Ambari Design

Design Goals

Platform Independence

The system must architecturally support any hardware and operating system, e.g. RHEL, SLES, Ubuntu, Windows, etc. Components which are inherently dependent on a platform (e.g., components dealing with yum, rpm packages, debian packages, etc) should be pluggable with well-defined interfaces.

Pluggable Components

The architecture must not assume specific tools and technologies. Any specific tools and technologies must be encapsulated by pluggable components. The architecture will focus on pluggability of Puppet and related components which is a provisioning and configuration tool of choice, and the database to persist the state. The goal is not to immediately support replacements of Puppet, but the architecture should be easily extensible to do so in the future.

The pluggability goal doesn't encompass standardization of inter-component protocols or interfaces to work with third-party implementations of components.

Version Management & Upgrade

Ambari components running on various nodes must support multiple versions of the protocols to support independent upgrade of components. Upgrade of any component of Ambari must not affect the cluster state.

Extensibility

The design should support easy addition of new services, components and APIs. Extensibility also implies ease in modifying any configuration or provisioning steps for the Hadoop stack. Also, the possibility of supporting Hadoop stacks other than HDP needs to be taken into account.

Failure Recovery

The system must be able to recover from any component failure to a consistent state. The system should try to complete the pending operations after recovery. If certain errors are unrecoverable, failure should still keep the system in a consistent state.

Security

The security implies 1) authentication and role-based authorization of Ambari users (both API and Web UI), 2) installation, management, and monitoring of the Hadoop stack secured via Kerberos, and 3) authenticating and encrypting over-the-wire communication between Ambari components (e.g., Ambari master-agent communication).

Error Trace

The design strives to simplify the process of tracing failures. The failures should be propagated to the user with sufficient details and pointers for analysis.

Near Real-Time and Intermediate Feedback for Operations

For operations that take a while to complete, the system needs to be able to provide the user feedback with intermediate progress regarding currently running tasks, % of operation complete, a reference to a operation log, etc., in a timely manner (near real-time). In the previous version of Ambari, this was not available due to Puppet's Master-Agent architecture and its status reporting mechanism.

Terminology

Service

Service refers to services in the Hadoop stack. HDFS, HBase, and Pig are examples of services. A service may have multiple components (e.g., HDFS has NameNode, Secondary NameNode, DataNode, etc). A service can just be a client library (e.g., Pig does not have any daemon services, but just has a client library).

Component

A service consists of one or more components. For example, HDFS has 3 components: NameNode, DataNode and Secondary NameNode. Components may be optional. A component may span multiple nodes (e.g., DataNode instances on multiple nodes).

Node/Host

Node refers to a machine in the cluster. Node and host are used interchangeably in this document.

Node-Component

Node-component refers to an instance of a component on a particular node. For example, a particular DataNode instance on a particular node is a node-component.

Operation

An operation refers to a set of changes or actions performed on a cluster to satisfy a user request or to achieve a desirable state change in the cluster. For example, starting of a service is an operation and running a smoke test is an operation. If a user requests to add a new service to the cluster and that includes running a smoke test as well, then the entire set of actions to meet the user request will constitute an operation. An operation can consist of multiple "actions" that are ordered (see below).

Task

Task is the unit of work that is sent to a node to execute. A task is the work that node has to carry out as part of an action. For example, an "action" can consist of installing a datanode on Node n1 and installing a datanode and a secondary namenode on Node n2. In this case, the "task" for n1 will be to install a datanode and the "tasks" for n2 will be to install both a datanode and a secondary namenode.

Stage

A stage refers to a set of tasks that are required to complete an operation and are independent of each other; all tasks in the same stage can be run across different nodes in parallel.

Action

An ‘action’ consists of a task or tasks on a machine or a group of machines. Each action is tracked by an action id and nodes report the status at least at the granularity of the action. An action can be considered a stage under execution. In this document a stage and an action have one-to-one correspondence unless specified otherwise. An action id will be a bijection of request-id, stage-id.

Stage Plan

An operation typically consists of multiple tasks on various machines and they usually have dependencies requiring them to run in a particular order. Some tasks are required to complete before others can be scheduled. Therefore, the tasks required for an operation can be divided in various stages where each stage must be completed before the next stage, but all the tasks in the same stage can be scheduled in parallel across different nodes.

Manifest

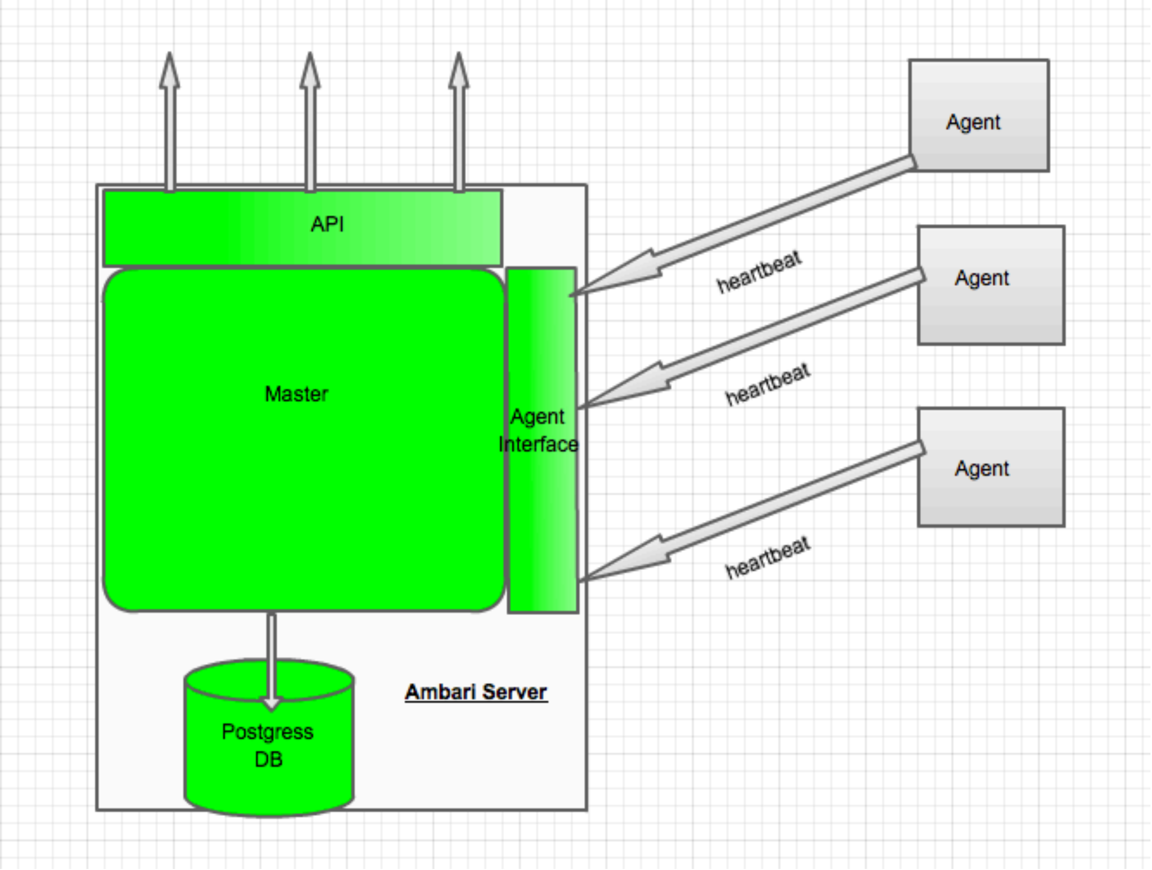
Manifest refers to the definition of a task which is sent to a node for execution. The manifest must completely define the task and must be serializable. Manifest can also be persisted on disk for recovery or record.

Role

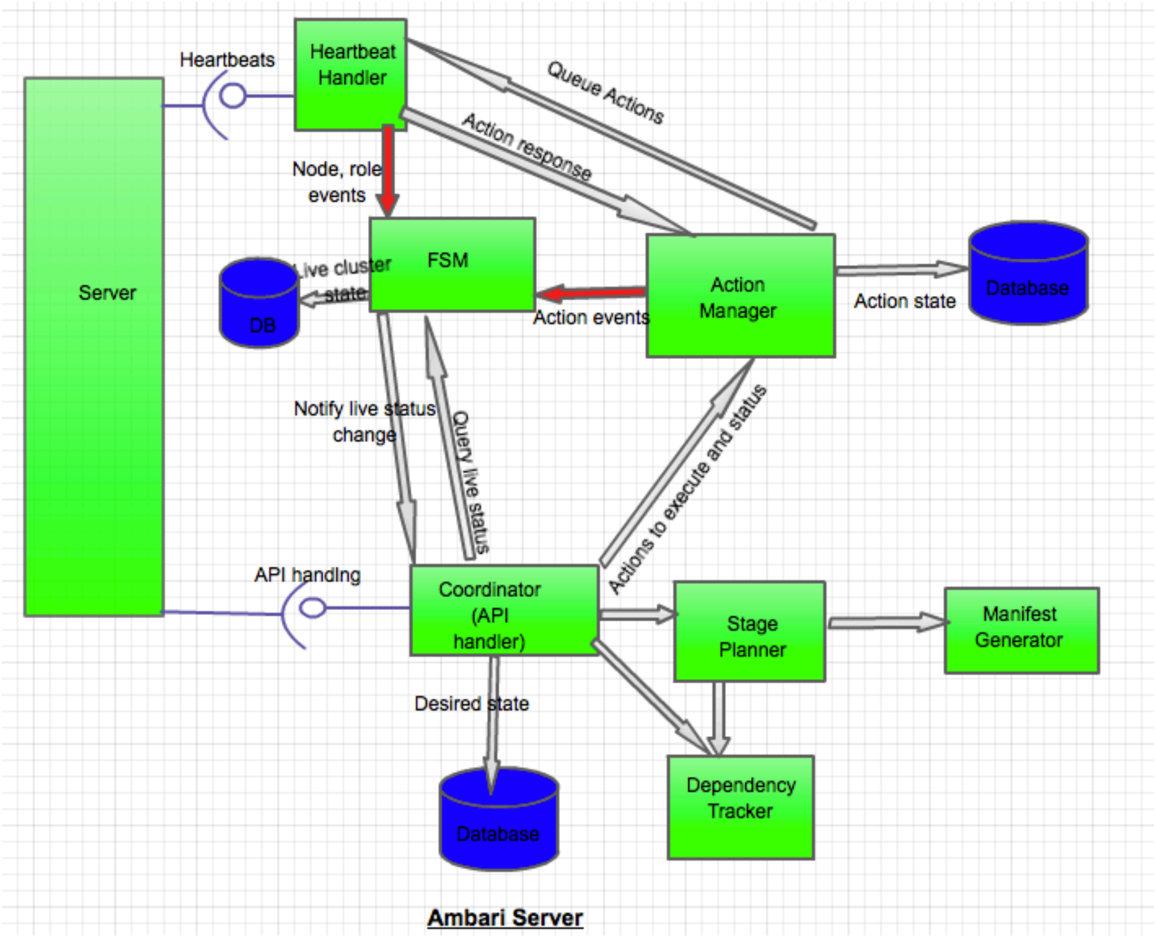
A role maps to either a component (e.g., NameNode, DataNode) or an action (e.g., HDFS rebalancing, HBase smoke test, other admin commands, etc.)

Ambari Architecture

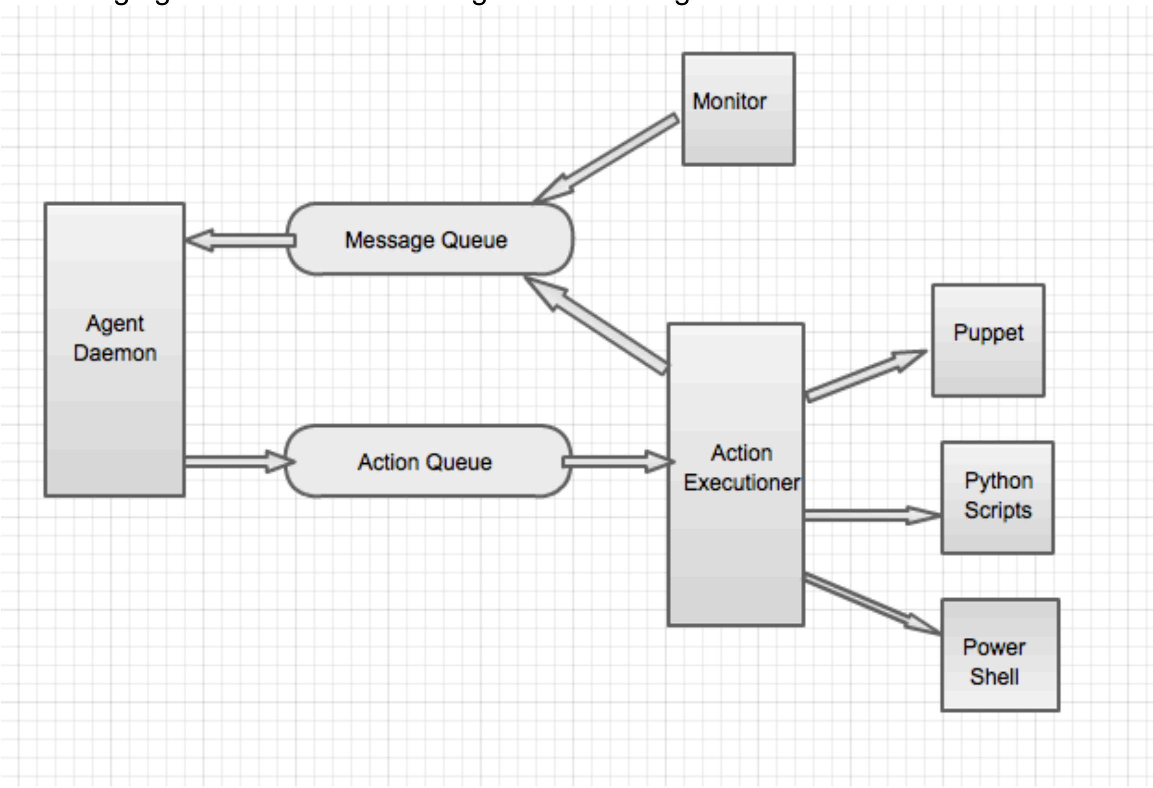
Following figure captures the high level architecture of Ambari.



Following figure describes the design of Ambari Server



Following figure describes the design of Ambari Agent.



Use cases

In this section we cover a few basic use cases and describe how the request are served by the system at a high level and how components interact.

1. Add service: Add a new service to an existing cluster. Let's take a specific example of adding Hbase service to an existing cluster, which is already running HDFS. HBase master and slaves will be added to the subset of existing nodes (no additional nodes). It will go through following steps:

- The request lands on the server via API and a request id is generated and attached to the request. A handler for this API is invoked in the Coordinator.
- The API Handler implements the steps needed to start a new service to an existing cluster. In this case the steps would be: install all the service components along with the required prerequisites, start the prerequisites and the service components in a specific order, and re-configure Nagios server to add new service monitoring as well.
- The Coordinator will lookup in Dependency Tracker and find the prerequisites for HBase. Dependency Tracker will return HDFS and ZooKeeper components. Coordinator will also lookup dependencies for Nagios server and which will return HBase client. Dependency Tracker will also return the required state of the required components. Thus, Coordinator will know the entire set of components and their required states. Coordinator will set the desired state for all these components in the DB.

- During the previous step, the Coordinator may also determine that it requires the user's input to select nodes for ZooKeeper and may return an appropriate response. This depends on the API semantics.
- The Coordinator will then pass the list of components and their desired states to the Stage Planner. Stage Planner will return the staged sequence of operations that need to be performed on each node where these components are to be installed/started/modified. The Stage Planner will also generate the manifest (tasks for each individual nodes for each stage) using the Manifest Generator.
- Coordinator will pass this ordered list of stages to the Action Manager with the corresponding request id.
- Action Manager will update the state of each node-component, in the FSM, which will reflect that an operation is in progress. Note that the FSM for each affected node-component is updated. In this step, the FSM may detect an invalid event and throw failure, which will abort the operation and all actions will be marked as failed with an error.
- Action Manager will create an action id for each operation and add it to the Plan. The Action Manager will pick first Stage from the plan and adds each action in this Stage to the queue for each of the affected nodes. The next Stage will be picked when first Stage completes. Action Manager will also start a timer for scheduled actions.
- Heartbeat Handler will receive the response for the actions and notify the Action Manager. Action Manager will send an event to the FSM to update the state. In case of a timeout, the action will be scheduled again or marked as failed. Once all nodes for an action have reached completion (response received or final timeout) the action is considered completed. Once all actions for a Stage are completed the Stage is considered completed.
- Action completion is also recorded in the database.
- The Action Manager proceeds to the next Stage and repeats.

2. **Run Smoke Test:** The cluster is already active with HDFS and HBase services active, and the user wants to run HBase smoke test.

- The request lands on the server via API and a request id is generated and attached to the request. A handler for this API is invoked in the Coordinator.
- The API Handler invokes Dependency Tracker and finds that HBase service should be up and running for this. The API Handler throws back error if hbase live status is not running. The Handler also determines where the smoke test should be run. Dependency Tracker will expose a method which will tell the Coordinator which client component is required on the host where smoke test should be run. The Stage Planner is invoked to generate a plan for smoke test. In this case the plan is simple one stage with single node-component.
- The rest of the steps are similar to previous use case. In this case the FSM will be specifically for hbase-smoketest role.

3. **Reconfigure Service**

- The cluster is already active with the services and its dependencies.
- A request to save the configuration lands on the server and the new config snapshot is stored on the server. This request should also have information on what service(s) and/or roles-hosts the config change affects. This implies

that the persistence layer needs to track on a per service/component/node basis as to what was the last config version that was changed that affects the object in question.

- A user can make multiple calls above to save multiple checkpoints.
- At some point, the user decides to deploy the new configs. In this scenario, the user will send a request to deploy the new configs to the required service/component/node-component.
- When this request lands on the server, via the coordinator handler, it will result in updating of the desired config version of the object in question to the version specified or the latest config based on the API specs.
- The Coordinator will execute re-configure in two steps. First, it will generate a stage plan to stop the services in question. Then it will generate a stage plan to start the services with new configurations. Coordinator will append the two plans, stop followed by start and will pass it to the Action Manager. Rest of the steps proceed as in previous use cases.

4. Ambari master crashed and restarted

- Assuming the Ambari master dies, there are multiple scenarios that need to be addressed.
- Assumptions:
  - Desired state has been persisted in the DB
  - All pending actions are defined in the database.
  - Live status is tracked in the DB. However, the agent cannot send back live status currently so require will be based on DB state.
  - All actions are idempotent.
- Actions required
  - Ambari Master
    - The action layer needs to re-queue all pending or previously queued (but incomplete) stages back to the Heartbeat Handler and allow the agents to re-execute the steps in question.
    - The Ambari server should not accept any new actions/requests until it has recovered completely.*
    - If there is any discrepancy between actual live state and the desired state (i.e live state is STARTED or STARTING but desired state is STOPPED), there should be a trigger to the stage planner/transaction manager to initiate the actions to get state to the desired state. However, if the live state is STOP_FAILED and desired_state is STOPPED, then this is a no-op for recovery as the Ambari server does not initiate actions itself in such scenarios. For the latter case, the onus is on the admin/user to re-initiate a stop for the nodes that failed [this may change depending on API specifications and product decision].
  - Ambari Agent
    - The agent should not die if the master suddenly disappears. It should continue to poll at regular intervals and recover as needed when the master comes back up.

- The Ambari agent should keep all the necessary information it planned to send to the master in case of a connection failure and re-send the information after the master comes back up.
- It may need to re-register if it was previously in the process of registering.

5. Decommission a subset of Datanodes.

- Decommissioning will be implemented as an action performed on hadoop-admin role in the Puppet layer. hadoop-admin will be a new role defined to cover hadoop admin operations. This role will require hadoop-client to be installed and admin privileges available.
- The manifest for the decommission operation will consist of hadoop-admin role with certain parameters like a list of datanodes and a flag specifying the action.
- Coordinator will identify a node which is designated to run hadoop-admin actions. This information must be available in cluster state.
- The decommission action will follow the state machine for node-roles that are actions. This will be considered successful when the decommissioning has been started at the namenode i.e. when the admin command to datanode succeeds.
- Information whether nodes have been decommissioned or not should be queried separately. This information is available in namenode UI. Ambari should have another API to query decommissioning or decommissioned datanodes.
- Ambari does not keep track whether a datanode is decommissioning or decommissioned. To get rid of decommissioned nodes from Ambari, a seperate API call should be made to Ambari to stop/uninstall those datanodes.

Agent:

Agents will heartbeat to the master every few seconds and will receive commands from the master in the heartbeat responses. Heartbeat responses will be the only way for master to send a command to the agent. The command will be queued in the **action queue,** which will be picked up by the action executioner. **Action executioner** will pick the right tool (Puppet, Python, etc) for execution depending on the command type and action type. Thus the actions sent in the hearbeat response will be processed asynchronously at the agent. The action executioner will put the response or progress messages on the message queue. The agent will send everything on the message queue to the master in the next heartbeat.

Recovery

There are two main choices in terms of master recovery.

- **Recovery based on actions**: In this case every action is persisted and upon a restart, the master checks for pending actions and reschedules them. The state of the cluster is also persisted in the database and master rebuilds the state machines upon a restart. There could be a race condition that some actions might be complete but master crashes before recording their completion, this will be handled by ensuring actions are idempotent and master will re-schedule all actions that are not marked as completed or failed in the DB. Persisted actions can be viewed as redo logs which is a very common approach.
- Alternating to the above approach is recovery based on the desired state. In this approach the master persists a desired state and upon restart it matches desired state with the live state and tries to restore the cluster to the desired state.

The approach based on actions gels better with our overall design because we pre-plan an operation and persist it, therefore even if we persist desired state, the recovery will require a re-planning of actions. Also the desired state approach doesn't capture certain operations that don't change the state of the cluster from Ambari point of view, e.g., smoke tests or hdfs re-balancing. Persisted actions can be viewed as redo logs.

Agent recovery requires just an agent restart because the agent is stateless. An agent failure will be detected by the master by heartbeat loss beyond a threshold of time.
TBD: How is agent restarted?

Security

API Authentication Methods

One way is to use HTTP Basic Authentication. This means the client would pass the credentials (or a base64 encoded version of it) to the server on every request. The server would have to validate credentials on every call. It makes it easy for CLI clients to use the API this way. Also the session state is not stored on the server.
Another way is to utilize HTTP Session and Cookies. Once the server validates the credentials, the server generates and stores the session ID in its store. The session ID is sent back to the client to be stored as a cookie. The client can pass this session ID on subsequent calls. This is more efficient in that credentials need not be validated on every API request. VMware's vCloud REST API 1.0 used to support this. However, they dropped this in later versions due to certain security concerns: http://kb.vmware.com/selfservice/microsites/search.do?language=en_US&cmd=displayKC&externalId=1025884
Also, this approach forces the server to store session state which is a big no no for RESTfulness (IF we care).
We can support both approaches.
For CLI, the first method is preferred. For browser based applications, the latter method avoids making redundant credential validations.

Both authentication methods (and any API call that requires an authenticated user) should be done over HTTPS. User credentials or the session token should never be transferred over HTTP.

Bootstrap

In HMC (released as 1.0) bootstrapping was a very integrated process of installing/configuring the hosts. In Ambari 1.1, bootstrapping will be a helper routine to install the Ambari agents on the hosts. In HMC 1.0, the bootstrap process figures out the information about a host and adds that to the database. This will now be done as a part of Ambari agent registering to the server. The bootstrap process will just SSH onto the hosts and install the Ambari agent. No database changes will be done as part of doing a bootstrap.