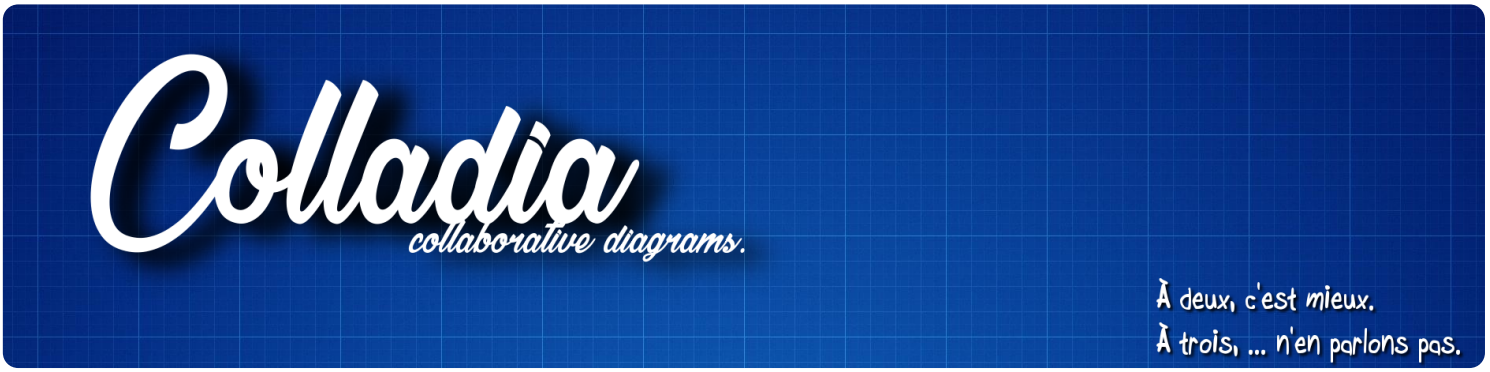


Jean Vintache
Florian Impellettieri
Charles Menier
Marouane Hammi
Adrien Jacquet

12 juin 2016

utc
Université de Technologie
Compiègne

Table des matières

1 Introduction

Ce document constitue le rapport de notre projet NF28/IA04. Au cours de ce projet, l'objectif a été de proposer à des utilisateurs une expérience d'interaction agréable, ceci à travers une interface graphique ergonomique adaptée aux besoins utilisateurs. Le second objectif était d'utiliser, d'une manière ou d'une autre, les fonctionnalités des systèmes multi-agents.

Dans le cadre de ce projet, nous avons choisi de réaliser un logiciel de tableau blanc collaboratif, qui permettrait de générer diverses formes, pour réaliser des diagrammes ou des schémas. L'architecture réalisée est composée de deux parties distinctes : la partie client qui présente les différentes interfaces graphiques correspondant au cours de NF28 et la partie serveur qui comporte un système multi-agents relatif au cours de IA04. Le projet a été réalisé sous Android natif ce qui permet de l'utiliser sur diverses tablettes et smartphones. La partie serveur, quant à elle, utilise la plateforme JADE pour faire fonctionner les agents.

2 Interface client

2.1 Intérêt principal

On dit souvent qu'un dessin vaut mieux qu'un long discours. Partant de ce principe, l'objectif de Colladia est de proposer aux utilisateurs une expérience leur permettant de pouvoir dessiner, ensemble, chacun sur leur propre appareil mobile.

Les utilisateurs peuvent ainsi modifier le document simultanément, et profiter des interactions tactiles qu'offrent ces supports. Que ce soit en déplacement ou en réunion, il n'est pas toujours possible d'avoir un ordinateur avec soi. Dans ce genre de situation, Colladia se révèle particulièrement intéressant.

2.2 Utilisateurs visés

Les premiers utilisateurs visés seront notamment les étudiants et les professionnels qui rencontrent la nécessité de réaliser des diagrammes ou des schémas dans leur quotidien. L'application est axée sur ces utilisateurs pour proposer des éléments de diagramme adaptés à leur besoins.

2.3 Étude des utilisateurs (interviews, enquêtes)

Afin de déterminer les besoins des utilisateurs de façon pertinente, un sondage a été réalisé auprès des étudiants de l'UTC venant de diverses branches. Les questions étaient relatives aux types d'outils avec lesquels ils sont habituées à réaliser des schémas.

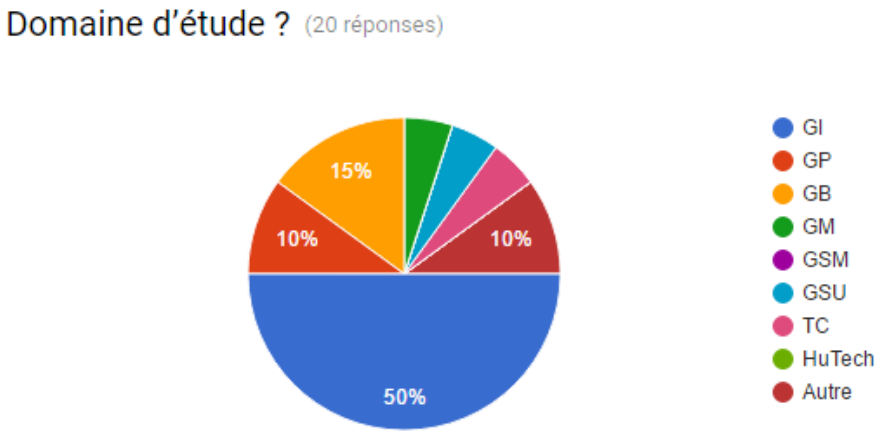


FIGURE 1 – Sondage - Répartition par branches

On observe que les avis sur les outils déjà existants sont assez mitigés. En effet, les inconvénients les plus cités sont le manque de personnalisation et le retour en arrière. D'un autres côté, les avantages cités sont la gratuité et la simplicité d'utilisation.

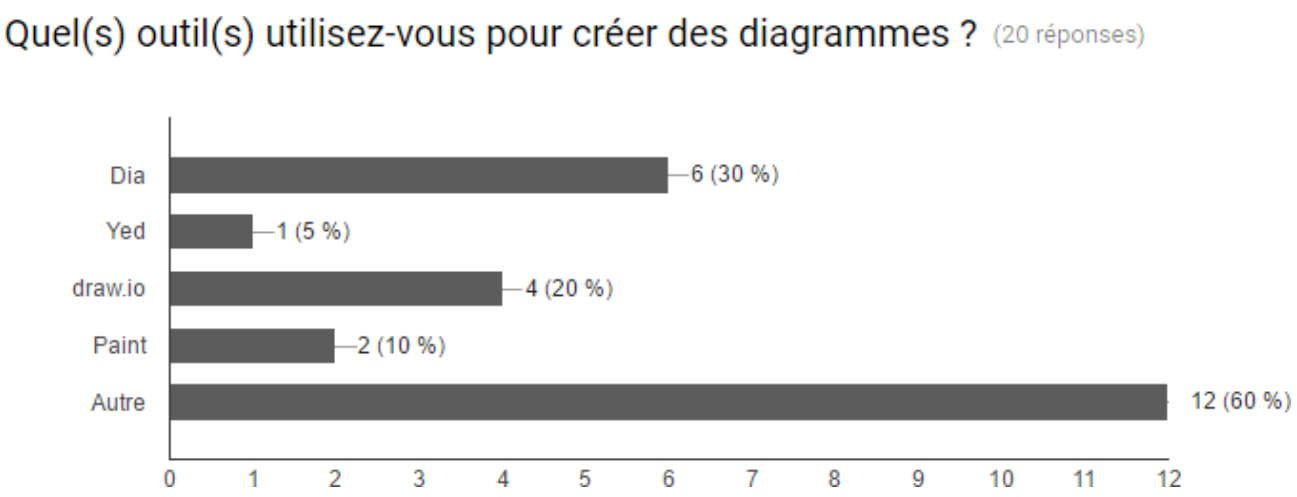


FIGURE 2 – Sondage - Outils utilisés pour la réalisation de diagrammes

FIGURE 3 – Sondage - Critiques des outils existants

On peut également voir que les avis quant à l'utilisation d'un appareil mobile pour travailler sont très partagés, on se retrouve à 50% pour et 50% contre. Enfin, l'idée d'une application permettant le travail collaboratif sur appareil mobile est aussi partagée mais tend à pencher pour un avis positif.

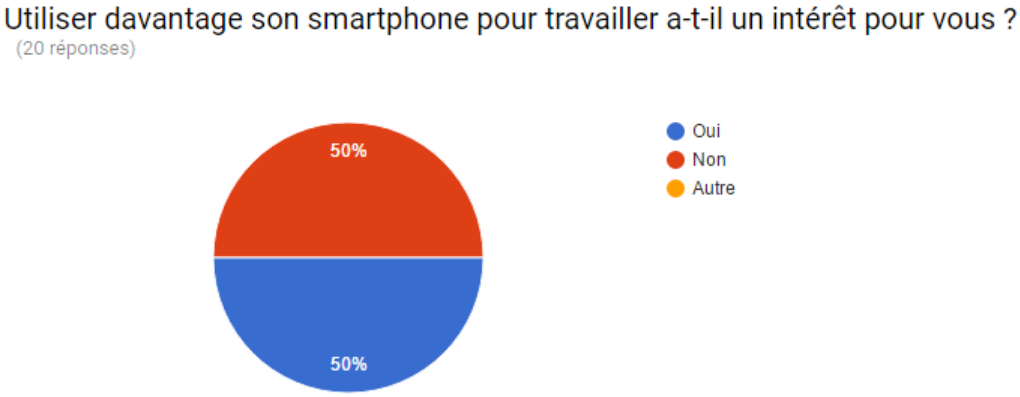


FIGURE 4 – Sondage - Critiques des outils existants

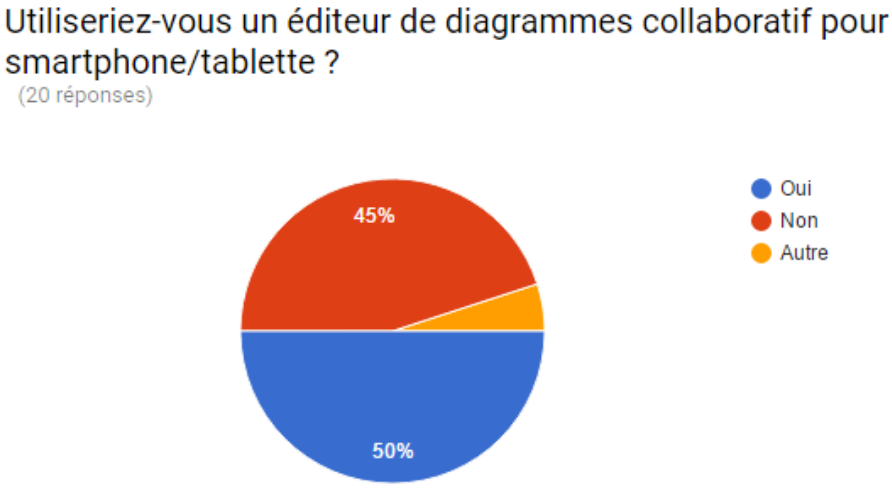


FIGURE 5 – Sondage - Avis sur Colladia

2.4 Étude de quelques logiciels concurrents

Lors de nos recherches, deux applications semblaient réellement en concurrence avec notre solution, à savoir draw.io de Google et DrawExpress de DrawExpress Inc.

2.4.1 Draw.io

Néanmoins draw.io, bien que collaboratif n'utilise pas pleinement les fonctionnalités tactiles des supports mobiles.

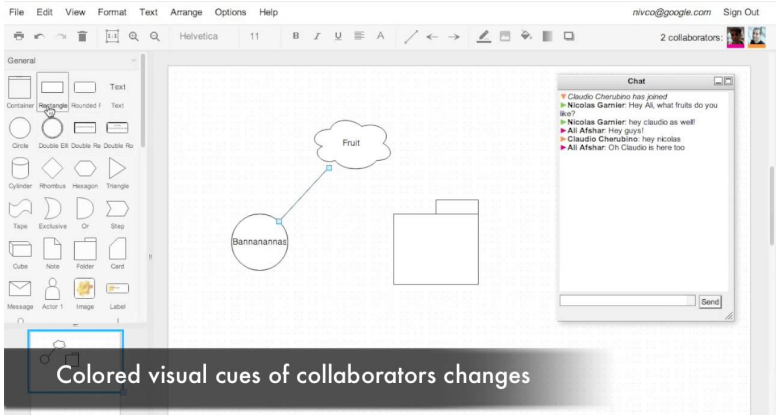


FIGURE 6 – draw.io - Édition en collaboration

2.4.2 DrawExpress

DrawExpress propose un éditeur de diagramme sur mobile. Il est cross-plateforme (IOS, Android et BlackBerry) mais il ne permet pas de travailler à plusieurs simultanément.

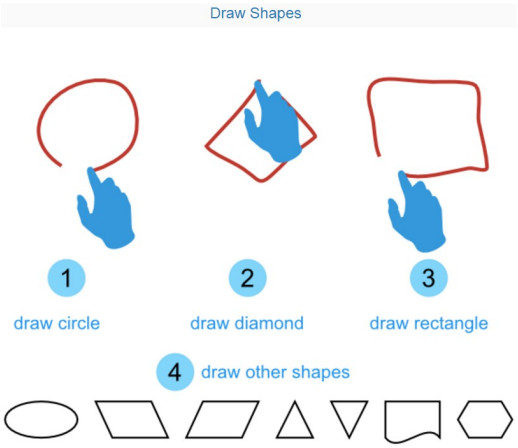


FIGURE 7 – Aperçu des interactions tactiles de DrawExpress - Reconnaissance de forme

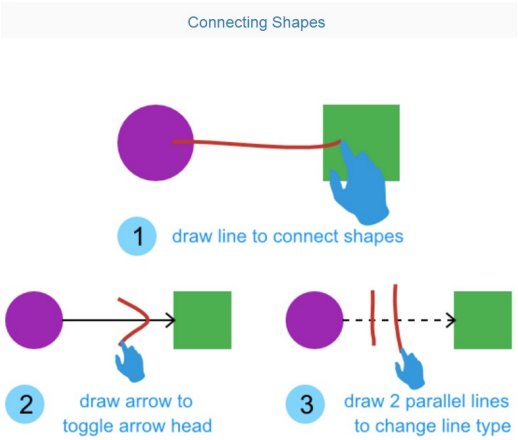


FIGURE 8 – Aperçu des interactions tactiles de DrawExpress - Tracé de lignes

Cette application constitue une bonne base d'inspiration pour Colladia puisqu'elle utilise particulièrement les interactions tactiles permises par les smartphones afin de créer, à partir de gestes simples à main levée, des formes prédéfinies qui constituent le diagramme. De cette manière, l'ergonomie est optimisée pour les écrans tactiles et de petite taille.

2.4.3 Positionnement

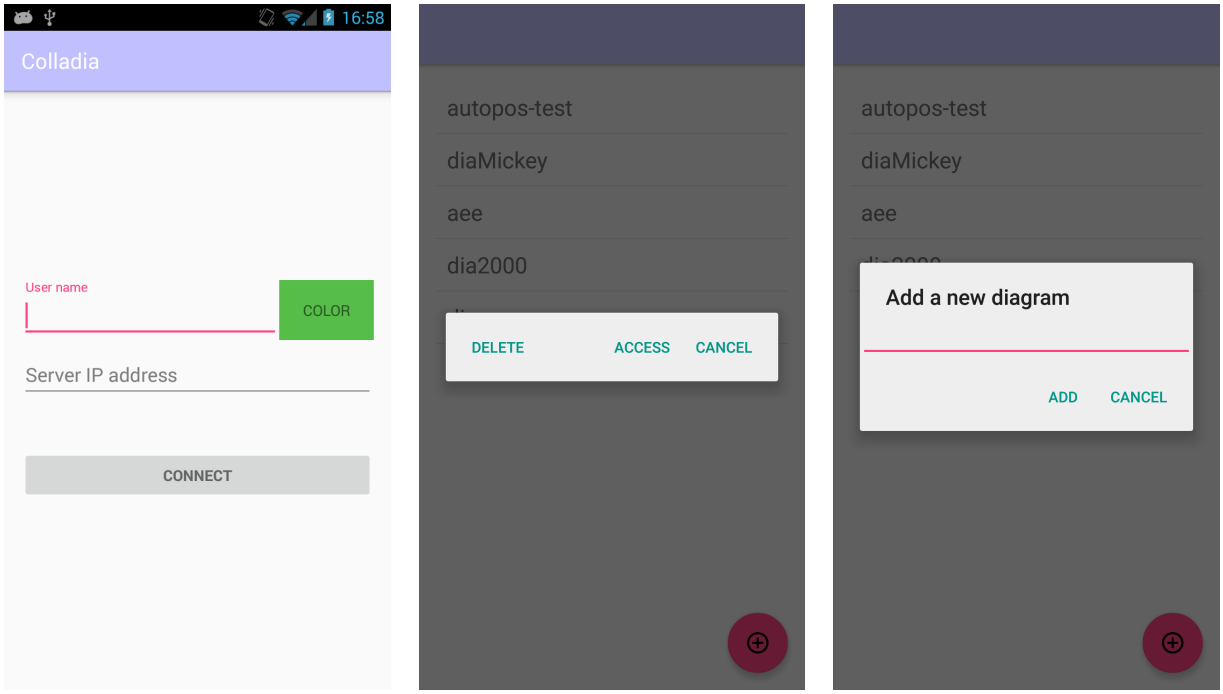
Le but de Colladia est donc de proposer les deux points forts ces deux logiciels concurrents, à savoir le travail collaboratif et les interactions tactiles.

2.5 Fonctionnalités

Dans cette partie, on rendra compte des fonctionnalités prévues du cahier des charges, en précisant les modifications qu'elles ont subies lors de leur implémentation.

2.5.1 Fonctionnalités générales

- La connexion au serveur avec un pseudo a été correctement réalisée. L'utilisateur a la possibilité d'entrer un pseudo et de préciser l'adresse du serveur auquel il faut se connecter pour créer des diagrammes. Il lui est aussi permis de sélectionner une couleur personnalisée.
- Dans la vue listant les diagrammes, il suffit de sélectionner le diagramme voulue et de cliquer sur "Access" pour le rejoindre. Il est aussi possible de supprimer le diagramme.



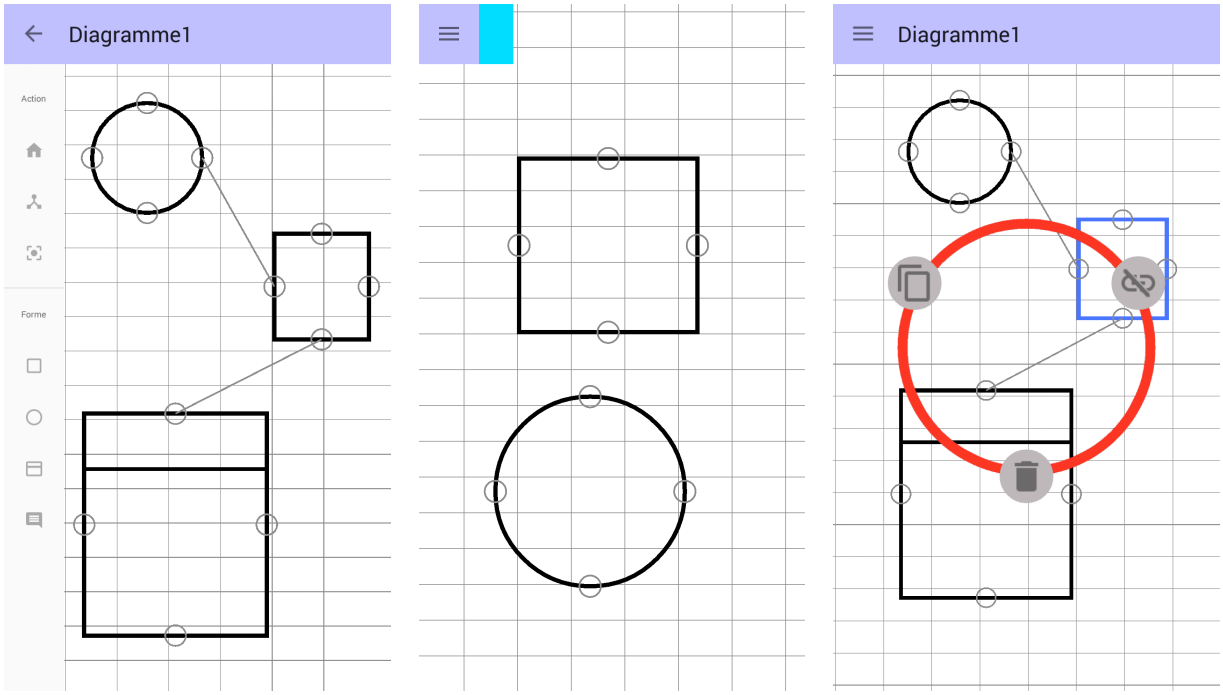
(a) Connexion (b) Liste des diagrammes (c) Création

FIGURE 9 – Colladia - Connexion à l'application et sélection d'un diagramme

- Inspiré des mouvements de déplacements habituelles sur les interfaces tactiles le mouvement de "swipe" permet de déplacer la vue. Il est aussi possible d'utiliser le mouvement de "pinch" pour effectuer un zoom.
- Les fonctionnalités d'ajout, de suppression et de modification d'éléments ont été implémentées et seront détaillées par la suite.

2.5.2 Fonctionnalités tactiles implémentées

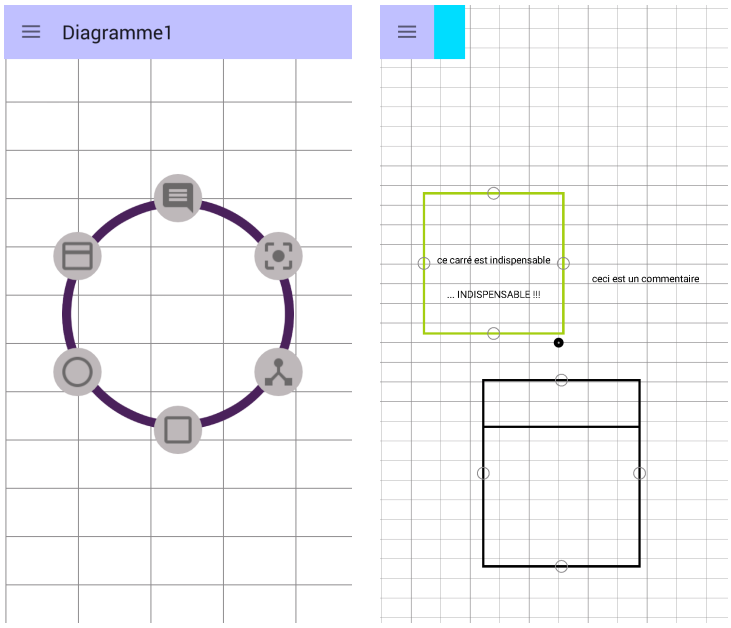
- Icône transparent dans le coin pour ouvrir le menu
- Ajout de formes prédéfinies (Classes UML, cercles, carrés, rectangles,...) en utilisant le menu principal ou le menu contextuel.
- Un appui long sur un élément permet d'ouvrir un menu contextuel spécifique à l'élément :
  - pour permettre de supprimer l'élément
  - pour permettre de dupliquer l'élément
  - pour permettre de supprimer les liens sortants



(a) Menu principal (b) Insertion de formes prédéfinies (c) Menu contextuel

FIGURE 10 – Colladia - Menu latéral et menu contextuel

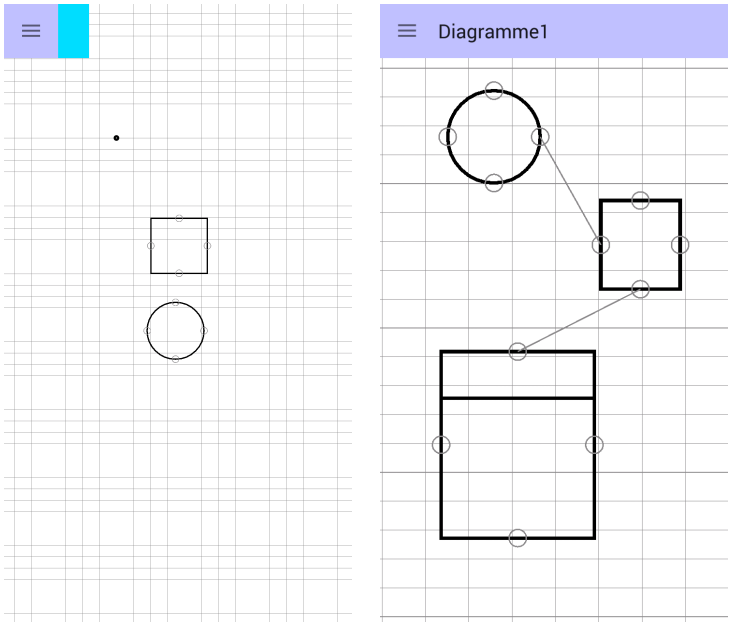
- Un appui long dans le vide permet d'ouvrir un menu pour ajouter des éléments et réaliser des actions :
  - Ajout de formes prédéfinies
  - Repositionner la vue au centre du diagramme
  - Position automatique des éléments
- Un "double-tap" sur un élément permet d'insérer du texte dans la forme.
- Un mouvement de "slide" permet de se déplacer sur la zone de dessin. Si un élément est sélectionné alors on déplace ce dernier.
- Un appui sur un élément le sélectionne et le colore de la couleur de l'utilisateur. En appuyant de nouveau sur un autre élément ou sur un espace vide, on dé-sélectionne l'élément, et il reprend sa couleur normale.



(a) Menu contextuel principal (b) Selection

FIGURE 11 – Colladia - Menu contextuel principal et sélection d'éléments

- Le mouvement de "pinch" habituel pour gérer les effets de zoom. Si un élément est sélectionné, ce mouvement permet d'effectuer le redimensionnement.
- Pour pouvoir représenter les flux dans les diagrammes, l'utilisateur peut sélectionner les ancres qui sont au nombre de 4 pour chaque élément, puis en restant appuyé sur l'écran et en rejoignant une seconde ancre un lien sera créée.



(a) Zoom arrière (b) Liens

FIGURE 12 – Colladia - Zoom et liens entre les éléments

2.5.3 Fonctionnalités tactiles non-implémentées

Dans la version actuelle de l'application, la gestion des collaborateurs n'est pas effectuée. Il n'est donc pas possible de voir les vues des collaborateurs, ni de connaître la liste des utilisateurs connectés à un instant précis. Néanmoins lorsqu'un élément est sélectionné, il prend la couleur spécifique de l'utilisateur. Ainsi il est possible de déterminer lorsqu'un item est modifié par un collaborateur. La gestion des différents niveaux de plans n'est pas proposé, ainsi que le fait d'attirer l'attention de l'utilisateur sur un élément en particulier, même si cette fonctionnalité est retrouvé lorsque l'on sélectionne l'élément, ce qui la colore.

2.5.4 Fonctionnalités tactiles supplémentaires

Afin de donner une expérience plus intéressante pour l'utilisateur et d'utiliser les capacités du système multi-agents du serveur, il est possible de :

- Replacer la vue de l'utilisateur au centre du diagramme pour s'assurer que l'utilisateur puisse toujours se repérer.
- Réorganiser l'ensemble des éléments existants en demandant au serveur de replacer les éléments sans que ces derniers ne se chevauchent.

2.5.5 Fonctionnalités optionnelles

Voici les fonctionnalités optionnelles pour améliorer l'expérience utilisateur qui avaient été proposées au début du projet :

- un chat pour laisser la possibilité aux membres de communiquer
- l'utilisation de commandes vocales pour faciliter l'utilisation de l'application
- un système de commentaire sur les diagrammes, pour fournir des informations complémentaires
- une fonction de recherche de texte
- la fusion de diagrammes
- l'exportation des diagrammes sous différents formats (graphml par exemple)
- la gestion des utilisateurs
- la possibilité de restreindre l'accès à un diagramme par un mot de passe
- la gestion des sauvegardes hors-ligne
- le dessin à main levé qui permet une reconnaissance de forme et d'ajout d'élément automatiquement

Parmi ces dernières la gestion des sauvegardes automatiques côté serveur a été implémenté, en sauvegardant dans des fichier les diagrammes sous format JSON. Les diagrammes sont donc persistants et accessibles d'une utilisation à l'autre.

2.6 Technologies

Concernant les technologies employées, le serveur utilise la plateforme multi-agent JADE et communique avec le client via une API Restlet. Côté client il avait été envisagé de réaliser l'application en utilisant Xamarin dans un premier temps, néanmoins la technologie ne fonctionnait pas correctement chez tous les membres du projet. Il a fallu réagir et prendre une décision pour pouvoir réaliser le projet dans le délai imparti. Le choix a été pris de réaliser l'application en Android natif, ce qui permet d'avoir accès à une documentation importante et d'avoir une application réactive.

2.7 Architecture de l'application

L'application cliente est constituée de 3 vues principales, qui s'occupent de l'essentiel des interactions avec l'utilisateur. Elles permettent de gérer les entrées d'utilisateurs. La première vue gère la connexion au serveur avec la création des données utilisateurs. Lors de la connexion, la deuxième vue est affichée pour lister les diagrammes existants et en créer de nouveau. Une fois un diagramme sélectionnée, la dernière vue est affichée pour permettre d'éditer le diagramme.

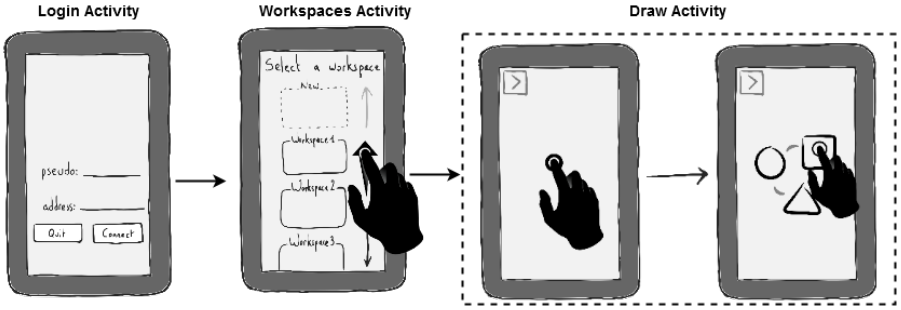


FIGURE 13 – Colladia - Vues principales

2.7.1 Architecture Général

L'application peut-être schématisée comme suit, à savoir le contrôleur situé entre la vue et le modèle. Dès qu'une modification est effectué au niveau de la liste des éléments du modèle le contrôleur met à jour la vue. Dans le sens contraire lorsque l'utilisateur interagit avec la vue une requête est envoyé au serveur pour mette à jour le diagramme sur le serveur pour les autres utilisateurs. Le modèle est mis à jour par le serveur par le retour d'une requête de modification ou par la réponse d'une requête GET qui est réalisé périodiquement pour savoir si de nouvelles modifications sont présentes sur le serveur.

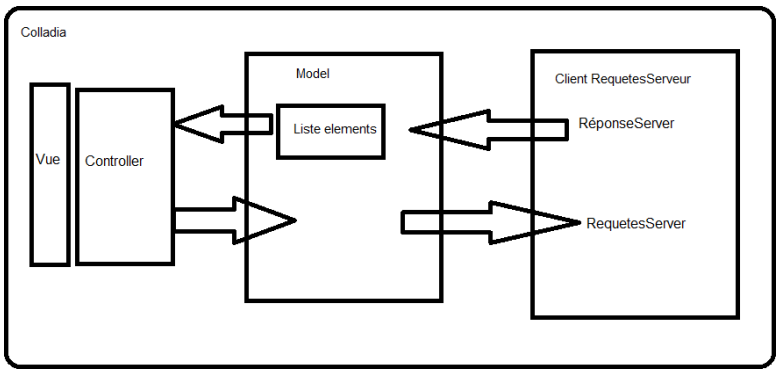


FIGURE 14 – Colladia - Schéma de l'architecture général

2.7.2 Structure de données

Outre les données utilisateurs, les structures de données principales concernent les éléments. Pour pouvoir proposer de nombreuses formes prédéfinis, il a été décidé de créer une classe Element abstraite dont hériteraient toutes les formes. Ainsi le contrôleur peut gérer les éléments sans avoir à connaître le type d'élément dont il s'agit. Ce polymorphisme induit un couplage plus léger et permet donc une plus grande souplesse du contrôleur. Certaines méthodes telles que `isTouch(PointF indexUser)` qui permet de savoir si un élément a été touché par l'index de l'utilisateur ou `draw()` qui permet de dessiner l'élément. En surchargeant ces différentes méthodes chaque forme peut proposer un comportement différent. Les éléments se distinguent entre eux par le stockage d'un identifiant `UUID` qui est unique, ce qui permet de sérialiser/dé-sérialiser sous format JSON les objets puis retrouver leurs références.

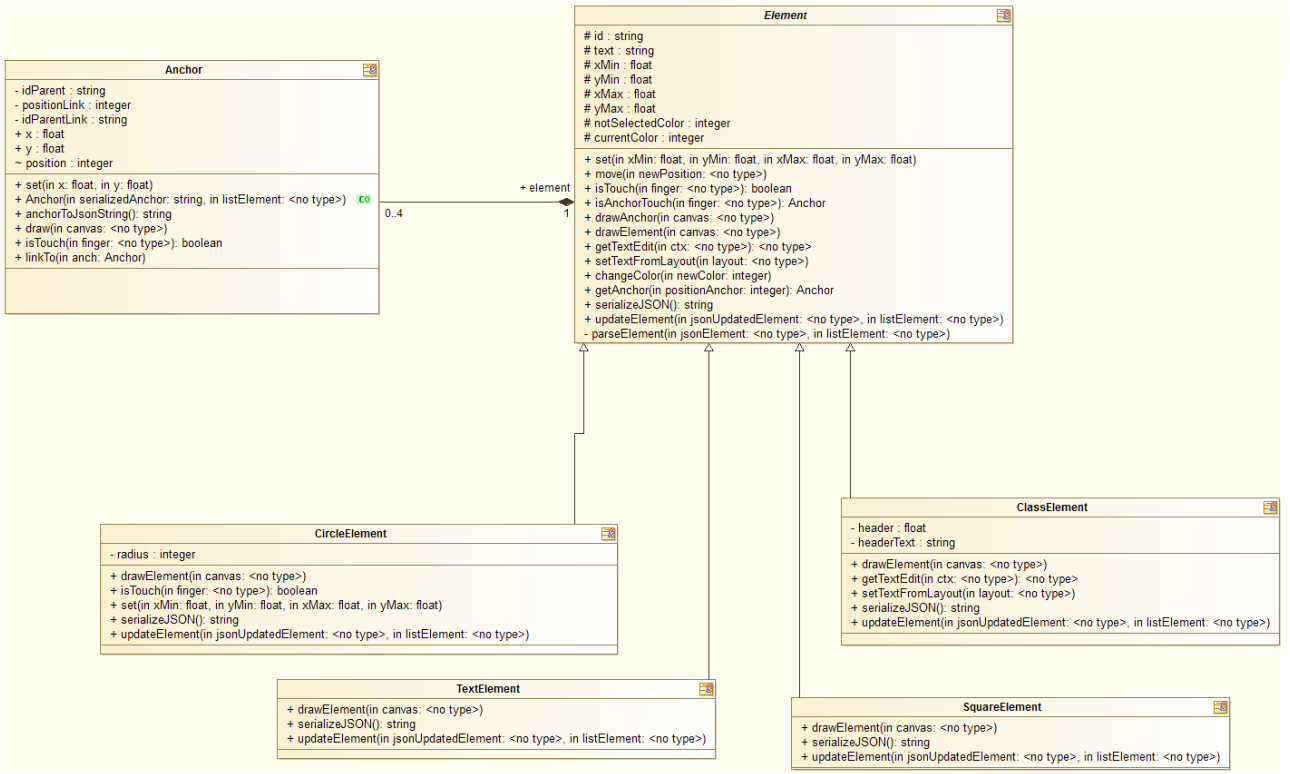


FIGURE 15 – Colladia - Structure de données simplifiée

Chaque élément possède 4 ancres (Top, Bottom, Left, Right) qui permettent de lier les éléments entre eux. Pour ce faire chaque ancre possède l'UUID de son parent ainsi que celui de l'ancre auquel il est associé. Lors de la sérialisation/dé-sérialisation des éléments il est possible de retrouver l'ancre associée sans pour autant avoir une référence sur l'objet constamment.

2.7.3 Gestion des données

Le singleton `Manager` propose des méthodes telles que `changeText(Element elmnt, Text text)` qui permet dans un premier temps de changer le texte l'élément, puis demande au `Requestator` d'envoyer une requête au serveur pour y mettre à jour l'élément. Le singleton `Requestator` s'occupe de réaliser les requêtes au serveur en utilisant le framework android Volley. Le modèle possède le diagramme couramment modifié ainsi que l'horloge logique utilisé par le serveur pour renvoyer les dernières modifications effectuées.

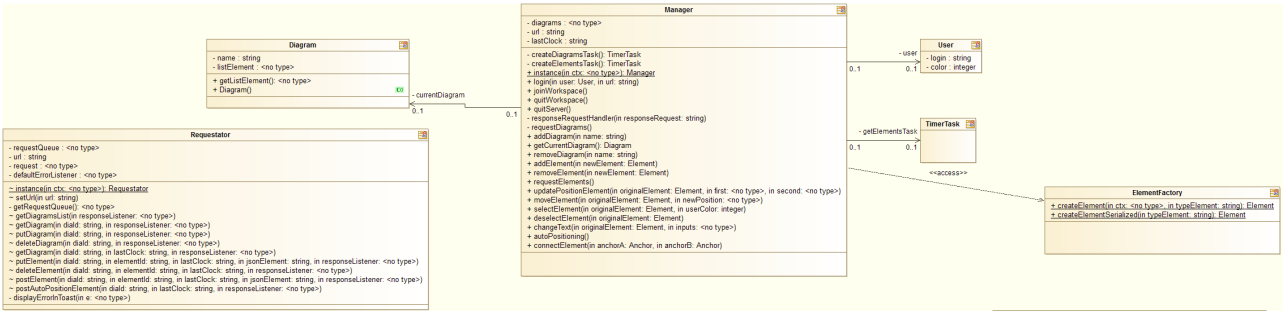


FIGURE 16 – Colladia - Gestion des données simplifiée

La classe `ElementFactory` permet de générer les éléments spécifiques tels un Cercle ou un Rectangle pour les proposer au Manager qui via le polymorphisme traitera l'objet comme un simple `Element`.

2.7.4 Différentes vues

On retrouve le concept du MVC au niveau des différentes vues que ce soit au niveau de la connexion au serveur avec les données utilisateurs, mais aussi avec la liste des diagrammes qui met à jour automatiquement la vue lorsqu'une modification arrive du serveur et modifie le modèle. Le design pattern Observable était une nécessité dans le contexte asynchrone de requêtage HTTP.

Concernant la zone de dessin il a fallu mettre en place une sorte de machine à états qui changerait de "mode" selon les interactions utilisateurs. Il existe différents modes (`ZOOM`, `SCROLL`, `INSERT`, ...), ce qui permet de déterminer le comportement à adopter selon la situation rencontrée.

La méthode `onTouchEvent()` réagit aux différentes interactions de l'utilisateur, puis appelle la méthode adéquate comme par exemple `moveTouch()` qui correspond au déplacement du doigt de l'utilisateur sur l'écran.

FIGURE 17 – Colladia - DrawColladiaView

2.8 Limites et améliorations

L'application possède une base intéressante et fonctionnelle, cependant on peut citer quelques limites principales. La première limite étant le nombre restreint d'éléments différents qui sont actuellement proposés. Un travail important été réalisé pour factoriser un maximum de code des éléments au niveau de la classe `Element`, ce qui permet de créer assez facilement des formes diverses et variés. Une des fonctionnalités supplémentaires qui pourrait aider l'utilisateur serait de permettre une sélection groupée, en plus des différentes fonctionnalités optionnelles décrites auparavant.

3 Serveur multi-agent

3.1 Description générale du système

Le serveur multi-agent a pour but de permettre la dimension collaborative de Colladia. C'est en effet son rôle d'informer les différents clients éditant un même diagramme des modifications effectuées par les autres utilisateurs. D'un point de vue technique, le serveur utilise un certain nombre de technologies :

- Les requêtes des clients sont reçues via une interface REST implémentée grâce au framework Java Restlet. Cette technologie étant par définition unidirectionnelle, les clients doivent effectuer des requêtes régulières sur le serveur pour être tenus au courant des modifications du diagramme.
- Le traitement des requêtes est effectué par un système multi-agent utilisant le framework JADE.
- Le contenu des requêtes REST et des messages du SMA sont sérialisés en JSON via la librairie Java Jackson.

La structure du SMA est divisée en deux conteneurs :

- Le conteneur principal où résident notamment les agents standards JADE (DF, AMS, etc.) ainsi que l'agent lié au serveur Restlet qui va transformer les requêtes REST reçues en messages pour le SMA (RestAgt). On y trouve aussi l'agent chargé de sauvegarder régulièrement l'état des différents diagrammes dans des fichiers de manière à pouvoir restaurer leur état après un éventuel redémarrage du serveur (SaveAgt).
- Le conteneur de diagramme contenant pour chaque diagramme :
  - Un ou plusieurs agents élément représentant le diagramme en soi ou des éléments présents dans ce dernier (EltAgt). Ces agents forment entre eux une arborescence dont l'élément représentant le diagramme est la racine.
  - Un agent horloge responsable de gérer une horloge logique pour la sauvegarde des modifications effectuées sur le diagramme (ClockAgt). L'horloge est initialisée à 0 au démarrage du serveur et est incrémentée de 1 à chaque modification du diagramme ou de ses sous-éléments.
  - Un agent historique sauvegardant une liste des dernières modifications (HistAgt).

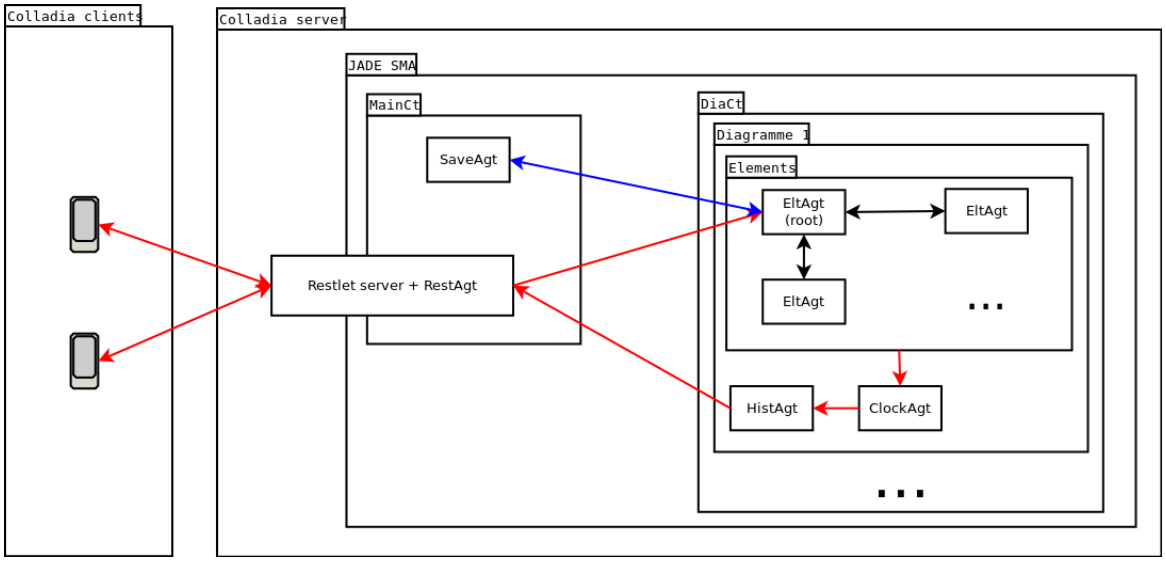


FIGURE 18 – Schéma de l'architecture générale du serveur

3.2 Description de l'interface REST

3.2.1 Champs communs

Un certain nombre de champs sont communs aux différentes requêtes REST. Pour les entrées le champ `last-clock`, optionnel, permettant au serveur de connaître la dernière valeur de l'horloge logique reçue par le client effectuant la requête.

Pour les sorties, il faut commencer par différencier les modifications sauvegardées par l'agent historique du message de retour des requêtes REST. Le retour comporte obligatoirement un champ `status` contenant un code d'erreur et un champ `clock` donnant la dernière valeur de l'horloge logique du diagramme. Le champ `status` peut prendre une des valeurs suivantes, inspirées des standards web :

- succès (2xx) :
    - 200 : OK
- redirection (3xx) :
    - 304 : non-modifié
- erreur client (4xx) :
    - 400 : requête mal formée
    - 401 : existe déjà
    - 404 : non-trouvé
- erreur serveur (5xx) :
    - 500 : erreur interne

Dans le cas où la requête n'a pas pu être exécutée (`status != 200`), un champ `error` est intégré au retour contenant une description de l'erreur. Si la requête à été exécutée correctement, qu'un champ `last-clock` a été spécifiée par le client et que sa valeur est valide par rapport aux modifications enregistrées par l'HistAgt du diagramme, alors on renvoie un champ `modification-list` qui est un tableau JSON contenant un ensemble de modifications telles que décrites ci-après. Si on est dans l'incapacité de produire une liste de modifications, on retourne un champ `description` contenant une description complète du diagramme et de ses sous-éléments sous-forme d'une sérialisation JSON.

Les modifications engendrées par certains types de requêtes (PUT, POST et DELETE) et stockées dans l'agent d'historique comportent forcément un champ `clock` indiquant l'horloge à laquelle la modification a eu lieu et un champ `type` donnant le type de la modification (PUT, POST ou DELETE).

3.2.2 Requêtes GET

Les requêtes de type GET permettent d'obtenir des informations sur l'état actuel du diagramme et n'engendrent jamais de modifications. Si l'URI est de la forme `<server adress>`, renvoie un champ `list` contenant un tableau JSON des noms des différents diagrammes stockés sur le serveur.

Si l'URI est de la forme `<server adress>/<diagram>[/<element> ...]`, renvoie un champ `path` donnant le chemin vers l'élément ciblé sous forme d'un tableau JSON ainsi qu'un champ `description` contenant la description récursive de cet élément.

3.2.3 Requêtes PUT

Les requêtes de type PUT permettent de créer de nouveaux diagrammes ou de nouveaux sous-éléments. Si l'URI est de la forme `<server adress>/<diagram>` c'est une création de diagramme, et dans le cas d'un succès (un diagramme possédant le même nom n'existe pas encore), on renvoie un champ `path` contenant le nom du diagramme.

Si l'URI cible un élément (`<server adress>/<diagram>/<element[/<element> ...]>`), on créé un nouveau sous-élément. La requête peut optionnellement contenir un champ `properties` contenant des couples (propriété, valeur) pour initialiser les propriétés de l'élément. Le retour contient un champ `path` ainsi qu'un champ `properties` contenant une description des propriétés de l'élément venant d'être créé.

3.2.4 Requêtes DELETE

Les requêtes de type DELETE permettent de supprimer un EltAgt ou bien une partie de ses propriétés. Dans les deux cas, l'URI est de la forme `<adrr>/<diagram>[/<element> ...]`. Si la requête possède un champ `properties-list` contenant une liste de nom de propriété à supprimer, alors on supprime uniquement ces propriétés au niveau de l'élément ciblé. Sinon, on supprime l'EltAgt ciblé ainsi que tous ses fils récursivement. Dans les deux cas, la modification a la même forme que la requête initiale, à ceci-près que le chemin de l'URI est intégré dans un champ `path` et le type de la requête dans un champ `type`.

3.2.5 Requêtes POST

Les requêtes REST de type POST permettent de modifier la valeur des propriétés d'un élément, une option permet d'exécuter un algorithme d'auto-positionnement sur les fils d'un EltAgt ciblé. Dans les deux cas, l'URI est de la forme `<adrr>/<diagram>[/<element> ...]`. Pour modifier directement des propriétés, un champ `properties` contenant des couples (propriété, valeur) doit être spécifié dans la requête. La modification possède une forme identique à la requête initiale.

Pour exécuter l'algorithme d'auto-positionnement, il faut spécifier dans la requête un champ `options` associé à une liste JSON contenant une valeur `auto-positioning`. Cette requête induit la sauvegarde de 0+ modifications de type POST, une par fils auto-positionnable. Un fils est dit auto-positionnable si il contient des propriétés suffisantes pour permettre un repositionnement, dans notre cas `xMin`, `xMax`, `yMin` et `yMax`. Chaque modification enregistrée contient un champ `path` contenant le chemin de l'EltAgt modifié ainsi qu'un champ `properties` récapitulant les altérations effectuées.

3.3 Description des agents et des comportements

3.3.1 RestAgt

Le RestAgt est couplé à une classe définissant un serveur Restlet de manière à pouvoir récupérer des requêtes REST puis à les transférer aux agents pouvant y répondre. L'agent est aussi responsable de la création des nouveaux diagrammes (PUT) et de fournir aux clients la liste des diagrammes existant sur le serveur (GET). À sa création, le RestAgt s'enregistre auprès du Directory Facilitator (DF) en tant qu'interface REST.

On peut dénombrer deux comportements pour cet agent. Le premier n'est pas une classe fille de la classe Behaviour, mais on peut considérer que le fait d'attendre les requêtes REST au niveau du serveur Restlet rentre dans la catégorie du comportement au sens multi-agent du terme. À la réception d'une requête, créé un nouveau diagramme ou retourne la liste des diagrammes disponibles si besoin. Sinon, envoie la requête sous forme de message ACL au EltAgt racine du diagramme ciblé par la requête et lance un ReceiveBhv. L'AID du ClockAgt du diagramme est aussi insérée au niveau du champ `reply-to` du message. Le `conversation-id` est généré par la classe Java UUID.

Le ReceiveBhv attend un message de retour pour une requête envoyée précédemment puis formule la réponse à la requête REST initiale du client.

3.3.2 SaveAgt

Le SaveAgt est responsable de la sauvegarde et de la restauration de l'état des diagrammes après un éventuel redémarrage du serveur. La description de chaque diagramme est stocké dans un fichier au format JSON. Le premier comportement, dit RestoreBhv, est un OneShotBehaviour lancé au démarrage de l'agent et qui, à partir des données sauvegardées, créé les différents diagrammes (EltAgt racine + ClockAgt + HistAgt) et envoie un message au nouveau EltAgt pour restaurer la valeur de ses propriétés et de ses sous-éléments. Une fois les différents messages envoyés, lance le TickerBhv et le ReceiveBhv.

Le TickerBhv est un comportement chargé d'envoyer régulièrement des messages GET aux différents EltAgt racine des diagrammes pour récupérer leur description complète. Le ReceiveBhv est un comportement cyclique qui récupère les réponses à ces messages et écrit les descriptions reçues dans des fichiers .json.

3.3.3 ClockAgt

Le ClockAgt est chargé de gérer l'horloge du diagramme et exprime un unique comportement cyclique, le ReceiveBhv. Il attend un message de type INFORM provenant d'un EltAgt, incrémente l'horloge si le message ne comporte par d'option `no-history` et est de type POST, PUT ou DELETE, puis, dans tous les cas, ajoute un champ `clock` au message comportant la valeur la plus récente de l'horloge. Le message est ensuite envoyé au HistAgt.

3.3.4 HistAgt

Le HistAgt est l'agent chargé de gérer l'historique des modifications et comporte un unique CyclicBehaviour encore une fois nommé ReceiveBhv. Il réceptionne les messages envoyés par le ClockAgt, et si le message ne comporte par d'option `no-history` et est de type POST, PUT ou DELETE, ajoute le contenu du message, la modification, à l'historique.

L'HistAgt est aussi chargé de formuler la réponse à la requête REST initiale si le message reçu du ClockAgt ne comporte pas une option `no-reply`. Dans ce cas, si le message intègre un champ `last-clock` et que sa valeur est supérieure à l'horloge de la première modification encore stockée dans la liste des modifications, l'agent renvoie au RestAgt la liste des modifications appliquée depuis celle portant l'horloge `last-clock`. Sinon envoie un message au EltAgt racine pour récupérer la description complète du diagramme en ajoutant un champ `reply-to` vers le RestAgt.

3.3.5 EltAgt

Agent chargé de stocker l'état courant d'un élément ou du diagramme lui-même. Pour chaque diagramme, les EltAgt forment une arborescence et la racine, qui représente le diagramme en soi, s'enregistre auprès du DF. Chaque EltAgt implémente des fonctions permettant entre autre d'ajouter, supprimer, modifier des éléments. C'est aussi eux qui délibèrent pour permettre un auto-positionnement des éléments à la demande du client.

3.3.5.1 ReceiveBhv

Le principal comportement utilisé, le ReceiveBhv, est un CyclicBehaviour lancé au démarrage de l'agent et qui attend des messages de requêtes des autres agents, que ce soit du RestAgt, du SaveAgt ou d'un autre EltAgt du diagramme. Le fonctionnement de ce comportement après réception d'un message de type REQUEST peut être résumé comme ceci :

- PUT :
    - Si l'élément courant est l'avant-dernier élément de l'URI, ajoute un nouvel élément si celui-ci n'existe pas déjà (création d'un EltAgt et ajout à l'arborescence).
    - Sinon, transfert du message au fils correspondant au prochain élément dans l'URI si il existe ou retourne une erreur.
- GET :
    - Si l'élément courant a une profondeur dans l'arborescence strictement inférieure à celle de la cible de l'URI, transfert du message au fils correspondant au prochain élément dans l'URI si il existe ou retourne une erreur.
    - Sinon :
        - Si l'élément est une feuille, répond au message en envoyant la liste des propriétés et leurs valeurs.
        - Sinon, transfert le message à tous les sous-éléments et lance un comportement WaitDescriptionBhv pour attendre les réponses.

- DELETE :
  - Si la requête contient un champ `properties-list` (suppression de propriété) :
    - Si l'élément courant est le dernier élément de l'URI, suppression des propriétés dans l'élément si elles existent.
    - Sinon, transfert du message au fils correspondant au prochain élément dans l'URI si il existe ou retourne une erreur.
  - Sinon (suppression d'élément) :
    - Si l'élément courant a une profondeur dans l'arborescence strictement inférieure à celle de la cible de l'URI, transfert du message au fils correspondant au prochain élément dans l'URI si il existe ou retourne une erreur.
    - Sinon :
      - Si l'élément courant est le dernier élément de l'URI, envoie une réponse à l'expéditeur initiale de la requête.
      - Transfert du message à tous les fils.
      - Autodestruction de l'agent.
- POST :
  - Si l'élément courant est le dernier élément de l'URI :
    - Si le message comporte une option `auto-positioning` (auto-positionnement des fils) :
      - Envoi un message de type CFP (Call For Proposal) à tous les fils comportant un angle pour la dispersion ainsi que la largeur et la longueur de l'élément courant pouvant prendre la valeur -1 pour indiquer une taille infinie.
      - Lance un nouveau comportement WaitProposalBhv avec en paramètre le nombre de messages envoyés vers les fils.
    - Sinon, si le message comporte un champ `properties`, modification des propriétés de l'élément et envoi une réponse à l'expéditeur initial de la requête.
  - Sinon, transfert du message au fils correspondant au prochain élément dans l'URI si il existe ou retourne une erreur.
- RESTORE (type supplémentaire du serveur pour la restauration des diagrammes) :
  - Pour chaque couple (clef, valeur) du champ `properties` :
    - Si la valeur est un objet JSON, créé un nouvel élément, l'ajoute à l'arborescence et lui envoie un message de restauration contenant la valeur du couple.
    - Sinon, c'est une propriété, et on l'ajoute à celles de l'élément courant.

3.3.5.2 Autres comportements

Le comportement WaitDescriptionBhv est lancé après avoir envoyé des messages aux fils suite à un GET et attend leurs descriptions respectives. Une fois la description de tous les fils reçus, formule une réponse au message de requête reçu initialement (en prenant en compte le `reply-to`).

Le ReceiveCFPBhv est un comportement cyclique lancé au démarrage de l'agent qui, à chaque réception d'un message CFP par l'agent lance un nouveau ProposeBhv. Ce dernier commence par placer l'élément au centre de son père, puis envoi ses nouvelles coordonnées (`x` et `y`) ainsi que ses dimensions (`width` et `height`) au père dans un message PROPOSE. Tant que ce dernier renvoi des REJECT_PROPOSAL, avance d'un pas fixé dans la direction donnée par l'élément père au lancement de l'algorithme d'auto-positionnement.

Une fois un ACCEPT_PROPOSAL reçu, il envoi un message au HistAgt du diagramme pour enregistrer la modification des coordonnées. Sachant que plusieurs modifications peuvent être engendrées par une seule requête d'auto-positionnement et qu'une requête REST n'accepte qu'une unique réponse, on ajoute un flag `no-reply` au message, de manière à ce que ce message n'engendre pas une réponse au RestAgt qui serait ensuite transférée au client.

Si, lors d'une itération l'élément ne peut plus se déplacer en x et en y car il a atteint les limites de son père, alors il intègre une option `force` à son message pour forcer l'acceptation de la proposition. Si l'élément ne comporte pas les propriétés nécessaires aux calculs d'auto-positionnement (`xMin`, `xMax`, `yMin` et `yMax`), il renvoie un message ayant un performatif REFUSE.

Le comportement WaitProposalBhv est lancé au début de chaque algorithme d'auto-positionnement en sachant combien de CFP ont été envoyés à des fils et entretien une liste des coordonnées et dimensions des éléments fils dont la nouvelle position a déjà été acceptée. À la réception d'un message de type REFUSE, décrémente le compteur de message attendu. À la réception d'un message PROPOSE, si la nouvelle position de l'élément fils chevauche celle d'un des éléments déjà accepté, retourne un message de type REJECT_PROPOSAL, sinon, un ACCEPT_PROPOSAL.

Dans le cas où le message reçu comporte une option `force`, renvoie toujours un ACCEPT_PROPOSAL. Chaque envoi d'un ACCEPT_PROPOSAL induit une décrémentation du compteur. Une fois que le compteur atteint 0, répond au message de requête initial en ajoutant une option `no-history`, de manière à fournir une réponse à la requête REST incluant uniquement les modifications réalisées par ses fils.

3.4 Description des messages

Pour tous les types de message décrits ci-après, le champ `conversation-id` est un UUID unique généré lors de la réception d'une requête REST par le serveur Restlet. Tous les messages ACL généré suite à cette requête utilisent ce même identifiant. Le champ FIPA `content` est toujours un dictionnaire JSON sérialisé. Les champs utilisés ci-dessous ne faisant pas parti du formalisme FIPA sont des champs de ce dictionnaire.

3.4.1 RestAgt

Tous les messages envoyés par le RestAgt respectent le patron suivant :

- `performative` : REQUEST
- `receivers` : un EltAgt racine
- `reply-to` : ClockAgt
- `type` : PUT, GET, DELETE ou POST dépendant du type de la requête REST initiale
- `path` : chemin du diagramme élément visé par la requête sous forme de tableau JSON
- `properties` : liste des propriétés et de leurs valeurs dans le cas d'une modification/création d'un élément
- `properties-list` : liste des propriétés à supprimer au sein d'un élément pour certaines requêtes DELETE
- `options` : `[auto-positionning]`
- `last-clock` : dernière horloge reçue par le client si spécifiée dans la requête REST

3.4.2 SaveAgt

Au démarrage du serveur, envoi des messages respectant le format suivant aux nouveaux EltAgt créés de manière à restaurer l'état du diagramme :

- `performative` : REQUEST
- `receivers` : un EltAgt racine (une fois créé)
- `type` : RESTORE
- `description` : description récursive des propriétés du diagramme et de ses éléments sous forme de dictionnaires JSON imbriqués

Une fois la phase de restauration terminée, le SaveAgt envoi à chaque tick de son TickerBhv le message suivant à tous les EltAgt racine :

- `performative` : REQUEST
- `receivers` : tous les EltAgt racine
- `type` : GET
- `path` : `[]` (liste vide pour que l'EltAgt retourne la description complète du diagramme)

3.4.3 ClockAgt

Les messages reçus par le ClockAgt sont transférés au HistAgt après ajout d'un champ `clock`, et respectent le format suivant :

- `performative` : INFORM
- `receivers` : HistAgt
- `content` : identique à celui du message reçu avec ajout d'un champ `clock` contenant la dernière valeur de l'horloge logique du diagramme

3.4.4 HistAgt

Si le HistAgt a pu récupérer la liste des modifications demandées par le client, envoi le message suivant au RestAgt :

- `performative` : INFORM
- `receivers` : RestAgt
- `status` : 200
- `clock` : dernière valeur de l'horloge logique (récupérée à partir du dernier message reçu)
- `modification-list` : liste des modifications appliquées au diagramme depuis l'horloge `last-clock` spécifiée par le client

Sinon, on envoie un message au EltAgt racine pour obtenir la description complète du diagramme et la transférer au RestAgt :

- `performative` : INFORM
- `receivers` : EltAgt racine
- `reply-to` : RestAgt
- `type` : GET
- `path` : `[]` (liste vide pour que l'EltAgt retourne la description complète du diagramme)
- `clock` : valeur de l'horloge logique insérée dans le contenu du message par le ClockAgt

3.4.5 EltAgt

Au niveau des éléments, on peut distinguer deux grandes familles de messages, les messages internes, envoyés d'un EltAgt à un autre EltAgt de l'arborescence, et les messages qui ont pour destinataires d'autres agents du diagramme (ClockAgt, RestAgt, etc.). Pour les messages dirigés vers l'extérieur, il existe deux possibilités. Soit l'exécution de la requête s'est déroulée sans encombres :

- `performative` : INFORM
- `receivers` : valeur du champ `reply-to`, ou `sender` si le premier est inexistant
- `status` : 200
- `content` : contenu de la requête initial plus les éventuels champs suivants :
- `description` : description récursive d'un élément du diagramme ou du diagramme complet dans le cas d'une requête GET

Soit elle a générée une erreur :

- `performative` : FAILURE
- `receivers` : valeur du champ `sender` de la requête (le `reply-to` est ignoré)
- `status` : un code d'erreur (3xx, 4xx ou 5xx)
- `error` : message de description de l'erreur

Pour les messages internes, on peut distinguer deux types de messages en plus de ceux liés à l'algorithme d'auto-positionnement. Lors du transfert d'un message vers le prochain fils de l'URI, l'entièreté des champs du message sont conservés, `reply-to` et `sender` compris.

Pour ce qui est des requêtes GET en revanche, une fois que la cible de l'URI a été passée, les messages envoyés aux différents fils en supprimant le champ `reply-to` et en mettant l'AID de l'EltAgt courant dans le champ `sender`. Ainsi, une fois qu'un élément à récupéré récursivement sa description, il crée une réponse au message de requête reçu. Le destinataire sera alors l'agent externe à l'arborescence ayant envoyé la requête ou le `reply-to` qu'il a spécifié si l'agent courant était la cible de l'URI, ou bien le père dans l'arborescence des éléments du diagramme.

3.4.5.1 Messages liés à l'algorithme d'auto-positionnement

À la réception d'une demande d'auto-positionnement, la cible de l'URI transfère à tous ces fils un message respectant le format suivant :

- `performative` : CFP
- `receivers` : un EltAgt fils
- `x`, `y`, `width` et `height` : valeurs calculées à partir des propriétés `xMin`, `xMax`, `yMin` et `yMax` de l'élément
- `angle` : un angle de dérivation donné par la formule $\frac{2\pi}{n} * i$ où n est le nombre de fils et i l'index du fils dans la liste des fils

L'EltAgt recevant ce message renvoi alors un message REFUSE si il n'est pas auto-positionnable :

- `performative` : REFUSE
- `receivers` : EltAgt père dont l'AID est donnée par le champ `sender` du CFP reçu

Une proposition dans le cas contraire ou si un REJECT_PROPOSAL a été reçu de la part du père suite à une proposition :

- `performative` : PROPOSE
- `receivers` : EltAgt père dont l'AID est donnée par le champ `sender` du dernier message CFP ou REJECT_PROPOSAL reçu
- `x`, `y`, `width` et `height` : nouvelles coordonnées proposées ainsi que les dimensions pour vérifier l'existence de chevauchement
- `options` : `[force]` si l'élément a atteint les limites de son père

Les messages permettant d'accepter ou de refuser une proposition sont relativement simples :

- `performative` : REJECT_PROPOSAL ou ACCEPT_PROPOSAL
- `receivers` : L'EltAgt fils ayant soumis une proposition

Une fois qu'un élément a vu sa proposition acceptée, il envoie un message contenant ses nouvelles coordonnées pour le HistAgt :

- `perfomative` : INFORM
- `receivers` : ClockAgt
- `type` : POST
- `properties` : `{xMin:<value>, xMax:<value>, yMin:<value>, yMax:<value>}`
- `options` : `[no-reply, auto-positionning]`

Une fois l'auto-positionnement de tous les fils réalisés, l'EltAgt père envoi un message au HistAgt pour qu'une réponse soit formulée à la requête REST initiale :

- `perfomative` : INFORM
- `receivers` : ClockAgt
- `options` : `[no-history, auto-positionning]`

3.5 Limites et améliorations

Concernant la partie serveur du projet on peut noter quelques points qui serait potentiellement limitants dans le cadre d'un déploiement à grande échelle :

- À part pour l'auto-positionnement, le serveur manipule les données envoyées par les clients sans se préoccuper de leur nature. Pour ce faire, toutes les données sont stockées sous forme de string et ce récursivement. Le contenu des messages transitant à la fois entre les différents agents et entre les clients et le serveur comporte de ce fait un grand nombre de caractères '\' utilisés pour échapper les caractères de début de chaîne ('"'). Pour des arborescences ayant une certaine profondeur, ceci pourrait être défavorable car les messages pourraient finir par contenir plus de caractères d'échappement que de caractères utiles.
- Le fait d'utiliser la technologie REST implique que les clients doivent réaliser régulièrement (un centième de seconde au moment de l'écriture de ce rapport) des requêtes de type GET pour être tenus informés des modifications produites par les autres clients. Si beaucoup de clients étaient connectés simultanément à un même serveur, celui-ci pourrait en pâtir. Pour éviter ce problème, il faudrait faire le choix d'une autre technologie bidirectionnelle, tel que les sockets.

L'algorithme d'auto-positionnement pourrait quant à lui aussi être amélioré sur certains points :

- Prise en compte des liens pour le positionnement.
- Redimensionnement intelligent des fils en fonction de leur contenu.
- Ajustement de la taille de police des textes.

4 Conclusion

Au cours de ce projet, il nous a été possible d'utiliser à la fois les notions du cours de IA04 et de NF28 ce qui nous a permis d'approfondir nos connaissances dans ces domaines. De plus, le fait de travailler à 5 sur ce projet nous a demandé de communiquer et d'organiser la répartition des taches de manière continue sur toute la durée du projet. Il a ainsi fallu utiliser des outils tel que Slack pour communiquer, git pour gérer les versions du code et Google Drive pour partager des documents.

Les difficultés rencontrées nous ont amenés à échanger nos idées pour permettre de les résoudre de la manière la plus adaptée possible, en se servant de l'expérience de chacun. Ce fut un réel plaisir de travailler sur ce projet durant le semestre, et nous avons pu apprendre beaucoup que se soit au niveau de la conception, de la programmation Android ou bien de la mise en place d'un SMA complexe.