PROJET CEMT

L'INTRODUCTION DE "PAST AWARENESS" DEDANS LE LOGICIEL BYZANCE

RAPPORT

Auteurs
Thiago Telecken (telecken@inf.ufrgs.br)
Cassiano Bergmann Maciel (cbm@inf.ufrgs.br)
Tharso de Bittencourt Borges (tharso@aton.inf.ufrgs.br)
Manuele Kirsch Pinheiro (manuele@inf.ufrgs.br)

26 Juin 2001

1 Introduction

Ce rapport décrit l'introduction d'un support à "past events awareness" (notification des événements passé) sur Byzance, un logiciel coopératif développé pour le Projet Opéra chez INRIA (Institut National de Recherche en Informatique et en Automatique − France) [OPE 99]. Ce support lá, il s'agit dans "framework", qui s'appelle BW, crée au but de fournit un support flexible à ce type de notification. Ce "framework" a été développer pour Pinheiro chez PPGC/UFRGS (Programa de Pós−Graduação em Computação / Universidade Federal do Rio Grande do Sul − Brésil) [PIN 01a]. Le résultat de cette introduction de "framework" BW dedans le logiciel Byzance, comme on verra après dans ce rapport, c'était une nouvelle version de Byzance capable de stocker et récupérer l'information de "awareness".

Ce rapport est organisé dans le parties suivantes :

- Past events awareness : une petite exposition sur ce qu'on appelle de "past events awareness";
- Framework BW : une exposition sur le fonctionnement et la structure de ce framework;
- Byzance : un résumé de la structure de ce logiciel;
- Structure de communication entre BW et Byzance : ça explique comme la communication entre BW et Byzance marche ;
- Événement de test choisi : ça expose le événement choisi pour fournit l'awareness sur ce premier test;
- Implémentation : ça donne de détails sur l'implémentation de ce test. Ces détails incluent des informations sur les caractéristiques généraux, le protocole de communication, et sur les modifications réalisé dans les codes de Byzance et BW;
- Conclusions : le bilan de ce processus d'inclusion de past events awareness dans le Byzance;

2 Past events awareness

Un très simple concept de "*awareness*" (notification) est le suivant: fournir le contexte des activités seules à travers de la compréhension des activités réalisées par des autres personne [ARA 97]. D'une façon plus complète, nous pouvons affirmer que *awareness* est la connaissance sur les activités du groupe, la connaissance sur ce qui a été produit dans un moment passé, ce qui en train d'être produire maintenant et ce qui pourrait être produire, et aussi la connaissance sur le propre travail et sur le propre groupe [PIN 01b].

En fait, être attentif sur les collègues et sur les activités qu'ils ont faites joue un très important rôle au but d'un travaille fluide et naturel [GUT 99]. Par conséquence, *awareness* devient crucial pour la coopération, puisque percevoir, reconnaître et comprendre les activités réalisées par des autres personne est un requis base à l'interaction humaine et à la communication en général [SOH 99].

Pinheiro et al. [PIN 01b] ont analysé plusieurs mécanismes du support à *awareness* et ils ont identifié des caractéristiques importantes pour ce support là. Ces caractéristiques sont organisées en six questions (*what, when, where, who, how, how much*), chacune identifiant des aspects cruciaux pour fournir awareness dedans les outils coopératifs. Dans ces questions−là, la question "quand" (*when*) est la plus intéressante pour nous. Ce question se réfère à quand sont gérer les événement qui produisent des informations de *awareness.* Nous considérons que les informations de *awareness* sont gérées par des événements, dont occurrent pendant le travail en groupe, et d'ailleurs, le moment où les événements occurrent détermine son importance pour la notification de l'utilisateur. Nous pouvons diviser les événements en quatre moments possibles: dans le passé (ces sont des événements qui ont commencé et finit dans un intervalle qui est déjà passé, et dont ses résultas peut−être ne sont plus valable), dans un passé continu (pour les événements qui ont commencé au passé, mais qui n'ont pas finit, ou dont ses résultats sont encore validés), au présent (les événements qui sont en train d'être réalisent maintenant), ou dans le futur (ça représente les options futures pour le groupe).

Nous appelons "*past events awareness*" ce support à des événement passé, et c'est l'introduction de ce type de support dans le logiciel Byzance que nous décrivons dans ce document.

3 Framework BW

Malgré son importance pour que le travail coopératif soit bien coordonné et structuré, le support à "*past events awareness*" est trop limités dans la majorité des outils coopératifs disponibles [PIN 01a]. Par conséquent, des situations comme l'absence d'une personne du groupe pendant un intervalle ne sont pas traitées. Ces situations−là sont beaucoup fréquentes durant le travail dans un groupe, donc les traiter est très important pour le bon fonctionnement d'un *groupware*. Alors, il faut donner un contexte des activités réalisées pour ceux−ci qui continuent en travaillant et aussi pour ceux−là qui sont en retournant au travail après une période d'absence.

Dans ce but de fournir un mécanisme flexible pour le support à "*past events awareness*" que le "*framework* BW" a été crée. Le BW (*Big Watcher*) est un *framework* pour fournir ce support−là. Il a été conçu d'une façon flexible au but de lui inclure dans plusieurs outils coopératifs [PIN 01a].

Le BW est organisé en quatre paquets (voir Figure 1). Trois (appelés *control*, *storage* et *Interface*) sont indépendants et changent des informations, lesquelles sont décrites dedans le quatrième paquet (appelée *kernel*). Ses informations sont essentiellement des événements, lesquels représentent les activités qui ont été conclut par quelqu'un (quelqu'un qui joue une rôle spécifique) dans le groupe. Ces activités sont registrées au BW pour l'outil coopératif qui l'utilise au but de fournir le contexte du travail aux ses utilisateurs. D'ailleurs, le propre outil coopératif peut aussi créer des classes spécifiques dès les classes proposées au BW. En conséquence, le *framework* BW est capable d'être inclus dedans presque quelconque outil coopératif asynchrone qui besoin d'un mécanisme de support à "past events awareness" [PIN 01a],[PIN 01c].

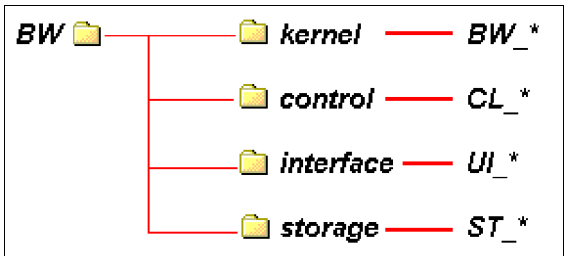

Figure 1 Organisation du framework BW

Le *framework* BW a été implémenté en Java (JDK 1.1.x) et son implémentation a suivi cette même division en paquet (voici Figure 1). Dans le paquet *kernel*, nous trouvons toutes les descriptions des événements et des autres informations utilisées dans BW, comme le groupe, ses membres et ses rôles. En addition, le paquet *control* est responsable par la gestion des informations, ce qui inclut le traitement des événements qui arrive et le traitement des informations à l'utilisateur. Finalement, les paquets *interface* et *storage* sont responsables, respectivement, par l'interface avec l'utilisateur et par le stockage des informations.

Tout le fonctionnement de *framework* BW s'organise en trois étapes : en premier, l'outil coopératif registre les événements dans le *framework* BW. Chaque type d'événement registré dans le BW correspond à un seul type d'activité possible dans l'outil. Ce processus permettra l'outil de choisir auxquelles activités sera fourni les informations d'*awareness*. Depuis, tandis que les activités finissent dans l'outil coopératifs, les événements correspondants sont gérés dans l'outil et envoyés pour celle−ci jusque le *framework*. Ce processus constitue la deuxième étapes de BW et appelé "occurrence". Cette deuxième étape continue jusque la troisième étape, qui est la propre notification d'utilisateur. Au cours de cette dernière étape, la notification, l'utilisateur reçoit les informations d'*awareness* qui lui intéresse. Il reçoit ces informations à travers de son interface, laquelle utilise quelques fonctions décrites dans le paquet Interface et doit aussi être bien intégrée à la propre interface de l'outil coopératif.

4 L’outil coopératif Byzance

Le logiciel Byzance est un éditeur HTML coopératif asynchrone. Il permet aux utilisateurs géographiquement distribués et connectés à Internet éditées conjointement des pages HTML. Son fonctionnement se base sur la fragmentation du document HTML en plusieurs parts, qui s’accorde avec les rôles joués par les auteurs de ce document. Ces rôles définissent les pouvoirs de chaque auteur dans chaque fragment. Par conséquent, les auteurs peuvent travailler sur quelques fragments du document même si sa connexion avec Internet est tombée. [DEC 98a], [DEC 98b]

Cette fragmentation dedans Byzance est résultat du travail de l’API Thot. Thot est utilisé pour le développent des applications, lesquelles manipulent des documents structurés. Il fournit un ensemble des fonctions qui manipulent la structure logique et le contenu des documents. D’ailleurs, Thot permettre aux applications inclurent de fonctions extra à travers de mécanisme de "call back", et aussi des fonctions pour l’interface graphique [DEC 98a], [DEC 98b], [QUI 99a].

Un document en Thot est représenté par sa structure logique, essentiellement hiérarchique, qui est appelée "arbre abstrait". Sur cette structure, il est ajouté des relations supplémentaires, en résultant aussi un hypertexte. Cette structure suit quelques règles, une espèce de type du document, qui aide (ou obligé) ses utilisateurs à produire des documents de ce type−là. En utilisant cette structure, le logiciel Byzance divise ses documents en fragments, lesquels sont distribués dans le *site* de chaque utilisateur. Cette division s’accorde avec les rôles de l’utilisateur, et à cause de ça, seulement un utilisateur du document pourrait changer un fragment à chaque fois. Cette garantie d’un seul utilisateur en écrivant dans un fragment a été obtenue à travers d’un schéma des copies "*master*" et "*slave*" pour chaque fragment. Ce schéma oblige l’existence d’une seule copie *master*, sur le site de seul utilisateur qui peut changer le fragment, et plusieurs copies *slave*, sur les sites des autres utilisateurs du document.

Ce pouvoir de changer (écrire) un fragment est donné par des rôles "manager" et "écrivain". Le rôle "manager" peut lire et changer le fragment, et encore déterminer les rôles des autres coauteurs, tandis que le rôle "écrivain" peut lire et changer le fragment. Les autres rôles possibles sont le rôle "lecteur", lequel peut lire le fragment, et aussi le rôle "nul" qui n’a pas aucun accès sur le fragment, ni lecture, ni écrire. [DEC 98b], [SAL 98].

Chaque coauteur a un ensemble des rôles potentiels et un ensemble de rôles effectives sur chaque fragment du document. Les rôles potentiels sont définis par le manager du document et ils représentent toutes les rôles possibles qui le coauteur peut jouer, tandis que les rôles effectives sont les rôles effectivement joué par le coauteur. Pendant la réalisation du travail, les utilisateurs (coauteurs) peuvent changer ses rôles effectifs en s’accordant avec ses rôles potentiels. Par conséquent, la fragmentation du document puisse changer en faisant une division ou union (*merge*) des fragments, parce qu’il faut adapter le document à la règle d’une seule copie *master* par fragment. À cause de tout ce mécanisme, plusieurs coauteurs peuvent écrire des fragments différents dans un même document durant un même intervalle du temps dedans Byzance. [SAL 98]

Byzance fournit un support à notification, appelé mode de conscience. Ce support permet que ses utilisateurs contrôlent l’environnement d’édition afin de recevoir

les avis sur des modifications dans les fragments. Ce support permet aussi à chaque coauteur règle les notifications sur les modifications réalisées aux autres fragments par ses collèges. Il a été conçu pour faciliter son intégration avec le processus d'édition, simplifier sa manipulation, réfléchir l'état de la coopération et être efficace à la mise en jour des fragments. [SAL 98]

Ce système de notification permet aux coauteurs régler ses propres niveaux de notification en choisissant entre les mises en jour automatique, ouvert ou verrouillé. Dans le premier choix, les modifications, réalisées par un collège au fragment, sont réfléchies dans l'interface de l'utilisateur lorsqu'elles sont disponibles. Ce choix−là n'équivaut pas à l'édition synchrone, puisqu'il pourrait avoir un retard entre ces modifications et sa divulgation pour les autres coauteurs. Dans le cas d'une mise en jour ouvert, l'utilisateur verra les modifications au fragment seulement après les demander volontairement. Par fin, dans une mise en jour verrouillé, l'utilisateur ne veut pas être averti sur les modifications sur le fragment, car il ne voudrait pas les considérer pendant son travail.

Le choix sur le système de notification et les rôles joués par l'utilisateur, tous les deux sont montrés graphiquement par des icônes intégrées à l'interface devant chaque fragment. Quelques modifications, soit dans système de notification, soit dans les rôles, sont représentées par des icônes différentes dont la transition entre un icône et autre icône est défini par un automate finit.[SAL 98]

Une conséquence de ce système de notification utilisé dedans Byzance est que nous n'avons pas un support à "*past events awareness*". Nous ne pouvons pas savoir, par exemple, si un fragment a été changé plusieurs fois. Nous ne pouvons connaître que la dernière modification sur le fragment, mais nous ne pouvons pas comment connaître les modifications réalisées avant cette dernière. Car Byzance a cette limitation, nous avons décidé d'introduire dedans Byzance un module de "*past events awareness*", en utilisant le *framework* BW, sur lequel nous avons déjà parlé.

5 Structure de communication entre BW et Byzance

Nous avons introduit au logiciel Byzance un module pour fournir le support à "*past events awareness*". Ce module a été conçu pour la manipulation des plusieurs types des événements réalisés dedans Byzance, donc des activités, au but de fournir un support pour la notification de ces activités déjà conclut. Ce support−là est fournit par le *framework* BW, que nous avons déjà discuté avant.

Le module que nous avons introduit est le seul responsable pour faire toutes les communications nécessaires avec le *framework* BW. Il se divise en deux parties: un client et un serveur d'*awareness.* Le client travail toujours directement avec Byzance, tandis que le serveur reçoit les descriptions des événements du client et leur passe au BW. L'architecture utilisée par la communication entre ces deux parties, client et serveur, est décrite dans la figure 2 au−dessous.

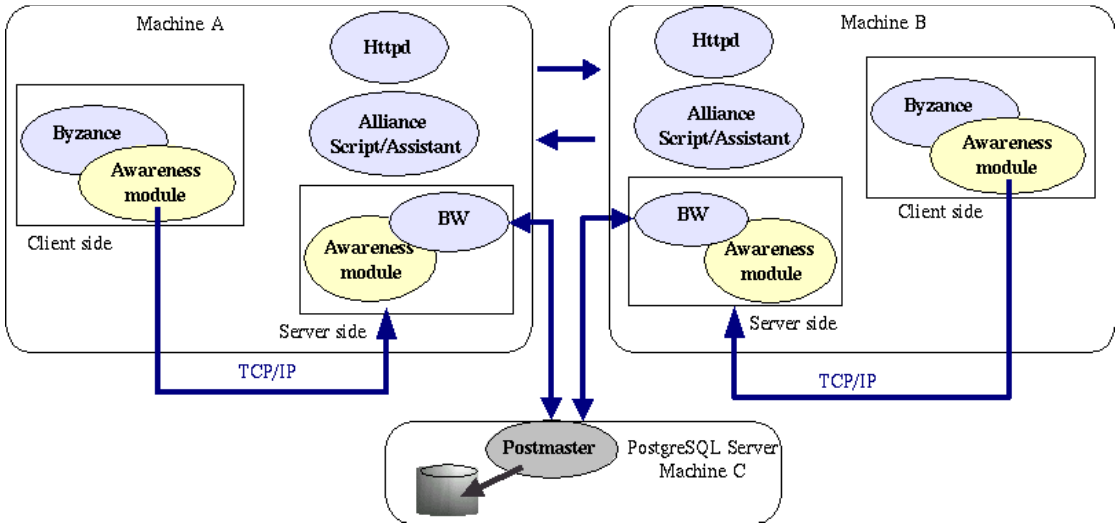

Figure 2 Architecture utilisée

Nous choisissons cette architecture parce que le Byzance et le BW ont été construits en différents langages, et cette architecture s'est montrée plus simple pour faire l'intégration entre les deux. Le Byzance est un logiciel écrit en C avec une base de donnée (cette base est fournie par la bibliothèque Thot), tandis que le *framework* BW a été écrit en Java. D'ailleurs, dans cette implémentation−ci, nous avons utilisé une base de donné centralisée (cette base est fournit par le logiciel PostgreSQL), criée dans une autre utilisation de *framework* BW. Alors, nous avons fait cette option pour le modèle client−serveur, en utilisant TCP/IP, dans l'architecture du système.

Le stockage et la lecture des informations dans ce système−ci suivent les deux stages suivants:

(1) Lorsqu'un utilisateur aura conclu une activité (par conséquent, un événement est crié), le Byzance réalisera ses tâches habituelles, et après, il enverra les informations d'*awareness* pour le client TCP/IP dedans le module de *awareness* (*Awareness*

Module). Ce client, lequel est écrit en langage C, va traiter et envoyer ces informations pour le côté serveur du module. Ce serveur, écrit en Java, se localise dans la même machine du client et inclut le *framework* BW. Il repasse au BW les informations reçues, lesquelles sont stockées par BW dans une base de donnée PostgreSQL distante;

(2) Lorsqu'un utilisateur demande un rapport des informations d'*awareness*, le Byzance passe les informations de cette demande−là au côté client du module d'*awareness*. Ce client passe la demande et ses informations pour le côté serveur du module, lequel, en utilisant le *framework* BW, lit les informations d'*awareness* de la base de données. Ces informations lues sont, alors, repassées au client TCP/IP, qui montre ces informations à l'utilisateur.

Après la définition de l'architecture, nous avons choisi un événement pour la première implémentation de ce système. Ce choix est décrit au−dessous.

6 L’événement du test

Nous avons choisi comme premier événement pour fournir des informations de "past events awareness" le "Save Document". Cet événement représente l’activité d’enregistrer un document. Cette activité est importante, car elle nous informe que les fragments, lesquels sont édités par un utilisateur sont stables.

Un événement "Save Document" est crée quand un utilisateur qui a la permission d’écrire (rôles manager ou écrivain) demande l’enregistrement du fragment après le change. Il est obligatoire qui le fragment a été changé par l’utilisateur, parce que, au contraire, l’événement ne sera pas créé. Par exemple, si l’utilisateur ne rien change et déclic sur le "save document" du Byzance, l’événement ne sera pas créé. Ces événements sont liés à chaque fragment. Ça veut dire que chaque fragment du document aurait ses propres informations d’awareness. Lorsqu’un nouveau fragment est identifié, il est registré dans Byzance, et alors chaque fois qui ce nouveau fragment est changé et enregistré, les événements correspondants seront gérés. Mais si deux fragments sont unis (ça est possible quand une tache, qui définit le fragment, est effacée), seulement les événements qui sont liés au premier de ces fragments resteront.

Les informations d’*awareness* sur un fragment, que nous appelons souvent "le rapport d’*awareness*", contiennent le suivant :

- Un titre avec le nom du document et un numéro qui identifie le fragment;
- une séquence de lignes qui montre des informations précises sur le fragment :
  - la date quand le fragment a été enregistré;
  - la machine où il a été enregistré;
  - le nom de l’utilisateur qui a l’enregistré;

Avec ce premier événement, nous ne voudrons que tester l’inclusion d’un support à "past events *awareness*". Il y a des autres événements aussi intéressants que nous pourrons inclure après la réussite de ce test−là. Par exemple, nous pourrons, avec ce support, montrer quels utilisateurs ont déjà vu la dernière version d’un fragment, ou montrer combien de versions d’un fragment ont été gérées dès que l’utilisateur a vu le fragment, etc.

7 L’implémentation

7.1 Caractéristiques Générales

L’implémentation de ce support à *awareness* dedans Byzance a suivi l’architecture que nous avons discuté dans la section 5 de ce rapport. Cette implémentation a été limitée, dans ce premier moment, à un seul événement, que nous avons discuté dans la dernière section de ce document. Dans cette section, nous irons donner de détails plus précis sur cette implémentation, en commençant par l’interface et la forme de communication utilisées.

Pour présenter les informations d’*awareness* aux utilisateurs, nous avons conçu un petit morceau d’interface dédié à cette tâche, en observant des suggestions disponibles en Pinheiro [PIN 01a]. Un suggestion de Pinheiro est l’intégration de l’interface du support à *awareness* et la propre interface de l’outil coopératif, au but de ne pas confondre l’utilisateur. Nous avons créé, en suivant cette suggestion, ce morceau d’interface complètement intégré à interface du Byzance. L’utilisateur, pour lire le rapport sur un fragment, doit utiliser les menus, exactement comme les autres fonctions du Byzance. Il touche sur l’option "Awareness" du menu "Cooperation" (le support à *awareness* est une partie très importante pour que la coopération réussisse). Comment résultat, le système montrera le rapport d’*awareness* sur le fragment actif dans une fenêtre très simple, laquelle a été construite avec la bibliothèque Thot. Finalement, le fragment actif est celui où le curseur est positionné, et l’utilisateur peut facilement changer le fragment actif en changent la position du curseur pour une autre fragment.

Pour avoir les informations d’*awareness,* ces informations doivent être envoyées et lues par le côté client du module d’awareness jusqu’au le côté serveur de ce même module. La communication entre le deux, client et serveur, est réalisée à travers des *sockets* de la communication TCP/IP, en utilisant des *stream* pour le flux des informations. Ces informations sont composées par strings. Ces strings respectent un protocole fixe, que nous irons discuter en détail après dans ce rapport. Il y a dans ces strings des caractères qui délimitent des blocs d’information et qui sont défini dans le même protocole. Grâce à ses caractères, ces blocs d’information sont bien compris et enregistrés.

Le premier passe de cette communication est la création, et postérieur identification, du *socket.* Ce processus utilise des fonctions patron des bibliothèques C/UNIX (*write,* pour envoyer les informations, et *read,* pour sa lecture). Depuis la création réussie de ce *socket*, nous pouvons demander la connexion avec le serveur, en utilisant une porte définie par avant (où le serveur attend les connexions des clients). Lorsque le serveur accepte cette connexion, un chemin pour le flux des informations continues est défini, et la communication en deux sens (client → serveur, et serveur → client) est établie. Par conséquent, nous pouvons utiliser le même *socket* pour envoyer et recevoir des informations.

Nous avons utilisé, pour recevoir les informations (*read*), des fonctions, non bloquant, donc l’insuccès ne bloque pas le processus et produit de nouveaux essais. Néanmoins, nous avons utilisé des fonctions bloquant pour envoyer des informations

(*write*). Ça veut dire que le processus arrêtera jusqu'au la réponse d'autre côté arrive. Par conséquent, nous avons garanti que le processus va attendre le message. Avec ces fonctions bloquant, nous pourrions empêcher l'utilisateur de travailler, si le temps d'attendre utilisé est long. Mais, comme le client et le serveur sont locaux, le temps d'attente ne sera pas trop long.

Du côté du serveur au module d'*awareness*, quand ce serveur a reçu les informations du client, il les transforme d'un ensemble des octets jusqu'au un ensemble des objets "événement", lesquels seront manipulés et stockés par BW.

Le choix pour tout le deux, le TCP/IP et les *stream*, donne une meilleure fiabilité dans la livraison des information, parce qu'ils facilitent la détection des erreurs et des paquets perdus pendant la communication. D'ailleurs, ils supportent l'envoi séquentiel des paquets et ils garantissent la livraison en ordre des paquets. C'est facile d'imaginer le problème causé par la livraison hors d'ordre, par exemple, si le message du fin arrive avant les informations. Dans ce cas−là, nous pourrions avoir des informations incomplètes ou nous pourrions perdre des informations intéressant.

7.2 Le protocole de communication

Pour réaliser cette communication décrit dans la section au−dessus, nous avons proposé un protocole très simple. La liste complète des commandes de ce protocole est dans le tableau 1.

Tableau 1 Commandes du protocole

Command	***Fonctionnalité***
BEGIN	Il commence toutes les transmissions
BEGINEVENT	Il indique le début d'un événement
ENDEVENT	Il indique le bout du dernier événement
CANCELEVENT	Il annule le dernier événement qui avait commencé
NEWUSER	Il additionne un nouvel utilisateur au groupe ou il change les informations sur un utilisateur
SETUSERPROFILE	Il change l'ensemble de profils (*profiles*) de l'utilisateur. Ces profils sont une structure du *framework* BW qui lui informe quels sont les événements intéressants pour l'utilisateur.
GETEVENTS	Il demande les événements, qui ont été produits dans un intervalle passé, et qui sont intéressants pour l'utilisateur.
BYE	Il finalise les transmission
OK	C'est la réponse du serveur à une commande qui a réussi
NOK	C'est la réponse du serveur à une commande qui n'a pas réussi

Toutes ces commandes ont une syntaxe similaire, sauf la commande BYE. Cette syntaxe est la suivante :

COMMANDE informations

Les informations, qui sont envoyées ci−joint, dépendent de la commande, et elles ont sa propre syntaxe générale (sauf les informations pour la commande SETUSERPROFILE) :

< attribut1="valeur" & attribut2="valeur" & ... & attributN="valeur" >

Ces attributs sont les mêmes attributs définis dans les classes "BW_Member", "BW_Event", "BW_Profile" et "BW_AwarenessProfile" du *framework* BW. Ces classes décrivent, respectivement, les utilisateurs, les événements, la généralisation des profils, et les profils des utilisateurs. La syntaxe complète de ces commandes est au−dessous.

Commande :
BEGIN < type="4" & login="login" & name="nom" & machine="nom de la machine" >
Informations :
La commande BEGIN passe les informations basique sur l'utilisateur actif.
Réponse du serveur :
OK → Si tout va bien.
NOK → Si il y a des problèmes au serveur.

Commande :
BEGINEVENT < type="n" & name="nom du événement" & description="description du événement" & details="détails sur l'événement" >
Informations :
La commande BEGINEVENT passe les informations sur l'événement qui commence. Les attributs "description" et "details" peuvent être vides (exemple : details="").
Réponse du serveur :
OK → Si tout va bien.
NOK → Si il y a des problèmes au serveur.

Commande :
ENDEVENT < type="n" & name="nom du événement" & description="description du événement" & details="détails sur l'événement" >
Informations :
La commande ENDEVENT passe quelques informations sur l'événement qui finit. Les attributs "description" et "details" peuvent être vides (exemple : details="").
Réponse du serveur :
OK → Si tout va bien.
NOK → Si il y a des problèmes au serveur.

Commande :
CANCELEVENT < type="n" & name="nom du événement" & description="description du événement" & details="détails sur l'événement" >
Informations :
La commande CANCELEVENT passe les informations sur l'événement qui a été annulé. Les attributs "description" et "details" peuvent être vides (exemple : details="")**.**

Réponse du serveur :
OK → Si tout va bien.
NOK → Si il y a des problèmes au serveur.

Commande :
NEWUSER < type="4" & login="login" & name="nom de l'utilisateur" & homepage="page web personnelle de l'utilisateur" & mail="courrier électronique de l'utilisateur" & machine="nom de la machine utilisée par l'utilisateur" & paper1="première rôle de l'utilisateur" & paper2="deuxième rôle de l'utilisateur" & ... & paperN="rôle N de l'utilisateur">
Informations :
La commande NEWUSER passe les informations sur un utilisateur. Les attributs "homepage", "mail" et "name" peuvent être vides (exemple : homepage="").
Observation :
Le numéro "4" est une suggestion d'un identification unique pour le type d'information "utilisateur".
Réponse du serveur :
OK → Si tout va bien.
NOK → Si il y a des problèmes au serveur.

Commande :
SETUSERPROFILE < type="n" & membertype="identification du type de l'utilisateur" & memberlogin="login de l'utilisateur" & membermachine="nom de la machine utilisée par l'utilisateur" & event1="type=n; name=nom" & ... & eventN="type=n; name=nom" & interval="jour:moins:an:heur:minute; jour:moins:an:heur:minute">
Informations :
La commande SETUSERPROFILE passe quelques informations sur un utilisateur et les informations sur son profil. Il y a une petite différence dans la structure de l'attribut "eventX". Cet attribut inclut le type de l'événement et le nom de l'événement, tout le deux séparés par un ";" et sans les guillemets. Dans l'attribut "interval", les informations, initiales et finales, de date et heure de l'événement sont aussi séparées par un ";", pendant que les informations individuelles dedans ceux−ci (jour, moins, an, heure et minute) sont séparés par ":" chaque un.
Réponse du serveur :
OK → Si tout va bien.
NOK → Si il y a des problèmes au serveur.

Commande :
GETEVENTS < type="n" & login="login de l'utilisateur" & machine="nom de la machine utilisée par l'utilisateur" >
Informations :
La commande GETEVENTS passe quelques informations sur un utilisateur, lequel voudrait les informations d'*awareness* de son intéresse.
Réponse du serveur :
La réponse du serveur sera les événements passées, qui s'accordant avec les profils de l'utilisateur. Cette réponse a la syntaxe suivante :
<details="détails sur l'événement" & type="n" & objid="n" & name="nom de l'événement" & description="description de l'événement" & interval="jour:moins:an:heure:minute; jour:moins:an:heure:minute" & memberlogin="login de l'utilisateur" & membername="nom de l'utilisateur" & memberhomepage="URL" & membermail="courrier elétronique " & membermachine="maquina" >
<details="détails sur l'événement" & type="n" & objid="n" & name="nom de l'événement" & description="description de l'événement" & interval="jour:moins:an:heure:minute; jour:moins:an:heure:minute" & memberlogin="login de l'utilisateur" & membername="nom de l'utilisateur" & memberhomepage="URL" & membermail="courrier elétronique " & membermachine="maquina" >

<details="détails sur l'événement" & type="n" & objid="n" & name="nom de l'événement" & description="description de l'événement" & interval="jour:moins:an:heure:minute; jour:moins:an:heure:minute" & memberlogin="login de l'utilisateur" & membername="nom de l'utilisateur" & memberhomepage="URL" & membermail="courrier elétronique " & membermachine="maquina" >

Commande :
BYE
Informations :
La commande BYE termine les transmissions entre client et serveur

7.3 Modifications dans Byzance

Pour inclure le module d'*awareness* dedans Byzance, nous avons changé quelques parties de la source du Byzance. Les parties changées ne représentent pas une grande portion de la source total, un résultat très positif, spécialement pour le framework BW, qui objective la flexibilité. Ces modifications réalisées dedans Byzance sont détaillées au−dessous.

(1)Schéma EDITOR.A : nous avons inséré dans ce schéma l'option "Awareness" dans le menu "Coopération", en causant l'insertion de cette même option dans l'aspect visuel du Byzance.

```
view:1 ALL_Cooperation separator;
view:1 ALL_Cooperation button :  Aw_Save_Doc                −> AwSaveDoc;
```

(2)Fichier managerSheet.c: nous avons additionné dans ce fichier−là la fonction "AWGetFragmentID", laquelle retourne l'identification du fragment actuel.

```
int AWGetFragmentID ()
{
int fragmentIndex;
...
return(aFragId (fragmentIndex).g_FragId);
}
```

(3)Fichier EDITORactions.c : dans ce fichier, nous avons inséré la fonction "AwSaveDoc", laquelle montrera le rapport d'*awareness*. L'insertion de cette fonction a aussi causé l'insertion de quelques bibliothèques ("stdio.h", "base.h" et "document.h").

```
/* Telecken: include for I/O file:w */
#include <stdio.h>
/* Telecken: include for get username,url,etc */
#include "base.h"
#include "document.h"
...
#ifdef __STDC__
void AwSaveDoc (Document document, View view)
#else /* __STDC__*/
void AwSaveDoc (document, view)
    Document document;
    View view;
#endif /* __STDC__*/
{ ... }
```

(4)Fichier documentOpen.c : dans ce fichier, nous avons inséré à la fin de la fonction "DocumentSave" un appel à la fonction "AwSendSaveDoc". Cet appel va se produire après l'nregistrement du document et il va actionner le *socket* de communication avec BW.

```
#ifdef __STDC__
T_bool    DocumentSave (Document document)
#else                                   /* __STDC__ */
T_bool    DocumentSave (document)
Document  document;
#endif                                  /* __STDC__ */
{
...
     if (newProducedDocumentVersion){
          signalModifiedDocToRemoteSites
(curDocIndex);
          AwSendSaveDoc (document);
     }
...
}
```

Tous les sources spécifiques du côté client du module de *awareness* est limité à un seul fichier (*awareness.h*), dans de répertoire "opera/byzance/awareness". Dans ce fichier, nous trouvons toutes les fonctions pour le traitement du protocole et d'autres fonctions de communication. Les fonctions les plus importantes sont décrites dans le tableau 2.

Tableau 2 Fonctions du côté client du module d'awareness

Fonction	***Description***
AwSendSaveDocument AwCallSaveDoc	Ces fonctions prennent des informations importantes du Byzance et leur repassent à fonction AwExecSaveeDocument.
AwExecSaveDocument	Cette fonction gère la communication TCP/IP entre le client et le serveur quand un événement "Save Document" se produit.
AwExecCallSaveDoc	Cette fonction gère la communication entre le client et le serveur quand un utilisateur demande son rapport d'*awareness.*
MountBegin	Cette fonction crée le *string* pour la commande BEGIN du protocole.
MountBeginEvent	Cette fonction crée le *string* pour la commande BEGINEVENT du protocole.
MountEndEvent	Cette fonction crée le *string* pour la commande ENDEVENT du protocole.
MountGetEvent	Cette fonction crée le *string* pour la commande GETEVENT du protocole.
TreatResp	Cette fonction traite la réponse envoyée par le serveur.
AwSendString	Cette fonction envoie les *strings* du protocole du client jusqu'au le serveur.

Finalement, nous avons aussi changé la procédure de compilation du Byzance, en changeant le fichier "makefile" (dans le répertoire "/opera/byzance/byzance"), au but d'introduire ces modifications et le nouveau module d'*awarenes.* Nous avons introduit le répertoire "opera/byzance/awareness" dans la liste des répertoires des bibliothèques, et avons introduit aussi une règle pour la compilation du module d'*awareness*. Ces changes sont détaillés au−dessous :

Fichier: opera/byzance/makefile

```
INCL_DIR = −I. −Ibyzance ... −Iawareness  −I/home/byzance/opera/Amaya/LINUX−ELF/libwww
...
$(OBJ_DIR)/Awareness.o: awareness/Awareness.c
        $(CC) $(PROF) $(FLAGS_CC) $(INCL_DIR) −c awareness/Awareness.c
        mv Awareness.o $(OBJ_DIR)
        $(COMMANDS)
...
ALL_OBJ =  $(OBJ_DIR)/AHTBridge.o $(OBJ_DIR)/AHTEvntrg.o \
        ...
        $(OBJ_DIR)/ANNOTnotif.o $(OBJ_DIR)/ANNOTutils.o $(OBJ_DIR)/Awareness.o
...
ANNOT_OBJS = $(OBJ_DIR)/ANNOTlink.o \
        ...
        $(OBJ_DIR)/Awareness.o \
        $(OBJ_DIR)/ANNOTnotif.o
```

Puisque nous avons introduit ces modifications dans Byzance, nous avons déterminé deux nouveaux flux d'information pour les fonctions d'*awareness*. Le premier se produit au moment de la création d'un événement. Quand un événement se produit, le Byzance active des fonctions (par exemple, la fonction AwSendSaveDocument), lesquelles capturent les informations nécessaires du Byzance et les repassent pour les fonctions gérantes (par exemple, la fonction AwExecSaveDocument). Ces fonctions gérantes contrôlent l'envoi de ces informations capturées du client jusqu'au le serveur, où se trouve le *framework* BW. Ce contrôle inclut la création des *strings* du protocole (fonctions MountBeginEvent, MountEndEvent, etc.) et l'envoi ou la réception de ces *strings*.

Le deuxième flux se produira quand un utilisateur demande les informations d'*awareness*. Le Byzance, au ce moment−là, active une fonction qui capture des informations nécessaires à ce processus, et leur repasse aux fonctions gérantes. Ces fonctions, à ses tournes, demandent la création du *string* du protocole pour la commande GETEVENT (fonction MountGetEvent), et l'envoi de cette commande. Quand le serveur reçut cette commande, il passe la demande au *framework* BW, lequel la traite, en lisant des informations d'*awareness* intéressantes pour l'utilisateur. Ces informations seront envoyées du serveur pour le client, lequel va leur traiter et montrer à l'utilisateur, en utilisant la fonction "AwSaveDoc".

7.4 Modifications dans BW

Le *framework* BW a été projeté pour être convenu à l'outil coopératif où il est intégré, sans obliger des grandes modifications dans les sources de l'outil. Au but de maintenir cette indépendance, le BW travaille avec l'idée d'un médiateur, un intermédiaire entre l'outil coopératif et le *framework* BW. Comme ça, l'implémentation du BW est cachée de l'outil, auquel il fournit support à notification, et le propre outil ne besoin pas de plusieurs modifications.

Nous avons construit, dans ce travail, ce médiateur entre le Byzance et le BW en langage Java. Ce médiateur représente le côté serveur du module d'*awarenes,* lequel recevra les informations (les événements, les informations sur l'utilisateur) du côté client dans Byzance, leur traitera et passera le résultat pour le *framework* BW. Nous avons basé ce médiateur sur une autre implémentation de médiateur (voyez Pinheiro [PIN 01a]), sur laquelle nous avons fait des modifications nécessaires pour l'utiliser dans nos cas ici. Cette section décrit ces modifications réalisées sur cette implémentation que nous avons utilisée comme base pour la construction du côté serveur du module d'*awareness*.

La première modification réalisée dans l'implémentation base était dans la classe responsable pour le remplissage initial de la base de donné, et pas dans le propre médiateur. Ce remplissage inclut dans la base des informations basiques pour le bon fonctionnement du BW, comme le registre de l'événement du test, et les rôles, l'utilisateur et le groupe modèle. Nous avons adapté ces informations de l'implémentation antérieure pour la cas du Byzance.

Après ces modifications pour le correct remplissage de la base de donné, nous avons remodelé le médiateur existant dès l'application antérieure pour les nécessités au

Byzance. En premier lieu, nous avons remplacé les définitions des événements de la vieille application pour la définition de l'événement test ("SAVEDOCUMENT") pour notre application. Nous avons aussi effacé quelques constantes et changé les valeurs des quelques autres.

En outre, nous avons changé quelques méthodes de la classe. Nous avons changé les méthodes "makePrototypes" et "getPrototypes", qui objectivent la création et la capture, respectivement, des prototypes des événements, afin qu'ils traitent correctement les attributs de l'événement test. Nous avons changé aussi la méthode "_start", qui fait l'initiation du médiateur, au but de faire cette initiation bien adaptée au cas du Byzance.

Nous avons aussi introduit des nouvelles méthodes, lesquelles sont essentiellement destinées au traitement de la communication pour *sockets* avec l'outil coopératif. Ces méthodes incluent ceux qui traitent les informations qui arrivent du client, comme, par exemple, les méthodes "AwtratarBeginEvent", "AwtratarEndEvent" et "AwtratarGetEvent", lesquelles traitent les commandes BEGINEVENT, ENDEVENT et GETEVENT, respectivement. Ils incluent aussi des méthodes auxiliaires, comme les méthodes "AWBuf" et "AWWriteBuf", qui aident dans le traitement de la commande GETEVENT. Les principaux méthodes inclus sont décrits dans le tableau 3.

Tableau 3 Les plus importantes méthodes introduites dans le médiateur

Méthodes	***Description***
AWtratarBeginEvent	Cette méthode reçoit le string du *socket* (la commande BEGINEVENT), la transforme en un objet BW_Event (événement), et signale le commencement d'un événement
AWtratarEndEvent	Cette méthode reçoit le string du *socket* (la commande ENDEVENT), la transforme en un objet BW_Event (événement), et signale la fin de cet événement
AWtratarGetEvent	Cette méthode reçoit le string du *socket* (la commande GETEVENT), et demande les événements passés au *framework* BW
AWBuf	Cette méthode aide la méthode AwtratarGetEvent, en organisant les informations des événements lus
AWWriteBuf	Cette méthode aide la méthode AwtratarGetEvent, en envoyant les informations des événements

Comme le médiateur joue aussi, dans le cas du Byzance, le rôle de serveur d'*awareness* (ou le côté serveur de ce module, comme nous avons aussi appelé), il doit se déclencher lui−même. À cause de ça, nous avons inclus la méthode "main" dans le médiateur, en transformant ceci d'une simple classe en une application Java. Cette application exécutera avant même qui l'outil coopératif, et n'ira pas s'arrêter quand cet outil arrête ses activités.

Une conséquence de ce comportement est la possibilité d'avoir plusieurs événements en se produisant au même temps (parfois, en plusieurs clients, dans la même machine). À cause de ça, nous avons introduit l'idée de *multi−threading*, afin de traiter chaque réquisition dans un nouveau flux de contrôle spécifique, et laisser le flux principal (serveur) beaucoup plus disponible pour les autres réquisitions.

Alors, quand le côté client du module d'*awareness* dans Byzance envoie des informations à travers du canal de communication TCP/IP, celles−ci sont reçues pour le médiateur par un de ses flux de contrôle, et elles sont traitées par des fonctions spécifiques (par exemple, la fonction "AwtratarBeginEvent"). En général, ces fonctions spécifiques divisent le string reçu par le *socket* de communication, et elles traitent chaque information individuellement. Ce traitement inclus la transformation de ces informations en objectes dont le *framework* BW espère.

D'une façon similaire, quand une demande, pour le rapport d'*awareness* (commande GETEVENT), arrive, le flux de contrôle qui l'a reçu appellera la fonction de traitement ("AwtratarGetEvent"). Cette fonction−là demande au *framework* BW les événements passés, et pour chaque événement récupéré, elle monte, avec l'aide des fonctions auxiliaires ("AWBuf" et "AWWriteBuf"), les strings de réponse et les envoie pour le client. Quand tous ces événements sont reçus par le côté client du module d'*awareness*, le client les présente à l'utilisateur dans une fenêtre d'interface similaire à l'interface utilisée dans Byzance.

L'interface avec l'utilisateur est un aspect très important pour le bon fonctionnement du support à *awareness*. Dans le *framework* BW, cette interface est définie d'une façon abstraite par les interfaces Java "UI_GUIElement" et "UI_GUIEvent". Ceux−ci sont implémentés par le propre médiateur, parce que la présentation graphique n'est pas définie par le serveur d'*awareness*, lequel dialogue avec le *framework* BW. Cette présentation est définie dans le côté client, ci−joint le logiciel Byzance, au but d'être bien intégré à celui−ci.

8 Conclusions

Nous avons trouvé quelques résultats intéressants pendant la réalisation de ce travail−ci. Les premiers résultats trouvés sont la capacité de l'expansion du Byzance et la flexibilité du *framework* BW. Ce framework−là a prouvé sa flexibilité (un parmi ses principaux objectifs), en acceptant l'intégration avec un outil complexe comme Byzance. Ceci a démontré, aussi, une capacité d'expansion, malgré sa grande complexité.

Cette capacité d'expansion du Byzance pourrait nous permettre d'améliorer le support à "*past events awareness*", que nous avons introduit. Nous pourrions, par exemple, introduire d'autres événements plus intéressants, ou améliorer l'interface de présentation des informations d'*awareness.*

Par fin, cette capacité démontre qu'il y a la possibilité de construire encore une nouvelle version de Byzance avec un support à *awareness* plus effectif, mais, principalement, une nouvelle version avec une structure du *workflow*. Cette nouvelle version, que nous pourrions appeler "Byzance_AW" (Byzance Awareness & Workflow), pourrait être utiliser pour la conception coopératif des *hyperdocuments* pour l'enseignement à distance.

9 Bibliographie

[ARA 97] Araújo, R.M.; Dias, M.S.; Borges, M.R.S. Suporte por Computador ao Desenvolvimento Cooperativo de Software: Classificação e Propostas. In: XI Simpósio Brasileiro de Engenharia de Software. **Anais**. Fortaleza, CE, outubro, 1997. P. 299−314

[DEC 98a] DECOUCHANT, Dominique; SALCEDO, Manuel Romero; QUINT, Vincent. **Structured Cooperative Authoring on the World−Wide Web.** Disponible: <http://www.w3.org/pub/Conferences/WWW/Papers/91/>. (set.1998).

[DEC 98b] DECOUCHANT, Dominique; SALCEDO, Manuel Romero; SERRANO, M. **Principes de Conception d'une Application d'Édition Coopérative de Documents sur Internet**. Disponible: <ftp://ftp.inrialpes.fr/pub/opera/publications/IHM97.ps.Z> (set.1998).

[GUT 99] Gutwin, C.; Greenberg, S. **A framework of awareness for small groups in shared−workspace groupware**. Techinical Report 99−1, Departament of Computer Science, University of Saskatchewan, Canadá. Disponible: <http://www.cpcs.ucalgary.ca/grouplab/papers/1999/99−AwarenessTheory/html/theory−tr99−1.html> (set., 1999).

[OPE 99] OPERA GROUP. **Byzance**. Disponble: <http://www.inrialpes.fr/opera/Byzance.fr.html> (jun.1999).

[PIN 01a] Pinheiro, Manuele Kirsch. **Mecanismo de suporte à percepção em ambientes cooperativos**. Dissertação de mestrado. Porto Alegre, Brasil : PPGC/UFRGS, 2001. Disponible: <http://www.inf.ufrgs.br/~manuele/teseFinalPDF.zip> (jun.,2001).

[PIN 01b] Pinheiro, Manuele Kirsch; Lima, José Valdeni; Borges, Marcos R.S. Awareness em Sistemas de Groupware. In IV Jornadas Iberoamericanas de Ingenieria de Requisitos y Ambientes de Software. **Memorias**. Santo Domingo,. Costa Rica : CIT, 2001.

[PIN 01c] Pinheiro, Manuele Kirsch; Lima, José Valdeni. **An Adaptable Framework for Past Events Awareness Support**. Disponible: <http://www.inf.ufrgs.br/~manuele/ArtBW.zip> (jun.,2001).

[QUI 99a] QUINT, Vincent;VATTON, Irène**. Thot Tool Kit API**. Disponble: <http://www.inrialpes.fr/Thot/Doc/APIman.html> (fev.1999).

[SAL 98] SALCEDO, Manuel Romero**. Alliance sur l'Internet**: support pour l'édition coopérative des documents struturés sur un réseau à grande distance**.** Grenouble, France: INPG, 1998.

[SOH 99] Sohlenkamp, M. **Supporting group awareness in multi−user enviroment through perceptualization**. Dissertation. Fachbereich Mathematik−Informatik der Universität − Gesamthochschule − Paderborn. Fevereiro, 1998. Disponible: <http://orgwis.gmd.edu/projects/POLITeam/poliawac/ms−diss/> (jul., 1999).