A YARN-913 service registry

Steve Loughran and Sanjay Radia
Revised September 3, 2014

Introduction and concepts

This document specifies a YARN service registry to address a problem: *how can clients talk to YARN-deployed services and the components which form such services?*

Service registration and discovery is a long-standing problem in distributed computing, dating back to Xerox's Grapevine Service. This proposal is for a registry for locating distributed applications deployed by YARN, and determining the binding information needed to communicate with these applications.

Definitions

Service: a potentially-distributed application deployed in —or reachable from— a Hadoop YARN cluster. Examples: Apache HBase, Apache hcatalog, Apache Storm. Services may be short-lived or long-lived.

Service Class: the name of a type of service, used as a path in a registry and matching the DNS-compatible path naming scheme. Examples: org-apache-hbase, org-apache-hcatalog

Component: a distributed element of a service. Examples: HBase master nodes, HBase region servers and HBase REST servers.

Service Instance: A single instance of an application. Example, an HBase cluster "demo1". A service instance is running if the instances the components which for the service are running. This does not imply “live” in the distributed computing sense, merely that the process are running.

Component Instance: a single instance of a component within a service instance. Examples: an HBase master node on host `"rack1server6"` or a region server on host `"rack3server40"`.

Endpoint: one means of binding with a service instance or a component instance. Examples: HBase's Apache Zookeeper binding, a Java JMX port on a region server, a Web UI on an HBase Master, and the REST API of an HBase REST component instance. Endpoints may be *internal:* for use within the service instance, or *external*: for use by clients of a service instance.

Service Record: a record in the registry describing a service instance or a component instance, including listing its endpoints.

YARN Resource Manager, "RM": the YARN component which allows client applications to submit work (including requests to deploy service instances) to a YARN cluster. The RM retains state on all running applications.

YARN Application: An application deployed via YARN. Every application instance has a unique application ID.

YARN Application Master, "AM": the application-specific component which is scheduled and deployed by the RM. It has the duty of maintaining the internal state of the application, including requesting and managing all other component instances of this application instance. The YARN RM will detect the failure of the AM, and respond by rescheduling it.

YARN Container: An allocation of resources, including CPU and RAM, for a component instance. The AM has the responsibility of requesting the containers its components need, and building the commands to instantiate its component instances onto allocated containers. Every allocated container has a unique container ID.

The binding problem

Hadoop YARN allows applications to run on the Hadoop cluster. Some of these are batch jobs or queries that can managed via Yarn's existing APi using its jobId. In addition YARN can deploy long-lived services instances such a pool of Apache Tomcat web servers or an Apache HBase cluster. YARN will deploy them across the cluster depending on the individual each component requirements and server availability. These service instances need to be discovered by clients; traditionally their IP added is registered in DNS or in some configuration file —but that is not feasible in YARN-deployed applications when neither the hostname nor network ports can be known in advance.

As a result there is no easy way for clients to interact with dynamically deployed applications.

YARN supports a rudimentary registry which allows YARN Application Masters to register a web URL and an IPC address. but is not sufficient for our purposes since it It does not allow any other *endpoints* to be registered —such as REST URLs, or zookeeper path or the endpoints of the tasks that the Application Master executes. Further, information that can be registered is mapped to the YARN application instance —a unique instance ID that changes whenever a YARN application is started. This makes it impossible to resolve binding information via a static reference to a named service, or to even probe for the existence of a service instance which is not currently live.

Use Cases

Service Name Examples

1. Core Hadoop services.
   These may be deployed statically, dynamically via an account with the permissions to write to the `/services` path, or even registrations of remote services accessible from within the Hadoop cluster
   a. `/services/hdfs  // the list of NN(s) of HDFS`
   b. `/services/yarn  // The RM of the Yarn service`
   c. `/services/oozie`
2. Yarn-deployed services belonging to individual users.
   a. `/users/joe/org-apache-hbase/demo1`
   b. `/users/joe/org-apache-hbase/demo1/components/regionserver1`

Registration Use Cases

1. A Hadoop core service that is not running under YARN example: HDFS) can be registered in for discovery. This could be done by the service or by management tools..
2. A long-lived application deployed by YARN registers itself for discovery by clients. The registration data is intended to outlive the application master, and perhaps the lifespan of a single deployment of the service instance.
3. Component instances of a service register themselves, publishing internal binding information, such as JMX ports.
4. A YARN-deployed application can bind to dependent service instances both static and dynamic.
   Example: a Tomcat web pool binding to the dynamic HBase service instance "`/users/joe/services/hbase/demo1`".
5. Component Instances use the registry to bind to an internal endpoint of their application master, to which they heartbeat regularly.

Unsupported Registration use cases:

1. A short-lived Yarn application is registered automatically in the registry, including all its containers. and unregistered when the job terminates. Short-lived applications with many containers will place excessive load on a registry. All YARN applications will be given the option of registering, but it will not be automatic —and application authors must be advised against registering short-lived containers.

Lookup Use Cases

1. A client application looks up a dynamically deployed service instance whose user, service class and instance name is known, e.g.

`"/users/joe/services/hbase/demo1",` and retrieves the information needed to connect to the service
2. A client application looks up a statically deployed Hadoop service
   Example: `"/services/hdfs"`.
3. An Application Master enumerates all registered component instances, discovers their listed JMX ports, and, inits own web UI, offers links to these endpoints.
4. A user connects to a private HBase service instance at `"/users/joe/services/hbase/demo1"`
5. A user connects to the cluster's HBase service at `"/services/hbase".`
6. A user looks up the binding information to a remote Hadoop cluster's filesystem at `"/net/cluster4/services/hdfs"`. The registration information includes the `webhdfs://` URL for the remote filesystem
7. A user lists their HBase service instances:
   `ls /users/joe/services/hbase`
8. User finds all Hbase services in the cluster:
   `find -endpointField.api=org.apache.hbase`
9. Possibly in future: looking up a service via DNS.

This registry proposal is intended to support these use cases by providing a means for applications to register their service endpoints, and for clients to locate them.

Key Requirements of a Service Registry

- Allow dynamic registration of service instances
  - YARN deployed services instances must be able register their bindings and be discovered by clients.
  - Core Hadoop service instances must be able to register their service endpoints.
  - The binding must be upgradable if the service moves or in case if HA fails over.
- A service instance must be able to publish a variety of endpoints for a service: Web UI, RPC, REST, Zookeeper, others. Furthermore one must also be able register certificates and other public security information may be published as part of a binding.
- Registry service properties:
  - The registry must be highly available.
  - Scale: many services and many clients in a large cluster. This will limit how much data a service can publish.
  - Ubiquity: we need this in every YARN cluster, whether physical, virtual or in-cloud.
- Must support hierarchical namespace and names. The name convention must match that of DNS so that we have the option of accessing the namespace via DNS protocol at a later phase of the project.
- Registry API Language/Protocols
  - Cross-language: independent of any language; client language != service
  - REST API for reading registry data
- Access Control:

    - Read access for all
    - Write is restricted so that squatting and impersonation can be avoided.
- Remote accessibility: supports remote access even on clusters which are only reachable via Apache Knox, or hosted in cloud environments.

Non-Requirements

- The registry is not intended for liveness detection, leader-election or perform other "shared consensual state" actions for an application itself, other than potentially sharing binding information between component instances.

- The registry is not intended to be a store for arbitrary application state, or for publishing configuration data other than binding information to endpoints offered by a service and its components. Such use would overload the registry and rapidly reach limits of what Zookeeper permits.

Architecture

We propose a base registry service that binds string-names to records describing service and component instances. We plan to use ZK as the base name service since it supports many of the properties, We pick a part of the ZK namespace to be the root of the service registry (say "`<ZK-ROOT>/serviceRegistryRoot`").

On top this base implementation we build our registry service API and the naming conventions that Yarn will use for its services. The registry will be accessed by the registry API, not directly via ZK - ZK is just an implementation choice (although unlikely to change in the future).

1. Services are registered by binding a ***path*** to a value called a ***Service Record***. Paths are hierarchical and use `"/"` as the root and `"/"` as the separator.
2. Service records are registered as persistent znodes. This ensures that the record remains present during planned and unplanned outages of the service, on the assumption that client code is resilient to transient outages.
3. Each service instance's service record lists the endpoints for its various protocols exported by that service instance.
4. For each protocol endpoint it must contain
    a. The *protocol* name including: Web, REST, IPC, zookeeper. (type:string)
    b. Its *address*: the specific details used to locate this endpoint
    c. Its *addressType*. This is the format of the binding string. (URL, ZK path, hostname:port pair). For the predefined protocols, we will define what format the binding string MUST be. Example: `protocol==REST` means binding type is [URL], `protocol==IPC` binding uses the addresstype [hostname, port]

    d. The *api*. This is the API offered by the endpoint, and is application specific. example: "org.apache.hadoop.namenode", "org.apache.hadoop.webhdfs"

5. Endpoints may be *external* —for use by programs other than the service itself, and *internal* —for connecting components within the service instance. They will be listed in different sections of the Service Record to distinguish them.
6. Core services will be registered using the following convention: `/services/<servicename>` e.g. "`/services/hdfs`".
7. Yarn services will registered using the following convention:
    a. `/users/<username>/<serviceclass>/<instancename>`
    b. Component instances will be registered under `/users/<username>/<serviceclass>/<instancename>/components/<componentname>`
    c. Every service instance must have a name that ise unique for all instances of that service class belonging to that user. Example "`hbase-1`". That is, the (user, service class, instance name) must be unique. This is the *service instance.*
    d. Each of the user's services must have unique service class names,
    e. Each component instance must have a name that is unique for all components in a service instance. For a YARN-deployed application, this can be trivially derived from the container ID.

    The requirements for unique names ensures that the path to a service instance or component instance is guaranteed to be unique, and that all instances of a specific service class can be enumerated by listing all children of the service class path.

8. Service Registry API: Both a Java API and Rest API will be provided (see below for the API). A Hadoop cluster *may* host one or more REST servers offering read access to the registry data. This may be in the YARN RM process, or standalone

Registry Model

Service entries MUST be persistent —it is the responsibility of YARN and other tools to determine when a service entry is to be deleted.

Path Elements

All path elements MUST match that of a lower-case entry in a hostname path as defined in RFC1123; the regular expression is:

```
([a-z0-9]|([a-z0-9][a-z0-9\-]*[a-z0-9]))
```

This policy will ensure that were the registry hierarchy ever to exported by a DNS service, all service classes and names would be valid.

A complication arises with user names, as platforms may allow user names with spaces, high unicode and other characters in them. Such paths must be converted to valid DNS hostname entries using the punycode convention used for internationalized DNS.

Service Record

A Service Record has some basic information and possibly empty lists of internal and external endpoints.

Service Record:

A Service Record contains some basic informations and two lists of endpoints: one list for users of a service, one list for internal use within the application.

.

Name	Description
`id: String`	YARN application or container ID (missing/empty for statically deployed services).
`description: String`	Human-readable description.
`registrationTime: long`	Registration time as a `System.getTimeMillis()` value seen at the service.
`persistence: int`	Persistence policy.
`external: List<Endpoint`	A list of service endpoints for external callers.
`internal: List<Endpoint>`	A list of service endpoints for internal use within the service instance.

The `persistence` attribute defines when a record *and any child entries* may be deleted.

Policy #	Name	Policy
0	Permanent	The record persists until removed manually.
1	Cluster Restart	Remove when the YARN cluster is restarted. This does not mean on HA failover; it means after a cluster stop/start.
2	Application	Remove when the YARN application defined in the `id` field terminates.
3	Application Attempt	Remove when the current YARN application attempt finishes.
4	Container	Remove when the YARN container in the ID field finishes

5	Ephemeral	Automatic deletion when the session is closed/times out.

The policies which clean up when an application, application attempt or container terminates require the `id` field to match that of the application, attempt or container. If the wrong ID is set, the cleanup does not take place —and if set to a different application or container, will be cleaned up according the lifecycle of that applicatin.

Endpoint:

Name	**Description**
`addresses: List<List<String>>`	a list of address tuples t uples whose format depends on the address type
`addressType: String`	format of the binding
`protocol: String`	Protocol. Examples: `http, https, hadoop-rpc, zookeeper, web, REST, SOAP, ...`
`api: String`	API implemented at the end of the binding

All string fields with have a limit on size, to dissuade services from hiding complex JSON structures in the text description.

Field: Address Type

The `addressType` field defines the string format of entries.

(We could eliminate this and mandate the format for every protocol instead, as there should be a single format for every protocol)

Having separate types is that tools (such as a web viewer) can process binding strings without having to recognize the protocol.

Format	**binding format**
`url`	`[URL]`
`hostname`	`[hostname]`
`inetaddress`	`[hostname, port]`
`path`	`[/path/to/something]`

zookeeper	[quorum-entry, path]

An actual zookeeper binding consists of a list of hostname:port bindings –the quorum— and the path within. In the proposed schema, every quorum entry will be listed as a 3-tuple of [hostname, port, path]. Client applications do not expect the path to de be different across the quorum. The first entry in the list of quorum hosts MUST define the path to be used by all clients. Later entries SHOULD list the same path, though clients MUST ignore these.

New Address types may be defined; if not standard please prefix with the character sequence "x-".

Field: API

APIs may be unique to a service class, or may be common across by service classes. They MUST be given unique names. These MAY be based on service packages but MAY be derived from other naming schemes:

Examples of Service Entries

Here is an example of a service entry for a YARN-deployed tomcat application.

After creation and registration of the application, the registry looks as follows:

```
/users
  /devteam
   /tomcat
    /components
     /container-1408631738011-0001-01-000002   *
     /container-1408631738011-0001-01-000001   *
```

The `/users/devteam/tomcat` service record describe the entire application. It lists the YARN application ID, and exports the URL to a load balancer. It's persistence is 0: "manual deletion"

```
{
  "registrationTime" : 1408638082444,
  "id" : "application_1408631738011_0001",
  "description" : "tomcat-based web application",
  "persistence" : "0",
  "external" : [ {
    "api" : "www",
    "addressType" : "uri",
    "protocolType" : "REST",
    "addresses" : [ [ "http://loadbalancer/" ] [ "http://loadbalancer2/" ] ]
  } ],
  "internal" : [ ]
}
```

The service instance is built from two component instances, each described with their container ID converted into a DNS-compatible hostname. The entries are marked as ephemeral. If the entries were set within the container, then when that container is released or if the component fails, the entries will be automatically removed. Accordingly, it's persistence policy is declared to be "1" —ephemeral

```
/users/devteam/tomcat/components/container-1408631738011-0001-01-000001 (ephemeral)
{
  "registrationTime" : 1408638082445,
  "id" : "container_1408631738011_0001_01_000001",
  "persistence" : "1",
  "description" : null,
  "external" : [ {
    "api" : "www",
    "addressType" : "uri",
    "protocolType" : "REST",
    "addresses" : [ [ "http://rack4server3:43572" ] ]
  } ],
  "internal" : [ {
    "api" : "jmx",
    "addressType" : "host/port",
    "protocolType" : "JMX",
    "addresses" : [ [ "rack4server3", "43573" ] ]
  } ]
}
```

The component instances list their endpoints: the public REST API as an external endpoint, the JMX addresses as internal.

```
/users/devteam/tomcat/components/container-1408631738011-0001-01-000002 (ephemeral)
{
  "registrationTime" : 1408638082445,
  "id" : "container_1408631738011_0001_01_000002",
  "persistence" : "1",
  "description" : null,
  "external" : [ {
    "api" : "www",
    "addressType" : "uri",
    "protocolType" : "REST",
    "addresses" : [ [ "http://rack1server28:35881" ] ]
  } ],
  "internal" : [ {
    "api" : "jmx",
    "addressType" : "host/port",
    "protocolType" : "JMX",
    "addresses" : [ [ "rack1server28", "35882" ] ]
  } ]
```

```
}
```

This information could be used by the (hypothetical) load balancer to enumerate the components and build a list of component instances to dispatch requests to. Similarly, a management application could enumerate all available component instances and their JMX ports, then connect to each to collect performance metrics.

Registry API

Here is the proposed registry API as seen from a Java application. The API is a thin layer above the ZK operations, essentially building up paths, reading, writing and updating entries, and enumerating children. The REST API is implemented inside a server and use this same API to implement its REST API.

The exceptions that are listed are only a subset of possible exception —the interface merely lists those that have special meaning.

All write operations must assume that they are communicating with a registry service with the consistency view of a Zookeeper client; read-only clients must assume that some operations may not be immediately visible to them.

Note that any non-ephemeral node in the namespace tree can be both a directory and service record.

The core interface RegistryOperations, extends the YARN service interface ... it is intended to be long lived. When the service is stopped, all ephemeral entries will expire.

The API does not itself expose security options to the caller. The implementation must create nodes which are writeable only by the owner, and readable by anyone.

```
/**
 * Registry Operations
 */
public interface RegistryOperations extends Service {

  /**
   * Create a path.
   *
   * It is not an error if the path exists already, be it empty or not.
   *
   * The createParents flag also requests creating the parents.
   * As entries in the registry can hold data while still having
```

```
 * child entries, it is not an error if any of the parent path
 * elements have service records.
 *
 * @param path path to create
 * @param createParents also create the parents.
 * @throws PathNotFoundException parent path is not in the registry.
 * @throws NoChildrenForEphemeralsException the parent is ephemeral.
 * @throws AccessControlException access permission failure.
 * @throws InvalidPathnameException path name is invalid.
 * @throws IOException Any other IO Exception.
 * @return true if the path was created, false if it existed.
 */
boolean mkdir(String path, boolean createParents)
    throws PathNotFoundException,
    NoChildrenForEphemeralsException,
    AccessControlException,
    InvalidPathnameException,
    IOException;


/**
 * Bind a name to a service record
 * @param path path to bind the service record
 * @param record service record to create/update
 * @param createFlags creation flags
 * @throws PathNotFoundException the parent path does not exist
 * @throws NoChildrenForEphemeralsException the parent is ephemeral
 * @throws FileAlreadyExistsException path exists but create flags
 * do not include "overwrite"
 * @throws AccessControlException access permission failure.
 * @throws InvalidPathnameException path name is invalid.
 * @throws IOException Any other IO Exception.
 */
void bind(String path, ServiceRecord record, int createFlags)
    throws PathNotFoundException,
    NoChildrenForEphemeralsException,
    FileAlreadyExistsException,
    AccessControlException,
    InvalidPathnameException,
    IOException;


/**
 * Resolve the record at a path
 * @param path path to service record
 * @return the record
 * @throws PathNotFoundException path is not in the registry.
 * @throws AccessControlException security restriction.
 * @throws InvalidPathnameException the path is invalid.
 * @throws IOException Any other IO Exception
```

```
 */

ServiceRecord resolve(String path) throws PathNotFoundException,
    AccessControlException,
    InvalidPathnameException,
    IOException;

/**
 * Get the status of a path
 * @param path
 * @return
 * @throws PathNotFoundException path is not in the registry.
 * @throws AccessControlException security restriction.
 * @throws InvalidPathnameException the path is invalid.
 * @throws IOException Any other IO Exception
 */
RegistryPathStatus stat(String path)
    throws PathNotFoundException,
    AccessControlException,
    InvalidPathnameException,
    IOException;

/**
 * List children of a directory
 * @param path path
 * @return a possibly empty array of child entries
 * @throws PathNotFoundException path is not in the registry.
 * @throws AccessControlException security restriction.
 * @throws InvalidPathnameException the path is invalid.
 * @throws IOException Any other IO Exception
 */
RegistryPathStatus[] listDir(String path)
    throws PathNotFoundException,
    AccessControlException,
    InvalidPathnameException,
    IOException;

/**
 * Delete a path.
 *
 * If the operation returns without an error then the entry has been
 * deleted.
 * @param path path delete recursively
 * @param recursive recursive flag
 * @throws PathNotFoundException path is not in the registry.
 * @throws AccessControlException security restriction.
 * @throws InvalidPathnameException the path is invalid.
 * @throws PathIsNotEmptyDirectoryException path has child entries, but
 * recursive is false.
 * @throws IOException Any other IO Exception
```

```
   *
   * @throws IOException
   */
  void delete(String path, boolean recursive)
      throws PathNotFoundException,
      PathIsNotEmptyDirectoryException,
      AccessControlException,
      InvalidPathnameException,
      IOException;

}


@JsonIgnoreProperties(ignoreUnknown = true)

/**
 * JSON-marshallable description of a single component
 */
public class ServiceRecord {

  /**
   * The time the service was registered -as seen by the service making
   * the registration request.
   */
  public long registrationTime;

  /**
   * ID. For containers: container ID. For application instances, application ID.
   */
  public String id;
  public int persistence;
  public String description;

  public List<Endpoint> external;
  public List<Endpoint> internal;
}


public class Endpoint {
  public String api;
  public String addressType;
  public String protocolType;
  public List<List<String>> addresses;
}


public class RegistryPathStatus {

  String name; // child Name
  long time;
```

```
  long size;

}

/**
 * Flags to use when creating a service entry
 */
public interface CreateFlags {

  /**
   * The entry is ephemeral, when this session is closed the entry
   * will be deleted.
   */
  int EPHEMERAL = 1;

  /**
   * If there is  existing entry, overwrite it.
   */
  int OVERWRITE = 2;

}

public interface PersistencePolicies {

  /**
   * The record persists until removed manually: {@value}.
   */
  int PERMANENT = 0;

  /**
   * Remove when the YARN cluster is restarted: {@value}.
   * This does not mean on HA failover; it means after a cluster stop/start.
   */
  int CLUSTER_RESTART = 1;

  /**
   * Remove when the YARN application defined in the id field
   * terminates: {@value}.
   */
  int APPLICATION = 2;

  /**
   * Remove when the current YARN application attempt ID finishes: {@value}.
   */
  int APPLICATION_ATTEMPT = 3;

  /**
   * Remove when the YARN container in the ID field finishes.
   */
  int CONTAINER = 4;
```

```
  /**
   * Automatic deletion when the session is closed/times out: {@value}.
   * This is implemented at the ZK layer, not in the RM.
   */
  int EPHEMERAL = 5;

}
```

Security

The registry will allow a service instance can only be registered under the path where it has permissions. Yarn will create directories with appropriate permissions for users where Yarn deployed services can be registered by a user. of the user account of the service instance. The admin will also create directories (such as /services) with appropriate permissions (where core Hadoop services can register themselves.

There will no attempt to restrict read access to registry information. The services will protect inappropriate access by clients by requiring authentication and authorization. There is a *scope* field in a service record , but this is just a marker to say "internal API only", rather than a direct security restriction. (this is why "internal" and "external" are proposed, not "public" and "private").

Rationale: the endpoints being registered would be discoverable through port scanning anyway. Having everything world-readable allows the REST API to have a simpler access model —and is consistent with DNS.

On a secure cluster, ZK token renewal may become an issue for long-lived services —if their token expires their session may expire. Renewal of such tokens is not part of the API implementation —we may need to add a means to update the tokens of an instance of the registry operations class.

Security Policy Summary

- The registry root is can only be written by: mapred, hdfs : `"rwcda"`
- The permissions are similarly restricted for `"/users/"`, and `"/services/"`
- Other entities can read: `"r"`
- Yarn will create a dir for each user that has a service running and will give that user write access `/users/${username}`
- The Web UI can be run under the identity of any user with read access, so does not need any transient user credentials to be propagated

Out of cluster and cross-cluster access

1. A client should be able to access the registry of another cluster in order to access services of that cluster. Detail of this need to further fleshed out.

2. Firewall services such as Apache Knox can examine the internal set of published services, and publish a subset of their endpoints. They MAY implement the REST API.

Limits

Entry Size
Zookeeper has a default limit of 1MB/node. If all endpoints of a service or component are stored in JSON attached to that node, then there is a total limit of 1MB of all endpoint registration data.

To prevent this becoming a problem, the client API should implement strict limits on the maximum length of fields, with low limits on the `addressType, protocol, and api` fields, something longer on the `description` and `addresses` elements —along with a limit on the number of elements in the `addresses` field.

Name size
To support DNS in future, there must be a limit of 63 bytes on all path elements. For non-ASCII User names, this restriction implies that a shorter path may be a limit.

Rate of Update
A rapid rate of entry change is considered antisocial in a ZK cluster. Implementations may throttle update operations.

Rate of Polling
Clients which poll the registry may be throttled.