Contents

This outlines possible approaches (and related difficulties) for a system that allows users to automate lumberjack configuration, focusing specifically on installing and configuring driver packages for the lumberjack. Note that this does not cover other considerations such as frontend components or other configuration features for lumberjack systems. The outline refers to several basic components of the system. These are described in more detail below, but can be basically defined as:

Configuration framework The component responsible for generating configuration scripts from user-defined configurations and interacting with our existing build system

Service component The component responsible for providing user-defined configurations to the configuration framework, handling and executing the resulting configuration scripts, and communicating with other processes, etc…

Configuration scripts The scripts that are generated by the configuration framework and perform the real-world effects necessary to install and configure drivers for a lumberjack

Configuration framework

Because we already use Nix as the basis for our build system, it will probably be easiest to interface with it directly by using Nix for the automation system. We could also use existing Haskell libraries to generate Nix expressions from the user-specified configuration. This introduces its own significant challenges to solve, however:

- We will need a way to connect a build-time configuration system with dynamic, user-specified configuration. This applies to Nix, but even if we were to use a different language or tool for configuration framework, we would likely still have this problem and would have the additional problem of a more awkward and fragile interface with our existing build system
- Because the complete configuration will need to use the results of impure process execution as inputs to subsequent steps, we will mostly likely need to split the configuration framework into smaller, modular pieces

Module system

The automation system will require dependencies between different options: configuring one option may depend on the results of different configuration options (computed either purely or impurely as may be the case). If we use Nix as a configuration system, the most obvious approach for solving the problem of mutual dependencies is to use the module system:

- We will have a large number of options with possibly recursive dependencies between them. This sort of problem is what the Nix module system is designed to solve
- We mostly likely want a method of verifying that options and configurations conform to some sort of schema, without limiting the complexity of the schema in order to facilitate verification (that is, we don't want to design a less powerful or flexible system merely to be able to verify it more easily). Attempting to do this outside of the module system would increase the ultimate complexity and fragility of the automation system

Configuration options

We will need to support several drivers (perhaps one dozen). Because all of these are configured using different processes and different serialized representations, we will need to write individual module configuration options for each. We will most likely also need to create different individual modules for each driver, to represent different stages of configuration. This does not mean the module system is impractical as a configuration framework, however. It would be easier to use the module system for this, in fact, because we can reuse and extend modules between different driver module options.

The Nix module system also simplifies the process of declaring dependencies between different options, which is preferable to an ad-hoc approach to managing option dependencies.

Dynamic results

We will most likely need to depend on dynamic results from external processes requiring network or database connections (e.g. submitting an add or update request to `single-well-controllogix`). Because a user may configure one service, which then performs a real-world effect and returns a result (e.g. a PID), this almost certainly cannot be accomplished within a single module evaluation because it cannot be executed in a Nix build sandbox. Because of the complexity and depth of the side effects required by configurations, we cannot have a single "configuration". Instead, we can expose a series of configurations that each evaluate to their own configuration scripts. There are two possible approaches to this:

Series of scripts

In this approach, the configuration framework wouldn't create a single configuration script in a single pass. Instead, several script-generating modules for each driver would be exposed. The service component would pass the user-defined configuration to the configuration framework to generate the initial scripts. The scripts would be executed, and the evaluation results would then be passed to a *separate* module; the results of the impure, real-world script execution would be represented in the module options of the next phase of the configuration framework. For example:

```
# The service component will evaluate this module first.
# It will produce a script that the service component
# will then execute. The result will be passed to the next
# module
{
  options.pureValue = { /*...*/ };
  config.getPidScript = pkgs.writeShellApplication {
    # This command may return a PID, for example
    text = ''
      get-pid-with-pure-value ${options.pureValue}
    '';
  };
}

# The service component will then evaluate this using
# the result of the previous step, and then create a
# *new* Nix expression to evaluate
{
  # This value was obtained impurely, but the configuration
  # system doesn't know that. The evaluation of this module
  # and the first one are separate
  options.thePid = { /*...*/ };
  config.doMoreScript = pkgs.writeShellApplication {
    text = ''
      do-something-else-with-pid ${options.thePid}
    '';
  };
}
```

Defer side effects

Instead of producing a series of scripts as part of a pipeline, we could generate all of the configuration for a single driver in a single pass. The configuration framework would generate a series of scripts for the given driver in a single evaluation. These scripts would take standard command-line arguments. The service component would execute the scripts in sequence and would supply the command-line arguments as needed. In this case, only the initial *pure* user-defined configuration options would be handled by the configuration framework, which would be used to generate the series of side-effecting scripts. For example:

```
# All of the Nix modules will be evaluated in a single
# pass. There will be no way of representing the values
# obtained impurely as part of the module system. Only
# the initial pure values are representable
{
  options.pureValue = { /*...*/ };
  config.getPidScript = pkgs.writeShellApplication {
    text = ''
      get-pid-with-pure-value ${options.pureValue}
    '';
  };
  config.doMoreScript = pkgs.writeShellApplication {
    # Because this is decoupled from the Nix evaluation,
    # it will be the service component's responsibility to
    # handle command-line arguments when executing the
    # scripts (for example, it needs to know that the
    # argument to this script is the result of the
    # previous script)
    text = ''
      do-something-else-with-pid $1
    '';
  };
}
```

Even though we can't represent *all* possible inputs using this approach, we would still be able to use as many pure values as possible, even at later stages. For example, in the `config.doMoreScript` above, we could still reference pure values when generating the script.

Configuration scripts

Format

Previous sections have referred to configuration scripts without defining the structure or format of these scripts. Two possible choices are:

Bash scripts

The most flexible approach would be to generate bash scripts that would be executed by the service component. We would largely lose the ability to maintain a structure or schema the configuration scripts, and the configuration framework would have to take a more imperative approach. If we choose to defer side effects to the latest possible point, as described above, we would mostly likely need to use this approach in order to pass execution results as a series of command-line arguments.

Systemd units

Systemd units would provide a more declarative and structured approach. There are existing Nix tools that simplify creating systemd units, and we would more easily be able to test and lint the internal Nix code. Systemd units themselves can be generated using the Nix module system, which would help prevent the creation of invalid scripts. We would lose the flexibility of bash scripts, however.

Phases

The scripts would need to be split into two broad sets or phases, installation and configuration:

Package installation

The first step in the automation system is to install the packages themselves. The first phase in the generated configuration scripts will accomplish this, mediated by the service component.

Assuming we have a "fresh" lumberjack and wish to install several driver packages on it, we will need to consider how the initial installation should be performed. Integrating the automation system with LAS is crucial to allow users to upgrade or remove packages that were installed via the automation system (of course, this applies to any operations we want to perform later via `mass-deploy` as well).

Presumably, we will want to use the existing LAS infrastructure (i.e. communicate with `lumberjack-deployment-server` to install the packages). This may require more up-front complexity in terms of connecting the two systems, but would would most likely result in less work compared to attempting to install packages in some other fashion (e.g. generating scripts that call `lumberjack-installer` directly and transmitting them to the lumberjack).

Package configuration

Once the installation of the package(s) is complete, the service component can execute scripts to configure each driver, as described in more detail above. Once the script is generated, the service component will be responsible for executing it and manipulating the results.

Service component

Regardless of the configuration framework we develop, we will need some sort of additional service that can process user-defined configurations and interface with the configuration framework. This service will have several important tasks:

Communicating with lumberjacks and other services

Because we will need to configure drivers that live on lumberjacks, the service component will need to be able to communicate with the given lumberjack and the services running on it. It would be best to reuse existing OnPing routes to accomplish this, rather than attempting to reimplement it. In addition, the service component will need to communicate with other services, e.g. LAS.

Processing user configurations

The service will need to translate user configurations (in whatever user-facing format we eventually choose) to the format of the configuration system (assumed to be Nix expressions). If the user interface is a series of forms or other inputs, this would be relatively straightforward. The frontend can post form data or JSON to the service. The structured data can be converted to Nix and the configuration scripts can be generated. If we provide a more dynamic option for user input, for example an Inferno script, this step would become significantly more difficult and may require extending Inferno itself.

The easiest interface, at least initially, would match the struture of the configuration framework exactly. For example, given configuration options for `single-well-controllogix`, we would provide a series of inputs closely matching the structure of the module options. Using the Nix module system will simplify this (for example, we may be able to automatically generate schema from the module declarations).

Storing execution results

The service component will need to use execution results from one configuration as inputs to another. Depending on the complexity involved, we could consider: simply using output redirection, i.e. capturing the output of one process to use as an input to the next; or using a database such as SQLite if we will need to use the same results in multiple places, or retrieve them at different stages in the execution pipeline

Connecting build-time configuration with dynamic input

This step is more complex and difficult to implement. Essentially, we need to bridge the user-defined configuration and the configuration framework, i.e. parameterize the build with runtime information. Of course, we can't use the typical approach of evaluating and building Nix ahead of time as the majority of possible inputs (i.e. inputs corresponding to module options) are unknown. Also, as mentioned above, there may be significant external processes required that would prevent us from building a single, comprehensive configuration script at once.

- Using a repo

  This is something of a hack, but is an established approach in the post-flakes Nix world:

  - A new repo would be created to house the module definitions defined in their own flake
  - It would have two or more inputs: `all` (in order to have access to our normal build artifacts) and a `source` input or series of inputs
  - The only artifacts exposed by the flake would be a series of packages; each package would be an applied function taking the

source as the argument. The function evaluates the Nix expression using the modules of the configuration framework described above. for example:

```
{
  packages = {
   # This is an applied function,
   # ‘buildDriver1Configuration‘ is
   # a function taking a configuration
   # and evaluating the modules
   driver1 = buildDriver1Configuration
     inputs.driver1-config-source;
  };
  # ...
}
```

  - The source input would be set by default to a Nix expression (e.g. a local path) representing a no-op
  - When overridden, the `source` input or inputs act(s) as a parameter to the function
  - After translating the user-defined configurations to Nix expressions, the service component would save it to a file. It would then build the configuration by overriding the `source` input:

```
nix build .#driver1Config \\
  --override-input driver1-config-source \\
  ./path/to/driver1/config
```

- Runtime interfacing with `all`
  - Module definitions could live in `all` or another repo
  - Nix functions would be exposed that take Nix expressions (or filepaths pointing to Nix expressions) as arguments, which are then evaluated by the configuration framework
  - Because this is necessarily impure, we could not use the standard flake interface to interact with the module definitions
  - Instead, `builtins.getFlake` could be used with `nix eval --impure --expr`
  - The service component would translate user-defined configuration into compatible Nix expressions, call `nix eval` with `builtins.getFlake`, and then execute the resulting configuration scripts

Other requirements

Because we will need to interface with either `all` directly or indirectly (if we use an auxiliary repo as described above), the service component will need to run on a machine with:

- Access to `all` (or the auxiliary repo). This is not a problem in practice as we can use flakes remotely, e.g. `nix build git+ssh://...`, provided that SSH is configured correctly
- Access to our Nix binary cache