OCI/Squashfs Container Runtime

What Is It?

OCI/Squashfs is a YARN nodemanager container runtime that allows containers to leverage Docker images. However unlike the existing Docker runtime, the images are fetched from HDFS via the YARN distributed cache rather than from the Docker registry or requiring images to be preloaded into Docker on each node. Docker does ***not*** need to be installed on the nodes in order for the runtime to work.

Why Another Docker Runtime?

Using the current Docker runtime has some drawbacks:

Docker Daemons Dependency

The Docker daemons, `dockerd` and `containerd`, must be running on the system in order for the runtime to function.

Single, Reliable Image Store Requirement

Docker supports only a single place where image data is stored and that storage must be 100% reliable. If the Docker image store drive fails then all future container launches can fail, meaning a single disk failure can take out the entire node.

Requiring a single reliable location prevents spreading the images across the node's data disks unless the data is replicated across the disks (e.g.: in a RAID configuration) which is not an efficient use of space. Creating the RAID setup would also require the admin to preconfigure a maximum amount of space for Docker images.

Docker Registry Issues at Scale

Using the Docker runtime on a large scale YARN cluster can overwhelm the Docker registry. In practice this requires admins to pre-load a Docker image on all the cluster nodes in a controlled fashion before a large job requesting the image can run.

Image Costs in Time and Space

Docker stores each image layer as a tar.gz archive. In order to use the layer, the compressed archive must be unpacked into the node's filesystem. This can consume significant disk space, especially when the reliable image store location capacity is relatively small. In addition unpacking an image layer takes time, especially when the layer is large or contains thousands of files. This additional time for unpacking delays container launch beyond the time needed to transfer the layer data over the network.

How Is This Runtime Different?

The new container runtime avoids the drawbacks listed above in the following ways:

No Docker Dependencies On The Node

Docker does ***not*** need to be installed on each node, nor is there a dependency on a daemon or service that needs to be started by an admin before containers can be launched. All that is required to be present on each node is an OCI-compatible runtime like `runc`.

Image Layers Stored Across Existing YARN Disks

Instead of requiring a single, reliable path, image data is stored on the same disks YARN uses for application data. The image layers can be spread across different disks, allowing the node to store significantly more images.

The YARN application data disks can fail which means image data may be lost. Similar to disk loss of application data in YARN today, containers running on an impacted image may fail. However future containers requesting the same image will trigger the impacted image layers to be re-downloaded to other disks, and those container launches can proceed normally. This prevents a single disk failure from disabling the node.

Leverages Distributed File Systems For Scale

Images can be fetched via HDFS or other distributed file systems instead of the Docker registry. This prevents a large cluster from overwhelming a Docker registry when a wide job causes all of the nodes to request an image at once. This also allows large clusters to run jobs more dynamically, as images would not need to be pre-loaded by admins on each node to prevent a large Docker registry image request storm.

Smaller, Faster Images On The Node

The new runtime handles layer localization directly, so layer formats other than tar archive can be supported. For example, each image layer can be converted to squashfs images as part of copying the layers to HDFS. squashfs is a file system optimized for running directly on a compressed image. With squashfs layers the layer data can remain compressed on the node saving disk space. Container launch after layer localization is also faster, as the layers no longer need to be unpacked into a directory to become usable.

How Does The Runtime Work?

Before requesting resources to be localized, each container calls a hook in the executor to possibly adjust the list of resources to request. By default all container runtimes will simply return the original list of resources requested by the container start context, but this runtime will

additionally request the layer data for the container's Docker image. The image layers will be localized to the YARN disks by the ResourceLocalizationService, just like other container resources. If the layers have already been requested by other containers and still remain in the local cache then they will not be downloaded again.

When the container launches the individual layers are stitched together into a single root filesystem via overlayfs. The container is then launched via the configured OCI runtime, using the root filesystem created earlier as the container's root filesystem. When the OCI runtime returns, the runtime tears down the root mount and any layer mounts that are no longer referenced by containers.

Localization of Image Data

The following figure shows the flow of how the Docker image is translated into resource localization requests for the container.

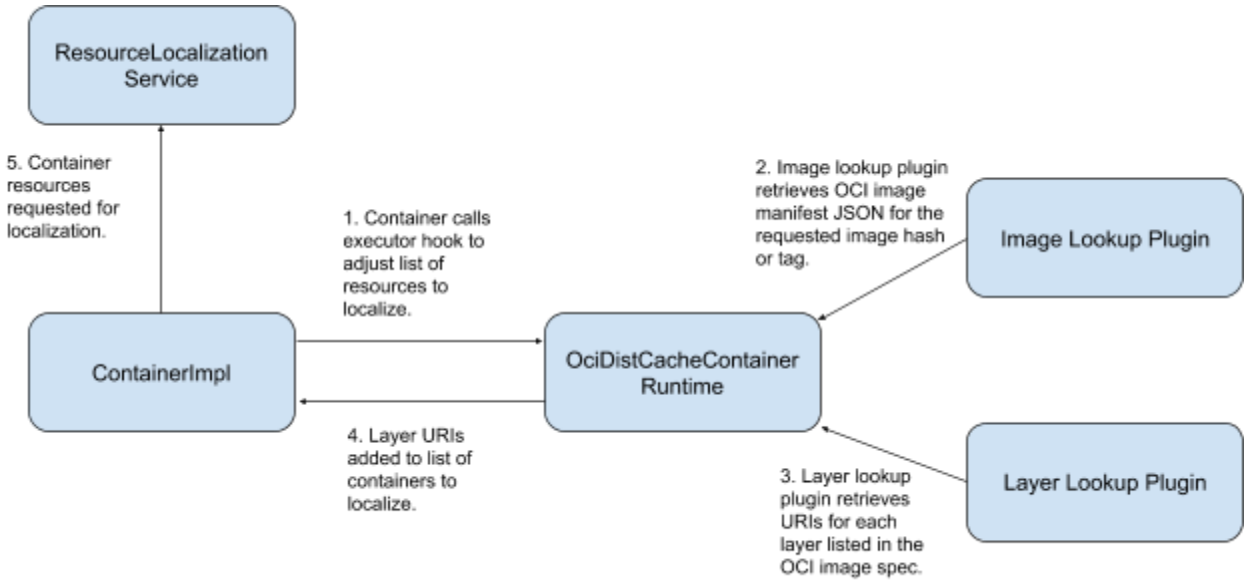


Image Lookup Plugin

The runtime abstracts how an image URI, hash, or tag is used to lookup the corresponding OCI image manifest. The plugin could query the Docker registry, search locations in HDFS, use a flat configuration file on the node, or any combination of these or other approaches. The plugin provides flexibility to tailor the image manifest lookup process to meet the needs of specific installations.

Layer Lookup Plugin

The runtime abstracts how layer hashes are mapped to URIs that can be used as resource localization requests. The default implementation would search a configured HDFS location for the corresponding layer data, but other implementations could leverage other distributed systems such as cloud storage.

Launching the Container

After the image layers have been localized along with the container's other resources, ContainerImpl will send a launch request to the ContainerLauncher which in turn will invoke the container executor to launch the container. The runtime in the nodemanager will emit a command file, formatted in JSON, to control the launch of the container. After this command file is written the native container executor is launched to handle the rest of the container launch process. The container executor is launched with an option specifying an OCI/squashfs container is desired and the path to the command file.

Container Executor Command File

The command file contains all of the information required for the native container executor to configure and launch the container. The config file contains a single JSON object that has the following contents:

- `username` : the name of the user running the container
- `applicationId` : the YARN application ID of the container
- `containerId` : the YARN container ID of the container
- `pidFile` : the path where the container's process ID will be written on launch
- `containerScriptPath` : path to the container launch script
- `containerCredentialsPath` : path to the container credentials file
- `localDirs` : array of YARN local directories to use
- `logDirs` : array of YARN log directories to use
- `layers` : array of layer descriptor objects where each object has the following contents:
    - mediaType : the MIME type of the layer data (application/vnd.squashfs)
    - path : the local path to the layer data
- `ociRuntimeConfig` : an partial OCI runtime configuration describing the required bind mounts, working directory, launch arguments, environment variables, etc. for the container. See the [OCI runtime config specification] for details on the contents of this object.

Container Executor Image and Layer File System Database

The runtime uses a file system database to store mounted image and layer file systems. Lock files in the database are used to coordinate multiprocess access to the mounted file systems. The database location can be configured in the container executor configuration, and by default this location is `/run/yarn-container-executor/`. The database should ***not*** persist

across reboots as it only tracks mounted file systems which are destroyed by the kernel when the system resets. Therefore ideally the database should be placed in a `tmpfs` file system or similar ephemeral storage that is cleared on a system reboot.

File System Database Layout

The database under the configured path is organized as follows:

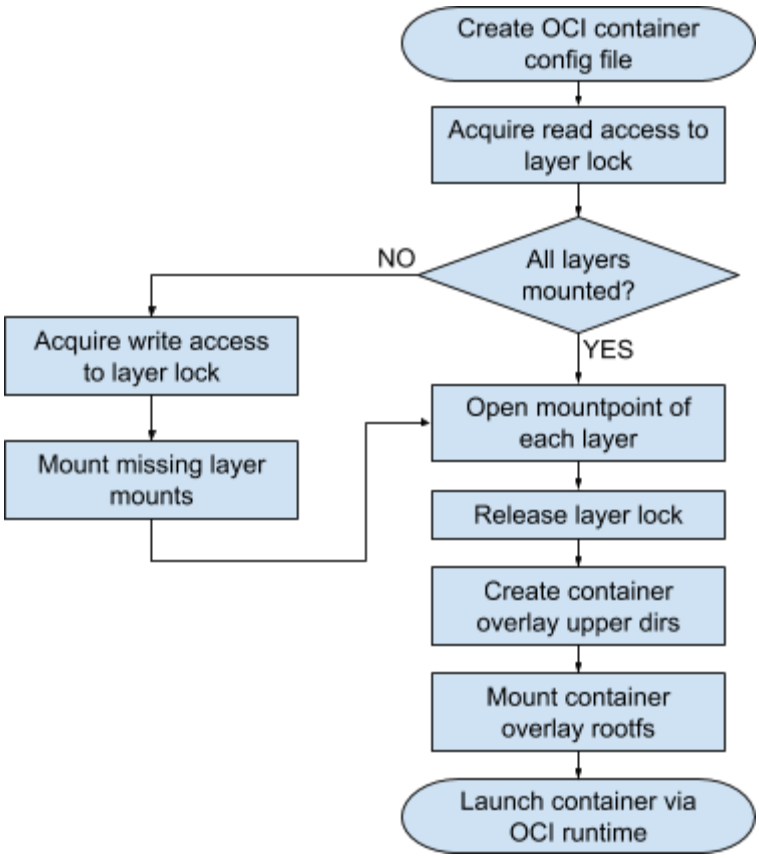
- `./layers/`
  - Layers that need to be mounted before use are mounted here with the layer's SHA256 as the mount point directory name
- `./layers/lock`
  - Used to control mulitprocess access to the layer mounts
- `./`*`container_id`*`/rootfs`
  - The mountpoint for the container's root filesystem
- `./`*`container_id`*`/upper`
  - The writeable, upper layer for the container's root filesystem
- `./`*`container_id`*`/work`
  - The overlay filesystem work directory for the container's root filesystem

File System Database Locking Policy

The layer lock file controls read/write access to layer mount points. If a mounted layer file system is needed after releasing the lock then it is critical to obtain an open file descriptor to that file system before releasing the lock. Failure to do so can lead to race conditions where another container's cleanup process deletes the just created but currently unreferenced file system before a new container launch opens the file system.

Container Executor Launch Steps

Container launch is described in the following flowchart:



Creating the OCI Container Config File

The runtime in the YARN nodemanager writes an OCI container configuration file to the container's nmPrivate area on one of the YARN disks. This allows the OCI container configuration file to be automatically deleted when the container completes and the nodemanager removes the container directories.

Mounting Layer File Systems

If the layer data is a squashfs or other file system image then the layer file system image must be mounted in order to make the layer data available for use. For squashfs layers, the container executor would setup a loopback device on the squashfs image and then mount a squashfs filesystem using that loopback device.

Mounting the Image Root File System

Image root filesystems are used by containers that have a read-only file system. The image root file system is mounted by overlayfs where each layer of the image is one of the lower file

systems in overlayfs. The lower file systems order in the overlayfs mount options matches the order of the layers in the image manifest. No upper file system is used for an image root file system since the file system is read-only.

Creating the Container Root File System

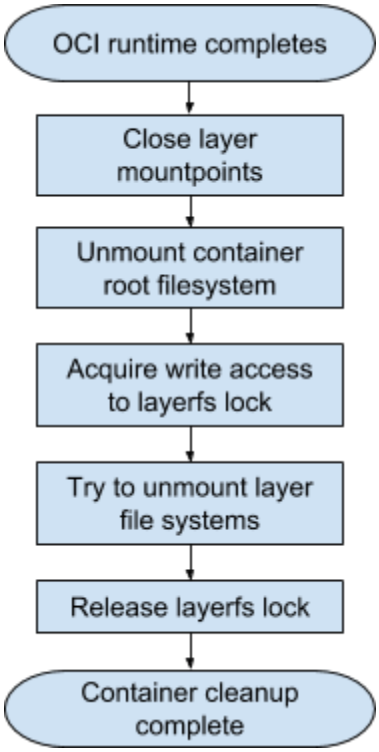
Containers with read-write root file system also have the root image mounted with overlayfs. The lower file system is the same image root file system that is mounted for read-only containers and the upper file system is where writes in the root file system should be stored. This would likely be somewhere under one of the container's directories on the YARN disks.

Launching the Container via OCI

The container executor uses the OCI runtime specified in the container executor configuration to launch the container. The default OCI runtime is `/usr/bin/runc`. The OCI container config file created in the previous step is passed as the configuration of the container to the OCI runtime.

Cleaning Up After Containers

Container cleanup is described in the following flowchart:



Layer File System Cleanup

The native container executor attempts to unmount every layer used by the container's image under `./layers/`. If the unmount is successful then the mount point entry under `./layers/`

is removed. If the umount returns an error code of `EBUSY` then this failure is ignored and the mount point left as-is, as this indicates the layer is still being used by another container.

TODOS FOR THIS DOCUMENT

- More details on default image tag/hash lookup plugin
- More details on default layer lookup plugin
- More details on how plugin results are cached (likely stashing layer order in ContainerImpl object or otherwise associating raw image hash with container in some way that is robust to the container failing before launch).