Rapport de projet CCS

Alexis BONNIN Ny Sitraka FIDIMIHAJAMANANA

Decembre 2017

1 Introduction

Dans le cadre du module *Architectures et style d'architectures* du M2 ALMA, ce projet a pour objectif d'implémenter un modèle Client-Serveur à partir du métamodèle CCS (Component and Connector Style). Le métamodèle a été construit à l'aide de la technologie EMF (ecore), ensuite, par héritage, nous avons mis en place le modèle Client-Serveur. Enfin, la dernière partie consiste à élaborer une implémentation du modèle précédent en code Java.

2 M2 : Le métamodèle CCS

Le métamodèle *Component & Connector Style* permet de définir une syntaxe permettant de créer des modèles représentant la réalité. Le schéma suivant décrit la syntaxe du métamodèle CCS à l'aide d'ecore :

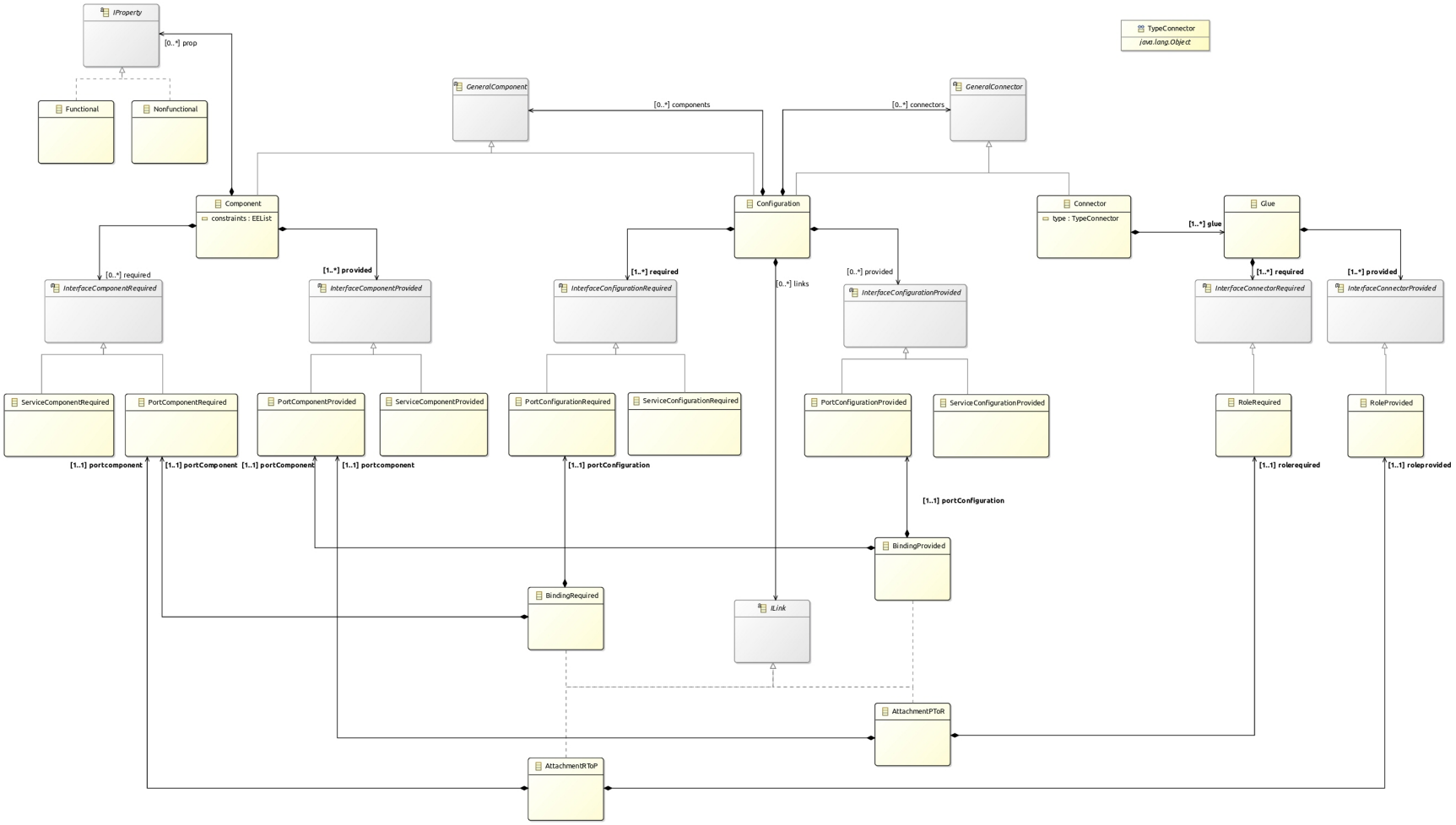
IProperty
[0..*] prop
Functional
Nonfunctional
TypeConnector
java.lang.Object
GeneralComponent
[0..*] components
[0..*] connectors
GeneralConnector
Component
constraints : EEList
Configuration
Connector
type : TypeConnector
[1..*] glue
Glue
[0..*] required
[1..*] provided
[1..*] required
[0..*] links
[0..*] provided
[1..*] required
[1..*] provided
InterfaceComponentRequired
InterfaceComponentProvided
InterfaceConfigurationRequired
InterfaceConfigurationProvided
InterfaceConnectorRequired
InterfaceConnectorProvided
ServiceComponentRequired
PortComponentRequired
PortComponentProvided
ServiceComponentProvided
PortConfigurationRequired
ServiceConfigurationRequired
PortConfigurationProvided
ServiceConfigurationProvided
RoleRequired
RoleProvided
[1..1] portcomponent
[1..1] portComponent
[1..1] portComponent
[1..1] portcomponent
[1..1] portConfiguration
[1..1] portConfiguration
[1..1] rolerequired
[1..1] roleprovided
BindingProvided
BindingRequired
ILink
AttachmentPToR
AttachmentRToP

Figure 1: *M2*

2.1 Présentation du schéma

Cette partie présente les principales classes du métamodèle.

2.1.1 Component

Cette classe représente un composant. Elle possède :

- Une ou plusieurs propriétés. Les propriétés sont de type *fonctionnel* ou *non fonctionnel*.
- Deux interfaces : une interface requis (Required) et une interface fournie (Provided). Ces deux interfaces sont elles mêmes composées de ports et de services. Celles-ci permettent de mettre à disposition de connecteurs des données ou des fonctions.

2.1.2 Connector

Cette classe représente un connecteur. Celui-ci est composé d'une ou plusieurs glues, elles-même composées d'interfaces (requis et fournies). Chacune des interfaces possède un rôle, correspondant à l'entrée et à la sortie du connecteur. Dans cette représentation du CCS, une glue permet donc de relier un rôle requis à un rôle fourni.

2.1.3 Configuration

La classe configuration regroupe les différents composants et connecteurs à l'aide du pattern composite (expliqué dans la partie 2.2.1). La configuration possède également deux interfaces: requis et fournie. Enfin, celle-ci contient les différents *attachment* et *binding* existants dans la configuration. La configuration permet donc de gérer l'ensemble des communications entre les composants et les connecteurs.

2.1.4 Links

L'interface *ILink* représente les liens entre les différentes entités présentées ci-dessus. Elle est implémentée par deux types de lien :

- Binding : lien entre un port de configuration et un port de composant, les deux liens sont de même type (fournis ou requis).
- Attachment : lien entre un port de composant et un rôle de connecteur, les deux liens sont de type différents (fournis ou requis).

2.2 Choix d'implémentation

2.2.1 Le composite

Le pattern composite est mis en place sur les classes *Component* et *Connector*. L'objectif est de concevoir un composant ou un connecteur qui contient un ou plusieurs objets similaires, celui-ci devient alors une configuration. Cela permet de manipuler un composant ou un connecteur comme une configuration.

2.2.2 Provided / Required

Le type (requis / fourni) d'une interface est défini par une séparation de classe : une classe *interfaceProvided* et une classe *interfaceRequired*. Chaque port, rôle ou service héritant de l'une de ces interfaces ne peut communiquer que dans un sens (défini par un lien d'attachement ou de binding). Cela permet d'assurer la cohérence d'un modèle concernant la liaison de composants, de connecteurs et de configurations.

2.2.3 Les glues

Afin de permettre une communication bi-directionnelle (lien entre deux rôles de type différent), nous avons fait le choix d'utiliser deux *Glue* au niveau des *Connector* . Un autre choix aurait été de créer deux *Connector* plutôt que de relier deux Glues à un même *Connector*.

3 M1 : Le modèle Client-Serveur

Afin de mettre en place le modèle Client-Serveur, nous avons utilisé le M2 précédemment décrit. A l'aide de l'héritage, les classes de ce modèle seront conformes à celles du méta-modèle M2. En effet, les classes du M1 sont réalisées en instanciant celles du M2. Nous avons utilisé la technologie EMF pour mettre en place ce concept. Deux parties du diagramme seront présentées ci-après, les autres parties étant disponible dans l'annexe.

3.1 La connexion Client-Server

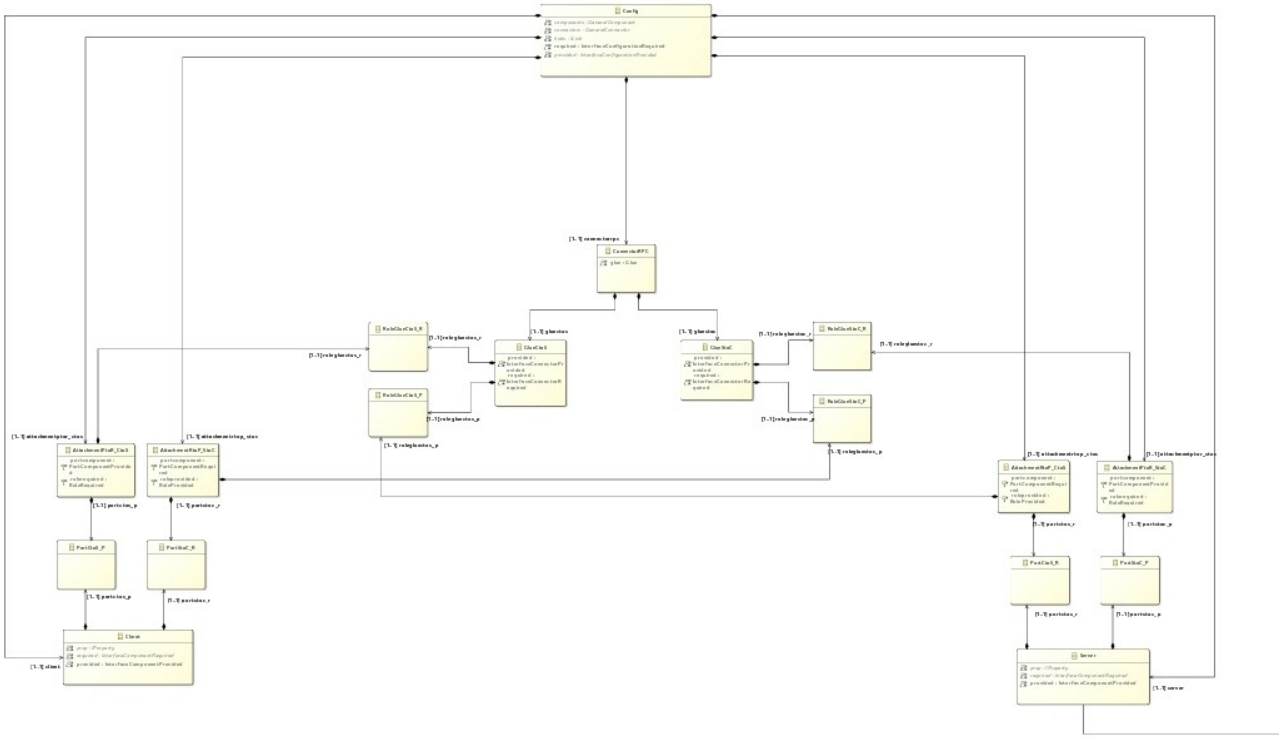

Figure 2: *M1- Connection Client Server*

Dans le schéma ci-dessus, nous avons un *Client* et un *Server* qui héritent de *Component*. Ces deux composants sont liés par un *ConnectorRPC* qui est de type *Connector*. Nous avons mis en place deux *Glues* qui représentent la communication bi-directionnelle entre le client et le serveur. Ces deux Glues sont intégrées dans le *ConnectorRPC*. Ensuite, pour établir la connexion entre un composant (Client ou Server) et un Connecteur, nous avons mis en place des liens de type *Attachment*. Un *Attachment* sert à lier un port du composant à un rôle du connecteur à travers la glue correspondante. Par exemple, nous avons *AttachmentPtoR_CtoS* qui lie *PortCtoS_P* à *RoleGlueCtoS_R*. Il est à remarquer que l'*Attachment* se fait entre un port et un rôle de différent types (Required/Provided).

La connexion entre le serveur et le client est un exemple de connexion bi-directionnelle. Les connexions entre *ConnectionManager* et *SecurityManager*, *SecurityManager* et *Database* ou encore entre *Database* et *ConnexionManager* sont réalisés de façon similaire, elle ne seront donc pas détaillées ici.

3.2 Lien entre ServerConfig et ConnectionManager

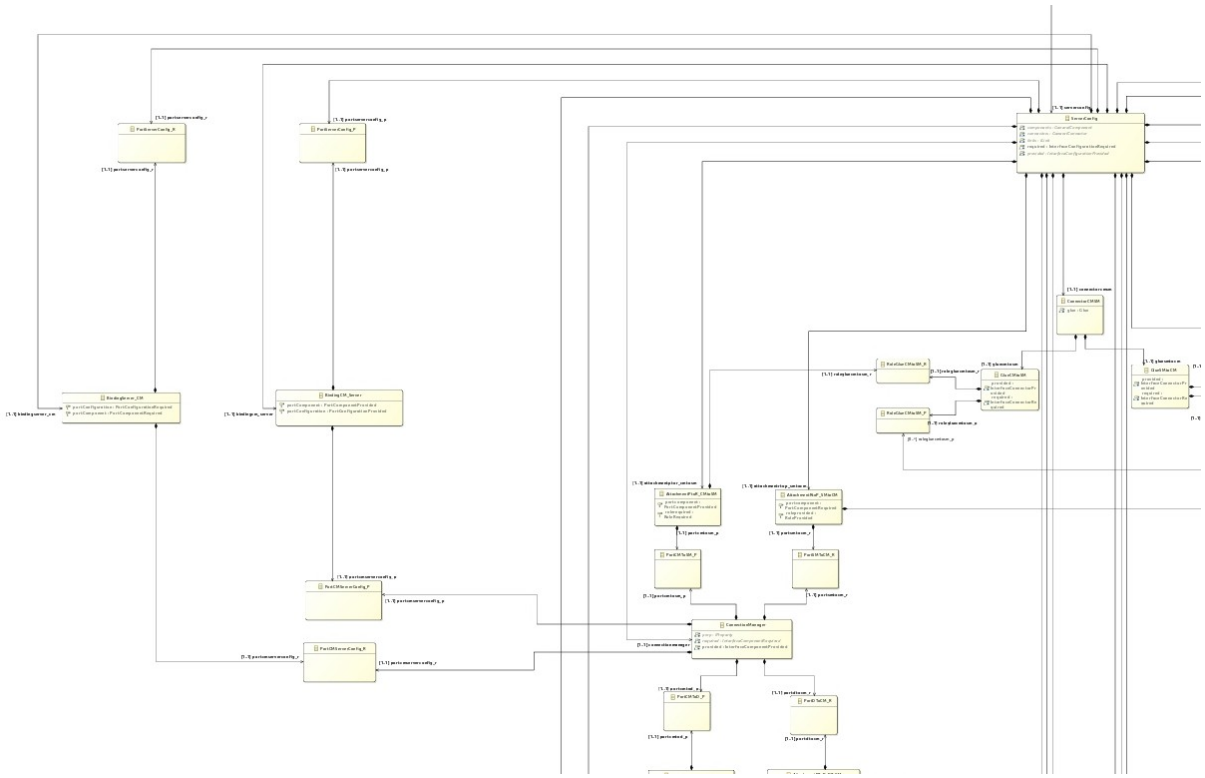

Figure 3: *M1- Binding*

Le lien entre *ServerConfig* et *ConnectionManager* se fait à l'aide de deux *Bindings* : **BindingServer_CM** et **BindingCM_Server**. En effet, la communication doit être bi-directionnelle et de ce fait, le *BindingServer_CM* s'occupe de l'acheminement des données du Serveur vers le ConnectionManager et *le BindingCM_Server* se charge de l'inverse.

3.3 Lien entre la configuration générale et la configuration du server

Figure 4: *M1- Référence server/serverConfig*

Pour établir la connexion entre le serveur et sa configuration, le *server* possède une référence vers le *serverConfig*. Ce lien permet au *server* d'accéder directement à l'interface de sa configuration pour récupérer

ou envoyer des données (communication permise dans les deux sens). Le détail de ce lien sera présenté dans la partie suivante (section 4).

4 M0 : L'implémentation du Client-Serveur

4.1 Présentation du code

Dans cette partie, une implémentation du modèle Cient-Server a été réalisée à l'aide du langage JAVA. Les classes sont directement créées à partir du modèle (M1), on retrouve donc un lien implicite entre le métamodèle (M2) et l'implémentation du code. Le projet Java *CCS-Implementation* est organisé en trois "packages" :

- **config** : ce package contient les classes principales qui forment la configuration telles que *Client*, *Config*, *ConnectorRPC* ainsi que le *Server*.
- **serverconfig** : ce package contient les classes *ConnectionManager*, *Database*, *SecurityManager*, *Server-Config* ainsi que les connecteurs.
- **utils** : ce dernier package regroupe les classes "outils" issus du modèle Client-Server. Il s'agit des classes représentant les liens (Attachment, Binding), la Glue, les types (required/provided), les ports, etc. Ces classes permettent de factoriser le code relatif aux éléments contenus dans les composants, les configuration et le connecteurs.

Nous avons fait le choix d'implémenter des singletons pour les deux configurations. Cela permet de simplifier les appels de méthodes depuis un composant. Une autre possibilité aurait été d'utiliser le pattern observer pour gérer les envois de messages. Cette possibilité étant plus complexe à mettre en oeuvre, nous avons opté pour la première solution.

Le scénario de test est le suivant : le premier message (String) envoyé au serveur doit être le mot de passe du serveur. Le message est traité par le connecteur RPC et est ensuite transmis au serveur. Le serveur transmet ensuite le message à sa configuration, qui le transmet ensuite au ConnectionManager via un binding. Le ConnectionManager vérifie si le client est connecté ou non. Si la connection est validée, le ConnectionManager renvoi un message à destination du client. Sinon, le ConnectionManager transmet le message au SecurityManager qui le transmet à la database pour vérifier le mot de passe. Si le mot de passe est correct, une réponse est transmise dans le sens inverse jusqu'au ConnctionManager qui active la connexion et qui renvoie une réponse au client.

5 Conclusion

En résumé, nous sommes parti d'un métamodèle *Component & Connector Style* pour établir un modèle *client-Server* par héritage, pour enfin aboutir à un code JAVA implémentant le Client-Server. La partie la plus difficile a été d'élaborer le modèle Client-Server à partir du métamodèle CCS. Ceci est dû au nombre élevé de classes à construire tout en respectant le M2. Néanmoins, grâce à ce projet, nous avons compris le principe du *CCS* et l'implémentation du Client-Serveur.