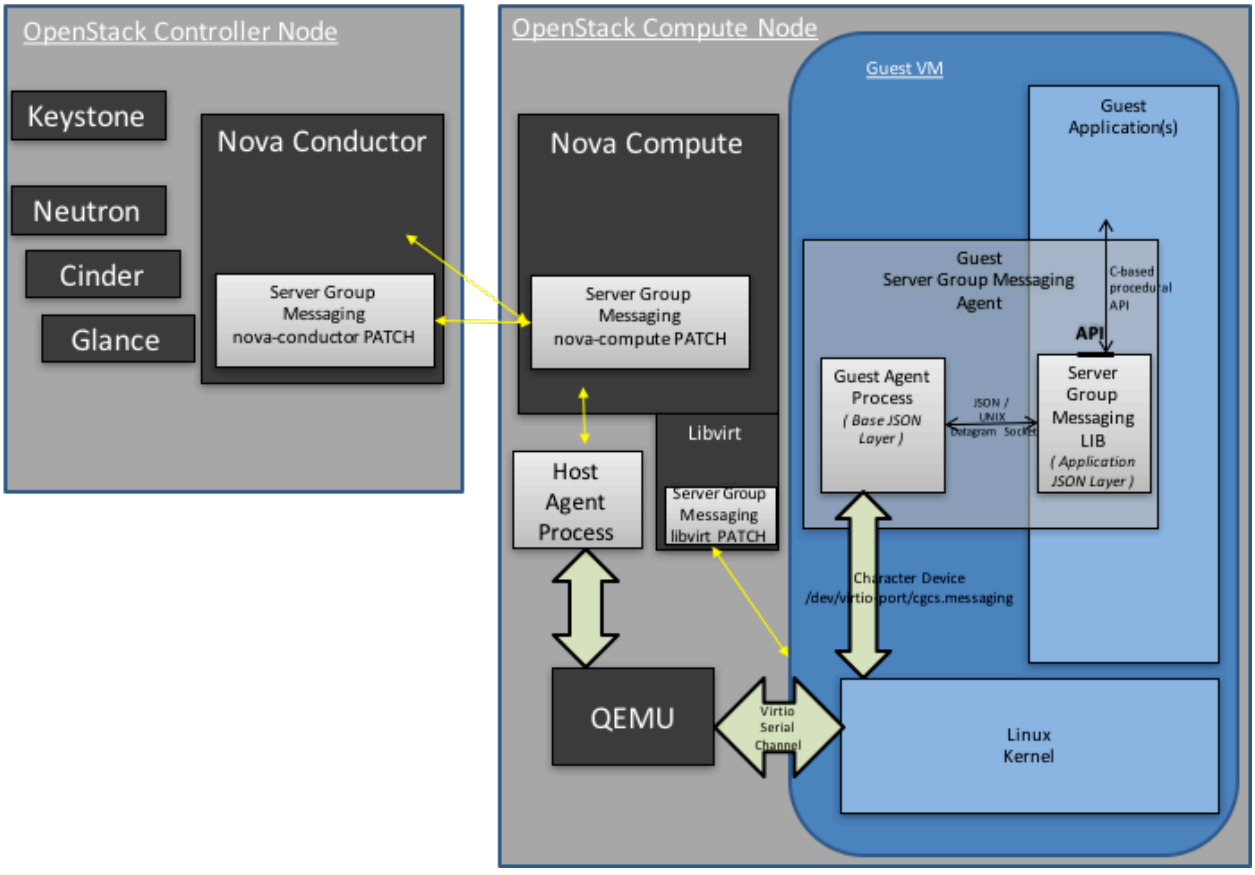

Figure 2 – Architecture for OpenStack Host support of Guest Server Group Messaging

Where:

- Libvirt Patch
  - Checks for a new ‘sw:wrs:srv_grp_messaging’ Boolean flavor extraspec which indicates whether the guest supports server group messaging or not,
  - If supported, the libvirt changes configure device emulation in QEMU to present a virtio-pci device to the VM, for the Host-to-Guest communications.

- Host Agent Process
  which implements the Base JSON Messaging Layer between the Host and Guest.
  This includes:
  - opening/reading,/writing and general management of the unix socket presented by QEMU for communicating with the Guest over the virtio-pci device,
  - parsing/processing/formatting of the Base JSON Messaging Layer of the Guest-Host interface, where processing of the messages involves:
    - the multiplexing/de-multiplexing of Application Layer messages to/from registered Host Application Layer Agents; in this particular case Nova-Compute who is responsible for handling Server Group Messaging to/from Guests of the local compute,
    - the interface between the Host Agent Process and Nova-Compute:
      - is a message-based interface;
      - specifically, a JSON Messaging Layer over a UNIX Datagram socket containing
        - the source-addr and dest-addr for the Base JSON Messaging Layer of the Guest-Host Interface,
        - the instance-address, and
        - an application-level JSON message to be put in the ‘data’ field of the Base JSON Messaging Layer of the Guest-Host Interface.

- Nova-Compute Patch
  - Manages Server Group Messaging on the Compute Node
  - Specifically, it implements the Application JSON Messaging Layer for Server Group Messaging on top of the Base JSON Messaging Layer UNIX Datagram socket provided by the Host Agent,
  - On receiving a Server Status Query from the Guest
    - Nova-Compute makes an RCP call to Nova-Conductor to request the status of all servers in its server group, and
    - On receiving these back from Nova-Conductor, forwards them on to the Host Agent and the Guest VM,
  - On receiving a Broadcast Message from the Guest
    - Nova Compute makes an RCP call to Nova-Conductor to request that the Broadcast message be sent to all servers of the server group, and

        - Again on receiving any Broadcast Messages from Nova-Conductor, forwards them on to the Host Agent and therefore the Guest VM.

- Nova-Conductor Patch
    - Provides centralized functions in support of Server Group Messaging
    - Supports an RCP query from nova-compute for the status of all servers within a server group
        - Nova-conductor looks up which instances are in the server group of the requesting instance, and then sends back a single message containing the status of all instances to nova-compute,
    - Supports an RCP query from nova-compute for the broadcasting of a message to all servers wthin the requesting instance's server group
        - nova-conductor looks up which instances are in the server group of the requesting instance, then figures out which compute nodes they're on, and sends one RPC message to each relevant compute node, with a list of instances to forward to.
    - Hooks into the Nova notification system in order to detect state changes in servers and then broadcast that state change notification to all servers of the server group; i.e. again by looks up which instances are in the server group of the requesting instance, then figures out which compute nodes they're on, and sends one RPC message (containing the state change notification) to each relevant compute node, with a list of instances to forward to.

Reference Implementation of Guest Server Group Messaging

This section provides an overview of the Linux-based reference implementation of the Guest-side software for implementing this Host-to-Guest Server Group Messaging API in the Guest.

This reference implementation can be found in the OPNFV High Availability 'Server Group Messaging' Module. This Module provides source code and make/build instructions which can be used strictly as reference or built and included 'as is' in your Guest image. Full build, install and usage instructions can be found in the README files included in the module. This section simply provides an overview of the reference implementation.

The diagram below provides the architecture diagram of the reference implementation:

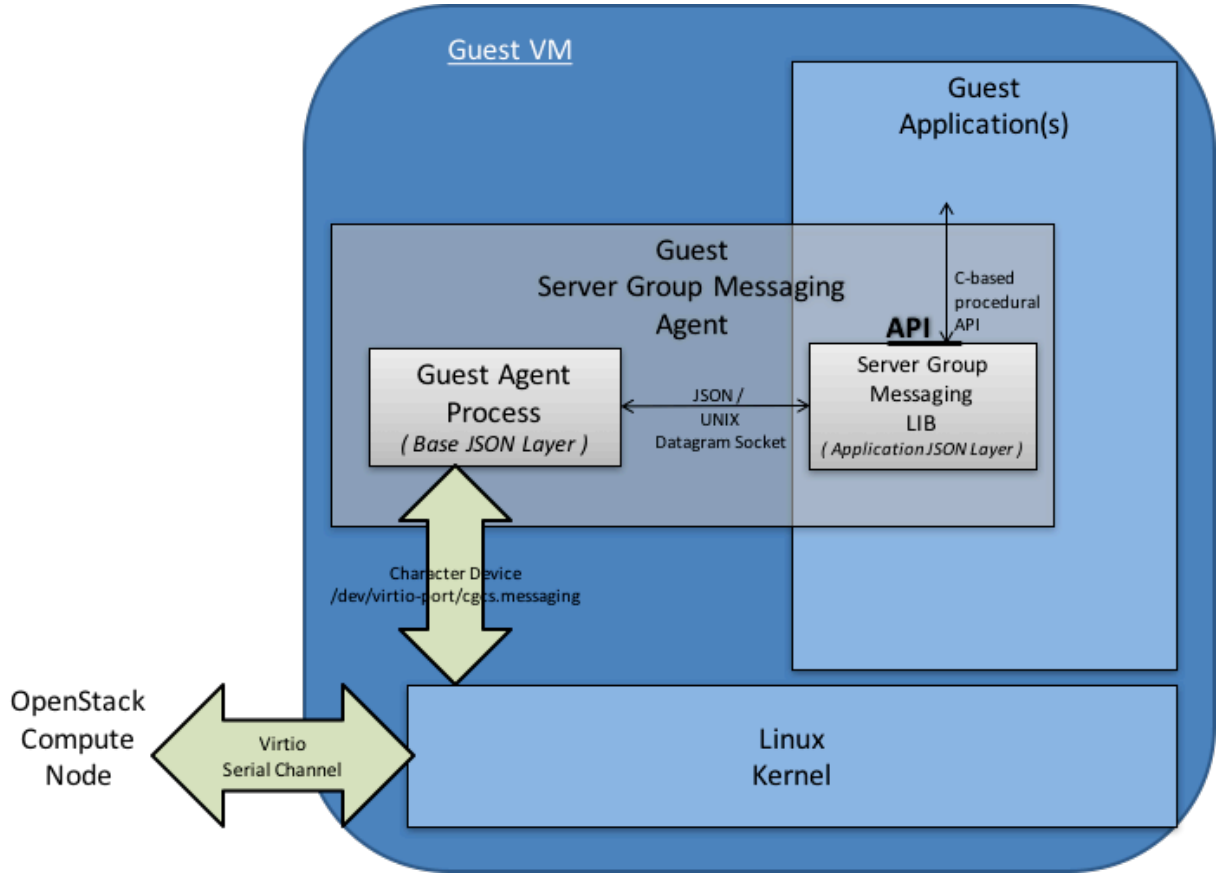

Figure 3 – Reference Implementation Architecture for Guest Server Group Messaging

Where:

- A Guest Agent Process implements the Base JSON Messaging Layer.
  This includes:
  - opening/reading,/writing and general management of the virtio serial device between the Guest and the Host,
  - parsing/processing/formatting of the Base JSON Messaging Layer of the Guest-Host interface, where processing of the messages involves:
    - the multiplexing/de-multiplexing of Application Layer messages to/from registered Guest Application Layer Agents; in this particular case a Guest Application Process responsible for handling Server Group Messaging for the Guest,
    - the interface between the Guest Agent Process and the Guest Application Process responsible for Server Group Messaging for the Guest:
      - is a message-based interface;
      - specifically a JSON Messaging Layer over a UNIX Datagram socket,
        - where the UNIX Socket Address is the Message Group Type (cgcs.server_grp in this particular case) specified within the Base JSON Messaging Layer and
        - where the JSON Message consists of the 'data' field contents specified within the Base JSON Messaging Layer.
  - NOTE
    - The implementation files for the Guest Agent Process within the OPNFV High Availability 'Server Group Messaging' module are:
      - misc.h, guest_host_msg.h, host_guest_msg_type.h,
      - guest_agent.c, host_guest_msg.c, lib_guest_host_msg.c

- A Server Group Messaging Lib which provides a C-based procedural API for a Guest Application Process to interface with the Guest Agent Process.

  Specifically this library implements:

  - the interface described above; a JSON Messaging Layer over a UNIX Datagram socket.
    - where the UNIX Socket Address is the Message Group Type (cgcs.server_grp in this particular case) specified within the Base JSON Messaging Layer and
    - the JSON Message consists of the 'data' field contents specified within the Base JSON Messaging Layer.
  - with a C-based procedural API.

High Availability for OPNFV

- NOTE
  - the definition and implementation of the Server Group Messaging Lib within the OPNFV High Availability 'Server Group Messaging' Module are:
    - server_group.h and server_group.c
    - server_group_app.c *(a sample usage of the API)*

```
server_group.h:

/* Function signature for the server group broadcast messaging callback function.
 * source_instance is a null-terminated string of the form "instance-xxxxxxxx".
 * The message contents are entirely up to the sender of the message.
 */
typedef void (*sg_broadcast_msg_handler_t)(const char *source_instance,
              const char *msg, unsigned short msglen);

/* Function signature for the server group notification callback function.  The
 * message is basically the notification as sent out by nova with some information
 * removed as not relevant.  The message is not null-terminated, though it is
 * a JSON representation of a python dictionary.
 */
typedef void (*sg_notification_msg_handler_t)(const char *msg, unsigned short msglen);

/* Function signature for the server group status callback function.  The
 * message is a JSON representation of a list of dictionaries, each of which
 * corresponds to a single server.  The message is not null-terminated.
 */
typedef void (*sg_status_msg_handler_t)(const char *msg, unsigned short msglen);



/* Get error message from most recent library call that returned an error. */
char *sg_get_error();

/* Allocate socket, set up callbacks, etc.  This must be called once before
 * any other API calls.
 *
 * Returns a socket that must be monitored for activity using select/poll/etc.
 * A negative return value indicates an error of some kind.
 */
int init_sg(sg_broadcast_msg_handler_t broadcast_handler,
              sg_notification_msg_handler_t notification_handler,
              sg_status_msg_handler_t status_handler);

/* This should be called when the socket becomes readable.  This may result in
 * callbacks being called.  Returns 0 on success.
 * A negative return value indicates an error of some kind.
 */
int process_sg_msg();

/* max msg length for a broadcast message */
#define MAX_MSG_DATA_LEN 3050



/* Send a server group broadcast message.  Returns 0 on success.
```

```
 * A negative return value indicates an error of some kind.
 */
int sg_msg_broadcast(const char *msg);

/* Request a status update for all servers in the group.
 * Returns 0 if the request was successfully sent.
 * A negative return value indicates an error of some kind.
 *
 * A successful response will cause the status_handler callback
 * to be called.
 *
 * If a status update has been requested but the callback has not yet
 * been called this may result in the previous request being cancelled.
 */
int sg_request_status();
```