University of São Paulo
Institute of Mathematics and Statistics
Bachelor of Computer Science

Integrating the KWorkflow system with the Lore archives

Enhancing the Linux kernel developer interaction with mailing lists

David de Barros Tadokoro

Final Essay

mac 499 — Capstone Project

Supervisor: Paulo Meirelles
Co-supervisor: Rodrigo Siqueira

São Paulo
2023

To all my family and friends.

Acknowledgedments

"Courage need not to be remebered, for it is never forgotten."
— Princess Zelda, The Legend of Zelda: Breath of the Wild

First, I would like to thank Rodrigo Siqueira and Paulo Meirelles for the continuous support given to me throughout the year related to the confection of this capstone project and in various aspects of my academic, professional, and personal life. Likewise, I would like to thank Aquila Macedo, Melissa Wen, Magali Lemes, Rubens Neto, and all the people involved in the KWorkflow community for the invaluable interactions that they provided me. I cannot forget to mention Nelson Lago for helping me structure experiments, fix LaTeX errors, and much more.

I would also like to thank the Instituto of Matemática e Estatística da Universidade de São Paulo (IME-USP) and all its professors, colleagues, and employees for providing me great experiences through these four years in the Bachelor of Computer Science course.

Last but certainly not least, I would like to thank all my family and friends for all the support and companionship given me throughout my life.

Resumo

David de Barros Tadokoro. **Integrando sistema KWorkflow os arquivos do Lore: *Aprimorando a interação do desenvolvedor do kernel Linux com as listas de email***. Monografia (Bacharelado). Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2023.

O *kernel do Linux* é um projeto de software livre bem estabelecido com trinta anos de desenvolvimento. Combinado com utilitários do *Projeto GNU*, o ecossistema *GNU/Linux* é fundamental para operar serviços críticos em nossa sociedade, como a Internet. Com um modelo de contribuição único, o kernel do Linux é desenvolvido colaborativamente por milhares de contribuidores e mantenedores que residem em todo o mundo por meio de correio eletrônico e listas de email. Muitos projetos oferecem soluções para os processos e práticas envolvidos no desenvolvimento do kernel do Linux, e o sistema *KWorkflow* (kw) tem como objetivo fornecer um ambiente unificado que aprimora os fluxos de trabalho desses desenvolvedores. O projeto kw carecia de uma ferramenta que contemplasse as interações dos desenvolvedores com as listas de email para revisão de patches, e o foco deste trabalho foi produzir um novo módulo (com essa funcionalidade) no ambiente kw que aprimorasse esse fluxo de trabalho. O módulo chama-se `patch-hub` e integra o ambiente kw com os *arquivos do Lore* - uma aplicação Web que hospeda arquivos atualizados das listas de email relacionadas ao desenvolvimento do kernel do Linux - para oferecer uma interface de usuário (UI) baseada em terminal para os patchsets enviados para as listas de email. Um *Sistema de Gerenciamento de Banco de Dados* (SGBD) foi integrado ao projeto kw como uma tarefa preliminar, constituindo uma base para o `patch-hub` e impactando positivamente o projeto em outras frentes, como na coleta de dados e no desempenho. Esse novo recurso foi implementada usando o padrão de projeto arquitetural *Model-View-Controller* (MVC) e o modelo matemático de computação *Finite-State Machine* (FSM), o que garantiu componentes fracamente acoplados que são simples de manter e expandir. Ao exibir sequências de menus de terminal para o usuário navegar, o `patch-hub` fornece os serviços de disponibilizar as listas de email arquivadas no Lore, exibir um registro dos patchsets mais recentes enviados para listas de email específicas, fazer consultas baseadas em strings, marcar patchsets para acesso rápido no futuro e aplicar ações sobre eles. O estado atual do módulo integrado ao projeto kw não apenas valida uma ferramenta que aprimora as interações com listas de email, como também proporciona oportunidades para trabalhos futuros que continuarão a aprimorar o `patch-hub` e a experiência do desenvolvedor do kernel do Linux.

Palavras-chave: Kernel Linux. KWorkflow. Arquivos do Lore. lore.kernel.org. Modelo de contribuição. Software livre.

Abstract

David de Barros Tadokoro. **Integrating the KWorkflow system with the Lore archives: *Enhancing the Linux kernel developer interaction with mailing lists***. Capstone Project Report (Bachelor). Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2023.

The *Linux kernel* is a well-established free software project with thirty years of development. Combined with utilities from the *GNU Project*, the *GNU/Linux* ecosystem is fundamental for operating critical services in our society, like the Internet. With a unique contribution model, the Linux kernel is collaboratively developed by thousands of contributors and maintainers who reside worldwide through electronic mail and mailing lists. Many projects offer solutions to the processes and practices involved in the Linux kernel development, and the *KWorkflow* (kw) system aims to provide a unified environment that enhances the workflows of these developers. The kw project lacked a tool that contemplated the developers' interactions with the mailing lists for patch reviewing, and this work's focus was to produce a feature in the kw environment that enhanced this workflow. The feature is named `patch-hub`, and it integrates the kw environment with the *Lore archives* - a Web application hosting updated archives of mailing lists related to the Linux kernel development - to offer a terminal-based user interface (UI) to the patchsets sent to the mailing lists. A *Database Management System* (DBMS) was integrated into the kw project as a preliminary task, constituting a base for `patch-hub` and positively affecting the project on other fronts, such as data collection and performance. The feature was implemented using the *Model-View-Controller* (MVC) architectural design pattern and the *Finite-State Machine* (FSM) mathematical model of computation, which ensured loosely coupled components that are simple to maintain and expand. By displaying sequences of terminal menus for the user to navigate, `patch-hub` provides the services of retrieving the mailing lists archived on Lore, displaying a record of the latest patchsets sent to target mailing lists, querying based on strings, bookmarking patchsets for future quick access, and applying actions on them. The current state of the feature merged in the kw project not only validates a tool that enhances the interactions with mailing lists but also lays the foundation for future works that will keep on enhancing `patch-hub` and the experience of Linux kernel developers.

Keywords: Linux kernel. KWorkflow. Lore archives. lore.kernel.org. Contribution model. Free software.

List of abbreviations

OS	*Operating System*
SCM	*Source Code Management*
DAWS	*Developer Automation Workflow System*
DBMS	*Database Management System*
DDL	*Data Definition Language*
DML	*Data Manipulation Language*
FLOSS	*Free/Libre/Open-Source Software*
CLI	*command-line interface*
SQL	*Structured Query Language*
ERD	*Entity-Relationship Diagram*
MVC	*Model-View-Controller*
UI	*User Interface*
FSM	*Finite-State Machine*
DFSM	*deterministic Finite-State Machine*
NFSM	*non-deterministic Finite-State Machine*
TUI	*terminal user interface*
API	*Application Programming Interface*
IPC	*Inter-Process Communication*
IP	*Internet Protocol*
URL	*Uniform Resource Locator*

List of Figures

List of Tables

List of Programs

Contents

Appendixes

Introduction

Computers are crucial tools that are pervasive in the modern world. Computer systems range from Personal Computers used by the general public for everyday activities to embedded systems that are present in a substantial portion of automobiles employing critical tasks. While not a strict requirement, computers commonly comprise two interacting parts: hardware and software. The physical electronic components of a computer are its hardware, while the set of programs and data that control the operation of the hardware is the software. In this sense, software can adopt various forms to instruct hardware. A program written in *Machine Language*, which uses zeroes and ones to code instructions specific to a piece of hardware, is a form of software. Programs can also rely on other programs to create abstractions that reduce the dependency on hardware specifications and provide a more readable and logical way of programming. Programs that more directly instruct hardware are of *Low-Level*, while the ones that use other programs to abstract its details are of *High-Level*.

From the perspective of High-Level programs that rely on Low-Level programs to use computer resources for any purpose, an *Operating System* (OS) is the complex piece of software that serves as the interface between these two levels of programming. From an alternative standpoint, an OS serves as the component responsible for managing these computer resources because they are both limited and require coordination to avoid conflicts when multiple pieces of software are using them. In summary, we can derive that OSes are fundamental in making computers more efficient and reliable because they manage computer resources efficiently and reliably. Also, OSes provide an environment for developers that is simpler and closer to natural language in terms of programming, which results in software production that is more diverse and faster in rate when compared with programming strictly at a Low Level.

The main component of an OS is its kernel, which commonly encapsulates all its core functionality, and the software that absorbs all hardware details of the components it supports (called *device drivers*). A great example of an OS kernel is the *Linux kernel*. The Linux kernel was officially released by Linus Torvalds on October 5, 1991, as version 0.0.2, inspired by the UNIX and MINIX OSes. As a kernel does not constitute a fully functional OS, Linux was combined with utilities developed by the *GNU Project* [1] to form the *Free/Libre/Open-Source Software* (FLOSS)[2] GNU/Linux OS, which meant that

[1] The GNU Project was announced by Richard Stallman on September 27, 1983, to provide a complete collection of free software to society, including a full operating system.

[2] In this work, the acronym “FLOSS” is used as a representative for “Free Software”, “Open Source Software” (OSS), and “Free/Open Source Software”(FOSS)

it could be freely obtained, run, copied, modified, studied and distributed. The creation of GNU/Linux constituted a revolution as, although the GNU Project provided almost all software components to build a FLOSS OS, attempts to make a kernel for it were unsuccessful. Countless modified versions of the original GNU/Linux exist that are fine-tuned to work best in specific scenarios. Today, the GNU/Linux ecosystem plays a critical role in various aspects of computing and technology on a global scale. The Internet infrastructure (routers, switches, and the like that compose the Internet core) and the majority of servers that provide essential services to society run some form of a GNU/Linux OS. Also, GNU/Linux plays an educational role as its source code is available for anyone to study and is a good reference for a real-world functional OS.

Focusing on the Linux kernel, the project has been collaboratively developed for more than 30 years, practically since its inception. As time passes, the project becomes progressively larger and more complex. Due to the fact that the Linux kernel plays a vital role in our society, that the project continues to grow in an accelerated manner, and that the workforce needs to be scaled and continuously supplanted, the processes and practices that constitute the workflows of both contributors and maintainers needs to be optimized, simplified, and automatized. The complexity of a system should not relate to the complexity of developing the system. In this context, many tools support Linux kernel developers with services that enhance their workflows. Among the tools that aim to enhance the Linux developer workflow, the *KWorkflow* (kw) system is distinguishable, as it intends to be a unified environment that provides solutions that act on diverse fronts. For example, the typical processes of compiling and installing a Linux kernel are contemplated in the system, as well as the practice of running a script to check for code style violations.

Besides getting more extensive and complex, the project influx of contributions also grows fast, so a single person or group cannot fully understand and maintain the whole codebase. As a solution, the Linux kernel contribution model employs a *Chain of Command* model to break down the responsibility of maintaining the project into smaller portions, called *subsystems*. Each subsystem has one or more *maintainers* responsible for deciding which changes are accepted into the correspondent subsystem. Other maintainers evaluate the accepted changes, repeating this process until Linus Torvalds merges them into an official Linux release. *Electronic mail* is the medium used to propagate changes between contributors, maintainers, and anyone interested in helping in the reviewing process. *Mailing lists* [3] are instrumental and act as the public record of all interaction between the developing communities of the Linux kernel (this includes discussions, considering that not only changes flow through the mailing lists). There are some disadvantages to using mailing lists from the contribution model perspective. However, the most outstanding is that it requires developers to actively subscribe to consult the messages that flow through a mailing list. An on-demand approach to consult these messages is possible through the *Lore archives*, which aggregate updated archives of all mailing lists related to Linux kernel development.

Before this work, kw lacked a tool (called *feature* in the project) that improved the interaction with the mailing list for reviewing changes, unlike sending changes to the

[3] In this text context, a mailing list is a list of email addresses, in which each address is a recipient to every message sent to the mailing list.

corresponding lists and maintainers that was already contemplated with a feature in the project. In other words, there was an essential workflow in Linux development that kw did not cover. With this in mind, this work focused on studying the interactions of Linux kernel developers with the mailing lists and implementing a new feature to the kw project named `patch-hub` that provided a user-friendly interface with the Lore archives integrated with the kw environment.

The `patch-hub` feature was implemented using the *Model-View-Controller* architectural design pattern and the *Finite-State Machine* computational model. Concepts of database systems, *User Interface* (UI), and Web applications were also necessary in designing and developing the feature. By the end of this work, although there are future works that this author will do, the feature reached a functional state that validated the idea of an integrated interface to the Lore archives in the kw project.

The remainder of this capstone project consists of six more chapters. Chapter 1 is dedicated to further discussing the Linux kernel, its contribution model, the kw system, and the Lore archives. Chapter 2 focuses on outlining the integration of kw with a *Database Management System* (DBMS) that was a pre-requisite for implementing `patch-hub`. Chapter 3 and Chapter 4 delve into the concepts and implementations of `patch-hub` UI and `patch-hub` Model, respectively, which are the two major components that constitute the feature. Chapter 5 are the final remarks that summarize and conclude this work and also present future works that are planned to be done by the author. Finally, Chapter 6 describes the author's learnings and experiences while working on this project and throughout his undergraduate education.

Chapter 1

Linux kernel development

In the context of Operating Systems (OSes), a kernel is the most fundamental part of the system. The kernel is the one program running at all times on a computer with complete control over everything in the system by managing hardware resources and providing an interface for hardware and software components (Silberschatz *et al.*, 2012).

The Linux kernel is a FLOSS project released under the GNU General Public License (GPL) created by Linus Torvalds in 1991. The project has been collaboratively developed and grown in size and scale. As an illustration of the kernel community size, version 5.8, released in August 2020, had a codebase consisting of almost 70 thousand files that amounted to approximately 28.4 million lines of code (Stewart *et al.*, 2020). During the development cycle for version 6.4, released in June 2023, a total of 1980 developers contributed to the project, from which 282 made their first contribution [1].

The Linux kernel is the base for many Operating Systems that play a critical role in the functioning of our society. As of October 23, 2023, Linux-based Operating Systems power 37.4% of all websites [2]. Beyond that, 96.3% of the top one million web servers run a Linux-base Operating System [3]. By these statistics, we can derive that the Linux kernel is fundamental for adequately operating the Internet.

1.1 Linux Kernel contribution model

The Linux project is extensive, and the flow of changes is high. As there are many contexts, the project is broken down into smaller and more logical parts called *subsystems*. Each subsystem generally has an official upstream repository (also called a *kernel tree*) associated with a dedicated portion of the code, an exclusive public mailing list, and a set of maintainers. The public mailing list serves as the medium in which changes are sent from the contributors and reviewed by the community, and overall discussions occur. By dividing the project into subsystems, maintainers can specialize in some portions of the

[1] Source: https://lwn.net/Articles/936113/

[2] Source: https://w3techs.com/technologies/details/os-linux

[3] Source: https://www.zdnet.com/home-and-office/networking/can-the-internet-exist-without-linux/

codebase, improving the code quality and making the reviewing process more efficient. *Patches* are the unit for changes in the Linux project and are similar to the *commit* entity of the Git Source Code Management (SCM) system [4]. Figure 1.1 is a diagram that illustrates a patch lifecycle from its conception until it is merged into the correspondent upstream repository.

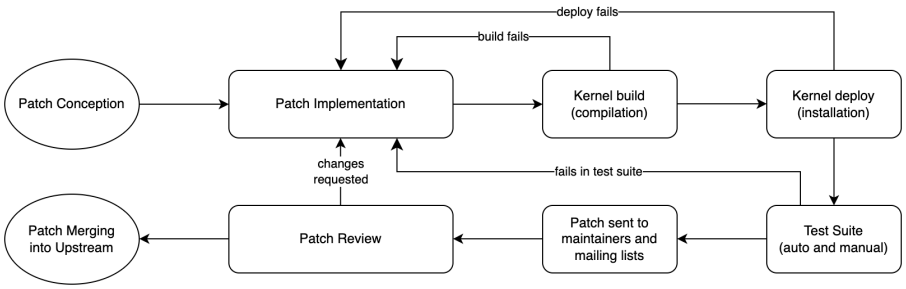


Figure 1.1: *Diagram of patch lifecycle on the Linux kernel project.*

There are subsystems for core parts of the Linux kernel, like Process Management, Memory Management, and Virtual File System. The vast majority of subsystems and lines of code, though, are destined for device drivers. These software components make hardware usable by the Linux kernel and, by extension, any software that builds upon the kernel. For example, the AMD Display Core subsystem handles the GPU scan-out in AMD *Graphics Processing Units* (GPUs).

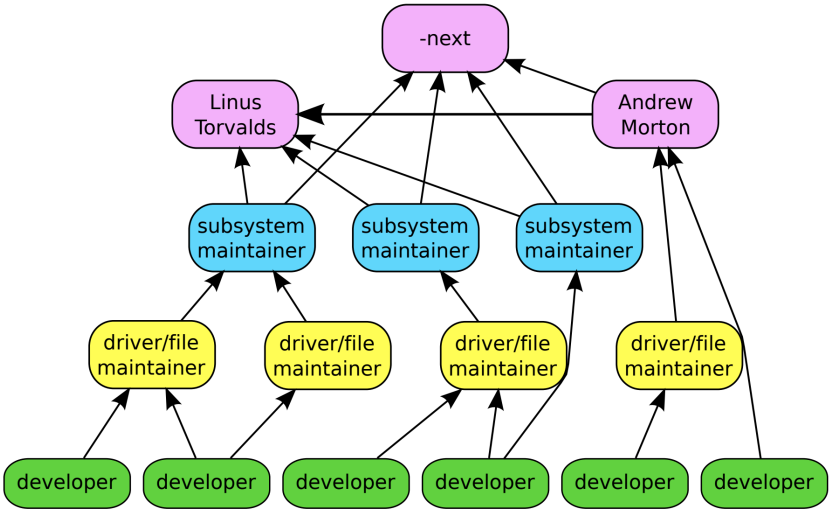


Figure 1.2: *Patch way until merging into Linux mainline.*

Most importantly, a unique repository called the *mainline* is maintained by Linus Torvalds, and it is the base for many other kernels, such as the stable version, which most of the distributions use. Every incorporated patch in any new main version of Linux must ultimately be merged by Linus Torvalds. However, just like the codebase is broken down

[4] https://git-scm.com/

into subsystems, the responsibility to select which and when patches are going into the mainline is shared with the correspondent maintainer or group of maintainers. This aspect of the contribution model is similar to a Chain of Command model, in which maintainers serve as gatekeepers to patches sent by contributors; those maintainers send the selected patches to other maintainers, and so on, until patches reach Linus Torvalds, who has the final say in if or when the patch will be incorporated, but, as the volume of patchsets is high, Linus Torvalds cannot thoroughly review each one and trusts that maintainers below the chain do not send bad patchsets. Figure 1.2 is a diagram illustrating the way a patch takes until it is merged into the Linux mainline.

In the contribution model, contributors send patches via email to the respective maintainers and dedicated mailing lists. Conversely, in the classic approach to consuming the influx of patches submitted, maintainers and anyone who desires to help in the reviewing process have to subscribe to the mailing lists, trust the correct mail distribution, and consult their mailboxes. In this approach, reviewers may lose patches if they are not subscribed to the mailing list when the patch was sent to the subscribers or if sending the patch to the given reviewer fails.

1.2 KWorkflow system

Linux kernel development includes many diverse sub-projects but entails general processes and practices that compose common workflows for most developers. For example, compiling the Linux kernel from source (also known as *building*) is a standard process in most workflows of Linux developers to test changes and a common practice to guarantee that changes did not introduce compilation errors. Building the kernel is a heavily automatable task, and using tools can simplify and accelerate development. Besides automation, the process can be optimized by tools that offer complimentary services, like listing the number of modules being compiled and kernel release name or integrate with the *LLVM* [5] toolchain. Other typical processes and practices present in Linux kernel development are installing a Linux kernel image (*deploying*), managing compilation configuration files, sending patches through email to the correct mailing lists and maintainers, reviewing patches, and debugging using the `dmesg` [6] log.

The Linux project provides a fundamental development tool, the `checkpatch.pl` script. The file is located in a Linux kernel source tree at `scripts/checkpatch.pl` and it is a *Perl* script used to check for *code style* [7] violations in patches. Note how, technically, a Linux developer can avoid using `checkpatch.pl` to check for code style violations in his/her patches; nonetheless, its use is almost standard because of its robustness and reliability for this practice.

Considering this context of tools that can enhance the workflows of Linux kernel

[5] The LLVM Project is a collection of modular and reusable compiler and toolchain technologies (for reference, see https://llvm.org/)

[6] `dmesg` is used to examine or control the kernel ring buffer (for reference, see https://man7.org/linux/man-pages/man1/dmesg.1.html).

[7] Set of rules and conventions that standardizes the writing of source code, making reading and understanding it more efficient, while also helping to avoid the introduction of errors.

developers, *KWorwflow*, also known as *KernelWorkflow* or just *kw*, is a *Developer Automation Workflow System* (DAWS) [8] that provides many features that simplify and optimize workflows related to Linux development. Quoting from the project website – "kw has a simple mission: reduce the environment and setup overhead of developing for Linux". Moreover, kw provides a unified interface for all its features.

Some kw features are more practical ones that automate and simplify core processes in Linux development, like `kw build` and `kw deploy`, which encompass many services related to compiling and installing Linux kernels, respectively. On the other hand, kw amasses other features that can be used by developers to indirectly improve their workflows, like `kw device` to retrieve basic information about the hardware of a target machine, `kw remote` to manage remote machines for easy access, and `kw pomodoro` to create and manage timeboxes using the Pomodoro time management technique. Table 1.1 exhibits some kw features and their surmised descriptions [9]. Currently, kw has a total count of 23 features that cover a wide variety of situations that fundamentally aim to improve the workflows of Linux developers.

kw feature	Description
kw build	Manage Linux kernel compilation
kw codestyle	Wrap and extend `checkpatch.pl`
kw config	Manage kw local and global configurations
kw deploy	Manage Linux kernel installation
kw init	Create local kw environment
kw kernel-config-manager	Manage Linux `.config` files
kw mail	Send patches through email
kw maintainers	Get maintainers and mailing lists of patches
kw patch-hub	User Interface to lore.kernel.org archives

Table 1.1: *List of kw features.*

In particular, this work focuses on the domain of `kw patch-hub`, its design and implementation. The kw is a FLOSS project, and its official repository is hosted in GitHub at https://github.com/kworkflow/kworkflow. From the contributor perspective, the kw project has well-defined code style rules and a modular architecture, which lowers the difficulty barrier for new contributors and makes the project more organized and robust. Figure 1.3 is a diagram with the project structure of kw emphasizing its modular architecture. Succinctly, the structure is composed of a file that serves as kw entry point named *kw (HUB)* on the diagram, dedicated files for each feature named *Components*, library files named *Libraries*, and specific files for code that is very mutable or too specific named *Plugins* (Neto, 2022). More details about the project structure can be found at https://kworkflow.org/content/project_structure.html.

[8] *Developer Automation Workflow System* is a definition proposed by this work that was created in conjunction with the kw community, based on the definition *Version Control System* used by Git.

[9] Man pages for all kw features, along with other relevant information, are available at the project website https://kworkflow.org

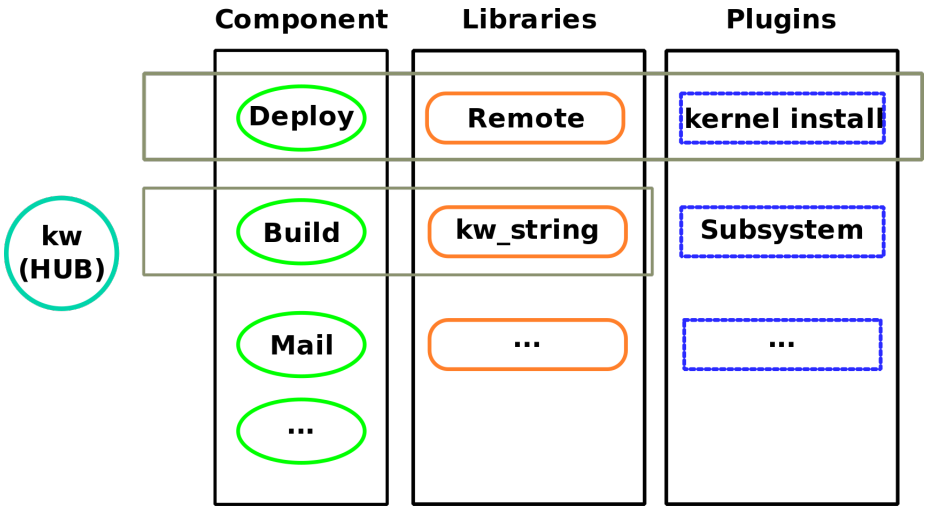


Figure 1.3: *kw project structure.*

1.2.1 Bash overview

KWorkflow is entirely implemented in *Bash* [10] script. As this text analyzes implementations for the `patch-hub` feature and other code excerpts from the kw project, this subsection aims to outline Bash concepts referenced in the following chapters.

Bash is an acronym for *Bourne-Again SHell* and is a wordplay with Stephen Bourne, the author of the *sh* shell, which is the ancestor of Bash. A *Shell* is a macro processor that executes commands, and a macro processor is a program that expands text and symbols to create more extensive expressions. In this sense, Bash is both a command interpreter and a programming language (known as Bash script). As a command interpreter, Bash provides an interface for the invocation of programs through commands [11] which are compositions of sequences of characters (called *tokens* in Bash). As a programming language, files containing sequences of commands combined with reserved keywords that provide fundamental Bash operations can constitute new programs from which a whole system can be built.

Bash is a command interpreter, so Bash scripts are interactively interpreted line by line. In summary, when executing a script, Bash follows these steps for each line [12]:

1. Reads its input from the script.
2. Breaks the input into tokens.
3. Parses the tokens into simple and compound commands.
4. Performs the various shell expansions.
5. Performs any necessary redirections.

[10] Bash is a typical shell that was first created to be the shell, or command language interpreter, for the GNU operating system (for reference, see https://www.gnu.org/software/bash/manual/bash.html).

[11] Shells are also known as *command-line interfaces* (CLIs).

[12] Source: https://www.gnu.org/software/bash/manual/bash.html#Shell-Operation

6. Executes the command.
7. Waits for the command to complete and collects its exit status.

Bash has many expansions, but the most important are shell parameter expansions, command substitutions, and arithmetic expansions. Shell parameter expansions are in the form of `$<parameter>`, in which `parameter` is the symbol that references a variable that stores a value (this value can be empty, or the variable can be unset), and the token is expanded to the value of `parameter`. Command substitutions are in the form of `$(<command>)`, in which `command` is a command that Bash can execute, and the expansion is the output from the execution of `command`. Arithmetic expansions are in the form of `$(( <expression> ))`, in which `expression` represents an arithmetic expression, and the result from the evaluation of the expression is the expansion. As an illustration, Program 1.1 is a Bash script that simply assigns the value 15 to `fifteen` using an arithmetic expansion, then assigns the value 0 to `zero` using a command substitution (the command `echo` outputs the values passed as arguments), and, finally outputs the value `15 minus 15 is 0` using two shell parameter expansions.

Program 1.1 Example of Bash script with expansions.

```
1    fifteen=$(( 10 + 5 ))
2    zero=$(echo '0')
3    echo "15 minus $fifteen is $zero"
```

From Program 1.1, another essential aspect of Bash syntax is noticeable: quoting. Character sequences enclosed in single quotes are treated as a single token, with the literal value of each character being preserved. In contrast, sequences enclosed in double quotes are susceptible to expansions. For example, if, in line 3 of Program 1.1, the argument to `echo` was `'15 minus $fifteen is $zero'`, the outputted value would be `15 minus $fifteen is $zero`, as no expansion would occur. It is imperative to note the special parameter expansions `$<positive-integer>`, that expands to the argument of number `positive integer`, and `$?`, that expands to the exit status of the last command executed.

The following reserved keywords constitute fundamental operations: `function` to define a function (that is treated like a command), `local` and `declare` to declare variables, `if-then-else-elif` for conditionals, and `for-until-while` for loops. Also, there are reserved metacharacters to denote input and output redirections, like `>` to redirect output, `<` to redirect input, and `|` to pipe the output of one command to the input of another. Program 1.2 exemplifies the use of some reserved keywords and redirections. The function `delete_foo` accepts an argument that is stored in the local variable `string`, then there is a conditional that, in case the value of `string` is `foo`, an empty value is outputted, and, otherwise pipes the `echo` and `sed` [13] commands to output the value of `string` without the literal `foo`. In the last line of Program 1.2, `delete_foo` is called with an argument `Hello fooWorld!` and redirects the output of the call to the file `hello_world.txt`, to which the contents of the file end up being `Hello World!`.

[13] `sed` is a command to manipulate and process streams (for reference, see https://man7.org/linux/man-pages/man1/sed.1p.html)

Program 1.2 Example of Bash script with keywords and redirections.

```
1   function delete_foo()
2   {
3     local string="$1"
4
5     if [[ "$string" == 'foo' ]]; then
6       echo ''
7     else
8       echo "$string" | sed 's/foo//'
9     fi
10  }
11
12  delete_foo 'Hello fooWorld!' > hello_world.txt
```

1.3 Lore archives

The Linux development mailing lists archives on lore.kernel.org, referred to as *Lore archives* or just *Lore* throughout the text, is a Web application that offers on-demand access to mailing lists related to the development of Linux, either directly or indirectly (for instance, there is a dedicated mailing list for the development of Git). Lore uses the *public-inbox* [14] technology, which describes itself as "an 'archives first' approach to mailing lists" and implements the sharing of a mailbox via Git to complement or replace traditional mailing lists. The technology also aims to be easy to deploy and manage. As such, it encourages projects to run their instances with minimal overhead.

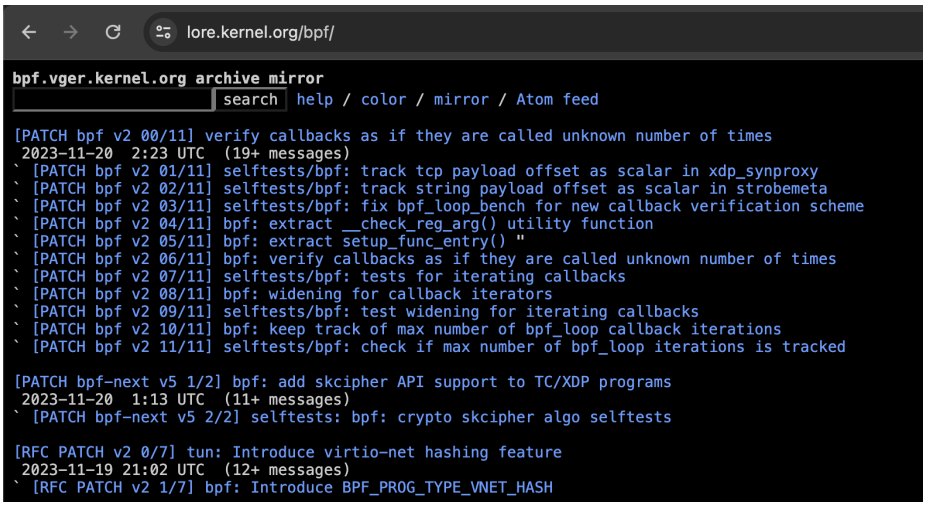


Figure 1.4: *Screenshot of the `bpf` mailing list archive on Lore accessed using a Web browser (visited on November 20, 2023).*

Lore offers on-demand access to the mailing lists related to Linux development because it consumes each list as a subscriber (a more reliable one) and provides the service of consulting these lists at the users will without needing a subscription or even an email account. Users can consult the archives in an HTML format using a Web browser, for example, or through Atom feeds [15]. Figure 1.4 is a screenshot of the top of the `bpf` mailing

[14] https://public-inbox.org/README.html

[15] Atom is an XML-based document format that describes lists of related information known as "feeds" (source:

list archive accessed using a Web browser.

Compared with the subscribing approach of consulting mailing lists, Lore presents a more flexible, reliable, and organized solution for reviewers on the Linux contribution model. There is no need to receive and store a high volume of mail just to consult a small portion of it, despite keeping a local copy of a complete archive through mirroring being arguably more straightforward and robust using Lore.

The Lore archives also offer an API that allows other applications to integrate. Although the documentation on the API is scarce, incorporating Lore into the workflow of Linux-related developers, especially maintainers, can enhance the process of patch reviewing. It is pertinent to note that not all messages that flow through the mailing lists are patches. Messages can also be discussions that do not contain code and replies to other messages. A more specific definition of a patch, in the context of Lore, is a message that contains *code differentials* [16]. It is also worth distincting patches from what we refer to in the text as *patchsets*. The former are individual messages with code differentials. At the same time, the latter indicates a set of patches related to a broader and common context (similar to commits and Pull Requests in GitHub [17]).

In the kw project, the `patch-hub` feature is being developed to integrate the Lore archives with the kw environment to provide for the patch-reviewing side of the Linux contribution model (`kw mail` already covers the patch-sending side of the model). The feature objective is to be a terminal-based hub for consulting patchsets from the available mailing lists, applying actions integrated with the kw environment on specific patchsets, and providing other services such as bookmarking patchsets. Chapter 3 and Chapter 4 discuss in details `patch-hub` concepts, design, and implementation. Chapter 2 disserts about the management of databases in the kw project as integrating the project with a Database Management System was a prerequisite for `patch-hub`.

https://datatracker.ietf.org/doc/html/rfc4287).

[16] A code differential, also called *code diff* or *diff*, is a text that represents changes made to a set of source code files between two different versions and that can be used to transition these files from one version to another.

[17] https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/proposing-changes-to-your-work-with-pull-requests/about-pull-requests

Chapter 2

Database Management System of KWorkflow

The kw project had many features that relied on storing and manipulating user data; the project employed data access by directly interacting with the files that composed the databases. This approach was functional but compromised maintainability and scalability.

As `patch-hub` (see Chapters 3 and 4) would need a more robust system to manage user data, and the whole project would benefit from an approach that provided developers with an easier way to interact with the databases, we integrated a *Database Management System* (DBMS) to kw. The DBMS chosen was SQLite3, as it fulfilled all the requirements from kw developing community.

We divided this chapter into three sections:

1. Overview of *flat-file databases*, and description of how kw used to manage its databases;
2. Overview of Database Management Systems and the requirements that led to choosing SQLite3;
3. Description of how kw currently manages its databases and an empirical benefit from adopting a DBMS.

2.1 File-Based Databases

A *file-based database*[1], also called a *flat-file database*, is a straightforward way to represent data records that have no **structured interrelationship**, leaving the database application (i.e., the part of the application that manages the storing, retrieving, and manipulation of data) with the responsibility of knowing **how** data is organized within files and **where** those files are stored.

[1] https://web.archive.org/web/20090320001015/http://knowledge.fhwa.dot.gov/tam/aashto.nsf/All+Documents/4825476B2B5C687285256B1F00544258/$FILE/DIGloss.pdf, pg. 11

As an illustration, imagine a developer building an application that manipulates data, and the developer has to store this data not on main memory but on persistent memory because, for example, the application does not run continuously, and it will need to be accessed later. One way to address this problem is by creating a file and outputting the data to this file. It can be a plain text file or a binary file, but in any case, the developer has to manage two things:

1. **Where** the file is being stored to insert, update, remove, and retrieve data from the correct file.
2. The format of **how** the data is stored to manipulate it correctly.

The file described in the illustration constitutes a *file-based database*. There are trade-offs to using this approach, and we list the pros and cons in Table 2.1 below.

Pro/Con	Description	Effect
Pro	Developer does not need to configure and learn an external system like a DBMS	More agility in development
Pro	Simpler databases like `/etc/passwd` (Program 2.1) do not need to scale	Straightforward implementation
Con	Application needs to know details about the file structure of stored data	Added complexity to application
Con	Application needs to know details about the format of stored data	Added complexity to application

Table 2.1: *Trade-offs of using a file-based database approach.*

Program 2.1 Example of part of `/etc/passwd`

```
1 root:x:0:0::/root:/bin/bash
2 bin:x:1:1::/:/usr/bin/nologin
3 daemon:x:2:2::/:/usr/bin/nologin
4 mail:x:8:12::/var/spool/mail:/usr/bin/nologin
5 ftp:x:14:11::/srv/ftp:/usr/bin/nologin
6 http:x:33:33::/srv/http:/usr/bin/nologin
```

2.1.1 How kw managed its databases

kw is an XDG-compliant[2] application. The *XDG Base Directory Specification* defines base directories relative to which files for different contexts should be located. As an example, the user-specific configuration files base directory is defined by the environment variable `$XDG_CONFIG_HOME` which, if empty or not set, a default equal to `~/.config` should be used.

In this sense, kw stores user-specific data files at `~/local/share/kw`, and three sub-directories contained files that represented the kw databases:

[2] https://specifications.freedesktop.org/basedir-spec/basedir-spec-latest.html

- `~/local/share/kw/statistics`: For all statistics collected by kw related to the `kw build` and `kw deploy` features, that provides users with many services for compiling and installing Linux kernels, respectively.
- `~/local/share/kw/pomodoro`: For the report on Pomodoro sessions related to the `kw pomodoro` feature, which provides users with a terminal-based utility to create Pomodoro sessions[3].
- `~/local/share/kw/configs`: For files and metadata managed by the `kw kernel-config-manager` feature that provides users services for managing Linux kernel configuration files.

For each of the directories above, there was a specific file structure that stored one or more *file-based databases*. In the following subsections, we will elaborate more on these databases.

Statistics database

In the case of statistics, a file path `statistics/<year>/<month>/<day>` represented statistics collected at <month>/<day>/<year>. For example, Program 2.2 could be a line in the file located at `statistics/23/08/23`, meaning that a `kw build` command started on August 23, 2023, and lasted for 8 minutes and 17 seconds (497 seconds).

Program 2.2 Example of entry in the kw former statistics database.

```
1 build 497
```

Pomodoro sessions database

For the report on Pomodoro sessions, there was a similar file structure to represent dates, with each line in a file `pomodoro/<year>/<month>/<day>` being a Pomodoro session that started at <month>/<day>/<year>. Differently from the statistics, though, each line/entry was comma separated, had a different number of attributes, and had an optional attribute. On top of that, there was a file `pomodoro/tags` for storing tags used by the `kw pomodoro` feature to group sessions.

Program 2.3 Example of contents from `pomodoro/tags`.

```
1 tag1
2 tag2
3 tag3
```

As an example, a valid file structure for the Pomodoro sessions database for report would be the file `pomodoro/tags` with the contents of Program 2.3, and a file `pomodoro/2023/10/26` with the contents of Program 2.4.

[3] https://en.wikipedia.org/wiki/Pomodoro_Technique

Program 2.4 Example of contents of Pomodoro sessions data from a single day.

```
1 tag1,300s,00:20:30,description 1
2 tag3,10m,12:10:00
3 tag1,1h,14:45:00,description 2
4 tag2,2h,20:05:00
```

This file structure meant that there were three groups for the Pomodoro sessions labeled `tag1`, `tag2`, and `tag3`, and that on October 26, 2023, each session pertained to the group in the first column, lasted for the time in the second column, started at the time in the third column, and, optionally, had a description in the fourth column.

Linux kernel `.config` files database

The Linux kernel compilation is configured via a special file named `.config`. When using `kw kernel-config-manager`, full `.config` files are stored inside `configs/configs`, and metadatafiles are stored inside `configs/metadata`.

For example, for a `.config` file managed by kw named `debian-optimized` there would be a file `configs/configs/debian-optimized` that would be the actual `.config` file, as well as a metadata file `configs/metadata/debian-optimized` containing a description for this particular configuration file.

2.2 Database Management Systems

Although the approach delineated in previous section was functional and fairly simple when considered separately, each feature had to implement and manage its own *file-based databases* with its own details. This made the code hard to scale and more coupled with these particularities of **where** and **how** the data was stored.

A Database Management System (DBMS) is a dedicated software that serves as an interface for applications (or end-users) and the database itself for storing, retrieving, and manipulating data. It specifically should provide the applications (or users) with the following (Ullman, 2007):

1. Ability to create databases (the actual data records) and their *schema* (the logical structure of the data) using a *Data Definition Language* (DDL).
2. Allow *Querying*, i.e., fetching the stored data using conditionals and modifying the data stored using a language called *Query Language*, or *Data Manipulation Language* (DML).
3. Support storage of large amounts of data persistently, keeping it secure from accidents and unauthorized use.
4. Control concurrent data accesses from many users simultaneously while keeping it consistent with each transaction.

Following the construction of the earlier sections, an application uses a DBMS to delegate the **how** and **where** the data is stored while also providing additional services like complex querying of data that use joins and aggregation, authentication, and crash recovery.

As mentioned in Section 2.1.1, replacing the *file-based databases* approach used by a DBMS, all the explicit data manipulation would be delegated to the DBMS. In turn, kw would only have to care for configuring the DBMS, creating the database schemas, and using a unified interface for interacting with the DBMS for data querying and manipulation.

2.2.1 The choice of SQLite3 as the DBMS

DBMSs are vast and diverse, proposing different solutions for different problems. To introduce a DBMS to kw, it needed to fulfill the following requirements:

- Be a FLOSS project: to keep up with the project philosophy.
- Have a CLI (Command-Line Interface) for easy integration with Bash: maintaining the project in pure Bash.
- Have a small footprint: to keep the kw installation light.
- Run on user space: for security.
- Be a Relational DBMS[4]: to leverage the high performance of the Structured Query Language (SQL) for queries.
- Be portable: to be something easy to set up.

The DBMS selected was SQLite3 [5] [6], as it is Public Domain (not exactly FLOSS, but far from proprietary), has a CLI interface, sizes less than 1 MB as of July 04, 2023[7], runs on user space, and is Relational. PostgreSQL[8] was a candidate, as it is a popular open-source Relational DBMS, has a CLI named *psql*, but is not as lightweight and portable as SQLite3. TinyDB[9] was also considered, as it is a lightweight open-source and portable DBMS but is not Relational (it is a document-oriented DBMS), and does not have a CLI (we would need to use Python).

[4] Also called a *SQL DBMS*, it has data structured in tables, favoring relationships by joining tables and aggregation of data.

[5] https://www.sqlite.org/index.html

[6] SQLite3 technically belongs to the family of embedded databases, as it is a library that developers embed in applications rather than being a standalone application. Nevertheless, in the kw project, SQLite3 absorbs most functionalities that a normal DBMS would, and SQLite3 is often defined as one, so we use this linguistic abuse throughout the text.

[7] https://www.sqlite.org/footprint.html, visited in November 01, 2023.

[8] https://www.postgresql.org/

[9] https://tinydb.readthedocs.io/en/latest/

2.3 From a *file-based database* to a DBMS approach

Neto, 2022 started the migration from a *file-based database* to a DBMS approach. This section describes the evolution of this migration provided in this current work.

The introduction of SQLite3 into kw entailed adapting all features that directly or indirectly used the kw databases by removing the explicit management of the data files and integrating them with the SQLite3 interface. These features are:

- `kw build`.
- `kw deploy`.
- `kw kernel-config-manager`.
- `kw report`.
- `kw backup`.

Function Signature	Operation
`insert_into <table> <columns> <entries>`	Insertion
`select_from <table> <columns>`	Querying
`remove_from <table> <match_conditions>`	Deletion
`replace_into <table> <columns> <entries>`	Update
`execute_sql_script <script_path>`	Execute SQL script
`execute_command_db <sql_command>`	Execute SQL command

Table 2.2: *kw library functions for interacting with SQLite3.*

With the SQLite3, all kw databases are stored in the file `~/.local/share/kw/kw.db` (instead of having multiple subdirectories at `~/.local/share/kw` for each database). Now, the code does not have to manage **where** the data is stored, reducing its complexity. Additionally, SQLite3 provides an efficient and straightforward CLI interface, which enables us to create a set of library functions that are wrappers for SQLite3 commands to simplify even further **how** the data is stored, retrieved, deleted, and manipulated by kw. These library functions are implemented in the file `src/lib/kw_db.sh`. Table 2.2 lists the functions' signatures and the operations to which they correspond.

Program 2.5 Function `is_tag_already_registered` from `src/pomodoro.sh` after SQLite3 introduction that uses a SQL conditional for a query.

```
1   function is_tag_already_registered()
2   {
3     local tag_name="$1"
4     local is_tag_registered=''
5
6     is_tag_registered=$(select_from "tag WHERE name IS '${tag_name}'")
7
8     [[ -n "${is_tag_registered}" ]] && return 0
9     return 1
10  }
```

These wrappers support most SQL expressions, and the first four functions cover practically all interactions needed between kw and the databases. Program 2.5 is an example of a query using the SQL conditional `WHERE <attribute> IS <value>` to check if there is a Pomodoro tag with a given name registered in the database in the function `is_tag_already_registered`. Program 2.6 shows the implementation of the same function before introducing SQLite3 into kw. By comparing the two implementations, we can see that the latter needs to know that the file that stores the target database is defined by the `KW_POMODORO_TAG_LIST` variable and that it contains only a tag name per line, while the former only needs to know the name of the table that represents the Pomodoro tags database.

Program 2.6 Function `is_tag_already_registered` from `src/pomodoro.sh` before SQLite3 introduction.

```
1    function is_tag_already_registered()
2    {
3      local tag
4
5      tag="$*"
6
7      tag="\<$tag\>" # \<STRING\> forces the exact match
8      grep -q "$tag" "$KW_POMODORO_TAG_LIST"
9      return "$?"
10   }
```

Although each database (statistics, Pomodoro sessions, and Linux kernel `.config` files) still has its entities and relationships, the way that any data is inserted, queried, deleted, and updated is the same by using these library calls. That standardizes how the data is stored, which further reduces its complexity.

Besides these benefits that were the actual motive for the DBMS introduction, we should notice a collateral benefit: **performance**. As kw used to manage many plain-text files spread across many directories and subdirectories, kw coordinated these input and output operations, which were slow compared to a DBMS that focuses on database management and accesses a single binary file.

	`perf stat` time output
Before Introduction	`55.084 +- 0.136 seconds time elapsed ( +- 0.25% )`
After Intoduction	`38.9413 +- 0.0955 seconds time elapsed ( +- 0.25% )`

Table 2.3: *Comparison of time using `perf stat` for running the whole test suite before and after introducing SQLite3 to kw.*

To further investigate this performance improvement, we used the script `run_tests.sh` for running the project automated tests was used. In combination with the `perf`[10] performance analyzing tool, the whole test suite was run ten times, both before and after the SQLite3 introduction by using the command `perf stat -repeat 10 ./run_tests.sh`. Table 2.3 compares the time output of `perf stat` for running the whole test suite before and after introducing SQLite3 to kw. By analyzing the table, we can notice almost 30% decrease in time.

[10] https://perf.wiki.kernel.org/index.php/Main_Page

Chapter 3

`patch-hub` User Interface

The `patch-hub` feature implements an adapted version of the *Model-View-Controller* (MVC) architectural design pattern (Fowler, 2012). We split the feature into two parts:

- The `patch-hub` User Interface (UI) that, in terms of the MVC design pattern, corresponds to the roles of the View and the Controller, and it is implemented using the *Finite-State Machine* (FSM) model and the `Dialog` tool[1]. Conceptually, the View is responsible for displaying information, and the Controller is responsible for taking user input, manipulating the Model, and updating the View accordingly.
- The `patch-hub` Model that corresponds to the Model role of the MVC design pattern. The Model role is to represent all the domain-related information.

In this chapter, we describe the `patch-hub` UI, and it is divided into four sections:

1. Background concepts: overviews of the View and Controller roles of the MVC architectural design pattern and Finite-State Machines.
2. Outline of the `Dialog` tool.
3. How the `Dialog` is used as a framework in kw.
4. Description of the implementation of `patch-hub` UI.

3.1 Background

3.1.1 The View and Controller roles of MVC

The MVC has two principal separations (Fowler, 2012):

1. Separation between the presentation (UI) and Model.
2. Separation between the View and Controller.

[1] https://invisible-island.net/dialog/manpage/dialog.html

The first separation is one of the most fundamental heuristics of good software design, while the second is desirable but less important (Fowler, 2012). A UI can be regarded as a combination of the View and Controller components. The former is responsible for displaying information to the user. At the same time, the latter is responsible for managing user interaction, manipulating the Model component (which represents all the information about the domain), and updating the View (or Views, if there is more than one) (Fowler, 2012).

As an illustration, consider an application that, at a certain point, displays a menu with a list of options and prompts the user to navigate the options and choose one. The View handles the design of the menu and how each piece of information is displayed. The Controller handles the press of a button to move the menu cursor to another option and signals the View to reflect this update. The Controller is also responsible for signaling the Model for any necessary action, for instance, storing the option chosen by the user in a database. In `patch-hub` UI, the `Dialog` tool serves as both components, as it will be discussed in Section 3.2.

Notice in the illustration that the View and Model components do not interact directly, only through the Controller component. Hence, the application can have multiple View components while implementing one Controller, one Model, and two interfaces: an interface for the Controller and Model and another for the Controller and the multiple Views. Figure 3.1 is a diagram describing the interaction between components in the MVC.

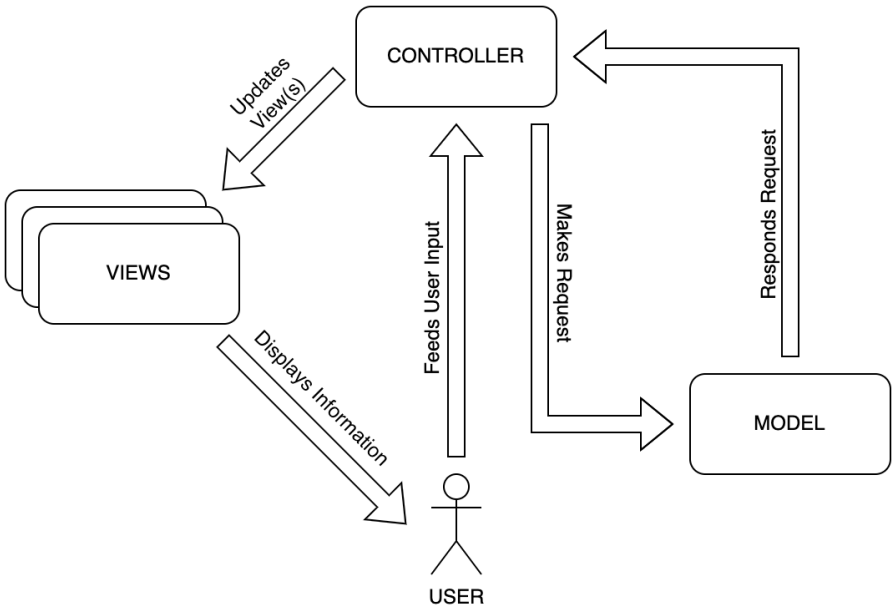


Figure 3.1: *Diagram describing the interaction between components in the Model-View-Controller architectural design pattern.*

To strictly adhere to the original MVC pattern, it is imperative to isolate the View and Controller components and implement a well-defined interface between them, as emphasized by the separation (2). Nevertheless, design patterns can be used as templates and adapted to solve specific problems better. In this regard, not enforcing separation (2) entails a trade-off.

The benefit of not following separation (2) is that the UI can be built as a single component while maintaining separation (1) between the UI and the Model. Keeping separation (1) decouples domain logic from presentation and allows Model developers - who have to be concerned about business policies, database interactions, authentication, and other problems - to focus on building a more robust application core. A good detachment between UI and the Model also allows better testability of those components, as it is easier to test the domain logic without having to mock visual and user-interactive components, and vice versa. The drawback of not following the mentioned rule is that implementing multiple Views is more complex, as adding a new View is not as straightforward as following a well-defined interface, and it is harder to implement and test visualization and user interaction separately.

3.1.2 Finite-State Machines

Finite-State Machine (FSM), or Finite-State Automaton (FSA), is a mathematical model of computation of an abstract machine that can model various problems in the computer software and hardware fields (Hopcroft *et al.*, 2006). A formal definition says that an FSM is a 5-tuple (a list of five objects) (Sipser, 1996):

1. A finite set of states the FSM can be on.

2. A finite set of input symbols that the FSM processes called *alphabet*.

3. Rules for moving, also called a *transition function*, that defines the transitions from a state A to a state B, depending on the state A and the input symbol X that the FSM receives.

4. A *start state*, that is the state the FSM begins execution.

5. A subset of the finite set of states called the *set of accept states*, which ends the FSM execution.

From the formal definition above, we can derive that an FSM is an abstract machine that can be on a finite number of states, but only one is active at once. The machine receives input symbols and transitions between states based on the *transition function*. Notice that not every two states must have a transition from one to the other. In other words, if the machine is in state A, there may be no input symbol in the *alphabet* that triggers a transition to a specific state B. From the definition, there can also be no transitions or states. Examples of FSM inputs are characters inputted by a human using a keyboard to software (the FSM) and signals being sent from a sensor to another device (the FSM).

Figure 3.2 is a diagram of an FSM that has four states, A, B, C, and D, and only receives inputs 0 and 1. The labeled circles represent the states, and the arrows represent the transitions. The 0s and 1s, on the side of the arrows, represent the input needed for the transition. We could omit transitions that take the machine to the same state, but they are explicit in the figure to illustrate that not every input triggers a state change.

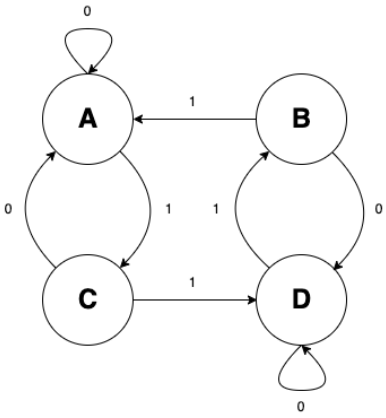


Figure 3.2: *Diagram of a Finite-State Machine with four states.*

Types of Finite-State Machines

FSMs can be of two types (Hopcroft *et al.*, 2006): a deterministic Finite-State Machine (DFSM) and a non-deterministic Finite-State Machine (NFSM). An FSM is a DFSM if it follows two restrictions:

1. Each transition is totally and uniquely defined by its starting state and necessary inputs.
2. For a transition to happen, the FSM needs to receive input.

Figure 3.2 is also an example of a DFSM.

NFSMs do not need to follow these restrictions. DFSMs are a subset of NFSMs. In simpler terms, for DFSMs, the machine only transitions between two states when well-defined inputs occur (that is why it is called deterministic), and for NFSMs, this is not true. Hence, a transition between two states has a probability of happening with the machine receiving, or not, a set of inputs.

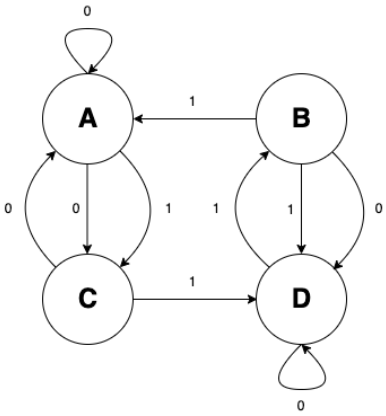


Figure 3.3: *Diagram of a non-deterministic Finite-State Machine with four states.*

Figure 3.3 is a diagram of an NFSM, which builds upon Figure 3.2. The only difference

is that the former adds two transitions to the latter:

1. Transition from state A to state C by receiving 0.
2. Transition from state B to state C by receiving 1.

These additions turn the previous DFSM into an NFSM because the machine in state A can, probabilistically, transition to state C or stay in state A by receiving 0. The same happens when the machine is in state B and receives 1. It can, probabilistically, transition to state A or state D.

Applications of the Finite-State Machine model

A classic example of an FSM is a traffic light, which has three states: green light, yellow light, and red light. Suppose the traffic light is in the green light state. It will transition to the yellow state when the green light timer expires. Then, when the yellow light timer expires, it transitions to the red light state, and so on. Notice that the input, in this case, is the expiration of a timer. Notice that a traffic light is a DFSM, as transitions only occur when input is received (a timer expires), and the starting state and the input totally define one.

A well-known application of the FSM model in software development is for building user interfaces. Usually, UIs display information to the user, collect and process input (not necessarily from the user), and then display more information to the user, repeating the loop. In this regard, a UI can be modeled as an FSM by abstracting each iteration of the loop or a sequence of iterations as states and the inputs as triggers for transitions. Depending on the application domain, a DFSM or a NFSM can be used to model the problem.

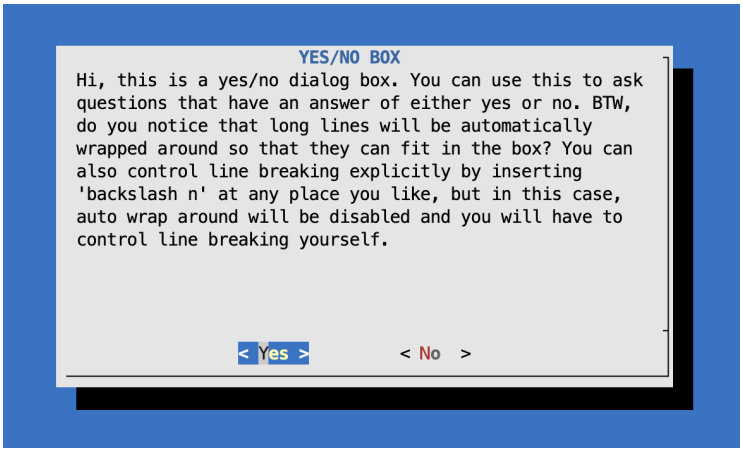
As Section 3.4 discusses further, a DFSM is used to model `patch-hub` UI since the `patch-hub` feature is designed to have a finite set of states (although the number increases with development) and a well-defined set of inputs. The inputs in the feature are user inputs and responses to web requisitions. Even though the response to a web requisition is not deterministic, considering there is a finite number of responses for each type of requisition, handling responses is deterministic, making the DFSM model suitable for `patch-hub` UI.

3.2 The `Dialog` tool

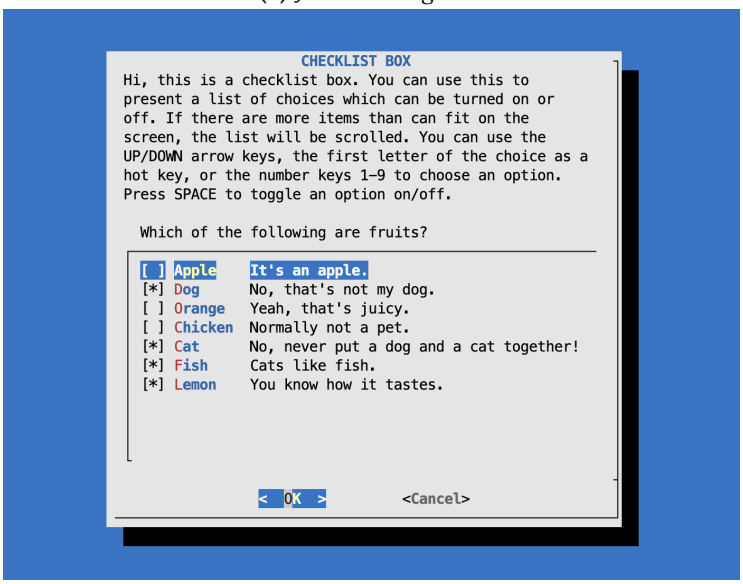
`Dialog` is a program capable of displaying dialog boxes from shell scripts. `Dialog` was written by Savio Lam and was first released on December 17, 1993. The project is now maintained by Thomas Dickey[2].

Through CLI commands, dialog boxes created by `Dialog` are used to expose information and collect input from the user. A dialog box is a terminal user interface (TUI) that provides several types of dialog boxes that `Dialog` can create, each adapted to a different user interaction.

[2] https://invisible-island.net/dialog/dialog.html



(a) *`yesno` dialog box.*



(b) *`checklist` dialog box.*

Figure 3.4: *Two dialog boxes created by `Dialog`, adapted from https://github.com/tolik-punkoff/dialog-examples.*

Figure 3.4a is a screenshot of the `yesno` dialog box, which displays a text message and two buttons, a *Yes* and a *No* button. The user can change the highlighted button by pressing the left and right arrow keys of the keyboard and select the highlighted button by pressing the enter key from the keyboard. Selecting a button closes the dialog box and sets the exit status (denoted by the parameter `$?`) to 0 if the *Yes* button was selected and to 1 if the *No* button was selected. Figure 3.4b is an example of the `checklist` dialog box that, similar to the `yesno` dialog box, displays a message with two buttons and, additionally, displays a list of items with descriptions in the format of a checklist. Besides using the left and right arrow keys to navigate through the buttons, the user can change the highlighted item of the list with the up and down keys and check it by pressing the space bar key from the keyboard. Like the `yesno` box, selecting the leftmost button (labeled *OK*) sets the exit status to 0, while selecting the rightmost button (labeled *Cancel*) sets the exit status to 1. Moreover, independent of the button selected, the list of items checked is outputted to the standard error stream (file descriptor number 2).

Each dialog box can be customized. For all boxes, the box title, the message inside the box, the button labels, and the dimensions of the box can be overridden. Other configurations specific to certain boxes can be done. The `checklist` box can have extra buttons, items in the list with or without description, and checked items when the box is first displayed. These customizations are passed as arguments to the `Dialog` command. For example, Program 3.1 is the entire command that generates the box in Figure 3.4b.

Program 3.1 `Dialog` command that produces Figure 3.4b box when run.

```
dialog --backtitle "No Such Organization" \
       --title "CHECKLIST BOX" "$@" \
       --checklist "Hi, this is a checklist box. You can use this to \n\
present a list of choices which can be turned on or \n\
off. If there are more items than can fit on the \n\
screen, the list will be scrolled. You can use the \n\
UP/DOWN arrow keys, the first letter of the choice as a \n\
hot key, or the number keys 1-9 to choose an option. \n\
Press SPACE to toggle an option on/off. \n\n\
 Which of the following are fruits?" 25 61 5 \
       "Apple" "It's an apple." off \
       "Dog" "No, that's not my dog." on \
       "Orange" "Yeah, that's juicy." off \
       "Chicken" "Normally not a pet." off \
       "Cat" "No, never put a dog and a cat together!" on \
       "Fish" "Cats like fish." on \
       "Lemon" "You know how it tastes." on
```

Theme customizations, i.e., the color scheme, box edges, along with other customizations, are set in the `.dialogrc` file. By default, `Dialog` uses the `$HOME/.dialogrc` file. Prepending a command with `DIALOGRC=<path_to_dialogrc>` uses a specific file.

3.3 Using `Dialog` as a framework in KWorkflow

As `Dialog` creates many types of dialog boxes, each customizable, the program can be used as a low-level framework for building TUIs. Using library functions to wrap `Dialog` commands makes the use of the program more robust because it reduces code duplication, allows the creation of function signatures that are both simpler and follow the project

code style, makes it possible to add optimizations (like default height and width of the dialog box), and enables unit testing of the functions.

In the kw project, the file `src/lib/dialog_ui.sh` implements many functions to build interactive terminal screens using `Dialog`. Program 3.2 is the implementation of the `create_directory_selection_screen` function, which creates a dialog box that allows the user to select a directory path in the file system.

Program 3.2 Implementation of the function `create_directory_selection_screen` that creates a dialog box to select a directory path in the file system.

```
1   function create_directory_selection_screen()
2   {
3     local starting_path="$1"
4     local box_title="$2"
5     local height="$3"
6     local width="$4"
7     local flag="$5"
8     local cmd
9
10    flag=${flag:-'SILENT'}
11    height=${height:-'15'}
12    width=${width:-'80'}
13
14    # Escape all single quotes to avoid breaking arguments
15    box_title=$(str_escape_single_quotes "$box_title")
16
17    cmd=$(build_dialog_command_preamble "$box_title")
18    # Add help button
19    cmd+=" --help-button"
20    # Add directory selection screen
21    cmd+=" --dselect '${starting_path}'"
22    # Set height and width
23    cmd+=" '${height}' '${width}'"
24
25    [[ "$flag" == 'TEST_MODE' ]] && printf '%s' "$cmd" && return 0
26
27    run_dialog_command "$cmd" "$flag"
28  }
```

All library functions in `src/lib/dialog_ui.sh` adhere to the pattern shown in Program 3.2, which is delineated as:

1. **Arguments of `Dialog` options are converted into function arguments**: Consider the function argument `box_title`, which is the argument of the `Dialog` option `–title`. In this case, the function caller must only conform to the function signature without knowing the correspondent option of `box_title`.
2. **Optional arguments fallback to default values**: Notice how the arguments `flag`, `height`, and `width` fallback to default values in lines 10, 11, and 12, respectively, of Program 3.2, in case they are empty.
3. **String arguments have single quotes escaped to avoid breaking `Dialog` commands**: As the commands are run as a single Bash command, unescaped single quotes dismember string argument, which can break the `Dialog` command, resulting in crashes.
4. **`Dialog` commands are built incrementally**: Building the command in steps makes the code more organized and allows the addition of common excerpts, like line 17 of Program 3.2, which adds a preamble with theme customizations, dialog box title, and dialog box message.
5. **Support for unit testing**: By passing the argument `flag` with the value `TEST_MODE`, the function builds the `Dialog` command and outputs it without running it, which allows unit testing, as it is only necessary to check the issued command. This procedure enhances the efficiency of the tests, given that dialog boxes are not rendered.

The pattern described above simplifies, standardizes, and optimizes the use of `Dialog` for creating TUIs, not only for the `patch-hub` feature but also throughout the whole kw project. Characteristics number (1), (2), and (3) of the pattern provide a more straightforward and solid API for creating dialog boxes, while characteristics number (4) and (5) result in more robust implementations of these library functions.

From the MVC architectural design pattern perspective, it is essential to point out that, when building TUIs using `Dialog`, the View and the Controller components are inherently coupled, as dialog boxes display information and collect user input. Besides the drawbacks of not separating View from Controller discussed in Section 3.1.1, `Dialog` imposes a limit on user interactiveness, such as four being the maximum number of buttons for some types of dialog boxes (this number is inferior for other types). However, `Dialog` fulfills `patch-hub` prerequisites for the Controller component in terms of user interactiveness.

3.4 `patch-hub` User Interface implementation

3.4.1 `patch-hub` entry point

The code style of kw demands that each feature have a dedicated file inside the `src/` directory. Henceforth, the file `src/patch_hub.sh` is dedicated to the `patch-hub` feature.

Program 3.3 Simplified listing of `src/patch_hub.sh`.

```
# The ‘patch_hub.sh‘ file is the entrypoint for the ‘patch-hub‘
# feature that follows kw codestyle. As the feature is screen-driven, it is implemented
# as a state-machine in files stored at the ‘src/ui/patch_hub‘ directory.

declare -gA options_values

include "${KW_LIB_DIR}/ui/patch_hub/patch_hub_core.sh"

function patch_hub_main()
{
  if [[ "$1" =~ -h|--help ]]; then
    patch_hub_help "$1"
    exit 0
  fi

  parse_patch_hub_options "$@"
  if [[ "$?" -gt 0 ]]; then
    complain "${options_values['ERROR']}"
    patch_hub_help
    return 22 # EINVAL
  fi

  patch_hub_main_loop
  return "$?"
}

function parse_patch_hub_options()
{
  ...
}

function patch_hub_help()
{
  ...
}
```

This file represents a component in the project structure, which must adhere to the component code style [3]. Program 3.3 is a simplified listing of `src/patch_hub.sh`.

Notice that at the top of the file, a function named `patch_hub_main` is defined, which is the entry point to the feature. At the end of the file, the functions `parse_patch_hub_options` and `patch_hub_help` are defined, which parses the options passed to the feature and displays the feature help (either a short help or the man-page), respectively. After entering the feature through `patch_hub_main`, there is a check to determine if the help should be displayed, then it parses the feature options, and, finally, it calls the function `patch_hub_main_loop`, which is not defined in `src/patch_hub.sh`, but rather in `src/ui/patch_hub/patch_hub_core.sh`.

3.4.2 The Finite-State Machine of `patch-hub`

Unlike other features in kw that have all feature-specific actions handled by functions defined in the file that represents a component, `patch-hub` implements the core of the feature in files at the `src/ui/patch_hub` directory.

The reasoning for this different approach, is that `patch-hub` is screen-driven, and each

[3] https://kworkflow.org/content/project_structure.html#components

screen is abstracted to a state of an FSM. For each state, a set of actions with a similar structure happens:

- Display dialog boxes of state.
- Collect (capture) and process (interpret) user input for each dialog box.
- Transition to next state.

Program 3.4 Simplified listing of `src/ui/patch_hub/patch_hub_core.sh`.

```
1    declare -gA screen_sequence
2
3    function patch_hub_main_loop()
4    {
5      while true; do
6        case "${screen_sequence['SHOW_SCREEN']}" in
7          'dashboard')
8            dashboard_entry_menu
9            ;;
10         'lore_mailing_lists')
11           show_lore_mailing_lists
12           ;;
13         'latest_patchsets_from_mailing_list')
14           show_latest_patchsets_from_mailing_list
15           ;;
16         'patchset_details_and_actions')
17           show_patchset_details_and_actions "${screen_sequence['SHOW_SCREEN_PARAMETER']}"
18           ;;
19       esac
20
21       handle_exit "$?"
22     done
23   }
```

Implementing these similar actions for each state on the same source file is not as beneficial as breaking each state of the FSM into small standardized files. Most states of the FSM used in `patch-hub` have a dedicated file in `src/ui/patch_hub`.

As mentioned in Subsection 3.4.1, the execution, eventually, enters the function `patch_hub_main_loop`. From this point until the end of the execution, the feature behaves as an FSM. Program 3.4 is a simplified listing of `src/ui/patch_hub/patch_hub_core.sh`.

Program 3.5 Listing of the `handle_exit`

```
1    function handle_exit()
2    {
3      local exit_status="$1"
4
5      case "$exit_status" in
6        1 | 22 | 255)
7          clear
8          exit 0
9          ;;
10     esac
11   }
```

Notice that each case in the switch-case of Program 3.4 is a state of the FSM, and the value of `screen_sequence['SHOW_SCREEN']` defines the active state. When the flow of

execution enters `patch_hub_main_loop`, it sets the default starting state to `dashboard` and enters a while loop with a conditional equal to the Bash boolean value of `true`, which causes an infinite while loop. The execution never returns from `patch_hub_main_loop`, only exits the shell with the built-in command `exit`[4] through the function `handle_exit`, that is implemented in `src/ui/patch_hub/patch_hub_core.sh` and is listed in Program 3.5.

Calls to `handle_exit` occur at least once at each iteration of the infinite while loop at line 47 of Program 3.4. The function causes the shell to exit if the argument passed is the integer 1, 22, or 255.

Program 3.6 Listing of the `show_dashboard` handler function

```
1    function show_dashboard()
2    {
3     menu_list_string_array=(
4       'Registered mailing list'
5       'Bookmarked patches'
6       'Search in Lore'
7       'Settings'
8     )
9
10    create_menu_options 'Dashboard' '' 'menu_list_string_array'
11    ret="$?"
12
13    [[ "$ret" != 0 ]] && handle_exit "$ret"
14
15    case "$menu_return_string" in
16      0) # Registered mailing list
17        screen_sequence['SHOW_SCREEN']='registered_mailing_lists'
18        ;;
19      1) # Bookmarked patches
20        screen_sequence['SHOW_SCREEN']='bookmarked_patches'
21        ;;
22      2) # Search in Lore
23        screen_sequence['SHOW_SCREEN']='search_string_in_lore'
24        ;;
25      3) # Settings
26        screen_sequence['SHOW_SCREEN']='settings'
27        ;;
28    esac
29   }
```

Each state of the FSM comprises one or more dialog boxes created by the handler functions prefixed with `show_` called inside each case of the switch case of Program 3.4. For example, the state `dashboard` is represented by only one dialog box that shows a menu for the user to choose one option from a list. Figure 3.5 is a screenshot of the single dialog box of the `dashboard` state, and Program 3.6 is the listing of the `show_dashboard` handler function.

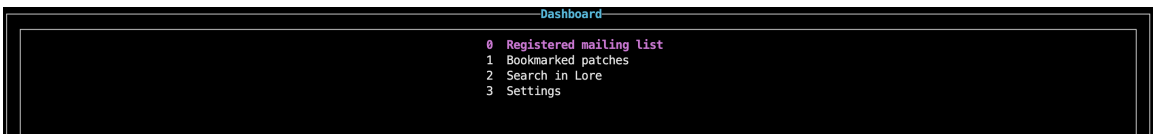


Figure 3.5: *Screenshot of the `patch-hub` Dashboard*

[4] https://man7.org/linux/man-pages/man1/exit.1p.html

The dialog box in Figure 3.5 is created by the `create_menu_options` function from the kw `Dialog` library discussed in Section 3.3. Depending on the user interaction, which can be selecting the button labeled *OK* or the button labeled *Exit* with a specific menu highlighted, the FSM transitions to a different state (or stops execution). In case the user selects the *OK* button, `create_menu_options` returns with a value 0 (this value is captured by `ret="$?"`), and the index of the selected menu in the screen is stored into the variable `menu_return_string`, and the next state in the FSM that represents the chosen menu is set to the variable `screen_sequence['SHOW_SCREEN']`. Then `show_dashboard` returns with a value zero, `handle_exit` is called, but the shell does not exit, and the next state is handled, repeating the loop. In case the user selects the *Exit* button, the return value is one instead of zero, which is used by `show_dashboard` to call `handle_exit` with an argument of 1, which causes the shell to exit and stop the execution of `patch-hub`.

3.4.3 View and Controller components of `patch-hub`

The `patch-hub` feature is implemented in three components that adapt the Model-View-Controller (MVC) architectural design pattern to the problem of creating a hub for the public mailing lists archived on https://lore.kernel.org.

The View component in `patch-hub` is represented by the `src/lib/dialog_ui.sh` file, which contains functions that create dialog boxes, as discussed in Section 3.3. The mentioned file is the View component, as all the display responsibilities discussed in Subsection 3.1.1 of the component are managed using only the functions defined in it. The same file partly represents the Controller component of the feature, given that dialog boxes collect (but do not process) user input.

Other responsibilities of the Controller for processing the user input, if necessary, interacting with the Model component, and updating the View are overseen by the files inside the directory `src/ui/patch_hub`. The functions defined in these files process the user input returned by the dialog boxes, interact with the Model (`src/lib/lore.sh`), if necessary, and creates new dialog boxes. Figure 3.6 is a diagram of the architecture of `patch-hub` considering the MVC pattern.

3.4.4 `patch-hub` execution example

It is important to stress that the FSM computational model and the MVC architectural design pattern implemented by `patch-hub` are complementary, as the abstraction of states for managing the flow of execution in the feature does not conflict with the components architecture of the feature.

The following example illustrates the flow of execution of `patch-hub` concerning the FSM model and MVC pattern. Consider the feature in execution in the state `dashboard`, like in Figure 3.5. At this moment, the execution is waiting for user input (from the MVC perspective, the execution is at the top of Figure 3.6). Suppose the user selects the *Bookmarked patches* menu, which displays a list of the *patchsets* that the user has bookmarked. The user input is collected by the dialog box and processed by the `show_dashboard` function, which transitions the FSM from the `dashboard` to the `bookmarked_patches` state (line 38 of Program 3.6), and the execution enters the `show_bookmarked_patches` function.

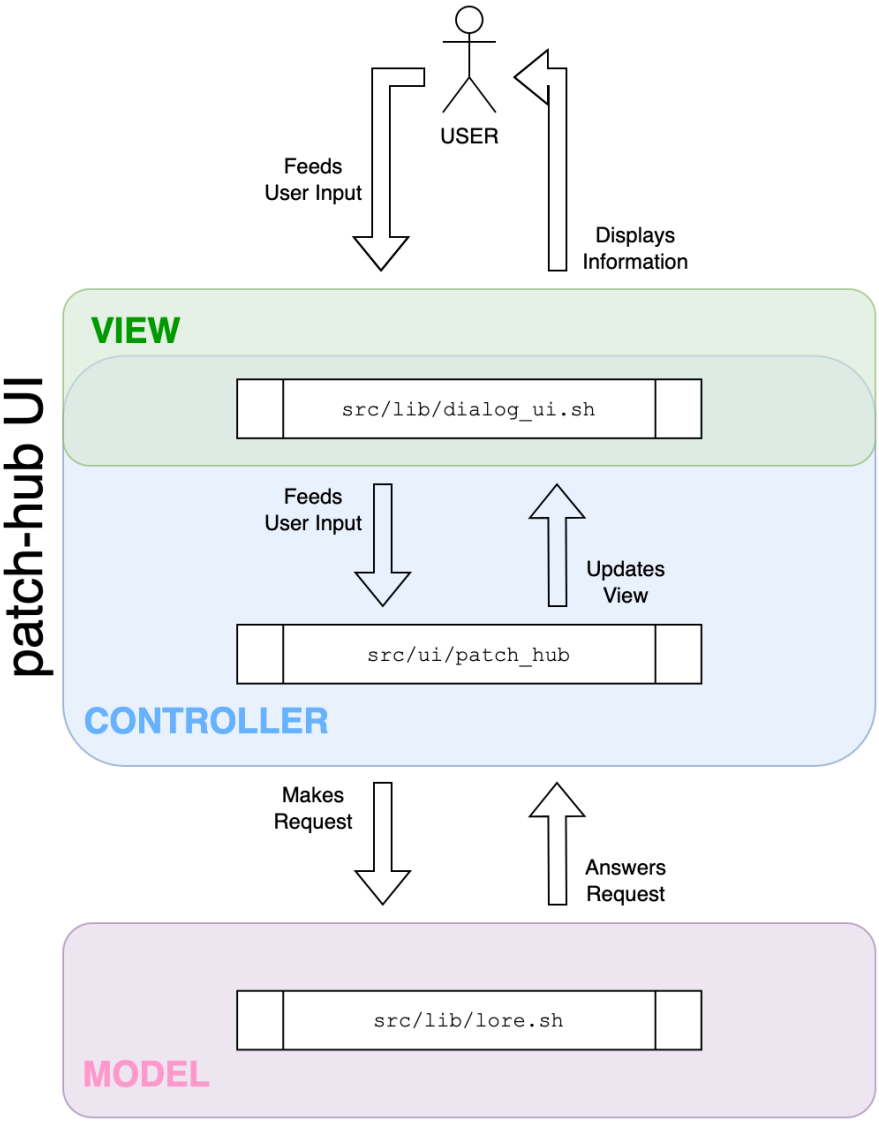


Figure 3.6: *Diagram of the architecture of* `patch-hub` *highlighting components of the MVC pattern.*

Program 3.7 Listing of `show_bookmarked_patches`.

```
1   function show_bookmarked_patches()
2   {
3     local fallback_message
4
5     get_bookmarked_series bookmarked_series
6
7     fallback_message='kw could not find any bookmarked patches.'$'\n'$'\n'
8     fallback_message+='Try bookmarking patches in the menu "Registered mailing list"'
9     list_patches 'Bookmarked patches' bookmarked_series "${fallback_message}"
10  }
```

Program 3.8 Listing of `list_patches`.

```
1   function list_patches()
2   {
3     local menu_title="$1"
4     local -n _target_array_list="$2"
5     local fallback_message="$3"
6
7     if [[ -z "${_target_array_list}" ]]; then
8       create_message_box 'Error' "${fallback_message}"
9       screen_sequence['SHOW_SCREEN']='dashboard'
10      return 2 # ENOENT
11    fi
12
13    create_menu_options "${menu_title}" '' '_target_array_list' '' '' 'Return'
14    ret="$?"
15
16    case "$ret" in
17      0) # OK
18        case "${screen_sequence['SHOW_SCREEN']}" in
19          'latest_patchsets_from_mailing_list')
20            screen_sequence['PREVIOUS_SCREEN']='latest_patchsets_from_mailing_list'
21            screen_sequence['SHOW_SCREEN_PARAMETER']=${list_of_mailinglist_patches["
                  $menu_return_string"]}
22            ;;
23          'bookmarked_patches')
24            screen_sequence['PREVIOUS_SCREEN']='bookmarked_patches'
25            menu_return_string=$((menu_return_string + 1))
26            screen_sequence['SHOW_SCREEN_PARAMETER']=$(get_bookmarked_series_by_index "
                  $menu_return_string")
27            ;;
28        esac
29        screen_sequence['SHOW_SCREEN']='patchset_details_and_actions'
30        ;;
31      1) # Exit
32        handle_exit "$ret"
33        ;;
34      3) # Return
35        screen_sequence['SHOW_SCREEN']='dashboard'
36        ;;
37    esac
38  }
```

Program 3.7 is the listing of the `show_bookmarked_patches` function, and Program 3.8 is the listing of the `list_patches` helper function used by the former.

At `show_bookmarked_patches`, the `get_bookmarked_patches` call happens, triggering a request to the Model component, which can result in one of two types of response:

1. An empty list, i.e., there are no *patchsets* bookmarked by the user.
2. A list with the *patchsets* bookmarked by the user.

Nevertheless, the execution returns from the Model and calls `list_patches`. In case number (1), the test in line 8 of Program 3.8 is successful, and the dialog box of Figure 3.7 is displayed. Then, when the user selects *OK*, a transition to the `dashboard` state occurs.

The test in line 8 of Program 3.8 fails for case number (2). As such, a dialog box with a list of the bookmarked *patchsets* (Figure 3.8) is shown to the user. The user can select the *OK* button with a specific bookmarked *patchset* highlighted, which triggers a transition

Figure 3.7: *Dialog box displayed when there are no bookmarked patchsets.*

Figure 3.8: *Dialog box displayed with list of bookmarked patchsets.*

to the state `show_patchset_details_and_actions`, or select the *Return* button, that triggers a transition to `dashboard`, instead, or even select the *Exit* button, stopping the execution of the feature.

Chapter 4

`patch-hub` Model

As discussed in Chapter 3, two major components constitute the architecture of `patch-hub`. One of them is the `patch-hub` Model. The `patch-hub` Model manages all data about the problem domain. Some of its responsibilities are:

- Fetching and manipulating data from the Linux mailing lists archives hosted at https://lore.kernel.org.
- Managing the database of domain-related data.
- Providing data to the UI for presentation.

This chapter is dedicated to describing the `patch-hub` Model, and it is divided into three sections:

1. Background concepts: overviews of the Model role of the MVC architectural design pattern and Web applications.
2. Description of the Lore archives API.
3. Description of the implementation of `patch-hub` Model.

4.1 Background

4.1.1 The Model role of MVC

The Model component can be described as the non-visual object containing all data and behavior not used for the UI (Fowler, 2012). This component represents the core of the application as it incorporates every domain logic, like business rules and database interactions.

The Model being a critical component in the architecture, it is beneficial to enforce the distinction between UI and Model, presented in Subsection 3.1.1, for better testability and to decouple the presentation and user interaction from domain logic. Automated tests are more straightforward to implement and more efficient to run because there is no need for mocking visual components or user interaction. The quality of the tests is also

enhanced because they make more fundamental assertions as they check the actual inputs and outputs of the Model, i.e., the requests and responses of the Model. The decoupling of the UI and the Model streamlines the development of the Model while also making it easier to add new Views or entire UIs. That is the rationale behind adapting the original MVC pattern by combining the View and Controller components while keeping a well-defined interface between the UI and the Model (see Figure 3.6).

To demonstrate the Model role, consider a Web application. This application exposes a single URL that, when visited, serves the user with an HTML page with a question and a field to input the answer to the question. For this event to happen, the application takes some actions. First, when the Web request to the URL arrives at the server, the Controller component processes it and requests the information needed to the Model, which, in this case, is the question to be presented to the user. Supposing several questions are available in the application database, the Model decides which question to return and interacts with the database to fetch it. Finally, it responds to the Controller requisition with the question, and the Controller requests to the View the creation and presentation of the HTML page.

Expanding upon this illustration, suppose that when the user submits the answer, the application posts the answer to a web forum. The Controller collects and processes the answer and requests the Model to post it in the forum. The Model, in turn, uses the forum API to post the answer directly and then responds to the Controller with some information (like the success or failure of the operation).

In the example, notice how the Model only needs to follow the API between it and the Controller, surmised by the requests and responses made to and from the Model. Besides, the Model encapsulates every action related to the domain logic, like fetching the question and posting the answer.

4.1.2 Web applications

The World Wide Web

The World Wide Web, commonly known as the Web, is a distributed application that uses the client-server architecture, with many public servers accessible through the Internet that serve to requesting clients webpages that reference other public servers [1].

The client-server architecture is a distributed application structure, which divides the application into two processes, the client and the server process [2], that communicate and synchronize their actions using an Inter-Process Communication (IPC) mechanism (Kurose and Ross, 2012). In the context of the Web, the server process is a perpetual process that waits for requests from client processes and responds them with a status code and, optionally, a document called a webpage.

In Web applications, the client and server processes are designed to be run on different machines connected to the Internet. By using an Internet Protocol (IP) address and port

[1] https://developer.mozilla.org/en-US/docs/Glossary/World_Wide_Web

[2] A process is an instance of a computer program that is being executed. A program is a static entity, while the process is a dynamic one.

number the processes are accessible, no matter where the machines physically are. It is important to distinct the Internet from the Web, as the former is an application that uses the Internet infrastructure to operate.

The most common type of webpages are HTML files, which use Hypertext to reference other webpages in, potentially, other Web servers. As such, the application name, *The Web*, can be understood as an analogy to a spider web, in which the nodes on the web are the many servers directly accessible through unique identifiers, and the edges are the references in the webpages served that interconnect the servers.

Web clients usually run on the end-users devices, like desktops, laptops, mobile devices, and the like, while Web servers often run on datacenters scattered around the world. Because servers are, in theory, perpetual processes that are always waiting for request or handling requests, the Web operates on-demand, meaning that end-users can access resources at any time.

The HTTP Protocol

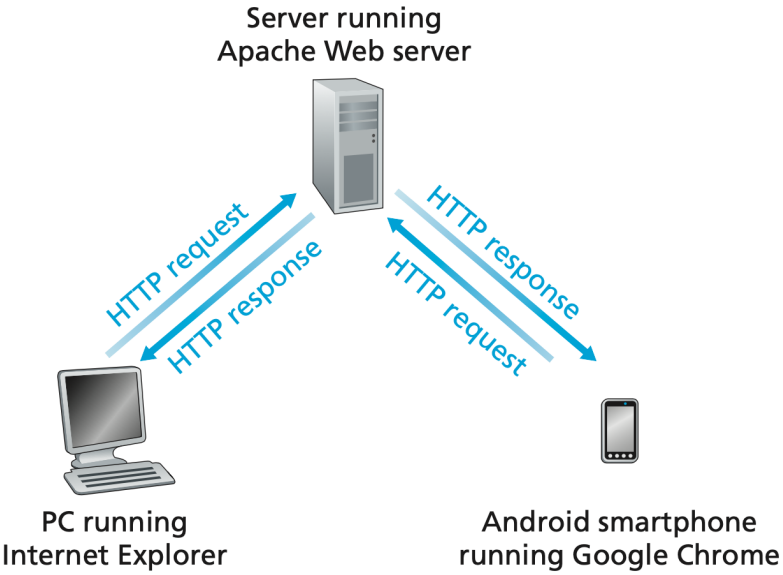


Figure 4.1: *Diagram exemplifying the role of the HTTP protocol in a Web application (source: Kurose and Ross, 2012).*

Although through the Internet infrastructure, the client and server processes of a Web application are addressable, the standardized communication that establishes well-defined requests and responses between the processes that make the Web viable is the HTTP Protocol. HTTP is the fundamental application layer protocol of the Web, in the five-layer model (Kurose and Ross, 2012). In every communication between the Web client and the Web server, there is an exchange of HTTP messages. There are two types of HTTP messages: request messages and response messages. Figure 4.1 is a diagram that exemplifies the role of HTTP in a Web application.

HTTP request messages are sent from the Web client to the Web server. Figure 4.2 is a

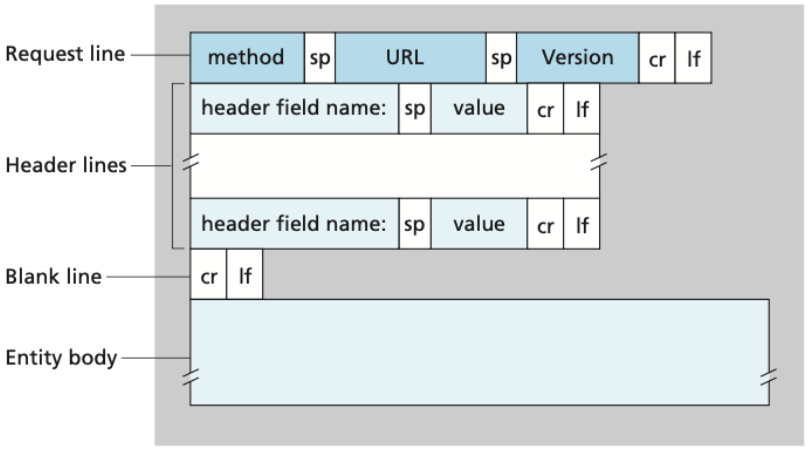


Figure 4.2: *General format of an HTTP request message (source: Kurose and Ross, 2012).*

Program 4.1 Example of HTTP request message.

```
1  GET /path/to/resource HTTP/1.1
2  Host: www.website.com
3  Connection: close
4  Accept-language: pt-BR
```

diagram of the format of HTTP request messages, and Program 4.1 is an example of an HTTP request.

HTTP request methods define the types of requests that a Web client can make. Standard HTTP methods are the GET method to retrieve whatever information (in the form of an entity) the resource represents, the POST and PUT to insert data through the message body in the Web server, and the HEAD method that generates the same HTTP response as the request with a GET method would, but without the actual resource.

Program 4.2 Example of HTTP response message.

```
1  HTTP/1.1 200 OK
2  Connection: close
3  Date: Fri, 17 Nov 2023 13:05:04 GMT
4  Server: Apache/2.2.3 (CentOS)
5  Last-Modified: Fri, 17 Nov 2023 13:05:04 GMT
6  Content-Length: 6821
7  Content-Type: text/html
8
9  (data data data data data ...)
```

After receiving and interpreting request messages from the client, the server responds with an HTTP response message. Figure 4.3 is a diagram of the format of HTTP response messages, and Program 4.2 is an example of an HTTP response.

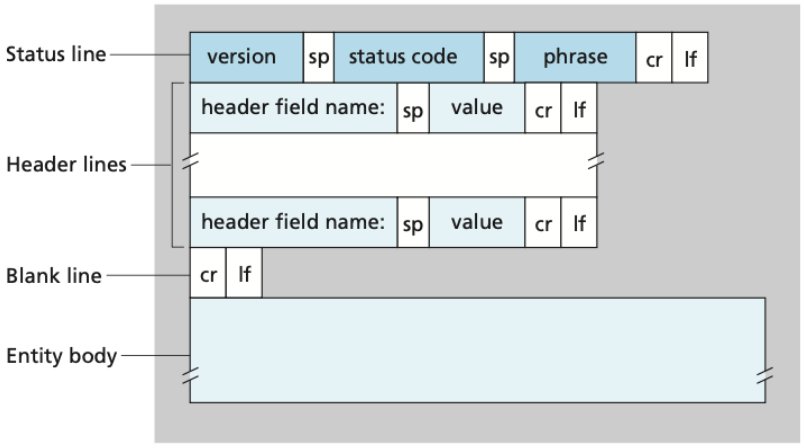


Figure 4.3: *General format of an HTTP response message (source: Kurose and Ross, 2012).*

Status codes of HTTP responses are three-digit positive integers that are the result of the server attempting to process and satisfy the HTTP request. For example, the status code and its explanatory phrase `200 OK` indicate that the server has successfully understood, accepted, and handled the request.

Query Strings

Even though the POST method is intended to pass data to the server in an HTTP request using its message body, this can also be achieved by using *query strings*. These strings are appended to the end of a base URL to pass data from the Web client to the Web servers using a GET or HEAD method. The question mark character `'?'` marks the start of a query string. The string itself consists of pairs of query parameters and values that are separated by the ampersand character `&` (parameter and value are separated by the equal sign character `'='`). To illustrate, Program 4.3 is a query string that assigns the values cat and yellow to the parameters animal and color, respectively, for the base URL https://url.com/resource.

Program 4.3 Example of query string that assigns cat and yellow to animal and color parameters, respectively.

```
1  https://url.com/resource?animal=cat&color=yellow
```

Web application API

As servers from Web applications are public and accessible through the Internet, not only can users make requests through Web browsers, but other applications can use the services of a Web application by consuming its API. In this context, a Web application API refers to the HTTP requests and responses interface that a Web application offers to other applications, no matter if this interface was created with the intent of being used by other developers to build applications or not.

An example of Web API intended to be used as a component for other applications is the Marvel Comics API [3], which demands an account creation to be used. It has extensive documentation that comprehensively explains how to make requests and interpret responses. Figure 4.4 is a screenshot of interactive documentation that lists the available endpoints (accessible URLs for Web clients) and provides a way to build URLs with query strings (Figure 4.5).

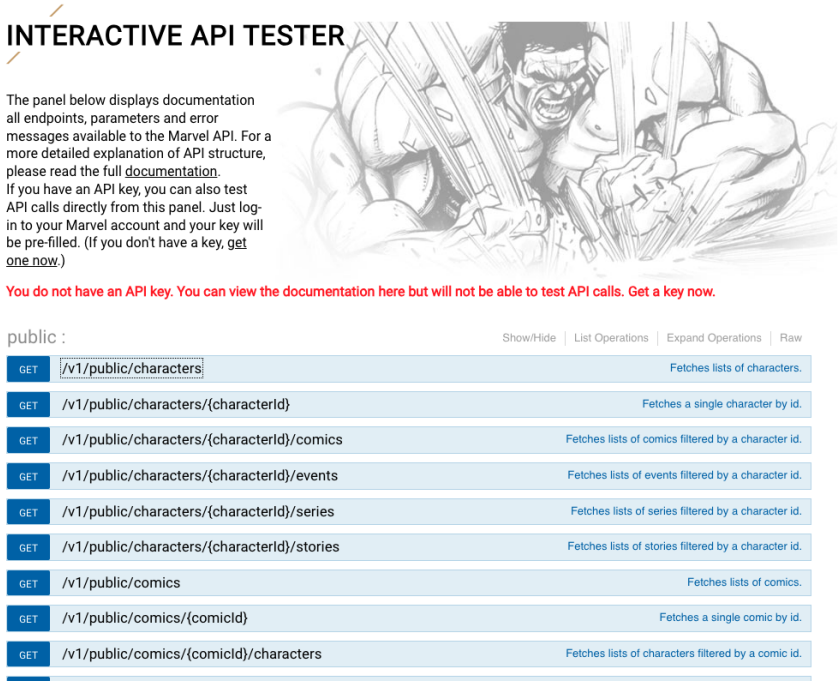


Figure 4.4: *Screenshot of Marvel Comics API interactive documentation.*

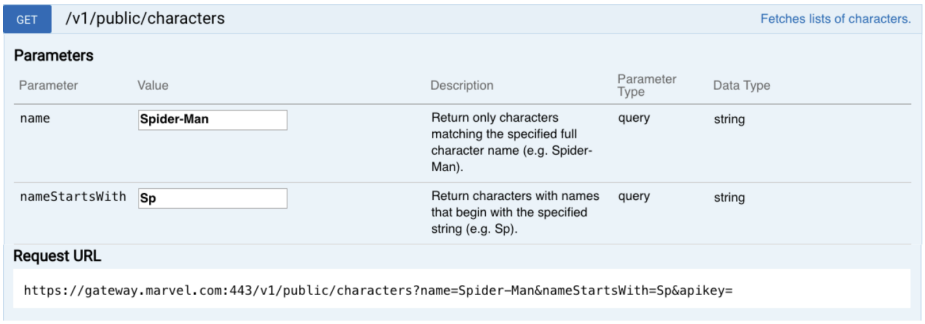


Figure 4.5: *Example of custom URL built with Marvel Comic API Interactive Documentation.*

There are also Web applications that provide little to no support for other developers, both in terms of documentation and the service not being designed to be used as a component by other applications. Considering that there are no legal infringements, it is possible to use reverse engineering to understand the API of most Web applications, as one can use the intended Web client (a Web browser, for instance) to use the Web service and experiment with the requests and responses of the interactions.

[3] https://developer.marvel.com/

4.2 Lore archives API

Lore offers an API [4] for requesting data about the archived mailing lists and the mails, called *messages*, that flow through them. More specifically, the API can be used for listing available mailing lists and messages of a given mailing list. For both types of listings, arguments can be passed to produce more specific queries based on time, author, string matching, and others. Individual messages are also downloadable as *mbox* [5] files using Lore API.

From the perspective of HTTP messages and considering the context of the `patch-hub` feature, the GET request method is used for all HTTP requests. Additional information is passed to a request with query strings instead of using the POST method. HTTP responses contain webpages in the form of HTML files, Atom feeds in the form of XML files, or individual messages in the form of mbox files.

Lore API has three main endpoints:

- https://lore.kernel.org or https://lore.kernel.org/lists.html: request listings of the available mailing lists archived on Lore.
- https://lore.kernel.org/<mailing-list>: request listings of the messages of an available mailing list with the name `mailing-list`.
- https://lore.kernel.org/<mailing-list>/<message-id>/raw: request mbox file of individual message, in which `message-id` is the identifier of a message on the `mailing-list` list.

Figure 4.6 is a diagram delineating the three types of requests and responses of Lore API that are used in the context of `patch-hub`, and Figures 4.7 are screenshots of each type of response viewed in a Web browser.

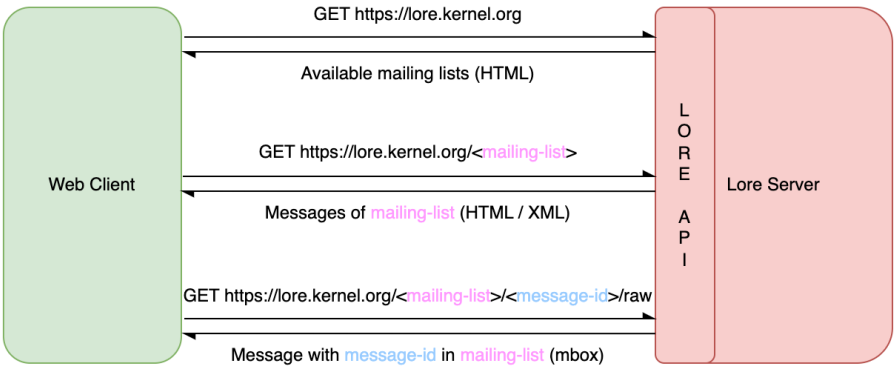
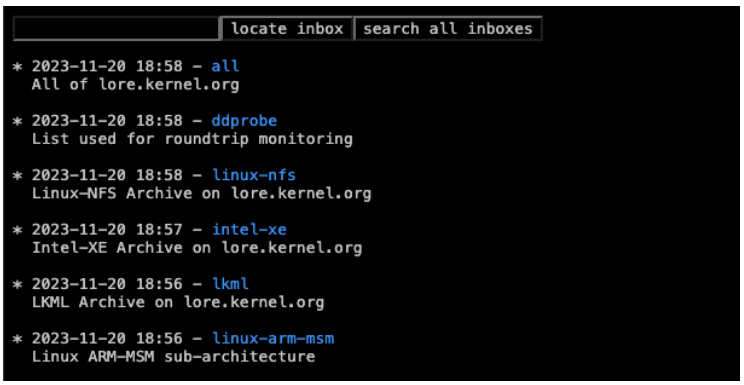


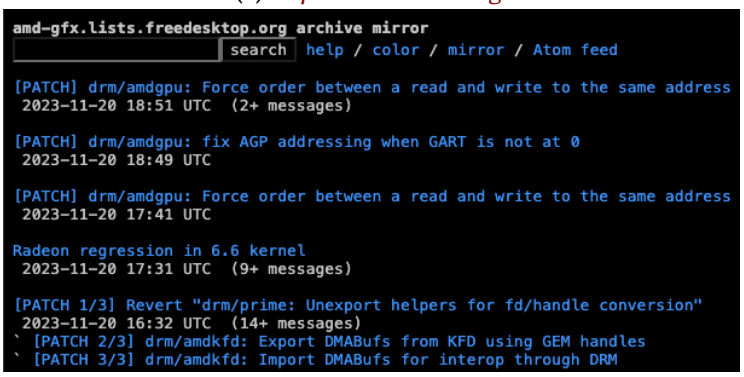
Figure 4.6: *Diagram delineating the three types of requests and responses of Lore API that occur in `patch-hub`.*

[4] The documentation on the API is scarce, so the description exposed in this section is based on testing and experimenting with the Lore archives.

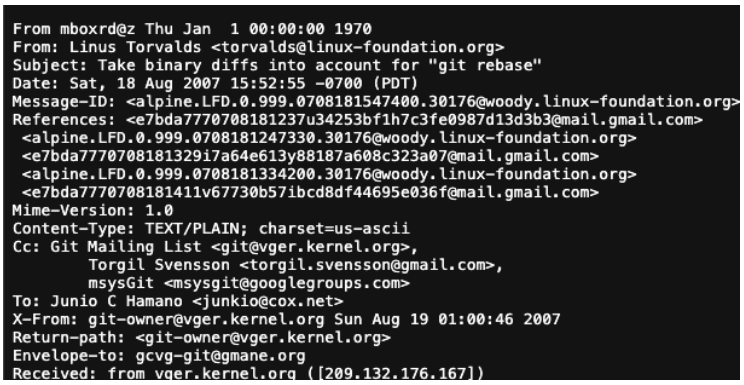
[5] Mbox is a standard format for storing messages. It is usually used for email files.



(a) *https://lore.kernel.org.*



(b) *https://lore.kernel.org/amd-gfx.*



(c) *https://lore.kernel.org/git/alpine.LFD.0.999.0708181547400.30176@woody.linux-foundation.org/raw.*

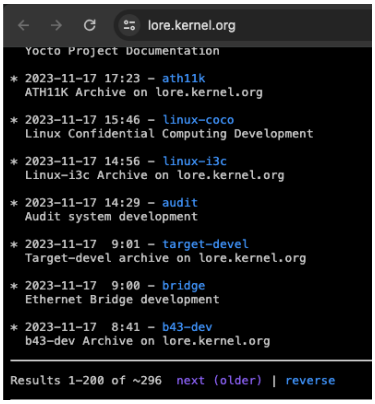
Figure 4.7: *Screenshots of the three types of responses from Lore API viewed in a Web browser.*

4.2.1 Query parameters

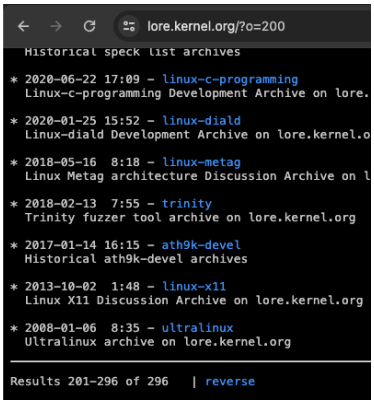
As mentioned earlier, arguments for listing requests (this does not apply to individual messages) to the Lore API are passed using query strings that are appended to the base URL of the request. This subsection describes the query parameters used in the context of the `patch-hub` feature.

Query parameter `o`

Lore API responses with listings are paginated. Pages have 200 entries at maximum, and the query parameter `o` value determines the starting index of the returned listing. Figure 4.8a is a screenshot of the first page of the available mailing lists, while Figure 4.8b is the second page. Notice at the bottom of the screenshots that the range of results corresponds to the value of the query parameter `o` (in the case of the first page, the value of `o` defaults to 0). This pagination also occurs when listing the messages of a given mailing list.



(a) *First page of available mailing lists.*

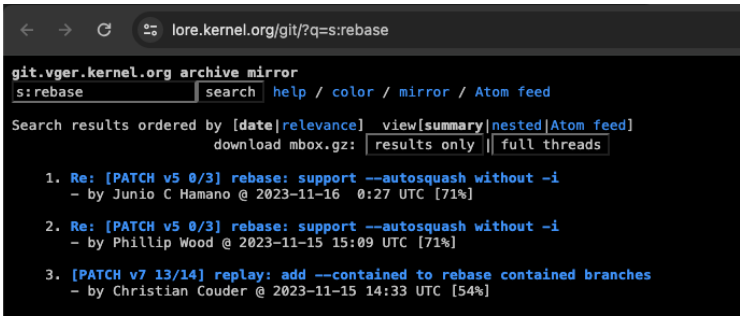


(b) *Second page of available mailing lists.*

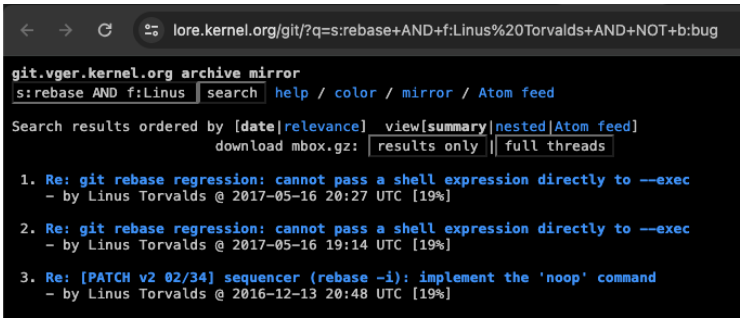
Figure 4.8: *Screenshots illustrating the pagination of Lore API responses.*

Query parameter `q`

The query parameter `q` filters messages from a given mailing list that fulfill certain conditions. In the form of `q=<string>`, this parameter filters messages that have a string match with `string` in any field of the message. Furthermore, Lore provides support for more fine-grained filtering based on specific fields of the message with the use of prefixes [6]. Common search engine operators like `AND`, `OR`, `+`, and `-` are also supported. For illustration, Figure 4.9a is a listing of the `git` mailing list filtering messages that have the term *rebase* in the subject field, while Figure 4.9b additionally filters messages sent by Linus Torvalds and that do not contain the term *bug* in the message body.



(a) *Messages from the `git` mailing list that contain the term rebase in the subject.*



(b) *Messages from the `git` mailing list that contain the term rebase in the subject, that were sent by Linus Torvalds and that do not have the term bug in the body.*

Figure 4.9: *Screenshots illustrating filtering of messages in Lore API.*

```
<?xml version="1.0" encoding="us-ascii"?>
<feed>
  <entry>
    <author><name>Linus Torvalds</name><email>torvalds@linux-foundation.org</email></author>
    <title>Re: git rebase regression: cannot pass a shell expression directly to --exec</title>
    <link href="http://lore.kernel.org/git/CA+55aFwX2RDwT=g4w55bbLV38ggaVpdcLpMcCvqMcKRziUSesQ@mail.gmail.com/"/>
    <content type="xhtml">…
    </content>
  </entry>
  <entry>
    <author><name>Linus Torvalds</name><email>torvalds@linux-foundation.org</email></author>
    <title>Re: git rebase regression: cannot pass a shell expression directly to --exec</title>
    <link href="http://lore.kernel.org/git/CA+55aFwB-MWASj7dZWkXWhgd4gvEfoOhL6Fo7kXeJSm9dht4Jg@mail.gmail.com/"/>
    <content type="xhtml">…
    </content>
  </entry>
  <entry>
    <author><name>Linus Torvalds</name><email>torvalds@linux-foundation.org</email></author>
    <title type="html">Re: [PATCH v2 02/34] sequencer (rebase -i): implement the &amp;#39;noop&amp;#39; command</title>
    <link href="http://lore.kernel.org/git/CA+55aFzxFFNY+dL6s7dLZeVXBsBKD0aeof5Bj2wcD1CpefVSAA@mail.gmail.com/"/>
    <content type="xhtml">…
    </content>
  </entry>
</feed>
```

Figure 4.10: *Simplified correspondent Atom feed of listing shown on Figure 4.9b.*

Query parameter x

When requesting the listing of messages from a mailing list, the exact itemization can be responded as an HTML webpage or as an XML Atom feed. The former behavior is the default, and the latter is achieved by setting the `x` query parameter to `A`. Even though the sequence of messages is the same in both types of responses, messages in the XML Atom feeds contain more attributes like author email address, timestamp, and message ID. Figure 4.10 is the correspondent simplified Atom feed of the listing shown on Figure 4.9b.

4.3 `patch-hub` Model implementation

As stated in Subsection 4.1.1, the Model component contains all application logic unrelated to the View or Controller. `patch-hub` Model comprises a few data structures used at runtime and several functions that manipulate domain-specific data or encapsulate some domain logic. These functions and data structures are organized in the library file `src/lib/lore.sh`, which is relatively different from other library files in the kw project. Due to the data structures that hold information about the state of the execution, `src/lib/lore.sh` is not just like a collection of functions with a shared context. Program 4.4 is a listing with the declaration of the mentioned data structures. Notice how these structures are declared as global variables to emulate the attributes of an object from the *Object-Oriented Programming* paradigm. As Bash does not have *Lexical Scoping* [7], the only way to represent a state that is manipulated by multiple function calls is through global variables.

Program 4.4 `patch-hub` Model data structures declared in `src/lib/lore.sh`.

```
1    # List of available mailing list in Lore
2    declare -gA available_lore_mailing_lists
3
4    # List of patchsets from a target mailing list. Each entry is a string with
5    # the following attributes separated by 'SEPARATOR_CHAR':
6    #
7    # author-name,author-email,version,total-patches,patchset-title,message-ID
8    declare -gA mailing_list_patchsets
9
10   # Number of patchsets processed in current Lore fetch session.
11   # Also, the size of 'mailing_list_patchsets'.
12   declare -g PATCHSETS_PROCESSED=0
13
14   # Value of query parameter 'o' from the Lore API that determines the minimum
15   # excluding entry index of the page. This value is associated with the
16   # current Lore fetch session.
17   declare -g MIN_INDEX=0
```

From the perspective of tasks that `patch-hub` Model has to accomplish, the itemization below surmises its responsibilities:

[6] Supported filters can be checked at https://lore.kernel.org/amd-gfx/_/text/help (this is the help page related to the `amd-gfx` list, but all lists support the same set of filters).

[7] Lexical scoping, also called *Static Scoping*, dictates that free variables in a procedure are taken to refer to bindings made by enclosing procedure definitions; that is, they are looked up in the environment in which the procedure was defined (Abelson and Sussman, 1996).

- Listing available mailing lists.
- Listing patchsets of a mailing list.
- Handling individual patchsets.
- Managing feature configurations.

The following subsections explain in depth what each task entails and how `patch-hub` Model is implemented to accomplish them.

4.3.1 Listing available mailing lists

To list the available mailing lists archived on Lore, `patch-hub` Model makes an HTTP GET request to the endpoint `https://lore.kernel.org`. This request is responded with an HTML file containing the record of the available mailing lists. `patch-hub` Model processes this HTML file and stores the processed mailing lists in the `available_lore_mailing_lists` global array. Program 4.5 is a listing of the `retrieve_available_mailing_lists` function that encapsulates this logic, and Program 4.6 is the implementation of the `download` function defined in `src/lib/web.sh`.

Program 4.5 Implementation of `retrieve_available_mailing_lists`.

```
1   function retrieve_available_mailing_lists()
2   {
3     local flag="$1"
4     local index=''
5     local pre_processed
6
7     flag=${flag:-'SILENT'}
8
9     setup_cache
10
11    download "$LORE_URL" "$MAILING_LISTS_PAGE" "$CACHE_LORE_DIR" "$flag" || return "$?"
12
13    # Note: "$LIST_PAGE_PATH" expands to "${CACHE_LORE_DIR}/${MAILING_LISTS_PAGE}"
14    pre_processed=$(sed -nE -e 's/^href="(.*)\/?">\1<\/a>$/\1/p; s/^ (.*)$/\1/p' "${LIST_PAGE_PATH}
          ")
15
16    while IFS= read -r line; do
17      if [[ -z "$index" ]]; then
18        index="$line"
19      else
20        available_lore_mailing_lists["$index"]="$line"
21        index=''
22      fi
23    done <<< "$pre_processed"
24  }
```

From the architectural aspect, the Controller component (that is part of `patch-hub` UI) requests to the Model a listing of the archived mailing lists using a call to `retrieve_available_mailing_lists`, that returns the lists through `available_lore_mailing_lists`; then, the Controller requests the View component to display this information using the same data structure.

Program 4.6 Implementation of `download`.

```
1   function download()
2   {
3     local url="$1"
4     local output=${2:-'page.xml'}
5     local output_path="$3"
6     local flag="$4"
7
8     if [[ -z "$url" ]]; then
9       complain 'URL must not be empty.'
10      return 22 # EINVAL
11    fi
12
13    flag=${flag:-'SILENT'}
14
15    output_path="${output_path:-${KW_CACHE_DIR}}"
16
17    cmd_manager "$flag" "curl --silent '$url' --output '${output_path}/${output}'"
18  }
```

4.3.2 Listing patchsets of a mailing list

To list patchsets of a target mailing list, `patch-hub` Model uses an analog approach to the one used to list the available mailing lists. Overall, it composes and makes an HTTP GET request to a Lore endpoint that is returned with a resource containing a list of entries, which are then processed and stored in a global array. Nevertheless, each step is considerably more complex when compared to listing the available mailing lists. Program 4.7 is a listing of `fetch_latest_patchsets_from`, which is the function that implements this behavior by dividing this task into subtasks and delegating responsibilities to other functions.

Overview of fetching latest patchsets

The first thing to notice in Program 4.7 is that the function is essentially a while loop that repeats a set of subtasks until the value `page` times `patchsets_per_page` is greater than `PATCHSETS_PROCESSED`. As context, `patch-hub` UI displays the latest patchsets of a target mailing list using pagination (this pagination is not the same as the one done by the Lore API mentioned in Subsection 4.2.1). The user can request older pages, as well as go back to previous rendered pages in this *fetch session*, so `patch-hub` UI, using `fetch_latest_patchsets_from`, requests a specific range of patchsets for `patch-hub` Model by passing the `page` and the `patchsets_per_page` arguments. The variable `PATCHSETS_PROCESSED` is one of the global variables used by `patch-hub` Model (Program 4.4) to store the number of patchsets that have already been processed in the current fetch session, to avoid fetching patchsets that were already processed.

The end of a fetch session is the resetting of the data structures that compose the state of the session, and it occurs when `patch-hub` UI uses the function `reset_current_lore_fetch_session` defined on `src/lib/lore.sh` that is listed in Program 4.8.

Program 4.7 Implementation of `fetch_latest_patchsets_from`.

```
function fetch_latest_patchsets_from()
{
  local target_mailing_list="$1"
  local page="$2"
  local patchsets_per_page="$3"
  local additional_filters="$4"
  local flag="$5"
  local raw_xml
  local lore_query_url
  local xml_result_file_name
  local pre_processed_patches
  local xml_result_file_name
  local lore_query_url
  local raw_xml
  local ret

  flag=${flag:-'SILENT'}
  xml_result_file_name="${target_mailing_list}-patches.xml"

  while [[ "$PATCHSETS_PROCESSED" -lt "$((page * patchsets_per_page))" ]]; do
    lore_query_url=$(compose_lore_query_url_with_verification "$target_mailing_list" "$MIN_INDEX"
          "$additional_filters")
    ret="$?"
    [[ "$ret" != 0 ]] && return "$ret"

    download "$lore_query_url" "$xml_result_file_name" "$CACHE_LORE_DIR" "$flag"
    ret="$?"
    [[ "$ret" != 0 ]] && return "$ret"

    if is_html_file "${CACHE_LORE_DIR}/${xml_result_file_name}"; then
      return 22 # ENOENT
    fi

    raw_xml=$(< "${CACHE_LORE_DIR}/${xml_result_file_name}")

    if [[ "$raw_xml" == '</feed>' ]]; then
      break
    fi

    pre_processed_patches=$(pre_process_xml_result "${CACHE_LORE_DIR}/${xml_result_file_name}")
    process_patchsets "$pre_processed_patches"

    MIN_INDEX=$((MIN_INDEX + LORE_PAGE_SIZE))
  done
}
```

Program 4.8 Implementation of `reset_current_lore_fetch_session`.

```
function reset_current_lore_fetch_session()
{
  mailing_list_patchsets=()
  PATCHSETS_PROCESSED=0
  MIN_INDEX=0
}
```

Building the request URL

The response of a request for the base URL `https://lore.kernel.org/<mailing-list>` is an HTML file with a list of message threads ordered by their received time on the Lore servers (Figure 1.4 is an example of this of response). Two characteristics of this type of response require `patch-hub` Model to adapt the request: the items in the list returned do not contain all the information needed by `patch-hub`, and discussion threads and replies are not filtered out. Another point to be considered is the pagination of listing responses by the Lore API.

To overcome these three complications, `patch-hub` Model uses the Lore API query parameters mentioned in Subsection 4.2.1 to build the request URL. The URL is built by the function `compose_lore_query_url_with_verification` (listed on Program 4.9) and captured by the variable `lore_query_url`. Notice on line 17 of Program 4.9 three assignments:

- `x=A` to request an Atom feed XML that contains the information needed by `patch-hub` for each patchset on the returned list.
- `o=$min_index` to set the earliest message of the page requested, managing the pagination of Lore response.
- `q=rt:..+AND+NOT+s:Re` to filter out reply messages. The parameter `q` accepts the composition of filters with the keyword `AND`. The prefix `s:` filters based on the subject of the message, so `NOT+s:Re` filters out messages that have the string `Re` on the subject, as reply messages subjects practically always start with this string. The prefix `rt:` filters based on the received time of the message accepting values as time ranges, with `rt:..` representing a time range that is open-ended on both ends; this is redundant as it means "messages that were received at any time", but it is used because Lore API does not accept a `NOT` keyword as the start of a value for parameter `q`.

Program 4.9 Implementation of `compose_lore_query_url_with_verification`.

```
1   function compose_lore_query_url_with_verification()
2   {
3     local target_mailing_list="$1"
4     local min_index="$2"
5     local additional_filters="$3"
6     local query_filter
7     local query_url
8
9     if [[ -z "$target_mailing_list" || -z "$min_index" ]]; then
10      return 22 # EINVAL
11    fi
12
13    if [[ ! "$min_index" =~ ^-?[0-9]+$ ]]; then
14      return 22 # EINVAL
15    fi
16
17    query_filter="?x=A&o=${min_index}&q=rt:..+AND+NOT+s:Re"
18    [[ -n "$additional_filters" ]] && query_filter+="+AND+${additional_filters}"
19    query_url="${LORE_URL}/${target_mailing_list}/${query_filter}"
20    printf '%s' "$query_url"
21  }
```

Take note of the optional argument `additional_filters` that is present in both `fetch_latest_patchsets_from` and `compose_lore_query_url_with_verification`. This argument can be used to apply any other filter supported by the query parameter `q` to the request, and, at the moment, `patch-hub` uses it to search arbitrary strings in the Lore archives.

Making the HTTP request and verifying the response

After building the correct URL, `patch-hub` makes the HTTP GET request to the Lore API (line 25 of Program 4.7), which responds with a resource. Then, there are two checks to account for special cases: the first to confirm that the returned resource is not an HTML file, which is responded when the request could not be handled, and the second to assert that it is not an empty XML file, which is responded when the value of the query parameter `o` is greater or equal than the index of the oldest message in the archive (i.e., there are no more patches to fetch from the target mailing list). If the first check fails, the function returns with an error code that the Controller can handle. If the second check fails, the `break` keyword is used to get out of the while loop, as having no patches left to fetch means that the function can return even though the number of patchsets processed did not reach the value required by the loop condition. In the common case, the request is correctly handled by Lore API, and the resource responded is an XML Atom feed that represents a single page of the target mailing list.

Processing patchsets

As mentioned in Section 1.3, we consider the word *patch* as an individual message in a mailing list containing code differentials. In contrast, *patchset* is a set of patches that are supposed to have a shared context. Also, for each patchset, a single message is elected as the representative of the series, usually the first message [8]. However, the listing returned by the HTTP GET request in the previous step is of patches, not patchsets. The design of `patch-hub` considers patchsets as the feature unit. Hence, the `patch-hub` Model has to process the response to consider only representative messages and to get the necessary information for them.

Program 4.10 Implementation of `pre_process_xml_result`.

```
1    function pre_process_xml_result()
2    {
3       local xml_file_path="$1"
4       local xpath_query
5       local raw_xml
6       local -r NAME_EXP='//entry/author/name/text()'
7       local -r EMAIL_EXP='//entry/author/email/text()'
8       local -r TITLE_EXP='//entry/title/text()'
9       local -r LINK_EXP='//entry/link/@href'
10
11      raw_xml=$(< "$xml_file_path")
12      xpath_query="${NAME_EXP}|${EMAIL_EXP}|${TITLE_EXP}|${LINK_EXP}"
13      printf '%s' "$raw_xml" | xpath -q -e "$xpath_query"
14   }
```

[8] In this case, we use *message* and not *patch*, because the first patch of a patchset can be a *cover letter*, which is an introductory message without code differentials that comments and delineates the context of the series.

On line 39 of Program 4.7, the XML resource with the list of patches is pre-processed using the function `pre_process_xml_result`, which consists of trimming the unnecessary information of the XML into the variable `pre_processed_patches`. Program 4.10 is a listing of `pre_process_xml_result`, that shows the use of the `xpath` tool [9] to filter four attributes of patches in the XML returned: author name and email, patch title, and URL of resource in the Lore archives (this is interchangeable with the message ID). It is important to point that `pre_process_xml_result` returns the same list of patches returned by the Lore API, but trimmed.

Program 4.11 Template of XML result of mailing list patches and correspondent pre-processed version.

```
1    # Simplified template of XML returned by Lore
2    <?xml version="1.0" encoding="us-ascii"?>
3    <feed>
4     <entry>
5      <author>
6       <name>AUTHOR-NAME</name>
7       <email>AUTHOR-EMAIL</email>
8      </author>
9      <title>MESSAGE-SUBJECT</title>
10     <updated>RECEIVED-TIME</updated>
11     <link href="MESSAGE-ID"/>
12     <id>...</id>
13     <thr:in-reply-to .../>
14     <content>...</content>
15    </entry>
16    <entry>
17    ...
18    </entry>
19    ...
20   </feed>
21
22   # Pre-processed version of XML returned by Lore
23   AUTHOR-NAME
24   AUTHOR-EMAIL
25   MESSAGE-SUBJECT
26   href="MESSAGE-ID"
27   ...
```

As an illustration, Program 4.11 is the general pattern of the XML file returned by Lore when requesting the listing of a target mailing list as an Atom feed with the correspondent pre-processed version.

On line 40 of Program 4.7, `process_patchsets` uses the pre-processed list of patches to generate the correspondent list of patchsets and stores it in the global array `mailing_list_patchsets`. Conceptually, `process_patchsets` parses the list of pre-processed patches, finding representative messages of patchsets. For each representative message, the function launches a *background process* [10] that parallelizes the processing of patchsets. The processed patchsets are stored in a temporary directory, and after all background processes finish execution, the processed patchsets are loaded into the `mailing_list_patches` data structure.

[9] Tool for querying *XPath* statements in XML files (for reference, see https://www.w3.org/TR/xpath/).

[10] A Bash command terminated with the ampersand character & executes asynchronously in a subshell (for reference, see https://www.gnu.org/software/bash/manual/bash.html#Lists).

Program 4.12 Implementation of `process_patchsets`

```
1     function process_patchsets()
2     {
3        local pre_processed_patches="$1"
4
5        shared_dir_for_parallelism=$(mktemp --directory)
6        starting_index="$PATCHSETS_PROCESSED"
7        count=0
8        i=0
9
10       while IFS= read -r line; do
11          if [[ "$line" =~ ^[[:space:]]href= ]]; then
12             patch_url=$(str_get_value_under_double_quotes "$line")
13
14             if is_introduction_patch "$patch_url"; then
15                thread_for_process_patch "$PATCHSETS_PROCESSED" "$shared_dir_for_parallelism" "
                       $processed_patchset" "$patch_url" "$patch_title" &
16                pids[i]="$!"
17                ((i++))
18                ((PATCHSETS_PROCESSED++))
19             fi
20
21             processed_patchset=''
22             count=0
23             continue
24          fi
25
26          case "$count" in
27             0) # NAME
28                processed_patchset="$(process_name "$line")${SEPARATOR_CHAR}"
29                ;;
30             1) # EMAIL
31                processed_patchset+="${line}${SEPARATOR_CHAR}"
32                ;;
33             2) # TITLE
34                patch_title="$line"
35                ;;
36          esac
37
38          ((count++))
39       done <<< "$pre_processed_patches"
40
41       for pid in "${pids[@]}"; do
42          wait "$pid"
43       done
44
45       for i in $(seq "$starting_index" "$((PATCHSETS_PROCESSED - 1))"); do
46          mailing_list_patchsets["$i"]=$(< "${shared_dir_for_parallelism}/${i}")
47       done
48    }
```

Program 4.12 is a listing of `process_patchsets`. Regarding the function implementation, it first creates a temporary directory using `mktemp`, then it enters the while loop and iterates through the pre-processed list of patches returned by `pre_process_xml_result` finding representative messages using the function `is_introduction_patch` and launching a background process using the function `thread_for_process_patch` to handle each representative message. Each background process executes a set of subtasks to define the data that compose an entry of `mailing_list_patches` (the attributes and format are described in Program 4.4) and stores this data in the temporary directory `shared_dir_for_parallelism`. Finally, after every background process finishes its tasks, the processed patchsets are loaded into the data structure `mailing_list_patchsets`.

It is pertinent to note that the executing shell that enters `process_patchsets` is the one that iterates through the while loop, launches the background processes, and halts execution until the termination of each one (lines 50 to 52 of Program 4.12). This original executing shell waits for the end of the background processes to synchronize them, as using the parallel processes to load data into `mailing_list_patchsets` would allow concurrent accesses to this shared data structure, which could lead to *race conditions* (Pacheco, 2011).

From the macroscopic view of listing the patchsets of a mailing list, in case the total number of patchsets processed in the current fetch session is less than what the caller of `fetch_latest_patchsets_from` required, another iteration of the while loop occurs for the next page of the target mailing list archive (the page is updated in line 42 of Program 4.7).

4.3.3 Handling individual patchsets

The `patch-hub` feature main objective is to provide a TUI of the Lore archives for patch reviewers. Listing the available mailing lists and the patchsets of a target list is not helpful if the feature does not allow patch reviewers to manipulate individual patchsets.

In `patch-hub` Model, each manipulation is implemented as one or more functions representing an action on the target patchset. Currently, two actions are supported: download the patchset applicable mbox file to a specific directory and add a bookmark to a patchset or remove it.

The download of a patchset to a specific directory entails the use of the request to Lore API for the mbox file of patches, as explained in Section 4.2. This request is not made directly by `patch-hub` Model, which uses the `b4` tool [11] to download the whole series in a single mbox file, without including the cover letter and that is ready to be applied in a Git repository with `git am`. Program 4.13 is a listing of `download_series`, which is the function used to handle the download of a patchset action.

`patch-hub` Model manages a database for bookmarks that serves as a record for easy access to selected patchsets. This database is implemented as a flat-file database (see Section 2.1). The actions provided by the model are the adding and removing of

[11] https://b4.docs.kernel.org/en/latest/

Program 4.13 Implementation of `download_series`

```
1   function download_series()
2   {
3     local series_url="$1"
4     local save_to="$2"
5     local flag="$3"
6     local series_filename
7     local cmd
8     local ret
9
10    flag=${flag:-'SILENT'}
11
12    if [[ -z "$series_url" || -z "$save_to" ]]; then
13      return 22 # EINVAL
14    fi
15
16    cmd_manager "$flag" "mkdir --parents '${save_to}'"
17    ret="$?"
18    if [[ "$ret" != 0 ]]; then
19      complain "Couldn't create directory in ${save_to}"
20      return "$ret"
21    fi
22
23    series_filename=$(extract_message_id_from_url "$series_url")
24
25    series_url=$(replace_http_by_https "$series_url")
26
27    cmd="b4 --quiet am '${series_url}' --no-cover --outdir '${save_to}' --mbox-name '${
          series_filename}.mbx'"
28    cmd_manager "$flag" "$cmd"
29    ret="$?"
30    if [[ "$ret" == 1 ]]; then
31      complain 'An error occurred during the execution of b4'
32      complain "b4 command: ${cmd}"
33    elif [[ "$ret" == 2 ]]; then
34      complain 'b4 unrecognized arguments'
35      complain "b4 command: ${cmd}"
36    else
37      printf '%s/%s.mbx' "$save_to" "$series_filename"
38    fi
39
40    return "$ret"
41  }
```

patchsets to and from this database. Implementing both involves explicitly managing the file used as the database for bookmarks. Program 4.14 is a listing of the function `add_patchset_to_bookmarked_database` that adds the patchset to the database. In contrast, Program 4.15 is a listing of `remove_patchset_from_bookmark_by_url` and `remove_patchset_from_bookmark_by_index`, which provide the removal of patchsets from the database.

As a side note on the development process of `patch-hub`, we used a flat-file database approach to implement the database for bookmarks since the requirements for the patchset entity (from a theoretical database modeling perspective) were not well-defined, so we chose to postpone the integration of the feature with the kw database system that uses SQLite3 (see Chapter 2).

New actions on individual patchsets are in development, like the integration of `patch-hub` with the features `kw build` and `kw deploy`, which would provide the compilation and installation of Linux kernel instances modified by patchsets. Other actions, from

Program 4.14 Implementation of `add_patchset_to_bookmarked_database`

```
function add_patchset_to_bookmarked_database()
{
  local raw_patchset="$1"
  local download_dir_path="$2"
  local timestamp
  local count

  create_lore_bookmarked_file

  timestamp=$(date '+%Y/%m/%d %H:%M')

  count=$(grep --count "${raw_patchset}" "${BOOKMARKED_SERIES_PATH}")
  if [[ "$count" == 0 ]]; then
    {
      printf '%s%s' "${raw_patchset}" "${SEPARATOR_CHAR}"
      printf '%s%s' "${download_dir_path}" "${SEPARATOR_CHAR}"
      printf '%s\n' "$timestamp"
    } >> "${BOOKMARKED_SERIES_PATH}"
  fi
}
```

Program 4.15 Implementations of `remove_patchset_from_bookmark_by_url` and `remove_patchset_from_bookmark_by_index`

```
function remove_patchset_from_bookmark_by_url()
{
  local patchset_url="$1"

  if [[ ! -f "${BOOKMARKED_SERIES_PATH}" ]]; then
    return 2 # ENOENT
  fi

  patchset_url=$(printf '%s' "$patchset_url" | sed 's/\//\\\//g')

  sed --in-place "/${patchset_url}/d" "${BOOKMARKED_SERIES_PATH}"
}

function remove_series_from_bookmark_by_index()
{
  local series_index="$1"

  if [[ ! -f "${BOOKMARKED_SERIES_PATH}" ]]; then
    return 2 # ENOENT
  fi

  sed --in-place "${series_index}d" "${BOOKMARKED_SERIES_PATH}"
}
```

displaying patch contents inside the feature to replying with tags like *Reviewed-by* and adding inline comments, are scheduled to be added in the future.

4.3.4 Managing feature configurations

Like other features in the kw project, `patch-hub` has a configuration file managed by kw. More specifically, a file named `lore.config` resides in the `etc` directory of the project repository that holds general configurations of the feature. For example, the number of patchsets per page displayed when listing from a target mailing list is set in this configuration file, as well as the available mailing lists that the user selected as *favorites* (referred to as *registered mailing lists*).

Program 4.16 Implementation of `save_new_lore_config`

```
1   function save_new_lore_config()
2   {
3     local setting="$1"
4     local new_value="$2"
5     local lore_config_path="$3"
6
7     if [[ ! -f "$lore_config_path" ]]; then
8       complain "${lore_config_path}: file doesn't exists"
9       return 2 # ENOENT
10    fi
11
12    sed --in-place --regexp-extended "s<(${setting}=).*<\1${new_value}<" "$lore_config_path"
13  }
```

`patch-hub` Model provides a unified interface for managing this configuration with the function `save_new_lore_config`, that is listed in Program 4.16.

Chapter 5

Final Remarks

The focus of this work was to delineate the integration of the *Lore archives* - a Web application that hosts archives of mailing lists related to the Linux kernel development - and *KWorkflow* (kw) - a system that aims to provide a unified environment that enhances the workflows of Linux developers. In terms of technological results, this study resulted in two main contributions to the kw project:

1. Integration of KWorkflow with a Database Management System (DBMS) (see Chapter 2).
2. Integration of KWorkflow with the Lore archives through the development of the `patch-hub` feature (see Chapters 3 and 4).

The kw project used a *file-based database* approach to manage its databases, which was functional but not scalable or efficient. The migration to a DBMS approach using the *SQLite3* DBMS was made as a preliminary task to implement the `patch-hub` feature, which would benefit from this approach change. The DBMS integration with kw had short-term positive effects, like considerably reducing the execution time of the automated unit tests suite, and will probably have future positive effects on the maintainability and scalability of the codebase.

Before this work, no feature in kw enhanced the Linux developer workflow of interacting with the public mailing lists for patch review. The on-demand characteristic of the Lore archives made possible the creation of `patch-hub`, a feature in the kw project to provide a user-friendly terminal-based interface to the flow of patchsets sent to the mailing lists related to the Linux kernel development. `patch-hub` also strives to offer integrations with other kw features by taking advantage of the unified environment provided by the system. The careful design of `patch-hub` architecture, which uses the *Model-View-Controller* (MVC) design pattern and the *Finite-State Machine* (FSM) mathematical computational model, assures that the feature has loosely coupled components that are well-defined and, therefore, makes the feature more maintainable, testable, and robust. A direct benefit of implementing the MVC pattern is that it is possible to add another type of UI, for example, a Web UI, using the same core component of the feature (i.e., reusing `patch-hub` Model). On the other hand, modeling `patch-hub` UI using the FSM model simplified and organized the implementation of the feature screens. Besides that, the screens of `patch-hub` use

the `Dialog` tool as a framework, and a side-effect of implementing the feature was the addition of a generic library for creating *terminal user interfaces* (TUIs) to the kw project. Finally, it is worth highlighting that, although it will be further expanded and refined, `patch-hub` is a functional feature that validated a tool in the kw environment to enhance the interaction with mailing lists. Screenshots that demonstrate the `patch-hub` feature are in Appendix A.

Moreover, the author became a maintainer of the kw project in 2023. Thus, other results were not discussed in this capstone project; the most notable [1] were the miscellaneous contributions to the kw project. A full listing is in Appendix B, but we emphasize these contributions:

1. Addition of Zsh native completions to the project.
2. Addition of tracing and profiling capabilities to the project.

A shell completion system suggests command names, option names, and values for choice, file, and path parameter types when a specific key is pressed (usually, the `TAB` key). Bash completions were natively implemented into the kw project, while Zsh completions were emulated using the Bash ones, producing errors that broke Zsh users' prompts. For Zsh users, not only did the addition of native completions fix the issues, but it also enhanced their user experience, as the Zsh completion system offers a great variety of functionalities that were explored in the implementation. This is further discussed in a blog post by the author [2].

Motivated by optimizing the execution time of fetching and processing patchsets from the Lore archives, two mechanisms were introduced into the kw project: tracing capabilities and generating execution profiles. The former relates to producing reports about the flow of execution, and the latter uses these reports to create different profiles that describe the flow of execution that can be used for debugging and finding performance bottlenecks. A dedicated page on the kw official website is a tutorial for using these two mechanisms [3].

The work done in this study creates opportunities for future work. Even though functional, the performance of `patch-hub` for fetching and processing patchsets is not ideal and can be optimized, as the bottleneck was determined using the tracing and profiling capabilities mentioned above, and a solution has already been drafted. With the desired performance, the next logical step would be to expand the functionalities of `patch-hub`, specifically integrating it with `kw build` and `kw deploy`, and adding the possibility of replying individual patches with commit tags like *Reviewed-by* and *Tested-by*, and with inline comments. Incorporating these expansions establishes the core functionalities of the feature, allowing for studies on the processes and practices in the Linux kernel development to be conducted using `patch-hub` to collect data.

---

[1] Three patchsets were sent and merged into the Linux kernel mainline and can be checked at the following links: https://lore.kernel.org/amd-gfx/20230306022427.437022-1-davidbtadokoro@gmail.com/, https://lore.kernel.org/amd-gfx/20230307225341.246596-1-davidbtadokoro@usp.br/, and https://lore.kernel.org/amd-gfx/20230307191417.150823-1-davidbtadokoro@usp.br/.

[2] https://davidbtadokoro.github.io/posts/adding-support-for-native-zsh-completions/

[3] https://kworkflow.org/content/tracingandprofiling.html

These future works (except the one about the conduction of studies) are tracked in a public *Kanban board* on GitHub Projects [4]. It is important to stress that the author was invited to continue this work in a doctorate in Computer Science to collaborate to evolve the research in FLOSS and Linux Kernel conducted by IME-USP Systems Group. Accordingly, the future works mentioned above will be studied, and the `patch-hub` feature will continue to be further developed.

[4] A Kanban board is an agile project management tool designed to help visualize work, limit work-in-progress, and maximize efficiency. The `patch-hub` tracking Kanban board is available at https://github.com/orgs/kworkflow/projects/2/views/1.

Chapter 6

Personal Appreciation

I had no programming or Computer Science background when I began my Bachelor of Computer Science undergraduate education. I have always had a great interest in anything related to computers, so when I decided to switch courses (I took three semesters in Bachelor of Physics prior), I knew for sure that Computer Science would be the right field for my life.

Almost four years after the start of my undergraduate studies, I came in contact with plenty of exciting topics in the Computer Science field, like building a virtual processor, designing and implementing a network protocol to play tic-tac-toe, writing a monography on Spectre and Meltdown, doing mathematical proof on graphs, and many others.

However, the experiences that I lived through this last year made me grow in all aspects of my academic, professional, and personal life. The interactions with the KWorkflow (kw) community - which I got to know through the *Laboratory of Extreme Programming* discipline - led to the work done in this capstone project, to the participation in the *Google Summer of Code 2023* (GSoC 23) as a contributor [1], and to put into practice all the knowledge that I obtained during my undergraduate education. From learning how to contribute to a free software project to having patches merged into the Linux mainline and diving deep into Git mechanics, Web applications, Linux kernel concepts, and the like, my experiences in the kw project truly consolidated the hard and soft skills that I amassed through these four years. Before my experiences with the kw project, I felt unsure about these skills and thought they were not solid because they were only applied in controlled environments of the Bachelor of Computer Science course. After a year of continuously contributing to the kw project, I became a maintainer and, today, help the project's progress on both sides by developing and reviewing code.

During the weekly meetings of the kw project, when I was just starting as a contributor, Rodrigo Siqueira presented me with two fronts that were merged and became the object of this study: the integrations of a *Database Management System* (DBMS) and the *Lore*

[1] Link to the GSoC 23 page in the past programs archive: https://summerofcode.withgoogle.com/archive/2023/projects/eqFrXfAz. Link to my GSoC 23 final report: https://davidbtadokoro.github.io/posts/gsoc23-final-report/

archives with the kw environment. These two topics quickly became my focus and my capstone project, from which I derived many learnings.

Compared to when I started the undergraduate course, I still feel overwhelmed by how diverse and vast the Computer Science field is and how fast it is evolving. Nevertheless, I understood that learning creates as many questions as it solves them and that there is no such thing as mastering an entire field. As I finish my undergraduate studies, I feel eager to keep on this unspeakably rewarding journey of learning about computers.

Appendix A

Demonstration of the `patch-hub` feature

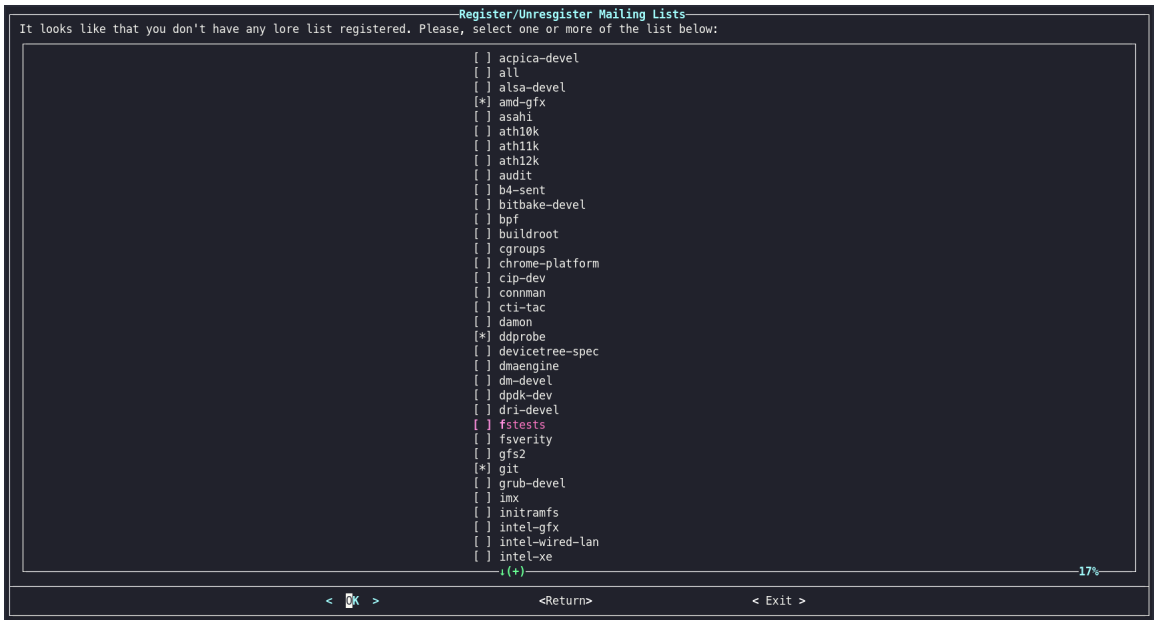
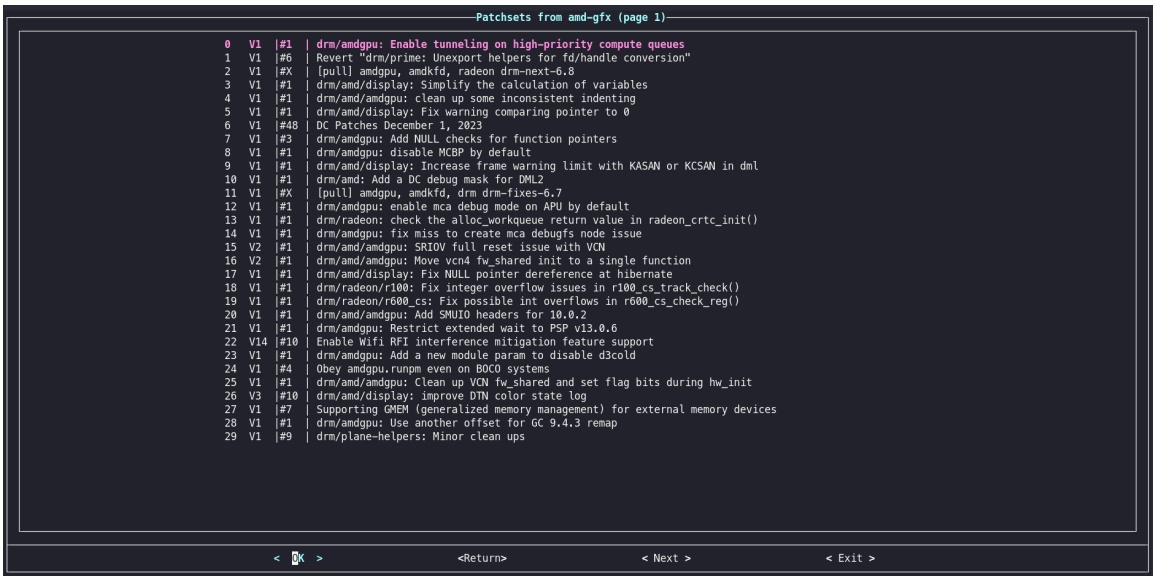
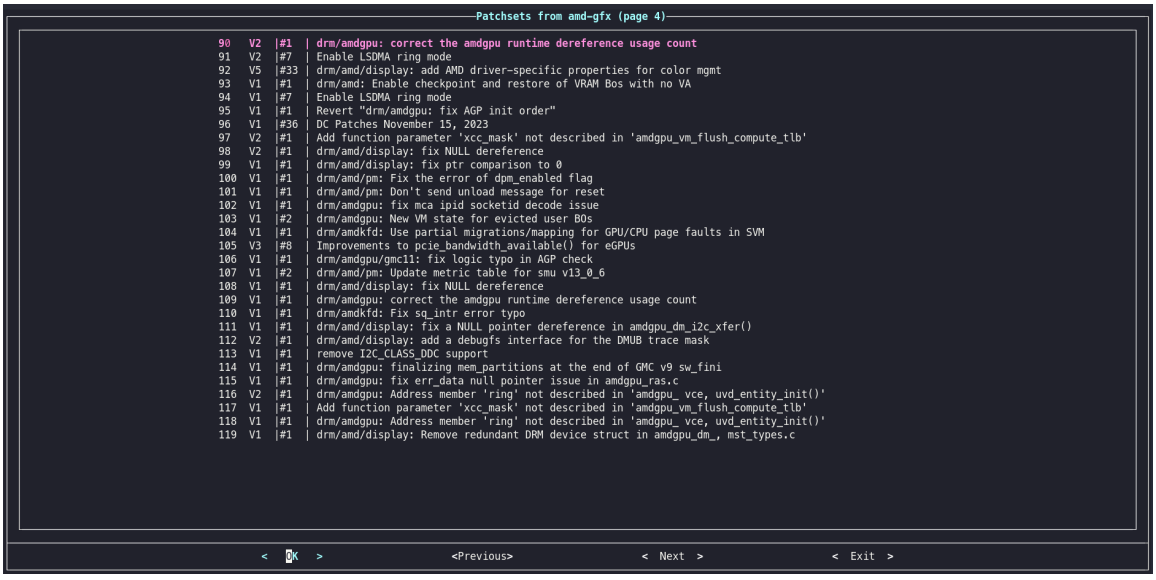


Figure A.1: *`patch-hub` listing of archived mailing lists on Lore.*

Figure A.2: *`patch-hub` menu with registered mailing lists.*

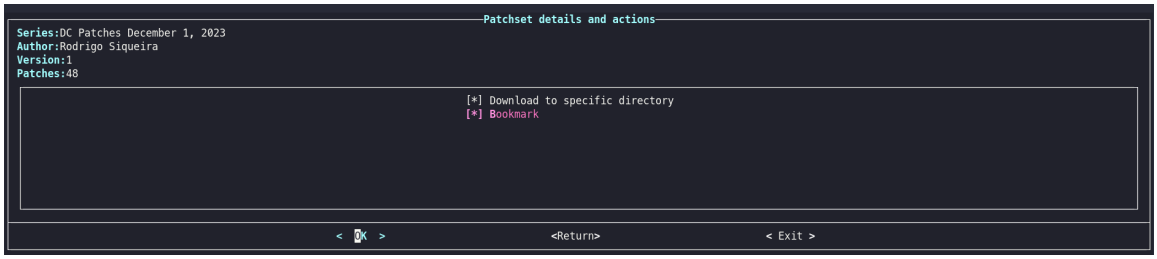


(a) *Page 1 of latest patchsets from amd-gfx mailing list.*

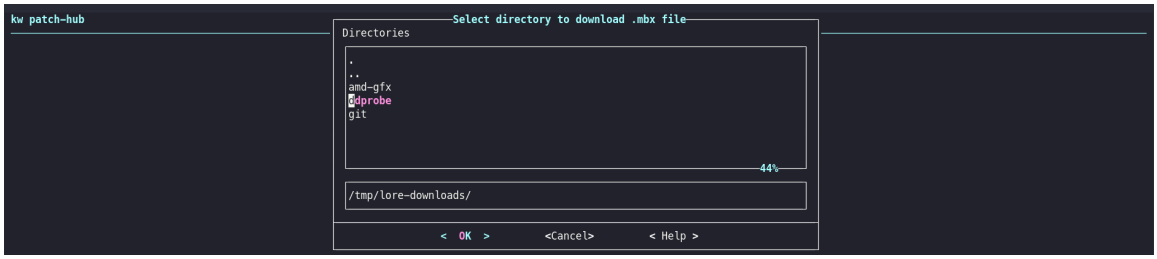


(b) *Page 4 of latest patchsets from amd-gfx mailing list.*

Figure A.3: *`patch-hub` listing of latest patchsets from target mailing list.*

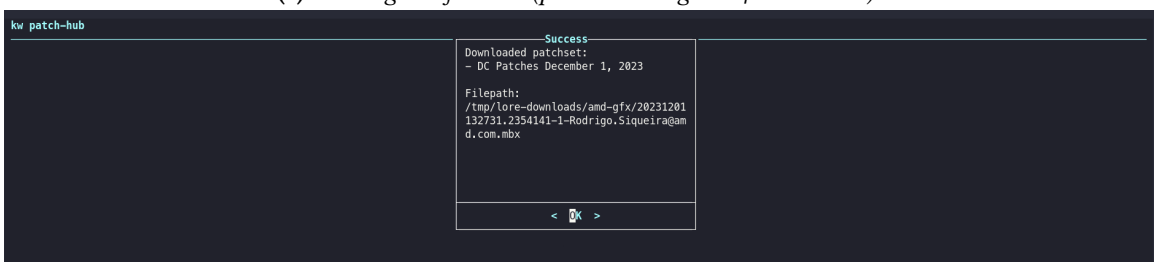


(a) *Individual patchset details and actions.*



(b) *Dialog box to select specific directory to download patchset.*

(c) *Loading notification (present throughout* `patch-hub`*).*



(d) *Dialog box confirming action (present throughout* `patch-hub`*).*

Figure A.4: `patch-hub` *handling of individual patchset.*

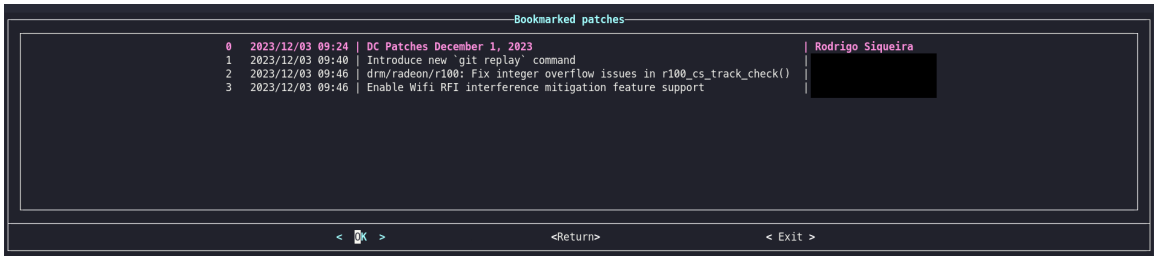


Figure A.5: `patch-hub` *menu with bookmarked patchsets.*

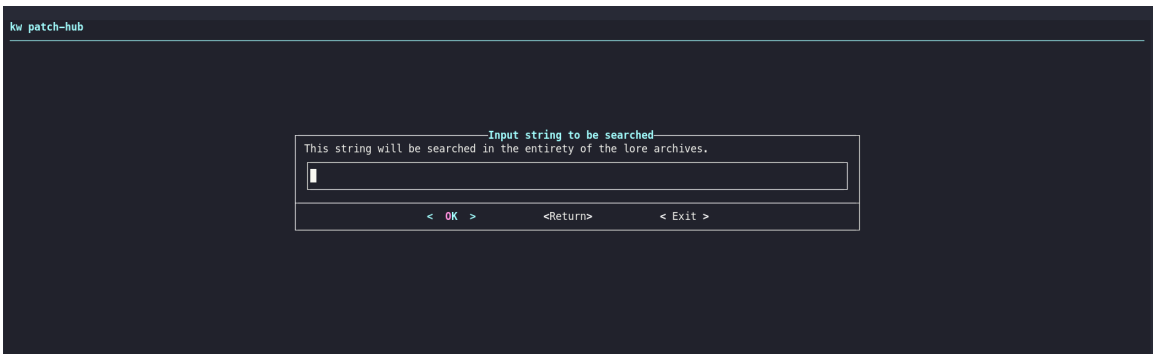
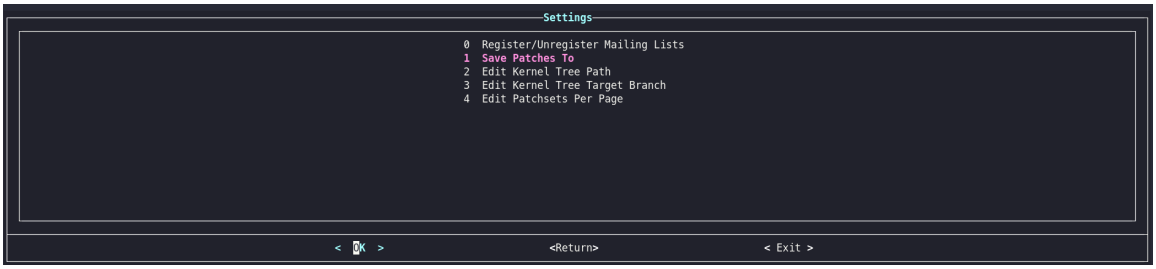
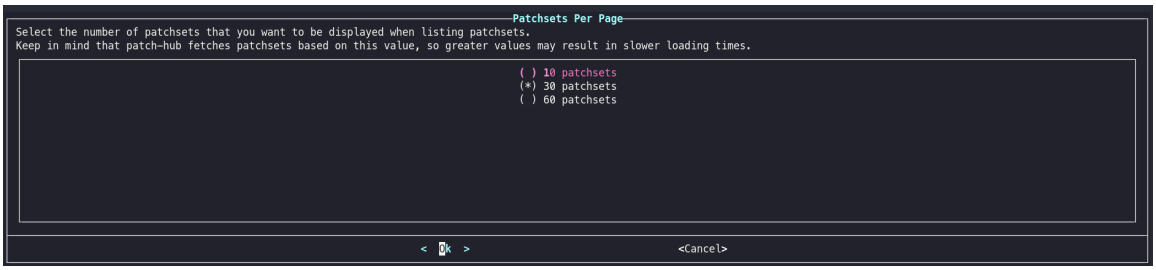


Figure A.6: *patch-hub capability of querying Lore archives based on string.*



(a) *Settings menu.*

(b) *Setting kernel tree target branch configuration.*



(c) *Setting patchsets per page configuration.*

Figure A.7: *patch-hub setting of configurations through the feature.*

Figure A.8: *patch-hub ability to adapt to terminal configurations (dimensions, color scheme, fonts, and the like).*

Appendix B

List of contributions to the KWorkflow project

Below is the complete listing, in chronological order, of the merged pull requests made by the author to the KWorkflow project, which accounts for the contributions made this year [1].

- src: update: add self-update mechanism to kw
- tests: report_test: Fix terminal and file outputs from test_save_data_to()
- Allow some kw deploy commands to be run outside kernel tree
- documentation: man: kw: Revise deploy subsection
- src: kw_remote: Fix not failing when missing valid options
- src: kw_remote: Fix remove remote that is prefix of other remote
- Revise kw remote man page
- Add support for native Zsh completions
- documentation: dependencies: Add curl and xpath dependencies
- src: upstream_patches_ui: Add help option
- src: upstream_patches_ui: Fix list_patches menu title
- src: upstream_patches_ui: Add loading screen for delayed actions
- src: upstream_patches_ui: Add bookmark feature
- src: upstream_patches_ui: Fix Dashboard screen message box
- Integrate kw_bd to the project and add migration script
- src: lib: lore: Use b4 tool for downloading patch series

[1] `patch-hub` used to be called `upstream-patches-ui`, so pull requests that refer to the latter are related to the former.

- Add Bash and Zsh completions for upstream-patches-ui
- src: upstream-patches-ui: Add basic feature documentation
- Add 'Settings' menu for upstream-patches-ui
- upstream-patches-ui: dialog's severe bugs with certain arguments
- src: upstream_patches_ui: Fix 'New Patches' screen title bug
- Adding 'Apply' action and 'Kernel Config File' setting menu to kw upstream-patches-ui
- src: upstream_patches_ui: Replace undefined help function call
- src: upstream_patches_ui: Fix relative paths in 'Kernel Tree Path'
- Refactor upstream-patches-ui feature
- upstream-patches-ui: Controller refactoring
- src: patch_hub: Rename upstream-patches-ui feature to patch-hub
- patch-hub: Revise 'Patchsets Details and Actions' screen
- patch-hub: Refactor lore mailing lists screen
- src: lib: remote: Fix ssh connection fail message with remote.config
- src/lib/dialog_ui: Reduce duplicated code and add pattern to file
- kw patch-hub: Add reliable fetch of latest patchsets from mailing list
- patch-hub: Fix bug and refactor 'Registered Mailing Lists' screen
- src: ui: patch_hub: patch_hub_core: Fix 'Registered Mailing Lists' message box
- patch-hub: Add query based on string functionality
- src: mail: Add 'additional_emails' configuration
- Add tracing to kw and introduce kw profiler
- src: _kw: Add Zsh completion for kw build –full-cleanup option
- documentation: conf: Fix deprecated navigation_with_keys setting bug
- patch-hub: Update handler function of dashboard state

References

[Abelson and Sussman 1996] Harold Abelson and Gerald Jay Sussman. *Structure and interpretation of computer programs*. The MIT Press, 1996 (cit. on p. 47).

[Fowler 2012] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2012 (cit. on pp. 21, 22, 37).

[Hopcroft *et al.* 2006] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. USA: Addison-Wesley Longman Publishing Co., Inc., 2006. isbn: 0321455363 (cit. on pp. 23, 24).

[Kurose and Ross 2012] James F. Kurose and Keith W. Ross. *Computer Networking: A Top-Down Approach (6th Edition)*. 6th. Pearson, 2012. isbn: 0132856204 (cit. on pp. 38–41).

[Neto 2022] Rubens G. Neto. *Simplificando o processo de contribuição para o kernel Linux*. 2022. url: https://www.linux.ime.usp.br/~rubensn/mac0499/monografia/monografia_entrega.pdf (cit. on pp. 8, 18).

[Pacheco 2011] Peter Pacheco. *An Introduction to Parallel Programming*. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011. isbn: 9780123742605 (cit. on p. 55).

[Silberschatz *et al.* 2012] Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. *Operating System Concepts*. 9th. Wiley Publishing, 2012. isbn: 1118063333 (cit. on p. 5).

[Sipser 1996] Michael Sipser. *Introduction to the Theory of Computation*. 1st. International Thomson Publishing, 1996. isbn: 053494728X (cit. on p. 23).

[Stewart *et al.* 2020] K Stewart, S Khan, D German, *et al.* "2020 linux kernel history report". *Linux Foundation, Version v5, Aug* 8 (2020) (cit. on p. 5).

[Ullman 2007] Jeffrey D Ullman. *A first course in database systems*. Pearson Education India, 2007 (cit. on p. 16).