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. Motivations

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 (flow 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 is another concept that needs to have the first-class treatment.

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

3. Object Model

This section describes all the major proposed changes to the existing object model as specified by the current API (TimelineClient, TimelineEntity, and TimelineEvent).

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 workflow, a Cascading application, or a Tez application.

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

A flow is a template. The actualization of a flow is a **flow run**.

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

- flow id: "alice@pig_analyze_browsers" (user "alice" running pig script "analyze_browsers")
- flow version: dc4309306ebb22f813588cc1b8c9762e2c44c1a1 (hash of the pig script)
- flow 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 flow 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. If the version is missing, the default is "1".

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

A flow run is a system entity (see below for system entities). A flow run should have flow id, flow version, and flow run id as required metadata.

3.1.2 Flows of flows

A flow may have another flow as its child (or more precisely a flow run may have another flow run as its child). For example, an Oozie flow may contain some pig scripts (thus pig flows) as its children.

- parent flow run (flow id / flow version / flow run id): “alice@oozie_system_test” / 8bad6cae758d35a0f26f6577136d5b278713f1bf / 1413323689
    - child flow run: “alice@oozie_system_test:pig_analyze_browsers” / dc4309306ebb22f813588cc1b8c9762e2c44c1a1 / 1413323689

3.1.3 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.

Here we refer to aggregation of metrics from individual YARN apps to flow runs and above (“run-level aggregation”). Aggregation up to YARN apps can be done by the AM or ATS writer itself.

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 does not happen for every write to the backing storage. Therefore, the aggregated values may be stale up to an extent. How often aggregation happens is up to the specific backing storage implementations.

In case of flow runs having child flow runs, the aggregation should happen only at the topmost flow runs and not at the nested flow runs.

3.2 Data

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

- events
- metrics
- configuration
- relationship

3.2.1 Metrics

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

We may reuse the well-known Hadoop metric types (counters, gauges, rates, etc.) or come up with new types of metrics that are suitable for this purpose. 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)

This data structure should be a map from the metric name (String) to this wrapped metric type. Methods such as addMetric(), addMetrics(), and setMetrics() should be added to TimelineEntity to support them.

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.

There are **system metrics** (or YARN-generic metrics) that are common across all frameworks, such as container memory size, and **framework-specific metrics**. Only system daemons such as Node Managers and Resource Managers may write system metrics. Any attempt to write system metrics by a non-system client is rejected.

3.2.2 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.

The data structure for the configuration is a map from string keys to string values.

There is no inheritance implied between parent configuration and child configuration.

Configuration is optional. Not all entity types need to configuration. Only certain types of entities, such as YARN applications, may have configuration.

The backing 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".

3.2.3 Parent-child relationship

There should be a strong parent-child relationship between entities so as to enable proper **aggregation of metrics**. For example, a flow run entity may be a parent entity of a set of YARN

application entities. Metrics of child entities that can be aggregated should be aggregated (rolled up) at the parent flow run entity.

For example, a typical flow may have the following entities in the lineage:

Cluster → User → Flow → Flow run → YARN app → ...

Aggregation of metrics is done along the lineage of parent-child hierarchy.

This should also 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”.

The parent relationship was previously captured in related entities as well as in the primary filters previously. Instead, there should be a first level parent information attribute. To support this, methods like setParent(), getParent(), and getChildren() should be part of TimelineEntity.

This aggregation is a primary aggregation. Separately, there can be a secondary aggregation of metrics. For example, although queues are not part of this lineage, we may want to aggregate metrics up to individual queues. This secondary aggregation may be done outside the primary linear aggregation.

3.3 Metadata

The metadata associated with timeline data should now be first-class attributes. To that end, methods such as addMetadata() and setMetadata() should be added to TimelineEntity. The data structure is a map from string keys to object values.

Specific frameworks can set their own metadata. System (YARN-generic) entities should set some well-known metadata. Those include

- flow id
- flow version
- flow run id
- YARN app id
- priority
- timestamp
- cluster id
- user
- queue

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 flow run should be an entity in its own right (in addition to metadata) because metrics can be associated with a flow 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).

3.4 Strongly-typed system entities

Some system entities should be strongly typed, and recognized as such in the client API and handled appropriately in the backing storage implementations.

Such system entities include (and may not be limited to) cluster, flow run, (YARN) application, container, app attempts, etc. These types should be derived from the underlying TimelineEntity type, but can add methods to support required data and metadata. For example, a flow run entity must have the associated flow id, flow version, and the run id as metadata.

The backing storage implementations can assume these are well-known and most common entities, and lay out the storage that optimizes for these entities.

3.5 Miscellaneous

3.5.1 Related entities

The related entities are deprecated in favor of the parents. If you need to capture any other entity relationship, it can be included in the other info.

3.5.2 Primary filters

The primary filters are also deprecated. The main reason for the primary filters was to optimize storage for efficient queries based on a primary filter. Many of them can still be implemented via metadata. We will introduce a pluggable solution for framework-dependent primary filters if necessary.

[TO-DO] More decision and clarification on primary filters is needed...

3.5.3 Sync and async write API

We need both synchronous and asynchronous write methods. For the most part, asynchronous writes are preferred as it does not impact the latency of the client and it enables more optimization on the part of the ATS writer companion.

For some critical data that we cannot afford to lose, the synchronous writes may be supported.

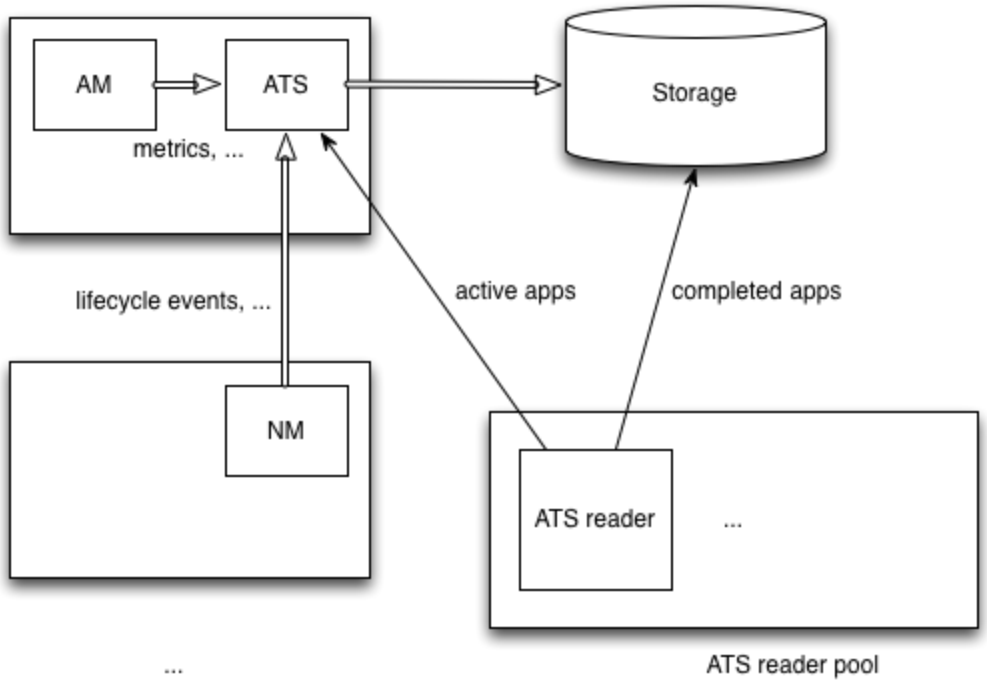
For a related discussion, see YARN-2517.

3.6 Queries (REST or RPC)

We should support all the existing queries that are currently supported by ATS. In addition, new queries based on flows, flow runs, and configuration should be supported.

[TO-DO] Spell out supported and unsupported queries some more

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. It would be more optimal to spawn the companion ATS instance on the same machine as where the AM is. When the AM container is allocated, the RM instructs the NM that it is requesting an AM container. The NM recognizes that it is creating an AM container, and also creates the special ATS writer container. The

allocation of these containers should be done atomically. Both containers should be allocated or neither.

If the AM is restarted the corresponding ATS could also be shut down and restarted on the node on which the new instance of AM is running.

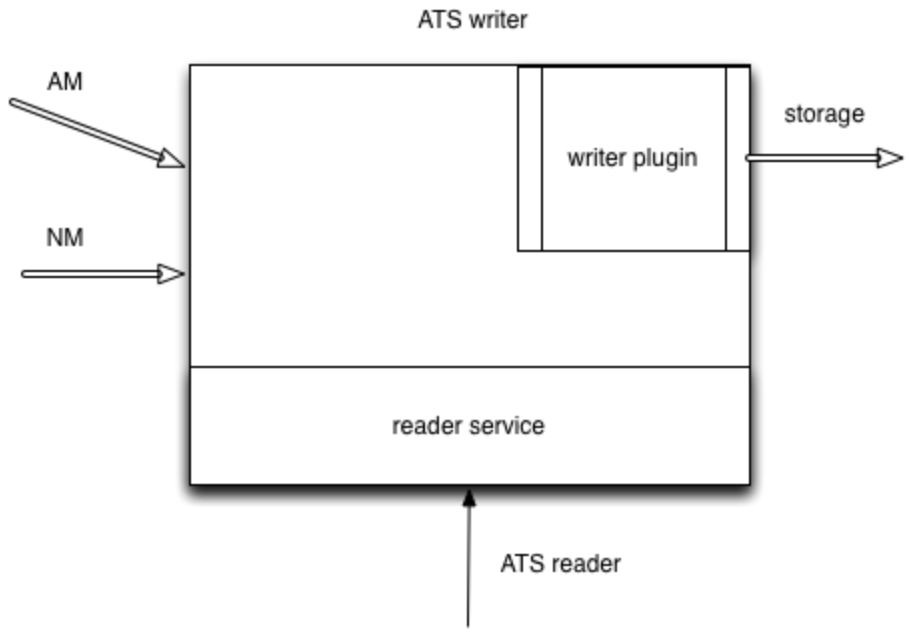
AMs write application-specific data (metrics and events) to their local ATS writers. Node managers can discover these ATS writers for a given app/container and write container-related events (e.g. YARN-generic container lifecycle events) to the respective (potentially remote) ATS writers. This requires 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 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.

The ATS writer is essentially a daemon. It runs no user code.

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

The ATS companion should run under a special user that has permission to write to the backing 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 backing 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 backing 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 pass and initialize information such as cluster, flow, user, queue, 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 has a reader service that exposes 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 backing storage (for completed jobs) and the reader service (for in-progress jobs).

4.1.3 Handling ATS writer failures

We assume that the failure semantics of the ATS writer companion is the same as the AM. If the ATS writer companion fails for any reason, we try to bring it back up up to a specified number of times. If the maximum retries are exhausted, we consider it a fatal failure, and fail the application.

The AM and the ATS writer are always considered as a pair, both in terms of resource allocation and failure handling. If the ATS writer fails, the AM should be shut down as well, and vice versa.

4.1.4 Unmanaged AMs

The unmanaged AM can be handled in the same manner. Although there may not be an actual AM instance on the cluster, we can still spawn the ATS writer instance. All cluster-side data and metrics are written using this instance.

4.1.5 Clients writing to ATS

There may be a need for clients to write directly to ATS. For example, a pig client (on which the pig execution engine runs) may want to write data to ATS directly. Since writes to ATS need a writer instance, we may need to create a generic global instance of the ATS writer so that clients or other entities off-cluster can write data to ATS.

One key use case is **initiating and creating the flow run entity**. There are two possible approaches for handling this:

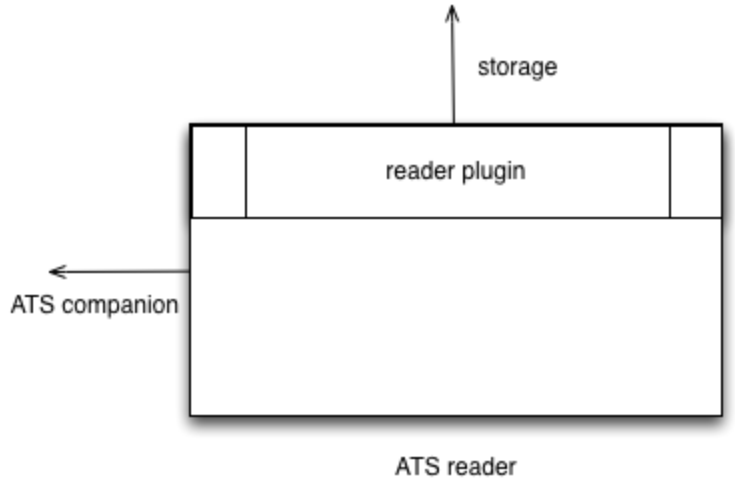
- have the NM for the first application write the flow run entity as part of writing events for that application
- have clients directly write the flow run entity just before submitting the YARN applications

Although the former approach may work in most cases, it's bit of a roundabout way of doing it. On the other hand, the latter approach requires an instance of the ATS writer that clients can write to. And there are issues such as establishing trust (whether based on the host or not) and rate limiting.

Off-cluster clients writing to ATS outside the context of YARN apps is out of scope. We will always rely on writing to ATS in the context of YARN apps.

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 backing 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.

The ATS reader instance can serve requests for a different cluster than it belongs to, as long as the data is found in the backing storage and with the user's access visibility.

5. Backing Storage Implementations

We should have at least the following backing storage implementations:

- (default) single HBase instance implementation
- Filesystem-based implementation (HDFS)
- HBase cluster 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).

The default implementation uses a single HBase instance.

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. Out of Scope

The following aspects are currently out of scope for this iteration of ATS. They may be considered in a future revision.

- Supporting clients writing to ATS outside the context of YARN apps
- Supporting multiple parents: having multiple parents may be helpful in terms of supporting multiple aggregation paths; we will defer this to a later phase
- Supporting more strongly typed configuration values (e.g. numeric) and enabling type-specific queries around them (e.g. number greater than a certain value)
- Any inheritance semantics between parent and child configuration
- Supporting a “logging level” feature where you can control levels of data that can be written and read
- Supporting inter-cluster flows

8. Unresolved Issues

The following issues need more discussion:

- Whether we would require a new incompatible client API
- It would be good to decide on supported and unsupported queries (via REST)
- Do we want to support secondary aggregation in this phase (e.g. queue aggregation)?
- What state should be recovered, if any, if the ATS writer instance is restarted (due to an AM crash)?
- How much change the current single-threaded implementation of RM requires to handle sync writes to ATS?
- A little more needs to be worked out on how to handle multiple parents and their aggregation semantics: example?
- We haven’t quite come to the conclusion on what to do with the primary filters; how about framework-specific primary filters? Other than (the clearer name), is there a big difference between metadata and primary filters? Is it just a renaming of the primary filters?