Manuel d'utilisation de PHOENIX

Assia DOUDOU

Rapport LSE EPFL

Juin 1995

Table des matières

1 Introduction

Ce manuel présente la marche à suivre pour réaliser une application tolérante aux pannes avec la boîte à outils PHOENIX. A cet effet, on va suivre les différentes étapes de conception d'une application simple qui utilise PHOENIX. Ceci aura pour intérêt d'une part, de vous faire ressentir l'utilité des outils offerts par PHOENIX, et d'autre part d'être capable au terme de la lecture de ce manuel d'écrire, en utilisant PHOENIX, une application tolérante aux pannes. PHOENIX duplique votre application sur plusieurs machines de votre choix et garantit la cohérence de toutes les copies de votre application. Une question qui vient légitimement à l'esprit est de se dire à quoi va servir cette duplication? Imaginez que vous n'ayez qu'une seule copie d'une application cruciale pour votre travail. Imaginez à présent que la machine sur laquelle tourne votre application tombe en panne. Vous venez de perdre toutes vos données et de plus votre activité est paralysée tant que votre machine n'est pas de nouveau opérationnelle. Mais si vous disposez d'un système qui vous permettrait de dupliquer votre application sur plusieurs machines, la panne d'une machine ne serait pas fatale pour votre travail, vu que d'autres copies seraient toujours disponibles. Ceci est justement le type de service que vous offre PHOENIX: dupliquer une application sur plusieurs sites et garantir la cohérence de toutes les copies se trouvant sur les sites opérationnels.

La première partie de ce manuel présente l'application qui va servir d'exemple, à savoir une application bancaire simple. La deuxième partie présente quelques concepts importants à connaître avant de commencer la mise en oeuvre de l'application. Enfin, la dernière partie donne tous les détails nécessaires à la mise en oeuvre de l'application.

2 Exemple d'application pour PHOENIX

L'application qui sert d'illustration est une application bancaire. La banque est composée de plusieurs guichets ou automates à partir desquels l'utilisateur peut effectuer des retraits ou des dépôts d'argent, après avoir introduit le numéro de compte. La gestion des comptes se fait grâce à un serveur de banque, qui met en oeuvre les traitements liés aux requêtes des clients. Ce serveur stocke également l'état des différents comptes d'utilisateurs. Par conséquent, le serveur de banque représente la partie cruciale de cette application. C'est donc cette partie qu'il faut dupliquer sur différentes machines afin de garantir une plus grande disponibilité des données. Les employés de la banque disposent également de tableaux d'affichage grâce auxquels ils peuvent voir, après chaque opération effectuée sur un compte, le nouvel état des comptes (points de consultation).

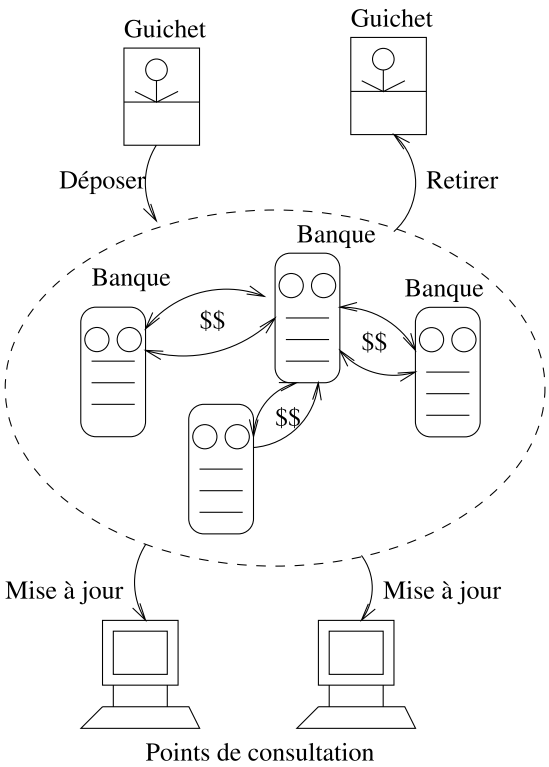

Figure 1: Application bancaire

3 Concepts clés dans PHOENIX

Le premier point très important à noter lors de l'utilisation de PHOENIX est de savoir que les outils fournis par ce dernier sont conçus à l'aide d'objets. Ce point est important dans la mesure ou votre application doit être, elle aussi, conçue selon une approche orientée objets, ou du moins la partie qui interagit avec PHOENIX.

Clarifions à présent une notion que l'on verra tout au long de l'utilisation de PHOENIX, à savoir la notion de **groupe**.

3.1 Groupe

Un groupe est un ensemble d'objets désignés comme une seule entité. Ces objets résident sur des machines différentes. Les objets qui composent un groupe sont de trois types:

- **Membres:** les membres *(members)* sont généralement les serveurs de l'application. L'ensemble des membres d'un groupe forme le noyau du groupe.
- **Clients:** les clients *(clients)* offrent une interface entre les utilisateurs et les serveurs.
- **Puits:** les puits *(sinks)* sont des objets qui reçoivent des messages envoyés par les serveurs. Les puits sont facultatifs dans une application PHOENIX.

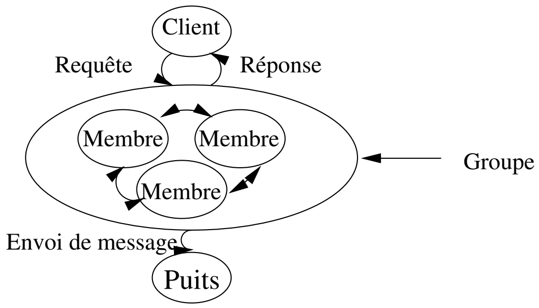

Figure 2: Les membres, clients et puits d'un groupe

Un groupe peut être composé de plusieurs membres, clients et puits. Un nouveau membre peut se joindre à tout moment au groupe et un autre peut le quitter volontairement ou involontairement, de même que pour les clients et les puits. Par conséquent, la composition d'un groupe peut varier au cours du temps, on parlera alors de groupe dynamique.

Notons que les classes qui décrivent les composants de PHOENIX sont liées par une relation d'héritage. La classe puits étant la classe de base, la classe client est une classe intérmédiaire et enfin la classe membre qui est la

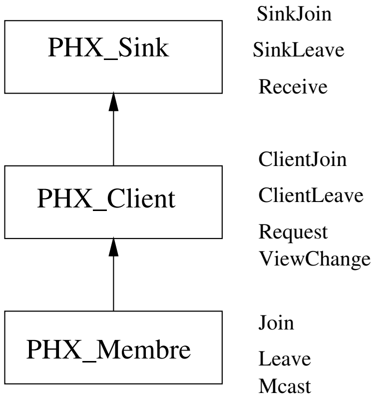

Figure 3: Héritage entre les différentes classes

classe la plus complète. Ceci explique que les composants d'un groupe sont présentés dans le manuel selon le lien d'héritage des classes qui les décrivent.

Voyons à présent d'un peu plus près les caractéristiques principales de chacune de ces classes et pourquoi il est nécessaire d'en savoir plus sur elles avant de commencer la mise en oeuvre de l'application.

3.1.1 Puits

Un objet puits est décrit par une classe appelé `PHX_Sink`. Le rôle d'un objet puits dans un groupe PHOENIX est de recevoir des informations qui lui sont envoyées par les membres. Ces informations concernent en général l'état dupliqué par les membres. L'état dupliqué est composé de l'ensemble des données gérées par un serveur et que l'on désire avoir en plusieurs exemplaires afin de garantir une plus grande disponibilité des données. Le puits peut éventuellement stocker les informations qu'il reçoit des serveurs sur une base de données ou bien les afficher. Le puits n'a pas de rôle actif quant à la gestion de l'état dupliqué. Par ailleurs, un objet puits peut être puits de plusieurs groupe en même temps.

Les méthodes les plus importantes à connaître afin de mettre en oeuvre un objet puits sont:

- `SinkJoin`: cette méthode permet à un objet puits de joindre un groupe en tant que puit.
- `SinkLeave`: cette méthode permet à un objet puits de quitter un groupe.

- `SinkReceive`: cette méthode permet à un objet puits de recevoir des informations venant des membres.

Utilisation d'un objet puits:
De manière générale, lors du dévelopement d'une application PHOENIX, la conception d'un objet puits n'est pas une nécessité.

Reprenons l'application bancaire, l'objet puits de cette application est le point de consultation sur lequel sont affichées les informations concernant l'état de l'ensemble des comptes. On parlera alors d'objet de la classe `CConsultationPoint`, qui devra hériter de la classe `PHX_Sink`.

3.1.2 Client

Un objet client est décrit par la classe `PHX_Client` qui hérite de la classe `PHX_Sink`. Le rôle d'un client dans un groupe est d'adresser les requêtes des utilisateurs vers les membres du groupe. Cet envoi peut avoir une sémantique bloquante ou pas. Le cas bloquant correspond au cas ou le client décide de ne pas envoyer de nouvelle requête tant que la précédente n'a pas été acquittée. Le cas non bloquant consiste à envoyer de nouvelles requêtes même si les anciennes ne sont pas acquittées. Un groupe PHOENIX peut avoir plusieurs clients, tout comme un client peut être client de plusieurs groupes. Les méthodes les plus importantes à connaître afin de mettre en oeuvre un client d'une application PHOENIX sont:

- `ClientJoin`: cette méthode permet à un client de joindre un groupe.
- `ClientLeave`: cette méthode permet à un client de quitter un groupe dont il etait client.
- `ViewChange`: cette méthode informe les clients d'un groupe que la composition du groupe a changée, et leur délivre la nouvelle composition.
- `Request`: cette méthode permet à un client d'envoyer une requête à un groupe.

Utilisation d'un objet client:
Un objet client est primordial dans une application PHOENIX, sans quoi les membres ne recevront jamais de requêtes. Dans l'application bancaire, il est clair que c'est les guichets et les automates qui se chargent d'adresser

les requêtes aux serveurs de la banque. On parlera alors d'objet de la classe `CBankOffice` qui décrit le comportement d'un guichet ou d'un automate de la banque. Cet objet doit hériter de la classe `PHX_Client`.

3.1.3 Membre

Un objet serveur est décrit par la classe `PHX_Member` qui hérite de la classe `PHX_Client`. Le rôle des membres est de gérer l'état dupliqué. Le traitement à effectuer sur cet état dépend des requêtes susceptible d'être envoyées par les clients.

Les membres disposent également de primitives qui leur permettent de garantir la cohérence de cet état dupliqué. En effet, la cohérence d'un état partagé peut facilement être compromise. Imaginons, que les requêtes envoyées par les clients n'arrivent pas dans le même ordre aux différents membres, si aucune précaution n'est prise, les états des membres seront différents. Nous verrons dans la section 3.3 un peu plus en détails certaines de ces primitives.

Mais contrairement aux clients, les membres ne peuvent être membres que d'un seul groupe à la fois.

Les méthodes les plus importantes à connaître lors de l'utilisation des membres sont:

- `Join`: cette méthode permet à un membre de rejoindre un groupe.
- `Leave`: cette méthode permet à un membre de quitter un groupe.
- `Mcast`: cette méthode permet de diffuser un message à l'intérieur d'un groupe.

Utilisation des membres:

Pour expliquer la nécessité d'avoir des membres dans une application reprenons notre application bancaire. Dans l'application bancaire, la gestion des comptes est le noyau de l'application. Cette gestion est faite par un serveur qui gère et traite l'ensemble des comptes des utilisateurs. Le serveur stocke l'état des comptes dans une liste qu'il maintient à jour. L'information critique de cette application est l'état de comptes des différents utilisateurs. Par conséquent, c'est cette information qu'il faut dupliquer afin de fournir aux utilisateurs une plus grandes disponibilité des données. A cet effet, l'objet serveur de l'application bancaire doit être considéré comme un membre au sens PHOENIX. Il sera décrit par la classe `CAccountDB` qui héritera de la classe `PHX_Member`.

En conclusion, grâce à l'utilisation de PHOENIX, il y aura autant d'objets serveurs que l'utilisateur le désire. Ainsi l'état dupliqué existera en plusieurs copies sur différents sites. De plus toutes les copies seront cohérente entre elles.

3.2 Changement de vue

La composition d'un groupe change au fil du temps. En effet, à tout moment un nouveau client, membre ou puits peut rejoindre ou quitter le groupe. PHOENIX dispose d'un mécanisme de changement de vue qui permet aux membres et aux clients d'un groupe de connaître la vue actuelle du groupe. Tout d'abord définissons ce que c'est qu'une vue: une vue est une liste de tous les membres présent à un instant donné dans le groupe. Ce mécanisme de changement de vue est enclenché si un nouveau membre rejoint le groupe, ou le quitte. Il a pour effet d'informer les membres et les clients du groupe de l'arrivée ou du départ d'un membre. On remarquera que ni l'arrivée ni le départ d'un client dans un groupe ne provoque de changement de vue. Par contre, le client est informé de la nouvelle vue, chaque fois que cette dernière change. Signalons finalement, que les puits ne provoquent pas de changements de vues ni n'en recoivent.

Rôle du changement de vue:

L'information délivrée par le mécanisme de changement de vue n'est pas vraiment exploitée dans l'application bancaire. Mais on peut imaginer des applications, dans lesquelles le programmeur décide que les tâches exécutées par les serveurs ne seront pas les mêmes. Par exemple, il n'y a qu'un seul serveur, appelé coordinateur, qui traite toutes les requêtes et il envoie le nouvel état périodiquement aux autres. Dans ce cas on peut exploiter le vue courante pour déterminer ce coordinateur.

3.3 Multicast

Le multicast est un mécanisme de communication qui permet à un objet membre de diffuser des messages dans un groupe, i.e., le message est diffusé à tous les membres du groupe. Ce mécanisme est couramment utilisé par les membres du groupe.

Toutefois, un client peut par indirection, diffuser un message dans un groupe. Pour cela le client envoie un message au groupe grâce à la primitive `Request`. Le message est reçu par un des membres qui se charge de le diffuser aux autres membres du groupe. L'avantage d'utiliser une primitive d'adressage basée sur le principe du **Multicast**, pour le client est que cela le décharge d'avoir à connaître la composition exacte d'un groupe afin de lui adresser une requête.

Cette communication est implémentée grâce à la méthode `Mcast` défini dans la classe `PHX_Member`. Cette primitive permet également de décider de l'ordre de livraison des messages ainsi que de l'atomicité d'exécution des requêtes, ce qui assure la consistance de l'état dupliqué. Prenons un exemple: si vous donnez comme paramètre du `Mcast` "Total_Order"ceci implique:

- D'une part, que si le message est délivré par un membre il le sera par tous les membres opérationnels du groupe. En d'autres termes, une requête est exécutée par tous les membres opérationnels ou par aucun; ceci résume la propriété dite d'atomicité.
- D'autres part, l'ordre de livraison des messages est identique pour tous les membres, i.e., si un membre délivre un message m_1 et ensuite m_2, on garantit que tous les autres membres délivreront les messages dans le même ordre. Ceci signifie que toutes les requêtes envoyées aux membres sont exécutées dans le même ordre.

L'interêt de sélectionner un ordre total est de garantir que toutes les requêtes délivrées par les serveurs opérationnels seront exécutées sur tous les sites dans le même ordre.

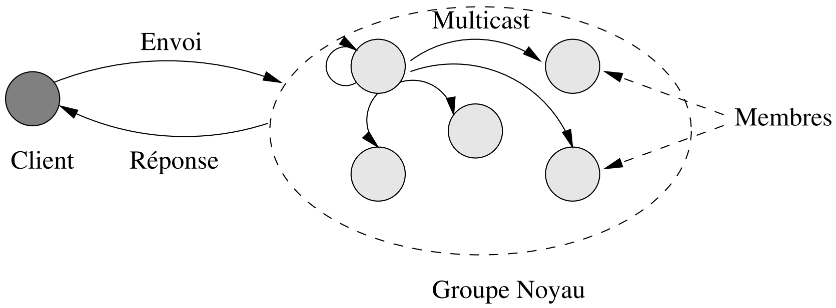

Figure 4: Exécution d'une requête

Manipulation du multicast:

Dans l'application bancaire vous n'aurez pas à manipuler la primitive du `Mcast`. Cependant, il est intéressant de comprendre son mécanisme afin d'avoir une idée sur le fonctionnement de la primitive `Request` (voir figure 4). Cette dernière est utilisée par le client pour adresser les requêtes aux serveurs.

3.4 Transfert d'état

Le transfert d'état peut être vu comme un mécanisme de synchronisation entre certains membres d'un groupe.

Lorsque un nouveau membre rejoint le groupe il n'a aucune connaissance de l'état dupliqué. Ce membre doit prendre connaissance de cet état pour pouvoir participer à ca gestion. Pour ce faire un membre du groupe transfère son état vers le nouveau membre[1]. Le nouveau membre ne peut exécuter aucunes requêtes tant que ce transfert n'est pas terminé.

Manipulation du transfert d'état:

Dans toutes applications PHOENIX le transfert d'état est une opération importante, sans quoi, les nouveaux membres d'un groupe ne pourront pas prendre connaissance de l'état dupliqué. Reprenons l'exemple de notre application bancaire: l'état dupliqué est la liste des comptes des utilisateurs. C'est donc cet état qu'il faut transmettre au nouveau membre. A cet effet, un membre du groupe doit construire un message, de type PHX_MSG[2], qui est une suite d'octets, dans lequel on doit mettre les informations relatives à l'état dupliqué. La méthode qui exécute cette opération est appelée `GetState`. De même, une opération symétrique, appelée `PutState` doit être définie afin de permettre au nouveau membre de reconstituer l'état dupliqué à partir du message reçu. Le mécanisme de transfert d'état est enclenché automatiquement par PHOENIX. A ce niveau on pourrait se demander pourquoi le `GetSate` et `PutState` ne sont pas des méthodes internes à PHOENIX et par conséquent, leur utilisation serait transparente à l'utilisateur. La raison pour laquelle ces deux méthodes ne sont pas prises en charge par PHOENIX est que l'utilisateur est seul à connaître l'état qu'il désire dupliquer et la structure des données de cet état. Par conséquent, c'est à lui de lire les données de son état et de les mettre dans un message PHOENIX. Quant à l'envoi effectif de ce message c'est PHOENIX qui détecte qu'il y a lieu de faire un transfert d'état et par conséquent il appelle la méthode `GetState`. De même, la méthode `PutState` doit être redéfinie

[1] Notons que lorsque un client rejoint le groupe les membres ne lui transfèrent pas d'état.

[2] voir manuel de référence.

par l'utilisateur, car la lecture du message PHOENIX reçu depend des structures de données associées à l'état partagé.

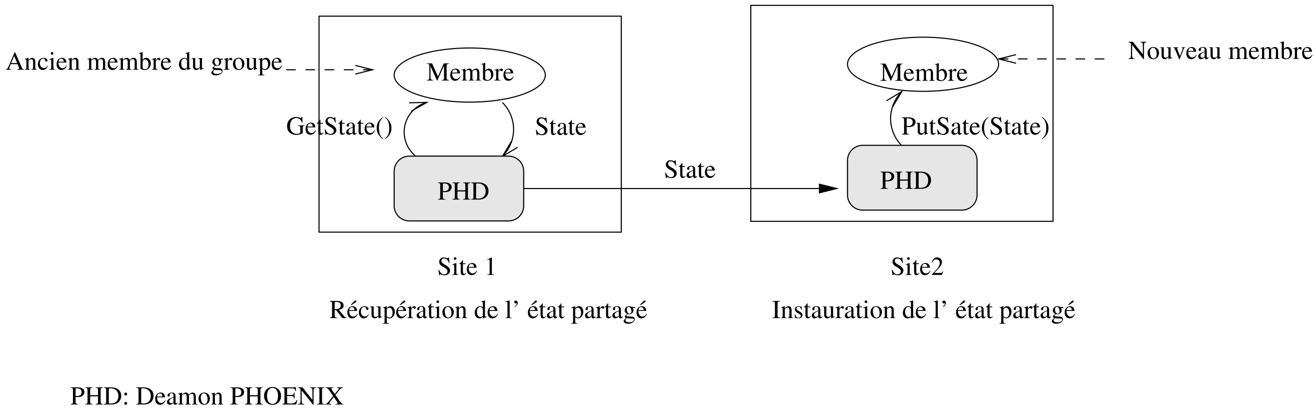

Figure 5: Mécanisme de transfert d'état

Le dernier concept que nous allons voir dans ce manuel est le concept de tâche *(thread)*. La compréhension de la notion de tâche est très utile pour comprendre le mécanisme d'invocation des méthodes dans PHOENIX.

3.5 Tâche

L'unité d'exécution traditionnelle manipulée par les systèmes d'exploitation est le processus, qui est défini comme étant un programme en exécution. Un processus traditionnel, ou processus au sens UNIX, est caractérisé par le fait qu'il ne possède qu'un fil de contrôle[3] et un contexte mémoire[4] dans lequel le fil de contrôle est seul à s'exécuter. Lorsque l'on a besoin de mettre en oeuvre l'exécution concurrente de plusieurs fils de contrôle, l'utilisation des processus traditionnels est coûteuse, car la commutation d'un processus est suivi de la sauvegarde ou de la restitution de tout son contexte mémoire. D'où l'alternative d'utiliser des processus légers *(lightweight processes)*, qui

[3] On appelle fil de contrôle *(thread of control)* l'exécution séquentielle d'une portion de code.

[4] Un contexte comprend le code et les données du programme en exécution, la pile et le contexte d'exécution qui contient des informations telle que l'état du processeur la valeur du compteur ordinal...

sont aussi appelés dans certain système **tâches**. Dans un système à processus légers, plusieurs fils de contrôle d'un même programme s'exécutent dans le même contexte mémoire.

Chaque tâche de contrôle dispose de son propre compteur ordinal et de sa pile. Cependant, contrairement aux processus traditionnels (aussi appelés processus lourds par analogie aux processus légers), plusieurs processus légers d'un même programme partagent le même espace d'adressage (voir figure 4).

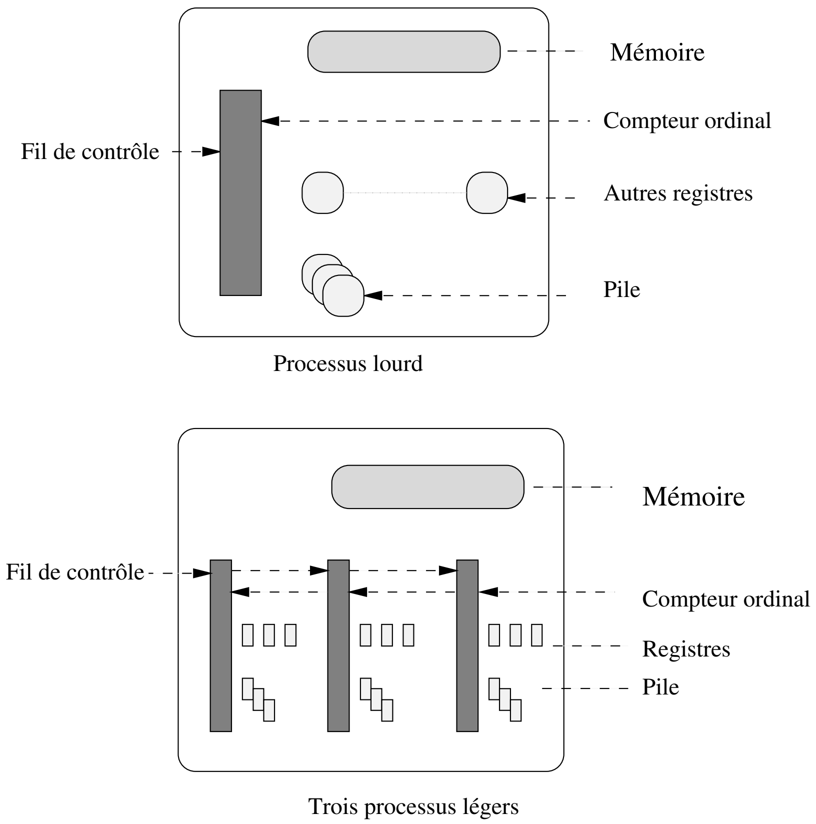

Figure 6: Processus lourd et processus légers

De par ce fait, la commutation de contexte entre deux processus légers est beaucoup moins coûteuse, mais le système ne satisfait plus la protection de la mémoire entre les différents fils de contrôle.

PHOENIX utilise des processus légers, qui sont couramment appelés

tâches . L'ordonnanceur de PHOENIX attribue par défaut aux tâches une sémantique de coroutine[5]. Par conséquent, si une tâche exécute une opération bloquante telle qu'une entrée/sortie, elle doit se détacher de l'ordonnanceur de PHOENIX afin de ne pas bloquer tout le système.

Manipulation des tâches:

Le mécanisme de tâche est un mécanisme interne à PHOENIX, donc en tant qu'utilisateur de PHOENIX, vous n'avez normalement pas besoin de l'utiliser explicitement.

Mais il faut savoir que les méthodes appelées suite à la reception d'un message ou d'un changement de vue s'exécutent dans des tâches. Typiquement, c'est ce qui se passe lorsque le client envoie un message au serveur. Ceci expliquera que vous pouvez avoir des exécutions quasi-paralléle de plusieurs méthodes d'un même objet. Par conséquent, il faut prendre, lors de la conception de vos méthodes, la précaution suivante: si une de vos méthode effectue une opération bloquante, il faut prévoir le détachement de la tâche de l'ordonnanceur juste avant l'exécution de la partie bloquante et son rattachement juste après.

3.6 Résumé des concepts

En guise de résumé, la figure suivante reproduit l´évolution d'un groupe PHOENIX et l'intervention des concepts clés présentés.

- **Partie I**: permet de voir comment une requête envoyée par un client est diffusée dans un groupe.
- **Partie II**: enclenchement du changement de vue et du transfert d'état suite à l'arrivée d'un nouveau membre.
- **Partie III**: Envoi d'une requête au nouveau groupe noyau et attente de toutes les réponse.
- **Partie IV**: changement de vue suite à la panne d'un des membres.
- **Partie V**: envoi de requête au groupe après la panne d'un des membres.

[5] Ceci permet d'éviter le déséquencement des messages et par la même occasion évite les conflits d'accès concurrent aux même zones mémoire.

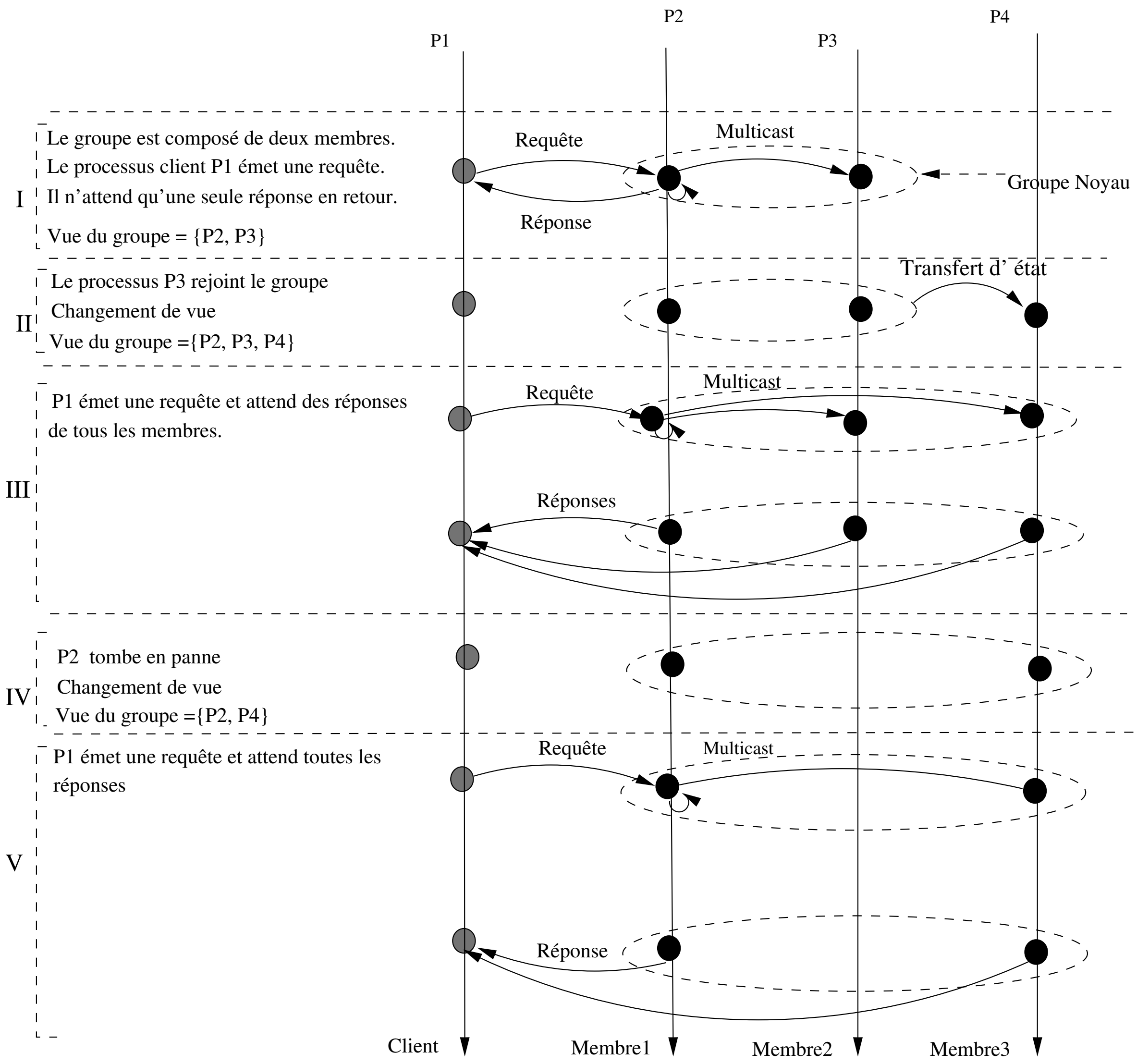

Figure 7: Intéraction entre les éléments d'un groupe

4 Mise en oeuvre d'une application PHOENIX

4.1 Présentation de l'architecture de l'application bancaire

L'application bancaire est constituée de trois composants:

- Serveur de banque: le serveur de banque est décrit par un objet qui gère l'état dupliqué. Cet état est composé de la liste des comptes des utilisateurs. Un objet serveur est une instance de la classe `CAccountDB`. Un objet compte est une instance de la classe `CAccount`. A fin de garantir la disponibilité des données, c'est l'objet serveur qui doit être dupliqué sur plusieurs sites. C'est donc un membre au sens PHOENIX, par conséquent il doit hériter de la classe `PHX_Member`.

- Interface client: l'interface entre le serveur et les utilisateurs est réalisée grâce à un objet client. Cet objet envoie les requêtes des utilisateurs vers les membres du groupe. Un objet client est une instance de la classe `CBankOffice`. A fin d'être considéré comme un client d'un groupe, il doit hériter de la classe `PHX_Client`.

- Point de consultation: un point de consultation reçoit des messages des membres concernant l'état partagé. Cet état est stocké dans une liste d'objets compte. Un objet compte est une instance de la classe `CAccount`. Un objet point de consultation est une instance de la classe `CConsultationPoint`. Afin, que cet objet soit considéré comme un puits d'une application PHOENIX, il doit hériter de la classe `PHX_Sink`.

Les membres d'une application sont forcément sur des machines distinctes. D'ailleurs, si vous tenter de lancer deux membres sur le même site PHOENIX vous signale une erreur. Cette contrainte est tout a fait justifiable, car l'objectif d'une plateforme telle que PHOENIX est de garantir la tolérences aux pannes par la duplication or si vos copies se trouvent toutes sur une même machine la duplication n'a plus de sens, vu que la panne de cette machine détruirait toutes les copies de l'application. Par contre, les sites ou résident les clients et les puits peuvent être quelconque.

La figure 7 présente l'architecture de l'application bancaire. Le client adresse une requête aux membres. Les membres recoivent la requêtes et l'effectue sur l'état partagé. Ensuite, ils envoient un acquittement aux clients et un message au puits. Le puits reçoit le message le stocke dans l'état qu'il gère et l'affiche.

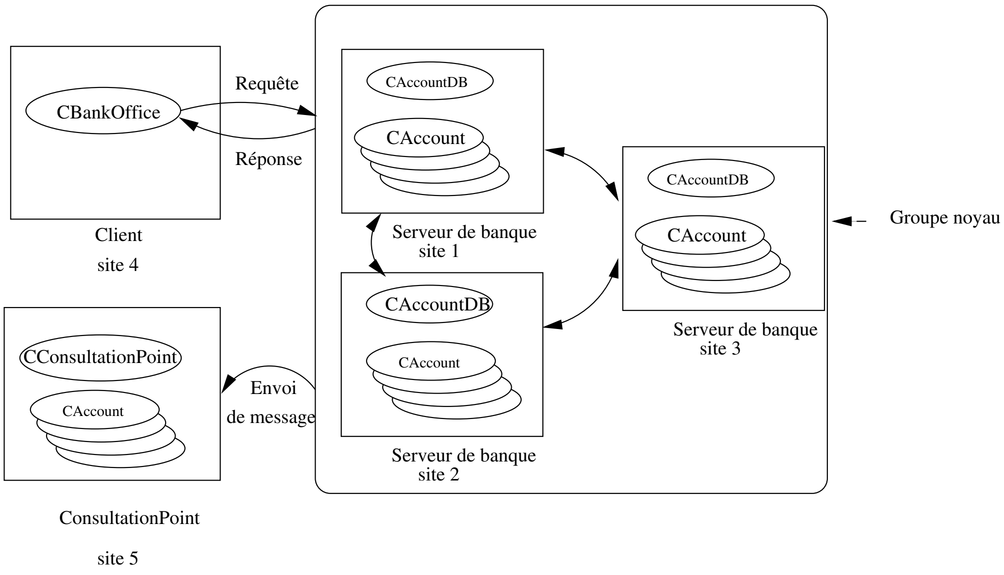

Figure 8: Architecture d'une application bancaire

4.2 Mise en oeuvre du serveur de banque

En général, un serveur d'une application PHOENIX doit remplir les quatre fonctions suivantes:

- Traiter les requêtes envoyées par les clients du groupe.
- Envoyer des réponses aux clients si nécessaire et des messages aux puits.
- Traiter les changements de vues.
- Traiter le transfert d'état.
- Envoyer des message aux puits relatifs à l'état dupliqué.

Un objet serveur est décrit par la classe `CAccountDB`. Chaque méthode de cette classe effectue un des traitement cité si-dessus. Le detail de l'interface est comme suit:

```
class CAccountDB : public PHX_Member {
 public:
// Constructor & Destructor
```

```
  CAccountDB();
  ~CAccountDB() {}

protected:
// Event handling
  void ViewChange(PHX_Group& grp); // Override
  // Called upon viewchange.
  PHX_Msg *GetState(); // Override
  void PutState(PHX_Msg& state); // Override
  // Called upon state transfer.

  void Body(); // Override
  // Thread associated with the object (the object is ACTIVE).

  void Dump(FILE *f);
  // Dump the internal state of the objects (i.e. all accounts).

private:
  void Deposit(PHX_Msg& msg, PHX_Id& srcid, ULONG reqid);
  // Perform a deposit on a particular account.
  void Withdraw(PHX_Msg& msg, PHX_Id& srcid, ULONG reqid);
  // Perform a withdrawal on a particular account.
  void SendStateToSinks();
  // Send the internal state (i.e. all accounts) to the sink members.
  void SendReply(char *msg, PHX_Id& srcid, ULONG reqid);
  // Send a reply to 'srcid'.

  PHX_List<CAccount> fAccounts;
  // List of all account.
};
```

Le serveur gère une liste d'objets compte, chaque compte est décrit par la classe `CAccount`. Les méthodes de cette classe permettent de manipuler un objet compte afin d'effectuer le dépôt et le retrait d'argent. Cette classe décrit également les attributs qui caractérisent un compte tel que son numéro et son solde actuel. Le détail de l'interface est donné comme suit:

```
class CAccount {
public:
// Constructor & Destructor
  CAccount()
    { fNb = 0; fAmount = 0; }
  CAccount(int nb)
    { fNb = nb; fAmount = 0; }
  CAccount(int nb, int amount)
    { fNb = nb; fAmount = amount; }

// Operators
  bool operator ==(const CAccount& a) const
    { return(fNb == a.fNb && fAmount == a.fAmount); }
  // Compare 2 accounts for equality.
```

```
// Access
  void Deposit(int amount) { fAmount += amount; }
  // Perform a deposit on the account.
  void Withdraw(int amount) { fAmount -= amount; }
  // Perform a withdrawal on the account.
  int Amount() { return(fAmount); }
  // Return the amount of the account.
  int Nb() { return(fNb); }
  // Return the account number.

private:
  int fNb;
  // Account number.
  int fAmount;
  // Amount of money on the account.
};
```

4.2.1 Traitement des requêtes

Le type de requêtes que doit traiter un serveur depend de la nature de l'application. Dans notre cas, le serveur de banque doit traiter deux requêtes: Le dépôt et le retrait d'argent.

Opération de dépôt:

Cette méthode, appelée `Deposit`, effectue un dépôt d'argent dans un compte. Lors d'un dépôt deux cas de figures sont à envisager. Premièrement, le compte n'existe pas: il faut donc le créer et ensuite y déposer l'argent. Deuxièmement, le compte existe déjà: le dépôt est immédiat. Les informations nécessaires à cette opération telles que le numéro de compte et la somme à déposer sont récupérées à partir du message PHOENIX envoyé par le client grâce à la méthode `Get` définie dans la classe `PHX_Msg`.

Une fois que le dépôt est effectué, la méthode envoie un acquittement au client en invoquant la méthode `SendReply`.

N.B. L'opération `SendStateToSink` permet d'envoyer le nouvel état aux puits du groupe; le détail de son implémentation est vu plus tard.

```
void CAccountDB::Deposit(PHX_Msg& msg, PHX_Id& srcid, ULONG reqid)
{
  int nb;
  int amount;

  msg.Get(&nb, sizeof(int));
  msg.Get(&amount, sizeof(int));
  // Read the account number and amount from the incoming message.
  for(Pix i = fAccounts.first(); i != 0; fAccounts.next(i)) {
    // Look for the account in the database.
```

```
    if(fAccounts(i).Nb() == nb) {
      fAccounts(i).Deposit(amount);
      this->SendReply(DEP_OK, srcid, reqid);
      // Send an OK reply.
      break;
    }
  }
  if(i == 0) {
    // We didn't find the account. We create it.
    fAccounts.add(CAccount(nb, amount));
    this->SendReply(DEP_NEW, srcid, reqid);
    // Send an NEW ACCOUNT reply.
  }
  if(this->FirstInGroup())
    SendStateToSinks();
}
```

Opération de retrait:

Le principe de cette méthode, appelée `Withdraw`, est similaire à celui du dépôt.

Entrées:

Vous devez associer à chaque méthode qui traite les requêtes envoyées par les clients une entrée au niveau du serveur. L'idée des entrées est de gérer différents types de requêtes sans que le programmeur ait à faire lui-même un "dispatching" explicite (i.e. déterminer à partir d'un message l'opération que le client veut invoquer).

Dans le cas de notre application, on associe deux entrées aux deux méthodes dépôt et retrait comme suit:

```
CAccountDB::CAccountDB() : PHX_Member(ACTIVE)
{
  this->CreateEntry(DEPOSIT, (TEntry)this->Deposit);
  this->CreateEntry(WITHDRAW, (TEntry)this->Withdraw);
  // Create the entries associated to messages.
  // E.g. when the object receives a message with key 'DEPOSIT',
  // its 'Deposit' method will be invoked.
}
```

4.2.2 Envois de réponses aux clients

Si vous désirez que les clients de votre application reçoivent des réponses, il est nécessaire de prévoir une opération qui s'en charge au niveau du serveur. La première étape consiste à définir l'ensemble des réponses possibles. Dans le cas de notre application cet ensemble est défini comme suit:

```
static char *WITH_OK = "Withdraw successful";
```

```
static char *WITH_KO = "Withdraw failed: no such account";
static char *DEP_OK = "Deposit successful";
static char *DEP_NEW = "Deposit successful: created new account";
```

La deuxième étape consiste à définir l'opération qui effectue l'envoi des réponses. Cette opération doit créer un message PHOENIX dans le quel elle met le message, après quoi elle invoque le méthode `Reply` de PHOENIX qui effectue l'envoi effectif.

```
void CAccountDB::SendReply(char *msg, PHX_Id& srcid, ULONG reqid)
{
  int len = strlen(msg) + 1;
  PHX_Msg rply(len);
  rply.Put(msg, len);
  this->Reply(rply, srcid, reqid);
  // Put the string in a Phoenix message and send the reply.
}
```

Finalement l'invocation de la méthode `SendReply` se fait au sein de chacune des méthodes qui traite les requêtes, i.e., la méthode de dépôt et de retrait. Les messages envoyés sont définis en fonction du succès ou de l'échec de la requête. Dans ce qui suit nous donnons deux exemples d'appel:

```
 this->SendReply(WITH_OK, srcid, reqid);
 this->SendReply(WITH_KO, srcid, reqid);
```

4.2.3 Traitement des changements de vues

Chaque site serveur de l'application bancaire va recevoir à son insu la composition du groupe à chaque fois que cette composition change. Dans notre application, cette information est utilisée afin d'être affichée, ce qui permet à l'utilisateur de voir quand les changements de vues surviennent et quels sont les serveurs encore opérationnels. Ceci est implémenté grâce à la méthode `ViewChange`.

```
void CAccountDB::ViewChange(PHX_Group& grp)
{
  cout << eoln << "********************" << eoln << eoln <<
    "Received ViewChange:" << eoln;
  grp.Dump(stdout);
}
```

Notons au passage, que cette méthode est tout à fait facultative. Par contre, on pourrait utiliser le changement de vue à d'autres fins que le simple affichage. Une autre utilisation serait de faire en sorte que les serveurs contrôlent à chaque fois les vues qu'ils reçoivent et dès qu'ils constatent qu'il y a moins de trois serveurs opérationnels, par exemple, ils décident de lancer un nouveau.

4.2.4 Traitement relatif au transfert d'état

Dans toute applications PHOENIX, vous aurez à implémenter des méthodes nécessaires au transfert d'état. Plus exactement, il y a deux méthodes à implémenter. La première lit l'état partagé et le stocke dans un message PHOENIX, cette méthode s'appelle `GetState`. Cette méthode est exécutée par un des anciens membres du groupe, lorsque PHOENIX détecte qu'un nouveau membre veut rejoindre le groupe. Son principe est d'allouer un message PHOENIX dont la taille correspond à la taille de l'état dupliqué. Les informations contenues dans cet état sont mises dans le message PHOENIX grâce à la méthode `Put` définie dans la classe `PHX_Msg`.

```
PHX_Msg *CAccountDB::GetState()
{
  // We put the state of all accounts in a Pheonix message.
  int size = sizeof(int) + (fAccounts.length() * 2 * sizeof(int));
  PHX_Msg *msg = new PHX_Msg(size);
  int len = fAccounts.length();
  msg->Put(&len, sizeof(int));
  for(Pix i = fAccounts.first(); i != 0; fAccounts.next(i)) {
    int nb = fAccounts(i).Nb();
    int amount = fAccounts(i).Amount();
    msg->Put(&nb, sizeof(int));
    msg->Put(&amount, sizeof(int));
  }
  return(msg);
}
```

La deuxième méthode est une méthode symétrique à la première. Cette méthode est invoquée par PHOENIX afin que le nouveau membre se mette à jour. Son principe consiste à extraire les données relatives à l'état partagé à partir du message PHOENIX reçu, grâce à la méthode `Get`. Cette méthode est implémentée comme suit:

```
void CAccountDB::PutState(PHX_Msg& state)
{
  // We get the state of all accounts from the message.
  int len;
  state.Get(&len, sizeof(int));
```

```
  for(int i = 0; i < len; i++) {
    int nb;
    int amount;
    state.Get(&nb, sizeof(int));
    state.Get(&amount, sizeof(int));
    fAccounts.add(CAccount(nb, amount));
  }
  printf("Received new state:\n");
  this->Dump(stdout);
}
```

4.2.5 Envois de messages aux puits

Dans notre application nous avons choisi de communiquer régulièrement l'intégralité de l'état dupliqué aux puits.

Ceci explique la présence de l'invocation `SendStateToSinks()` à la fin des méthodes de dépôt et de retrait d'argent. Cette invocation est faite par un membre du groupe. La méthode `GetState` est utilisé pour construire le message PHOENIX à envoyer. Du côté du puits, la méthode `Receive` se charge de reconstituer l'état partagé à partir du message recu.

```
void CAccountDB::SendStateToSinks()
{
  // Send the internal state (i.e. all accounts) to the sink members.
  PHX_Msg *msg = this->GetState();
  this->SendToSinks(*msg);
  delete msg;
}
```

4.2.6 Activation de l'objet serveur

L'objet serveur est activé grâce à sa méthode `Body`. En effet, lors de l'instanciation de l'objet serveur, le constructeur de PHX_Member est appelé avec le paramètre `Active`. Par conséquent, la méthode `Body` est lancée dans un nouveau thread et on parlera alors d'objet actif. La méthode `Body` peut être vue comme une opération qui s'exécute en tâche de fond. La première opération qu'elle invoque est le `Join`, afin que l'objet serveur de banque rejoigne le groupe des serveurs de l'application bancaire. Une fois que le serveur est membre du groupe, il est intéressant de pouvoir laisser la liberté à l'utilisateur d'intervenir afin d'arrêter l'activité d'un membre donné du groupe. Afin que ceci puisse se faire, la méthode `Body` met en oeuvre un menu permettant à l'utilisateur d'interrompre l'exécution du serveur. Mais comme nous l'avons vu dans le §3.5, une tâche doit se détacher de l'ordonnanceur de PHOENIX, avant d'effectuer une opération bloquante, en l'occurrence une entrée/sortie. Une fois que l'utilisateur sélectionne la touche qui permet de quitter l'application, la méthode `Body` sort de sa boucle d'attente, se rattache à l'ordonnanceur de PHOENIX. Après quoi elle invoque la méthode `Leave`, qui permet au serveur de quitter le groupe. Avant que la méthode `Body` ne se termine, il est nécessaire d'invoquer la méthode `ExitMainLoop`, qui avertit PHOENIX que l'exécution du serveur s'achève.

```
void CAccountDB::Body()
{
  printf("Joining group\n");
  this->Join(gName);
  // Join the specified group.

  PHOENIX::DetachTask();
  // Since we perform a blocking operation (reading a character from the
  // keyboard), we detach our thread from the Phoenix scheduling. That
  // way, we do not block other operations.
  char c;
  do {
    cout << eoln << "(Q)uit" << eoln << eoln;
    cin >> c;
  } while(toupper(c) != 'Q');
  PHOENIX::AttachTask();
  // Before calling a Phoenix function, we need to reenter the Phoenix
  // scheduling.

  printf("Leaving group\n");
  this->Leave();
  // Leave the previously joined group.

  PHOENIX::ExitMainLoop();
  // Exit Phoenix main loop.
}
```

4.2.7 Programme principal

Le programme principal contrôle les paramètres de la ligne de commande. Puis invoque la méthode `Init` définie dans la classe `PHOENIX` qui initialise les structures de données nécessaires à PHOENIX. Ceci est suivi par l'instanciation de l'objet serveur de la classe `CAcountDB`. Une fois que l'objet serveur est créé, le programme principal invoque la méthode `MainLoop`, qui donne la main à PHOENIX. Grâce à cette méthode tous les messages qui viennent de l'extérieur et qui sont destinés au serveur, seront interceptés et dirigés vers le serveur. Comme nous l'avons vu précédemment la création de l'objet serveur est suivi du lancement d'un thread dans lequel la méthode `Body` va s'exécuter. Tant que la méthode `Body` n'invoque pas `ExitMainLoop` PHOENIX continue à envoyer des messages à l'objet serveur. Une fois que l'invocation d'`ExitMainLoop` est faite le programme principal reprend la main et exécute les instructions qui se trouvent après le `MainLoop`.

```
int main(int argc, char **argv)
{
  if(argc != 2) {
    printf("Usage: %s groupname\n", argv[0]);
    exit(1);
  }

  gName = argv[1];
  // Store the group name.

  PHOENIX::Init();
  CAccountDB *accounts = new CAccountDB;
  PHOENIX::MainLoop();
  delete accounts;
  PHOENIX::Exit();
  return(0);
}
```

4.3 Mise en oeuvre d'un client

La mise en oeuvre d'un client est considérablement plus simple que celle d'un serveur. Un client doit remplir les deux fonctions suivantes:

- Envoyer les requêtes des utilisateurs vers les membres. Ce traitement s'effectue dans la méthode principale de la classe client, à savoir, la méthode `Body`.
- Traiter les changements de vue.

Un objet client de l'application bancaire est décrit par la classe `CBankOffice` dont l'interface est donnée comme suit:

```
class CBankOffice : public PHX_Client {

public:
// Constructor & Destructor
  CBankOffice() : PHX_Client(ACTIVE) {}
  ~CBankOffice() {}
```

```
protected:
// Event handling
  void ViewChange(PHX_Group& grp); // Override
  // Called upon viewchange.

  void Body(); // Override
  // Thread associated with the object (the object is ACTIVE).

private:

  PHX_Group fGroup;
  // current state of the group of accounts databases.
};
```

4.3.1 Activation de l'objet client

Tout le traitement qui consiste à récupérer les requêtes des utilisateurs et à les envoyer aux serveurs est fait au niveau de la méthode `Body`. Cette méthode est la méthode principale de la classe `CBankOffice`. Si lors de l'instanciation de l'objet client, le constructeur de `PHX_Client` est appelé avec le paramètre `Active`, la méthode `Body` est lancée dans un nouveau thread et on parlera alors d'objet client actif. Le premier appel effectué par la méthode `Body` est le `ClientJoin`, grâce à quoi l'objet client pourra joindre le groupe. Cette méthode implémente également un menu permettant à l'utilisateur de sélectionner une requête parmi une liste de requêtes proposées.

Remarque: Les identificateurs utilisés pour le choix des requêtes dans le menu doivent correspondre aux entrées spécifiés au niveau du serveur.

Une fois que le choix est effectué, on récupère le numéro de compte ainsi que le montant à déposer ou à retirer. Un message PHOENIX est crée et on met toutes les informations relatives à la requête grâce à la méthode `Put`. Après quoi, la méthode `Request` envoie le message vers le groupe et attend des réponses de tous les membres. Cette méthode n'est pas bloquante. Par contre, l'opération d'attente des acquittements `Replies` est bloquante, jusqu'à ce que toutes les réponses soient reçues. La méthode `Body` bouclera sur la séquence de code qui vient d'être décrite tant que l'utilisateur ne fait pas le choix de quitter l'application client. Si tel est le cas, on invoque la méthode `ClientLeave` afin que l'objet client quitte effectivement le groupe. Finalement, on appelle la méthode `ExitMainoop`.

```
void CBankOffice::Body()
{
  this->ClientJoin(gName);
  // Join the specified group as a client.

  PHOENIX::DetachTask();
  // Since we perform a blocking operation (reading a character from the
  // keyboard), we detach our thread from the Phoenix scheduling. That
  // way, we do not block other operations.

  while(1) {
    cout << eoln << "(D)eposit" << eoln << "(W)ithdraw" << eoln <<
      "(Q)uit" << eoln << eoln;

    String str;
    cin >> str;
    char c = toupper(str[0]);

    if(c == 'Q')
      break;

    if(c == 'D' || c == 'W') {

      cout << "Account nb: ";
      int nb;
      cin >> nb;
      cout << "Amount: ";
      int amount;
      cin >> amount;

      PHX_Msg msg(2 * sizeof(int));
      msg.Put(&nb, sizeof(int));
      msg.Put(&amount, sizeof(int));
      // Put the account number and amount in a Phoenix message.

      PHOENIX::AttachTask();
      // Before calling a Phoenix function, we need to reenter the Phoenix
      // scheduling...
      PHX_Reply *rply = this->Request(fGroup, msg, ALL_REPLIES, c);
      // Send a request to the group, waiting for all replies. 'c' is used
      // as the key of the message. It will be redirected in the correct entry
      // at the server side.
      PHX_List<PHX_Msg>& replies = rply->Replies();
      // Read the list of replies from the reply object. This call is blocking
      // if all replies have not yet been received.
      PHOENIX::DetachTask();
      // ...and exit it once more.

      cout << "Received " << replies.length() << " replies: " << eoln;
      for(Pix i = replies.first(); i != 0; replies.next(i))
cout << '\t' << replies(i).Data() << eoln;
      // Print the replies to the screen.
      delete rply;
    }
  }
```

```
  PHOENIX::AttachTask();

  printf("Leaving group\n");
  this->ClientLeave(gName);
  // Leave the previously joined group.

  PHOENIX::ExitMainLoop();
  // Exit Phoenix main loop.
}
```

4.3.2 Traitement des changements de vue

Un client, tout comme un serveur, reçoit les changements de vues du groupe. Dans le cas de notre application cette information est utilisée afin d'être affichée. D'autres types d'utilisation peuvent être envisagés.

```
void CBankOffice::ViewChange(PHX_Group& grp)
{

  printf("\nReceived ViewChange:\n");
  grp.Dump(stdout);
  // Dump the new group composition to the screen.
}
```

4.3.3 Programme principal

Le programme principal contôle tout d'abord la ligne de commande, puis invoque la méthode `Init`. Il crée un objet client. Après quoi le MainLoop est lancé. L'exécution du MainLoop ne s'arrêtera que lorsque le Body exécutera `ExitMainLoop`. Cette invocation a pour effet d'indiquer à PHOENIX de ne plus adresser de message à cette objet client. Par conséquent, on peut détruire l'objet client et finalement quitter PHOENIX.

```
int main(int argc, char **argv)
{
  if(argc != 2) {
    printf("Usage: %s groupname\n", argv[0]);
    exit(1);
  }

  gName = argv[1];
  // Store the group name.

  PHOENIX::Init();
  // Initialize Phoenix.
```

```
  CBankOffice *bank = new CBankOffice;
  // Create a BankOffice object.

  PHOENIX::MainLoop();
  // Call Phoenix main loop.

  delete bank;
  PHOENIX::Exit();
  // Leave Phoenix cleanly.

  return(0);
}
```

4.4 Mise en oeuvre du points de consultation

Le rôle d'un point de consultation est résumé dans les points suivants:

- Recevoir les messages envoyés par les membres du groupe.
- Reconstituer l'état partagé à partir de ces messages.
- Afficher le contenu de l'état partagé.

Un objet point de consultation est décrit par la classe `CConsultationPoint`, dont l'nterface est donné comme suit:

```
class CConsultationPoint : public PHX_Sink {

public:
// Constructor & Destructor
  CConsultationPoint() : PHX_Sink(ACTIVE) {}
  ~CConsultationPoint() {}

protected:
// Event handling
  void Receive(PHX_Msg& msg, PHX_Id& srcid, ULONG msgid); // Override
  // Called upon message reception.

  void Body(); // Override
  // Thread associated with the object (the object is ACTIVE).

  void Dump(FILE *f);
  // Dump the internal state of the objects (i.e. all accounts).

private:
  PHX_List<CAccount> fAccounts;
  // List of all account.
};
```

L'objet point de consultation gère une liste d'objets compte dans lesquels il stocke l'état qui lui ait envoyé par les membres. Chaque objet compte est décrit par la classe `CAccount`, dont l'interface est donnée comme suit:

```
class CAccount {

public:
// Constructor & Destructor
  CAccount()
    { fNb = 0; fAmount = 0; }
  CAccount(int nb, int amount)
    { fNb = nb; fAmount = amount; }

// Operators
  bool operator ==(const CAccount& a) const
    { return(fNb == a.fNb && fAmount == a.fAmount); }
  // Compare 2 accounts for equality.

// Access
  int Amount() { return(fAmount); }
  // Return the amount of the account.
  int Nb() { return(fNb); }
  // Return the account number.

private:
  int fNb;
  // Account number.
  int fAmount;
  // Amount of money on the account.
};
```

Rôle d'un point de consultation

4.4.1 Réception d'un message en provenance des membres

La réception d'un message PHOENIX et la reconstitution de l'état dupliqué se fait grâce à la méthode `Receive`. Cette méthode lit le message PHOENIX reçu. Les données qui compose l'état dupliqué sont extraites du message grâce à la méthode `Get` et mises dans une liste des objets compte. Après quoi la liste est reparcourue depuis le début et l'état des comptes est affiché.

```
void CConsultationPoint::Receive(PHX_Msg& msg, PHX_Id& srcid, ULONG msgid)
{
  while(fAccounts.length() > 0)
    fAccounts.remove_front();
  // Since we receive the state of all accounts, we need to free the
  // previous state.

  // Read the accounts from the incoming message and store them.
  int len;
  msg.Get(&len, sizeof(int));
  for(int i = 0; i < len; i++) {
    int nb;
    int amount;
```

```
    msg.Get(&nb, sizeof(int));
    msg.Get(&amount, sizeof(int));
    fAccounts.add(CAccount(nb, amount));
  }
  cout << eoln << "********************" << eoln << eoln <<
    "Received new accounts:" << eoln;
  this->Dump(stdout);
  // Dump the accounts to the screen.
}
```

4.4.2 Activation de l'objet point de consultation

L'activation du point de consultation se fait (tout comme pour le membre et le client) grâce à la méthode `Body`. Cette méthode est lancée dans un nouveau thread, vu que le constructeur de la classe héritée `PHX_Sink` est invoqué avec le paramètre `Active`. Cette méthode commence par invoquer le `SinkJoin`, ce qui permettra à notre objet de rejoindre le groupe en tant que puits. L'utilisateur a la liberté de quitter l'application quand il le désire, grâce à un menu. Si ce choix est fait, la méthode `SinkLeave` est appelée. Par conséquent, l'objet point de consultation n'est plus considéré comme puits du groupe. Et finalement `ExitMainLoop` est appelée pour arrêter l'interaction entre le puits et PHOENIX..

```
void CConsultationPoint::Body()
{
  printf("Joining group\n");
  this->SinkJoin(gName);
  // Join the specified group as a sink.

  PHOENIX::DetachTask();
  // Since we perform a blocking operation (reading a character from the
  // keyboard), we detach our thread from the Phoenix scheduling. That
  // way, we do not block other operations.
  char c;
  do {
    cout << eoln << "(Q)uit" << eoln << eoln;
    cin >> c;
  } while(toupper(c) != 'Q');
  PHOENIX::AttachTask();
  // Before calling a Phoenix function, we need to reenter the Phoenix
  // scheduling.

  printf("Leaving group\n");
  this->SinkLeave(gName);
  // Leave the previously joined group.

  PHOENIX::ExitMainLoop();
  // Exit Phoenix main loop.
}
```

4.4.3 Programme principal

Le programme commence par contrôler la ligne de commande, puis appelle la méthode d'intialisation de PHOENIX. Après quoi, il crée un objet point de consultation et lance le `MainLoop`. Cette méthode s'exécutera tant que le `Body` n'aura pas invoqué `ExitMainLoop`. Une fois que cet appel est fait, on quitte l'application PHOENIX et on détruit l'objet point de consultation.

```
int main(int argc, char **argv)
{
  if(argc != 2) {
    printf("Usage: %s groupname\n", argv[0]);
    exit(1);
  }

  gName = argv[1];
  // Store the group name.

  PHOENIX::Init();
  // Initialize Phoenix.

  CConsultationPoint *cp = new CConsultationPoint;
  // Create a ConsultationPoint object.

  PHOENIX::MainLoop();
  // Call Phoenix main loop.

  delete cp;
  PHOENIX::Exit();
  // Leave Phoenix cleanly.

  return(0);
}
```

5 Annexe

Annexe A

Code du serveur de banque:

```
#include "phoenix.h"
#include "debug.h"

#include <ctype.h>

#define DEPOSIT 'D'
#define WITHDRAW 'W'

#define _MODULE_ _APPL_

static char *WITH_OK = "Withdraw successful";
static char *WITH_KO = "Withdraw failed: no such account";
static char *DEP_OK = "Deposit successful";
static char *DEP_NEW = "Deposit successful: created new account";

/************************************************************
 * Type definitions
 ************************************************************/

// A simple class for representing an account

class CAccount {

public:
// Constructor & Destructor
  CAccount()
    { fNb = 0; fAmount = 0; }
  CAccount(int nb)
    { fNb = nb; fAmount = 0; }
  CAccount(int nb, int amount)
    { fNb = nb; fAmount = amount; }

// Operators
  bool operator ==(const CAccount& a) const
    { return(fNb == a.fNb && fAmount == a.fAmount); }
  // Compare 2 accounts for equality.

// Access
  void Deposit(int amount) { fAmount += amount; }
  // Perform a deposit on the account.
  void Withdraw(int amount) { fAmount -= amount; }
  // Perform a withdrawal on the account.
  int Amount() { return(fAmount); }
  // Return the amount of the account.
  int Nb() { return(fNb); }
  // Return the account number.
```

```
private:
  int fNb;
  // Account number.
  int fAmount;
  // Amount of money on the account.
};

// A class for representing a database of accounts.

class CAccountDB : public PHX_Member {

public:
// Constructor & Destructor
  CAccountDB();
  ~CAccountDB() {}

protected:
// Event handling
  void ViewChange(PHX_Group& grp); // Override
  // Called upon viewchange.
  PHX_Msg *GetState(); // Override
  void PutState(PHX_Msg& state); // Override
  // Called upon state transfer.

  void Body(); // Override
  // Thread associated with the object (the object is ACTIVE).

  void Dump(FILE *f);
  // Dump the internal state of the objects (i.e. all accounts).

private:
  void Deposit(PHX_Msg& msg, PHX_Id& srcid, ULONG reqid);
  // Perform a deposit on a particular account.
  void Withdraw(PHX_Msg& msg, PHX_Id& srcid, ULONG reqid);
  // Perform a withdrawal on a particular account.
  void SendStateToSinks();
  // Send the internal state (i.e. all accounts) to the sink members.
  void SendReply(char *msg, PHX_Id& srcid, ULONG reqid);
  // Send a reply to 'srcid'.

  PHX_List<CAccount> fAccounts;
  // List of all account.
};

/************************************************************
 * Local Scope Variables
 ************************************************************/

char *gName;
int toto;

// Name of the group of accounts databases.

/************************************************************
 * Class Methods
```

```
 **********************************************************/

CAccountDB::CAccountDB() : PHX_Member(ACTIVE)
{
  printf("Creating account\n");
  this->CreateEntry(DEPOSIT, (TEntry)this->Deposit);
  this->CreateEntry(WITHDRAW, (TEntry)this->Withdraw);
  // Create the entries associated to messages.
  // E.g. when the object receives a message with key 'DEPOSIT',
  // its 'Deposit' method will be invoked.
}

void CAccountDB::ViewChange(PHX_Group& grp)
{
  cout << eoln << "********************" << eoln << eoln <<
    "Received ViewChange:" << eoln;
  grp.Dump(stdout);
  // Dump the new group composition to the screen.
}

void CAccountDB::Body()
{
  printf("Joining group\n");
  this->Join(gName);
  // Join the specified group.

  PHOENIX::DetachTask();
  // Since we perform a blocking operation (reading a character from the
  // keyboard), we detach our thread from the Phoenix scheduling. That
  // way, we do not block other operations.
  char c;
  do {
    cout << eoln << "(Q)uit" << eoln << eoln;
    cin >> c;
  } while(toupper(c) != 'Q');
  PHOENIX::AttachTask();
  // Before calling a Phoenix function, we need to reenter the Phoenix
  // scheduling.

  printf("Leaving group\n");
  this->Leave();
  // Leave the previously joined group.

  PHOENIX::ExitMainLoop();
  // Exit Phoenix main loop.
}

void CAccountDB::SendReply(char *msg, PHX_Id& srcid, ULONG reqid)
{
  int len = strlen(msg) + 1;
  PHX_Msg rply(len);
  rply.Put(msg, len);
  this->Reply(rply, srcid, reqid);
  // Put the string in a Phoenix message and send the reply.
}
```

```
void CAccountDB::Deposit(PHX_Msg& msg, PHX_Id& srcid, ULONG reqid)
{
  int nb;
  int amount;

  msg.Get(&nb, sizeof(int));
  msg.Get(&amount, sizeof(int));
  // Read the account number and amount from the incoming message.

  cout << eoln << "********************" << eoln << eoln <<
    "Deposit $" << amount << " on account " << nb << eoln;

  for(Pix i = fAccounts.first(); i != 0; fAccounts.next(i)) {
    // Look for the account in the database.
    if(fAccounts(i).Nb() == nb) {
      fAccounts(i).Deposit(amount);
      // Perform the deposit.
      this->SendReply(DEP_OK, srcid, reqid);
      // Send an OK reply.
      break;
    }
  }
  if(i == 0) {
    // We didn't find the account. We create it.
    printf("Creating account\n");
    fAccounts.add(CAccount(nb, amount));
    this->SendReply(DEP_NEW, srcid, reqid);
    // Send an NEW ACCOUNT reply.
  }
  if(this->FirstInGroup())
    SendStateToSinks();
  // The first member of the group sends the new state to the sinks.
}

void CAccountDB::Withdraw(PHX_Msg& msg, PHX_Id& srcid, ULONG reqid)
{
  int nb;
  int amount;

  msg.Get(&nb, sizeof(int));
  msg.Get(&amount, sizeof(int));
  // Read the account number and amount from the incoming message.

  cout << eoln << "********************" << eoln << eoln <<
    "Withdraw $" << amount << " on account " << nb << eoln;

  for(Pix i = fAccounts.first(); i != 0; fAccounts.next(i)) {
    // Look for the account in the database.
    if(fAccounts(i).Nb() == nb) {
      fAccounts(i).Withdraw(amount);
      // Perform the withdrawal.
      this->SendReply(WITH_OK, srcid, reqid);
      // Send an OK reply.
      break;
```

```
    }
  }
  if(i == 0) {
    // We didn't find the account. This is an error.
    printf("ERROR: Account does not exist\n");
    this->SendReply(WITH_KO, srcid, reqid);
    // Send an ERROR reply.
    return;
  }
  if(this->FirstInGroup())
    SendStateToSinks();
  // The first member of the group sends the new state to the sinks.
}

PHX_Msg *CAccountDB::GetState()
{
  // We put the state of all accounts in a Pheonix message.
  int size = sizeof(int) + (fAccounts.length() * 2 * sizeof(int));
  PHX_Msg *msg = new PHX_Msg(size);
  int len = fAccounts.length();
  msg->Put(&len, sizeof(int));
  for(Pix i = fAccounts.first(); i != 0; fAccounts.next(i)) {
    int nb = fAccounts(i).Nb();
    int amount = fAccounts(i).Amount();
    msg->Put(&nb, sizeof(int));
    msg->Put(&amount, sizeof(int));
  }
  return(msg);
}

void CAccountDB::PutState(PHX_Msg& state)
{
  // We get the state of all accounts from the message.
  int len;
  state.Get(&len, sizeof(int));
  for(int i = 0; i < len; i++) {
    int nb;
    int amount;
    state.Get(&nb, sizeof(int));
    state.Get(&amount, sizeof(int));
    fAccounts.add(CAccount(nb, amount));
  }
  printf("Received new state:\n");
  this->Dump(stdout);
}

void CAccountDB::SendStateToSinks()
{
  // Send the internal state (i.e. all accounts) to the sink members.
  PHX_Msg *msg = this->GetState();
  this->SendToSinks(*msg);
  delete msg;
}

void CAccountDB::Dump(FILE *f)
```

```
{
  // Dump the internal state of the objects (i.e. all accounts).
  for(Pix i = fAccounts.first(); i != 0; fAccounts.next(i))
    fprintf(f, "Account %4d contains %d\n", fAccounts(i).Nb(), fAccounts(i).Amount());
}

int main(int argc, char **argv)
{
  if(argc != 2) {
    printf("Usage: %s groupname\n", argv[0]);
    exit(1);
  }

  gName = argv[1];
  // Store the group name.

  PHOENIX::Init();
  // Initialize Phoenix.

  CAccountDB *accounts = new CAccountDB;
  // Create an AccountDB object.

  PHOENIX::MainLoop();
  // Call Phoenix main loop.

  delete accounts;
  PHOENIX::Exit();
  // Leave Phoenix cleanly.

  cout << "End of program" << eoln;
  return(0);
}
```

Annexe B

Code de l'interface client

```
#include "phoenix.h"
#include "debug.h"

#include <ctype.h>

#define DEPOSIT 'D'
#define WITHDRAW 'W'

#define _MODULE_ _APPL_

/***********************************************************
 * Type definitions
 ***********************************************************/

// A class for representing a bank office (from where you can
// deposit or withdraw money).

class CBankOffice : public PHX_Client {

public:
// Constructor & Destructor
  CBankOffice() : PHX_Client(ACTIVE) {}
  ~CBankOffice() {}

protected:
// Event handling
  void ViewChange(PHX_Group& grp); // Override
  // Called upon viewchange.

  void Body(); // Override
  // Thread associated with the object (the object is ACTIVE).

private:

  PHX_Group fGroup;
  // current state of the group of accounts databases.
};

/***********************************************************
 * Local Scope Variables
 ***********************************************************/

char *gName;
// Name of the group of accounts databases.

/***********************************************************
 * Class Methods
 ***********************************************************/

void CBankOffice::ViewChange(PHX_Group& grp)
```

```
{
  fGroup = grp;
  printf("\nReceived ViewChange:\n");
  grp.Dump(stdout);
  // Dump the new group composition to the screen.
}

void CBankOffice::Body()
{
  printf("Joining group\n");
  this->ClientJoin(gName);
  // Join the specified group as a client.

  PHOENIX::DetachTask();
  // Since we perform a blocking operation (reading a character from the
  // keyboard), we detach our thread from the Phoenix scheduling. That
  // way, we do not block other operations.
  while(1) {
    cout << eoln << "(D)eposit" << eoln << "(W)ithdraw" << eoln <<
      "(Q)uit" << eoln << eoln;

    String str;
    cin >> str;
    char c = toupper(str[0]);

    if(c == 'Q')
      break;

    if(c == 'D' || c == 'W') {

      cout << "Account nb: ";
      int nb;
      cin >> nb;
      cout << "Amount: ";
      int amount;
      cin >> amount;

      PHX_Msg msg(2 * sizeof(int));
      msg.Put(&nb, sizeof(int));
      msg.Put(&amount, sizeof(int));
      // Put the account number and amount in a Phoenix message.

      PHOENIX::AttachTask();
      // Before calling a Phoenix function, we need to reenter the Phoenix
      // scheduling...
      PHX_Reply *rply = this->Request(fGroup, msg, ALL_REPLIES, c);
      // Send a request to the group, waiting for all replies. 'c' is used
      // as the key of the message. It will be redirected in the correct entry
      // at the server side.
      PHX_List<PHX_Msg>& replies = rply->Replies();
      // Read the list of replies from the reply object. This call is blocking
      // if all replies have not yet been received.
      PHOENIX::DetachTask();
      // ...and exit it once more.
```

```
      cout << "Received " << replies.length() << " replies: " << eoln;
      for(Pix i = replies.first(); i != 0; replies.next(i))
cout << '\t' << replies(i).Data() << eoln;
      // Print the replies to the screen.
      delete rply;
    }
  }
  PHOENIX::AttachTask();

  printf("Leaving group\n");
  this->ClientLeave(gName);
  // Leave the previously joined group.

  PHOENIX::ExitMainLoop();
  // Exit Phoenix main loop.
}

int main(int argc, char **argv)
{
  if(argc != 2) {
    printf("Usage: %s groupname\n", argv[0]);
    exit(1);
  }

  gName = argv[1];
  // Store the group name.

  PHOENIX::Init();
  // Initialize Phoenix.

  CBankOffice *bank = new CBankOffice;
  // Create a BankOffice object.

  PHOENIX::MainLoop();
  // Call Phoenix main loop.

  delete bank;
  PHOENIX::Exit();
  // Leave Phoenix cleanly.

  return(0);
}
```

Annexe C

Code d'un point de consultation

```
#include "phoenix.h"
#include "debug.h"

#include <ctype.h>

#define _MODULE_ _APPL_

/**********************************************************
 * Type definitions
 **********************************************************/

// A simple class for representing an account

class CAccount {

public:
// Constructor & Destructor
  CAccount()
    { fNb = 0; fAmount = 0; }
  CAccount(int nb, int amount)
    { fNb = nb; fAmount = amount; }

// Operators
  bool operator ==(const CAccount& a) const
    { return(fNb == a.fNb && fAmount == a.fAmount); }
  // Compare 2 accounts for equality.

// Access
  int Amount() { return(fAmount); }
  // Return the amount of the account.
  int Nb() { return(fNb); }
  // Return the account number.

private:
  int fNb;
  // Account number.
  int fAmount;
  // Amount of money on the account.
};

// A class for representing a place where people can consult their
// accounts.

class CConsultationPoint : public PHX_Sink {

public:
// Constructor & Destructor
  CConsultationPoint() : PHX_Sink(ACTIVE) {}
  ~CConsultationPoint() {}
```

```
protected:
// Event handling
  void Receive(PHX_Msg& msg, PHX_Id& srcid, ULONG msgid); // Override
  // Called upon message reception.

  void Body(); // Override
  // Thread associated with the object (the object is ACTIVE).

  void Dump(FILE *f);
  // Dump the internal state of the objects (i.e. all accounts).

private:
  PHX_List<CAccount> fAccounts;
  // List of all account.
};

/************************************************************
 * Local Scope Variables
 ************************************************************/

char *gName;
// Name of the group of accounts databases.

/************************************************************
 * Class Methods
 ************************************************************/

void CConsultationPoint::Receive(PHX_Msg& msg, PHX_Id& srcid, ULONG msgid)
{
  while(fAccounts.length() > 0)
    fAccounts.remove_front();
  // Since we receive the state of all accounts, we need to free the
  // previous state.

  // Read the accounts from the incoming message and store them.
  int len;
  msg.Get(&len, sizeof(int));
  for(int i = 0; i < len; i++) {
    int nb;
    int amount;
    msg.Get(&nb, sizeof(int));
    msg.Get(&amount, sizeof(int));
    fAccounts.add(CAccount(nb, amount));
  }
  cout << eoln << "********************" << eoln << eoln <<
    "Received new accounts:" << eoln;
  this->Dump(stdout);
  // Dump the accounts to the screen.
}

/************************************************************
 * Class Methods
 ************************************************************/

void CConsultationPoint::Body()
```

```
{
  printf("Joining group\n");
  this->SinkJoin(gName);
  // Join the specified group as a sink.

  PHOENIX::DetachTask();
  // Since we perform a blocking operation (reading a character from the
  // keyboard), we detach our thread from the Phoenix scheduling. That
  // way, we do not block other operations.
  char c;
  do {
    cout << eoln << "(Q)uit" << eoln << eoln;
    cin >> c;
  } while(toupper(c) != 'Q');
  PHOENIX::AttachTask();
  // Before calling a Phoenix function, we need to reenter the Phoenix
  // scheduling.

  printf("Leaving group\n");
  this->SinkLeave(gName);
  // Leave the previously joined group.

  PHOENIX::ExitMainLoop();
  // Exit Phoenix main loop.
}

void CConsultationPoint::Dump(FILE *f)
{
  // Dump the internal state of the objects (i.e. all accounts).
  for(Pix i = fAccounts.first(); i != 0; fAccounts.next(i))
    fprintf(f, "Account %4d contains $%d \n", fAccounts(i).Nb(), fAccounts(i).Amount());
}

int main(int argc, char **argv)
{
  if(argc != 2) {
    printf("Usage: %s groupname\n", argv[0]);
    exit(1);
  }

  gName = argv[1];
  // Store the group name.

  PHOENIX::Init();
  // Initialize Phoenix.

  CConsultationPoint *cp = new CConsultationPoint;
  // Create a ConsultationPoint object.

  PHOENIX::MainLoop();
  // Call Phoenix main loop.

  delete cp;
  PHOENIX::Exit();
  // Leave Phoenix cleanly.
```

```
  return(0);
}
```