Auteur: **Eric S. Raymond** **(esr@thyrsus.com)**

Traduction: **Sébastien Blondeel**

Date: 1998/08/11 20:27:29

J'analyse le succès d'un projet de logiciel dont le code source est ouvert,

> *NdT traduction de open-source software, terme par lequel l'auteur a délibérément remplacé le terme précédent free software (logiciel libre). ``Ouvert'' signifie ici à la fois public et susceptible d'être commenté et corrigé par ceux qui s'y intéresseront suffisamment.*

fetchmail (``va chercher le courrier''), qui a été lancé délibérément pour tester certaines théories surprenantes du génie logiciel suggérées par l'histoire de Linux. Je discute ces théories en termes de deux styles de développement fondamentalement différents, le modèle ``cathédrale'' de la majorité du monde commercial, à opposer au modèle ``bazar'' du monde de Linux. Je montre que ces modèles dérivent d'hypothèses opposées sur la nature de la tâche consistant à déboguer un logiciel. Enfin, je m'efforce de démontrer, à partir de l'expérience apportée par Linux, qu'``Étant donnés suffisamment d'observateurs, tous les bogues sautent aux yeux'', je suggère des analogies productives avec d'autres systèmes auto-correcteurs par agents égoïstes, et je conclus en réfléchissant aux implications de ces idées sur l'avenir du logiciel.

---

1. La cathédrale et le bazar

Linux est subversif. Qui aurait imaginé, il y a seulement cinq ans, qu'un système d'exploitation de classe internationale prendrait forme comme par magie à partir de bidouilles

> NdT Aux termes *to hack, hacker*, Eric S. Raymond a consacré un article complet. Il s'agit de l'action de comprendre comment les choses fonctionnent, de vouloir savoir ce qui se cache dans les mécaniques diverses, et de les réparer ou de les améliorer. En aucun cas, ce terme ne doit être confondu, sous cette plume, avec les pirates de l'informatique, ou *crackers*. J'ai choisi de traduire ce terme par ``bidouille'', et je vous demande de ne pas y attacher d'a priori négatif. Un bidouilleur construit des choses parfois très belles et très compliquées, alors qu'un pirate cherche à détruire.

faites pendant le temps libre de plusieurs milliers de développeurs disséminés de par le monde, et reliés seulement par les les liens ténus de l'Internet ?

Certainement pas moi. Quand, début 1993, Linux est apparu pour la première fois sur mon écran radar, cela faisait déjà dix ans que j'étais impliqué dans le développement sous Unix et dans la programmation de logiciels dont le code source est ouvert. Au milieu des années 1980, j'étais l'un des premiers contributeurs à GNU. J'avais distribué sur le réseau une bonne quantité de logiciels dont le code source est ouvert (nethack, les modes VC et GUD pour Emacs, xlife, et d'autres),

encore largement utilisés de nos jours. Je pensais savoir comment tout cela fonctionnait.

Linux a remis en cause une grande partie de ce que je croyais savoir. J'avais prêché l'évangile selon Unix sur l'utilisation de petits outils, le prototypage rapide et la programmation évolutive, depuis des années. Mais je pensais aussi qu'il existait une certaine complexité critique au delà de laquelle une approche plus centralisée, plus a priori, était nécessaire. Je pensais que les logiciels les plus importants (comme les systèmes d'exploitation et les très gros outils comme Emacs) devaient être conçus comme des cathédrales, soigneusement élaborés par des sorciers isolés ou des petits groupes de mages travaillant à l'écart du monde, sans qu'aucune version bêta ne voie le jour avant que son heure ne soit venue.

Le style de développement de Linus Torvalds - distribuez vite et souvent, déléguez tout ce que vous pouvez déléguer, soyez ouvert jusqu'à la promiscuité - est venu comme une surprise. À l'opposé de la construction de cathédrales, silencieuse et pleine de vénération, la communauté Linux paraissait plutôt ressembler à un bazar, grouillant de rituels et d'approches différentes (très justement symbolisé par les sites d'archives de Linux, qui acceptaient des contributions de *n'importe qui*) à partir duquel un système stable et cohérent ne pourrait apparemment émerger que par une succession de miracles.

Le fait que ce style du bazar semblait fonctionner, et bien fonctionner, fut un choc supplémentaire. Alors que j'apprenais à m'y retrouver, je travaillais dur, non seulement sur des projets particuliers, mais encore à essayer de comprendre pourquoi le monde Linux, au lieu de se disloquer dans la confusion la plus totale, paraissait au contraire avancer à pas de géant, à une vitesse inimaginable pour les bâtisseurs de cathédrales.

Vers le milieu de 1996, je pensais commencer à comprendre. Le hasard m'a donné un moyen incomparable de mettre ma théorie à l'épreuve, sous la forme d'un projet dont le code source est ouvert et que je pourrais consciemment faire évoluer dans le style du bazar. Ce que je fis -- et je rencontrai un franc succès.

Dans le reste de cet article, je conterai l'histoire de ce projet, et je l'utiliserai pour proposer des aphorismes relatifs au développement efficace de logiciels dont le code source est public. Je n'ai pas appris toutes ces choses dans le monde Linux, mais nous verrons de quelle manière le monde Linux les place au devant de la scène. Si je ne me trompe pas, elles vous aideront à comprendre exactement ce qui fait de la communauté Linux une telle manne de bons logiciels - et à devenir plus productif vous-même.

2. Le courrier doit passer !

Depuis 1993, j'étais aux commandes du service technique d'un modeste fournisseur de services Internet proposant un accès libre, appelé Chester County Interlink (CCIL) et situé dans le West Chester, en Pennsylvanie (je suis l'un des co-fondateurs de CCIL et j'ai écrit notre unique logiciel de ``bulletin-board system'' (BBS, systèmes de discussions entre utilisateurs) multi-utilisateurs -- vous pouvez vérifier cela en accédant par telnet à locke.ccil.org. Aujourd'hui, ce service supporte presque trois mille utilisateurs sur trente lignes.) Ce travail m'a permis d'accéder au réseau 24 heures sur 24, à travers la ligne de 56K de CCIL -- en réalité, c'était presque une nécessité !

De cette manière, je m'étais habitué au courrier électronique instantané par l'Internet. Pour des raisons compliquées, il était difficile de faire fonctionner SLIP entre ma machine située à mon domicile (`snark.thyrsus.com`) et CCIL. Quand j'ai enfin réussi, j'ai trouvé que me connecter régulièrement à `locke` pour vérifier que je n'avais pas de courrier électronique était ennuyeux. Je souhaitais que mon courrier me parvienne sur `snark` de telle sorte que je sois averti dès son arrivée et que je puisse le traiter à l'aide de mes outils locaux.

Indiquer simplement à sendmail de faire suivre le courrier n'aurait pas fonctionné, parce que ma machine personnelle n'est pas sur le réseau en permanence et ne dispose pas d'une adresse IP statique. J'avais besoin d'un programme qui étende une main tentaculaire à travers ma connexion SLIP, et me rapporte mon courrier afin de le distribuer de manière locale. Je savais que de telles choses existaient, et que la plupart d'entre elles utilisait un simple protocole d'application (application protocol) appelé POP (Post Office Protocol, protocole du bureau de poste). Et bien sûr, le système d'exploitation BSD/OS de `locke`, incluait un serveur POP3.

J'avais besoin d'un client POP3. Alors je suis allé sur le réseau et j'en ai trouvé un. En fait, j'en ai trouvé trois ou quatre. J'ai utilisé pop-perl pendant un moment, mais il lui manquait une fonctionnalité qui semblait évidente, la possibilité de bidouiller les adresses sur le mail rapporté afin que les réponses fonctionnent correctement.

Le problème était le suivant: un dénommé `joe' sur `locke` m'envoyait du courrier. Si je rapportais le courrier sur `snark` et tentais d'y répondre, mon programme de courrier essayait en toute bonne foi de le faire parvenir à un `joe' sur `snark`, qui n'existait pas. Éditer les adresses de retour à la main pour leur ajouter `@ccil.org' est rapidement devenu assez pénible.

C'était typiquement le genre de choses que l'ordinateur devait faire à ma place. Mais aucun des clients POP existants ne savait comment faire ! Et ceci nous amène à la première leçon:

1. Tout bon logiciel commence par gratter un développeur là où ça le démange.

Ceci est peut-être une évidence (il est bien connu, et depuis longtemps, que ``Nécessité est mère d'Invention'') mais trop souvent on voit des développeurs de logiciels passer leurs journées à se morfondre à produire des programmes dont ils n'ont pas besoin et qu'ils n'aiment pas. Ce n'est pas le cas dans le monde Linux -- ce qui peut expliquer pourquoi la plupart des programmes issus de la communauté Linux sont de si bonne facture.

Alors, me suis-je précipité frénétiquement dans le codage d'un client POP3 tout nouveau tout beau, qui soutienne la comparaison avec ceux qui existent déjà ? Jamais de la vie ! J'ai examiné avec beaucoup d'attention les utilitaires POP que j'avais à ma disposition, en me demandant ``lequel de ces programmes est le plus proche de ce que je cherche ?''. Parce que

2. Les bons programmeurs savent quoi écrire. Les grands programmeurs savent quoi réécrire (et réutiliser).

N'ayant pas la prétention de me compter parmi les grands programmeurs, j'essaie seulement de les imiter. Une caractéristique importante des grands programmeurs est la paresse constructive. Ils savent qu'on n'obtient pas 20/20 pour les efforts fournis, mais pour le résultat obtenu, et qu'il est pratiquement toujours plus facile de partir d'une bonne solution partielle que de rien du tout.

Linus Torvalds, par exemple, n'a pas essayé d'écrire Linux en partant d'une page blanche. Au

contraire, il a commencé par réutiliser le code et les idées de Minix, un minuscule système d'exploitation ressemblant à Unix tournant sur les 386. La totalité du code de Minix a été abandonnée ou réécrite complètement depuis -- mais tant qu'il était là, ce code a servi de tuteur à l'enfant qui deviendrait Linux.

Dans le même esprit, je me suis mis en quête d'un utilitaire POP raisonnablement bien écrit, afin de l'utiliser comme base pour mon développement.

La tradition de partage du code source du monde Unix a toujours favorisé la réutilisation de code (c'est pourquoi le projet GNU a choisi Unix comme système d'exploitation de travail, malgré de sérieuses réserves quant au système d'exploitation lui-même). Le monde Linux a pratiquement emmené cette tradition jusqu'à sa limite technologique; il dispose de téraoctets (millions de millions d'octets) de codes sources ouverts généralement disponibles. De telle sorte que passer du temps à chercher le ``presqu'assez bon" de quelqu'un d'autre a plus de chances de donner de bons résultats dans le monde Linux que nulle part ailleurs.

Et pour moi, cela a marché. Avec ceux que j'avais déjà trouvés, ma deuxième recherche aboutit à un total de neuf candidats -- fetchpop, PopTart, get-mail, gwpop, pimp, pop-perl, popc, popmail et upop. Je me suis d'abord concentré sur le `fetchpop' de Seung-Hong Oh. J'y ai ajouté ma fonctionnalité de réécriture des entêtes, et j'ai fait un certain nombre d'améliorations que l'auteur a acceptées dans sa version 1.9.

Quelques semaines plus tard, cependant, je suis tombé sur le code de `popclient' par Carl Harris, et j'ai rencontré un problème. Fetchpop contenait des idées bonnes et originales (comme son mode démon), mais il ne pouvait gérer que POP3 et il était programmé d'une manière quelque peu amateur (Seung-Hong est un programmeur brillant mais inexpérimenté, et ces deux caractéristiques apparaissaient dans son travail). Le code de Carl était meilleur, assez professionnel et solide, mais son programme ne proposait pas certaines fonctionnalités de fetchpop, importantes et plutôt difficiles à coder (en particulier, celles que j'avais programmées moi-même).

Continuer, ou basculer ? Si je basculais, cela revenait à abandonner le travail que j'avais déjà réalisé, en échange d'une meilleure base de développement.

Une raison pragmatique me poussant à changer était la présence d'un support pour plusieurs protocoles. POP3 est le protocole de bureau de poste

> NdT traduction littérale de *POP*

le plus communément utilisé, mais ce n'est pas le seul. Fetchpop et ses semblables n'étaient pas compatibles avec les protocoles POP2, RPOP, ou APOP, et j'envisageais déjà vaguement d'ajouter IMAP (Internet Message Access Protocol, protocole d'accès aux messages par Internet, le protocole de bureau de poste le plus récent et le plus puissant), rien que pour le plaisir.

Mais j'avais une raison plus théorique de penser que changer serait une bonne idée malgré tout, une raison que j'avais apprise bien avant Linux.

3. ``Prévoyez d'en jeter un, car de toute manière, vous le ferez." (Fred Brooks, ``The Mythical Man-Month", chapitre 11)

> ``Le mythe du mois-homme", traduction de Frédéric Mora, ITP, ISBN 2-84180-081-4,

est un livre où il explique que l'ensemble des croyances relatives à l'unité de mesure ``mois-homme'' est un mythe.

Ou, dit autrement, on ne comprend souvent vraiment bien un problème qu'après avoir implanté une première solution. La deuxième fois, on en sait parfois assez pour le résoudre correctement. Ainsi, si vous voulez faire du bon travail, soyez prêt à recommencer *au moins* une fois.

Bien (me suis-je dit), les modifications à fetchpop furent un coup d'essai. J'ai donc basculé.

Après avoir envoyé à Carl Harris mon premier lot de modifications apportées à popclient, le 25 juin 1996, j'ai découvert qu'il avait pratiquement perdu tout intérêt pour popclient depuis quelque temps. Le code était un peu poussiéreux, et de petits bogues y subsistaient. J'avais de nombreuses modifications à apporter, et nous nous sommes rapidement mis d'accord pour que je reprenne le programme.

Avant même que je m'en rende compte, le projet avait monté d'un cran. Je n'envisageais plus d'apporter des modifications mineures à un client POP existant. Je m'engageais à maintenir un client POP dans son intégralité, et je savais que certaines des idées qui germaient dans mon cerveau m'amèneraient probablement à effectuer des modifications majeures.

Dans une culture du logiciel qui encourage le partage du code, il est naturel pour un projet d'évoluer ainsi. J'étais en train d'expérimenter le fait suivant:

4. Si vous avez la bonne attitude, les problèmes intéressants viendront à vous.

Mais l'attitude de Carl Harris était encore plus importante. Il comprit que

5. Quand un programme ne vous intéresse plus, votre dernier devoir à son égard est de le confier à un successeur compétent.

Sans même avoir à en discuter, Carl et moi savions que nous poursuivions un objectif commun: chercher la meilleure solution. La seule question d'importance pour chacun de nous était de nous assurer que le programme serait en de bonnes mains avec moi. Une fois que j'en apportai la preuve, il me fit place nette avec bonne volonté. J'espère agir de même quand mon tour viendra.

3. De l'importance d'avoir des utilisateurs

C'est ainsi que j'héritai de popclient. De même, et c'est tout aussi important, c'est ainsi que j'héritai de l'ensemble des utilisateurs de popclient. Il est merveilleux de disposer d'utilisateurs, et pas seulement parce qu'ils vous rappellent que vous remplissez un besoin et que vous avez fait une bonne chose. Éduqués de façon appropriée, ils peuvent devenir vos co-développeurs.

Le fait que beaucoup d'utilisateurs sont également des bidouilleurs est une autre force de la tradition Unix, et c'est une caractéristique que Linux a poussée avec bonheur dans ses derniers retranchements. De plus, la libre disponibilité du code source en fait des bidouilleurs *efficaces*. Ceci peut se révéler incroyablement utile pour réduire le temps de débogage. Avec un peu d'encouragement (ils n'en réclament pas beaucoup), vos utilisateurs diagnostiqueront les problèmes, suggéreront des corrections, et vous aideront à améliorer le code bien plus rapidement que vous n'auriez pu le faire, si vous étiez laissé à vous même.

6. Traiter vos utilisateurs en tant que co-développeurs est le chemin le moins semé d'embûches vers une amélioration rapide du code et un débogage efficace.

Il est facile de sous-estimer la puissance de ce phénomène. En fait, la quasi-totalité d'entre nous, appartenant au monde du logiciel dont le code source est ouvert, sous-estimions effroyablement la facilité avec laquelle il s'adapterait à l'augmentation du nombre d'utilisateurs et de la complexité des systèmes, jusqu'à ce que Linus Torvalds ne nous démontre le contraire.

En réalité, je pense que la bidouille la plus ingénieuse de Linus, et celle qui a eu le plus de conséquences, n'a pas été la construction du noyau de Linux en lui-même, mais plutôt son invention du modèle de développement de Linux. Un jour où j'exprimais cette opinion en sa présence, il souria et répéta tranquillement cette pensée qu'il a si souvent exprimée: ``Je suis tout simplement une personne très paresseuse, qui aime se faire remercier pour le travail effectué par d'autres." Paresseux comme un renard. Ou, comme Robert Heinlein aurait pu le dire, trop paresseux pour pouvoir échouer.

Rétrospectivement, on peut voir dans le développement des bibliothèques Lisp et des archives de code Lisp pour GNU Emacs un précédent aux méthodes et au succès de Linux. Au contraire du coeur d'Emacs, construit en C à la manière des cathédrales, et au contraire de la plupart des autres outils proposés par la FSF (Free Software Foundation, Fondation pour le Logiciel Libre), l'évolution de l'ensemble du code Lisp a été très fluide et très dirigée par les utilisateurs. On a souvent réécrit les idées et les prototypes des modes trois ou quatre fois avant d'atteindre une forme finale stable. Et les exemples de collaborations occasionnelles rendues possibles par l'Internet, à la manière de Linux, ne manquent pas.

En fait, ma bidouille personnelle qui a rencontré le plus de succès, avant fetchmail, fut probablement le mode VC pour Emacs, issu d'une collaboration par courrier électronique à la Linux avec trois autres personnes, et je n'ai toujours rencontré que l'une d'entre elles (Richard Stallman, l'auteur d'Emacs et le fondateur de la FSF) à ce jour. C'était une interface sous Emacs pour SCCS, RCS et plus tard pour CVS, qui fournissait toutes les opérations de contrôle de version avec un seul bouton. Il était parti d'un mode sccs.el très rudimentaire et tout petit, écrit par quelqu'un d'autre. Et le développement de VC fut réussi parce que, à la différence d'Emacs lui-même, le code Lisp pour Emacs pouvait traverser très rapidement les cycles de mise à jour/test/amélioration.

4. Distribuez tôt, mettez à jour souvent

Un élément essentiel du système de développement de Linux est la mise à jour rapide et fréquente des nouvelles versions. La plupart des développeurs (moi y compris) pensait que ce n'était pas une bonne méthode pour des projets de taille non triviale, parce que des versions prématurées sont quasiment, par définition, des versions boguées, et qu'il n'est pas dans votre intérêt d'abuser de la patience des utilisateurs.

C'est cette croyance qui a consolidé l'attachement général au style de développement de type cathédrale. Si l'objectif premier était de présenter aux utilisateurs une version aussi dépouillée de bogues que possible, alors vous ne faisiez une mise à jour que tous les six mois (voire moins souvent encore), et vous travailliez d'arrache-pied au débogage entre les mises à jour. C'est de cette manière qu'a été mis au point le coeur d'Emacs, écrit en C. Ce ne fut pas le cas, en pratique, de la

bibliothèque Lisp -- parce qu'il existait des archives Lisp ``vivantes'' hors de portée de la FSF, et où on pouvait trouver de nouvelles versions de code de développement indépendamment du cycle de mises à jour d'Emacs.

La plus importante de ces archives, l'archive elisp de l'État de l'Ohio, était en avance sur son temps et avait déjà deviné l'esprit et la plupart des spécificités des archives Linux actuelles. Mais bien peu d'entre nous réfléchissaient vraiment beaucoup sur ce que nous faisions ou sur les problèmes liés au développement de style cathédrale adoptés par la FSF que l'existence même de cette archive pouvait suggérer. J'ai vraiment tenté, une fois, vers 1992, de faire passer de manière formelle, un bon morceau du code d'Ohio dans la bibliothèque Lisp officielle d'Emacs. Je n'ai rencontré que des guerres d'opinions et je suis rentré bredouille dans les grandes longueurs.

Mais un an plus tard, alors que Linux devenait de plus en plus apparent, il était clair que quelque chose de différent, de plus sain, était en train de se produire. La politique de développement ouvert de Linus était l'antithèse même du modèle des cathédrales. Les archives de sunsite et de tsx-11 bourgeonnaient, de multiples distributions étaient mises à l'eau. Et tout cela était rythmé par une fréquence de mise à jour du coeur même du système jusqu'alors inégalée.

Linus traitait ses utilisateurs comme des développeurs de la manière la plus efficace:

7. Distribuez tôt. Mettez à jour souvent. Et soyez à l'écoute de vos clients.

La grande innovation de Linus ne fut pas tant de faire cela (c'était une tradition du monde Unix depuis un bon moment, en tout cas sous une forme proche), que de donner à cette idée un niveau d'intensité correspondant à la complexité de ce qu'il développait. En ces temps reculés (autour de 1991), il lui arrivait de mettre à jour son noyau plusieurs fois par *jour !* Comme il cultivait sa base de co-développeurs et recherchait des collaborations sur Internet plus que quiconque jusque là, cela a marché.

Mais *comment* cela a-t-il marché ? Et, était-ce quelque chose que je pouvais dupliquer, ou cela reposait-il uniquement sur quelque trait de génie propre à Linus Torvalds ?

Je ne le pensais pas. Je vous l'accorde, Linus est un bidouilleur sacrément bon (combien parmi nous pourraient mettre au point l'intégralité du noyau d'un système d'exploitation prêt à être mis en production ?). Mais Linux ne constituait pas vraiment un réel progrès conceptuel. Linus n'est pas (en tout cas, pas encore) un génie de l'innovation dans la conception comme peuvent l'être Richard Stallman ou James Gosling (de NeWS et Java). Au contraire, Linus est à mes yeux un génie de l'ingéniérie, doué d'un sixième sens qui lui permet d'éviter les bogues et les impasses de développement et surtout d'un authentique talent pour trouver le chemin de moindre effort reliant A à B. En effet, la conception de Linux dans son intégralité est imprégnée de cette qualité et reflète l'approche conservatrice et simplificatrice qu'a Linus de la conception.

Ainsi, si la rapidité des mises à jour et l'extraction de la substantifique moelle de l'Internet en tant que moyen de communication n'étaient pas des hasards, mais faisaient partie intégrante du côté visionnaire génial de Linus vers le chemin le plus facile à suivre, qu'est-ce donc qu'il maximisait ? Qu'est-ce donc qu'il tirait de toute cette machinerie ?

Posée de cette manière, la question contient sa propre réponse. Linus stimulait et récompensait ses utilisateurs/bidouilleurs en permanence -- il les stimulait par la perspective auto-gratifiante de

prendre part à l'action, et il les récompensait par la vue constante (et même *quotidienne*) des améliorations de leur travail.

Linus cherchait directement à maximiser le nombre de personnes-heures jetées dans la bataille du débogage et du développement, au prix éventuel d'une certaine instabilité dans le code et de l'extinction de sa base d'utilisateurs si un quelconque bogue sérieux se révélait insurmontable. Linus se comportait comme s'il croyait en quelque chose comme:

8. Étant donné un ensemble de bêta-testeurs et de co-développeurs suffisamment grand, chaque problème sera rapidement isolé, et sa solution semblera évidente à quelqu'un.

Ou, moins formellement, ``Étant donnés suffisamment d'observateurs, tous les bogues sautent aux yeux.'' C'est ce que j'appelle: ``La Loi de Linus''.

Ma première formulation de cette loi était que chaque problème ``semblera clair comme de l'eau de roche à quelqu'un''. Linus a objecté qu'il n'était pas nécessaire, et que d'ailleurs ce n'était pas le cas en général, que la personne qui comprenne un problème soit la personne qui l'ait d'abord identifié. ``Quelqu'un trouve le problème,'' dit-il, ``et c'est quelqu'un d'*autre* qui le comprend. Je pousserai le bouchon jusqu'à dire que le plus difficile est de trouver le problème.'' Mais le principal est que ces deux événements se produisent en général vite.

C'est là, je pense, la différence fondamentale sous-jacente aux styles de la cathédrale et du bazar. Dans la programmation du point de vue de la cathédrale, les bogues et les problèmes de développement représentent des phénomènes difficiles, ennuyeux, insidieux, profonds. Il faut à une poignée de passionnés des mois d'observations minutieuses avant de bien vouloir se laisser convaincre que tous les bogues ont été éliminés. D'où les longs intervalles séparant les mises à jour, et l'inévitable déception quand on se rend compte que la mise à jour tant attendue n'est pas parfaite.

Dans le point de vue bazar, d'un autre côté, vous supposez qu'en général, les bogues sont un phénomène de surface -- ou, en tout cas, qu'ils sautent rapidement aux yeux lorsqu'un millier de co-développeurs avides se précipitent sur toute nouvelle mise à jour. C'est pourquoi vous mettez à jour souvent afin de disposer de plus de corrections, et un effet de bord bénéfique est que vous avez moins à perdre si de temps en temps, un gros bogue vous échappe.

Et voilà. C'est tout. Si la ``Loi de Linus'' est fausse, alors tout système aussi complexe que le noyau Linux, subissant les bidouilles simultanées d'autant de mains que ce fut le cas du noyau Linux, aurait dû à un certain moment s'effondrer sous le poids des interactions néfastes et imprévues, sous le poids de bogues ``profonds'' non découverts. D'un autre côté, si elle est juste, elle suffit à expliquer l'absence relative de bogues dans Linux.

Et peut-être bien, après tout, que cela ne devrait pas être aussi surprenant. Il y a des années que les sociologues ont découvert que l'opinion moyenne d'un grand nombre d'observateurs (tous aussi experts, ou tous aussi ignorants) est d'un indicateur beaucoup plus fiable que l'opinion de l'un des observateurs, choisi au hasard. Ils ont appelé ce phénomène l'``effet de *(l'oracle de)* Delphes''. Il apparaît que Linus a fait la preuve que cela s'applique même au débogage d'un système d'exploitation -- que l'effet de Delphes peut apprivoiser la complexité du développement même au niveau de complexité atteint par le noyau d'un système d'exploitation.

Je dois à Jeff Dutky <dutky@wam.umd.edu> la remarque qu'on peut reformuler la Loi de Linus

sous la forme ``On peut paralléliser le débogage''. Jeff fait remarquer que, même si le débogage requiert que les débogueurs communiquent avec un développeur chargé de la coordination, une réelle coordination entre débogueurs n'étant pas indispensable. Ainsi, le débogage n'est pas la proie de cette augmentation quadratique de la complexité et des coûts d'organisation qui rend problématique l'ajout de nouveaux développeurs.

En pratique, le monde Linux n'est quasiment pas affecté par la perte théorique d'efficacité qui découle du fait que plusieurs débogueurs travaillent sur la même chose au même moment. L'une des conséquences de la politique du ``distribuez tôt, mettez à jour souvent'' est de minimiser les pertes de ce type en propageant au plus vite les corrections qui sont revenues au coordinateur.

Brooks a même fait une observation annexe proche de celle de Jeff: ``Le coût total de maintenance d'un programme largement utilisé est typiquement de 40 pour cent ou plus de son coût de développement. De façon surprenante, ce coût dépend énormément du nombre d'utilisateurs. *Un plus grand nombre d'utilisateurs trouve un plus grand nombre de bogues*.'' (l'emphase est de mon fait).

Plus d'utilisateurs trouvent plus de bogues parce que l'ajout de nouveaux utilisateurs introduit de nouvelles manières de pousser le programme dans ses derniers retranchements. Cet effet est amplifié quand les utilisateurs se trouvent être des co-développeurs. Chacun d'entre eux a une approche personnelle de la traque des bogues, en utilisant une perception du problème, des outils d'analyse, un angle d'attaque qui lui sont propres. L'``effet de Delphes'' semble précisément fonctionner grâce à cette variabilité. Dans le contexte spécifique du débogage, la diversité tend aussi à réduire la duplication des efforts.

Ainsi, introduire de nouveaux bêta-testeurs ne va sans doute pas réduire la complexité du bogue le plus ``profond'' du point de vue du *développeur*, mais cela augmente la probabilité que la trousse à outils d'un bêta-testeur sera adaptée au problème de telle sorte que le bogue saute aux yeux *de cette personne*.

Mais Linus assure ses arrières. Au cas où il y *aurait* des bogues sérieux, les versions du noyau Linux sont numérotées de telle sorte que des utilisateurs potentiels peuvent faire le choix d'utiliser la dernière version désignée comme étant ``stable'', ou de surfer à la pointe de la technique courant le risque que quelques bogues accompagnent les nouvelles fonctionnalités. Cette tactique n'est pas encore formellement imitée par la plupart des bidouilleurs Linux, mais c'est sans doute un tort; le fait d'avoir une alternative rend les deux choix séduisants.

5. De la chenille au papillon

Après avoir étudié le comportement de Linus et formulé une théorie expliquant les raisons de son succès, je décidai volontairement de mettre cette théorie en pratique sur mon nouveau projet (quoique bien moins complexe et bien moins ambitieux).

Mais la première chose que je fis fut de réorganiser et de simplifier popclient en profondeur. L'implantation de Carl Harris était très correcte, mais elle trahissait cette complexité inutile qui caractérise de nombreux programmeurs C. Il pensait que le code primait sur les structures de données, qui le servaient. Le résultat était un code assez esthétique, mais une conception improvisée

des structures de données, assez laides (en tout cas aux yeux d'un vieux baroudeur du Lisp comme moi).

J'avais malgré tout autre chose en tête que simplement améliorer le code et les structures de données. Mon but était de transformer popclient en quelque chose que je comprenne entièrement. Ce n'est pas drôle de devoir corriger des bogues dans un programme que vous comprenez mal.

Pendant environ un mois, je me contentai de suivre les conséquences de la conception simpliste de Carl. Le premier changement significatif que je fis fut d'introduire le support pour IMAP. Je le fis en réorganisant les pilotes de protocoles sous la forme d'un pilote générique et de trois tables de méthodes (pour POP2, POP3, et IMAP). Cela, et les modifications antérieures, illustre un principe général qu'il est bon pour les programmeurs de garder à l'esprit, en particulier dans des langages comme le C qui ne permettent pas de procéder à un typage dynamique de manière naturelle:

9. Il vaut mieux avoir des structures de données intelligentes et un code stupide que le contraire.

Si on prend le chapitre 9 de Fred Brooks: ``Montre-moi ton *code*, dissimule tes *structures de données*, je continuerai à être mystifié. Montre-moi tes *structures de données* et je n'aurai sans doute pas besoin de voir ton *code*, il me semblera évident."

En fait, il parlait d'``organigrammes" et de ``descriptions de données". Mais si on traduit cela en termes d'aujourd'hui, après trente ans d'évolution culturelle et technologique, cela revient pratiquement au même.

C'est à ce moment (début septembre 1996, six semaines après avoir commencé) que j'ai commencé à penser à procéder à un changement de nom -- après tout, il ne s'agissait plus là uniquement d'un client POP. Mais j'hésitai, puisque rien dans ma conception n'était vraiment neuf. Il fallait encore que mon programme développe une identité propre.

Cela s'est produit de manière radicale quand fetchmail apprit à faire suivre le courrier rapporté sur le port SMTP. J'y viendrai bientôt. Il me faut d'abord signaler ceci: j'ai dit plus haut que j'avais décidé d'utiliser ce projet pour mettre en pratique ma théorie sur les raisons du succès de Linus Torvalds. Comment ai-je procédé, vous direz-vous ? Comme suit:

1. J'ai distribué tôt et mis à jour souvent (au moins tous les dix jours; pendant les périodes de développement effréné, une fois par jour).
2. J'ai agrandi ma liste de bêta-testeurs en y ajoutant tout ceux qui me contactaient et me parlaient de fetchmail.
3. À chaque mise à jour, j'envoyais un petit palabre sur la liste bêta, encourageant les gens à participer.
4. Et j'ai écouté mes bêta-testeurs, en les sondant sur les choix de conception et en les caressant dans le sens du poil à chaque fois qu'ils m'envoyaient leur avis ou des corrections.

Le résultat de ces mesures élémentaires ne se fit pas attendre. La plupart des développeurs tueraient père et mère pour avoir des notifications de bogues de la qualité de celles que je reçus dès le début du projet, et la plupart du temps, elles étaient accompagnées de bonnes solutions. J'ai reçu des critiques réfléchies, du courrier d'admirateurs, des suggestions intelligentes proposant de nouvelles fonctionnalités. Ce qui nous donne:

10. Si vous traitez vos bêta-testeurs comme ce que vous avez de plus cher au monde, ils réagiront en

devenant effectivement ce que vous avez de plus cher au monde.

Une mesure intéressante du succès de fetchmail est l'impressionnante taille de la liste bêta du projet, les amis de fetchmail. Au moment ou j'écris ces lignes, elle compte 249 membres et elle grandit de deux ou trois membres chaque semaine.

En fait, alors que je relis maintenant ces lignes en mai 1997, la liste commence à diminuer (elle a atteint à son heure de gloire, la taille de 300 membres) pour une raison intéressante. Plusieurs personnes m'ont demandé de résilier leur inscription parce que fetchmail fonctionne si bien pour eux qu'ils n'éprouvent plus le besoin de suivre les discussions de la liste ! Cela fait sans doute partie du cycle de vie d'un projet dans le style bazar, lorsqu'il est parvenu a maturité.

6. Et popclient devint fetchmail

Le projet prit un virage radical le jour où Harry Hochheiser m'envoya un premier jet de son code destiné à faire suivre le courrier sur le port SMTP de la machine faisant tourner le client. J'ai réalisé pratiquement tout de suite qu'une implantation fiable de cette fonctionnalité rendrait pratiquement obsolètes tous les autres modes de distribution du courrier.

Cela faisait plusieurs semaines que je peaufinais fetchmail, pas à pas, tout en trouvant mon interface fonctionnelle mais un peu pataude -- inélégante et truffée de trop nombreuses options exiguës qui sortaient de toutes parts. J'étais particulièrement turlupiné par les options proposant de déposer le courrier récupéré dans un fichier boîte aux lettres ou sur la sortie standard, sans trop savoir pourquoi.

Je compris, quand l'idée de faire suivre le courrier sur le port SMTP traversa mon esprit, que popclient cherchait à trop en faire. Il avait été conçu pour jouer à la fois le rôle de la Poste (MTA, agent de transport du courrier) et du facteur (MDA, agent de distribution du courrier). En choisissant la solution SMTP, il n'aurait plus à s'occuper de tout ce qui est MDA et pourrait se concentrer sur le côté MTA, en passant le relais à d'autres programmes pour la distribution en local, tout comme sendmail le fait.

Pourquoi s'encombrer de toute la complexité inhérente à la configuration d'un facteur, pourquoi verrouiller un fichier boîte aux lettres, y ajouter du courrier, alors que le port 25 nous tendait les bras de façon pratiquement assurée sur toute plate-forme proposant TCP/IP depuis toujours ? Surtout que choisir cette solution garantit au courrier récupéré sa ressemblance avec du courrier normalement envoyé par son véritable auteur via SMTP, et que c'est exactement ce que nous cherchions.

Il y a plusieurs morales à cette histoire. Tout d'abord, l'idée de faire suivre le courrier vers le port SMTP fut le plus grand profit que je tirai de la tentative consciente de copier les méthodes de Linus. C'est un utilisateur qui m'a soufflé cette idée formidable -- et je n'ai eu qu'à en comprendre les conséquences.

11. Il est presque aussi important de savoir reconnaître les bonnes idées de vos utilisateurs que d'avoir de bonnes idées vous-même. C'est même préférable, parfois.

De manière surprenante, vous comprendrez vite que si vous êtes parfaitement honnête quant à ce

que vous devez aux autres, tout en restant en retrait, on finira par considérer que c'est vous qui avez tout écrit tout seul et que vous restez modeste par humilité. On sait tous comment cela a bien fonctionné pour Linus !

(Quand j'ai présenté cet article à la conférence sur Perl en août 1997, Larry Wall, l'inventeur de Perl, était assis au premier rang. Alors que j'abordai le paragraphe précédent il cria, d'un ton enthousiaste et fervent:

> NdT Larry se moquait gentiment de lui-même; il fréquente des groupes chrétiens qui font des réunions assez pittoresques.

``Oui, dis-le leur, mon frère !", déclenchant un rire généralisé. Tout le monde dans l'assistance savait comment cela avait bien fonctionné pour l'inventeur de Perl, également.)

Après quelques semaines de travail sur le projet dans cet esprit-là, je commençai à être félicité par des gens qui n'étaient pas mes utilisateurs, mais qui avaient entendu parler du projet. J'ai archivé certains de ces courriers; je les consulterai de nouveau si un jour je me demande avec angoisse si ma vie a été utile à quelque chose :-).

Mais on doit retenir deux autres leçons essentielles, apolitiques, valables pour toutes sortes de conceptions de procédés nouveaux.

12. Bien souvent, les solutions les plus innovantes, les plus frappantes, apparaissent lorsque vous réalisez que votre approche du problème était mauvaise.

J'avais tenté de résoudre le mauvais problème en m'acharnant à développer popclient comme un MTA/MDA combinés, m'encombrant d'un tas de modes de distribution exotiques en local. Il fallait repenser la conception de fetchmail du tout au tout, se contenter d'en faire un MTA, un maillon de la chaîne du courrier géré par SMTP sur l'Internet.

Quand vous vous heurtez à un mur au cours du développement d'un projet -- quand vous avez du mal à réfléchir au-delà de la prochaine génération de corrections -- c'est souvent le bon moment pour vous demander, non pas si vous apportez la bonne réponse, mais plutôt si vous posez la bonne question. Il se peut qu'il faille recadrer le problème.

Je le recadrai. Clairement, il me fallait maintenant (1) bidouiller pour ajouter la possibilité de faire suivre le courrier vers le port SMTP dans le pilote générique, (2) en faire le mode par défaut, et (3), finalement, abandonner tous les autres modes de livraison, en particulier les options de livraison dans un fichier ou sur la sortie standard.

J'ai eu des scrupules à franchir la troisième étape pendant un moment, craignant le courroux de mes utilisateurs fidèles, utilisant les autres modes de distribution. En théorie, il leur suffisait de mettre en place des fichiers .forward ou leur équivalent dans un autre système que sendmail pour obtenir exactement les mêmes effets. Dans la pratique, il se pouvait que la transition soit difficile.

Mais quand je le fis, finalement, les avantages furent énormes. Les parties les plus alambiquées du code propre au pilote disparurent. La configuration devenait simplissime -- plus de baratin sur la manière de gérer le MDA et la boîte aux lettres de l'utilisateur, plus d'ennuis quant à savoir si le système d'exploitation sous-jacent permettait de verrouiller des fichiers.

Il faut également noter que l'unique manière de perdre son courrier disparut par la même occasion.

Si vous demandiez la distribution du courrier vers un fichier et que le disque était plein, votre courrier était perdu. Cela ne peut pas se produire en faisant suivre le courrier vers le port SMTP, parce que la machine distante ne vous dira pas que tout va bien si elle ne peut effectivement distribuer le message, ou tout du moins le mettre de côté pour plus tard.

De plus, les performances de mon programme s'en trouvaient améliorées (mais pas au point de le remarquer au cours d'une seule session). Un autre bénéfice non négligeable de ce changement de cap fut la simplification drastique de la documentation (la page de `man`).

Plus tard, il me fallut revenir à un mode de distribution (MDA) en local spécifié par l'utilisateur, de manière à me permettre de gérer certaines situations obscures liées au SLIP dynamique. Mais j'ai trouvé une façon de faire bien plus simple.

La morale ? N'hésitez pas à jeter aux orties des fonctionnalités dépassées quand vous le pouvez sans perdre en efficacité. Antoine de Saint-Exupéry (qui, lorsqu'il n'écrivait pas des classiques pour enfants, pilotait et construisait des avions), a dit:

13. ``La perfection est atteinte non quand il ne reste rien à ajouter, mais quand il ne reste rien à enlever.''

C'est quand votre code devient meilleur et plus simple, que vous *savez* qu'il est bon. Et au passage, la conception de fetchmail développa sa propre identité, différente de son ancêtre popclient.

Le temps était venu pour un changement de nom. La nouvelle conception du nouveau programme ressemblait beaucoup plus à une réplique de sendmail qu'à l'ancien popclient; tous deux sont des MTAs, mais alors que sendmail envoie puis distribue, le nouveau popclient récupère avant de distribuer. Deux mois après le démarrage du projet, j'en changeai le nom en fetchmail.

7. Fetchmail grandit

Me voici donc avec une conception propre et novatrice, un code dont je savais qu'il fonctionnait bien puisque je l'utilisais tous les jours, et une liste bêta bourgeonnante. Je me rendis peu à peu compte que je n'étais plus engagé dans une bidouille personnelle et triviale que d'autres pourraient éventuellement trouver utile. J'étais le responsable d'un programme réellement utile à tout bidouilleur possédant une babasse sous Unix et une connexion par SLIP/PPP.

Quand j'y eus injecté la fonctionnalité de SMTP, il surpassa tellement ses concurrents qu'il se mua en un ``programme de référence'', un de ces programmes classiques qui remplissent si bien leur rôle qu'il élimine toute compétition parce qu'on en oublie complètement les autres solutions, au lieu de simplement les laisser de côté.

Je ne pense pas qu'on puisse vraiment viser ou planifier un tel but. On y est attiré par des idées de conception si puissantes qu'après coup les résultats paraissent tout naturels, inévitables, prédestinés. La seule manière de tomber sur ce genre d'idées est d'avoir beaucoup d'idées -- ou d'avoir assez de discernement pour reconnaître les bonnes idées des autres, même si elles vous mènent plus loin que vous ne pensiez aller.

Andrew Tanenbaum eut l'idée originale de construire un Unix simple en natif pour le 386, afin de l'utiliser comme outil pour l'enseignement. Linus Torvalds a poussé le concept de Minix bien plus

loin sans doute que ce qu'Andrew avait en tête -- et il le transforma en quelque chose de merveilleux. De la même manière (mais pas à la même échelle), j'avais pris quelques idées de Carl Harris et de Harry Hochheiser et je les avais exploitées à fond. Aucun de nous n'est 'original' de la manière, romantique, dont les gens imaginent le génie. Mais la majeure partie des sciences, de l'ingéniérie et du développement logiciel n'est pas le fruit du génie à l'état pur, malgré ce que peuvent laisser croire les mythes des bidouilleurs.

Les résultats furent à peu près une excitation identique -- en fait, le genre de choses dont tous les bidouilleurs rêvent ! Et ils signifiaient simplement qu'il me faudrait être encore plus exigeant avec moi-même. Pour rendre fetchmail aussi bon que j'entrevoyais qu'il pouvait l'être, il ne faudrait plus me contenter d'écrire du code répondant à mes propres besoins, il me faudrait prendre en compte les désirs et les souhaits des autres pour des fonctionnalités que je n'utiliserais pas. Tout en gardant le programme simple et robuste.

Le premier ajout que j'écrivis après cela, et il fut de taille, fut la possibilité de distribuer du courrier dans des 'immeubles' (multidrop) -- récupérer du courrier à partir d'une boîte aux lettres correspondant à une adresse commune, dans laquelle s'étaient accumulés des plis, et redistribuer chacun d'entre eux dans les boîtes personnelles de leur véritable destinataire.

J'ai en partie accepté d'ajouter le multidrop parce qu'on me le réclamait à grands cris, mais surtout parce que je pensais qu'il m'aiderait à traquer les bogues subsistant dans le code 'single-drop' (mono-destinataire) en me forçant à gérer les adresses dans toute leur généralité. C'est ce qui s'est passé. Il me fallut extrêmement longtemps pour gérer la syntaxe du RFC 822

> NdT les RFC sont des protocoles décidés en commun sur Internet. Ce sont les initiales de *Request For Comments*, littéralement ``(nous) réclamons des commentaires (sur ce qui suit)".

, non qu'il contienne des choses extrêmement compliquées, mais que dans son ensemble il fait intervenir un enchevêtrement de détails un peu tordus.

Mais l'adressage multidrop s'avéra également être une excellente décision de conception. Voici comment je m'en rendis compte:

14. Tout outil doit être utile par rapport aux utilisations qu'il a été prévu d'en faire. Mais on reconnaît un outil vraiment excellent au fait qu'il se prête à des usages totalement insoupçonnés.

L'utilisation inattendue d'un fetchmail multi-destinataires est la gestion de listes de courrier électronique où la liste est conservée et où le développement des alias est fait du côté *client* de la connexion SLIP/PPP. Cela signifie qu'on peut modérer une liste de courrier électronique à partir d'une machine personnelle, via un compte chez un fournisseur de services pour Internet

> NdT ISP, parfois FAI

sans devoir accéder sans cesse aux fichiers d'alias du fournisseur.

Une autre amélioration importante exigée par mes bêta-testeurs, fut la possibilité d'utiliser le format MIME en 8 bits. Cela fut assez simple à faire, parce que j'avais pris soin de laisser mon code compatible 8 bits. Non que je prévoyais ce souhait des utilisateurs, mais plutôt en accord avec une

autre règle:

*15. Quand vous écrivez un logiciel jouant le rôle d'une passerelle quelconque, prenez soin de perturber le moins possible le flot de données -- et ne perdez *jamais* d'éléments d'information, à moins que la machine destinataire vous y oblige !*

Si je n'avais pas obéi à cette règle, le support du MIME en 8 bits aurait été difficile à implanter et instable. En l'état, il me suffit de lire le RFC 1652 et d'ajouter un petit peu de logique triviale pour la génération des en-têtes.

Quelques utilisateurs européens m'ont ennuyé jusqu'à ce que j'ajoute une option pour limiter le nombre de messages transférés à chaque session (pour qu'ils puissent contrôler le prix de la communication téléphonique, sur leur onéreux réseau). J'ai mis du temps à m'y résoudre, et aujourd'hui encore je n'en suis pas très content. Mais quand vous êtes au service du monde entier, il vous faut écouter vos consommateurs -- le fait qu'ils ne vous paient pas en monnaie sonnante et trébuchante ne change rien à l'affaire.

8. Quelques enseignements supplémentaires tirés de fetchmail

Avant de retourner à des sujets plus généraux du génie logiciel, il nous reste à méditer sur quelques leçons supplémentaires tirées de l'expérience de fetchmail.

La syntaxe du fichier rc spécifie quelques mots clés ``parasites'' optionnels, complètement ignorés par l'analyseur syntaxique. Ils permettent l'utilisation d'une syntaxe proche de la langue anglaise, nettement plus lisible que les traditionnels, et spartiates, couples mot clé-valeur qu'on peut lire quand on ne garde que l'indispensable.

L'idée m'est venue tard un soir, alors que je remarquais combien ces déclarations du fichier rc commençaient à prendre l'allure d'un mini-langage impératif. (C'est aussi la raison pour laquelle j'ai remplacé le mot clé `server' (serveur), originalement présent dans popclient, par `poll' (vérification de la présence d'activité)).

Il m'a semblé que rendre ce mini-langage plus proche de l'anglais en rendrait l'utilisation plus aisée. Il me faut préciser que, bien que farouche partisan de l'école de conception ``faites-en un langage'', illustrée par des outils comme Emacs, HTML et de nombreux moteurs de bases de données, je ne suis pas, d'habitude, un admirateur des syntaxes ``à l'anglaise''.

De manière traditionnelle, les programmeurs ont eu tendance à favoriser des syntaxes de contrôle très précises et compactes, sans redondance. Il s'agit d'un héritage culturel du temps où les ressources informatiques coûtaient cher, et où il fallait donc rendre les étapes d'analyse syntaxique aussi courtes et simples que possible. La langue anglaise, pour moitié redondante, semblait alors un modèle très peu approprié.

Ce n'est pas la raison pour laquelle je dénonce la non utilisation de syntaxes naturelles; je n'en fais mention ici que pour mieux la démolir. Maintenant que les ressources informatiques sont pléthore, l'austérité ne doit pas être une fin en soi. De nos jours, il est plus important qu'un langage soit facilement compréhensible par les humains plutôt que par les ordinateurs.

Il existe, en revanche, de bonnes raisons pour se méfier. L'une d'entre elles est le coût en complexité

de la phase d'analyse syntaxique -- vous n'avez certainement pas envie d'en faire une source non négligeable de bogues. Vous n'avez pas envie non plus que la complexité de votre langage déroute l'utilisateur. Une autre raison est que les tentatives de rendre la syntaxe d'un langage similaire à l'anglais, aboutissent le plus souvent à un telle déformation de cet ``anglais'' que cette ressemblance superficielle au langage naturel provoque autant de confusion que ne l'aurait fait une syntaxe traditionnelle. (C'est le cas de nombreux langages prétendument de quatrième génération et des langages de requêtes dans des bases de données commerciales.)

La syntaxe permettant de contrôler fetchmail évite ces deux écueils parce que le domaine du langage est très restreint. Il n'est en rien un langage généraliste; il se contente de dire des choses simples, de telle sorte qu'il est difficile de se tromper en transposant mentalement un minuscule sous-ensemble de l'anglais au langage de contrôle en lui-même. Je pense qu'il y a ici matière à une leçon plus générale:

16. Quand votre langage est loin d'être Turing équivalent

> *NdT c'est le cas par exemple des langages de programmation*

, un peu de ``sucre syntaxique'' ne peut qu'aider.

Une autre leçon concerne la sécurité par hermétisme (utilisant par exemple des mots de passes secrets). Certains utilisateurs de fetchmail m'ont demandé de modifier le logiciel de manière à stocker des mots de passe chiffrés dans le fichier rc, de telle sorte que les petits malins ne puissent les lire directement.

Je ne l'ai pas fait, parce que cela n'apporte aucune protection supplémentaire. Quiconque a obtenu le droit de lire votre fichier rc pourra lancer fetchmail sous votre identité de toutes manières -- et s'il en a après votre mot de passe, il pourra toujours extraire du code de fetchmail le décodeur nécessaire pour le déchiffrer.

Tout ce qu'un chiffrement de mots de passe dans le fichier .fetchmailrc aurait apporté, c'est un sentiment de sécurité infondé à ceux qui connaissent mal les problèmes de sécurité. La règle générale est ici:

17. Un système de sécurité n'est pas plus sûr que le secret (la clé) qui le garde. Gare aux pseudo secrets !

9. Prérequis nécessaires au style bazar

Les premiers relecteurs et les premiers publics auprès desquels cet article a été testé ont posé de manière régulière la question des prérequis nécessaires à la réussite d'un développement dans le style bazar, en incluant dans cette question aussi bien les qualifications du chef de projet que l'état du code au moment où il est rendu public et tente de rallier autour de lui une communauté de co-développeurs.

Il est assez évident qu'il n'est pas possible de commencer à coder dans le style bazar dès le début. On peut tester, déboguer et améliorer un programme dans le style bazar, mais il sera très difficile de *faire démarrer* un projet dans le mode bazar. Linus ne l'a pas tenté, moi non plus. Il faut que votre communauté naissante de développeurs ait quelque chose qui tourne, qu'on puisse tester, avec quoi

elle puisse jouer.

Quand vous initiez un travail de développement en communauté, il vous faut être capable de présenter une *promesse plausible*. Votre programme ne doit pas nécessairement fonctionner très bien. Il peut être grossier, bogué, incomplet, et mal documenté. Mais il ne doit pas manquer de convaincre des co-développeurs potentiels qu'il peut évoluer en quelque chose de vraiment bien dans un futur pas trop lointain.

Linux et fetchmail furent tous deux rendus publics avec des conceptions simples, fortes et séduisantes. Nombreux sont ceux qui, après avoir réfléchi au modèle du bazar tel que je l'ai présenté ici, ont très correctement considéré que cela était critique, et en ont rapidement conclu qu'il était indispensable que le chef de projet fasse preuve au plus haut point d'astuce et d'intuition dans la conception.

Mais Linus se fonda sur la conception d'Unix. Ma conception provenait initialement du vieux programme popclient (bien que beaucoup de choses aient changé à terme, proportionnellement bien plus que dans Linux). Alors faut-il que le chef/coordinateur d'un effort commun mené dans le style bazar ait un talent exceptionnel pour la conception, ou peut-il se contenter d'exalter le talent des autres ?

Je pense qu'il n'est pas critique que le coordinateur soit capable de produire des conceptions exceptionnellement brillantes. En revanche, il est absolument critique que le coordinateur soit capable de *reconnaître les bonnes idées de conception des autres*.

Les projets Linux et fetchmail en sont tous deux la preuve. Linus, bien qu'il ne soit pas (comme nous l'avons vu précédemment) un concepteur spectaculairement original, a démontré une puissante capacité à reconnaître une bonne conception et à l'intégrer au noyau Linux. Et j'ai déjà décrit comment l'idée la plus puissante dans la conception de fetchmail (faire suivre le courrier vers le port SMTP) me fut soufflée par quelqu'un d'autre.

Les premières personnes auxquelles j'ai présenté cet article m'ont complimenté en me suggérant que je suis prompt à sous-évaluer dans la conception des projets menés dans le style bazar, leur originalité -- principalement parce que j'en suis pas mal pourvu moi-même, ce qui me conduit à considérer cela comme acquis. Il peut y avoir un fond de vérité là dedans; la conception (à l'opposé de la programmation ou du débogage) est certainement mon point fort.

Mais le problème avec l'intelligence et l'originalité dans la conception de logiciels, c'est que ça devient une habitude -- presque par réflexe, vous commencez à faire dans l'esthétique et le compliqué alors qu'il faudrait rester simple et robuste. Certains de mes projets n'ont jamais abouti à cause de cette erreur, mais ce ne fut pas le cas pour fetchmail.

Aussi suis-je convaincu que le projet fetchmail a réussi en partie parce que j'ai refoulé ma tendance à donner dans la subtilité; cela contredit (au moins) l'idée que l'originalité dans la conception est essentielle dans des projets réussis menés dans le style bazar. Et pensez à Linux. Imaginez que Linus ait tenté de mettre en pratique des innovations fondamentales dans la conception de systèmes d'exploitation au cours du développement; est-il probable que le noyau qui en résulterait soit aussi stable et populaire que celui que nous connaissons ?

Il faut, bien sûr, une certaine habileté dans la conception et le codage, mais je pense que quiconque

envisage sérieusement de lancer un projet dans le style en possède déjà le minimum nécessaire. Les lois du marché de la réputation interne au monde du logiciel dont le code source est ouvert exercent des pressions subtiles décourageant ceux qui pensent mettre à contribution d'autres développeurs alors qu'ils n'ont pas eux-mêmes la compétence d'assurer le suivi. Jusqu'à présent tout cela semble avoir très bien marché.

Il existe une autre qualité qui n'est pas normalement associée au développement logiciel, mais je pense qu'elle est aussi importante qu'une bonne intelligence de la conception pour les projets de style bazar -- et elle est peut-être bien plus importante. Le chef, ou coordinateur, d'un projet dans le style bazar doit être bon en relations humaines et avoir un bon contact.

Cela devrait être évident. Pour construire une communauté de développement, il vous faut séduire les gens, les intéresser à ce que vous faites, et les encourager pour les petits bout du travail qu'ils réalisent. De bonnes compétences techniques sont essentielles, mais elles sont loin de suffire. La personnalité que vous projetez compte aussi.

Cela n'est pas une coïncidence que Linus soit un chic type qu'on apprécie volontiers et qu'on a envie d'aider. Cela n'est pas fortuit non plus que je sois un extraverti énergique qui aime animer les foules et qui se comporte et réagit comme un comique de théâtre. Pour que le modèle du bazar fonctionne, une petite touche de charme et de charisme aide énormément.

10. Le contexte social du logiciel dont le code source est ouvert

Cela est bien connu: les meilleures bidouilles commencent en tant que solutions personnelles aux problèmes de tous les jours rencontrés par leur auteur, et elles se répandent parce que ce problème est commun à de nombreuses personnes. Cela nous ramène à la règle numéro 1, reformulée de manière peut-être plus utile:

18. Pour résoudre un problème intéressant, commencez par trouver un problème qui vous intéresse.

Ainsi fut-il de Carl Harris et du vieux popclient, ainsi fut-il de moi et de fetchmail. Mais cela est bien compris depuis longtemps. Ce qui compte, le détail sur lequel les histoires de Linux et de fetchmail semblent nous demander de nous concentrer, est l'étape suivante -- l'évolution du logiciel en présence d'une communauté d'utilisateurs et de co-développeurs importante et active.

Dans ``Le mythe du mois-homme'', Fred Brooks observa qu'on ne peut pas diviser et répartir le temps du programmeur; si un projet a du retard, lui ajouter des développeurs ne fera qu'accroître son retard. Il explique que les coûts de communication et de complexité d'un projet augmentent de manière quadratique avec le nombre de développeurs, alors que le travail réalisé n'augmente que linéairement. Depuis, cela est connu sous le nom de ``loi de Brooks'', et on la considère en général comme un truisme. Mais si seule la loi de Brooks comptait, Linux serait impossible.

Le classique de Gerald Weinberg ``La psychologie de la programmation sur ordinateur'' apporta ce qu'on pourrait considérer après coup comme une correction vitale à Brooks. Dans sa discussion sur la ``programmation non égoïste'', Weinberg observa que dans les boîtes où les programmeurs ne marquent pas le territoire de leur code, et encouragent les autres à y chercher les bogues et les améliorations potentielles, les améliorations sont drastiquement plus rapides qu'ailleurs.

Les termes choisis par Weinberg l'ont peut-être empêché de recevoir toute la considération qu'il

méritait -- et l'idée que les bidouilleurs sur Internet puissent être décrits comme ``non égoïstes'' fait sourire. Mais je pense que cet argument semble aujourd'hui plus vrai que jamais.

L'histoire d'Unix aurait dû nous préparer à ce que nous découvrons avec Linux (et à ce que j'ai expérimenté à une échelle plus modeste en copiant délibérément les méthodes de Linus): alors que l'acte de programmation est essentiellement solitaire, les plus grandes bidouilles proviennent de la mise à contribution de l'attention et de la puissance de réflexion de communautés entières. Le développeur qui n'utilise que son propre cerveau dans un projet fermé ne tiendra pas la route face au développeur qui sait comment créer un contexte ouvert, susceptible d'évoluer, dans lequel la traque des bogues et les améliorations sont effectuées par des centaines de gens.

Mais le monde traditionnel d'Unix n'a pas poussé cette approche dans ses derniers retranchements pour plusieurs raisons. L'un d'entre eux concernait les contraintes légales des diverses licences, secrets de fabrication, et autres intérêts commerciaux. Un autre (mieux compris plus tard) était que l'Internet n'était pas encore assez mûr.

Avant que les coûts d'accès à Internet ne chutent, il existait quelques communautés géographiquement compactes dont la culture encourageait la programmation ``non égoïste'' à la Weinberg, et un développeur pouvait facilement attirer tout un tas de co-développeurs et autres touche-à-tout doués. Les laboratoires Bell, le laboratoire d'intelligence artificielle de l'institut de technologie du Massachusetts (MIT), l'université de Californie à Berkeley, devinrent les foyers d'innovations légendaires qui sont restés puissants.

Linux fut le premier projet qui fit un effort conscient et abouti pour utiliser le *monde entier* comme réservoir de talent. Je ne pense pas que cela soit une coïncidence que la période de gestation de Linux ait coïncidé avec la naissance du World Wide Web, ni que Linux ait quitté le stade de la petite enfance en 1993-1994, au moment où l'intérêt général accordé à Internet et que l'industrie des fournisseurs d'accès explosèrent. Linus fut le premier à comprendre comment jouer selon les nouvelles règles qu'un Internet omniprésent rendait possibles.

Même s'il fallait qu'Internet ne coûte pas cher pour que le modèle Linux se développe, je ne pense pas que cela soit suffisant. Il y avait un autre facteur vital: le développement d'un style de direction de projet et d'un ensemble de coutumes de coopération qui permettaient aux développeurs d'attirer des co-développeurs et de rentabiliser au maximum ce nouveau média.

Quel est donc ce style de direction, quelles sont donc ces coutumes ? Ils ne peuvent pas être fondés sur des rapports de force -- même si c'était le cas, la direction par coercition ne produirait pas les résultats qu'on peut observer. Weinberg cite fort à propos l'autobiographie de Pyotr Alexeyvitch Kropotkine, anarchiste russe du XIXe siècle, ``Mémoires d'un révolutionnaire'':

``Élevé dans une famille possédant des serfs, j'entrai dans la vie active, comme tous les jeunes gens de mon époque, avec une confiance aveugle dans la nécessité de commander, d'ordonner, de brimer, de punir et ainsi de suite. Mais quand, assez tôt, je dus diriger d'importantes affaires et côtoyer des hommes *libres*, et quand chaque erreur pouvait être immédiatement lourde de conséquences, je commençai à apprécier la différence entre agir selon les principes du commandement et de la discipline et agir selon le principe de la bonne intelligence. Le premier fonctionne admirablement dans un défilé militaire, mais ne vaut rien dans la vie courante, où on ne peut atteindre son but que grâce à l'effort soutenu de nombreuses volontés travaillant dans le même sens."

``L'effort soutenu de nombreuses volontés travaillant dans le même sens'', c'est exactement ce qu'un projet comme Linux demande -- et le ``principe de commandement'' est en effet impossible à appliquer aux volontaires de ce paradis de l'anarchie que nous appelons Internet. Pour opérer et se mesurer les uns aux autres de manière efficace, les bidouilleurs qui cherchent à prendre la tête d'un projet de collaboration commune doivent apprendre à recruter et à insuffler de l'énergie dans les communautés d'intérêts convergents à la manière vaguement suggérée par le ``principe de bonne intelligence'' de Kropotkine. Ils doivent apprendre à utiliser la loi de Linus.

Plus haut, je me référais à ``l'effet de Delphes'' comme une possible explication de la loi de Linus. Mais on ne peut s'empêcher de penser à des analogies très puissantes avec des systèmes adaptatifs en biologie et en économie. Le monde Linux sous de nombreux aspects, se comporte comme un marché libre ou un écosystème, un ensemble d'agents égoïstes qui tentent de maximiser une utilité, ce qui au passage produit un ordre spontané, auto-correcteur, plus élaboré et plus efficace que toute planification centralisée n'aurait pu l'être. C'est ici qu'il faut chercher le ``principe de bonne intelligence''.

La ``fonction d'utilité'' que les bidouilleurs Linux maximisent n'est pas classiquement économique, c'est l'intangible de leur propre satisfaction personnelle et leur réputation au sein des autres bidouilleurs. (On peut être tenté de penser que leur motivation est ``altruiste'', mais c'est compter sans le fait que l'altruisme est en soi une forme d'égoïsme pour l'altruiste). Les cultures volontaires qui fonctionnent ainsi ne sont pas si rares; j'ai déjà participé à des projets semblables dans les fanzines de science-fiction, qui au contraire du monde de la bidouille, reconnaissent explicitement que l'``egoboo'' (le fait que sa réputation s'accroisse auprès des autres fans) est le principal moteur qui propulse les activités volontaires.

Linus, en se positionnant avec succès comme gardien des clés d'un projet où le développement est surtout fait par les autres, et en y injectant de l'intérêt jusqu'à ce qu'il devienne auto-suffisant a montré une intelligence profonde du ``principe de bonne intelligence'' de Kropotkine. Cette vision quasi-économique du monde de Linux nous permet de comprendre comment cette bonne intelligence est mise en oeuvre.

On peut analyser la méthode de Linus comme un moyen de créer de manière efficace, un marché de l'``egoboo'' -- relier les égoïsmes individuels des bidouilleurs aussi fermement que possible, dans le but de réaliser des tâches impossibles sans une coopération soutenue. Avec le projet fetchmail, j'ai montré (à une échelle plus modeste, je vous l'accorde) qu'on peut dupliquer ses méthodes avec de bons résultats. Peut-être même l'ai-je fait un peu plus consciemment et plus systématiquement que lui.

Nombreux sont ceux (en particulier ceux dont les opinions politiques les font se méfier des marchés libres) qui s'attendraient à ce que la culture d'égoïstes sans autre maître qu'eux mêmes soit fragmentée, territoriale, source de gâchis, pleine de petits secrets, et hostile. Mais cette idée est clairement mise en défaut par (pour ne donner qu'un seul exemple) l'époustouflante variété, qualité et profondeur de la documentation relative à Linux. Il est pourtant légendaire que les programmeurs *détestent* écrire des documentations; comment se fait-il, alors, que les bidouilleurs de Linux en produisent tant ? Il est évident que le marché libre d'egoboo de Linux fonctionne mieux que tout autre pour générer des comportements vertueux, serviables, mieux que les services documentaires largement subventionnés des producteurs de logiciels commerciaux.

Les projets fetchmail et du noyau de Linux montrent tous deux qu'en flattant à bon escient l'ego de beaucoup d'autres bidouilleurs, un coordinateur-développeur fort peut utiliser l'Internet pour tirer parti du fait d'avoir énormément de co-développeurs sans que le projet ne s'effondre en un capharnaüm chaotique. C'est pourquoi je propose la contrepartie suivante à la loi de Brooks:

19: Pour peu que le coordinateur du développement dispose d'un moyen de communication au moins aussi bon que l'Internet, et pour peu qu'il sache comment mener ses troupes sans coercition, il est inévitable qu'il y ait plus de choses dans plusieurs têtes que dans une seule.

Je pense qu'à l'avenir, le logiciel dont le code source est ouvert sera de plus en plus entre les mains de gens qui savent jouer au jeu de Linus, des gens qui abandonnent les cathédrales pour se consacrer entièrement au bazar. Cela ne veut pas dire que les coups de génie individuels ne compteront plus; je pense plutôt que l'état de l'art du logiciel dont le code source est ouvert appartiendra à ceux qui commencent par un projet individuel génial, et qui l'amplifieront à travers la construction efficace de communautés d'intérêt volontaires.

Et peut-être même qu'il n'est pas question ici que de l'avenir du logiciel *dont le code source est ouvert*. Aucun développeur qui fonctionne avec un code source fermé ne peut mobiliser autant de talent que celui que la communauté Linux peut consacrer à un problème donné. Bien rares même sont ceux qui auraient eu les moyens de rémunérer les deux cents et quelques contributeurs à fetchmail !

Peut-être qu'à terme, la culture du logiciel dont le code source est ouvert triomphera, non pas parce qu'il est moralement bon de coopérer, non pas parce qu'il est moralement mal de ``clôturer'' le logiciel (en supposant que vous soyez d'accord avec la deuxième assertion; ni Linus ni moi ne le sommes), mais simplement parce que le monde dont le code source est fermé ne peut pas gagner une course aux armements évolutive contre des communautés de logiciel libre, qui peuvent mettre sur le problème un temps humain cumulé plus important de plusieurs ordres de grandeurs.

11. Remerciements

Un grand nombre de personnes m'ont aidé, au travers de conversations, à améliorer cet article. Je remercie tout particulièrement Jeff Dutky <dutky@wam.umd.edu> qui m'a suggéré la formule ``le débogage est parallélisable'', et qui m'a aidé à tirer l'analyse qui en découle. Merci aussi à Nancy Lebovitz <nancyl@universe.digex.net> qui m'a suggéré que je suivais l'exemple de Weinberg en citant Kropotkine. J'ai reçu aussi des critiques utiles de la part de Joan Eslinger <wombat@kilimanjaro.engr.sgi.com> et de Marty Franz <marty@net-link.net> de la liste de General Technics. Je remercie les membres du PLUG, le groupe d'utilisateurs de Linux de Philadelphie (jeu de mot signifiant aussi ``branché''), pour avoir joué le rôle du premier public critique de cet article. Enfin, les commentaires de Linus Torvalds me furent utiles, et son soutien des premières heures m'encouragea beaucoup.

12. Pour aller plus loin

J'ai cité plusieurs extraits du classique de P. Brooks *The Mythical Man-Month* (*Le mythe du mois-homme*) parce que, sous bien des aspects, ses conclusions doivent encore être affinées. Je

recommande chaleureusement l'édition faite par la société Addison-Wesley (ISBN 0-201-83595-9) à l'occasion du vingt-cinquième anniversaire de ce livre, qui ajoute son article de 1986 intitulé ``*No Silver Bullet*'' (pas de balle en argent).

Cette nouvelle édition est préfacée par une inestimable rétrospective des 20 dernières années dans laquelle Brooks reconnaît les quelques affirmations de son texte original qui n'ont pas résisté à l'épreuve du temps.

J'ai lu la rétrospective pour la première fois après avoir écrit la quasi totalité du présent article, et j'ai été surpris de constater que Brooks attribue des pratiques de type bazar à Microsoft !

De Gerald M. Weinberg, *The Psychology Of Computer Programming* (la psychologie de la programmation des ordinateurs) (New York, Van Nostrand Reinhold 1971) introduisit le concept assez maladroitement appelé de ``programmation non égoïste''. Bien que n'étant pas le premier à se rendre compte de la futilité du ``principe de commandement'', il fut probablement le premier à reconnaître cet aspect et à argumenter sur ce dernier par rapport au développement logiciel.

Richard P. Gabriel, en contemplant la culture Unix de l'avant-Linux, argumenta à contrecoeur en faveur de la supériorité d'un modèle de type bazar primitif dans son article de 1989 *Lisp: Good News, Bad News, and How To Win Big* (Lisp: bonnes nouvelles, mauvaises nouvelles, et comment gagner gros). Bien qu'il ait mal vieilli sous certains aspects, cet essai est toujours adulé à juste titre dans les cercles de fans de Lisp (dont je fais partie). Un correspondant m'a rappelé que la section intitulée ``Worse Is Better'' (le pire est l'ami du mieux) peut presque se lire comme une anticipation de Linux. On peut trouver ce papier sur la Toile (WWW) à l'adresse http://www.naggum.no/worse-is-better.html.

Peopleware: Productive Projects and Teams (ressources humaines: projets et équipes productifs) de De Marco et Lister's (New York; Dorset House, 1987; ISBN 0-932633-05-6) est un joyau méconnu, et j'ai été enchanté de voir Fred Brooks le citer dans sa rétrospective. Bien que la matière directement applicable au monde Linux ou au monde du logiciel dont le code source est ouvert en général soit rare, les auteurs ont de bonnes intuitions sur les prérequis nécessaires au travail créatif qui intéresseront quiconque envisage d'importer les vertus du modèle du bazar dans un contexte commercial.

Enfin, je dois admettre que j'ai failli appeler cet article ``La cathédrale et l'agora'', du mot grec désignant un marché ouvert ou un lieu public de rencontres. Les articles fondateurs ``agoric systems'' (systèmes agoriques), de Mark Miller et Eric Drexler, en décrivant les propriétés émergentes d'écosystèmes informatiques similaires au libre marché, m'ont aidé à avoir les idées plus claires sur des phénomènes analogues dans la culture du logiciel dont le code source est ouvert quand Linux m'a mis le nez dedans, cinq ans plus tard. On peut trouver ces papiers sur la Toile à l'adresse http://www.agorics.com/agorpapers.html.

13. Épilogue: Netscape embrasse la méthode du bazar !

Ça fait tout drôle de réaliser qu'on participe à l'Histoire...

Le 22 janvier 1998, environ sept mois après ma première publication de cet article, Netscape Communications, Inc. a rendu public son projet de donner libre accès au code source de Netscape

Communicator. Je n'avais pas la moindre idée que cela se produirait avant qu'ils ne l'annoncent.

Eric Hahn, vice-président et responsable en chef de la technologie à Netscape, m'a envoyé un courrier électronique peu après en me disant: ``Au nom de tout le monde à Netscape, je souhaite vous remercier pour nous avoir, le premier, aidés à comprendre tout cela. Votre réflexion et vos écrits ont été des inspirations cruciales dans la prise de cette décision."

La semaine suivante, à l'invitation de Netscape, je me suis rendu dans la ``Silicon Valley"

> NdT c'est l'endroit, près de San Francisco, où sont concentrées de nombreuses entreprises qui travaillent sur l'électronique des ordinateurs et sur les ordinateurs eux-mêmes.

pour suivre une conférence de stratégie d'un jour (le 4 février 1998) en compagnie de certains de leurs cadres et de leurs techniciens les plus importants. Nous avons mis au point la stratégie de mise à disposition du code source de Netscape ainsi que la licence sous laquelle il serait rendu public, et nous avons fait quelques projets dont nous espérons qu'ils auront, à terme, un impact très étendu et très positif sur la communauté du logiciel dont le code source est ouvert. Au moment où j'écris ces lignes, il est un peu trop tôt pour donner plus de détails; mais vous en prendrez connaissance dans les semaines à venir.

Netscape est sur le point de nous proposer une expérience à grande échelle, une expérience dans des conditions réelles du modèle du bazar dans une optique commerciale. La culture du logiciel dont le code source est ouvert affronte maintenant un danger; si le projet de Netscape ne donne pas satisfaction, cela risque de jeter tant de discrédit sur le concept de logiciel dont le code source est ouvert qu'il faudra attendre dix ans de plus pour que des sociétés commerciales s'y intéressent.

D'un autre côté, c'est également une opportunité spectaculaire. Les réactions à chaud à cet événement, à Wall Street (la bourse de New York) comme ailleurs, ont été circonspectes mais positives. On nous donne aussi l'opportunité de faire nos preuves. Si, grâce à cette décision, Netscape regagne des parts de marché de façon substantielle, cela risque de provoquer une révolution dans l'industrie du logiciel, une révolution qui aurait dû prendre part il y a si longtemps.

L'année qui vient devrait être très instructive et très intéressante.

14. Historique des versions et des modifications

$Id: cathedral-bazaar.sgml,v 1.40 1998/08/11 20:27:29 esr Exp $

J'ai lu la version 1.16 au Linux Kongress le 21 mai 1997.

J'ai ajouté la bibliographie dans la version 1.20 le 7 juillet 1997.

J'ai ajouté l'anecdote concernant la conférence sur Perl le 18 novembre 1997 dans la version 1.27.

J'ai changé ``logiciel libre" en ``logiciel dont le code source est ouvert" le 9 février 1998 dans la version 1.29.

J'ai ajouté ``Épilogue: Netscape embrasse la méthode du bazar !" le 10 février 1998 dans la version 1.31.

J'ai ôté le commentaire de Paul Eggert concernant l'opposition entre le modèle du bazar et la GPL à la suite d'arguments pertinents émis par RMS le 28 juillet 1998.

Les autres numéros de versions correspondent à des corrections éditoriales et de structuration du document mineures.

Sébastien Blondeel <sbi@linux-france.org> est responsable de la traduction en français. Il sera heureux de recueillir vos commentaires et vos remarques sur les choix qu'il a faits. Ce document aussi est ``ouvert'' !

Merci à Mark Hoebeke <mh@bambou.jouy.inra.fr> et à Julien Vayssiere <Julien.Vayssiere@sophia.inria.fr> pour leurs relectures et conseils éclairés.