Timeline Reader Design

Varun Saxena

This document is intended to cover initial design for timeline reader.

Introduction

Timeline Reader will be a single daemon(in the initial phase) which will serve as a REST endpoint for client queries. Reader will serve these requests by either querying the collector for live apps or by querying the backend storage (HBase/FileSystem, etc.) for completed apps. No internal state will be maintained by readers hence no state store is necessary. Readers will be the only interface for querying by clients.

Implementation

Reader will launch a web-server on a configurable host and port to serve REST calls from clients. It will create a response by either querying the relevant collector(which is serving the application queried) if it is a live or in-progress app or it will directly get the response from backing storage

In phase 1, we will support three of the existing ATS v1 APIs' i.e. getEntities, getEntity and getEvents.

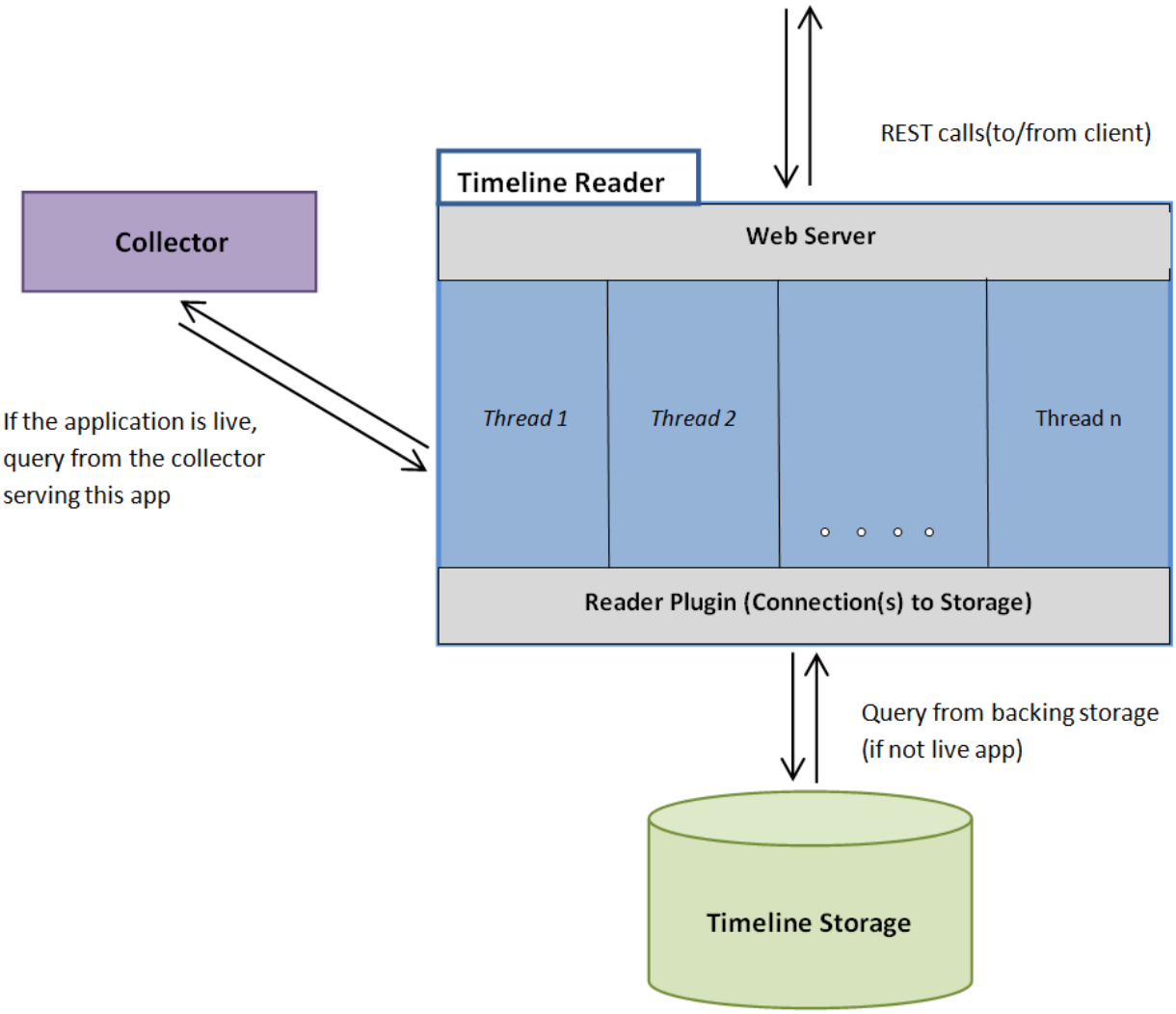
Timeline Reader needs to apply ACLs' as well so that an unauthorized user does not get access to application date he is not supposed to view.

Diagram on the next page gives an overview of reader. Multiple threads will process REST queries/requests coming from the client. Reader also provides an interface for reader plugin which can be implemented and used to configure a custom backing storage implementation. Certain implementations using Phoenix/HBase, File system, etc. will be implemented. Number of connections opened to query the backing store depends on store implementation.

Query from collector for live app will be a synchronous operation

Reader will create a JSON response based on the response received from collector or backing store and send it back to the client.

Reader overview



Multiple options to determine whether an app is active

As mentioned above, reader will query a live or in-progress app from the collector for the application queried by the client. But for this to happen, Reader needs to first ascertain if an application is active or not. For this, we have two options :

1. NM notifies Reader whenever a collector is bound on launching of AM.

- NodeManager will notify Reader whenever a collector service is bound same as it notifies RM right now.
- Reader will cache this collector info alongwith the application ID which is being served by the collector.
- NM(s) and Reader will exchange a HB. During this collector info can be sent by NM and reader can refresh it. This will be useful for handling failure scenarios(i.e. when reader goes down). If NM goes down, collector info is anyways refreshed via HB from RM.

Pros of this approach :

Reader will immediately be able to find out which collector it should query.

Cons of this approach :

As timeline reader does not have an associated state store, collector list will be lost if reader goes down. Complexity hence will be added to make sure NM resyncs the collector info when reader goes down. Moreover, when we move from a single reader to a pool of readers, this approach will not be easily scalable. It will largely depend on how we design pool of readers.

2. Reader queries RM to find the collector address if app is live.

- RM stores collector info and also stores it in state store so the collector info is available across restarts.
- Reader can hence make a RPC call to RM to get collector address based on application id.
- Upon receiving the response, reader checks if collector address is present. If it is present, collector is queried. Otherwise reader will query backing storage.
- If response from collector is unsuccessful, backing storage will also be tried. As it is possible that an app which was active at the time of RM query, might have completed by the time collector was queried.

Pros of this approach :

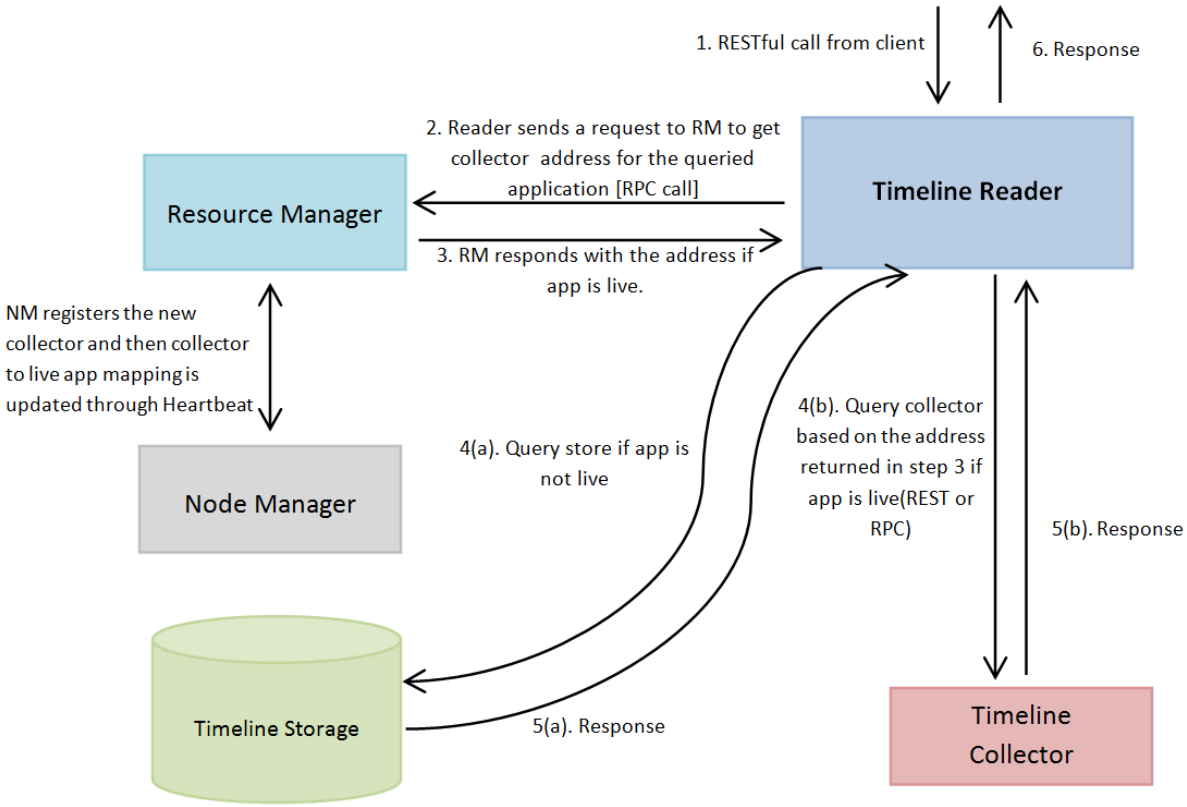
This will be useful when multiple readers are added. Reader(s) just need to know the RM address. Also collector info is preserved across RM restarts. Hence, collector info will be available for reader at all times too.

Cons of this approach :

A RPC call will have to be made on each client query.

Based on pros and cons of each approach, second approach is preferable and hence will be implemented.

This approach is further explained in diagram below. Please note Timeline Collector will need to start a Reader Service to serve the request from reader. We need to decide whether it will be a REST call or a RPC call.



Query APIs'

In phase 1, we will support 3 of the ATS v1 APIs' i.e. getEntity, getEntities and getEvents.
Primary filters will no longer be supported. Now "filters" will act similar to secondary filters in v1
and will match with "info" in TimelineEntity.

Other query parameters such as limits, fields, etc. will remain the same as v1. Fields though will be according to new data model.

For instance if client wants to get all the applications run by a user, it will match with entities of type YARN_APPLICATION with user passed in filters being matched in "info" field of entity stored.

How an entity will be stored depends on store implementation.

Major classes

TimelineReaderServer : Main starting point for single node reader. Launches the web server and store implementations.

TimelineWebServices : Acts as REST endpoint for user queries.

REST URLs' will be as under :

For getEntities, http://<reader host>/ws/v2/timeline/{entityType}

For getEntity, http://<reader host>/ws/v2/timeline/{entityType}/{entityId}

For getEvents, http://<reader host>/ws/v2/timeline/{entityType}/events

TimelineReaderDataManager : TimelineWebServices parses the request and forwards it to TimelineReaderDataManager. This class processes the request, gets data from backend storage and will apply ACLs' and limits on the result.

TimelineReaderDataManager instantiates the store implementation.

TimelineReader : TimelineReader is the interface for accessing backing storage. This will be implemented by respective store implementation to fetch data from storage.

Major requirements for Reader

- ✓ YARN-3047 : Start reader with basic request serving mechanism. Involves starting up of a web server and handling REST requests. Also involves additions in script to start up the reader.
- ✓ YARN-3051 : Define the interface for store implementation. Would involve deciding basic query APIs' to be supported. As part of this JIRA, FS based implementation will also be done.
- ✓ Apply ACLs'. This can either be done in TimelineReaderDataManager or in the store. Need to discuss this as store implementation such as LevelDb limit the number of records retrieved which can be further reduced when ACLs' are applied. Will raise a separate JIRA for this.
- ✓ Query RM to get collector address to query live apps. Explained in the document above. Will raise a new JIRA for this.
- ✓ Reader Service in collector to serve queries for live apps coming from reader.
- ✓ Support certain YARN CLI commands. For commands such as "yarn application", if the app was not found in RM, we used to query AHS. To support these commands now, we will have to query timeline reader. This will now be done via REST instead of RPC. Will raise a new JIRA to handle it.
- ✓ YARN-3118 : Will implement multiple reader instances. This would involve load balancing. Need discussion on the design. Probably can have a single master daemon which will launch readers on the fly or can have a predefined pool, one of which master will choose from. Probably latter.
- ✓ Phoenix/Hbase based store implementation.