MuntsOS

Application Note #25: RabbitMQ Enterprise Message Broker Client Programs

Revision 1
31 December 2025

by Philip Munts
***dba* Munts Technologies**
http://tech.munts.com

Introduction

This application note describes some examples of and best practices for **MuntsOS Embedded Linux** (hereafter just **MuntsOS**) target programs that send messages to and/or receive messages from a **RabbitMQ Enterprise Message Broker** (hereafter just **RabbitMQ**), which is often used in backend systems to implement an information bus architecture.

An information bus architecture works well for IoT (Internet of Things) networks, with each IoT end node running software that pushes messages to and/or pulls messages from the information bus.

Broker

A **RabbitMQ** Broker is an instance of **RabbitMQ** installed, configured, and running on a server computer accessible from a **MuntsOS** target computer. Installation and configuration of **RabbitMQ** are beyond the scope of this document. A broker, *per se*, is invisible to client programs.

Virtual Host

A **RabbitMQ** Virtual Host (hereafter, **vhost**) is analogous to a web server virtual host. Just as tech.munts.com and repo.munts.com are web servers with separate name spaces that run on the same piece of hardware, each vhost creates a distinct name space at the connection level.

Every broker has an anonymous default vhost sometimes referred to as "/".

Named vhosts can be managed (created and destroyed) using the `rabbitmqctl` command.

Each vhost its own URI of the form:

```
scheme://user:password@host[:port][/[vhost]]
```

where:

`scheme` can be either `amqp` (unencrypted) or `amqps` (encrypted).

`user` and `password` have default values `guest` and `guest`.

`host` is an IP address or domain name

`port` is a TCP port number. If omitted, one of the default port numbers is selected: 5672 (unencrypted) or 5671 (encrypted).

`vhost` is the name of a vhost. If omitted, with or without the trailing `/`, the anonymous default vhost is selected.

A client program may connect to more than one vhost, each with a distinct URI and a separate connection object.

Exchange

A **RabbitMQ** Exchange (hereafter, **exchange**) is a named entity inside a vhost name space to which a client program can write/publish/push/send messages, each of can include a routing key string. The exchange writes each message to one or more queues. There are many exchange types, each with different forwarding policies. The most common and useful types are:

A **fanout** exchange writes every incoming message to every queue bound to the exchange. The routing key is ignored.

A **direct** exchange writes incoming messages to every queue bound to the exchange with a matching (exactly) routing key.

A **topic** exchange also writes incoming message to every queue bound to the exchange with a matching routing key. However, each queue's routing key is a rudimentary regular expression consisting of tokens separated by periods, *e.g.* **`House.Bedroom.Table`**. Two special wildcard tokens are defined: **`*`** matches exactly one token and **`#`** matches zero or more tokens.

So, a topic exchange will write an incoming message with the routing key **`House.Bedroom.Table`** to queues created with routing keys **`House.Bedroom.Table`**, **`House.Bedroom.*`**, **`House.*.Table`**, **`House.*.*`**, **`*.Bedroom.*`** , **`*.*.Table`** and **`House.#`**.

Every vhost will contain default exchanges named **`amq.fanout`**, **`amq.direct`**, and **`amq.topic`** among others.

Queue

A **RabbitMQ** Queue (hereafter, **queue**) is a **FIFO** (First In First Out) data structure subsystem inside a vhost name space. **RabbitMQ** queues are always *output* queues. Only exchanges can write to queues and only client programs can read from queues. A queue must be bound to an exchange before a client program can read from it.

In the context of this Application Note, the **ephemeral** queue is most useful. An ephemeral queue must be named (preferably with a name generated by the broker or a UUID string), created, and bound with a routing key to an exchange by a client program. Every ephermal queue must be marked as **exclusive**, which means it is private to the client program that created it and that it will be deleted when the client program closes its connection to the vhost.

Example Client *aka* End Node Programs

The Linux Simple I/O Library contains the following Ada and C# example **RabbitMQ** client *aka* end node programs, which can be cross-compiled to run on **MuntsOS** target computers:

- `test_rabbitmq_consume.adb`
- `test_rabbitmq_produce.adb`
- `test_rabbitmq_consume`
- `test_rabbitmq_produce`

Each of these example programs obtain their runtime configuration from the following environment variables:

● `RABBITMQ_SCHEME`	(default `amqp`)
● `RABBITMQ_USER`	(default `guest`)
● `RABBITMQ_PASS`	(default `guest`)
● `RABBITMQ_SERVER`	(default `localhost`)
● `RABBITMQ_PORT`	(default `5672`)
● `RABBITMQ_VHOST`	(default `/`)
● `RABBITMQ_EXCHANGE`	(default `amq.topic`)
● `RABBITMQ_ROUTING`	(default empty string "")

The minimum set of environment variables that you will need to define in the file `/etc/environment` on your **MuntsOS** target computer is:

- `RABBITMQ_USER`
- `RABBITMQ_PASS`
- `RABBITMQ_SERVER`
- `RABBITMQ_VHOST`

Additionally, you may need to define `RABBITMQ_ROUTING` to set the **RabbitMQ** routing key (*e.g.* topic) if the client program does not generate a custom routing key on the fly.

Case Study #1 -- LoRa Sensor Network

Imagine some Raspberry Pi microcomputers running **MuntsOS Embedded Linux** are members of a LoRa sensor network using the Amateur Radio LoRa P2P Network Flavor #1 Protocol.

One Raspberry Pi, with radio node ID `N7AHL-1`, is a data aggregator node running the .Net Core program `wioe5_ham1_rabbitmq`. It listens for incoming radio messages containing sensor data samples. For each sensor data radio message received, it extracts the sensor data, adds a timestamp, adds some radio network metadata, and forwards the result with a routing key like `N7AHL-1.N7AHL-2.Sensor.Temperature` to the `amqp.topic` exchange in a **RabbitMQ** vhost.

Another Raspberry Pi microcomputer, with radio node ID `N7AHL-2`, and without Internet access, is a sensor node running the Python3 program `wioe5_ham1_thermometer.py`. It periodically measures the temperature of a Type K thermocouple and transmits a radio message containing the measurement to `N7AHL-1`.

Elsewhere, another computer named `logger` runs a **RabbitMQ** client program connected to the same **RabbitMQ** vhost as `N7AHL-1`. The client program on `logger` has created an ephemeral queue and bound it to the `ampq.topic` exchange. It drains temperature sample messages from the ephemeral queue and stores the time stamped temperature samples to a database server.

`logger` will do different things depending on what routing key it used to connect to the **RabbitMQ** vhost:

`N7AHL-1.N7AHL-2.Sensor.Temperature`:	`logger` stores temperature samples received by `N7AHL-1` from `N7AHL-2`.
`N7AHL-1.*.Sensor.Temperature`:	`logger` stores temperature samples received by `N7AHL-1` from any sensor node that is a member of the same radio network.
`SA7CHS-1.*.Sensor.Temperature`:	`logger` stores temperature samples from another completely separate radio network in Sweden.
`*.*.Sensor.Temperature`:	`logger` stores temperatue samples from any sensor node anywhere in the world.
`*.*.Sensor.*`:	`logger` stores all kinds of sensor data (at least what it can parse) from any sensor node anywhere in the world.

The last two routing keys probably make the most sense for `logger` to use. After the temperature samples have been stored in the database, an you can perform database queries to calculate average temperatures, produce reports, etc.

Now suppose you want an eye candy real time display (*e.g.* a flat screen TV on a wall) driven by a computer named `monitor`, of the most recent temperatures measured by each of the sensor nodes. All you would have to do is write another **RabbitMQ** client program, perhaps a .Net WPF application, that connects to the same vhost as `logger` and creates and binds an output queue using the `*.*.Sensor.Temperature` routing key.

Neither `logger` nor `monitor` are aware of each other and neither affects the other unless the sheer number of sensor nodes, aggregator nodes, and client programs is large enough to overwhelm the hardware the broker is running on.