ATS (Application Timeline Server) v.2

1. Introduction

This document captures a proposal that improves upon the current ATS design to deliver scalability, reliability, and usability.

2. Motivation

There are two key motivations for the new design.

2.1 Scalability and reliability

As it stands today, ATS uses a single instance to handle all read/write traffic. Especially on the write-side, this poses scalability challenges once the cluster is beyond a certain size, and reliability challenges as well.

2.2 Key usability enhancements

YARN apps rarely run in isolation. They usually run in the bigger context of a workflow, whether it is Pig, Oozie, Cascading, Hive, or Tez. It is critical to treat flows and their concrete instances (runs) as first-class concepts in ATS.

There are other important metadata that need to be recognized and captured, such as cluster id and priority.

Metrics are an important part of the ATS data that need to be treated as first-class concepts. Metrics can be updated frequently, and they need to be aggregated from the containers to the app, and from apps to the flow (run). We should be able to update and aggregate metrics in an efficient manner.

Configuration (as string-based key-value pairs) is another concept that needs to have the first-class treatment.

Efficient queries around things such as flows, runs, configuration, and so on, should be possible.

3. Object Model

[NOTE] *There are still some more details that need to be worked out regarding the object model. Please treat this section as tentative.*

3.1 Flows

Flows are supported as a first-class concept in ATS.

3.1.1 Definitions

A **flow** is defined as a high level “application” that can spawn multiple YARN applications to complete the work on a cluster. Examples may be a Pig script, an Oozie flow, a Cascading application, or a Tez application.

A flow is further qualified by the user that runs the particular application. If two different users run the same pig script for example, those are two separate flows.

A flow is a template. The actual realization of a flow is a **run**.

Flows should have reasonably unique identifiers. Flows may also have versions which is used to identify changes in the flows, such as code changes or script changes. Runs for a given flow must have unique and totally ordered run identifiers. For example,

- flow id: “analyze_browsers:foo” (user “foo” running pig script “analyze_browsers”)
- flow version: dc4309306ebb22f813588cc1b8c9762e2c44c1a1 (hash of the pig script)
- run id: 1413323689 (unix timestamp when the run was started; ensuring that run id is unique is the framework’s responsibility)

The actual format of the flow id and the run id is up to the individual frameworks that set them. A client-side API should be provided (e.g. as part of the application submission context) so frameworks can set them.

If the flow id is missing, ATS can assume a default behavior of reusing the YARN application name in lieu of the flow id. Similarly, in the absence of the run id, the app start/submit timestamp can be used as the default.

Flows (and runs) can be implemented in terms of YARN tags. An AM would get the flow and the run id it belongs to, and can embed that information in ATS events and metrics.

3.1.2 Run-level aggregation

Some metrics such as counters need to be aggregated at the encompassing flow run level from individual YARN apps. It needs to be specified exactly what metrics should be

aggregated and what not. This information is necessarily application-specific. This could be implemented with a properties file for example.

This aggregation may happen on the read path; i.e. the ATS reader instances may aggregate these metrics on the fly and serve them.

Certain storage types (e.g. HBase) may choose to aggregate on the write path using mechanisms such as coprocessors, for example, as apps are completed, mostly for performance reasons.

As a rule, it is acceptable that the run-level aggregation of metrics happens only for apps that are completed so the aggregated values do not include values from active apps.

3.2 Metadata

ATS should communicate the key metadata associated with timeline data such as

- flow id
- run id
- YARN app id
- priority
- timestamp
- cluster id

in addition to the existing metadata.

We may also add the framework version (e.g. pig version), OS version, JVM version, and Hadoop version as metadata.

Furthermore, at least the run should be an entity in its own right (in addition to metadata) because metrics can be associated with a run. The same may possibly apply flows.

The cluster id is by default the same cluster in which ATS is running, unless explicitly specified (again in a YARN tag).

3.3 Data

The data basically should have the following top-level elements:

- events
- metrics
- configuration
- relationship

3.3.1 Metrics

Metrics should be elevated into first-class objects, as opposed to part of the “other info” as it is done today.

We can reuse the well-known Hadoop metric types (counters, gauges, rates, etc.), instead of creating new types. In addition, the following attributes are added:

- whether the values should be aggregated
- whether it is critical (i.e. should never be dropped in any circumstance)

The metrics update should be as fine-grained as possible. If a single metric value is updated, only that metric should be updated in the backing store, and nothing else. This is in contrast with today’s behavior where metrics are part of the other info, and the entire other info field is updated even if only one metric is updated.

3.3.2 Configuration

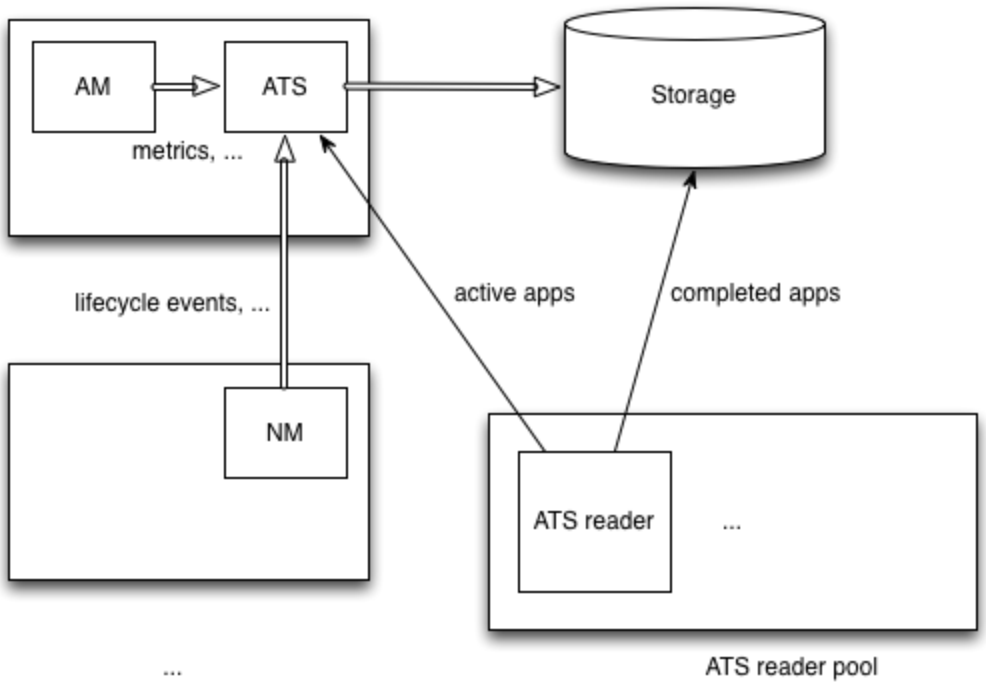
The configuration should be considered a top-level object, rather than part of the “other info”. It is defined as string-based key-value pairs, as in Hadoop’s Configuration. Furthermore, it should be stored in a format that can be queried easily. For example, storing it as an opaque XML blob might not be the best way.

The backend storage implementations should support efficient queries based on the configuration; e.g. “return 10 most recent apps where config X = Y”, or “return all apps in a flow run where config X = Y”. Also, it should support queries that return config values selectively; e.g. “return value of config X of all apps in a flow run”.

3.3.3 Relationship

There should be a strong parent-child relationship between key entities so as to enable efficient queries in both directions. For example, it should be easy and efficient to query “all apps in a given flow run”, or “the flow run given an app”.

4. Distributed Writers/Readers



4.1 ATS writers as local companions to AMs

The key aspect of this design is to spawn an instance of ATS for each AM. When the node manager allocates a container for an AM, it also spawns a companion ATS writer instance that is dedicated to that particular YARN application (see below for the benefits). The capacity that is used by the ATS should be attributed to the correct user. Although it would be more optimal to spawn the companion ATS instance on the same machine as where the AM is, it is not strictly required (and up to how we implement it).

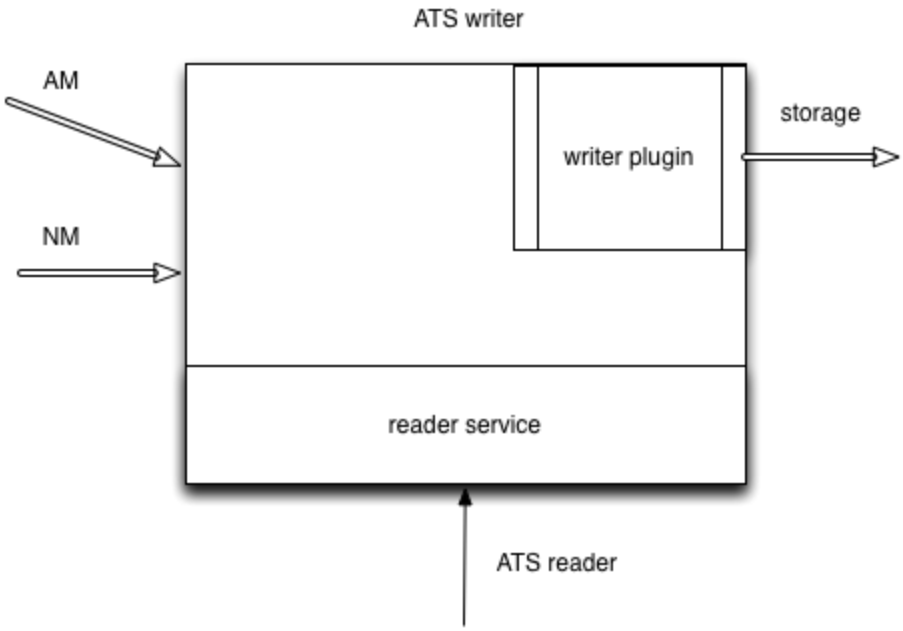
If the AM is restarted the corresponding ATS could also be shutdown and restarted on the node that the new instance of AM is running.

AMs write application-specific data (metrics and events) to their local ATS writers. Node managers can discover these ATS writers for a given app/container and write container-related events (e.g. YARN-generic container lifecycle events) to the respective (potentially remote) ATS writers. This requires a **service discovery** feature to be able to discover the right ATS writer instances, such as YARN-913.

The ATS writer has a **writer plugin** that will handle different storage types, whether it be LevelDB, HDFS, or HBase. The writer plugin would encapsulate not only different storage types but also strategies of handling things like flow-level aggregation of metrics, etc.

RM itself has its ATS instance to be able to write RM-specific timeline events (e.g. application lifecycle, scheduler, etc.). RM can also use YARN tags to associate events with a specific flow/run/app. The volume of data coming directly from RM is not expected to be great.

The ATS companion should run under a special user that has permission to write to the backend storage.



This approach has several key benefits.

First, this essentially distributes writes from a single and global instance of ATS to a number of distributed writers, thereby eliminating the scalability bottleneck for the write path (and for the read path as well).

Second, this also accomplishes a fairly optimal number of connections to the backend ATS storage. It is based on the assumption that the number of nodes in a cluster usually outstrips the number of active applications. Since there is a strong affinity between an application and the ATS instance, the number of active connections to the backend would be around the number of active apps, and those connections would be stable (long-lived). This has many beneficial effects on storage types like HBase for example.

Third, this also provides better isolation between different applications. Whereas writes from all applications had to flow through a single instance of ATS and there was no predicting whether one chatty application may impact other applications, ATS instances dedicated to applications provide better (although not perfect) protection for its writes. This also gives us a good accountability towards the ATS resources (they are correctly attributed to the app/user).

4.1.1 Writer plugin

For critical data such as YARN-generic lifecycle events and app-level roll-up metrics, the expectation is that they will be written out to the storage before ATS returns a response to the client.

On the other hand, for data such as container-level metrics, the ATS writer may buffer them, acknowledge the client, and flush them to the storage on a much less frequent basis. For example, AMs may emit container-level metrics every minute while ATS may flush the data every 10 minutes. The actual strategies of how often the data is flushed to the storage can vary: it can be on a periodic basis, it can depend on the size of data buffered, and so on.

The writer API should support both synchronous and asynchronous mode of writing as before, with the understanding that the guarantees are different.

The writer plugin can be implemented as a YARN service. The key methods it needs to provide are:

- init()
- start()
- recover()
- write()
- stop()

The init() or start() method can be used to establish and pass information such as cluster, flow, user, etc.

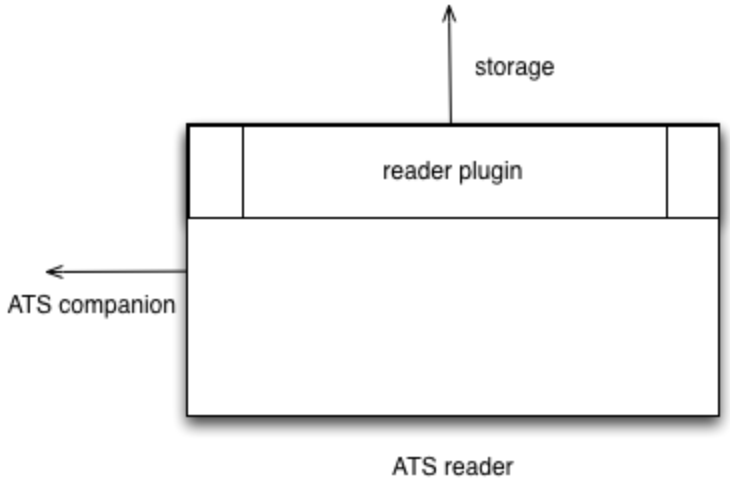
The recover() method is needed in case an AM crashes (and therefore the corresponding ATS writer instance shuts down) and a new AM/ATS pair is created somewhere else.

4.1.2 Reader service

Each writer instance is also accompanied with a reader service that expose the live data to ATS Reader(s). Each ATS Reader is a stateless server that forms the response to the clients by querying both the backend storage (for completed jobs) and the reader service (for in-progress jobs).

4.2 ATS readers

We also have a stateless ATS reader instance or a pool of instances that are dedicated to serving queries.



As a rule, the reader will query the ATS writers themselves for apps that are still active (i.e. AMs and the ATS writers are running). For apps that are completed, the reader will query the backend storage.

The mapping from the flow run to the launched apps can be retrieved from the backing storage. The ATS reader can query either use the service discovery itself or RM to determine whether a certain YARN app is active or not.

Similarly to the ATS writers, we need a **reader plugin** that can handle different storage types and strategies.

The ATS reader should enforce security. The existing access domain and its ACL should be enforced on the data visibility.

5. Backing Storage Implementations

We should have at least the following backing storage implementations:

- LevelDB implementation
- Filesystem-based implementation (HDFS)
- HBase implementation

Each implementation would implement its own writer and reader plugins.

In general, a 1-to-1 correspondence between a YARN cluster and a backing store should **NOT** be required; i.e. the backing storage implementation should support using a single backing store for multiple YARN clusters.

Backing storage implementations may choose to use write-side optimizations (e.g. buffering data before flushing it out to the backing store) as well as failure handling scenarios (e.g. store data in HDFS temporarily if the backing store is unavailable).

In case of LevelDB, there can be only a single (local?) writer to a LevelDB instance. Therefore, either we revert to a single global ATS writer instance for LevelDB, or the LevelDB writer plugin would proxy writes to the global ATS write instance local to the LevelDB instance. The latter option enables us to keep the same ATS writer allocation, regardless of the backing storage type.

The HDFS writer plugin would write data by creating HDFS files and appending data to them. We may consider optimization such as parquet to improve the throughput.

The HBase writer plugin would use a HBase cluster to be able to scale. In the case the HBase storage is not available, the plugin should buffer the writes temporarily (e.g. HDFS), and flush them once the storage comes back online. Reading and writing to hdfs as the the backup storage could potentially use the HDFS writer plugin unless the complexity of generalizing the HDFS writer plugin for this purpose exceeds the benefits of reusing it here.

6. Handling Migration

There are several issues that need to be handled in terms of migrating existing users of ATS.

6.1 Data compatibility

The data schema of the backing store may change considerably from the existing implementation to this iteration. Although it would be ideal to preserve the schema compatibility, it may not always be possible. It would be good to provide a policy on how the old existing data can be accessed when the new version of ATS is in place. Transforming the old data into the new schema may be a good idea if it is feasible.

6.2 Client library API compatibility

Although it would be good to preserve the API compatibility, again we may need to change/evolve the API in an incompatible manner to satisfy the requirements. If we can affect all cases of client implementations, we could upgrade them to the newer version. If that is deemed not feasible, we may want to provide some type of a shim layer to ease the transition.

7. Unresolved Issues

The following issues need more discussion:

- The LevelDB case needs to be hashed out more; can we have the ATS companion model in the LevelDB case, considering the fact that there can be only one writer to a LevelDB instance?
- Unmanaged AMs
- Whether we would require a new incompatible client API
- What state should be recovered, if any, if the ATS writer instance is restarted (due to an AM crash)?
- How to handle ATS crashes (separate from the AM crash)
    - fail the AM?
    - restart the ATS?
    - consider the data for this app unreliable?
- How much change the current single-threaded implementation of RM requires to handle sync writes to ATS?