YARN Rolling Upgrades

Arun C. Murthy, Bikas Saha, Hitesh Shah, Jason Lowe, Siddharth Seth, Vinod Kumar Vavilapalli

Requirements

- Possible to upgrade a YARN cluster without brining the entire cluster down.
- Possible to upgrade YARN independent of HDFS/Common
- Within a major version, addition of new functionality/APIs should be possible, without breaking existing clients.
- Within a major version, a client compiled against a specific minor version should continue to work with all YARN versions which are greater than or equal to this minor version.

Current Features in the RPC layer to support this

- PB used as the wire format – has in-built support for unknown fields, required and optional fields.
- Versioning of Protocols – Protocols are annotated with a version number. When connecting to a server, the client must talk to the same version of the protocol on the server.
- Handshake between the server and client, which is used to share the signature methods supported on the interface. (This includes no information about required or optional fields). TODO: This is not supported by the ProtobufRPCEngine. Is this something that we'll need ?
- Support for multiple version of a protocol to be served on a single RPC server.

Considerations

1) **Data persisted by YARN** – An upgraded version of a YARN daemon should continue to work with data persisted by a previous version of the daemon. This includes data persisted for RM restart, the Aggregated Log format, NodeManager directory hierarchy. (Not including history server persistence since that is part of MapReduce)
2) **Handling various upgrade scenarios**
    a. With no API changes. Listed in the Upgrade Scenario section.
    b. With compatible API changes.
3) **API Changes –** detailed below.
4) **Changes related to security** – As an example, addition of a new field to a TokenIdentifier (AMToken). Currently, old NodeManagers will not be able to validate tokens issues by a new ResourceManager if a field is added.
5) **Upgrading the MapReduce runtime** (not necessarily part of YARN itself) – This comes into play since the MapReduce runtime is installed along with YARN. On a partially upgraded cluster, a MapReduce AM can end up using upgraded YARN libraries, which may require a specific version of YARN.

API Changes

If not mentioned under the compatible section – the change is likely to be incompatible (and requires a bump in the protocol version)

Considered to be compatible (no protocol version change)

- Addition of a new optional field to a method.
- Addition of a new method. (TODO: Upgrade mechanisms will likely play a role in this (see section on steps to upgrade a YARN cluster), TODO: Impact on behavior – if an older version of an NM does not know how to interpret a field, and hence cannot act on it – is this considered compatible since the actual behavior varies from the expected behavior)

Incompatible changes (protocol version changes)

- Changing a field from being optional to required.
- Deletion of a field
- Deletion of a method
- Change in API semantics

Guidelines for changing APIs

TBD. Similar to the 'Considered compatible' part of the API Changes section.

Steps to upgrade a YARN cluster

The version of a YARN cluster can be considered to be the lowest version of any (YARN) component running on the cluster. This, however, is not enforced in any way by YARN itself. TODO: Does it make sense to include a YARN version in all YARN APIs, which is independent of the protocol being used? This would allow clients to check against the RM itself to see whether they can run.

1) Upgrade YARN daemons
    a. Typically, the first step will be to upgrade the ResourceManager.
    b. Upgrade NodeManagers in batches based on the cluster size. Useful when only NMs need to be upgraded. Until this point, an RM upgrade / restart effectively kills all containers on the cluster, and staggered NM restarts are not extremely useful.
2) Upgrade MapReduce libraries once the entire YARN cluster is on a version required by an upgraded MapReduce application. (MapReduce needs to be upgraded independent of YARN)
3) Likewise, YARN may need to be upgraded independent of HDFS and Common.

Upgrade Scenarios

1) Bug fix upgrade to the RM only – without any API changes. Covered by RM restart.
2) Bug fix upgrades to the NM only – without any API changes. Currently, all containers running on the NM will have to 'complete'. In the future, we may want to support preserving containers across NM restarts – especially for

longer running containers. For now, a mechanism is required for the NM upgrades to be managed in a controlled manner – Drain with a timeout. (Drain decommission)
3) RM/NM upgrades which involve 'compatible' API changes.

Required Changes

1) Introduction of a 'drain/maintenance state' on the NodeManager – When in this mode, the NM does not accept any new 'startContainer' requests. It does, however, continue to accept stopContainer and getContainerStatus requests. In addition, while in this state, a standard exception should be thrown when a startContainer call is received. A similar API may not be required on the RM considering the current approach used for RM restart.
2) An API on the ResourceManager to set specific NodeManagers into this 'maintenance state'. When put into this state, the RM will stop allocating containers on these NodeManagers. In addition, AMs will be informed of this changed state, so that they can take appropriate action.
3) TODO: What changes are required to support multiple versions of a YARN interface on the same RPC server?
4) Versioning of all persisted data.
    a. Log formats are already versioned.
    b. Data persisted for RM restart may require versioning. We can likely get away without doing this – since Alternately, this can be handled by upgrade scripts to convert from one data format to another – or by the upgraded RM itself.
    c. NodeManager directory structures – the same applies. Either have upgrade scripts, or have the upgraded NM support / translate from the old structure. This is not so important at the moment since NMs are not setup to recover state.
5) TokenIdentifier serialization should consider unknown fields.
6) For components like ContainerTokens – which are shared between the RM and NM via user code in the AM, consider changing the fields to be ByteBuffers (with versioned data?), which can be interpreted by YARN itself. The AM does not need to know how to pull information from such fields, and should not rely on information available in such fields.
7) Define a clear API / error for method invocations by a client which are currently not supported by the server. This is already present as a RPC layer exception. Do we want to make additional information available – like the version of the protocol.
8) Handling of Required fields – This is related more to YARN-386 (API changes) than to rolling upgrades. Implement this in the server side API instead of in the protobuf layer. Use case to support this: Changing the type of a field, effectively means adding a new optional field. Older clients would set the old field, newer ones would set the new field. The server is only interested in one of the two fields being set.

9) Changes to YARN to make use of the RPC layer functionality to check if a method is supported. The RPC layer will throw an appropriate exception. YARN client libraries may however choose to throw a different exception.
10) Handling exceptions on the Client RPC layer (ClientPBImpl) – Handling of new exceptions added to the remote end, even if they sub-class YarnRemoteException.

Concerns

MapReduce being deployed as part of the YARN installation – The classpath for the AM is constructed based on the environment on the NodeManager, which launches the AM (specifically HADOOP_PREFIX and related variables). If a MR job is started during an upgrade process, it may end up using an API, which is not available on all NodeManagers. It needs to know how to handle this. Similarly for HDFS and Common libraries.
This can be avoided by deploying MapReduce independent of YARN, or shipping MapReduce libraries from the client itself via the DistributedCache.

Interaction of YARN with deployed versions of HDFS and Common – YARN (and possibly client code) include the deployed version of YARN, Common and HDFS as part of their classpath. All the classes may not be loaded by the JVM up-front. In case of an upgrade, a process may end up running with a mix of classes from different versions (Pulling JARs from under a running Java process).
Could be avoided by not using symbolic links to point to the current version. Instead have multiple deployments, with environment variables (HADOOP_PREFIX) etc. updated to point to the latest install. In general, avoid changing JARs which are being used by a running process.

Use Cases

- Addition of APIs to control log-aggregation
- Addition of fields to TokenIdentifiers – e.g. a username to the AppToken
- All of the Allocate API changes under YARN-397

Longer Term

- Avoid killing containers during an NM restart process.
- Recover distributed cache etc. across restarts. Some of this may be considered during work-preserving RM restarts.