

Redox OS Service Monitor

JIF 4328

Charles Tomas Ordonez, Charles Phillips, Devan Gandhi, Julianna
Cutter, Matthew McCollum, Donald Nelson

Client: Redox OS

Repository: <https://github.com/CharliePhillips/JIF-4328>

Table of Contents

Table of Figures	3
Terminology.....	4
Introduction.....	6
Background	6
Document Summary	6
System Architecture	7
Introduction.....	7
Static System Architecture	8
Diagram	8
Description	9
Dynamic System Architecture	10
Diagram	10
Description	11
Component Detailed Design	12
Introduction.....	12
Static Elements	12
Dynamic Elements.....	14
Data Storage Design	16
Introduction.....	16
Database Use	16
File Use	17
Data Exchange.....	18
UI Design	19
Introduction.....	19
The Command Line	19
Graphical UI.....	21
Appendix A – CLI Commands	22
Appendix B – Team Member Information	23

Table of Figures

Figure 1 - Static System Diagram	8
Figure 2 - Dynamic System Diagram.....	10
Figure 3 - Static Component Diagram	13
Figure 4 - Dynamic Component Diagram	15
Figure 5 - File Use Diagram.....	17
Figure 6 - Command Line Interface	19
Figure 7 - Services Registry Help.....	20
Figure 8 - List UI.....	21

Terminology

CLI: Command Line Interface. A text-based interface that allows users to interact with the computer by typing commands in a terminal/console.

COSMIC: Cross platform GUI library written in rust. Primarily for use with the COSMIC desktop environment, which is still in alpha. The GUI library itself is also still under heavy development.

File Descriptor: An integer representing a stream of bytes that could be from a file, device, network, etc. It is tracked and handled by the **kernel** as a key abstraction in many operating systems.

GUI: Graphical User Interface. Allows users to more easily interact with the computer using visual elements like icons, buttons, and menus.

Kernel: Handles the core functions of any operating system, including process scheduling and memory. Monolithic kernels tend to have drivers and other services built into them, while Microkernels aim to leave as much as possible up to user programs for modularity.

Mutex: A common programming tool used to ensure memory safety between multiple threads of execution running at the same time. If one thread of execution is attempting to change the value of variable X and another is trying to read the value of X at the same time, a mutex ensures that one operation completes before the other begins. Mutex is short for mutual exclusion.

Redox OS Kernel: The “core” of Redox OS is responsible for scheduling programs, handling system calls, and other crucial tasks. Redox OS uses a fully Microkernel architecture, unlike other major operating systems.

Rust: A general-purpose programming language focused on performance and memory safety.

Scheme: The storage of data for a service, accessible through system calls as a file descriptor. The service a scheme belongs to can access it directly.

Service/Daemon: A service or daemon (interchangeable) is a program that runs in a constant loop in the background to manage a particular part of the system. For example, a device driver or network functionality would be implemented as a service/daemon.

Signal: Standard for inter-process communication. More detailed information can be found within the Linux manual: <https://man7.org/linux/man-pages/man7/signal.7.html>

Syscall: Operating system abstraction that gives programmers access to critical system functions managed by the kernel like memory allocation and device access. As an example, C's infamous 'malloc' function invokes either the 'brk' or 'mmap' syscall to obtain memory.

Introduction

Background

Our application consists of a command line tool and system service designed to monitor, log, and start/stop services inside of Redox OS to maintain stability and usability. It runs in the background, fetching information from the various services running to monitor them. It provides access to logs, starting and stopping services, and a variety of other details to monitor the stability of services throughout the system. New services can be added to the registry, so that they can be automatically started and monitored on system boot. Additionally, it can handle simple cases of restarting services on its own, hopefully preventing manual intervention from the user. Our application is built fully in Rust and includes various Rust libraries. We also plan to implement a desktop-app style GUI for the tool, allowing a user less familiar with the command line to still utilize our tool.

Document Summary

System Architecture: This section provides a high-level view of how a major piece of our system – the Service Monitor – interacts with the Redox OS Kernel, other services, and the command line. This includes static interaction as well as dynamic interaction through the stop command.

Component Detailed Design: This section describes the components of our system and interactions in more detail through a static component diagram and a dynamic interaction diagram.

Data Storage Design: This section describes how we store data in our application. Our only long-term data storage is the ‘smregistry.toml’ file, which stores information on services that our Service Monitor will start and manage.

UI Design: This section describes and shows the design of the graphical UI for our program. This is a layer on top of the command line interface, that provides a more polished and user-friendly interface for interaction.

System Architecture

Introduction

Our application consists of a command line tool and system service running natively in Redox OS. It is a service on its own, that will have the ability to control and communicate with other services in the kernel. This architecture fits in with the pre-existing state that Redox OS was developed for, as it is a fully microkernel architecture. Within our service, other services can be registered and tracked. The Service Monitor will continuously collect various statistics from each registered service to monitor their stability; more specifically, it will rely on these statistics to diagnose errors for each service consistently and accurately.

In the diagrams below, we will present a static look at our system architecture (Figure 1) from a high-level view, as well as a showcase of how the stop operation would work within our service (Figure 2). These showcase the underlying architecture of how the Service Monitor will operate for most functions. Generally, the Service Monitor will exchange data with its client services through the kernel using system calls. Any applications wishing to interact with the Service Monitor or its services will have access through the Service Monitor only.

Rationale

This leaves room for our command line application, GUI application, and future Redox applications to interact with the Service Monitor in a standardized way. Our design leverages Redox's existing system-call interface for detailed inter-process communication. In all, our application runs completely locally on the user's computer and lacks any private or vulnerable data. The data that it does collect is counts of read/write calls, not the contents of those calls. Most data collected is not stored outside of system logs, which are already available to a normal user. As such, there are no concerns over the security of the data collected at this time.

Static System Architecture

The Service Monitor starts its managed services and communicates with them through the kernel. The CLI app can be used to manually control the behavior of the Service Monitor. The diagram below demonstrates how the Service Monitor and its managed services (example services 1 and X) interact with each other and with the rest of the system.

Diagram

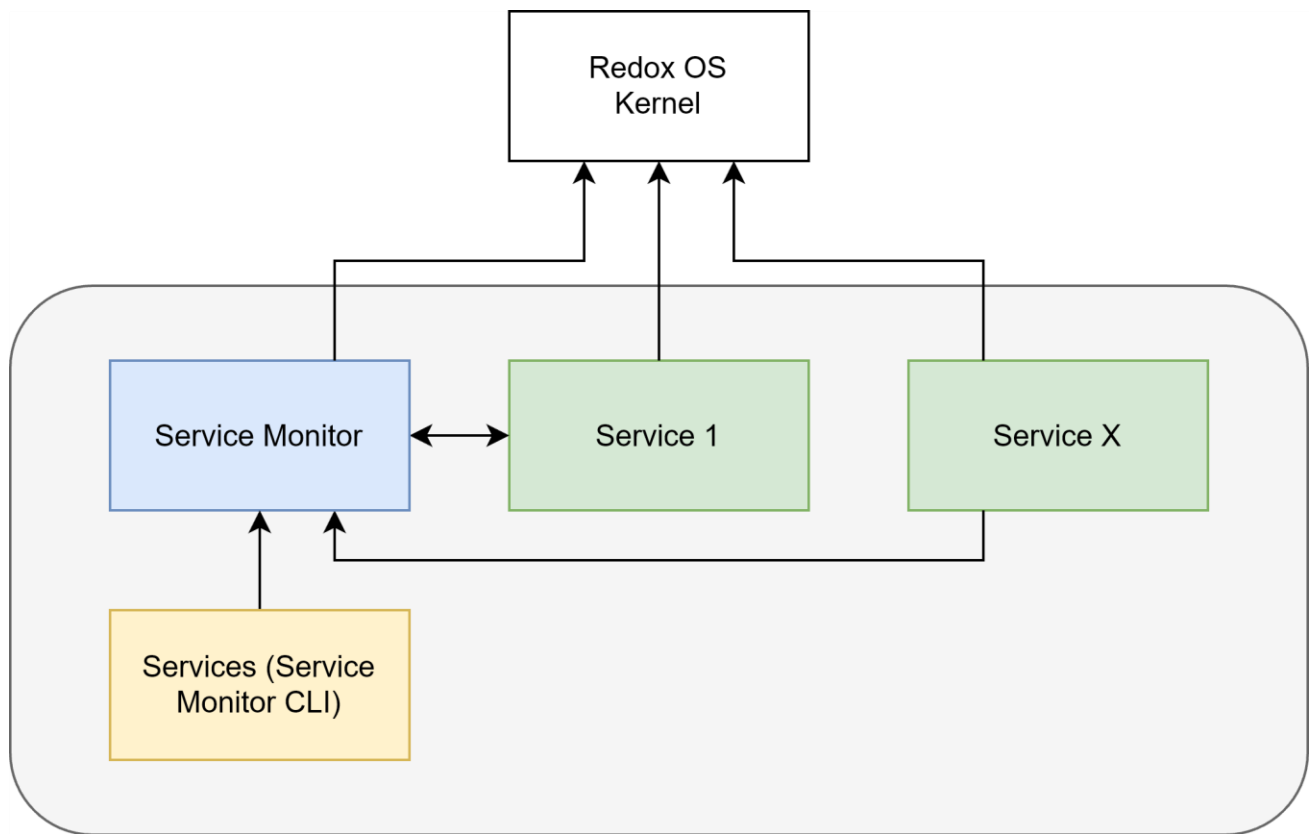


Figure 1 - Static System Diagram

Description

The first step in booting Redox OS is for the kernel to boot our service (the Service Monitor). After that, our service can boot the rest of the services on the system through the system call interface (read/write). Since those additional services are reliant on the kernel for communication, that is reflected in our diagram.

The Service Monitor Command Line Interface (CLI) is our form of user input. Using the command line, the user can instruct our application to control, log, and collect data on services. Because of the current userbase of Redox OS, we can rely on their technical knowledge and trust that a command line is approachable for them. However, we recognize that for Redox OS to expand its reach, we also need a desktop-friendly GUI. That will act in a similar manner to the command line.

Each service started by our Service Monitor will be logged and have various information tracked on its status. This could include uptime, time to init (initialize), total number of syscalls, Scheme size, or last response time. Collecting this data on each service is a core function of our monitor and is information that each service would need to generate to function. We know every service will generate this information, and if they don't, that could be a critical error. From there, the Service Monitor could attempt to kill that service and restart it or take other actions to recover that piece of software. The Service Monitor will have some information that it uses to start and control services and must communicate with the kernel for additional information under certain circumstances.

Dynamic System Architecture

The user issues commands to the CLI app which parses them and relays them to the Service Monitor. The Service Monitor takes the command and performs the appropriate action returning information from the results of this action.

Diagram

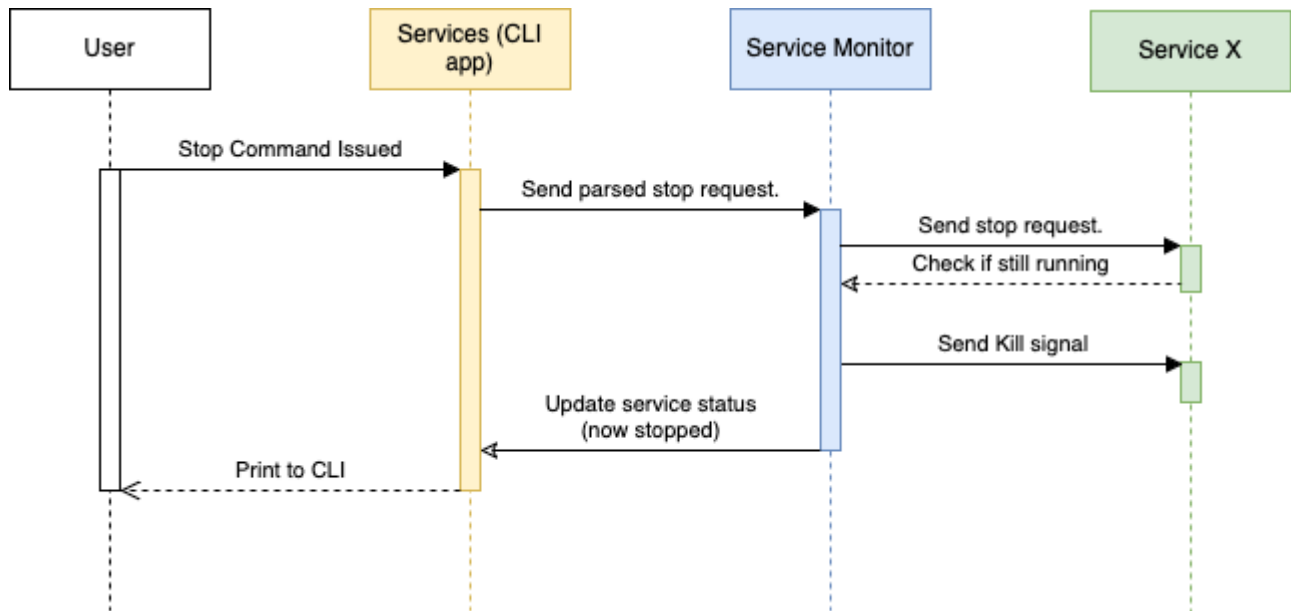


Figure 2 - Dynamic System Diagram

Description

The dynamic architecture diagram depicts the flow through the system when the user issues a stop command to a specific service. It shows the key interactions between each portion of the process and what each piece sends to the next. The Redox kernel is responsible for facilitating communication between each process. In Figure 2, the arrows represent the kernel facilitating communication via the ‘read’, ‘write’, or ‘kill’ system calls. In the future the services CLI could be replaced with a GUI that makes the same or similar syscalls but displays the results in a window instead of a terminal. This also leaves room for future applications to interface with the Service Monitor to perform more complex commands.

Upon the User’s issuance of a stop command for a specific service, that command is presented to the CLI app. The CLI portion will then take that request in and relay it to the Service Monitor itself. This allows the user to rely on human readable commands to accomplish their goal. The CLI passes the parsed stop request to the Service Monitor.

When the Service Monitor receives a parsed stop request, it can identify which service it is targeting. The Service Monitor then sends out a stop request to the service, waits for a set amount of time, and receives info back on from the service to tell if it is still running. If it is still running, the system monitor will follow up with a kill signal, forcing the service to terminate. The first stop is not a force kill to allow the service to terminate safely if needed, but for services that are not responding, a stronger signal is needed (the kill signal). The service does not return anything after being forcefully killed. After the completion of the Service Monitor’s routine, it updates the status of the service and sends that information back to the command line application.

After receiving that update, the command line app prints the new status back to the main command line for the user to see. This completes the stop instruction.

Component Detailed Design

Introduction

Figures 3 and 4 show the static and dynamic levels of the components in our system. There are 3 main components to our system: the Service Monitor API, the Service Monitor daemon, and the items interacted with by the service monitor, such as various registry files or schemes that store data and other services throughout the system. Each component maintains conceptual integrity as they are shown in different contexts, such as System Architecture previously, at varying levels of abstraction.

Static Elements

Our static component diagram (figure 3) shows which individual components communicate with and contain each other. The BaseScheme is in our application logic because it serves as the primary interface between the actual Service Monitor daemon and each of the services registered to it. That BaseScheme defines all the data to be collected from each service and in what format it should be read. Within BaseScheme, we have the ManagementSubScheme, a generic-type wrapper that allows for the different sub-schemes of BaseScheme to be accessed in a thread-safe manner (using mutexes). This also allows for the actual service scheme to be accessible to the service monitor, leaving room for future development. Additionally, the Service Monitor relies on a “registry processor” module to define and manage its list of services. This module contains the code responsible for maintaining the ‘smregistry.toml’ file.

The Service Monitor API clients, the CLI and GUI, access the Service Monitor API and allow users to manually interact with the Service Monitor. In addition, we have “Other Redox OS Components;” with continuing development of Redox OS, there could be other components that work in tandem with our application. Designing the application to work well with these future components is an important consideration.

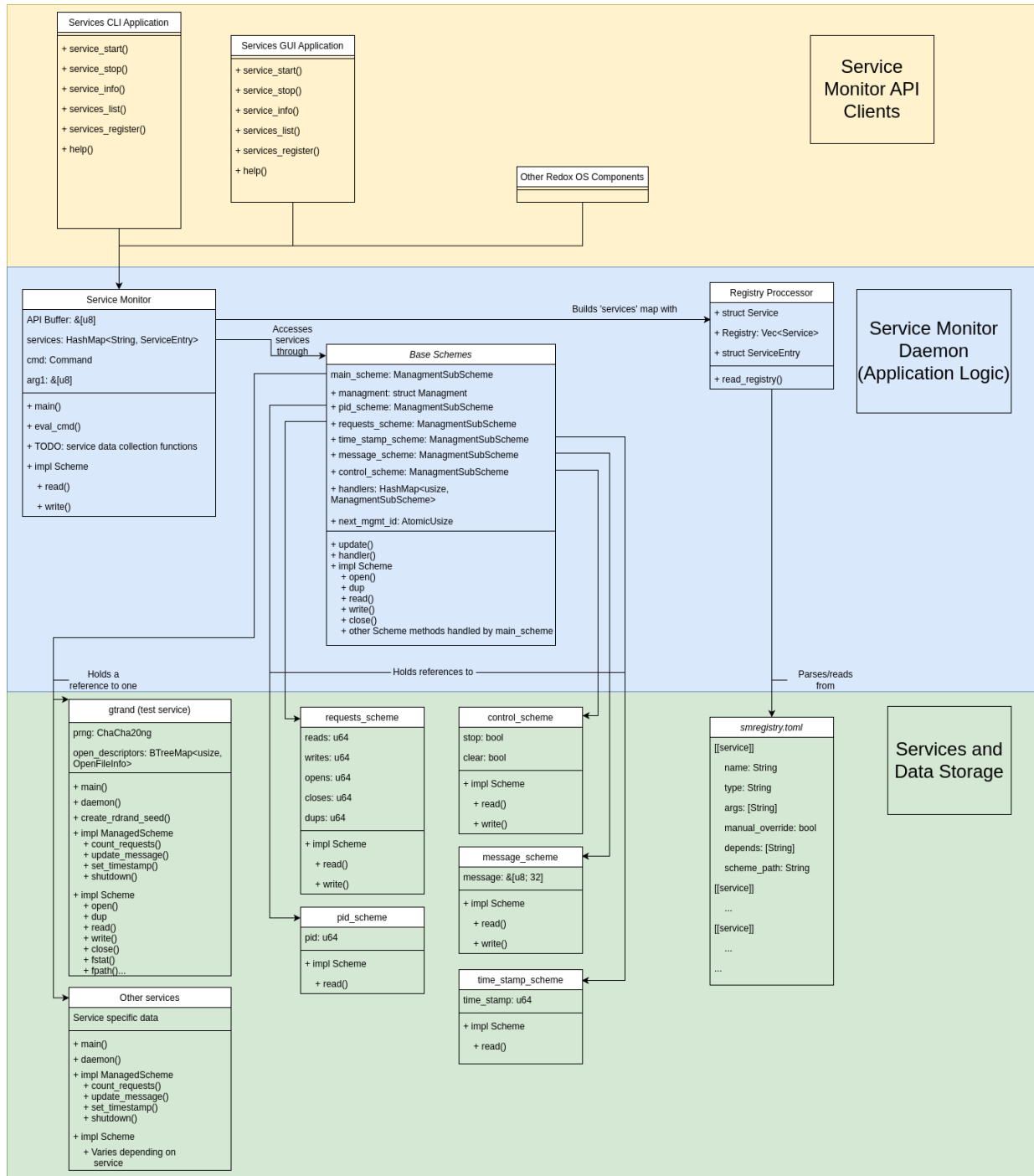


Figure 3 - Static Component Diagram

Dynamic Elements

The dynamic interaction diagram for the application shows an example of when the list and stop commands would be executed to identify a service having problems and stop it. This specific instance is of a user performing the interactions, but this could easily be extended to the Service Monitor performing the same task. Upon request from the user, the Service Monitor opens the main scheme and requests/receives relevant data from the respective service. After completing the collection of data from a service, the scheme is closed, and the monitor moves onto the next service. Once the user has had time to view the services and data, they can stop the service if needed. This is a three-step process involving the Service Monitor requesting a stop, checking if the service stopped, and sending a kill signal to force a stop if needed. The user will receive an indication when the stop is complete. The process to start a service in the application is not included in the diagram but is relatively similar. It would involve reading from the service registry, starting the service, and checking if the service started successfully by getting the process id. Note that there is never a separate call to any sort of data storage as the services themselves store the data we need.

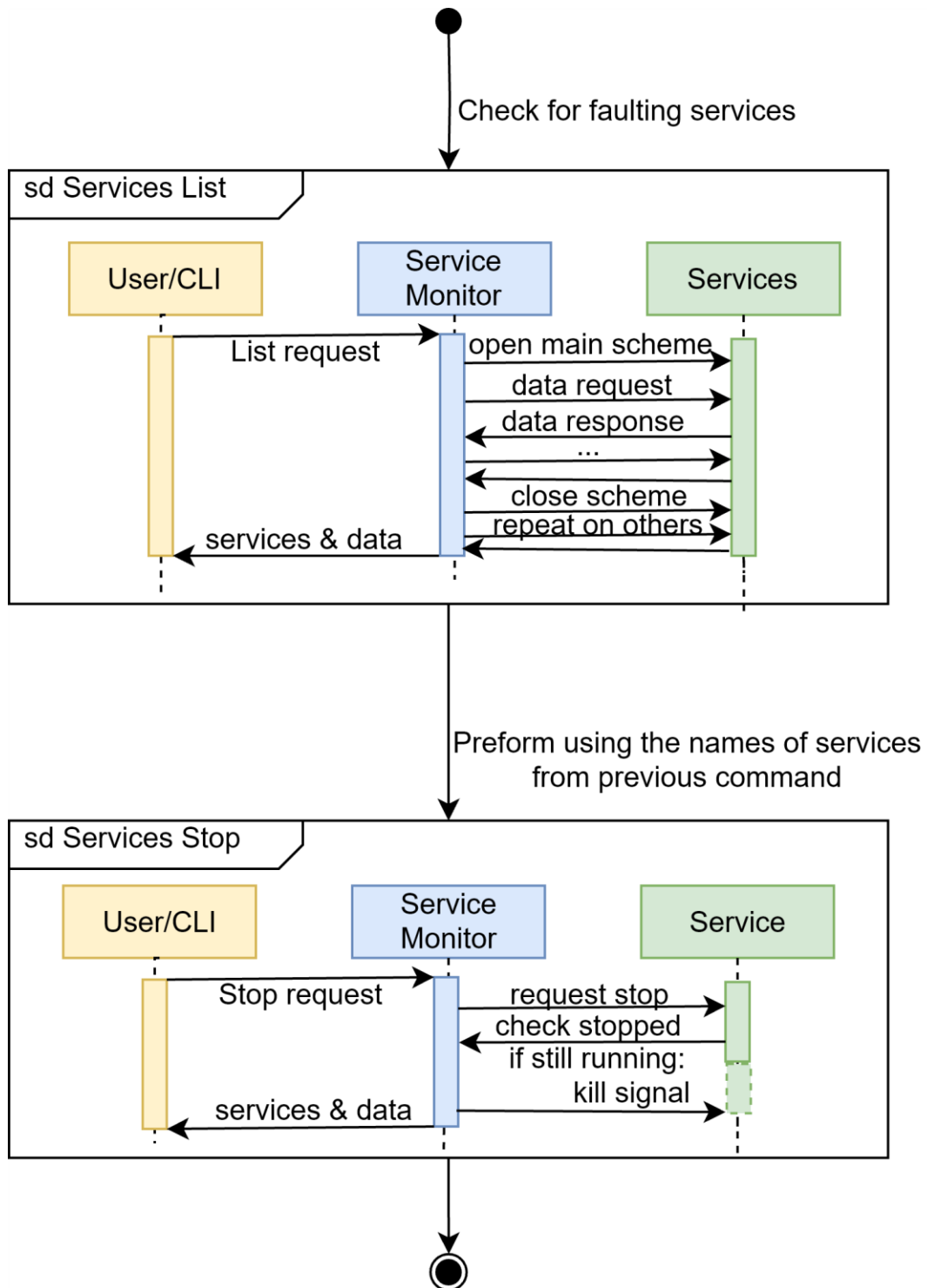


Figure 4 - Dynamic Component Diagram

Data Storage Design

Introduction

The diagram below shows how we use the 'smregistry.toml' file within the Service Monitor for storing information on all the services that the monitor will start and manage. This includes information on the service's name, type, arguments, dependencies, and the path to its scheme. This is the only long-term storage for our program, and the rest of the data collected by our program does not persist on restart.

Database Use

We do not use an SQL or NoSQL-based database in our application. Our registry is stored in a .toml file in Redox's file system. The Service Monitor does track an additional set of data for each service, but it is not necessary to retain this data after rebooting.

File Use

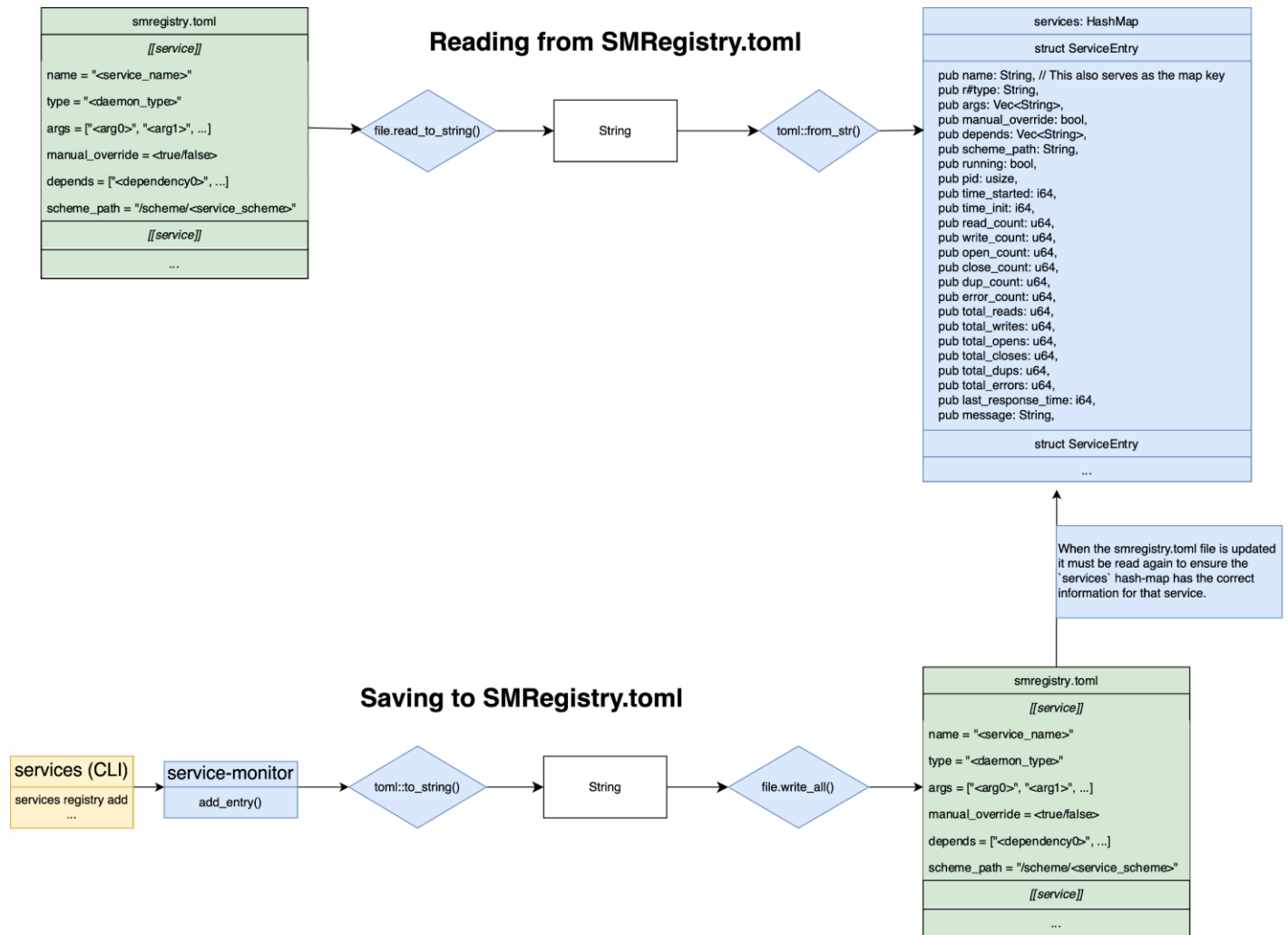


Figure 5 - File Use Diagram

The 'smregistry.toml' is read when the service-monitor starts, this file contains the information necessary for starting and managing each listed service. The user can also read the registry entry for a service with the 'services registry view...' command.

When the user wants to add an entry in the registry, they issue the 'services registry add ...' command in the terminal. This tells the service-monitor to execute the function responsible for the arguments passed. The arguments passed through the command line are then written to 'smregistry.toml'. The file is then read to ensure the service-monitor's data (the 'services' hash-map) is up-to-date. The service's information in the 'smregistry.toml' file cannot be removed or modified while it is running.

Our client asked us specifically to use a .toml file instead of a database. TOML (Tom's Obvious, Minimal Language) is a markup language designed to be easily readable and easily mapped to a hash-table. TOML has similar use cases to JSON and is supported by many popular languages. This file is intended to be stored within the OS and modified primarily through the command line, although careful manual edits could occur as well.

Data Exchange

Our application does not involve the physical transfer of data between different devices. It only monitors the current system's services and does not transmit data outside of the system it is running on.

Within our application, there are little to no security concerns surrounding the transfer of data. The data being transferred is non-sensitive (application statistics) info that would already be available to any user on the system, and there is no separate login required for using our application. There are no privacy or security concerns with the data we are exchanging, and outside of the 'smregistry.toml' file, none of it persists on restart.

All data that is exchanged with our application (service to application or application to service) is done through the system call interface, which is part of the OS's kernel.

UI Design

Introduction

In this section, we will present both user interfaces for the Service Monitor. Users can use either to interact with the monitor, getting the same info and having the same options for starting and stopping services. Our program has two distinct interfaces for interaction to give the most usability possible to a wider range of users. For those who may not be as familiar with the command line, the graphical interface will be their best option for starting and stopping services. For the more technical users who know what they need, they can use the command line to quickly access that information.

The primary way the current users of Redox OS will interact with the Service Monitor is through the command line. As the OS is in development, its audience is almost exclusively technically skilled developers who want to contribute to the project. To this userbase, the command line is a standard usage method.

The Command Line

```
user:~# services --help
The command-line interface for the Redox service monitor

Usage: services <COMMAND>

Commands:
  start    Start a service
  stop     Stop a service
  list     List all services and their respective statuses
  clear    Clear short-term stats for a service
  info     Get info about a service
  registry Change and view the registry. Try 'services registry --help' for more information

Options:
  -h, --help    Print help
  -V, --version  Print version
user:~# services start gtrand
Unable to start 'gtrand': Already running
user:~# services stop gtrand
Stopped service 'gtrand'
user:~# services info gtrand
Service:      gtrand
Message:
Total READ count: 0
Total WRITE count: 0
Total OPEN count: 2
Total CLOSE count: 1
Total ERROR count: 0
user:~# services list


| Name    | PID  | Uptime                    | Message   | Status      |
|---------|------|---------------------------|-----------|-------------|
| gtrand  | None | None                      | None      | Not running |
| gtrand2 | 76   | 1 minutes, 01.888 seconds | starting! | Running     |


user:~#
```

Figure 6 - Command Line Interface

Depending on the command the user runs, they will receive a response from the monitor containing information on the services and commands in the program. They have the freedom to run whatever commands they wish and can see further info on how to use the command line if needed.

```
user:~# services registry -h
Change and view the registry. Try 'services registry --help' for more information

Usage: services registry <COMMAND>

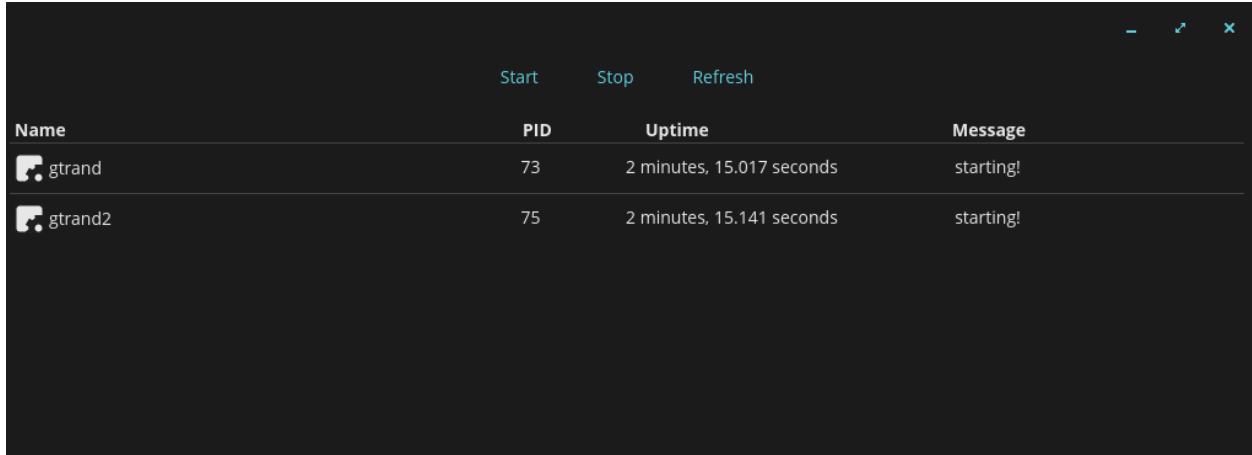
Commands:
  add      Add a service to the registry
  remove   Remove a service's entry from the registry
  view     Print a service's entry in the registry
  edit     Edit a service's entry in the registry

Options:
  -h, --help  Print help
```

Figure 7 - Services Registry Help

While it is difficult to apply usability heuristics to a command line interface, there are some that are relevant to how our users will interact with the system using our commands. First, the commands for our program enable a new level of system stability for the entire OS, and the statistics recorded help diagnose and identify issues with other services tracked by it. The tool also provides feedback to the user as they use the command line, letting them know if any issues occurred in their usage. The presence of both a command line interface and GUI gives two forms of usage for users, providing flexibility and user control over the technical level they wish to engage with. Most command line flags also feature flexibility, with both short and long versions (-h or --help) to better explain their function or provide a quick shortcut to their functionality. This contributes partly to our commands maintaining consistency and standards with other command-line tools, enabling users familiar with other terminal applications to quickly gain an understanding of how to use this version of the program. Help and documentation of service commands is also easily available for users who need it through widely-recognized means such as “--help”.

Graphical UI





Start Stop Refresh			
Name	PID	Uptime	Message
 gtrand	73	2 minutes, 15.017 seconds	starting!
 gtrand2	75	2 minutes, 15.141 seconds	starting!

Figure 8 - List UI

Above is a look at the main screen of our Service Monitor. It incorporates the list of services that the Service Monitor is currently tracking, their information, and buttons to control the services and refresh the data. In this format, the information gathered by the Service Monitor is presented in a more user-friendly way, appealing to those who may not be used to a command line. As of right now, this version of the UI is not a priority for the program as the percentage of the userbase who would use it is minimal. In addition, some of the COSMIC components are not as feature-rich as other more mature GUI libraries; for example, only dark mode is available, and there is no way to grey out or disable the “Start” and “Stop” buttons when no service is selected. This UI does, however, provide an easier way to view important information collected by the program.

While this screen conforms to all the heuristics mentioned in the command line above, there are also some new ones that it brings to the program. First, with all the key information presented on a single screen, users avoid needing to remember service names and IDs, and can instead recognize the information being displayed. The UI also features a minimalist design, avoiding any unnecessary info from being presented to the user and prioritizing key system messages and uptime. Additional info is hidden until the user clicks on a service. Lastly, the UI enables a quick view of the last message sent by the service. This message is intended to be human-readable, allowing any user to understand what is happening with that specific service.

Appendix A – CLI Commands

Command	Description
services list	Lists all daemons with their PID, uptime, status message, and state (running, stopped, restarted)
services info <daemon_name>	Shows detailed status and stats of a specific daemon (uptime, last time to init, total # of requests, scheme size, error count, last response time)
services clear <daemon_name>	Clears short-term stats (requests, errors, message, last response time & timeout) for a specific daemon
services start <daemon_name>	Starts a registered daemon, ensuring dependencies are running or prompting the user
services stop <daemon_name>	Stops a daemon with multiple attempts (gracefully, then forcefully)
services / services --help	Displays the help page with available commands for the Service Monitor
services registry view <daemon_name>	Views the registry entry for a service, or indicates it's not registered
services registry add <--old> <daemon_name> "[arg1', 'arg2'...]" <--override> "[dep1', 'dep2'...]" <scheme_path>	Adds a new service entry to the registry with optional tags -old (type = old daemon) and -override (flag for manual_override), and arguments and dependencies
services registry remove <daemon_name>	Removes the registry entry for the specified service if it exists; does not affect the instance of the service that is currently running, only the entry in the registry
services registry edit <--old> <daemon_name> "[arg1', 'arg2'...]" "[dep1', 'dep2'...]" <scheme_path>	Updates the registry entry for a service with optional tag --old (type = old daemon), arguments, scheme path, and dependencies
services registry / services registry --help	Displays a help page detailing the available commands for changing and viewing the registry

Appendix B – Team Member Information

Name	Contributions	Contact Information
Charles Tomas Ordonez	Registry Re-Reading, List, RFC	cordonez8@gatech.edu
Charles Phillips	BaseScheme API, Automatic Recovery, GUI List, RFC	Charlie_L_phillips@proton.me
Devan Gandhi	Add/View/Remove Registry, List, Design Document, RFC	dgandhi46@gatech.edu
Julianna Cutter	Refactoring/helpers, Edit Registry, GUI Start/Stop, RFC	jcutter7@gatech.edu
Matthew McCollum	Testing Services, Service Monitor Info, Refactoring, RFC	Mmcollum31@gatech.edu
Donald Nelson	Registry Parsing, Refactoring, Command Line/Data Formatting Improvements, RFC	donald.nelson@gatech.edu