Programmation d'une Application Distribuée Résistante Aux Pannes avec l'environnement GARF [1]

Karim R. Mazouni, Benoît Garbinato, Rachid Guerraoui
Laboratoire de Systèmes d'Exploitation
Département d'Informatique
Ecole Polytechnique Fédérale de Lausanne
Lausanne, Suisse
e-mail: garf@lse.epfl.ch

Résumé: L'environnement GARF a pour objectif de simplifier la programmation d'applications distribuées et résistantes aux pannes. La programmation avec GARF est modulaire: deux niveaux orthogonaux d'objets expriment respectivement les aspects fonctionnels et les aspects comportementaux d'une application. Au niveau *fonctionnel*, les objets *données* décrivent les fonctionnalités de l'application dans un contexte centralisé, sans concurrence, sans persistance, et sans pannes. Au niveau *comportemental*, les objets données sont associés à des *encapsulateurs* et à des *messagers* qui implémentent les comportements de ces objets données concernant la concurrence, la persistance et la résistance aux pannes dans un contexte distribué. Cet article illustre la démarche de programmation avec GARF sur un exemple concret: le Gestionnaire Réparti d'Agendas Dupliqués.
Mots clés: distribution, résistance aux pannes, encapsulateurs, messagers, programmation orientée-objet.

1 Introduction

La programmation d'applications distribuées résistantes aux pannes est rendue difficile par les problèmes inhérents à la distribution, la concurrence, la persistance et à la résistance aux pannes. L'environnement de Génération Automatique d'applications Résistantes aux Fautes (GARF) [10] a pour objectif de simplifier la programmation de telles applications en ajoutant "automatiquement" le comportement distribué et résistant aux pannes désiré aux fonctionnalités d'une application. Cet article présente l'environnement GARF et illustre la démarche de programmation d'une application avec cet environnement. L'idée fondamentale de GARF est de programmer séparément les aspects *fonctionnels* (spécifiques à chaque application) d'une part, et les aspects *comportementaux* ("factorisables" d'une application à l'autre) d'autre part. Cette approche a pour avantage d'être modulaire: la modification du comportement d'un composant d'une application est effectuée sans altérer la programmation des fonctionnalités de ce composant et vice-versa. A titre d'exemple, la duplication d'un composant particulier (pour le rendre résistant aux pannes) est réalisée sans modifier la programmation de ses fonctionnalités. La programmation avec GARF est basée sur un modèle objet. Le concept d'objet favorise la modularité et implémente naturellement le paradigme client-serveur, concept clé de la programmation d'applications distribuées. Le concept de classe d'objets permet de définir des comportements réutilisables d'une application à l'autre. Enfin, le concept d'héritage permet simplement de définir de nouveaux comportements à partir de comportements existants.

La partie 2 décrit l'environnement GARF en présentant les deux niveaux orthogonaux du modèle objet de GARF et certaines classes de messagers et d'encapsulateurs fournies par GARF. Ces classes sont utilisées dans les parties 3 et 4 qui décrivent la démarche de programmation avec GARF sur un exemple d'application distribuée résistante aux pannes: le Gestionnaire Réparti d'Agendas Dupliqués (GRAD). La partie 3 décrit la programmation au niveau fonctionnel et la partie 4 décrit la programmation au niveau comportemental. La partie 5 termine cet article en comparant l'approche GARF aux projets similaires décrits dans la littérature et en présentant les directions des travaux futurs .

2 L'environnement GARF

Cette partie donne un aperçu des composantes de l'environnement au GARF, présente les objets comportementaux et décrit les classes d'encapsulateurs et de messagers utilisées dans les parties 3 et 4.

2.1 Architecture générale

L'environnement GARF s'appuie sur deux composantes principales: un langage de programmation orientée objets et une plateforme logicielle de services distribués. Le langage permet de définir des objets *données* et des objets *comportementaux*. Ces derniers utilisent les services de la plateforme à travers un interface composé d'objets *système*. Les objets de l'environnement GARF sont regroupés en classes organisées en une arborescence d'héritage dont la racine est la classe *GARFObject*. Toutes les classes de cet arborescence sont construites à partir des classes de bases fournies par le langage orienté objet. Plus précisément, cette hiérarchie des classes se subdivise en deux sous-arborescences disjointes: la

[1] Recherche financée par le Fonds National Suisse pour la Recherche Scientifique (projet Mesures Spéciales, no. contrat: 5003-034344.)

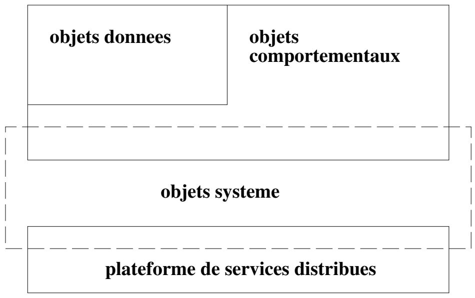

Figure 1: Architecture générale de l'environnement GARF

hiérarchie des objets données (racine *DataObject*), la hiérarchie des objets comportementaux (racine *BehaviorObject*). La hiérarchie des objets données décrit les classes utilisées pour programmer les fonctionnalités des applications développées avec GARF. La hiérarchie des objets comportementaux permet de décrire les aspects comportementaux des applications tels que la concurrence, la persistance et la résistance aux pannes. La plateforme de services distribués implémente des

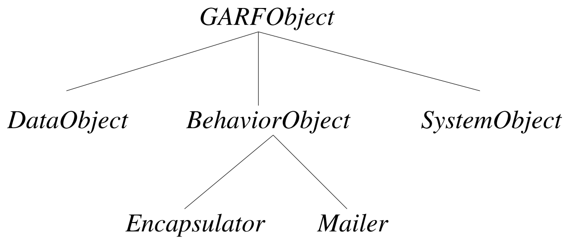

Figure 2: Arborescence des objets GARF

services système que les objets comportementaux utilisent à travers les objets système.

2.2 Objets données et objets comportementaux

Les objets données expriment les fonctionnalités d'une application dans un contexte centralisé, sans concurrence, sans persistance et sans pannes. Un objet donnée est un objet au sens classique de la programmation orientée objet [4, 15]. L'information que représente un objet donnée ne peut être manipulée qu'à travers les méthodes définies par la classe de l'objet. Un objet donnée c communique avec un autre objet donnée s en invoquant une méthode de s. Une invocation de s par c consiste en deux phases: l'envoi par c d'une requête destinée à s et la réception par c d'un résultat envoyé par s. Après avoir envoyé sa requête, l'objet client c attend le résultat de l'invocation que l'objet serveur s est censé lui renvoyer. La requête (resp. le résultat) peut être vu comme un message. Ainsi, on considére que les objets données communiquent à l'aide d'envois de messages *point-à-point synchrones*.

Le comportement d'un objet données dans un contexte distribué concerne la concurrence, la persistance et/ou la résistance aux pannes. Ce comportement est exprimé par un *encapsulateur* et par une classe de *messagers* associés à l'objet données. Un encapsulateur *E(o)* est un objet comportemental qui implémente le comportement de l'objet donnée o. Le comportement de o défini par *E(o)*, complète *sans la modifier* la programmation des fonctionnalités de o. Le comportement de o est exprimé en termes de pré-actions et de post-actions exécutés par *E(o)* respectivement avant et après chaque invocation de o. Pour réaliser ce comportement, *E(o)* intercepte tous les messages dont o est l'émetteur ou le destinataire. A chaque interception de message, *E(o)* effectue des actions (ex: contrôle des accès concurrents, journalisation, envois de messages de manière asynchrone, ...) modélisant le comportement de o. Pour l'objet donnée o, l'encapsulateur *E(o)* est complétement transparent.

Un messager *Mi(s)* est un objet comportemental chargé d'acheminer les messages liés à la i-ième invocation de s. A chaque invocation de s, un messager est instancié à partir de la classe de messagers *M(s)* associée à s. La classe *M(s)* définit la politique d'invocation de l'objet s. Cette politique tient compte notamment de la sémantique de la méthode invoquée (ex: lecture, écriture) et du comportement de s (ex: duplication) pour choisir la primitive de communication adéquate avec laquelle la requête et le résultat sont transmis. Par exemple, si s est un objet dupliqué et que la méthode invoquée est une écriture, le messager va utiliser une primitive de diffusion pour communiquer la requête à tous les duplicas de l'objet s. On assure ainsi la cohérence des duplicas de l'objet s.

Dans un contexte distribué, plusieurs machines (ou *noeuds*) dialoguent à travers un réseau de lignes de communication. Le placement des objets données sur les différents noeuds se fait à travers celui de leurs encapsulateurs respectifs. Quand

E(o) est créé sur un noeud donné, *o* est créé automatiquement sur ce noeud. Par conséquent, un objet donnée et son encapsulateur résident toujours sur le même noeud. Par contre, un messager *Mi(s)* réside sur le noeud du client *c* qui a effectué la *i*-ième invocation de *s*. Le messager représente, le temps de l'invocation, l'objet *o* sur le noeud du client *c*: il joue le rôle d'un *proxy* [6]. La localisation du messager sur le noeud *c* s'avère particulièrement utile lorsque *s* est dupliqué: la localisation de *s* (et par conséquent sa duplication) est complètement transparente au client *c*. En effet, le messager donne l'illusion à *c* qu'il communique avec un seul objet *s* alors que celui-ci est en réalité constitué de plusieurs duplicas résidant chacun sur un noeud distinct.

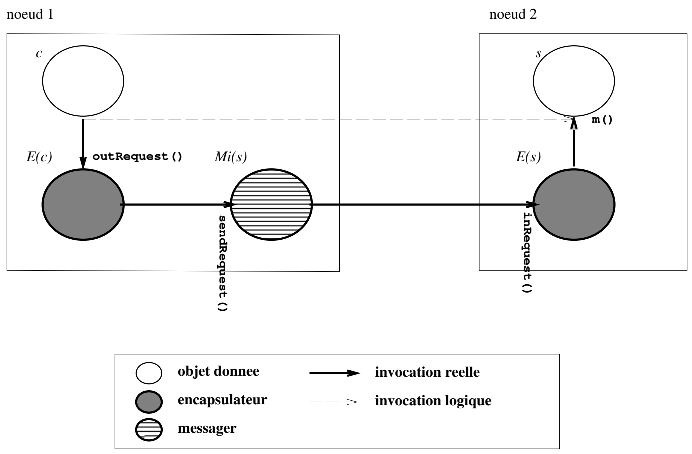

Figure 3: Communication client-serveur dans GARF

La figure 3 illustre la communication client-serveur dans GARF au cours d'une invocation. Un objet client *c* résidant sur le noeud 1 invoque une méthode `m()` sur un objet serveur *s* résidant sur le noeud 2. La requête est interceptée par le système[2] qui instancie le messager *Mi(s)* sur le noeud 1 et qui passe (invocation de `outRequest()`) la requête et le nom du messager à l'encapsulateur de *c*, *E(c)*. Celui-ci effectue éventuellement des actions comportementales (p.ex: sauvegarde de l'état de *c* en mémoire stable) avant de transmettre (invocation de `sendRequest()`) la requête à *Mi(s)*. Le messager utilise une primitive de communication distante pour transmettre (invocation de `inRequest()`) la requête à *E(s)*, l'encapsulateur de *s*. Cet encapsulateur effectue éventuellement des pré-actions avant d'invoquer effectivement la méthode `m()` sur l'objet *s*. Lorsque l'exécution de `m()` se termine, *E(s)* effectue éventuellement des post-actions avant de renvoyer (retour de `inRequest()`) le résultat à *Mi(s)*. Ce dernier transmet (retour de `sendRequest()`) le résultat à *E(c)* avant de disparaître, l'invocation étant terminée. L'encapsulateur *E(c)* effectue éventuellement des actions comportementales avant de renvoyer (retour de `outRequest()`) le résultat à l'objet client *c*.

2.3 Hiérarchie des objets comportementaux

Cette hiérarchie se subdivise en deux arborescences distinctes: la hiérarchie des encapsulateurs (racine *Encapsulator*) et la hiérarchie des messagers (racine *Mailer*). La classe *Encapsulator* définit les opérations `inRequest()` et `outRequest()`. La classe *Mailer* définit l'opération `sendRequest()`. La classe *DataObject* définit l'opération `garfNew()` exécutée à chaque fois qu'un objet donnée est instancié. Cette opération peut être redéfinie dans des sous-classes pour associer un encapsulateur et une classe de messagers à un objet donnée.
Les classes *Encapsulator* et *Mailer* n'implémentent aucun aspect comportemental si ce n'est les communications distantes. Pour ce faire, elles se basent sur une couche RPC fournie par la plateforme de services. Un comportement particulier est obtenu en redéfinissant `inRequest()` et `outRequest()` dans une sous-classe de *Encapsulator*, et/ou en redéfinissant `sendRequest()` dans une sous-classe de *Mailer*. Ce comportement s'appuye sur des services existants de la plateforme. Par exemple, le comportement persistant est basé sur la mémoire stable implémentée au niveau de la plateforme de services. Pour chaque aspect comportemental (concurrence, persistance et résistance aux pannes), l'environnement GARF fournit un ensemble de classes d'encapsulateurs et de messagers à la disposition du programmeur. Quelques exemples de classes utilisées dans la suite de l'article (voir 4), sont décrites ci-dessous.

2.3.1 Concurrence

Comme les objets données peuvent être invoqués de façon concurrente, il est indispensable de contrôler les accès concurrents pour garantir la cohérence des objets. L'accès à un objet en *exclusion mutuelle* permet de résoudre simplement

[2] Il s'agit en fait du run-time de l'environnement GARF.

le problème des invocations concurrentes. L'exclusion mutuelle garantit qu'une seule invocation est faite sur un objet à un instant donné. Si une invocation est en cours, les autres invocations sont mises en attente dans une file. Lorsque l'invocation en cours se termine, l'invocation en tête de la file est soumise à l'objet. Ce mécanisme est implémenté par la classe d'encapsulateurs *Mutex*. Chaque instance de cette classe maintient une file d'invocations destinées à l'objet donnée auquel l'instance est associée. Ce type de contrôle de concurrence très restrictif est raffiné par la classe *ReadsWrite* qui implémente le modèle lecteurs/rédacteur. Un encapsulateur de cette classe n'autorise que les invocations concurrentes de méthodes de lecture, les invocations de méthodes d'écriture devant se faire en exclusion mutuelle. Au moment de l'association de l'encapsulateur à un objet donnée, le programmeur doit préciser (p.ex. à l'aide de paramètres de la méthode de création) quelles méthodes correspondent à des lectures et lesquelles correspondent à des écritures.

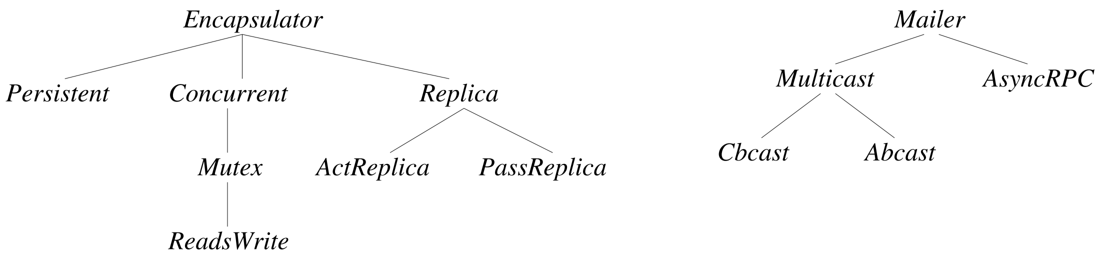

Figure 4: Hiérarchies des encapsulateurs et des messagers

2.3.2 Persistance

Généralement, un objet est dit persistant [13] si il ne disparaît pas avant que le programmeur ne le détruise explicitement. Un objet persistant survit d'une session à l'autre. Implémenter des objets persistants nécessite des noms globaux persistants pour désigner les objets et de la mémoire stable pour stocker l'état des objets. La classe d'encapsulateurs *Persistent* définit deux méthodes `store()` et `retrieve()` qui permettent respectivement de stocker et de retrouver l'état d'un objet donnée en mémoire stable. Lorsqu'un objet donné est créé puis associé à un encapsulateur (dérivé) de la classe *Persistent*, cet encapsulateur vérifie (à l'aide du nom global de l'objet) que cet objet n'a pas été déjà créé au cours d'une session antérieure. Le cas échéant, l'état est rechargé à partir de la mémoire stable. Après chaque invocation d'un objet donnée persistant, son encapsulateur stocke l'état de l'objet en mémoire stable.

2.3.3 Résistance aux pannes

La duplication est un moyen efficace de masquer les pannes et d'implémenter la résistance aux pannes. En présence de canaux de communication fiables[3] et de noeuds affectés uniquement par des pannes franches (*crash failures*) [5], un objet est accessible tant qu'il reste au moins une copie de cet objet sur un noeud opérationnel. Si un objet est dupliqué $k+1$ fois sur des noeuds distincts, il est dit résistant à k pannes. La classe d'encapsulateurs *Replica* et ses sous-classes implémentent la duplication dans GARF. Les instances de ces classes constituent des groupes d'encapsulateurs: à chaque objet dupliqué correspond un groupe d'encapsulateurs particulier. Chaque membre du groupe est responsable d'une copie de l'objet donnée. La composition du groupe est géré par un objet système particulier appelé *GroupManager*. Celui-ci constitue l'interface d'un gestionnaire de groupe distribué implémenté par la plateforme de services distribués. Lorsqu'un membre du groupe disparaît (suite à la panne de son noeud), le *GroupManager* le retire de la liste des encapsulateurs appartenant au groupe. Plusieurs politiques de duplication peuvent être implémentées en définissant un schéma de communication particulier entre les membres d'un même groupe.

Duplication active Chaque membre du groupe d'encapsulateurs reçoit la requête, la transmet à sa copie de l'objet donnée et renvoie une réponse au messager responsable de l'invocation [16]. Les membres du groupe effectuent exactement le même traitement sans communiquer entre eux. Pour dupliquer un objet donnée avec cette politique, il faut lui associer un encapsulateur de la classe *ActReplica* (en précisant le taux de duplication et la localisation des duplicas) et la classe de messagers *Abcast*. Les messagers de cette classe diffuse la requête au groupe des encapsulateurs concernés en utilisant une primitive de diffusion fiable et totalement ordonnée [3]. Cette primitive garantit que tous les membres reçoivent la requête ou aucun ne la reçoit, et que toutes les requêtes sont traitées dans le même ordre par tous les membres du groupe d'encapsulateurs.

[3] Un canal fiable garantit (en utilisant la retransmission) qu'un message arrive à son destinataire au bout d'un temps fini pour autant que ni l'émetteur, ni le destinataire ne tombent en panne entre temps.

Duplication passive Un seul membre du groupe d'encapsulateurs (appelé le primaire) reçoit la requête, la transmet à sa copie de l'objet donnée et renvoie une réponse au messager responsable de l'invocation [16]. Régulièrement, le primaire envoie son état aux autres membres du groupe qui servent de copies de secours. Si le primaire tombe en panne, les membres du groupe désignent l'un d'entre eux pour être le nouveau primaire. Pour dupliquer un objet donnée avec cette politique, il faut lui associer un encapsulateur de la classe *PassReplica* (en précisant le taux de duplication et la localisation des duplicas) et la classe de messagers *Mailer*. Cette classe de messagers implémente une communication point-à-point qui correspond exactement à l'interaction entre le messager et le primaire dans un schéma de duplication passive.

3 Programmation au niveau fonctionnel

Cette section et la suivante décrivent la démarche de programmation d'une application distribuée résistante aux pannes en utilisant l'environnement GARF. La démarche est appliquée à un exemple concret: le Gestionnaire Réparti d'Agendas Dupliqués (GRAD), une application pilote permettant d'expérimenter les concepts de GARF. La programmation d'une application avec GARF se fait au niveau fonctionnel, puis au niveau comportemental.

3.1 Programmation des objets données

Au niveau fonctionnel, le programmeur se place dans un contexte centralisé, sans concurrence, sans persistance et sans pannes. Le point de départ est une description des fonctionnalités de l'application. Cette description peut être exprimée aussi bien de manière informelle en langage naturel que de manière précise à l'aide d'un langage de spécification. Un exemple de description informelle des fonctionnalités du GRAD est:
"Le GRAD gère un agenda pour chaque utilisateur. Un agenda est constitué par la liste des rendez-vous auxquels le propriétaire de l'agenda est censé participer. Un rendez-vous comporte une période (date, durée) et une liste de participants. Les fonctionnalités offertes à un utilisateur lors d'une session avec le GRAD sont:

- *consulter la liste des propriétaires d'agendas;*
- *créer un agenda (lors de la première session d'un nouvel utilisateur avec le GRAD);*
- *consulter les rendez-vous inscrits dans son agenda;*
- *planifier/annuler un rendez-vous avec d'autres propriétaires d'agendas;*
- *effacer son agenda (lors de la dernière session d'un utilisateur du GRAD)."*

A partir d'une telle description, le programmeur définit des classes d'objets données en suivant une approche orientée objets classique [4, 15]. Les cinq classes d'objets données définies pour le GRAD sont: *DiaryIndex*, *Diary*, *Meeting*, *Period* et *DiaryService*. La classe *DiaryIndex* implémente les opérations applicables à l'annuaire des propriétaires d'agendas. L'annuaire associe à chaque nom d'utilisateur le nom de son agenda. Les méthodes définies par *DiaryIndex* permettent de consulter la liste des propriétaires d'agenda, d'ajouter/retirer un couple (propriétaire, agenda) à cette liste, d'obtenir le nom de l'agenda d'un utilisateur donné. La classe *Diary* implémente les opérations applicables à un agenda. Pour chaque utilisateur enregistré dans l'annuaire, un agenda est créé. Les méthodes définies par *Diary* sont: consulter la liste des rendez-vous contenus dans un agenda, ajouter/retirer un rendez-vous d'un agenda, consulter/mettre à jour les plages libres d'un agenda (opérations utilisées lors de la prise d'un rendez-vous). La classe *Meeting* implémente les opérations applicables à un rendez-vous. A chaque fois qu'un nouveau rendez-vous est fixé entre plusieurs propriétaires d'agendas, une instance de cette classe est créée. Les méthodes définies par *Meeting* permettent de consulter la période d'un rendez-vous, de consulter/mettre à jour la liste des participants au rendez-vous. La classe *Period* implémente une méthode permettant de consulter une période. Une période se compose d'une date et d'une durée. Il y en a une par rendez-vous fixé. La classe *DiaryService* implémente les opérations directement accessibles à un utilisateur du GRAD en se basant sur les classes décrites précédemment. A chaque fois qu'un utilisateur commence une session avec l'application, une instance de cette classe est créée. Les méthodes définies par *DiaryService* sont: enregistrer/effacer un utilisateur, débuter/terminer une session, lister les propriétaires d'agendas, fixer un rendez-vous avec plusieurs propriétaires d'agendas, permettre à un utilisateur de se désister d'un rendez-vous de son choix.

3.2 Graphes de dépendances client-serveur

La dépendance entre un objet client c et un objet serveur s s'exprime par la relation $uses$ définie par: $c\ uses\ s$ si c est suceptible d'invoquer s. Cette relation découle de la relation plus générale $\overline{uses}$ qui lie deux classes d'objets données $\overline{c}$ et $\overline{s}$ et qui s'exprime par: $\overline{c}\ \overline{uses}\ \overline{s}$ si une méthode définie par $\overline{c}$ contient une invocation d'une méthode définie par $\overline{s}$.

Si $\overline{c}\ \overline{uses}\ \overline{s}$ il peut exister, au cours d'une exécution de l'application, un objet c et un objet s tels que $c\ uses\ s$. Par conséquent, pour déterminer les dépendances client-serveur entre les objets données de son application, le programmeur construit un graphe orienté[4] dont les noeuds représentent les classes d'objets données et dont les arcs représentent les dépendances.

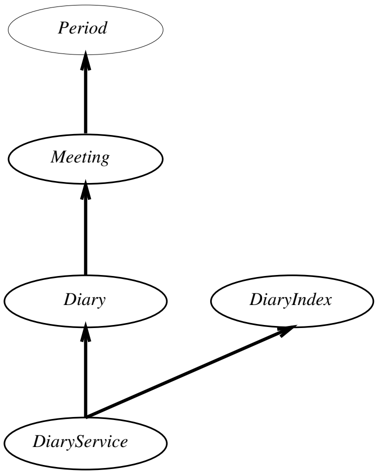

Figure 5: Exemple de graphe de la relation $\overline{uses}$

La figure 5 présente l'allure du graphe dans le cas des cinq classes du GRAD. Le petit nombre de classes rend le graphe très simple. A partir de ce graphe, le programmeur affine les dépendances en construisant un graphe analogue pour la relation *uses*. Ce deuxième graphe s'obtient à partir du premier en "instanciant" un ou plusieurs noeuds à partir de chaque noeud du premier graphe. Par exemple, sur la figure 6, les noeuds *diary1* et *diary2* sont obtenus à partir du noeud *Diary* de la figure 5. Cela signifie qu'il peut exister à un instant donné plusieurs agendas au sein du GRA. Par contre, comme l'annuaire est unique, le noeud *diary index* représente l'unique instance de la classe *DiaryIndex*.

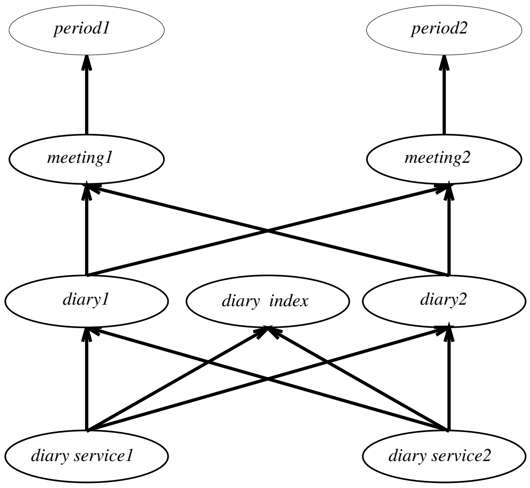

Figure 6: Exemple de graphe de la relation $uses$

Les arcs de ce graphe sont construits à partir de ceux du graphe précédent en tenant compte de la sémantique de partage des objets de l'application. Si un objet peut être serveur de plusieurs objets (objet *partagé*), plusieurs arcs arrivent au noeud correspondant. Par exemple, les agendas *diary1* et *diary2* peuvent être partagés par deux instances de l'interface utilisateur que représentent les noeuds *diary service1* et *diary service2*. Par contre, la période d'un rendez-vous est un objet *privé* du rendez-vous qui en est le client: les noeuds *period1* (resp. *period2*) ne reçoit qu'un seul arc provenant de *meeting1* (resp. *meeting2*).
Un objet privé a la particularité de n'être utilisé que par un seul objet (appelé propriétaire). Cette particularité a des conséquences importantes sur le comportement d'un tel objet. Par définition, un objet privé n'est jamais invoqué de façon concurrente par plusieurs objets[5]. Généralement, un objet privé est créé sur le même noeud que son propriétaire afin de

[4]Ce graphe a pu déjà être construit lors de la conception orientée objets au niveau fonctionnel [4].
[5]Si le propriétaire fait des accès concurrents à l'objet privé, le contrôle des accès concurrents doit faire partie du comportement du propriétaire.

tirer partie de la localité des accès à l'objet privé. Ainsi, l'objet privé ne reçoit que des invocations locales (provenant d'un objet dans le même espace d'addressage) et n'a pas besoin d'être associé à des objets comportementaux gérant l'invocation distante. Un objet privé est rendu persistant si son propriétaire est persistant: les objets comportementaux responsables de la persistance du propriétaire considèrent un objet privé comme faisant partie intégrante de l'état du propriétaire qu'ils sauvegardent en mémoire stable. Cette remarque s'applique aussi à la résistance aux pannes.
Le comportement d'un objet *non* privé (i.e un objet partagé ou un objet "autonome" comme l'interface utilisateur) doit être exprimé explicitement par des objets comportementaux adéquats. Par conséquent, lors de la programmation au niveau comportemental, on ne définit que le comportement des objets non privés.

4 Programmation au niveau comportemental

Au niveau comportemental, le programmeur considère dans un contexte distribué, la concurrence, la persistance et la résistance aux pannes. Il analyse le comportement désiré pour son application afin de choisir les encapsulateurs et les classes de messagers adéquats. En utilisant la sémantique de l'application et les graphes construits précédemment, le programmeur décrit le comportement de son application en abordant successivement la concurrence, la persistance et la résistance aux pannes. Pour chaque aspect, un même comportement peut être défini pour toute une classe d'objets donnés et/ou des comportements différents peuvent être associés à des instances particulières. Ceci vient du fait que chaque objet donnée a un encapsulateur et une classe de messagers qui lui sont propres et qui peuvent être choisis de manière indépendante par le programmeur.

4.1 Concurrence

La concurrence concerne les objets partagés, suceptibles d'être invoqués simultanément par plusieurs objets clients. Les fonctionnalités de l'application programmées au niveau fonctionnel ne garantissent pas la cohérence d'un objet partagé. Le programmeur décide au niveau comportemental, de la politique de contrôle des accès concurrents qu'il doit associer à chaque objet partagé. A partir du graphe de la relation *uses*, le programmeur détermine l'ensemble des objets partagés de l'application. La figure 6 montre que les objets partagés du GRAD sont les rendez-vous, les agendas et l'annuaire. L'annuaire et les agendas peuvent être consultés ou modifiés en adoptant un contrôle de concurrence classique: les lectures peuvent être concurrentes mais les écritures doivent se faire en exclusion mutuelle.
Pour un agenda, le contrôle des accès concurrents est rendu plus complexe par la sémantique de l'opération permettant de fixer un rendez-vous. Cette opération met en jeu une instance de la classe *DiaryService* et plusieurs agendas. La prise de rendez-vous se fait selon l'algorithme suivant: l'interface utilisateur *diary service*, interroge tour à tour les agendas des participants au rendez-vous afin que ceux-ci lui communiquent leurs plages de temps libre. A partir de ces informations, *diary service* détermine (par intersection des plages libres) la période du rendez-vous en considérant une information de durée éventuellement fournie par l'utilisateur. Une fois que la période du rendez-vous est fixée, le *diary service* crée un objet rendez-vous composé de cette période et de la liste des participants. Puis, le rendez-vous est inscrit dans les agendas des participants.
Si plusieurs prises de rendez-vous concernant les mêmes agendas sont concurrentes, le contenu des agendas risquent d'être incohérents même si un contrôle de concurrence du type "lecture concurrentes, écriture exclusive" est effectué. Le problème provient du fait qu'un rendez-vous est construit en invoquant chaque agenda à deux reprises: une lecture pour consulter les plages libres et une écriture pour écrire le rendez-vous. Si deux opérations de prise de rendez-vous sont entrelacées (ex: lecture, lecture, écriture, écriture), une même plage de temps libre peut se voir affectée aux deux rendez-vous! Par conséquent, la prise de rendez-vous doit se faire de façon indivisible: tant que le rendez-vous n'est pas inscrit dans les agendas de tous les participants, aucune prise de rendez-vous concernant ces agendas ne doit débuter. On peut voir la prise de rendez-vous comme une transaction.

4.2 Persistance

Le programmeur utilise la sémantique de l'application et les dépendances client-serveur afin de déterminer quels objets données doivent être rendus persistants. Le comportement persistant se "propage" le long des dépendances client-serveur: si c est persistant alors l'objet s tel que $c\ uses\ s$ est persistant. Pour rechercher les objets persistants, il suffit d'abord de trouver, en partant des racines (les noeuds n'ayant pas de prédécesseurs) du graphe de la relation *uses*, des objets dont la persistance est "triviale". Dès qu'un tel objet est trouvé, tous les noeuds contenus dans le sous-graphe qui en découle doivent être rendus persistants. La recherche s'arrête lorsque l'on a parcouru tout le graphe.
Dans le contexte du GRAD, Les propriétaires d'agendas travaillent avec l'application de façon discontinue au cours de sessions de travail espacées dans le temps. D'une session à l'autre, le contenu d'un agenda doit être conservé tant que son propriétaire ne l'efface pas explicitement. Par conséquent, chaque agenda est "trivialement" persistant. Il en découle que tous les rendez-vous sont persistants. L'annuaire répertorie tous les agendas de l'application et il est impossible de

retrouver un agenda sans consulter l'annuaire. Par conséquent l'annuaire est lui aussi "trivialement" persistant. Chaque instance de l'interface utilisateur (classe *DiaryService*) n'existe que le temps d'une session et n'est donc pas persistante.

4.3 Résistance aux pannes

Afin de définir le comportement résistant aux pannes désiré pour une application, le programmeur inventorie les pannes potentielles et évalue leurs fréquences d'apparition, puis il analyse l'effet de ces pannes sur le fonctionnement de l'application. A partir de cette analyse de risques, le programmeur décide des conditions (en terme d'occurences de panne de chaque type) dans lesquelles l'application doit continuer à fonctionner. Cette décision est un compromis entre le coût engendré par les effets des pannes sur le fonctionnement de l'application et le coût nécessaire pour protéger l'application contre ces effets. L'analyse de risques n'est pas intégrée dans l'environnement GARF dont l'objectif est de rendre une application résistante à des pannes *déjà* identifiées au préalable. Cette résistance aux pannes est implémentée en associant un encapsulateur et une classe de messagers adéquats aux objets concernés.

Dans un premier temps, les pannes de type *crash failure* [5] sont les seules considérées lors de la programmation d'une application avec GARF. L'effet d'une telle panne est l'arrêt brutal d'un noeud et la perte des objets qui y résidaient. L'analyse des effets d'une panne sur une application consiste à envisager la perte subite d'un ou plusieurs objets et à déterminer les conséquences de cette perte sur le fonctionnement ultérieur de l'application. Suivant l'importance des objets perdus, l'application continue à fonctionner en mode dégradé ou bien ne fonctionne plus du tout. Le programmeur utilise les dépendances client-serveur pour mesurer cette importance.

Concrètement, le graphe de la figure 6 montre que l'annuaire *diary index* est crucial pour le fonctionnement du GRAD. La perte de cet objet interdit toute nouvelle session avec un utilisateur, car l'application est alors incapable de retrouver son agenda. Toutes les sessions en cours sont paralysées: chaque utilisateur ne peut faire que consulter son agenda. Toute tentative d'accéder à l'annuaire engendre une exception (non traitée par *diary service*) qui provoque la terminaison prématurée de la session. La perte d'un agenda introduit une incohérence: l'annuaire répertorie un agenda qui n'existe plus. Cette perte peut provoquer une erreur lors de l'invocation de l'agenda perdu. Par exemple, lorsqu'un utilisateur désire fixer un rendez-vous avec le propriétaire de l'agenda perdu, l'objet *diary service* correspondant à cet utilisateur invoque une méthode sur un objet inexistant. Cette erreur provoque la terminaison de la session demandant le rendez-vous. La perte d'un rendez-vous a le même effet que celle d'un agenda. Le risque d'erreur concerne toutes les sessions des participants au rendez-vous perdu. Par exemple, si un participant désire annuler le rendez-vous perdu, son *diary service* va invoquer une méthode sur le rendez-vous pour mettre à jour la liste des participants et provoquer une exception. La perte d'un objet interface utilisateur correspond à l'arrêt prématuré d'une session. Si au moment de la panne, cet objet reportait un rendez-vous dans les agendas des participants, il est possible que ce rendez-vous n'ait pas pu être inscrit dans tous les agendas avant que la panne ne survienne. Bien que l'objet rendez-vous existe, certains participants ne le trouvent pas dans leurs agendas.

La duplication est le paradigme utilisé dans GARF pour rendre une application résistante aux pannes. Pour chaque objet, il faut définir le taux de duplication et le placement des duplicas sur les noeuds. La figure 7 illustre le placement des

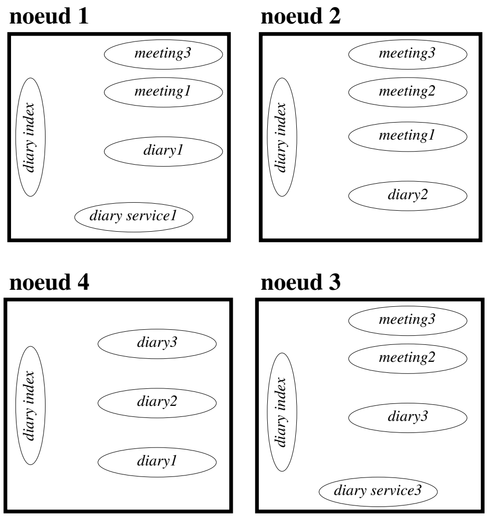

Figure 7: Duplication des objets du GRAD

duplicas du GRAD sur quatre noeuds. On suppose dans ce contexte que l'occurence d'une seule panne est courante, celle de deux pannes est rare et celle de trois pannes exceptionelle. L'annuaire *diary index* est dupliqué sur tous les noeuds: même si trois pannes surviennent il est toujours possible de consulter l'annuaire, voire de créer un nouvel agenda (qui

ne sera pas dupliqué). Suivant quelle machine est encore active, certains utilisateurs pourront encore consulter leurs agendas. En effet, on suppose que chaque utilisateur travaille toujours à partir de la même machine, soit les noeuds 1, 2 ou 3 sur la figure 7. Le noeud 4 joue le rôle d'une machine de secours lorsque la machine d'un utilisateur est en panne. Par conséquent, chaque agenda est dupliqué en deux exemplaires dont un sur le noeud 4. Après une panne n'importe quel agenda est encore opérationnel. Sur la figure 7, trois agendas *diary1*, *diary2* et *diary3* sont représentés. Chaque *diaryi* réside sur le noeud i et sur le noeud 4. Un rendez-vous est dupliqué sur chaque noeud où travaille un participant. Par exemple, *meeting1* est dupliqué sur le noeud 1 et le noeud 2, car il figure dans *diary1* et dans *diary2*. De même, *meeting3* est dupliqué sur les noeuds 1, 2 et 3 car il figure dans *diary1*, *diary2* et *diary3*. Cette politique de duplication permet à un rendez-vous d'être accessible tant qu'un agenda qui le contient est opérationnel.
Une instance de la classe *DiaryService* représente une session avec un utilisateur travaillant sur un noeud donné. La panne de ce noeud empêche l'utilisateur de continuer son travail. Par conséquent, les risques d'incohérences mis à part, l'interface utilisateur n'a pas besoin de résister à la panne du noeud sur lequel il s'exécute: il n'est donc pas dupliqué. La figure 7 montre que les propriétaires de *diary1* et de *diary3* travaillent avec le GRAD, comme en témoignent les instances *diary service1* et *diary service3*.

4.4 Choix des objets comportementaux

Lorsque le comportement de tous les objets de l'application a été défini pour chaque aspect (concurrence, persistance et résistance aux pannes) on procède au choix des objets comportementaux qu'il faut associer à chaque objet. Pour effectuer ce choix, il est utile de résumer le comportement de chaque objet à l'aide d'un tableau (cf. table 1). Les comportements des objets du GRAD sont définis par classe afin de simplifier l'exposé. Cependant, rien n'empêche le programmeur d'associer des comportements différents aux objets d'une même classe. Par exemple, on peut imaginer des politiques de duplication particulières pour certains agendas.

Classe	**Concurrence**	**Persistance**	**Distribution/Résistance aux pannes**	
			nb. exemplaires	**placement**
DiaryIndex	1 écriture, n lectures	oui	4	1 / noeud
Diary	idem	idem	2	1 / noeud du propriétaire, 1 / noeud 4
Meeting	idem	idem	1 / participant	1 / noeud du participant
DiaryService	non	non	1	noeud de l'utilisateur

Table 1: Comportement des objets du GRAD

Pour chaque classe d'objets données, le programmeur choisit la classe des encapsulateurs[6] et la classe des messagers. Lors de ce choix, deux situations peuvent se présenter: soit les classes d'objets comportementaux existent et le programmeur les utilise telles quelles, soit le programmeur construit de nouvelles classes en s'appuyant sur des classes existantes.
Dans le contexte du GRAD, les classes d'objets comportementaux nécessaires n'existent pas encore dans la hiérarchie de l'environnement GARF. Le programmeur construit de nouvelles classes en utilisant la notion d'héritage multiple. L'annuaire, unique instance de la classe *DiaryIndex*, est concurrent, persistant et dupliqué. Pour modéliser ce comportement, le programmeur construit la classe *E_DiaryIndex* qui hérite de méthodes définies dans les classes *ReadsWrite*, *Persistent* et *ActReplica*. La classe de messagers associée à l'annuaire est *Abcast* (pour implémenter la duplication active (voir 2.3.3)). Les rendez-vous, instances de la classe *Meeting*, ont le même comportement que l'annuaire. On associe à la classe *Meeting* les mêmes classes d'objets comportementaux qu'à la classe *DiaryIndex*. Pour les agendas, instances de la classe *Diary*, le comportement lié à la concurrence (voir 4.1) est complexe: il pourrait être obtenu à l'aide d'un gestionnaire de transactions implémenté au niveau de la plateforme des services. Dans le contexte d'une première implémentation du GRAD, le comportement des instances de la classe *Diary* est partiellement décrit par les classes *E_DiaryIndex* et *Abcast*. Les instances de la classe *DiaryService* ont un comportement très simple: ni concurrence, ni persistance, ni résistance aux pannes. En réalité, ce dernier aspect devrait être considéré: la panne d'un interface utilisateur pendant la prise d'un rendez-vous entraine des incohérences dans les agendas (voir 4.3). Cependant, dans le contexte d'une première implémentation du GRAD, le comportement des instances de la classe *DiaryService* est décrit par les classes de bases *Encapsulator* et *Mailer*.

5 Conclusion

L'originalité de l'approche GARF réside dans la programmation séparée des fonctionnalités et des aspects comportementaux. Les aspects comportementaux concernent la concurrence, la persistance et la résistance aux pannes. Cette

[6] il s'agit bien d'une classe puisque l'on suppose, pour simplifier l'exposé, que chaque instance d'une classe d'objets données a le même comportement, i.e un encapsulateur appartenant à la même classe.

programmation à deux volets est toutefois complètement homogène car elle repose sur un seul paradigme: l'objet.
La littérature contient de nombreuses références à des environnements de programmation d'applications distribuées basés sur les objets. Ces environnements peuvent être classés en trois catégories. La première regroupe les systèmes basés sur un seul niveau d'objets, comme Argus [12], Guide [8] et Arjuna [18] par exemple. Les fonctionnalités et les aspects comportementaux sont décrits par les mêmes objets, ce qui rend difficile la réutilisation des comportements. La seconde catégorie regroupe les systèmes dits *réflexifs* comme Apertos [19] et ABCL/R2 [14]. Dans un système réflexif, chaque objet est associé à plusieurs *méta-objets* qui décrivent son comportement. Chaque méta-objet est lui même associé à d'autre méta-objets décrivant son comportement. Cette approche est séduisante par sa flexibilité mais son inconvénient majeur réside dans sa complexité. La dernière catégorie regroupe des systèmes assez proches de l'environnement GARF. Il s'agit de systèmes comme SOS [6], XENOOPS [1] et Sina [7]. Le modèle proposé dans Sina associe à chaque objet des filtres d'entrée et des filtres de sortie, dont le rôle est comparable à celui des encapsulateurs dans GARF. La différence majeure entre le modèle de Sina et celui de GARF est le concept de messagers (qui n'existe pas dans Sina) qui permet de rendre transparente la duplication ds objets données.
Un prototype de l'environnement GARF est en cours d'implémentation. Il utilise Smalltalk [11] comme langage de programmation et ISIS [2] comme plateforme de services distribués. Ce premier prototype permet d'appliquer les idées proposées dans GARF au GRAD. Cette application s'est avérée être une application pilote idéale car malgré la simplicité apparente de ses fonctionnalités, elle pose des problèmes comportementaux non triviaux à résoudre. Les développements de l'environnement GARF vont dans plusieurs directions: l'enrichissement de la hiérarchie des objets comportementaux (ex: communication entre clients dupliqués et serveurs dupliqués) et la définition précise des objets système nécessaires pour implémenter GARF sur d'autres plateformes telles que Horus [17] ou Transis [9].

References

[1] S. Bijnens, W. Joosen, and P. Verbaeten. A reflective invocation scheme to realise advanced object management. In R. Guerraoui, O. Nierstrasz, and M. Riveill, editors, *Object Based Distributed Programming*. Springer Verlag, 1994.

[2] K. Birman and R. Cooper. The isis project: Real experience with a fault tolerant programming system. Technical Report TR90-1138, Computer Science Dept. of Cornell University, 1990.

[3] K. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic multicast. *Transactions on Computer Systems*, 9(3):272–314, August 1991.

[4] G. Booch. *Object Oriented Design With Applications*. The Benjamin/Cummings Publishing Company, Inc, 1991.

[5] F. Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2):56–78, february 1991.

[6] P. Dickman and M. Makpangou. Refinement of the fragmented object model. In *International Workshop on Object orientation in Operating Systems Proceedings*, 1992.

[7] M. Aksit et al. Abstracting inter-object communications using composition filters. Technical Report 0924-3755, University of Twente, 1987.

[8] S. Krakowiak et al. Design and implementation of an object-oriented, strongly typed language for distributed applications. *Journal of Object-Oriented Programming*, 1990.

[9] Y. Amir et al. Transis: A communication sub-system for high availability. Technical Report CS91-13, The Hebrew University of Jerusalem, April 1992.

[10] B. Garbinato, R. Guerraoui, and K.R. Mazouni. Distributed programming in GARF. In R. Guerraoui, O. Nierstrasz, and M. Riveill, editors, *Object Based Distributed Programming*. Springer Verlag, 1994.

[11] A.J. Goldberg and A.D. Robson. *SMALLTALK-80: The Language and its Implementation*. Addison Wesley, 1983.

[12] B. Liskov and R. Sheifler. Guardians and actions: Linguistic support for robust, distributed programs. In *Principles of Programming Languages Symposium Proceedings*, 1982.

[13] C. Low. A shared persistent object store. In *European Conference on Object Oriented Computing Proceedings*, 1988.

[14] S. Matsuoka, T. Watanabe, and A. Yonezawa. Hybrid group reflective arcjitectures for object-oriented reflective programming. In *European Conference on Object Oriented Programming Proceedings*, 1991.

[15] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 1988.

[16] D. Powell, editor. *Delta4: A Generic Architecture for Dependable Computing*. Springer-Verlag, 1991.

[17] R. Van Renesse, K. Birman, and R. Cooper. The horus system. Technical report, University of Cornell (NY), 1993.

[18] S.K. Shrivastava, G.N. Dixon, and G.D. Parrington. An overview of the arjuna distributed programming system. *IEEE Software*, 1991.

[19] Y. Yokote. The apertos reflective operating system: The concept and its implementation. In *Object-Oriented Programming Systems, Languages and Applications Conference Proceedings*, 1988.