# Scheduler

Feb 2nd, 2020
Paul Homer
Doc Version: 0.1.0

## Summary

Run processes remotely, in a safe and secure manner. Make it really easy to setup and use (set and forget)

## Requirements

- Install a scheduler on the server, its cli on client.
- Setup authentication & authorization.
- Define a job.
- Start a job from client.
- Stop a job from client.
- Query the status of a job from client.
- Get the output (tail -f) of a running or completed process.
- Minimize code size and complexity.

## Additional Goals

- Simplify the authentication, avoid having a password.
- Have the server-side assemble all of the details for the connection, so the user doesn't have to know, they just have access their server account and get the info.
- Move around the authentication/connection information as a single file. User is responsible for protecting this.
- Scope user's authorization to the server account's authorization (can do anything that they could do if they were logged in (they don't have a tty, so this isn't quite true)).
- Have an account on the server act as the implicit proof that the user has already gained permissions to access the server.
- Simplify the cli, avoid as many flags as possible.
- Works for both MacOS and Linux.

## Out of Scope

- Realtime scheduling.
- Tight guarantees on start times.
- Pausing jobs.
- Killing jobs mid-run.
- Support for pipes and multiple commands (stdout -> stdin).

- Second level security. If the client is compromised, the server is too. (Could easily be enhanced by putting the key in a password encrypted zip and forcing the client to ask for a password (so it can be optionally more secure).
- Ignoring long-running connection issues and keepalives. Gorilla can utilize web sockets, gRpc has settings, but that is a version 2 issue.
- Not going to split stdout and stderr, just combine them, e.g. 2>&1, not going to worry about flushing the output buffers or non-atomic writes in the combined stream.
- Not going to handle user/job/instance. Just a user job id, command string is unique to a user (but immediate and delayed are separate), and there is no instance tracking (the last run overwrites the previous ones). So, if the user runs the same command twice, the job id is identical and there is only one output stored on the server, no history.
- Not going to use a database for persistence, not going to worry about indexing for speed. Just files of JSON, locked, and any required searching is essentially a full-table scan.

# Issues

Need to see if Gorilla will allow flushing part of the response, to support long-polling...

If the user issues a long running immediate job then exits after it has started, if they reissue the same command again fast enough, it would be useful to reconnect to that last run (map the command to a jobid, and then see that that id is currently active, then spin off a goroutine to push the output into the response).

Kinda want an immediate command to jump the queue, but I would also like it to reconnect if the same thing is already running. Both scenarios are practically useful.

If I use the user + command + args as a unique key, then I don't really need a jobid, but for some commands the args matter, for some they don't. If I use user + [frequency] + command + args then I can differentiate between running something immediately and running it later, but does that make any reconnect logic weird?

Make jobids relative to each user. No reason not to.

# Proposed Interface

Keep it simple to set up, easily scriptable with as few flags as possible.

1. Setup the server
    a. Install the binaries as root, daemon is call scheduled
    b. Start it running (it will daemonize itself)
    `c. scheduled &`
2. Client Authentication
    a. Log in to the server as user account
    b. run `register`, default filename is hostname.pkg
        i. The key package will contain everything that the client needs to find, authenticate and connect to the server.
    c. cp hostname.key file to client in ~/.schedule or working dir
3. Define and start a job to run immediately

    a. `schedule ls -l`
    b. Returns output, ^C to exit
4. Define a job to run later
    a. `schedule 5mins ps -elf`
    b. returns a jobid
5. Get back the output, most recent or all of it
    a. `schedule --tail {jobid}`, ^C to exit
    b. `schedule --output {jobid}`, ^C to exit
6. Return execution or exit status from a job
    a. `schedule --status {jobid}`
    b. returns “running”, “unknown” or exit status
7. Define a job to run at a specific interval (optional)
    a. `schedule 2days 7days ping www.google.com`
    b. First arg is odd first execution period, second arg is continuing frequency
    c. `schedule 42hrs 1week ping www.google.com`
    d. supports sec(s), min(s), hour(s), day(s), week(s)
8. Remove a scheduled job ids
    a. `schedule --remove {jobid}`

# Client Code

- Check path (., ~/.schedule) for valid key packages
    - Error if none found
    - Print options and then prompt for choice if ambiguous (3 files found in ~/.schedule, for example)
- Process args (pick out any flags or time params, the rest is job definition)
    - No need to quote the full command to run, handled by vargs
    - Can’t handle pipes or multi-statements, won’t support \| or \; properly
    - vet first arg against regex for number + period, if match assume parameter, if not, assume immediate execution. Typos will cause an attempt to execute immediately: i.e. 20mims instead of 20mins
- Establish connection, issue instructions+data
- Leave connection open if running as tail/output, stream everything to tty stdout.
- Catch ^C, close connection nicely.

# Registration Code

User should just run a simple command to prove authentication, but the modified data is owned and controlled by the daemon. This occurs for classic unix commands like `passwd`; use that same technique (setuid flag) to solve it here.

- No args, just `register,` always writes pkg to local directory
- Setuid on executable, lock account file, updates server persistent account file with name, token, etc. (use file locking from github.com/juju/fslock)
- Send sighup to daemon to reload configuration.
- Assembles file(s) necessary for connection, hostname, port, certs, auth, etc. Zip if more than one file.

- Duplicate calls are idempotent (will overwrite older entries).
- Output file is hostname.key, binary zipped.

# Server Code

- Handle incoming HTTPS connections (Gorilla)
    - Validate credentials
    - 4 different types of actions
        - execute command (immediate or delayed)
        - return output tail/output
        - get status
        - remove job
- SIGHUP reloads jobs and accounts data
- Startup scheduler thread.
    - Sleeps until next job has to run, fork, chown, reset file descriptors, drop tty, exec
    - Interruptible (by os, also by job requests, by persistent changes) tied to SIGHUP
    - Serial, first in, first out, for scheduled jobs, but immediate jobs jump the queue
        - (can be operationally useful if the machine is thrashing, or having recurring problems, or running out of space, etc).
    - Simple queue for next job, allowing inserts at start and is recalculatable from the JSON file. Won't let the user run the same job twice, first instance wins if active. Reconnects.
    - Spawn off executing jobs (old style was fork/exec, but it seems Go's mechanics may interfere with this pattern).
- Locking:
    - 1 file lock on accounts config + 1 big course lock on its data structure
    - 1 big course lock (mutex) on jobs queue + active
    - 1 big course lock (mutex) on persistent jobs file + its data structure
    - Persistence: 2 files. Single JSON data-structure listing jobids and commands, account information JSON with one entry per use. Locking for first file is internal (mutex) and external for second one (file lock).
- SIGCHILD handler to reap zombies, update jobid child status.
- Spin off goroutines for streaming output, they access output files directly, watching (fsnotify) or timing out and then checking child status each time. Close when child reaped or socket closed.
- Job output: flat directory structure, directories for each user, files for output, named by jobsid & stream, not concerned about too many files per folder, or cleaning up the files at some point.

# Environment

- Go: 1.13.7
- Github