An easy method to make dialogue systems incremental

Hatim KHOUZAIMI
Orange Labs
Laboratoire Informatique d'Avignon
hatim.khouzaimi@orange.com

Romain LAROCHE
Orange Labs,
Issy-les-Moulineaux,
France
romain.laroche@orange.com

Fabrice LEFEVRE
Laboratoire Informatique d'Avignon,
Avignon, France
fabrice.lefevre@univ-avignon.fr

Abstract

Incrementality as a way of managing the interactions between a dialogue system and its users has been shown to have concrete advantages over the traditional turn-taking frame. Incremental systems are more reactive, more human-like, offer a better user experience and allow the user to correct errors faster, hence avoiding desynchronisations. Several incremental models have been proposed, however, their core underlying architecture is different from the classical dialogue systems. As a result, they have to be implemented from scratch. In this paper, we propose a method to transform traditional dialogue systems into incremental ones. A new module, called *the Scheduler* is inserted between the client and the service so that from the client's point of view, the system behaves incrementally, even though the service does not.

1 Introduction

An incremental compiler (Lock, 1965) processes each instruction irrespectively from the others so that local modifications of the source code do not affect the global result. This idea of incrementality has been adapted to the field of natural language analysis (Wirén, 1992): instead of feeding modules with full utterances, the input signal is delivered and processed chunk by chunk (word by word for example) and each new piece engenders a new output hypothesis.

Human beings behave similarly when interacting with each other (Levelt, 1989; Clark, 1996). They understand each other gradually when they speak, they can interrupt each other and the listener is able to predict the end of an utterance before it is fully pronounced by the speaker (Tanenhaus et al., 1995; Brown-Schmidt and Hanna, 2011; DeVault et al., 2011). Reading is also a task that we perform incrementally (Ilkin and Sturt, 2011).

Traditional dialogue systems[1] work in a turn-taking manner. The user pronounces his request and after a silence is detected, the systems starts processing the utterance and planning an answer. Some systems can even allow the user to barge in on them, however, they do not take the timing of the interruption into account nor try to link it with the system's utterance. On the other hand, incremental dialogue systems process the user's request chunk by chunk as the latter is divided in several *incremental units* (IU) (Schlangen and Skantze, 2011). They keep a hypothetical user request that evolves as new IUs arrive as input. The response to this hypothesis can be used to make live feedback to the user using voice or other modalities if available. As opposed to traditional systems, when the user interrupts the system, the content and the timing of its utterance are taken into account (Matsuyama et al., 2009; Selfridge et al., 2013) to determine how to act on it. Therefore, incremental systems have been shown to be more reactive, to offer a more human-like experience (Edlund et al., 2008) and to correct errors faster hence achieving better results in terms of user experience (Skantze and Schlangen, 2009; Baumann and Schlangen, 2013; El Asri et al., 2014) and task completion (Matthias, 2008; El Asri et al., 2014).

Many incremental architectures have already been proposed. Nevertheless, designing systems based on them requires an implementation from scratch as they are fundamentally different from traditional dialogue systems. The objective of this paper is to propose a method of transforming a traditional system into an incremental one at minimal cost. A new module called *the Scheduler* is inserted between the client and the service so that

[1] We will use the expression *traditional dialogue systems* to talk about non incremental ones.

Proceedings of the SIGDIAL 2014 Conference, pages 98–107,
Philadelphia, U.S.A., 18-20 June 2014.

from the client's point of view, the system behaves incrementally, even though the service works in a traditional way.

Section 2 draws a state-of-the-art concerning incremental dialogue systems. The architecture proposed here and the role of the Scheduler are presented in Section 3. In Section 4, two implementations of our method are presented: CFAsT and DictaNum. Then, a discussion is held in Section 5 before concluding the paper and presenting our next objectives in Section 6.

2 Related work

Dialogue systems can be split into four groups according to how they integrate incrementality in their behaviour. Traditional dialogue systems (Laroche et al., 2011) form the first category whereas the second one refers to systems that propose some incremental strategies among traditional others (El Asri et al., 2014). The architecture presented in this paper belongs to the third group which contains incremental systems based on a traditional inner behaviour (Hastie et al., 2013; Selfridge et al., 2012). The fourth category contains incremental systems where internal modules work incrementally (Dohsaka and Shimazu, 1997; Allen et al., 2001; Schlangen and Skantze, 2011). Figure 1 discussed later provides a list of the features that are available in each category.

Several dialogue strategies have been implemented in NASTIA (El Asri et al., 2014), a dialogue system helping the user to find a date and a time for an appointment with a technician (completing the work made during the European project CLASSiC (Laroche and Putois, 2010)). Among them, List of Availabilities is an incremental strategy where the system enumerates a list of alternatives for the appointment. The user is supposed to interrupt this enumeration when he hears an option that is convenient for him. An experiment showed that List of Availabilities produced better results than other traditional strategies in terms of task completion and user satisfaction.

PARLANCE (Hastie et al., 2013) is an example of a third category system (it was developed in the European project PARLANCE). Its architecture is similar to the traditional ones but it integrates a new module, called MIM (Micro-turn Interaction Manager), which decides when the system should speak, listen to the user and when it should generate back-channels. The closest approach to the method introduced in this paper is presented in (Selfridge et al., 2012) : the IIM (Incremental Interaction Manager) is an intermediate module between an incremental ASR and a TTS on the one hand and the service on the other hand. Instead of replicating the dialogue context as it is suggested in this paper, different instances of the service are run. Moreover, the IIM is introduced as preliminary work in order to simulate incremental dialogue whereas in this paper, the Scheduler approach is fully studied and placed into the context of the current state-of-the-art concerning incremental dialogue. It is also viewed as a new layer that can be extended later on, into a smart turn-taking manager.

The architecture proposed in (Dohsaka and Shimazu, 1997) contains eight modules that work in parallel: the Speech Recognizer, the Response Analyzer, the Dialogue Controller, the Problem Solver, the Utterance Planner, the Utterance Controller, the Speech Synthesizer and the Pause Monitor. The user asks the system to solve a problem. Then, his request is submitted incrementally to the Speech Recognizer which sends its output text to the Response Analyzer that figures out concepts to be sent to the Dialogue Controller. The latter interacts with the Problem Solver and the Utterance Planner in order to compute a solution that is communicated to the user through the Utterance Controller then the Speech Synthesizer. This system belongs to the fourth category as all its modules behave incrementally in order to start suggesting a solution to the user's problem before it is totally computed. In the same category, (Allen et al., 2001) proposes another architectures split in three main modules: the Interpretation Manager, the Behavioral Agent and the Generation Manager. The first module catches the user's request and broadcasts it incrementally inside the system. The second one manages the system's action plan and the third is in charge of the response delivery.

A general and abstract model is introduced in (Schlangen and Skantze, 2011). A dialogue system can be viewed as a chain of modules. Each module has a Left Buffer (LB) where its inputs are pushed, an Internal State (IS) and a Right Buffer (RB) where it makes its outputs available. Data (audio, text, concepts...) flows through these modules in the form of Incremental Units (IU). When an IU is put in the LB of a module, it can be pro-

cessed immediately hence modifying its RB. For example, every 500 ms, a new IU in the form of a chunk of audio signal can be put into the LB of the ASR which can modify its output according to what the user said during this time window. All dialogue systems from the four categories can be viewed as instances of this general model: *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*.

In this paper, we introduce an architecture that belongs to the third category. In comparison with the first two categories, these systems behave incrementally during the whole dialogue. On the other hand, they can be built at a lower cost than the systems from the fourth category.

3 Architecture

Traditional dialogue systems are generally composed of a client on the user's terminal and a service that is deployed on a remote machine. They work in a turn-taking manner as when the user speaks, the system waits until the end of his request before processing it and vice versa (except for some systems where the user can interrupt the system). To make such a system incremental, we suggest inserting a new module between the client and the service: the Scheduler (this denomination is taken from (Laroche, 2010)). This new architecture can be cast as an instance of the general abstract model of (Schlangen and Skantze, 2011). The client, the Scheduler and the service are the three modules that compose the system. The first two ones are incremental but the last one is not. We will not use the notions of LB and RB and will consider that these modules interact with each other through some channel (network in the case of our implementation, see Section 4).

3.1 The traditional architecture

In a traditional architecture, the client receives a stream of data (audio signal, string...). If it is not the case (a web interface where each button represents a request for example), it does not make sense to transform such a system in an incremental one, so they are out of the scope of this paper. The end of a request is determined by a condition $EndTurnCond$. It can be a long enough silence (Raux and Eskenazi, 2008; Wlodarczak and Wagner, 2013) in the case of vocal services or a carriage return for text systems. A *dialogue turn* is the time interval during which the user sends a request to the system and gets a response. These turns will be called T^1, T^2, ..., T^k... and each one of them can be split into a *user turn* $T^{k,U}$ and a *system turn* $T^{k,S}$: $T^k = T^{k,U} \cup T^{k,S}$. During the user turn, a request Req^k is sent and during the system turn, the corresponding response $Resp^k$ is received. The instant when a condition goes from $false$ to $true$ will be called its *activation time*. As a consequence, $T^{k,U}$ ends at the activation time of $EndTurnCond$ and $T^{k,S}$ is finished when the system gives the floor to the user.

The service is made up of three parts: the internal interface, the internal context and the external interface. The internal interface manages the interactions between the service and the client. The internal context handles the way the client's requests should be acted on and the external interface is in charge of the interactions with the external world (database, remote device...).

3.2 Incrementality integration

The way the client sends the user's request to the service should be modified in order to make the system incremental. A new sending condition is defined: $EndMicroTurnCond$ and it is less restrictive than $EndTurnCond$ (which makes the latter imply the former). Therefore, the new client sends requests more frequently than the traditional one. A user micro-turn is the time interval between two activation times of $EndMicroTurnCond$ so the user turn $T^{k,U}$ can be divided into $n^{k,U}$ user micro-turns $\mu T_i^{k,U}$: $T^{k,U} = \bigcup_{i=1}^{n^{k,U}} \mu T_i^{k,U}$. We also define the p^{th} *sub-turn* of the user turn $T^{k,U}$ as: $T_p^{k,U} = \bigcup_{i=1}^{p} \mu T_i^{k,U}$. The union symbol is used as we concatenate time intervals. In general, $EndMicroTurnCond$ can be activated at a constant frequency or at each new input made by the user. Moreover, when $EndTurnCond$ is activated, the Scheduler is informed by the client thanks to a dedicated signal: *signal_ETC*. At each $T^{k,S}$, the user makes a new request but at the micro-turn $\mu T_i^{k,S}$ with $i < n^{k,U}$, the complete request is not available yet. Consequently, a temporary request which we will call *sub-request* (Req_i^k) is sent. Sending the whole request from the beginning of the turn at each micro-turn is called *restart incremental* processing (Schlangen and Skantze, 2011). Let us notice that if $i_1 < i_2$ then $Req_{i_1}^k$

is not necessarily a prefix of $Req_{i_2}^k$ (in spoken dialogue, a new input in the ASR can modify the whole or a big part of the output).

The Scheduler is an intermediate module between the client and the service whose aim is to make the combination {Scheduler + Service} behave incrementally from the client's point of view. We define $ServiceReqCond$ as the condition constraining the Scheduler to send a request to the system or not. At each user micro-turn $\mu T_i^{k,S}$, it receives a sub-request Req_i^k. If $ServiceReqCond$ is true, the latter is sent to the system and the corresponding response $Resp_i^k$ is stored so that the client can ask for it later. For example, $ServiceReqCond$ can be constantly true which makes the Scheduler send all the sub-requests that it receives or it can be activated only if the new sub-request is different from the previous one (if the client already behaves the same way through $EndMicroTurnCond$ it is redundant to do so in $ServiceReqCond$ too).

The end of a turn is determined by the Scheduler. This module decides when to validate the current sub-request and to no longer wait for new information to complete it. It engages the dialogue in the direction of this hypothesis as it is considered as the user's intent. The Scheduler is said to *commit* the sub-request (Schlangen and Skantze, 2011) (this notion is described in Section 3.3). We define $CommitCond$ as the condition for the Scheduler to commit a hypothesis. For example, in the case of a system that asks for a 10 digits phone number, $CommitCond = (length(num) == 10)$ where $length(num)$ is the number of digits in each sub-request. Hence, a user turn ends at the activation time of $CommitCond$ and not when a *signal_ETC* is received. However, $EndTurnCond$ implies $CommitCond$.

The client is made of two threads: the *sending thread* and the *recuperation thread*. The first one is in charge of sending sub-requests at each micro-turn and the second one gets the last response hypothesis available in the Scheduler. The recuperation thread is activated at the same frequency as micro-turns so that the client is always up to date. In the case of vocal services, it is the Scheduler's task to decide which intermediate responses should be pronounced by the system and which ones should be ignored. Therefore, a flag in the message must be set by this module to declare whether it has to be outputted or not. When the recuperation thread gets new messages from the Scheduler, it decides whether to send it to the Text-To-Speech module or not based on the value of this flag.

The service in our architecture is kept unchanged (apart from some changes at the applicative level, see Section 4.2). The only functional modification is that the context is duplicated: *the simulation context* (see Section 3.3) is added. When a new sub-request is received by the Scheduler and $ServiceReqCond$ is true, an incomplete request (sub-request) is sent to the service. Therefore, the system knows what would be the response of a sub-request if it has to be committed. As the service is not incremental and cannot process the request chunk by chunk, all the increments from the beginning of the turn have to be sent and that is what justifies the choice of the *restart incremental* mode.

The service can also order the Scheduler to commit. This behaviour is described in (Schlangen and Skantze, 2011) where the IUs in the RB of a module are *grounded in* the ones in the LB that generated them. Consequently, when a module decides to commit to an output IU, all the IUs that it is grounded in must be committed. In our architecture, when the service commits to the result of a request (if it already started delivering the response to the user for example), this request has to be committed by the Scheduler.

On the other hand, as we defined the user micro-turn, we can introduce the *system micro-turn*. In traditional systems, the service's response is played by the TTS during the system turn $T^{k,S}$. In incremental dialogue, this turn can be divided into n_S^k system micro-turns $\mu T_i^{k,S}$: $T^{k,S} = \bigcup_{i=1}^{n_S^k} \mu T_i^{k,S}$. Their duration depends on the way the service decides to chunk its response (for example, every item in an enumeration can be considered as a chunk). When the user interrupts the system, the timing of his interruption is given by the micro-turn during which he reacted. Moreover, when the user barges in, a new tour is started. Only vocal systems are concerned with this behaviour as textual systems cannot be interrupted (the whole service response is displayed instantly).

3.3 Commit, rollback and double context

The request hypothesis fluctuates as long as new increments are taken into account. However, at

some point, the system has to take an action that is based on the last hypothesis and visible by the user. For example, a response may be sent to the TTS or a database can be modified. At that point, the system is said to *commit* to its last hypothesis which means that it engages the dialogue according to its understanding of the request at that moment. It no longer waits for other incremental units to complete the request as it can no longer change it. On the contrary, the system can decide to forget its last hypothesis and come back to the state it was in at the moment of the last commit. This operation is called *rollback* (both terms are taken from the database terminology).

Most of the requests sent by the Scheduler to the service are aimed to know what would the latter respond if the current hypothesis contains all the information about the user's intent. Consequently, these requests should not modify the current context of the dialogue. We suggest that the service maintains two contexts: the *real context* and the *simulation context*. The first one plays the same role as the classical context whereas the second one is a buffer that can be modified by partial requests.

In our architecture, committing to a hypothesis will be made by copying the content of the simulation context (generated by the current request hypothesis) into the real context. On the opposite, a rollback is performed by copying the real context into the simulation one, hence going back to the state the system was in right after the last commit.

Every user micro-turn, the client sends to the Scheduler the whole user's sub-request since the last commit. This incomplete request is then sent to the service and the answer is stored in the Scheduler. If during the next micro-turn, the Scheduler does not ask for a commit but needs to send a new sub-request instead, a rollback signal is sent first as the system works in a restart incremental way (in this paper, rollbacks are only performed in this case). Figures A.1 and A.2 represent the way our three modules interact and how the double context is handled. In Figure A.1, the conditions $EndTurnCond$, $EndMicroTurnCond$, $ServiceReqCond$ and $CommitCond$ are written on the left of the streams they generate. On the left of the figure, the times where the sending thread of the client is active and inactive are represented and dashed arrows represent streams that are received by the recuperation thread. They are not synchronized with the rest of the streams, even though they are in this figure (for more clarity). Also, the commit decision has been taken by the Scheduler after it received a *signal_ETC* which is not always the case.

We call $ctxt(T^k)$ the real context at the end of T^k ($ctxt(T^0)$ being the initial context at the beginning of the dialogue). The context is not modified during the system turn, hence, we may notice that $ctxt(T^{k,U}) = ctxt(T^k)$. During the commit at the end of $T^{k,U}$, the simulated context is copied into the real context: $ctxt(T^k) = ctxt(T^{k-1} + T^{k,U}_{n^{k,U}})$.

4 Implementations

We implemented our method in the case of two dialogue systems developed at Orange Labs. The first one is a text service where the client is a web interface and the second one is a vocal service designed to record numbers. With only a few modifications, these two systems have been made incremental, showing that our solution is easy to implement, and demonstrating the incremental behaviour of the transformed systems, in the limit of the implemented strategies and according to the modalities that have been used (text and vocal modes).

4.1 CFAsT: Content Finder AssitanT

CFAsT is an application developed at Orange Labs and which can be used to generate textual dialogue systems and whose objective is to help the user search for some specific content in a database.

The client is a web page with a text-box where the user can type a request using natural language (validated by a carriage return or by clicking on the `validate` button). This page also contains buttons representing keywords or content suggestions. In this implementation, the content base chosen is the list of accepted papers at the NIPS 2013 conference. A list of keywords is maintained through the interaction. It is initially empty and for each new request, if new keywords are detected, they are added to the list. The interaction ends when the user selects a unique content.

In our implementation[2], the client is made of two screens. The first one contains the same view as the classical system and the second one fluctuates incrementally. When the user is typing a request, the second view is updated

[2]http://dialogue.orange-labs.fr/CFAsTIncrEng/

each time the space button is hit: it shows the answer that the service would have given in case of a carriage return at that moment. When a request is validated, a commit is performed and the two views are identical. In this case: $EndMicroTurnCond$ is activated when the space button is hit ($ServiceReqCond$ does not filtrate any sub-request) and $EndTurnCond$ is $true$ if a carriage return is detected. In addition, $CommitCond = EndTurnCond$.

4.2 DictaNum

The second implementation is a vocal system called DictaNum and inspired by NUMBERS dialogue system (Skantze and Schlangen, 2009). It asks the user to dictate a number and then gives a feedback to confirm that it has been well understood. The client is an HTML web page that interacts with the Scheduler using AJAX[3]. It uses the Google web API for both the ASR and TTS tasks. Other existing systems ask the user to dictate a number (telephone, credit card...) such as *How may I help you?* (Langkilde et al., 1999). Here is an example of an interaction with the traditional version of DictaNum:

System: Welcome to the DictaNum service. Please dictate your number.
User: 01 45 65 79 98.
System: The dictated number is: 01 45 67 79 98. Is that correct?
User: No.
System: Sorry. Let us try again. Please dictate your number.
User: 01 45 65 79 98.
System: The dictated number is: 01 47 65 79 98. Is that correct?
User: No.
System: Sorry. Let us try again. Please dictate your number.
[...]
System: The dictated number is: 01 45 65 79 98. Is that correct?
User: Yes.
System: Thank you for using DictaNum. Bye.

If the user's request has not been understood correctly in one shot, he has to dictate it again. This is not an optimal strategy. DictaNum has been modified to support chunk by chunk dictation as in the following example:

[3]http://dialogue.orange-labs.fr/DictaNumEng/

System: Welcome to the DictaNum service. Please dictate your number.
User: 01 45
System: 01 45
User: 65 79
System: 67 79
User: No, 65 79
System: Sorry, 65 79
User: 98
System: 98
User: ...
System: The dictated number is: 01 45 65 79 98. Is that correct?
User: Yes.
System: Thank you for using DictaNum. Bye.

In order to generate feedbacks, the service has been modified at the applicative level. It maintains a string that is supposed to contain the complete number at the end of the interaction. Initially, this string is empty. When a silence is detected after a chunk is dictated, the system repeats the latter as a feedback to the user and concatenates it at the end of the number string. If the user begins a new chunk with *No*, the system understands that it has to correct the last one (starting the next feedback with *Sorry...*), otherwise, it keeps it and moves forward in the dictation. Finally, if after a feedback a silence is detected with nothing dictated, the system understands that the dictation is over and makes a general feedback over the whole number.

These modifications are not enough for the system to be used in an incremental way. It is not optimal for the user to insert silences in his dictation. Of course, he can, but it is not convenient nor natural. The client has been modified so that it no longer waits for a silence to send the user's request, instead, it sends a partial request every 500 ms ($EndMicroTurnCond$). The partial request is sent on a restart incremental mode.

Also, DictaNum can detect silences in a microturn level. We call Δ_s the silence threshold used to determine the end of a request in the traditional system and we introduce a new threshold δ_s such as $\delta_s \leq \Delta_s$. A silence whose duration is greater than δ_s is called *micro-silence*. The system has been modified in order to detect these shorter silences during the dictation, to commit ($EndTurnCond = CommitCond$) and deliver a feedback right after. Additionally, our system's

response time is very short, the feedback message is available before the end of the micro-silence, so it is fed to the TTS without any delay. If $\delta_s = \Delta_s$, it is more convenient to dictate the number in one shot. Therefore, moving δ_s between zero and Δ_s creates a continuum between traditional systems and incremental ones. One may argue that these modifications are enough and no incremental behaviour is required, but the response delay will be higher, hence, the user will not wait for any feedback and will try to dictate his number in one shot.

If the user manifests a silence that is longer than Δ_s right after a feedback, the dictation ends and a general feedback is made to confirm the whole number. In our system, silences are determined by the number micro-turns during which there is no new input from the ASR but we could have used the VAD (Voice Activity Detection) (Breslin et al., 2013).

We set $EndMicroTurnCond$ to be activated by a 2 Hz clock and at every micro-turn, the Scheduler checks whether the new request is different from the previous one ($ServiceReqCond$). If that is the case, a rollback signal is sent followed by all the digits in the current number fragment. When a micro-silence is detected, a string *silence* is sent to the Scheduler (as *signal_ETC*) and that is when the Scheduler decides to commit. The recuperation thread requests the last message from the service with the same frequency as micro-turns, so when $CommitCond$ is activated, the feedback is already available and is delivered instantly to the TTS.

Finally, it is also possible for the user to interrupt the system during the final feedback. To do so, the service sends a feedback message in the following format: *The dictated number is: 01 <sep> 45 <sep> 65 <sep> 79 <sep> 98. Is that correct?*. The *<sep>* is a separator that is used to delimit the system micro-turns $\mu T_i^{k,S}$. They are pronounced one after another by the TTS. As a result, a dictation may end like this:

System: The dictated number is: 01 45 67 ...
User: No, 65.
System: Sorry. The dictated number is: 01 45 65 79 98. Is that correct?
User: Yes.
System: Thank you for using DictaNum. Bye.

After the interruption, a message sent to the service under the following format: {*part of the request that has been pronounced so far | barge-in content*}. In our example, this message is {*The dictated number is: 01 45 67 | No, 65*} which makes the service know how to perform the correction (or not, if the interruption is just a confirmation for example).

5 Discussion

Incremental dialogue systems present new features compared to traditional ones. In this section, we analyse the abilities of these systems given the way they integrate incrementality. To do so, we classify them as suggested in Section 2. Figure 1 summarizes the features discussed. These features are specific to incremental dialogue systems, so they do not exist in the first category. On the contrary, they have all been implemented in systems from the fourth category.

To interact with the NASTIA service, the user has to call a vocal platform which handles the ASR and TTS tasks. It has been configured in order to interrupt the TTS when activity is detected in the ASR. When using the List of Availabilities strategy, each item during an enumeration is a dialogue turn where the timeout duration is set to a low value (time to declare that the user did not answer) so that if he does not barge-in, the system moves to the next item of the list. If the user speaks, the TTS is stopped by the vocal platform and the user's utterance and its timing are communicated to the service. The latter can ignore the barge-in (if the user says *No* for example) or select an item in the list according to this input. Some traditional systems allow the user to interrupt them but they do not take the content of the utterance into account nor its timing (in order to make the link with the utterance of the TTS). Hence, these two features can be implemented in a dialogue system provided that it is permanently listening to the user and that it catches his utterance and its timing. These conditions are true for systems from the third category which make it possible for them to integrate these features.

Incremental dialogue systems can sometimes detect desynchronisations before the user has finished his utterance. Therefore, the dialogue would take less time if the system can interrupt the user asking him to repeat his request. Feedbacks are also a form of interrupt as it is the case for DictaNum because they are uttered after a short si-

Features	Category 1	Category 2	Category 3	Category 4
TTS interruption after input analysis	-	+	+	+
Link interruption time with TTS	-	+	+	+
User interruption by the system	-	-	+	+
Better reactivity	-	-	+	+
Optimal processing cost	-	-	-	+

Figure 1: Available features for dialogue systems given the way they integrate incrementality

lence (micro-silence). These features can only be implemented in systems from the third and the fourth group, as for the the first two ones, the system is only requested at the end of a user's utterance.

As far as reactivity is concerned, systems from the third and the fourth category process the user's request every time that a new increment is pushed into the system. Therefore, when the end of the request is detected (long enough silence), the service's response is already ready and can be delivered immediately. On the other hand, systems from group 1 and 2 wait until the end of the user's utterance to send the request to the service, hence, being less reactive. However, systems from the third group work on a restart incremental, reprocessing the whole request at each new increment. On the contrary, systems from the fourth category can process the request increment by increment hence optimizing the processing cost. Sometimes, a new increment can modify the whole request (or a part of it) and those systems are designed to handle this too by canceling some previous processing (*revoke* mechanism (Schlangen and Skantze, 2011)). While integrating incrementality in CFAsT and DictaNum, we noticed that the system responded so quickly that no efforts are necessary to optimise the processing time. However, systems from the fourth group can make the difference if the system needs to process tasks that create a delay (slow access to a remote database for example).

In our method, the service is not modified in a functional level (except from the double context management). However, as it is the case for DictaNum, some modifications at the applicative level might be compulsory. The Scheduler is not supposed to generate messages by himself or to perform traditional dialogue management tasks. As a consequence, when one needs to add some new feedback messages at the micro-turn level or the possibility to correct an utterance, these features must be implemented in the service.

Finally, in order for the Scheduler to decide when to commit and when to take the floor in an optimal way, it might need information coming from the back-end modules. Once again, this should be handled in the applicative level. A future paper, focused on how to implement systems using the Scheduler, will cover the ideas briefly described in the last two paragraphs.

6 Conclusion and future work

This paper describes a method for transforming a traditional dialogue system into an incremental one. The Scheduler is an intermediate module that is inserted between the client and the service. From the client's point of view, the system's behaviour is incremental despite the fact that the service works in a traditional turn-taking manner. Most requests that are sent by the Scheduler to the service are aimed to see what would be the answer if the current request hypothesis is the final one. In this case, the service's context should not be modified. Therefore, two context have to be maintained: the real context and the simulated one.

This solution has been implemented in the case of a textual dialogue system generated by the CFAsT application. It helps the user navigate through the NIPS 2013 proceedings titles. It has also been used to make a vocal system incremental: DictaNum. This service asks the users to dictate a number and confirms that it has been well understood.

In the future, we will explore how to make the Scheduler learn when to commit the current request hypothesis and when to take the floor. We will use reinforcement learning to figure out the optimal strategies.

References

James Allen, George Ferguson, and Amanda Stent. 2001. An architecture for more realistic conversational systems. In *6th international conference on Intelligent user interfaces*.

Timo Baumann and David Schlangen. 2013. Open-ended, extensible system utterances are preferred, even if they require filled pauses. In *Proceedings of the SIGDIAL 2013 Conference*.

Catherine Breslin, Milica Gasic, Matthew Henderson, Dongho Kim, Martin Szummer, Blaise Thomson, Pirros Tsiakoulis, and Steve Young. 2013. Continuous asr for flexible incremental dialogue. In *ICASSP*, pages 8362–8366.

Sarah Brown-Schmidt and Joy E. Hanna. 2011. Talking in another person's shoes: Incremental perspective-taking in language processing. *Dialogue and Discourse*, 2:11–33.

Herbert H. Clark. 1996. *Using Language*. Cambridge University Press.

David DeVault, Kenji Sagae, and David Traum. 2011. Incremental interpretation and prediction of utterance meaning for interactive dialogue. *Dialogue and Discourse*, 2:143–170.

Kohji Dohsaka and Akira Shimazu. 1997. A system architecture for spoken utterance production in collaborative dialogue. In *IJCAI*.

Jens Edlund, Joakim Gustafson, Mattias Heldner, and Anna Hjalmarsson. 2008. Towards human-like spoken dialogue systems. *Speech Communication*, 50:630–645.

Layla El Asri, Remi Lemonnier, Romain Laroche, Olivier Pietquin, and Hatim Khouzaimi. 2014. NASTIA: Negotiating Appointment Setting Interface. In *Proceedings of LREC*.

Helen Hastie, Marie-Aude Aufaure, 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*.

Zeynep Ilkin and Patrick Sturt. 2011. Active prediction of syntactic information during sentence processing. *Dialogue and Discourse*, 2:35–58.

Irene Langkilde, Marilyn Anne Walker, Jerry Wright, Allen Gorin, and Diane Litman. 1999. Automatic prediction of problematic human-computer dialogues in how may i help you? In *ASRU99*.

R. Laroche and G. Putois. 2010. D5.5: Advanced appointment-scheduling system "system 4". Prototype D5.5, CLASSIC Project.

R. Laroche, G. Putois, et al. 2011. D6.4: Final evaluation of classic towninfo and appointment scheduling systems. Report D6.4, CLASSIC Project.

Romain Laroche. 2010. *Raisonnement sur les incertitudes et apprentissage pour les systemes de dialogue conventionnels*. Ph.D. thesis, Paris VI University.

Willem J. M. Levelt. 1989. *Speaking: From Intention to Articulation*. Cambridge, MA: MIT Press.

Kenneth Lock. 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.*

Kyoko Matsuyama, Kazunori Komatani, Tetsuya Ogata, and Hiroshi G. Okuno. 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*.

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

Antoine Raux and Maxine Eskenazi. 2008. Optimizing endpointing thresholds using dialogue features in a spoken dialogue system. In *SIGDIAL*.

David Schlangen and Gabriel Skantze. 2011. A general, abstract model of incremental dialogue processing. *Dialogue and Discourse*, 2:83–111.

Ethan O. Selfridge, Iker Arizmendi, Peter A. Heeman, and Jason D. Williams. 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*, July.

Ethan Selfridge, Iker Arizmendi, Peter Heeman, and Jason Williams. 2013. Continuously predicting and processing barge-in during a live spoken dialogue task. In *Proceedings of the SIGDIAL 2013 Conference*.

Gabriel Skantze and David Schlangen. 2009. Incremental dialogue processing in a micro-domain. In *ACL*.

Michael K. Tanenhaus, Michael J. Spivey-Knowlton, Kathleen M. Eberhard, and Julie C. Sedivy. 1995. Integration of visual and linguistic information in spoken language comprehension. *Science*, 268:1632–1634.

Mats Wirén. 1992. *Studies in Incremental Natural Language Analysis*. Ph.D. thesis, Linköping University, Linköping, Sweden.

Marcin Wlodarczak and Petra Wagner. 2013. Effects of talk-spurt silence boundary thresholds on distribution of gaps and overlaps. In *INTERSPEECH Proceedings*.

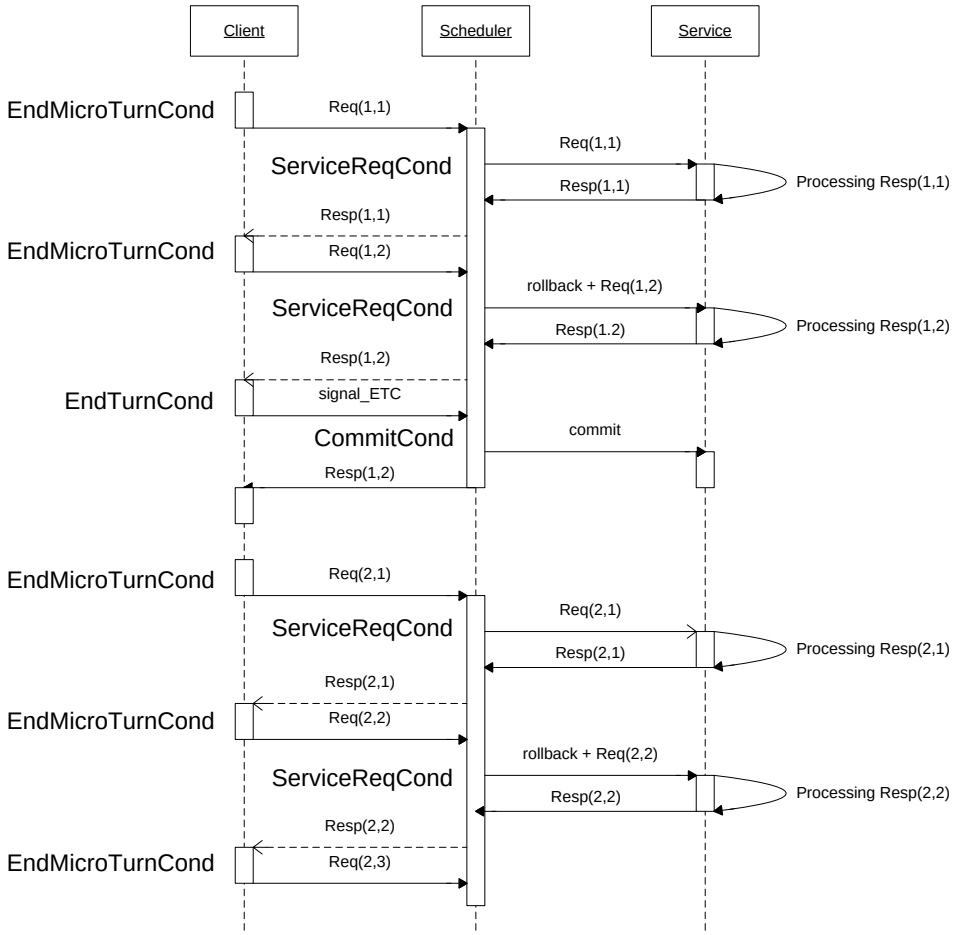

Figure A.1: The scheduler sub-requests management (The streams in dashed lines are received by the recuperation thread of the client).

Turn	**User sub-turn**	**Input**	**Real context**	**Simulation context**
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 A.2: A double context: the real context and the simulation context.