# Fedmsg Migration Tools Documentation

*Release 0.1.2*

**Jeremy Cline**

**Nov 12, 2020**

# Contents

A set of tools to assist in Fedora's migration from fedmsg to an broker-based message system.

CHAPTER 1

# Migration Documentation

## 1.1 Migration Overview

Fedora's infrastructure makes heavy use of the event-driven services. In this pattern, one service emits a message when a particular event occurs and a different service uses that event to trigger some work.

As a concrete example, release-monitoring.org polls upstream projects for new versions. When it discovers a new version, it emits a message. When the-new-hotness receives this message, it tries to automatically update the project's spec file and build a new version for testing.

### 1.1.1 Migrate from what?

Fedora currently sends messages using ZeroMQ. ZeroMQ provides a familiar socket API and all the basic building blocks necessary for common (and not-so-common) message passing patterns.

### 1.1.2 Why Migrate?

Our current usage of ZeroMQ is limited to its PUB/SUB sockets. There's no broker. Although it's true that a broker introduces a point of failure, the broker offers a number of useful features out of the box like guaranteed delivery, durable message queues, authentication and authorization, monitoring, etc.

It's true ZeroMQ can be used to achieve all the features a broker like RabbitMQ provides. It's a great library to start with if if you're interested in building a message broker. Fedora shouldn't try to build a broker, though.

### 1.1.3 Why AMQP and RabbitMQ

The core requirements for Fedora is a messaging protocol that offers:

- Messages must be delivered to consumers at least once
- Clients must be authenticated to publish messages and authorized to publish using a given topic

RabbitMQ is a mature, well-established broker that supports AMQP v0.8.0, v0.9.0, and v0.9.1 as well as 0.9.1 with a number of protocol extensions. It also supports the STOMP (v1.0, v1.1, and v1.2) and MQTT (v3.1.1 at this time) protocols via plugins.

We already run RabbitMQ in Fedora infrastructure and have some familiarity with it. It supports authentication and topic-based authorization. With the RabbitMQ publisher ack extension, publishers can be confident their messages were published and consumers will receive it at least once.

There are other message protocols: STOMP, MQTT, and AMQP 1.0 just to name a few. There are other message brokers: Apache ActiveMQ, Qpid, and Mosquitto, for example. Some other protocols are capable of meeting our needs, and the broker Fedora infrastructure wishes to run is a choice best left to the system administrators.

This document assumes AQMP and RabbitMQ for simplicity and to demonstrate a concrete system that can meet Fedora's needs. It would be reasonably straight- forward to implement the Python API (and message bridges) using, for example, STOMP.

### 1.1.4 The Plan

This plan assumes the following requirements:

- No flag day.
- Don't disrupt any services or applications.
- Don't break any services outside of Fedora's infrastructure relying on these messages.

#### Deploy a Broker

The first step is to deploy a broker in Fedora to use. The broker should support (at a minimum) AMQP. RabbitMQ is probably a safe choice, but other brokers that support AMQP (0.9.1) are fine.

#### Building Bridges

In order to avoid a flag day, bridges from AMQP to ZeroMQ and ZeroMQ to AMQP need to be implemented and deployed. Some care needs to be taken to ensure messages don't get looped endlessly between AMQP and ZeroMQ. In order to avoid endless loops, two AMQP topic exchanges will be used. These will be used to separate those messages originally published to ZeroMQ from those originally published to AMQP. We'll call these exchanges "amq.topic" and "zmq.topic". The setup is as follows:

1. The ZeroMQ to AMQP bridge publishes message to the "zmq.topic" exchange, using the ZeroMQ topic as the AMQP topic.
2. AMQP publishers publish to the "amq.topic" exchange.
3. The AMQP to ZeroMQ bridge binds a queue to the "amq.topic" exchange and publishes all messages to a ZeroMQ PUB socket. This socket is added to the list of sockets all fedmsg consumers connect to.
4. When a ZeroMQ consumer is migrated to AMQP, the queue it sets up is bound to both the "zmq.topic" and "amq.topic" with the topics it's interested in.

Fig. 1: A diagram of how messages are routed using the AMQP <-> ZMQ bridging.

This allows both fedmsg publishers and subscribers to migrate at their leisure. Once all Fedora services are migrated, the ZeroMQ to AMQP bridge can be turned off. If Fedora wishes to continue supporting the external ZeroMQ interface, the AMQP to ZeroMQ bridge should continue to be run.

#### Testing

In order to validate that the bridges are functioning, a small service will be run during the transition period that connects to fedmsg and to the AMQP queues to compare messages. This will help catch format changes, configuration issues that lead to message loss, etc. It will also likely have false positives since ZeroMQ PUB/SUB sockets are designed to be unreliable and determining if a message is lost or merely slow to be delivered is a difficult problem.

An initial implementation of this service is provided by these tools, see the fedmsg-migration-tools CLI for details.

#### Converting Applications

After the bridges are running, applications are free to migrate. Rather than attempting to add support to the existing fedmsg APIs, a new API has been created. The reason for this is that the fedmsg API doesn't offer useful exceptions on errors and has a huge dependency chain we don't need.

This new library, fedora-messaging, handles a lot of the boilerplate code and offers some useful APIs for the simple publisher and consumer. However, users are free to use the AMQP client libraries directly if they so choose.

#### Supporting External Consumers

Access to the AMQP broker won't be available to outside consumers, so even after all applications have migrated to AMQP, we can continue to run the AMQP to ZeroMQ bridge so external users can receive messages via ZeroMQ.

### 1.1.5 Demo

Using the fedora-messaging library, fedmsg-migration-tools offers alpha-quality implementations of the bridges as well as the verification service. You can set these up at home with relative ease:

1. Install RabbitMQ. On Fedora, `sudo dnf install rabbitmq-server`
2. Start RabbitMQ: `sudo systemctl start rabbitmq-server`
3. Enable the management plugin for a nice HTTP interface with `sudo rabbitmq-plugins enable rabbitmq_management`
4. Navigate to http://localhost:15672/ and log in to see the monitoring dashboard. The default username is `guest` and the password is `guest`.
5. Install the migration tools:

```
mkvirtualenv --python python3 fedmsg_migration_tools
pip install fedmsg-migration-tools
```

6. Start the ZeroMQ to AMQP bridge:

```
fedmsg-migration-tools zmq_to_amqp --zmq-endpoint "tcp://fedoraproject.org:9940" -
↪-zmq-endpoint "tcp://release-monitoring.org:9940"
```

7. In a second terminal, start the AMQP to ZeroMQ bridge:

```
workon fedmsg_migration_tools
fedora-messaging consume --callback="fedmsg_migration_tools.bridges:AmqpToZmq"
```

8. Congratulations, you now have a functional bridge to and from AMQP. ZeroMQ messages are being published to the "zmq.topic" exchange, and any messages published to the "amq.topic" are bridged to ZeroMQ publishing sockets bound to all available interfaces on port 9940.

9. Run the verification service to confirm messages are available via both AMQP and ZeroMQ:

```
fedmsg-migration-tools verify_missing --zmq-endpoint "tcp://fedoraproject.org:9940
↪" --zmq-endpoint "tcp://release-monitoring.org:9940"
```

## 1.2 Performance

One concern about using a broker rather than ZeroMQ has been that a broker could not keep up with the scale of Fedora's messaging needs. See the old fedmsg documentation for details on these concerns. However, we now have a multi-year history of messaging trends in Fedora's infrastructure thanks to datagrepper, and we can compare these with the throughput of RabbitMQ using various durability settings, server configuration, and hardware.

### 1.2.1 Current Message Rates

Datagrepper connects to all ZeroMQ publishers and records the published messages in a PostgreSQL database. Using this, we can get data about the number of messages published per second.

**Note:** All numbers were collected using 1500 UTC on April 26, 2018 as the starting point.

- Over the last hour 3,905 messages were published which is on average 1.085 messages per second.
- Over the last 24 hours 70,802 messages were published which is on average 0.819 messages per second
- Over the last 4 weeks, 1,653,835 messages were published which is on average 0.684 messages per second
- Over the last 365.25 days, 20,896,220 messages were published which is on average 0.662 messages per second

Obviously, these are just averages and will vary by time of day, year, etc.

### 1.2.2 RabbitMQ Performance

Before looking at actual numbers, it is important to understand what affects RabbitMQ's performance. Note that these are all trade-offs between reliability and performance, so we are free to tweak them as necessary to get the balance we want.

#### Publishing

The largest factor that impacts publishing is the level of durability used.

- Transient - Messages are not durable; if the broker restarts queued messages are lost. Messages may be paged out to disk if there are memory constraints, of course.
- Durable without transactions - Messages persist across broker restarts. However, publishers receive an acknowledgment from the broker before the message is persisted to disk so there is a window of time when messages can be lost without publishers knowing it.
- Durable with transactions - Use standard AMQP 0.9.1 channel transactions to ensure messages are published. According to RabbitMQ documentation this can decrease throughput by a factor of 250.
- Durable with Confirm extension - RabbitMQ extension to AMQP which sends an acknowledgment to the publisher only after the broker has successfully persisted the message in the same way consumers acknowledge messages.

### Consuming

When consuming, using a low pre-fetch value and acknowledging messages individually can impact message throughput.

### Both

- RabbitMQ performs best when queues are empty. If messages are stacking up in queues, more CPU time is required per message.
- Queues are are bound to a single core. If a queue's throughput is too slow, multiple queues should be used if possible.
- In a high availability setup, queues can be mirrored to multiple nodes in the cluster, which adds additional latency to publishing and consuming.

## The Numbers

### Single Node Cluster

All numbers were obtained using a single node RabbitMQ 3.6.15 cluster running on a Pine64 LTS which has a Cortex A-53 ARM processor and 2GB of RAM. The publisher used Pika as the client with a blocking (synchronous) connection.

### Transient Publishing

With no consumer attached to the queue, message throughput peaked at around 2,000 messages per second. Since memory was limited, messages regularly had to be paged out to the disk, during which time new messages were not accepted. After 10 minutes, the queue accumulated 675,255 messages. The process memory for the queue was approximately 55MB. Message bodies combined to be around 560MB.

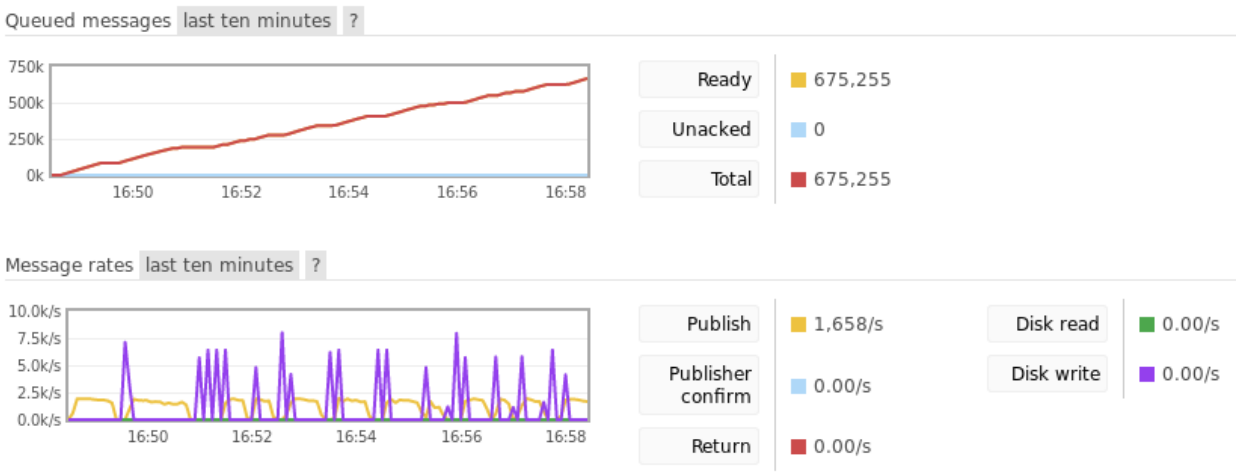


Fig. 2: This image shows the RabbitMQ management console after 10 minutes of publishing. Note the dips in publishing due to paging messages out to permanent storage.

#### Durable Publishing

With no consumer attached to the queue, message throughput peaked at around 1,500 messages per second. After around 10 minutes, the queue accumulated 676,214 messages. Again, the process memory was approximately 55MB and message bodies combined to be around 560MB. Interestingly, this performs as well as transient messages, but this is likely due to the extremely limited memory on the Pine64.

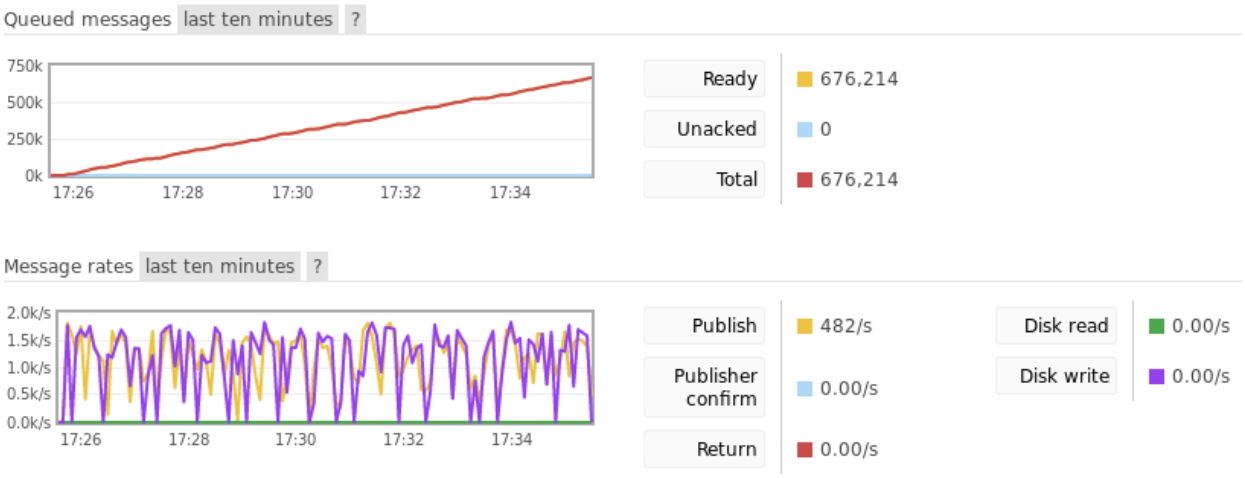


Fig. 3: This image shows the RabbitMQ management console after 10 minutes of publishing.

#### Three Node Cluster

All numbers were obtained using a three node RabbitMQ 3.6.15 cluster running on a Pine64 LTS, a Raspberry Pi 3B+, and a Minnowboard Turbot Dual-Core. The publisher used Pika as the client with a blocking (synchronous) connection. The queue was mirrored to all three nodes in the cluster.

#### Durable Publishing

With no consumers attached to the queue, message throughput peaked at around 800 messages per second. After around 10 minutes, the queue accumulated 383,428 messages.

## 1.3 Reliability

RabbitMQ deployments support clustering, high availability queues, and federation. Since Fedora does not want a single point of failure or regular downtime for updates, it would be best to use clustering with high availability queues.

### 1.3.1 Clustering

For complete details, consult the clustering documentation. A few things to note:

- All nodes in the cluster need to be running the same minor (same y version in a x.y.z release) so a major upgrade to Erlang requires downtime.

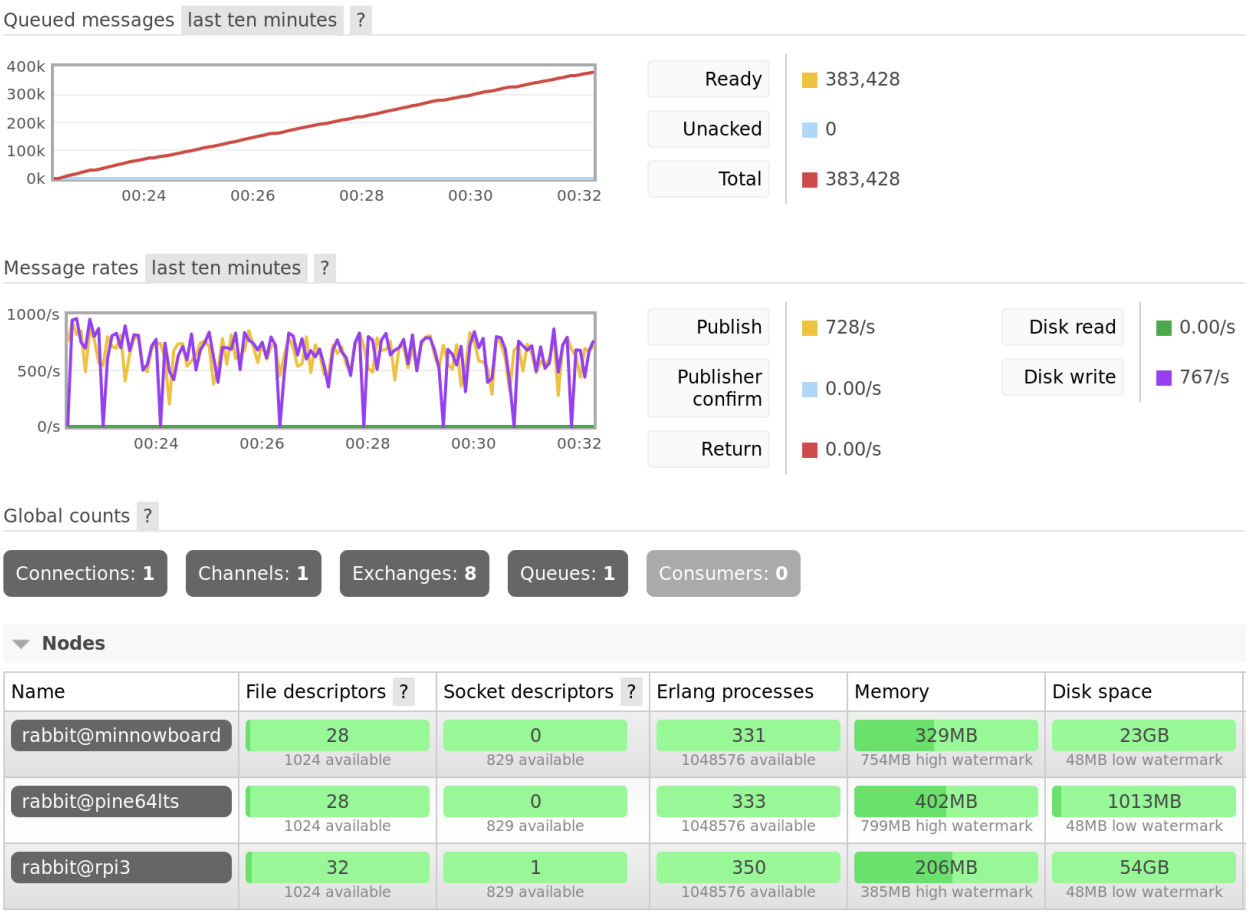


Fig. 4: Publishing to a queue mirrored across 3 nodes for 10 minutes.

- Clustering should only be used on a LAN. See the documentation on network partitions for details. Federation is designed for sharing across WANs.

### 1.3.2 High Availability Queues

RabbitMQ supports mirroring queues across nodes in a cluster in order to provide high availability queues.

Again, consult RabbitMQ documentation for the details, but a few highlights are:

- Mirroring can be applied to a subset or all queues
- Queues can be mirrored to all nodes in a cluster or just a few.

CHAPTER 2

# Project Documentation

## 2.1 Using a container

### 2.1.1 Building the container

You can create a Docker image suitable for running on OpenShift using the source-to-image tool:

```
sudo s2i build https://github.com/fedora-infra/fedmsg-migration-tools registry.
↪fedoraproject.org/f28/python3 fedmsg-migration-tools
```

The produced image is capable of running the 3 included commands:

- `amqp-to-zmq`: the AMQP (fedora-messaging) to ZeroMQ (fedmsg) bridge
- `zmq-to-amqp`: the ZeroMQ (fedmsg) to AMQP (fedora-messaging) bridge
- `verify-missing`: the verification service that checks that all messages are sent to both networks.

### 2.1.2 Running the container

You can choose which command the container will run by using the `APP_SCRIPT` environment variable. If you are running the container with Docker directly, you can use a command similar to:

```
sudo docker run -e APP_SCRIPT=./.s2i/amqp-to-zmq fedmsg-migration-tools
```

If you are running the container in OpenShift, set the environment variable in the deployment options.

### 2.1.3 Configuration

If you're running the container in OpenShift, you can use the `configmap` feature to set the `/etc/fedora-messaging/config.toml` file to your liking.

If you are running the container in Docker, you can use the `--volume` option of `docker run`.

## 2.2 Contributing

Thanks for considering contributing to fedmsg-migration-tools, we really appreciate it!

Quickstart:

1. Look for an existing issue about the bug or feature you're interested in. If you can't find an existing issue, create a new one.
2. Fork the repository on GitHub.
3. Fix the bug or add the feature, and then write one or more tests which show the bug is fixed or the feature works.
4. Submit a pull request and wait for a maintainer to review it.

More detailed guidelines to help ensure your submission goes smoothly are below.

---

**Note:** If you do not wish to use GitHub, please send patches to infrastructure@lists.fedoraproject.org.

---

### 2.2.1 Guidelines

#### Python Support

The fedmsg-migration-tools run on Python 3.4 or greater. This is automatically enforced by the continuous integration (CI) suite.

#### Code Style

We follow the PEP8 style guide for Python. This is automatically enforced by the CI suite.

We are using *Black <https://github.com/ambv/black>* to automatically format the source code. It is also checked in CI. The Black webpage contains instructions to configure your editor to run it on the files you edit.

#### Tests

The test suites can be run using tox by simply running `tox` from the repository root. All code must have test coverage or be explicitly marked as not covered using the `# no-qa` comment. This should only be done if there is a good reason to not write tests.

Your pull request should contain tests for your new feature or bug fix. If you're not certain how to write tests, we will be happy to help you.

#### Release notes

To add entries to the release notes, create a file in the `news` directory with the `source.type` name format, where `type` is one of:

- `feature`: for new features
- `bug`: for bug fixes
- `api`: for API changes
- `dev`: for development-related changes

- `author`: for contributor names
- `other`: for other changes

And where the `source` part of the filename is:

- `42` when the change is described in issue `42`
- `PR42` when the change has been implemented in pull request `42`, and there is no associated issue
- `Cabcdef` when the change has been implemented in changeset `abcdef`, and there is no associated issue or pull request.
- `username` for contributors (`author` extention). It should be the username part of their commits' email address.

A preview of the release notes can be generated with `towncrier --draft`.

### Licensing

Your commit messages must include a Signed-off-by tag with your name and e-mail address, indicating that you agree to the Developer Certificate of Origin version 1.1:

```
Developer Certificate of Origin
Version 1.1

Copyright (C) 2004, 2006 The Linux Foundation and its contributors.
1 Letterman Drive
Suite D4700
San Francisco, CA, 94129

Everyone is permitted to copy and distribute verbatim copies of this
license document, but changing it is not allowed.


Developer's Certificate of Origin 1.1

By making a contribution to this project, I certify that:

(a) The contribution was created in whole or in part by me and I
    have the right to submit it under the open source license
    indicated in the file; or

(b) The contribution is based upon previous work that, to the best
    of my knowledge, is covered under an appropriate open source
    license and I have the right under that license to submit that
    work with modifications, whether created in whole or in part
    by me, under the same open source license (unless I am
    permitted to submit under a different license), as indicated
    in the file; or

(c) The contribution was provided directly to me by some other
    person who certified (a), (b) or (c) and I have not modified
    it.

(d) I understand and agree that this project and the contribution
    are public and that a record of the contribution (including all
    personal information I submit with it, including my sign-off) is
    maintained indefinitely and may be redistributed consistent with
    this project or the open source license(s) involved.
```

Use `git commit -s` to add the Signed-off-by tag.

#### Releasing

When cutting a new release, follow these steps:

- update the version in `fedmsg_migration_tools/__init__.py`
- generate the changelog by running `towncrier`
- change the `Development Status` classifier in `setup.py` if necessary
- commit the changes
- tag the commit
- push to GitHub
- generate a tarball and push to PyPI with the commands:

```
python setup.py sdist bdist_wheel
twine upload -s dist/*
```

## 2.3 Release Notes

### 2.3.1 0.1.2 (2018-09-13)

#### Features

- Add Systemd service files and an RPM spec file. (PR#10)
- Add validation and signing for bridges. (PR#2)

#### Bug Fixes

- Fix up verify_missing config. (PR#2)
- Wrap messages sent to ZMQ with fedmsg format. (PR#9)

#### Development Changes

- Use towncrier to generate the release notes, and check our dependencies' licenses. (PR#10)
- Use Mergify. (PR#7)
- Use Black. (PR#8)

#### Contributors

Many thanks to the contributors of bug reports, pull requests, and pull request reviews for this release:

- Aurélien Bompard
- Jeremy Cline