Vers une approche simplifiée pour introduire le caractère incrémental dans les systèmes de dialogue

Hatim Khouzaimi[1,2] Romain Laroche[1] Fabrice Lefèvre[2]
(1) Orange Labs, 38-40 rue du Général Leclerc 92794 Issy-les-Moulineaux, France
(2) Laboratoire Informatique d'Avignon, 339 chemin des Meinajaries 84911 Avignon, France
hatim.khouzaimi@orange.com, romain.laroche@orange.com, fabrice.lefevre@univ-avignon.fr

Résumé. Le dialogue incrémental est au cœur de la recherche actuelle dans le domaine des systèmes de dialogue. Plusieurs architectures et modèles ont été publiés comme (Allen *et al.*, 2001; Schlangen & Skantze, 2011). Ces approches ont permis de comprendre différentes facettes du dialogue incrémental, cependant, les implémenter nécessite de repartir de zéro car elles sont fondamentalement différentes des architectures qui existent dans les systèmes de dialogue actuels. Notre approche se démarque par sa réutilisation de l'existant pour tendre vers une nouvelle génération de systèmes de dialogue qui ont un comportement incrémental mais dont le fonctionnement interne est basé sur les principes du dialogue traditionnel. Ce papier propose d'intercaler un module, appelé *Scheduler*, entre le service et le client. Ce *Scheduler* se charge de la gestion des événements asynchrones, de manière à reproduire le comportement des systèmes incrémentaux vu du client. Le service, de son côté, ne se comporte pas de manière incrémentale.

Abstract. Incremental dialogue is at the heart of current research in the field of dialogue systems. Several architectures and models have been published such as (Allen *et al.*, 2001; Schlangen & Skantze, 2011). This work has made it possible to understand many aspects of incremental dialogue, however, in order to implement these solutions, one needs to start from scratch as the existing architectures are inherently different. Our approach is different as it tends towards a new generation of incremental systems that behave incrementally but work internally in a traditional way. This paper suggests inserting a new module, called the *Scheduler*, between the service and the client. This *Scheduler* manages the asynchronous events, hence reproducing the behaviour of incremental systems from the client's point of view. On the other end, the service does not work incrementally.

Mots-clés : Systèmes de Dialogue, Traitement Incrémental, Architecture des Systèmes de Dialogue.

Keywords: Dialogue Systems, Incremental Processing, Dialogue Systems Architecture.

1 Introduction

Les systèmes de dialogue traditionnels [1] fonctionnent au tour par tour. L'utilisateur parle et quand il se tait, il donne la parole au système. Certains systèmes permettent à l'utilisateur de les interrompre pour éviter les désynchronisations (*barge-in*) mais sans relier le moment et le contenu de son intervention avec la phrase du système et sans pouvoir continuer à parler en ignorant certaines interruptions (confirmations, bruit...). Ce modèle de dialogue a certes l'avantage d'être simple, mais il est loin de la réalité du dialogue naturel entre humains (Edlund *et al.*, 2008). Quand ceux-ci interagissent entre eux, ils se comprennent au fur et à mesure qu'ils parlent, peuvent s'interrompre mutuellement et peuvent même deviner la fin d'une proposition avant que celle-ci ne soit totalement prononcée par la personne qui parle (Tanenhaus *et al.*, 1995; Brown-Schmidt & Hanna, 2011; DeVault *et al.*, 2011). Des travaux ont également mis en valeur un procédé par étapes, lors de la construction du sens pendant la lecture (Ilkin & Sturt, 2011) et d'autres documents plus généraux en psycholinguistique évoquent ce phénomène (Levelt, 1989; Clark, 1996).

Intégrer ce genre de comportement dans les systèmes de dialogue permet d'avoir des systèmes plus réactifs et qui offrent une expérience utilisateur potentiellement plus agréable car plus proche du dialogue homme-homme. On parle de dialogue incrémental. De nombreuses études ont montré la supériorité des stratégies de dialogue incrémentales en termes de

1. Tout au long de ce papier, nous utiliserons l'adjectif *traditionnel* pour qualifier les systèmes de dialogue non-incrémentaux.

satisfaction utilisateur (Skantze & Schlangen, 2009; Baumann & Schlangen, 2013; El Asri *et al.*, 2014) et de complétion de tâche (Matthias, 2008; El Asri *et al.*, 2014).

Le terme *incrémental* a été utilisé initialement dans le domaine de l'informatique. Un compilateur incrémental (Lock, 1965) compile chaque ligne indépendamment des autres, ainsi, une modification locale ne peut pas affecter la compilation globale. (Wirén, 1992) utilise pour la première fois cette notion pour l'analyse du langage naturel. Les données en entrée d'un module incrémental ne lui sont pas communiquées en un seul bloc mais elles sont divisées en plusieurs morceaux. Dès le premier fragment, ce module commence déjà son traitement et cela donne lieu à des sorties hypothétiques disponibles dès qu'elles sont calculées.

Un système de dialogue incrémental fonctionne selon le même principe. La requête de l'utilisateur est divisée en *unités incrémentales* (Schlangen & Skantze, 2011) (e. g. division temporelle du signal audio pour les systèmes vocaux, division en mots pour les systèmes texte...) qui sont envoyés à la suite au système. Ce dernier maintient une hypothèse de réponse qui évolue au fur et à mesure avec l'arrivée de nouvelles informations. Cette évolution peut être visible par l'utilisateur ou cachée. Dans le cas des systèmes multimodaux, les canaux autres que la parole peuvent être exploités pour faire un *feedback* à l'utilisateur pendant qu'il parle (Fink *et al.*, 1998) (par exemple, un avatar qui hoche la tête quand il comprend une nouvelle information et fronce les sourcils quand il détecte une incohérence...). Par ailleurs, un système de dialogue incrémental reste à l'écoute de l'utilisateur même quand il prend la parole. Ainsi, outre la réactivité, un des avantages majeurs des stratégies de dialogue incrémentales réside dans la possibilité pour l'utilisateur d'interrompre le système (Matsuyama *et al.*, 2009; Selfridge *et al.*, 2013) quand il pense que celui-ci a mal compris sa requête. Contrairement au *barge-in* autorisé par certains systèmes traditionnels, le contenu et l'instant de l'intervention sont mis en relation avec l'énoncé interrompu pour en dégager le sens. De plus, le système peut choisir de ne pas réagir à certaines interruptions (comme les confirmations par exemple). Par conséquent, l'intervention de l'utilisateur peut ne porter que sur un fragment de la réponse du système comme c'est le cas dans les stratégies d'énumération (El Asri *et al.*, 2014). Il peut donc rattraper une désynchronisation plus rapidement et plus facilement. En plus, suivant le sens donné à l'interruption, celle-ci peut donner lieu à des actions différentes ou peut être ignorée.

Plusieurs travaux proposant des architectures de systèmes de dialogue incrémentaux existent déjà. Néanmoins, ces architectures présupposent de construire de tels systèmes en partant de rien, ce qui constitue un investissement considérable. Un système de dialogue est composé de plusieurs modules constituant *la chaîne de dialogue*, qui généralement se présente comme suit : la reconnaissance vocale, le traitement du langage naturel, la gestion du dialogue, la génération de langage naturel puis la synthèse de la parole. Pour rendre un tel système incrémental, des travaux antérieurs proposent de rendre la plupart de ses modules incrémentaux. Par opposition, notre démarche vise à transformer un système de dialogue traditionnel en système incrémental à moindre coût, en procédant par étapes.

La section 2 décrit l'état de l'art concernant les systèmes de dialogue incrémentaux. Dans le cadre de ce papier, la solution proposée consiste à introduire un module entre le client et le service de dialogue : le *Scheduler*. Ce concept sera introduit en Section 3. En Section 4, la méthode sera appliquée à un assistant texte de recherche de contenu puis à un système vocal de dictée de numéros : DictaNum. Enfin, une discussion est menée en Section 5 et la Section 6 conclut ce papier par des perspectives d'amélioration.

2 Travaux précédents

Nous distinguons quatre catégories de systèmes de dialogue suivant leur niveau d'intégration du caractère incrémental. La première concerne les systèmes traditionnels (Laroche *et al.*, 2011). La seconde englobe les systèmes de dialogue traditionnels avec quelques stratégies isolées (El Asri *et al.*, 2014) et la troisième se compose des systèmes qui reproduisent un comportement incrémental tout en gardant un fonctionnement interne traditionnel (Selfridge *et al.*, 2012; Hastie *et al.*, 2013). Enfin, les systèmes incrémentaux dont les composants internes le sont également constituent le dernier groupe (Dohsaka & Shimazu, 1997; Allen *et al.*, 2001; Schlangen & Skantze, 2011). La figure 5, discutée en section 5, propose une comparaison entre ces catégories en termes de fonctionnalités offertes par chacune.

NASTIA (El Asri *et al.*, 2014) est un système de dialogue destiné à la prise de rendez-vous avec un technicien pour une intervention à domicile. Ce système vient à la suite des travaux menés lors du projet européen CLASSiC (Laroche & Putois, 2010). Il interagit avec l'utilisateur en utilisant plusieurs stratégies de dialogue destinées à la récupération d'un créneau durant lequel le technicien peut se déplacer. Parmi celles-ci, la stratégie nommée *List of Availabilities* (LA) consiste à lister les créneaux disponibles et à attendre que l'utilisateur interrompe le système quand il entend une option qui l'intéresse. Cette dernière stratégie relève du dialogue incrémental car elle ne fonctionne pas en tour de parole.

L'expérimentation menée dans (El Asri *et al.*, 2014) montre qu'elle permet d'avoir un gain de près de 10% de taux de complétion de tâche ainsi qu'une amélioration significative de l'expérience utilisateur.

PARLANCE (Hastie *et al.*, 2013) est un exemple de système de dialogue appartenant à la troisième catégorie. Il a été développé au sein du projet européen du même nom. Son architecture conserve les même modules qu'une architecture classique mais avec quelques fonctionnalités en plus pour supporter un comportement incrémental : "the PARLANCE system architecture [...] maintains the modularity of a traditional SDS while at the same time allowing for complex interaction at the micro-turn level between components". Le principal module qui fait la différence avec les autres architectures est le MIM (Micro-turn Interaction Manager). Il gère les prises de parole du système, les périodes d'écoute et la génération des backchannels, tout ceci à l'échelle du micro-tour. L'architecture la plus proche de celle proposée ici est introduite dans (Selfridge *et al.*, 2012). Un module est introduit entre la reconnaissance vocale incrémentale et la synthèse de parole d'un côté et un service de dialogue de l'autre. Néanmoins, il ne s'agit pas de l'idée centrale du papier qui présente cette démarche comme étant un travail préliminaire pour pouvoir simuler un dialogue incrémental à moindre coût, et ainsi faire une expérimentation illustrant d'autres points qui ne nous intéressent pas ici. Dans cet article, nous avons choisi d'étudier cette démarche dans le détail, d'y apporter une formalisation, de l'ancrer dans la littérature existante et de prouver son fonctionnement à travers deux implémentations.

L'architecture proposée dans (Dohsaka & Shimazu, 1997) contient huit modules fonctionnant en parallèle : le Speech Recognizer, le Response Analyzer, le Dialogue Controller, le Problem Solver, l'Utterance Planner, l'Utterance Controller, le Speech Synthesizer et le Pause Monitor. La requête de l'utilisateur est vue comme un problème que l'utilisateur soumet au système. Celle-ci est captée par le Speech Recognizer, transmise sous forme de texte au Response Analyzer qui en dégage des concepts compréhensibles par le Dialogue Controller. Ce-dernier communique à la fois avec le Problem Solver et l'Utterance Planner qui délivre une réponse à l'utilisateur à travers l'Utterance Controller et le Speech Synthesizer. L'objectif est de pouvoir commencer à proposer une solution au problème alors qu'il est toujours en cours de résolution. Cette architecture appartient à la dernière catégorie de systèmes de dialogue sur l'échelle relative au caractère incrémental. De même, dans (Allen *et al.*, 2001), l'architecture proposée se compose de trois modules principaux : l'Interpretation Manager, le Behavioral Agent et le Generation Manager. Chaque nouvelle requête de l'utilisateur est captée par l'Interpretation Manager. Celui-ci diffuse cette information dans le système de manière incrémentale. Le Behavioral Agent est chargé de gérer le plan d'action du système et le Generation Manager s'occupe des interactions avec l'utilisateur. Tous deux sont construits de façon à agir de façon incrémentale compte-tenu des entrées venant de l'Interpretation Manager.

Une architecture plus générale est décrite dans (Schlangen & Skantze, 2011). Un système de dialogue peut être vu dans le cadre général comme une chaîne de modules séparés ayant chacun une tâche précise. Pour réaliser un système incrémental, (Schlangen & Skantze, 2011) part de ce postulat de départ et sépare chacun de ces modules en 3 parties : le Left Buffer, l'Internal State et le Right Buffer. Le Left Buffer représente l'entrée du module, l'Internal State désigne l'état de celui-ci et le Right Buffer contient la sortie. L'information est transportée et propagée dans le système sous forme d'IUs (Incremental Units). Par exemple, toutes les 500 ms, une nouvelle IU sous forme de signal sonore est placée dans le Left Buffer de la reconnaissance vocale (ASR) qui la transforme en IU au format texte. Le Right Buffer d'un module est le Left Buffer du suivant ce qui assure la propagation de l'information. Ce modèle générique englobe les systèmes de dialogue des quatre catégories, les systèmes non-incrémentaux étant perçus comme des cas particuliers de systèmes incrémentaux : "we can now see that a non-incremental system can be characterised as a special case of an incremental system, namely one where IUs are always maximally complete [...] and where all modules update in one go".

L'architecture proposée ici fait partie de la troisième catégorie de notre classification. Par comparaison à PARLANCE, le *Scheduler* a un rôle similaire au MIM, cependant, il est construit pour s'interfacer avec une architecture traditionnelle préexistante, rajoutant ainsi une couche au service pour simuler un comportement incrémental. Le MIM, quant à lui, s'insère dans une architecture qui est faite de façon à communiquer avec lui (plusieurs autres modules s'interfacent directement avec lui), jouant ainsi un rôle central dans le système de dialogue. L'avantage de ce type de systèmes réside dans le fait qu'ils se comportent de manière incrémentale tout au long du dialogue (contrairement aux systèmes des deux premières catégories) et sont donc plus réactifs, plus naturels tout en présentant moins de problèmes de désynchronisation. En outre, par opposition à la dernière catégorie, la conception de tels systèmes est moins coûteuse car elle permet de partir de systèmes non-incrémentaux existants et de les rendre incrémentaux. Dans ce qui suit, nous formalisons notre approche en utilisant les concepts introduits dans le modèle général et abstrait de (Schlangen & Skantze, 2011).

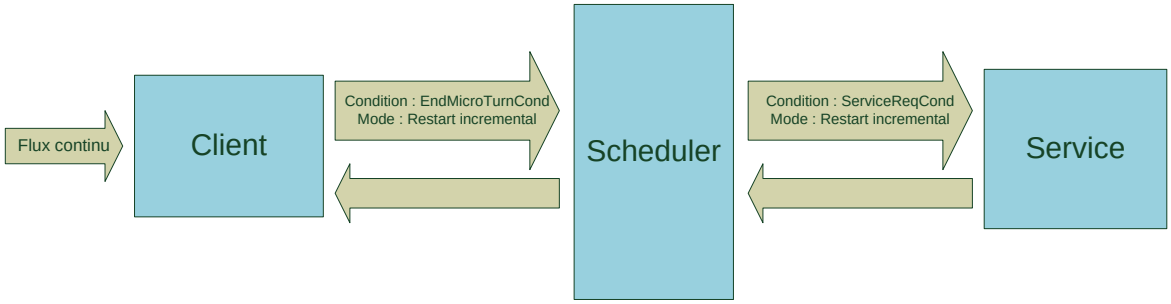

FIGURE 1 – Le *Scheduler* : module intermédiaire entre le client et le système de dialogue.

3 Le Scheduler

En règle générale, les systèmes de dialogue traditionnels se composent d'un client installé sur le terminal de l'utilisateur et d'un service déployé sur un serveur. Ils adoptent une approche tour par tour. Nous appellerons *tour de dialogue* l'intervalle de temps durant lequel l'utilisateur prend la parole une fois puis le système lui répond. Ainsi, le temps du dialogue peut être vu comme un enchaînement de tours de dialogue : T^1, T^2,..., T^k... Au tour T^k, le client envoie une requête complète Req^k au service qui la traite et renvoie la réponse Rep^k correspondante (il peut éventuellement effectuer un autre traitement en plus comme la modification d'une base de données). Par conséquent, ce tour peut lui-même être divisé en deux intervalles temporels, le *tour utilisateur* et le *tour système* $T^k = T^{k,U} \cup T^{k,S}$. En partant d'un tel système, et sans aucune autre hypothèse supplémentaire, nous allons montrer comment le rendre incrémental. La méthode consiste à intercaler un module intermédiaire entre le client et le service : le *Scheduler* (cf. Figure 1). Le rôle de celui-ci sera de simuler un comportement incrémental vu du client (le terme *Scheduler* est emprunté à (Laroche, 2010)). Cette architecture est une instanciation du modèle abstrait proposé dans (Schlangen & Skantze, 2011). Le client, le *Scheduler* et le service en constituent les modules. Les deux premiers sont incrémentaux alors que le troisième ne l'est pas. Il n'est pas pertinent pour nous de parler de Left Buffer et de Right Buffer, les liens entre les modules sont vus comme des canaux d'information (réseau dans notre implémentation). Dans la suite de cette section, nous décrirons les comportements traditionnels du client et du service, avant d'aborder les modifications qui leurs sont apportées dans notre nouvelle architecture.

3.1 Rôle du client et du service dans une architecture traditionnelle

Le client reçoit un signal provenant de l'utilisateur. Généralement, celui-ci se présente sous forme d'un flux continu d'information (signal audio en vocal, suite de caractères en texte...). Si cela n'est pas vérifié (par exemple, une interface web où chaque bouton désigne une requête), la notion de dialogue incrémental n'a pas de sens. Ainsi, le client doit avoir une condition $EndTurnCond$ (End of Turn Condition) lui permettant de savoir à quel moment envoyer la requête au service. Cette condition correspond généralement à un silence suffisamment long (Raux & Eskenazi, 2008; Wlodarczak & Wagner, 2013) pour les systèmes vocaux et à un retour chariot pour les systèmes textuels. Nous appellerons *instant d'activation* d'une condition le moment où elle passe de $faux$ à $vrai$. Un dialogue traditionnel est une alternance de tours utilisateur dont la fin est déterminée par l'instant d'activation de $EndTurnCond$ et de tours du système qui se terminent quand celui-ci redonne la parole à l'utilisateur. Il se peut que ce dernier ne dise rien durant le temps qui lui est imparti, nous appellerons tout de même cela un tour de parole.

Le service se compose d'une interface avec le client, d'un contexte interne et d'une interface avec le monde extérieur. L'interface avec le client gère la communication avec celui-ci ainsi que la compréhension de ses requêtes et le contexte interne correspond à toutes les informations dont dispose le service sur l'état du dialogue à chaque instant. L'interface avec le monde extérieur lui permet d'envoyer des requêtes et d'agir sur des modules externes au système de dialogue (bases de données ou appareils en domotique par exemple).

3.2 Passage en mode incrémental

Pour rendre un tel système incrémental, nous modifions la façon dont les requêtes sont envoyées. Une nouvelle condition définit l'envoi d'une nouvelle requête de la part du client. Elle est notée $EndMicroTurnCond$ (End of Micro-Turn

Condition, cf. Figure 1) et est moins restrictive que $EndTurnCond$: $EndTurnCond$ implique $EndMicroTurnCond$ (le client envoie des requêtes plus souvent). Nous appellerons *micro-tour utilisateur* l'intervalle temporel compris entre deux instants d'activation de $EndMicroTurnCond$. $T^{k,U}$ peut être divisé en $n^{k,U}$ micro-tours utilisateur $\mu T_i^{k,U}$: $T^{k,U} = \bigcup_{i=1}^{n^{k,U}} \mu T_i^{k,U}$. Nous définissons également le *sous-tour utilisateur* $T_p^{k,U} = \bigcup_{i=1}^{p} \mu T_i^{k,U}$ où $1 \leq p \leq n^{k,U}$. Nous manipulons des intervalles temporels, d'où le choix de l'opérateur d'union. En général, $EndMicroTurnCond$ correspond à un cycle d'horloge précis (nouveau micro-tour toutes les 500 ms...) ou à l'arrivée d'une nouvelle information (modification de la sortie de l'ASR par exemple). Notons également qu'à chaque instant d'activation de $EndTurnCond$, le client envoie un signal dédié noté *signal_ETC* pour signaler cet événement au *Scheduler*.

Le *Scheduler* est un module qu'on propose de placer entre le client et le service. Cet intermédiaire a pour rôle de rendre l'ensemble {Scheduler + Service} équivalent à un système incrémental vu du client sans modifier le fonctionnement du service. À chaque micro-tour utilisateur, il reçoit une nouvelle entrée venant du client. Le *Scheduler* est également muni d'une condition $ServiceReqCond$ (Service Request Condition, cf. Figure 1) sous laquelle il transmet la requête venant du client au service, récupère la réponse correspondante et la stocke pour qu'elle puisse être récupérée ultérieurement par le client. Cette condition peut être vraie tout le temps, auquel cas, à chaque requête provenant du client, le service est sollicité pour obtenir une réponse. Une façon simple et plus économe en nombre de requêtes est de définir $ServiceReqCond$ comme l'arrivée d'une requête différente de la précédente. Si le *Scheduler* intervient juste après l'ASR, il pourra vérifier que le texte correspondant à la requête a changé (Si $EndMicroTurnCond$ inclut déjà cette condition, il est inutile de la rajouter à $ServiceReqCond$). De même, s'il intervient après l'analyse sémantique, la vérification se portera sur l'arrivée d'un nouveau concept.

L'autre rôle clé du *Scheduler* est de décider du moment où il engage le système à valider l'hypothèse de requête en cours et à s'en tenir à elle (en n'attendant plus de nouvelles informations de la part du client pour venir la confirmer ou l'infirmer). Ceci marque la fin du tour en cours et correspond à la notion de *commit* (Schlangen & Skantze, 2011) décrite en détail dans la section 3.3. On notera $CommitCond$ (Commit Condition) la condition sous laquelle le *Scheduler* prend cette décision. Par exemple, si l'utilisateur doit fournir son numéro de téléphone à 10 chiffres au service, on pourra prendre $CommitCond = (length(num) == 10)$ où $length(num)$ est la longueur du numéro reconnu à chaque instant. Il est important de noter que dans le cadre incrémental proposé, c'est l'instant d'activation de $CommitCond$ qui marque la fin d'un tour et non pas celui de $EndTurnCond$ (le *Scheduler* peut décider d'effectuer un *commit* sans recevoir de *signal_ETC* de la part du client). Néanmoins, $EndTurnCond$ implique $CommitCond$.

À chaque nouveau tour utilisateur $T^{k,U}$, l'utilisateur formule une nouvelle requête et au micro-tour $\mu T_i^{k,U}$, celle-ci n'est pas encore accessible dans son intégralité (sauf si $i = n^{k,U}$), néanmoins, le système dispose déjà d'une hypothèse provisoire qui sera appelée *sous-requête* et notée Req_i^k. Celle-ci sera envoyée au *Scheduler* à chaque micro-tour utilisateur. Le fait d'envoyer toute la requête depuis le début du tour utilisateur en cours correspond au mode *restart incremental* introduit dans (Schlangen & Skantze, 2011). Remarquons au passage que si $i_1 < i_2$ alors $Req_{i_1}^k$ n'est pas obligatoirement un préfixe de $Req_{i_2}^k$ (en vocal, l'arrivée d'un nouvel incrément de signal audio au niveau de l'ASR peut changer l'hypothèse de sortie et pas seulement la compléter).

Le client est composé de deux processus indépendants : le premier se charge de l'envoi des requêtes vers le *Scheduler* en suivant le fonctionnement décrit plus haut, et le second récupère les réponses stockées dans ce-dernier. La récupération de ces réponses se fait à une fréquence de l'ordre de la fréquence des micro-tours utilisateur afin de s'assurer que le client est en permanence à jour (qu'il a récupéré la dernière réponse du service disponible dans le *Scheduler*). Dans le cas des systèmes vocaux, le *Scheduler* a pour tâche de déterminer quelles réponses du système devraient être prononcées par la synthèse vocal et lesquelles devraient être ignorées. Pour cela, il ajoute un marqueur dédié aux réponses qu'il sélectionne avant qu'elles ne soient récupérées par le client. Cela permet au *Scheduler* de gérer les sorties système de manière incrémentale.

Tout l'intérêt de la solution proposée ici est que le service reste quasiment inchangé au niveau fonctionnel (quelques modifications au niveau applicatif peuvent s'avérer nécessaires, voir la section 4.2 pour un exemple). Le seul changement concerne le maintien d'un double contexte : le contexte réel et le contexte simulé (voir la section 3.3). À chaque fois que $ServiceReqCond$ est vraie, le *Scheduler* envoie au service une requête incomplète et récupère la réponse. Ainsi, il permet au client (si celui-ci choisit de récupérer les réponses intermédiaires) de voir ce qu'aurait répondu le service avant que la requête ne soit totalement formulée. Le fait de conserver un comportement non-incrémental du service et d'avoir besoin de toutes les IU depuis le début de la requête à chaque fois que ce-dernier est sollicité justifie le choix du mode *restart incremental*.

Au même titre que le *Scheduler*, le service peut également ordonner à ce dernier d'effectuer un *commit*. Une telle action doit être remontée au *Scheduler* pour l'informer qu'il ne peut pas remplacer la dernière hypothèse de requête par une autre (voir la notion de *rollback* décrite dans la section 3.3). La relation de *grounding* présentée dans (Schlangen & Skantze, 2011) permet d'avoir un tel mécanisme. À la sortie d'un module, chaque IU connaît l'ensemble des IU en entrée qui l'ont engendrée : on dit qu'elle est basée sur ces IU (*grounded in*). Ainsi, quand le module décide de valider une IU par un *commit*, cette validation se propage à toutes les IU sur laquelle elle est basée et ainsi de suite.

Enfin, tout comme nous avons défini la notions de micro-tour utilisateur, nous faisons de même du côté du système. Dans un cadre classique, la synthèse vocale (TTS) joue la réponse du système durant chaque tour système $T^{k,S}$. En dialogue incrémental, cette durée peut être divisée en $n^{k,S}$ micro-tours système $\mu T_i^{k,S}$: $T^{k,S} = \bigcup_{i=1}^{n^{k,S}} \mu T_i^{k,S}$. Cette division est déterminée par le service au moment où il envoie sa réponse (cf. 4.2 pour un exemple). Quand l'utilisateur interrompt le système, le timing de son interruption est donné par le micro-tour système durant lequel il est intervenu. Par ailleurs, au moment du *barge-in*, on passe au tour suivant T^{k+1}. Remarquons que cela ne concerne que les systèmes vocaux, les systèmes textuels ne pouvant être interrompus.

3.3 Commit, rollback et double contexte

Dans le cadre incrémental, le système formule des hypothèses de réponse au fur et à mesure que l'utilisateur lui fournit des informations en entrée en complétant sa requête. Ces hypothèses fluctuent en fonction des nouvelles entrées, de la base des connaissances actuelles ou d'autres paramètres du système. Cependant, au bout d'un moment, le système doit effectuer une action qui engage le dialogue dans la direction imposée par la dernière hypothèse faite. Par exemple, si le système commence à formuler une réponse ou à modifier une base de données, il ne peut pas se permettre d'ignorer son hypothèse et de la modifier car celle-ci a donné naissance à une action concrète perceptible par l'utilisateur. À partir de ce moment là, on dit que le système a effectué un *commit* de sa dernière hypothèse.

L'opération inverse du *commit* sera appelée *rollback* (terme emprunté au lexique de la gestion de base de données). Tant que le *commit* d'une hypothèse n'est pas effectif, le système peut décider d'oublier cette hypothèse et de revenir à l'état dans lequel il était au moment du dernier *commit*.

Certaines requêtes du *Scheduler* peuvent être amenées à modifier le contexte du dialogue or cela n'est pas l'effet recherché. Ce module est censé communiquer avec le système de dialogue principalement pour voir ce qu'il aurait répondu à la requête à un instant donné (la plupart du temps encore incomplète). Pour remédier à cela, une solution consiste à maintenir deux contextes au sein des systèmes de dialogue : un contexte *réel* qui est l'équivalent du contexte traditionnel, sauvegardé à chaque *commit*, et un contexte *de simulation* avec lequel travaillera le *Scheduler* pour obtenir les résultats intermédiaires du système.

Cette notion de double contexte est indissociable des idées de *commit* et *rollback*. Quand le *Scheduler* décide d'effectuer un *commit*, **il copie le contexte de simulation dans le contexte réel**. À l'inverse, il fait le *rollback* d'une hypothèse quand **il copie le contexte réel dans le contexte de simulation**. Il oublie ainsi ce qui s'est passé depuis le dernier *commit*.

À chaque micro-tour utilisateur, le client envoie au *Scheduler* tout ce que l'utilisateur a prononcé (ou tapé) depuis le dernier *commit*. Le *Scheduler* envoie ensuite cette requête (encore incomplète) au service et récupère la réponse. Si au prochain micro-tour, le *Scheduler* ne décide pas d'effectuer un *commit* mais souhaite plutôt envoyer une nouvelle requête, alors un ordre de *rollback* est envoyé au service avant cette requête, comme l'envoi se fait en mode *restart incremental* (dans le cadre de ce papier, l'opération de *rollback* intervient uniquement dans ce cas là). Les Figures 2 (diagramme de séquence) et 3 illustrent ce fonctionnement. Sur la Figure 2, les conditions $EndTurnCond$, $EndMicroTurnCond$, $ServiceReqCond$ et $CommitCond$ sont écrites à gauche des flux qu'ils génèrent. Les zones d'activité et d'inactivité du client concernent le processus d'envoi alors que les flèches en pointillées désignent les retours vers le processus de récupération des réponses. Ceux-ci ne sont pas synchronisés avec le reste des flux, même si, par souci de clarté, ils le sont sur le diagramme. Par ailleurs, sur cette figure, la décision de *commit* a été prise suite à un *signal_ETC* or ce n'est pas toujours le cas.

Nous notons $ctxt(T^k)$ le contexte réel obtenu à l'issue du tour T^k, $ctxt(T^0)$ étant le contexte initial au début du dialogue. Ce contexte fluctue durant les tours utilisateur mais reste fixe durant les tours système ($ctxt(T^k) = ctxt(T^{k,U})$). Au moment du *commit* qui marque la fin du tour T^k, le contexte réel prend la valeur du contexte simulé à cet instant là : $ctxt(T^k) = ctxt(T^{k-1} + T^{k,U}_{n^{k,U}})$.

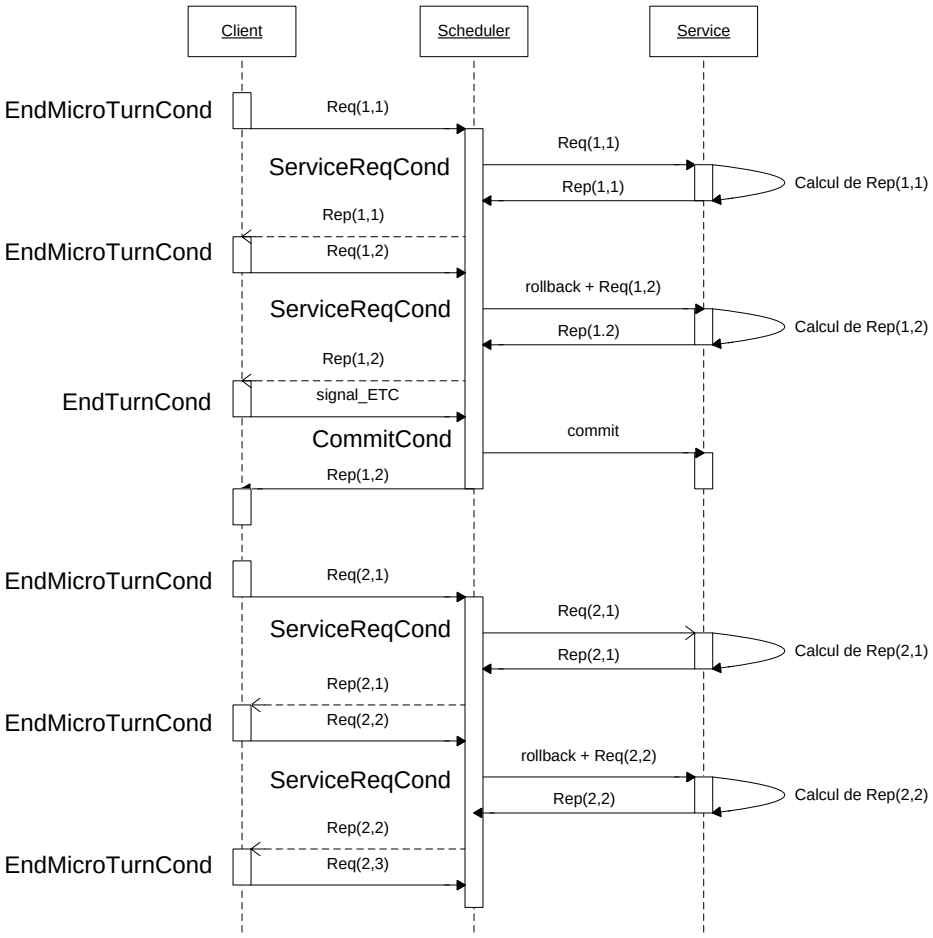

FIGURE 2 – Principe de fonctionnement du *Scheduler* (les flèches en pointillées désignent les flux vers le processus de récupération dans le client).

4 Implémentations

Dans le but de montrer la faisabilité de cette solution, deux systèmes de dialogue existants (développés à Orange Labs) ont été rendus incrémentaux à l'aide de l'introduction d'un *Scheduler*. Le premier est un service texte où le client se présente sous forme d'interface web et le second est un service vocal où l'utilisateur interagit avec une application Android. Avec un minimum de modifications, un comportement incrémental a été intégré à ces deux exemples ce qui montre la facilité d'implémentation et d'adaptation de la solution tout en démontrant le comportement incrémental du résultat obtenu, à la fois dans le cadre des stratégies implémentées et des modalités d'interaction utilisées (mode texte et interface web d'un côté et mode vocal de l'autre).

4.1 CFAsT : Content Finder AssistanT

Le CFAsT est une application permettant de générer automatiquement et rapidement un assistant de navigation en mode texte à partir d'une base de contenus donnée. Il a été développé au sein d'Orange Labs et les services qu'il génère se composent d'un service web déployé sur un serveur et d'une interface web où l'utilisateur peut entrer du texte (en langage naturel) ou appuyer sur des boutons (propositions de mots-clés ou de cibles).

Tour	Sous-tour utilisateur	Entrée	Contexte réel	Contexte simulé
T^1	$T_1^{1,U}$	Req_1^1	ctxt(T^0)	ctxt($T^0 + T_1^{1,U}$)
	$T_2^{1,U}$	Req_2^1	ctxt(T^0)	ctxt($T^0 + T_2^{1,U}$)
	...	...	ctxt(T^0)	...
	$T_{n^{1,U}}^{1,U}$	$Req_{n^{1,U}}^1$	ctxt(T^0)	ctxt($T^0 + T_{n^{1,U}}^{1,U}$)
	COMMIT : $ctxt(T^1) = ctxt(T^0 + T_{n^{1,U}}^{1,U})$			
T^2	$T_1^{2,U}$	Req_1^2	ctxt(T^1)	ctxt($T^1 + T_1^{2,U}$)
	...	...	ctxt(T^1)	...

FIGURE 3 – Un double contexte : le contexte réel et le contexte simulé.

Initialement, l'utilisateur est invité à entrer sa requête en langage naturel ou à choisir parmi un certain nombre de mots-clés. Ensuite, il se voit proposer des contenus en plus des mots-clés. Dans le cadre de cette étude, la base de contenus choisie est la FAQ de l'ANPE (Agence Nationale Pour l'Emploi). Un contenu correspond à un ensemble question/réponse de la FAQ. L'utilisateur peut rentrer sa requête en langage naturel ou sélectionner un des trois mots-clés qui lui sont présentés, ensuite, en plus de l'entrée texte, cinq propositions de contenus sont faites ainsi que trois nouvelles propositions contextuelles de mots-clés. L'interaction s'arrête quand un contenu est sélectionné par l'utilisateur.

Le système maintient une liste de mots-clés qu'il met à jour au fur et à mesure de l'interaction. Ainsi, la requête de l'utilisateur se construit et s'enrichit au fur et à mesure jusqu'à ce que l'utilisateur choisisse un contenu.

Le client de démonstration [2] se compose de deux vues : la première contient l'interface classique et la seconde est mise à jour à chaque fois que l'utilisateur tape un espace (à chaque nouveau mot). Elle représente ce qu'aurait répondu le système si l'utilisateur avait validé la requête à ce stade là. À chaque fois que l'utilisateur valide une requête par un retour chariot (cette validation entraîne un *commit* de la part du *Scheduler*), les deux vues sont identiques. Cette application de notre méthode n'a pas d'utilité pratique mais elle sert à en prouver le fonctionnement. En reprenant les concepts introduits ici : {**EndMicroTurnCond** : appui sur la barre d'espace, **EndTurnCond** : retour chariot, **ServiceReqCond** : EndMicroTurnCond, **CommitCond** : EndTurnCond}.

4.2 DictaNum

DictaNum [3] est un système de dialogue (inspiré de NUMBERS (Skantze & Schlangen, 2009)) qui recueille des numéros et confirme leur bonne compréhension. L'utilisateur interagit avec le service en utilisant un client web. La reconnaissance vocale et la synthèse de la parole sont assurées par l'API Web Speech sur Google Chrome. D'autres systèmes existants permettent de dicter des numéros (téléphone, carte bancaire...) comme *How may I help you ?* (Langkilde *et al.*, 1999).

L'intérêt principal de la version incrémentale est de pouvoir dicter son numéro par morceaux. Par conséquent, le système a été modifié au niveau applicatif. Au départ, le numéro est une chaîne vide et à chaque fois qu'un silence est détecté ($EndTurnCond$), la sortie de l'ASR est concaténée à celle-ci et le système émet un *feedback* en la répétant. L'utilisateur peut ensuite continuer sa dictée ou indiquer qu'il n'est pas d'accord avec ce *feedback* et le corriger en commençant sa phrase par 'Non' (voir l'exemple d'application plus bas). Le service comprendra qu'il faut remplacer le dernier fragment dicté et fera un second *feedback* en commençant par s'excuser : 'Désolé,...'. Cependant, effectuer une dictée par morceaux dans un cadre non-incrémental est un processus qui n'est pas naturel et qui manque de réactivité car la durée des silences est trop importante. L'utilisateur préférera effectuer sa dictée en une seule fois car cela prend moins de temps. Le client a été modifié de façon à ne plus attendre un silence de la part de l'utilisateur pour envoyer la requête. Toutes les 500 ms, il envoie tout ce que l'utilisateur a dit depuis le dernier *commit*. On note Δ_s le seuil de silence dans le cadre non-incrémental et δ_s un nouveau seuil utilisé pour morceler la dictée du numéro ($\delta_s \leq \Delta_s$). Au cours de celle-ci, quand l'utilisateur se tait pendant une période supérieure à δ_s (on parle de *micro-silence*), le système prend en compte le nouveau morceau de numéro et le répète à l'utilisateur jusqu'à ce qu'après un *feedback*, l'utilisateur manifeste un silence d'une durée supérieure à Δ_s (indiqué par l'envoi de la chaîne 'longSilence'). Le client détermine les durées du silence à l'aide du nombre de micro-tours consécutifs durant lesquels on ne détecte aucune sortie de l'ASR. Alternativement, nous aurions pu utiliser la VAD (Voice Activity Detection) comme dans (Breslin *et al.*, 2013).

2. http ://dialogue.orange-labs.fr/CFAsTIncr/
3. http ://dialogue.orange-labs.fr/DictaNum/

Pendant la dictée, à chaque micro-tour utilisateur ($EndMicroTurnCond$ = horloge de fréquence 2 Hz), le *Scheduler* vérifie que la requête du client est différente de la précédente ($ServiceReqCond$ = modification de la requête) auquel cas il effectue un *rollback* puis envoie une requête au service depuis le début du morceau de numéro en cours. Quand le client détecte un micro-silence, il envoie la chaîne de caractère 'silence' au *Scheduler* (non concaténée au fragment de numéro en cours) et c'est comme cela que $CommitCond$ est définie (un *commit* est donc effectué). Par ailleurs, le client requête la dernière réponse disponible dans le *Scheduler* à l'aide d'un processus indépendant, et ce, à la même fréquence que celle des micro-tours utilisateur. Ainsi, le micro-silence étant de l'ordre de quelques micro-tours, au moment où $CommitCond$ est vérifiée, le client dispose déjà du *feedback* du service et peut le jouer immédiatement.

Faire varier δ_s entre une petite valeur (quelques micro-tours utilisateur) et Δ_s crée un continuum entre un comportement incrémental et non incrémental. Si $\delta_s = \Delta_s$, il est plus confortable pour l'utilisateur de dicter son numéro en une seule fois (situation décrite plus haut), par contre si δ_s est sensiblement plus petit que Δ_s, l'utilisateur sera interrompu dans sa dictée à chaque micro-silence pour avoir un *feedback* (il ne faut pas que δ_s soit trop petit pour que les *feedbacks* ne deviennent pas trop fréquents et donc gênants pour la dictée).

Une interaction avec la version originale du système ressemble à ceci :

– **Système :** Bienvenue dans le service DictaNum. Veuillez dicter votre numéro.
– **Utilisateur :** 01 45 65 79 98.
– **Système :** Le numéro dicté est : 01 45 67 79 98. Est-ce bien cela ?
– **Utilisateur :** Non.
– **Système :** Désolé. Nous allons reprendre du début. Veuillez dicter votre numéro.
– **Utilisateur :** 01 45 65 79 98.
– **Système :** Le numéro dicté est : 01 47 65 79 98. Est-ce bien cela ?
– **Utilisateur :** Non.
– **Système :** Désolé. Nous allons reprendre du début. Veuillez dicter votre numéro.
– [...]
– **Système :** Le numéro dicté est : 01 45 65 79 98. Est-ce bien cela ?
– **Utilisateur :** Oui.
– **Système :** Merci d'avoir utilisé DictaNum. À bientôt.

En lui intégrant un comportement incrémental, le dialogue se déroule comme suit :

– **Système :** Bienvenue dans le service DictaNum. Veuillez dicter votre numéro.
– **Utilisateur :** 01 45 (silence de durée δ_s)
– **Système :** 01 45
– **Utilisateur :** 65 79 (silence de durée δ_s)
– **Système :** 67 79
– **Utilisateur :** Non, 65 79 (silence de durée δ_s)
– **Système :** Désolé, 65 79
– **Utilisateur :** 98 (silence de durée δ_s)
– **Système :** 98
– **Utilisateur :** ... (silence de durée Δ_s)
– **Système :** Le numéro dicté est : 01 45 65 79 98. Est-ce bien cela ?
– **Utilisateur :** Oui.
– **Système :** Merci d'avoir utilisé DictaNum. À bientôt.

La Figure 4 illustre les opérations effectuées par le client et le *Scheduler* durant le dernier dialogue. Par ailleurs, il est possible d'interrompre le système durant le *feedback* final. Pour ce faire, le service renvoie le message de *feedback* sous la forme : *Le numéro dicté est : 01 <sep> 45 <sep> 65 <sep> 79 <sep> 98. Est-ce bien cela ?*. La balise *<sep>* joue le rôle de séparateur entre les différents fragments qui sont envoyés à la suite à la TTS. Une dictée peut se terminer ainsi :

– **Système :** Le numéro dicté est : 01 45 67...
– **Utilisateur :** Non, 65.
– **Système :** Pardon. Le numéro dicté est : 01 45 65 79 98. Est-ce bien cela ?
– **Utilisateur :** Oui.
– **Système :** Merci d'avoir utilisé DictaNum. À bientôt.

Tour	Micro-tour utilisateur	Action client	Action Scheduler	Commentaire
T^1	$\mu T_1^{1,U}$	requête initiale	requête initiale	Requête initiale
	$\mu T_2^{1,U}$	envoi(‘01’)	envoi(‘01’)	Obtention réponse à ‘01’
	$\mu T_3^{1,U}$	envoi(‘01 45’)	*rollback* + envoi(‘01 45’)	Obtention réponse à ‘01 45’
	$\mu T_4^{1,U}$	envoi(‘silence’)	*commit*	*Commit*
				Réponse ‘01 45’
T^2	$\mu T_1^{2,U}$	envoi(‘67’)	envoi(‘67’)	Obtention réponse à ‘67’
	$\mu T_2^{2,U}$	envoi(‘67 79’)	*rollback* + envoi(‘67 79’)	Obtention réponse à ‘67 79’
	$\mu T_3^{2,U}$	envoi(‘silence’)	*commit*	*Commit*
				Réponse ‘67 79’
T^3	$\mu T_1^{3,U}$	envoi(‘Non’)	envoi(‘Non’)	Obtention réponse à ‘Non’ (Ici, ‘Désolé’)
	$\mu T_2^{3,U}$	envoi(‘Non, 65’)	*rollback* + envoi(‘Non, 65’)	Obtention réponse à ‘Non, 65’
	$\mu T_3^{3,U}$	envoi(‘Non, 65 79’)	*rollback* + envoi(‘Non, 65 79’)	Obtention réponse à ‘Non, 65 79’
	$\mu T_4^{3,U}$	envoi(‘silence’)	*commit*	Correction et *commit*
				Réponse ‘Désolé, 65 79’
T^4	$\mu T_1^{4,U}$	envoi(‘98’)	envoi(‘98’)	Obtention réponse à ‘98’
	$\mu T_2^{4,U}$	envoi(‘silence’)	*commit*	*Commit*
				Réponse ‘98’
T^5	$\mu T_1^{5,U}$	envoi(‘longSilence’)	envoi(‘longSilence’)	Détection fin numéro
				Demande confirmation numéro
T^6	$\mu T_1^{6,U}$	envoi(‘Oui’)	envoi(‘Oui’)	Obtention réponse à ‘Oui’
	$\mu T_2^{6,U}$	envoi(‘silence’)	*commit*	*Commit*
				Message fin

FIGURE 4 – Actions effectuées par le client et le *Scheduler* dans l’exemple d’interaction avec DictaNum

Au moment de l’interruption, le message envoyé au service se présente sous la forme {*texte déjà énoncée par la TTS* | *contenu du barge-in*}. Dans le cas du dernier exemple, le service reçoit {*Le numéro dicté est : 01 45 67* | *Non, 65*} ce qui lui permet d’effectuer la correction (ou pas si l’interruption n’est qu’une confirmation par exemple).

5 Discussion

La Figure 5 présente une analyse des fonctionnalités accessibles suivant le degré d’intégration du caractère incrémental dans un système. La classification présentée en Section 2 est reprise ici. Les fonctionnalités abordées concernent les systèmes de dialogue incrémentaux et donc les systèmes de la première catégorie ne proposent aucune d’elles. À l’opposé, ces fonctionnalités ont déjà été implémentées dans des systèmes appartenant au dernier groupe.

La stratégie *List of Availabilities* de NASTIA (El Asri *et al.*, 2014) a recours à deux fonctionnalités liées du système : la possibilité de s’interrompre après avoir analysé l’entrée et le fait de pouvoir relier l’instant d’interruption de l’utilisateur au signal sonore joué par la TTS. La capacité à détecter les interruptions de l’utilisateur existe déjà dans certains systèmes traditionnels (Laroche *et al.*, 2011) mais le moment et le sens de l’intervention ne sont pas pas mis en relation avec l’énoncé du système. De plus, celui-ci ne peut pas choisir de continuer à parler en ignorant certaines interruptions. En revanche, NASTIA est conçu pour effectuer une énumération que l’utilisateur peut interrompre puis réagir en fonction du contenu de son intervention et du moment auquel elle est faite. L’utilisateur communique avec le service à l’aide d’une plate-forme vocale qui interrompt la synthèse vocale si l’utilisateur commence à parler. Pendant que la TTS est en train de prononcer une alternative, l’utilisateur peut se taire ou intervenir. Dans le premier cas, le système est relancé automatiquement pour fournir l’alternative suivante car son timeout (temps au bout duquel le système est relancé en absence de réponse de l’utilisateur) a été réglé à une valeur très proche du temps mis pour énoncer une alternative. Dans le second cas, la TTS s’interrompt et le résultat de l’ASR ainsi que le moment de l’intervention sont transmis au service qui se charge de les analyser et de les traiter. Par conséquent, deux fonctionnalités propres aux systèmes incrémentaux peuvent être reproduites dans un système de dialogue non incrémental à condition qu’il reste à l’écoute de l’utilisateur (même si la TTS est en train de jouer un message) et qu’il capte le contenu de l’intervention et le moment auquel elle a été faite. Ces conditions sont vérifiées pour les systèmes de catégorie 3 ce qui leur donne également accès à ces deux fonctionnalités.

Fonctionnalité	Catégorie 1	Catégorie 2	Catégorie 3	Catégorie 4
Le système décide de s'interrompre après analyse de l'entrée	-	+	+	+
Relier l'instant d'interruption à la TTS	-	+	+	+
Le système interrompt l'utilisateur	-	-	+	+
Réactivité améliorée	-	-	+	+
Coût de traitement optimisé	-	-	-	+

FIGURE 5 – Analyse des fonctionnalités accessibles suivant le degré d'intégration du comportement incrémental

À l'inverse, il peut être intéressant pour le système d'interrompre l'utilisateur. Par exemple, en cas de désalignement, cela ne sert à rien de laisser l'utilisateur finir sa requête et le système peut directement lui demander de la reformuler. Les *feedbacks* générés par DictaNum constituent un autre cas d'utilisation, bien que les interruptions soient moins marquées car elles interviennent après un micro-silence. Cependant, dans un système utilisant un *Scheduler*, rien n'empêche le service d'intervenir de manière spontanée à n'importe quel micro-tour utilisateur. En revanche, les systèmes de la catégorie 2 ne permettent de soumettre une requête au service qu'à la fin de l'énoncé de l'utilisateur. Ils ne peuvent donc pas interrompre l'utilisateur. Notons qu'après une interruption du système, il faut arrêter d'écouter l'utilisateur pendant quelques micro-tours afin de ne pas considérer la fin de sa phrase (le temps qu'il se rende compte que le système l'a interrompu) comme un *barge-in* de sa part.

Un des avantages majeurs du dialogue incrémental réside dans l'amélioration de la réactivité du système. Les systèmes de type 1 et 2 attendent la fin de la requête de l'utilisateur (marquée par un silence) avant de la traiter. En revanche les systèmes des catégories 3 et 4 traitent chaque nouvel incrément d'information dès son arrivée, ainsi, avant la détection du silence, la réponse du système est déjà disponible. Cela permet de l'envoyer au client à l'instant même où le seuil de silence est dépassé (retour chariot pour les services textuels). Néanmoins, les systèmes du groupe 3 ont un fonctionnement du type *restart incremental* et de ce fait, à chaque nouvel incrément d'information injecté par l'utilisateur, la requête (incomplète) est traitée intégralement depuis le début après avoir effectué un *rollback*. En revanche, les architectures de type 4 traitent les nouveaux incréments comme compléments des informations déjà disponibles. Cela permet de n'effectuer que le traitement nécessaire pour intégrer le nouvel incrément à son arrivée, optimisant ainsi les coûts de traitement. Remarquons au passage qu'une nouvelle unité incrémentale peut modifier une partie ou toute la requête, néanmoins, les systèmes du groupe 4 sont capables de gérer une telle situation en annulant les traitement qui ne sont plus en accord avec la nouvelle requête. Ce mécanisme est désigné par le terme *revoke* dans (Schlangen & Skantze, 2011). Enfin, sur nos deux exemples d'implémentation, les délais de réponse du service sont très courts. Par conséquent, il n'est pas utile d'optimiser les délais de traitement si le service répond rapidement et qu'il ne doit pas effectuer de tâches qui créent un délai (accès lent à des bases de données par exemple).

Notre solution permet d'éviter de modifier le service de dialogue au niveau fonctionnel (mis à part l'ajout du contexte de simulation). Cependant, comme c'est le cas pour DictaNum, certaines modifications au niveau applicatif peuvent être indispensables. Le rôle du *Scheduler* n'est pas de générer lui même des messages ou de faire de la gestion de dialogue au sens classique. Ainsi, quand le passage en incrémental nécessite de nouveaux types de messages tels que les *feedbacks* à l'échelle du micro-tour, ou encore la prise en compte des corrections (*Non, 65...*), il faut les implémenter dans le service.

Enfin, afin de pouvoir prendre des décisions de prise de parole optimales, le *Scheduler* pourrait avoir besoin de certaines informations provenant de modules plus en aval dans la chaîne de dialogue (au niveau du service). Encore une fois, cela doit être géré au niveau applicatif. Un futur papier, dédié à l'implémentation des systèmes de dialogue en utilisant le *Scheduler*, traitera les idées brièvement décrites dans les deux derniers paragraphes.

6 Conclusion et travaux futurs

En partant d'un système de dialogue non-incrémental, ce papier montre qu'il est possible d'en rendre le fonctionnement incrémental avec peu de modifications. Le *Scheduler*, un module intermédiaire entre le client et le service de dialogue permet de simuler un comportement incrémental vu du client tout en laissant le service pratiquement inchangé. Au fur et à mesure que l'utilisateur parle, des incréments de requête sont formés et le *Scheduler* détermine les réponses correspondantes en les envoyant au service. Au bout d'un moment, le *Scheduler* prend la décision d'engager le dialogue dans le sens de la dernière hypothèse de requête en cours en envoyant un signal de *commit* au système. Ce dernier peut ainsi se baser sur cette hypothèse pour agir sur le monde extérieur (formulation d'une réponse, modification d'une base de données...).

Traditionnellement, le service maintient un contexte de dialogue constitué de toutes les informations relatives au déroulement du dialogue actuel. Dans le cadre de notre solution, il faudra maintenir un second contexte qui sera modifié lors des requêtes non validées par un *commit*. La plupart des requêtes adressées au service servent à voir ce qu'il aurait répondu dans le cas où l'hypothèse actuelle est la bonne et on ne veut pas que ces requêtes modifient le contexte principal.

Comme application de la solution proposée ici, deux services de dialogue déjà existants ont été transformés en systèmes incrémentaux. Le premier est un service en mode texte dont l'objectif est d'aider l'utilisateur à naviguer dans une base de contenus (la FAQ de l'ANPE en ce qui nous concerne). Le second, en mode vocal, invite l'utilisateur à dicter un numéro.

Concevoir un système de dialogue incrémental capable de décider quand réaliser un *commit* et prendre la parole n'est pas facile à concevoir de manière experte. Par la suite, nous envisageons d'explorer des approches basées sur l'apprentissage statistique pour optimiser en ligne ces comportements et ainsi atteindre cet objectif.

Références

ALLEN J., FERGUSON G. & STENT A. (2001). An architecture for more realistic conversational systems. In *6th international conference on Intelligent user interfaces*.

BAUMANN T. & SCHLANGEN D. (2013). Open-ended, extensible system utterances are preferred, even if they require filled pauses. In *Proceedings of the SIGDIAL 2013 Conference*.

BRESLIN C., GASIC M., HENDERSON M., KIM D., SZUMMER M., THOMSON B., TSIAKOULIS P. & YOUNG S. (2013). Continuous asr for flexible incremental dialogue. In *ICASSP*, p. 8362–8366.

BROWN-SCHMIDT S. & HANNA J. E. (2011). Talking in another person's shoes : Incremental perspective-taking in language processing. *Dialogue and Discourse*, **2**, 11–33.

CLARK H. H. (1996). *Using Language*. Cambridge University Press.

DEVAULT D., SAGAE K. & TRAUM D. (2011). Incremental interpretation and prediction of utterance meaning for interactive dialogue. *Dialogue and Discourse*, **2**, 143–170.

DOHSAKA K. & SHIMAZU A. (1997). A system architecture for spoken utterance production in collaborative dialogue. In *IJCAI*.

EDLUND J., GUSTAFSON J., HELDNER M. & HJALMARSSON A. (2008). Towards human-like spoken dialogue systems. *Speech Communication*, **50**, 630–645.

EL ASRI L., LEMONNIER R., LAROCHE R., PIETQUIN O. & KHOUZAIMI H. (2014). NASTIA : Negotiating Appointment Setting Interface. In *Proceedings of LREC*.

FINK G. A., SCHILLO C., KUMMERT F. & SAGERER G. (1998). Incremental speech recognition for multimodal interfaces. In *IECON*.

HASTIE H., AUFAURE M.-A. *et al.* (2013). Demonstration of the parlance system : a data-driven incremental, spoken dialogue system for interactive search. In *Proceedings of the SIGDIAL 2013 Conference*.

ILKIN Z. & STURT P. (2011). Active prediction of syntactic information during sentence processing. *Dialogue and Discourse*, **2**, 35–58.

LANGKILDE I., WALKER M. A., WRIGHT J., GORIN A. & LITMAN D. (1999). Automatic prediction of problematic human-computer dialogues in how may i help you ? In *ASRU99*.

LAROCHE R. (2010). *Raisonnement sur les incertitudes et apprentissage pour les systemes de dialogue conventionnels*. PhD thesis, Paris VI University.

LAROCHE R. & PUTOIS G. (2010). *D5.5 : Advanced Appointment-Scheduling System "System 4"*. Prototype D5.5, CLASSIC Project.

LAROCHE R., PUTOIS G. *et al.* (2011). *D6.4 : Final evaluation of CLASSiC TownInfo and Appointment Scheduling systems*. Report D6.4, CLASSIC Project.

LEVELT W. J. M. (1989). *Speaking : From Intention to Articulation*. Cambridge, MA : MIT Press.

LOCK K. (1965). Structuring programs for multiprogram time-sharing on-line applications. In *AFIPS '65 (Fall, part I) Proceedings of the November 30–December 1, 1965, fall joint computer conference, part I*.

MATSUYAMA K., KOMATANI K., OGATA T. & OKUNO H. G. (2009). Enabling a user to specify an item at any time during system enumeration – item identification for barge-in-able conversational dialogue systems –. In *Proceedings of the INTERSPEECH 2009 Conference*.

MATTHIAS G. M. (2008). Incremental speech understanding in a multimodal web-based spoken dialogue system. Master's thesis, Massachusetts Institute of Technology.

RAUX A. & ESKENAZI M. (2008). Optimizing endpointing thresholds using dialogue features in a spoken dialogue system. In *SIGDIAL*.

SCHLANGEN D. & SKANTZE G. (2011). A general, abstract model of incremental dialogue processing. *Dialogue and Discourse*, **2**, 83–111.

SELFRIDGE E., ARIZMENDI I., HEEMAN P. & WILLIAMS J. (2013). Continuously predicting and processing barge-in during a live spoken dialogue task. In *Proceedings of the SIGDIAL 2013 Conference*.

SELFRIDGE E. O., ARIZMENDI I., HEEMAN P. A. & WILLIAMS J. D. (2012). Integrating incremental speech recognition and pomdp-based dialogue systems. In *Proceedings of the 13th Annual Meeting of the Special Interest Group on Discourse and Dialogue*.

SKANTZE G. & SCHLANGEN D. (2009). Incremental dialogue processing in a micro-domain. In *ACL*.

TANENHAUS M. K., SPIVEY-KNOWLTON M. J., EBERHARD K. M. & SEDIVY J. C. (1995). Integration of visual and linguistic information in spoken language comprehension. *Science*, **268**, 1632–1634.

WIRÉN M. (1992). *Studies in Incremental Natural Language Analysis*. PhD thesis, Linköping University, Linköping, Sweden.

WLODARCZAK M. & WAGNER P. (2013). Effects of talk-spurt silence boundary thresholds on distribution of gaps and overlaps. In *INTERSPEECH Proceedings*.