Jindigo: a Java-based Framework for Incremental Dialogue Systems

Gabriel Skantze

Department of Speech, Music and Hearing, KTH, Sweden
gabriel@speech.kth.se

Abstract

This paper presents Jindigo – a Java-based open source framework for implementing and experimenting with incremental dialogue systems. The framework is based on a general, abstract model of incremental processing. The paper describes how the framework supports revisions, incremental feedback, incremental reference resolution and incremental production of spoken utterances.

Index Terms: dialogue systems, incremental processing

1. Introduction

A traditional simplifying assumption for spoken dialogue systems is that the dialogue proceeds with strict turn-taking between user and system. The minimal unit of processing in such systems is the *utterance*, which is processed in whole by each module of the system before it is handed on to the next. Contrary to this, humans understand and produce language *incrementally* – they use multiple knowledge sources to determine when it is appropriate to speak, they give and receive backchannels in the middle of utterances, they start to speak before knowing exactly what to say, and they incrementally monitor the listener's reactions to what they say [1]. The benefits of incremental processing in spoken dialogue systems have recently attracted an increasing amount of attention [2, 3, 4]. However, most of these efforts have focused on specific aspects of incremental processing (ASR, parsing, reference resolution, etc), and not on how to build complete systems.

This paper presents a new open source framework for implementing and experimenting with completely incremental dialogue systems, called Jindigo[1]. The development of this framework is based on the experience of developing the Higgins dialogue system [5]. However, although Higgins has recently been used in some experiments on incremental processing [6], it is not built from ground-up on a unified model of incremental processing. In Jindigo, on the other hand, all datatypes are derived from a general model of incremental processing. Another difference is that all modules in Jindigo run as separate threads within a single Java process, whereas in Higgins, the modules run in separate processes and are implemented in various programming languages. A single process allows the modules to share the same memory space.

The description of the framework will move from abstract to concrete: We start by briefly describing the general, abstract model of incremental processing that Jindigo is based on. We then describe how Jindigo can be seen as a general middleware layer implementing this model. Then we describe a typical Jindigo system architecture and how dialogue may be processed. Finally we illustrate the potential benefits of incremental processing, using some simple example applications implemented within the framework.

[1] Binaries and source code can be downloaded from http://www.jindigo.net, where a video showing an example application can be seen as well.

2. The IU-model of incremental processing

Schlangen & Skantze [7] describes a general, abstract model of incremental dialogue processing, which Jindigo is based on. We only have room for a very brief overview here. In this model, a system consists of a network of processing modules. Each module has a left buffer, a processor, and a right buffer, where the normal mode of processing is to take input from the left buffer, process it, and provide output in the right buffer, from where it goes to the next module's left buffer. (Top-down, expectation-based processing would work in the opposite direction.) An example is shown in Figure 1. Modules exchange incremental units (IUs), which are the smallest 'chunks' of information that can trigger connected modules into action (such as words, phrases, communicative acts, etc). IUs typically are part of larger units: individual words are parts of an utterance; concepts are part of the representation of an utterance meaning. This relation of being part of the same larger unit is recorded through *same-level links*. In the example below, IU_2 has a same-level link to IU_1 of type PREDECESSOR, meaning that they are linearly ordered. In a tree structure (such as a parse tree), IUs could be connected by CHILD links. The information that was used in creating a given IU is linked to it via *grounded-in* links. In the example, IU_3 is grounded in IU_1 and IU_2, while IU_4 is grounded in IU_3.

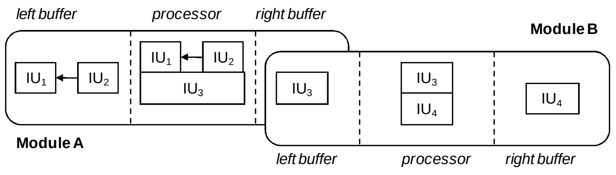


Figure 1: *Two connected modules.*

A challenge for incremental systems is to handle *revisions*. For example, as the first part of the word "forty" is recognised, the best hypothesis might be "four". As the speech recogniser gets more input, it might need to revise its previous output, which might cause a chain of revisions in all subsequent modules. To cope with this, modules have to be able to react to three basic situations: that IUs are *added* to a buffer, which triggers processing; that IUs that were erroneously hypothesized by an earlier module are *revoked*, which may trigger a revision of a module's own output; and that modules signal that they *commit* to an IU, that is, won't revoke it anymore.

3. Jindigo as a middleware layer

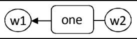
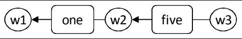
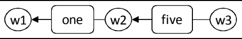
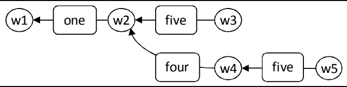
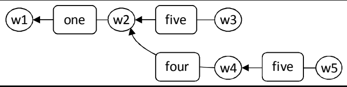
Jindigo can be regarded as a middleware layer for building incremental modules based on the general model outlined above, and it provides a set of standard modules with which it is possible to implement incremental spoken dialogue systems.

In Jindigo, IUs are modelled as typed objects, where all IUs are derived from the base class INCREMENT that handle grounded-in relations, same-level links, etc. The modules in the system are also typed objects, but buffers are not. Instead, a buffer can be regarded as a set of IUs that are connected by (typed) same-level links. Since all modules have access to the same memory space, they can follow the same-level links to examine (and possibly alter) the buffer. Update messages between modules are relayed based on a system specification that defines which types of update messages from a specific module go where. Since the modules run asynchronously, update messages do not directly invoke methods in other modules, but are put on the input queues of the receiving modules. The update messages are then processed by each module in their own thread.

3.1. A graph-based update model

In a system where modules run in separate processes and memory is not shared, modules have to somehow synchronize their buffers when changes are being made. Since memory space is shared in Jindigo, this is not necessary, which facilitates a more efficient processing and more compact update messages. However, having concurrent threads altering and accessing the state of the IUs can lead to problems of synchronization (see for example [8] for a discussion). There are different ways of dealing with this problem; one approach is to make objects (partly) immutable. Jindigo provides a mechanism for defining immutable IUs, while still allowing revision and commitment. In this model, IUs are connected by predecessor links, which gives each IU a position in a (chronologically) ordered stream. Positional information is reified by super-imposing a network of position nodes over the IU network, with the IUs being associated with edges in that network. An example is shown in Table 1, which shows the right buffer of an ASR at different time-steps.

Table 1: *The right buffer of an ASR module, and update messages at different time-steps.*

String	**Right buffer**	**Update message**
t_1: one		[w1, w2]
t_2: one five		[w1, w3]
t_3: one		[w1, w2]
t_4: one four five		[w1, w5]
t_5: [commit]		[w5,w5]

As the example shows, the positional nodes are used to represent the update stage at different time-steps. Each update message consists of a pair of pointers [*C*, *A*]: (*C*) what the newest committed IU is and (*A*) what the newest non-revoked or active IU is. So, the change between the state at time t_1 and t_2 is signalled by *A* taking on a different value. This value (w3) has not been seen before, and so the consuming module can infer that the network has been extended; it can find out which IUs have been added by going back from the new *A* to the last previously seen position (in this case, w2). At t_3, a retraction of a hypothesis is signalled by a return to a previous state, w2. All consuming modules have to do now is to return to an internal state linked to this previous input state. Commitment is represented similarly through a pointer to the rightmost committed node; in the figure, that is for example w5 at t_5.

Since information about whether an IU has been revoked or committed is not stored in the IU itself, all IUs can (if desirable) be defined as immutable objects. The model also supports parallel hypotheses, in which case the positional network would turn into a lattice (as shown in Figure 5).

In the example above, the output of the module is a simple stream of words. However, it is also possible to communicate more complex structures using this method. Figure 2 illustrates the right buffer of a parser. Here, the chronologically ordered stream represents the current best traversal of the widest spanning phrases found so far. These phrases in turn point to the underlying phrase structure (IUs connected by CHILD links, forming a tree structure). By following these pointers, IUs that are not directly part of the ordered buffer may be accessed.

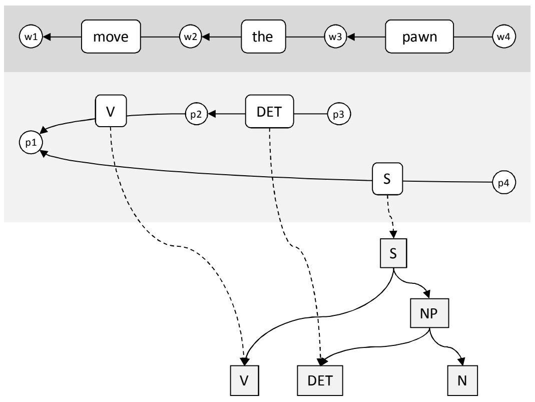


Figure 2: *Top: Right buffer of an ASR module. Middle: The linear right buffer of a parser parsing the ASR output (containing the current best sequence of top phrases). Vertices are numbered in the order they are produced; p1 is associated with w1, p2 with w2, etc. Bottom: The underlying parse tree that is incrementally produced.*

3.2. Semantic representation

Jindigo provides a model for semantic representation using typed concepts and arguments. The type hierarchy of concepts is defined in an XML-based domain specification, which can then be compiled into a Java class hierarchy. For example, there may be a concept `Red` which inherits the concept `Property` which in turn inherits the base class `Concept`. Each concept may have a number of typed arguments, which are pointers to other concepts. This model is in turn backed by the IU-model described in Section 2, so that the `Concept` class is inherited from the `Increment` class and arguments are represented as same-level links. An example of semantic representations in a chess domain is shown in Figure 3. In this example, the concept `Move` may take the argument `piece` of type `Piece` and the argument `position` of type `Position`. The concepts `Bishop` and `Queen` are inherited from `Piece` and `FrontOf` inherited from `Position`.

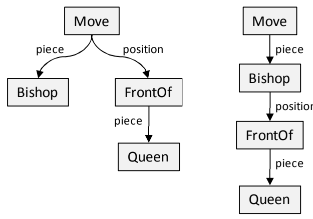


Figure 3: *Two alternative semantic representations of "move the bishop in front of the queen".*

A domain definition usually needs to be defined for each new application, but it is of course possible to reuse sub-domains such as number, time and colour domains and their corresponding grammars.

3.3. Typical system architecture

Jindigo provides base classes for implementing any kind of module, but it also comes with a set of standard modules. A typical system architecture is shown in Figure 3. The buffers between these modules are illustrated in Figure 4.

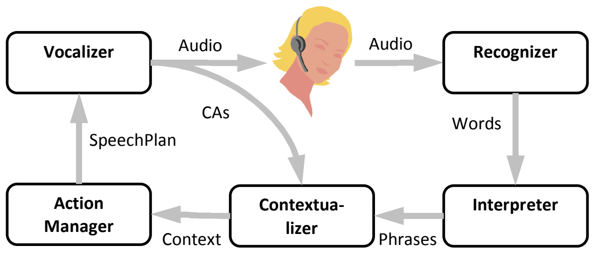


Figure 3: *A typical Jindigo system architecture.*

The standard Recognizer is based on CMU Sphinx 4 [9], which has been adapted to act as a Jindigo module. The word buffer is parsed by an Interpreter (similar to [10]) which tries to find an optimal sequence of top phrases and their semantic representations (`Concepts`). These phrases are then packaged as Communicative Acts (CAs) by a Contextualizer (similar to [11]). As can be seen in Figure 4, the Contextualizer maintains two linear graphs, one for user CAs and one for system CAs, and the pointers to these graphs forms the current `Context`. Each concept may implement the method `contextualize`, which is called by the Contextualizer with the current `Context` as argument. This method then returns a reinterpretation of the concept structure, taking the current context into account. As a very simple example, the Interpreter may interpret the elliptical utterance "red" as RED, but when calling `contextualize` (which has probably been inherited from some super-class) for this concept, the method will find the system request "what colour is it" (in its semantic form) in the dialogue history and transform the ellipsis into a semantic representation of a statement about the colour of a specific object. Similarly, the `contextualize` method may perform reference resolution and create anaphoric pointers to other concepts in the dialogue history.

The Action Manager consults the current `Context` and generates a `SpeechPlan` that is sent to the Vocalizer. As Figure 4 illustrates, the `SpeechPlan` contains a lattice of `SpeechSegments` that represents possible renderings of the utterance, allowing the system to generate a flexible, varied output. Each `SpeechSegment` may be associated with a `Concept` (the semantic equivalent to the segment). When a `SpeechSegment` has been rendered by the Vocalizer, this `Concept` is sent to the Contextualizer, allowing the system to self-monitor its own output. This way, user CAs may be interpreted in the context of system CAs (such as an answer to a request). We currently use MaryTTS [12] as a speech synthesizer backing the standard Vocalizer module. The incremental speech production in Jindigo is further described in [13]. This division of labour makes it possible for the Action Manager to generate utterances asynchronously, while the Contextualizer and Vocalizer takes care of the synchronization of utterances as the dialogue unfolds (cf. [14]).

Jindigo provides a base class called `ActionManager`. To implement an application within Jindigo, the developer may inherit this class and override methods that define the behaviour of the system. One possibility is to extend this with a hierarchy of `ActionManager` classes that handle different types of dialogue behavior, such as form-filling, etc.

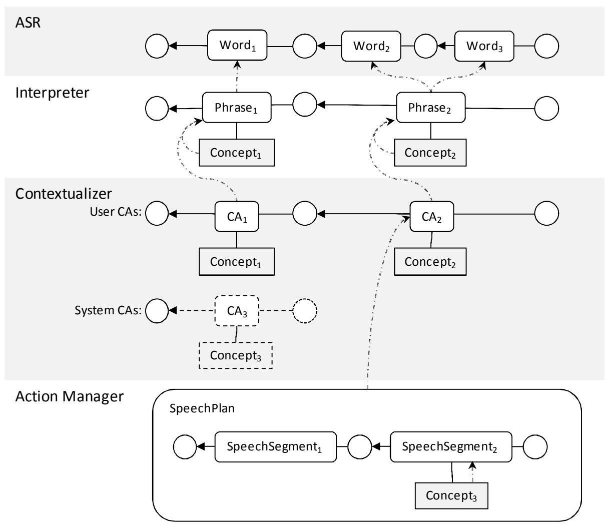


Figure 4: *The right buffers of the different modules in Figure 3. Semi-dotted lines show grounded-in pointers.*

4. Example applications

To illustrate the potential benefits of incremental processing, we will now describe three different simple applications based on Jindigo and discuss how they utilize incrementality in different ways.

4.1. Numbers: Incremental feedback

Skantze & Schlangen [6] describe a completely incremental system in a very limited domain: number dictation. The system was based on the Higgins framework (the predecessor to Jindigo), but has now been re-implemented in Jindigo. Number dictation in commercial telephone-based systems is typically not incremental and the user has to read the whole sequence before getting any response. A study of human-human number dictation revealed that humans typically package the sequence in smaller groups, or *installments* [1]. A rising pitch at the end of an installment tells the receiver that more is to be expected, but also leaves room for an acknowledgement ("mhm", "yeah") or a clarification request ("four *two* three?"). A falling pitch signals that the sequence has ended.

If a dialogue system should be able to mimic this behaviour, it needs to be able to give very fast responses based on this pattern. In the Numbers system, the ASR module also annotates each word based on the slope of the pitch (as described in [6]). Each installment is modelled as a CA, where a `Group` concept may have a set of `Number` concepts as arguments. By following the grounded-in links of the CA, the Action Manager may find the last involved word from the ASR and consult the pitch slope in order to determine whether it should give an acknowledgement (or a clarification request in case of a low confidence score) or start to repeat the whole sequence. As described in 3.3, each time a new word is produced by the ASR, the Action Manager creates a new `SpeechPlan` and the Vocalizer prepares to start speaking. This allows the system to give backchannels after mid-sequence installments in about 200 ms.

4.2. Chess: Incremental reference resolution

Another simple application that has been implemented in Jindigo is Chess, a speech controlled chess game. The user can move the pieces on the board by saying commands such as "take the right knight" or "move the pawn in front of the queen". The system can make clarification requests such as "which pawn?" or "how many steps?". In this domain, incremental processing allows the system to highlight the possible interpretations on the chess board as the user is speaking, and of course quickly make backchannels or clarification requests. But it also allows the system to incrementally resolve referring expressions as the utterance is parsed, similarly to [2]. As a new reference to a piece or a move is created by the Interpreter, it is sent directly to the Action Manager for evaluation. The Action Manager compares the concept structure to the current state of the chess board and annotates it based on whether the description may apply. If it doesn't, the chart parser may prune the edge from its chart as soon as it has been created, which will both make it more efficient and help it to output the correct interpretation. As an example, take the syntactic ambiguity in Figure 3, which renders two different solutions. By consulting the current state of the board, one of them may (possibly) be excluded. Another case where this might be useful is if we parse a word lattice from the ASR, as shown in Figure 5. This will render six possible parses, but the number of parses can easily be pruned by consulting the chess board. Even if the utterance will continue (e.g., with "two steps"), the chart has already been pruned.

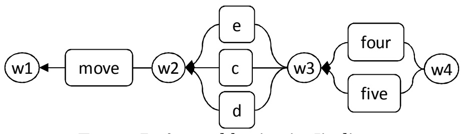


Figure 5: *A word lattice in Jindigo.*

This example shows how information may go both left-to-right and right-to-left in the system. In the buffer between the Interpreter and the Action Manager (not present in Figure 3), the order between the concepts that are to be annotated is not important and there is no point in revoking such IUs (since they do not affect the state of the Action Manager). Therefore, this buffer is not represented with the model presented in 3.1, but as a simple queue.

4.3. Deal: Incremental speech production

As a third example, the conversational language learning game Deal [15] has been re-implemented in Jindigo. In this game, the system is a flea market dealer and the user takes the role of a buyer trying to haggle the price of different objects. Within this domain, we have used Jindigo to explore how to do incremental speech production. Incremental speech production can be useful if the system detects that the user has stopped speaking and it is appropriate for the system to start speaking, but the system hasn't completed the processing of the user utterance and a new complete `SpeechPlan` cannot be constructed yet. The reason could be that the system still need time to access some data source, or that the ASR is given more time for processing, or that a Wizard-of-Oz setup is used where the Wizard needs more time to fully transcribe the user's utterance. In Jindigo, it is possible for the Action Manager to output a tentative `SpeechPlan`, which is amended as the system is speaking and processing continues. The system could for example grab the floor by using a filled pause, or something more meaningful based on the input so far. If revisions to the `SpeechPlan` contradict what has been spoken so far, the Vocalizer will automatically make self-repairs. The system has been evaluated with 10 users, showing that incremental speech production makes the system more efficient and pleasant to use. This is described in detail in [13].

5. Future work

Jindigo is still at an early stage of development and has so far only been used for prototype research applications. A lot of work remains to be done to build a stable platform for implementing dialogue systems. An important next step is to implement general dialogue behaviours, such error handling [5], in the framework, relieving the application developer of this. All IUs carries a property called confidence, and since all IUs are grounded in lower level IUs, it is for example straightforward to derive the ASR confidence score of a `Concept`. Another interesting extension is to implement probabilistic dialogue models. It is of course also possible to implement non-incremental systems using the framework.

As the examples above show, we have already started to use the framework for exploring issues related to incremental dialogue processing. We hope that it will also be of use for other people in the research community.

6. References

[1] Clark, H. H. (1996). *Using language.* Cambridge, UK: Cambridge University Press.

[2] Stoness, S. C., Tetreault, J., & Allen, J. (2004). Incremental parsing with reference interaction. In *Proceedings of the ACL Workshop on Incremental Parsing* (pp. 18-25).

[3] Baumann, T., Buß, O., Atterer, M., & Schlangen, D. (2009). Evaluating the Potential Utility of ASR N-Best Lists for Incremental Spoken Dialogue Systems. In *Proceedings of Interspeech*. Brighton, UK.

[4] DeVault, D., Sagae, K., & Traum, D. (2009). Can I Finish? Learning When to Respond to Incremental Interpretation Results in Interactive Dialogue. In *Proceedings of SigDial* (pp. 11-20). London, UK.

[5] Skantze, G. (2007). *Error Handling in Spoken Dialogue Systems - Managing Uncertainty, Grounding and Miscommunication*. Doctoral dissertation, KTH, Department of Speech, Music and Hearing.

[6] Skantze, G., & Schlangen, D. (2009). Incremental dialogue processing in a micro-domain. In *Proceedings of EACL-09*. Athens, Greece.

[7] Schlangen, D., & Skantze, G. (2009). A general, abstract model of incremental dialogue processing. In *Proceedings of EACL-09*. Athens, Greece.

[8] van Roy, P., & Haridi, S. (2004). *Concepts, Techniques, and Models of Computer Programming.* Cambridge, MA: The MIT Press.

[9] Lamere, P., Kwok, P., Gouvea, E., Raj, B., Singh, R., Walker, W., Warmuth, M., & Wolf, P. (2003). The CMU SPHINX-4 speech recognition system.. In *Proceedings of the IEEE Intl. Conf. on Acoustics, Speech and Signal Processing*. Hong Kong.

[10] Skantze, G., & Edlund, J. (2004). Robust interpretation in the Higgins spoken dialogue system. In *Proceedings of ISCA Tutorial and Research Workshop (ITRW) on Robustness Issues in Conversational Interaction*. Norwich, UK.

[11] Skantze, G. (2008). Galatea: A discourse modeller supporting concept-level error handling in spoken dialogue systems. In Dybkjær, L., & Minker, W. (Eds.), *Recent Trends in Discourse and Dialogue*. Springer.

[12] Schröder, M., & Trouvain, J. (2003). The German Text-to-Speech Synthesis System MARY: A Tool for Research, Development and Teaching. *International Journal of Speech Technology, 6*, 365-377.

[13] Skantze, G., & Hjalmarsson, A. (submitted). Towards Incremental Speech Production in Dialogue Systems. Submitted to *Proceedings of SigDial*. Tokyo, Japan.

[14] Raux, A., & Eskenazi, M. (2007). A multi-Layer architecture for semi-synchronous event-driven dialogue Management. In *ASRU 2007*. Kyoto, Japan..

[15] Hjalmarsson, A., Wik, P., & Brusk, J. (2007). Dealing with DEAL: a dialogue system for conversation training. In *Proceedings of SigDial* (pp. 132-135). Antwerp, Belgium.