```
Title: [YARN-1530] Store, manage and serve per-framework
                    application-timeline data
By: Arun C Murthy, Billie Rinaldi, Gopal Vijayaraghavan,
    Hitesh Shah, Mayank Bansal,
    Vinod Kumar Vavilapalli, Zhijie Shen
Last modified: January 8 2014
```

1.0 Preamble

YARN-321 addresses the problem of storing, managing and retrieving *generic historical* information about applications. The application-history problem is solved by making ResourceManager write the events as and when they happen (application-start, container-start, app-finish etc.) to a pluggable store (default implementation based on HDFS files) and by serving the information via a generic **ApplicationHistoryServer** (**AHS**). AHS deliberately avoids solving issues with storing, managing, retrieving and rendering framework-specific data - both current and historical - to manage complexity. YARN-1530 aims to address this gap as detailed in this document.

1.1 Problem statement

Today, each and every application/framework has to store, manage and serve per-framework data all by itself as YARN doesn't have a common solution. MapReduce does (and by extension any application/framework can do) this by

- Running a web-server inside the **ApplicationMaster** (**AM**) to serve non-generic information about the running application
- Making ApplicationMaster emit events to what are known as Job-History files, manage them on HDFS and serve them via a MR (framework) specific JobHistoryServer (daemon).
- Clients use RPCs, web-UI and web-services on JobHistoryServer to learn about finished applications.

YARN needs a common solution for handling non-generic information about running applications as well as about finished applications; with plugin points for frameworks to do their own rendering.

1.2 Proposal

YARN-1530 JIRA ticket attempts to solve the storage, management and serving of per-framework data from various applications, **both** *running* and *finished*. The aim is to change YARN to enable collection, aggregation and storage of this data in a generic manner with plugin points for frameworks to do their own thing w.r.t data-interpretation and serving. The solution is

in a way a generalization of MapReduce JobHistory events and thus involves

- **A timeline writer API/Library**: A simple REST API and/or a more involved client library that AMs and containers can use to send data across.
- **Event aggregation**: Once AMs and clients write their data to a API/library, YARN uses an aggregation system to aggregate all the events from different AMs & containers across applications; which eventually feeds into a storage system. The event-aggregation is a function of the API/library.
    - We can have REST APIs via which the clients (AMs and/or containers) directly post the events to a central location. In this case, the event-aggregation merges itself with the writer library + storage system.
    - On the other hand, event-aggregation itself can be a pluggable system that makes use of external frameworks like Kafka/Flume and potentially gives better guarantees w.r.t reliability, fault-tolerance, etc.
- **Data storage**: A pluggable storage to persist all the events into a centralized durable storage for serving and analysis. This can include
    - An embedded or local file-system based storage for simple and small deployments.
    - An implementation for larger scale deployments based on a system like HBase.
- **Serving**
    - Reusing the serving layer in Application History Server (from YARN-321) and extending it to be a generic **Application Timeline Server** (**ATS**). At that point, as should be obvious, it can serve both generic history as well as per-framework current and historical data in the form of web-services (and/or read-only RPC end-points)
    - Query API: Web service functionality exposed by the Application Timeline Server either gives out the complete set of events or after filtering them based on time, event-type, etc. It *DOES NOT* do any rendering *by default*.
- **Rendering/User interfaces**:
    - To facilitate framework specific user-interfaces, *Application Timeline Server* enables administrators to optionally install rendering-libraries for *specific versions* of specific *well-known*, commonly used applications/frameworks.
    - Other than the admin-ratified rendering libraries, YARN neither encourages nor blocks users from writing their own user-interfaces making use of the serving APIs. It won't directly link to them for reasons of security.

1.3 Details

The system is centered on a (multiple-writer) framework for a given application to persist events of its lifetime in a centralized storage. The Application Timeline Server, the storage layer,

ApplicationMasters/containers are the main players in the event cycle. The AM/containers have to log the events that happen during their life-time. The storage layer and ATS are responsible for providing services to the clients for persistence, extraction and display of this information. More details on each of these individual parties follows:

1.3.1 Timeline writer API/Library

There are multiple options for writing events. Existing systems tend to write them to files or sockets directly or use some kind of an interface like a log-appender, a REST end-point or by using a client library. We've chosen the latter.

The API for application writers to use in their AMs/containers is either a simple REST API and a more involved client library that takes care of more things on behalf of the user. There are several things that may need to be done on top of a simple event-post, these things are implemented by the client library. The choice of the API has much larger implications on the event-aggregation which we inspect more in the corresponding section.

Application timeline

Writers (AMs/containers) write per-app/per-framework data using events. We call these the **Application Timeline events** as they are modeled around timestamps and specific to an application. Because of the distributed nature, the event-stream for the timeline can potentially be fed into the system by the application related entities - AM, containers - in any order and it is the responsibility of the serving layer to reorder the events to be sorted depending on the need. There are two APIs for the writers : (1) a REST API that provides minimum guarantees & enables cross-platform access and (2) an embeddable library that provides more features like buffering, failover, etc.

In the remainder of this section, we showcase the REST APIs. Corresponding APIs can be demonstrated trivially from a library standpoint too.

HTTP Request

```
Request: PUT http://<host>:<port>/ws/v1/timeline
Response: The status of the request

Description: Put information about a given entity of a given type.
```

- In general, adds to existing information about the entity, with the following exceptions:
    - If an identical event is inserted later with different event info, the info will be overwritten.
    - If an other info name/value pair is inserted later with a different value, the value will be overwritten
    - If a start time is provided that does not match the existing start time for the entity,

it will be ignored and the existing start time will be used (see below for more information on start time).

- Not all information needs to be filled in with every request.
- Can post a batch of events about different entities for efficiency.

An example of a request follows. The request includes an entity and entity type (e.g. task_1381946403170_0003_3_00_000000 and TezTaskId). The entity and entity type pair should be unique over all applications, so it may be desirable to include the application name in the entity type, as in TezTaskId.

An optional start time is used to index this entity by time. If not provided, the store will attempt to use the start time that already exists for this entity. If none exists, it will be set to the minimum timestamp of the events in the put request. This effectively indexes entities by start time. The start time for an entity never changes once it is set.

The request also includes an array of events, each of which must include a timestamp and event type, and may include optional event info. If an identical event type and timestamp are written for this entity later, the event info will be overwritten.

The example assumes a Tez application creates a DAG that creates vertices that create tasks (only a request for a single task is shown below). Here, the 'relationship' between a task and a vertex is established by specifying a vertex ID as a related entity of a task ID. The task ID will then be retrievable as a related entity of the vertex ID -- **the reverse of how the relationship was put**. Related entities are specified as a map from entity type to array of entities and are optional. Additional related entities for the same entity type are added to any existing related entities.

Optional primary filters specify additional name/value pairs under which the entity information in the put request will be indexed. To be indexed, the information must be provided in the same put request; it will not use previously put information for the entity. Primary filters may also be used for secondary (unindexed) filtering.

Other entity info are name/value pairs that are stored and retrievable by entity, but will not be used to index the entity. These name/value pairs are added to any that already exist for the entity, but if a value already exists for a given name, it will be overwritten by the new value. Other info may be used for secondary (unindexed) filtering.

```
PUT http://<host>:<port>/ws/v1/timeline/
{
 "entities": [
  {
    "entity": "task_1390516007863_0003_1_00_000000",
    "entitytype": "TEZ_TASK_ID",
    "starttime": 1390516418713,
    "events": [
      {
        "timestamp": 1390516418713,
        "eventtype": "TASK_STARTED"
        "eventinfo": {
          <name> : <value>,
          ...
        }
      },
      ...
    ],
    "relatedentities": {
      "TEZ_VERTEX_ID": ["vertex_1390516007863_0003_1_00", ...],
      ...
    },
    "primaryfilters" : {
      "user" : "hadoop",
      <name> : <value>,
      ...
    },
    "otherinfo": {
      "startTime": 1390516418713,
      "scheduledTime": 1390516418713,
      <name> : <value>,
      ...
    }
  },
  ...
 ]
}
```

More details about the API:

- Examples of anticipated primary filters at the application level include user, queue, etc. and at the task level may include node, rack, container, etc.
- Some applications may wish to use entities as primary filters for other entities.
- Depending on the implementation, primary filters may be expensive in terms of storage (involving storing the entire entity information again for each primary filter) and thus

primary filters should be used sparingly.

1.3.2 Event aggregation

YARN needs to use an aggregation system to aggregate all the events from different AMs & containers across applications and then to eventually persist them into a storage layer. As mentioned before, the event-aggregation is a function of the API/library.

- Clients (AMs and/or containers) directly post the events to a central location via REST APIs. In this case, the event-aggregation merges itself with the writer library + storage system.
- On the other hand, event-aggregation itself can be a pluggable system that makes use of external frameworks like Kafka/Flume and potentially gives better guarantees w.r.t reliability, fault-tolerance, etc.

A quick description of various '*ilities*' desired with the event-aggregation sub-component:

- **1.3.2.1 Scalability**:
    - A REST API based system has scalability issues with tens of thousands of AMs + containers pushing events out. The fact that events can have custom, application-specific payload too (and not just control-data) which also is routed through the web-service layer only exacerbates this. It can be scaled on the server-side by, say, using a farm of web-proxies. The good part about this is that it is a known path towards scaling web-services. The bad is that it adds to the operator burden.
    - A library can let administrators choose a scalable event-aggregation framework like Kafka/Flume completely transparent to the user. The con to this approach is adding yet another dependency to YARN. The library for example could use the REST API itself for smaller installations and seamlessly change to a scalable aggregation framework.
    - There is a third option of choosing a scalable framework and directly expose users to that API. But our requirement of flexibility between small and large clusters, extra vs no external dependencies pushes us to not pursue this.
- **1.3.2.2 Reliability**: If applications wish to use the aggregated timeline say as a transaction log for their recovery (this is what MapReduce ApplicationMaster does with its JobHistory), we will need reliability guarantees for the event-storage. For e.g., MR JobHistory is flushed every so often by the MR AppMaster so that it can get as many recorded events as possible when recovery happens.
    - A REST based API implies that an event-write acked by a 200(OK) status code amounts to a success.
    - A library based approach may involve exposing APIs to flush and/or client call-back handlers. It can explicitly implement it itself or delegate it to an underlying aggregation system if it supports such APIs.

- **1.3.2.3 Fault tolerance**: Clients writing the events may go down before the acknowledgement comes in. The aggregation-system may also go down in a similar fashion. Network failures may also happen. The Application-writers need some guarantees on the API if they are to rely on the written data for more than UI purposes.
    - A REST based API may force client to explicitly retry.
    - A library can wrap the retry functionality within itself.
- **1.3.2.4 Buffering**: This is listed as a separate requirement and not as a solution as it is one of the natural choices to address some of the scalability and the fault-tolerance problems listed above. Buffering helps scalability by queueing events locally when destination systems are loaded. Events can also be queued in the presence of a fault somewhere - client, aggregation-system or the network - and can be resent once the system recovers from the fault.
    - A REST based user-interface needs APIs that accept batches of events. On top of that, it requires each and every client to buffer events for a while before sending them across.
    - A library can wrap the buffering functionality itself. It can implement buffering itself or delegate it to the underlying aggregation system if it supports buffering.
- **1.3.2.5 Timeliness**: For serving the information while the application is running, the event-aggregation as well as the storage need to be as real-time as possible. Some lag for the end-user is acceptable but it shouldn't go beyond a few seconds at a maximum.
- **1.3.2.6 Forwarding**: Unlike other messaging/pub-sub frameworks, the destination of the events in our case is a single storage system, we do not have a need for multiple destinations. The underlying storage itself may be distributed. So we don't need to explicitly implement forwarding for the sake of load-balancing, scalability etc.

1.3.3 Data storage

To enable site-specific implementation of the storage of application data, we will use a pluggable storage to store all the events coming across from AMs & containers of various applications. Having the event-data in a centralized storage is desired both for scalable serving and for centralized analysis. There are few requirements here:

- Pluggable system to cater to small and large installations
- The primary storage that will be shipped needs to enable structured data for enabling near real-time querying (with indexes etc) **and** rendering as well as better analytics (without say running batch jobs).

For simple and small deployments, we do not want to put the burden of installing one more system as a YARN dependency. With YARN being the lowest level of the architecture, forced addition of more systems underneath is a non-starter. So an embedded implementation that works on top of local file-system based storage will be shipped as default. For enabling a

structured system that scales up, we are looking at one of the basic implementations to be based on Level-DB that can work against local-system and scale up by integrating with a nosql database like HBase.

1.3.4 Serving

Clients are served by the web-services layer in the Application Timeline Server (Previously AHS). The web-service acts as the serving layer that bridges the gap between the storage-layer and the client which is rendering the information in various ways possible. We can imagine implementing a RPC layer too (as with other things in YARN) but it isn't a priority as of now.

We now describe the APIs exposed by the serving layer.

1.3.4.1 Application Timeline REST API for retrieval

An example API follows:

HTTP Request I

```
Request: GET http://<host>:<port>/ws/v1/apptimeline/<entityType>
                [?primaryFilter=<name>:<value>]
                [&secondaryFilter=<name>:<value>[,<name>:<value>]]
                [&windowStart=<startTime>][&windowEnd=<endTime>]
                [&limit=<limit>][&fields=<field>[,<field>]]

Description: Retrieves entities of a given type with entity start time in a time
window matching given filters.
```

- With the primaryFilter option, the request performs an indexed retrieval of entities matching the primary filter. Only data matching the primary filter will be scanned. Only one primary filter may be specified.
- With the secondaryFilter option, the request only returns entities matching all of the specified secondary filters, though unmatching entities may be scanned and rejected. Matching in this case means having a primary filter or other info with the specified value for the given name.
- The time window is specified with the windowStart and windowEnd options. If windowEnd is not specified, MAX_LONG will be used. If windowStart is not specified, all earlier entities will be processed unless a limit is specified.
- With the limit option, one can request the N most recently active entities or, if an windowEnd is specified, the entities active before a given end time (inclusive). The limit defaults to 100.
- The fields option controls which information to retrieve about an entity. Possible fields are EVENTS, RELATEDENTITIES, PRIMARYFILTERS, OTHERINFO, and LASTEVENTONLY

(retrieves only the most recent event). These may be specified in any combination, with EVENTS overriding LASTEVENTONLY. The default when unspecified is to retrieve all fields.

Example

```
Request: GET http://<host>:<port>/ws/v1/apptimeline/TEZ_VERTEX_ID?
                fields=events,relatedentities,otherinfo
Response:

{
  "entities": [
    {
        "entity": "vertex_1390516007863_0003_1_00",
        "entitytype": "TEZ_VERTEX_ID",
        "starttime": 1390516418713,
        "events": [
            {
                "timestamp": 1390516419283,
                "eventtype": "VERTEX_FINISHED",
                "eventinfo": {}
            },
            {
                "timestamp": 1390516411332,
                "eventtype": "VERTEX_STARTED",
                "eventinfo": {}
            },
            {
                "timestamp": 1390516410986,
                "eventtype": "VERTEX_INITIALIZED",
                "eventinfo": {}
            }
        ],
        "relatedentities": {
            "TEZ_TASK_ID": [
                "task_1390516007863_0003_1_00_000000"
            ]
        },
        "primaryfilters": {},
        "otherinfo": {
            "numTasks":1,
            "status":"SUCCEEDED",
            "vertexName":"Reducer 2",
            ...
        },
    },
    ...
  ]
}
```

The response contains an array of entity information objects which always contains an entity,

entity type, and start time. These are sorted in start time order, decreasing. Additional information is filled in based on which fields are requested. The events array contains events sorted in decreasing time order.

HTTP Request II

```
Request: GET http://<host>:<port>/ws/v1/apptimeline/<entitytype>/
                <entityId>[?fields=<field>[,<field>]]

Description: Retrieves information about a given entity of a given type.
```

Example

```
Request: GET http://<host>:
<port>/ws/v1/apptimeline/TezTaskId/task_1381946403170_0003_3_00_000000?
                fields=events,relatedentities,otherinfo
Response:

{
  "entity": "task_1390516007863_0003_1_00_000000",
  "entitytype": "TEZ_TASK_ID",
  "starttime": 1390516418713,
  "events": [
      {
          "timestamp": 1390516419282,
          "eventtype": "TASK_FINISHED",
          "eventinfo": {}
      },
      {
          "timestamp": 1390516418713,
          "eventtype": "TASK_STARTED",
          "eventinfo": {}
      }
  ],
  "relatedentities": {
      "TEZ_TASK_ATTEMPT_ID": [
          "attempt_1390516007863_0003_1_00_000000_0"
      ]
  },
  "primaryfilters": {},
  "otherinfo": {
      "status":"SUCCEEDED",
      "timeTaken":472,
      "counters": ...,
      ...
  }
}
```

HTTP Request III

```
Request: GET http://<host>:<port>/ws/v1/apptimeline/<entityType>/events
                [?entityID=<entityID>[,<entityID>]]
                [&windowStart=<startTime>][&windowEnd=<endTime>]
                [&limit=<limit>]
                [&eventType=<eventType>[,<eventType>]]


Description: Retrieves events for one or more given entities of a given type.
```

- A time window for the events can be specified with the windowStart and windowEnd options.
- The events can be limited to given event types (though events of other types may be scanned).
- A limit on number of events per entity can be specified. The limit defaults to 100.

Example

```
Request: GET http://<host>:<port>/ws/v1/apptimeline/
            TEZ_APPLICATION_ATTEMPT/events?
            entityId=tez_appattempt_1390516007863_0001_000001,
               tez_appattempt_1390516007863_0002_000001
Response:

{
  "events": [
    {
        "entity": "tez_appattempt_1390516007863_0001_000001",
        "entitytype": "TEZ_APPLICATION_ATTEMPT",
        "events": [
            {
                "timestamp": 1390516132507,
                "eventtype": "AM_STARTED",
                "eventinfo": {}
            },
            {
                "timestamp": 1390516131185,
                "eventtype": "AM_INITIALIZED",
                "eventinfo": {}
            }
        ]
    },
    {
        "entity": "tez_appattempt_1390516007863_0002_000001",
        "entitytype": "TEZ_APPLICATION_ATTEMPT",
        "events": [
            {
                "timestamp": 1390516277437,
                "eventtype": "AM_STARTED",
                "eventinfo": {}
            },
            {
                "timestamp": 1390516276322,
                "eventtype": "AM_INITIALIZED",
                "eventinfo": {}
            }
        ]
    }
  ]
}
```

1.3.5 Rendering/User interfaces

The client library can use the APIs exposed by the serving layer and build framework specific UIs. One obvious choice is to build it in JavaScript, serve them somewhere and let users access them.

To facilitate the ease of deployment of the framework specific user-interfaces, *Application Timeline Server* enables administrators to optionally install rendering-libraries for *specific versions* of specific *well-known*, commonly used applications/frameworks.

- As should be obvious, any changes in the UI at run-time by the application-developer need ratification by admins before deployment. This comes in the way of YARN's application agility requirement - other than the UI, developers are free to, and even encouraged, to evolve applications in an agile manner.
- Due to security reasons, we cannot let arbitrary code to be installed dynamically in the ATS. Because of this, at any point of time, ATS only allows specific versions for each framework to be installed by the admins. Managing lot of versions will be a burden to admins and we don't forsee more than a few versions being supported.
- ApplicationMasters can set the finalTrackingUrl as part of Application unregistration to this URL so that users are transparently redirected to the right UI on the ATS.

As mentioned previously, other than the admin-ratified & installed rendering libraries, YARN neither encourages nor blocks users from writing their own user-interfaces making use of the serving APIs, and setting up their own daemons. YARN cannot explicitly link to them for reasons of security - AMs can set finalTrackingURLs pointing to these locations, but the content may be filtered through the central web-application-proxy for security.

1.3.6 Miscellaneous

- **Time**: For the timeline to make any sense, the timestamps all need to be globally ordered and so will require us to depend on synchronized clocks.
- **Use beyond AM/Containers**: One can imagine events flowing out of YARNClient, framework clients like MR JobClient, or even upper layers like Pig, Hive, Oozie etc. We provide the API, but leave correlation across entities and their visualization to the framework specific UIs.

1.4 Merging the generic and per-framework data

It is very conceivable to use this same system with ResourceManager as the client/source of events for persisting generic historical data.

- ResourceManager can add extra fields which indicate the lifetime of a container and its allocation - e.g. container start, finish etc.
- It can add queue, user, application_type, application_name, application_id, application_state, container_id, rack and node_id etc too.

- These fields added by the RM can in addition be indexable by the storage layer for faster access of generic information.

But we are keeping the scope of the integration of generic data into this framework out of this document and for later.