Universidade de São Paulo
Instituto de Matemática e Estatística
Bacharelado em Ciência da Computação

Automatização do fluxo de submissões de patches para o Kernel Linux através do kworkflow

João Guilherme Barbosa de Souza

Monografia Final

mac 499 — Trabalho de Formatura Supervisionado

Supervisor: Prof. Dr. Paulo Roberto Miranda Meirelles
Cossupervisor: David de Barros Tadokoro

São Paulo
2025

Resumo

João Guilherme Barbosa de Souza. **Automatização do fluxo de submissões de patches para o Kernel Linux através do kworkflow**. Monografia (Bacharelado). Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2025.

Palavras-chave: Palavra-chave1. Palavra-chave2. Palavra-chave3.

Abstract

João Guilherme Barbosa de Souza. **Automating the Linux Kernel patch submission flow using kworkflow**. Capstone Project Report (Bachelor). Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2025.

Keywords: Keyword1. Keyword2. Keyword3.

Lista de abreviaturas

Lista de figuras

Lista de tabelas

Lista de programas

Sumário

Introdução

Capítulo 1

O Kernel Linux

Em um computador, o Sistema Operacional é a parte responsável por lidar com as interações entre o hardware e o software, permitindo, de maneira eficiente, a interação final máquina-usuário. Para que isso seja possível, o sistema operacional precisa ser dividido em diversas partes, dentre essas, o kernel, considerado o núcleo dos sistemas operacionais. Em geral, o kernel é um programa que opera a todo momento e é responsável pelo gerenciamento dos processos do sistema, alocando recursos para outros programas em atividade conforme a necessidade e prioridade de cada um.

Dentre os muitos sistemas de kernels existentes, o Kernel Linux é um projeto desenvolvido por Linus Torvalds em 1991 como alternativa ao Unix, pioneiro no mercado de sistemas operacionais. Hoje, o Kernel Linux é o maior projeto de software livre do mundo, utilizado por algumas das maiores empresas de tecnologia, software e computação no mercado, possuindo diversas distribuições e constituindo aproximadamente 57% dos websites na internet cujos sistemas operacionais puderam ser identificados.[1]

1.1 O Modelo de desenvolvimento do Kernel Linux

Ainda que tenha sido lançado há mais três décadas, novas versões do Kernel Linux continuam sendo lançadas até hoje. Essas versões, chamadas "kernels estáveis", são construídas, em parte, através da contribuição de diversos desenvolvedores ao redor do mundo, através da submissão de *patches* de melhorias que são integrados à versão principal para o desenvolvimento de futuras versões. De acordo com Feitelson, 2012, esse desenvolvimento do Kernel Linux segue um modelo de perpetual development, no qual novas funcionalidades, correções e versões de produção são liberadas continuamente, havendo também a manutenção de versões mais antigas.

Segundo a The Linux Kernel Documentation, 2023, esse processo divide-se em três etapas bem distintas, a janela de mesclagem, o período de estabilização e a manutenção contínua:

[1] Fonte: https://w3techs.com/technologies/details/os-linux

Durante a primeira etapa a maior parte das alterações serão integradas à nova versão do kernel. Essas mesclagens ocorrem a partir de *patches* que foram previamente preparados, testados e coletados. Com base nessa nova versão, o primeiro kernel RC (Release Candidate) será lançado, encerrando a janela de mesclagem e iniciando a próxima etapa.

Durante a segunda etapa, apenas *patches* que sirvam para correção de *bugs* deverão ser enviados e novas versões de RC serão lançadas periodicamente até que uma versão estável seja atingida. De forma objetiva, uma versão estável é atingida quando todas as regressões, erros conhecidos superados por versões anteriores, reintroduzidos durante a janela de mesclagem, são corrigidas.

Contudo, dado o tempo limitado em que as etapas precisam ocorrer, eliminar todas as regressões das versões estáveis nem sempre é um desafio que pode ser atingido. Por conta desse fato, após a criação da versão estável, uma equipe de desenvolvedores é designada para a terceira etapa, a manutenção contínua, lançando novas atualizações ocasionais com correções para essa versão por um período de tempo enquanto a janela de mesclagem se reinicia para a nova versão.

1.2 Software livre e contribuição para o Kernel Linux

1.2.1 Software livre x Software proprietário

No mercado computacional, dois conceitos principais dominam o cenário no que se refere ao desenvolvimento como software proprietário ou software livre. Em geral, define-se como software proprietário, o software que é desenvolvido de maneira privada, com apenas a aplicação sendo acessível para usuários. Enquanto que, softwares livres são projetos cujo código fonte é de livre acesso para qualquer um que queira estudar.

Historicamente, o mercado computacional era dominado por grandes corporações, que detinham o monopólio do processo e conhecimento e recursos necessários para o desenvolvimento e do software como um todo, dificultando o ingresso de outros competidores no mercado. Nesse cenário, projetos de software livre surgem como um processo disruptivo, compartilhando o acesso a esse conhecimento, de modo a promover a colaboração e inovação na indústria (Avatavului *et al.*, 2023).

Comparativamente, softwares proprietários são geridos por empresas, que contratam equipes fixas de funcionários para trabalhar em período integral e de maneira exclusiva no projeto. Dessa maneira, a decisão quanto às novas implementações para o software são centradas, com o principal objetivo, em muitos casos, sendo o de ganho financeiro. Essa prática, contudo, muitas vezes implica em que o foco constante seja em implementações de novas ferramentas ao invés da melhoria do software como um todo, favorecendo com que esses softwares sejam mais propensos à erros e falhas ocasionais. Por outro lado, a existência de responsáveis legais pelo projeto fazem com que esses softwares contem em grande parte com a existência de um suporte especializado, característica valorizada no mercado corporativo.

De maneira contraditória, softwares livres são desenvolvidos de maneira colaborativa,

contando com a contribuição voluntária de grandes quantidades de desenvolvedores ao redor do mundo inteiro. Por conta disso, as demandas surgem de forma espontânea, muitas vezes da necessidade do próprio usuário que depende do software para usos pessoais. Essa grande quantidade de contribuidores, aliada à dependência mútua destes com o software, garante atualizações frequentes de segurança e qualidade. Como consequência, esses sistemas tendem a ser menos propensos a erros, mas não contam com um suporte dedicado na maioria dos casos.

Hoje, softwares livres e proprietários continuam coexistindo no mercado, sendo constantemente comparados quanto à sua efetividade observada em projetos reais. Porém, a inegável vantagem do conhecimento colaborativo e do grande volume de contribuição que softwares livres conseguem apresentar, fazem com que hoje ele esteja em grande crescente no mercado computacional, sendo o método de desenvolvimento de diversos softwares relevantes mundialmente, como no caso do Kernel Linux.

1.2.2 Contribuição em Software Livres

Em projetos de software livre, para que a gestão das contribuições seja possível, os sistemas geralmente contam com uma equipe mantenedores, grupo interno de desenvolvedores que possuem responsabilidade geral pelo código principal. Desse modo, para que sejam integradas, as contribuições enviadas precisam passar por revisões por parte dos mantenedores para garantir que atendam aos requisitos técnicos definidos para o projeto. De acordo com Tan *et al.*, 2020, os mantenedores devem avaliar principalmente se um *patch* é necessário, se uma implementação apresenta falhas ou se existem eventuais melhorias na forma como a solução foi feita.

Em alguns casos, principalmente devido ao crescimento dos projetos, torna-se necessário também uma divisão em componentes do sistema principal, de modo que cada parte possui seus mantenedores dedicados. Dessa maneira, para que a contribuição seja feita de maneira correta, os *patches* precisam ser enviados diretamente para o responsável do subsistema que será alterado.

1.2.3 Contribuindo para o Kernel Linux

No caso do Kernel Linux, essa divisão lógica é feita com base em conjuntos de subsistemas, como, por exemplo, os sistema de rede, gerenciamento de memória, dispositivos de vídeo, etc. Dentro desses subsistemas, cada mantenedor responsável administra uma árvore de fontes do kernel, gerindo os *patches* enviados ao seu subsistema. Assim, quando a janela de mesclagem é aberta, às contribuições previamente aprovadas pelos mantenedores serão enviadas diretamente ao Linus e, caso aceitas, integradas ao código principal para gerar a nova versão. (The Linux Kernel Documentation, 2023)

Ainda segundo a documentação oficial (The Linux Kernel Documentation, 2023), eventualmente, esses subsistemas podem ser identificados de modo que um subsistema principal seja constituído por subsistemas menores, como, por exemplo, o subsistema de rede, que agrega também as árvores dedicadas a drivers de dispositivos de rede, redes sem fio. Desse modo, além dos *patches* recebidos diretamente por contribuidores, os

mantenedores podem receber também *patches* já aprovados por outros mantenedores, formando uma cadeia de confiança até que os *patches* cheguem a serem integrados.

Assim como o kernel, os *patches* que são incorporados durante a janela de mesclagem precisam ser previamente preparados e, durante esse preparo, passam por algumas etapas. Esse processo, ainda que informal, serve para garantir que cada *patch* possa ser revisado e tenha sua qualidade garantida antes que a alteração seja incorporada ao kernel principal (The Linux Kernel Documentation, 2023).

Os principais estágios que um *patch* deve passar, são:

1. **Design**: Nesta etapa, serão levantados os requisitos do *patch* e a forma com que serão atingidos, ou seja, a identificação dos seus objetivos e as necessidades técnicas que devem constar nessa implementação.
2. **Revisão antecipada**: Publicação dos *patches* na lista de discussão relevante para que desenvolvedores possam responder com comentários e ajudar a revelar quaisquer problemas iniciais.
3. **Revisão mais ampla**: Antes que o *patch* seja considerado para inclusão na versão principal, ele deve ser aceito por um mantenedor de subsistema, que o incluirá nas árvores *-next*. Com essa etapa, revisões mais elaboradas e possíveis problemas de integração com outras implementações poderão ser verificados.
4. **Mesclagem e manutenção de longo prazo**: Ainda que o *patch* possa ser mesclado e chegar efetivamente à versão estável do kernel, futuros problemas podem vir a aparecer durante essas fases, dessa forma, o desenvolvedor original deve continuar a assumir a responsabilidade da manutenção do código no futuro.

1.2.4 O problema da contribuição em desenvolvimento de software

O sistema de controle de versão é indispensável no desenvolvimento de software contemporâneo. Antes do advento dos sistemas de controle de versão, os programadores dependiam de métodos manuais para gerenciar suas modificações de código. Eles costumavam fazer backups regulares de seus arquivos de código ou adotar convenções de nomenclatura para distinguir entre as várias versões. Esse processo era bastante inconsistente e difícil de gerenciar, especialmente quando alguns desenvolvedores estavam trabalhando no mesmo projeto. (Devineni, 2020)

Gerenciar as versões de um software se torna um problema ainda maior dependendo do tamanho total do software, do número de contribuidores e da quantidade de contribuições sendo realizadas nele de maneira simultânea. No kernel, por exemplo, a versão 6.13, lançada em 19/01/2025, contou com mais de 206 contribuições por dia por parte de 2085 colaboradores, resultando em um código fonte final com mais de 39 milhões de linhas Kroah-Hartman, 2025. Segundo a tendência, esses números devem seguir aumentando de forma constante conforme novas versões forem sendo desenvolvidas.

Buscando superar parte dessas dificuldades e melhorar o processo colaborativo de desenvolvimento de software, foram desenvolvidos os sistemas de controle de versão.

Segundo Devineni (2020),

Os primeiros Version Control Systems (VCS), permitiam que os desenvolvedores mantivessem um histórico das alterações realizadas nos arquivos, o que facilitava a reversão de mudanças e oferecia visibilidade sobre a evolução do código. No entanto, o potencial colaborativo ainda era limitado, exigindo muitos acordos e gestões manuais por parte dos colaboradores.

Como segunda opção, surgem os *Concurrent Versions System (CVS)*, baseados em um modelo de repositório central. Nele, os desenvolvedores podiam obter os arquivos, aplicar suas modificações e submetê-las novamente ao repositório. Esse modelo contribuiu para maior agilidade em equipes de desenvolvimento, ao permitir que várias pessoas trabalhassem simultaneamente na mesma base de código. Ainda assim, em projetos de grande porte ou com equipes distribuídas geograficamente, os sistemas centralizados apresentavam limitações no gerenciamento eficiente do trabalho.

Por fim, surgem os modelos mais utilizados atualmente, os *Distributed Concurrent Versions System - DVCS*, como o Git. Esses sistemas, ao contrário da versão anterior, distribuía as cópias do código central entre os desenvolvedores, permitindo um método mais flexível de colaboração. Como cada colaborador poderia ter uma versão local do código, as mudanças realizadas por ele ao código principal poderiam ser administradas localmente antes de serem integradas, permitindo trabalhos offline e que alterações fossem submetidas em lotes ao invés de individualmente.

Contudo, de acordo com Greg Kroah Hartman, 2016, ainda que softwares como *github*,[2] *gerrit*[3] ou outros DVCS possam ser úteis para gerir o fluxo de submissões de softwares menores, eles ainda apresentam muitos problemas para escalar para softwares maiores. Dentre os principais motivos, são citados, por exemplo, a maneira como o fluxo para revisão desses softwares é mais demorado e diminui a produtividade dos mantenedores, a dificuldade de gerenciar e categorizar os inúmeros problemas e submissões com os recursos oferecidos, a maneira como as discussões e comentários dentro da comunidade são pouco acessíveis à outros contribuidores, dificultando a propagação de informação e gerando retrabalho, a dificuldade para que desenvolvedores possam se conectar à listas de discussões e serem notificados sempre que uma novidade relevante ocorra, entre outros. Parte desses problemas da comunidade, porém, ainda segundo Greg Kroah Hartman, 2016, são solucionados ao se substituir os softwares de DVCS por servidores de email, como é feito para a contribuição do kernel.

Essa substituição, entre tanto, também apresenta suas dificuldades, uma vez que, sendo um sistema com perspectiva muito mais abrangente, servidores de email não apresentam funcionalidades e melhorias para esse fluxo. Entre os diversos problemas enfrentados pelo usuário, destacam-se principalmente o grande *overhead* inicial para novos contribuidores, a má rastreabilidade do históricos de submissões e revisões, a escalabilidade limitada, sobrecarregando a lista de email de alguns mantenedores, problemas de corrupção de arquivos, e a dificuldade de se capturar métricas. Em resposta, hoje, muitos softwares vêm sendo desenvolvidos, buscando oferecendo automatizações para esse fluxo na busca

[2] https://github.com

[3] https://www.gerritcodereview.com

de suprir algumas de suas lacunas, como é o caso do *b4*,[4] *Patchwork*,[5] *lore.kernel.org*,[6] *kworkflow*,[7] entre outras soluções encontradas na comunidade.

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

[5] https://patchwork.kernel.org/

[6] https://lore.kernel.org/

[7] https://github.com/kworkflow/kworkflow

Referências

[Avatavului *et al.* 2023] Cristian Avatavului *et al.* "Open-source and closed-source projects: a fair comparison". *Journal of Information Systems & Operations Management* 17.2 (dez. de 2023) (citado na pg. 4).

[Devineni 2020] Siva Karthik Devineni. "Version control systems (vcs) the pillars of modern software development: analyzing the past, present, and anticipating future trends". *International Journal of Science and Research* 9.12 (2020), pp. 1816–1829. doi: 10.21275/SR24127210817 (citado na pg. 6).

[Feitelson 2012] Dror G. Feitelson. "Perpetual development: a model of the linux kernel life cycle". *Journal of Systems and Software* 85.4 (2012), pp. 859–875. url: https://www.cs.huji.ac.il/~feit/papers/LinuxDev12JSS.pdf (citado na pg. 3).

[Greg Kroah Hartman 2016] Greg Kroah Hartman. *Kernel Recipes 2016 - Patches carved into stone tablets… - Greg KH*. Acessado em: 03 set. 2025. 2016. url: https://youtu.be/L8OOzaqS37s?si=zPnIcyGIlu7lIcmK (acesso em 03/09/2025) (citado na pg. 7).

[Kroah-Hartman 2025] Greg Kroah-Hartman. *kernel-history: Linux kernel history logs and stats*. Repositório GitHub. Acesso em: 06 set. 2025. 2025. url: https://github.com/gregkh/kernel-history (citado na pg. 6).

[Tan *et al.* 2020] Xin Tan, Minghui Zhou e Brian Fitzgerald. "Scaling open source communities: an empirical study of the linux kernel". In: *Proceedings of the 42nd International Conference on Software Engineering (ICSE 2020)*. ACM / IEEE Computer Society, 2020, pp. 1222–1234. doi: 10.1145/3377811.3380920 (citado na pg. 5).

[The Linux Kernel Documentation 2023] The Linux Kernel Documentation. *How the development process works*. Parte de "A guide to the Kernel Development Process" — versão v4.14. 2023. url: https://www.kernel.org/doc/html/v4.14/process/2.Process.html (acesso em 04/09/2025) (citado nas pgs. 3, 5, 6).