Release Versioning Strategy (proposal for Honolulu)

- Objective
- ONAP Versioning Strategy
  - What are the 2 big issues to address?
    - Issue of dependency on Snapshot
    - Issue of identifying which version you depend on
  - Nexus
- Versioning of Repos
  - Example: 2 major releases
  - Example: 1 Major, 1 Service and 1 Enhancement Releases
- Semantic Versioning
- Branching and Merging
  - Basic Principles
  - Tagging Daily Builds in ONAP
  - Branching and Merging in ONAP
- API Versioning

Objective

The goal of the ONAP versioning strategy is to provide a compromise addressing the following competing goals:

- Minimize work and complexity for project teams
- Provide an easy to use configuration to ONAP users and the integration team
- Be intuitive

ONAP Versioning Strategy

What are the 2 big issues to address?

Issue of dependency on Snapshot

> *Example of the problem:*
>
> *Imagine you have your cpt A build working fine. Your build is depending on cpt B SNAPSHOT build.*
>
> *For some good reasons, cpt B makes a new build. As cpt A relies on B SNAPSHOT (meaning latest build) when you rebuild cpt A if the build breaks you will not know if the problem is within cpt A or within its dependency cpt B.*

To address the Snapshot dependency scenario, the idea is to move away from Snapshot dependency to "Release Artifact" dependency only. The word "Release Artifact" is loaded with the sense that the build is stable, has been thoroughly tested and can be used by another team. The "Released Artifact" is available within the "Nexus Release" repo (refer to Nexus section below for further description of Nexus repos). The PTLs and the project team decide on their own when to create a "Release Artifact". This approach allows to truly decouple the builds from each other and avoid the situation where something that was working fine before doesn't suddenly break due to snapshot changes.

The "Release Artifact" dependency also address the fact that "mega build" are no longer necessary.

Issue of identifying which version you depend on

ONAP is composed of more than 30 projects and 200 repositories. Dealing with components dependencies and documenting the version tree for each build can become a Dantesque activity.

In order to know whether you're depending on the right version of a particular artifact, ONAP will be maintaining a master "version manifest" for each named release (e.g. Amsterdam) specifying the version of each component that is to be delivered within a release. The manifest is a structured list of components and their versions. The manifest is intended to track a particular ONAP release and be used to track all versions of all components within a release.

One early idea was to embed the version manifest within the oparent artifact, perhaps as version properties within the POM file that can be directly consumed by downstream POMs. This approach, however, has some issues:

> *Just think about this scenario:*
>
> *cpt A version 2.3.0 depends on o-parent 0.9.0*
>
> *cpt A releases 2.3.1, which still depends on o-parent 0.9.0*
>
> *Now, cpt A version in o-parent needs to be updated*

Due to the change of the cpt A version in o-parent, o-parent itself needs to be bumped to 0.9.1. Now we have a circular dependency.

Each PTLs will need to update the manifest to declare the "Released" version available. In order to avoid the circular dependency issue, any updates to the version manifest, must NOT require any artifact to be version bumped and re-released.

Understanding the above, ONAP is maintaining a manifest of "Released Artifacts" outside the build process and outside of any binary artifacts. However, the manifest is still in source control. A specific version of this manifest will be blessed by TSC and used to make the "Named Release" (see definition of "Named Release" below).

Nexus

ONAP has 3 types of Nexus repos for artifacts:

1. **Snapshot repo**: this should no longer be needed since there should no longer be cross-project SNAPSHOT dependencies and therefore there should be no reason to make SNAPSHOT artifacts available via Nexus.
2. **Staging repo**: used for artifacts built from master and staged for further verification - those that have passed CSIT should be valid candidates for Release. The Staging artifacts are used primarily by the Team for their own testing and for E2E Release testing. The Staging artifacts are not meant for public consumption. Once a day, a new clean build is automatically performed. All Staging artifacts have same version number.
3. **Release repo**: this is the place where the project Team (or Linux Foundation Releng Team) stores the artifacts that are deemed stabled for being consumed by the other project teams. Each team decides when to release. It is not expected to get a new release every day. No TSC approval is required for getting a new release artifact.

Versioning of Repos

Each project team decides on when version numbers of repos under its control are incremented and artifacts are place in a Nexus Release repo. In particular the version numbers do not have to be in sync across projects and do not have to be aligned with the release version number.

ONAP has 2 types of artifacts:

1. **Artifact Release**: this refers to all the jar and docker files that are under the control of a project. No TSC approval is necessary for the team to move artifacts within Nexus Release
2. **Named Release**: this refers to the Marketing name that is used externally to evangelize ONAP. The "Named Release" is a collection of properly versioned "Artifact Release". The "Named Release" required TSC approval and is published within Docker Hub.

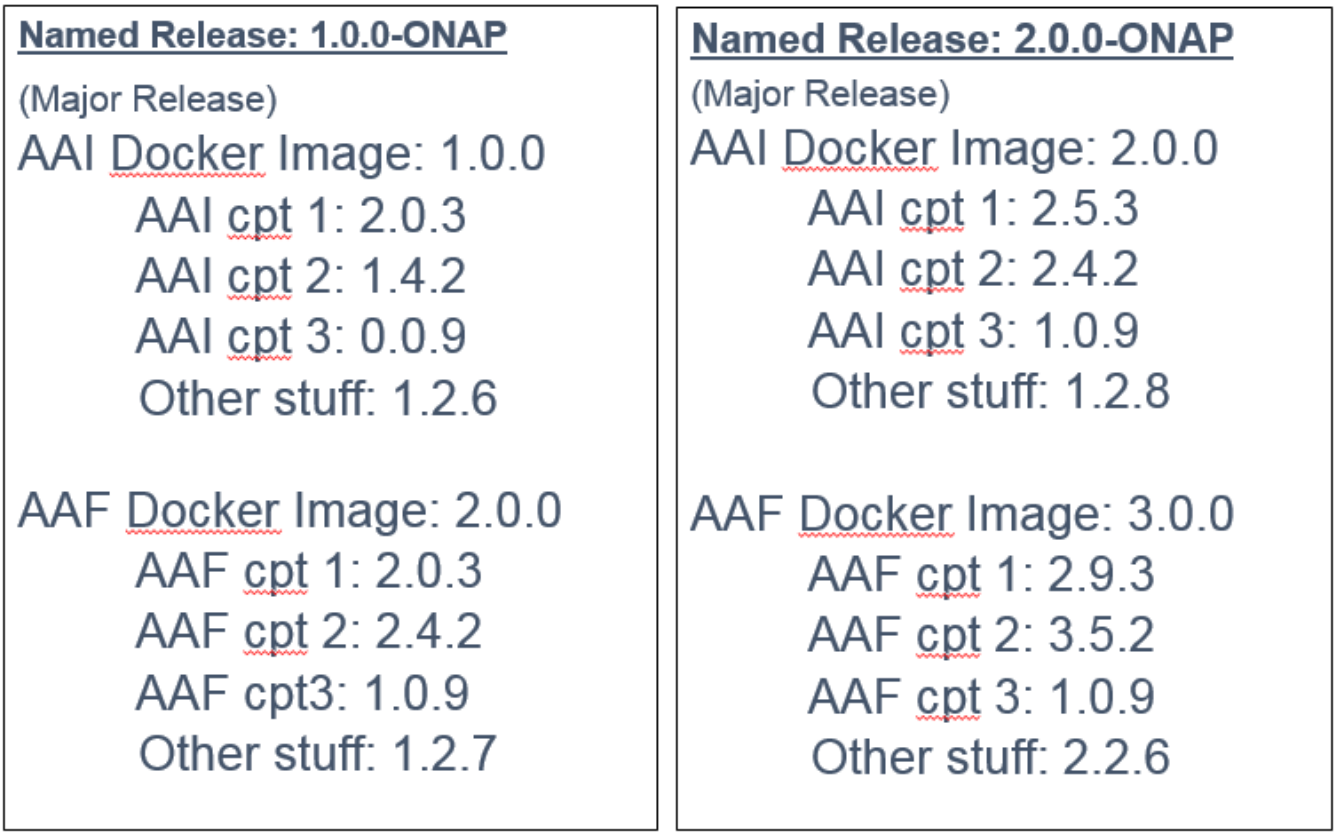
Example: 2 major releases

The diagram below has 2 Major releases. Theses are Named Release "1.0.0-ONAP" and "2.0.0-ONAP".

1.0.0-ONAP is a Major Release, and embeds (in this example) 2 Artifact Release, "AAI Docker" version 1.0.0 and "AAF Docker" version 2.0.0.

Note that the Artifact Release have different version number and do embed components that also have different version number.

The version numbers are documented in the o-parent manifest.



Example: 1 Major, 1 Service and 1 Enhancement Releases

Using the Semantic versioning, the Service Release represents the Patch Release, the Enhancement Release represents the Minor Release.

The diagram below has 3 Named Releases (delivered chronologically in that order):

1. 1.0.0-ONAP
2. 1.0.1-ONAP
3. 1.1.0-ONAP

1.0.0-ONAP is a Major Release.

1.0.1-ONAP is a Service Release and is released to deliver (for example) a security issue in "AAI cpt 1" component. Note the third digit is now "**4**".

1.1.0-ONAP is an Enhancement Release and is released to deliver (for example) a backward compatible functionality in "AAF cpt 1" and "AAF cpt 2" components. Note the second digits are now "**1**" and "**7**". Note also there is no version continuity in "AAF cpt 2" between the Service (4) and the Enhancement (7) Releases. This is acceptable as the team may have delivered intermediate "Artifact Release" that were not embedded with a "Named Release".

Named Release: 1.0.0-ONAP	Named Release: 1.0.1-ONAP	Named Release: 1.1.0-ONAP
(Major Release)	(Service Release)	(Enhancement Release)
AAI Docker Image: 1.0.0	AAI Docker Image: 1.0.1	AAI Docker Image: 1.0.1
AAI cpt 1: 2.0.3	AAI cpt 1: 2.0.4	AAI cpt 1: 2.0.4
AAI cpt 2: 1.4.2	AAI cpt 2: 1.4.2	AAI cpt 2: 1.4.2
AAI cpt 3: 0.0.9	AAI cpt 3: 0.0.9	AAI cpt 3: 0.0.9
Other stuff: 1.2.6	Other stuff: 1.2.6	Other stuff: 1.2.6
AAF Docker Image: 2.0.0	AAF Docker Image: 2.0.0	AAF Docker Image: 2.1.0
AAF cpt 1: 2.0.3	AAF cpt 1: 2.0.3	AAF cpt 1: 2.1.3
AAF cpt 2: 2.4.2	AAF cpt 2: 2.4.2	AAF cpt 2: 2.7.2
AAF cpt3: 1.0.9	AAF cpt 3: 1.0.9	AAF cpt 3: 1.0.9
Other stuff: 1.2.7	Other stuff: 1.2.7	Other stuff: 1.2.7

Semantic Versioning

All versioning uses a semantic versioning approach. In particular:

Given a version number MAJOR.MINOR.PATCH, increment the:

- MAJOR version when you make incompatible API changes,
- MINOR version when you add functionality in a backwards-compatible manner, and
- PATCH version when you make backwards-compatible bug fixes.

More details at: http://semver.org

Branching and Merging

Basic Principles

- Keep things as simple as possible: 1 Development branch on central repo (Gerrit)
- Do not create on central repo tons of intermediate development branches but one development branch.
- Do whatever you want and need on your local repository.
- **Push your changes to central repository at least once a day**. The more often you push your change, the easier it is to fix problem (simply because you just did the work) and fixing things should become a daily routine, not an end of release nightmare.
- Stay up to date: **often fetch and pull** from central repo. That will avoid you staying late at night.
- In case you break the build refer to Continuous Integration Practice.

To avoid breaking build refer to Continuous Integration Practice. Yes, CI is the foundation in ONAP.

Tagging Daily Builds in ONAP

Builds that have passed CI testing that includes automated deployment of the base ONAP system to either Openstack or Kubernetes and a minimum spanning set of test including Robot testing, selected REST calls up to and including the entire vFirewall use case - will be tagged across all the repos. This will aide developers, testers and deployers when they wish to target a known working build or reproduce an issue flagged to a particular tagged build. Using tagged builds will allow use of ONAP away from the head of the master branch.

Branching and Merging in ONAP

As a guiding principle, all ONAP development occurs in the **Master branch**.
First ONAP Release is named Amsterdam. Second ONAP Release is named Beijing. By convention, ONAP follows the major cities system naming.

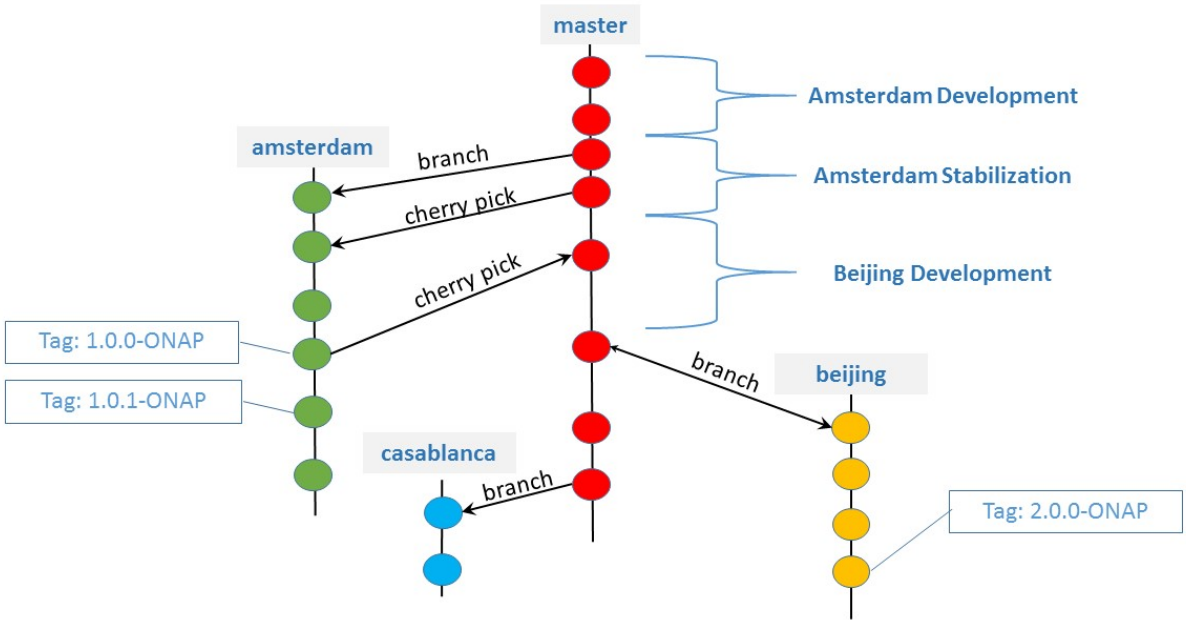
At one point in the Release (around RC0-RC1), we have to limit what get into the final Release.

To make this happens, we branch out from Master, create a delivery branch (Amsterdam, Beijing, Casablanca,...), and lock the delivery branch to **strictly control** what goes into delivery branch. Developers continue their **non delivery branch work in Master**.
At the end of each Release, artifacts are **tag** in Gerrit, according to above versioning principles.

Now things happen and we may need to deliver a hot fix (also called patch) into the latest available Release.
To perform the patch delivery, we simply perform the hot fix in the delivery branch, and tag it (1.0.1-ONAP).
Once the hot fix is delivered (and everyone is back to happiness), we will decide how to carry over the changes into the current Release branch (Beijing branch on the case below). Most probably cherry pick technique will apply.
It may also happen that we need to carry over into the latest available Release some fixes or new backward functionalities that were discovered while developing current Release (Beijing). Here too, we will decide on the best options to carry over changes into the current available Release and make a new Release (1.1.0-ONAP).

As a guiding principle, the ONAP community keeps its focus on developing in Master branch. **Hot fix process will be more the exception than the rule**.

After the delivery branch is created, we will have a very short delivery team to control the final content of the Release. The delivery Team composed of 1 member representing each project will analyze the impact of every defect and decide its inclusion or not into the delivery release. **Jira** will be used to support all documentation and **decisions**.
The Gerrit merge process will be limited to the delivery team.



API Versioning

For the API versioning strategy, refer to: ONAP API Common Versioning Strategy (CVS) Guidelines

While this addresses a few of the questions below, some thought is required in the relation to a release. Questions to answer:

- Who assigns version numbers?
- same as for repos. Projects decide
- Where do we track available APIs for a given release ?
- -> we need something that has a similar role to O-Parent
- What's our policy of depreciating APIs?
- -> I assume we would require MAJOR API versions to be available for extended periods of time