GIT TOOLBOX VERSION 2.0 : LES PRINCIPALES NOUVEAUTES

1) On joue à cache-cache ?

Afin de minimiser le plus possible les accès au serveur de métadonnées Git/GCO, un système de cache a été mis en place sur l'ensemble des machines sur lesquelles la « toolbox » Git a été installée : merou, beaufix (noeuds de login), prolix (noeuds de login), alose/orphie/pagre/rason, et mirage (machine de COMPAS). Ces caches seront mis à jour :

- dès qu'une commande *git_post* sera exécutée ;
- dès qu'une branche sera créée avec *git_branch* ;
- dès que le statut public/privé d'une branche aura été modifié avec *git_public* .

A noter que ce mécanisme de mise à jour des caches a déjà été greffé sur la version actuelle du serveur, de façon transparente pour les utilisateurs.

2) Mais à quoi va donc servir *git_login* ?

Exit l'authentification via *git_login* ! Elle sera désormais basée sur votre nom d'utilisateur système, tous les alias possibles (ex : mrpm602/khatib/elkhatibr) sont – en principe... - pris en compte (le cas échéant, prévenir GCO). Toutefois, si vous êtes déjà authentifié sur une machine en tant qu'utilisateur Git, cette authentification prévaudra tant que vous ne vous êtes pas déconnectés avec *git_logout* .

La commande git_login peut néanmoins toujours être utilisées pour se connecter sous un nom d'utilisateur différent, à condition de connaître son mot de passe.

3) Plus rapide...

Conséquence directe du point (1), la commande *git_view* devient très rapide, les commandes *git_diff* et *git_history* sont dans certains cas plus rapides (autour de 10%), dans d'autres cas bien plus rapides (autour de 70%), tout dépend de l'historique de la routine concernée...

4) ...mais asynchrone :

L'utilisation de *git_post* & *git_branch* se fera désormais de manière (plus ou moins) asynchrone.

- Pour *git_post*, tout sera fait de façon asynchrone. Les opérations de « fetch » du dépôt de l'utilisateur, de mise à jour de la base de métadonnées, et de mise à jour des caches, se feront en « background ». Il est donc fortement recommandé d'attendre le retour par mail de l'exécution de *git_post* avant de faire quoi que ce soit d'autre !
  Cela dit, il a été finalement décidé de pouvoir utiliser *git_post* de façon synchrone, avec l'ajout d'une nouvelle option ***–sync*** . Dans ce cas, la mise à jour du cache de la machine à partir de laquelle a été exécuté *git_post* se fera de façon synchrone, le cache des autres machines sera toujours mis à jour de façon asynchrone.

- Pour *git_branch*, la création de la branche dans le dépôt local se fera bien évidemment de façon synchrone, seule la partie mise à jour de la base de métadonnées et des caches se fera de façon asynchrone.

5) On casse tout et on recommence !

Bonne nouvelle : on reconnait désormais le droit à l'erreur ! En clair, vous avez créé une branche, vous y

avez fait des modifications, vous les avez postées... et vous vous apercevez trop tard qu'il y a des erreurs dans ce que vous avez livré, voire même un nombre tellement important d'erreurs que vous avez juste envie de tout effacer et de tout recommencer ! Et bien ça sera désormais possible, de multiples façons...

- Si la quasi totalité des 150 routines de votre branche est OK, mais que vous devez retoucher quelques routines, vous pourrez utiliser l'option ***–amend*** de *git_commit*, équivalent à l'option du même nom de la commande native Git *commit* . Cette option permet d'amender le commit précédent, même si celui-ci a déjà été posté. Toutefois, il conviendra dans ce cas de poster à nouveau ses modifications avec *git_post*, avec l'option ***–force***, sinon ça ne fonctionnera pas .

- Si vous souhaitez mettre à la poubelle la totalité de votre branche alors que celle-ci a déjà été postée, ou alors revenir à une version antérieure de votre branche, ou encore revenir à une version antérieure pour un certain nombre de fichiers, ça sera possible avec la commande *git_reset*, qui remplace bien avantageusement la commande *git_rmbranch* . Dans le cas d'un « reset » de fichiers, il sera nécessaire de faire un *git_commit* et un *git_post* .

- Et de toute façon, même si vous avez fait un « reset » de votre branche avec la commande Git native *reset*, et bien pas de problème ! Il suffit d'utiliser *git_post* avec l'option ***–force***, et le tour est joué !

- Enfin, il sera possible de regrouper une série de commits faits sur une même branche en utilisant la nouvelle option ***–join*** de *git_commit* .

Petit bémol tout de même : dans le cas où vous auriez (vraiment) envie de procéder à un tel jeu de massacre alors que votre branche a déjà fait l'objet d'un « merge » à GCO, il serait judicieux d'en discuter justement avec GCO avant de le faire...

6) Refonte de l'option *-k* de *git_edit* :

L'option ***-k*** de *git_edit* m'a toujours laissé quelque peu dubitatif.,. Elle a donc été complètement revue : désormais l'utilisation de cette option aura pour conséquence la sauvegarde de la version courante de la routine éditée, mais à l'extérieur du dépôt Git. En complément, l'option ***-r*** a été ajoutée à *git_edit* : on pourra alors restaurer dans une branche une version précédente d'une routine éditée avec l'option ***-k*** dans cette même branche.

7) Disparition de *git_rmbranch* :

Comme cela a déjà été évoqué au point n°4, la commande *git_rmbranch* disparait. Pour supprimer le contenu d'une branche, il faudra désormais passer par *git_reset*. Pour supprimer la branche elle-même, il faudra utiliser *git_branch* avec la nouvelle option ***-d*** .

8) Nouvelles commandes :

- ***git_cache_show :***

Cette commande permet de savoir ce qu'il y a dans le cache (donc dans le dépôt central) pour une branche, un commit, une version, un tag, ou même un utilisateur.

- ***git_editor :***

Cette nouvelle commande permettra de définir ses éditeurs par défaut pour l'édition, le « diff », le merge, parmi la liste des éditeurs disponibles donnée par la commande *git_editor –list –long* . Si un éditeur pourtant disponible sur la machine courante n'apparait pas dans cette liste, il sera possible de l'ajouter (et par la suite de le définir par défaut) avec l'option ***–add*** . A noter que pour utiliser un éditeur sans toutefois en faire la version par défaut, il est possible d'utiliser les variables d'environnement GIT_EDITOR,

GIT_DIFF_EDITOR, et GIT_MERGE_EDITOR .

Pour information, les éditeurs par défaut sont les suivants :
- sur merou : edit=gvim, diff=xcleardiff, merge=xcleardiff ;
- sur beaufix & prolix : edit=gvim, diff=meld, merge=meld ;
- sur alose/orphie/pagre/rason : edit=gvim, diff=gvimdiff, merge=gvimdiff.

- ***git_restore* :**

La nouvelle commande *git_restore* permettra de restaurer une version précédente d'un fichier (choisie dans son historique), ou la version précédente de ce fichier. A ne pas confondre avec l'option ***-r*** de *git_edit* ! Avec *git_edit -r* on restaure une version d'un fichier sauvegardée à l'extérieur du dépôt Git, avec *git_restore* on restaure une version historisée sous Git d'un fichier.

- ***git_sync :***

La commande *git_sync* remplace l'ancienne *git_fetch*. Elle commence par faire un *git fetch*, et elle permet de synchroniser la branche courante (ou une autre branche) avec le dépôt central Git. Par exemple, en lançant simplement *git_sync*, on saura immédiatement si la branche courante...
- ...est à jour par rapport au dépôt central (statut : *is up-to-date*) ;
- ...est en retard par rapport au dépôt central (statut : *update needed*) ;
- ...est en avance par rapport au dépôt central (statut : *post needed*) ;
- ...est désynchronisée par rapport au dépôt central (statut:f*orced updated needed*).

- ***git_fmerge :***

Malgré son nom, cette commande n'a strictement rien à voir avec l'ancien script ClearCase *cc_fmerge* !! Elle va permettre de pallier à deux problèmes inhérents au fonctionnement de la commande Git native *git merge* :
- lorsqu'on merge une branche dans la branche courante, on va en fait merger tous les commits « vus » par la branche à merger qui n'ont pas encore été mergés dans la branche courante, donc pas forcément que les commits effectivement faits sur la branche à merger ;
- on ne peut merger qu'une branche entière, pas une sélection de fichiers modifiés dans cette branche.

La commande *git_fmerge* permet de pallier à ces 2 problèmes, vu que d'une part elle ne prend en considération que les modifications faites dans la branche à merger, avec possibilité de n'en sélectionner qu'une partie. Elle n'utilise donc pas la commande Git native *git merge* : elle va comparer pour chaque fichier à merger les versions locales/distantes et la base commune à ces 2 versions. Si au moins 2 versions sur 3 sont identiques le merge sera automatique (soit on garde la version locale, soit on la remplace par la version distante), dans le cas contraire le merge sera manuel via son éditeur favori. Ce n'est donc pas véritablement un merge au sens Git du terme...

Il est important de signaler que dans la très grande majorité des cas l'utilisation de la commande *git_merge* est préconisée !!

- ***git_export :***

La commande *git_export* permettra d'exporter...
- ...le contenu de la branche courante ;
- ...le contenu complet du dépôt Git ;
- ...une version de source donnée, qui sera complète (ex : CY43), ou incrémentale (ex : CY41T1..CY41T1_op1.15).

Si l'option ***–tgz*** est utilisée, on obtiendra en sortie un fichier archive, et non une arborescence. En outre, il est tout à fait possible d'exporter les sources sur une machine distante.