Flume NG

5outline

Overview

This is the top level section for all Flume NG documentation. Flume NG is a refactoring of Flume and was originally tracked in FLUME-728. From the JIRA's description:

> *To solve certain known issues and limitations, Flume requires a refactoring of some core classes and systems. This bug is a parent issue to track the development of a "Flume NG" - a poorly named, but necessary refactoring. Subtasks should be added to track individual systems and components.*
>
> *The following known issues are specifically to be addressed:*
>
> - *Code complexity; Flume has evolved over the last few years and has a fair amount of extraneous code.*
> - *Core component lifecycle standardization and control code (e.g. anything that can be start()ed or stop()ed, sources, sinks).*
> - *(Static) Configuration access throughout the code base.*
> - *Drastic simplification of common data paths (e.g. durability as an element of the source rather than a disconnected sink).*
> - *Heartbeat and master rearchitecture.*
> - *Renaming packages to org.apache.flume.*
>
> *This is a large and far reaching set of tasks. The intent is to perform this work in a branch as to not disrupt immediate releases or short term forthcoming releases while still allowing open development in the community.*

For reference, we refer to the code branch flume-728 (named for the refactoring JIRA) as "Flume NG." We call the current incarnation of Flume "Flume OG" ("original generation" or the slightly funnier definition, "original gangsta") which corresponds to the code branch trunk and that which was previously released under the 0.9.x stream.

Historically, NG code has been worked on by Arvind Prabhakar, Prasad Mujumdar, and E. Sammer (me). Jon Hsieh, Patrick Hunt, and Henry Robinson have provided help in vetting design. Will McQueen has provided usability and correctness testing. Development is obviously open to all and we'd greatly appreciate anyone who wants to jump in and help!

It goes without saying that NG is based on the fantastic work led by Jon Hsieh and all of the other contributors put into Flume (OG).

Architecture

Flume NG's high level architecture solidifies a few concepts from Flume OG and drastically simplifies others. As our goals state, we are focused on a streamlined codebase that meets the common use cases in a "batteries included," easy to use, easy to extend package. Flume NG retains Flume OG's general approach to data transfer and handling (a N:M push model data transport, where N is big and M is significantly smaller).

The major components of the system are:

- Event
  An event is a singular unit of data that can be transported by Flume NG. Events are akin to messages in JMS and similar messaging systems and are generally small (on the order of a few bytes to a few kilobytes). Events are also commonly single records in a larger dataset. An event is made up of headers and a body; the former is a key / value map and the latter, a arbitrary byte array

  In the future, an event body may be a Java `ByteBuffer`.

- Source
  A source of data from which Flume NG receives data. Sources can be *pollable* or *event driven*. Pollable sources, like they sound, are repeatedly polled by Flume NG *source runners* where as event driven sources are expected to be driven by some other force. An example of a pollable source is the sequence generator which simple generates events whose body is a monotonically increasing integer. Event driven sources include the Avro source which accepts Avro RPC calls and converts the RPC payload into a Flume event and the netcat source which mimics the `nc` command line tool running in server mode. Sources are a user accessible API extension point.

- Sink
  A sink is the counterpart to the source in that it is a destination for data in Flume NG. Some of the builtin sinks that are included with Flume NG are the Hadoop Distributed File System sink which writes events to HDFS in various ways, the logger sink which simply logs all events received, and the null sink which is Flume NG's version of `/dev/null`. Sinks are a user accessible API extension point.

- Channel
  A channel is a conduit for events between a source and a sink. Channels also dictate the *durability* of event delivery between a source and a sink. For instance, a channel may be in memory, which is fast but makes no guarantee against data loss, or it can be fully durable (and thus reliable) where every event is guaranteed to be delivered to the connected sink even in failure cases like power loss. Channels are a user accessible API extension point.

- Source and Sink Runners
  Flume NG uses an internal component called the source or sink *runner*. The runner is mostly responsible for driving the source or sink and is mostly invisible to the end user. Developers of sources and sinks may need to understand the details of runners and the difference between pollable and event driven sources, for instance, but shouldn't usually matter. Every source or sink is wrapped and controlled by a runner.

- Agent
  Flume NG generalizes the notion of an agent. An agent is any physical JVM running Flume NG. Flume OG users should discard previous notions of an agent and mentally connect this term to Flume OG's "physical node." NG no longer uses the physical / logical node terminology from Flume OG. A single NG agent can run any number of sources, sinks, and channels between them.

  Subject to available CPU, memory, blah blah blah.

- Configuration Provider
  Flume NG has a pluggable configuration system called the *configuration provider*. By default, Flume NG ships with a Java property file based configuration system that is both simple and easy to generate programmatically. Flume OG has a centralized configuration system with a master and ZooKeeper for coordination and we recognize this is very appealing to some users where as others see it as overhead they simply don't want. We opted to make this a pluggable extension point and ship a basic implementation that would let many users get started quickly and easily. There's almost certainly enough desire for a similar implementation to that of Flume OG, but it isn't yet implemented. Users may also implement arbitrary plugins to integrate with any type of configuration system (JSON files, a shared RDBMS, a central system with ZooKeeper, and so forth). We see this as something more interesting to system integrators.

- Client
  The *client* isn't necessarily a Flume NG component as much as something that *connects* to Flume NG and sends data to a source. One popular and good example of a client would be a logging package like a log4j appender that directly sends events to Flume NG's Avro source. Another example might be a syslog daemon.

Data Delivery Semantics

TODO.

Everything below this line is outdated. In the process of updating things now... -esammer

Notes

These are esammer's raw notes while hacking on NG. There's no guarantee they match the code exactly as they're taken in vim during development and then dropped here for reference. Ideally, they wil be refined over time and integrated into a developer handbook for Flume.

Critical Features

Having spoken to a large number of both potential and current Flume users, the following features seem to be the most important (beyond "transfer this data").

- General features
  - Configurable source / destination (i.e. the notion of sources / sinks makes a lot of sense to people)
  - Configurable degree of event durability (best effort and "end to end" seem the most desirable)
  - Multiple "channels" of data through a single physical node (i.e. the notion of multiple logical nodes per physical node makes sense, although nomenclature is confusing)
  - Centralized is important to about 50% of users. The other half don't want it and would prefer simple local files. Everyone agrees dynamic configuration (runtime reconfiguration) is critical as downtime for data ingest systems hurts.
  - HA features: multiple options for data delivery, multiple masters if there is to be a master
  - Scriptability of configuration and APIs to automate tasks
  - Deep metrics and monitoring (both health and performance) of the physical components, the service as a whole, and each end point in a data flow. JMX or REST / JSON seem acceptable to most.
- SPIs for
  - sources / sinks
  - input readers / output writers (a desire to have them disconnected from sources / sinks)
  - configuration interface (e.g. load conf data from zk, local fs)
  - possibly durability manager
  - RPC points (mostly to be able to plugin TLS)
- Data path features
  - Fan out and load balancing constructs
  - Logical groups of equivalent nodes (i.e. for `a -> b | c`, we don't want to have to specify b and c explicitly)

Common Use Cases

Route Java application logs to HDFS

- Application uses log4j or similar framework
- Use an appender that is Flume aware to send events to an RPC interface (e.g. Avro, Thrift)
- Durability usually critical (logs are used for monitoring or debugging)
- Low traffic

"Full event collection"

- Web application logs every user action (i.e. standard Apache log case)
- Folks tend to use tail techniques here
- Durability differs between orgs. Throughput trumps having all data most of the time.
- This is the scale case: common to have multiple TB / day.

Known Issues, Limitations, Concerns

...of the NG branch.

- A bunch of things that need to be thread safe aren't. We deal with some raciness and memory consistency stuff because the external contracts are still properly maintained. This usually means that sources / sinks may take an extra attempt to close or open, but externally, they do what's expected.
- Context objects are not used properly. There should be a notion of different scopes for life cycle ops than for normal next / append ops.
- There's still no (static) configuration system. We're configuring components using pojos. This is on purpose to prevent configuration references from leaking into the core, low level code.
- The thread interruption stuff needs mega-review.
- I'm 99.9% positive sources / sinks should implement LifecycleAware and use start / stop rather than open / close.
- The Reporter is not threaded through all the code properly. In other words, it's not respected. The intended behavior is that sources / sinks must call context.getReporter().progress() to indicate they're working if what they're doing takes longer than `timeout` millis.
- I'm becoming increasingly convinced that there should be a global lifecycle operation timeout. If this existed, we would change the policy on LifecycleAware state transitions to simply be "within X or die," where die means to -> ERROR and refuse to continue. This only leaves the case of stopping a collection of LifecycleAware services; to continue stopping if one fails to do so.
- Many of the unit tests are non-deterministic (Thread.sleep() ahoy).

Traits

It's super common that sources and sinks share common attributes / behaviors / configuration parameters. To simplify documentation and ensure feature parity across end points, we define the notion of *traits*. A trait is a set of shared behaviors and related configuration parameters. Traits act like mixins; any end point that wants to adopt common behavior / configuration (and user understanding) can simply reuse a trait. Note that traits do not prescribe a specific implementation strategy (e.g. inheritance, scala style traits, etc.). This is purely a logical convention (at least at this point).

It's legal to implement traits and not support all options (think UnsupportedOperationException). End points are validated prior to configuration being accepted so they have a chance to consider all options. All traits are theoretically orthogonal although, in practice, some may be correlated.

The following traits exist

- file output traits (file)
    - path
    - prefix
    - flush-interval-events
    - flush-interval-ms
    - output-format
- client traits (client)
    - group
- server traits (server)
    - port
    - bind
    - group
- queue traits (queue)
    - size
    - reject-policy
- agent traits (agent)
    - durability
        - mode
        - max-wal-size

Groups (client / server trait param)

In the existing Flume config language there's an incongruence in how logical nodes and sources / sinks are presented. This is largely the side effect of a Way-Back Decision(tm) so I won't go into details. Here's the issue.

User defines:

The problem: The user has to know that collector must be hostname2 or the config fails. Autochains were meant to solve this but the notion of flow is so under-documented it's not clear how to separate sets of compatible agents and collectors. Let's use a naming convention people get.

Clients are **always** in a *group*. Servers are **always** in a *group*. There's **no** way (in the built in sources / sinks) to communicate to a specific machine from a client; you **always** specify a group. The result is the same, but I think it's clearer. The inability to specify hosts simplifies the code in that group resolution must always occur (and is a first class feature).

The same config in "groups" (in pseudo-new-config-slash-old-config):

Groups probably also support a notion of *mode*. A mode is one of round-robin, fan-out, or least-loaded. This becomes both fan-out and load balancing across active-active collectors.

Catalog of Sources / Sinks

End point	Traits
thrift-agent	agent, server
avro-agent	agent, server
exec-agent	agent
scribe-agent	agent, server
syslog-agent	agent, server
collector-client	client, queue
collector-server	server
fs	file, queue
hdfs	file, queue
hbase	client, queue
batch	queue

Possible filter-ish thing?

Diagrams

TODO.