Event-Driven Messaging for Offline Data Quality Monitoring at ATLAS

Peter Onyisi, on behalf of the ATLAS Collaboration
University of Texas at Austin, Austin, TX 78712, United States
E-mail: ponyisi@utexas.edu

Abstract. During LHC Run 1, the information flow through the offline data quality monitoring in ATLAS relied heavily on chains of processes polling each other’s outputs for handshaking purposes. This resulted in a fragile architecture with many possible points of failure and an inability to monitor the overall state of the distributed system. We report on the status of a project undertaken during the LHC shutdown to replace the ad hoc synchronization methods with a uniform message queue system. This enables the use of standard protocols to connect processes on multiple hosts; reliable transmission of messages between possibly unreliable programs; easy monitoring of the information flow; and the removal of inefficient polling-based communication.

1. Introduction
The ATLAS detector [1] offline data quality (DQ) system [2, 3] consists of a heterogeneous set of applications running on multiple nodes at CERN. These include management of the histograms produced in reconstruction jobs, automatic analysis of produced histograms and detector conditions, visualization and notification applications, and so on. These programs are written in C++ and Python and are scheduled in various ways: as part of of Tier-0 reconstruction jobs, as part of web servers, as standalone daemons, or as short-lived tasks started by cron. In addition information extracted by the DQ system (histograms, results of automatic checks, run summaries, etc.) are made available to detector subsystems and others through web service interfaces.

Since these applications form a distributed system, loose coupling between components is desirable. In Run 1 communication was achieved primarily through synchronizing based on file or database polling. The communication between each pair of applications was largely ad hoc and reuse of communication code was rare. There were also some inefficient design choices made in order to decouple processes, such as continuously polling large directory trees for updates.

For Run 2 it was decided to implement a “message bus” connecting all offline DQ applications. In this model, applications communicate by exchanging messages using a message broker. Applications publish messages to the bus as necessary, and subscribe to the subset of messages they are interested in receiving.

2. ATLAS Offline Data Quality Infrastructure
The ATLAS offline DQ infrastructure includes two major loci of action: the Tier-0 reconstruction farm, which reconstructs raw data and produces and analyzes monitoring histograms, and the
central DQ server, where almost all other tasks are performed. In the Tier-0 farm, applications are scripted by Python control scripts. On the central server, most of the software is written in Python, executing in a variety of environments (as web server applications in two different frameworks, as standalone continuously-running applications managed by Supervisor, and as scripts started by cron).

The applications need to communicate with each other. For example, the histogram display service needs to maintain an index of which runs have histograms available, their timestamps and reconstruction software configurations, and whether the histograms are final or the result of intermediate fast monitoring. The Tier-0 jobs directly place the histogram files in the target directories in order to avoid race conditions and the locks needed to avoid them; to avoid tightly coupling the Tier-0 jobs and the histogram display service, this is done asynchronously to updates of the index. The index updates were implemented in Run 1 by polling the relevant directory tree structure which contained nearly 800,000 files, too big for the standard Linux file notification APIs to be used\(^1\). The polling results in a significant continuous I/O load on the relevant server (of order 10\% iowait). The index only needs to be updated when new histograms are uploaded from the Tier-0 reconstruction and analysis, which is a few times per minute at most. By replacing the polling by a notification system, a significant decrease in I/O requirements can be achieved.

3. Messaging Bus Concepts
A messaging bus is a way to achieve loose coupling between applications in a distributed system by exchanging messages via a logical bus. The bus transmits messages between the connected applications; the listening applications decide what messages they will listen to. Software participating in the system only needs to be aware of how to connect to the bus and how to publish or consume appropriate messages. Applications that publish information to the bus do not need to know about which applications are consuming that information, and similarly the applications consuming data do not need to know about those publishing it.

There are two typical models of communication though a message broker: the “queue” and the “topic.” A queue will deliver a message to one consumer; if no consumer is available, it implements a first-in-first-out queue of messages that will be delivered when consumers come online. This provides a way to ensure transfer of information between programs that may not be active at the same time, and to implement resilient load balancing. In contrast, a topic delivers a message to all consumers connected at a given time; in the pure model, any consumer not connected when the message is published will not receive the message. Figure 1 shows a graphical depiction of these flows. Modern messaging queue software provides the ability to create more complex topologies from these building blocks (such as connecting a queue to the output of a topic to create a “durable topic” which stores messages for clients while they are disconnected).

4. Messaging Broker Configuration
There are many messaging brokers on the market. Two of the best supported are Apache’s ActiveMQ and Pivotal Software’s RabbitMQ. ActiveMQ is supported by CERN IT which runs and manages central instances. Both provide support for the standard STOMP protocol, which is a text-based wire protocol for sending and receiving messages from a broker. A Python STOMP client library (stomp.py) is included in the LCG software distribution and so is available on every machine relevant to offline ATLAS DQ without additional intervention, making this protocol a natural choice.

\(^1\) in addition, neither inotify nor fanotify handle recursive monitoring of subdirectory trees when they can be created or deleted by external programs, as in this use case.
Both RabbitMQ (as a standalone server, managed by the DQ group) and ActiveMQ (managed by CERN IT) were evaluated for ATLAS DQ. RabbitMQ was found to be simple to install and configure. Integration with CERN’s single sign on authentication system could be achieved for RabbitMQ by writing an Erlang plugin, although the lack of relevant language expertise in the group did not favor this solution. ActiveMQ permits very complex message routing configurations through its “Virtual Destination” mechanism, which for example can deliver the functionality of a queue listening to a topic (so a consumer can receive messages generated while it was offline, while not interfering with any other consumers subscribed directly to the topic). In the end the support of the servers and infrastructure provided by CERN IT strongly motivated the selection of ActiveMQ.

Running on centrally managed instances imposes certain restrictions. In particular creating topics and queues, or altering their properties and users’ access privileges, requires coordination with CERN IT. As a result only a minimal system has been configured so far, which does not take full advantage of features such as virtual destinations. As appropriate use cases are identified more ActiveMQ features will be used.

Three message broker servers are used in a load-sharing configuration. The brokers do not communicate with each other, so it is necessary for consumers to subscribe to all three brokers to be sure of receiving all messages. For Python applications, a small utility function was written to simplify this process. A publisher may communicate with any of the brokers (the choice is generally made by round-robin DNS).

The following queues and topics were created:

- **atlas.dqm.panic** (queue): Consumers will trigger notification of administrators when a message is received on this queue. Messages can be published from any source.
- **atlas.dqm.logging** (queue): Non-urgent logging information to be archived centrally is published on this queue. A single consumer will handle the archiving.
- **atlas.dqm.progress** (topic): General information on ATLAS DQ and run status is published on this topic. This includes run start and end messages and the status of Tier-0 data reconstruction and histogram processing.

Because the **atlas.dqm.progress** topic carries traffic of different types, the “selector” mechanism is used to ensure that consumers only receive messages that they are interested in. Publishers can include arbitrary headers that describe the type of message that is being sent. For ATLAS DQ, messages are required to have **MsgClass** set to **DQ** and to have **MsgType** set to a descriptive string (e.g. **RunStart**, **RunEnd**). When subscribing to the topic, a consumer may specify a SQL-like filter specification and will only receive messages satisfying the filter.

The body of a DQ message is required to be in JSON. The actual content depends on the type of message specified by **MsgType**. An example message, indicating the completion of a histogram
Figure 2. Example message, indicating that the histogram production from Tier-0 reconstruction is complete for the express reconstruction of the minimum bias stream of run 264525. The payload includes various items of histogram automatic check configuration information as well as the run and reconstruction setup.

creation job for a specific run and data stream, is shown in figure 2. On receipt of this message, an indexer will add this information to its database and delete the temporary histograms from the intermediate updates for that run and stream.

5. Operational Experience and Current Status
The most problematic aspect of the use of ActiveMQ has been the authentication/authorization system. This concern is specific to the central CERN deployment; in general site-specific Java authentication plugins can be written for ActiveMQ. In this instance either full user certificates (not proxy certificates) or application-specific passwords can be used. Application-specific passwords are most commonly used in practice. To make it easier for ATLAS collaborators to use the messaging bus system without having to get their own password, the relevant file is stored in CERN AFS and protected with relevant ACLs.

Python client libraries have been written to make publishing and subscribing to messages as simple as possible, in order to reduce barriers to adoption. For example, a panic message can be sent using two lines of Python code, without needing to know any details of the messaging bus implementation (or indeed any passwords).

The following services are now implemented:
- Run Start and Run End notifications, created by a program that polls online databases.
• Tier-0 histogram processing status. In this case the messages are sent by the reconstruction and histogram analysis jobs themselves, which may be running on any node in the Tier-0 farm. This information is consumed by a number of clients which handle bookkeeping tasks, and which previously polled directories for files created by Tier-0 jobs.
• a panic notification program (currently sends emails to responsible parties).

The following services are anticipated:
• Central log collection and analysis using software such as logstash and elasticsearch
• A status board system using a Redis key/value store to provide snapshots of system status
• Using the durability of messages stored in queues to ensure that applications that are temporarily unable to run (due to server outages, for example) execute the tasks they missed when they restart

It is also expected that additional clients, such as luminosity monitoring, will subscribe to the messaging bus. The system is being configured with such use cases in mind (for example, the MsgClass header which allows different domains to keep their message traffic separate).

6. References
[1] ATLAS Collaboration (ATLAS Collaboration) 2008 JINST 3 S08003
[2] Adelman J et al. 2010 J. Phys. Conf. Ser. 219 042018
[3] Golling T, Hayward H, Onyisi P, Stelzer H and Waller P 2012 Eur. Phys. J. C 72 1960 (Preprint arXiv:1110.6119)