|  |  |  |
| --- | --- | --- |
| D:\Learning\UMPC\Stage\upmc-logotype.gif |  | D:\Learning\UMPC\Stage\logo_tecomptech_200.gif |

**MASTER INFORMATIQUE**

**SPÉCIALITÉ SCIENCE ET TECHNOLOGIE DU LOGICIEL**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

RAPPORT DE STAGE DE FIN D'ÉTUDES

**OUTILS POUR LA DECOUVERTE ET LA COMPOSITION DES SERVICES**

**D'INTERFACE UTILISATEUR EN ENVIRONNEMENT PERVASIF**

|  |  |  |
| --- | --- | --- |
| ***Responsable du stage*** | ***:*** | ***Prof. Jean-Claude Moissinac***  ***Prof. Emmanuel Chailloux***  ***Doctorante. Bertha Helena Rodriguez*** |
| ***Étudiant*** | ***:*** | ***Xuan-Hoa Nguyen*** |
| ***Formation*** | ***:*** | ***Master 2 STL - APR*** |
| ***Année scolaire*** | ***:*** | ***2011 - 2012*** |
| ***Période du stage*** | ***:*** | ***04/2012 - 09/2012*** |
| ***Laboratoire d'accueil*** | ***:*** | ***Télécom ParisTech*** |

**Paris 08/2012**

# REMERCIEMENT

***Table de matières***

[REMERCIEMENT 2](#_Toc334028033)

[CHAPITRE 1: INTRODUCTION 6](#_Toc334028034)

[1. Contexte 6](#_Toc334028035)

[2. Le stage recherche STL 6](#_Toc334028036)

[2.1. Choix du sujet 6](#_Toc334028037)

[2.2. Description du sujet 6](#_Toc334028038)

[2.3. Missions 6](#_Toc334028039)

[CHAPITRE 2: ARCHITECTURE MMI 8](#_Toc334028040)

[1. État de l'art 8](#_Toc334028041)

[2. Aperçu de l'architecture 8](#_Toc334028042)

[3. Composants principaux 9](#_Toc334028043)

[3.1. InteractionManager 9](#_Toc334028044)

[3.2. ModalityComponent 10](#_Toc334028045)

[3.3. DataComponent 10](#_Toc334028046)

[4. Communication entre les composants 10](#_Toc334028047)

[4.1. Événements MMI 11](#_Toc334028048)

[4.2. Notifications MMI 12](#_Toc334028049)

[CHAPITRE 3: TRAVAIL RÉALISÉ 14](#_Toc334028050)

[1. Environnement et méthodologie de travail 14](#_Toc334028051)

[1.1. Environnement de travail 14](#_Toc334028052)

[1.2. Méthodologie de travail 14](#_Toc334028053)

[2. Mise en œuvre de l'architecture MMI 15](#_Toc334028054)

[2.1. Couche Réseau: XHR Lib 16](#_Toc334028055)

[2.2. Couche Transport: MMI Lib 17](#_Toc334028056)

[2.3. Couche Session: Architecture Lib 19](#_Toc334028057)

[3. Services MMI 24](#_Toc334028058)

[3.1. Enregistrement du son 24](#_Toc334028059)

[3.2. Transformation de la parole en texte (Speech-to-Text) 24](#_Toc334028060)

[3.3. Transformation du texte en parole (Text-to-Speech) 25](#_Toc334028061)

[3.4. Reconnaissance des gestes 26](#_Toc334028062)

[3.5. Découverte des services locaux 27](#_Toc334028063)

[3.6. Autres services 27](#_Toc334028064)

[3.6.1. Horloge 27](#_Toc334028065)

[3.6.2. Commandeur 28](#_Toc334028066)

[3.6.3. Calendrier 28](#_Toc334028067)

[4. Critiques 30](#_Toc334028068)

[4.1. Résultat obtenus 30](#_Toc334028069)

[4.2. Comparaison 30](#_Toc334028070)

[4.3. Travail à faire 30](#_Toc334028071)

[CHAPITRE 4: CONCLUSION 31](#_Toc334028072)

[BIBLIOGRAPHIE 32](#_Toc334028073)

[ANNEXES 33](#_Toc334028074)

[1. Recherches en multimédia 33](#_Toc334028075)

[1.1. Gpac 33](#_Toc334028076)

[1.2. SVG 33](#_Toc334028077)

[1.3. UPnP 33](#_Toc334028078)

[2. Diagrammes 33](#_Toc334028079)

Table de figures

[Figure 1: Informatique pervasif 8](#_Toc334028080)

[Figure 2: Architecture Multimodale. 9](#_Toc334028081)

[Figure 3: Framework et ses composants 10](#_Toc334028082)

[Figure 4: Communication entre les composants 11](#_Toc334028083)

[Figure 5:Événements MMI 11](#_Toc334028084)

[Figure 6: Méthode d'itération 14](#_Toc334028085)

[Figure 7: Gestion de version utilisant SVN 15](#_Toc334028086)

[Figure 8: Suivi des tâches et gestion des erreurs 15](#_Toc334028087)

[Figure 9: Fonctionnalités de XHR Lib 16](#_Toc334028088)

[Figure 10: Traitement des messages 17](#_Toc334028089)

[Figure 11: Rôles de la couche de transport 18](#_Toc334028090)

[Figure 12: MMI Lib 18](#_Toc334028091)

[Figure 13: Bibliothèque Architecture Lib 20](#_Toc334028092)

[Figure 14: Enregistrement de l'application avec le serveur 21](#_Toc334028093)

[Figure 15: Deux types de commandes 22](#_Toc334028094)

[Figure 16: Notification des événements. 23](#_Toc334028095)

[Figure 17: Enregistrement du son, fixer error 24](#_Toc334028096)

[Figure 18: Transformation de la parole en texte 25](#_Toc334028097)

[Figure 19: Reconnaissance des gestes 27](#_Toc334028098)

[Figure 20: Le calendrier. 29](#_Toc334028099)

# INTRODUCTION

Ce chapitre vous présentera une vue globale sur le contexte et sur mon stage à Télécom ParisTech. Il a deux sous parties:

1. Contexte
2. Le stage recherche STL

## Contexte

Le stage recherche STL s'inscrit dans le cadre des études en Master 2, parcours Algorithmique et Programmation pour la Recherche - spécialité Science et Technologie du Logiciel à l'Université Pierre et Marie Curie.

Pendant la période de stage durant cinq ou six mois, chaque étudiant devrait faire son stage chez un laboratoire avec le sujet conformant à ce que dit la spécialité STL. Le but c'est d'aider les étudiants à s'adapter à l'environnement de recherche et à avoir ses premières expériences professionnelles.

## Le stage recherche STL

### Choix du sujet

Le stage, dont le sujet est "*outils pour la découverte et la composition de services d'interface utilisateur en environnement pervasif*", proposé par l'équipe Multimédia à Télécom ParisTech beaucoup m'intéresse. Il s'agit d'une continuité de travaux débutés dans le cadre du projet européen Intemedia[[1]](#footnote-2) et en liaison avec le laboratoire Ubimedia, Alcatel-Lucent.

Après avoir bien réfléchi, j'ai choisi ce sujet. C'est vraiment un sujet très intéressant dans lequel je pourrais à la fois étudier les nouvelles tendances du web et travailler comme un ingénier. De plus, mes compétences de recherche seraient aiguisées grâce à un environnement très excellent chez Télécom ParisTech

### Description du sujet

Mon stage a débuté le 02 avril 2012 et se terminera le 14 septembre 2009 au site de Télécom ParisTech, 37-39 rue Dareau - 75014 Paris.

Il se passe sous l'encadrement de monsieur Jean-Claude MOISSINAC et madame Bertha Helena RODRIGUEZ au sein d l'équipe multimédia et réseau du département TSI

Notre équipe travaille sur diverses solutions technologiques permettant d’assurer l'interopérabilité entre des services multimédia dans un environnement de plus en plus riche en terminaux multimédia : téléviseurs, baladeurs multimédia, tablettes, smartphone...

### Missions

Quant à moi, mon travail porte sur l'implémentation d'un protocole permettant un service de s'enregistre sa présence avec le serveur suivant la spécification MMI [1] et sur le développement des services d'interface d'utilisateur en environnement pervasif.

Concrètement, mon travail se divise en des missions ci-dessous:

**Au niveau de la recherche**

* Études du projet Open Source GPAC[[2]](#footnote-3), des extension .bifs (BInary Format for Scenes), .xmt (eXtensible MPEG-4 Textual Format) [2]
* Études de la norme SVG [3], du protocole UPnP
* Études de l'architecture MMI proposé par W3C donc l'équipe Multimedia fait ses contributions.
* Études de la norme Soa2m proposé par Télécom ParisTech

**Au niveau de développement**

* Implémentation le cœur de l'architecture MMI
  + Couche Réseau : XHR Lib
  + Couche Transport : MMI Lib
  + Couche Session Architecture Lib
* Implémentation des services (Couche Application) :
  + Enregistrement du son
  + Transformation de la parole en texte (Speech-To-Text)
  + Transformation du texte en parole (Text-To-Speech)
  + Reconnaissance des gestes.
  + Découverte des services locaux
  + Autre:
    - Commandeur : Les boutons simple
    - Afficheur: L'horloge est le calendrier.

// Des conclusion....

# ARCHITECTURE MMI

Ce chapitre va vous présentera l'architecture MMI sur laquelle notre implémentation se base. À la fin de ce chapitre, vous sauriez pourquoi on a proposé cette architecture, c'est quoi son architecture, quels sont ses composants principaux, ainsi que la communication entre les composants. Il se comporte des sections suivantes:

|  |  |
| --- | --- |
| 1. État de l'art 2. Aperçu de l'architecture | 1. Composants principaux 2. Communication entre les composants |

## État de l'art

|  |  |
| --- | --- |
| Une des tendances de développement de la technologie d'information aujourd'hui, c'est le système d'information pervasifs. Il s'agit d'un système dans lequel tous objets seraient enrichi de capacités de traitement d'information. Ces objets pourraient non seulement présenter un comportement dépendant du contexte mais aussi être si interactifs qu'ils puissent communiquer avec d'autres objets intelligents dans les réseaux (voir ).  Au niveau de la description, nous pouvons voir une perspectif très brillante. Évidemment, si c'était réalisable, notre vie deviendrait beaucoup meilleure. Néanmoins, sachant que les dispositifs sont très diversités au niveau de matérielle, de plateforme, d'implémentation, il est encore un gros défi de construire un tel système comme ça.  Tenir compte ce défi, quelque organisations ont proposé des solutions pour s'adresser ces problèmes. Parmi eux, celui de W3C s'appelant *Muitimodal Architecture and* | Figure 1: Informatique pervasif  *Source : Internet* |

*Interfaces (ou l'architecture MMI)* est très complet et détaillé. Alors, qu'est ce qu'il a proposé ? Pour mieux comprendre sa solution, continuons-nous à la section suivent : l'aperçu de l'architecture MMI.

## Aperçu de l'architecture

L'architecture multimodale est un standard développé par W3C depuis 2005 (voir Figure 2). Le standard proposé une architecture et un protocole pour que les modules dans le système puissent se communiquer l'un avec les autres dans un environnement hétérogène.

Les communications entre les composants se font de façon asynchrone suivant l'architecture orientée événement. Ainsi, à condition que les normes soient respectées, chaque composant peut être implémenté de telle façon qu'il ne dépend pas de l'implémentation d'autres composants.

L'architecture affirmerait également que les exigences ci-dessous soient assurées si l'on respectait sa specification:

* **Encapsulation**: L'implémentation détaillée de chaque composant est considérée comme une boîte noire.
* **Distribution:** L'implémentation de façon distribuée est supportée
* **Extensibilité**: Ce n'est pas difficile d'ajouter des nouveaux composants en n'importe quel moment.
* **Récursivité:** Chaque composant pourrait avoir des sous-composants. Cela diversifie ses fonctionnalités ainsi que son autonome.
* **Modularité**: Dans l'architecture, les données, les contrôles, et les présentations sont toujours séparés l'un avec l'autres.

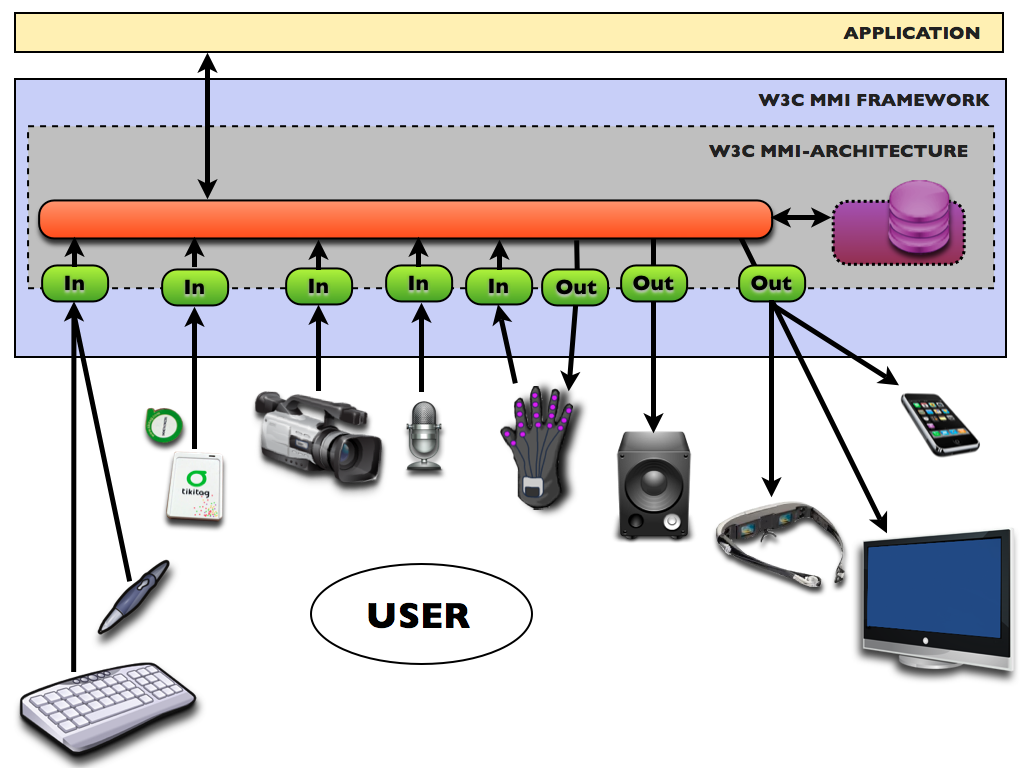


Figure 2: Architecture Multimodale.

*Source: http://en.wikipedia.org/wiki/Multimodal\_Architecture\_and\_Interfaces*

Maintenant, quand nous avons une vue globale sur l'architecture MMI, nous détaillons un peu sur sa structure. La section suivante discutera ses composants principaux:

Au niveau de composant, l'architecture MMI conformant au modèle MVC avec les trois composants principaux:

* InteractionManager (Le contrôleur)
* ModalityComponent (La Vu)
* DataComponent (Le Modèle)

Dans la section suivante, nous vous détaillerons ces composants.

## Composants principaux

### InteractionManager

*InteractionManager*, ou le contrôleur, (voir Figure 3) est un composant logique responsable d'échanger les messages entre les composants. C'est dans ce composant qu'on va gérer la communication, la gestion des événements. Chaque application devrait avoir au moins un comme son contrôleur, Lui, à son tour, il va:

* Gérer les événements, ainsi que les mécanismes pour la production, le traitement d'un événement.
* Gérer les communications entre les composants dedans l'application et les communications en dehors de l'application.
* Gérer les la synchronisation des données.

### ModalityComponent

*ModalityComponent,* ou la Vue*,* est normalement un dispositif terminal (l'écran, le microphone...). Son travail lié aux tâches concernant les données, l'entrée, la sortie des données...

D'après la spécification quelques ses responsabilités principalles *ModalityComponent* sont:

* Gérer les commandes de l'entrée et la reconnaissance d'entrée
* Gérer l'intégration de l'entrée s'il y a plusieurs entrées de types différentes (par exemple, traiter à la fois le son et la vidéo d'entrée.
* Gérer la génération du contenu de sortie (par exemple, mélanger du son et de la vidéo pour créer des flux de vidéo).

### DataComponent

*DataComponent,* ou le Modèle*,* est chargé de stocker les données utilisées par l'application. Nous ne pouvons pas lui directement accéder, mais via le *InteractionManager*. Du coup, tous les *ModalityComponent* devraient utiliser l'*InteractionManager* comme une porte d'accès à des données de l'application.

Néanmoins, chaque *ModalityComponent* pourrait avoir son propre *DataController* s'il a besoin de stocker ses données privées. C'est ici l'architecture récursive qu'on a abordée tout à l'heure.

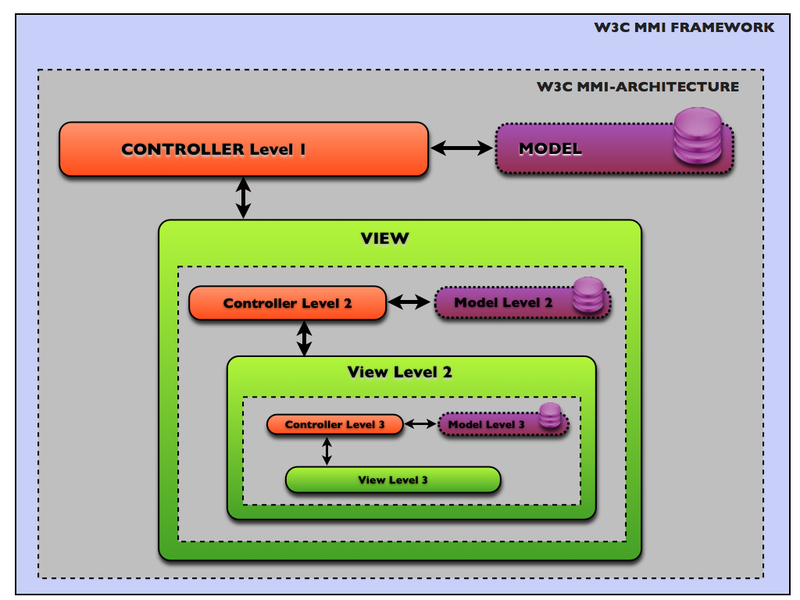


Figure 3: Framework et ses composants

*Source: http://en.wikipedia.org/wiki/Multimodal\_Architecture\_and\_Interfaces*

Après avoir étudié le composant principal selon l'architecture MMI, nous passons par la section suivante, qui vous expliquera la façon avec laquelle les composants se communiquent entre eux.

## Communication entre les composants

Pour la communication, l'architecture MMI propose son propre protocole. Le protocole est asynchrone, bidirectionnel. Il se construit à partir des événements appelant *Life-Cycle Events (voir* Figure 5*)*. Son but est de définit une façon pour échanger des informations, pour configurer la communication (établir et terminer une communication...).

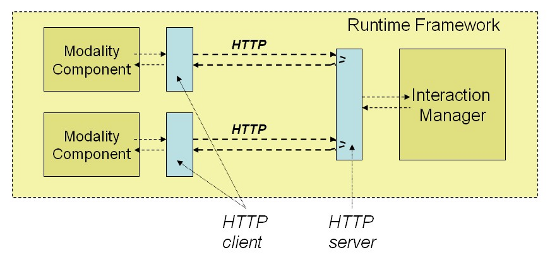


Figure 4: Communication entre les composants

*Source: http://www.w3.org/TR/2012/PR-mmi-arch-20120814/*

La spécification a proposé six types d'événement standard (qui sont en paire requête / réponse) pour la communication et deux types de notification pour la surveillance l'état courant du système. Nous allons étudier un par un:

### Événements MMI

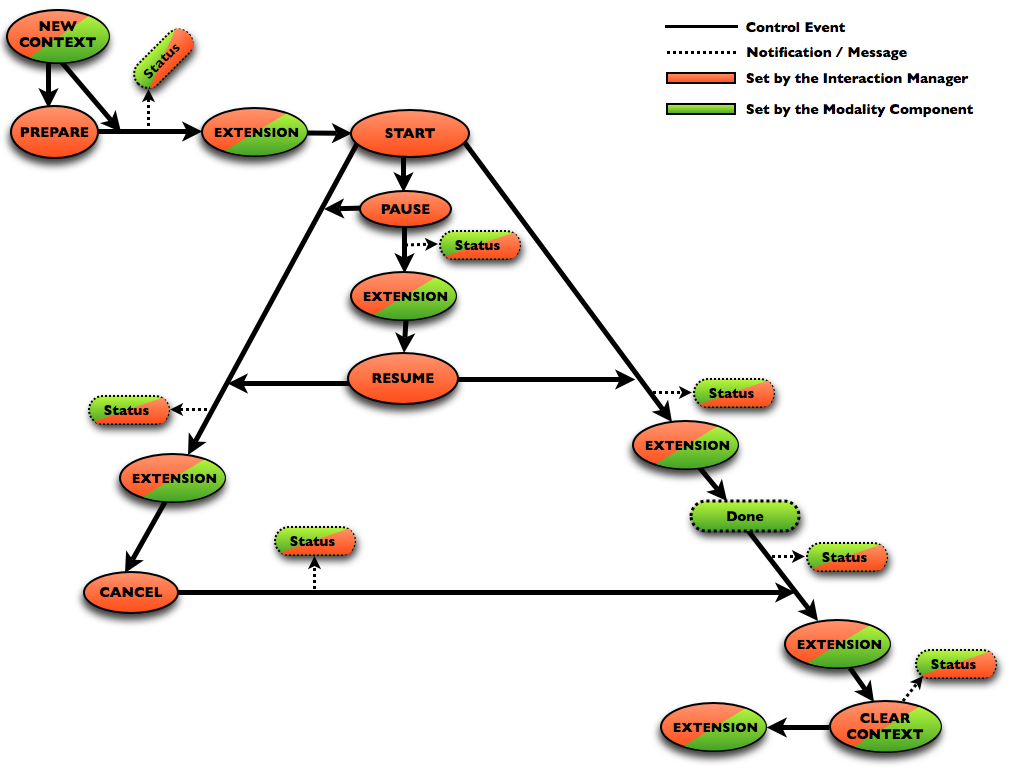


Figure 5:Événements MMI

*Source: http://en.wikipedia.org/wiki/Multimodal\_Architecture\_and\_Interfaces*

Comme on a abordé tout à l'heure, il y a six types d'événement standard:

* **NewContext (NewContextRequest / NewContextResponse):** Utilisé par un *ModalityComponent* pour demander à l'InteractionManager de lui donner un contexte. Un contexte est un cycle d'interaction dans laquelle il y a plusieurs utilisateurs et plusieurs *ModalityComponent.* Le contexte cela signifie également une période d'interaction la plus longue où les modules doivent rendre les informations disponibles.

Considérons un exemple: un *ModalityComponent* chargé de gérer les commandes produites par les boutons (voir CHAPITRE 3:3.6). Avant qu'il puisse communiquer avec d'autres composants. Il devrait tout d'abord demander le contexte avec la requête suivante:

|  |
| --- |
| <mmi:mmi xmlns:mmi="http://www.w3.org/2012/08/mmi-arch" version="1.0">  <mmi:newContextRequest  requestID="myReq1"  source="command.html"  target="server.php"  data="myMCStatus.xml" />  </mmi:mmi> |

Et la réponse:

|  |
| --- |
| <mmi:mmi xmlns:mmi="http://www.w3.org/2012/08/mmi-arch" version="1.0">  <mmi:newContextResponse  requestID="myReq1"  source="server.php"  target="command.html"  context="myContextID1"  status="success" />  </mmi:mmi> |

* **ClearContext (ClearContextRequest / ClearContextResponse)** Contrairement à la paire *NewContextRequest/Response*, ces deux événements indiquent la terminaison d'un cycle d'interaction. La requête est envoyée par l'Interaction pour but d'arrêter le *ModalityComponent* concerné. Lors de la réception, le composant devrait libérer les ressources et s'arrête.
* **Prepare (PrepareRequest / PrepareResponse)** Envoyé par l'*InteractionManager* vers les *ModalityComponent*s*,* pour les prévenir de préparer l'environnement avant de commencer son travail. Par exemple, lors de la réception du *PrepareRequest* un écran pourrait être demandé de changer son message affiché depuis français vers anglais avant de l'afficher.

S'il y a plusieurs données, il est possible d'envoyer le *PrepareRequest* plusieurs fois. Néanmoins, pour chaque événement reçu, il faut que le *ModalityComponent* lerépondrait

* **Start (StartRequest / StartResponse)** Envoyé par *l'InteractionManager*, cet événement signale le *ModalityComponent* qu'il peut commencer son travail.

Si le *ModalityComponent* reçu un autre START lors qu'il est un état START, en dépendant de la situation courante, il peut à la fois déclencher la nouvelle tâche, ou annoncer une erreur.

* **Cancel (CancelRequest/ CancelResponse)** Envoyé par *l'InteractionManager,* cet événement demande le *ModalityComponent* d'arrêter sa tâche courante.
* **Pause (PauseRequest/ PauseResponse)** Envoyé par *l'InteractionManager,* cet événement demande le *ModalityComponent* de pauser sa tâche courante.
* **Resume (ResumeRequest/ ResumeResponse)** Envoyé par l*'InteractionManager*, il demande le *ModalityComponent* de résumer la tâche vient d'être pausée.
* **Status (StatusRequest/ StatusResponse)** Le status peut être envoyé par l'*InteractionManager* ou *ModalityComponent* pour annoncer l'état courant du système.

### Notifications MMI

Pour les notifications, comme son nom indiqué, le récepteur n'a pas besoin de les répondre. Souvent, ils se servent juste à informer l'état courant du système. Il existe deux types de notification

* **Extension (ExtensionNotification)** Cette notification est utilisée pour envoyer des données supplémentaires
* **Done (DoneNotification)** Cette notification est envoyée par ModalityComponent vers l'interaction Manager pour notifier sa finition.

// Conclusion, connecteur entrer les deux chapitre

# TRAVAIL RÉALISÉ

À travers ce chapitre, nous allons vous présenter ce que nous avons fait en se basant sur la théorie et l'architecture présentée dans le chapitre 2.

Plus détaillé, nous aborderons tout d'abord l'environnement et méthodologie de travail, ceux qui sont très importants au sein d'un projet informatique. En suit, nous vous détaillerons la mise en œuvre de la plateforme/architecture MMI. Puis, nous créerons quelques services au delà de la plateforme venant d'être construites. En fin, nous vous donnerons quelques critiques sur notre travail.

Au niveau de la structure, ce chapitre se comporte de quatre sections suivantes:

|  |  |
| --- | --- |
| 1. Environnement et méthodologie de travail 2. Mise en oeuvre de l'architecture MMI | 1. Services MMI 2. Critiques |

## Environnement et méthodologie de travail

### Environnement de travail

Au niveau de technique, l'environnement de travail suivant est utilisé:

* **IDE**: Eclipse IDE, Flash Builder, Flash Professional, Aptana Studio.
* **OS**: Linux Ubuntu 11.10, Linux Ubuntu 12.04, Window 7
* **Langage**: Java, Javascript, ActionScript
* **Autre**: Firebug...

### Méthodologie de travail

Une partie très important dans les projets informatiques est la gestion de projet. C'est elle qui décide la répartition du travail, du budget, du temps, la coopération entre les membres, ainsi que le succès du projet. Dans cette section, nous allons aborder sur ce sujet.

À travers le projet, nous avons appliqué une **méthode agile** comprenant des boucles, chaque boucle se divise en trois étapes principales:

|  |  |
| --- | --- |
| * **Première étape:** En étudiant la spécification, nous proposons une conception détaillée utilisant les modèles d'UML. * **Deuxième étape:** Quand la conception est bien finie, nous commençons à l'implémenter. À la fin de cette étape, nous avons une application exécutable. * **Troisième étape:** L'application sera | Figure 6: Méthode d'itération |

ensuite testée pour assurer la fonctionnalité , l'exactitude. Les résultats de teste ainsi que les feedbacks serons utilisés comme l'entrée pour la première étape dans la boucle prochaine.

Pour ça, à chaque phase d'itération, nous pouvons trouver rapidement des fautes/erreurs et les fixer. Nous pouvons également adapter notre implémentation vers les spécifications de MMI pour avoir le meilleur résultat.

Au niveau de **gestion de version**, Subversion est utilisé pour gérer les sources, Ce la facilite la modification, le stockage des codes source. De plus, Il a également utile pour la restauration à partir de l'erreur lors qu'elle se produit.



Figure 7: Gestion de version utilisant SVN

|  |  |
| --- | --- |
| **Suivi des tâches et gestion des erreurs**: Pour faciliter le travail, un projet informatique se divise toujours en plusieurs tâches. Chaque tâche sera ensuite assignée à un ou plusieurs personnes. Ainsi, il faut un outil pour la gestion des tâches et des erreurs.  Notre projet n'est pas une exception, nous avons utilisé Mantis Bug Tracker[[3]](#footnote-4) au sein du projet. Pour ça, nous pouvons enregistrer la déclaration d'un bug informatique, mettre à jour l'avancement de sa résolution, jusqu'à sa clôture.... Les pris des notes, l'attachement des fichiers sont aussi supporté pour mieux décrire la tâche. | Figure 8: Suivi des tâches et gestion des erreurs |

## Mise en œuvre de l'architecture MMI

La mise en œuvre de l'architecture est réalisée dans trois langages de programmation: javascript, actionscript et java. Sachant qu'il n'y a pas beaucoup d'implémentation de l'architecture avant. Nous aimerons bien faire des comparaisons entre l'un avec l'autre. Le but final est de donner des contributions à l'architecture en se basant sur les qualités, défauts de chaque implémentation.

### Couche Réseau: XHR Lib

Description:

La couche de réseau est responsable d'envoyer, de recevoir des événements MMI à travers les réseaux.

Des messages SOAP, JSON ainsi que les méthodes GET et POST sont supportés.

Au niveau d'implémentation:

* XmlHttpRequest est utilisé pour javascript
* EventListener, avec des fonctions de callback est utilisé pour actionscript et java

Travail à réaliser:

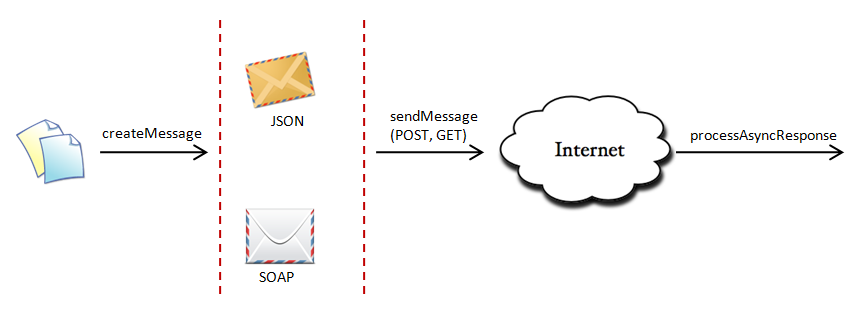


Figure 9: Fonctionnalités de XHR Lib

En se basant sur la responsabilité, nous pouvons se diviser sur les trois rôles principaux (voir Algorithme 3.1):

* Création du message (lignes 5-12)
* Envoie du message (lignes 14-21)
* Traitement du message (lignes 23-27)

|  |
| --- |
| 1 // Create and Sending message  2 function sendAsynchronous(specification, method, type)  3 var message;  4  5 // Create message  6 if (type == "SOAP")  7 message = createSoapMessage(specification);  8 else if (type == "JSON")  9 message = createJsonMessage(specification);  10 else  11 throw "Unknown type exception"  12 end if  13  14 // Send message  15 if (method == "POST")  16 sendPostMessage(message);  17 else if (method == "GET")  18 sendGetMessage(message);  19 else  20 throw "Unknown method exception";  21 end function  22  23 // Process Asynchronous response  24 function onResponseReceived(event)  25 - Check message's validation  26 - Delegate it to the Transport layer  27 end function |

Algorithme 3.1: Algorithme pour l'envoie des messages

Création du message:

La bibliothèque nous donne deux possibilités de créer le message: soit SOAP, soit JSON. Une fois que la fonction *sendAsynchronous* est appelée, nous examinerons le paramètre *method*pour savoir quel type de message devons-nous créer. Sachant que JSON est non seulement plus simple que SOAP mais aussi supporté par plusieurs systèmes par défaut, nous préférons JSON dans notre implémentation.

Envoie du message:

Quand le message est prêt, nous l'envoyons vers le serveur. Le choix de POST ou GET dépend de chaque application. Pour GET, le contenu sera attaché directement dans le lien. Par exemple, vous cherchez le mot clé "UPMC" dans *https://www.google.com/* le lien sera *https://www.google.com/search?q=upmc.* Par contre, dans la requête POST, le contenu sera dans le corps de la requête.

Du coup, si la taille (par example: les *forms* de grande taille) ou la sécurité (le mot de passe) sont considérés, vous devrais utiliser la méthode POST. Si non, GET est aussi un bon choix.

Traitement de la réponse:

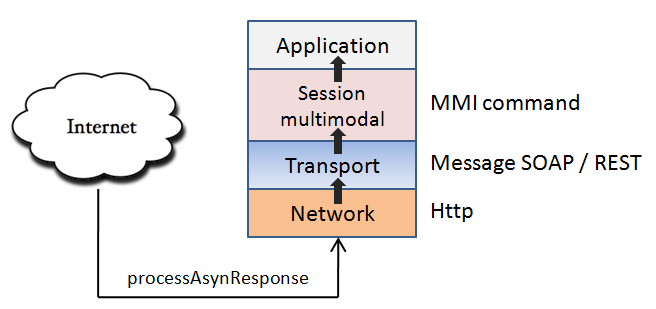


Figure 10: Traitement des messages

Lors que le serveur nous envoyons la réponse, la fonction *onResponseReceived* est appelée. Elle examine le message reçu. Si tout est bien, elle le délègue vers la couche de transport. Ici, le système reconstruira le message SOAP (ou REST) et l'envoie ensuit à la couche de session. Dans la couche de session, nous analyserons ce MMI message pour récupérer des commandes, des paramètres. En fin, nous les envoyons vers la couche d'application.

### Couche Transport: MMI Lib

Description:

La couche de transport est utilisée pour la création des événements MMI. Ils sont des événements proposés par la spécification qui jouent le rôle comme une interface entre l'application et le réseau. Du coup, l'architecture est indépendante du langage utilisé: dès que l'implémentation conforme à la spécification, l'application pourrait communiquer avec d'autres applications implémentées par d'autres langages (voir Figure 11).

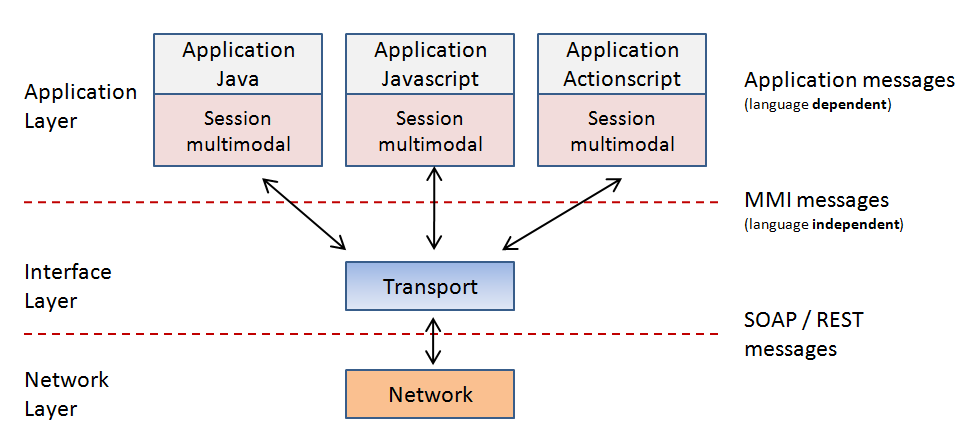


Figure 11: Rôles de la couche de transport

Au niveau d'implémentation, MMI Lib est implémenté en trois langages: Javascript, Actionscript et Java.

Travail à réaliser:

D'après la spécification, il y a deux types d'événement: celui qui existe en paire (la requête et la réponse):

|  |  |
| --- | --- |
| * NewContext Request/Response * Prepare Request/Response * Start Request/Response * Cancel Request/Response | * Pause Request/Response * Resume Request/Response * ClearContext Request/Response * Status Request/Response |

Et celui qui n'existe pas en paire:

|  |  |
| --- | --- |
| * DoneNotification | * ExtensionNotification |

|  |  |
| --- | --- |
|  |  |

Figure 12: MMI Lib

Malgré des attributs différents, ils partagent plusieurs attributs communs. C'est pourquoi nous les avons implémentés utilisant l'héritage. La classe de base s'appelle *MessageBase.* En se basant sur cette classe, nous avons construit d'autres classes: *MessageRequest*, *MessageResponse*pour ceux qui existe en paire et *Notification*pour ce qui n'existe pas en paire (voir Figure 12). Notons que nous utilisons le *dessign pattern Factory.* C'est un *design pattern* pour la séparation des dépendances entre les classes, celle qui est très importante dans notre architecture multimodal.

Voici Algorithme pour la création des messages de requête (pour les messages de réponse, les notifications, l'algorithme est pareil)

|  |
| --- |
| 1 function createRequest(spec)  2 var msgType = spec.getMsgType();  3 if (msgType == MessageType.NEW\_CONTEXT\_REQUEST;  4 // Create New Ctx Request  5 return NewContextRequest;  6 end if (msgType == MessageType.START);  7 // ...........  8 // ...........  9 else  10 throw "Unknown type exception"  11 end if  12 end function |

Algorithme 3.2: Création des messages

Après la récupération le type du message (ligne 2), nous le tester contre les prédéfinis types (lignes 3-8) pour créer le message convenable. Si non, on lancera une exception (ligne 10).

### Couche Session: Architecture Lib

Description:

La couche Session est responsable de:

* Enregistrer l'application avec le serveur.
* Stocker des informations concernant la session (l'id, le contexte, la durée de vie, les configurations... de l'application).
* Analyser les messages reçus pour donner des commandes vers la couche application.

Au niveau de la structure de la bibliothèque, cette couche est représentée par la bibliothèque Architecture Lib dans laquelle la classe *Controller* est la classe principale. Près que tous les actions importantes liées à la session sont contrôlées par des méthodes se trouvant dedans cette classe. Ces méthodes peuvent directement lancer des traitements ou bien, demander à des méthodes d'autres classes pour lui aider.

Travail à réaliser:

|  |  |
| --- | --- |
|  |  |

Figure 13: Bibliothèque Architecture Lib

Enregistrement de l'application:

Nous examinerons tout d'abord les protocoles pour l'enregistrement d'un composant avec le serveur (voir la Figure 14). En fait, le processus se comporte des étapes suivantes

* Chargement des configurations via la fonction *load()*: Pendant cette étape, des informations comme l'adresse du serveur, l'adresse des services web, les variables... sont chargés. Ces variables se servent à instancier le *Parser*, *Timer*, *SocketManager*...
* Enregistrement de l'application via la fonction *register()*: Nous envoyons une requête pour savoir s'il le serveur et disponible ou pas.
  + Si non, nous attendrons jusqu'à ce qu'un intervalle indiqué dans le Timer se soit passé. Et relancer la demande.
  + Si oui, nous continuerons l'étape suivante.
* Avertissement des changements via la fonction *inform()*: Après recevoir la réponse depuis le serveur, nous mettons à jour notre système. Le changement est ensuite informé au serveur par la fonction *inform()*. Quand le serveur reçoit la requête, il confirme sa réception en donnant des informations supplémentaires vers le client. Ces informations sont utilisées pour mettre à jour l'état ainsi que l'horloge du système.
* Envoie du message *NewContextRequest*: Une fois que le client est synchronisé (au terme des données) avec le système. Il commence à envoyer sa requête vers le serveur pour demander le nouveau contexte.

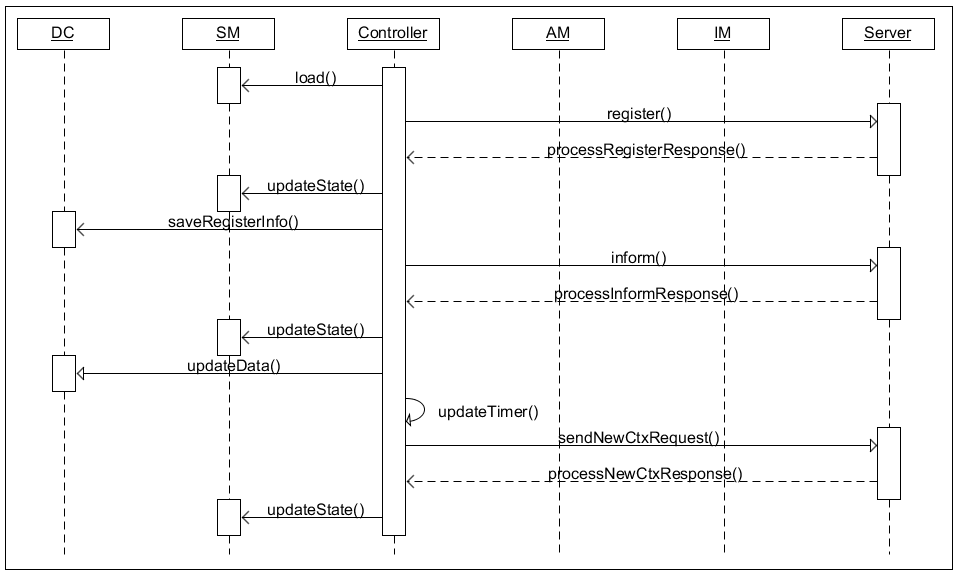


Figure 14: Enregistrement de l'application avec le serveur

Voici l'algorithme pour l'enregistrement de l'application:

|  |
| --- |
| 1 loadConfigFile()  2  3 while (hasRegister == false)  4 sendRegisterRequest();  5 end while  6  7 // have registerId, update the system  8 inform();  9 updateState();  10 updateDate();  11  12 while(hasContext == false)  13 sendContextRequest()  14 end while  15  16 // found context, change to stand by state  17 // ............. |

Algorithme 3.3: Enregistrement d'une application

Notons que nous faisons une boucle (ligne 12-14) jus ce que le contexte est trouvé. La différence entre un *contextId* et un *registerId* est: *ContextId* est partagé entre plusieurs applications dans la même session gérée par le serveur tans dis que *RegisterId* est per chaque application.Il est utilisé par le serveur pour lui identifier dans la session indiquée par *contextId.*

Ainsi, une application devrait s'enregistrer deux fois avec le serveur: la première fois pour demander son identifier et la deuxième fois pour demander la session avec laquelle elle va travailler. Les deux informations sont stockées dans le fichier de configuration pour que l'application ne doive pas se réenregistrer.

Une fois que nous trouvons le contexte, c'est-à-dire l'Algorithme 3.3 s'est bien passé, nous commençons la deuxième étape: Interaction avec des commandes.

Interaction avec des commandes:

Maintenant, supposons que l'application a son identifier et l'identifier du contexte auquel il participera, il pourrait attendre le commande depuis le serveur.

Comme nous avons abordé tout à l'heure: il y a deux types d'application:

* Composant contrôlant (dit A1) qui produit des commandes, par exemple, le clavier, il produit des événements en se basant sur quels boutons sont tapés.
* Composant contrôlé (dit A2) qui reçoit des commandes, par exemple, l'écran, un écran reçoit des messages et lui affiche en suite.

Pour les applications de type A2, il n'y a qu'un type de commande envoyé pas le serveur. Néanmoins, pour le type A1, il existe deux types de commande totalement différentes (voir la Figure 15):

* Les commandes produites par l'application elle-même pour contrôler d'autre application (en couleur rouge, dit **commande de l'application**) à condition que l'application soit en mode activé
* Les commande produits par le serveur pour déclencher, terminer l'application (en couleur noir, dit **commande du système**). Par exemple, la commande pour désactiver le clavier. À ce moment là, les boutons (en rouge) sont inutiles.

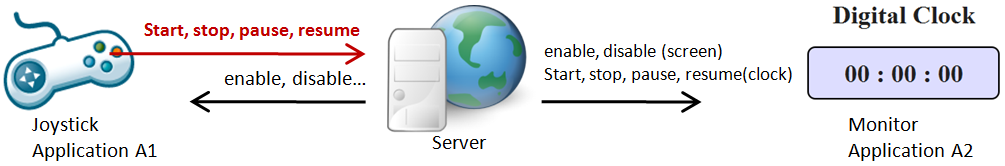


Figure 15: Deux types de commandes

Quand nous avons bien distingué entre les deux types, nous allons examiner comment le serveur pourrait contrôler ses clients via les commandes du système, et comment les applications contrôlant pourraient envoyer des événements.

Recevoir ses commandes:

Nous avons utilisé deux algorithmes: *push* et *pull* et trois implémentations:

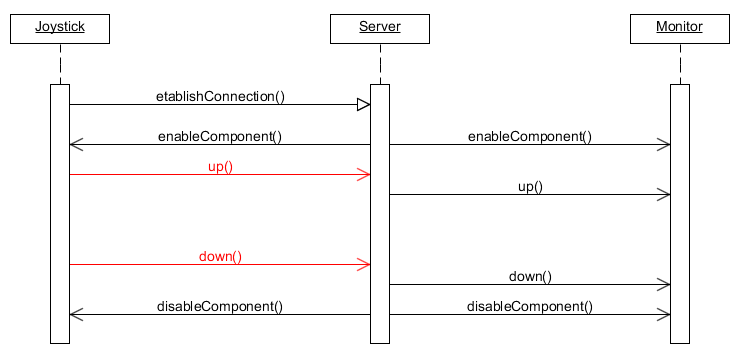
* Comet (push) pour les composants javascript
* Socket (push) pour les composants java, actionscript
* Sondage périodique (pull) pour tous les trois composants.

*L'implémentation du Comet* n'est pas complexe (par exemple, Monitor et Server). Il suffit d'intégrer la balise *<iframe>* dans notre composant. Le serveur va communiquer avec le client via la balise. La technique détaillé a été décrite dans la spécification de *Comet.*

Quant à *l'implémentation utilisant Socket* (Joystick et Server), nous devons établir la connexion. Si tout va bien, l'application tournera en arrière en attendant le message. Si le serveur voulait l'envoyer un message, par exemple, pour activer le composant, il utiliserait le socket. À côté client, le message est traité de couche après couche (voir la Figure 10).

En fin, c'est l'implémentation utilisant le mécanisme de sondage. Il sera abordé dans la partie **Error! Reference source not found.**

Notifier ses applications:



Dans la mode asynchrone, chacun fait son travail: la couche session tourne en arrière juste pour attendre des commandes et l'application, en haute, fait son métier. Du coup, il faut une façon pour la communication entre les deux. D'ou la notification arrive. En actionscript, l'événement et la notification est native, en javascript, nous pouvons le faire grâce aux variables globales. Néanmoins, pour Java, en suivant le principe de l'objet orrienté, les deux sont strictement séparés. Alors, qu'on va faire ? Veuillez consulter la Figure 16pour l'explication:

|  |  |
| --- | --- |
|  | 1 // In SocketManager Object  2 this.addEventListener(Controller)  3  4 function fireEvent(SocketEvent)  5 for (listener : listeners)  6 listener.onDataReceived(SocketEvent)  7 end for  8 end function  9  10 // In Controller Object  11 function onDatareceived(SocketEvent)  12 interactionManager.process(SocketEvent)  13 end function |

Figure 16: Notification des événements.

Dedans le *SocketManager*, nous ajoutons le *Controller* comme un EventListener (ligne 2), Une fois que le *SocketManager* reçoit un événement depuis le Serveur, il le lance via la fonction *fireEvent*. Comme le Controller est déjà enregistré pour ce type d'événement, il se réveille et demande l'InteractionManager de faire son travail (lignes 10-12). Lui, à son tour, va ensuite mettre à jour l'interface GUI.

## Services MMI

### Enregistrement du son

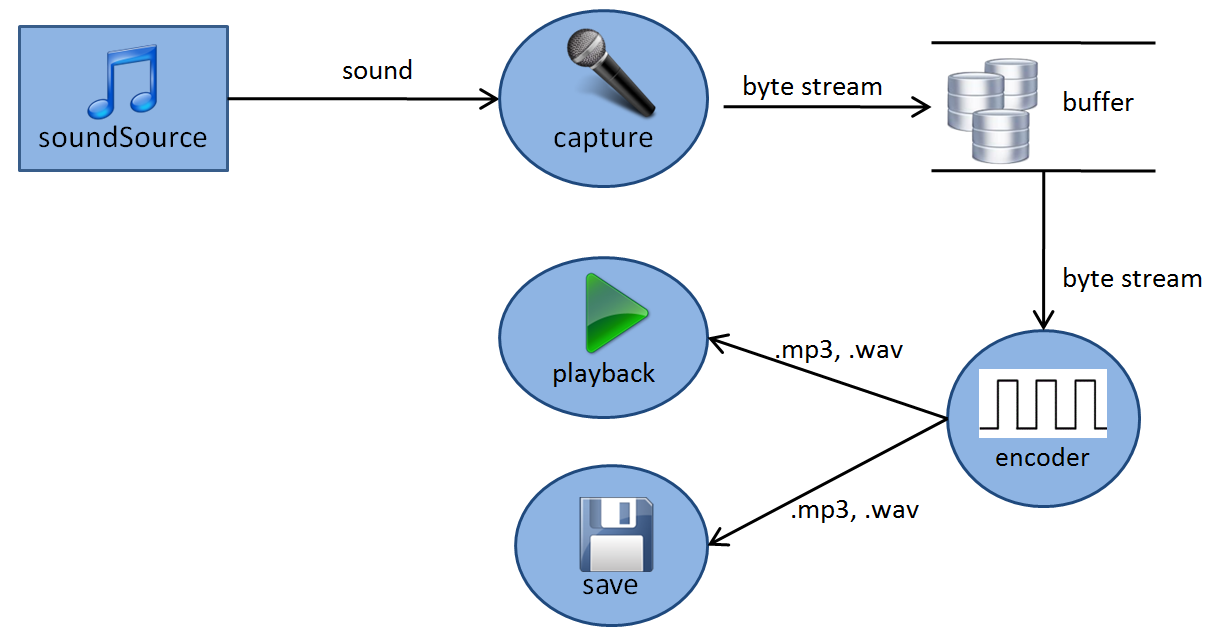


Figure 17: Enregistrement du son, fixer error

Description:

Cette tâche est pour but de créer un composant capable d'enregistrer et stocker du son (Figure 17)

Travail réalisé:réré qsdf

L'enregistrement du son est implémenté en Adobe Air, utilisant actionscript.

Pour la capture du son, un microphone est utilisé. Le son capturé est stocké dans un buffer. L'algorithme est comme suivant:

|  |
| --- |
| 1 mic.addListener(onRecording);  2 function onRecording (Event event)  3 while (event.hasData)  4 sample = readData(event);  5 writeToBuffer(sample);  6 end while  7 end function |

Algorithme 3.4: Enregistrement du son

Tout d'abord, la fonction **onRecording** est attachée au microphone un tant que un *eventListener (*ligne 1). Lors qu'il y a du son, cette fonction sera appelée et lui, à son tour, va lire les données. En suit, elle les stocke dans le tampon (ligne 4-5).

En suit, on peut à la fois re-jouer le son, ou bien le sauve sous un format supporté. Notre application ne supporte que trois types d'extension: mp3, wav et flac. Pour chaque type, il faut une étape de prétraitement s'appelant *encode.* Ce processus prend les flux d'octets comme l'entrée et produit des fichiers de type mp3, wav ou flac selon notre choix.

### Transformation de la parole en texte (Speech-to-Text)

Description:

La transformation de la parole en texte est implémentée en Adobe Air, utilisant actionscript.

Comme son nom indiqué, cette application prend du son en entrée et sort le texte-format contenant le contenu du son d'entrée.

Nous avons deux approches:

* Soit construire notre propre transformateur tout au début.
* Soit utiliser d'autres APIs qui sont disponibles en ligne.

Sachant que la transformation demande des connaissances en signaux numériques qui ne sont pas ma spécialité, que ce module se sert juste pour montrer la faisabilité de l'architecture MMI et que Google a crée un très bien API pour nous, j'ai choisi la deuxième approche.

Travail réalisé:réré qsdf



Figure 18: Transformation de la parole en texte

Comme nous avons abordé, la transformation est réalisée grâce à un api de Google[[4]](#footnote-5). En fait, c'est un api un-officiel réservé uniquement pour Google Chromium. Ce qu'on va faire, c'est de construire une application pour imiter le comportement de Google Chromium. Le travailse compose de deux étapes (voir Algorithme 3.5)

**Conversion**: Faire la conversion de *.extension vers .flac* ou .extension soit *.mp3*, soit *.wav*. Comme Google ne supporte que *.flac* comme entrée de la fonction de transformation, cette étape est obligatoire.

**Traitement:** Le traitement se divise encore en deux sous parties:

* **Envoyer**: Nous devons fournir des variables avant qu'on appelle l'API.
* **Récupérer les données:** En appelant la fonction nous récupérons un String contenant le contenu du son. ce variable sera sauvegardé pour l'utilisation après.

|  |
| --- |
| 1 // Conversion  2 var flacData = convertToFlac(mp3Data);  3 // Traitement  4 var language = "FR";  5 var resultTxt = convertSpeechToTxt(flacData, language);  6 save(resultTxt); |

Algorithme 3.5: Transformation de la parole en texte.

### Transformation du texte en parole (Text-to-Speech)

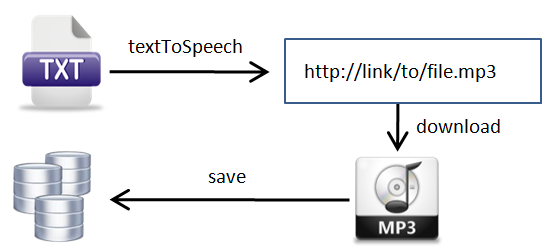
Description:

La transformation du texte en parole est implémentée en Adobe Air, utilisant actionscript.

Au niveau de la fonctionnalité, cette application est opposée à celle décrite dans 3.2: Il prend un String et essaie de produire le fichier .mp3 équivalent.

Comme 3.2, nous avons utilisé un autre API de Google: Google Translate. Ce qui est intéressant c'est que Google Translate supporte près que toues les langues dans le mode. Autrement dire, vous pouvez travailler avec autant nombre de langues comme Google Translate, mais directement depuis notre application.

Travail réalisé:réré qsdf



Algorithme 3.6: Transformation du texte en parole.

Pour la partie d'implémentation, nous devons tout d'abord choisir quelques lignes de texte d'entrée ainsi que la langue. Ensuite, nous appelons Google Translate utilisant les configurations ci-dessus (texte, langage).

|  |
| --- |
| 1 var language = "FR";  2 // Convert  3 var mp3Link = convertTxtToSpeech;  4 var mp3Sound = dowload(mp3Link);  5 save (mp3Sound); |

Algorithme 3.7: Transformation du texte en parole.

Au niveau de résultat, si tout est bien, nous recevrons un lien contenant le fichier mp3. Nous téléchargeons ce fichier là et lui stockons pour l'utilisation après.

### Reconnaissance des gestes

Description:

Le but de ce composant est de simuler la fonctionnalité de l'écran tactile. En se basant sur les gestes de l'utilisateur, une commande se produira (voir Figure 19). Cette commande va ensuite envoyer vers la couche de session, puis, la couche de transport...

Cette section va vous présenter l'algorithme pour la production des commandes à partir des gestes de l'utilisateur.

Travail réalisé:réré qsdf

À partir des gestes, nous passons par trois étapes jusqu'à la sortie des commandes correspondantes:

* Récupération des données de gestes.
* Adaptation des données récupérées.
* Interpolation

Pour mieux comprendre l'algorithme, considérons un exemple dans lequel un utilisateur utilise son doigt pour désigner (v) caractère sur l'écran tactile. Nous passons par trois étapes suivantes:

|  |  |
| --- | --- |
| **Récupération des données de gestes**: Dans ces étapes, nous construisons des vecteurs de direction à partir des gestes.  Pour chaque l'intervalle de temps, nous récupérons la coordonnée du doigt et la stocke dans un point (disons P2). Avec la dernière coordonnée (disons P1), nous créons un vecteur de direction et les stocker (lignes 3-8)  À la fin de cette étapes, nous avons une liste des vecteurs de directions (v1....v6) décrite la trace du doigt d'utilisateur. | 1 function getCommand  2 while (collectMode)  3 P1 = P2;  4 P2 = getCurrentPos();  5 vector = Vector(P1, P2);  6 vectorList.add(vector);  7 sleep(interval);  8 end while  9 return vectorList  10 end function |

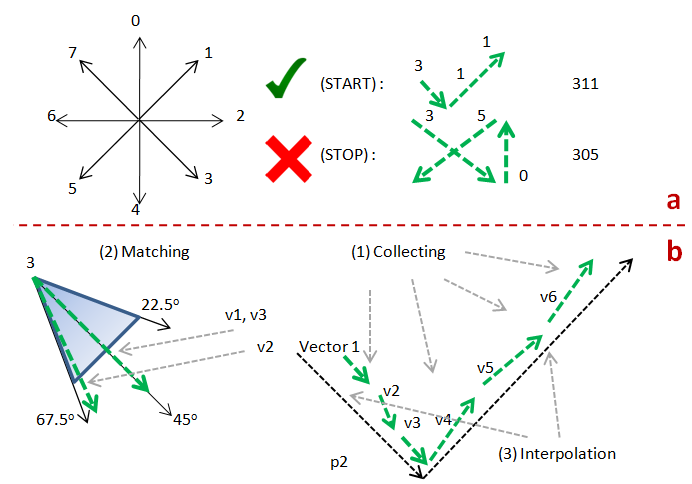


Figure 19: Reconnaissance des gestes

**Adaptation des données récupérées:** Avec la liste des vecteurs. Nous effectuons une adaptation pour chaque vecteur. Autrement dire, il s'agit d'une normalisation d'un vecteur quelconque vers un des huit vecteurs de direction décrits ci-dessus. Ainsi, les trois vecteur v1, v2, v3 sont considérés comme les trois vecteur de type 3. Comme ça, la sortie de cette étape est le string 333111. Avec ce string, nous passons à l'étape suivante.

|  |  |
| --- | --- |
| **Interpolation:** Maintenant, quand nous avons le résultat obtenu (333111), Supposons que les données d'échantillon sont stockées dans une liste, nous parcourons la liste. Pour chaque élément, nous calculons la Distance de Levenshtein entre les deux et sauvegardons la valeur la plus petite.  À la fin de la boucle, nous avons une échantillon la plus proche. Dans notre cas, c'est 311. Nous l'utilisons pour obtenir le command START. | 1 function getCommand(gesture)  2 for each sample in sampleList  3 distance = getDistance(gesture, sample)  4 if (distance < min) then  5 min = sample  6 end if  7 end for  8 command = getCommandFromId(min)  9 return command  10 end function |

### Découverte des services locaux

### Autres services

#### Horloge

L'horloge est un composant contrôlé très simple implémenté en Javascript qui affiche le temps sur l'interface GUI. Il est crée juste pour montrer la réception d'une commande depuis le serveur. (En fait, le serveur est juste un courtier, il reçoit la commande depuis des composants contrôlant. Et puis, il la transfère vers l'horloge, qui est un composant contrôlé.

Etant un composant MMI, le fonctionnement de l'horloge passe de l'étape par l'étape (voir la Figure 14).

* La première étape est *LOAD,* ce composant, via son contrôleur, charge sa configuration. Entre temps, le contrôleur lance un événement pour indiquer son état courant. En implémentant un listener d'événement, notre l'interface GUI le récupère et affiche "*Now Loading*" comme la réponse.

|  |  |
| --- | --- |
| * La deuxième étape comporte des messages *register(), inform()...* Pendantcette étape, le système lance l'événement *PREPARE\_BEFORE\_NEW\_CTX*. Comme la réponse, l'interface affiche "*Prepare before new context request sent...*". Le texte ici joue le rôle comme une espace réservé. En fait, au lieu d'afficher du texte, nous pouvons faire des préparations pour ce composant avant l'envoie du message *NewContextRequest*   Dans troisième étape, le système périodiquement envoie la requête *NewContextRequest*. Pour les trois premières fois, le serveur n'est pas disponible, du coup, il devrait redemander (avec l'intervalle indiquée dans le *TimerData).* |  |

* En fin, une fois que le serveur accepte sa requête, il lui réponse avec une *contextId*. L'horloge est désormais disponible et pourrait recevoir des commandes (START, PAUSE...)

#### Commandeur

|  |  |
| --- | --- |
| Le commandeur est un composant contrôlant implémenté en Javascript, il s'agit d'un composant très simple ayant des boutons capable d'envoyer des commandes (START, STOP, PAUSE, RESUME).  Pour que ces soit fonctionné, il devait passer par des étapes: le chargement, l'enregistrement, la demande du contexte. Ces étapes seront détaillées dans le composant d'horloge.  Une fois qu'il est prêt, il pourrait envoyer des commandes vers les composants contrôlés (indirectement via le serveur). |  |

#### Calendrier

Calendrier est aussi un composant contrôlé implémenté en Java. Il passe également par trois étapes comme notre l'horloge. Pour simplifier les choses, supposons que ce composant est déjà prêt (c'est-à-dire, il a déjà le *contextId)* et il a reçu toutes les données, les informations nécessaires pendant l'étape *PREPARE\_BEFORE\_NEW\_CTX.*

Via la connexion Socket, il reçoit la commande *START (a).* En traitant la commande, il rend disponible l'interface GUI (b et c). Les informations affichées se trouvent dans le fichier XML donc la structure est dans la Figure 20

|  |  |
| --- | --- |
| Le contenu du fichier sera analysé par le *XmlParser* et sera stocké dans les objets Java:   * la balise <dailyData> devient la classe *DailyData* * la balise < dailyEvent > devient la classe *DailyEvent* |  |

Comme ça, le composant lit les objets et affiche des informations sur l'interface GUI jusqu'à ce que la commande STOP soit reçue.

|  |  |
| --- | --- |
|  | <dailyData date="13/08/12">  <dailyEvent>  <startHour>9</startHour>  <endHour>11</endHour>  <room>DB101</room>  <description>  Soutenance de thèse  </description>  </dailyEvent>  </dailyData>  <dailyData date="14/08/12">  <dailyEvent>  <startHour>7</startHour>  <endHour>9</endHour>  <room>DB401</room>  <description>  Printing  </description>  </dailyEvent>  <dailyEvent>  <startHour>13</startHour>  <endHour>15</endHour>  <room>DB206</room>  <description>  Entretien téléphonique  </description>  </dailyEvent>  ........................  </dailyData> |

Figure : Le calendrier.

## Critiques

### Résultat obtenus

voir hut report semestre 8

### Comparaison

### Travail à faire

# CONCLUSION

# BIBLIOGRAPHIE

1. http://www.w3.org/TR/2012/PR-mmi-arch-20120814/
2. Michelle Kim, Steve Wood, Lai-Tee Cheok , Extensible MPEG-4 textual format (XMT), Proceeding MULTIMEDIA '00 Proceedings of the 2000 ACM workshops on Multimedia Pages 71-74
3. http://www.w3.org/Graphics/SVG/

1. Maes, P., Concepts and Experiments in Computational Reection, In Proceedings of OOPSLA'87, Vol. 22 of ACM SIGPLAN Notices, pp147-155, ACM Press, 1987.

2. Gordon S. Blair , Geo Coulson , Nigel Davies , Philippe Robin , Tom Fitzpatrick , Adaptive Middleware for Mobile Multimedia Applications, 1997

3. G.S. Blair , G. Coulson , P. Robin , M. Papathomas, An Architecture for Next Generation Middleware, Middleware, IFIP International Conference on Distributed Systems Platform and Open Distributed Processing (Middleware'98), Lake District

# ANNEXES

## Recherches en multimédia

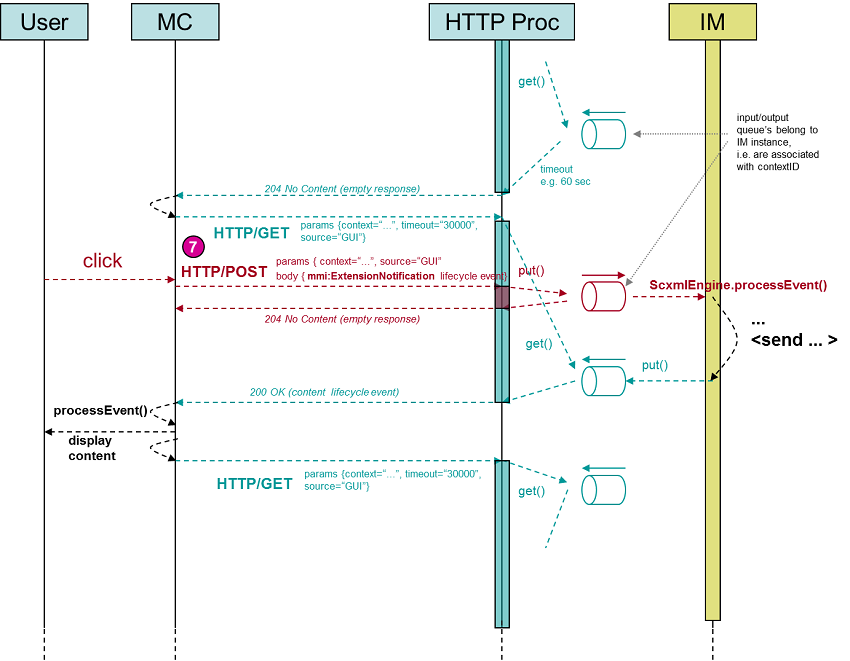
### Gpac

### SVG

### UPnP

## Diagrammes

## http://www.w3.org/TR/2012/PR-mmi-arch-20120814/Images/HTTP_lifecycle_transport_4.png



1. http://intermedia.miralab.ch/ [↑](#footnote-ref-2)
2. gpac.sourceforge.net [↑](#footnote-ref-3)
3. http://www.mantisbt.org/ [↑](#footnote-ref-4)
4. https://www.google.com/speech-api/v1/recognize?xjerr=1&client=chromium [↑](#footnote-ref-5)