Apache Aurora Incubating

Documentation

Aurora is a service scheduler that schedules jobs onto *Mesos*, which runs tasks at a specified cluster. Typical services consist of up to hundreds of task replicas.

Aurora provides a *Job* abstraction consisting of a *Task* template and instructions for creating near-identical replicas of that Task (modulo things like "instance id" or specific port numbers which may differ from machine to machine).

Terminology Note: *Replicas* are also referred to as *shards* and *instances*. While there is a general desire to move to using "instances", "shard" is still found in commands and help strings.

Typically a Task is a single *Process* corresponding to a single command line, such as `python2.6 my_script.py`. However, sometimes you must colocate separate Processes together within a single Task, which runs within a single container and `chroot`, often referred to as a "sandbox". For example, if you run multiple cooperating agents together such as `logrotate`, `installer`, and master or slave processes. *Thermos* provides a Process abstraction under the Mesos Tasks.

To use and get up to speed on Aurora, you should look the docs in this directory in this order:

1. How to deploy Aurora or, how to install Aurora on virtual machines on your private machine (the Tutorial uses the virtual machine approach).
2. As a user, get started quickly with a Tutorial.
3. For an overview of Aurora's process flow under the hood, see the User Guide.
4. To learn how to write a configuration file, look at our Configuration Tutorial. From there, look at the Aurora + Thermos Reference.
5. Then read up on the Aurora Command Line Client.
6. Find out general information and useful tips about how Aurora does Resource Isolation.

To contact the Aurora Developer List, email dev@aurora.incubator.apache.org. You may want to read the list archives. You can also use the IRC channel `#aurora` on `irc.freenode.net`

Contents:

CHAPTER 1

Aurora Client Commands

The most up-to-date reference is in the client itself: use the `aurora help` subcommand (for example, `aurora help` or `aurora help create`) to find the latest information on parameters and functionality. Note that `aurora help open` does not work, due to underlying issues with reflection.

1.1 Introduction

Once you have written an `.aurora` configuration file that describes your Job and its parameters and functionality, you interact with Aurora using Aurora Client commands. This document describes all of these commands and how and when to use them. All Aurora Client commands start with `aurora`, followed by the name of the specific command and its arguments.

Job keys are a very common argument to Aurora commands, as well as the gateway to useful information about a Job. Before using Aurora, you should read the next section which describes them in detail. The section after that briefly describes how you can modify the behavior of certain Aurora Client commands, linking to a detailed document about how to do that.

This is followed by the Regular Jobs section, which describes the basic Client commands for creating, running, and manipulating Aurora Jobs. After that are sections on Comparing Jobs and Viewing/Examining Jobs. In other words, various commands for getting information and metadata about Aurora Jobs.

1.2 Cluster Configuration

The client must be able to find a configuration file that speciies available clusters. This file declares shorthand names for clusters, which are in turn referenced by job configuration files and client commands.

The client will load at most two configuration files, making both of their defined clusters available. The first is intended to be a system-installed cluster, using the path specified in the environment variable `AURORA_CONFIG_ROOT`, defaulting to `/etc/aurora/clusters.json` if the environment variable is not set. The second is a user-installed file, located at `~/.aurora/clusters.json`.

A cluster configuration is formatted as JSON. The simplest cluster configuration is one that communicates with a single (non-leader-elected) scheduler. For example:

```
[{
  "name": "example",
  "scheduler_uri": "localhost:55555",
}]
```

A configuration for a leader-elected scheduler would contain something like:

```
[{
  "name": "example",
  "zk": "192.168.33.2",
  "scheduler_zk_path": "/aurora/scheduler"
}]
```

1.3 Job Keys

A job key is a unique system-wide identifier for an Aurora-managed Job, for example `cluster1/web-team/test/experiment204`. It is a 4-tuple consisting of, in order, *cluster*, *role*, *environment*, and *jobname*, separated by /s. Cluster is the name of an Aurora cluster. Role is the Unix service account under which the Job runs. Environment is a namespace component like `devel`, `test`, `prod`, or `stagingN`. Jobname is the Job's name.

The combination of all four values uniquely specifies the Job. If any one value is different from that of another job key, the two job keys refer to different Jobs. For example, job key `cluster1/tyg/prod/workhorse` is different from `cluster1/tyg/prod/workcamel` is different from `cluster2/tyg/prod/workhorse` is different from `cluster2/foo/prod/workhorse` is different from `cluster1/tyg/test/workhorse`.

Role names are user accounts existing on the slave machines. If you don't know what accounts are available, contact your sysadmin.

Environment names are namespaces; you can count on `prod`, `devel` and `test` existing.

1.4 Modifying Aurora Client Commands

For certain Aurora Client commands, you can define hook methods that run either before or after an action that takes place during the command's execution, as well as based on whether the action finished successfully or failed during execution. Basically, a hook is code that lets you extend the command's actions. The hook executes on the client side, specifically on the machine executing Aurora commands.

Hooks can be associated with these Aurora Client commands.

- `cancel_update`
- `create`
- `kill`
- `restart`
- `update`

The process for writing and activating them is complex enough that we explain it in a devoted document, Hooks for Aurora Client API.

1.5 Regular Jobs

This section covers Aurora commands related to running, killing, renaming, updating, and restarting a basic Aurora Job.

1.5.1 Creating and Running a Job

```
aurora create <job key> <configuration file>
```

Creates and then runs a Job with the specified job key based on a `.aurora` configuration file. The configuration file may also contain and activate hook definitions.

`create` can take four named parameters:

- `-E NAME=VALUE` Bind a Thermos mustache variable name to a value. Multiple flags specify multiple values. Defaults to `[]`.
- `-o, --open_browser` Open a browser window to the scheduler UI Job page after a job changing operation happens. When `False`, the Job URL prints on the console and the user has to copy/paste it manually. Defaults to `False`. Does not work when running in Vagrant.
- `-j, --json` If specified, configuration argument is read as a string in JSON format. Defaults to False.
- `--wait_until=STATE` Block the client until all the Tasks have transitioned into the requested state. Possible values are: `PENDING`, `RUNNING`, `FINISHED`. Default: `PENDING`

1.5.2 Running a Command On a Running Job

```
aurora run <job_key> <cmd>
```

Runs a shell command on all machines currently hosting shards of a single Job.

`run` supports the same command line wildcards used to populate a Job's commands; i.e. anything in the `{{mesos.*}}` and `{{thermos.*}}` namespaces.

`run` can take three named parameters:

- `-t NUM_THREADS`, `--threads=NUM_THREADS`The number of threads to use, defaulting to `1`.
- `--user=SSH_USER` ssh as this user instead of the given role value. Defaults to None.
- `-e, --executor_sandbox` Run the command in the executor sandbox instead of the Task sandbox. Defaults to False.

1.5.3 Killing a Job

```
aurora kill <job key> <configuration file>
```

Kills all Tasks associated with the specified Job, blocking until all are terminated. Defaults to killing all shards in the Job.

The `<configuration file>` argument for `kill` is optional. Use it only if it contains hook definitions and activations that affect the kill command.

`kill` can take two named parameters:

- `-o, --open_browser` Open a browser window to the scheduler UI Job page after a job changing operation happens. When `False`, the Job URL prints on the console and the user has to copy/paste it manually. Defaults to `False`. Does not work when running in Vagrant.
- `--shards=SHARDS` A list of shard ids to act on. Can either be a comma-separated list (e.g. 0,1,2) or a range (e.g. 0-2) or any combination of the two (e.g. 0-2,5,7-9). Defaults to acting on all shards.

1.5.4 Updating a Job

```
aurora update [--shards=ids] <job key> <configuration file>
aurora cancel_update <job key> <configuration file>
```

Given a running job, does a rolling update to reflect a new configuration version. Only updates Tasks in the Job with a changed configuration. You can further restrict the operated on Tasks by using `--shards` and specifying a comma-separated list of job shard ids.

You may want to run `aurora diff` beforehand to validate which Tasks have different configurations.

Updating jobs are locked to be sure the update finishes without disruption. If the update abnormally terminates, the lock may stay around and cause failure of subsequent update attempts. `aurora cancel_update`unlocks the Job specified by its `job_key` argument. Be sure you don't issue `cancel_update` when another user is working with the specified Job.

The `<configuration file>` argument for `cancel_update` is optional. Use it only if it contains hook definitions and activations that affect the `cancel_update` command. The `<configuration file>` argument for `update` is required, but in addition to a new configuration it can be used to define and activate hooks for `update`.

`update` can take four named parameters:

- `--shards=SHARDS` A list of shard ids to update. Can either be a comma-separated list (e.g. 0,1,2) or a range (e.g. 0-2) or any combination of the two (e.g. 0-2,5,7-9). If not set, all shards are acted on. Defaults to None.
- `-E NAME=VALUE` Binds a Thermos mustache variable name to a value. Use multiple flags to specify multiple values. Defaults to `[]`.
- `-j, --json` If specified, configuration is read in JSON format. Defaults to `False`.
- `--updater_health_check_interval_seconds=HEALTH_CHECK_INTERVAL_SECONDS` Time interval between subsequent shard status checks. Defaults to `3`.

1.5.5 Renaming a Job

Renaming is a tricky operation as downstream clients must be informed of the new name. A conservative approach to renaming suitable for production services is:

1. Modify the Aurora configuration file to change the role, environment, and/or name as appropriate to the standardized naming scheme.
2. Check that only these naming components have changed with `aurora diff`.

   ```
   aurora diff <job_key> <job_configuration>
   ```

3. Create the (identical) job at the new key. You may need to request a temporary quota increase.

   ```
   aurora create <new_job_key> <job_configuration>
   ```

4. Migrate all clients over to the new job key. Update all links and dashboards. Ensure that both job keys run identical versions of the code while in this state.
5. After verifying that all clients have successfully moved over, kill the old job.

   ```
   aurora kill <old_job_key>
   ```

6. If you received a temporary quota increase, be sure to let the powers that be know you no longer need the additional capacity.

1.5.6 Restarting Jobs

`restart` restarts all of a job key identified Job's shards:

```
aurora restart <job_key> <configuration file>
```

Restarts are controlled on the client side, so aborting the `restart` command halts the restart operation.

`restart` does a rolling restart. You almost always want to do this, but not if all shards of a service are misbehaving and are completely dysfunctional. To not do a rolling restart, use the `-shards` option described below.

Note: `restart` only applies its command line arguments and does not use or is affected by `update.config`. Restarting does ***not*** involve a configuration change. To update the configuration, use `update.config`.

The `<configuration file>` argument for restart is optional. Use it only if it contains hook definitions and activations that affect the `restart` command.

In addition to the required job key argument, there are eight `restart` specific optional arguments:

- `--updater_health_check_interval_seconds`: Defaults to `3`, the time interval between subsequent shard status checks.

- `--shards=SHARDS`: Defaults to None, which restarts all shards. Otherwise, only the specified-by-id shards restart. They can be comma-separated `(0, 8, 9)`, a range `(3-5)` or a combination `(0, 3-5, 8, 9-11)`.
- `--batch_size`: Defaults to `1`, the number of shards to be started in one iteration. So, for example, for value 3, it tries to restart the first three shards specified by `--shards` simultaneously, then the next three, and so on.
- `--max_per_shard_failures=MAX_PER_SHARD_FAILURES`: Defaults to `0`, the maximum number of restarts per shard during restart. When exceeded, it increments the total failure count.
- `--max_total_failures=MAX_TOTAL_FAILURES`: Defaults to `0`, the maximum total number of shard failures tolerated during restart.
- `-o, --open_browser` Open a browser window to the scheduler UI Job page after a job changing operation happens. When `False`, the Job url prints on the console and the user has to copy/paste it manually. Defaults to `False`. Does not work when running in Vagrant.
- `--restart_threshold`: Defaults to `60`, the maximum number of seconds before a shard must move into the `RUNNING` state before it's considered a failure.
- `--watch_secs`: Defaults to `30`, the minimum number of seconds a shard must remain in `RUNNING` state before considered a success.

1.6 Cron Jobs

You will see various commands and options relating to cron jobs in `aurora -help` and similar. Ignore them, as they're not yet implemented. You might be able to use them without causing an error, but nothing happens if you do.

1.7 Comparing Jobs

```
aurora diff <job_key> config
```

Compares a job configuration against a running job. By default the diff is determined using `diff`, though you may choose an alternate diff program by specifying the `DIFF_VIEWER` environment variable.

There are two named parameters:

- `-E NAME=VALUE` Bind a Thermos mustache variable name to a value. Multiple flags may be used to specify multiple values. Defaults to `[]`.
- `-j, --json` Read the configuration argument in JSON format. Defaults to `False`.

1.8 Viewing/Examining Jobs

Above we discussed creating, killing, and updating Jobs. Here we discuss how to view and examine Jobs.

1.8.1 Listing Jobs

```
aurora list_jobs
Usage: `aurora list_jobs cluster/role
```

Lists all Jobs registered with the Aurora scheduler in the named cluster for the named role.

It has two named parameters:

- `--pretty`: Displays job information in prettyprinted format. Defaults to `False`.
- `-c`, `--show-cron`: Shows cron schedule for jobs. Defaults to `False`. Do not use, as it's not yet implemented.

1.8.2 Inspecting a Job

```
aurora inspect <job_key> config
```

`inspect` verifies that its specified job can be parsed from a configuration file, and displays the parsed configuration. It has four named parameters:

- `--local`: Inspect the configuration that the `spawn` command would create, defaulting to `False`.
- `--raw`: Shows the raw configuration. Defaults to `False`.
- `-j`, `--json`: If specified, configuration is read in JSON format. Defaults to `False`.
- `-E NAME=VALUE`: Bind a Thermos Mustache variable name to a value. You can use multiple flags to specify multiple values. Defaults to `[]`

1.8.3 Versions

```
aurora version
```

Lists client build information and what Aurora API version it supports.

1.8.4 Checking Your Quota

```
aurora get_quota --cluster=CLUSTER role
```

Prints the production quota allocated to the role's value at the given cluster.

1.8.5 Finding a Job on Web UI

When you create a job, part of the output response contains a URL that goes to the job's scheduler UI page. For example:

```
vagrant@precise64:~$ aurora create example/www-data/prod/hello /vagrant/examples/jobs/hello_world.au
INFO] Creating job hello
INFO] Response from scheduler: OK (message: 1 new tasks pending for job www-data/prod/hello)
INFO] Job url: http://precise64:8081/scheduler/www-data/prod/hello
```

You can go to the scheduler UI page for this job via `http://precise64:8081/scheduler/www-data/prod/hello` You can go to the overall scheduler UI page by going to the part of that URL that ends at `scheduler`; `http://precise64:8081/scheduler`

Once you click through to a role page, you see Jobs arranged separately by pending jobs, active jobs and finished jobs. Jobs are arranged by role, typically a service account for production jobs and user accounts for test or development jobs.

1.8.6 Getting Job Status

```
aurora status <job_key>
```

Returns the status of recent tasks associated with the `job_key` specified Job in its supplied cluster. Typically this includes a mix of active tasks (running or assigned) and inactive tasks (successful, failed, and lost.)

1.8.7 Opening the Web UI

Use the Job's web UI scheduler URL or the `aurora status` command to find out on which machines individual tasks are scheduled. You can open the web UI via the `open` command line command if invoked from your machine:

```
aurora open [<cluster>[/<role>[/<env>/<job_name>]]]
```

If only the cluster is specified, it goes directly to that cluster's scheduler main page. If the role is specified, it goes to the top-level role page. If the full job key is specified, it goes directly to the job page where you can inspect individual tasks.

1.8.8 SSHing to a Specific Task Machine

```
aurora ssh <job_key> <shard number>
```

You can have the Aurora client ssh directly to the machine that has been assigned a particular Job/shard number. This may be useful for quickly diagnosing issues such as performance issues or abnormal behavior on a particular machine.

It can take three named parameters:

- `-e`, `--executor_sandbox`: Run `ssh` in the executor sandbox instead of the task sandbox. Defaults to `False`.
- `--user=SSH_USER`: `ssh` as the given user instead of as the role in the `job_key` argument. Defaults to none.
- `-L PORT:NAME`: Add tunnel from local port `PORT` to the remote named port `NAME`. Defaults to `[]`.

1.8.9 Templating Command Arguments

```
aurora run [-e] [-t THREADS] <job_key> -- <<command-line>>
```

Given a job specification, run the supplied command on all hosts and return the output. You may use the standard Mustache templating rules:

- {{`thermos.ports[name]`}} substitutes the specific named port of the task assigned to this machine
- {{`mesos.instance`}} substitutes the shard id of the job's task assigned to this machine
- {{`thermos.task_id`}} substitutes the task id of the job's task assigned to this machine

For example, the following type of pattern can be a powerful diagnostic tool:

```
aurora run -t5 cluster1/tyg/devel/seizure -- \
  'curl -s -m1 localhost:{{thermos.ports[http]}}/vars | grep uptime'
```

By default, the command runs in the Task's sandbox. The `-e` option can run the command in the executor's sandbox. This is mostly useful for Aurora administrators.

You can parallelize the runs by using the `-t` option.

CHAPTER 2

Aurora + Thermos Configuration Reference

CHAPTER 3

Introduction

Don't know where to start? The Aurora configuration schema is very powerful, and configurations can become quite complex for advanced use cases.

For examples of simple configurations to get something up and running quickly, check out the Tutorial. When you feel comfortable with the basics, move on to the Configuration Tutorial for more in-depth coverage of configuration design.

For additional basic configuration examples, see the end of this document.

CHAPTER 4

Process Schema

Process objects consist of required `name` and `cmdline` attributes. You can customize Process behavior with its optional attributes. Remember, Processes are handled by Thermos.

`Process` Objects

Attribute Name	Type	Description
name	String	Process name (Required)
cmdline	String	Command line (Required)
max_failures	Integer	Maximum process failures (Default: 1)
daemon	Boolean	When True, this is a daemon process. (Default: False)
ephemeral	Boolean	When True, this is an ephemeral process. (Default: False)
min_duration	Integer	Minimum duration between process restarts in seconds. (Default: 15)
final	Boolean	

When True, this process is a finalizing one that should run last. (Default: False)

4.1 name

The name is any valid UNIX filename string (specifically no slashes, NULLs or leading periods). Within a Task object, each Process name must be unique.

4.2 cmdline

The command line run by the process. The command line is invoked in a bash subshell, so can involve fully-blown bash scripts. However, nothing is supplied for command-line arguments so `$*` is unspecified.

4.3 max_failures

The maximum number of failures (non-zero exit statuses) this process can have before being marked permanently failed and not retried. If a process permanently fails, Thermos looks at the failure limit of the task containing the process (usually 1) to determine if the task has failed as well.

Setting `max_failures` to 0 makes the process retry indefinitely until it achieves a successful (zero) exit status. It retries at most once every `min_duration` seconds to prevent an effective denial of service attack on the coordinating Thermos scheduler.

4.4 daemon

By default, Thermos processes are non-daemon. If `daemon` is set to True, a successful (zero) exit status does not prevent future process runs. Instead, the process reinvokes after `min_duration` seconds. However, the maximum failure limit still applies. A combination of `daemon=True` and `max_failures=0` causes a process to retry indefinitely regardless of exit status. This should be avoided for very short-lived processes because of the accumulation of checkpointed state for each process run. When running in Mesos specifically, `max_failures` is capped at 100.

4.5 ephemeral

By default, Thermos processes are non-ephemeral. If `ephemeral` is set to True, the process' status is not used to determine if its containing task has completed. For example, consider a task with a non-ephemeral webserver process and an ephemeral logsaver process that periodically checkpoints its log files to a centralized data store. The task is considered finished once the webserver process has completed, regardless of the logsaver's current status.

4.6 min_duration

Processes may succeed or fail multiple times during a single task's duration. Each of these is called a *process run*. `min_duration` is the minimum number of seconds the scheduler waits before running the same process.

4.7 final

Processes can be grouped into two classes: ordinary processes and finalizing processes. By default, Thermos processes are ordinary. They run as long as the task is considered healthy (i.e., no failure limits have been reached.) But once all regular Thermos processes finish or the task reaches a certain failure threshold, it moves into a "finalization" stage and runs all finalizing processes. These are typically processes necessary for cleaning up the task, such as log checkpointers, or perhaps e-mail notifications that the task completed.

Finalizing processes may not depend upon ordinary processes or vice-versa, however finalizing processes may depend upon other finalizing processes and otherwise run as a typical process schedule.

CHAPTER 5

Task Schema

Tasks fundamentally consist of a `name` and a list of Process objects stored as the value of the `processes` attribute. Processes can be further constrained with `constraints`. By default, `name`'s value inherits from the first Process in the `processes` list, so for simple `Task` objects with one Process, `name` can be omitted. In Mesos, `resources` is also required.

`Task` Object

param

type

description

name

String

Process name (Required) (Default: {{processes0.name}})

processes

List of Process objects

List of Process objects bound to this task. (Required)

constraints

List of Constraint objects

List of Constraint objects constraining processes.

resources

Resource object

Resource footprint. (Required)

max_failures

Integer

Maximum process failures before being considered failed (Default: 1)

max_concurrency

Integer

Maximum number of concurrent processes (Default: 0, unlimited concurrency.)

finalization_wait

Integer

Amount of time allocated for finalizing processes, in seconds. (Default: 30)

`name`

`name` is a string denoting the name of this task. It defaults to the name of the first Process in the list of Processes associated with the `processes` attribute.

`processes`

`processes` is an unordered list of `Process` objects. To constrain the order in which they run, use `constraints`.

`constraints`

A list of `Constraint` objects. Currently it supports only one type, the `order` constraint. `order` is a list of process names that should run in the order given. For example,

```
process = Process(cmdline = "echo hello {{name}}")
task = Task(name = "echoes",
            processes = [process(name = "jim"), process(name = "bob")],
            constraints = [Constraint(order = ["jim", "bob"])])
```

Constraints can be supplied ad-hoc and in duplicate. Not all Processes need be constrained, however Tasks with cycles are rejected by the Thermos scheduler.

Use the `order` function as shorthand to generate `Constraint` lists. The following:

```
order(process1, process2)
```

is shorthand for

```
[Constraint(order = [process1.name(), process2.name()])]
```

`resources`

Takes a `Resource` object, which specifies the amounts of CPU, memory, and disk space resources to allocate to the Task.

`max_failures`

`max_failures` is the number of times processes that are part of this Task can fail before the entire Task is marked for failure.

For example:

```
template = Process(max_failures=10)
task = Task(
  name = "fail",
  processes = [
     template(name = "failing", cmdline = "exit 1"),
     template(name = "succeeding", cmdline = "exit 0")
  ],
  max_failures=2)
```

The `failing` Process could fail 10 times before being marked as permanently failed, and the `succeeding` Process would succeed on the first run. The task would succeed despite only allowing for two failed processes. To be more specific, there would be 10 failed process runs yet 1 failed process.

`max_concurrency`

For Tasks with a number of expensive but otherwise independent processes, you may want to limit the amount of concurrency the Thermos scheduler provides rather than artificially constraining it via `order` constraints. For example,

a test framework may generate a task with 100 test run processes, but wants to run it on a machine with only 4 cores. You can limit the amount of parallelism to 4 by setting `max_concurrency=4` in your task configuration.

For example, the following task spawns 180 Processes ("mappers") to compute individual elements of a 180 degree sine table, all dependent upon one final Process ("reducer") to tabulate the results:

```
def make_mapper(id):
  return Process(
    name = "mapper%03d" % id,
    cmdline = "echo 'scale=50;s(%d\*4\*a(1)/180)' | bc -l >
               temp.sine_table.%03d" % (id, id))

def make_reducer():
  return Process(name = "reducer", cmdline = "cat temp.\* | nl \> sine\_table.txt
                 && rm -f temp.\*")

processes = map(make_mapper, range(180))

task = Task(
  name = "mapreduce",
  processes = processes + [make\_reducer()],
  constraints = [Constraint(order = [mapper.name(), 'reducer']) for mapper
                 in processes],
  max_concurrency = 8)
```

`finalization_wait`

Tasks have three active stages: `ACTIVE`, `CLEANING`, and `FINALIZING`. The `ACTIVE` stage is when ordinary processes run. This stage lasts as long as Processes are running and the Task is healthy. The moment either all Processes have finished successfully or the Task has reached a maximum Process failure limit, it goes into `CLEANING` stage and send SIGTERMs to all currently running Processes and their process trees. Once all Processes have terminated, the Task goes into `FINALIZING` stage and invokes the schedule of all Processes with the "final" attribute set to True.

This whole process from the end of `ACTIVE` stage to the end of `FINALIZING` must happen within `finalization_wait` seconds. If it does not finish during that time, all remaining Processes are sent SIGKILLs (or if they depend upon uncompleted Processes, are never invoked.)

Client applications with higher priority may force a shorter finalization wait (e.g. through parameters to `thermos kill`), so this is mostly a best-effort signal.

`Constraint` Object

Current constraint objects only support a single ordering constraint, `order`, which specifies its processes run sequentially in the order given. By default, all processes run in parallel when bound to a `Task` without ordering constraints.

param

type

description

order

List of String

List of processes by name (String) that should be run serially.

Specifies the amount of CPU, Ram, and disk resources the task needs. See the Resource Isolation document for suggested values and to understand how resources are allocated.

---

param

type

description

cpu

Float

Fractional number of cores required by the task.

ram

Integer

Bytes of RAM required by the task.

disk

Integer

Bytes of disk required by the task.

CHAPTER 6

Job Schema

`Job` Objects

task

Task

The Task object to bind to this job. Required.

name

String

Job name. (Default: inherited from the task attribute's name)

role

String

Job role account. Required.

cluster

String

Cluster in which this job is scheduled. Required.

environment

String

Job environment, default devel. Must be one of prod, devel, test or staging<number>.

contact

String

Best email address to reach the owner of the job. For production jobs, this is usually a team mailing list.

instances

Integer

Number of instances (sometimes referred to as replicas or shards) of the task to create. (Default: 1)

cron_schedule (Present, but not supported and a no-op)

String

UTC Cron schedule in cron format. May only be used with non-service jobs. Default: None (not a cron job.)

cron_collision_policy (Present, but not supported and a no-op)

String

Policy to use when a cron job is triggered while a previous run is still active. KILL_EXISTING Kill the previous run, and schedule the new run CANCEL_NEW Let the previous run continue, and cancel the new run. RUN_OVERLAP Let the previous run continue, and schedule the new run. (Default: KILL_EXISTING)

update_config

update_config object

Parameters for controlling the rate and policy of rolling updates.

constraints

dict

Scheduling constraints for the tasks. See the section on the constraint specification language

service

Boolean

If True, restart tasks regardless of success or failure. (Default: False)

daemon

Boolean

A DEPRECATED alias for "service". (Default: False)

max_task_failures

Integer

Maximum number of failures after which the task is considered to have failed (Default: 1) Set to -1 to allow for infinite failures

priority

Integer

Preemption priority to give the task (Default 0). Tasks with higher priorities may preempt tasks at lower priorities.

production

Boolean

Whether or not this is a production task backed by quota (Default: False) Production jobs may preempt any non-production job, and may only be preempted by production jobs in the same role and of higher priority. To run jobs at this level, the job role must have the appropriate quota.

health_check_config

heath_check_config object

Parameters for controlling a task's health checks via HTTP. Only used if a health port was assigned with a command line wildcard.

Jobs with the `service` flag set to True are called Services. The `Service` alias can be used as shorthand for `Job` with `service=True`. Services are differentiated from non-service Jobs in that tasks always restart on completion, whether successful or unsuccessful. Jobs without the service bit set only restart up to `max_task_failures` times and only if they terminated unsuccessfully either due to human error or machine failure.

`UpdateConfig` Objects

Parameters for controlling the rate and policy of rolling updates.

batch_size

Integer

Maximum number of shards to be updated in one iteration (Default: 1)

restart_threshold

Integer

Maximum number of seconds before a shard must move into the RUNNING state before considered a failure (Default: 60)

watch_secs

Integer

Minimum number of seconds a shard must remain in RUNNING state before considered a success (Default: 30)

max_per_shard_failures

Integer

Maximum number of restarts per shard during update. Increments total failure count when this limit is exceeded. (Default: 0)

max_total_failures

Integer

Maximum number of shard failures to be tolerated in total during an update. Cannot be greater than or equal to the total number of tasks in a job. (Default: 0)

Parameters for controlling a task's health checks via HTTP.

initial_interval_secs

Integer

Initial delay for performing an HTTP health check. (Default: 60)

interval_secs

Integer

Interval on which to check the task's health via HTTP. (Default: 30)

timeout_secs

Integer

HTTP request timeout. (Default: 1)

max_consecutive_failures

Integer

Maximum number of consecutive failures that tolerated before considering a task unhealthy (Default: 0)

CHAPTER 7

Specifying Scheduling Constraints

Most users will not need to specify constraints explicitly, as the scheduler automatically inserts reasonable defaults that attempt to ensure reliability without impacting schedulability. For example, the scheduler inserts a `host:limit:1` constraint, ensuring that your shards run on different physical machines. Please do not set this field unless you are sure of what you are doing.

In the `Job` object there is a map `constraints` from String to String allowing the user to tailor the schedulability of tasks within the job.

Each slave in the cluster is assigned a set of string-valued key/value pairs called attributes. For example, consider the host `cluster1-aaa-03-sr2` and its following attributes (given in key:value format): `host:cluster1-aaa-03-sr2` and `rack:aaa`.

The constraint map's key value is the attribute name in which we constrain Tasks within our Job. The value is how we constrain them. There are two types of constraints: *limit constraints* and *value constraints*.

Limit Constraint

A string that specifies a limit for a constraint. Starts with 'limit: followed by an Integer and closing single quote, such as 'limit:1'.

Value Constraint

A string that specifies a value for a constraint. To include a list of values, separate the values using commas. To negate the values of a constraint, start with a !.

You can also control machine diversity using constraints. The below constraint ensures that no more than two instances of your job may run on a single host. Think of this as a "group by" limit.

```
constraints = {
  'host': 'limit:2',
}
```

Likewise, you can use constraints to control rack diversity, e.g. at most one task per rack:

```
constraints = {
  'rack': 'limit:1',
}
```

Use these constraints sparingly as they can dramatically reduce Tasks' schedulability.

CHAPTER 8

Template Namespaces

Currently, a few Pystachio namespaces have special semantics. Using them in your configuration allow you to tailor application behavior through environment introspection or interact in special ways with the Aurora client or Aurora-provided services.

`mesos` Namespace

The `mesos` namespace contains the `instance` variable that can be used to distinguish between Task replicas.

instance

Integer

The instance number of the created task. A job with 5 replicas has instance numbers 0, 1, 2, 3, and 4.

The `thermos` namespace contains variables that work directly on the Thermos platform in addition to Aurora. This namespace is fully compatible with Tasks invoked via the `thermos` CLI.

ports

map of string to Integer

A map of names to port numbers

task_id

string

The task ID assigned to this task.

The `thermos.ports` namespace is automatically populated by Aurora when invoking tasks on Mesos. When running the `thermos` command directly, these ports must be explicitly mapped with the `-P` option.

For example, if '`{{thermos.ports[http]}}`' is specified in a `Process` configuration, it is automatically extracted and auto-populated by Aurora, but must be specified with, for example, `thermos -P http:12345` to map `http` to port 12345 when running via the CLI.

CHAPTER 9

Basic Examples

These are provided to give a basic understanding of simple Aurora jobs.

```
    hello_world.aurora
```

Put the following in a file named `hello_world.aurora`, substituting your own values for values such as `cluster`'s.

```
import os
hello_world_process = Process(name = 'hello_world', cmdline = 'echo hello world')

hello_world_task = Task(
  resources = Resources(cpu = 0.1, ram = 16 * MB, disk = 16 * MB),
  processes = [hello_world_process])

hello_world_job = Job(
  cluster = 'cluster1',
  role = os.getenv('USER'),
  task = hello_world_task)

jobs = [hello_world_job]
```

Then issue the following commands to create and kill the job, using your own values for the job key.

```
       `aurora create cluster1/$USER/test/hello_world hello_world.aurora`

       `aurora kill cluster1/$USER/test/hello_world `

Environment Tailoring
```

Put the following in a file named `hello_world_productionized.aurora`, substituting your own values for values such as `cluster`'s.

```
include('hello_world.aurora')

production_resources = Resources(cpu = 1.0, ram = 512 * MB, disk = 2 * GB)
staging_resources = Resources(cpu = 0.1, ram = 32 * MB, disk = 512 * MB)
hello_world_template = hello_world(
    name = "hello_world-{{cluster}}"
    task = hello_world(resources=production_resources))

jobs = [
```

```
  # production jobs
  hello_world_template(cluster = 'cluster1', instances = 25),
  hello_world_template(cluster = 'cluster2', instances = 15),

  # staging jobs
  hello_world_template(
    cluster = 'local',
    instances = 1,
    task = hello_world(resources=staging_resources)),
]
```

Then issue the following commands to create and kill the job, using your own values for the job key

```
aurora create cluster1/$USER/test/hello_world-cluster1 hello_world_productionized.aurora

aurora kill cluster1/$USER/test/hello_world-cluster1
```

CHAPTER 10

Aurora Configuration Tutorial

How to write Aurora configuration files, including feature descriptions and best practices. When writing a configuration file, make use of `aurora inspect`. It takes the same job key and configuration file arguments as `aurora create` or `aurora update`. It first ensures the configuration parses, then outputs it in human-readable form.

You should read this after going through the general Aurora Tutorial.

10.1 The Basics

To run a job on Aurora, you must specify a configuration file that tells Aurora what it needs to know to schedule the job, what Mesos needs to run the tasks the job is made up of, and what Thermos needs to run the processes that make up the tasks. This file must have a `.aurora` suffix.

A configuration file defines a collection of objects, along with parameter values for their attributes. An Aurora configuration file contains the following three types of objects:

- `Job`
- `Task`
- `Process`

A configuration also specifies a list of `Job` objects assigned to the variable `jobs`.

- `jobs` (list of defined Jobs to run)

The `.aurora` file format is just Python. However, `Job`, `Task`, `Process`, and other classes are defined by a type-checked dictionary templating library called *Pystachio*, a powerful tool for configuration specification and reuse. Pystachio objects are tailored via {{}} surrounded templates.

When writing your `.aurora` file, you may use any Pystachio datatypes, as well as any objects shown in the *Aurora+Thermos Configuration Reference*, without `import` statements - the Aurora config loader injects them automatically. Other than that, an `.aurora` file works like any other Python script.

Aurora+Thermos Configuration Reference has a full reference of all Aurora/Thermos defined Pystachio objects.

Use Bottom-To-Top Object Ordering

A well-structured configuration starts with structural templates (if any). Structural templates encapsulate in their attributes all the differences between Jobs in the configuration that are not directly manipulated at the `Job` level, but typically at the `Process` or `Task` level. For example, if certain processes are invoked with slightly different settings or input.

After structural templates, define, in order, `Process`es, `Task`s, and `Job`s.

Structural template names should be *UpperCamelCased* and their instantiations are typically *UPPER_SNAKE_CASED*. `Process`, `Task`, and `Job` names are typically *lower_snake_cased*. Indentation is typically 2 spaces.

10.2 An Example Configuration File

The following is a typical configuration file. Don't worry if there are parts you don't understand yet, but you may want to refer back to this as you read about its individual parts. Note that names surrounded by curly braces {{}} are template variables, which the system replaces with bound values for the variables.

```
# --- templates here ---
class Profile(Struct):
  package_version =      Default(String, 'live')
  java_binary =          Default(String,
                           '/usr/lib/jvm/java-1.7.0-openjdk/bin/java')
  extra_jvm_options =   Default(String, '')
  parent_environment =   Default(String, 'prod')
  parent_serverset =     Default(String,
                           `/foocorp/service/bird/{{parent_environment}}/bird')

# --- processes here ---
main = Process(
  name = 'application',
  cmdline = '{{profile.java_binary}} -server -Xmx1792m '
            '{{profile.extra_jvm_options}} '
            '-jar application.jar '
            '-upstreamService {{profile.parent_serverset}}'
)

# --- tasks ---
base_task = SequentialTask(
  name = 'application',
  processes = [
    Process(
      name = 'fetch',variablesvv
      cmdline = 'curl -O
              https://packages.foocorp.com/{{profile.package_version}}/application.jar'),
  ]
)

    # not always necessary but often useful to have separate task
    # resource classes
    staging_task = base_task(resources =
                    Resources(cpu = 1.0,
                              ram = 2048*MB,
                              disk = 1*GB))
```

```
  production_task = base_task(resources =
                          Resources(cpu = 4.0,
                                    ram = 2560*MB,
                                    disk = 10*GB))

  # --- job template ---
  job_template = Job(
    name = 'application',
    role = 'myteam',
    contact = 'myteam-team@foocorp.com',
    instances = 20,
    service = True,
    task = production_task
  )

  # -- profile instantiations (if any) ---
  PRODUCTION = Profile()
  STAGING = Profile(
    extra_jvm_options = '-Xloggc:gc.log',
    parent_environment = 'staging'
  )

  # -- job instantiations --
  jobs = [
        job_template(cluster = 'cluster1', environment = 'prod')
                .bind(profile = PRODUCTION),

        job_template(cluster = 'cluster2', environment = 'prod')
                 .bind(profile = PRODUCTION),

        job_template(cluster = 'cluster1',
                     environment = 'staging',
          service = False,
          task = staging_task,
          instances = 2)
          .bind(profile = STAGING),
  ]
```

Defining Process Objects

Processes are handled by the Thermos system. A process is a single executable step run as a part of an Aurora task, which consists of a bash-executable statement.

The key (and required) `Process` attributes are:

- `name`: Any string which is a valid Unix filename (no slashes, NULLs, or leading periods). The `name` value must be unique relative to other Processes in a `Task`.
- `cmdline`: A command line run in a bash subshell, so you can use bash scripts. Nothing is supplied for command-line arguments, so `$*` is unspecified.

Many tiny processes make managing configurations more difficult. For example, the following is a bad way to define processes.

```
copy = Process(
  name = 'copy',
  cmdline = 'curl -O https://packages.foocorp.com/app.zip'
)
```

```
unpack = Process(
  name = 'unpack',
  cmdline = 'unzip app.zip'
)
remove = Process(
  name = 'remove',
  cmdline = 'rm -f app.zip'
)
run = Process(
  name = 'app',
  cmdline = 'java -jar app.jar'
)
run_task = Task(
  processes = [copy, unpack, remove, run],
  constraints = order(copy, unpack, remove, run)
)
```

Since `cmdline` runs in a bash subshell, you can chain commands with `&&` or `||`.

When defining a `Task` that is just a list of Processes run in a particular order, use `SequentialTask`, as described in the **Defining* ‘‘Task‘ Objects <#Task>‘__ section. The following simplifies and combines the above multiple `Process` definitions into just two.

```
stage = Process(
  name = 'stage',
  cmdline = 'curl -O https://packages.foocorp.com/app.zip && '
            'unzip app.zip && rm -f app.zip')

run = Process(name = 'app', cmdline = 'java -jar app.jar')

run_task = SequentialTask(processes = [stage, run])
```

`Process` also has five optional attributes, each with a default value if one isn't specified in the configuration:

- `max_failures`: Defaulting to `1`, the maximum number of failures (non-zero exit statuses) before this `Process` is marked permanently failed and not retried. If a `Process` permanently fails, Thermos checks the `Process` object's containing `Task` for the task's failure limit (usually 1) to determine whether or not the `Task` should be failed. Setting `max_failures`to `0` means that this process will keep retrying until a successful (zero) exit status is achieved. Retries happen at most once every `min_duration` seconds to prevent effectively mounting a denial of service attack against the coordinating scheduler.
- `daemon`: Defaulting to `False`, if `daemon` is set to `True`, a successful (zero) exit status does not prevent future process runs. Instead, the `Process` reinvokes after `min_duration` seconds. However, the maximum failure limit (`max_failures`) still applies. A combination of `daemon=True` and `max_failures=0` retries a `Process` indefinitely regardless of exit status. This should generally be avoided for very short-lived processes because of the accumulation of checkpointed state for each process run. When running in Aurora, `max_failures` is capped at 100.
- `ephemeral`: Defaulting to `False`, if `ephemeral` is `True`, the `Process`‘ status is not used to determine if its bound `Task` has completed. For example, consider a `Task` with a non-ephemeral webserver process and an ephemeral logsaver process that periodically checkpoints its log files to a centralized data store. The `Task` is considered finished once the webserver process finishes, regardless of the logsaver's current status.
- `min_duration`: Defaults to `15`. Processes may succeed or fail multiple times during a single Task. Each result is called a *process run* and this value is the minimum number of seconds the scheduler waits before re-running the same process.
- `final`: Defaulting to `False`, this is a finalizing `Process` that should run last. Processes can be grouped into two classes: *ordinary* and *finalizing*. By default, Thermos Processes are ordinary. They run as long as the `Task`

is considered healthy (i.e. hasn't reached a failure limit). But once all regular Thermos Processes have either finished or the `Task` has reached a certain failure threshold, Thermos moves into a *finalization* stage and runs all finalizing Processes. These are typically necessary for cleaning up after the `Task`, such as log checkpointers, or perhaps e-mail notifications of a completed Task. Finalizing processes may not depend upon ordinary processes or vice-versa, however finalizing processes may depend upon other finalizing processes and will otherwise run as a typical process schedule.

10.3 Getting Your Code Into The Sandbox

When using Aurora, you need to get your executable code into its "sandbox", specifically the Task sandbox where the code executes for the Processes that make up that Task.

Each Task has a sandbox created when the Task starts and garbage collected when it finishes. All of a Task's processes run in its sandbox, so processes can share state by using a shared current working directory.

Typically, you save this code somewhere. You then need to define a Process in your `.aurora` configuration file that fetches the code from that somewhere to where the slave can see it. For a public cloud, that can be anywhere public on the Internet, such as S3. For a private cloud internal storage, you need to put in on an accessible HDFS cluster or similar storage.

The template for this Process is:

```
<name> = Process(
  name = '<name>'
  cmdline = '<command to copy and extract code archive into current working directory>'
)
```

Note: Be sure the extracted code archive has an executable.

Defining Task Objects

Tasks are handled by Mesos. A task is a collection of processes that runs in a shared sandbox. It's the fundamental unit Aurora uses to schedule the datacenter; essentially what Aurora does is find places in the cluster to run tasks.

The key (and required) parts of a Task are:

- `name`: A string giving the Task's name. By default, if a Task is not given a name, it inherits the first name in its Process list.
- `processes`: An unordered list of Process objects bound to the Task. The value of the optional `constraints` attribute affects the contents as a whole. Currently, the only constraint, `order`, determines if the processes run in parallel or sequentially.
- `resources`: A `Resource` object defining the Task's resource footprint. A `Resource` object has three attributes: - `cpu`: A Float, the fractional number of cores the Task requires. - `ram`: An Integer, RAM bytes the Task requires. - `disk`: An integer, disk bytes the Task requires.

A basic Task definition looks like:

```
Task(
    name="hello_world",
    processes=[Process(name = "hello_world", cmdline = "echo hello world")],
    resources=Resources(cpu = 1.0,
                        ram = 1*GB,
                        disk = 1*GB))
```

There are four optional Task attributes:

- `constraints`: A list of `Constraint` objects that constrain the Task's processes. Currently there is only one type, the `order` constraint. For example the following requires that the processes run in the order `foo`, then `bar`.

  ```
  constraints = [Constraint(order=['foo', 'bar'])]
  ```

  There is an `order()` function that takes `order('foo', 'bar', 'baz')` and converts it into `[Constraint(order=['foo', 'bar', 'baz'])]`. `order()` accepts Process name strings `('foo', 'bar')` or the processes themselves, e.g. `foo=Process(name='foo', ...)`, `bar=Process(name='bar', ...)`, `constraints=order(foo, bar)`

  Note that Thermos rejects tasks with process cycles.

- `max_failures`: Defaulting to `1`, the number of failed processes needed for the `Task` to be marked as failed. Note how this interacts with individual Processes' `max_failures` values. Assume a Task has two Processes and a `max_failures` value of `2`. So both Processes must fail for the Task to fail. Now, assume each of the Task's Processes has its own `max_failures` value of `10`. If Process "A" fails 5 times before succeeding, and Process "B" fails 10 times and is then marked as failing, their parent Task succeeds. Even though there were 15 individual failures by its Processes, only 1 of its Processes was finally marked as failing. Since 1 is less than the 2 that is the Task's `max_failures` value, the Task does not fail.

- `max_concurrency`: Defaulting to `0`, the maximum number of concurrent processes in the Task. `0` specifies unlimited concurrency. For Tasks with many expensive but otherwise independent processes, you can limit the amount of concurrency Thermos schedules instead of artificially constraining them through `order` constraints. For example, a test framework may generate a Task with 100 test run processes, but runs it in a Task with `resources.cpus=4`. Limit the amount of parallelism to 4 by setting `max_concurrency=4`.

- `finalization_wait`: Defaulting to `30`, the number of seconds allocated for finalizing the Task's processes. A Task starts in `ACTIVE` state when Processes run and stays there as long as the Task is healthy and Processes run. When all Processes finish successfully or the Task reaches its maximum process failure limit, it goes into `CLEANING` state. In `CLEANING`, it sends `SIGTERMS` to any still running Processes. When all Processes terminate, the Task goes into `FINALIZING` state and invokes the schedule of all processes whose final attribute has a True value. Everything from the end of `ACTIVE` to the end of `FINALIZING` must happen within `finalization_wait` number of seconds. If not, all still running Processes are sent `SIGKILL`s (or if dependent on yet to be completed Processes, are never invoked).

  `SequentialTask`: Running Processes in Parallel or Sequentially

By default, a Task with several Processes runs them in parallel. There are two ways to run Processes sequentially:

- Include an `order` constraint in the Task definition's `constraints` attribute whose arguments specify the processes' run order:

  ```
  Task( ... processes=[process1, process2, process3],
        constraints = order(process1, process2, process3), ...)
  ```

- Use `SequentialTask` instead of `Task`; it automatically runs processes in the order specified in the `processes` attribute. No `constraint` parameter is needed:

  ```
  SequentialTask( ... processes=[process1, process2, process3] ...)
  ```

  `SimpleTask`

For quickly creating simple tasks, use the `SimpleTask` helper. It creates a basic task from a provided name and command line using a default set of resources. For example, in a `.aurora` configuration file:

```
SimpleTask(name="hello_world", command="echo hello world")
```

is equivalent to

```
Task(name="hello_world",
     processes=[Process(name = "hello_world", cmdline = "echo hello world")],
     resources=Resources(cpu = 1.0,
                         ram = 1*GB,
                         disk = 1*GB))
```

The simplest idiomatic Job configuration thus becomes:

```
import os
hello_world_job = Job(
  task=SimpleTask(name="hello_world", command="echo hello world"),
  role=os.getenv('USER'),
  cluster="cluster1")
```

When written to `hello_world.aurora`, you invoke it with a simple `aurora create cluster1/$USER/test/hello_world hello_world.aurora`.

`Tasks.concat` and `Tasks.combine` (`concat_tasks` and `combine_tasks`)

`Tasks.concat`(synonym,‘‘concat_tasks‘‘) and `Tasks.combine`(synonym,‘‘combine_tasks‘‘) merge multiple Task definitions into a single Task. It may be easier to define complex Jobs as smaller constituent Tasks. But since a Job only includes a single Task, the subtasks must be combined before using them in a Job. Smaller Tasks can also be reused between Jobs, instead of having to repeat their definition for multiple Jobs.

With both methods, the merged Task takes the first Task’s name. The difference between the two is the result Task’s process ordering.

- `Tasks.combine` runs its subtasks’ processes in no particular order. The new Task’s resource consumption is the sum of all its subtasks’ consumption.
- `Tasks.concat` runs its subtasks in the order supplied, with each subtask’s processes run serially between tasks. It is analogous to the `order` constraint helper, except at the Task level instead of the Process level. The new Task’s resource consumption is the maximum value specified by any subtask for each Resource attribute (cpu, ram and disk).

For example, given the following:

```
setup_task = Task(
  ...
  processes=[download_interpreter, update_zookeeper],
  # It is important to note that {{Tasks.concat}} has
  # no effect on the ordering of the processes within a task;
  # hence the necessity of the {{order}} statement below
  # (otherwise, the order in which {{download_interpreter}}
  # and {{update_zookeeper}} run will be non-deterministic)
  constraints=order(download_interpreter, update_zookeeper),
  ...
)

run_task = SequentialTask(
  ...
  processes=[download_application, start_application],
  ...
)
```

```
combined_task = Tasks.concat(setup_task, run_task)
```

The `Tasks.concat` command merges the two Tasks into a single Task and ensures all processes in `setup_task` run before the processes in `run_task`. Conceptually, the task is reduced to:

```
task = Task(
  ...
  processes=[download_interpreter, update_zookeeper,
             download_application, start_application],
  constraints=order(download_interpreter, update_zookeeper,
                    download_application, start_application),
  ...
)
```

In the case of `Tasks.combine`, the two schedules run in parallel:

```
task = Task(
  ...
  processes=[download_interpreter, update_zookeeper,
             download_application, start_application],
  constraints=order(download_interpreter, update_zookeeper) +
                    order(download_application, start_application),
  ...
)
```

In the latter case, each of the two sequences may operate in parallel. Of course, this may not be the intended behavior (for example, if the `start_application` Process implicitly relies upon `download_interpreter`). Make sure you understand the difference between using one or the other.

Defining `Job` Objects

A job is a group of identical tasks that Aurora can run in a Mesos cluster.

A `Job` object is defined by the values of several attributes, some required and some optional. The required attributes are:

- `task`: Task object to bind to this job. Note that a Job can only take a single Task.
- `role`: Job's role account; in other words, the user account to run the job as on a Mesos cluster machine. A common value is `os.getenv('USER')`; using a Python command to get the user who submits the job request. The other common value is the service account that runs the job, e.g. `www-data`.
- `environment`: Job's environment, typical values are `devel`, `test`, or `prod`.
- `cluster`: Aurora cluster to schedule the job in, defined in `/etc/aurora/clusters.json` or `~/.clusters.json`. You can specify jobs where the only difference is the `cluster`, then at run time only run the Job whose job key includes your desired cluster's name.

You usually see a `name` parameter. By default, `name` inherits its value from the Job's associated Task object, but you can override this default. For these four parameters, a Job definition might look like:

```
foo_job = Job( name = 'foo', cluster = 'cluster1',
          role = os.getenv('USER'), environment = 'prod',
          task = foo_task)
```

In addition to the required attributes, there are several optional attributes. The first (strongly recommended) optional attribute is:

- `contact`: An email address for the Job's owner. For production jobs, it is usually a team mailing list.

Two more attributes deal with how to handle failure of the Job's Task:

- `max_task_failures`: An integer, defaulting to `1`, of the maximum number of Task failures after which the Job is considered failed. `-1` allows for infinite failures.
- `service`: A boolean, defaulting to `False`, which if `True` restarts tasks regardless of whether they succeeded or failed. In other words, if `True`, after the Job's Task completes, it automatically starts again. This is for Jobs you want to run continuously, rather than doing a single run.

Three attributes deal with configuring the Job's Task:

- `instances`: Defaulting to `1`, the number of instances/replicas/shards of the Job's Task to create.
- `priority`: Defaulting to `0`, the Job's Task's preemption priority, for which higher values may preempt Tasks from Jobs with lower values.
- `production`: a Boolean, defaulting to `False`, specifying that this is a production job backed by quota. Tasks from production Jobs may preempt tasks from any non-production job, and may only be preempted by tasks from production jobs in the same role with higher priority. **WARNING**: To run Jobs at this level, the Job role must have the appropriate quota.

The final three Job attributes each take an object as their value.

- `update_config`: An `UpdateConfig` object provides parameters for controlling the rate and policy of rolling updates. The `UpdateConfig` parameters are:
  - `batch_size`: An integer, defaulting to `1`, specifying the maximum number of shards to update in one iteration.
  - `restart_threshold`: An integer, defaulting to `60`, specifying the maximum number of seconds before a shard must move into the `RUNNING` state before considered a failure.
  - `watch_secs`: An integer, defaulting to `30`, specifying the minimum number of seconds a shard must remain in the `RUNNING` state before considered a success.
  - `max_per_shard_failures`: An integer, defaulting to `0`, specifying the maximum number of restarts per shard during an update. When the limit is exceeded, it increments the total failure count.
  - `max_total_failures`: An integer, defaulting to `0`, specifying the maximum number of shard failures tolerated during an update. Cannot be equal to or greater than the job's total number of tasks.
- `health_check_config`: A `HealthCheckConfig` object that provides parameters for controlling a Task's health checks via HTTP. Only used if a health port was assigned with a command line wildcard. The `HealthCheckConfig` parameters are:
  - `initial_interval_secs`: An integer, defaulting to `60`, specifying the initial delay for doing an HTTP health check.
  - `interval_secs`: An integer, defaulting to `30`, specifying the number of seconds in the interval between checking the Task's health.
  - `timeout_secs`: An integer, defaulting to `1`, specifying the number of seconds the application must respond to an HTTP health check with `OK` before it is considered a failure.
  - `max_consecutive_failures`: An integer, defaulting to `0`, specifying the maximum number of consecutive failures before a task is unhealthy.
- `constraints`: A `dict` Python object, specifying Task scheduling constraints. Most users will not need to specify constraints, as the scheduler automatically inserts reasonable defaults. Please do not set this field unless you are sure of what you are doing. See the section in the Aurora + Thermos Reference manual on Specifying Scheduling Constraints for more information.

  Defining The `jobs` List

At the end of your `.aurora` file, you need to specify a list of the file's defined Jobs to run in the order listed. For example, the following runs first `job1`, then `job2`, then `job3`.

jobs = [job1, job2, job3]

10.4 Templating

The `.aurora` file format is just Python. However, `Job`, `Task`, `Process`, and other classes are defined by a templating library called *Pystachio*, a powerful tool for configuration specification and reuse.

Aurora+Thermos Configuration Reference has a full reference of all Aurora/Thermos defined Pystachio objects.

When writing your `.aurora` file, you may use any Pystachio datatypes, as well as any objects shown in the *Aurora+Thermos Configuration Reference* without `import` statements - the Aurora config loader injects them automatically. Other than that the `.aurora` format works like any other Python script.

10.4.1 Templating 1: Binding in Pystachio

Pystachio uses the visually distinctive {{}} to indicate template variables. These are often called "mustache variables" after the similarly appearing variables in the Mustache templating system and because the curly braces resemble mustaches.

If you are familiar with the Mustache system, templates in Pystachio have significant differences. They have no nesting, joining, or inheritance semantics. On the other hand, when evaluated, templates are evaluated iteratively, so this affords some level of indirection.

Let's start with the simplest template; text with one variable, in this case `name`;

```
Hello {{name}}
```

If we evaluate this as is, we'd get back:

```
Hello
```

If a template variable doesn't have a value, when evaluated it's replaced with nothing. If we add a binding to give it a value:

```
{ "name" : "Tom" }
```

We'd get back:

```
Hello Tom
```

We can also use {{}} variables as sectional variables. Let's say we have:

```
{{#x}} Testing... {{/x}}
```

If `x` evaluates to `True`, the text between the sectional tags is shown. If there is no value for `x` or it evaluates to `False`, the between tags text is not shown. So, at a basic level, a sectional variable acts as a conditional.

However, if the sectional variable evaluates to a list, array, etc. it acts as a `foreach`. For example,

```
{{#x}} {{name}} {{/x}}
```

with

```
{ "x": [ { "name" : "tic" } { "name" : "tac" } { "name" : "toe" } ] }
```

evaluates to

```
tic tac toe
```

Every Pystachio object has an associated `.bind` method that can bind values to {{}} variables. Bindings are not immediately evaluated. Instead, they are evaluated only when the interpolated value of the object is necessary, e.g. for performing equality or serializing a message over the wire.

Objects with and without mustache templated variables behave differently:

```
>>> Float(1.5)
Float(1.5)

>>> Float('{{x}}.5')
Float({{x}}.5)

>>> Float('{{x}}.5').bind(x = 1)
Float(1.5)

>>> Float('{{x}}.5').bind(x = 1) == Float(1.5)
True

>>> contextual_object = String('{{metavar{{number}}}}').bind(
... metavar1 = "first", metavar2 = "second")

>>> contextual_object
String({{metavar{{number}}}})

>>> contextual_object.bind(number = 1)
String(first)

>>> contextual_object.bind(number = 2)
String(second)
```

You usually bind simple key to value pairs, but you can also bind three other objects: lists, dictionaries, and structurals. These will be described in detail later.

Structurals in Pystachio / Aurora

Most Aurora/Thermos users don't ever (knowingly) interact with `String`, `Float`, or `Integer` Pystashio objects directly. Instead they interact with derived structural (`Struct`) objects that are collections of fundamental and structural objects. The structural object components are called *attributes*. Aurora's most used structural objects are `Job`, `Task`, and `Process`:

```
class Process(Struct):
  cmdline = Required(String)
  name = Required(String)
  max_failures = Default(Integer, 1)
  daemon = Default(Boolean, False)
  ephemeral = Default(Boolean, False)
  min_duration = Default(Integer, 5)
  final = Default(Boolean, False)
```

Construct default objects by following the object's type with (). If you want an attribute to have a value different from its default, include the attribute name and value inside the parentheses.

```
>>> Process()
Process(daemon=False, max_failures=1, ephemeral=False,
  min_duration=5, final=False)
```

Attribute values can be template variables, which then receive specific values when creating the object.

```
>>> Process(cmdline = 'echo {{message}}')
Process(daemon=False, max_failures=1, ephemeral=False, min_duration=5,
        cmdline=echo {{message}}, final=False)

>>> Process(cmdline = 'echo {{message}}').bind(message = 'hello world')
Process(daemon=False, max_failures=1, ephemeral=False, min_duration=5,
        cmdline=echo hello world, final=False)
```

A powerful binding property is that all of an object's children inherit its bindings:

```
>>> List(Process)([
... Process(name = '{{prefix}}_one'),
... Process(name = '{{prefix}}_two')
... ]).bind(prefix = 'hello')
ProcessList(
  Process(daemon=False, name=hello_one, max_failures=1, ephemeral=False, min_duration=5, final=False)
  Process(daemon=False, name=hello_two, max_failures=1, ephemeral=False, min_duration=5, final=False)
  )
```

Remember that an Aurora Job contains Tasks which contain Processes. A Job level binding is inherited by its Tasks and all their Processes. Similarly a Task level binding is available to that Task and its Processes but is *not* visible at the Job level (inheritance is a one-way street.)

Mustaches Within Structurals

When you define a `Struct` schema, one powerful, but confusing, feature is that all of that structure's attributes are Mustache variables within the enclosing scope *once they have been populated*.

For example, when `Process` is defined above, all its attributes such as {{`name`}}, {{`cmdline`}}, {{`max_failures`}} etc., are all immediately defined as Mustache variables, implicitly bound into the `Process`, and inherit all child objects once they are defined.

Thus, you can do the following:

```
>>> Process(name = "installer", cmdline = "echo {{name}} is running")
Process(daemon=False, name=installer, max_failures=1, ephemeral=False, min_duration=5,
        cmdline=echo installer is running, final=False)
```

WARNING: This binding only takes place in one direction. For example, the following does NOT work and does not set the `Process` `name` attribute's value.

```
>>> Process().bind(name = "installer")
Process(daemon=False, max_failures=1, ephemeral=False, min_duration=5, final=False)
```

The following is also not possible and results in an infinite loop that attempts to resolve `Process.name`.

```
>>> Process(name = '{{name}}').bind(name = 'installer')
```

Do not confuse Structural attributes with bound Mustache variables. Attributes are implicitly converted to Mustache variables but not vice versa.

10.4.2 Templating 2: Structurals Are Factories

A Second Way of Templating

A second templating method is both as powerful as the aforementioned and often confused with it. This method is due to automatic conversion of Struct attributes to Mustache variables as described above.

Suppose you create a Process object:

```
>>> p = Process(name = "process_one", cmdline = "echo hello world")

>>> p
Process(daemon=False, name=process_one, max_failures=1, ephemeral=False, min_duration=5,
        cmdline=echo hello world, final=False)
```

This `Process` object, "p", can be used wherever a `Process` object is needed. It can also be reused by changing the value(s) of its attribute(s). Here we change its `name` attribute from `process_one` to `process_two`.

```
>>> p(name = "process_two")
Process(daemon=False, name=process_two, max_failures=1, ephemeral=False, min_duration=5,
        cmdline=echo hello world, final=False)
```

Template creation is a common use for this technique:

```
   >>> Daemon = Process(daemon = True)
   >>> logrotate = Daemon(name = 'logrotate', cmdline = './logrotate conf/logrotate.conf')
   >>> mysql = Daemon(name = 'mysql', cmdline = 'bin/mysqld --safe-mode')

Advanced Binding
```

As described above, `.bind()` binds simple strings or numbers to Mustache variables. In addition to Structural types formed by combining atomic types, Pystachio has two container types; `List` and `Map` which can also be bound via `.bind()`.

Bind Syntax

The `bind()` function can take Python dictionaries or `kwargs` interchangeably (when "`kwargs`" is in a function definition, `kwargs` receives a Python dictionary containing all keyword arguments after the formal parameter list).

```
>>> String('{{foo}}').bind(foo = 'bar') == String('{{foo}}').bind({'foo': 'bar'})
True
```

Bindings done "closer" to the object in question take precedence:

```
>>> p = Process(name = '{{context}}_process')
>>> t = Task().bind(context = 'global')
>>> t(processes = [p, p.bind(context = 'local')])
Task(processes=ProcessList(
  Process(daemon=False, name=global_process, max_failures=1, ephemeral=False, final=False,
          min_duration=5),
  Process(daemon=False, name=local_process, max_failures=1, ephemeral=False, final=False,
          min_duration=5)
))
```

Binding Complex Objects

Lists

```
>>> fibonacci = List(Integer)([1, 1, 2, 3, 5, 8, 13])
>>> String('{{fib[4]}}').bind(fib = fibonacci)
String(5)
```

Maps

```
>>> first_names = Map(String, String)({'Kent': 'Clark', 'Wayne': 'Bruce', 'Prince': 'Diana'})
>>> String('{{first[Kent]}}').bind(first = first_names)
String(Clark)
```

Structurals

```
   >>> String('{{p.cmdline}}').bind(p = Process(cmdline = "echo hello world"))
   String(echo hello world)

Structural Binding
```

Use structural templates when binding more than two or three individual values at the Job or Task level. For fewer than two or three, standard key to string binding is sufficient.

Structural binding is a very powerful pattern and is most useful in Aurora/Thermos for doing Structural configuration. For example, you can define a job profile. The following profile uses `HDFS`, the Hadoop Distributed File System, to designate a file's location. `HDFS` does not come with Aurora, so you'll need to either install it separately or change the way the dataset is designated.

```
class Profile(Struct):
  version = Required(String)
  environment = Required(String)
  dataset = Default(String, hdfs://home/aurora/data/{{environment}}')

PRODUCTION = Profile(version = 'live', environment = 'prod')
DEVEL = Profile(version = 'latest', environment = 'devel', dataset = 'hdfs://home/aurora/data/test')
TEST = Profile(version = 'latest', environment = 'test')

JOB_TEMPLATE = Job(
  name = 'application',
  role = 'myteam',
  cluster = 'cluster1',
  environment = '{{profile.environment}}',
  task = SequentialTask(
    name = 'task',
    resources = Resources(cpu = 2, ram = 4*GB, disk = 8*GB),
    processes = [
  Process(name = 'main', cmdline = 'java -jar application.jar -hdfsPath
            {{profile.dataset}}')
    ]
   )
 )
```

```
jobs = [
  JOB_TEMPLATE(instances = 100).bind(profile = PRODUCTION),
  JOB_TEMPLATE.bind(profile = DEVEL),
  JOB_TEMPLATE.bind(profile = TEST),
 ]
```

In this case, a custom structural "Profile" is created to self-document the configuration to some degree. This also allows some schema "type-checking", and for default self-substitution, e.g. in `Profile.dataset` above.

So rather than a `.bind()` with a half-dozen substituted variables, you can bind a single object that has sensible defaults stored in a single place.

Configuration File Writing Tips And Best Practices

Use As Few `.aurora` Files As Possible

When creating your `.aurora` configuration, try to keep all versions of a particular job within the same `.aurora` file. For example, if you have separate jobs for `cluster1`, `cluster1` staging, `cluster1` testing, and`cluster2`, keep them as close together as possible.

Constructs shared across multiple jobs owned by your team (e.g. team-level defaults or structural templates) can be split into separate `.aurora`files and included via the `include` directive.

Avoid Boilerplate

If you see repetition or find yourself copy and pasting any parts of your configuration, it's likely an opportunity for templating. Take the example below:

`redundant.aurora` contains:

```
download = Process(
  name = 'download',
  cmdline = 'wget http://www.python.org/ftp/python/2.7.3/Python-2.7.3.tar.bz2',
  max_failures = 5,
  min_duration = 1)

unpack = Process(
  name = 'unpack',
  cmdline = 'rm -rf Python-2.7.3 && tar xzf Python-2.7.3.tar.bz2',
  max_failures = 5,
  min_duration = 1)

build = Process(
  name = 'build',
  cmdline = 'pushd Python-2.7.3 && ./configure && make && popd',
  max_failures = 1)

email = Process(
  name = 'email',
  cmdline = 'echo Success | mail feynman@tmc.com',
  max_failures = 5,
  min_duration = 1)

build_python = Task(
  name = 'build_python',
  processes = [download, unpack, build, email],
  constraints = [Constraint(order = ['download', 'unpack', 'build', 'email'])])
```

As you'll notice, there's a lot of repetition in the `Process` definitions. For example, almost every process sets a `max_failures` limit to 5 and a `min_duration` to 1. This is an opportunity for factoring into a common process template.

Furthermore, the Python version is repeated everywhere. This can be bound via structural templating as described in the Advanced Binding section.

`less_redundant.aurora` contains:

```
class Python(Struct):
  version = Required(String)
  base = Default(String, 'Python-{{version}}')
  package = Default(String, '{{base}}.tar.bz2')

ReliableProcess = Process(
  max_failures = 5,
  min_duration = 1)

download = ReliableProcess(
  name = 'download',
  cmdline = 'wget http://www.python.org/ftp/python/{{python.version}}/{{python.package}}')

unpack = ReliableProcess(
  name = 'unpack',
  cmdline = 'rm -rf {{python.base}} && tar xzf {{python.package}}')

build = ReliableProcess(
  name = 'build',
  cmdline = 'pushd {{python.base}} && ./configure && make && popd',
  max_failures = 1)

email = ReliableProcess(
  name = 'email',
  cmdline = 'echo Success | mail {{role}}@foocorp.com')

build_python = SequentialTask(
  name = 'build_python',
  processes = [download, unpack, build, email]).bind(python = Python(version = "2.7.3"))
```

10.4.3 Thermos Uses bash, But Thermos Is Not bash

Bad

Many tiny Processes makes for harder to manage configurations.

```
copy = Process(
  name = 'copy',
  cmdline = 'rcp user@my_machine:my_application .'
 )

 unpack = Process(
   name = 'unpack',
   cmdline = 'unzip app.zip'
 )

 remove = Process(
   name = 'remove',
   cmdline = 'rm -f app.zip'
```

```
)

run = Process(
  name = 'app',
  cmdline = 'java -jar app.jar'
)

run_task = Task(
  processes = [copy, unpack, remove, run],
  constraints = order(copy, unpack, remove, run)
)
```

Good

Each `cmdline` runs in a bash subshell, so you have the full power of bash. Chaining commands with `&&` or `||` is almost always the right thing to do.

Also for Tasks that are simply a list of processes that run one after another, consider using the `SequentialTask` helper which applies a linear ordering constraint for you.

```
  stage = Process(
    name = 'stage',
    cmdline = 'cmdline = 'rcp user@my_machine:my_application . && ' 'unzip app.zip && rm -f app.zip'

    run = Process(name = 'app', cmdline = 'java -jar app.jar')

    run_task = SequentialTask(processes = [stage, run])
```

Rarely Use Functions In Your Configurations

90% of the time you define a function in a `.aurora` file, you're probably Doing It Wrong(TM).

Bad

```
def get_my_task(name, user, cpu, ram, disk):
  return Task(
    name = name,
    user = user,
    processes = [STAGE_PROCESS, RUN_PROCESS],
    constraints = order(STAGE_PROCESS, RUN_PROCESS),
    resources = Resources(cpu = cpu, ram = ram, disk = disk)
)

task_one = get_my_task('task_one', 'feynman', 1.0, 32*MB, 1*GB)
task_two = get_my_task('task_two', 'feynman', 2.0, 64*MB, 1*GB)
```

Good

This one is more idiomatic. Forced keyword arguments prevents accidents, e.g. constructing a task with "32*MB" when you mean 32MB of ram and not disk. Less proliferation of task-construction techniques means easier-to-read, quicker-to-understand, and a more composable configuration.

```
TASK_TEMPLATE = SequentialTask(
  user = 'wickman',
  processes = [STAGE_PROCESS, RUN_PROCESS],
)

task_one = TASK_TEMPLATE(
  name = 'task_one',
  resources = Resources(cpu = 1.0, ram = 32*MB, disk = 1*GB) )

task_two = TASK_TEMPLATE(
  name = 'task_two',
  resources = Resources(cpu = 2.0, ram = 64*MB, disk = 1*GB)
)
```

CHAPTER 11

Getting your ReviewBoard Account

Go to https://reviews.apache.org and create an account.

CHAPTER 12

Setting up your email account (committers)

Once your Apache ID has been set up you can configure your account and add ssh keys and setup an email forwarding address at

http://id.apache.org

Additional instructions for setting up your new committer email can be found at

http://www.apache.org/dev/user-email.html

The recommended setup is to configure all services (mailing lists, JIRA, ReviewBoard) to send emails to your @apache.org email address.

Setting up your ReviewBoard Environment

Run `./rbt status`. The first time this runs it will bootstrap and you will be asked to login. Subsequent runs will cache your login credentials.

CHAPTER 14

Submitting a Patch for Review

Post a review with `rbt`, fill out the fields in your browser and hit Publish.

```
./rbt post -o -g
```

CHAPTER 15

Updating an Existing Review

Incorporate review feedback, make some more commits, update your existing review, fill out the fields in your browser and hit Publish.

```
./rbt post -o -r <RB_ID>
```

CHAPTER 16

Merging Your Own Review (Committers)

Once you have shipits from the right committers, merge your changes in a single commit and mark the review as submitted. The typical workflow is:

```
git checkout master
git pull origin master
./rbt patch -c <RB_ID>  # Verify the automatically-generated commit message looks sane,
                        # editing if necessary.
git show master         # Verify everything looks sane
git push origin master
./rbt close <RB_ID>
```

Note that even if you're developing using feature branches you will not use `git merge` - each commit will be an atomic change accompanied by a ReviewBoard entry.

CHAPTER 17

Merging Someone Else's Review

Sometimes you'll need to merge someone else's RB. The typical workflow for this is

```
git checkout master
git pull origin master
./rbt patch -c <RB_ID>
git show master  # Verify everything looks sane, author is correct
git push origin master
```

The Aurora scheduler is responsible for scheduling new jobs, rescheduling failed jobs, and killing old jobs.

CHAPTER 18

Installing Aurora

Aurora is a standalone Java server. As part of the build process it creates a bundle of all its dependencies, with the notable exceptions of the JVM and libmesos. Each target server should have a JVM (Java 7 or higher) and libmesos (0.15.0) installed.

18.1 Creating the Distribution .zip File (Optional)

To create a distribution for installation you will need build tools installed. On Ubuntu this can be done with `sudo apt-get install build-essential default-jdk`.

```
git clone http://git-wip-us.apache.org/repos/asf/incubator-aurora.git
cd incubator-aurora
./gradlew distZip
```

Copy the generated `dist/distributions/aurora-scheduler-*.zip` to each node that will run a scheduler.

18.2 Installing Aurora

Extract the aurora-scheduler zip file. The example configurations assume it is extracted to `/usr/local/aurora-scheduler`.

```
sudo unzip dist/distributions/aurora-scheduler-*.zip -d /usr/local
sudo ln -nfs "$(ls -dt /usr/local/aurora-scheduler-* | head -1)" /usr/local/aurora-scheduler
```

CHAPTER 19

Configuring Aurora

19.1 A Note on Configuration

Like Mesos, Aurora uses command-line flags for runtime configuration. As such the Aurora "configuration file" is typically a `scheduler.sh` shell script of the form.

```
#!/bin/bash
AURORA_HOME=/usr/local/aurora-scheduler

# Flags controlling the JVM.
JAVA_OPTS=(
  -Xmx2g
  -Xms2g
  # GC tuning, etc.
)

# Flags controlling the scheduler.
AURORA_FLAGS=(
  -http_port=8081
  -thrift_port=8082
  # Log configuration, etc.
)

# Environment variables controlling libmesos
export JAVA_HOME=...
export GLOG_v=1
export LIBPROCESS_PORT=8083

JAVA_OPTS="${JAVA_OPTS[*]}" exec "$AURORA_HOME/bin/aurora-scheduler" "${AURORA_FLAGS[@]}"
```

That way Aurora's current flags are visible in `ps` and in the `/vars` admin endpoint.

Examples are available under `examples/scheduler/`. For a list of available Aurora flags and their documentation run

```
/usr/local/aurora-scheduler/bin/aurora-scheduler -help
```

19.2 Replicated Log Configuration

All Aurora state is persisted to a replicated log. This includes all jobs Aurora is running including where in the cluster they are being run and the configuration for running them, as well as other information such as metadata needed to

reconnect to the Mesos master, resource quotas, and any other locks in place.

Aurora schedulers use ZooKeeper to discover log replicas and elect a leader. Only one scheduler is leader at a given time - the other schedulers follow log writes and prepare to take over as leader but do not communicate with the Mesos master. Either 3 or 5 schedulers are recommended in a production deployment depending on failure tolerance and they must have persistent storage.

In a cluster with `N` schedulers, the flag `-native_log_quorum_size` should be set to `floor(N/2) + 1`. So in a cluster with 1 scheduler it should be set to `1`, in a cluster with 3 it should be set to `2`, and in a cluster of 5 it should be set to `3`.

Number of schedulers (N)

-native_log_quorum_size setting (floor(N/2) + 1)

1

1

3

2

5

3

7

4

Incorrectly setting this flag will cause data corruption to occur!

19.3 Network considerations

The Aurora scheduler listens on 3 ports - a Thrift port for client RPCs, an admin web UI, and a libprocess (HTTP+Protobuf) port used to communicate with the Mesos master and for the log replication protocol. These can be left unconfigured (the scheduler publishes all selected ports to ZooKeeper) or explicitly set in the startup script as follows:

```
# ...
AURORA_FLAGS=(
  # ...
  -http_port=8081
  -thrift_port=8082
  # ...
)
# ...
export LIBPROCESS_PORT=8083
# ...
```

CHAPTER 20

Running Aurora

Configure a supervisor like Monit or supervisord to run the created `scheduler.sh` file and restart it whenever it fails. Aurora expects to be restarted by an external process when it fails. Aurora supports an active health checking protocol on its admin HTTP interface - if a `GET /health` times out or returns anything other than `200 OK` the scheduler process is unhealthy and should be restarted.

For example, monit can be configured with

```
if failed port 8081 send "GET /health HTTP/1.0\r\n" expect "OK\n" with timeout 2 seconds for 10 cycle
```

assuming you set `-http_port=8081`.

CHAPTER 21

Maintaining an Aurora Installation

21.1 Monitoring

Aurora exports performance metrics via its HTTP interface `/vars` and `/vars.json` contain lots of useful data to help debug performance and configuration problems. These are all made available via twitter.common.http.

Java code in the aurora repo is built with Gradle.

CHAPTER 22

Getting Started

You will need Java 7 installed and on your `PATH` or unzipped somewhere with `JAVA_HOME` set. Then

```
./gradlew tasks
```

will bootstrap the build system and show available tasks. This can take a while the first time you run it but subsequent runs will be much faster due to cached artifacts.

22.1 Running the Tests

Aurora has a comprehensive unit test suite. To run the tests use

```
./gradlew build
```

Gradle will only re-run tests when dependencies of them have changed. To force a re-run of all tests use

```
./gradlew clean build
```

22.2 Creating a bundle for deployment

Gradle can create a zip file containing Aurora, all of its dependencies, and a launch script with

```
./gradlew distZip
```

or a tar file containing the same files with

```
./gradlew distTar
```

The output file will be written to `dist/distributions/aurora-scheduler.zip` or `dist/distributions/aurora-scheduler.tar`.

CHAPTER 23

Developing Aurora Java code

23.1 Setting up an IDE

Gradle can generate project files for your IDE. To generate an IntelliJ IDEA project run

```
./gradlew idea
```

and import the generated `aurora.ipr` file.

23.2 Adding or Upgrading a Dependency

New dependencies can be added from Maven central by adding a `compile` dependency to `build.gradle`. For example, to add a dependency on `com.example`'s `example-lib` 1.0 add this block:

```
compile 'com.example:example-lib:1.0'
```

NOTE: Anyone thinking about adding a new dependency should first familiarize themself with the Apache Foundation's third-party licensing policy.

CHAPTER 24

Developing Aurora UI

24.1 Installing bower (optional)

Third party JS libraries used in Aurora (located at 3rdparty/javascript/bower_components) are managed by bower, a JS dependency manager. Bower is only required if you plan to add, remove or update JS libraries. Bower can be installed using the following command:

```
npm install -g bower
```

Bower depends on node.js and npm. The easiest way to install node on a mac is via brew:

```
brew install node
```

For more node.js installation options refer to https://github.com/joyent/node/wiki/Installation.

More info on installing and using bower can be found at: http://bower.io/. Once installed, you can use the following commands to view and modify the bower repo at 3rdparty/javascript/bower_components

```
bower list
bower install <library name>
bower remove <library name>
bower update <library name>
bower help
```

CHAPTER 25

Developing the Aurora Build System

25.1 Bootstrapping Gradle

The following files were autogenerated by `gradle wrapper` using gradle 1.8's Wrapper plugin and should not be modified directly:

```
./gradlew
./gradlew.bat
./gradle/wrapper/gradle-wrapper.jar
./gradle/wrapper/gradle-wrapper.properties
```

To upgrade Gradle unpack the new version somewhere, run `/path/to/new/gradle wrapper` in the repository root and commit the changed files.

CHAPTER 26

Hooks for Aurora Client API

Introduction Hook Types Execution Order Hookable Methods Activating and Using Hooks `'.aurora` Config File Settings <#auroraConfigFileSettings>'__ Command Line Hooks Protocol `'pre_` Methods <#preMethods>'__ `'err_` Methods <#errMethods>'__ `'post_` Methods <#postMethods>'__ Generic Hooks Hooks Process Checklist

26.1 Introduction

You can execute hook methods around Aurora API Client methods when they are called by the Aurora Command Line commands.

Explaining how hooks work is a bit tricky because of some indirection about what they apply to. Basically, a hook is code that executes when a particular Aurora Client API method runs, letting you extend the method's actions. The hook executes on the client side, specifically on the machine executing Aurora commands.

The catch is that hooks are associated with Aurora Client API methods, which users don't directly call. Instead, users call Aurora Command Line commands, which call Client API methods during their execution. Since which hooks run depend on which Client API methods get called, you will need to know which Command Line commands call which API methods. Later on, there is a table showing the various associations.

Terminology Note: From now on, "method(s)" refer to Client API methods, and "command(s)" refer to Command Line commands.

26.2 Hook Types

Hooks have three basic types, differing by when they run with respect to their associated method.

`pre_<method_name>`: When its associated method is called, the `pre_` hook executes first, then the called method. If the `pre_` hook fails, the method never runs. Later code that expected the method to succeed may be affected by this, and result in terminating the Aurora client.

Note that a `pre_` hook can error-trap internally so it does not return `False`. Designers/contributors of new `pre_` hooks should consider whether or not to error-trap them. You can error trap at the highest level very generally and always pass the `pre_` hook by returning `True`. For example:

```
def pre_create(...):
    do_something()  # if do_something fails with an exception, the create_job is not attempted!
    return True

# However...
```

```
def pre_create(...):
    try:
      do_something()  # may cause exception
    except Exception:  # generic error trap will catch it
      pass  # and ignore the exception
    return True  # create_job will run in any case!
```

post_<method_name>: A post_ hook executes after its associated method successfully finishes running. If it fails, the already executed method is unaffected. A post_ hook's error is trapped, and any later operations are unaffected.

err_<method_name>: Executes only when its associated method returns a status other than OK or throws an exception. If an err_ hook fails, the already executed method is unaffected. An err_ hook's error is trapped, and any later operations are unaffected.

26.3 Execution Order

A command with pre_, post_, and err_ hooks defined and activated for its called method executes in the following order when the method successfully executes:

1. Command called
2. Command code executes
3. Method Called
4. pre_ method hook runs
5. Method runs and successfully finishes
6. post_ method hook runs
7. Command code executes
8. Command execution ends

The following is what happens when, for the same command and hooks, the method associated with the command suffers an error and does not successfully finish executing:

1. Command called
2. Command code executes
3. Method Called
4. pre_ method hook runs
5. Method runs and fails
6. err_ method hook runs
7. Command Code executes (if err_ method does not end the command execution)
8. Command execution ends

Note that the post_ and err_ hooks for the same method can never both run for a single execution of that method.

26.4 Hookable Methods

You can associate `pre_`, `post_`, and `err_` hooks with the following methods. Since you do not directly interact with the methods, but rather the Aurora Command Line commands that call them, for each method we also list the command(s) that can call the method. Note that a different method or methods may be called by a command depending on how the command's other code executes. Similarly, multiple commands can call the same method. We also list the methods' argument signatures, which are used by their associated hooks.

Aurora Client API Method

Client API Method Argument Signature

Aurora Command Line Command

cancel_update

self, job_key

cancel_update

create_job

self, config

create, runtask

restart

self, job_key, shards, update_config, health_check_interval_seconds

restart

update_job

self, config, health_check_interval_seconds=3, shards=None

update

kill_job

self, job_key, shards=None

kill

Some specific examples:

- `pre_create_job` executes when a `create_job` method is called, and before the `create_job` method itself executes.
- `post_cancel_update` executes after a `cancel_update` method has successfully finished running.
- `err_kill_job` executes when the `kill_job` method is called, but doesn't successfully finish running.

26.5 Activating and Using Hooks

By default, hooks are inactive. If you do not want to use hooks, you do not need to make any changes to your code. If you do want to use hooks, you will need to alter your `.aurora` config file to activate them both for the configuration as a whole as well as for individual `Jobs`. And, of course, you will need to define in your config file what happens when a particular hook executes.

26.6 .aurora Config File Settings

You can define a top-level hooks variable in any .aurora config file. hooks is a list of all objects that define hooks used by Jobs defined in that config file. If you do not want to define any hooks for a configuration, hooks is optional.

```
hooks = [  Object_with_defined_hooks1,
           Object_with_defined_hooks2
]
```

Be careful when assembling a config file using include on multiple smaller config files. If there are multiple files that assign a value to hooks, only the last assignment made will stick. For example, if x.aurora has hooks = [a, b, c] and y.aurora has hooks = [d, e, f] and z.aurora has, in this order, include x.aurora and include y.aurora, the hooks value will be [d, e, f].

Also, for any Job that you want to use hooks with, its Job definition in the .aurora config file must set an enable_hooks flag to True (it defaults to False). By default, hooks are disabled and you must enable them for Jobs of your choice.

To summarize, to use hooks for a particular job, you must both activate hooks for your config file as a whole, and for that job. Activating hooks only for individual jobs won't work, nor will only activating hooks for your config file as a whole. You must also specify the hooks' defining object in the hooks variable.

Recall that .aurora config files are written in Pystachio. So the following turns on hooks for production jobs at cluster1 and cluster2, but leaves them off for similar jobs with a defined user role. Of course, you also need to list the objects that define the hooks in your config file's hooks variable.

```
jobs = [
        Job(enable_hooks = True, cluster = c, env = 'prod')
        for c in ('cluster1', 'cluster2')
       ]
jobs.extend(
   Job(cluster = c, env = 'prod', role = getpass.getuser())
   for c in ('cluster1', 'cluster2')
   # Hooks disabled for these jobs
```

26.7 Command Line

All Aurora Command Line commands now accept an .aurora config file as an optional parameter (some, of course, accept it as a required parameter). Whenever a command has a .aurora file parameter, any hooks specified and activated in the .aurora file can be used. For example:

```
aurora restart cluster1/role/env/app myapp.aurora
```

The command activates any hooks specified and activated in myapp.aurora. For the restart command, that is the only thing the myapp.aurora parameter does. So, if the command was the following, since there is no .aurora config file to specify any hooks, no hooks on the restart command can run.

```
aurora restart cluster1/role/env/app
```

26.8 Hooks Protocol

Any object defined in the .aurora config file can define hook methods. You should define your hook methods within a class, and then use the class name as a value in the hooks list in your config file.

Note that you can define other methods in the class that its hook methods can call; all the logic of a hook does not have to be in its definition.

The following example defines a class containing a `pre_kill_job` hook definition that calls another method defined in the class.

```
# An example of a hook that defines a method pre_kill_job
class KillConfirmer(object):
    def confirm(self, msg):
        return True if raw_input(msg).lower() == 'yes' else False

    def pre_kill_job(self, job_key, shards=None):
        shards = ('shards %s' % shards) if shards is not None else 'all shards'
        return self.confirm('Are you sure you want to kill %s (%s)? (yes/no): '
                            % (job_key, shards))
```

26.8.1 `pre_` Methods

`pre_` methods have the signature:

```
pre_<API method name>(self, <associated method's signature>)
```

`pre_` methods have the same signature as their associated method, with the addition of `self` as the first parameter. See the chart above for the mapping of parameters to methods. When writing `pre_` methods, you can use the `*` and `**` syntax to designate that all unspecified parameters are passed in a list to the `*`ed variable and all named parameters with values are passed as name/value pairs to the `**`ed variable.

If this method returns False, the API command call aborts.

26.8.2 `err_` Methods

`err_` methods have the signature:

```
err_<API method name>(self, exc, <associated method's signature>)
```

`err_` methods have the same signature as their associated method, with the addition of a first parameter `self` and a second parameter `exc`. `exc` is either a result with responseCode other than `ResponseCode.OK` or an `Exception`. See the chart above for the mapping of parameters to methods. When writing `err_` methods, you can use the `*` and `**` syntax to designate that all unspecified parameters are passed in a list to the `*`ed variable and all named parameters with values are passed as name/value pairs to the `**`ed variable.

`err_` method return codes are ignored.

26.8.3 `post_` Methods

`post_` methods have the signature:

```
post_<API method name>(self, result, <associated method's signature>)
```

`post_` method parameters are `self`, then `result`, followed by the same parameter signature as their associated method. `result` is the result of the associated method call. See the chart above for the mapping of parameters to methods. When writing `post_` methods, you can use the `*` and `**` syntax to designate that all unspecified arguments are passed in a list to the `*`ed parameter and all unspecified named arguments with values are passed as name/value pairs to the `**`ed parameter.

`post_` method return codes are ignored.

26.9 Generic Hooks

There are five Aurora API Methods which any of the three hook types can attach to. Thus, there are 15 possible hook/method combinations for a single `.aurora` config file. Say that you define `pre_` and `post_` hooks for the `restart` method. That leaves 13 undefined hook/method combinations; `err_restart` and the 3 `pre_`, `post_`, and `err_` hooks for each of the other 4 hookable methods. You can define what happens when any of these otherwise undefined 13 hooks execute via a generic hook, whose signature is:

```
generic_hook(self, hook_config, event, method_name, result_or_err, args*, kw**)
```

where:

- `hook_config` is a named tuple of `config` (the Pystashio `config` object) and `job_key`.
- `event` is one of `pre`, `err`, or `post`, indicating which type of hook the genetic hook is standing in for. For example, assume no specific hooks were defined for the `restart` API command. If `generic_hook` is defined and activated, and `restart` is called, `generic_hook` will effectively run as `pre_restart`, `post_restart`, and `err_restart`. You can use a selection statement on this value so that `generic_hook` will act differently based on whether it is standing in for a `pre_`, `post_`, or `err_` hook.
- `method_name` is the Client API method name whose execution is causing this execution of the `generic_hook`.
- `args*`, `kw**` are the API method arguments and keyword arguments respectively.
- `result_or_err` is a tri-state parameter taking one of these three values:

1. None for `pre_`hooks
2. `result` for `post_` nooks
3. `exc` for `err_` hooks

Example:

```
# An example of a hook that overrides the standard do-nothing generic_hook
# by adding a log writing operation.

from twitter.common import log
  class Logger(object):
    '''Just adds to the log every time a hookable API method is called'''
    def generic_hook(self, hook_config, event, method_name, result_or_err, *args, **kw)
      log.info('%s: %s_%s of %s' % (self.__class__.__name__, event, method_name, hook_config.job_key)
```

26.10 Hooks Process Checklist

1. In your `.aurora` config file, add a `hooks` variable. Note that you may want to define a `.aurora` file only for hook definitions and then include this file in multiple other config files that you want to use the same hooks.

   ```
   hooks = [ ]
   ```

2. In the `hooks` variable, list all objects that define hooks used by `Jobs` defined in this config:

   ```
   hooks = [ Object_hook_definer1,
             Object_hook_definer2
           ]
   ```

3. For each job that uses hooks in this config file, add `enable_hooks = True` to the `Job` definition. Note that this is necessary even if you only want to use the generic hook.

4. Write your `pre_`, `post_`, and `err_` hook definitions as part of an object definition in your `.aurora` config file.
5. If desired, write your `generic_hook` definition as part of an object definition in your `.aurora` config file. Remember, the object must be listed as a member of `hooks`.
6. If your Aurora command line command does not otherwise take an `.aurora` config file argument, add the appropriate `.aurora` file as an argument in order to define and activate the configuration's hooks.

CHAPTER 27

Resource Isolation and Sizing

NOTE: Resource Isolation and Sizing is very much a work in progress. Both user-facing aspects and how it works under the hood are subject to change.

Introduction CPU Isolation CPU Sizing Memory Isolation Memory Sizing Disk Space Disk Space Sizing Other Resources

Introduction

Aurora is a multi-tenant system; a single software instance runs on a server, serving multiple clients/tenants. To share resources among tenants, it implements isolation of:

- CPU
- memory
- disk space

CPU is a soft limit, and handled differently from memory and disk space. Too low a CPU value results in throttling your application and slowing it down. Memory and disk space are both hard limits; when your application goes over these values, it's killed.

Let's look at each resource type in more detail:

CPU Isolation

Mesos uses a quota based CPU scheduler (the *Completely Fair Scheduler*) to provide consistent and predictable performance. This is effectively a guarantee of resources – you receive at least what you requested, but also no more than you've requested.

The scheduler gives applications a CPU quota for every 100 ms interval. When an application uses its quota for an interval, it is throttled for the rest of the 100 ms. Usage resets for each interval and unused quota does not carry over.

For example, an application specifying 4.0 CPU has access to 400 ms of CPU time every 100 ms. This CPU quota can be used in different ways, depending on the application and available resources. Consider the scenarios shown in this diagram.

- *Scenario A*: the application can use up to 4 cores continuously for every 100 ms interval. It is never throttled and starts processing new requests immediately.
- *Scenario B* : the application uses up to 8 cores (depending on availability) but is throttled after 50 ms. The CPU quota resets at the start of each new 100 ms interval.

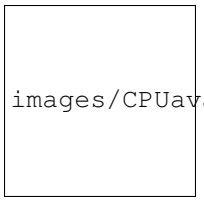

Figure 27.1: CPU Availability

- *Scenario C* : is like Scenario A, but there is a garbage collection event in the second interval that consumes all CPU quota. The application throttles for the remaining 75 ms of that interval and cannot service requests until the next interval. In this example, the garbage collection finished in one interval but, depending on how much garbage needs collecting, it may take more than one interval and further delay service of requests.

Technical Note: Mesos considers logical cores, also known as hyperthreading or SMT cores, as the unit of CPU.

CPU Sizing

To correctly size Aurora-run Mesos tasks, specify a per-shard CPU value that lets the task run at its desired performance when at peak load distributed across all shards. Include reserve capacity of at least 50%, possibly more, depending on how critical your service is (or how confident you are about your original estimate : -)), ideally by increasing the number of shards to also improve resiliency. When running your application, observe its CPU stats over time. If consistently at or near your quota during peak load, you should consider increasing either per-shard CPU or the number of shards.

Memory Isolation

Mesos uses dedicated memory allocation. Your application always has access to the amount of memory specified in your configuration. The application's memory use is defined as the sum of the resident set size (RSS) of all processes in a shard. Each shard is considered independently.

In other words, say you specified a memory size of 10GB. Each shard would receive 10GB of memory. If an individual shard's memory demands exceed 10GB, that shard is killed, but the other shards continue working.

Technical note: Total memory size is not enforced at allocation time, so your application can request more than its allocation without getting an ENOMEM. However, it will be killed shortly after.

Memory Sizing

Size for your application's peak requirement. Observe the per-instance memory statistics over time, as memory requirements can vary over different periods. Remember that if your application exceeds its memory value, it will be killed, so you should also add a safety margin of around 10-20%. If you have the ability to do so, you may also want to put alerts on the per-instance memory.

Disk Space

Disk space used by your application is defined as the sum of the files' disk space in your application's directory, including the `stdout` and `stderr` logged from your application. Each shard is considered independently. You should use off-node storage for your application's data whenever possible.

In other words, say you specified disk space size of 100MB. Each shard would receive 100MB of disk space. If an individual shard's disk space demands exceed 100MB, that shard is killed, but the other shards continue working.

After your application finishes running, its allocated disk space is reclaimed. Thus, your job's final action should move any disk content that you want to keep, such as logs, to your home file system or other less transitory storage. Disk reclamation takes place an undefined period after the application finish time; until then, the disk contents are still available but you shouldn't count on them being so.

Technical note : Disk space is not enforced at write so your application can write above its quota without getting an ENOSPC, but it will be killed shortly after. This is subject to change.

27.1 Disk Space Sizing

Size for your application's peak requirement. Rotate and discard log files as needed to stay within your quota. When running a Java process, add the maximum size of the Java heap to your disk space requirement, in order to account for an out of memory error dumping the heap into the application's sandbox space.

Other Resources

Other resources, such as network bandwidth, do not have any performance guarantees. For some resources, such as memory bandwidth, there are no practical sharing methods so some application combinations collocated on the same host may cause contention.

CHAPTER 28

Aurora Tutorial

Before reading this document, you should read over the (short) README for the Aurora docs.

- Introduction
- Setup: Install Aurora
- The Script
- Aurora Configuration
- What's Going On In That Configuration File?
- Creating the Job
- Watching the Job Run
- Cleanup
- Next Steps

CHAPTER 29

Introduction

This tutorial shows how to use the Aurora scheduler to run (and "`printf-debug`") a hello world program on Mesos. The operational hierarchy is:

- *Aurora manages and schedules jobs for Mesos to run.*
- *Mesos manages the individual tasks that make up a job.*
- *Thermos manages the individual processes that make up a task.*

This is the recommended first Aurora users document to read to start getting up to speed on the system.

To get help, email questions to the Aurora Developer List, dev@aurora.incubator.apache.org

CHAPTER 30

Setup: Install Aurora

You use the Aurora client and web UI to interact with Aurora jobs. To install it locally, see vagrant.md. The remainder of this Tutorial assumes you are running Aurora using Vagrant.

CHAPTER 31

The Script

Our “hello world” application is a simple Python script that loops forever, displaying the time every few seconds. Copy the code below and put it in a file named `hello_world.py` in the root of your Aurora repository clone (Note: this directory is the same as `/vagrant` inside the Vagrant VMs).

The script has an intentional bug, which we will explain later on.

```
import sys
import time

def main(argv):
  SLEEP_DELAY = 10
  # Python ninjas - ignore this blatant bug.
  for i in xrang(100):
    print("Hello world! The time is now: %s. Sleeping for %d secs" % (
      time.asctime(), SLEEP_DELAY))
    sys.stdout.flush()
    time.sleep(SLEEP_DELAY)

if __name__ == "__main__":
  main(sys.argv)
```

CHAPTER 32

Aurora Configuration

Once we have our script/program, we need to create a *configuration file* that tells Aurora how to manage and launch our Job. Save the below code in the file `hello_world.aurora` in the same directory as your `hello_world.py' file. (all Aurora configuration files end with`.aurora‘ and are written in a Python variant).

```
import os

# copy hello_world.py into the local sandbox
install = Process(
  name = 'fetch_package',
  cmdline = 'cp /vagrant/hello_world.py . && chmod +x hello_world.py')

# run the script
hello_world = Process(
  name = 'hello_world',
  cmdline = 'python2.6 hello_world.py')

# describe the task
hello_world_task = SequentialTask(
  processes = [install, hello_world],
  resources = Resources(cpu = 1, ram = 1*MB, disk=8*MB))

jobs = [
  Job(name = 'hello_world', cluster = 'example', role = 'www-data',
      environment = 'devel', task = hello_world_task)
]
```

For more about Aurora configuration files, see the Configuration Tutorial and the Aurora + Thermos Reference (preferably after finishing this tutorial).

CHAPTER 33

What's Going On In That Configuration File?

More than you might think.

1. From a "big picture" viewpoint, it first defines two Processes. Then it defines a Task that runs the two Processes in the order specified in the Task definition, as well as specifying what computational and memory resources are available for them. Finally, it defines a Job that will schedule the Task on available and suitable machines. This Job is the sole member of a list of Jobs; you can specify more than one Job in a config file.
2. At the Process level, it specifies how to get your code into the local sandbox in which it will run. It then specifies how the code is actually run once the second Process starts.

Creating the Job

We're ready to launch our job! To do so, we use the Aurora Client to issue a Job creation request to the Aurora scheduler.

Many Aurora Client commands take a *job key* argument, which uniquely identifies a Job. A job key consists of four parts, each separated by a "/". The four parts are `<cluster>/<role>/<environment>/<jobname>` in that order. When comparing two job keys, if any of the four parts is different from its counterpart in the other key, then the two job keys identify two separate jobs. If all four values are identical, the job keys identify the same job.

`/etc/aurora/clusters.json` within the Aurora scheduler has the available cluster names. For Vagrant, from the top-level of your Aurora repository clone, do:

```
$ vagrant ssh aurora-scheduler
```

Followed by:

```
vagrant@precise64:~$ cat /etc/aurora/clusters.json
```

You'll see something like:

```
[{
  "name": "example",
  "zk": "192.168.33.2",
  "scheduler_zk_path": "/aurora/scheduler",
  "auth_mechanism": "UNAUTHENTICATED"
}]
```

Use a `"name"` value for your job key's cluster value.

Role names are user accounts existing on the slave machines. If you don't know what accounts are available, contact your sysadmin.

Environment names are namespaces; you can count on `prod`, `devel` and `test` existing.

The Aurora Client command that actually runs our Job is `aurora create`. It creates a Job as specified by its job key and configuration file arguments and runs it.

```
aurora create <cluster>/<role>/<environment>/<jobname> <config_file>
```

Or for our example:

```
aurora create example/www-data/devel/hello_world /vagrant/hello_world.aurora
```

Note: Remember, the job key's `<jobname>` value is the name of the Job, not the name of its code file.

This returns:

```
[tw-mbp13 incubator-aurora (master)]$ vagrant ssh aurora-scheduler
Welcome to Ubuntu 12.04 LTS (GNU/Linux 3.2.0-23-generic x86_64)

 * Documentation:  https://help.ubuntu.com/
Welcome to your Vagrant-built virtual machine.
Last login: Fri Jan  3 02:18:55 2014 from 10.0.2.2
vagrant@precise64:~$ aurora create example/www-data/devel/hello_world /vagrant/hello_world.aurora
 INFO] Creating job hello_world
 INFO] Response from scheduler: OK (message: 1 new tasks pending for job www-data/devel/hello_world)
 INFO] Job url: http://precise64:8081/scheduler/www-data/devel/hello_world
vagrant@precise64:~$
```

CHAPTER 35

Watching the Job Run

Now that our job is running, let's see what it's doing. Access the scheduler web interface at `http://$scheduler_hostname:$scheduler_port/scheduler` Or when using `vagrant`, `http://192.168.33.5:8081/scheduler` First we see what Jobs are scheduled:

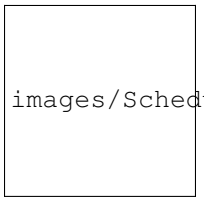

Figure 35.1: Scheduled Jobs

Click on your user name, which in this case was `www-data`, and we see the Jobs associated with that role:

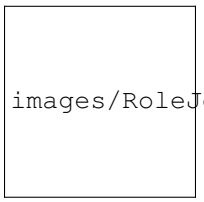

Figure 35.2: Role Jobs

Uh oh, that `Unstable` next to our `hello_world` Job doesn't look good. Click the `hello_world` Job, and you'll see:

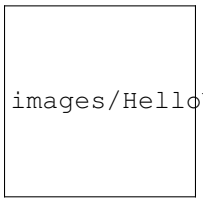

Figure 35.3: hello_world Job

Oops, looks like our first job didn't quite work! The task failed, so we have to figure out what went wrong.

Access the page for our Task by clicking on its Host.

Figure 35.4: Task page

Once there, we see that the `hello_world` process failed. The Task page captures the standard error and standard output streams and makes them available. Clicking through to `stderr` on the failed `hello_world` process, we see what happened.

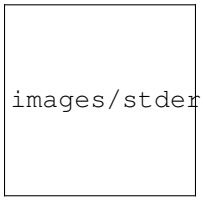

Figure 35.5: stderr page

It looks like we made a typo in our Python script. We wanted `xrange`, not `xrang`. Edit the `hello_world.py` script, save as `hello_world_v2.py` and change your `hello_world.aurora` config file to use `hello_world_v2.py` instead of `hello_world.py`.

Now that we've updated our configuration, let's restart the job:

```
$ aurora update example/www-data/devel/hello_world /vagrant/hello_world.aurora
```

This time, the task comes up, we inspect the page, and see that the `hello_world` process is running.

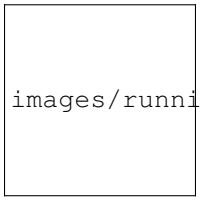

Figure 35.6: Running Task page

We then inspect the output by clicking on `stdout` and see our process' output:

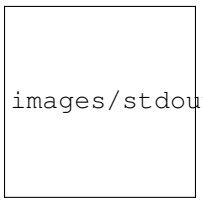

Figure 35.7: stdout page

CHAPTER 36

Cleanup

Now that we're done, we kill the job using the Aurora client:

```
vagrant@precise64:~$ aurora kill example/www-data/devel/hello_world
 INFO] Killing tasks for job: example/www-data/devel/hello_world
 INFO] Response from scheduler: OK (message: Tasks killed.)
 INFO] Job url: http://precise64:8081/scheduler/www-data/devel/hello_world
vagrant@precise64:~$
```

The Task scheduler page now shows the `hello_world` process as `KILLED`.

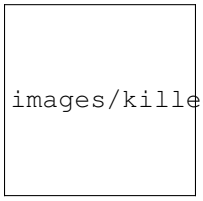

Figure 36.1: Killed Task page

CHAPTER 37

Next Steps

Now that you've finished this Tutorial, you should read or do the following:

- The Aurora Configuration Tutorial, which provides more examples and best practices for writing Aurora configurations. You should also look at the Aurora + Thermos Configuration Reference.
- The Aurora User Guide provides an overview of how Aurora, Mesos, and Thermos work "under the hood".
- Explore the Aurora Client - use the `aurora help` subcommand, and read the Aurora Client Commands document.

CHAPTER 38

Aurora User Guide

Overview Aurora Job Lifecycle Life Of A Task `PENDING to RUNNING states <#Pending>‘__ Task Updates *Giving Priority to Production Tasks: ‘‘PREEMPTING‘* <#Giving>‘__ *Natural Termination: ‘‘FINISHED‘*, FAILED <#Natural>‘__ *Forceful Termination: ‘‘KILLING‘*, RESTARTING <#Forceful>‘__ Configuration Creating Aurora Jobs Interacting With Aurora Jobs

CHAPTER 39

Overview

This document gives an overview of how Aurora works under the hood. It assumes you've already worked through the "hello world" example job in the Aurora Tutorial. Specifics of how to use Aurora are **not** given here, but pointers to documentation about how to use Aurora are provided.

Aurora is a Mesos framework used to schedule *jobs* onto Mesos. Mesos cares about individual *tasks*, but typical jobs consist of dozens or hundreds of task replicas. Aurora provides a layer on top of Mesos with its `Job` abstraction. An Aurora `Job` consists of a task template and instructions for creating near-identical replicas of that task (modulo things like "shard id" or specific port numbers which may differ from machine to machine).

How many tasks make up a Job is complicated. On a basic level, a Job consists of one task template and instructions for creating near-idential replicas of that task (otherwise referred to as "instances" or "shards").

However, since Jobs can be updated on the fly, a single Job identifier or *job key* can have multiple job configurations associated with it.

For example, consider when I have a Job with 4 instances that each request 1 core of cpu, 1 GB of RAM, and 1 GB of disk space as specified in the configuration file `hello_world.aurora`. I want to update it so it requests 2 GB of RAM instead of 1. I create a new configuration file to do that called `new_hello_world.aurora` and issue a `aurora update --shards=0-1 <job_key_value> new_hello_world.aurora` command.

This results in shards 0 and 1 having 1 cpu, 2 GB of RAM, and 1 GB of disk space, while shards 2 and 3 have 1 cpu, 1 GB of RAM, and 1 GB of disk space. If shard 3 dies and restarts, it restarts with 1 cpu, 1 GB RAM, and 1 GB disk space.

So that means there are two simultaneous task configurations for the same Job at the same time, just valid for different ranges of instances.

This isn't a recommended pattern, but it is valid and supported by the Aurora scheduler. This most often manifests in the "canary pattern" where instance 0 runs with a different configuration than instances 1-N to test different code versions alongside the actual production job.

A task can merely be a single *process* corresponding to a single command line, such as `python2.6 my_script.py`. However, a task can also consist of many separate processes, which all run within a single sandbox. For example, running multiple cooperating agents together, such as `logrotate`, `installer`, master, or slave processes. This is where Thermos comes in. While Aurora provides a `Job` abstraction on top of Mesos `Tasks`, Thermos provides a `Process` abstraction underneath Mesos `Tasks` and serves as part of the Aurora framework's executor.

You define `Jobs`,`Tasks`, and `Processes` in a configuration file. Configuration files are written in Python, and make use of the Pystachio templating language. They end in a `.aurora` extension.

Pystachio is a type-checked dictionary templating library.

TL;DR

- Aurora manages jobs made of tasks.

- Mesos manages tasks made of processes.
- Thermos manages processes.
- All defined in `.aurora` configuration file.

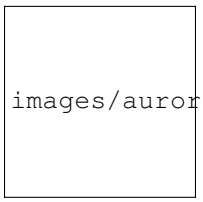

Figure 39.1: Aurora hierarchy

Each `Task` has a *sandbox* created when the `Task` starts and garbage collected when it finishes. All of a `Task`'s processes run in its sandbox, so processes can share state by using a shared current working directory.

The sandbox garbage collection policy considers many factors, most importantly age and size. It makes a best-effort attempt to keep sandboxes around as long as possible post-task in order for service owners to inspect data and logs, should the `Task` have completed abnormally. But you can't design your applications assuming sandboxes will be around forever, e.g. by building log saving or other checkpointing mechanisms directly into your application or into your `Job` description.

CHAPTER 40

Aurora Job Lifecycle

When Aurora reads a configuration file and finds a `Job` definition, it:

1. Evaluates the `Job` definition.
2. Splits the `Job` into its constituent `Task`s.
3. Sends those `Task`s to the scheduler.
4. The scheduler puts the `Task`s into `PENDING` state, starting each `Task`'s life cycle.

Note: It is not currently possible to create an Aurora job from within an Aurora job.

40.1 Life Of A Task

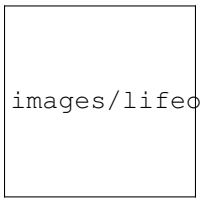

Figure 40.1: Life of a task

40.2 `PENDING` to `RUNNING` states

When a `Task` is in the `PENDING` state, the scheduler constantly searches for machines satisfying that `Task`'s resource request requirements (RAM, disk space, CPU time) while maintaining configuration constraints such as "a `Task` must run on machines dedicated to a particular role" or attribute limit constraints such as "at most 2 `Task`s from the same `Job` may run on each rack". When the scheduler finds a suitable match, it assigns the `Task` to a machine and puts the `Task` into the `ASSIGNED` state.

From the `ASSIGNED` state, the scheduler sends an RPC to the slave machine containing `Task` configuration, which the slave uses to spawn an executor responsible for the `Task`'s lifecycle. When the scheduler receives an acknowledgement that the machine has accepted the `Task`, the `Task` goes into `STARTING` state.

`STARTING` state initializes a `Task` sandbox. When the sandbox is fully initialized, Thermos begins to invoke `Process`es. Also, the slave machine sends an update to the scheduler that the `Task` is in `RUNNING` state.

If a Task stays in ASSIGNED or STARTING for too long, the scheduler forces it into LOST state, creating a new Task in its place that's sent into PENDING state. This is technically true of any active state: if the Mesos core tells the scheduler that a slave has become unhealthy (or outright disappeared), the Tasks assigned to that slave go into LOST state and new Tasks are created in their place. From PENDING state, there is no guarantee a Task will be reassigned to the same machine unless job constraints explicitly force it there.

If there is a state mismatch, (e.g. a machine returns from a netsplit and the scheduler has marked all its Tasks LOST and rescheduled them), a state reconciliation process kills the errant RUNNING tasks, which may take up to an hour. But to emphasize this point: there is no uniqueness guarantee for a single shard of a job in the presence of network partitions. If the Task requires that, it should be baked in at the application level using a distributed coordination service such as Zookeeper.

40.3 Task Updates

Job configurations can be updated at any point in their lifecycle. Usually updates are done incrementally using a process called a *rolling upgrade*, in which Tasks are upgraded in small groups, one group at a time. Updates are done using various Aurora Client commands.

For a configuration update, the Aurora Client calculates required changes by examining the current job config state and the new desired job config. It then starts a rolling batched update process by going through every batch and performing these operations:

- If a shard ID is present in the scheduler but isn't in the new config, then that shard is killed.
- If a shard ID is not present in the scheduler but is present in the new config, then the shard is created.
- If a shard ID is present in both the scheduler the new config, then the client diffs both task configs. If it detects any changes, it performs a shard update where it kills the old config shard and adds the new config shard.

The Aurora client continues through the shards list until all tasks are updated, in RUNNING, and healthy for a configurable amount of time. If the client determines the update is not going well (a percentage of health checks have failed), it cancels the update.

Update cancellation runs a procedure similar to the described above update sequence, but in reverse order. New instance configs are swapped with old instance configs and batch updates proceed backwards from the point where the update failed. E.g.; (0,1,2) (3,4,5) (6,7, 8-FAIL) results in a rollback in order (8,7,6) (5,4,3) (2,1,0).

Giving Priority to Production Tasks: PREEMPTING

Sometimes a Task needs to be interrupted, such as when a non-production Task's resources are needed by a higher priority production Task. This type of interruption is called a *pre-emption*. When this happens in Aurora, the non-production Task is killed and moved into the PREEMPTING state when both the following are true:

- The task being killed is a non-production task.
- The other task is a PENDING production task that hasn't been scheduled due to a lack of resources.

Since production tasks are much more important, Aurora kills off the non-production task to free up resources for the production task. The scheduler UI shows the non-production task was preempted in favor of the production task. At some point, tasks in PREEMPTING move to KILLED.

Note that non-production tasks consuming many resources are likely to be preempted in favor of production tasks.

Natural Termination: FINISHED, FAILED

A RUNNING Task can terminate without direct user interaction. For example, it may be a finite computation that finishes, even something as simple as echo hello world. Or it could be an exceptional condition in a long-lived

service. If the `Task` is successful (its underlying processes have succeeded with exit status `0` or finished without reaching failure limits) it moves into `FINISHED` state. If it finished after reaching a set of failure limits, it goes into `FAILED` state.

Forceful Termination: `KILLING`, `RESTARTING`

You can terminate a `Task` by issuing an `aurora kill` command, which moves it into `KILLING` state. The scheduler then sends the slave a request to terminate the `Task`. If the scheduler receives a successful response, it moves the Task into `KILLED` state and never restarts it.

The scheduler has access to a non-public `RESTARTING` state. If a `Task` is forced into the `RESTARTING` state, the scheduler kills the underlying task but in parallel schedules an identical replacement for it.

CHAPTER 41

Configuration

You define and configure your Jobs (and their Tasks and Processes) in Aurora configuration files. Their filenames end with the `.aurora` suffix, and you write them in Python making use of the Pystashio templating language, along with specific Aurora, Mesos, and Thermos commands and methods. See the Configuration Guide and Reference and Configuration Tutorial.

CHAPTER 42

Creating Aurora Jobs

You create and manipulate Aurora Jobs with the Aurora client, which starts all its command line commands with `aurora`. See Aurora Client Commands for details about the Aurora Client.

CHAPTER 43

Interacting With Aurora Jobs

You interact with Aurora jobs either via:

- Read-only Web UIs

  Part of the output from creating a new Job is a URL for the Job's scheduler UI page. For example:

  ```
  vagrant@precise64:~$ aurora create example/www-data/prod/hello /vagrant/examples/jobs/hello_worl
  INFO] Creating job hello
  INFO] Response from scheduler: OK (message: 1 new tasks pending for job www-data/prod/hello)
  INFO] Job url: http://precise64:8081/scheduler/www-data/prod/hello
  ```

  The "Job url" goes to the Job's scheduler UI page. To go to the overall scheduler UI page, stop at the "scheduler" part of the URL, in this case, `http://precise64:8081/scheduler`

  You can also reach the scheduler UI page via the Client command `aurora open`:

  ```
  aurora open [<cluster>[/<role>[/<env>/<job_name>]]]
  ```

  If only the cluster is specified, it goes directly to that cluster's scheduler main page. If the role is specified, it goes to the top-level role page. If the full job key is specified, it goes directly to the job page where you can inspect individual tasks.

  Once you click through to a role page, you see Jobs arranged separately by pending jobs, active jobs, and finished jobs. Jobs are arranged by role, typically a service account for production jobs and user accounts for test or development jobs.

- The Aurora Client's command line interface

  Several Client commands have a `-o` option that automatically opens a window to the specified Job's scheduler UI URL. And, as described above, the `open` command also takes you there.

  For a complete list of Aurora Client commands, use `aurora help` and, for specific command help, `aurora help [command]`. **Note**: `aurora help open` returns `"subcommand open not found"` due to our reflection tricks not working on words that are also builtin Python function names. Or see the Aurora Client Commands document.

Aurora includes a `Vagrantfile` that defines a full Mesos cluster running Aurora. You can use it to explore Aurora's various components. To get started, install VirtualBox and Vagrant, then run `vagrant up` somewhere in the repository source tree to create a team of VMs. This may take some time initially as it builds all the components involved in running an aurora cluster.

The scheduler is listening on http://192.168.33.5:8081/scheduler The observer is listening on http://192.168.33.4:1338/ The master is listening on http://192.168.33.3:5050/

Once everything is up, you can `vagrant ssh aurora-scheduler` and execute aurora client commands using the `aurora` client.

CHAPTER 44

Troubleshooting

Most of the vagrant related problems can be fixed by the following steps: * Destroying the vagrant environment with `vagrant destroy` * Cleaning the repository of build artifacts and other intermediate output with `git clean -fdx` * Bringing up the vagrant environment with `vagrant up`

CHAPTER 45

Indices and tables

- *genindex*
- *modindex*
- *search*