Binge: Processing All of the Things with a Binary-at-the-Edge

Kevin M. Greenan
BRB Inc.
Santa Cruz, CA 950602

January 12, 2021

Abstract

Most stream and event processing is done using popular Stream Processing Engines (SPE), such as Apache Storm, using event-based design patterns within a custom application stack, or a combination of the two. In either case, the foundation of these architectures relies on a centralized event bus or pub/sub system that acts as a buffer for processing events. While this approach is well understood and ubiquitous, it is not well-suited to many current and future applications within edge computing and modern SaaS applications. While there will likely always be a need for centralized event buses and custom application stacks, a great deal of stream processing can be done without an SPE or core application stack.

In this paper, we present Binge (binary-at-the-edge), a lightweight, durable, scalable, stream processing daemon that can run on commodity hardware without the need for complex application and infrastructure configurations. Each binge instance only relies on its own local configuration, which allows it to scale horizontally and heterogeneously. We show that binge can be leveraged for simple stream processing tasks at the IoT edge, VPC edge, PoP edge and as a mesh of coordinating endpoints.

1 Introduction

The ubiquity of software-as-a-service (SaaS) (e.g., Salesforce, Slack, GitHub, etc.) and cloud platform services (PaaS) (e.g., AWS, Google Cloud and Azure) has created a complex ecosystem of integration platforms that integrate SaaS services via events and APIs. There are a great deal of products aimed at automating decision making, tracking customer experience, automating engineering processes, and so on. These products are effectively processing event data from SaaS services and managing it in managed PaaS services. Integrating with a SaaS platform typically means subscribing to events, consuming events and calling their APIs. A basic integration platform may consume all events from any number of SaaS integrations and publish them to Kafka topics to be consumed by SPEs, custom microservice applications or big data systems such as BigQuery.

The many emerging IoT use cases are very similar. That is, consuming disperate events, publishing them to a centralized event bus and using SPEs to process. The main difference between the IoT use cases and the integration platform use case is the definition of edge. In the case of IoT, the edge is as close to the devices as possible, while the integration platform is usually a point-of-presence (PoP) or a load balancer in the platform's VPC or data center. In each case, there are different assumptions around what resources are available. For example, it might not be safe to assume now-latency access to a Kafka broker in the IoT use case, but the integration platform edge may be on the same network as a Kafka cluster. In the most ideal case, all filtering, transformation and processing can be done as close to the edge as possible. In reality, most of this is still done in a centralized fashion, albeit in distributed SPEs and microservice architectures.

The goal of binge is to simplify moving as much

event processing as possible to the edge (depending on the application) in a way that is durable, flexible and easy to operate. The goal is not to usurp existing SPEs or event based architectures, but to compliment them in a way that performs processing in the most appropriate tier (edge vs. hub) depending on cost, resources, performance, etc.

2 Outline

The remainder of this article is organized as follows. Sections 3 and 4 cover the design and implementation of Binge. We go through a few potential use cases for using Binge in Section 5. We evaluate the performance of Binge in Section 6. Section 7 gives account of previous work. Finally, Section 8 covers future work and we conclude in Section 9.

3 Binge Design

The high-level components of Binge are illustrated in Figure 1. Opposed to most SPEs and microservice architectures, which require a great deal of configuration and moving parts, Binge is composed of a single binary that can run as a standalone command (e.g. process a single event in a Lmabda, for testing or debugging) or as a daemon. The figure shows the components used for daemon mode. Binge exposes a HTTP endpoint that accepts POST requests containing JSON-formatted content, each representing an event. All events consumed by the daemon are persisted to a durable queue, which are consumed by workers. Each event will be processed by a worker in one or more pipelines defined in the configuration. Once a worker has completed processing an event, it will ack the event and pick up more work. This in combination with checkpointing (discussed later) allows the each daemon to be killed without losing events or processing state. Later we will discuss tradeoffs with the various durability configurations.

Today, the daemon can be also be configured in stateless mode, which disables the durable queue. This configuration can be used in cases where reliable delivery is less important than performance. In addition, there are some use cases where HTTP is not a sufficient interface. Adding new endpoint interfaces is relatively easy. For example, we can create a consumer interface that consumes messages from MTTQ queues when running in an IoT edge.

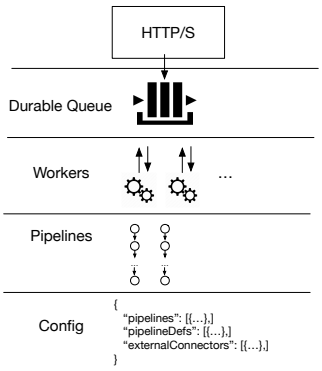

Figure 1: This

The rest of this section will be devoted to digging deeper into the durable queueing mechanism, pipelines, configuration and tradeoffs between different configurations.

3.1 Durable Queue

All events posted to the binge daemon are immediately placed into a durable queue before replying with success to the caller. This is done for two reasons. First, it allows this or another daemon to process events that were either unprocessed or in-flight after a crash. Second, it provides a buffering mechanism between the incoming events and the workers, preventing the need to apply backpressure. The remainer of this section is devoted to both of these aspects of the durable queue.

3.1.1 Event Processing and Crashes

The durable queue maintains three buckets: in-flight, unprocessed and an internal bucket for queue metadata, such as head and tail location of the unprocessed events. Figure 2 shows the basic data structures and a simple example. The queue can techni-

cally be backed by any underlying data structure that implements the following interface:

```
type DQueue interface {
    Dequeue() (*QueueItem, error)
    Enqueue(v []byte) error
    Ack(*QueueItem) error
}
```

We currently rely on BoltDB (https://github.com/etcd-io/bbolt) for persistence. Swapping out backends is trivial as long as the backing system can be mapped to a Key-Value interface. We chose BoltDB because it is fast, stable and runs in-process, which allows us to minimize the number of external dependencies. Running BoltDB also allows for configurations to easily leverage external block stores for persistence, which lends itself to more ephemeral environments.

As shown in Figure 2, we have 5 unprocessed events a 0 in-flight, before a worker pulls an event off the queue. Prior to returning the event to the worker, Evt_3 is atomically swapped to the in-flight queue. While processing the event, the daemon (as well as the worker) crashes and restarts. Before accespting any new connections the daemon will process the unprocessed and in-flight queues. Depending on the current status of each event, some may remain on the in-flight queue after the start-up process finishes. This could be due to an external resource being unavailable or an issue with the state of the event or checkpoint. The proper action depends mostly on the use case and could be a combination of: throw the events away, fire an alert, or forward the events to another system [1]

The self-contained nature of binge also allows many instances to serve the same event streams where daemons fail and recover without direct coordination. For example, if running in Kubernetes, a binge pod can be bounced and will simply continue using the same persistent volume when it restarts. We get similar behavior when running in VMs or on physical hardware, provided a supervisor detects the daemon stopped and requires restart.

[1] ToDo: Add recovery rules to the pipelines. For example, add an OnFail section to a process, which can take an action when it fails

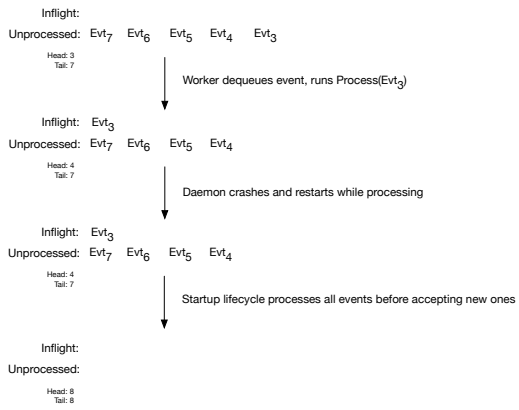

Figure 2: This

3.1.2 Backpressure and Tracing

We want to ensure all events are eventually processed, but there are times when a daemon gets overloaded and applies backpressure, usually in the form of a HTTP 429 response. One way to avoid the need to apply backpressure is to put a buffer between the enpoint serving the request and the processing. Here, the tradeoff is that returning a 200 OK only means the event has been persisted and the event is hopefully processed. As we have shown, we use a durable queue as our buffer. We use tracing to ensure we have visibility into the state of all events.

Given that binge may be running in a resource constrained environment, it is possible that a daemon is overloaded and the queue exhausts disk space in either a local or remote volume. There are two complimentary ways to ensure events are not lost: spin-up more instances (if possible) and/or specify high-water marks used to offload the latest events to an external system until a low-water mark is hit and we can pull those events in [2].

In the worst case, the daemon is running in a resource constrained environment, runs out of disk space and eventually has to resort to backpressure. In any case, the daemon itself can be configured to mitigate this issue by offloading newly consumed events

[2] ToDo: This can be worked into the configuration, likely as a command line option for the daemon

until a low-water mark is hit.

3.2 Pipelines

3.3 Configuration

4 Binge Implementaion

5 Example Use Cases

6 Performance Evaluation

7 Previous work

8 Limitations and Future Work

9 Conclusions

We worked hard, and achieved very little.