RabbitMQ - Message Format

The messages sent to the broker should be encoded dictionaries. The contents of the dictionary should always follow a predefined protocol. There are several pragmatics to take into consideration first, in regards to the communication:

1. Direct Communication;
2. Fanout (Broadcast) Communication;

Direct Communication

A direct exchange is **an exchange which route messages to queues based on the message routing key**. The suggested payload consists in a dictionary with the following keys:

1. **destination** - the receiver's identifier of the message: what entity do I want to send this message to?
2. **origin** - the sender's identifier of the message: what entity sends this message?
3. **method** - the identifier of the method to be executed: what method do I want the receiver to execute/trigger?
4. **type** - the message's type: is it a request or a response?
5. **content** - the content needed to execute the method: what information does the receiver need to execute the method properly? It's essentially the arguments/parameters of the method.

The message routing key can be calculated with the **'destiny'** and **'type'** field, the latter being optional. If each entity listens to a single queue of messages, the 'destination' field suffices.
However, if each entity listens to 2 queues of messages, one for the requests and another for the responses, the 'type' of the message must be taken into account.

Example:
Let's assume that the central system (CSMS) wishes a certain charger to cancel a given reservation. The CSMS will send a

CancelReservationRequest(reservation_id=1) to the OCPP Client of the charger. Then, the OCPP Client will redirect that message to the entity responsible for executing the request (Decision Point).

Let's assume the message to send before encoding is the following dictionary:

```
{
    “destination": "decision_point",
    "origin": "ocpp_client",
    “method": "request_cancel_reservation",
    "type": "request",
    "content": {
        "reservation_id": 1
    }
}
```

The Decision Point is listening to 2 queues, namely the 'decision_point_requests' and the 'decision_point_responses'.

If the type of the message is 'request' and the destination of the message is 'decision_point', the message will be sent to the 'decision_point_requests'. And so on.

Fanout (Broadcast) Communication

A fanout exchange routes messages to all of the queues that are bound to it and the routing key is ignored. The suggested payload consists in a dictionary with the following keys:

1. **intent** - the intent of the message: what functionality/method do I want the receiver to execute/trigger?
2. **type** - the message's type: is it a request or a response?
3. **content** - the content needed to execute the method: what information does the receiver need to execute the method properly? It's essentially the arguments/parameters of the method.

Since it's fanout (broadcast), it's not needed to calculate the specific queue to send the message to. It's assumed that each entity will know what to do with every message it receives: ignore or take action.

Example:
Let's assume that the central system (CSMS) wishes a certain charger to cancel a given reservation. The CSMS will send a CancelReservationRequest(reservation_id=1) to the OCPP Client of the charger. Then, the OCPP Client will redirect that message to the entity responsible for executing the request (Decision Point).

Let's assume the message to send before encoding is the following dictionary:

```
{
        "intent": "cancel_reservation",
        "type": "request",
        “content": {
                "reservation_id": 1
        }
}
```

Each entity is listening to a single queue of messages. In this case, every entity listening to a queue will receive this message. However, some entities will ignore the message since they are not responsible for the received intent. In this case, the decision point is responsible for canceling a reservation, so it will take action.

Example of Constants

In this section, it's presented some examples of possible entities' identifiers, types of message, possible methods, possible intents, and so on. It's important to notice that the lists are not completed and are simply presented to give clarity.

Identifiers of entities

DECISION_POINT = 'decision_point'

OCPP_CLIENT = 'ocpp_client'

Identifiers of message

REQUEST = 'request'

RESPONSE = 'response'

Identifiers of methods/intents

AUTHORIZE = 'request_authorize'

BOOT_NOTIFICATION = 'request_boot_notification'

DIAGNOSTICS_STATUS_NOTIFICATION = 'request_diagnstics_status_notification'

FIRMWARE_STATUS_NOTIFICATION = 'request_firmware_status_notification'

HEARTBEAT = 'request_heartbeat'

METER_VALUES = 'request_meter_values'

START_TRANSACTION = 'request_start_transaction'

STATUS_NOTIFICATION = 'request_status_notification'

STOP_TRANSACTION = 'request_stop_transaction'

CANCEL_RESERVATION = 'cancel_reservation'

CHANGE_AVAILABILITY = 'change_availability'

CHANGE_CONFIGURATION = 'change_configuration'

CLEAR_CACHE = 'clear_cache'

CLEAR_CHARGING_PROFILE = 'clear_charging_profile'

GET_COMPOSITE_SCHEDULE = 'get_composite_schedule'

GET_CONFIGURATION = 'get_configuration'

GET_DIAGNOSTICS = 'get_diagnostics'

GET_LOCAL_LIST_VERSION = 'get_local_list_version'

REMOTE_START_TRANSACTION = 'remote_start_transaction'

REMOTE_STOP_TRANSACTION = 'remote_stop_transaction'

RESERVE_NOW = 'reserve_now'

RESET = 'reset'

SEND_LOCAL_LIST = 'send_local_list'

SET_CHARGING_PROFILE = 'set_charging_profile'

TRIGGER_MESSAGE = 'trigger_message'

UNLOCK_CONNECTOR = 'unlock_connector'

UPDATE_FIRMWARE = 'update_firmware'