**Introduction**

IoTa is an IoT message handling platform implemented as a set of microservices. Most services are implemented in Python, but one is JavaScript. Postgres is the database, and the services communicate via RabbitMQ.

See <https://github.com/DPIclimate/broker> for an overview of the system.

A component view of IoTa is given at the end of this document.

This project is about improving the implementation and deployment of IoTa based upon the first 12 months of running the system.

**Suggested method of development**

The IoTa github repo (link above) should be forked and all development take place on the fork. If the fork is kept up to date during the project, then pulling the changes from there to the IoTa repo can be done at the end of the project, or even earlier if some changes are finalised before the end of the project.

Another team is developing a timeseries database for IoTa, and it would be worth communicating when them throughout the project in an attempt to ensure the changes they are making fit with the changes in this project.

**Possible improvements**

The following list of improvements are not all mandatory for the project to be considered a success. They are listed in order of priority. Even if only the two mandatory improvements are implemented the project will be considered successful.

**Kubernetes (mandatory)**

IoTa containers are run using docker compose. This is inflexible because it is difficult or impossible to rebuild and restart single containers and implement service replication for performance or fault tolerance.

It would be good to have IoTa running under Kubernetes rather than docker compose.

Additionally, demonstrating how one or more services can be replicated would be good. The logical mapper is the easiest service to demonstrate because it has no interactions with external systems.

*Explain the problem with RabbitMQ fanout exchanges for replicated services and how a round-robin exchange will be required to distribute the messages to multiple instances of a service.*

**Refactor MQTT front end processor (mandatory)**

The Wombat and YDOC front end processors receive their messages via MQTT topics, and it is reasonable to expect there could be more MQTT-based services in future.

To avoid creating a new service each time there should be a single (replicable) MQTT front end processor that dynamically discovers and executes a message parser plugin based upon the topic an incoming message is received from.

These dynamically discovered modules should:

1. Provide one or more MQTT topic patterns, which can include MQTT wildcards, that they will handle messages for.
2. Accept a message in text format (to handle cases where the message is not JSON).
3. Return a list of messages in IoTa’s internal JSON format generated from the given input message.
4. It might be good to cache them in-process so they are not loaded anew for every message.

Before going too far with the implementation of this one I’d like to review the design and then see a toy implementation of the agreed design that can locate and load a couple of plugins, showing their topic patterns and their ability to transform a canned message.

*If multiple instances of the refactored MQTT front end processor are running, how will they ensure each received message is only processed by one of them? This might be handled by a single receiving service pushing the messages it gets onto a round-robin exchange and using the MQTT topic as a routing tag? Would that very front single service still get bogged down or will it be quick enough because it is only pushing the messages to a new exchange and not doing any real processing?*

**Metrics (highly desirable)**

Add Prometheus metrics to IoTa, and a Grafana dashboard.

In addition to the metrics provided by the Python Prometheus client, obvious metrics are:

* Number of messages received.
* Number of messages sent.
* Number of messages where processing failed.

The number of messages received/sent over a time period (eg per second/minute/hour) would be good too.

The Prometheus server and Grafana must be integrated into IoTa set of containers, and the necessary nginx reverse proxy configuration defined.

**Improvements to the management application (highly desirable)**

IoTa has a webapp for device management. There are various improvements that can be made to this, not limited to:

Filter the list of physical devices to only include those with no current mapping to a logical device.

Use checkboxes to select multiple devices on the physical and logical devices list rather than the current multiple selection mechanism.

**Refactor front end processors so they can be run stand-alone (desirable)**

It would be extremely useful to be able to run the front end processes in a stand-alone mode where they take their messages from one or more file(s) or from the raw\_messages or physical\_timerseries table. This would allow easier re-processing of messages. The process would still run in a container so it has access to the database, but when run with certain arguments would know to read a set of messages from the file(s) or database and send them on to the mid level process(es) in the usual way.

If the messages are read from file(s) or the raw\_messages table they need to be transformed into the standard internal format as usual. Messages from the physical\_timeseries table are already in the standard internal format so could even be read directly by the logical mapper rather than a front-end.

This change requires the logical mapper to look up the physical -> logical device mapping that was in effect at the timestamp of the message, rather than assuming the current mapping as it does now. In turn, that change could benefit from some caching of the mappings so the mapping table wasn’t hit for every message, with the postgres notify/listen mechanism used to keep the cache up to date with changes to the mapping table.

**Fix RabbitMQ async code (desirable)**

IoTa has some classes used for sending and receiving RabbitMQ messages. These classes are based upon the Pika client, using async programming. These classes are buggy and can cause the delivery services to stop working.

**Improve the unit tests (desirable)**

The DAO unit tests are extremely slow at present due to the retry on error mechanism. Many unit tests are forcing and expecting failures, so this retry mechanism is a problem there. A technique called ‘monkey patching’ may be a work-around for this.

Try to find a way to disable the retry mechanism when running unit tests where failures are expected.

Other unit test improvements could be:

1. Reworking the RESTAPI unit tests to use FastAPI unit testing idioms.
2. Reworking the DAO unit tests to use the PyTest framework.

