

CORRIDOR: Cobalt Realtime RabbitMQ Messaging Stream

Barry Greengus, Benjamin Walters
Student Research Participation Program
University of Illinois at Chicago, Illinois Institute of Technology
Argonne National Laboratory
Lemont, IL

Prepared in partial fulfillment of the requirements of the Student Research Participation Program under the direction of William Allcock in the Leadership Computing Facility at Argonne National Laboratory.

Participants:

Research Advisor:

Corridor: Cobalt Realtime RabbitMQ Messaging Stream. BARRY GREENGUS (University of Illinois at Chicago, Chicago, IL 60607), BENJAMIN WALTERS (Illinois Institute of Technology, Chicago, IL 60616) WILLIAM ALLCOCK (Argonne National Laboratory, Lemont, IL 60439)

In order to analyze performance and better appropriate usage of the Argonne Leadership Computing Facility's (ALCF) Blue Gene/Q supercomputers, collection of job performance data and metadata is crucial. Due to the complexity and parallelism of Cobalt - ALCF's BG/Q job scheduling program - it is difficult to have a good picture of what is happening in the system. Currently, the only way to monitor the system remotely is to periodically request specific information from Cobalt. The aim of this work is to implement and utilize an interface for providing accurate data about the system's activity in real-time. An interface was designed for Cobalt that enables asynchronous publishing of AMQP (Advanced Message Queuing Protocol) messages to a remote AMQP broker. This broker allows many clients to subscribe to the Cobalt's message stream. Corridor, a client of this stream, was created to provide a sanitized subset stream to the public. It handles dropped messages from Cobalt, and keeps its own copy of non-sensitive information from Cobalt's job queue. Corridor also provides an interface for public clients to request its state. This stream of real-time updates could replace the periodic polling functionality of the current monitoring system. The messaging framework created is language and platform independent, easily expandable to include more streams and events, and allows many internal and external clients. Therefore, this framework can be used to provide a constant, real-time picture of the ALCF's Blue Gene/Q systems to both system administrators and outside clients.

Research Category: Computer Science

Schools Authors Attend: University of Illinois at Chicago (Barry Greengus)

Illinois Institute of Technology (Benjamin Walters)

DOE National Laboratory Attended: Argonne National Laboratory

Mentor's Name: William Allcock

Phone: (630) 252-7573

e-mail Address: allcock@anl.gov

Barry Greengus: barry.LG.62@gmail.com

Benjamin Walters: benjaminwalters20@gmail.com

Biographies

Barry Greengus is a first year student in Computer Science at the University of Illinois at Chicago. He has taken multiple programming classes, and has experience writing small games.

Ben Walters is a first year Computer Science student at Illinois Institute of Technology. He works as an undergraduate research assistant in the Data-Intensive Distributed Systems Laboratory at IIT.

Introduction

The Argonne Leadership Computing Facility (ALCF) is home to Mira, a 48 rack IBM Blue Gene /Q supercomputer. With 786,432 processors and 768 terabytes of memory, Mira is ranked the 5th fastest supercomputer in the world (“Mira - BlueGene/Q”, 2014) at around 10 petaflops. Like other machines of its class, a batch scheduler manages Mira's workload. ALCF develops its own scheduler, Cobalt, which invites research into software architecture, user interfaces, and job scheduling (“Cobalt: Component Based”, n.d.). Cobalt is composed of components which communicate via XML-RPC. XML-RPC, like all remote procedure call implementations, is client-server. Likewise, the XML packaging of RPC in XML-RPC comes with significant data overhead. In Corridor, we explore the use of AMQP as a potential replacement for XML-RPC by using it to address the issue of sharing Cobalt state with many clients.

For users and system administrators, knowing what Cobalt and Mira are doing is extremely useful. Currently, the only way to see the queue outside of logging into the system, is to check the ALCF status website (<http://status.alcf.anl.gov>). The site provides a javascript browser client that polls a web server called Gronkulator for information about Mira. Gronkulator maintains information about Mira by polling gronkd, a daemon running on one of the system's login nodes. Gronkd's job is to poll Cobalt for very specific information that should be available to the public (Figure 1).

This system of requesting information has downsides. If a user or administrator wants information about the system, they have to go and request the exact information they want. This leaves the client “in the dark” about the system until he/she requests information. Also, catastrophic changes could occur and the client would not be the wiser. Even if the client did request information, they might not request the relevant information. Additionally, the paradigm of requesting information creates the possibility of over-requesting. For example, the Gronkulator polls every ten seconds, but if nothing actually changes on the machine for minutes at a time, those requests are fruitless.

The goal of this work is to create an interface for Cobalt that will enable it to send messages with Advanced Message Queuing Protocol (AMQP). The interface will have to be able to send the messages when events internal to Cobalt occur without hampering the scheduler’s main task, i.e. scheduling jobs. Another aim of this work is to use this interface to provide a public stream of queue updates from Cobalt. This will be achieved by creating a “middle-man” program that will be a client of Cobalt’s stream. The client will then publish selected messages to a second stream that will be available to the public.

Materials (systems) and Methods

To create a message stream intended for many clients, a standard must be decided upon to allow interconnectivity. While there are many other message queuing protocols, ZeroMQ and STOMP, for instance, AMQP was chosen because it is the most popularly adopted standard and has all of the features needed: queuing, language independence, publish/subscribe capability, reliability, and security (“AMQP is the Internet”, 2014). Queueing, reliability, and security are necessary to help prevent missing messages and security flaws. Being language independent adds flexibility because no matter

what programming languages clients want to use, they can listen to the stream. This is opposed to only clients using Python (the language Cobalt is written in) being able to listen. AMQP's support for publish/subscribe routing is needed because it allows for many subscribers getting the same messages from one publisher. The publish/subscribe model is also extended to include different topics that client can subscribe (Figure 2). It also allows for the philosophy of "fire and forget" because publishers are only responsible for delivery to an AMQP broker. The broker is then responsible for getting the message to all subscribers. To lessen the overhead on Cobalt, firing off the message and forgetting about it is better than waiting until all clients receive the message. This system also allows separating messages by topics, which gives clients the power to choose which topics they follow.

An important part of AMQP is the broker program, which manages the delivery of messages from sender to receiver. Although many implementations are available, they all must support messaging features defined by the protocol. RabbitMQ was chosen, mainly because for prototype purposes it was the easiest to learn to use. Written in Erlang, it runs on all major operating systems, supports a large number of developer platforms, is easily scalable on clusters, and is both open source and commercially supported ("What is RabbitMQ?", 2014). Because Cobalt is written in Python, a Python RabbitMQ client was needed to add the message functionality. There are many of these for RabbitMQ, but Pika was chosen also because it seemed to be the easiest to pick up and learn. This project is only a prototype, so which broker and Python client should be used in production still needs to be considered.

Changes needed to be made to Cobalt, so a way to test those changes was needed. However, Cobalt is a job scheduler for Blue Gene/Qs, and ALCF didn't have extra supercomputers lying around for interns to test schedulers. Instead, the initial testing and development was done by running our fork of Cobalt on a Blue Gene simulator called Brooklyn running on a Macbook Pro. Brooklyn pretends to

run jobs, but actually just sleeps for the allotted time. It is missing many other features of the Blue Gene/Q, but none that were necessary for the project. Because our system had an internal and an external feed, two AMQP brokers were required. In the test environment, one broker ran on the local machine, and one ran on a VirtualBox virtual machine running Ubuntu. Once the project was mostly completed, it was tested on a partition on Vesta, ALCF's Blue Gene/Q testbed.

Results

After some deliberation and meeting with network security administrators, a system architecture was devised. (Figure 3). Parts of Cobalt run on the service node that controls the supercomputers, and thus shouldn't interact with anything outside of the ALCF internal network. So, it sends its AMQP messages to a RabbitMQ broker running on a virtual machine called Falcon. Also running on Falcon is the middle-man program, Corridor. Both Corridor and any computers in the ALCF internal network can connect to the RabbitMQ broker. Corridor listens to those messages, and keeps its own copy of Cobalt's state. If a message is dropped, Corridor makes an XML-RPC to Cobalt to request its state the same way that Gronkd currently does. Also, if a client sends a request for state through a RabbitMQ-RPC request (Figure 4), Corridor has its own copy to send back to the client. It also subsets the messages coming out of Cobalt into ones with only public-safe information and then sends them as AMQP messages to another RabbitMQ broker. This broker is running on a different virtual machine called RedWing that lives outside of ALCF's internal network. Public clients would both listen to the update stream through the broker on Redwing, and request a copy of initial state from Corridor through it.

When the project was ready to be tested on the real machine instead of the simulator on a Macbook, it ran as a secondary Cobalt instance on a partition of Vesta. The virtual machines, Falcon and Redwing, were set up and running, and heavy loads of jobs and requests were thrown at the system. At its peak, there were over 500 jobs between the queue and those actually running. It performed remarkably well, though at one point, the RabbitMQ broker itself became overloaded with messages and raised errors, and minor issues with exception handling could be improved.

Creating a Production Ready System

Although this system was only intended to be a prototype of an AMQP messaging stream in Cobalt, it could be made production ready with minimal work. Several instrumentation calls inside of Cobalt would need to be moved to better capture when events are actually being triggered. Also, more queue events would need to be instrumented with AMQP messages. Some other work could be done inside of MQWriter and Corridor to further abstract some of the AMQP functionality. Much of the code for connecting to brokers and sending messages could be pulled into its own module so that it could be shared between the two.

In addition to code modifications, several other things need to be considered before the system could be ready for production. Many AMQP implementations would need to be investigated in order to choose one that satisfies many requirements. Some of the requirements may include scalability, security, documentation, reliability, and support. Although many implementations claim to have several of these features, they would need to be stress tested extensively before being put into production on something as large as Mira.

Security needs to be a major consideration not only in the AMQP implementation, but also in the system as a whole. The prototype allows anybody inside of the ALCF internal network to subscribe to

any message stream desired. In reality, different internal streams could exist with different people allowed to subscribe. Ideally, the chosen AMQP implementation should support using some sort of authentication system that ALCF already has in place.

Discussion and Conclusions

Once there is a system ready for production, there are large number of things that can be added. First of all, the public stream created in this project has almost all pieces of information that are displayed by the Gronkulator. Thus, only a few pieces of information would need to be added for the public stream to feed the Gronkulator. The Gronkulator web server would need to be modified to listen for AMQP messages, and to maintain the state of Cobalt's queue, using the messages it receives to modify the state it keeps. Much or all of the functionality used to display the information, however, would remain the same. Many of the same changes could also be applied to the javascript browser client so that individual browsers are actually being pushed live updates.

Currently, Cobalt only sends a small group of messages from one of its seven components, 'queue-manager'. If more types of messages were added to Cobalt, and from multiple components, Corridor would need to be modified to handle state of multiple components and XML-RPC calls to all of those components. Additional internal clients could be added, each to do different things. For example, a message send could be registered with the sys logger, and one internal client could collect every log message. Another could send out personal updates to a smartphone app giving users personal updates whenever something happens with a job of theirs.

This project was inspired by a much larger scoped plan of changing the messaging protocol that is used inside of Cobalt. Cobalt's components currently communicate using XML-RPC. However, using AMQP would open up a world of possibilities. For example, if you used an rpc-style request over

AMQP, in addition to the caller listening for the result, a logging program could listen for the result of that call. That level of real time logging would allow for a much greater awareness of what is happening in the system.

Acknowledgements

This project was made possible by Argonne National Laboratory, Leadership Computing Division, and the U.S. Department of Energy. Special thanks to Bill Allcock, the advisor of the students on the project, William Scullin, Paul Rich, and Matt Kemp for all of their help and willingness to offer advice and answer questions.

References

What is RabbitMQ? (2014). Retrieved August 1, 2014, from <https://www.rabbitmq.com>

AMQP is the Internet protocol for Business Messaging (2014). Retrieved July 30, 2014, from <http://www.amqp.org/about/what>

Mira - BlueGene/Q, Power BQC 16C 1.60GHz, Custom. (2014, July 1). Top500.org. Retrieved July 31, 2014, from <http://top500.org/system/177718>

Cobalt: Component Based Lightweight Toolkit (n.d) Retrieved July 28, 2014, from <http://trac.mcs.anl.gov/projects/cobalt>

Figures

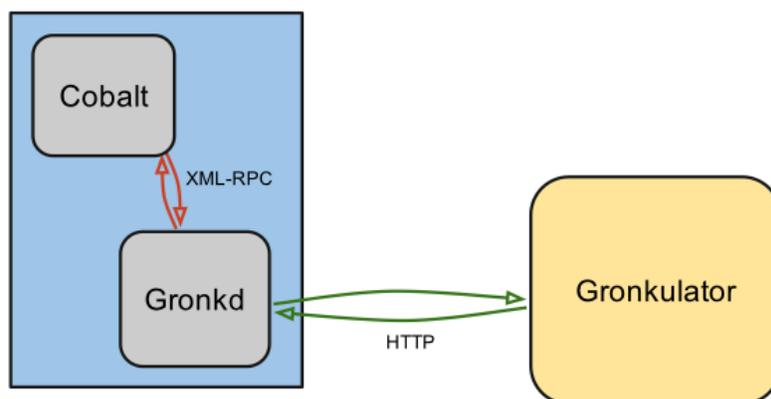


Figure 1. Shows the polling system previously used to monitor Cobalt

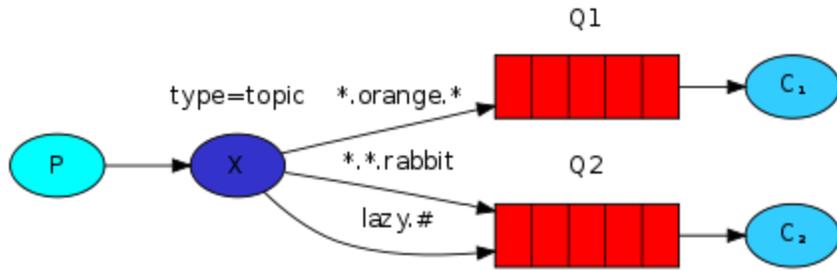


Figure 2. Shows a server publishing to a RabbitMQ exchange and subscribers listening to different topics

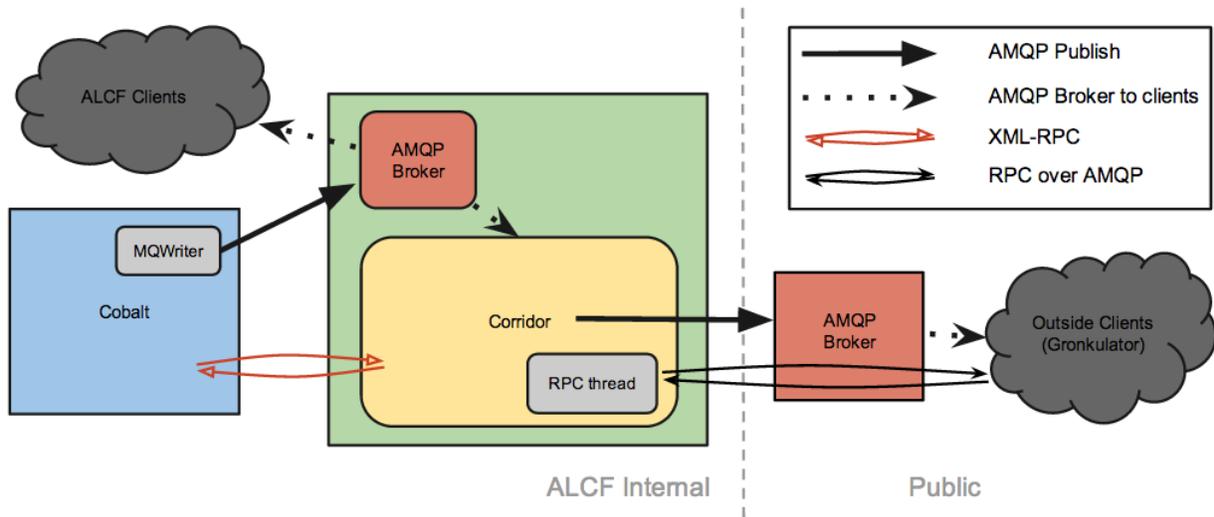


Figure 3. Shows the new system, which includes two AMQP publisher/subscriber relationships and two different request/response interfaces

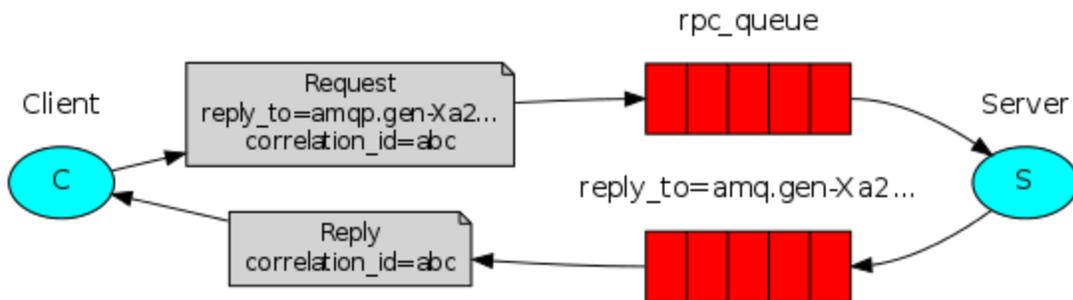


Figure 4. Shows a RabbitMQ client making a Remote Procedure Call over AMQP