Git et le modèle de développement du noyau Linux

Julien Kirch

2021-07-01

Git est un outil très maléable qui s'adapte à de nombreuses manières de travailler.

Git est aussi un outil qui tient son origine dans le noyau Linux. Je pense qu'à cause de cela, et de l'aura du noyau Linux dans le monde du logiciel, le modèle de développement du noyau Linux a une influence importante sur l'usage de Git en général, même dans des cas très éloignés.

Or le noyau Linux a un modèle de développement assez spécifique, notamment quand on le compare à celui de la majorité des projets web.

Ces spécificités proviennent à la fois du type de logiciel qu'est le noyau Linux (le noyau d'un système d'exploitation), de son organisation (hiérarchique et répartie entre plusieurs organisations) et d'autres enfin sont des choix historiques ou des préférences.

Je pense que comprendre ces aspects du développement du noyau Linux est intéressant pour prendre du recul sur ses habitudes de développement, par exemple pour se rendre compte que des choses qu'on prend pour acquises ne le sont pas forcément.

Je ne veux pas dire qu'il est souhaitable de recopier ou de s'inspirer de ces approches, mais ma conviction est que les connaître permet de mieux éclairer nos choix.

Une connaissance minimale du fonctionnement d'un projet et de Git est nécessaire pour comprendre ce qui suit.

Le process de développement

Si le noyau Linux n'est pas la propriété d'une entreprise ou d'une organisation, son process de développement est assez codifié et hiérarchique.

En haut de la pyramide Linus Torvald est responsable de la branche de développement principale, c'est-à-dire que c'est lui qui fait les merges depuis les autres branches.

Différents sous-ensembles du noyau Linux appelés *subsystem* sont sous la responsabilité chacun d'au moins une personne appelée *maintainer*. Dans le code ces sous-ensembles

correspondent à des répertoires ou à des fichiers. Un fichier définit les périmètres des différentes personnes.

Dans un sous-ensemble, le ou les responsables ont une certaine autonomie tant que les règles générales du fonctionnement du noyau sont respectées.

C'est Linus Torvald qui nomme ces responsable, et il peut à tout moment leur retirer leur rôle ou revenir sur leurs décisions.

Le développement de la branche principale du noyau Linux se fait de manière itérative, avec une nouvelle version approximativement tous les 2 mois.

Lors de l'ouverture d'une nouvelle version du noyau Linux au développement, Linus Torvald va commencer par récupérer les commits disponibles dans différentes branches préparées par ces responsables et qui en principe contiennent du code "prêt à être mergé".

Une partie de ce code est écrit par ces responsables. Le reste provient d'autre personnes.

Pour voir leur code inclus dans le noyau Linux, ces autres personnes doivent convaincre au moins un responsable du domaine en question de le merger.

En effet Linus Torvald ne prend pas proposition de code "en direct" mais se repose sur les responsables qu'il nommé.

Ce qui signifie que ces personnes ont droit de vie ou de mort sur votre code. Quand vous voulez proposer une évolution, vous avez besoin de la "vendre" pour que les personnes acceptent d'y prêter attention, et potentiellement acceptent de la merger.

Cela signifie aussi que les revues peuvent être fastidieuses à cause du rapport de force très déséquilibrés. L'équivalent des pull requests sont ainsi parfois réécris 10 ou 20 fois jusqu'à être validées, ou jusqu'à ce que la personne qui propose la modification se lasse ou change de job.

Le noyau Linux continuant à être modifié pendant ce temps, qui peut prendre plusieurs mois ou années, il faut aussi réadapter votre code à ces évolutions et donc le rebaser régulièrement, car les pull request ne doivent pas contenir de commits de merge.

Par rapport à un projet développé dans une organisation unique, où des mécanismes de régulation peuvent aider à débloquer des choses, les personnes qui travaillent sur le noyau Linux font parties de structures différentes (et une partie continue à le faire de manière indépendante sur ton temps de loisir). Il est toujours possible de s'adresser à Linus Torvald directement, mais c'est l'équivalent dans ne organisation de contacter votre N+2 quand vous avez un problème avec votre N+1 : vous pouvez le faire mais ça peut avoir des conséquences, surtout si vous avez l'intention ensuite de continuer à travailler sur cette partie du code.

Valider du code sans pouvoir tester

Une partie conséquente du noyau Linux est composé des pilotes qui sont les liens entre les périphériques matériels et le noyau, fournissant les implémentations de différentes API en fonction du matériel en question (générer du son ou de la vidéo, réagir à l'appui d'une touche sur un clavier ou une souris…).

Une personne responsable d'un sous-système couvrant de pilotes possède en principe quelques périphériques du domaine correspondant (comme quelques modèles de souris ou de cartes graphiques). Si on lui propose une évolution du code d'un pilote et qu'elle possède le matériel en question, elle pourra alors le tester.

Mais si on lui propose un changement et qu'elle ne possède pas le matériel, elle va devoir faire une revue du code sans pouvoir le tester en condition réelle.

Les tests automatisés peuvent permettre de détecter des changements de comportements entre des versions en mockant le matériel, mais ne sont pas forcément suffisant pour valider que le code va véritablement fonctionner .

En effet les périphériques ont la mauvaises habitudes de ne pas suivre exactement leurs spécifications, et certains vendeurs fournissent des spécifications insuffisantes voire pas de spécification du tout, le développement doit alors se faire par essai et erreur.

Il faut alors documenter de manière précise ce qui se passe, le code n'étant souvent pas assez explicite en lui-même, pour qu'une personne qui n'a pas le matériel puisse suivre ce qui se passe et éventuellement faire des retours.

La phase de merge passée, il arrive régulièrement que les personnes tentent de corriger des bugs sans pouvoir tester, quand la personne qui a initialement soumis le code est passée à autre chose.

Les revues par mail

Le média privilégié pour les discussions d'évolution de code du noyau Linux est l'email. Cela tient aux habitudes, mais aussi au fait que de nombreux outils ont été conçus autour des échanges par mail, par exemple pour déterminer à qui une pull request doit être envoyée, ou pour savoir qui a validé un changement.

Git contient des fonctionnalités facilitant ce type d'approche, par exemple `git format-patch`.

Cela signifie que les revues de code se font par email. Même si les personnes peuvent importer les changements à discuter dans leur éditeur de code, les discussions se font par mails.

Pour rendre ces échanges gérables, les pulls request sont donc découpées en suite de changement aussi courts que possibles et s'appuyant les uns sur les autres. Voici par exemple une pull request en 21 commits.

Parvenir à parvenir à découper son changement demande des compétences bien spécifiques.

Lorsque des changements sont demandés dans une revue (le patch ci-dessus en est à sa troisième version), cela demande de reprendre les différents commits, parfois d'en ajouter d'intermédiaires ou d'en supprimer, tout en s'assurant que tout fonctionne bien à chaque étape.

Les reports de code entre les versions

À l'inverse d'une application web qui n'a qu'une version en production à un moment donné, plusieurs versions du noyau Linux sont supportées à un moment donné.

Cela permet à certaines distributions Linux de ne pas changer de version du noyau après leur sortie, et ainsi d'essayer de limiter les risques d'introduire de nouveaux bugs.

Cela ne signifie pas que le noyau des versions passées n'évolue pas, mais que les changements qu'on va y appliquer sont soit des corrections de problèmes de sécurité, soit des corrections de bugs qui ont un rapport gravité sur risque d'introduire d'autres problèmes suffisamment élevé.

Concrètement il va s'agir d'identifier les changements pertinents quand ils sont ajoutés à la version en cours, et de les appliquer sur les différentes versions encore supportées.

Bien entendu, vouloir porter seulement certains changements et pas tous signifie qu'il faudra peut-être les adapter pour les faire correspondre au code des anciennes versions. C'est un peu la même chose que de réadapter du code pas encore mergé quand la branche de développement évolue, mais à l'envers.

Si besoin la personne en charge de telle ou telle version en maintenance peut demander de l'aide à personne responsable du sous-système correspondant, mais elle n'a pas de garantie de l'obtenir, car tout le monde ne donne pas la même priorité à cette partie du développement.

Et pour revenir sur la partie précédente, parfois ni la personne en charge de la version ni la personne responsable du sous-système n'a le moyen de tester que le changement fonctionne bien.

Ici aussi la capacité du code à pouvoir être compris sans être exécuté est importante.

Le fait d'avoir des versions supportées pendant longtemps amène aussi à ne pas vouloir faire de changements à l'utilisé insuffisante dans le code.

Si les refactoring nécessaires sont réguliers quand une API change ou qu'une nouvelle approche est choisie, les choses comme des reformatage de code ou des renommages de variables mineurs sont proscrites pour ne pas rendre inutlilement complexe le portage de code.

Des PR sur plusieurs années

Incorporer du code dans le noyau Linux peut être long, voir très long. Par exemple certaines fonctionnalités nécessaire pour du temps réel ont été mergées cet été alors qu'elles avaient été proposées pour la première fois en 2004. Pendant 17 ans, les personnes qui ont travaillé sur cette partie du projet on investi du temps et de l'énergie pour convaincre les responsables de différentes parties du noyau Linux de merger une à une les modifications nécessaires.

Cela signifie que pendant 17 ans, il a fallu maintenir ce fork et l'adapter aux différents changement amonts, et notamment à ceux qui étaient demandés dans les composants temps réels pour qu'ils soient acceptés.

Si c'est le seul chantier de cet ampleur par sa durée, et le fait qu'une seule personne le dirige depuis le début.

Une situation plus classique est qu'une personne propose une modification, se décourage, et qu'une autre reprenne le chantier quelques temps plus tard parce qu'elle-même en a besoin.

Par exemple ce changement de 2017 visant à améliorer le comportement quand des données sont indisponibles en mémoire est basé sur un travail démarré en 2009 et qui a connu plusieurs reprises.

Parfois cela finit par passer, d'autres patchs reviennent encore et encore, souvent parce que l'approche proposée n'est pas jugée satisfaisante, mais qu'il n'en existe pas de meilleure.

Conclusion

Le modèle de développement du noyau Linux accorde une grande importance aux commits individuels et à l'historique du code. Être capable de faire merger du code dans le noyau Linux est ainsi une compétence très recherchée par certaines organisations.

Ce fonctionnement quelque chose qui répond aux enjeux du projet (un système d'exploitation supportant des versions sur le long terme) et qui est inscrit dans les habitudes et l'organisation du projet.

J'espère que vous ne travaillez pas sur une branche mise à jour pendant 17 ans pour enfin espérer la merger, et que vous n'avez pas besoin de valider du code sans pouvoir le tester.

Si vous travaillez (et je vous le souhaite) dans un environnement qui n'a pas ces besoins, je vous invite à vous interroger sur vos propres pratiques.

Il est normal que quand on développe on soit attaché au produit de son travail, et vouloir faire les choses proprement est louable, mais j'ai l'impression qu'on a parfois tendance à porter une attention excessive aux commits et à l'historique git.

Cela ne veut pas dire systématiquement prendre le contre-pied du noyau Linux, mais qu'il faut savoir investir son énergie là où elle est le plus utile.