GATD: A Robust, Extensible, Versatile Swarm Dataplane

Pat Pannuto, Brad Campbell, Prabal Dutta
Electrical Engineering and Computer Science Department
University of Michigan
Ann Arbor, MI 48109
{ppannuto,bradjc,prabal}@umich.edu

Abstract

We propose Get All The Data (GATD), a data collection and dissemination system for the Swarm. GATD offers a flexible architecture to connect arbitrary producers of data and consumers of events. Too many sensor networks are fragile, vertical silos, with a series of one-off handlers written to shuttle data, manipulate it, process it, and present it. Instead of being mired in details and rigid schemas, we argue that sensor network deployment should be simple. The key in GATD's design is the observation that there are common patterns to how disparate sensor network applications handle and process their generated data.

To take advantage of these similarities we present a system comprised of common modules that different applications can leverage. To join GATD, new sensors simply send raw data. GATD will buffer this raw data indefinitely until an application specific *formatter* is written to map the raw data to key-value pairs. These streams can be combined, processed, graphed, stored, or otherwise manipulated by a standard set of transforms or new custom drivers. With this architecture, we argue that GATD provides a robust, extensible, and versatile Swarm dataplane.

1 Introduction

Efficient incoming data stream management, storage, processing, fusing, and publishing is critical for handling the influx of data from new swarms of sensors. However, building a data management system for each application or deployment is unmaintainable and, we argue, unnecessary. From our experience, the process of managing and storing data for a variety of different applications follows a similar pattern. Many common operations, for instance converting raw sensor values, streaming data in real-time to a display, and storing data, can be shared across different applications.

GATD is a strawman system that attempts to exploit these patterns and divide common operations into reusable modules to explore the viability of such a general framework. It is based on three design tenets:

- **Robust.** All data must be accepted. If it is unknown it will be archived and processed later.
- **Extensible.** New applications will arise. The system must be extensible with minimal to no downtime.
- **Versatile.** Real-time streaming, long-term archival, and anything in-between must be supported.

2 System Architecture

Our system architecture is divided into a number of purpose-centric modules. These modules communicate using a message broker model, similar to SEDA [2], which provides significant decoupling and allows them to run in a distributed manner. Using message queues provides a robust layer to connect the modules and dynamic load balancing between different versions or copies of modules.

2.1 Ideal Model: Producers / Consumers

Conceptually, we model GATD as a series of producer and/or consumer modules interconnected with queues of infinite depth. Each queue may have an arbitrary number of independent readers, all of whom receive copies of each message. The boundless queue abstraction is important for presenting a simple, unified interface: GATD transparently morphs into a data archival service if a reader attaches itself to a queue but never actually consumes any messages.

These magic structures do not exist today and the modules and queues are forced to take on a more limited capacity in our GATD implementation. It is easy to map pieces such as the archive module as an implementation detail that facilitates the infinite-depth queue abstraction.

2.2 Implementation: GATD Modules

The basic breakdown of modules is shown in Figure 1. Many instantiations of each type can exist, to support, for instance, different communication protocols.

Receiver: A receiver is a generic interface for getting data into GATD. The purpose of receivers is to abstract away the transport medium used to communicate with GATD. Receivers do no processing on data, they simply timestamp its receipt and pass it on as an opaque block. Currently we have raw UDP and TCP receivers, and a HTTP module that accepts POST requests, though others could be easily added.

Queryer: The queryer is a compatibility receiver-like module to convert a polled data source to a push stream. A queryer can be configured to emit packets at a constant frequency or every time the polled data changes.

Formatter: GATD formatters are responsible for canonicalizing raw data from a receiver into a series of key-value pairs. Formatters have great leeway in their action. The simplest formatters are near no-ops, copying to their outputs what they received as inputs. Others may convert raw data to standard units or unpack a data struct. In some cases, formatters subscribe to multiple producers and aggregate them.

In lieu of a magical queue, when a formatter is not available for incoming data it is stored in an archive database. This database is replayed when a formatter comes online than can handle the data. Once data is formatted it is stored in a database and pushed to any subscribed streaming queues.

Streamer: Streamers are the output analog of receivers. They are initiated in response to a stream subscription sent to GATD and send all formatted data blobs that match a given query. Currently there exists a streamer for sending data to

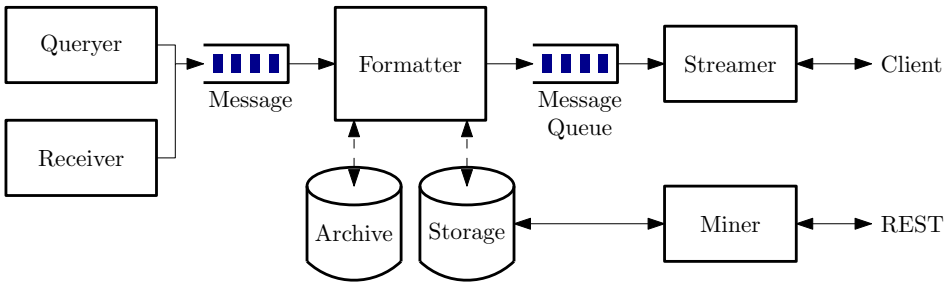

Figure 1: *Architecture of GATD*—Queryers and receivers push messages into the system. These messages are timestamped upon receipt and buffered in a queue indefinitely until retrieved by a formatter. Formatted messages are written out to long-term storage for future queries. Formatters also write out messages in real-time after formatting to any services that have subscribed to the formatter. A subscription creates another queue of infinite depth, allowing the formatter to emit messages independent of the rate that a subscriber is able to consume them.

web browsers and one that uses a raw TCP connection for streaming to arbitrary clients.

Miner: Miner modules are used as an optimized interface for historical data queries, allowing direct database query access. From the model perspective, a miner is a transient block that consumes multiple messages from multiple streams and produces only one message in an output queue.

In practice, the miner interface is a band-aid that solves the absence of a GATD control plane that could dynamically create a purely computational producer/consumer block on the fly to satisfy a subscription request for a currently nonexistent queue. Our GATD hypothesis argues that if a producer and consumer subscription model is sufficient, then complex queries can be expressed as a subscription to a specific queue that an intelligent GATD control plane could dynamically create, optimized with the knowledge of current compute resources and required data location.

3 Limitations / Future Work

GATD has been a very useful tool for the collection and processing of sensor network data, however there remain challenges to expanding its design if it is the be the foundation of a more general purpose SwarmOS dataplane.

3.1 Latency Sensitive Applications / QoS

GATD only has one quality of service (QoS) mode: eventual delivery of all messages exactly and in-order (for a given producer). Many applications have different demands. We postulate that a consumer-driver subscription model may be sufficient to express QoS more generally. When a consumer subscribes to a queue, it expresses its QoS requirements. These requirements can be "bubbled up" as the subscribee must verify that it can provide the consumer's request before allowing the subscription. In such a model, the long-term data storage model can be conceptualized as a consumer of the current type—requiring every message but not requiring anything better than eventual delivery.

3.2 Global Time / Global Order

Currently, when each receiver receives a message it adds a timestamp. This happens to generate a total global ordering of events as all of the receivers are currently running on the same machine. Once GATD components are actually distributed across machines clock drift will eliminate this total ordering property. As precise network time is challenging, we look to the distributed systems community for possible solutions. Lloyd et al's recent scalable consistency work provides a nice summary of the state of distributed consistency in addition to its new "causal consistency" primitive, which may be very apropos for an event-based system [1]. While formatters could network together to build an "overlay consistency network", ordering is a property that may be better expressed as a more fundamental primitive. It is not clear, however, how to discover and partition modules that require ordering. We need a better understanding of systems that require ordering before attempting to propose a solution.

3.3 Security and Privacy

When data streams come in to the same server, are stored together, and combined or aggregated, obvious data privacy concerns arise. Should sensitive data streams be combined with completely public streams? If two streams have overlapping access controls can they be combined by GATD? How are the access controls specified, and at what level? If streams have only best-effort QoS constraints, what side-channel information is potentially leaked?

The current GATD implementation does very little to address this. Streams are marked as either public (available to all users) or private (requires authentication to get data from the stream). An attempt to read from an unauthenticated private stream will simply return no data (so as not to leak the presence or absence of the stream itself). Authentication is handled out of band, issues of distributing trust remain an issue.

3.4 Location

With a large spread of sensors reporting data, it may be important to know where the sample was taken. Therefore, at some point, location data needs to be appended to the data packet. However, where in the system this should be added? What happens with mobile nodes? If a node can localize itself, how should it report this data?

4 References

[1] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. SOSP '11, pages 401–416, New York, NY, USA, 2011.

[2] M. Welsh, D. Culler, and E. Brewer. SEDA: An architecture for well-conditioned, scalable internet services. SOSP '01, 2001.