Version 0.5
Dec 22, 2021

Introduction

This document describes the design of JobManager – a job worker service that enables authorized users to securely run arbitrary Linux processes on remote hosts. Users can start, stop, list, and query job status, as well as stream the output of those processes.

The JobManager service is composed of three high-level components:

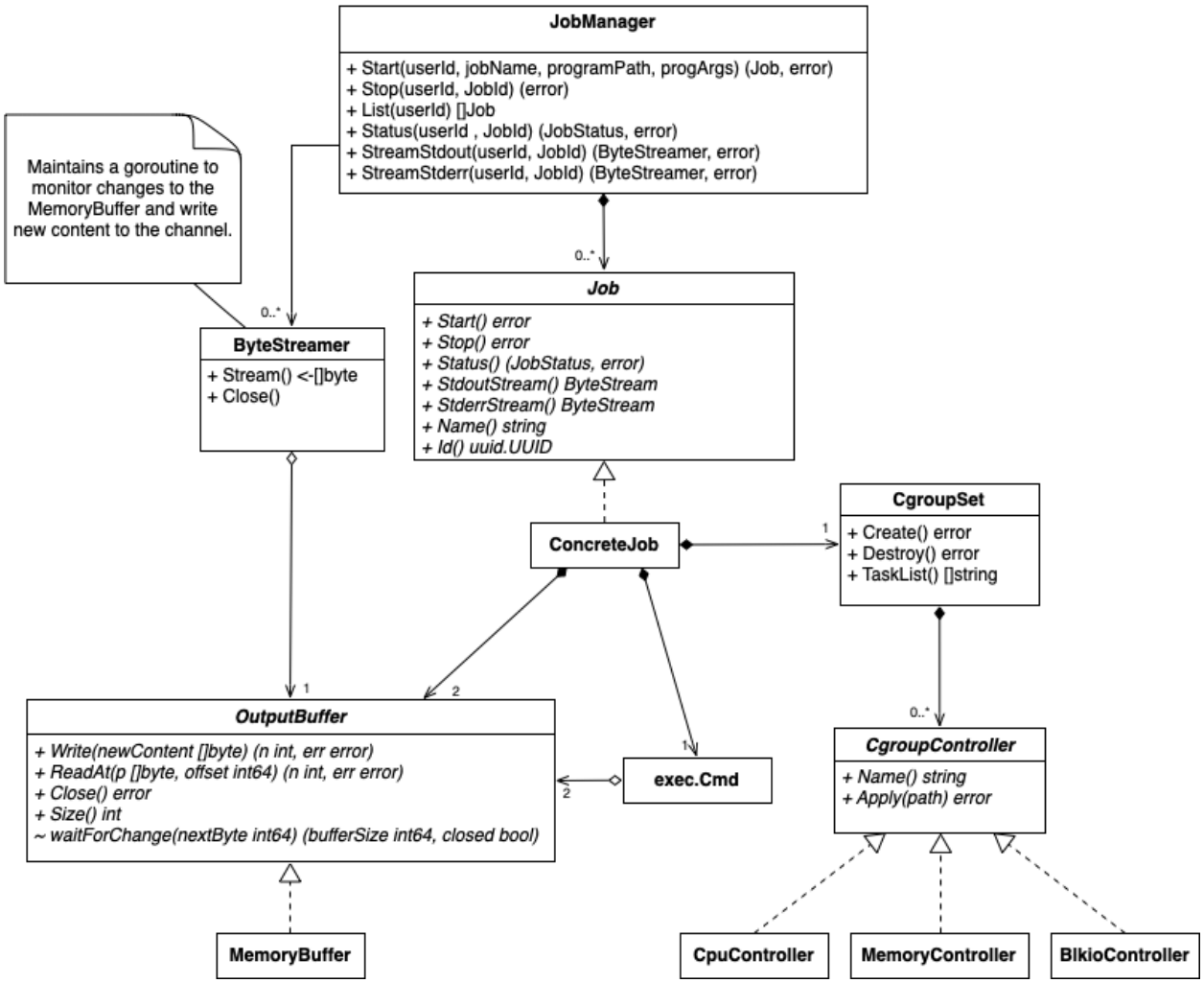
- **Library** – A Go library that exposes an API that enables callers to start, stop, list, and query the status of a job, as well as to consume a stream of the output generated by the job. The library must support multiple concurrent stream readers. The full output of the job must be available to any streaming client. The library will assign each job to its own pid, mount, and network namespaces. The library will use the cpu, memory, and blkio cgroups to limit process resource consumption.
- **API** – A gRPC API that enables clients to authenticate and securely exercise the above library. The API must use mTLS authentication and use a strong set of cipher suites along with a simple authorization scheme. The API must handle communication errors gracefully.
- **Client** – A gRPC client application that enables users to exercise the API exposed by the service. The client will enable users to start, stop, and query the status of jobs as well as view the output stream generated by the jobs.

Library

The library will consist of a number of software components. The diagram on the following page models the major components and their interactions. Note that the names of some types in the library match those defined in the gRPC API; however, the types exposed by the library will be different types defined by the library. The library will not be coupled with any gRPC components.

- The gRPC API interacts with the `JobManager` component.
- The `JobManager` component is responsible for maintaining the jobs for all users and enforcing the authorization policy (see the Security section for details on the authorization policy). The `JobManager` component maintains a collection of `Job` objects. The `Job` type exposes an interface to enable callers to manage a single job.
- The ConcreteJob object implements the Job interface and manages a single Job. It maintains a `CgroupSet` to assign the newly-created process to a set of cgroups. It maintains an `exec.Cmd` object (from the Go standard library) to execute the process. It also maintains two `OutputBuffers`, The `exec.Cmd` writes the content from the process' standard output stream to one of the buffers, and writes the content from the process' standard error stream to the other.

  A `ConcreteJob` can create any number of `ByteStream` objects associated with either of its two `OutputBuffers` to enable callers to stream the content of the buffers to clients.



Each `ByteStream` will have an associated goroutine to monitor the `OutputBuffer` for changes. The goroutine will use a condition variable to block while there is no more data to read and while the associated buffer has not been closed. The `Write` and `Close` operations on the `OutputBuffer` will broadcast on the condition variable to unblock any blocked goroutines associated with active `ByteStreamer` objects.

- The `CgroupSet` will maintain a collection of `CgroupControllers` and expose an API to enable a `Job` to create, destroy, and assign a process to the collection of cgroups.
- `CgroupController` is an interface that defines an abstraction over the different concrete cgroup controllers and enables the `CgroupSet` to apply a cgroup configuration to a cgroup at a given path.
- The `CpuController` will configure the cpu cgroup for the process. The configuration will be a floating point number that represents the “amount” of the available CPUs to allow the process to use (e.g., 1.5 -> the process can use 1.5 CPUs).
- The `MemoryController` will configure the memory cgroup for the process. The configuration will be in terms of the process memory limit in bytes.

- The `BlkioController` will configure the blkio cgroup for the process. The configuration will be in terms of read and write bytes per second.

Cgroup Configuration

The following are the proposed configuration knobs for the cgroups that JobManager will use to limit job resource utilization.

- cpu
  - `cpu.cfs_period_us = 100000`
  - `cpu.cfs_quota_us  = 100000 * <desired # of cpus>`
- memory
  - `memory.limit_in_bytes = <desired limit>`
- blkio
  - `throttle.read_bps_device = <desired read limit>`
  - `throttle.write_bps_device = <desired read limit>`

API

The following is the proposed API definition of the JobManager service.

```
// The jobmanager.v1 package includes the API for the first major
// version of the JobManager API.  This API can be extended with
// non-breaking changes. If breaking changes are required, then
// we can introduce new versions of the API alongside this one to
// maintain backward compatibility.
package jobmanager.v1;

// The JobManager service models the API exposed by the JobManager.
service JobManager {
    // Starts a new job.  The given JobCreationRequest captures the
    // details needed by the service to create the job.  Returns a
    // Job, that will enable further operations on the created job.
    rpc Start(JobCreationRequest)         returns (Job)             {}

    // Terminates a (potentially running) Job by sending it the
    // SIGKILL signal.  If the specified job is no longer running,
    // this function has no effect.
    rpc Stop(JobId)                       returns (NilMessage)      {}

    // Queries the state of the given Job.
    rpc Query(JobId)                      returns (JobStatus)       {}

    // List all jobs and their status.  Possible extensions to this
```

```
    // might enable clients to specify a filter to reduce the
    // resulting set.  Depending on the desired scale of the system,
    /// paging might also be desired.
    rpc List(NilMessage)                  returns (JobStatusList)   {}

    // Streams the output of the running job to the client.
    // The stream begins with the initial output generated by the Job
    // and ends when the Job is finished.
    rpc StreamOutput(StreamOutputRequest) returns (stream JobOutput){}
}

// A JobCreationRequest is a message that clients use to request
// the service to create a new Job.  Possible extensions to this
// would enable clients to include additional metadata (e.g., labels)
// to be associated with the newly-created jobs.
message JobCreationRequest {
    // A client-specified name for the job.  The user cannot have
    // any other current job with the same name.
    string name = 1;

    // The path of the program to run
    string programPath = 2;

    // Arguments to pass to the the program
    repeated string arguments = 3;
}

// A JobId is a message that client use to uniquely identify a job
// managed by the JobManager.
message JobId {
    // A server-assigned UUIDv4 for the job
    string id = 1;
}
```

```
// A Job is a message that uniquely identifies one of the Jobs managed
// by the JobManager service.
message Job {
    // The server-assigned ID
    JobID id = 1;

    // The client-specified name for the job
    string name = 2;
}

// The JobStatus message is used to communicate the status of a job.
message JobStatus {
    // The Job with which this status is associated
    Job job = 1;

    // The user who started the job
    string owner = 2;

    // Is the job running?
    bool isRunning = 3;

    // The process ID of the job
    int32 pid = 4;

    // If the job is not running and was terminated by a signal, which
    // signal?  If the process is terminated as a result of the Stop
    // API, then this will indicate the KILL signal (9).  If the
    // process was terminated by an extraneous signal, then this field
    // will indicate the signal number that was received.
    int32 signalNumber = 5;

    // If the job is not running, what was its exit code?
    int32 exitCode = 6;

    // If a job failed to start, what was the cause of the failure?
    string errorMessage = 7;
}
```

```
// The JobOutput message is used to stream the output of the command.
// This message can be enhanced in the future to include information
// about the byte offset into the output if this information would
// be useful to clients.
message JobOutput {
    // An array of bytes corresponding to the next “chunk” of command
    // output (either stdout or stderr).
    bytes output = 1;
}

// The JobStatusList message is used to communicate the list of jobs
// managed by the JobManager and their status.
message JobStatusList {
    repeated JobStatus jobStatusList = 1;
}

// The OutputStream enumeration captures the set of output stream
// the JobManager can stream from the process.
enum OutputStream {
    // The value if the client didn’t specify the stream
    OutputStream_UNPSECIFIED = 0;

    // Stream the process’ standard output stream
    OutputStream_STDOUT = 1;

    // Stream the process’ standard error stream
    OutputStream_STDERR = 2;
}

// The StreamOutputRequests requests that the server stream the
// selected OutputStream for the job with the given jobId.
message StreamOutputRequest {
    JobId jobId = 1;
    OutputStream outputStream = 2;
}

// The NilMessage message is used when no other message is needed.
message NilMessage {}
```

Client

The client application will provide a minimal interface to enable users to exercise the API exposed by the server. The CA will be hard coded. The command will take an optional argument to specify the userID. The default userID will be `client1`.

Note that the authorization policy of the server will enable a user — uniquely identified by the provided certificate — to operate only on jobs created by the same user or by the administrator user.

```
$ jobctl --help
jobctl enables uses to manage JobManager jobs via the gRPC API

Usage:
 jobctl [flags]
 jobctl [command]

Available Commands:
 completion  Generate the autocompletion script for the
             specified shell
 help        Help about any command
 list        List job
 query       Query job state
 start       Start a new job
 stop        Stop a job
 stream      Stream job output

Flags:
 -h, --help              help for jobctl
     --hostPort string   The <hostName>:<portNumber> of the
                         jobmanager server (default ":24482")
 -u, --userID string     The name of the user (selects the
                         appropriate credential)
                         (default "client1")

Use "jobctl [command] --help" for more information about a command.
```

The following subsections will show each command. In the examples, the `key` and `cert` parameters are omitted for brevity.

Start Command

The `start` command is used to start a new job:

```
$ jobctl start --help
Starts a new job with the given parameters on the JobManager

Usage:
 jobctl start [flags]

Examples:
jobctl start -j myJob -c /usr/bin/find -- /dir -type f

Flags:
 -c, --command string   The command for the job to run; must
                        supply full path
 -h, --help             help for start
 -j, --jobName string   The name of the job to create; must
                        be unique

$ jobctl start -j date -c /bin/date
+------+--------------------------------------+
| NAME |                  ID                  |
+------+--------------------------------------+
| date | be3936f3-b4a2-486a-a1d8-df885e34386c |
+------+--------------------------------------+
```

Stop Command

The `stop` command is used to terminate an existing job owned by the same user:

```
$ jobctl stop --help
Stap a job managed by the JobManger.  If the job is not running, this has no
effect.

Usage:
 jobctl stop [flags]

Examples:
jobctl stop 8de11b74-5cd9-4769-b40d-53de13faf77f

$ jobctl stop be3936f3-b4a2-486a-a1d8-df885e34386c
```

Query Command

The `query` command is used to retrieve information about a previously-created job owned by the same user:

```
$ jobctl query --help
Query the state of a job managed by JobManager

Usage:
 jobctl query [flags]

Examples:
jobctl query ba90b623-3dae-4bdd-8b96-c1ea4a999c44

$ jobctl query be3936f3-b4a2-486a-a1d8-df885e34386c
+------+--------------------------------------+---------+--------+
| NAME |                  ID                  | RUNNING |  PID   | …
+------+--------------------------------------+---------+--------+
| date | be3936f3-b4a2-486a-a1d8-df885e34386c | false   | 122796 |
+------+--------------------------------------+---------+--------+
```

Note the output also includes the Exit Code, termination signal, and any error encountered while starting the process. They are omitted here for display.

Also note that the output of the `query` command for the `administrator` user includes an additional column – the name of the user that owns the job:

```
$ jobctl -u administrator query be3936f3-b4a2-486a-a1d8-df885e34386c
+-------+------+--------------------------------------+---------+
| OWNER | NAME |                  ID                  | RUNNING | …
+-------+------+--------------------------------------+---------+
| user1 | date | be3936f3-b4a2-486a-a1d8-df885e34386c | false   |
+-------+------+--------------------------------------+---------+
```

List Command

The `list` command displays information about all jobs owned by the user.

```
$ jobctl list --help
List jobs managed by the JobManager

Usage:
 jobctl list [flags]

Examples:
jobctl list
```

```
$ jobctl list be3936f3-b4a2-486a-a1d8-df885e34386c
+------+--------------------------------------+---------+--------+
| NAME |                  ID                  | RUNNING |  PID   | …
+------+--------------------------------------+---------+--------+
| date | be3936f3-b4a2-486a-a1d8-df885e34386c | false   | 122796 |
| find | 4a451804-871c-44c1-97d2-36101ec21e8d | true    | 122819 |
+------+--------------------------------------+---------+--------+
```

Note the output also includes the Exit Code, termination signal, and any error encountered while starting the process. They are omitted here for display.

Also note that the output of the `list` command for the `administrator` user includes an additional column – the name of the user that owns the job:

```
$ jobctl -u administrator list be3936f3-b4a2-486a-a1d8-df885e34386c
+-------+------+--------------------------------------+---------+
| OWNER | NAME |                  ID                  | RUNNING | …
+-------+------+--------------------------------------+---------+
| user1 | date | be3936f3-b4a2-486a-a1d8-df885e34386c | false   |
| user1 | find | e944f047-4629-4ad9-b63c-86a07cca8a1d | true    |
| user2 | bar  | 7ddeb9d9-87b1-4b85-b906-e75e96b57aa4 | false   |
+-------+------+--------------------------------------+---------+
```

Stream Command

The `stream` command enables users to stream either stdout or stderr from a process.

```
$ jobctl stream –help
Stream the output (either stdout or stderr) generated by a job

Usage:
 jobctl stream [flags]

Examples:
jobctl stream 8de11b74-5cd9-4769-b40d-53de13faf77f

Flags:
 -h, --help            help for stream
 -s, --stream string   The output to stream ('stdout' or
                       'stderr') (default "stdout")

$ jobctl stream be3936f3-b4a2-486a-a1d8-df885e34386c
Sun Dec 19 09:41:57 EST 2021
```

```
$ jobctl stream -s stdout e944f047-4629-4ad9-b63c-86a07cca8a1d
find: ‘/etc/cron.hourly’: Permission denied
```

Note that the output of the `stream` command will continue until the job terminates. If the job has already completed, then the stream command will terminate after it receives and prints all of the requested output.

Security

The JobManager service must be able to trust that clients are legitimate, and clients must be able to trust that the service is legitimate. To facilitate this requirement, the JobManager service will employ mTLS authentication between clients and servers. The client/server pair must use strong cipher suites for TLS and a good cryptographic configuration for certificates. The server will expose a simple authorization scheme to ensure that the user is allowed to manage jobs on the target server.

TLS Configuration

The SSL Server Rating Guide [1] suggests the following configuration parameters to receive a high score from SSL Labs’ test [2]:

- TLS version 1.2 or greater
- Key or DH parameter strength >= 4096 bits
- Cipher strength >= 256 bits

Based those references, and a blog article [3] that references [2] and that tested against [2], JobManager will use the following TLS configuration:

```
cfg := &tls.Config{
   MinVersion:               tls.VersionTLS12,
   CurvePreferences:         []tls.CurveID{
       tls.CurveP521,
       tls.CurveP384,
       tls.CurveP256,
   },
   CipherSuites:             []uint16{
       tls.TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384,
       tls.TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA,
       tls.TLS_RSA_WITH_AES_256_GCM_SHA384,
       tls.TLS_RSA_WITH_AES_256_CBC_SHA,
   },
   PreferServerCipherSuites: true, /* client-side */
}
```

Authentication

Each user will be issued a client certificate that is signed by a single certificate authority. The JobManager service will authenticate a user by verifying that the supplied client certificate is signed by the trusted certificate authority, and JobManager clients will authenticate the server by verifying that the supplied server certificate is signed by the trusted certificate authority.

Authorization

Each user's client certificate will contain a `commonName` that is unique to the user. The JobManager service will extract the `commonName` field from the client-supplied certificate and use it to authorize the API operation. When a user creates a job, JobManager will associate the `commonName` in the client-supplied certificate with the job. Other API will similarly extract the `commonName` and enable the operation only on jobs owned by that user. This mechanism will enable JobManager to transparently support multiple users.

There will be an administrative user whose `commonName` field is "administrator". JobManager will enable the administrative user to access and control any job in the system.

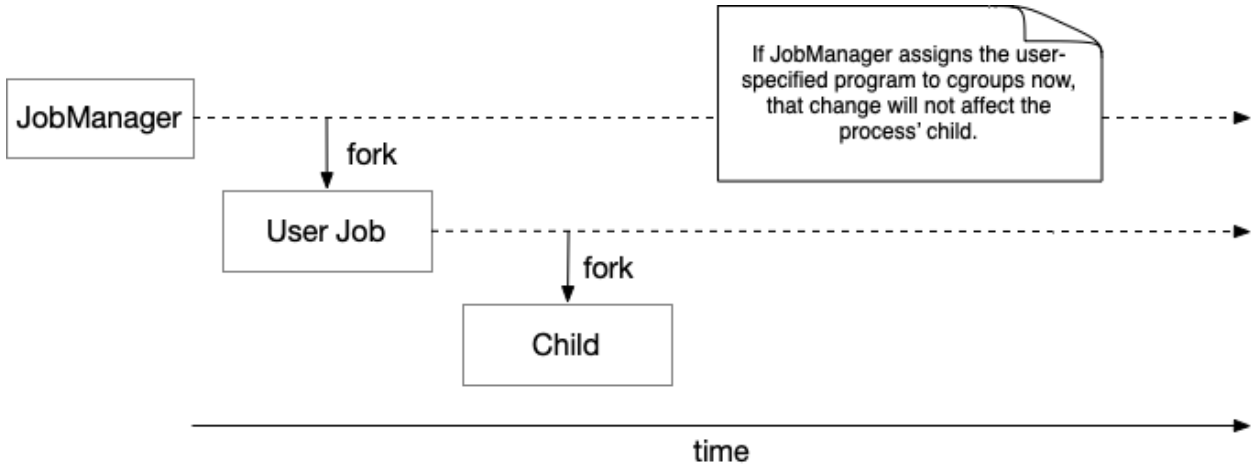
Implementation Considerations

This section highlights some of the parts of the implementation that may require special attention.

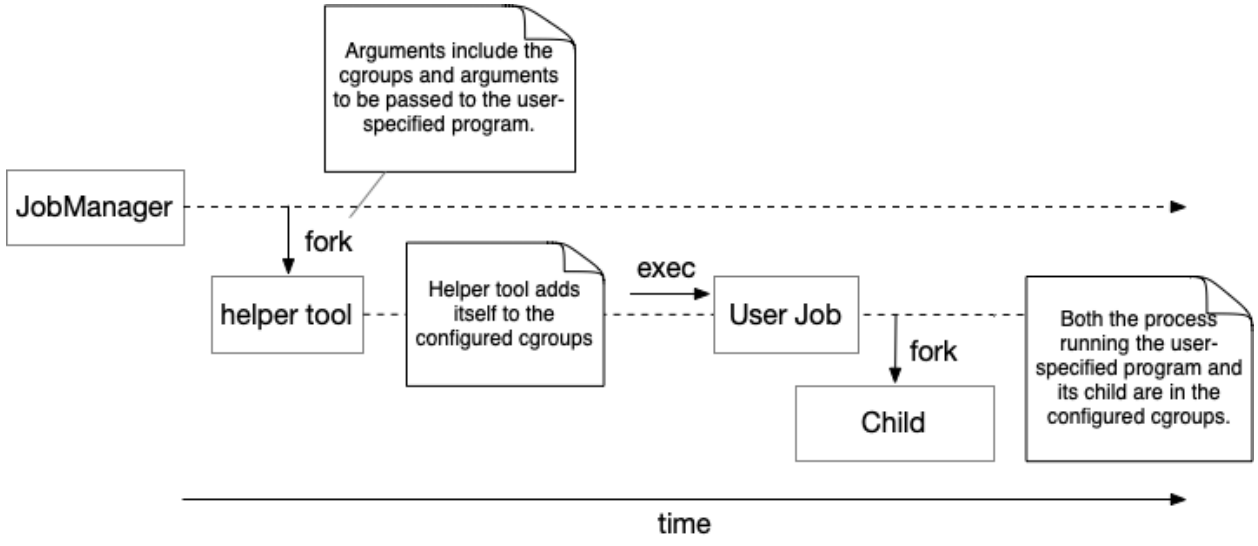
There exists a requirement that the JobManager service supports multiple concurrent clients, and that those concurrent clients can stream the output from a job. This motivates the creation of two software components, (1) an in-memory buffer to collect and store the output generated by the process, and (2) a streaming component that will monitor the buffer and stream the content written to it. One goroutine will be responsible for writing to the buffer, and there will be a goroutine associated with each instance of the streaming component. The streaming component will block while it has read all available data from the MemoryBuffer and while the MemoryBuffer has not been closed. Write and Close operations on the MemoryBuffer will unblock any blocked streaming components to enable them to either copy the newly-written data to the channel or terminate their goroutines. At any given time, there may be 0 or more of the streaming components associated with a single buffer. Proper synchronization between these components will be necessary to avoid race conditions and deadlocks. Care will need to be taken to clean up streaming components in the event of a communication issue with an API client.

There exists a requirement that the JobManager service run the processes associated with user-submitted jobs in a set of cgroups. Assigning processes to cgroups is accomplished by writing the PID of the process to the 'tasks' file in the cgroup. Child processes inherit the cgroup

assignment of their parents. If the JobManager attempts to (1) start the user-specified program immediately, then (2) assign that process to the cgroups, there exists a race condition – the newly-created process might create children before the JobManager assigns the parent to the cgroups. In that case, the children would not have the required resource constraints.



The JobManager service must ensure that the job process is assigned to the required cgroups before the user-specified program starts execution. JobManager will use a separate small helper application to exec the client-specified program, but only after it adds itself to the required cgroups. JobManager will know the cgroup mount structure within the rootfs of the process' mount namespace and will pass arguments to the helper tool that correspond to the cgroup task files to which the program should add itself.



Design Tradeoffs

A number of simplifications have been made to the design of the JobManager service to enable its completion as a challenge exercise.

- The project will be delivered with pre-generated keys and certificates. A production system would use a more controlled method for securely generating, managing, and storing those keys and certificates.
- For simplicity, the certificate hierarchy will be rooted at a self-signed root certificate. That root certificate will be used to directly sign the server certificate and each of the client certificates. A production system would use a “real” certificate authority, and might use a longer chain of trust, depending on the certificate management scheme.
- The project will not include a way to delete “old” jobs. It will maintain state about all jobs until it terminates. A production system might expose an API to enable clients to discard jobs that are no longer needed.
- The project will not include any sort of component to manage tunable configuration options. It will instead use hard-coded constants. A production system might have such a configuration component, possibly with an API to enable those values to be adjusted dynamically.
- The project will not attempt to sanitize the input provided by the user. Depending on what environment this service will be deployed, a production system might be required to prevent users from running some set of commands. That constraint might also be supported by limitations on what content is available within the chroot for the command.
- The resource constraints that the service applies to the jobs that it manages are currently hard-coded constants. In a production system, those might need to be configurable, and might need to differ between jobs.
- The project will hard code the device portion of the blkio cgroup limits. In a production system JobManager could parse `/proc/partitions` to programmatically determine that information.
- There will be no network connectivity between the network namespace in which the job is running and the outside world. A production system would set up a bridge and create tap interfaces if network access is required.
- The project will not attempt to manage different root filesystems for each job. Switching to a different mount namespace would often be accompanied by a chroot to an isolated directory on the filesystem. Each job would have its own directory, and the JobManager service would have to manage those directories, their contents, and their mounts (e.g., procfs, sysfs, devtmpfs).
- The project will not focus on scaling considerations. All jobs will run directly on the machine hosting the server. The output of each job will be stored in an in-memory buffer. A production system at scale might need to be able to distribute jobs across a set of machines, scale out that set of machines if necessary, and store job output persistently (e.g., file or a database).
- The project will not attempt to persist information across process restarts. A production system might need to persist the set of jobs and their output for future consumption, even in the face of a process restart.
- The project will not include explicit support for logging and metrics. A production system would need to provide additional visibility into the behavior of the server to enable administrators to identify when it's overloaded/misbehaving/etc.

- For the project, we use a separate go program to place jobs in their respective cgroups before execing the user-specified program. In a production system, we might use an existing tool that is packaged with libcgroup, `cgexec`, that exposes this behavior [4].

References

[1] https://github.com/ssllabs/research/wiki/SSL-Server-Rating-Guide
[2] https://www.ssllabs.com/ssltest/
[3] https://blog.bracebin.com/achieving-perfect-ssl-labs-score-with-go
[4] https://github.com/matsumotory/libcgroup/blob/master/src/tools/cgexec.c