Plateformes de Développement d'Applications Distribuées

Travaux Dirigés N° 01

CCM

Préliminaires : installation de OpenCCM et de JacORB

L'installation de l'environnement de développement **OpenCCM 0.9 + JacORB 2.2 + J2SDK 1.4** sur les machines de l'ENIS se fera par les étudiants sur les postes Ubuntu. Il s'agit simplement de récupérer une archive précompilée préparée pour les TDs. Cette archive peut aussi être utilisée sur tout PC exécutant le système d'exploitation Ubuntu (10.04 ou 10.10).

Récupération et extraction de l'archive

Après avoir récupéré l'archive **`ENIS_CCM_Package.tgz`** (l'adresse sera communiquée au cours de la séance de TD), il faudra l'extraire. Ensuite, dans un terminal, saisir les commandes suivantes :

```
~% tar xzvvf /chemin/vers/ENIS_CCM_Package.tgz
```

À l'issue des deux commandes ci-dessus, on disposera d'un répertoire **`ENIS_CCM_Package`** contenant l'installation de **OpenCCM** ainsi que des outils de développement et d'exécution **JAVA 4** de SUN. On disposera aussi d'un fichier nommé **`env.sh`** qui permet de placer correctement les variables d'environnement nécessaires pour la compilation, le déploiement et l'exécution des applications réparties. Ce fichier doit être **"chargé" chaque fois** qu'un nouveau terminal (shell) est ouvert. Le chargement du fichier **`env.sh`** s'effectue, **dans le nouveau terminal**, à l'aide des commandes suivantes :

```
~% cd /chemin/vers/ENIS_CCM_Package
~% source env.sh
```

Exercice 1: Conception et Génération de code

Il s'agit dans cet exercice de réaliser une version orientée composant de l'application classique *« Le diner des philosophes »*. Six philosophes sont assis autour d'une table. Une fourchette est placée entre chaque deux philosophe. Un philosophe peut être dans divers états (*Mange, Réfléchit, A faim, Affamé* et *Mort*). Il ne peux manger que s'il dispose de deux fourchettes (celle située à sa gauche et celle située à sa droite).

La fourchette sera modélisée par un composant **`FourchetteManager`** ayant une facette de type **`Fourchette`** permettant aux philosophes de prendre et de rendre la fourchette (méthodes `prendre` et `liberer`).

Le philosophe sera modélisé par un composant **CCM** ayant deux réceptacles (une pour la fourchette gauche et une autre pour la fourchette droite. Il possède aussi une source d'évènement (de type `InfoEtat` comme vu dans le cours) permettant d'envoyer son état à un observateur externe de l'application.

L'observateur externe est, lui aussi, un composant connecté à tous les philosophes. Il possède un

puits d'évènement sur lequel il reçoit les différents états des philosophes.

L'application comprendra un observateur, 6 fourchettes et 6 philosophes: *Aristote, Descartes, Kant, Nietzsche, Platon* et *Socrate*.

Dans la suite de cet exercice, on effectuera la conception de l'application, la génération du code et sa compilation.

Tout au long du TD, on utilisera diverses commandes de **OpenCCM**. La plupart de ces commandes ont besoin d'un dépôt d'interface en cours d'exécution. Pour cela, il faut démarrer ce dépôt d'interface avec la commande suivante :

```
~% ir3_start
```

Cette commande place les informations nécessaire pour consulter et gérer le dépôt d'interface dans le dossier pointé par la variable d'environnement **${OpenCCM_CONFIG_DIR}**. Dans notre cas, il s'agit du dossier **ENIS_CCM_Package/OpenCCM_CONFIG_DIR**. Pour arrêter le dépôt d'interface, il suffit de saisir la commande suivante :

```
~% ir3_stop
```

Question 1 : Création du modèle IDL3

Créer, en vous inspirant des exemples du cours, un fichier **`DinerDesPhilosophes.idl3`** décrivant les 3 types de composants de l'application.

On vérifiera la validité du modèle à l'aide de la commande suivante :

```
~% idl3_check DinerDesPhilosophes.idl3
```

Question 2 : Génération d'une partie du code du conteneur

Dans la suite, on va effectuer diverses génération de code à partir du modèle **IDL3**. Pour mieux organiser les fichiers sources, nous choisissons de mettre tous les fichiers générés sous l'hiérarchie du dossier **`generated`**.

```
~% mkdir generated
```

Pour cette question, on créera une partie des squelettes utilisés par les conteneurs pour manipuler les composants. Ce code sera placé sous le dossier `skeletons` du dossier `generated`

```
~% mkdir generated/skeletons
```

La génération du code de ces squelettes se fait en deux étapes :

1) Chargement du modèle IDL3 dans le dépôt d'interfaces. Puisque le modèle **`DinerdesPhilosophes.idl3`** importe le paquetage **`Components`** de **CCM**, il faut indiquer à l'outil de chargement l'endroit où il trouvera le fichier `Components.idl` :

   ```
   ~% ir3_feed -I"${OpenCCM_HOMEDIR}/idl" DinerDesPhilosophes.idl3
   ```

2) Génération des fichiers du conteneur :

   ```
   ~% cd generated/skeletons
   ```

```
~% ir3_java ::DinerDesPhilosophes
~% cd ../..
```

À l'issue ce cette commande, beaucoup de fichiers gérant l'interaction entre le composant et son conteneurs seront générés.

Question 3 : Génération des fichiers IDL2

Nous allons maintenant générer des modèles **IDL2** à partir du modèle **IDL3** chargé dans le dépôt d'interfaces. Ces modèles serviront à générer le code du coté client ainsi que les interfaces locales du coté serveur. Le code **IDL2** sera généré dans le dossier **`generated/idl`** :

```
~% mkdir generated/idl
```

La génération du code **IDL2** se fait avec l'outil `ir3_idl2`. Il faut préciser à cet outil le nom du fichier cible :

```
~% ir3_idl2 -o ./generated/idl/DinerDesPhilosophes.idl ::DinerDesPhilosophes
```

On vérifiera la production automatique de deux fichiers : **`DinerDesPhilosophes.idl`** et **`DinerDesPhilosophes_local.idl`.** Le premier contient le code du coté client et le second contient les interface locales du coté serveur. On comparera contenus respectifs avec les règles de projections vues dans le cours.

Question 4 : Cadre d'implantation du composant (CIF)

En s'inspirant des exemples du cours, crée le fichier **`DinerDesPhilosophes.cidl`** qui décrit la composition des composants. **On choisira des composants monolithiques**. Le composant Observateur sera de type service. Les composants Philosophe et FourchetteManager seront de type session.

On vérifiera la validité de notre modèle d'implantation avec la commande suivante :

```
~% cidl DinerDesPhilosophes.cidl
```

Le fichier **CIDL** servira pour la génération automatique des interfaces **IDL** des exécuteurs de composants et de homes. La génération du code **IDL2** se fait avec l'outil **`cidl_cif`**. Cet outil produit, en plus des fichiers **IDL2** un ensemble de fichiers `.dep` qui sont très utiles lors de l'emballage des composants car ils permettent d'indiquer les classes exactes utilisés par un composants donnés. Ces fichiers seront produit dans le dossier **`generated/dependencies`** :

```
~% mkdir ./generated/dependencies

~% cidl_cif -o ./generated/idl/DinerDesPhilosophes_cif.idl \
      -d ./generated/skeletons \
      -dep ./generated/dependencies \
      -ipath DinerDesPhilosophes_local.idl \
            DinerDesPhilosophes.cidl
```

Question 5 : Génération de code JAVA

À partir des fichiers **IDL2**, on générera automatiquement le code JAVA. Les fichiers JAVA seront

générés dans le dossier generated/stubs qui devra être créé préalablement :

```
~% mkdir ./generated/stubs
```

Les commandes pour générer ce code sont les suivantes :

1) Pour le code du coté client :

```
~% idl2java -d ./generated/stubs \
      -I./generated/idl/ \
      -I${OpenCCM_HOMEDIR}/idl \
      ./generated/idl/DinerDesPhilosophes.idl
```

2) Pour les interfaces locales du serveur :

```
~% idl2java -d ./generated/stubs \
      -I./generated/idl/ \
      -I${OpenCCM_HOMEDIR}/idl \
            ./generated/idl/DinerDesPhilosophes_local.idl
```

3) Pour les interfaces de exécuteurs :

```
%~ idl2java -d ./generated/stubs \
      -I./generated/idl/ \
      -I${OpenCCM_HOMEDIR}/idl \
            ./generated/idl/DinerDesPhilosophes_cif.idl
```

Question 6 : Squelettes d'implantation et compilation du code généré

Pour pouvoir compiler le code généré, il va falloir fournir des squelettes pour l'implantation des composants et des Homes. Heureusement, les outils de **OpenCCM** nous permettent de créer des fichiers d'implantation compilables. On les placera dans le dossier **`src`** :

```
~% mkdir ./src
```

Deux méthodes existent pour produire des squelettes incomplètes d'implantations. Les commandes pour produire les implantations sont les suivantes :

1) À partir du modèle IDL3 en utilisant la commande **`ir3_jimpl`**. Cette Méthode ne peut générer le code que pour des composants monolithiques :

```
~% cd src
~% ir3_jimpl ::DinerDesPhilosophes
~% cd ..
```

2) À partir de la description CIDL en utilisant la commande **`cif_jimpl`**. Cette méthode permet de générer du code pour des composants monolithiques ainsi que pour des composants segmentés. **C'est elle que l'on va choisir pour produire les implantation incomplètes** :

```
~% cif_jimpl -d ./src DinerDesPhilosophes.cidl
```

Nous disposons maintenant de tout le code nécessaire pour compiler nos composants. La production

des fichiers de classes se fera dans le dossier **`generated/classes`** qu'il faudra créer :

```
~% mkdir ./generated/classes
```

1) Compilation des souches :

```
~% javac -d ./generated/classes \
      -sourcepath ./generated/stubs \
            ./generated/stubs/DinerDesPhilosophes/**/*.java
```

2) Compilation des squelettes :

```
~% javac -d ./generated/classes \
      -sourcepath ./generated/skeletons \
            ./generated/skeletons/DinerDesPhilosophes/**/*.java
```

3) Compilation des implantations :

```
~% javac -d ./generated/classes \
      -sourcepath ./src \
            ./src/DinerDesPhilosophes/**/*.java
```

Exercice 2: Emballage, Assemblage et Déploiement

Il s'agit dans cet exercice d'effectuer l'emballage, l'assemblage et le déploiement de notre application (*Diner des Philosophes*) et de l'exécuter en utilisant les outils de la plateforme **OpenCCM**.

Il va falloir, pour cela, télécharger l'archive **`DinerDesPhilosophes.tar.gz`**. L'emplacement de cette archive sera donné au cours de la séance de TD. Cette archive contient les éléments suivants :

1) Un dossier `src/` qui contient les implantations des composants complétés pour réaliser une interface graphique pour l'application. On remarquera notamment l'ajout de la classe `PhilosophePanel` qui illustre qu'il est possible pour le développeur d'ajouter ses propres classe pour finir l'implantation des composants.
2) Un dossier `ressources/` qui contient les images nécessaires à l'interface graphique.
3) Un dossier `META-INF/` qui contient les descripteurs de paquetages, de composants et d'assemblage de l'application.
4) Un fichier `build.xml` utilisable avec l'outil de construction automatique `ant`. Ce fichier permet d'exécuter plus facilement les actions effectuées dans l'exercice 1 de ce TD. Ce fichier utilise des propriétés déclarées dans un second fichier `diner.properties`. On disposera particulièrement des cibles suivantes pour effectuer les différentes tâches de construction de l'application :
   1. **etape0** : nettoyage de l'application
   2. **etape1** : initialisation; notamment, on créera les dossiers nécessaires dans l'hiérarchie `generated/`
   3. **etape2** : génération automatique de code (IDL2, souches et squelettes)
   4. **etape3** : compilation des souches, squelettes et implantations
   5. **etape4** : archivage de l'application. Plus précisément, on construira une archive JAR pour chaque composant.

      On construira également une archive CAR pour chaque composant. Cette dernière archive contient l'archive JAR du composant correspondant ainsi que les descripteur de paquetage logiciel (***CORBA Software Descriptor*, .csd**) et le descripteur de composant (***CORBA Component Descriptor*, .ccd**).

      Finalement, on construira une archive JAR pour toute l'application ainsi qu'une archive AAR d'assemblage qui contient, entre autres, le descripteur d'assemblage de l'application (***Component Assembly Descriptor*, .cad**).
5) Deux scripts `deploy` et `stop` qui permettent de lancer/arrêter le déploiement de l'application.

Question 1 : Compilation de l'application avec les nouvelles implantations

Après avoir déplacé les fichiers IDL et CIDL dans le dossier `DinerDesPhilosophes/`, générer les souches et les squelettes et les compiler en utilisant la commande suivante :

```
%~ ant etape2 etape3
```

Résoudre les éventuels problèmes liés à la manière adoptée pour nommer les modèles de l'application.

Question 2 : Descripteurs XML

Pour pouvoir archiver l'application. Il est nécessaire de créer les différents descripteurs (de paquetage, de composant et d'assemblage). **OpenCCM** dispose d'un outil graphique pour créer ces fichiers XML (**`ccm_assembling`**).

Utiliser cet outil pour explorer le contenu des différents fichiers présents dans le dossier `META-INF/`. On remarquera notamment les éléments suivants :

1) La description des interfaces de composants présentes dans les descripteurs de composants (**.ccd**)
2) Les connections entre les instances de composants présentes dans le descripteur d'assemblage (**.cad**)
3) L'allocation de ces instance sur les serveurs de composants

Question 3 : Emballage et Assemblage

Créer les différentes archives de l'application en utilisant la commande suivante :

```
%~ ant etape4
```

On disposera à la fin de cette commande d'un dossier nommé `archives/` contenant les archive JAR, CAR et AAR de l'application.

Explorer le contenu de cette archive avec la commande `unzip`.

Question 4 : Déploiement et exécution de l'application

Après avoir exploré le contenu des scripts `deploy` et `stop`, déployer votre application en exécutant :

```
%~ deploy
```

L'exécution de ce script va effectuer les opérations suivantes :

1) Installation de la plateforme CCM : crée le dossier `${OpenCCM_CONFIG_DIR}/`
2) Démarrage du service de nommage : ce service est nécessaire pour la communication entre les composants
3) Démarrage d'un serveur de composants : ce serveur va abriter les différents conteneurs
4) Démarrage d'un nœud principal : Ce nœud contiendra, entre autres, les homes de composants
5) Déploiement et début de l'application : l'exécution de l'application sera automatiquement effectuée après le déploiement.

Pour arrêter l'application, il suffit d'exécuter le script `./stop` qui effectuera dans l'ordre inverse l'arrêt des tâches lancées par le script `./deploy`.