Teleport coding challenge design document by Jin Huang

## I) Architecture:

The design would consist of 3 parts, an API handler, a server, and a main. Error handling will be present appropriately in each of the functions. To use this API, the client would request to the server with curl to start, stop, or query.

Server:

- http server:
  - address: local host
  - handler: a router defined to accept start, query, and stop requests to be handled by the api handler; could use gorilla/mux for better parameter parsing over the default ServeMux
  - Log errors and messages using the logger in api handler

Main:

- Goroutine: Run the http server via ListenAndServe with caution to data races and deadlocks
- Shutdown: server will initialize quitting on OS interrupt and wait for all resources to release before shutdown to prevent leaks

API Handler (REST):

- Start, query, and stop will make use of the os/exec package to execute commands parsed from client request (e.g. ps -q pid, kill -9 pid) and return desired response or error.
- Logger: Prints out error and status messages via the log package

## II) Tradeoffs and scope:

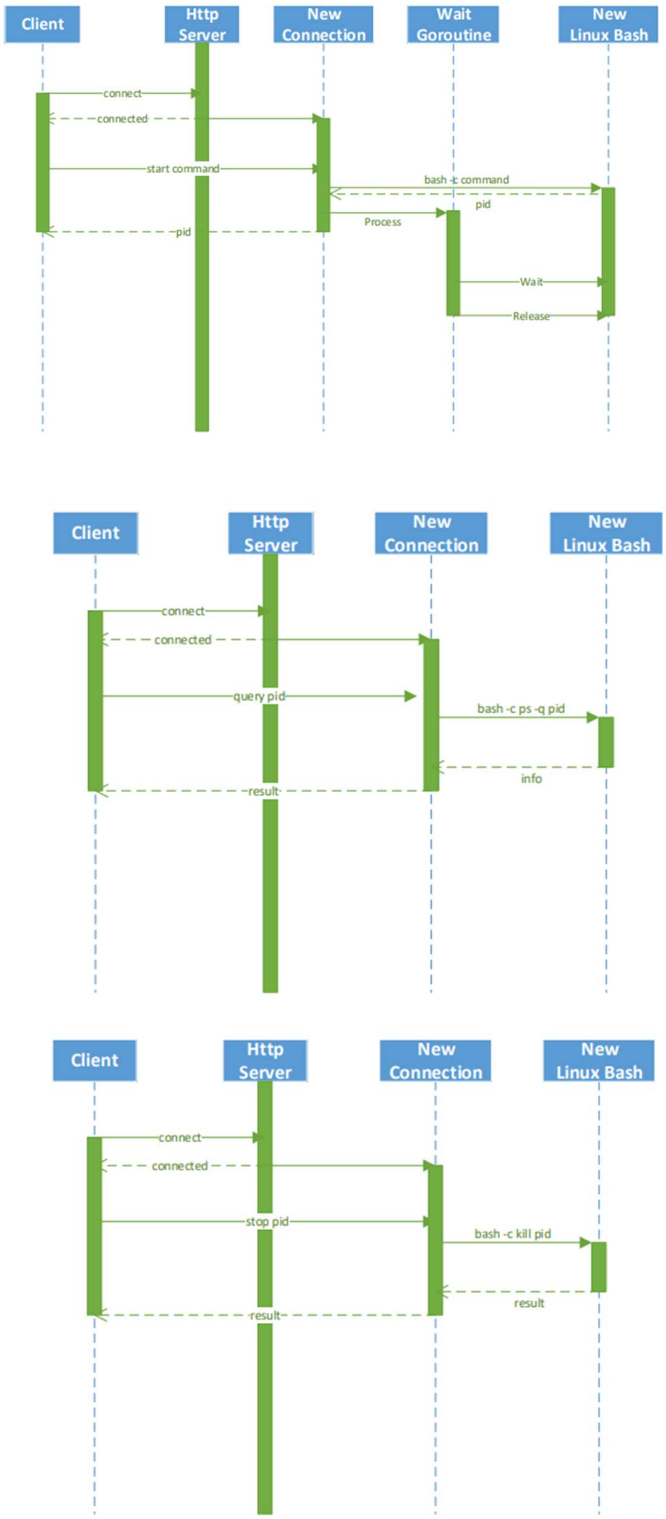
If not for simplicity, the following important features should be implemented:

1. In a scalable version of this project, the jobs would ideally be backed with database and caching such that recovery is possible when a sudden failure is encountered.
2. Authentication and security are issues that needs to be addressed, which ideally an SSL handshake should be required to ensure secure communication b/t client and the server with https.
3. Clients that access this API would ideally be registered in a database with their credentials, with a client interface that accept authorized requests to protect server against malicious requests
4. Logs should be kept consistent and saved to enable referencing and debugging, could use a library like Logrus and send log files to a centralized logging platform.
5. Could use services like Docker to containerize the development environment for dependencies, and packages

## III) Edge cases consideration:

1. Jobs may not be all complete on server shutdown, and leaks need to be prevented in goroutines (handled in main)
2. Too many concurrent running jobs with long execution time and high resource usage may cause problems (should not worry about in this current implementation)
3. Output of processes may not always be short or formatted to fit back to the client for output when requested (assume to have enough memory for output)

Example sequence diagram for start, query and stop:


Client
Http Server
New Connection
Wait Goroutine
New Linux Bash
connect
connected
start command
bash -c command
pid
Process
pid
Wait
Release
Client
Http Server
New Connection
New Linux Bash
connect
connected
query pid
bash -c ps -q pid
info
result
Client
Http Server
New Connection
New Linux Bash
connect
connected
stop pid
bash -c kill pid
result
result