cli_filter

command line filtration, manipulation, and introspection of job submissions

Douglas Jacobsen

Systems Software Engineer, NERSC

Slurm User Group * 2017/09/25

What is cli_filter

cli_filter is a new stackable plugin infrastructure that adds hooks to allow site-definable, configurable behavior of the `salloc`, `sbatch`, `srun` plus limited support for `sbcast`.

Motivation

Slurm job submission is enormously flexible with over a hundred potential options and many fold more valid permutations.

Challenges

- enforcing site policy challenges solely with job_submit
- detailed communication with users via job_submit is limited
- discovery/tracking of user behavior

Enforcing custom site policy

job_submit plugins are the *de facto* mechanism for customizing Slurm policy enforcment, e.g., by rejecting some jobs, rewriting job based on user requests, etc.

Issues with server side policy processing

job_submit plugins

- run while locks are held in slurmctld, must avoid long-running checks
- busy slurmctld may get busier processing job requests that will be rejected
- some desirable checks may not be runnable from controller node
- some functionality is CLI only, not exposed to server side

Client side policy enforcement

site policy enforced in sbatch, salloc, srun

Client-side enforcement of site policy can

- reduce the workload of slurmctld
- run processes on parallel filesystems you might prefer your controller node didn't get jammed up upon.

WARNING

client side manipulation (wrapper scripts, cli_filter) cannot be relied upon for security needs.

Simple job_submit/lua

```
function slurm_job_submit(req, partitions, uid)
  return check_qos(req, nil)
end

function slurm_job_modify(req, desc, partitions, uid)
  return check_qos(req, desc)
end

function check_qos(req, desc)
  local qos = read_str_field(req, desc, "qos")
  if qos == "premium" then
    slurm.log_user('premium job disabled at present, please use regular')
    return slurm.ERROR
  end

  local user_balance, acct_balance, cost_est
  user_balance = get_user_balance(job, desc)
  cost_est = get_cost_estimate(job, desc)

  if user_balance < cost_est  then
    req['qos'] = 'scavenger'
  end

  return slurm.SUCCESS
end
```

User Communication

- job_submit provides `slurm.log_user()` to return a message back user CLI
    - only in case of job rejection

Often, we want to warn the user about a modification automatically made, or some more visible message (unless the CLI `--parsable` flag is set).

Discovery/tracking of user behavior

Monitoring user behavior pervasively helps site staff to

- provide support for user requests
- discover patterns of usage to set staff priorities
- proactively identify users and workloads that may benefit from consulting assistance — or other intervention

Slurm CLI *are* the User Interface

Users interact with them explicitly with

- command line options (command-specific options)
- #SBATCH statements (script-specific options)
- environment variables set explicitly by the user (*e.g.*, `$SBATCH_RESERVATION`)

and implicitly

- by environment variables propagated within the job, like srun responding to `$SLURM_JOB_ID`

Example script

- test.sh:

```
#!/bin/bash
#SBATCH -p regular
#SBATCH -t 5:00:00
#SBATCH --constraints=haswell

srun ./my_openmp_app "$@"
```

- Execution:

```
sbatch -N 5 --reservation=dmj test.sh input.
```

Analysis of the example

Script makes it clear that the user requested

- the **regular** partition
- a **5 hour** time limit
- **haswell** nodes.

Issues

- The number of nodes, reservation, and script arguments are not recorded in the script.
- It appears to be an openmp application, was `$OMP_NUM_THREADS` set? cpu_binding style?
- Debugging this user's experience will rely somewhat on their memory of the job submission.

Monitoring Slurm Data

Needed Data beyond the Slurm Database

- slurmctld data structure representations of job/step data
  - jobcomp/nersc
- capturing and logging all job and step submissions options, including aspects of the environment
  - **cli_filter (this topic)**
- jobacctgather profiling data (site enforced)
  - need more scalable backends, hdf5 file per node per job doesn’t scale well
  - early 2018 NERSC priority

What was cli_filter again?

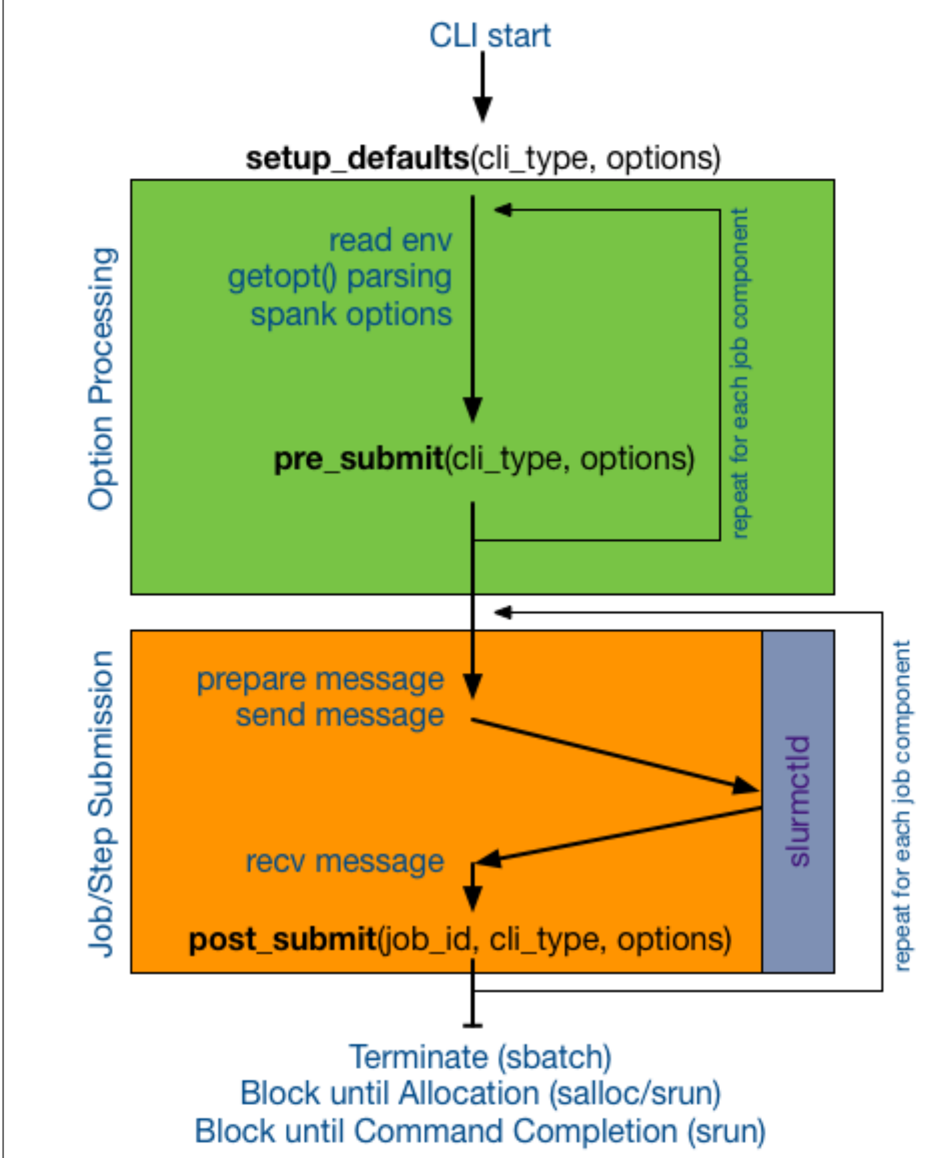
cli_filter adds hooks to allow site-definable, configurable behavior of the `salloc`, `sbatch`, `srun` plus limited support for `sbcast`.

Implemented for slurm versions

- 17.02
- 17.11

Not included with SchedMD distribution (in progress), should be considered **tech preview** right now.

cli_filter hook flowchart


CLI start
setup_defaults(cli_type, options)
Option Processing
read env
getopt() parsing
spank options
pre_submit(cli_type, options)
repeat for each job component
Job/Step Submission
prepare message
send message
slurmctld
recv message
post_submit(job_id, cli_type, options)
repeat for each job component
Terminate (sbatch)
Block until Allocation (salloc/srun)
Block until Command Completion (srun)

non-zero exit from **setup_defaults**() or **pre_submit**() cause error exit of the CLI app.

exit status of **post_submit**() is not meaningful (the RPCs are already sent)

Programming Interface

cli_filter plugins work by manipulating the `opt` data structure used in each slurm client executable. Use code generator to map option data structure to get/set functions

- lua interface handles string/boolean/numeric types in their appropriate mappings
- C-interface indirectly handles everything through C-strings to avoid cli_plugins from directly having to handle native cli types
- C-based plugins can still use native data structures if preferred

cli_filter setup_defaults()

setup_defaults

- Runs once per cli_filter plugin per CLI execution
- Non-zero exit will terminate the CLI execution
- Runs after opt data structure allocation and initialization, before environment or option processing
- Run long-running checks exactly once

Implementations

- *cli_filter/user_defaults* reads ~/.slurm_defaults to set options
- *cli_filter/lua* to set site default options

cli_filter/user_defaults

- Set defaults command line options in `$HOME/.slurm_defaults`.

  Accepts `(<command>:?)(<cluster>:?)<option> = <value>` syntax.

- **`$HOME/.slurm_defaults`** example:

```
partition = regular
cori:constraints = knl,quad,cache
edison:constraints = ivybridge
salloc:*:qos = premium
```

cli_filter pre_submit()

pre_submit

- Runs once per job-pack per cli_filter plugin per CLI execution
- Non-zero exit will terminate CLI execution
- Runs after all option processing but before slurmctld message preparation (can change options here)

Implementations

- *cli_filter/lua* plugin can be used to read options, implement policy, change options or terminate job submission

cli_filter/lua example

```
function slurm_cli_pre_submit(cli_type, options)
  -- dangerous to run on controller node, may get stuck if PFS misbehaving
  local fs_quota_auth = os.execute("/usr/bin/myquota -c")
  if fs_quota_auth ~= 0 then
    slurm.log_error("ERROR: in violation of quota limits. " ..
                    "Job submission disabled.")
    return slurm.ERROR
  end
  -- TODO: check options['workdir'] to check aux filesystem quotas

  if cli_type == CLI_ALLOC and options["qos"] ~= nil
          and options["qos"] == "interactive" then

    options["immediate"] = 30
  end

  local balance = io.popen("/something/to/get/external/accounting")
  local time_requested = calculate_time(options)
  if balance > time_requested and not options["parsable"] then
    slurm.log_info("WARNING: Low on allocation, your job moving to scavenger")
  end
  return slurm.SUCCESS
end
```

cli_filter post_submit()

post_submit

- Runs once per job-pack per cli_filter plugin per CLI execution
- Non-zero exit will attempt to terminate job (invalid for sbatch)
- Runs after all option processing but before slurmctld message preparation (can change options here)

Implementations

- *cli_filter/lua* plugin can get data and log it
- *cli_filter/syslog* dumps json record of submission to syslog

cli_filter/syslog Example output

```
Sep 22 22:08:49 slurmdev srun/syslog[24345]: post_submit: {"job_id":182,"accel_bind_
  "alloc_nodelist":"slurmdev","allocate":"false",
  "argc":"1","argv":"hostname|",
  "bcast_flag":"false","begin":"0","ckpt_dir":"\/var\/slurm\/checkpoint",
  "ckpt_interval":"0","cmd_name":"hostname","compress":"0",
  "contiguous":"false","core_spec":"65534",
  "core_spec_set":"false","cores_per_socket":"-2",
  "cpu_bind_type":"0","cpu_bind_type_set":"false",
  "cpu_freq_gov":"4294967294","cpu_freq_max":"4294967294",
  "cpu_freq_min":"4294967294","cpus_per_task":"0","cpus_set":"false","cwd":"\/home\/
  "cwd_set":"false","deadline":"0","debugger_test":"false",
  "delay_boot":"4294967294","disable_status":"false",
  "distribution":"1","egid":"-1","euid":"-1",
  "exclusive":"false","extra_set":"false","gid":"100",
  "hint_set":"false","hold":"false","immediate":"0",
  "job_flags":"0","job_name":"bash","job_name_set_cmd":"false",
  "job_name_set_env":"true","jobid":"182","jobid_set":"false",
  "join":"false","kill_bad_exit":"-2","labelio":"false",
  "launch_cmd":"false","mail_type":"0","max_exit_timeout":"60",
  "max_launch_time":"0","max_nodes":"1","max_threads":"60",
  "max_wait":"0","mem_bind_type":"0","mem_per_cpu":"-2",
  "min_nodes":"1","msg_timeout":"10","multi_prog":"false",
  "multi_prog_cmds":"0","network_set_env":"false","nice":"-2",
  "no_alloc":"false","no_kill":"false","no_rotate":"false",
  "nodes_set":"true","nodes_set_env":"true",
```

Configuration

- `slurm.conf`

```
CliFilterPlugins = lua,user_defaults
```

- `$sysconfdir/cli_filter.lua` - for cli_filter/lua

```
function slurm_cli_setup_defaults(cli, opts)
  return slurm.SUCCESS
end

function slurm_cli_pre_submit(cli, opts)
  return slurm.SUCCESS
end

function slurm_cli_post_submit(cli_opts)
  return slurm.SUCCESS
end
```

One lua script to rule them all

- /etc/slurm/job_submit.lua

```
<variable definitions>

package.path = package.path .. ';/usr/lib/nersc-slurm-plugins/?.lua'
require "lib_job_submit"
```

- /etc/slurm/cli_filter.lua

```
<variable definitions>

package.path = package.path .. ';/usr/lib/nersc-slurm-plugins/?.lua'
require "lib_job_submit"
```

- The lua scripts are just stubs and call shared code. This shared code can potentially be shared amongst all clusters (NERSC does this), with the variable definitions covering the local cluster configurations.

One lua script to rule them all

- /usr/lib/nersc-slurm-plugins/lib_job_submit.lua

```
function slurm_job_submit(req, part, uid)
  return check_qos(req, nil)
end

function slurm_job_modify(req, rec, part, uid)
  return check_qos(req, rec)
end

function slurm_cli_setup_defaults(cli, opt)
  return slurm.SUCCESS
end

function slurm_cli_pre_submit(cli, opt)
  return check_qos(opt, nil)
end

function slurm_cli_post_submit(jobid, cli, opt)
  local json
  local json_env
  json = slurm.cli_json(job, opts)
  json_env = slurm.cli_json_env()
  local msg = '{"host":"cluster","type":"slurm","job":' .. json .. ',"env":"' .. json_env
  proc = io.popen("/usr/bin/nc <loghost> <port>", "w")
  proc.write(msg)
  proc.close()
```

Current State

Code currently at

- 17.11: **https://github.com/dmjacobsen/slurm/tree/cli_filter**
- 17.02: **https://github.com/dmjacobsen/slurm/tree/cli_filter-17.02**

Future Work

Upcoming

- Working with SchedMD to explore merge options
- Ideally, functionalize routines called by getopt() switch() so cli_filter updates can (optionally) re-run slurm actions upon pre_submit() update
- Mechanism for transmit data from cli_filter to job_submit, and verify the message is truly from cli_filter pre_submit()

Questions?