layout: documentation

Mesos Fetcher

Experimental support for the Mesos fetcher *cache* is introduced in Mesos 0.23.0.

In this context we loosely regard the term "downloading" as to include copying from local file systems.

What is the Mesos fetcher?

The Mesos fetcher is a mechanism to download resources into the sandbox directory of a task in preparation of running the task. As part of a TaskInfo message, the framework ordering the task's execution provides a list of CommandInfo::URI protobuf values, which becomes the input to the Mesos fetcher.

By default, each requested URI is downloaded directly into the sandbox directory and repeated requests for the same URI leads to downloading another copy of the same resource. Alternatively, the fetcher can be instructed to cache URI downloads in a dedicated directory for reuse by subsequent downloads.

The Mesos fetcher mechanism comprises of these two parts:

1. The slave-internal Fetcher Process (in terms of libprocess) that controls and coordinates all fetch actions. Every slave instance has exactly one internal fetcher instance that is used by every kind of containerizer (except the external containerizer variant, which is responsible for its own approach to fetching).
2. The external program "mesos-fetcher" that is invoked by the former. It performs all network and disk operations except file deletions and file size queries for cache-internal bookkeeping. It is run as an external OS process in order to shield the slave process from I/O-related hazards. It takes instructions in form of an environment variable containing a JSON object with detailed fetch action descriptions.

The fetch procedure

Frameworks launch tasks by calling the scheduler driver method launchTasks(), passing CommandInfo protobuf structures as arguments. This type of structure specifies (among other things) a command and a list of URIs that need to be "fetched" into the sandbox directory on the the slave node as a precondition for task execution. Hence, when the slave receives a request go launch a task, it calls upon its fetcher, first, to provision the specified resources into the sandbox directory. If fetching fails, the task is not started and the reported task status is TASK_FAILED.

All URIs requested for a given task are fetched sequentially in a single invocation of mesos-fetcher. Here, avoiding download concurrency reduces the risk of bandwidth issues

somewhat. However, multiple fetch operations can be active concurrently due to multiple task launch requests.

The URI protobuf structure

Before mesos-fetcher is started, the specific fetch actions to be performed for each URI are determined based on the following protobuf structure. (See "include/mesos/mesos.proto" for more details.)

```
message CommandInfo {
  enum Cache {
    FETCH = 0;
    FORCE = 1;
  }

  message URI {
    required string value = 1;
    optional bool executable = 2;
    optional bool extract = 3 [default = true];
    optional Cache cache = 4;
  }
  ...
  optional string user = 5;
}
```

The field "value" contains the URI.

If the "executable" field is "true", the "extract" field is ignored and has no effect.

Specifying a user name

The framework may pass along a user name that becomes a fetch parameter. This causes its executors and tasks to run under a specific user. However, if the "user" field in the CommandInfo structure is specified, it takes precedence for the affected task.

If a user name is specified either way, the fetcher first validates that it is in fact a valid user name on the slave. If it is not, fetching fails right here. Otherwise, the sandbox directory is assigned to the specified user as owner (using chown) at the end of the fetch procedure, before task execution begins.

The user name in play has an important effect on caching. Caching is managed on a per-user base, i.e. the combination of user name and "uri" uniquely identifies a cacheable fetch result. If no user name has been specified, this counts for the cache as a separate user, too. Thus cache files for each valid user are segregated from all others.

This means that the exact same URI will be downloaded and cached multiple times if different users are indicated.

Executable fetch results

By default, fetched files are not executable.

If the field "executable" is set to "true", the fetch result will be changed to be executable (by "chmod") for every user. This happens at the end of the fetch procedure, in the sandbox directory only. It does not affect any cache file.

Archive extraction

If the "extract" field is "true", which is the default, then files with extensions that hint at packed or compressed archives (".zip", ".tar", et.al.) are unpacked in the sandbox directory.

In case the cache is bypassed, both the archive and the unpacked results will be found together in the sandbox. In case a cache file is unpacked, only the extraction result will be found in the sandbox.

Bypassing the cache

By default, the URI field "cache" is not present. Then the fetcher downloads directly into the sandbox directory.

The same also happens dynamically as a fallback strategy if anything goes wrong when preparing a fetch operation that involves the cache. In this case, a warning message is logged. Possible fallback conditions are:

- The server offering the URI does not respond or reports an error.
- The URI's download size could not be determined.
- There is not enough space in the cache, even after attempting to evict files.

Fetching through the cache

If the URI's "cache" field has the value "FETCH", then the fetcher cache is in effect. If a URI is encountered for the first time (for the same user), it is first downloaded into the cache, then copied to the sandbox directory from there. If the same URI is encountered again, and a corresponding cache file is resident in the cache or still en route into the cache, then downloading is omitted and the fetcher proceeds directly to copying from the cache. Competing requests for the same URI simply wait upon completion of the first request that occurs. Thus every URI is downloaded at most once (per user) as long as it is cached.

Every cache file stays resident for an unspecified amount of time and can be removed at the fetcher's discretion at any moment, except while it is in direct use:

- It is still being downloaded by this fetch procedure.
- It is still being downloaded by a concurrent fetch procedure for a different task.
- It is being copied or extracted from the cache.

Once a cache file has been removed, the related URI will thereafter be treated as described above for the first encounter.

Forcing a cache update

If the URI's "cache" field has the value "FORCE", then the fetcher first removes the cache file related to the given URI if there is one. Thereafter it proceeds exactly as described above for the value "FETCH". This effectively forces an unconditional cache content update for one URI (and user).

Recommended practice:

1. Every framework should use "FORCE" every time it requests any given URI for the first time since framework startup and until a download succeeds.
2. Only after at least one successful download, it should use "FETCH".
3. Once "FETCH" can be used, it should always be used.

Thus the framework is not exposed to stale content from prior frameworks that may potentially have executed a long time ago.

(A future feature may force updates based on checksums.)

Determining resource sizes

Before downloading a resource to the cache, the fetcher first determines the size of the expected resource. It uses these methods depending on the nature of the URI.

- Local file sizes are probed with systems calls (that follow symbolic links).
- HTTP/HTTPS URIs are queried for the "content-length" field in the header. This is performed by CURL. The reported asset size must be greater than zero or the URI is deemed invalid.
- FTP/FTPS is not supported at the time of writing.
- Everything else is queried by the local HDFS client.

If any of this reports an error, the fetcher then falls back on bypassing the cache as described above.

WARNING: Only URIs for which download sizes can be queried upfront and for which accurate sizes are reported reliably are eligible for any fetcher cache involvement. Do not use any cache feature with any URI for which you have any doubts! If actual cache file sizes exceed the physical capacity of the cache directory in any way, all further slave behavior is completely unspecified.

The fetcher cache has limited means to correct for falsely advertised file sizes after downloading. It will dynamically adjust its own bookkeeping according to the actual sizes of its cache files. But this only works as long as there is enough physical space. If you know for sure that size aberrations are within certain limits you can specify a cache directory size that is smaller than your actual physical volume and this may work.

Cache eviction

After determining the prospective size of a cache file and before downloading it, the cache attempts to ensure that at least as much space as is needed for this file is available and can

be written into. If this is immediately the case, the requested amount of space is simply marked as reserved. Otherwise, missing space is freed up by "cache eviction". This means that the cache removes files at its own discretion until the given space target is met or exceeded.

The eviction process fails if too many files are in use and therefore not evictable or if the cache is simply too small. Either way, the fetcher then falls back on bypassing the cache for the given URI as described above.

If multiple evictions happen concurrently, each of them is pursuing its own separate space goals. However, leftover freed up space from one effort is automatically awarded to others.

Slave flags

It is highly recommended to set these flags explicitly to values other than their defaults or to not use the fetcher cache in production.

- "fetcher_cache_size", default value: enough for testing.
- "fetcher_cache_dir", default value: somewhere inside the directory specified by the "work_dir" flag, which is OK for testing.

Recommended practice:

- Use a separate volume as fetcher cache. Do not specify a directory as fetcher cache directory that competes with any other contributor for the underlying volume's space.
- Set the cache directory size flag of the slave to less than your actual cache volume's physical size. Use a safety margin, especially if you do not know for sure if all frameworks are going to be compliant.

Ultimate remedy:

You can disable the fetcher cache entirely on each slave by setting its "fetcher_cache_size" flag to zero bytes.