

Chapitre

7

Paquets, sorties et développement quotidien

Ce chapitre se rapporte à la création et à la publication de logiciels par les projets de logiciels libres ainsi qu'à l'organisation du processus de développement tout entier autour de ces objectifs.

L'une des grandes différences entre les projets Open Source et les projets propriétaires est l'absence de contrôle centralisé sur l'équipe de développement. C'est lorsqu'arrive le temps de préparer une nouvelle sortie que cette différence est particulièrement frappante : une entreprise peut demander à toute son équipe de développement de se concentrer sur la sortie en préparation en mettant de côté le développement de nouvelles fonctionnalités et les corrections de bogues

mineurs jusqu'à la publication de la nouvelle version. Les groupes de volontaires ne sont pas aussi monolithiques. Les membres travaillent sur le projet pour de nombreuses raisons différentes, et sont libres de ne pas chambouler leurs priorités à cause d'une sortie imminente. Comme le développement ne s'arrête jamais, les procédures de sortie dans un environnement Open Source tendent à être plus longues, mais sont moins perturbantes que celles des logiciels commerciaux. On peut faire une analogie avec l'entretien d'une autoroute. Vous avez deux options pour la réparer : vous pouvez la fermer complètement afin que les équipes de travaux puissent l'envahir à loisir jusqu'à ce que le problème soit résolu, ou bien vous pouvez travailler sur quelques voies en laissant les autres ouvertes au trafic. La première solution est très efficace pour l'équipe de réparation, mais la voie est complètement fermée aux autres jusqu'à la fin du chantier. La deuxième solution implique beaucoup plus de contraintes pour l'équipe de chantier (ils doivent travailler avec moins de monde et moins d'équipement, dans des conditions plus restreintes, affecter des préposés à la circulation, etc.), mais au moins la voie reste utilisable, même si ce n'est pas à sa capacité maximale.

Les projets Open Source ont tendance à adopter la deuxième solution. En fait, pour un logiciel évolué ayant plusieurs lignes de sortie maintenues simultanément, le projet est en quelque sorte toujours en travaux. Il y a toujours quelques voies fermées, un niveau constant, mais bas, d'inconfort est toujours toléré par l'ensemble du groupe afin que les sorties puissent être faites à intervalles réguliers.

Le modèle qui rend tout cela possible ne s'applique pas qu'aux sorties. C'est le principe de parallélisation des tâches qui ne sont pas mutuellement interdépendantes, un principe qui n'est en aucun cas spécifique au développement Open Source évidemment, mais un principe que les projets Open Source adaptent à leurs besoins particuliers. Ils ne peuvent pas se permettre de trop importuner l'équipe de chantier, ou le trafic habituel, mais ils ne peuvent pas non plus se permettre d'employer du personnel spécialement pour se poster près des plots oranges et agiter un drapeau pour faire la circulation. Ils s'orientent donc vers des processus qui demandent un niveau soutenu d'administration plutôt que vers des processus irréguliers. Les volontaires s'accrochent très bien d'un niveau constant, mais faible, de désagréments. Comme tout est prévisible, ils peuvent aller et venir sans se demander si leur emploi du temps entrera en conflit avec ce qui se passe dans le projet. Mais si le projet était soumis à un emploi du temps dans lequel certaines activités en excluraient d'autres, on se retrouverait finalement avec beaucoup de développeurs n'ayant rien à faire pendant une grande partie de ce temps,

ce qui serait non seulement inefficace, mais aussi ennuyeux, et donc dangereux (il y a de fortes chances qu'un développeur qui s'ennuie quitte le projet).

Le travail de publication est généralement celui qui est le plus visible parmi les tâches qui ne sont pas directement liées, mais parallèles au développement. Ainsi, les méthodes décrites dans les parties suivantes concernent principalement les sorties. Cependant, notez qu'elles s'appliquent également à d'autres tâches parallélisables, comme les traductions et l'internationalisation, les vastes changements d'API apportés graduellement à l'ensemble du code source, etc.

7.1 Numérotation des versions

Avant d'aborder la conception d'une version, parlons d'abord de la façon de nommer les versions, et pour cela, il faut que vous connaissiez la signification des sorties pour les utilisateurs. Une sortie signifie que :

- Les anciens bogues ont été corrigés. C'est quasiment un impératif à chaque sortie.
- De nouveaux bogues ont été ajoutés. Les utilisateurs s'y attendent aussi, mais pas quand il s'agit d'un correctif de sécurité ou de quelques autres cas exceptionnels (voir la section appelée « Correctifs de sécurité » plus loin dans ce chapitre).
- De nouvelles fonctionnalités peuvent avoir été ajoutées.
- De nouvelles options de configuration peuvent avoir été ajoutées et certaines anciennes options ont pu légèrement changer. Les procédures d'installation peuvent, également, avoir été légèrement modifiées depuis la version précédente également, bien qu'en général ce ne soit pas souhaitable.
- Des changements apportés peuvent rendre certaines données incompatibles et le format de données utilisé dans d'anciennes versions du logiciel n'est plus utilisable sans passer par une étape de conversion unidirectionnelle (potentiellement manuelle).

Comme vous pouvez le voir, ce ne sont pas que de bonnes choses. C'est la raison pour laquelle les utilisateurs expérimentés appréhendent toujours un peu les sorties, surtout quand le logiciel est mature et réalise déjà presque tout ce qu'ils veulent faire avec (ou du moins, espèrent pouvoir faire). Même l'arrivée de nouvelles fonctionnalités a ses avantages et ses inconvénients puisque le comportement du logiciel peut être modifié.

La numérotation a donc deux buts : de toute évidence le numéro devrait indiquer sans ambiguïté l'ordre de sortie (c'est-à-dire qu'en regardant deux numéros de version, on devrait savoir quelle est la plus récente), mais elle devrait également indiquer de manière aussi condensée que possible l'importance et la nature des changements de cette version.

Tout cela dans un numéro ? Eh bien, plus ou moins, oui. La stratégie de numérotation des sorties est une des plus vieilles controverses (voir la section appelée « Plus le sujet est facile, plus long sera le débat » dans le chapitre 6) et je doute que l'on puisse s'entendre sur une norme unique et complète d'ici peu. Cependant, quelques bonnes stratégies se sont démarquées et un principe est accepté par tous : soyez cohérents. Choisissez une stratégie de numérotation, décrivez-la et restez-y fidèle. Les utilisateurs vous en seront reconnaissants.

7.1.1 Les éléments de la numérotation

Cette partie décrit en détail les conventions formelles de la numérotation de version et ne nécessite que peu de connaissances préalables. Vous devez la considérer comme une référence. Si vous êtes déjà familier avec ces conventions, vous pouvez vous rendre directement à la section suivante.

Les numéros de versions sont des groupes de chiffres séparés par des points :

Scanley 2.3

Singer 5.11.4

... et ainsi de suite. Les points ne sont pas des virgules pour séparer les décimales, ce sont simplement des séparateurs ; 5.3.9 sera suivi par 5.3.10. Certains projets ont parfois dévié de cette voie, l'exemple le plus connu est le noyau Linux avec sa suite 0.95, 0.96, 0.99 pour arriver à Linux 1.0, mais la convention, selon laquelle les points ne séparent pas les décimales, est maintenant bien établie et devrait être considérée comme une norme. Il n'y a aucune limite au nombre de composantes (portions de chiffres ne contenant pas de points), mais la plupart des projets ne dépassent pas trois ou quatre. Les raisons vous apparaîtront plus clairement par la suite.

En plus de la composante chiffrée, les projets ajoutent encore une étiquette telle que Alpha ou Beta, par exemple :

Scanley 2.3.0 (Alpha)

Singer 5.11.4 (Beta)

Une étiquette Alpha ou Beta signifie que cette sortie précède une sortie prochaine avec le même numéro, mais sans l'étiquette. Ainsi, 2.3.0 (Alpha) aboutira finalement à 2.3.0. Afin de laisser de la place pour plusieurs pré-sorties de suite, les étiquettes elles-mêmes peuvent porter une méta-étiquette. Par exemple, voici une série de sorties dans l'ordre où elles seraient mises à disposition du public :

```
Scanley 2.3.0 (Alpha 1)
Scanley 2.3.0 (Alpha 2)
Scanley 2.3.0 (Beta 1)
Scanley 2.3.0 (Beta 2)
Scanley 2.3.0 (Beta 3)
Scanley 2.3.0
```

Vous remarquerez que, quand il possède le qualificatif Alpha, Scanley 2.3 est écrit comme 2.3.0. Les deux nombres sont équivalents, le 0 en fin de numérotation peut toujours être abandonné par souci de concision, mais lorsqu'un qualificatif est présent, on utilise plutôt le nom complet.

D'autres qualificatifs utilisés relativement souvent sont Stable, Unstable, Development et RC (pour « Release Candidate »). Les plus utilisés restent Alpha et Beta, RC n'est pas loin derrière à la troisième place, mais vous remarquerez que RC est toujours suivi d'un méta-qualificatif numérique. C'est-à-dire que vous ne publiez pas Scanley 2.3.0 (RC), vous publiez Scanley 2.3.0 (RC1), suivi par RC2, etc.

Ces trois étiquettes, Alpha, Beta et RC sont largement connues maintenant, car ce sont des mots compréhensibles, pas du jargon, et je ne recommanderai pas l'usage des autres, même si *a priori* ce sont de meilleurs choix. Les gens qui installent des logiciels non-finalisés sont déjà familiers avec ce triptyque, et il n'y a pas de raison de réinventer la roue.

Même si les points dans les numéros de version ne sont pas des virgules, ils indiquent tout de même un certain ordre. Toutes les versions 0.x.y précèdent la 1.0 (qui est équivalente à la 1.0.0, bien sûr). La 3.14.158 précède immédiatement la 3.14.159 et ne précède pas directement la 3.14.160 ou encore la 3.15.0 et ainsi de suite.

Une politique cohérente de numérotation de version permet aux utilisateurs de distinguer, uniquement en comparant les chiffres pour un même logiciel, les différences importantes qui existent entre deux variantes. Dans le cas classique d'une numérotation à trois composantes, le premier est le nombre principal, le deuxième est le nombre mineur et le troisième est le micro-nombre. Par exemple, la version

2.10.17 est la dix-septième micro sortie de la dixième sortie mineure au sein de la deuxième version principale. Les mots « lignes » et « séries » sont utilisés de manière informelle ici, mais ils veulent dire ce qu'ils veulent dire. Une série majeure comprend toutes les sorties qui partagent le même nombre principal, et une série mineure (une ligne mineure) contient toutes les versions qui partagent le même nombre mineur et le même nombre principal. Par exemple, la 2.4.0 et la 3.4.1 ne sont pas dans la même série mineure, même si elles ont toutes les deux le 4 comme chiffre mineur, à l'opposé, la 2.4.0 et la 2.4.2 sont dans la même ligne mineure bien qu'elles ne soient pas adjacentes si la 2.4.1 a été publiée entre les deux.

La signification de ces nombres est très logique et sans surprise : une incrémentation du nombre principal indique que de grands changements se sont produits, une incrémentation du nombre mineur indique des modifications mineures et une incrémentation du micro-nombre indique des changements vraiment triviaux. Certains projets ajoutent une quatrième composante, appelée habituellement le numéro du correctif, pour un contrôle particulièrement fin des différences entre les versions (certains projets utilisent « correctif » comme synonyme de « micro » dans un système à trois composantes, ce qui peut prêter à confusion). Il y a aussi d'autres projets qui utilisent la dernière composante comme un numéro de « build ». Tous les rapports de bogue sont ainsi liés à un « build » particulier. Cette méthode est sûrement la mieux adaptée pour les logiciels distribués sous forme de paquets binaires.

Bien qu'il existe de nombreuses conventions différentes concernant le nombre de composantes que vous devriez utiliser et leur signification, les différences restent minimales, vous avez une certaine marge, mais pas tant que ça. Les deux prochaines parties portent sur les conventions les plus utilisées.

7.1.2 La stratégie simple

Chaque projet possède ses propres règles définissant les modifications autorisées dans une version si l'on incrémente seulement le micro-nombre, d'autres règles pour le nombre mineur et d'autres encore pour le nombre principal. S'il n'existe aucune norme encore, voici celle qui est appliquée avec succès par plusieurs projets. Même si votre projet n'adopte pas cette numérotation, vous trouverez dans les paragraphes suivants un bon exemple du type d'informations que doit véhiculer le numéro de version. Ces règles sont tirées du système de numérotation utilisé par le projet APR, voir <http://apr.apache.org/versioning.html>.

1. Les modifications du micro-nombre seul (c'est-à-dire les modifications au sein de la même ligne mineure) doivent être compatibles à la fois avec les versions suivantes et les versions précédentes. Les modifications ne devraient être, par conséquent, que des correctifs de bogue ou de petites améliorations de fonctionnalités déjà existantes. De nouvelles fonctionnalités ne devraient pas être introduites dans une micro-sortie.
2. Les modifications du nombre mineur (c'est-à-dire, dans la même ligne principale) doivent être rétro-compatibles, mais pas nécessairement avec les versions futures. Il est normal d'introduire de nouvelles fonctionnalités dans une sortie mineure, mais en nombre limité normalement.
3. Des modifications du nombre majeur marquent des limites de compatibilité. Une nouvelle sortie majeure peut être compatible avec les versions suivantes et précédentes. Une sortie majeure est censée posséder de nouvelles fonctionnalités, et peut même avoir tout un nouvel ensemble de fonctionnalités.

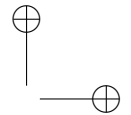
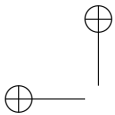
La signification exacte de la compatibilité avec les versions précédentes et suivantes dépend de ce que le logiciel fait, mais dans le contexte du logiciel la signification est relativement claire. Par exemple, si votre projet est une application client/serveur, alors « rétro-compatible » signifie que mettre à jour le serveur avec la version 2.6.0 ne devrait pas empêcher les clients utilisant la version 2.5.4 de fonctionner comme avant (hormis pour les bogues corrigés évidemment). D'un autre côté, mettre à jour l'un de ces clients avec la version 2.6.0, en même temps que le serveur, peut apporter de nouvelles fonctionnalités au client, des fonctionnalités que les clients avec la version 2.5.4 ne peuvent pas utiliser. Si cela se produit, alors la mise à jour n'est pas compatible : vous ne pouvez évidemment pas revenir en arrière et remettre la version 2.5.4 sur le poste client en gardant les fonctionnalités obtenues avec la 2.6.0 puisque certaines de ces fonctionnalités sont apparues avec la version 2.6.0.

C'est pourquoi les micro-sorties ne font quasiment que corriger des bogues. Elles doivent être compatibles dans les deux sens : si vous faites une mise à jour de la version 2.5.3 vers la version 2.5.4, et qu'ensuite vous changez d'avis pour revenir à la version 2.5.3, vous ne devriez pas perdre de fonctionnalités. Évidemment, les bogues corrigés dans la version 2.5.4 ré-apparaîtront après le retour en arrière, mais vous ne devez pas perdre de fonctionnalité, sauf si, éventuellement, un bogue corrigé par la nouvelle version empêche l'exécution de fonctionnalités existantes.

Les protocoles client/serveur ne représentent qu'un domaine de compatibilité parmi tant d'autres. On peut également citer le format des données : est-ce que le logiciel écrit des données sur une unité de stockage permanent ? Si c'est le cas, les formats de lecture et d'écriture doivent répondre aux mêmes normes imposées par les règles de numérotation. La version 2.6.0 doit pouvoir lire les fichiers écrits avec la version 2.5.4 mais peut très bien mettre à jour discrètement le format, le rendant non-interopérable avec la version 2.5.4. La compatibilité avec les versions antérieures n'est pas requise quand on change de numéro mineur. Si votre projet distribue des librairies de code que d'autres programmes emploient, alors les API sont aussi un autre domaine de compatibilité : vous devez vous assurer que les règles de compatibilité de la source et des binaires sont bien énoncées de manière à ce que les utilisateurs informés n'aient pas à se demander s'ils peuvent mettre à jour leur version sans risque. Ils pourront regarder la numérotation et seront instantanément fixés.

Ce système vous empêche de changer les choses profondément tant que vous n'incrémentez pas le nombre majeur. Cela peut souvent devenir un vrai handicap : vous avez peut-être envie d'ajouter des fonctionnalités, ou de repenser certains protocoles, mais vous ne pourrez pas le faire par souci de compatibilité. Il n'existe pas de solution miracle, à part peut-être si, dès le départ, vous décidez de tout construire de façon extensible (il faudrait un livre entier pour couvrir ce sujet, il dépasse largement les buts de cet ouvrage). Mais vous devez publier des règles de compatibilité et vous y tenir, c'est une obligation. Une mauvaise surprise pourrait vous mettre de nombreux utilisateurs à dos. Les règles décrites précédemment sont bonnes, en partie parce qu'elles sont répandues, mais aussi parce qu'elles sont simples à expliquer et à mémoriser, même pour ceux qui n'en sont pas familiers.

De manière générale, ces règles ne s'appliquent pas aux versions précédant la version 1.0 (bien que vos règles puissent l'établir explicitement, juste pour être clair). Un projet qui en est toujours à un stade de développement initial peut sortir les versions 0.1, 0.2, 0.3 et ainsi de suite l'une après l'autre jusqu'à ce qu'il soit prêt pour la version 1.0, et les différences entre ces versions peuvent être arbitrairement importantes. Les micro-nombres pour les sorties avant la 1.0 sont optionnels. Selon la nature de votre projet et les différences entre les sorties, vous jugerez utile ou non d'avoir des versions 0.1.0, 0.1.1, etc. Les conventions pour les sorties précédant la version 1.0 ne sont pas très strictes, les gens comprennent que de fortes contraintes de compatibilité entraveraient le dévelop-



pement initial et les premiers utilisateurs pardonnent facilement de toute manière.

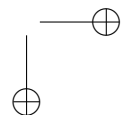
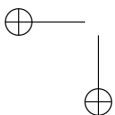
Souvenez-vous que toutes ces règles ne s'appliquent qu'à ce système, à trois composantes particulier. Votre projet pourrait facilement élaborer un système différent à trois composantes, ou même décider que vous n'avez pas besoin d'une telle finesse, et plutôt adopter un système à deux composantes. Il faut surtout que vous décidiez rapidement du système que vous allez adopter, que vous publiiez ce que signifient exactement les composantes et vous y tenir.

7.1.3 La stratégie pair/impair

Certains projets se servent de la parité du nombre mineur pour indiquer la stabilité du logiciel : pair signifie stable, impair signifie instable. Ceci ne s'applique qu'au nombre mineur, pas au nombre principal ou aux micro-nombres. L'incréméntation du micro-nombre indique toujours des corrections de bogues (pas de nouvelles fonctionnalités) et l'incréméntation du nombre principal indique toujours de grands changements, un ensemble de nouvelles fonctionnalités, etc.

L'avantage du système pair/impair, qui est employé par le projet du noyau Linux entre autres, est qu'il offre un moyen de sortir de nouvelles fonctionnalités, pour qu'elles soient testées, sans exposer les utilisateurs à un code potentiellement instable. Les gens comprennent grâce aux chiffres que la version 2.4.21 est suffisamment stable pour être installée sur leur serveur Web, mais que la version 2.5.1 devrait rester confinée aux expérimentations sur une machine de test. L'équipe de développement traite les rapports de bogue qui arrivent pour la série instable (dont le nombre mineur est impair), et quand les choses commencent à se calmer, après quelques micro-sorties dans cette série, ils incrémentent d'une unité le nombre mineur, le rendant donc pair, remettent le micro-nombre à 0 et sortent un paquet présumé stable.

Ce système préserve les indications de compatibilité données précédemment, ou du moins n'entre pas en conflit avec. Le nombre mineur véhicule simplement plus d'informations. Son incréméntation est deux fois plus fréquente que nécessaire, mais il n'y a rien de grave là-dedans. Le système pair/impair est sans doute mieux adapté aux projets qui ont de longs cycles de développement et qui, par nature, ont une grande proportion d'utilisateurs conservateurs qui préfèrent la stabilité à de nouvelles fonctionnalités. Cependant, ce n'est pas l'unique moyen pour que ces nouvelles fonctionnalités soient testées en « situation réelle ». La partie de ce chapitre intitulée « Stabiliser



une version » en décrit une autre, peut-être plus commune, pour proposer du code potentiellement instable au public, avec un avertissement dans le nom grâce auquel on peut évaluer plus simplement le rapport risque/bénéfice.

7.2 Les branches de publication

Du point de vue d'un développeur Open Source, le projet se trouve dans un état permanent de finalisation. Les développeurs utilisent généralement le code le plus à jour pour y repérer les bogues et, connaissant bien le projet, ils peuvent éviter les fonctionnalités encore instables. Il est courant pour eux de mettre très fréquemment à jour leur logiciel, et lorsqu'ils publient une modification du code, ils estiment raisonnable d'attendre des autres développeurs qu'ils l'aient intégrée à leur version du logiciel dans les 24 heures qui suivent.

Comment, alors, le projet doit-il s'y prendre pour proposer une version finalisée? Doit-il, par exemple, se contenter d'utiliser une image de l'arborescence à un moment précis, d'en faire un beau paquet binaire et de l'offrir au monde en annonçant « voici la version 3.5.0 »? Non évidemment! D'abord, l'arborescence de développement ne sera jamais complètement propre et publiable. La branche peut contenir diverses nouvelles fonctionnalités en chantier. Quelqu'un pourrait avoir publié un correctif important pour un bogue précis, mais qui est controversé et encore débattu au moment où vous figez le programme. Si tel est le cas, repousser la livraison, jusqu'à ce que le débat prenne fin, n'aiderait en rien. Un autre débat, pas forcément corrélé, pourrait naître entre-temps et vous devriez, dans ce cas, patienter jusqu'à sa fin, au risque d'attendre indéfiniment.

Dans tous les cas, figer l'arborescence entière entrerait forcément en conflit avec le développement en cours, même si l'arborescence courante était publiable. Si son image devient la version 3.5.0, l'image suivante, vraisemblablement la 3.5.1, contiendra principalement des correctifs des bogues introduits dans la version 3.5.0. Mais si les deux sont des images de la même arborescence supposée faite par les développeurs entre deux versions? Ils n'ont pas l'autorisation d'ajouter de nouvelles fonctionnalités, les directives de compatibilité les en empêchent. Or, tout le monde n'est pas motivé par l'écriture de correctifs pour les bogues de la version 3.5.0. Certains pourraient désirer achever de nouvelles fonctionnalités et seraient irrités que la seule possibilité qu'on leur laisse est de choisir

entre se tourner les pouces ou travailler sur ce qui ne les intéresse pas, simplement parce que le processus de publication du projet impose à l'arborescence de développement de rester dans un état de stabilité artificielle.

Les branches de publication sont la solution à ces problèmes. Ces branches sont utilisées au sein d'un logiciel de gestion de versions (voir *branche*) pour que le code destiné à être publié puisse être isolé du développement principal. Les logiciels libres ne sont pas les seuls à emprunter cette méthode, beaucoup d'équipes de développement de logiciels propriétaires font de même. Mais dans les environnement propriétaires, ces dernières sont parfois considérées comme un luxe, une « bonne pratique », dont on peut, compte tenu de contraintes de temps, se dispenser lorsque l'ensemble des équipes travaillent à la stabilisation de l'arborescence principale.

En revanche, les branches de publication sont incontournables dans le cadre d'un projet libre. J'ai vu des projets publier sans y avoir recours, mais le résultat est toujours le même : une partie des développeurs n'a rien à faire, alors que les autres, en général une minorité, s'échinent à sortir enfin la version. Le résultat est en général mauvais pour plusieurs raisons. D'abord, l'effort de développement général est ralenti. Ensuite, la qualité de la version s'en ressent puisque seules quelques personnes y travaillant sont pressées de terminer pour que tout le monde puisse se remettre au travail. Enfin, cette situation, où les différents types de travaux interfèrent, crée une fracture psychologique entre les équipes. Les développeurs qui se tournent les pouces aimeraient probablement contribuer aux branches de publication, lorsqu'ils en ont le temps et l'envie. Mais sans la branche, leur choix devient « Vais-je participer au code aujourd'hui ou non ? » plutôt que « Vais-je travailler aujourd'hui sur la version en instance ou sur la nouvelle fonctionnalité que j'ai amorcée dans le code principal ? »

7.2.1 Fonctionnement des branches de publication

Le processus exact de création d'une branche de publication dépend, bien entendu, de votre logiciel de gestion de versions, mais les concepts généraux sont les mêmes pour la plupart des systèmes. Une branche est en général issue d'une autre branche ou du tronc. Le tronc désigne généralement la partie où se fait le gros du développement, loin des contraintes liées aux sorties. La première branche de publication, celle qui mène à la sortie de la version « 1.0 », provient du tronc. Dans CVS, la commande de branche ressemble à ceci :

210 Paquets, sorties et développement quotidien

```
$ cd trunk-working-copy
$ cvs tag -b RELEASE_1_0_X
```

ou avec Subversion à cela :

```
$ svn copy http://.../repos/trunk http://.../repos/ →
↳ branches1.0.x
```

(Ces exemples impliquent un modèle de numérotation de version à 3 chiffres. L'espace faisant défaut ici pour fournir les commandes exactes de chaque logiciel de gestion de versions, j'utiliserai des exemples tirés de CVS et de Subversion en espérant que les commandes correspondantes dans les autres systèmes pourront aisément en être déduites.)

Notez que nous avons créé la branche 1.0.x (la lettre *x*) plutôt que 1.0.0. C'est parce que la même ligne mineure, c'est à dire la même branche, va être employée pour toutes les micro publications dans cette ligne. Le véritable processus de stabilisation d'une branche, en vue de sa sortie, est discuté dans la section appelée « Stabiliser une version » plus loin dans ce chapitre. Nous nous concentrons ici uniquement sur l'interaction entre le logiciel de gestion de versions et le processus de publication. Lorsque la branche est stabilisée et prête, c'est le moment d'en faire une image instantanée :

```
$ cd RELEASE_1_0-working-copy
$ cvs tag RELEASE_1_0_0
```

ou

```
$ svn copy http://.../repos/branches/1.0.x http://.../ →
↳ repos/tags/1.0.0
```

Cette étiquette représente l'état exact de l'arborescence source du projet pour la publication de la version 1.0.0 (c'est utile au cas où quelqu'un aurait besoin de restaurer une ancienne version, après que les paquets et binaires ne sont plus disponibles). La micro publication suivante, dans la même ligne, est préparée également sur la branche 1.0.x et lorsqu'elle est prête, une étiquette est créée pour la version 1.0.1. Répétez l'opération pour la version 1.0.2 et ainsi de suite... Lorsque le moment est venu de publier une version 1.1.x, créez une nouvelle branche partant du tronc :

```
$ cd trunk-working-copy
$ cvs tag -b RELEASE_1_1_X
```

ou

```
$ svn copy http://.../repos/trunk http://.../repos/ →  
↳ branches/1.1.x
```

La maintenance des versions 1.0.x et 1.1.x se fait en parallèle, et les sorties peuvent être indépendantes pour chaque ligne. Il est courant de publier de nouvelles versions presque simultanément pour deux lignes différentes. L'ancienne série est recommandée aux administrateurs plus conservateurs qui ne veulent pas sauter le pas vers la version 1.1 sans s'y être consciencieusement préparés. Mais les personnes plus aventureuses peuvent également disposer d'une version plus récente (possédant la numérotation la plus élevée), afin de profiter des dernières fonctionnalités, même au prix d'une plus grande instabilité.

Ce n'est naturellement pas la seule stratégie de publication de branches. Et elle n'est pas forcément la meilleure non plus selon les circonstances, bien qu'elle ait fait ses preuves dans bien des projets auxquels j'ai participé. Choisissez la stratégie qui vous convient le mieux, mais souvenez-vous des points essentiels : une branche de publication sert à isoler le travail à publier de l'agitation du développement quotidien et à structurer le processus de publication. Ce dernier est décrit plus en détails dans la prochaine partie.

7.3 Stabiliser une version

La stabilisation consiste à rendre publiable une branche, c'est à dire décider des changements qui seront inclus dans cette version et ainsi élaborer son contenu.

Le terme « décider » est polémique. La course à la fonctionnalité de dernière minute est un phénomène courant pour les projets de logiciels collaboratifs : aussitôt que les développeurs s'aperçoivent qu'une sortie est imminente, ils se bousculent pour terminer leur travaux en cours et ne pas rater le train. C'est évidemment exactement ce que vous voulez éviter au moment de sortir une nouvelle version. Il serait nettement préférable qu'ils continuent à travailler sur ces fonctionnalités à leur rythme, sans se préoccuper du fait que leurs modifications seront incluses dans la version en préparation ou dans la suivante. Plus vous essaieriez de surcharger une version à la dernière minute, plus le code sera instable et plus vous introduirez de bogues.

La plupart des ingénieurs logiciels s'accordent sur des critères de base permettant de décider quels types de changements devraient être autorisés lors de la période de stabilisation d'une version. Bien

sûr, les correctifs de bogues graves sont autorisés, en particulier ceux qui ne possèdent pas de contournements. Les mises à jour de la documentation sont acceptables, tout comme les corrections de messages d'erreur (sauf lorsqu'ils sont considérés comme faisant partie de l'interface et doivent rester stables). De nombreux projets autorisent également, durant la stabilisation, certains changements peu risqués ou sans impact sur le cœur du programme et peuvent avoir des critères stricts pour mesurer les risques. Mais quel que soit le niveau de formalisme, le jugement humain reste indispensable. Il reste toujours des cas où le projet doit prendre la décision d'accepter ou non un changement. Comme chacun souhaite voir ses modifications préférées être incorporées à la nouvelle version, nombreux sont ceux qui voudront autoriser les modifications, tandis que ceux voulant les bloquer le sont nettement moins, voilà le risque.

Par conséquent, le moteur du processus de stabilisation d'une version est le « non ». Le plus délicat, en particulier pour les projets libres, est de trouver comment refuser tout en évitant de blesser ou de décevoir les développeurs et sans pour autant poser d'entraves aux modifications méritant d'être adoptées. Différents moyens le permettent, et il est assez simple de créer des systèmes satisfaisant à ces critères une fois que l'équipe les a identifiés. Je ferai ici une brève description de deux systèmes parmi les plus populaires malgré leurs différences, mais n'hésitez pas à vous montrer plus créatif : de nombreuses autres possibilités existent, il s'agit de deux exemples simples dont j'ai pu constater l'efficacité en pratique.

7.3.1 La dictature du propriétaire de version

Le groupe désigne un propriétaire de version. Cette personne décide finalement des changements qui seront inclus ou non dans la version. Bien sûr, il est normal, et attendu, qu'il y ait des discussions et des débats, mais le groupe doit accorder une autorité suffisante au propriétaire de version pour qu'il prenne les décisions finales. Pour que ce système fonctionne, la personne choisie doit disposer des compétences techniques requises pour comprendre toutes les modifications, mais aussi avoir le charisme et les compétences nécessaires pour mener la discussion et parvenir à la publication sans froisser excessivement les susceptibilités.

Le propriétaire de version est souvent amené à dire par exemple : « Je ne pense pas qu'il y ait de problème avec cette modification, mais nous n'avons pas assez de temps pour la tester maintenant, on ne devrait donc pas l'inclure dans cette version ». S'il possède une connaissance technique générale du projet, il serait judicieux de sa

part d'expliquer en quoi le changement pourrait impacter la version (par exemple, par ses interactions avec d'autres parties du logiciel, ou pour une question de portabilité). Certains demanderont parfois des justifications, ou affirmeront que leurs modifications ne sont pas si risquées. Ces conversations ne seront pas conflictuelles tant que le propriétaire de version sera en mesure de peser tous les arguments, objectivement et sans camper systématiquement sur ses positions.

Notez que le propriétaire de version n'est pas nécessairement le chef du projet (si tant est qu'il y en ait un, voir la section « Les dictateurs bienveillants » du chapitre 4). Il est souvent préférable que ce ne soit pas une seule et même personne qui endosse les deux rôles. Les aptitudes requises sont assurément différentes. Pour une chose aussi importante que le processus de publication, il est sage d'apporter un contrepois au jugement du chef de projet.

Comparez le rôle du propriétaire de version à celui, moins dictatorial, décrit dans la section « Contrôleur de version » plus loin dans ce chapitre.

7.3.2 Le choix des évolutions par vote

À l'extrême opposé du concept de dictature du propriétaire de version, les développeurs peuvent simplement voter pour décider des évolutions à inclure dans la version. Cependant, la stabilisation n'étant pas une opération portes ouvertes, il est important de concevoir le système de vote en gardant à l'esprit que l'ajout de modifications doit exiger l'implication de plusieurs développeurs. Une simple majorité devrait être une condition nécessaire, mais non suffisante, à un ajout (voir la section « Qui vote ? » dans le chapitre 4). Dans le cas contraire, un vote « pour » et aucun vote « contre » une modification suffirait à la valider et une dynamique indésirable se mettrait en place : chaque développeur voterait pour son propre changement et serait peu enclin à voter contre les modifications des autres par peur de représailles. Pour éviter cela, le système devrait favoriser la coopération entre sous-groupes de développeurs militant pour un changement. Non seulement plus de gens vérifient alors les modifications, mais en plus, les développeurs, en tant qu'individus, sont davantage enclins à voter contre. Ils savent, en effet, que personne, en particulier parmi ceux qui ont voté « pour », ne prendra leur vote « contre » comme un affront personnel. Plus il y a de personnes impliquées, plus la discussion se focalise sur les changements et non plus sur les individus.

Le système que nous employons dans le projet Subversion semble avoir atteint un bon équilibre, je le recommande donc. Pour qu'une

modification soit appliquée à une branche de publication, au moins trois développeurs doivent voter en sa faveur et aucun contre. Un seul vote « contre » est suffisant pour que le changement ne soit pas ajouté, c'est à dire qu'un vote « contre » équivaut à un veto (voir la section intitulée « Vetos »). Naturellement, un tel vote doit être justifié et, en théorie, le veto peut être outrepassé si suffisamment de personnes pensent qu'il est excessif et forcent un vote spécial pour le contourner. En pratique, ceci ne s'est jamais produit, et je ne pense pas que ce soit le cas dans l'avenir. Les participants deviennent naturellement plus conservateurs quand on s'approche d'une publication et lorsque quelqu'un est suffisamment déterminé pour opposer son veto à une modification, ses raisons sont généralement bonnes.

La procédure de publication étant délibérément biaisée en faveur du conservatisme, les justifications données pour les vetos sont parfois plus procédurières que techniques. Par exemple, une personne peut être convaincue qu'une modification est correcte et ne causera probablement aucun nouveau bogue et voter pourtant contre son inclusion dans une version de mise à jour intermédiaire, simplement parce que cette modification est trop lourde, ou qu'elle apporte peut-être de nouvelles fonctionnalités ou risque de ne pas remplir totalement les règles de compatibilité ascendante. J'ai même déjà vu, à une occasion, des développeurs opposer leur veto simplement parce qu'ils avaient le pressentiment que la modification nécessitait des tests plus poussés, sans toutefois détecter le moindre bogue lors des relectures. Malgré les récriminations, le veto fut respecté et la modification n'a pas été incluse dans cette version (d'ailleurs, je ne me souviens pas si des bogues ont finalement été découverts ou non).

7.3.2.1 Gérer une stabilisation collégiale

Si votre projet opte pour un système de vote, vous devez rendre aussi accessibles que possible les outils du vote ainsi que le vote en lui-même. Bien qu'il existe de nombreux logiciels Open Source créés pour faciliter le vote électronique, un simple fichier texte (que l'on nommera par exemple « STATUTS » ou « VOTES ») dans la branche de publication se révélera plus pratique. Le fichier, éditable par tous les développeurs, liste chaque évolution proposée ainsi que les votes pour ou contre. Les votes sont fréquemment accompagnés de notes et commentaires. Toutefois, proposer une modification ne signifie pas nécessairement voter pour, bien que les deux aillent souvent de pair. Voici un exemple de ce à quoi peut ressembler ce type de fichier :

```
* r2401 (problème #49)
Éviter la redondance de la connexion client/serveur.
```

```
Justification:
Supprime des accès réseaux inutiles; changement
minime et facile à vérifier.
Notes:
Sujet discuté sur http://.../mailing-lists/message
-7777.html et autres messages dans ce sujet.
Votes:
+1: jsmith, kimf
-1: tmartin (viole la compatibilité avec certains
serveurs pre-1.0; il est vrai que ces serveurs sont
bogués, mais il faut préserver la compatibilité
autant que faire se peut)
```

Ici, la modification a reçu deux votes pour mais tmartin a opposé son veto, et l'a justifié dans une note entre parenthèses. La forme exacte de cette entrée importe peu : l'explication de tmartin pourrait se trouver dans la section « Notes » ou dans la description de la modification marquée par un titre « Description » pour correspondre aux autres sections... Il faut principalement vous assurer que toutes les informations nécessaires à l'évaluation soient disponibles, et que le système de vote soit aussi simple que possible. La modification soumise à débat est désignée par son numéro de révision dans le dépôt de sources (dans ce cas, il contient une seule révision, la r2401, mais une modification peut en référencer plusieurs). La révision pointe implicitement vers un changement du tronc puisqu'une modification issue de la branche de publication n'aurait pas eu à être soumise au vote. Si votre logiciel de gestion de versions ne dispose pas d'une syntaxe explicite pour se référer à des modifications individuelles, alors, le projet devrait en prévoir une. Pour que le vote soit fonctionnel, chaque modification doit être identifiable sans ambiguïté.

Ceux qui ont proposé ou voté pour une modification doivent s'assurer que cette dernière s'applique correctement et sans conflit à la branche de publication (voir Conflits). Si des conflits subsistent, la modification doit pointer vers un correctif ou vers une branche temporaire proposant une version ajustée de la modification, par exemple :

```
*r13222, r13223, r13232
Réécrire l'algorithme de fusion automatique
libsvn_fs_fs
Justification:
Performances inacceptables (>50 minutes pour
un petit ajout) dans un dépôt contenant 300 000
révisions
Branche:
1.1.x-r13222@13517
Votes:
+1: epg, ghudson
```

Cet exemple est authentique, il provient du fichier `STATUTS` utilisé lors de la publication de Subversion 1.1.4. Notez qu'il utilise les révisions originales comme identifiants uniques des modifications, malgré l'utilisation d'une branche contenant une version adaptée de la modification (la branche combine les trois révisions du tronc en une seule, la r13517, afin de faciliter sa fusion si elle est acceptée). Les révisions originales sont présentes pour faciliter leur consultation car les commentaires originaux y sont liés. La branche temporaire ne doit pas reprendre ces derniers pour éviter de dupliquer l'information (voir la section « Unicité de l'information » dans le Chapitre 3, Infrastructure Technique), le commentaire de la révision r13517 devrait simplement contenir « Ajuster r13222, r13223 et r12232 pour portage à la branche 1.1.x. ». Toute autre information peut être retrouvée dans les révisions originales.

7.3.2.2 Le responsable de version

La fusion (voir fusion ou portage) des modifications approuvées vers la branche de publication peut être effectuée par les différents développeurs. Un poste spécifique, dédié aux fusions, n'est pas indispensable. Si la charge de travail devient trop conséquente, il vaut mieux la répartir entre plusieurs développeurs.

Malgré le caractère décentralisé des votes et des fusions, une ou deux personnes guident le processus de publication. On utilise parfois le terme formel de « responsable de version » pour ce rôle sensiblement différent du « propriétaire de version » (voir la section « La dictature du propriétaire de version » précédemment dans ce chapitre) lequel garde le dernier mot sur les changements à appliquer. Les responsables de version effectuent le suivi des modifications actuellement en cours de traitement : combien ont été approuvées, combien sont en voie d'obtenir l'approbation, etc. C'est leur rôle aussi de suggérer subtilement aux développeurs d'examiner certains changements ne bénéficiant pas de l'attention qu'ils méritent. Lorsqu'un lot de modifications est approuvé, ils se chargent (en général de leur propre initiative) de les insérer dans la branche de publication. Les autres développeurs doivent cependant comprendre que tout ce travail n'incombe pas aux seuls responsables de version, sauf prérogatives explicites. Lors de la publication de la version (voir la section « Tests et sortie » plus loin dans ce chapitre), les responsables de version sont également responsables de la création des paquets définitifs, de la collecte des signatures numériques, de la mise à disposition des paquets et de l'annonce publique.

7.4 La création de paquets

Les logiciels libres sont traditionnellement distribués sous la forme de code source, interprété (Perl, Python, PHP, etc.) ou nécessitant une compilation préalable (C, C++, Java, etc.). Les logiciels compilés ne seront généralement pas construits par les utilisateurs eux-mêmes, mais installés sous la forme d'un paquet binaire prêt à l'exécution (voir la section appelée « Paquets binaires » plus loin dans ce chapitre). Ces paquets sont néanmoins toujours issus d'une source de distribution de référence. Le paquet source doit porter un identifiant de version le rendant parfaitement identifiable. Un paquet nommé « Scanley 2.5.0 » désigne en fait « l'arborescence de fichiers sources constituant Scanley 2.5.0 une fois compilée (si nécessaire) et installée ».

Cette arborescence répond à des normes plutôt strictes qui sont, à quelques rares exceptions près, toujours suivies. Sauf cas de force majeure, votre projet devrait lui aussi suivre ces normes.

7.4.1 Le format

Le code source devrait être mis à disposition dans le format standard du transport d'arborescences de répertoires. Pour les systèmes d'exploitation Unix ou dérivés, la convention est le format TAR, compressé par compress, gzip, bzip ou bzip2. Pour MS Windows, c'est le format zip. Ce dernier assure également la compression, ce qui élimine le besoin de compresser l'archive après sa création.

Les fichiers TAR

TAR signifie « Tape ARchive », car le format tar représente une arborescence de répertoires comme un flux linéaire de données, ce qui le rend idéal pour sauvegarder des répertoires sur cassette. Cette même propriété en fait le standard idéal pour distribuer les arborescences sous la forme d'un fichier unique. La création de fichiers tar compressés (ou tarballs) est assez facile. Sur certains systèmes, la commande tar peut produire elle-même des archives compressées alors que sur d'autres, un programme de compression distinct est nécessaire.

7.4.2 Nom et structure

Le nom du paquet est composé du nom du logiciel suivi du numéro de version et du suffixe du format correspondant au type d'archive. Par exemple, le nom pour Scanley 2.5.0, pour Unix, compressé avec GNU Zip (gzip) ressemble à ceci :

```
scanley-2.5.0.tar.gz
```

ou pour Windows, en utilisant une compression de type zip :

```
scanley-2.5.0.zip
```

Ces deux archives, une fois décompressées, créent un nouveau dossier nommé `scanley-2.5.0` dans le dossier courant. Dans ce nouveau dossier, le code source est prêt à être compilé (si une compilation est requise) et à être installé. À la racine du nouveau dossier se trouve un fichier texte nommé `README` (ou `LISEZ-MOI`) expliquant le but du logiciel, sa version ainsi que d'autres ressources comme le site Web du projet, d'autres fichiers pertinents, etc. Parmi ces autres fichiers, on trouve un fichier `INSTALL`, semblable au fichier `README`, qui fournit les instructions permettant de construire et d'installer le logiciel pour tous les systèmes d'exploitation pris en charge. Comme nous l'avons déjà évoqué dans la section « Comment appliquer une licence à votre logiciel » du chapitre 2, on trouve également un fichier `COPYING` ou `LICENCE`, précisant les conditions de distribution du logiciel.

On doit également y trouver un fichier `CHANGES` ou `MODIFICATIONS` (parfois appelé `NEWS` ou `NOUVEAUTÉS`) reprenant les améliorations apportées par cette version. Le fichier `CHANGES` liste les modifications portant sur toutes les versions, dans l'ordre anti-chronologique, afin que les modifications de la version courante apparaissent en premier. Compléter cette liste est, en général, la dernière chose devant être effectuée sur une branche de publication en cours de stabilisation. Certains projets rédigent cette liste au fur et à mesure du développement, d'autres préfèrent conserver cette étape pour la fin et confier sa rédaction à une personne. Les modifications sont tirées des journaux de gestion de versions et ressemblent à cet extrait :

```
Version 2.5.0 (20 décembre 2004) depuis
/branches/2.5.x)
http://svn.scanley.org/repos/svn/tags/2.5.0/
Nouvelles fonctionnalités, améliorations:
* Ajout des requêtes d'expressions régulières
* Ajout du support UTF-8 et UTF-16 pour les documents
* Documentation traduite en polonais, russe
* et malgache
* ...
```

```
Correctifs:  
* Correction du bogue de ré-indexation  
(bogue #945)  
* Correction de quelques bogues de requêtes  
(bogues #815, #1007, #1008)
```

La liste peut être aussi longue que nécessaire, mais il n'est pas indispensable d'y inclure chaque petite correction ou amélioration. Elle sert à fournir aux utilisateurs une vision générale de ce qu'ils gagneront à utiliser la nouvelle version. En fait, la liste des modifications apparaîtra en général dans l'e-mail d'annonce (voir la section appelée « Tests et sortie » plus loin dans ce chapitre), rédigez-la donc en gardant en tête sa future audience.

CHANGES et ChangeLog

Traditionnellement, un fichier nommé ChangeLog liste l'intégralité des changements apportés à un projet, c'est -à-dire chaque révision du logiciel de gestion de versions. Il existe différents formats pour les fichiers ChangeLog, mais les détails du format ne sont pas importants ici puisqu'ils contiennent tous les mêmes informations : la date de la modification, son auteur ainsi qu'un bref résumé (ou simplement le message de journal pour ce changement).

Un fichier CHANGES est différent. Il ne reprend que les modifications jugées importantes pour un certain public : certaines méta-données comme la date exacte ou l'auteur ont été supprimées. Pour éviter les confusions, n'intervertissez pas ces deux termes. Certains projets utilisent « NEWS » plutôt que « CHANGES ». Même si la confusion potentielle avec « ChangeLog » est ainsi évitée, le terme est mal choisi puisque le fichier CHANGES répertorie les modifications portant sur toutes les versions antérieures, et comprend donc beaucoup d'anciennes « nouveautés » en plus de celles figurant en haut du fichier.

Les fichiers ChangeLog semblent, de toute façon, en voie de disparition. Ils étaient utiles à l'époque où CVS était le logiciel de gestion de versions incontournable et dont il n'était pas simple d'extraire ces données. Les logiciels de gestion de versions plus modernes peuvent facilement restituer par requêtes les données qui étaient jadis inscrites dans le ChangeLog, ce qui rend inutile la maintenance d'un fichier statique contenant ces données, puisque le ChangeLog se résume alors à une réplique du journal de messages déjà enregistré dans le dépôt.

La structure du code source, dans l'arborescence, doit être aussi proche que possible de celle du projet telle qu'elle apparaît dans le dépôt de gestion de versions. Quelques différences peuvent exister, par exemple, le paquet peut contenir certains fichiers générés nécessaires à la configuration et à la compilation (voir la section nommée « Compilation et installation » plus loin dans ce chapitre), ou il peut contenir un programme tiers requis, mais qui n'est pas maintenu par le projet, et que les utilisateurs ne possèdent probablement pas encore. Mais, même si l'arborescence distribuée correspond exactement à une arborescence de développement dans le dépôt de gestion de versions, la distribution elle-même ne doit pas être une copie de travail (voir « Copie de travail » dans la section « Gestion de versions » du chapitre 3) et doit représenter un point de référence immuable, une configuration particulière et non-modifiable de fichiers sources. Une copie de travail pourrait être modifiée par l'utilisateur, lequel penserait alors disposer de la version officielle alors qu'il ne s'agirait que d'une version modifiée.

Souvenez-vous que le contenu est toujours le même, indépendamment du paquet. La version, c'est à dire l'entité précise à laquelle on fait référence quand on parle de « Scanley 2.5.0 », est l'arborescence créée lorsqu'on extrait les fichiers d'une archive zip ou tarball. Le projet devrait donc proposer tous ces fichiers au téléchargement :

```
scanley-2.5.0.tar.bz2
scanley-2.5.0.tar.gz
scanley-2.5.0.zip
```

... mais l'arborescence issue de leur extraction doit être rigoureusement la même. Cette arborescence source est la distribution ; la forme sous laquelle elle est téléchargée est une simple question de commodité. Certaines différences triviales entre les paquets sources sont possibles : par exemple, dans les fichiers textes de paquets Windows, les lignes devraient se terminer par CRLF (Carriage Return and Line Feed¹ tandis que les paquets Unix ne devraient utiliser que LF. Les arborescences peuvent être légèrement arrangées en fonction du système d'exploitation cible si ce dernier demande une structure spécifique pour la compilation. Il s'agit néanmoins de transformations triviales. Les fichiers sources de base devraient être identiques pour tous les paquets d'une version donnée.

7.4.2.1 De l'utilisation des majuscules

Les gens utilisent naturellement des majuscules pour les noms de projets, alors traités comme des noms propres. Il en est de même

1. NdT : retour du chariot et passage à la ligne.

pour les acronymes : « MySQL 5.0 », « Scanley 2.5.0 », etc. C'est au projet de décider si le nom du paquet doit utiliser la même casse. `Scanley-2.5.0.tar.gz` ou `scanley-2.5.0.tar.gz` sont tous les deux valables par exemple (je préfère la dernière proposition pour ne pas forcer les gens à appuyer sur MAJ, mais de nombreux projets utilisent une majuscule dans le nom du paquet). Il faut néanmoins que le répertoire créé en extrayant le tarball utilise la même convention. Évitez les surprises, l'utilisateur devrait pouvoir prédire sans ambiguïté le nom exact du répertoire créé quand il extraira la distribution.

7.4.2.2 Pré-version

Quand vous rendez publique une pré-version ou une version candidate, le qualificatif fait concrètement partie du numéro de version, par conséquent ajoutez-le au nom du paquet. Par exemple, les phases successives alpha et bêta déjà abordées dans la section « Les composants d'un numéro de version » correspondraient aux noms de paquets suivants :

```
scanley-2.3.0-alpha1.tar.gz
scanley-2.3.0-alpha2.tar.gz
scanley-2.3.0-beta1.tar.gz
scanley-2.3.0-beta2.tar.gz
scanley-2.3.0-beta3.tar.gz
scanley-2.3.0.tar.gz
```

Le premier paquet est extrait dans un répertoire nommé `scanley-2.3.0-alpha1`, le second dans un répertoire `scanley-2.3.0-alpha2` et ainsi de suite.

7.4.3 Compilation et installation

Lorsqu'une compilation ou une installation depuis la source sont nécessaires, il existe en général des procédures standardisées que les utilisateurs expérimentés connaissent. Par exemple, pour les programmes écrits en C, C++ ou certains autres langages compilés, la méthode standard sous Unix et dérivés est la suivante :

```
$ ./configure
$ make
# make install
```

La première de ces commandes détecte automatiquement autant d'informations que possible sur l'environnement et prépare l'étape de construction, la seconde commande construit le logiciel (mais

ne l'installe pas) et la dernière commande l'installe sur le système. Les deux premières sont effectuées en tant qu'utilisateur simple, la troisième en tant qu'administrateur. Pour plus de détails à propos de la mise en place de ce système, je vous renvoie à l'excellent livre *GNU Autoconf, Automake et Libtool* par Vaughan, Elliston, Tromeey et Taylor. Il est publié en tant que « treeware » par New Riders et son contenu est accessible librement en ligne¹.

Ce n'est pas le seul standard qui existe, mais c'est le plus répandu. Le système de construction ANT² gagne en popularité, particulièrement auprès des projets écrits en Java, et possède ses propres standards de procédures pour la construction et l'installation. Certains langages de programmation, tels que Perl et Python, recommandent également une méthode standard pour les programmes écrits dans ces langages (par exemple, les modules Perl emploient la commande `perl Makefile.pl`). Si les standards applicables à votre projet ne vous paraissent pas évidents, demandez à un programmeur expérimenté ; il existe très probablement un standard que vous ne connaissez pas encore.

Choisissez le standard le plus adapté à votre projet et ne changez qu'en cas de nécessité absolue. Les procédures d'installation standard sont devenues des réflexes conditionnés pour beaucoup d'administrateurs système. Des commandes familières, dans le fichier `INSTALL`, les rassureront sur l'importance que vous accordez aux standards, et ils auront une meilleure approche *a priori* de votre logiciel. De même, comme nous l'avons vu dans la section « Téléchargements » du chapitre 2, utiliser une séquence de construction standard plaît aux développeurs potentiels.

Sous Windows, les standards de construction et d'installation ne sont pas aussi bien définis. Pour les projets nécessitant une compilation, il semblerait que l'habitude soit de créer une arborescence s'intégrant à l'espace de travail des environnements de développement standards de Microsoft (Developer Studio, Visual Studio, VS.NET, MSVC++, etc.). Selon la nature de votre logiciel, il vous sera possible de proposer une option de construction semblable à celle d'Unix sur Windows grâce à l'environnement Cygwin³. Et bien sûr, si vous utilisez un langage ou framework qui possède ses propres conventions comme Perl ou Python, vous devriez simplement employer la méthode standard adéquate indépendamment du système d'exploitation Windows, Unix, Mac OS ou autres.

1. <http://sources.redhat.com/autobook/>
 2. <http://ant.apache.org/>
 3. <http://www.cygwin.com/>

Soyez prêt à faire de nombreux efforts supplémentaires pour rendre votre projet conforme aux standards de construction et d'installation. Ce sont des étapes clés : les choses peuvent devenir plus complexes par la suite, si vraiment c'est nécessaire, mais il serait fortement dommageable pour l'image de votre projet que les utilisateurs aient à fournir plus d'efforts que prévus dès ce premier contact avec le logiciel.

7.4.4 Paquets binaires

Les paquets embarquant le code source constituent le format canonique d'une version, mais la plupart des utilisateurs installeront un paquet binaire, fourni par le système de distribution de logiciels de leur système d'exploitation ou téléchargé manuellement depuis le site Web du projet ou d'un tiers. « Binaire » ne signifie pas forcément « compilé », cela veut simplement dire que le paquet est pré-configuré de sorte que l'utilisateur puisse l'installer sur son ordinateur sans avoir à passer par les procédures habituelles de construction/installation du fichier source. On trouve sous Redhat GNU/Linux le système RPM, sous Debian GNU/Linux le système APT (.deb), sur MS Windows en général des fichiers .MSI ou des fichiers d'installation automatique .exe.

Qu'ils viennent d'un proche du projet ou d'une personne qui y est complètement étrangère, ces paquets sont officiels et les problèmes qu'ils rencontrent seront ajoutés au système de suivi de bogues. Le projet a donc tout intérêt à travailler en étroite collaboration avec ceux qui préparent les paquets et à leur fournir des directives claires afin que les paquets soient fidèles au logiciel.

Les assembleurs doivent impérativement baser leurs paquets binaires sur une version officielle du code source. Ils seront parfois tentés d'extraire une version plus évoluée du dépôt ou d'y inclure des modifications plus récentes pour apporter aux utilisateurs certaines corrections de bogues ou d'autres améliorations. En agissant ainsi, ils pensent rendre service en proposant le code le plus récent, mais gare à la confusion. Les projets savent traiter les rapports concernant les bogues rencontrés dans les versions publiées, dans les ajouts récents au tronc ou dans les branches principales du code (des bogues trouvés par ceux qui font tourner délibérément le code le plus récent). Quand un rapport de bogue est enregistré pour une version ou une branche précise, la personne traitant le rapport sera dans la plupart des cas capable de confirmer la présence du bogue et, s'il a été corrigé, il pourra alors conseiller à l'utilisateur de mettre son logiciel à jour ou d'attendre la prochaine version. Et enfin, si le bogue

n'est pas encore répertorié, son classement nécessite de connaître la version affectée.

Mais les projets ne sont pas préparés à recevoir des rapports de bogues basés sur des versions mal définies ou hybrides. Ces bogues sont très difficiles à reproduire et peuvent être le résultat d'interactions imprévisibles entre des changements isolés extraits d'un développement plus récent. Ils peuvent donc causer des comportements inattendus sans que les développeurs du projet en soient responsables. Ces rapports particuliers peuvent faire perdre un temps considérable à votre projet, j'en ai moi-même été témoin : quelqu'un rencontrait un bogue avec une version corrigée du code source officiel. Personne dans l'équipe de développement ne parvenait à reproduire le bogue et chacun a dû chercher en vain l'origine de ce comportement inexplicable.

Certains assembleurs insisteront toujours pour inclure certaines modifications du code en développement. Il faut les encourager à avertir les développeurs du projet de leurs intentions. Même si les développeurs ne le voient pas d'un bon œil, ils seront au moins prévenus en cas de rapport de bogue inattendu. Ils pourront ainsi publier une mise en garde sur le site du projet et demander au créateur du paquet d'en faire de même à un endroit bien en vue afin que les personnes utilisant ce paquet binaire sachent que ce qu'ils ont entre les mains n'est pas exactement identique à ce que le projet a officiellement publié. Même si c'est difficile, il faut éviter d'envenimer la situation. Les assembleurs ne partagent simplement pas la vision des développeurs. Ils cherchent principalement à apporter aux utilisateurs le meilleur du logiciel. C'est aussi le but des développeurs bien entendu, mais ces derniers doivent également maîtriser la version exacte des paquets utilisés pour rendre les rapports de bogues cohérents et pour assurer la compatibilité de leur travail. Ces objectifs entrent parfois en conflit. Il faut alors se souvenir que le projet n'a aucun contrôle sur le travail des assembleurs et que chacun a des obligations envers l'autre. En effet, le projet produit généreusement le code mais les assembleurs, de leur côté, effectuent un travail souvent ingrat pour rendre le projet largement accessible et l'ouvrir à une communauté que le projet seul n'aurait peut être pas atteinte. Vous pouvez vous trouver en désaccord, mais restez courtois pour que tout se passe pour le mieux.

7.5 Tests et sorties

Une fois le tarball des sources créé à partir d'une branche de publication stabilisée, la partie visible de la publication commence.

Mais avant que le tarball soit mis à la disposition du monde entier, il doit être testé et approuvé par un minimum de développeurs, en général trois ou plus. Cette validation ne sert pas seulement à chercher des bogues évidents : idéalement les développeurs téléchargent l'archive, la construisent et l'installent sur un système propre, puis effectuent l'ensemble des tests de non-régression (voir la section appelée « Tests automatisés » dans le chapitre 8) et quelques tests manuels. En supposant que l'archive passe ces épreuves, ainsi que toute autre série de tests que le projet pourrait avoir fixée, les développeurs signent numériquement l'archive en utilisant GnuPG¹, PGP², ou tout autre programme capable de produire une signature compatible PGP.

Dans la plupart des projets, les développeurs utilisent leurs propres signatures numériques plutôt qu'une clé partagée par le projet. Tous les développeurs qui le souhaitent peuvent apposer leur signature (il y en a donc un nombre minimum, mais pas de maximum). Un grand nombre de signatures prouve que le programme a fait l'objet de nombreux tests, ce qui augmente la confiance des utilisateurs avertis.

Une fois approuvés, tous les fichiers (c'est à dire, les tarballs, les fichiers zip et autres formats de paquet) devraient être placés dans la section téléchargement du projet, accompagnés d'une signature numérique et d'une empreinte MD5/SHA1 (voir l'article Wikipédia sur la fonction de hachage³). Plusieurs standards existent. Une solution consiste à mettre à disposition, avec le paquet, le fichier de signature numérique correspondante et le fichier d'empreinte. Par exemple, pour le paquet `scanley-2.5.0.tar.gz`, placez dans le même répertoire, un fichier `scanley-2.5.0.tar.gz.asc` contenant la signature numérique de l'archive, un fichier `scanley-2.5.0.tar.gz.md5` contenant l'empreinte MD5 et, à votre convenance, un fichier `scanley-2.5.0.tar.gz.sha1` contenant l'empreinte SHA1. Il est également possible de rassembler toutes les signatures pour tous les paquets proposés dans un fichier unique, `scanley-2.5.0.sigs`, et de faire de même avec les empreintes.

Peu importe la solution que vous retiendrez tant que le système est simple, documenté et cohérent entre les publications. Le but de toutes ces signatures et empreintes est de permettre aux utilisateurs de vérifier que la copie reçue n'a pas été falsifiée à des fins malveillantes. Les utilisateurs s'appêtent à faire tourner ce code sur leur ordinateur, or, s'il a été altéré, une personne mal intentionnée

1. <http://www.gnupg.org/>

2. <http://www.pgpi.org/>

3. http://fr.wikipedia.org/wiki/Fonction_de_hachage

peut ainsi créer une porte dérobée pour accéder à toutes leurs données. Dans la section « Les mises à jour de sécurité », plus loin dans ce chapitre, nous reviendrons sur la paranoïa.

7.5.1 Versions candidates

Pour les versions introduisant beaucoup de changements, de nombreux projets publient une version candidate (Release Candidate ou RC), par exemple `scanley-2.5.0-beta1` avant `scanley-2.5.0`. Le but d'une version candidate est de soumettre le code à un maximum de testeurs avant de le labelliser « version officielle ». Si des problèmes apparaissent, ils sont corrigés dans la branche de publication, et une nouvelle version candidate est proposée (`scanley-2.5.0-beta2`). Le cycle se poursuit jusqu'à ce qu'il n'y ait plus de bogues rédhibitoires. La version candidate devient alors la version officielle. Tout ce qui différencie la dernière version candidate de la version officielle est donc le qualificatif qui est retiré du numéro de version.

Pour le reste, une version candidate doit être traitée comme une version finale. Les qualificatifs *alpha*, *beta* ou *rc* suffisent à avertir les utilisateurs conservateurs de patienter jusqu'à la version finale et les e-mails d'annonce de la version candidate doivent mettre en avant qu'elle a pour but d'obtenir des retours de la part des utilisateurs. Sinon, accordez aux versions candidates la même attention qu'aux versions officielles car leur mise à disposition est la meilleure façon de trouver de nouveaux bogues, et parce que vous ne savez jamais, après tout, quelle version candidate deviendra la version officielle.

7.5.2 Annoncer une publication

Annoncer une publication, c'est comme annoncer n'importe quel autre évènement et les procédures décrites dans la section « Les annonces » du chapitre 6 s'appliquent. Il y a cependant quelques points particuliers à prendre en compte dans cette situation.

À chaque fois que vous fournissez l'URL d'un tarball, assurez-vous de fournir également les empreintes MD5/SHA1 et des liens vers les fichiers contenant les signatures numériques. Puisque l'annonce se fera sur plusieurs médias (listes de diffusion, pages d'informations, etc.), les utilisateurs peuvent obtenir les empreintes depuis différentes sources, ce qui donne aux utilisateurs les plus exigeants sur la sécurité une assurance supplémentaire que ces dernières n'ont pas été altérées. Proposer différents liens vers les fichiers de signatures

numériques ne rend pas ces signatures plus sûres, mais démontre aux gens (en particulier ceux qui ne suivent pas le projet de près) que la sécurité est prise au sérieux.

Dans l'e-mail d'annonce et dans les pages d'information contenant plus qu'un simple argumentaire « commercial », assurez-vous d'inclure les passages pertinents du fichier `CHANGES` pour informer les utilisateurs des avancées apportées par cette mise à jour. Cela vaut autant pour les versions candidates que pour les versions finales, la présence de correctifs de bogues et de nouvelles fonctionnalités est importante pour pousser les gens à tester les versions candidates.

Pour finir, n'oubliez pas de remercier l'équipe de développement, les testeurs et toutes les personnes qui ont pris le temps de faire de bons rapports de bogues. Ne citez personne précisément, sauf si quelqu'un de particulier est à l'origine d'un travail singulièrement important, un travail que tout le monde a pu apprécier au sein du projet. Ne vous laissez pas entraîner sur la pente glissante de la surenchère de remerciements (voir la section « Remerciements » dans le chapitre 8).

7.6 Maintenir plusieurs versions

Les projets les plus matures maintiennent parallèlement plusieurs versions. Une fois publiée, la version 1.0.0 peut continuer à vivre via des micro versions (correctifs) 1.0.1, 1.0.2, etc. jusqu'à ce que le projet décide explicitement de mettre un terme à son support. Le simple fait de publier la version 1.1.0 n'est pas une raison suffisante pour clore la série des versions 1.0.x. Certains utilisateurs se fixent comme règle de ne jamais utiliser une première version mineure ou majeure, laissant aux autres le soin de dénicher les bogues de la version 1.1.0 et attendent la version 1.1.1. N'y voyez pas forcément une attitude égoïste (souvenez-vous qu'ils se privent ainsi des correctifs et des nouvelles fonctionnalités) : ils ont simplement décidé de ne prendre aucun risque avec les mises à jour à cause de contraintes spécifiques. De la même manière, si le projet découvre un bogue important dans la version 1.0.3, peu de temps avant de sortir la version 1.1.0, il serait sévère de n'effectuer la correction que dans la version 1.1.0 et de demander à tous les utilisateurs des versions 1.0.x de se mettre à jour. Pourquoi ne pas publier à la fois les versions 1.1.0 et 1.0.4 pour contenter tout le monde ?

Une fois la ligne de version 1.1.x bien avancée, vous pouvez déclarer officiellement la fin de support des versions 1.0.x. Que cette

annonce bénéficie d'un communiqué particulier ou qu'elle soit mentionnée lors de la sortie de la version 1.1.x importe peu, les utilisateurs doivent être informés que l'ancienne ligne n'est plus supportée pour être en mesure de prendre des décisions sur une éventuelle mise à jour.

Certains projets fixent une durée de support pour les anciennes lignes. Dans un contexte Open Source, « supporter » signifie accepter les rapports de bogues et publier des versions de maintenance lorsque des bogues importants sont trouvés. D'autres projets ne se fixent pas de durées précises, mais s'appuient sur le nombre de rapports de bogue reçus pour évaluer le nombre d'utilisateurs de l'ancienne ligne. Lorsque la proportion descend sous un certain seuil, ils déclarent la fin de cette ligne et en stoppent le support.

À chaque publication, assurez-vous que le système de suivi de bogues propose bien les versions et jalons nécessaires pour permettre aux personnes rapportant les problèmes de les associer à la version adéquate. N'oubliez pas non plus de proposer un jalon « Développement » ou « Récent » pour la version de développement puisque certaines personnes (et pas uniquement des développeurs actifs) gardent souvent une longueur d'avance sur la version officielle.

7.6.1 Mises à jour de sécurité

La plupart des détails concernant la gestion des failles de sécurité ont été abordés dans la section « Annoncer les failles de sécurité » du chapitre 6, mais il nous reste à préciser certains points concernant la publication des mises à jour de sécurité.

Une mise à jour de sécurité est une version dédiée à la résolution d'une vulnérabilité. Le code qui résout le problème ne doit pas être rendu public avant la mise à disposition d'un correctif, ce qui signifie non seulement que le correctif ne peut être enregistré dans le dépôt avant le jour de sa sortie, mais également, que cette version ne peut faire l'objet de tests publics avant sa parution officielle. Les développeurs peuvent, bien entendu, examiner le correctif en interne et tester la version en privé, mais il est impossible de faire des tests à grande échelle.

Pour cette raison, une mise à jour de sécurité, par rapport à la version précédente, n'ajoute que le correctif de sécurité : aucune autre modification ne doit y figurer. La raison est simple : plus vous ajoutez de modifications non testées, plus vous augmentez le risque que l'une d'entre elles contienne un nouveau bogue, peut-être même

un nouveau bogue de sécurité! Ce conservatisme va aussi dans le sens des administrateurs qui pourraient avoir besoin de déployer le correctif de sécurité sans surprise.

Les mises à jour de sécurité vous contraindront, parfois, à quelques tours de passe-passe. Par exemple, le projet pourrait déjà travailler sur la sortie de la 1.1.3 (déjà publiquement annoncée) corrigeant quelques bogues de la version 1.1.2, lorsqu'un rapport de sécurité arrive. Évidemment, les développeurs ne peuvent pas parler du problème de sécurité avant d'avoir mis un correctif à disposition. Ils doivent donc continuer à parler en public de la version 1.1.3 avec son contenu prévu initialement. Mais la version 1.1.3 réelle n'embarquera finalement que le correctif de sécurité et tous les autres changements devront être reportés à la version 1.1.4 (qui contiendra bien sûr le correctif de sécurité, comme toutes les versions ultérieures).

Vous pouvez utiliser le système de numérotation pour signaler les mises à jour de sécurité. Par exemple, le numéro de version 1.1.2.1 pourrait signaler qu'il s'agit d'une mise à jour de sécurité de la version 1.1.2 et que toutes les versions supérieures (1.1.3, 1.2.0, etc.) contiennent le même correctif de sécurité. Pour les utilisateurs avertis, ce système fournit de nombreuses informations. Cela risque néanmoins de sembler étrange à ceux qui ne suivent pas le projet. Ils auront l'habitude de voir un numéro à trois chiffres la majeure partie du temps puis un quatrième apparaît de temps en temps. La plupart des projets que j'ai suivis choisissent la constance et utilisent simplement le numéro suivant pour les mises à jour de sécurité, même si cela implique de repousser d'un numéro les versions prévues.

7.7 Publications et développement quotidien

Le maintien de plusieurs versions simultanées a des conséquences sur le développement au quotidien. Le projet doit s'astreindre à une certaine discipline (recommandée de toute façon) : chaque modification n'apporte qu'un changement à la fois et des changements isolés ne sont jamais mélangés au sein d'un même *commit*. Si un changement est trop important ou trop impactant, il est conseillé de le diviser en plusieurs *commits* plus petits, chacun étant une sous-partie bien définie et n'embarquant rien qui ne soit relatif au changement global.

Voici un exemple de modification mal conçue :

```
-----  
r6228 / toto / 30-06-2004 22:13:07 -0500 (Mercredi,
```

```
30 juin 2004) / 8 lignes
Corrige le problème #1729: Faire que l'indexation
avertisse l'utilisateur quand un fichier est modifié
lors de son indexation.

* ui/repl.py
(ChangingFile): Nouvelle classe d'exception.
(DoIndex): Prise en charge de la nouvelle exception.

* indexer/index.py
(FollowStream): Afficher une nouvelle exception
si un fichier change durant l'indexation.
(BuildDir): Sans lien, retire des commentaires
devenus obsolètes, refonte d'une partie du
code et corrige les vérificateurs d'erreurs
lors de la création d'un répertoire.

Autres nettoyages sans lien:

* www/index.html: Correction de quelques
coquilles, date de la prochaine mise à jour
-----
```

Le problème apparaît dès que quelqu'un veut transposer le correctif portant sur le vérificateur d'erreur `BuildDir` dans une branche de maintenance. Les autres modifications ne sont pas souhaitées : le correctif du problème #1729 n'a pas forcément été approuvé pour la branche de maintenance et les améliorations de `index.html` sont ici sans objet. Mais il ne peut pas extraire facilement la modification apportée à `BuildDir` grâce aux outils de fusion du logiciel de gestion de versions puisque la modification groupe logiquement diverses modifications sans rapport. En fait, pas besoin d'attendre la fusion pour voir le problème apparaître. Le simple fait de lister la modification pour la soumettre au vote serait problématique : au lieu de simplement fournir le numéro de révision, la personne qui propose le vote devrait préparer un correctif spécial, ou changer de branche simplement pour isoler la partie du changement qu'il désire faire valider. Tout ceci rend le travail des autres plus difficile, uniquement parce que celui qui a enregistré ces modifications n'a pas pris la peine de les séparer en groupes logiques isolés.

Concrètement, ce *commit* unique aurait dû être séparé en quatre *commits* distincts : un pour le problème #1729, un autre pour retirer les commentaires devenus obsolètes et re-formater le code dans `BuildDir`, un autre pour réparer la vérification d'erreurs dans `BuildDir` et le dernier portant sur les améliorations de `index.html`, le troisième étant le changement proposé pour la branche de la version de maintenance.

Les changements doivent être atomiques, pas seulement pendant la période de stabilisation pré-publication, mais tout le temps. Psy-

chologiquement, un changement unifié autour d'un thème précis est plus facile à vérifier et à annuler si nécessaire (dans certains logiciels de gestion de versions, l'annulation est en fait un type particulier de fusion). Si chacun fait preuve d'un peu de discipline de gros problèmes ultérieurs peuvent être évités.

7.7.1 Planifier les publications

L'un des domaines où les projets Open Source diffèrent historiquement des projets propriétaires, est celui de la planification des publications. Les projets propriétaires ont, en général, des dates butoirs bien définies. Parfois parce qu'on a promis aux clients une mise à jour à une date précise, parce que la nouvelle version doit être coordonnée avec d'autres actions liées aux marketing, ou parce que les partenaires financiers qui ont investi dans le projet demandent à voir des résultats avant de s'engager plus avant. Les projets de logiciels libres, d'un autre côté, étaient encore récemment motivés principalement par l'« amateurisme » pris dans son sens le plus littéral : les développeurs écrivaient simplement par plaisir. Personne ne ressentait l'obligation de livrer le produit avant que toutes les fonctionnalités ne soient prêtes, et d'ailleurs pourquoi auraient-ils dû le faire ? Ce n'était pas comme si l'emploi de quelqu'un était en jeu.

De nos jours, de nombreux projets libres sont financés par de grands groupes et sont donc, de plus en plus, influencés par la culture d'entreprise et ses dates butoirs. C'est positif à bien des égards, mais cela peut créer des conflits entre les priorités des développeurs rémunérés et celles des bénévoles. La question de la date de publication et de son contenu cristallise souvent ces divergences. Les développeurs salariés, sous pression, voudront naturellement choisir une date pour la prochaine version et souhaiteront que l'activité de chacun s'aligne dessus. Mais les priorités des volontaires peuvent être bien différentes, peut-être souhaitent-ils achever des fonctionnalités ou quelques tests, ils estimeront donc que la sortie peut attendre.

Seuls la discussion et le compromis permettent de résoudre ce genre de problème. Mais vous pouvez limiter la fréquence et l'importance des frictions en dissociant version future et date de sortie. Essayez d'orienter la discussion sur les versions à court et moyen termes et sur les fonctionnalités que ces dernières devraient incorporer. N'avancez pas encore de date, sauf pour donner un ordre d'idée, et encore, avec une grande marge d'erreur¹. Une fois les évo-

1. Pour un autre point de vue je vous conseille la lecture de la thèse de doctorat de Martin Michlmayr : « Quality Improvement in Volunteer Free

lutions déterminées, les discussions autour de chaque version sont plus cadrées et donc moins sujettes à controverses. Vous créez en même temps une certaine inertie à vaincre pour quiconque souhaiterait proposer de nouvelles fonctionnalités ou complications. Si le périmètre d'une version est bien défini, il appartient à celui qui propose l'élargissement de le justifier, même si la date de publication n'a pas encore été fixée.

Dans sa biographie en plusieurs volumes de Thomas Jefferson, intitulée *Jefferson et son temps*, Dumas Malone raconte comment Jefferson a dirigé la première réunion tenue pour décider de l'organisation de la future Université de Virginie. L'université était au départ une idée de Jefferson, mais (comme c'est le cas partout, pas uniquement dans les projets libres) beaucoup d'autres personnes se sont rapidement jointes au projet, chacun avec ses propres intérêts et ses propres plans. Lorsqu'ils se sont réunis la première fois pour débroussailler le terrain, Jefferson s'est présenté avec des plans architecturaux méticuleusement préparés, un budget détaillé pour la construction et le fonctionnement, une proposition de cursus et les noms de professeurs qu'il voulait faire venir d'Europe. Personne d'autre dans l'assemblée, et de loin, n'était aussi bien préparé, le groupe a donc dû s'en remettre à la vision de Jefferson et finalement l'université a été conçue à quelques détails près selon ses plans. Il avait certainement prévu que le coût de la construction dépasserait largement le budget et que beaucoup de ses idées ne fonctionneraient finalement pas (pour diverses raisons). Son but était purement stratégique : se présenter devant l'assemblée avec quelque chose de si solide que les autres en seraient réduits au rôle de consultants et qu'ils ne pourraient plus proposer que de simples modifications. Il s'assurait ainsi que le projet se déroulerait comme il le souhaitait dans les grandes lignes et selon le planning qu'il avait imaginé.

Concernant les logiciels libres, il n'y a pas de grande assemblée générale mais plutôt une série de petites propositions faites principalement par le biais du système de suivi de bogues. Mais si vous avez déjà une certaine crédibilité au sein du projet, et que vous commencez à prévoir différentes fonctionnalités, améliorations et bogues pour des versions précises et en accord avec un plan général annoncé, les participants vous suivront sans problème. Une fois que vous avez

and Open Source Software Projects : Exploring the Impact of Release Management » (<http://www.cyrius.com/publications/michlmayr-phd.html>) [NdT : « Amélioration de la qualité dans les projets volontaires de logiciels libres et Open Source : l'impact de la gestion des sorties »]. Elle traite de la gestion des sorties non plus sur la base des fonctionnalités, mais sur une base temporelle dans les projets importants de logiciels libres. Michlmayr s'est aussi exprimé sur le sujet chez Google, et la vidéo est disponible sur Google Vidéo à l'adresse <http://video.google.com/videoplay?docid=-5503858974016723264>.

établi les choses à peu près comme vous le désirez, les conversations à propos de la date de sortie seront moins problématiques.

Il est bien sûr crucial de ne jamais présenter une décision unilatérale comme définitive. Dans les commentaires associés à chaque attribution d'un problème à une version particulière, initiez des discussions ou des protestations et montrez-vous authentiquement ouvert à la persuasion quand c'est raisonnable. Ne dirigez pas simplement pour le plaisir de diriger : plus les autres personnes participent au planning de sortie (voir la section « Partager les tâches de gestion aussi bien que les tâches techniques » du chapitre 8), plus il vous sera facile de les convaincre du bien-fondé de vos priorités sur les points qui vous tiennent à cœur.

Pour éviter de cristalliser les tensions autour du planning, le projet peut aussi proposer un rythme soutenu de publication. Quand le temps entre deux sorties est long, l'importance d'une nouvelle version est décuplée dans l'esprit des développeurs et ils sont beaucoup plus affectés si leur code n'y est pas incorporé parce qu'ils savent qu'il va leur falloir attendre longtemps avant d'avoir une nouvelle chance. Selon la complexité du processus de publication et la nature de votre projet, un délai de trois à six mois est en général un bon écart entre deux sorties, bien que les lignes de maintenance puissent soutenir un rythme plus important pour les micro-versions si elles sont nécessaires.