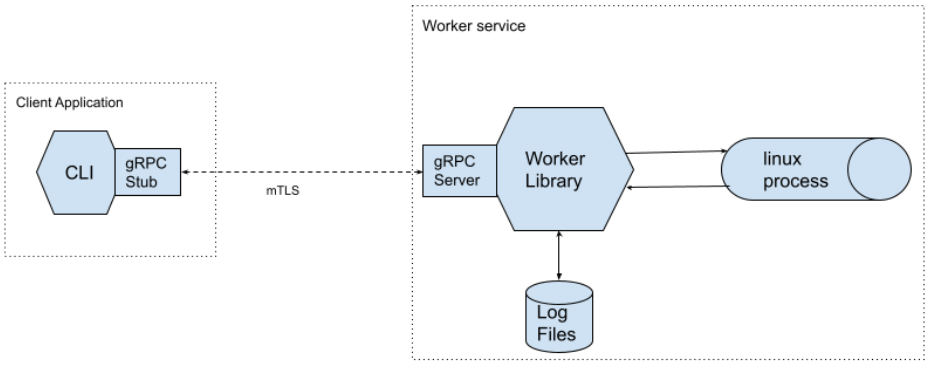
Job worker service proposal

Design of a prototype job worker service that provides an API to run arbitrary Linux processes. This implementation is aiming for the level 3 of the coding challenge given by Teleport.

I. Key designs

1. Worker library

- **General architecture:**



- **Process execution:** For arbitrary Linux processes execution, Golang supports the well defined "**os/exec**" package. With **exec.Cmd** we will allow users to execute a command with its arguments. For stopping a job, sending **SIGTERM** to the corresponding process is a graceful way to terminate it. In addition, we also provide a "-force" option to users to terminate the process immediately with **SIGKILL**.
- **Job representation:** At the programming level, a Job is an encapsulation of exec.Cmd, associated with **Job ID**, **Job status** and **Log**.
- **Job ID** is the UUID assigned to a job in the system. This ID will be used for job mapping and querying.
- **Job Status** is the status of execution of a job. This status should provide the essential information about the executed job like its owner (common name of user who started the job), the command, PID, process state (RUNNING, STOPPED, EXITED), and process exit code.
- **Job Result and Log**: During job execution, service logs, process logs and output shall be written to a log file, in order to reduce primary memory usage. In order to limit the occupied disk space by logs, we can implement log rotation or cleanup policy. For now, to simplify the implementation, we will keep log files in a temporary directory, which will result in the deletion at system reboot.
- **Log streaming**: Upon a streaming request, the worker will start a stream and subscribe to a data hub associated with a job. From this hub, the stream will gain access to the buffer, where the content of the log file is exported to. This buffer is created when there is a stream subscribed to and will be removed after a certain time of inactivity. For each hub, we will watch its associated log file and continue to export data until either the job is finished and all logs are exported, or the buffer is destroyed. In order to receive the notification of changes in the log file, we can use the fsnotify package. Users can either wait for the server to send all the data, or close the channel, to finish the streaming.

For this streaming process, we also need to take into account **long-lived streaming** for the case where a job is a long running process. For this, we need to handle **connection backoff** and **keepalive**. For a connection backoff, in order to continue the transporting stream, the client needs to specify the sequence number of the frame, where it wants to resume the streaming.

2. gRPC APIs

In order to allow the client to access the service, we can defined the following RPCs:

- **rpc StartJob(Command) returns (Job) {}:** Start a job specified by the command and return a Job with its ID. If the job fails to start, the RPC will fail with a specified error.
- **rpc StopJob(StopRequest) returns (JobStatus) {}:** Stop a job specified by the stop request then return the status of the specified job. It will return the Job status if the job is terminated successfully. Otherwise, the RPC will fail with a specified error.
- **rpc QueryJob(Job) returns (JobStatus) {}:** Query a job status, specified by its ID. The returned Job status includes the essential information about the job execution.
- **rpc StreamLog(Job) returns (stream Log) {}:** Retrieve a stream of log and output of a job, specified by job ID.

The complete service and messages definitions can be found in the **.proto** file.

3. Security

- **mTLS:** To ensure the integrity of messages, TLS v1.2 and v1.3 are both without known security issues. In this exercise, we will use **TLS v1.3**, which gives us some peak benefits like: improved latency, improved security and more secure cipher suites. Cipher suite defines the building blocks of TLS. It defines how secure the communication can take place. With TLS v1.3, we ensure perfect forward secrecy in our encryption. In the current implementation of golang "crypto/tls" package, TLS v1.3 was supported with the following cipher suites:
    + TLS_AES_128_GCM_SHA256
    + TLS_AES_256_GCM_SHA384
    + TLS_CHACHA20_POLY1305_SHA256
- **Authentication:** In mTLS, security starts with the cryptographic identity of server and client, which include the private key and certificate. For modern web applications, a **256-bit ECDSA** key is sufficient. Compared to RSA, ECDSA provides better security, with smaller keys and better performance. After having a strong private key, to achieve a high security level of certificates, we need a strong certificate signature algorithm, such as **SHA256**. For this exercise, we will create CAs for servers and clients to request their certificates. The keys and certificates shall be kept with the source code for clients and servers to load directly into their program.
- **Authorization:** For authorization, we can use Role-based access control to limit user access. We can enable **RBAC** in gRPC services with **gRPC Interceptor**. Since we already have mTLS communication in our system, we can us JWT with public key/ private key signature to control user access. For the prototype, with RBAC, we can define 2 primitive roles for our service:
    + **Admin**: Gain access right to start/stop/query/stream all jobs in the system.
    + **User:** Gain access right to start/query jobs but can only stop or stream log their created jobs.
    + **Observer**: Gain read-only access to query job status.

4. Client CLI

For the client to use the service, we can support the following command line executions:

- **worker_cli help:** Providing the user instruction on each command.
- **worker_cli start -a=domain.name:port -cert=cert.pem -key=key.pem -ca=server-ca-cert.pem -cmd="ls" "-la":** For starting a job with the command "ls -la" where domain.name:port is the address-port of the service.
- **worker_cli stop [-force] -a=domain.name:port -cert=cert.pem -key=key.pem -ca=server-ca-cert.pem -job=job-id:** For stop a job with a job ID. With the option "-force", users can force the job to terminate using SIGKILL.
- **worker_cli query -a=domain.name:port -cert=cert.pem -key=key.pem -ca=server-ca-cert.pem -job=job-id:** For query status of job.
- **worker_cli stream -a=domain.name:port -cert=cert.pem -key=key.pem -ca=server-ca-cert.pem -job=job-id:** For starting a stream of log from job.

II. Key trade-offs

1. Scalability

For the moment, because of the stateful nature, the prototype is implemented to support direct connection between client and a single worker. For production scale, we need improve system availability and scalability by applying the following changes:

- **Data store:** With a dedicated data store for job storage for all information related to jobs, we can make all workers stateless, hence, making all workers interchangeable. For this, we can use a log aggregator and persistent storage, or in-memory data storage.
- **Load balancing and Messages queue**: By making the application stateless, we can place a load balancer between client and the service. With the current architecture, we have a traditional setup where clients connect to services inside a cluster, Proxy Load Balancing is a suitable choice. Furthermore, to limit the resource usages of workers, we need a distributed scheduling mechanism, which can be done by a message queue.

2. Security

There are several ways for us to enhance the security of the service.

- **Private key protection:** The private keys are the most important assets. Therefore the keys need to be protected by an encrypted key store. If a key is compromised, the certificate needs to be revoked and a new key needs to be generated.
- **Restricted shell access:** Our service allows users to pass any unstructured request to our service. Therefore, we need to ensure that the execution of user commands does not affect our system. Because of the lack of structured data, it is not effective to use text processing or pattern matching to detect malicious requests. A way to avoid this problem is to execute our service in a containerized environment. Furthermore, we can add Process, Network and Mount namespaces to isolate the process. These isolation mechanisms hide the information of the host from the job and prevent it from affecting other users as well as the host system.
- **Credentials-based authentication and authorization:** Currently, we use certificates for authentication and JWT for authorization. The JWT is managed locally by the application by the open storage toolkit. For future development, it is better to have a dedicated service, acting as token issuer.