[Design] Unified Resource Statistics Collection per node

Vinod Kumar Vavilapalli and Sanjay Radia
Last Modified: March 10 2015

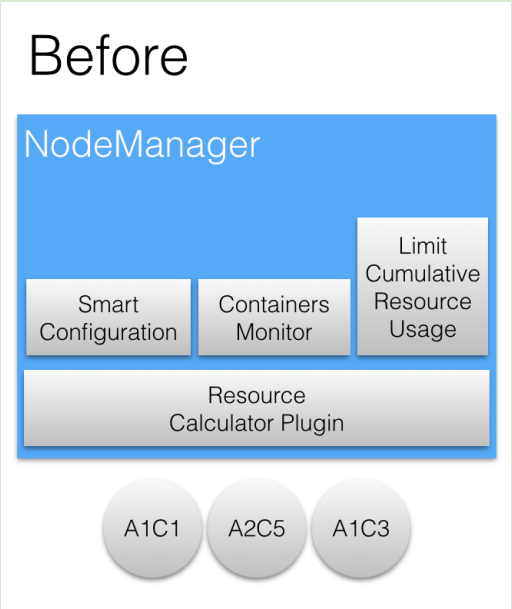
Current State of Resource Statistics in YARN

Today in YARN, NodeManager collects statistics like per container resource-usage and overall physical resources available on the machine. Currently this is used internally in YARN by the NodeManager for

- **Automatically determining the capacity of resources on node**: We call this smart configuration: NodeManager auto detects the capacity of memory/cpu available on the machine and uses it for YARN unless explicitly overridden by admins.

- **Enforcing resource usage to what is reserved per container**: Currently only memory resource is enforced explicitly in YARN. CPU for example is enforced through OS mechanisms like cgroups. Memory usage enforcement is applied on each container depending on a default container size unless explicitly overridden by applications. In addition, the platform can also apply limits on the cumulative resource usage across all YARN containers running on a node.

This current state of the art thus looks like the following:



Next steps in the evolution

It is useful to extend the existing architecture and collect statistics for usage *beyond* the existing use-cases. Statistics collected can be used in various ways:

1. Surface statistics (machine level and container specific) to end-users on its hosted UI

2. Report container statistics to per-application Timeline Aggregators (the Next-gen Timeline service effort - YARN-2928) so that users can reason about individual application's resource usages - point in time as well as historical.

3. Report statistics of running containers to ResourceManager so that it can do smarter scheduling based on actual resources usage (like YARN-2745) instead of limiting itself to "resource reservations".

4. Expose the statistics to applications, users and external systems via a generally accessible web-service and/or a client library. Other Hadoop Systems like HDFS can figure out the point-in-time resource usage on a machine - for e.g. current per-disk used bandwidth - and make intelligent decisions.

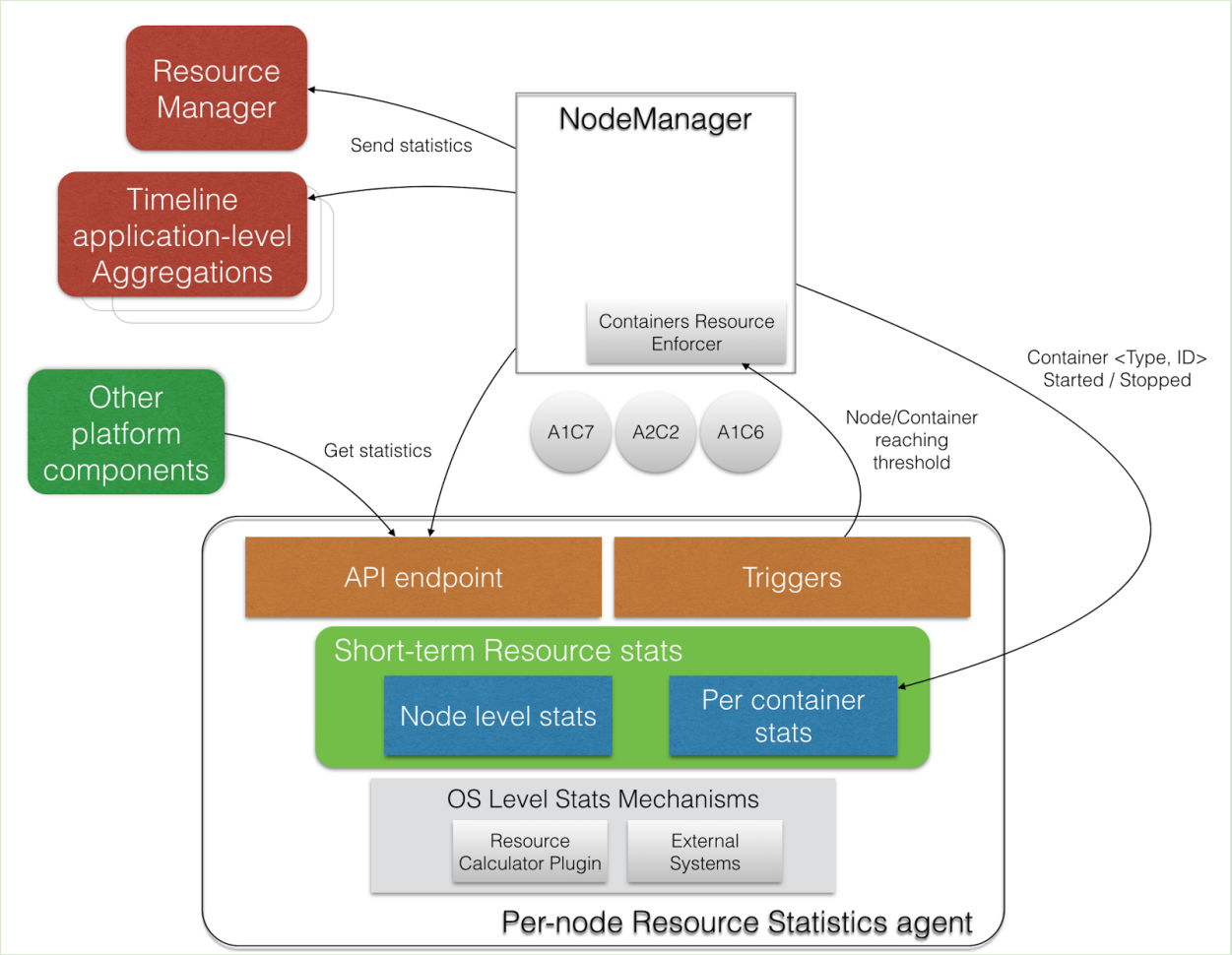
Today, use-cases (2) and (3) are being worked upon in parallel, with potential code duplication. The proposal in this document aims to reconcile them in addition to enabling (1) and (4). Note that (2) and (3) can still be addressed without new layers being suggested in

this document because they are internal to YARN. In order to also enable (4), I propose we introduce a new common layer.

Proposal

Architecture

We add a new agent singularly responsible for collecting resource-statistics per machine as shown below.



Chief architectural design choices

- The agent is **decoupled** from YARN (though deployed **inside** NodeManager in practice, either embedded or separate processes) so that users and external systems alike do not need to speak YARN's parlance. This translates to a generic web-service for reading the collected information.

- The agent will be **lightweight** w.r.t state, only be a per node aggregator: it simply collects information, keeps them in memory for a short duration (admin configurable, order of mins), is **not** responsible for storage.

- Systems that require the statistics to be either rolled up in some dimension and/or persisted for longer durations are implemented on **top** of this layer.

- The agent understands the machine's resource usage and a generic concept of a container.

Details

Statistics collection

At the lowest level, we have mechanisms to obtain OS level statistics. This includes the following types of information

- ***Capacity of machine resources***: total RAM available, number of disks attached, their max speed and throughput, CPU speeds and number of cores, network speed etc. This can be determined automatically at startup time though can be explicitly overridden by admin.

- ***Current usage/free resources on the node***: current total RAM free, total disks space free per disk, unused CPU and network.

- ***Per-container resource usage***: For each container, total physical memory used, total virtual memory used, total CPU, disk space etc.

We **do not** wish to reinvent the statistics collection. We will continue to use existing mechanisms to collect stats including

- Existing ***ResourceCalculatorPlugin utilities*** in YARN. Plugins are already available for Linux/Windows to obtain machine level statistics

- Existing ***Procfs based stats collection per container*** in YARN. Today we use this information to enforce memory limits on each containers

- We will further enhance our existing stats collection via OS utilities like hdParm, ethtool and other existing libraries.

- We can also have a small plugin architecture to interface with **external systems** like ganglia etc.

Note that the mechanisms used are a function of (OS type, supported Container types, utilities available, List of Statistics to be collected).

Entities tracked

The agent deals with two types of entities - (1) the machine itself and (2) containers

Containers

Containers tracked by the agent are considered to be generic and are identified by the tuple ***<Container Type, Container Identifier>***. Today, YARN supports “Process-trees” as *Container-Type* and “root PID” as the *Container-Identifier*. This can be extended to include other new notions of containers. For example, we can have a “Docker Container” *Container-Type* and a “Docker ID” *Container-Identifier*. Depending on the Container-Type, the agent will use the right underlying mechanisms of statistics collection, for e.g. Docker has ways of obtaining usage per Docker Container.

Life cycle of Containers

To be able to start and stop tracking of each <Container Type, Container Identifier>, the agent needs to be able to know when a Container started and when it stopped. For YARN’s containers, NodeManager will need to send an event to the agent whenever a container gets started or stopped. This can happen via many ways

- A file based communication - NM writes the ContainerType and ID to a file and the agent reads it - lightweight, easy to secure etc.
- Add a new protocol for this
- Or extend the read end-point to accept PUTs and let NodeManager to be the only one register/deregister containers as needed.

Container memory enforcement

In the updated architecture, it’s the agent that is periodically obtaining memory usage. To enforce today’s memory limits on each container, we have two choices (1) implement the enforcement in the agent or (2) leave it inside the NodeManager with notifications from the agent. To keep the architecture clean, and avoiding putting policies (of when to kill something) inside the agent, I opted to have a pluggable trigger mechanism inside the agent. Whoever starts the agent can choose to configure some triggers like the following

- Send an event when a container goes beyond a specific physical memory limit, virtual memory limit or CPU usage
- Inform me when the overall machine is beyond a specific threshold of CPU load

YARN is completely out of way w.r.t isolation with things like Cgroups on Linux,, so that doesn't have any significance in this doc. But wherever YARN is involved w.r.t isolation, this architecture should work.

We now consider each of the new use cases and how they are implemented in this architecture.

NodeManager surfacing statistics on its UI

As discussed before, NodeManager/ResourceManager today do not expose the dynamically changing, point-in-time resource-usage of the machine and/or the individual YARN containers. Using the agent now, NM can extract this information, surface it on its UI. One item ***todo*** is to see if we should have a shortcut read from the NM to the agent in the embedded deployment mode or the NM always goes through the web-service.

Per-application Timeline Aggregators

With the agent in place, NodeManager can periodically extract per-container statistics and send them across to the per-application Timeline Aggregators sitting elsewhere. If and when we implement things like rack-level Timeline Aggregators to collate information for a single rack, the same can be forwarded via the Agent->NM statistics flow-path.

Smart scheduling

Similar to Timeline Aggregators, NodeManager can explicitly report statistics of the machine and/or running containers to the ResourceManager for smarter scheduling. I deliberately chose not to have a direct push from the agent to the RM for clearer separation of concerns and in order to optimize on the number of connections originating from each node to the ResourceManager.

This will also enable smarter scheduling decisions by the individual YARN *applications*. By accessing the same data, ApplicationMasters can observe the actual resource usage of their work, compare that to the expected usage and can size their containers better and make optimizations at runtime.

Usage by other systems

HDFS NameNode/DataNode can now directly reach into the agent's web-service to understand point-in-time state of the machine (per-disk bandwidth etc) and use them to do better isolation/scheduling. The fact that the agent is a generic entity and exposes a simple web-service should help make the integration with other platform components straightforward.

Extensions

So far, I've deliberately scoped the agent to be responsible for only resource-statistics *aggregation*. Forwarding and storage are left for layers above.

One interesting extension of this is to also have this agent as the central place where platform components and/or containers can **push** metrics information. So that entities like UI, ResourceManager, Timeline Aggregators can consume that information via the same interfaces.

The above is useful, for example, in scaling Timeline Service next-gen to also be able to let individual containers send Timeline information through this channel. Today, it is left up to individual YARN frameworks to establish a channel from the containers to the AM which then sends Timeline information to the Aggregators.