# Autonomous Agent Upgrade

**Status: Complete**

**Sponsor User:** *IBM*

**Date of Submission:** 16 Aug 2021

**Submitted by:** David Booz (booz@us.ibm.com)

**Affiliation(s):** IBM

***<Please fill out the above fields, and the Overview, Design and User Experience sections below for an initial review of the proposed feature.>***

**Scope and Signoff:** *(to be filled out by Chair)*

# Overview

*<Briefly describe the problem being solved, not how the problem is solved, just focus on the problem. Think about why the feature is needed, and what is the relevant context to understand the problem.>*

Installing the agent and managing edge nodes needs to be a zero touch experience. With the centralization of node configuration in the management hub and SDO support to perform the initial agent install, the zero touch experience is almost complete. This missing piece is automatically updating the agent software and/or it's configuration (e.g. the management hub cert or even the hub URLs). Today, the supported agent install procedures are able to upgrade an already installed agent, but those procedures require an administrator to gather the host login credentials for each edge node so that the install process can access the node's host remotely. A zero touch experience requires that agent upgrades are done by the agent in concert with the management hub, using the agent's node credentials, reaching out to get the updated software and config, while running behind a network firewall. While it is reasonable to expect an administrator to initiate agent upgrades, in order to support 1000s of nodes the upgrade process should not require the administrator to explicitly address each node. Agent software and configuration upgrades should be autonomous from the perspective of the agent and the node owner. Management hub administrative users need to have control over the upgrade process, but they should not need direct access to the node's host.

# Design

*<Describe how the problem is fixed. Include all affected components. Include diagrams for clarity. This should be the longest section in the document. Use the sections below to call out specifics related to each aspect of the overall system, and refer back to this section for context. Provide links to any relevant external information.>*

## Theory

The essential aspect of policy within OH is less about deploying services than it is about an expression of intent on the part of the user, coupled with an implied set of lifecycle actions. The existing deployment policy feature is a conflation of intent and action. The user intends to deploy a service and wants the system to perform all the implicit actions of that deployment autonomously. The act of publishing a deployment policy signals intent to deploy a service, and initiates autonomous management of the following lifecycle steps in the agent:

- telling the agent what to deploy and how to configure it
- giving each agent a chance to decline the intent
- download the containers
- verify the container hash
- create the container
- start the container
- create a network
- attach the container to the network
- monitor and report container status

As a theoretical exercise, consider that a deployment policy could include (and may in the future) the definition of a custom lifecycle flow which is enacted by the system to achieve the user's intent. The default lifecycle would be as stated above. Further consider that the user wants to perform some action on the host before the container is started. The user would provide the details of the additional lifecycle step(s) within the deployment policy (though maybe not literally encoded within the JSON) enabling the agent to enact the custom lifecycle actions. It is this core concept within policy of user intent plus lifecycle actions that is also exploited in this design for autonomous agent upgrades.

Also consider that the deployment policy has an implied result: the service running. This is necessary for any intent-based policy system, so the system knows whether the intent has already been accomplished or not. This understanding is important to understand this design.

## Node Management Policy (NMP)

Node Management Policy (NMP) is introduced to provide a framework for automatic agent updates, and other similar management tasks not yet fully realized. It is a policy definition that holds agent management policy settings. A NMP contains a set of policy actions that apply to node(s), which each agent is responsible for enacting when the NMP applies to its node. A NMP applies to nodes that have a compatible node policy or pattern match with the policy and pattern selectors in the NMP. In this sense, and NMP is similar to a deployment policy. An NMP is different from a deployment policy in that the agent acts on the policy directly, the agbot is not involved.

A NMP is backed by a resource in the exchange. It resides within an org and applies only to agents in the same org. The exchange API to interact with a NMP is **/api/v1/orgs/<org>/managementPolicy/<name>.** The GET, PUT, POST, PATCH verbs would be supported as usual.

...verb details here when they are ready...

The pseudo-schema for an NMP looks like this, a description of each field follows:

```
{
  "owner":"",               // The user that created this resource
  "label": "",              // (Optional) A short description of the policy
  "description": "",        // (Optional) A much longer description of the policy
  "constraints": [""],      // (Optional) Typical constraint expression used to select nodes
  "properties": [{}],       // (Optional) Typical property expressions used by nodes to select policy
  "patterns": [""],         // (Optional) This policy applies to nodes using one of these patterns
  "enabled": boolean,       // Is this policy enabled or disabled, default false == disabled
  "agentUpgradePolicy": {},  // (Optional) Assertions on how the agent should update itself
  "lastUpdated":"<time-stamp>"
}
```

The "pattern" field is NOT mutually exclusive with "properties" and "constraints". This allows a single NMP to apply to nodes which use patterns or policies. The patterns list may contain the string "*", indicating that the NMP applies to any node that is using a pattern.

Because patterns can be public, the pattern array can contain org qualified pattern names. That is, the format of the strings in the patterns field is org /pattern-name, where the character '/' is used as a separator. If org is omitted, the org of the NMP is the default org.

The constraints field contains the same text based constraint language found in deployment, service, model and node policy constraints. The same language parser and therefore the same syntax is supported. The properties field contains policy properties, using the same syntax as deployment, service, model and node policies. The properties and constraints are used to determine which policy based nodes are subject to the intent of the NMP. This determination includes the same bi-directional policy compatibility calculations used for deployment, service, model and node policy.

An important new concept with NMP is the ability to administratively enable and disable the policy. This allows an NMP to be published but not acted on by the agent until the policy is enabled. Retrofitting this concept into deployment policy is not within the scope of this design.

**Question: Is there a flavor of NMP that deactivates itself after it is complete? How do we know when an NMP is complete? Do we need to know? ???**

The purpose of the "agentUpgradePolicy" field is to enable an org admin user to declare an intent to upgrade the agent version or configuration of a set of agents. Note that there are additional actions (with a different implied lifecycle) which an NMP might contain in the future. Those are discussed later in this design document.

In a large scale environment with thousands of nodes, an enabled NMP with empty constraints and/or patterns set to "*" could be quite disruptive to the management hub or even the entire system of nodes by attempting to upgrade all agents at approximately the same time. Publishing a NMP that meets these criteria requires the user to confirm that they understand the potentially disruptive nature of the policy before the NMP is published in the exchange.

Note that a default NMP is equivalent to having no NMP at all because the default for the enabled field is false.

## Agent Upgrade Policy

There are several discrete lifecycle steps associated with upgrading the agent software and/or configuration to a new version:

- a stimulus to begin the process
- download the new version
- verify a hash of the downloaded package
- install the new package (if there is one)
- update any new or changed configuration (if any)
- restart the agent process
- report the new (or failed) version info to the node/status resource in the exchange

This is the implied lifecycle for a new kind of policy called "agentUpgradePolicy" defined within a NMP.

For any given node, the stimulus to initiate the "agentUpgradePolicy" occurs when a NMP is successfully enabled (or created already enabled) that matches the node (policy or pattern). The node self discovers this stimulus (through the exchange notification framework, i.e. the /changes API) and initiates the policy actions on its own. Packages are downloaded from the hub (CSS), the package hash is verified and the package is installed and/or the config is updated. The agent is restarted (see below) and updates it's new package version in its exchange node/status resource.

The "agentUpgradePolicy" section in a NMP contains the specifics of the agent upgrade intent being expressed by the administrator, and conforms to the following JSON snippet:

```
"agentUpgradePolicy": {
      "atLeastVersion": "<version> | current", // Specify the minimum agent version these nodes should have,
default "current"
      "start": "<RFC3339 timestamp> | now",    // When to start an upgrade, default "now"
      "duration": seconds                      // Enable agents to randomize upgrade start time within start +
duration, default 0
},
```

The atLeastVersion field indicates the minimum version the agents should upgrade to when instructed. When atLeastVersion is set to **current**, the agent will periodically check for new versions and automatically upgrade to the newest version.

The start field contains an RFC3339 timestamp indicating when the upgrade should start. The timestamp is in UTC format so that affected nodes anywhere in the world are acting based on the same clock.

The duration field is specified in seconds, and is added to the start time to form a window of time within which the agent will initiate an update. Each agent will randomly select a point within that window to begin the upgrade process. This prevents all affected agents from upgrading at the same time, possibly overwhelming the management hub. The combination of selecting nodes based on properties and constraints plus the start time and duration is intended to enable org administrators sufficient control over mass upgrades to prevent overwhelming the management hub.

If an agent is instructed to upgrade to a version which it is already running, it will not perform the upgrade.

If an agent detects a policy conflict, it will log the conflict in its event log.

If an agent upgrade fails, the agent will be rolled back to the previous version. The failure status will be recorded in the node's /status resource in the exchange, and the upgrade will not be attempted again until a newer version of the package is made available.

**Question: Can agents be instructed to downgrade?**

## NMP Constraint Collisions

Given the flexibility of policy within OH, it is possible that NMPs with conflicting actions could be compatible with the same node(s). There are a myriad of circumstances in which this could occur. Within this design, the conflict could occur in relation to the atLeastVersion setting of the agent update policy. Features (in the future) which add additional policy to a NMP could create additional conflicting situations. Two or more NMPs cannot be in conflict if they are enacting different kinds of policy. This is important for the future when additional management policy kinds are added.

The agent is responsible for interpreting each NMP and acting on it accordingly, therefore an agent is only concerned with the NMPs that apply to it. If there are any conflicts, the agent will not enact the conflicting policy action. For example, NMP1 enables policy intent1 at time t1, NMP2 enables policy intent2 at time t2, and NMP3 enables policy intent1 at time t3 with a policy setting that conflicts with NMP1. For the purposes of conflict resolution, the only potential conflicts are between NMP1 and NMP3. Agents will implement the policy intents in NMP1 and 2. NMP3 is ignored because NMP3 conflicts over intent1 and was added after NMP1 was already enacted.

It is possible that conflicting actions could be introduced in close temporal proximity. This could result in actions that are immediately reversed. The agent should be prepared to handle these cases. For example, NMP1 and NMP2 are introduced/enabled a few seconds apart and both adjust the same policy setting (e.g. autoUpgrade). Some of the nodes that match NMP1 and NMP2 will be notified of both NMP1 and NMP2 at the same time. These nodes will enact the actions in NMP2, completely ignoring NMP1. Other nodes might only be notified of NMP1, and will enact the policy in NMP1. Very soon after that, these same nodes are notified of NMP2, and recognizing that NMP2 is newer than NMP1, will enact the action in NMP2, reversing what was done when NMP1 was enacted.

Here are the changes that might result in a management policy collision:

1. new/changed node policy - this can occur at node registration (which is essentially creation of a node policy) or when there are changes to an existing node policy
2. enablement of an existing NMP - an NMP can be published in the disabled state
3. new/changed NMP that is also enabled

In the absence of any enabled NMPs, an agent never checks for an update.

## Agent Version Status

The agent will maintain it's version in the exchange, in its /status resource. Following is the JSON encoded pseudo-schema for the new version status:

```
"agentVersion": {
      "package":{}, // The version(s) of the most recent package installed on the agent
      "failed":"<version>", // The last version upgrade failed on this version, or empty
      "upgradeStart":"<RFC3339 timestamp>", // The time (in UTC) when the last upgrade started
      "upgradeEnd":"<RFC3339 timestamp>", // The time (in UTC) when the last upgrade ended
}
```

The package field contains the JSON object from the package manifest of the most recently successful agent install. See the "Agent package" section.

## Agent package:

**BP:** I think it is important for this and agent-install.sh to work from the same files, because they are doing essentially the same thing.

An updated package downloaded by the agent from the management hub, may contain any or all of the following:

1. Package manager specific packages for updating agent binaries
2. Management hub URLs
3. SSL certificate for the management hub

The package will be structured as follows. It is stored as a gzipped tar file in the management hub, where the tar file has the following directory structure:
/packages - Contains the binary packages for the node's package manager, e.g. debs, rpms, etc.
/ssl - Contains a PEM encoded certificate, similar to the agent-install.crt file.
/config - contains the env var settings to bootstrap the node to the hub, similar to the agent-install.cfg file.

Any of these directories may be omitted or empty, in which case the agent will simply skip the directory and move on to the next one.

At the root of the package is a metadata file called the package manifest; it is named package.json. It contains a JSON encoded object defining the contents of the package:

```
{
  "version: "<semantic-version>",
  "packagesVersion": "<semantic-version>",
  "sslVersion":"<semantic-version>",
  "configVersion":"<semantic-version>"
}
```

The version field corresponds to the version of the overall package. The other version fields correspond to the version of the contents of the similarly named directory in the package. The version field is required, the other version fields are optional and are used strictly for documentation. The agent makes use of ONLY the version field to determine if the package is different from the agent's current version.

The version field is a semantic versions, based on the standard 3 part dotted decimal syntax; e.g. 1.0.2.

Agent packages can be added to the management hub at any time, but will not be acted upon by any registered agent until an NMP is enabled that matches the agent.

Agent packages are stored in the MMS, specifically in the CSS in the management hub. The MMS object which defines the package has a version field which MUST be the same as the version of the package.

Agent packages all reside within a single organization within the management hub. It does not matter which org because the packages are declared to be public objects in the MMS, meaning that they can be downloaded by an agent in any org. A NMP (which is org specific) initiates the deployment of agent packages, it is not the act of uploading a new package that causes existing agents to update themselves.

Agent packages have an object type of "openhorizon.agent-package", to discriminate them from other objects in the MMS.

Agent packages can be created using any tools the user chooses, as long as the package conforms to the design described above. The hzn CLI is extended to provide some simple commands to help the user get started.

hzn [agentpackage | ap] [new | list | publish]

...describe more details...

## Edge Clusters

Agents running in an edge cluster are running within a container. The container's guest OS and package manager will be used to upgrade the agent binaries. The agent will upgrade itself within the container, without terminating the container.

**Question: Still deciding on this. The alternative is to have the agent installed via an operator (which we dont have today) and allow the operator to do the upgrade. Perhaps this is a better way? The agent process would have to communicate with the operator (through a CRD) in order to trigger an upgrade. ???**

**Question: Is there a case where we would need to update the container in some way that is different from updating the agent binary of the agent config ???**

## Agent in Linux Container

The existing upgrade script depends on pulling image from dockerhub OH
...design for this is tbd...

### Additional management policy

Thinking forward a bit, NMP could be extended with the following schema to support additional scenarios:

1. The ability to scale to 100s millions of nodes will require multiple management hubs working together in the horizontal scaling sense (there will be limits to the vertical scale of a single management hub). This will require that an agent communicates with multiple management hubs.
2. The ability to configure a failover hub.
3. The ability for an agent to upgrade a top level service's dependencies without a new agreement.
4. The ability for nodes to self align into HA groups, ensuring that at least one of them is always running a stable version of edge services, even when some of those services are being upgraded to new versions.

Here's some schema examples:

```
{
  "service-dependency-upgrade": { // enable service dependencies to upgrade without a new agreement
    "enable": boolean,
  },
  "management-hub": {       // management hub configuration, if the primary doesn't work, use an alternate.
    "primary" : {
      "exchange": "<url>",
      "mms": "<url>",
      "tls-cert": "<url>"
    },
    "alternate": [          // optional
      {"exchange": "<url>",
      "mms": "<url>",
      "tls-cert": "<url>"},
    ],
  "ha-group": {                           // establish a group of nodes that are working together, service
upgrades happen one at a time
    "partners": ["<node-id>|<node-name>"], // An explicit list of partner nodes, XOR
    "constraint": [""]
  }
}
```

# User Experience

*<Describe which user roles are related to the problem AND the solution, e.g. admin, deployer, node owner, etc. If you need to define a new role in your design, make that very clear. Remember this is about what a user is thinking when interacting with the system before and after this design change. This section is not about a UI, it's more abstract than that. This section should explain all the aspects of the proposed feature that will surface to users.>*

As a hub admin, I want to install updated agent packages into the management hub so that nodes in any org can receive agent updates.

As an org admin, I want to declare that some or all nodes in my org will automatically upgrade the agent when a new version becomes available in the management hub.

As an org admin, I want to declare that some or all nodes in my org will never automatically upgrade the agent.

As an org admin, I want to declare that some or all nodes in my org should start upgrading at a given time.

As an org admin, I want to declare that some or all nodes in my org should upgrade to a specific version.

As an org admin, I want to know the agent version of any given node or group of nodes in my org.

As a hub admin, I want to change the self signed SSL certificate used for TLS between agents and the hub.

As a hub admin, I want to declare the default agent upgrade behavior for all nodes in the org, when creating a new org.

As an org admin, I want to change the default agent upgrade behavior for all nodes in the org, after the org has been created.

As a node owner, I want to override the org's agent upgrade behavior, I want to control when to upgrade the agent on my node.

As an org admin, I want to get a list of nodes that my NMP applies to.

**Some additional, but related use cases to ponder. These are NOT covered in detail by this design document:**

As an org admin, I want some or all of the nodes in my org to upgrade dependent services.

As a cluster admin, I want to change one of the URLs that agents use to communicate with the hub.

As a node owner, I want to declare which nodes are part of an HA group.

# Command Line Interface

*<Describe any changes to the hzn CLI, including before and after command examples for clarity. Include which users will use the changed CLI. This section should flow very naturally from the User Experience section.>*

hzn exchange [ nm | nodemanagement ] [ list | new | publish | remove ]

NOTE: verb to ask if all nods have implemenetd a given NMP policy.

For the hzn exchange nm publish command, when the user is trying to publish a NMP with empty constraints and/or patterns == "*", they are required to co nfirm that they understand the potentially disruptive nature of the policy.

# External Components

*<Describe any new or changed interactions with components that are not the agent or the management hub.>*

None.

# Affected Components

*<List all of the internal components (agent, MMS, Exchange, etc) which need to be updated to support the proposed feature. Include a link to the github epic for this feature (and the epic should contain the github issues for each component).>*

Agent

Exchange (new resource and updates of that resource surfaced through /changes API)

# Security

*<Describe any related security aspects of the solution. Think about security of components interacting with each other, users interacting with the system, components interacting with external systems, permissions of users or components>*

# APIs

*<Describe and new/changed/deprecated APIs, including before and after snippets for clarity. Include which components or users will use the APIs.>*

# Build, Install, Packaging

*<Describe any changes to the way any component of the system is built (e.g. agent packages, containers, etc), installed (operators, manual install, batch install, SDO), configured, and deployed (consider the hub and edge nodes).>*

# Documentation Notes

*<Describe the aspects of documentation that will be new/changed/updated. Be sure to indicate if this is new or changed doc, the impacted artifacts (e.g. technical doc, website, etc) and links to the related doc issue(s) in github.>*

# Test

*<Summarize new automated tests that need to be added in support of this feature, and describe any special test requirements that you can foresee.>*