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
		Thésarde. Bertha Helena Rodriguez
Étudiant	***:***	***Xuan-Hoa Nguyen***
Formation	***:***	***Master 2 STL - APR***
Année universitaire	***:***	***2011 - 2012***
Période du stage	***:***	***04/2012 - 09/2012***
Laboratoire d'accueil	***:***	***Télécom ParisTech***

Paris 09/2012

FICHE DE SYNTHÈSE

1. Contexte général du stage

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.

État de l'art:

Dans nos jours, le rôle des dispositifs intelligents, par exemple les baladeurs multimédia, tablettes, smartphone, est indéniable. Néanmoins, sachant que les dispositifs sont très divers au niveau du matériel, des plateformes et des implémentations, il est encore un gros défi pour les réunir dans un seul système multimodal qui simplifiant notre vie quotidienne. Afin d'adresser ce problème, Télécom ParisTech, en collaboration avec Alcatel-Lucent, a réalisé un projet de recherche nommé Ubimedia. Mon stage s'agit d'une continuité de travaux de ce projet.

Sujet de stage:

Le sujet de stage est "*Outils pour la découverte et la composition de services d'interface utilisateur en environnement pervasif*". Concrètement, le travail porte sur l'implémentation de l'architecture MMI[1] architecture proposé par W3C dans le cadre du projet Soa2M (Service-Orriented Architectures for Multimedia) au sein de l'équipe Multimédia de Télécom ParisTech.

2. Problème étudié

À travers le stage, les problèmes suivants sont considérés:

- Études de l'architecture MMI (Multimodal Architecture and Interfaces) [1], proposé par W3C.
- Implémentation de cette architecture.
- Implémentation des composants de modalité (*Multimodal Component*).

3. Contribution

Lors de mon stage, mes contributions notables sont:

Premièrement, j'ai implémenté le cœur de l'architecture MMI en utilisant *Java*, *JavaScript* et *ActionScript*. Plus en détail, Il comprend des missions suivantes:

- Implémentation des mécanismes pour la communication entre les composants dans l'architecture MMI.
- Implémentation des messages MMI suivant sa spécification.
- Implémentation un cycle d'interaction selon ce que propose la spécification MMI.

Deuxièmement, j'ai construit une application en *ActionScript* capable de faire la transformation de la parole en texte. Dans le meilleur de mes connaissances, c'est la première implémentation en *ActionScript*[2]. De plus, j'ai également implémenté un algorithme pour la reconnaissance des gestes.

Troisièmement, j'ai proposé un nouveau mécanisme pour la synchronisation entre le serveur et ses clients nommé *Adaptative Pull* ainsi qu'un mécanisme de fonctionnement en se basant sur des états.

---

[1] http://www.w3.org/TR/2012/PR-mmi-arch-20120814/

[2] Pour plus d'info, veuillez consulter un article sur mon blog:
http://nxhoaf.wordpress.com/2012/05/11/speech-to-text-in-action-script-3/

4. Analyse du travail

Au point de vue théorique, mon travail donne des contributions à l'architecture MMI. De plus, ma proposition du mécanisme *Pull Adaptative* définit une nouvelle façon pour la synchronisation des donnés entre le serveur et le client sans consommer trop de ressource du réseau.

Au point de vue expérimental, mes implémentations (en *Java, JavaScript et ActionScript*) est l'une des premières implémentations de l'architecture MMI. Avec les libraires implémentées dans les trois langages, ce serait très facile pour d'autres personnes de les réutiliser, ou de les étendre.

La plus grande limitation insiste sur la sûreté des librairies. Pendant mon stage, j'ai dû passer un gros nombre de travail dans un temps assez cours. Du coup, je n'ai pas pu tout tester ces librairies. Cela devrait y aller comme je continuerai mon travail avec l'équipe Multimédia après mon stage.

5. Bilan et perspectives

Pour la récapitulation, mon travail porte sur le côté client avec plus de 10 000 lignes de code sources environs.

Au niveau de la recherche, mes contributions par rapport à la spécification sont:

- Proposition d'un mécanisme de communication nommé *Adatative Pull.*
- Proposition d'un mécanisme de fonctionnement en se basant sur des états.
- Ajout du patron de conception *Factory* en spécification.
- Proposition des services de notification pour la surveillance de la session multimodale, de l'état du système.
- Proposition d'un mécanisme de délégué pour que le composant de contrôle puisse déléguer ses travails vers ses sous composants.

Au niveau d'implémentation, vous voyez ci-dessous un tableau détaillé qui résume ce que j'ai effectué ainsi que son résultat lors de mon stage:

Description	Fini	Non fini
Recherche		
Études du projet Open Source GPAC	x	
Études des extensions .bifs, .xml	x	
Études de la norme SVG	x	
Études de l'architecture MMI proposé par W3C	x	
Études du langage SCXML		x
Développement		
Le cœur de l'architecture MMI en JavaScript	x	
Le cœur de l'architecture MMI en ActionScript	x	
Le cœur de l'architecture MMI en Java	x	
Transformation de la parole en texte (STT)	x	
Transformation du texte en parole (TTS)	x	
Reconnaissance des gestes	x	
Autre applications: Découverte des services en GPAC, Calendrier en Java...	x	
Le moteur de décision		x

Dans les jours qui viennent, nous continuerons notre travail sur l'implémentation d'un moteur de décision en utilisant le sémantique web à côté serveur. Ainsi, le serveur pourrait donner ses meilleures décisions en fonction du contexte courant.

REMERCIEMENTS

Je voudrais manifester ma reconnaissance sincère à M. Jean-Claude MOISSINAC, mon maître de stage, de m'avoir sélectionné pour ce stage, de sa surveillance constante et de son intérêt tout au long de mon stage. Il m'a donné de bons conseils et m'a transmis beaucoup de nouvelles connaissances professionnelles grâce auxquels je pourrais perfectionner mes compétences de recherches.

Je remercie sincèrement M. Emmanuel CHAILLOUX, mon responsable, pour ses conseils, ses encouragements et son suivi tout au long de mon stage.

Mes remerciements vont également à Mme Bertha Helena RODRIGUEZ, doctorante au sein de l'équipe, avec qui j'ai partagé le bureau, qui m'a laissé toujours la porte ouverte et m'aidé en toute circonstance, qui m'a posé des problèmes à résoudre. C'est elle qui a toujours répondu, avec enthousiasme, à mes questions.

Je tiens aussi à remercier M. Jean-Claude DUFOURD, M. Jean LE FEUVRE, M. Cyril CONCOLATO pour ses aides ainsi que ses conseils qui m'ont permis de me progresser sans cesse durant ces six mois de stage.

Ma gratitude va à toute l'équipe TSI de Télécom ParisTech pour leur accueil, ce qui a rendu agréable ma mission.

Paris, le 04/09/2012

NGUYEN Xuan Hoa

TABLE DE MATIÈRES

TABLE DE FIGURES

CHAPITRE 1 : 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

1. 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.

2. Le stage recherche STL

3. 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 m'a beaucoup intéressé. Il s'agit d'une continuité de travaux qui a débuté dans le cadre de la collaboration entre Telecom ParisTech et Alcatel-Lucent sur le programme de recherche Ubimedia.

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 recherche. De plus, mes compétences de recherche seraient aiguisées grâce à un environnement très excellent chez Télécom ParisTech.

3.1. Description du sujet

Mon stage a débuté le 02 avril 2012 et se terminera le 14 septembre 2012 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 de 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...

3.2. Missions

Mon travail porte sur l'implémentation d'un protocole de communication des données et de contrôle pour des systèmes multimodaux suivant la spécification MMI (Multimodal Architecture)[1]

Concrètement, le travail se divise en deux axes: la recherche et le développement avec les missions ci-dessous:

Au niveau de la recherche

- Études du projet Open Source GPAC[2], des extension .bifs (BInary Format for Scenes), .xmt (eXtensible MPEG-4 Textual Format) [4].
- Études de la norme SVG [4], du protocole UPnP.
- Études de l'architecture MMI proposée par W3C à laquelle l'équipe Multimedia contribue.

[1] www.w3.org/TR/2012/PR-mmi-arch-20120814/
[2] gpac.sourceforge.net

- Études de l'architecture Soa2m (Service-Oriented Architectures for Multimedia) proposée par Télécom ParisTech.

Au niveau de développement

- Implémentation du cœur de l'architecture MMI.
  - Communication via les réseaux : *XHR Lib*.
  - Événements MMI *: MMI Lib*.
  - Cycle d'interaction: *Architecture Lib*.
- Implémentation des composants de modalité:
  - 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.
  - Autres:
    - Commandeur simple avec des boutons.
    - Afficheur: L'horloge.
    - Afficheur: Le calendrier.

CHAPITRE 2 : ARCHITECTURE MMI

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

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

1. État de l'art

Une des tendances de développement de la technologie d'information aujourd'hui, c'est le système d'information pervasif. 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 selon le contexte mais aussi être si interactifs qu'ils puissent communiquer avec d'autres objets intelligents dans les réseaux [Figure 1].

Evidement, la perspective est très brillante. Néanmoins, sachant que les dispositifs sont très divers au niveau du matériel, des plateformes et des implémentations, il est encore un gros défi de construire un tel système.

Pour tenir compte de ce défi, quelques organisations ont proposé les solutions pour adresser ces problèmes. Parmi eux, celui du W3C nommé *Muitimodal Architecture and Interfaces (ou l'architecture MMI)* est très complet et détaillé.

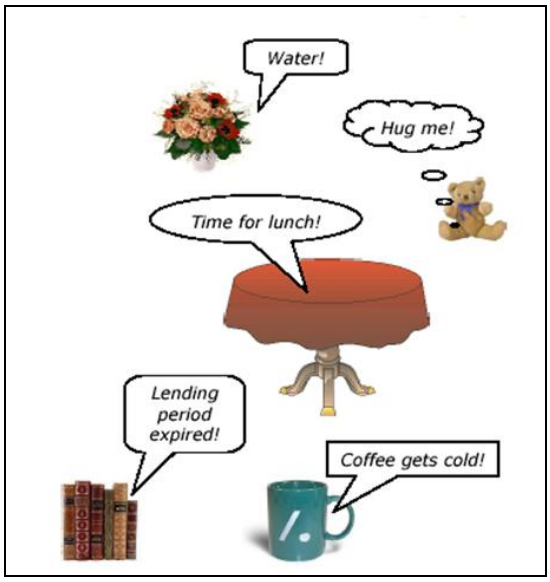


Figure 1: Informatique pervasif
Source : Internet

Alors, qu'est ce qu'il a proposé? Nous continuons à la section suivant : **l'aperçu de l'architecture MMI** pour mieux comprendre la réponse.

2. Aperçu de l'architecture

L'architecture multimodale est un standard développé par W3C depuis 2002 [Figure 2]. Le standard propose une architecture pour faciliter la communication entre ses composants dans un environnement réparti, hétérogène.

La communication se fait de façon asynchrone suivant une architecture orientée événement. Ainsi, à condition que les normes soient respectées, chaque composant peut être implémenté de telle façon qu'il ne dépend pas du langage utilisé, mais ayant capable de communiquer l'un avec l'autres.

L'architecture affirme que les exigences ci-dessous doivent être assurées:

- **Encapsulation**: L'implémentation détaillée de chaque composant est considérée comme une boîte noire.
- **Distribution:** L'implémentation de manière répartie est supportée.
- **Extensibilité**: Ce n'est pas difficile d'ajouter de nouveaux composants à n'importe quel moment.
- **Récursivité:** Chaque composant pourrait avoir ses sous-composants. Cela diversifie ses fonctionnalités ainsi que son autonomie.
- **Modularité**: L'architecture assure la séparation entre les trois tiers: données, contrôle, et présentation.

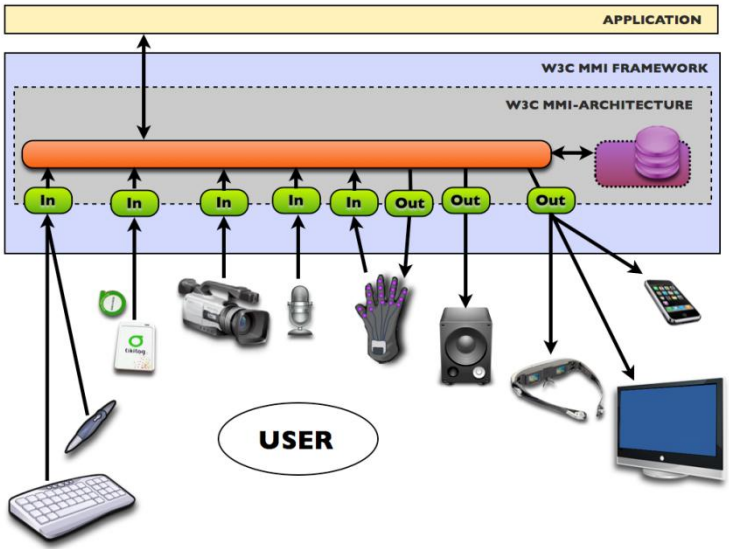


Figure 2: Architecture Multimodale.
Source: http://en.wikipedia.org/wiki/Multimodal_Architecture_and_Interfaces

Avec ces concepts de base, nous attaquons maintenant sur la section suivante qui discute sur des composants principaux de l'architecture.

3. Composants principaux

L'architecture MMI est en accord avec le modèle MVC, les trois composants principaux sont:

- *InteractionManager* (Contrôleur)
- *ModalityComponent* (Vue)
- *DataComponent* (Modèle)

3.1. InteractionManager

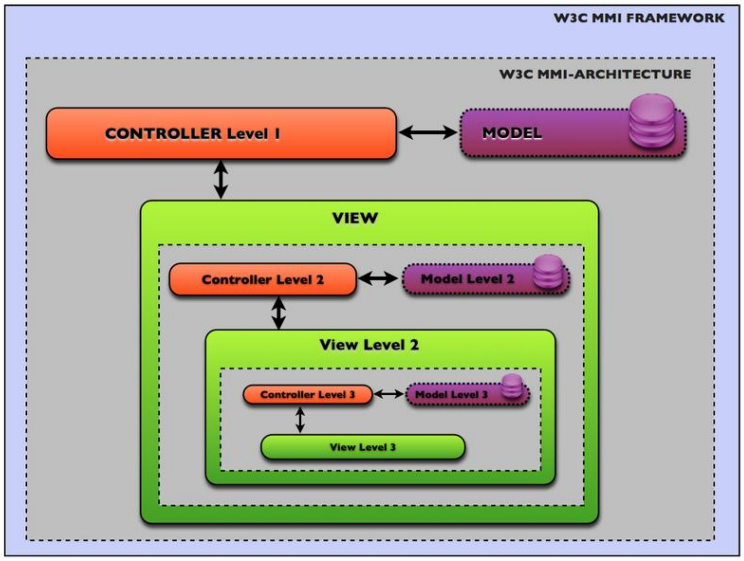


Figure 3: Framework et ses composants
Source: http://en.wikipedia.org/wiki/Multimodal_Architecture_and_Interfaces

InteractionManager, ou bien Contrôleur [Figure 3] est un composant logique chargé d'échanger les messages entre les composants. C'est dans ce composant qu'on gérera la communication et la gestion des événements. Chaque application devrait avoir au moins un contrôleur pour:

- Gestion des événements, ainsi que du mécanisme pour la production, le traitement d'un événement.
- Gestion des communications entre les composants de modalité

- Gestion de la synchronisation des données.

3.2. ModalityComponent

ModalityComponent, ou bien Vue. Son travail lié aux tâches pour l'entrée, la sortie des données...

D'après la spécification, ses responsabilités sont:

- Gestion des commandes d'entrée et la reconnaissance d'entrée.
- Gestion de l'intégration de l'entrée s'il y a une combinaison des entrées (par exemple, mixer le son et la vidéo).
- Gestion de la génération du contenu de sortie.

3.3. DataComponent

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

Pourtant, chaque *ModalityComponent* pourrait avoir son propre *DataController* en cas de stocker des données privées.

4. Communication entre les composants

Pour la communication, l'architecture MMI propose son propre protocole [voir Annexes, section 3]. Le protocole est asynchrone, bidirectionnel. Il se construit à partir des événements nommés *Life-Cycle Events* [Figure 4]. Son but est de définir une façon pour échanger des informations, pour gérer la communication (établir et terminer une communication...).

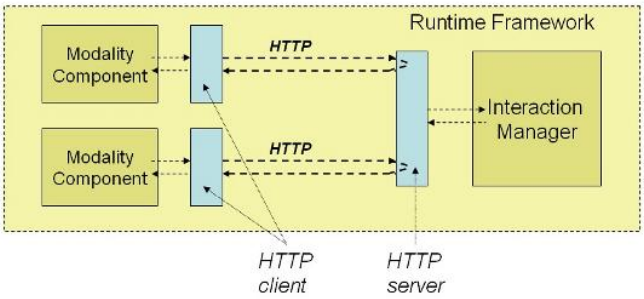


Figure 4: Communication entre les composants
Source: http://www.w3.org/TR/2012/PR-mmi-arch-20120814/

Afin de faciliter la communication, 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 d'état courant du système.

Nous avons également **proposé deux événements** qui seront détaillés dans le chapitre 3, section 3.3.3*: "*

Interaction avec les commandes: La technique l'Adaptative Pull". Dans le cadre de cette section, nous abordons tout d'abord les six événements et les deux notifications MMI. Dans ce contexte, *ModalityComponent* est considéré comme un client, et *InteractionManager* est considéré comme un serveur.

4.1. Événements MMI

Les six types d'événement standard sont:

- **NewContext (NewContextRequest/NewContextResponse):** Utilisé par un *ModalityComponent* pour demander un contexte au sein de *l'InteractionManager*. Un contexte est un cycle d'interaction dans lequel il se trouve plusieurs utilisateurs et

plusieurs *ModalityComponent*. Le contexte signifie également une période d'interaction la plus longue où les *ModalityComponent* doivent rendre ses informations disponibles.

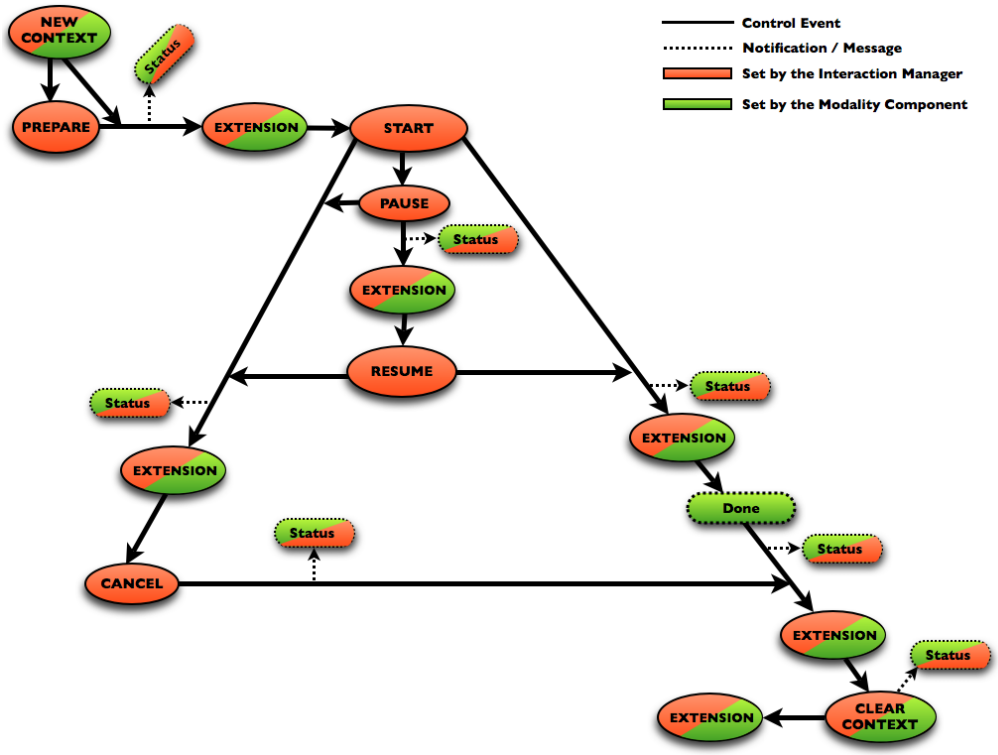


Figure 5: Événements MMI

Source: http://en.wikipedia.org/wiki/Multimodal_Architecture_and_Interfaces

Considérons un exemple: si un *ModalityComponent* voulait participer à une session d'interaction. Il devrait demander le contexte via 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>
```

Si tout va bien, le serveur lui répondra:

```
<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'InteractionManager* pour arrêter un *ModalityComponent*. Le composant concerné devrait libérer les ressources et s'arrêter.
- **Prepare (PrepareRequest/PrepareResponse)** Envoyé par l'*InteractionManager* vers les *ModalityComponent*, pour leur dire de préparer l'environnement avant de commencer son travail.

  Par exemple, un *PrepareRequest* contenant d'une vidéo est envoyé vers un écran. Si cet écran reçoit ensuite le `START`, il jouera cette vidéo.

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

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

  Si le *ModalityComponent* reçoit un autre `START` tandis qu'il est déjà dans état `START`, en fonction de la situation courante, il peut, soit déclencher une nouvelle tâche, soit signaler une erreur.

- **Cancel (CancelRequest/ CancelResponse)** Envoyé par *l'InteractionManager*, cet événement demande au *ModalityComponent* d'arrêter sa tâche courante.
- **Pause (PauseRequest/ PauseResponse)** Envoyé par *l'InteractionManager*, cet événement demande au *ModalityComponent* de mettre en pause sa tâche courante.
- **Resume (ResumeRequest/ ResumeResponse)** Envoyé par l'*InteractionManager*, il demande au *ModalityComponent* de reprendre la tâche qui vient d'être mise en pause.
- **Status (StatusRequest/ StatusResponse)** Envoyé par l'*InteractionManager* ou le *ModalityComponent* pour annoncer l'état courant du système.

4.2. Notifications MMI

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

- **Extension (ExtensionNotification)** Utilisée pour envoyer des données supplémentaires
- **Done (DoneNotification)** Envoyée par *ModalityComponent* vers *l'InteractionManager* pour notifier de la fin de l'expéditeur.

Après avoir étudié l'architecture MMI, nous passons à l'implémentation. Cette partie vous détaillera également mon travail pendant le stage.

CHAPITRE 3 : 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 en détail, nous aborderons tout d'abord l'environnement et la méthodologie de travail, ceux qui est très important dans un projet informatique. Ensuite, nous vous détaillerons la conception de notre implémentation, en suivant sa mise en œuvre. Puis, nous créerons quelques composants de modalité au delà de la plateforme venant d'être construites. Enfin, nous vous donnerons quelques critiques sur notre travail.

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

1. Environnement et méthodologie de travail
2. Analyse et Conception de l'architecture
3. Mise en œuvre
4. Services MMI
5. Critiques

1. Environnement et méthodologie de travail

1.1. Environnement de travail

Au niveau 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...

1.2. Méthodologie de travail

Une partie très importante 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 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 biEnfinie, nous commençons à l'implémenter. À la fin de cette étape, nous avons un composant exécutable.
- **Troisième étape:** L'application sera ensuite testée pour assurer sa fonctionnalité, son exactitude. Les résultats de teste et les feedbacks seront utilisés comme entrée de la première étape dans la prochaine boucle.

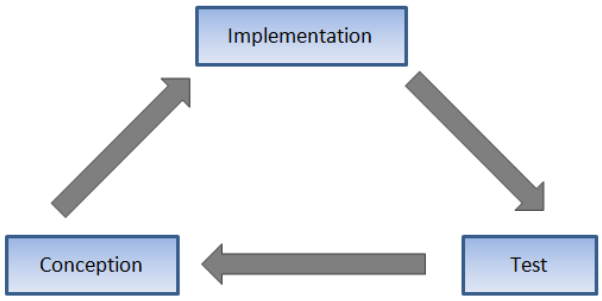


Figure 6: Méthode d'itération

À chaque phase d'itération, nous pourrions rapidement trouver des fautes/erreurs. 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 code. Cela facilite la modification, le stockage des codes source. Il est également utile pour le retour en arrière à partir d'une erreur lors qu'elle se produit.

Figure 7: Gestion de version utilisant SVN

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 **le suivi des tâches et la gestion des erreurs**:

Quant à nous, Mantis Bug Tracker[1] est considéré. Quelques ses caractéristiques sont:

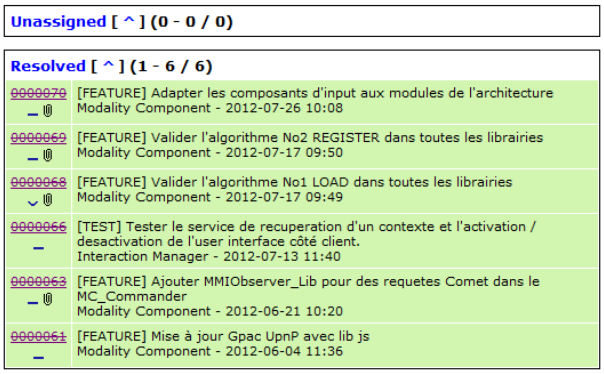


Figure 8: Suivi des tâches et gestion des erreurs

- Assigner les tâches à des personnes concernées.
- Signaler les bugs se produit lors du développement.
- Mettre à jour sur l'avancement de sa résolution, jusqu'à sa clôture....
- Prendre des notes, Attacher des fichiers pour mieux décrire la tâche.

Avec ces méthodologie de travail, nous continuerons Ensuite à la phase d'analyse et conception.

2. Analyse et Conception

2.1. Système dans son ensemble

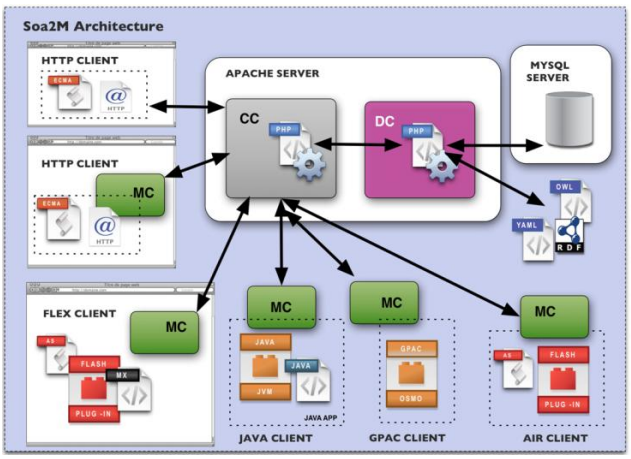


Figure 9: Structure du système
Source: document interne

[1] http://www.mantisbt.org/

Nous proposons une architecture multi-niveaux [Figure 9]: L'accès aux données (*DataComponent*) et le contrôleur (*ControllerComponent*) se trouve dans un serveur web (Apache). En revanche, les composants de modalité sont sous la forme des clients HTTP ou des applications autonomes. La technologie pour le côté serveur est Php et MySQL tandis que HTML, Css, Svg, JavaScript, ActionScript, Flex, Flash, Air, Gpac et Java sont utilisés pour le côté client.

Comme mon travail porte sur le côté client, nous analyserons maintenant la structure du côté client.

2.2. Structure d'un composant de modalité

Au cœur du côté client, c'est le composant de modalité (*ModalityComponent)*. Nous avons proposé deux types de composant de modalité: simple et complexe. Le deuxième pourrait contenir d'autres composants [voyez le chapitre 2, section 2 **Aperçu de l'architecture**]

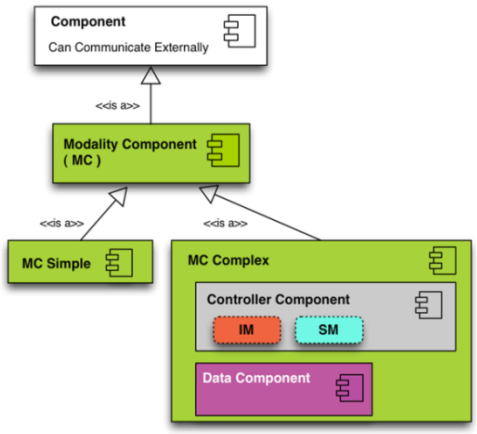


Figure 10: Composant de Modalité
Source: document interne

Un composant de modalité communique avec le monde via les messages MMI, en utilisant un protocole de communication. Les deux seront abordés dans les sections suivantes.

2.3. Structure d'un message MMI

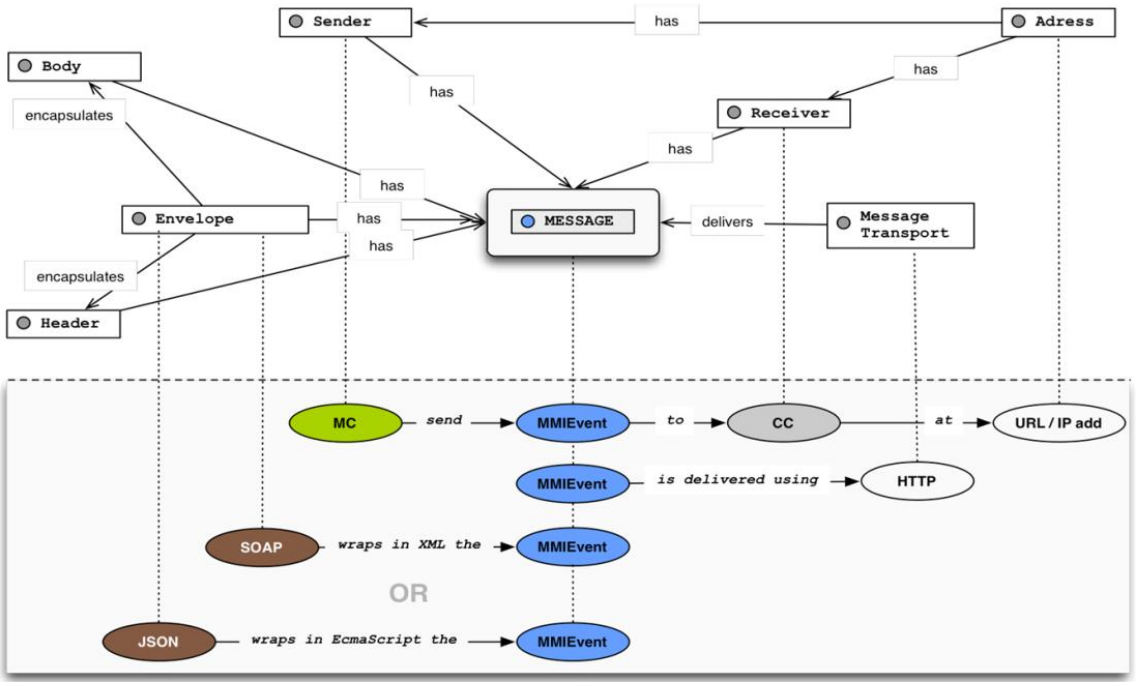


Figure 11: Message MMI
Source: document interne

Quand le composant de modalité est au cœur du côté client, Les messages MMI sont considérés comme le cœur de la communication. [Figure 11] décrit sa structure ainsi que lé mécanisme de transmission. Un message MMI, qui pourrait avoir des *MMIEvent* [voir chapitre 2 - section 4], est un engagement entre l'expéditeur et le récepteur lors de ses communications. Pour pouvoir traverser le réseau, le message MMI est enveloppé pas un message SOAP ou JSON. Le protocole de communication utilisé est présenté dans la section suivante.

2.1. Protocole de communication

La communication client - serveur conforme au protocole de communication décrit dans [Figure 12]: S'il un composant de modalité voulait faire partie de la session, il devrait demander le contexte. Le serveur lui répondra (indiqué par la flèche doté dans la figure) en le lui donnant. Quelque part, le serveur pourrait lui prévenir de préparer son environnement. Quand ce client finit sa préparation, il devrait envoyer une confirmation via le message *PrepareResponse*. Le client tourne désormais en arrière pour attendre la commande, par exemple, le *StartRequest.*

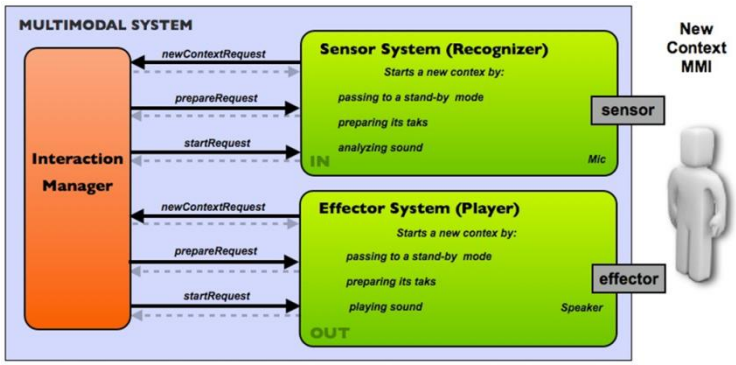


Figure 12: Protocole de communication
Source: document interne

Maintenant, après avoir conçu le système, nous passons à l'étape suivant: Mise en œuvre de l'architecture.

3. Mise en œuvre de l'architecture MMI

La mise en œuvre est réalisée en utilisant trois langages de programmation: *JavaScript*, *ActionScript* et *Java*. Le but est de valider l'architecture MMI au sein du W3C, qui est encore en état de divulgation et de vérification, ainsi que de donner une implémentation utilisable.

Pour visualiser la mise en œuvre, considérons un cas d'études: un professeur entre dans son bureau. Son portable a une petite application s'appelant "*Calendrier*". Supposons que cette application est déjà activée. Lors que le portable se connecte au réseau local, l'application s'enregistre lui-même au réseau en même temps, il découvre des services locaux au tour de lui, parmi eux se trouve un grand écran dans son bureau. Quand tout sera fini, le grand écran (qui est aussi une part de l'application Calendrier) affichera le message "Bonjour...".

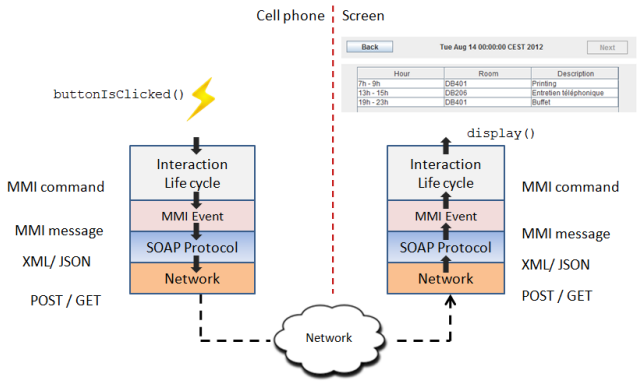


Figure 13: Cas d'étude

De son côté, il souhaiterait avoir le plan d'enseignement d'aujourd'hui. Ainsi, il appuie sur le bouton "*Voir Calendrier*" sur l'écran tactile de son portable. Tout de suite le grand écran affiche son plan d'enseignement.

Mais comment ça marche ? Cette section et la section suivante vous expliquera tout:

- La transmission des commandes via le réseau sera présentée dans la **section 3.1: Communication via les réseaux: XHR Lib**
- Les messages MMI, les événements seront présentée dans la **section 3.2: Événements MMI: MMI Lib**
- L'enregistrement de l'application avec le serveur se faite via le processus de *handshaking* et sera présenté dans la **section 3.3: Cycle d'interaction: Architecture Lib**

Maintenant, On va s'attaquer la première partie: **Communication via les réseaux:** XHR Lib

3.1. Communication via les réseaux: XHR Lib

Description:

La couche de réseau est chargée d'envoyer des messages HTML (POST ou GET) via les réseaux. Elle est également chargée d'encapsuler les MMI messages sous le format SOAP ou JSON

Implémentation:

Nous avons implémenté la couche de réseau en basant sur ses trois responsabilités principales:

- Création du message
- Envoi du message
- Traitement de la réponse

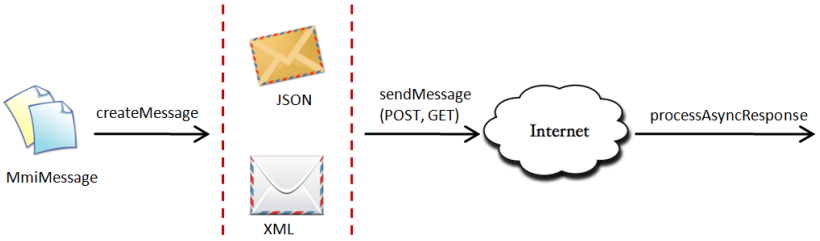


Figure 14: Fonctionnalités de XHR Lib

L'implémentation détaillée varie entre les langages. Mais au niveau de l'algorithme, les idées principales restent les mêmes.

Création du message:

La bibliothèque nous donne deux possibilités de créer le message: soit SOAP avec XML, soit SOAP avec JSON. En fonction de la valeur du paramètre d'entrée *type*, le message approprié sera crée.

```
1    // Create message
2    if (type == "SOAP")
3       message = createSoapMessage(mmiMsg);
4    else if (type == "JSON")
5       message = createJsonMessage(mmiMsg);
6    else
7       throw "Unknown type exception"
8    end if
```

Envoi du message:

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

```
1    // Send message
2    if (method == "POST")
3       sendPostMessage(message);
4    else if (method == "GET")
5       sendGetMessage(message);
6    else
7       throw "Unknown method exception";
8    end if
```

En gros, si la taille des messages ou la sécurité sont considérés, nous devrions utiliser la méthode POST. Si non, GET est aussi un bon choix.

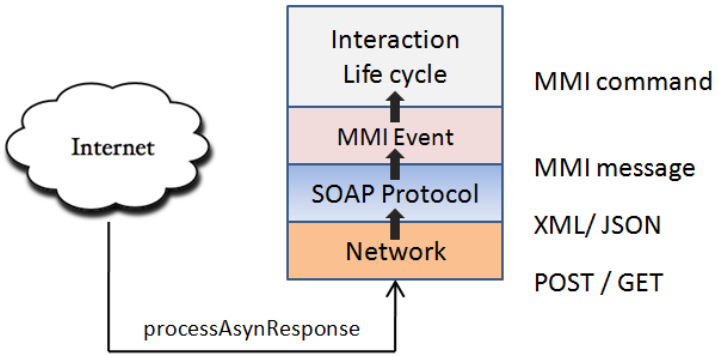


Figure 15: Traitement des messages

Traitement de la réponse:

Lors que le serveur nous répond la fonction *onResponseReceived* est appelée. Elle analyse la requête HTTP reçue pour récupérer le contenu dedans, dison le message SOAP ou JSON. Ce message est ensuite envoyé à la couche plus haute pour le traitement.

```
1    // Process Asynchronous response
2    function onResponseReceived(event)
3        - Check message's validation
4        - Delegate it to the upper layer
5    end function
```

Grâce au mécanisme de transmission, les services MMI pourraient communiquer l'un avec l'autre. Pour la suite, nous étudierons dans la section suivant les événements utilisés pour la communication.

3.2. Événements MMI: MMI Lib

Description:

La librairie MMI est chargée de:

- Créer des messages (événements) MMI à partir des attributs.
- Analyser les messages XML / JSON pour la récupération de message MMI.

Ces événements MMI sont proposés par la spécification. Ils sont considérés comme une interface entre l'application et le réseau. Grâce à la communication utilisant ces événements, l'architecture est langage-indépendante, plateforme-indépendante: dès qu'ils respectent la spécification, les composants pourraient communiquer entre eux sans le problème de hétérogénéité [Figure 16].

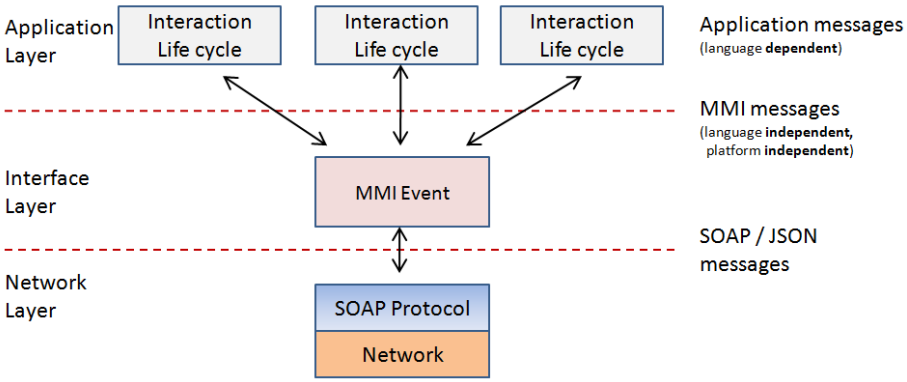
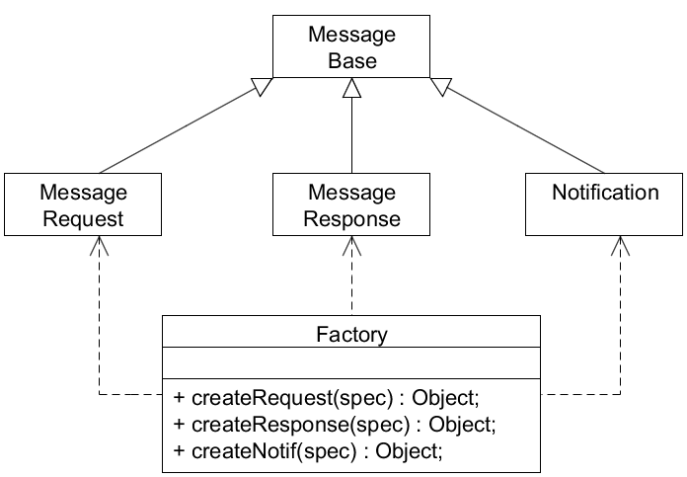


Figure 16: Rôles des événements

Implémentation:

Suite à la spécification, il y a deux types d'événement: celui qui existe en paire (une requête et une réponse). Et celui qui n'existe pas en paire (les notifications) [voyez la chapitre 2, section 4]. Sachant que ces événements (message et notification) partagent plusieurs attributs en communs, nous proposons une solution se basant sur le principe de l'héritage [Figure 17]



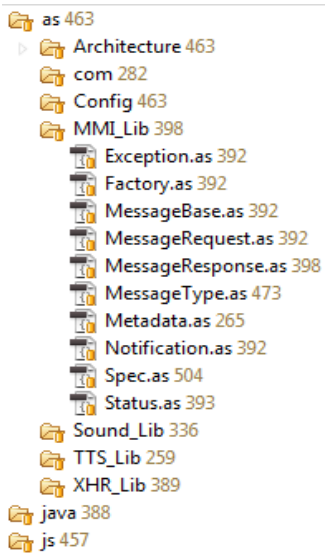


Figure 17: MMI Lib et son implémentation en action script

Il existe une classe de base s'appellant *MessageBase*. À partir d'elle, nous avons construit trois sous classes: *MessageRequest*, *MessageResponse* et *Notification*.

Nous avons également utilisé le patron de conception *Factory* pour la création des messages. Les intérêts est de cacher l'implémentation détaillée et de séparer la dépendance entre les composants. En examinant le variable *spec*, nous créerons des types de message différents (voir l'algorithme ci-dessous) comme *NewContext, Start, Pause*...

Il se trouve également dans la librairie quelques classes utilitaires:

- *Exception.as*[1] pour la gestion des exceptions qui sont produites lors de la création des messages
- *Status.as* pour la surveillance des états d'un message.

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

Quand nous avons le type (ligne 2), nous le comparons aux prédéfinis types pour savoir quel type de message nous devrions créer (lignes 3-8). Si le type est inconnu, nous lancerons une exception (ligne 10).

```
1  function createRequest(spec)
2      var msgType = spec.getMsgType();
3      if (msgType == NEW_CONTEXT_REQUEST;
4          // Create New Ctx Request
5          return NewContextRequest;
6      end if (msgType == START);
7          // Create Start Request
8          // ...........
9      else
10         throw "Unknown type exception"
11     end if
12 end function
```

3.3. Cycle d'interaction: Architecture Lib

Description:

Le cycle d'interaction est chargée de:

- Enregistrer l'application avec le serveur.
- Analyser les messages reçus pour donner des commandes vers la couche application.

Implémentation:

L'implémentation de la librairie se trouve en package *architecture* dans laquelle la classe *Controller* est la classe principale. Presque toutes 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 d'aide.

L'implémentation se découpe selon deux grandes fonctionnalités:

- Enregistrement d'un service.

[1] Ils sont pareils pour l'implémentation en java et en javascript.

- Interaction avec des commandes.

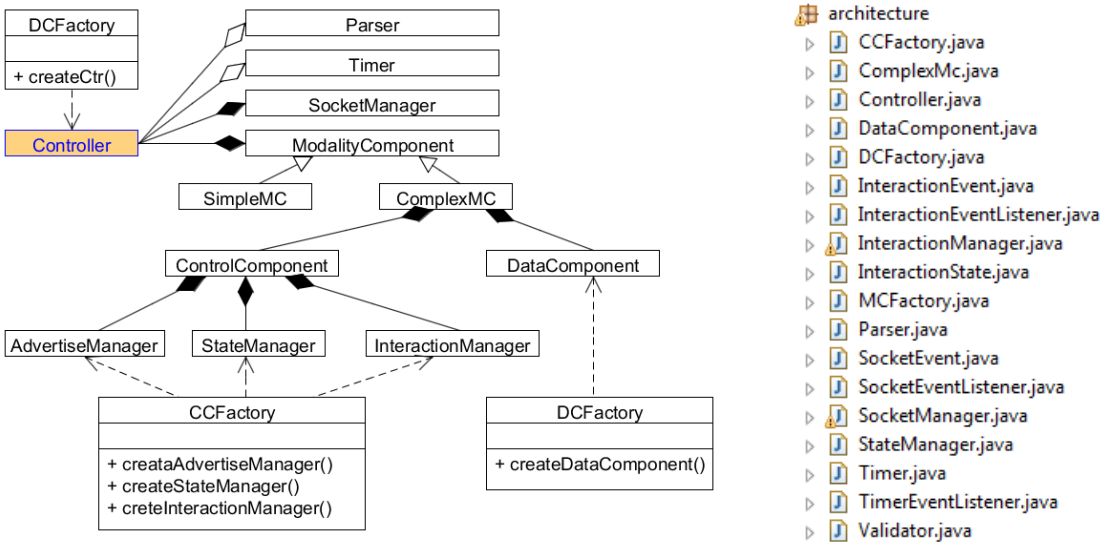


Figure 18: Le package architecture

3.3.1. Enregistrement d'un service

Nous examinerons tout d'abord les protocoles pour l'enregistrement d'un service avec le serveur En fait, le processus se comporte des étapes suivantes [Figure 19]:

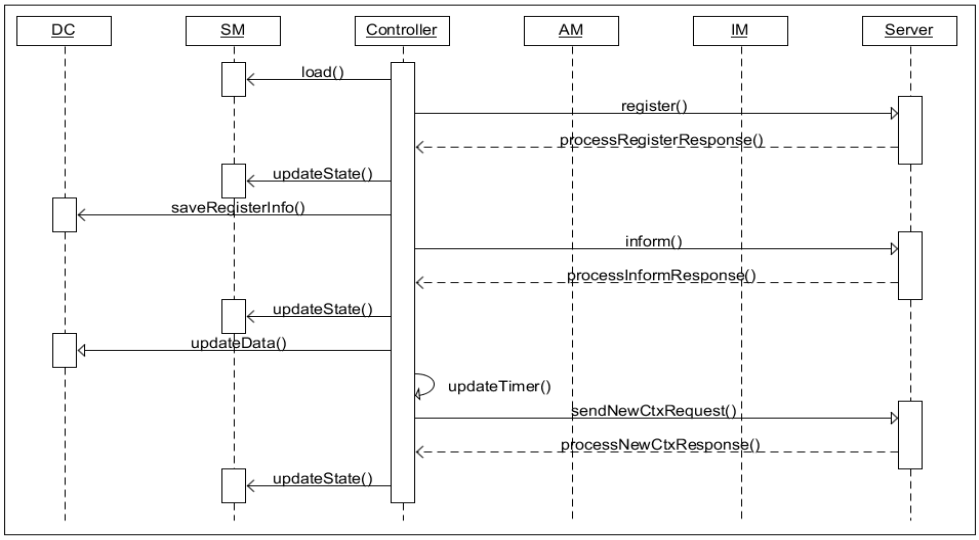


Figure 19: Enregistrement de l'application avec le serveur

`DC:DataComponent` `SM:StateManager` `AM:AdvertiseManager` `IM:InteractionManager`

1. **Chargement des configurations**: Le *Controller* demande au *StateManager* de charger les configurations via la fonction `StageManager.load()` . Ainsi, les informations comme l'adresse du serveur, l'adresse des services web, les variables seront chargés. Elles serviront ensuite à instancier le *Parser*, *Timer*, *SocketManager*...
2. **Enregistrement**: Après le chargement, le client appelle `Controller.register()` pour s'enregistrer avec la serveur. Le serveur répondra via la fonction asynchrone `processInformResponse()` contenant le variable `requestId` dedans, nous examinons ce variable:
   - S'il est inconnu, nous re-faisons l'étape 2.
   - Si non, nous utilisons les informations dans cette réponse pour mettre à jour l'état du système: `StateManager.updateState()` . En même temps, nous les sauvegardons: `StateMachine.updateRegisterInfo()` . Après, nous passons à l'étape 3

3. **Notification la mise à jour**: quand la mise à jour finit, nous l'informons au serveur via grâce au `inform()`, le serveur confirme sa réception et répondra avec des informations supplémentaires, qui peuvent être utilisées pour la synchronisation entre les deux côtés.
4. **Demande du *NewContextRequest***: Après la synchronisation, le client envoie sa requête vers le serveur pour demander le nouveau contexte.
    - S'il y a un `contextId` dans la réponse, le processus d'enregistrement se termine avec le succès.
    - Par contre, nous re-faisons l'étape 4.

Notons qu'ici, il existe deux boucles:

- La première boucle: le client demande son identifier.
- La deuxième boucle: le client demande la session avec laquelle il travaillera.

Ces informations sont stockées dans le fichier de configuration pour que l'application ne doive pas se réenregistrer.

Après l'enregistrement, le client passe au mode d'interaction où il traite des commandes produites par les services. Elle sera abordée dans la partie suivante.

3.3.2. Interaction avec les commandes: Le principe

Après l'enregistrement, le composant de modalité pourrait interagir avec les commandes en provenant de serveur.

Nous distinguons deux types de composant de modalité:

- Composant de modalité contrôlant (dit **entrée**) qui produit des commandes, par exemple, un clavier produit des événements en se basant sur des boutons tapés.
- Composant de modalité contrôlé (dit **sortie**) qui reçoit des commandes, par exemple, un écran reçoit des messages et les affiche.

Pour les Service de type de **sortie**, il n'y a qu'un type de commande envoyé pas le serveur. Néanmoins, **pour ceux de type d'entrée, il nous faut distinguer entre deux types de commande** totalement différentes [Figure 20]. (La reconnaissance des gestes ainsi que l'enregistrement du son seront présentés dans la section suivante) :

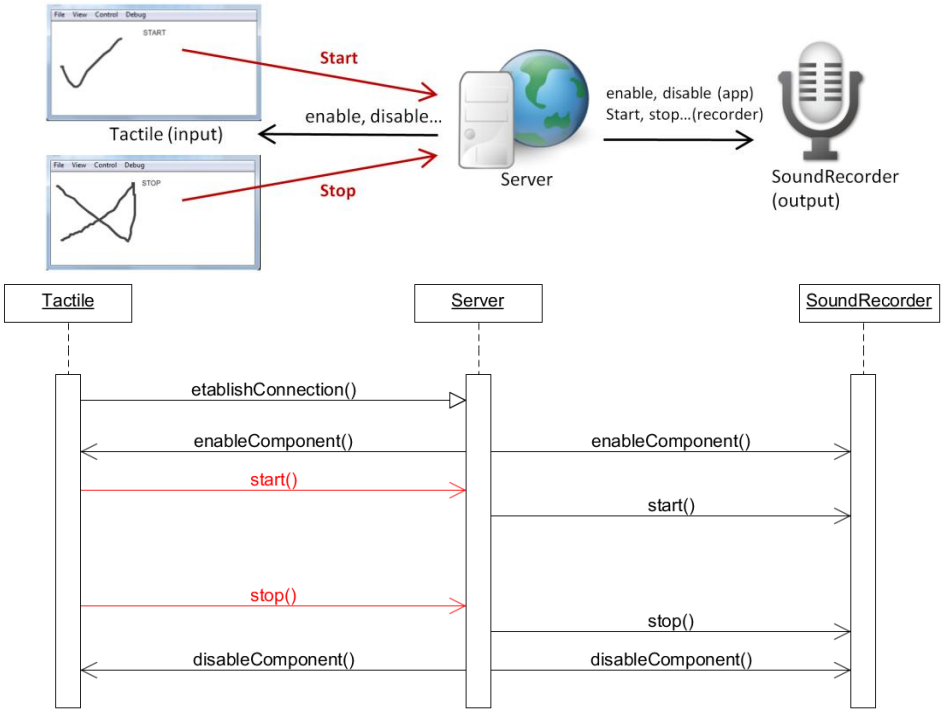


Figure 20: Deux types de commandes

- Commandes produites par le service elle-même pour contrôler d'autre application (en couleur rouge, dit **commande du composant modalité**),
  Les exemples sont des commandes pour starter, stopper le processus d'enregistrement.
- Commandes produites par le serveur pour déclencher, terminer l'application (en couleur noir, dit **commande du système**).
  Les exemples sont des commandes pour désactiver le composant de modalité chargé d'enregistrer du son.

Quand nous avons bien distingué entre les deux types de commandes, nous considérons ensuite comment elles sont traitées. En fait, l'interaction avec les commandes se passe par deux étapes:

- Réception d'une commande (en provenance du composant de modalité au delà ou de la couche la plus bas)
- Diffusion d'une commande (en provenance du composant de modalité au delà ou de la couche la plus bas)

Nous les détaillerons tout de suite:

a, Réception des commandes:

S'il s'agit d'une commande produite par le composant de modalité construit au delà, il suffit d'appeler directement sa méthode, parce qu'ils sont tous dans une même application.

S'il s'agit d'une commande en provenant du serveur, c'est plus compliqué. Nous devons toujours garder la connexion avec le serveur, si non nous ne savons pas quand il veut nous envoyer une commande.

Pour adresser ce problème, il y a deux solutions.

- *Push* le serveur qui initialise la connexion:
  - En *JavaScript*, nous avons utilisé la technique *Comet.* Il suffit d'intégrer la balise `<iframe>` dans notre composant. La communication se fait via cette balise. Pour la technique détaillé, voyez la spécification de *Comet*.
  - En *Java* et *ActionScript*, le *Socket* est ainsi utilisé. Nous devons établir une connexion et la garder tout le temps. L'application donc tournera en arrière en attendant le message.
- *Pull* le client périodiquement envoie des messages de sondage vers le serveur. L'intérêt de cette méthode est:
  - Nous pourrions modifier l'intervalle entre les sondages. Ainsi, il client, qui sait quand il est prêt, est plus autonome, plus actif.
  - Nous ne devons pas garder la connexion tout le temps.

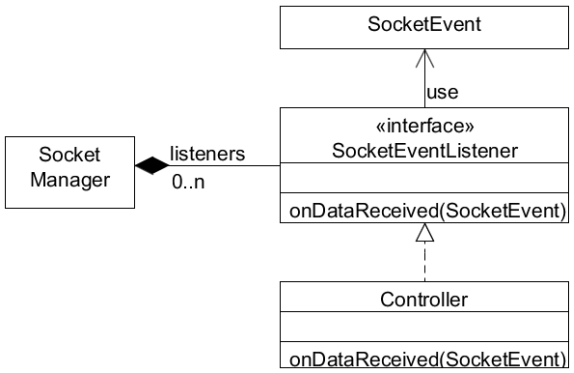
Parmi eux, *Pull* **n'est pas prévu par la spécification**, mais c'est nous qui l'avons proposé lors de notre implémentation. Son mécanisme sera détaillé dans la section suivante.

b, Diffusion d'une commande:

Après avoir reçu la commande, nous les enverrons vers la cible.

Si la cible est un composant de modalité de la couche plus haute, il suffit d'appeler une fonction de traitement fournie par ce composant là, parce qu'il est dans la même application.

Si la cible est une couche plus base (le cas où le composant de modalité en haut est envie d'envoyer sa commande vers d'autre composant de modalité dans le réseau) nous utilisons le mécanisme de notification. Considérons l'implémentation en *Java* [Figure 21].




```
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     // Process
13  end function
```

Figure 21: Notification des événements.

Nous ajoutons au *SocketManager* une implémentation de *SocketEventListener*, disons *Controller* (ligne 2). Lors que le *SocketManager* reçoit un événement, il le diffuse via la fonction *fireEvent()*. Comme notre *Controller* s'est inscrit à cet événement, il se réveille fait son travail (lignes 10-12).

À travers ce chapitre, nous vous avons présenté les algorithmes, les librairies principales utilisés dans l'implémentation de l'architecture MMI. Pour la continuation, nous vous présenterons quelques services que nous avons construits en se basant sur cette implémentation.

3.3.3. Interaction avec les commandes: La technique l'Adaptative Pull

Revenez-vous à la technique Pull [Figure 22], elle permet au client de subjectivement demander de nouvelles données en provenance du serveur. Avec cette technique, le changement est initié par le client: Il demande périodiquement au serveur ce qu'il faut mettre à jour. La connexion est tout de suite fermée après chaque transfert. Le client dira quand il faut ouvrir une nouvelle connexion.

Ce qui est important, c'est la fréquence de sondage (*pulling frequency*). Il faut faire attention ici: une haute fréquence peut causer des redondants en augmentant le trafic du réseau tandis que une basse fréquence pourrait causer la manque des informations du serveur.

Ainsi, dans le cas idéal, l'intervalle de *Pull* doit être égal au taux de changement de l'état au sein du serveur. Comme ce fréquence est fluctuée, il vaut mieux d'avoir un mécanisme adaptative dans lequel nous pouvons facilement modifier la valeur de la fréquence de mise à jour.

Du coup, nous proposons une structure de données dédiée s'appelant *timeout* contenant trois éléments:

```
timeout = {sleep, lifeTime, interval}
```

Avec:

- `sleep` : le délai d'attente avant l'envoi de la première demande.
- `lifeTime` : la durée de vie de la communication.
- `interval` : l'intervalle entre les deux requêtes.

Chaque client (disons *ModalityComponent*) dort pendant un certain temps, puis il se réveille et vérifie pour voir s'il y a des changements prévus au côté serveur (disons *ControllerComponent*). Entre temps, il met un réveil pour se réveiller lui-même plus tard. La valeur `sleep` est calculée au côté serveur en se basant sur l'état courant du système.

Le second élément est la durée de vie de communication. Un client quitte le système multimodal lorsque sa durée de vie est dépassée. S'il voulait le rejoindre, il devrait redémarrer son mécanisme de s'inscrire [voir 3.3.1] pour obtenir son nouveau identifiant et son nouveau `timeout`.

Le troisième élément est l'intervalle de communication, cet élément indique le client de la fréquence d'envoie des messages autorisée par le serveur. Cette valeur est échangée à chaque demande, ce qui signifie qu'elle peut être mise à jour à tout moment.

Concernant cette structure de données, nous proposons également deux nouveaux événements: *CheckUpdate* et *UIUpdate*

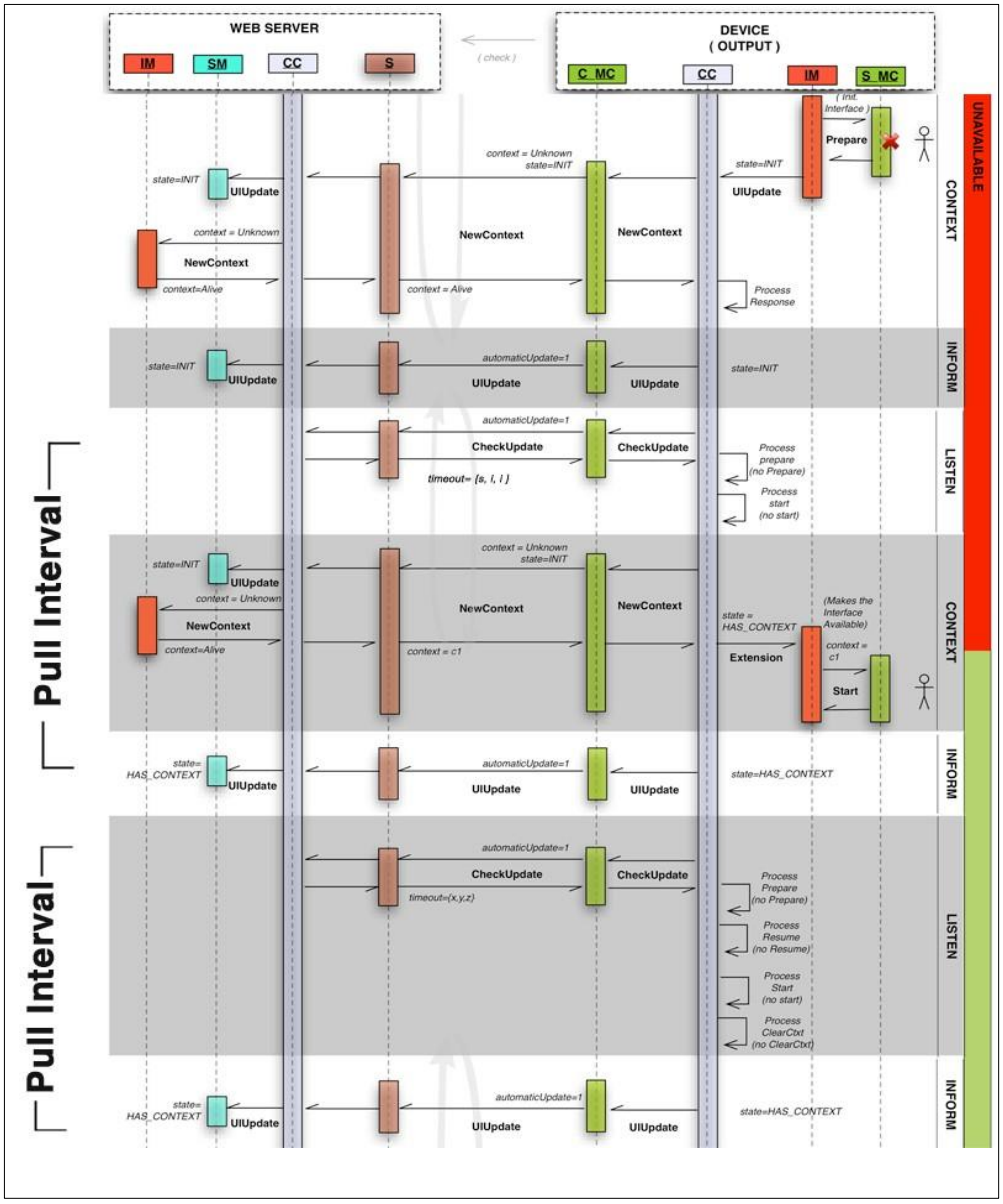


Figure 22: La technique "Pull"
Source: document interne

CheckUpdate est chargé de:

- Vérifier s'il y a des changements au côté serveur
- Mettre à jour le `timeout` si cela est nécessaire.
- Déclencher des notifications automatiques concernant l'état du client (si le champ `automaticUpdate` dans la réponse est défini).

UIUpdate est utilisé pour périodiquement informer l'état du client (*ModalityComponent)* au serveur.

Pour un mot, si la fréquence de sondage indiquant dans `interval` est inférieure à la fréquence de changement au serveur, nous risquons de manquer quelques données. Dans le cas contraire, la performance du réseau sera influencée. Grace à notre proposition, la fréquence de sondage serait égale à la fréquence de changement au serveur. De cette façon, les données, la cohérence ainsi que la performance du réseau seraient assurées.

4. Composant de modalité MMI

Dans cette partie, l'implémentation des services suivants seront présentés:

- Service pour l'enregistrement du son.
- Service pour la transformation de la parole en texte.
- Service pour la transformation du texte en parole.
- Service pour la reconnaissance des gestes.

4.1. Enregistrement du son

Description:

Cette tâche est pour but de créer un composant capable d'enregistrer du son [Figure 23]. L'utilisateur va utiliser son propre microphone pour recorder ce qu'il dit.

Implémentation:

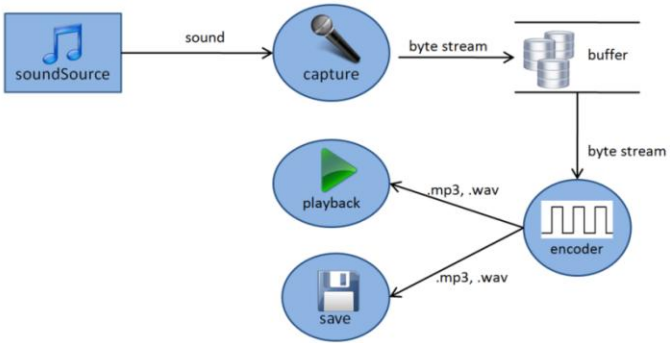


Figure 23: Enregistrement du son

L'enregistrement du son est implémenté en *AdobeAir*, utilisant *ActionScript*. Il comprend des étapes suivantes:

- Capture du son.
- Encodage du son.

Capture du son: nous avons utilisé un microphone configuré par les codes source d'*ActionScript*. Le son capturé est stocké ensuite dans un tampon. L'algorithme est comme suivant:

Pour être en courant, nous attachons la fonction **onRecording** au microphone *(*ligne 1). Elle sera réveillée chaque fois qu'il y a un flux d'octets en provenance du microphone. Après, elle écrit les données en tampon (ligne 4-5).

```
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
```

Encodage du son: Après l'enregistrement, le service pourrait à la fois faire la lecture ou la sauvegarde. Mais il faut tout d'abord encoder les flux d'octets vers le format qu'on veut. Ce composant supporte quelques formats principaux: *.mp3, .wav*.

Adobe a fournit une librairie non-officielle pour l'encodage vers l'extension *.wav*. Il suffit de l'utiliser pour récupérer un fichier *.wav*

Quant au format *.mp3*, c'est un peu plus compliqué parce qu'il n'est pas supporté pas *ActionScript.* Heureusement, il existe un encodeur de MP3 construit en C nommé *Shine MP3 Encoder*[1]. En utilisant la technique l'Alchemy[2] , nous pouvons l'utiliser en *ActionScript* pour créer des fichiers mp3

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

Description:

Cette application prend du son en entrée et sort un texte indiquant ce que dit le son d'entrée.

Implémentation:

La transformation de la parole en texte est implémentée en *AdobeAir*, utilisant *ActionScript*.

[1] http://code.google.com/p/flash-kikko/
[2] http://labs.adobe.com/technologies/alchemy/

Concernant l'implémentation, nous avons deux approches:

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

Figure 24: Transformation de la parole en texte

Sachant que la transformation demande une base en signaux numériques et que Google a crée un très bien API, nous avons choisi la deuxième approche.

Le travail se comporte de deux étapes: la **conversion** du son vers l'extension .flac et la **transformation** du texte en parole.

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

La librairie utilisée pour la conversion est *FlacAlchemyEncoder*[1]. Lors de l'utilisation, il y a un souci: la structure de données de l'*ActionScript* ne support que les flux d'octets de 32bit par défaut. Néanmoins, la librairie supporte seulement celui de 16bit. À l'époque, ce n'était pas noté dans la documentation et nous avons été bloquée par cette raison. Grâce à des correspondances avec l'auteur, nous avons fixé ce problème en écrivant une procédure en *ActionScript*

Transformation: La transformation se divise encore en trois sous parties:

- **Préparation**: Nous devons préparer quelques configurations. Pour pouvoir récupérer la réponse, nous ajoutons des fonctions callback en indiquant la langue concernée. Ce sont des fonctions pour signaler des événements comme le commencement, la fin, ainsi que les erreurs lors du traitement.

  Nous choisissons également une méthode pour la requête (POST/GET). Dans notre cas, nous utilisons POST parce que la taille du flux des octets est considérable.

- **L'envoi des données:** Une fois que la requête est prête, nous l'enverra vers le service de Google. Notons que la communication est asynchrone, le système pourrait ainsi effectuer d'autre tâche en attendant la réponse.
- **Récupération des données:** Nous récupérons la réponse via une fonction callback. À ce moment là, nous pouvons soit l'afficher sur l'interface GUI, soit l'envoyer vers d'autres composant de modalité.

```
1   // Conversion
2   var data = toFlac(mp3Data);
3   // Transformation
4   var configInfo = getConfigInfo();
5   save(resultTxt);
```

4.3. Transformation du texte en parole (Text-to-Speech)

Description:

Ce composant de modalité est opposé à celle décrite dans 4.2: Il prend une chaîne de caractères et produit un fichier *.mp3*.

Comme 4.2, nous avons utilisé un API de Google: Google Translate. Ce qui est intéressant c'est que Google Translate supporte près que toutes les langues. Autrement dire, vous pouvez avoir autant nombre de langues comme Google Translate, mais directement depuis notre service.

Implémentation:

La transformation du texte en parole est implémentée en *AdobeAir*, utilisant *ActionScript*.

L'implémentation comprend de trois étapes:

[1] https://github.com/lukeweber/flac-as3-alchemy-wrapper

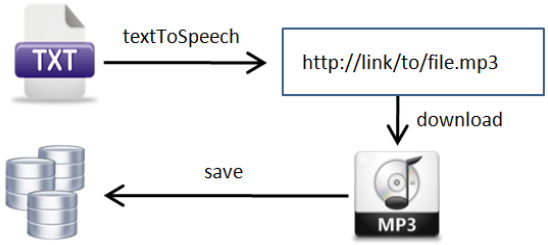


Figure 25: Transformation du texte en parole.

- **Récupère du texte**: Nous devons premièrement récupérer le texte utilisé pour la transformation. Le texte peut provient du serveur en tant qu'un champ dans la requête PREPARE_REQUEST [voir le chapitre 2, section 4.2]. Il peut également venir de l'interface GUI de ce composant de modalité.
- **Envoi du texte ver *Google Translate:*** Quand nous avons le texte, nous l'enverrons vers Google. Une requête demandant la version *.mp3* du mot "*bonjour*" aura le lien:

```
http://translate.google.com/translate_tts?tl=fr&q=bonjour
```

- **Récupère de la réponse:** Si la transformation s'est bien passée, nous recevrons un lien contenant un fichier *.mp3*. En le téléchargeant fichier, nous pouvons soit l'envoyer vers d'autre composant de modalité soit le stocker pour l'utilisation après

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

4.4. Reconnaissance des gestes

Description:

Le but de ce composant est de simuler la fonctionnalité d'un écran tactile. En se basant sur les gestes de l'utilisateur, une commande sera produite. Elle sera ensuite envoyée vers la couche en basse pour la création d'un message MMI équivalant.

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

Implémentation:

L'implémentation passe par trois étapes suivantes [Figure 26]:

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

Considérons un exemple dans lequel un utilisateur utilise son doigt pour désigner le *caractère V* sur l'écran tactile |Figure 26]. Nous passons par trois étapes suivantes:

Récupération des données de gestes: À partir des gestes, nous construisons des vecteurs de direction via a une boucle. À chaque étape:

- Nous récupérons la coordonnée du doigt (disons `P2`). Avec la dernière coordonnée (disons `P1`) récupérée à toute à l'heure, nous créons un vecteur de direction entre eux (lignes 3-7)
- À la fin de cette étape, nous avons une liste des vecteurs de directions (`v1....v6` dans notre exemple) 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
```

Adaptation des données récupérées: Ayant la liste des vecteurs dans la main, nous effectuons une adaptation pour chaque vecteur. Ainsi, les trois vecteur `v1, v2, v3` sont considérés comme les

trois vecteur de *type 3* [Figure 26]. Pareil, trois vecteur `v4, v5, v6` sont à *type 1*. La sortie de cette étape est donc la chaîne `333111`. Avec cette chaîne, nous passons à l'étape suivant.

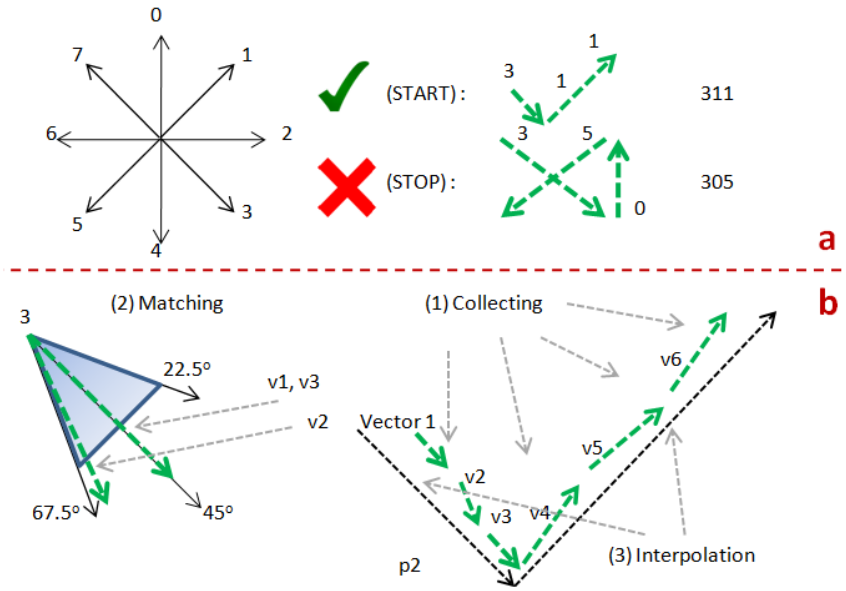


Figure 26: Reconnaissance des gestes

Interpolation:

Considérons l'algorithme au dessus en supposant que `333111` est la valeur du variable `gesture`.

```
1   function getCommand(gesture) // 333111
2       // init
3       bestMatchedId = -1;
4       min = Interger.Max;
5       sampleList = {-1, 311, 305} // UNKNOWN, START, STOP

6       // find command
7       for each sample in sampleList
8           distance = getDistance(gesture, sample)
9           if (distance < min) then
10              min = distance
11              bestMatchedId = sample
12          end if
13      end for
14      command = getCommandFromId(bestMatchedId)// START
15      return command
16  end function
```

- `gesture` l'entrée que l'utilisateur fournit. Dans notre cas, c'est "`333111`"
- `sampleList` Une liste contenant des vecteurs d'exemplaire. Dans notre cas, c'est une liste contenant deux éléments: `311 (START), 305 (STOP) et -1 (UNKNOWN)`. Et parmi eux, la variable `nearestValue` est `311`.
- `command` est donc `START`.

Après la phase d'initialisation, nous parcourons la liste `sampleList` (ligne 7). Pour chaque élément, nous calculons la *Distance de Levenshtein*[1] et gardons ceux dont la distance est la plus petite.

À la fin de la boucle, nous avons un échantillon la plus proche dans le variable `bestMatchedId`. (`311` dans notre cas). Nous l'utilisons pour obtenir la commande `START`.

[1] La distance de Levenshtein mesure la similarité entre deux chaînes de caractères. Pour plus d'info, voyez: http://fr.wikipedia.org/wiki/Distance_de_Levenshtein

Maintenant, quand l'implémentation de l'architecture a été présentée, nous aimerons vous donner quelques critiques sur cette implémentation.

5. Critiques

Après avoir implémenté l'architecture MMI utilisant quelques langages, nous aimerons sortir une petite comparaison entre eux. Comme que je ne travail pas avec le côté Serveur qui utilise Php, ce langage n'est pas montré ici.

La notion utilisée est **1, 2, 3 ou 1 est le mieux**.

	ActionScript	Java	JavaScript
Gestion du projet	2	1	3
Gestion des événements	1	3	2
Fléxibilité	2	2	1
Multi threading	3	1	3
Line de code source	2	2	1
Réutiliser du code	3	1	2
Communauté	2	1	1

- **Gestion de projet**: Parmi les trois, *Java* offre la mieux façon de la gestion de projet. Il me semble que *JavaScript* n'aie son IDE dédié pour gérer le projet.
- **Gestion des événements**: *ActionScript* a montré sa puissance, *JavaScript* est acceptable. Pour *Java*, la note 3 ne dire pas qu'elle a du mal à gérer les événements. En fait, elle les a gérés très bien. Néanmoins, *ActionScript* est *JavaScript* sont mieux.
- **Flexibilité**: *JavaScript* est le mieux, comme il a des caractères du langage fonctionnel, c'est facile d'ajouter des méthodes, des variables. Pour l'exécution, il suffit de lancer le browser. Nous n'avons pas besoin de le recompiler.
- **Multi threading**: Java supporte le multi-threading tandis que *JavaScript* et *ActionScript* ne le supporte pas.
- **Ligne de code source**: *JavaScript* est le numéro un, Le nombre de code source de *Javascript* est le plus moins. Pour Java, c'est acceptable et pour *JavaScript*, c'est trop.
- **Distribution du code:** *Java* est très forte dans ce terme. JavaScript est pas mal. Pour *ActionScript*, c'est super difficile comme il est fermé pas Adobe. 1000€ pour une licence, cela tout dit!
- **Communauté:** *JavaScript* et *Java* sont fortement supportés par la communauté. Pour *ActionScript*, même s'il est beaucoup supporté par la communauté. Mais ce n'est que la communauté Adobe. Toujours le problème de la licence...

Pour la conclusion, *Java* et *JavaScript* est le mieux pour ce projet. Quant à *ActionScript*, ce serait mieux, s'il était plus ouvert.

CHAPITRE 4 : CONCLUSION

À travers mon stage au sein de l'équipe Multimédia à Télécom ParisTech, j'ai fait la recherche en domaine de multimédia, j'ai également étudié et implémenté une spécification proposé par W3C. Mon travail porte sur le côté client avec plus de 10 000 lignes de code sources environs.

Mes contributions par rapport avec la version originale de la spécification sont:

- Proposition d'un mécanisme de communication nommé *Adatative Pull.*
- Proposition d'un mécanisme de fonctionnement en se basant sur des états.
- Ajout du patron de conception *Factory*.
- Proposition des services de feedback pour la surveillance de la session multimodale, de l'état du système.
- Proposition d'un mécanisme de délégué pour le contrôleur.

Et au niveau d'implémentation, vous voyez ci-dessous un tableau détaillé récapitule ce que j'ai effectué ainsi que son résultat lors de mon stage:

Description	Fini	Non fini
Recherche		
Études du projet Open Source GPAC	x	
Études des extensions .bifs, .xml	x	
Études de la norme SVG	x	
Études de l'architecture MMI proposé par W3C	x	
Études du langage SCXML		x
Développement		
Le cœur de l'architecture MMI en JavaScript	x	
Le cœur de l'architecture MMI en ActionScript	x	
Le cœur de l'architecture MMI en Java	x	
Transformation de la parole en texte (STT)	x	
Transformation du texte en parole (TTS)	x	
Reconnaissance des gestes	x	
Autre applications: Découverte des services en GPAC, Calendrier en Java...	x	
Le moteur de décision		x

Concernant les difficultés, ma principale difficulté est due au problème d'hétérogénéité. Le fait de travailler dans un projet multimédia hétérogène me demander de travailler dans des environnements différents: Linux, Window, et aussi, le smartphone. Cela me prend pas mal de temps pour la configuration, pour m'adapter...

J'ai également des difficultés de la licence avec les produits Adobe. En fait, pour pouvoir sortir la version actionscript en Air, Flash... j'ai du travailler avec les versions trial, et pour prolonger la période d'essai, je ai du l'implémenter dans quelques ordinateur.

En dehors de ces problèmes là, mon stage s'est très bien passé. Comme j'ai de la chance de à travailler à Télécom ParisTech en tant qu'un ingénieur de recherche, nous continuerons à compléter notre implémentation, et à ajouter le moteur de décision chez le serveur.

BIBLIOGRAPHIE

[1] http://www.w3.org/TR/2012/PR-mmi-arch-20120814/

[2] http://www.w3.org/TR/mmi-framework/

[3] http://www.w3.org/TR/mmi-use-cases/

[4] http://www.w3.org/Graphics/SVG/

[5] http://www.w3.org/2002/mmi/2012/mmi-arch-ir/

[6] http://en.wikipedia.org/wiki/Multimodal_Architecture_and_Interfaces

[7] http://www.openstream.com/

[8] http://gpac.wp.mines-telecom.fr/

[9] 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

ANNEXES

1. Découverte des services locaux avec GPAC

SERVICE NAME	CURRENT AVAILABLE SERVICE DATA
migration.001	URN: urn:intermedia, URL: 137.194.23.205
ConnectionManager	URN: urn:schemas-upnp-org, URL: 137.194.23.205
X_MS_MediaReceiverRegistrar	URN: urn:microsoft.com, URL: 127.0.0.1
X_MS_MediaReceiverRegistrar	URN: urn:microsoft.com, URL: 137.194.22.243
X_MS_MediaReceiverRegistrar	URN: urn:microsoft.com, URL: 137.194.22.243
X_MS_MediaReceiverRegistrar	URN: urn:microsoft.com, URL: 137.194.23.18
X_MS_MediaReceiverRegistrar	URN: urn:microsoft.com, URL: 137.194.22.55
SyncManager	URN: urn:dmc-samsung-com, URL: 137.194.22.158

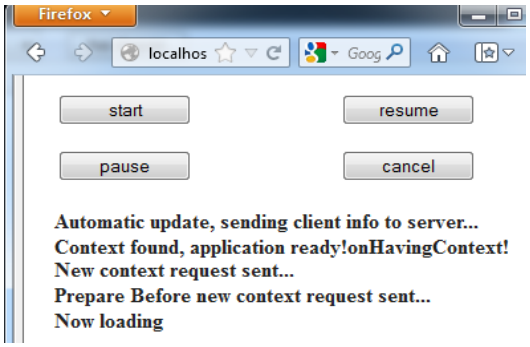
Figure 27: Découverte des services locaux utilisant GPAC

La découverte des services locaux s'est fait avec GPAC. J'ai utilisé les API du GPAC pour découvrir des services. Ensuit, je structure les données récupérées utilisant JSON. Puis, ces données sont envoyées vers une interface GUI construit avec SVG pour l'affichage. Le langage utilisé est JavaScript est SVG. [Figure 27] montre les services au tour de moi au moment que j'écris mon rapport.

2. Composants de modalité

2.1. Commandeur

Le commandeur est un composant de sortie [Chapitre 3, section 3.3.2: Interaction avec les commandes] implémenté en *JavaScript*, il s'agit d'un composant très simple ayant des boutons capable d'envoyer des commandes (START, STOP, PAUSE, RESUME).



Pour qu'il fonctionne, il devait passer par des étapes: le chargement, l'enregistrement, la demande du contexte [Chapitre 3, section 3.3.1]

Une fois qu'il est prêt, il pourrait envoyer des commandes vers les composants contrôlés (indirectement via le serveur).

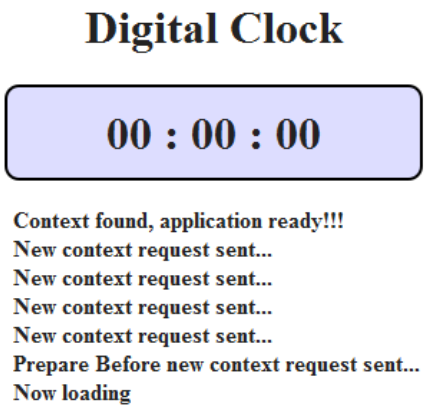
2.2. Horloge

L'horloge est un composant d'entrée très simple implémenté en *JavaScript* qui affiche le temps sur l'interface GUI. Son fonctionnement montre la réception d'une commande en provenance du serveur. (En fait, le serveur est juste un courtier, il reçoit la commande depuis des composants d'entrée. Et puis, il la transfère vers d'autre composant de sortie). Dans notre cas, l'horloge est un composant de sortie.

Etant un service construit au delà de la plateforme MMI, son fonctionnement passe de l'étape par des étapes [Figure 19].

- La première étape est *LOAD:* ce composant de modalité via son contrôleur charge sa configuration. Entre temps, le contrôleur lance un événement indiquant son état courant en affichant "*Now Loading*" dans l'interface GUI

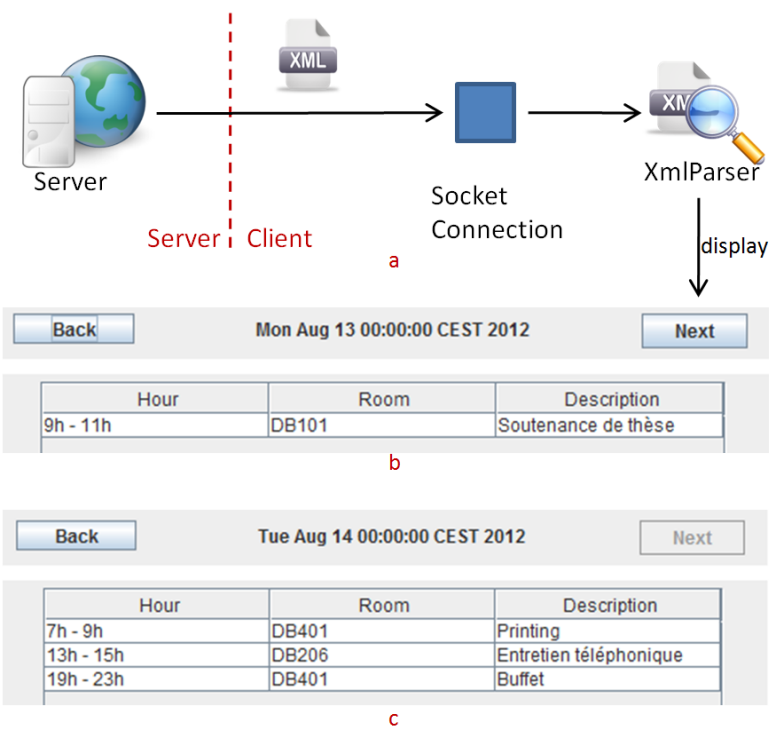
- Dans la deuxième étape, le système s'inscrit lui-même avec le serveur. S'il réussit à le faire, il lance l'événement *PREPARE_BEFORE_NEW_CTX* qui sera ensuite capturé par le service d'horloge. Ce service confirme sa réception via l'affichage du texte "*Prepare before new context request sent...*"
  Pour la troisième étape, le système périodiquement envoie la requête *NewContextRequest*. Dans notre exemple, après la quatrième demande, le service réussit à un *contextId* et il affiche "*Context found, application ready!!!*". L'horloge est désormais disponible et pourrait recevoir des commandes (START, PAUSE...)



2.3. Calendrier

Calendrier est d'autre 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

Considérons [Figure 28]: Via la connexion `Socket`, il reçoit la commande `START` *(a)*. En traitant cette commande, il rend disponible l'interface GUI (b et c). Les données se trouvent dans un fichier XML en provenance du serveur donc la structure est comme la suivante:

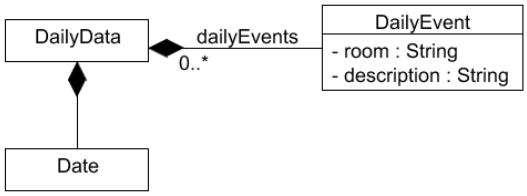



```
<dailyData date="13/08/12">
  <dailyEvent>
    <startHour>9</startHour>
    <endHour>11</endHour>
    <room>DB101</room>
    <description>
        Soutenance de thèse
    </description>
  </dailyEvent>
</dailyData>

<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 28: Le calendrier.

Le contenu du fichier sera analysé par le *XmlParser* et sera stocké dans les objets Java, disons *DailyData* et *DailyEvent*:



- La balise `<dailyData>` devient la classe *DailyData*
- La balise `< dailyEvent >` devient la classe *DailyEvent*

L'interface GUI de ce calendrier les lit pour l'affichage après [Figure 28 - a,b].

3. Diagrammes

3.1. Processus de hand-shaking

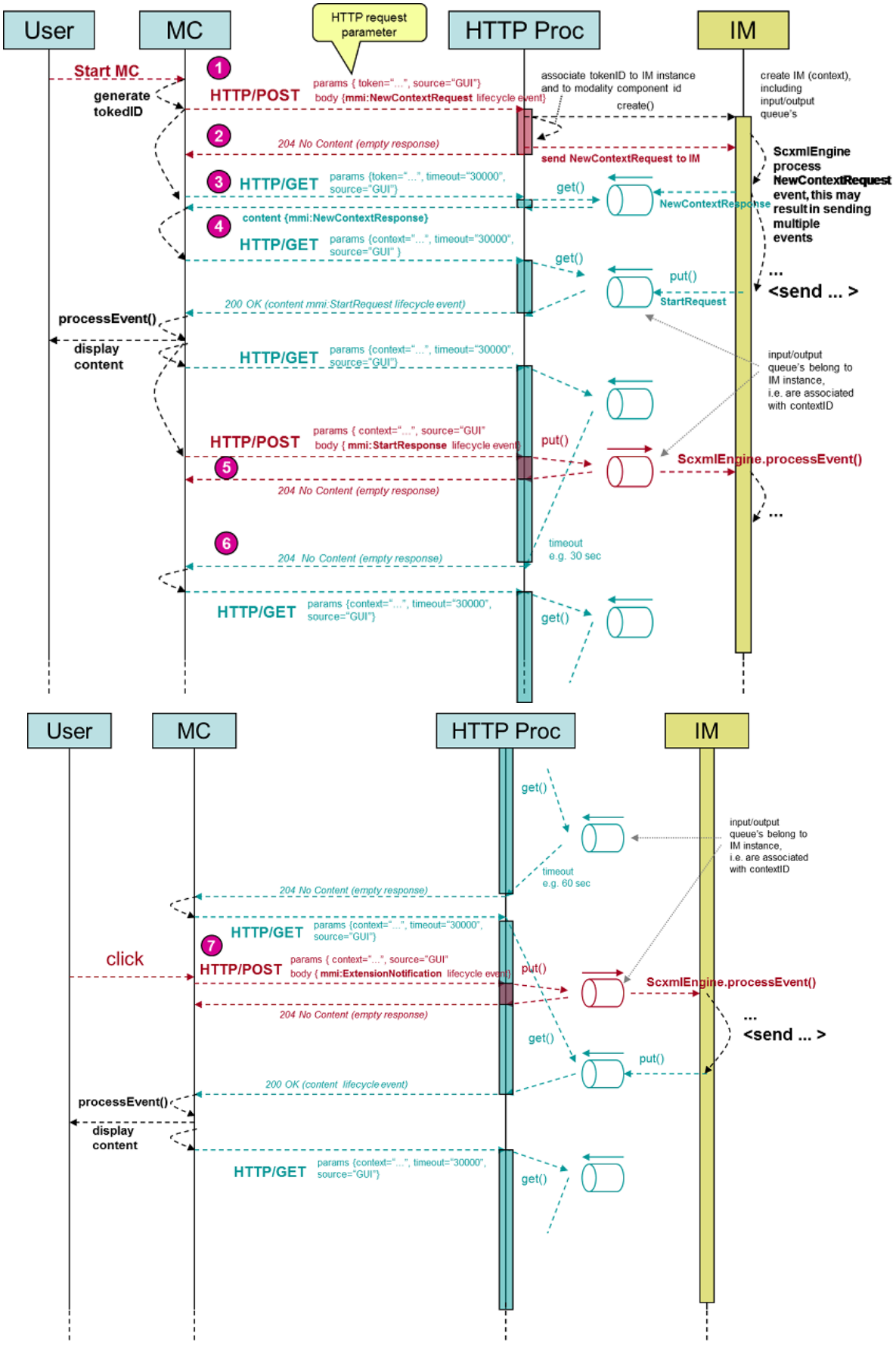


Figure 29: Processus de hand-shaking
Source: www.w3.org/TR/2012/PR-mmi-arch-20120814/

3.2. Cycle d'interaction

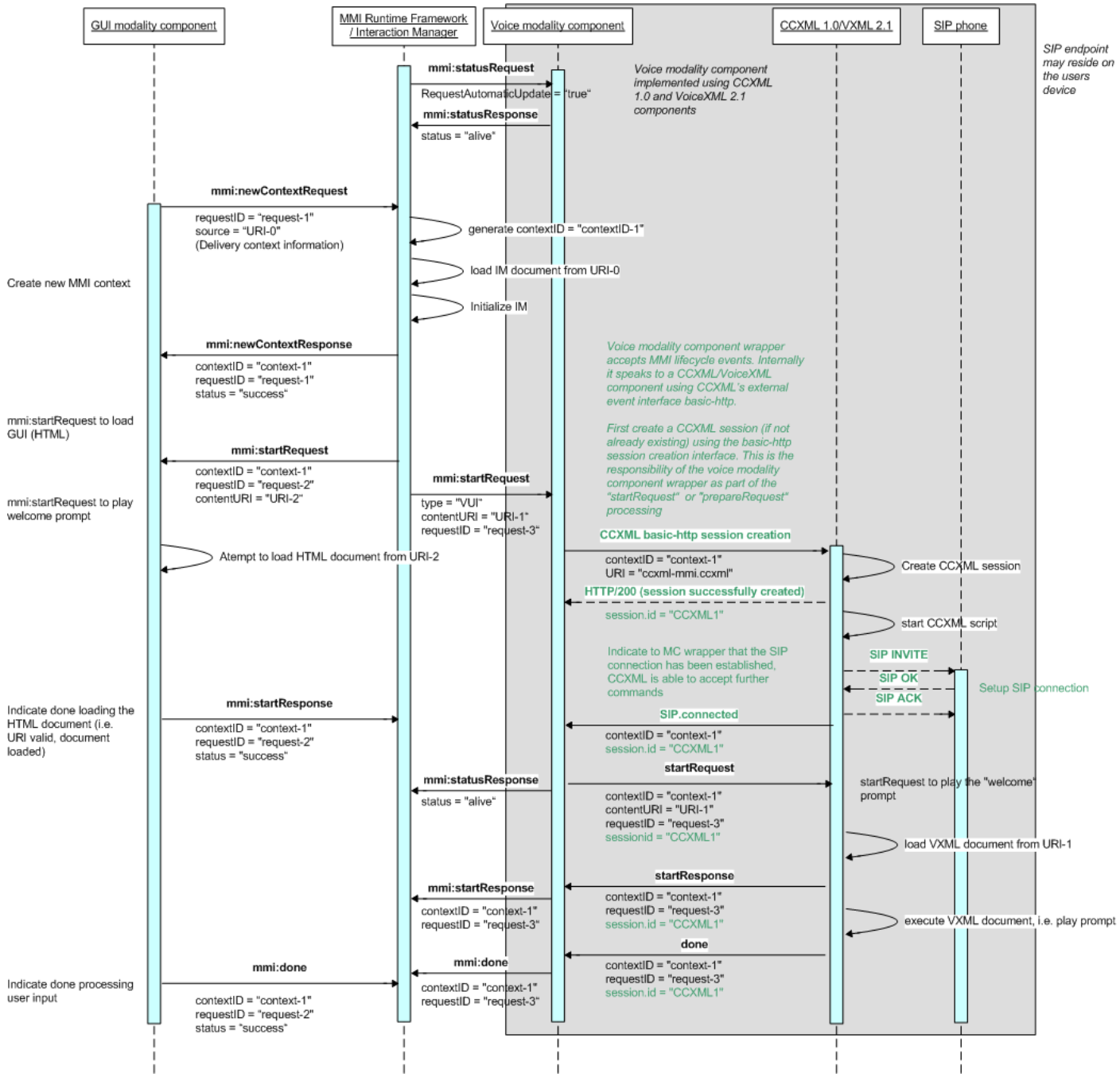


Figure 30: Cycle d'interaction

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

3.3. Traitement d'entrée d'utilisateur

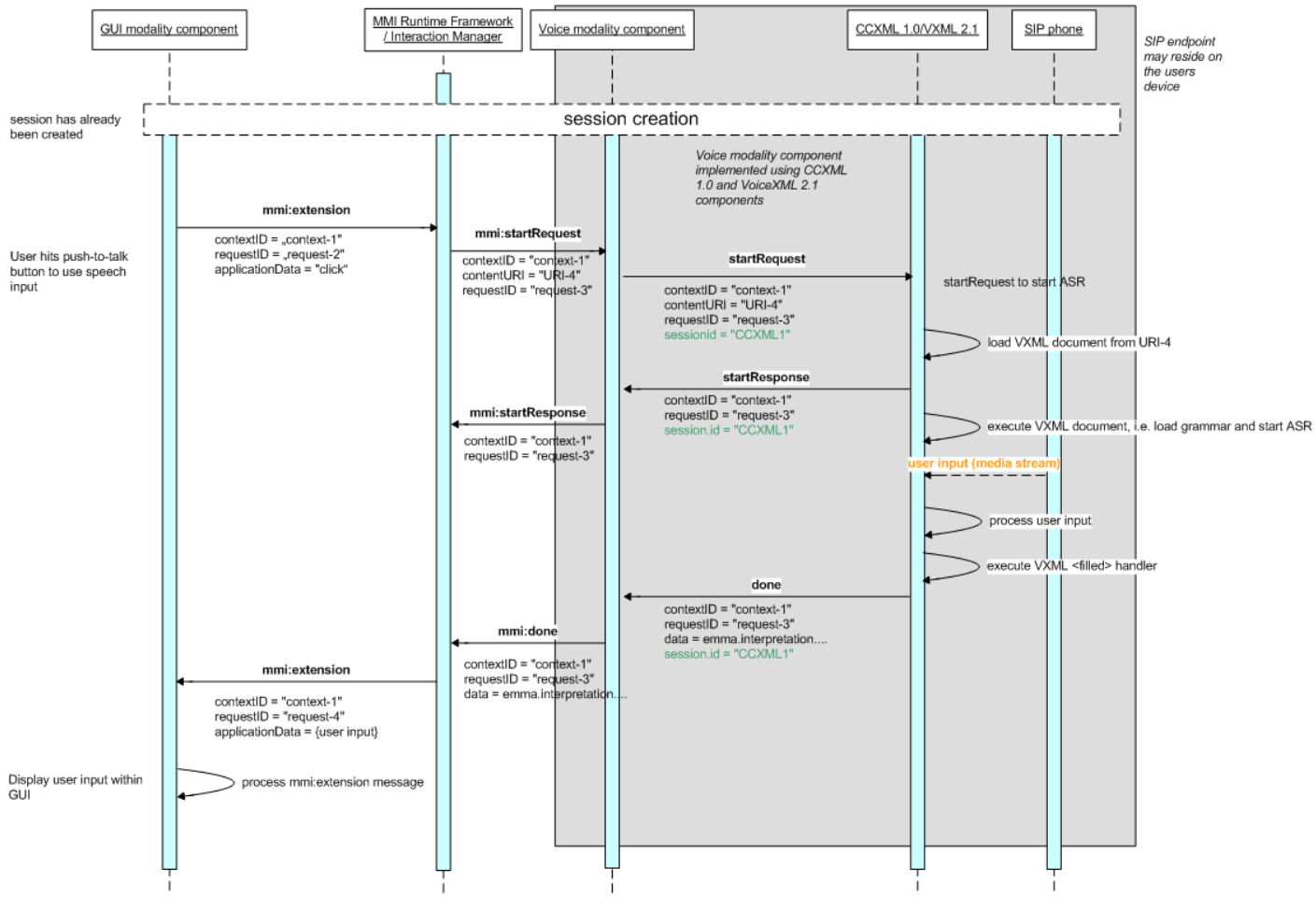


Figure 31: Traitement d'entrée d'utilisateur
Source: www.w3.org/TR/2012/PR-mmi-arch-20120814/

3.4. Terminaison de la session

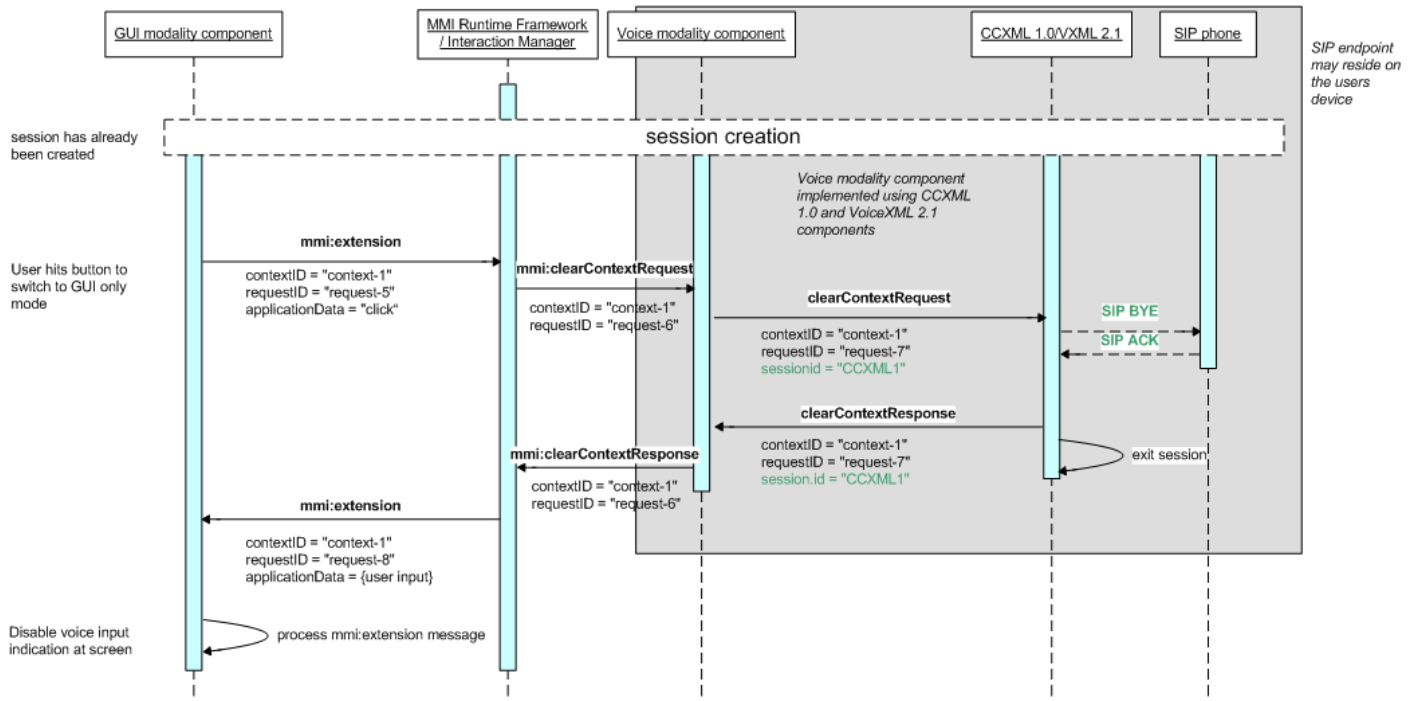


Figure 32: Terminaison de la session
Source: www.w3.org/TR/2012/PR-mmi-arch-20120814/