Lisp in the middle

using Lisp to manage a Linux system

Michael Raskin[*]
raskin@mccme.ru,raskin@in.tum.de
Technical University of Munich
Garching bei München

ABSTRACT

In the Lisp community one can still find some nostalgia for the time of Lisp machines. The defining feature that has been since lost is having a powerful programming language as the main method of controlling the system behaviour.

Unfortunately, to the best of our knowledge, there are few modern systems that try to revive this approach. Moreover, regardless of the configuration language in use, managing the system as a whole is usually associated purely with managing a global persistent state, possibly with parts of it getting enabled or disabled in runtime.

We present a system design and a description of a partial implementation of Lisp-in-the-middle, a system based on the common GNU/Linux/X11 stack that uses Common Lisp for runtime system policy and per-user policy. We prioritise ease of achieving compatibility with niche workflows, low rate of purely maintenance changes, and minimising the unnecessary interactions between the parts of the system unless requested by user.

CCS CONCEPTS

• **Software and its engineering** → **Operating systems**; *Application specific development environments*;

KEYWORDS

operating systems, Linux, Lisp machines

ACM Reference Format:
Michael Raskin. 2021. Lisp in the middle: using Lisp to manage a Linux system. In *Proceedings of the 14th European Lisp Symposium (ELS'21)*. ACM, New York, NY, USA, 9 pages. https://doi.org/10.5281/zenodo.4724166

1 INTRODUCTION

Since the time when Lisp machines were in use, a variety of software has emerged that allows to use some flavour of Lisp to control some subset of the computer environment.

Among these projects, Emacs [1] probably enjoys the widest adoption. While many people use it mainly for editing text, there are plugins that add functionality ranging from email client to a window manager. Other projects used as parts of their main environment by some people are StumpWM [2] (window manager for X11) and Guix/Guix-SD [3] with Shepherd [4] (package manager based on Guile Scheme, a service management tool based on Guile, and a GNU/Linux distribution based on them).

There are interesting projects implementing entire bare-hardware OS in Lisp, such as Mezzano [5], Movitz [6] and LOSAK [7]. To the best of our knowledge, all of them were (and Mezzano currently is) developed inside virtual machines.

We experiment with using Common Lisp for access policies and glue code in the context of runtime system management. Unlike many modern tools for system state management and system policy management, we consider the case of a user with programming experience and explicit preferences about the details of the workflows and access control. Thus we do not consider the simplicity of a language used to be important, but the expressive power of a language is an important advantage. As the user has specific preferences related to the workflows, interactions working without the need to enable them explicitly are not as important as the reduction of unexpected changes caused by the updates. Additionally, tools to explicitly restrict unwanted interactions can be useful.

We aim to provide an environment close to the range of exotic GNU/Linux distributions with some access to the Lisp-implemented system functionality to shell-based workflows.

The Lisp code in the system generally implements the answers to the two main questions. What things should happen together, and what access control is necessary for privileged operations.

For example, when opening a laptop at home, one might want multiple things to happen. After a WiFi connection is established, the instant messaging client should reconnect and set the status. If the password manager has been locked, it should ask for the master password. If email fetching is paused for some reason, it should be enabled again. Screen brightness that matches the environment should be set. At the same time, screen brightness change and WiFi reconfiguration are operations requiring root access, so there should be some policy describing under which condition these changes are allowed to happen. Both parts are usually managed via a mix of manual actions, shell scripts, and service configuration settings in various languages.

Another example requiring configurable combination of system management operations with a subset of them being privileged is launching an application in a restricted and partially isolated environment. This is desirable both from the point of view of reducing the average impact of vulnerabilities in unreasonably complex applications such as office suites and modern graphical web browsers, and from the point of view of restricting unintended interactions between simultaneously running applications or the runs of the same application.

[*]The author is supported by an ERC Advanced Grant (787367: PaVeS)

For example, a user might want to launch a Firefox instance that only has network access via the university proxy, no access to most of the user files, and read/write access to a specified directory with article PDF files.

A large part of the impulse for experimentation was getting tired of keeping track of changed systemd default settings that needed to be reconfigured back to the previous default values or worked around. Note that such motivations raises importance of some *negative* requirements, i.e. a tool not doing undesired things is no less important than the tool having the desired features.

In the current state the Lisp-in-the-middle system replaces some configuration and tools with Lisp code and keeps the general structure of a GNU/Linux environment. As it is intended to adapt to niche workflows, it does not impose much of a global structure; it is more a collection of tools that turned out to be convenient for specific tasks. We believe that experiments with alternative system structures or alternative interaction approaches require either a massive upfront investment, or something that can be almost immediately used as a day-to-day environment and gradually extended from the inside. While the former approach might allow a much better eventual outcome in the case of success, we follow the latter one to reduce the risk. We hope that future expansion of the Lisp layer in the Lisp-in-the-middle system will lead to accumulation of tools and experience useful both for building a non-standard system on top of the standard low-level parts of GNU/Linux software stack, and for providing isolated and controllable instances of software packages required for interoperability. Examples of hard-to-replace packages are web browsers and office file format editors.

Centralised management of the system by interoperating daemons is what systemd does after expanding its scope from being an init system and daemon supervisor. Unlike systemd, we aim to provide configuration by defining or replacing the functions that make policy decisions. We also want to allow multiple replaceable daemons to manage different parts of the system without tight coupling between the daemons.

Currently the system exists as a small set of libraries and tools, some in Common Lisp, some in a mix of Nix/Shell/C, and an example system definition using these libraries and tools.

2 AN EXAMPLE INTERACTION: REDUCING POWER CONSUMPTION

We proceed to describe a small example of the events triggered when a user requests to reconfigure the system in the runtime. The user has unplugged the laptop from an AC power supply and plans to use the laptop without AC power supply for a significant amount time. Thus the user wishes to reduce the power consumption of the system. To achieve that, the user calls the disconnect function in an unprivileged Common Lisp image. This might be done via a REPL or via a command-line parameter to a newly started image. The disconnect function is defined in a way similar to the following.

```
(defun disconnect
     (&key kill-ssh kill-wifi kill-bg (brightness 1)
           (cpu-frequency "min") kill-matrixcli
           standby-options standby kill-mounts)
  (when kill-mounts
    (loop for d in (directory (~ "mnt/*/")) do
          (& fusermount -u (namestring d))))
  (alexandria:write-string-into-file "10"
  (format nil "~a/.watchperiod" (uiop:getenv "HOME"))
    :if-exists :supersede)
  (! web-stream-updater-starter quit)
  (uiop:run-program "rm ~/.update-web-streams-*"
    :ignore-error-status t)
  (ask-with-auth
    (:presence t)
    `(list
       (set-cpu-frequency ,cpu-frequency)
       (set-brightness ,brightness)
       ,@(when kill-wifi `((kill-wifi "wlan0")))))
  (when kill-ssh
  (ignore-errors (stumpwm-eval `(close-ssh-windows)))
  (! pkill "-HUP" ssh-fwd) (! pkill "-HUP" -f /ssh-fwd))
  (when kill-bg (kill-background-process-leaks))
  (when kill-matrixcli
   (! pkill -f /matrixcli) (! pkill -f " matrixcli"))
  (! x-options)
  (when standby (apply 'standby standby-options)))
```

Depending on the current needs, it might stop unnecessary background processes (SSH sessions, stuck Firefox instances in the background X session, IM clients, retrieval of web feeds), unmount network filesystems, change the refresh rate of the status bar, reduce CPU frequency and screen brightness, turn off the WiFi interface, reconfigure the X session in case an external monitor has been disconnected, and suspend the laptop to RAM. Some of these operations are performed using previously written shell scripts, some are implemented inside this function, some are called from another part of the configuration.

Some parts of the disconnect functions require privileged operations. In particular, we will consider the part corresponding to the change in the CPU frequency and the screen brightness. As the function is executed inside a non-privileged process, it asks the system management daemon to perform the privileged operations.

2.1 Inter-process requests

If the disconnect function is called with the default arguments, the following form gets executed.

```
(ask-with-auth
  (:presence t)
  `(list
     (set-cpu-frequency "min")
     (set-brightness 1)))
```

The ask-with-auth macro expands to the code that connects to the system management daemon socket to send the request, possibly wrapping it into some authentication or authorisation exchange. In this section we describe how the requests are represented and sent.

The main Common Lisp system management daemon creates a listening socket with an address on the file system to receive requests. The requests are s-expressions, containing strings and numeric literals. Symbols (except NIL) are deliberately not allowed for two reasons. First, there are differences in symbol treatment between Lisps (mainly related to representing packages or modules when serialising a symbol), while lists, strings and most numbers are represented in the same way. The identical representation across the Lisp language family should make migration (as well as using multiple policy daemons in different languages for managing different parts of the system) easier. Second, one of the obvious use cases for allowing some symbols in the requests is passing some part of the request as keyword parameters to a function. We want to discourage such an approach, because policies defined in such a way are likely to allow more than intended. Utility functions for personal use are likely to gain additional keyword parameters over time, and expanding their interface should not require complete review of all the possible requests. As a simple example, a function that converts files to a different format might eventually gain an extra keyword argument to remove original files in case of success. In that case, the right to pass it arbitrary keyword arguments might become more sensitive.

There is a special package for request handlers. The request must be a list; the first element must be a string, which is looked up (after converting to the upper case) in the package to find the handler that will be executed with the rest of data as parameters. Each connection also has a hashtable of extra data. This extra data can be used, for example, in the process of authentication of requests. The extra data can be freely read and modified by the request handlers.

For example, a request `("SET-CPU-FREQUENCY" "min")` will lead to a call

```
(socket-command-server-commands::set-cpu-frequency
  context "min")
```

where `context` might contain, for example, information whether this is a part of a larger request that has been authorised by the physically present user.

For convenience, the `ask-with-auth` macro allows the user to pass symbols inside the requests. Each symbol is replaced with a string representing its name. Hence calling the `disconnect` function leads to the following request content after replacing the symbols with their names.

```
("LIST"
   ("SET-CPU-FREQUENCY" "min")
   ("SET-BRIGHTNESS" 1))
```

However, we do not allow these operations to be requested by an arbitrary process without confirmation from the user, thus the conversations with the server is more complicated.

2.2 Authentication and authorisation

Before executing a request the daemon usually needs to check if the policy allows such a request. The policies are defined by Lisp code executed by the daemon. The management daemon provides a few ways to either verify the client's claims about its identity or to ask the physically present user whether the command should be permitted.

The simplest and fully automatic way of authentication is verifying that the client has the claimed user ID. This is done by writing a random token into a file readable only by the target UID and asking the client process to provide the token. A simple way of authorising a request with slightly larger impact is to ask the physically present user to confirm that the request should be executed.

Consider for example the `ask-with-auth` invocation inside the `disconnect` function. This macro always performs UID authentication, and additional authorisation by the physically present user is requested in this case. First the client process sends a request to confirm having access to the files only readable as user `raskin`.

```
("REQUEST-UID-AUTH" "raskin")
```

The server replies with the filename containing the key. The client reads the key and uses it to send an authenticated request, which requires additional authorisation via confirmation from the physically present user. Here the timeout for the user reaction is 15 seconds. The request demands to change two settings: CPU frequency and screen brightness.

```
("WITH-UID-AUTH" "JDPBTP56Y3LALB6FMA54"
  ("WITH-PRESENCE-AUTH" "T"
    ("PROGN" ("SET-CPU-FREQUENCY" "min")
             ("SET-BRIGHTNESS" 1))
    15))
```

The daemon switches to a specially reserved virtual terminal to separate the communication between the daemon and the user from the programs in the normal user session. After a switch to a dedicated virtual terminal, the user is presented with the list of operations to confirm or cancel. If the user confirms the request, the daemon adjusts CPU frequency and screen brightness using `sysfs` virtual filesystem provided by the Linux kernel for interaction with drivers. Afterwards the daemon reports the success or failure to the client process.

As screen resolutions vary in a wide range, but font size configuration in plain Linux kernel-rendered virtual terminal using the `fbcon` driver is limited, we use a reduced interface to graphics on virtual terminals (framebuffer) provided by the Linux kernel with `fbterm`, a framebuffer terminal emulator with vector font support.

2.3 Integration with existing scripts

Sometimes an existing script in another programming language is modified to use management daemon requests instead of, for example, `sudo`.

We mainly use a saved Lisp image with all the relevant library code loaded and allow the scripts to just execute it passing the Lisp code for performing a request. A shell script can then include something similar to the following.

```
lisp-shell.bin --eval '(ask-with-auth (:presence t)
                          `("SET-BRIGHTNESS" 1))'
```

3 OBTAINING INFORMATION FROM THE WEB

Web browsing is performed in a tiered manner: in the best case, plain HTML is downloaded, then it is parsed using `cl-html5-parser` [12], and then the parsed data is converted to a plain text representation using our Thoughtful Theridion library [13]. The download might be automated using Throughful Theridion's web page walker DSL. If the page requires Javascript to fetch the content, a background Firefox instance with Marionette remote control is started to obtain the data, save the HTML and also the corresponding text representation. In the unfortunate case where an interactive web browser is required, an isolated (typically per site) instance of Firefox (or Chromium for sites failing to work in Firefox) is used.

The latter two cases include starting a browser in an isolated environment. This happens with the assistance of the system management daemon. As an example, evaluating the following code in an unprivileged Lisp instance will start a virtual X session, launch Firefox with full network access and limited file system access, open the ELS page, select the current year, wait for the second page to load and return the URL of the image.

```
(uiop:launch-program "xdummy :9")
(with-new-firefox-marionette
  () (() :netns nil :display 9)
  (prog1 (progn
      (marionette-set-url
        "https://european-lisp-symposium.org/")
      (ask-marionette-parenscript
        `(ps:chain document document-element
           (query-selector "a.current") (click))
        :wait-ready t)
      (first
        (ask-marionette-parenscript
        `(return (ps:chain
             document document-element
             (query-selector "span.imagery img")
             src)))))
      (marionette-close)))
```

While the substance of the task is the code passed to the calls to `ask-marionette-parenscript`, we are interested in the request to the daemon that happens due to the `with-new-firefox-marionette` macro.

The main part of the expansion is a call to the `firefox` function that launches Firefox in an isolated environment with limited file system access and, if requested, network access restricted to accessing the selected proxy. The function also tells Firefox to enable the Marionette external control framework. This framework has been initially implemented for automated testing of Firefox itself. Marionette is also used as a foundation for Geckodriver, a tool for automated testing of websites in Firefox. The instance can be used for interactive browsing or for automated interaction with some websites.

To run Firefox (or any other command) in an isolated environment, the client process sends a request to the system management daemon. We do not show a complete request here because it has too many parameters. The request contains the configuration for the files that should be available inside, which devices such as sound cards and video cameras should be made accessible to the application, what network access should be provided, etc.

3.1 Sandboxing

The request to run some command in an isolated environment allows a client process to claim a fresh user ID and run some code under this UID for partial access control. It is also possible to request isolation using some of the technologies developed for Linux containers. For example, many vulnerabilities in the desktop programs when opening a malformed document have significantly less impact if the corresponding program has neither network access nor write access to anything outside the current directory. Using a unique UID also allows controlling the scope of daemons like PulseAudio that insist to be run only once in a user session and are required by some software. This functionality permits controlling the interaction between the non-Lisp components of the system.

The user has access to launching programs in isolated environments via a normal Lisp function, similar in purpose to `uiop:run-program` but with numerous keyword arguments to describe the desired environment and functionality available to the program.

3.2 Window management integration

If the `firefox` function was called with the `:stumpwm-tags` keyword argument, all the windows created by this Firefox instance will be automatically assigned the specified window tags by StumpWM. These tags can be used to manipulate the windows as desired. The implementation is as follows.

GUI applications launched in isolated environments can be distinguished, for example, by the internal host name of the environment. We have expanded StumpWM support of the X11 protocol window properties and provide functions to assign window tags based on window properties. In particular, the Inter-Client Communication Conventions Manual standard provides access to the hostname of the client application and the Extended Window Manager Hints standard provides access to the process ID and hostname of the client application via X11 window properties.

On the StumpWM side we add a socket with an address on the file system. Before launching the isolated application, we evaluate code in StumpWM that establishes correspondence between the chosen hostname of the isolated environment and the desired tags.

4 UPDATING THE SYSTEM

An important operation on most systems is updating the system itself. When a user updates the system, there are two parts of the task: a new persistent state of the system needs to be constructed, and a subset of runtime state needs to be reinitialised based on the new persistent state. The user initialises the system update by calling `full-refresh` function in an unprivileged Lisp instance.

For the first part we use Nix package manager [8]. First we call Nix from the unprivileged client process to prepare the new persistent state. As Nix installs each package, from `glibc` to the overall system state, into its own path, asking the Nix daemon (provided by Nix) to construct the new persistent state is non-destructive and can be done without special privileges. Then the client process asks the daemon to set the newly created system state as the active system state. This is a privileged operation; unlike more frequent operations like adjusting the screen brightness here the user is asked to enter the system root password for confirmation instead of just pressing enter. This operation essentially just changes the target of one symbolic link on the filesystem.

To update the runtime state of the daemon, the client process sends the request to the daemon to exit; a new instance of the daemon is automatically started whenever the old one exits. Any data that needs to survive such restarts is stored in external SQLite databases, mostly on RAM-backed filesystems. Such external storage of the runtime daemon state reduces the risks in case of errors in policy code leading to unhandled conditions in the daemon, as well as makes it easier to share the runtime state with a different daemon. In addition to the fully functional Common Lisp system daemon, updating the system is also supported a proof-of-concept Guile system daemon.

Some of the other daemons running in the system can also be restarted after the update by normal requests to the daemon.

5 LISP IMPLEMENTATION LIMITATIONS AND WORKAROUNDS

Common Lisp implementations (including SBCL), as well as portability libraries such as `uiop`, often expect that spawning other programs will be done in a relatively limited way. There are two kinds of cases where we currently choose to use a workaround.

The first case is starting programs that need access to the terminal input. The most annoying situation is the case of a long running program that we might want to interrupt. Unix shells intended to spawn such a case have special code for handling interruptions while a program is running in the foreground. However, if the program is launched from an SBCL REPL using `sb-ext:run-program` or `uiop:run-program`, pressing `Ctrl-c` leads to a different, less desirable behaviour. Both the program and the Lisp session interpret `Ctrl-c` normally; the program exists as desired, and the REPL starts a debugger caused by an interactive interrupt. Similarly, it is also possible to break some expectations regarding the input/output stream (for example, `Ctrl-d` can be caught both by the program invoked and the Common Lisp implementation).

A simple way to avoid this problem is to make sure the spawned program has its own separate pseudo-terminal. Using a terminal multiplexer like `screen` and running interactive programs in a new screen window is usually an acceptable workaround. We provide a macro `!!` which interprets its argument in the same as the `!` macro does, but asks `screen` to start the command in a new window instead of starting it directly. Of course, an obvious way to avoid that is to run workflows in a traditional shell, spawning a Lisp instance when necessary.

The second case is using special system functionality, for example file locks. For Common Lisp implementations it is more natural to implement concurrency as multiple threads in a shared address space. However, this means that system APIs intended for use in a multi-process model are not fully usable. We use existing wrapper programs for such functionality or write small helper executables in C. For example, if the use of POSIX `flock` function for file locking is desired, we spawn a separate process that acquires a lock, reports success when the lock is acquired, then waits for the command to release the lock.

6 OVERALL STRUCTURE OF THE SYSTEM

The initial boot is performed in a way pretty similar to other GNU/Linux distributions. The customisation of this stage is done via a shell script. Once the system runs with the normal storage properly set up, a minimalistic init process `sinit` is launched for the core system tasks (releasing the process memory structures of the processes that terminate after their corresponding parent processes has terminated, and handling the system shutdown), and the main Lisp process is started.

The Lisp process starts the login processes on the virtual terminals, and launches the daemons (such as CUPS printing daemon and OpenSSH) according to the policy. Afterwards the system Lisp daemon starts listening on a Unix domain socket on the file system for requests. The requests are used for operations requiring special access in a way similar to the way `sudo` or `doas` are used. Using a Unix domain socket for communication instead of e.g. a local HTTP server allows us to send file descriptors. Passing file descriptors is used in the context of isolated environments.

A large part of the scripts for launching the daemons and of the daemon configuration files is generated using the code from NixOS GNU/Linux distribution.

We do not currently use any service supervision system, mostly because restricting the global interactions leads to a limited set of system-wide services with failure modes that are invisible to a service supervision system anyway.

A small library is provided for writing both the system-wide policy code and the user-level policy code. We will now describe the provided functionality.

7 OTHER RELATED FUNCTIONALITY

7.1 Authentication and authorisation

Request authorisation by the physically present user supports some extra features.

The client can provide extra text to explain to the user why the request has been sent. Sometimes authorising the request will require the user to enter a password, either their own, or belonging to a specified other user on the system (usually the `root` user). The password request and entry will happen in the same reserved virtual terminal to prevent abuse of input interception in the X11 protocol. The same mechanism can be used by a client process to request an application-specific password from the user without the password going through the normal input channels of the current session.

7.2 Isolated environments

Our protocol for requests supports forwarding some status messages to the client as they happen without waiting for the entire request to finish. For example, when starting a command in an isolated environment, it is possible to have the standard output (or the exit code of the program) as a return value, but forward the error messages as they appear. This is based on sending the file descriptors through the socket and thus requires the socket to be a Unix domain socket as opposed to a TCP socket. The same functionality also allows to run a command-line program in an isolated environment but provide it with the input from the current pseudo-terminal.

While the client process has a lot of freedom in configuring what files are accessible inside the sandbox, it may be sensible to impose some limitations. For example the following policy says that an empty `tmpfs` (temporary memory-based filesystem) can be mounted wherever the requesting process wishes, but real files and directories can either be provided at their true location, or inside `/tmp/` or `/home/`, or at a location without any well-known meaning. This restricts the possibilities for requesting access to an installed setuid program inside the sandbox and confusing the program with fake entries inside /etc/.

```
(defun nsjail-mount-allowed-p (from to type)
  (or
    (equal type "T") (equal from to)
    (and
      (or
        (alexandria:starts-with-subseq "/home/" from)
        (alexandria:starts-with-subseq "/tmp/" from))
      (or
        (alexandria:starts-with-subseq "/home/" to)
        (alexandria:starts-with-subseq "/tmp/" to)
        (equal "/tmp" to)
        (not (find
           (second (cl-ppcre:split "/" to))
          *well-known-directories* :test 'equalp))))))
```

On the current system the risks of abusing setuid binaries are further reduced by having only 4 setuid binaries. There is some separate support for generating safe contents for specific files inside /etc/ such as /etc/passwd (but not /etc/shadow).

7.3 Miscellaneous Lisp utilities

For many system administration tasks, the simplest way to perform them on GNU/Linux involves invoking utility programs and parsing their output. We provide some tools to simplify manipulating shell commands, and tool-specific wrappers for some commands.

We provide a function to parse the network address list returned by the ip network configuration tool, as well as a few trivial wrappers for the most common network configuration modifications, including WiFi state control via wpa_supplicant. We currently assume that the configuration of wpa_supplicant is managed separately.

We wrap some of the basic Nix package management operations, basic user account manipulation operations, power management etc.

In the spirit of the standard with-open-file macro and multiple with-. . . macros provided by Lisp libraries, we implement shell-with-mounted-devices macro that attaches a file system or multiple file systems on removable devices, launches a specified shell command (by default it opens a new shell window in the current screen session) and detaches the file systems when the command finishes. Many graphical environments based on GNU/Linux offer graphical shortcuts to attach a file system on the USB drive and to launch a file manager or a terminal for that file system, but we are not aware of environments that provide an option to tie automatic disconnection of a file system with the lifetime of the corresponding terminal or file manager.

Of course, the functionality provided for lifetime tracking of a non-blocking command can be used for implementing other related functionality, such as deactivating the screen saver for the lifetime of a video player when the video player doesn't implement screen saver interaction on its own. From the low-level point of view, the lifetime tracking of the child process is based on letting it inherit an open file descriptor; in most cases the file descriptor will be inherited by any descendant processes, which we consider a desirable behaviour.

7.4 Tracking process windows with StumpWM

We also provide the automatic-tagging functionality for unsandboxed applications, however this only works if the initially started process is the one creating windows. This restriction is absent when isolation is used for launching the application, because we can assign a unique hostname to the container and the application will not change it. Note that the hostname and process ID are reported by the application who can deliberately violate the protocol, but most applications use some GUI libraries that reveal the data correctly.

8 COMPARISON TO OTHER APPROACHES

8.1 Similarities and differences with Guix-SD goals

While Guix and Guix-SD also use a Lisp (namely, Guile Scheme) to configure the system, our goals are different. Guix and Guix-SD are intended to provide Lisp-based management and Lisp APIs for constructing the system. GNU Shepherd manages the system in the runtime, but it is intended for managing only the set of running daemons, and for use via command-line. It also provides APIs that mirror the command line invocations.

In our case, the goal is to experiment with controlling the runtime state of a system and with providing Lisp APIs for runtime fine-tuning as opposed to building a preconfigured system once. An example of a difference is access to privileged commands. Guix-SD uses the standard sudo mechanism and allows to specify the contents of the /etc/sudoers configuration file. Our system implements a custom protocol for requesting privileged operations, and the access policy is configured by defining Lisp functions.

Guix could have been used instead of Nix for package management; the choice of Nix here was motivated by package availability.

8.2 Request authentication and authorisation

A custom protocol for checking the UID of the other process is providing functionality pretty similar to the getsockopt system call with SO_PEERCRED option. However, we verify the current user ID of the client. In contrast, getsockopt checks the identity of the process that has initially established the connection; this could be, for example, the parent process of the current one that did not close the socket before spawning the current process. The behaviour we use is more similar to the behaviour achieved when using sudo. (As an aside, iolib does not have full support for SO_PEERCRED.)

Our protocol allows authorising multiple operations as a batch. In this case the user will need to take an action only once, but full information about all the actions will still be provided. This differs from the typical situation with sudo where either each operation requires separate authorisation, or authenticating a session leads to all requests in the next few minutes are granted.

In many cases, the physical presence of a user is relevant: for example, a user with physical access can shutdown the system using the power button, so physical presence can be enough to initiate safe shutdown; but the same user logged in via SSH might not want to be able to shutdown the system too easily. Our solution seems to be more robust than the usual ones when a user has both SSH and physical login session and also has some programs running in a pseudo-terminal accessed from both of the sessions (e.g. using `screen`). In such a setup the standard policy mechanisms of checking whether a user has any physical session, or whether the current program is a descendant of physical login do not answer the question whether the request comes from a physically present user.

8.3 Isolated environments

The goals of the system include running software in isolated environments. Linux namespaces support permits to construct some isolated environments even without administrator access. It is still desirable to assign single-use user IDs to different isolated environments as an extra layer of protection. However, for things like network isolation it does not matter whether the namespace has been constructed as root or as user. This allows us to run a relay for limited network access in an environment that is almost fully isolated save for the networking, then create a nested sandbox with access to the relay socket but not the host networking (figure 1).

Thus it would be convenient to have a tool with nested isolation support. This makes service-oriented isolation options, including Shepherd service sandboxing, less attractive. An intermediate option would be something like `systemd-nspawn`, but it requires that the entire system is managed by `systemd`.

Standalone options range from pretty limited `unshare` to fully featured options such as Firejail, `nsjail`, Bubblewrap, and others.

We use `nsjail` [9] tool for the low-level work necessary for creation of isolated environments, as it supports many various options and seems stable. For example, `nsjail` supports presenting specified parts of filesystem inside the isolated environment, choosing whether to use or not each kind of isolation supported by the Linux kernel, resource limits, user and group ID handling, etc. As the invocations of `nsjail` are generated by the Common Lisp code, flexibility and feature availability are more important than convenience of manual invocations of availability of predefined isolation profiles.

8.4 Integration with shell scripts

Of course, it is possible to reimplement the authentication protocol in the language of the script. On the other hand, maintaining implementations of the protocol in multiple programming languages increases the cost of updating the protocol.

Another approach is to have a separate management daemon running with the user's permissions, and use file system access control to allow requests to such a daemon only from the client programs with the same user ID. The user daemon will execute the requests. It can have access to both the user Lisp scripts not requiring special permissions and to sending a request to the system management daemon.

A benefit of the chosen approach with starting a saved Lisp image is the possibility to see the parent process ID. Another benefit, which is only relevant for scripts running in a terminal, is that the Lisp image providing the communication with the system management daemon can write to the terminal. A possible drawback is a higher memory use, although most of the data in a saved Lisp image is mapped read-only from the image on the disk and can be shared between multiple process.

8.5 Package management

Most system-wide package managers such as `dpkg` and `rpm` typically contain some notion of package conflicts, leading to annoying interactions between the parts that we would prefer to keep separated and not interacting.

To have a solid foundation for managing the runtime state, we choose a tool for managing the packages that minimises unwanted interactions such as package conflicts, and ensure a requested state regardless of the previous state. If we want these properties to apply to the base system, the most natural options are Nix [8] and GNU Guix [3]. Both the Nix package manager and the Guix package manager inspired by Nix restrict the notion of package conflicts by installing each package into its own directory. That way, installation of a package is not seen as a change in the globally shared mutable system state; instead, the installed packages are treated as a garbage-collected pool of immutable data structures. Updating the system with this approach is closer to constructing a new list out of cons cells (reusing some of list elements), than to updating the elements in-place in a sequence.

Both package managers have distributions based on them, NixOS and GuixSD; in this case the entire system is defined as a package with some dependencies, allowing to keep multiple versions of the entire system and choose between them on boot without unnecessary duplication of the installed packages. This allows both to install individual packages and to reuse parts of the system (e.g. configuration files and service definitions) without running the entire distribution.

Of course some persistent data such as PostgreSQL databases needs to be converted to newer formats during some upgrades, so the abstraction is not perfect. However, the cost and the risk of a typical update of a system with some exotic packages installed are lowered significantly.

We use the Nix [8] package manager. While Guix, which has started as a rewrite of Nix into Guile, seems to be a natural choice for a package manager in a Lisp OS, we currently use Nix [8] because of a larger packaging community. In particular, Nixpkgs have better Common Lisp package coverage than GuixPkgs. However, we use only some parts of NixOS, because NixOS uses `systemd`.

8.6 Communication with StumpWM

There are alternative ways to send commands to StumpWM. One of them relies on setting the window properties of the root window. Unfortunately, this approach has no access control and cannot handle concurrent commands. The generic SWANK [14] debugging interface is often used for a similar purpose. However, it uses a local network port that also can be accessed by all the applications granted unconstrained network access.

8.7 Choice of implementation language

The overall design of the system and the supporting tools do not require specifically Common Lisp or a Lisp-family language. However, many languages in the Lisp family possess all the strong sides we consider required for comfortable use in such a setup, such as REPL and macro support.

While scsh [10] would be a natural choice for migrating Shell workflows into a REPL with a more powerful language, Common

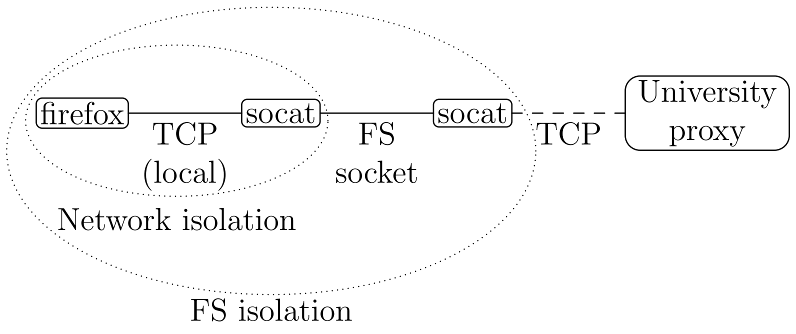

Figure 1: Providing limited network access inside isolated environment

Lisp library availability provides an advantage for some parts of the system, as libraries such as SQLite3 FFI are readily available. Another language option could be Julia [11], which has macro support and REPL, as well as solid FFI support and a very active ecosystem. However, a higher level of activity comes with a drawback as breaking changes happen more often. A tangible benefit of the Common Lisp ecosystem is the tendency to expand the tools more often than to change them with an impact on existing use.

In any case, we want to avoid a system design that would lock in too many decisions. We want to make gradual migration between multiple different system management daemons reasonably easy, including the case when they are written in different languages (maybe even outside the Lisp family). In particular, we want to allow running two system management daemons simultaneously and making requests to both of them. Moreover, it would be nice to allow the daemons to have overlapping areas of control.

Note the system also has some small utility programs implemented in C specifically to be invoked from Common Lisp code. Such programs are useful when it is desirable to invoke low-level system functionality from a separate process, not just a separate thread, for example due to thread-safety concerns.

9 FUTURE DIRECTIONS

There are many things that need to be wrapped in a Lisp API; too many to attempt listing them.

We will introduce support for more authentication checks, such as a protocol for confirming the process ID of the program that makes a request and looking up the corresponding executable. On the one hand, this requires adding support for `getsockopt` with `SO_PEERCRED` to some FFI library; on the other hand, the race conditions need to be evaluated to avoid the case where a program sends a request and immediately uses `exec` to replace itself with a different executable. This can be useful to provide some reliable context.

Implementing service supervision could be useful for some cases.

A modern computing environment often consists of multiple interacting devices. Our model already includes multiple communicating agents without complete trust between them. It would be interesting to try managing a multi-device environment using interacting policy daemons.

An interface to prevent a denial of service by a misbehaving program issuing too many requests requiring user confirmation is needed.

Some alternative interfaces based on JSON or XML could be provided for easier integration with client code in different languages.

Providing an option to use lightweight VMs instead of containers, probably reusing some of the SpectrumOS [15] work, would expand the options for isolation.

Two daemons could be used to make it easier to fix broken updates without reboot and without logging in as root.

10 CONCLUSIONS AND LESSONS LEARNED

Using a system management daemon running Lisp policy code is a simple way for gradually increasing the part of the environment managed in runtime via Lisp, reducing the amount of necessary upfront work. Unlike existing solutions, authorisation policies for privileged system reconfiguration actions are written in a high-level general-purpose programming language (in our case, Common Lisp). Another atypical feature naturally arising from our use of an extensible protocol is the possibility to authorise multiple related actions via a single user interaction without loss of transparency. Tools developed for isolation and testing can be reused by such a system for reducing unwanted interaction between the non-Lisp system parts. We think such an approach will be an integral part of a realistic project to regain some of the benefits of Lisp OS on the modern hardware without losing the ability to interact with the full range of tools and formats considered portable.

In general we observe that the optimistic expectations about the effort of making a system managed by Lisp code the day-to-day system were not too far from truth.

We observe that implementing the system-level and user-level policies in Lisp immediately suggests convenient system-level functionality in the spirit of idiomatic Lisp code that is usually overlooked in other environments. An example is the use of `with-...` macros.

In a somewhat disappointing way, it turned out that for many tasks the author finds Shell workflows running Common Lisp where needed more convenient than alternative workflows based on the Common Lisp REPL.

The current code of the system is available at https://github.com/7c6f434c/lang-os.

ACKNOWLEDGMENTS

The author would also like to thank the anonymous referees for their valuable comments and helpful suggestions on presentation.

REFERENCES

[1] GNU Emacs project page. Retrieved on 16 February 2018. https://www.gnu.org/software/emacs
[2] StumpWM project page. Retrieved on 16 February 2018. https://github.com/stumpwm/stumpwm
[3] Ludovic Courtès. Functional Package Management with Guix. European Lisp Symposium 2013, Madrid, Spain. Retrieved on 16 February 2018. https://arxiv.org/abs/1305.4584
[4] GNU Shepherd project page. Retrieved on 07 March 2021. https://www.gnu.org/software/shepherd
[5] Mezzano project page. Retrieved on 16 February 2018. https://github.com/froggey/Mezzano
[6] Movitz project page. Retrieved on 16 February 2018. https://www.common-lisp.net/project/movitz/
[7] LOSAK project page. Retrieved on 18 February 2018. http://losak.sourceforge.net/
[8] Eelco Dolstra, Merijn De Jonge, Eelco Visser. Nix: A Safe and Policy-free System for Software Deployment. Large Installation System Administration Conference 2004. Retrieved on 16 February 2018. http://nixos.org/~Eeelco/pubs/nspfssd-lisa2004-final.pdf
[9] nsjail project page. Retrieved on 07 March 2021. https://github.com/google/nsjail
[10] Olin Shivers, Brian D. Carlstrom, Martin Gasbichler, and Mike Sperber. Scsh Reference Manual Retrieved on 07 March 2021. https://scsh.net/docu/html/man.html
[11] Jeff Bezanson, Stefan Karpinski, Viral B. Shah, Alan Edelman. Julia: A Fast Dynamic Language for Technical Computing. Retrieved on 07 March 2021. https://arxiv.org/abs/1209.5145
[12] cl-html5-parser project page. Retrieved on 07 March 2021. https://github.com/rotatef/cl-html5-parser
[13] Thoughtful Theridion project page. Retrieved on 07 March 2021. https://gitlab.common-lisp.net/mraskin/thoughtful-theridion
[14] Superior Lisp Interaction Mode for Emacs project page. (Includes Swank debugging protocol implementation) Retrieved on 07 March 2021. https://common-lisp.net/project/slime/
[15] SpectrumOS project page. Retrieved on 22 March 2021. https://spectrum-os.org/