Collaboration avec des projets libres - enjeux, difficultés et bonnes pratiques

Olivier Berger, Quang Vu Dang, Christian Bac

Collaboration avec des projets libres - enjeux, difficultés et bonnes pratiques

Christian Bac
christian.bac@int-edu.eu

Vu Dang Quang
quang_vu.dang@int-edu.eu

Olivier Berger
olivier.berger@int-edu.eu

GET/INT – Département INF – projet PFTCR
9, rue Charles Fourier, 91011 Evry Cedex France

Résumé

Nous souhaitons proposer quelques pistes permettant d'affiner des stratégies de collaboration avec les projets de développement de logiciels libres, pour les organisations basant le développement de leurs systèmes d'information sur l'intégration et la customisation d'applications libres existantes.

Mots clefs

contribution, logiciel libre, open source, meilleures pratiques, debian, packages, maintenance

1 Introduction

Dans cet article, nous nous plaçons du point de vue d'une organisation souhaitant mettre en œuvre son système d'information en se basant sur des produits logiciels libres existants.

Bien souvent, une telle démarche nécessite d'adapter les logiciels choisis, et de collaborer ainsi avec d'autres acteurs ayant participé au développement de ces logiciels, en dehors des limites de l'organisation utilisatrice. Il semble alors très prometteur de pouvoir bénéficier gracieusement des efforts externes d'autres contributeurs bénévoles. Mais cela ne se fait pas de façon magique, sans qu'un certain nombre de conditions nécessaires soient réunies, sans qu'une démarche de collaboration fructueuse soit instaurée.

Ce type de collaboration est novateur pour beaucoup d'organisations, notamment pour leur management, et nécessite un minimum de préparation pour qu'elle soit réussie. Elle permet, en cas de succès, de tirer un réel bénéfice de l'utilisation et de l'enrichissement des logiciels libres mis en œuvre, y compris en tirant parti des efforts réalisés en dehors de l'organisation.

Notre présentation s'appuie sur l'expérience acquise lors du développement d'applications de travail collaboratif au Groupe des Écoles des Télécommunications[1] (GET) au cours des dernières années. Ces applications[2] sont toutes à bases de logiciels libres. Nous utilisons aussi la connaissance des projets logiciels libres acquise au travers de plusieurs projets de recherche pluridisciplinaires d'étude du logiciel libre[3].

Nous ne visons pas l'exhaustivité dans cet article. Nous espérons dresser un panorama qui aide chaque organisation confrontée à cette problématique à construire sa propre stratégie de coopération, nécessairement singulière, du fait de ses spécificités et des caractéristiques propres à chacune des communautés de développement de logiciels libres auxquelles elle sera confrontée.

2 Tirer parti des logiciels libres

2.1 Qu'est-ce qu'un logiciel libre ?

Le logiciel libre est un concept universellement associé aux notions de liberté pour les utilisateurs des logiciels. Ces notions ont été formalisées par la *Free Software Foundation* (FSF) il y a maintenant une vingtaine d'années au travers de 4 libertés fondamentales[4].

Tout logiciel libre ainsi défini apporte des avantages indéniables à ses utilisateurs, en rééquilibrant les droits et devoirs des auteurs ou éditeurs par rapport aux utilisateurs, notamment :

- diminution du risque de clients captifs par une concurrence accrue,
- partage de connaissance entre tous les acteurs,
- respect des standards, portabilité,
- réversibilité sur les tâches sous-traitées, etc.

Nous n'adresserons pas ici tous ces avantages généraux de l'utilisation des logiciels libres, la littérature (et la propagande) étant suffisamment abondante.

[1] http://www.get-telecom.fr/

[2] successivement, dans le cadre du projet PFTCR : *PicoLibre*, *ProGET* [1], *PicoForge* (http://www.picoforge.org/) [2] et *Perseus.*

[3] projet Européen FP6 IST *CALIBRE*, projet GET *Contrib2.0.*

[4] soit : « *La liberté d'exécuter le programme, pour tous les usages (liberté 0). La liberté d'étudier le fonctionnement du programme, et de l'adapter à vos besoins (liberté 1). Pour ceci l'accès au code source est une condition requise. La liberté de redistribuer des copies, donc d'aider votre voisin (liberté 2). La liberté d'améliorer le programme et de publier vos améliorations, pour en faire profiter toute la communauté (liberté 3). Pour ceci l'accès au code source est une condition requise.* » (http://www.gnu.org/philosophy/free-sw.fr.html)

2.2 Rapide panorama d'un écosystème

Si un logiciel libre est souvent caractérisé de façon non-ambiguë par l'examen de sa licence, il y a bien d'autres aspects essentiels à considérer.

Comme nous le verrons, les qualités d'un logiciel libre ne sont pas qu'internes, strictement logicielles, mais aussi liées à un environnement, l'écosystème d'acteurs impliqués dans son développement et son utilisation.

On peut distinguer différents acteurs clés :

- les développeurs initiaux, en « amont » (*upstream*)
- les « packageurs » dans les différentes distributions
- les autres utilisateurs
- des prestataires spécialisés

L'ensemble de ces acteurs contribue, chacun à son niveau, à déterminer l'évolution d'un logiciel libre.

Les développeurs initiaux peuvent être un groupe d'individus fortement organisé, ou très peu structuré, selon les cas. Un logiciel libre est parfois développé par un éditeur, dans une configuration assez classique, parfois dans une communauté virtuelle sur l'Internet, à titre purement bénévole, parfois avec un mélange de ces deux cas de figure.

La stratégie de publication des développeurs « amont » peut varier du tout au tout : parfois très planifiée, parfois uniquement dictée par des périodes de temps écoulé (*snapshots*) [3].

Les « packageurs » jouent un rôle essentiel pour industrialiser les productions des développeurs amont, par rapport à une distribution logicielle. Ils jouent un rôle d'intermédiaire avec les utilisateurs finaux, en s'assurant que le logiciel est configurable simplement, en filtrant les niveaux de criticité des mises à jour, en filtrant les remontées de bugs ou demandes d'améliorations.

Certains prestataires peuvent se spécialiser sur un logiciel libre, et assurer la sous-traitance de certaines tâches pour des utilisateurs finaux. Ils contribuent souvent (espérons-le) aux projets à plusieurs niveaux, par exemple en sponsorisant certains développeurs amont, ou en reversant, après un certain temps, du savoir-faire financé par leurs clients.

Il convient souvent d'interagir avec un ou plusieurs de ces acteurs. Tous n'ont pas les mêmes agendas, ni le même souci de leurs utilisateurs. Par exemple, ce n'est pas parce qu'une nouvelle fonctionnalité est transmise comme contribution vers le projet amont, qu'elle sera packagée très rapidement dans les distributions logicielles populaires. Certains acteurs peuvent par exemple se « renvoyer la balle » par rapport au traitement de certains bugs, difficiles à reproduire en dehors d'un environnement ou d'une plate-forme spécifique.

Les exemples ne manquent pas de difficultés rencontrées dans les interactions avec les communautés libres. Même si dans certains cas les relations seront assez formelles, voire contractuelles, par exemple avec les acteurs commerciaux éditant un logiciel libre ou proposant des offres de support, la plupart du temps, il n'y a aucune garantie de pouvoir régler un problème, face à des participants bénévoles.

Il faut alors réussir à mettre en œuvre des stratégies de relations sociales permettant d'inciter les autres à nous aider. Dans une telle relation plus ou moins désintéressée, l'espoir d'accès à une aide sera bien souvent conditionné par l'implication affichée vis-à-vis du projet.

On retrouve dans les communautés du logiciel libre les mêmes rapports que dans toute organisation humaine[5], avec tous les enjeux psychologiques et sociaux, les rapports méritocratiques, les enjeux de don/contre-don, de pouvoir, de réputation, etc.

2.3 Qualités particulières d'un logiciel libre

Quand on souhaite mettre en œuvre un logiciel, notamment à des fins d'utilisation dans un contexte professionnel, de nombreux facteurs sont à prendre en compte quant aux qualités de ce logiciel, et la licence de celui-ci, son caractère libre, n'est alors qu'un critère d'évaluation parmi d'autres.

Les caractéristiques propres de ce logiciel sont évidemment essentielles : absence de défauts, documentation existante, respect des standards, adéquation aux besoins, complexité à administrer, etc. Ces caractéristiques, ces critères de qualité ne sont pas spécifiques de son caractère libre – voir la littérature abondante sur la qualité du logiciel.

Par contre, pour les logiciels libres, du fait de l'absence, en général, de garantie et de contrat avec un éditeur, il faut ajouter à ces critères classiques, et de façon beaucoup plus importante, l'évaluation précise de la qualité du processus de production, la vitalité de la communauté développant et utilisant ce logiciel (cf. 3.2).

3 Conduite de projet

Conduire un projet informatique est toujours un art subtil. Dans un environnement où les logiciels libres entrent en jeu, la gestion du projet peut en être complexifiée. Mais c'est le prix à payer pour bénéficier des autres avantages.

Avant tout, il convient d'avoir une vision claire de l'état des projet libres, et des contraintes internes, et d'affiner la stratégie à adopter. L'essentiel des difficultés, comme souvent, n'est pas dans les aspects purement techniques, mais bien plus dans le management, les relations humaines, la communication, l'anticipation des problèmes.

Nous allons passer en revue brièvement les différentes phases d'un projet informatique, et mentionner, à chaque fois des précautions à prendre par rapport à certaines spécificités du logiciel libre.

3.1 Degrés d'interaction avec les communautés

Pour l'organisation utilisatrice, trois degrés principaux d'utilisation des logiciels libres peuvent être distingués :

- utilisation « simple » de solutions packagées correspondant directement au besoin, sans nécessité d'adaptation,
- « customisation » de solutions existantes,

[5] cf. [4], entre autres, pour des analyses plus approfondies.

– intégration lourde de différentes solutions, packaging et déploiement d'un ensemble plus large, potentiellement « mission critical ».

Ces degrés d'utilisation impactent différemment l'exigence d'interaction, de collaboration avec les communautés externes de développement des logiciels.

Dans la première situation, le langage de programmation peut n'avoir qu'un incidence mineure, par exemple.

Dans la seconde situation, la vitalité des communautés de développement peut être un facteur clé.

Par exemple, en connaissant la feuille de route (*roadmap*) d'un projet, et la probabilité qu'elle soit réalisée dans les temps prévus, on peut identifier les adaptations qui sont déjà à l'agenda du projet libre et qui correspondent à des améliorations que l'on aurait entreprises indépendamment. La recherche d'information sur les fonctionnalités nouvelles qui ne sont pas encore complètement intégrées permet souvent de limiter l'effort à fournir au seul niveau des test plutôt que d'une implémentation complète.

Dans la troisième situation, il faut probablement se concentrer sur quelques éléments clés, sur lesquels l'essentiel de l'effort doit être porté (adapteurs, customisation, packaging, tests), et chercher l'assistance de prestataires spécialisés sur des éléments très spécifiques, évaluer des solutions de repli, etc. Dans une telle situation, il semble bien difficile de pouvoir interagir en même temps avec chacune des communautés des différents composants intégrés.

3.2 Sélection d'un produit

Au delà des principes de fond et des avantages généraux apportés par une licence libre, il convient d'évaluer chaque logiciel, et chaque écosystème associé, chaque communauté, comme un cas particulier.

3.2.1 Simplicité et réactivité

Les logiciels libres sont largement disponibles (voire pléthoriques[6]), faciles à tester, et la réalisation d'un prototype est souvent assez simple. La documentation est également disponible, et du support facile à trouver pour peu qu'on ait accès aux forums en ligne et du temps devant soi.

À la condition de disposer des compétences nécessaires, tester un logiciel libre pour construire un prototype est la situation idéale. Il n'y a en effet pas de surcoût de licence, pas de problème de « piratage », pas de nécessité de rentrer dans un processus d'achat (notamment pour un organisme public), pas de mise en concurrence à réaliser. Cette simplicité améliore la réactivité pour proposer aux utilisateurs une solution à court terme.

3.2.2 Ne pas confondre vitesse et précipitation

Ce n'est pas parce qu'un logiciel est facile à tester et qu'un prototype semble utilisable rapidement, que ce logiciel répond à toutes les qualités attendues, notamment vis-à-vis de la pérennité de la solution si la communauté de développement n'est plus active (failles de sécurité, etc.).

Même si les phases de test et de sélection d'une solution potentiellement utilisable sont réduites par rapport au monde « propriétaire » dans lequel il est nécessaire de négocier l'achat ou le prêt d'un logiciel pour les besoins d'une évaluation, cela ne doit surtout pas dispenser de faire des études poussées, de planifier.

3.2.3 Sous-traiter ce qui peut l'être

Il peut être tentant, pourvu qu'on ait la compétence pour ce faire, de tout faire soi-même, pour maîtriser l'ensemble de la chaîne en choisissant progressivement chacun des éléments, chaque version, chaque niveau de patch appliqué, d'autant que le code source et toutes les informations sont disponibles en direct, du producteur au consommateur.

Ce n'est pas parce que, en général, le logiciel libre est gratuit, qu'il faut imaginer que toute relation marchande doive échapper au monde du logiciel libre. Et ce n'est pas parce que l'Internet regorge de documentation sur le libre, qu'on doit tout apprendre par soi-même. D'un point de vue financier, et pour maîtriser les délais, il conviendra d'évaluer, comme ailleurs, ce qui peut être sous-traité.

Les sociétés de service généralistes ou spécialisées sont largement à même de répondre en compétences sur les logiciels libres. Autant envisager cette possibilité, d'autant que c'est aussi ainsi que le développement des logiciels libres peut être financé, au travers des développeurs embauchés chez ces prestataires[7]. Il reste à identifier le niveau d'exigence, et les prestataires à même de répondre, ce qui est loin d'être simple.

3.2.4 Identifier les éléments critiques et monter en compétence

Il sera important d'identifier le plus tôt possible les logiciels critiques sur lesquels on s'appuiera, pour « monter en compétence » sur ces éléments.

L'activité de veille devra alors être assez importante, même si elle ne s'avère pas immédiatement productive, mais elle permettra de mieux connaître le projet, les acteurs, et d'identifier l'état du processus de développement, les risques, et de mieux anticiper sur les éventuels problèmes externes, pour mieux prévoir des contournements. L'abonnement aux différents flux d'information (blogs, forums, listes de discussion) sera alors indispensable (cf. 4.2.2).

Mais identifier « le bon » logiciel n'est pas forcément chose aisée même si des méthodes d'évaluation et de comparaison commencent à être disponibles[8].

3.2.5 Préférer les solutions déjà packagées

Mais même s'il est louable de vouloir maîtriser complètement les réalisations, d'auditer le code, il est important de garder une certaine capacité d'automatisation des processus.

Plusieurs décennies de génie logiciel nous interdisent en effet de constituer des solutions assemblées de bas en haut,

[6] voir les catalogues comme Freshmeat (`http://freshmeat.net/`) ou celui de la FSF et de l'Unesco (`http://directory.fsf.org/`).

[7] salariés à plein temps, ou aussi des experts « free-lance » intervenant au cas-par-cas.

[8] voir les pointeurs réunis dans l'article WikiPedia `http://fr.wikipedia.org/wiki/Méthode_d'évaluation_de_logiciels_libres`

non packagées et non supportées, en déployant des versions instables en perpétuelle évolution, ou ne répondant pas à leur cahier des charges.

Hormis dans des situations très particulières sur des logiciels critiques, il est préférable de s'appuyer sur des solutions packagées, et minimiser les manipulations du code. Il sera plus simple en général d'intervenir sur un système intégré à partir de packages standards, plutôt que sur un assemblage manuel complet :

- le *turn-over* dans les équipes doit être anticipé, et les solutions génériques sont plus simples à comprendre,
- en cas de *crash*, la mise en œuvre du plan de secours en sera aussi grandement facilitée,
- la réorganisation de la plate-forme sera facilitée une fois le système mis en production (montée en charge, etc.)

Parfois, dans les distributions qui ont des cycles de parution de versions stables assez espacées (Debian par exemple), les versions packagées sont parfois en retard sur les versions amont[9]. Il pourra alors être nécessaire d'acquérir des compétences en matière de packaging, et pourquoi pas contribuer à la mise au point d'une version packagée générique plus rapidement disponible (*backports* par exemple, cf. 3.5.3). D'autres distributions, *gentoo* par exemple, ne connaissent pas ces problèmes de fraîcheur[10].

3.2.6 Choix du système de packages

Pour maîtriser la question clé de la gestion de configuration, notamment pour le déploiement, il faut pouvoir s'appuyer sur un système de packaging maîtrisé (gestion de dépendances, alternatives, etc.). Debian propose un tel système qui nous semble donner d'excellent résultats (y compris dans les distributions dérivées, telle Ubuntu).

D'où l'importance d'avoir une conception des développements qui privilégiera un approche modulaire, et le packaging, de façon impérieuse, plutôt que la construction d'applications monolithiques.

3.3 Maintien en condition opérationnelle

Nous abordons cette phase avant celle du développement proprement dit car elle nous semble une des phases clés à prendre en considération, souvent sous-estimée, et dont les contraintes doivent impacter fortement la planification des phases antérieures.

En effet, les développeurs ont trop tendance à considérer un projet comme terminé une fois la première version réalisée, et à négliger les étapes ultérieures qui ont tendance à coûter souvent beaucoup plus cher.

Certains mécanismes permettent de mieux maîtriser la traçabilité des mises à jour, des *patches*, la cohabitation de versions standard et de customisations spécifiques, qui aideront à diminuer la charge de maintien en condition opérationnelle.

3.3.1 Réactivité pour les mises à jour

Les failles dans les solutions génériques largement répandues (logiciels libres y compris) sont très vite connues et exploitables, d'autant que nombre d'applications sont déployées via des services (sur le) Web en Intranet ou sur Internet. Ceci a tendance à plus exposer les systèmes d'information (D.O.S.[11], intrusions, etc.) que pour des solutions spécifiques. Il est donc très important de garantir une grande réactivité et la capacité de déployer des mises à jour de sécurité sur les systèmes en production exposés.

Lorsqu'un système est en condition opérationnelle, la tentation est grande de ne pas y toucher de peur qu'il ne tombe en panne. Le plus simple pour ne pas avoir de problème est alors de tout figer, de ne rien changer.

C'est aujourd'hui illusoire : les mises-à-jour de sécurité sur les logiciels libres paraissent en flux constant. Face à cela, les éditeurs de distributions, notamment, ont des stratégies de support à moyen terme de versions stabilisées qui peuvent sembler séduisantes. Pour autant qu'elles assurent un socle stable, sur lequel les mises à jour sont limitées au minimum, elles risquent de tendre à une certaine obsolescence fonctionnelle : les utilisateurs ont tendance à demander les nouvelles fonctionnalités.

Le choix de la version cible des plate-formes hôtes, pour le déploiement et la maintenance, devra donc balancer entre ces tendances contradictoires.

3.3.2 Diminuer l'adhérence dans les composants spécifiques

Face à ce flux de mises à jour, il convient de diminuer la charge de ré-installation, re-paramétrage et re-modification à faire à chaque nouvelle version. C'est notamment le cas lorsque des développements spécifiques ont été réalisés.

Parfois, les plate-formes hôtes des applications proposent des mécanismes de *plugins* permettant d'isoler le périmètre des adaptations[12]. Alors les mises à jour n'impactent pas nécessairement tous les plugins installés.

Quand un tel mécanisme de plugins n'existe pas, il est souvent nécessaire de « tailler dans l'applicatif », par exemple sous forme de patches d'extension spécifiques à installer. Dans ce cas, il faudra limiter au maximum l'ampleur de ces patches, par exemple en les concevant comme des adapteurs (*wrappers*) qui isoleront « proprement » ce qui resterait spécifique. Seuls les patches ajoutant ces adaptateurs seraient à réinstaller à chaque mise à jour du code standard.

Autre possibilité, diminuer l'ampleur des éléments spécifiques à gérer de façon particulière, et transférer un maximum de choses dans le tronc commun générique (cf. 4.1.3).

---

[9] voir à ce sujet : *Vérifier qu'un paquet est toujours maintenu* (`http://www.ouaza.com/wp/2007/08/06/verifier-quun-paquet-est-toujours-maintenu/`)

[10] d'autres encore proposent plusieurs versions (au moins deux) du même logiciel.

[11] D.O.S. (*Denial Of Service*) : déni de service

[12] les technologies de type Serveur d'Applications et modèles de composants visent aussi à résoudre ce type de problématiques, sans qu'elles s'imposent notoirement, aujourd'hui, dans le logiciel libre. Des projets comme OW2 (ex ObjectWeb) et les technologies Java, désormais libres, sont porteuses d'espoir dans ce domaine.

3.4 Développement

Quand des développements sont nécessaires, de très nombreuses problématiques entrent en jeu, dont certaines sont liées à l'organisation interne des équipes.

En général, quand un ensemble de fonctionnalités nouvelles sont à ajouter dans un logiciel libre existant, il y a de grandes chances que des interactions fortes soient nécessaires avec les développeurs externes.

De façon générale, on privilégiera une méthode de conduite de projet souple, dite « développement agile »[13], permettant de réévaluer constamment les risques, de re-calibrer l'effort et de procéder par incréments successifs, au fur et à mesure des interactions avec les développeurs externes et les utilisateurs.

3.4.1 Rendre générique ce qui peut l'être

Souvent, les besoins d'un projet amènent à implémenter des fonctions assez spécifiques dans un ensemble plus générique (cf. 3.3.2).

Il est délicat, voire inutile d'essayer de faire intégrer dans les projets externes des éléments très spécifiques relatifs aux besoins internes. Mais quoiqu'il en soit, plus on y arrivera, moins la charge de maintenance ultérieure sera grande (*outsourcing* de la maintenance), ce qui s'avérera important pour la vie ultérieure du système (Cf. 4.1.3).

D'où l'importance de réaliser les développements en adoptant dès le début une méthodologie compatible avec celle des projets externes (cf. 4.2.7).

3.4.2 Méthodologie d'intégration

Il peut être assez tentant de réutiliser des morceaux de solutions piochés à droite ou à gauche, pour construire un assemblage qui convienne aux besoins spécifiques qu'on rencontre.

C'est souvent faisable rapidement, et d'autant plus facilement en procédant « à la hache », plutôt qu'en essayant de créer des adapteurs, des *wrappers*, en essayant d'intégrer des solutions d'interopérabilité dans des applications différentes. Pourtant la deuxième approche est nettement préférable pour la maintenabilité future (cf. 3.3.2).

On préférera ainsi un découpage modulaire maximum, avec une intégration de haut niveau, permettant d'isoler les modifications, d'assembler sans avoir à dégrader chacune des applications intégrées.

3.5 Déploiement

En environnement GNU/Linux notamment, le choix d'une distribution cible pour les déploiements est relativement sensible.

3.5.1 Support de certains matériels

Certains matériels spécifiques nécessitent des drivers ou des applications particulières de monitoring (pas toujours libres) pour fonctionner au mieux.

L'installation d'un OS certifié par le constructeur, accroît la garantie de disponibilité des matériels de manière sensible, ce qui contre-balence le coût supplémentaire (licences, support).

3.5.2 Virtualisation

L'utilisation d'une version du système d'exploitation pour laquelle le support matériel est garanti n'empêche pas de déployer des logiciels sur un niveau de virtualisation intermédiaire.

L'approche de virtualisation permet de ne pas mettre tous ses œufs dans le même panier. Par exemple, il est possible de déployer le matériel avec un OS RedHat ou Suse, puis d'ajouter une couche de virtualisation (Xen[14] par exemple), et d'installer des systèmes Debian dans des serveurs virtualisés pour bénéficier des apport du système de packaging.

En outre, cela permettra de réorganiser la plate-forme, par exemple en cas de montée en charge, au gré des migrations des machines virtuelles.

3.5.3 Savoir packager

Nous souhaitons insister sur la nécessité de maîtriser les techniques de déploiement de packages, et aussi, souvent, de construction/adaptation de packages.

Nous mentionnons, à titre d'exemple dans Debian, quelques techniques utiles à explorer :

- conversion de format packages avec l'outil *alien*, puis customisation via les scripts *debconf*,
- *backports* de packages récents dans des distributions anciennes[15],
- surcharge de certains fichiers originaux, via les *overrides* dans des packages de customisation,
- miroirs de déploiement internes (pour *apt*, par exemple), etc.

Nous engageons le lecteur intéressé par ce type de fonctionnalités à se référer aux manuels des développeurs de sa distribution, pour plus de détails.

4 Contribuer : une nécessité

Au travers des différentes phases de la vie d'un projet informatique que nous avons examinées, nous avons mis en évidence la nécessité de collaborer avec d'autres acteurs qui sont impliqués dans le développement des logiciels libres utilisés.

Cette collaboration se concrétise au mieux via une contribution bénévole substentielle, d'un élément logiciel ou apparenté (documentation, jeux de tests, support), qui permet d'ancrer une confiance réciproque entre acteurs bénévoles, et qui bénéficie aux différentes parties.

4.1 Pourquoi contribuer

Longtemps, les organisations ont eu trop tendance à ré-inventer en interne plutôt que de privilégier la réutilisation

[13] voir une intriduction, par exemple, dans WikiPedia, dans : `http://fr.wikipedia.org/wiki/Méthode_agile`

[14] Xen : `http://xen.xensource.com/`

[15] par exemple, certains packages sont déjà disponibles sur `http://www.backports.org/`

de composants standards[16]. De même, il n'est pas naturel d'abandonner le contrôle sur un élément produit en interne, de le donner à des tiers, pour de nombreuses raisons d'ordre psychologique ou stratégique.

4.1.1 Cercle vertueux des contributions

Il est très facile d'être un « passager clandestin » du logiciel libre, en tirant parti du pot commun, sans jamais y reverser. C'est naturel, c'est autorisé, mais une trop grande disproportion entre ceux qui donnent et ceux qui prennent n'est pas un gage de pérennité du modèle.

Cela frise l'évidence : contribuer au projet libre un maximum de ce qui peut l'être est la meilleure garantie, le meilleur investissement dans la pérennité de celui-ci. Ce n'est pas une question d'ordre moral, mais seulement une recherche de mutualisation où chacun doit prendre sa part pour que le modèle puisse passer à l'échelle de façon réaliste.

4.1.2 Quasi-obligation du fait des licences

Certaines licences de logiciels libres ont des clauses de réciprocité (*copyleft*) qui requièrent la publication, sous la même licence que l'original, de toute modification diffusée à des tiers[17].

Cette obligation ne concerne pas les adaptations destinées à une utilisation en interne dans une organisation. Certains peuvent estimer pouvoir s'abstraire de cette exigence, dans un premier temps, car ils n'envisagent pas qu'une distribution à des tiers survienne. Cependant, il est difficile d'augurer du devenir d'un logiciel, et diffuser dès le début ses adaptations conformément aux exigences de la licence peut être une sage précaution.

Tout ceci suppose que les développeurs et les décisionnaires soient informés sur les licences de logiciels, leurs différences, leurs exigences, et que les contraintes propres à chaque licence soient bien connues de tous (cf. 4.2.7).

4.1.3 Externalisation de la maintenance

De nombreuses entreprises rêvent aujourd'hui de surfer sur la vague du libre, du gratuit, des réseaux sociaux, pour diminuer leurs coûts de R&D, de marketing, en externalisant aux « communautés » anonymes et bénévoles une partie de leurs activités (*crowdsourcing*, Web 2.0).

Une telle stratégie peut être recherchée dans le logiciel libre par tous les acteurs qui cherchent à mutualiser les coûts de maintenance [5].

Le modèle semble simple : chaque nouveauté est développée par celui (ou ceux) qui en a besoin le premier, puis reversée au pot commun, et ensuite maintenue collectivement par tous les utilisateurs qui en bénéficieront.

Bien entendu cette hypothèse séduisante ne porte pas ses promesses si personne n'amorce le processus, ou si personne ne fait le travail d'intérêt général sur la conservation du « patrimoine » assemblé au cours des évolutions du logiciel.

[16] syndrôme NIH (*Not Invented Here*)

[17] par exemple la licence GNU General Public licence (GPL), pour la plus connue (`http://www.gnu.org/copyleft/gpl.html`).

4.1.4 Se faire plaisir et apprendre en vraie grandeur

Un des principaux facteurs de motivation des développeurs de logiciels libres, mis en évidence par l'observation et des enquêtes sociologiques [6] est l'envie d'apprendre, de se former, de faire partager son savoir-faire dans une démarche de qualité.

Cette dimension n'est pas anecdotique pour la formation (continue) des informaticiens, notamment des développeurs, et la motivation des équipes. Il est vraiment très satisfaisant pour un informaticien de voir son travail reconnu par des tiers, pour ce qu'il est et non par la seule satisfaction d'un « client ».

Mettre en place une démarche coordonnée de contribution (limitée) à des projets libres, même en dehors des objectifs immédiats du service, et parallèle à l'activité de l'organisation, est un facteur d'épanouissement des collaborateurs dans les équipes techniques.

4.1.5 Se faire connaître et reconnaître

Une dimension importante de la motivation, pour les individus et pour les organisations qui contribuent au libre, est la possibilité de se faire connaître, de se faire reconnaître, au travers des contributions.

Toutes les contributions doivent alors être transmise selon un protocole plus ou moins formel (y compris du fait des contraintes en terme de propriété intellectuelle, ou du fait des contraintes de contrôle du processus de développement d'un projet). Cela passe par la signature, l'échange d'emails, et les mentions de propriété, qui popularisent le nom des développeurs ou de leurs employeurs.

Cela peut assurer une solide renommée dans certaines communautés spécialisées (pairs) permettant de légitimer des décisions, de nouer des partenariats plus efficacement.

4.1.6 Influer sur le pilotage d'un projet

Les projets libres sont pilotés de façons multiples, parfois très directrice, parfois assez lâche, souvent avec de multiples acteurs aux agendas différents. Dans un système essentiellement méritocratique (sans diminuer le poids des facteurs financiers, ou des relations personnelles), contribuer c'est pouvoir faire entendre sa voix.

À partir du moment où la contribution à un projet libre est conséquente, il est plus aisé d'avoir voix au chapitre, d'influer sur les orientations d'un projet, de pérenniser l'investissement en renforçant des éléments clés qui peuvent apparaître comme secondaires pour d'autres organisations.

4.2 Comment bien contribuer

Pour autant qu'une stratégie bien comprise de contribution aux projets libres soit définie au sein d'une organisation, il ne suffit pas de la décréter pour que cela ait un effet utile. Ce n'est pas parce qu'on met sur Internet, au coin d'un site Web ou d'un forum, un morceau de code source, que quelqu'un pourra en faire utilement quelque chose.

Il y a un certain nombre d'étapes à respecter si l'on veut que la contribution, l'investissement consenti, puisse faire son chemin et trouver son potentiel en retour pour l'organisation.

4.2.1 Il n'y a pas besoin de savoir coder

Le monde du logiciel libre repose sur les programmeurs, bien entendu, mais aussi sur nombre d'acteurs qui assurent l'ensemble des tâches autour de la simple écriture du logiciel :

- documentation,
- support aux autres utilisateurs,
- mise en place de sites de ressources,
- tests,
- rapport des bugs,
- triage des bugs, qualification et suivi,
- traductions,
- assurance qualité,
- packaging, etc.

Chacune de ces tâches peut être prise en charge, et toutes ne nécessitent pas une compétence technique.

Souvent les compétences humaines, de relation aux individus, d'organisation, sont tout aussi importantes. Elles permettent de garantir la cohésion, l'état d'esprit d'une communauté souvent composée d'individus très différents, de gérer le « turn-over », d'assurer la survie de l'entreprise virtuelle, au delà des parcours individuels.

4.2.2 Assurer une veille régulière

Nous avons déjà mentionné l'importance de comprendre la dynamique communautaire autour du développement d'un logiciel (cf. 2.2).

Pour mieux comprendre ce qui se passe dans un projet, la documentation noir sur blanc et les grandes déclarations d'intentions ne sont pas toujours très fiables. C'est par une veille précise, sur une période de temps assez importante qu'on peut réellement comprendre ce qui se passe, et où cela se passe, les points chauds (crises larvées, risques, conflits de personnes ou d'organisations), quels sont les acteurs clé.

Les canaux de communication sont nombreux et parfois inaccessible au premier coup d'œil, répartis entre différents outils, sur différentes forges logicielles : wikis, canaux IRC, mailing-lists, blogs, commentaires dans les blogs, etc.

Parfois, il est judicieux d'aller jusqu'à s'abonner aux flux (RSS ou emails) de notifications des outils de suivi des packages ou des gestionnaires de bugs, pour suivre l'intégralité des échanges, afin de mieux comprendre les problèmes en cours.

4.2.3 Beaucoup d'effort même pour des choses simples

Corriger un bug dans un programme est parfois ardu, mais en général limité dans le temps. Influencer la décision d'une personne bénévole est plus délicat. Synchroniser le travail de nombreuses personnes est souvent difficile [7]. Comment aider les nouveaux entrants à respecter des procédures partiellement implicites ?

Aucun moyen de coercition, de contrainte, n'opérera sur des éléments externes, qui plus est bénévoles, et aucune décision ne pourra être acquise sans un argumentaire sérieux, voire une procédure de vote formel[18].

L'écueil principal des problèmes de management des projets libres est souvent lié aux relations humaines, la planification s'en ressentira souvent. La loi de Pareto (règle des 20/80) se retrouvera souvent vérifiée, donc autant être préparé à ce que 20% de l'effort prenne 80% du délai.

4.2.4 Minimum légal : faire vivre la base de bugs

Une contribution minimale mais essentielle aux projets libres consiste à contribuer à la « gestion des bugs ». C'est une activité que tout utilisateur peut effectuer, et qui facilite la relation avec les développeurs :

- signaler systématiquement les problèmes, en respectant les développeurs, en leurs donnant des éléments factuels,
- tracer le signalement ou la résolution dans les divers bug-trackers (amont, de distribution, internes)[19],
- éliminer les doublons, aider à mieux qualifier les problèmes,
- tester les propositions de résolution, et contribuer des jeux de test, etc.

Cette tâche, réalisée au fil de l'eau ou ponctuellement, est le « minimum légal » que toute organisation utilisatrice devrait s'efforcer de faire systématiquement, comme gage de coopération vis-à-vis des projets libres utilisés.

4.2.5 Accepter des usages sociaux différents

Au-delà des modèles caricaturaux mis en évidence par Raymond dans « Cathédrale et Bazar » [8], il y a de nombreuses variantes, qui changent complètement les modes de fonctionnement, la façon d'accepter les contributions. Comme dans toute société humaine, certains projets libres ont développé des mécanismes sophistiqués de socialisation (constitutions, procédures de vote, mais aussi rites de passage, etc.).

Pénétrer dans une communauté vieille d'une dizaine d'années va avec l'acceptation des us-et-coutumes locaux. Donc avant tout, il convient d'observer les communautés virtuelles auxquelles on pourrait être amener à prendre part, avant de faire une demande, de proposer une contribution, pour s'y prendre correctement.

Depuis peu, certains projets mettent en œuvre des groupes d'accueil et d'entre-aide des nouveaux entrants, et travaillent utilement à l'amélioration des relations humaines au sein du projet : lutte contre l'élitisme forcené, égalité de traitement hommes/femmes, etc.

4.2.6 Communiquer avant tout

Bien-sûr, dans la plupart des projets internationaux, la langue anglaise est importante, et il est parfois nécessaire, lorsque l'implication dans un projet croît, d'acquérir une bonne maîtrise de l'anglais, pour être capable de contribuer utilement dans des discussions, au fil des arguties, des

[18] voir par exemple : *Constitution for the Debian Project* : `http://www.debian.org/devel/constitution.html`

[19] voir à ce sujet : *Faire avancer un bogue où rien ne se passe* (`http://www.ouaza.com/wp/2007/08/04/faire-avancer-un-bogue-ou-rien-ne-se-passe/`)

conflits, sans « violenter » les autres participants anglophones.

Nous recommandons vivement de prendre contact avec les projets libres au plus tôt, avant même de démarrer un développement complémentaire, par exemple. Il n'y a rien de pire pour un projet libre, que de voir des inconnus qui proposent une nouvelle fonctionnalité très intéressante arriver après des mois d'effort, mais de réaliser qu'ils ont développé sans tenir compte des règles de programmation, ou qui ont dupliqué un effort conduit par ailleurs, sans que personne ne l'ait su.

4.2.7 Jouer le jeu selon les règles

Il y a autant de modèles d'organisation, de sous-cultures, de conventions implicites, que de communautés de développement de logiciels libres.

Nous l'avons mentionné plus haut, il vaut mieux adopter les règles d'écriture de code (*coding standards*) pour faciliter l'acceptation des contributions.

De même, le respect des licences, et plus largement des règles du droit d'auteur, est fondamental.

La sensibilisation de l'ensemble des acteurs (stagiaires compris) de l'organisation aux règles du droit d'auteur est donc fondamentale dès le début des projets.

Certains projets demanderont une signature d'une cession de droits formelle (projet GNU, par exemple[20]), qui risque de prendre du temps, et de nécessiter un argumentaire détaillé auprès des services juridiques. Autant entamer les préparatifs le plus tôt possible, et non en fin de projet.

4.2.8 Eviter l' « abandonware » non déclaré

Le logiciel libre peut être un moyen de « recycler » des vieux logiciels inutilisés, mais il vaut mieux l'annoncer clairement.

Si l'on n'est pas prêt à assurer un minimum de support pour les utilisateurs éventuels, autant le leur dire explicitement.

4.2.9 Contribuer un nouveau module au bon endroit

Mieux vaut utiliser les outils et les procédures d'un projet pour publier une contribution.

En terme de visibilité, il est parfois tentant de préférer garder chez soi ce qu'on a réalisé. Mais il faut alors être prêt à synchroniser souvent les informations avec le projet principal. Et en termes de lisibilité pour les utilisateurs potentiels, c'est loin d'être idéal.

Les projets disposent la plupart du temps d'outils de support et l'utilisation de ces outils permet une centralisation des informations. Cette utilisation évite de devoir installer tous les outils de support (forum, gestionnaires de bugs, etc.) dans une infrastructure dédiée. Mieux vaut donc contribuer dans les référentiels de codes, les wikis des projets, en apposant une notice précise qui trace l'origine des contributions.

[20] voir `http://www.gnu.org/licenses/why-assign.html` : *Why the FSF gets copyright assignments from contributors*

4.2.10 Investir dans une forge pour un nouveau logiciel

La disponibilité et la fiabilité des infrastructures de support du processus de développement et d'accueil de la communauté des utilisateurs sont très importantes lors de la publication d'un nouveau logiciel libre complet que l'on souhaite « porter » en propre.

Une attention particulière devra être apportée au choix d'une « forge logicielle » hébergeant le projet[21], en interne ou en externe. Les coûts indirects affectés à la maintenance de cette infrastructure (y compris en personnels) devront être intégrés à l'ensemble.

5 Conclusion

Chaque projet informatique est singulier. Il en va de même des projets libres avec toute la complexité apportée par les relations sociales entre les différents acteurs qui y participent.

Pour les organisations qui réalisaient traditionnellement leurs projets en relation avec des entreprises dans un cadre contractuel, de nombreuses règles du jeu sont chamboulées.

Pour autant, avec de la méthode, et une volonté de jouer le jeu vertueux des contributions au logiciel libre, il est possible de bénéficier du potentiel formidable d'une mutualisation de l'effort et des coûts, et d'une meilleure maîtrise de ses systèmes d'information, tout en valorisant le travail des informaticiens.

Mais c'est un chemin difficile, dont nous n'avons fait qu'esquisser les grands principes. Il reste à adapter ces contraintes à chaque cas particulier, et le rôle du management des projets sera la clé de la réussite.

Bibliographie

[1] Bac C., O. Berger and B. Hamet, 2005, Intégration d'applications logicielles libres pour la réalisation d'une plate-forme de travail collaboratif destinée aux enseignants/chercheurs du GET, in: *Actes du congrès 6èmes journées RESeaux - JRES2005*, p. 151-160.

[2] O. Berger, C. Bac, Plate-forme de travail collaboratif PicoForge (poster), à paraître in *actes du congrès 7èmes journées RESeaux – JRES2007*

[3] Michlmayr, M., Hunt, F., Probert, D. R. (2007). Release Management in Free Software Projects: Practices and Problems. In: Feller, J., Fitzgerald, B., Scacchi, W., Silitti, A. (Eds.), *Open Source Development, Adoption and Innovation*. 295–300.

[4] Perspectives on Free and Open Source Software (2005), edited by J. Feller, B. Fitzgerald, S. Hissam, and K. R. Lakhani (MIT Press)

[5] Bac C., O. Berger, V. Deborde and B. Hamet, 2005, Why and how to contribute to libre software when you integrate them into an in-house application ?,

[21] citons des plate-formes comme *GForge*, *Trac*, *LibreSource* ou encore *PicoForge* [2], qui offrent toutes des fonctions de support au développement collaboratif [9], et peuvent être installées localement.

in: *Proceedings of the First International Conference on Open Source Systems*, p. 113-118.

[6] Le travail des développeurs de logiciels libres. Didier Demazière, Nicolas Jullien, François Horn (novembre 2004) Publié dans *les Cahiers lillois d'économie et de sociologie*, n°46, 2e semestre. pp 171-194.

[7] Michlmayr, M., Robles, G., Gonzalez-Barahona, J. M. (2007). Volunteers in Large Libre Software Projects: A Quantitative Analysis Over Time. In: Sowe, S. K., Stamelos, I. G., Samoladas, I. (Eds.), *Emerging Free and Open Source Software Practices*. 1–24.

[8] Eric Raymond, Bob Young, The Cathedral & the Bazaar, *O'Reilly*, 2001, 208 p. (ISBN 9780596001087)

[9] O. Abdoun, B. Lange et al., Collaborative development environnement : A state of the art. Technical report, QualiPSo project, february 2007