Communication in large distributed AI Systems for Natural Language Processing

Jan W. Amtrup*
University of Hamburg, Comp. Sci. Dept.
Vogt-Kölln-Str. 30
D-22527 Hamburg
amtrup@informatik.uni-hamburg.de

Jörg Benra*
DFKI GmbH
Erwin-Schrödinger-Str. (Bau 57)
D-67663 Kaiserslautern
benra@dfki.uni-kl.de

Abstract

We are going to describe the design and implementation of a communication system for large AI projects, capable of supporting various software components in a heterogeneous hardware and programming-language environment. The system is based on a modification of the channel approach introduced by Hoare (1978). It is a three-layered approach with a de facto standard network layer (PVM), core routines, and interfaces to five different programming languages together with support for the transparent exchange of complex data types. A special component takes over name service functions. It also records the actual configuration of the modules present in the application and the created channels.

We describe the integration of this communication facility in two versions of a speech-to-speech translation system, which differ with regard to quality and quantity of data distributed within the applications and with regard to the degree of interactivity involved in processing.

1 Introduction

Currently, there is a trend of building large AI-systems in a distributed, agent-oriented manner. The complex tasks performed e.g. by systems with multimodal user interfaces or by systems tackling the processing of spontaneous speech often require more than one computer in order to run acceptably fast. If pure speed is not the primary motivation, the incorporation of several modules, each of them possibly being realized in a different programming language or even a different programming paradigm, demands complex interfaces between these modules. Furthermore, only modularization makes it possible to develop applications in a truly distributed manner without the need to copy and install versions repeatedly over.

The actual realization of the interfaces should ground on a sound theoretical framework, and it should be as independent as possible from the module implementations. Typically, when an interface between two subcomponents of a system is needed, at first very simple means -- e.g. file interfaces or simple pipes -- are considered. This does not only lead to a variety of different protocols between components -- which is natural to a certain degree due to the different tasks performed by the components and the purpose of the interface data -- but also to a number of different implementation strategies for interfaces.

In this paper, we present ICE, the *I*ntarc *C*ommunication *E*nvironment (Amtrup, 1995), an implementation of a channel-oriented, multi-architecture, multi-language communication module for large AI-systems, which is particularly useful for systems integrating speech and language processing.

A channel-oriented model for interaction relations between software modules seemed to be the most suitable system for our needs. We adopted the CSP-approach (Hoare, 1978) and its actual realization in the transputer hardware (Graham and King, 1990). This core functional model was slightly modified to satisfy the needs emerging from experiences with actual systems. We decided not to implement all communication functions from scratch, but instead we use PVM, the *P*arallel *V*irtual *M*achine (Geist et al., 1994), a widespread process-communication software, which turned out to be extremely reliable.

We will desribe how the communication system has been integrated within Verbmobil, a large research project for automatic speech-to-speech translation (Wahlster, 1993). ICE is used for the various prototypes of the interpretation system.

*This research was funded by the Federal Ministry of Education, Science, Research and Technology (BMBF) in the framework of the VERBMOBIL Project under Grants 01 IV 101 A/O and 01 IV 101 G.

We describe experiences and results of the work on the first demonstrator. Furthermore, we show that ICE is flexible enough to be used in architectural experiments and we are going to report some of the experiences made with them.

2 Application acrchitecture

Verbmobil, the primary application for which ICE was built, aims at developing an automatic interpreting device for a special type of negotiation between business people. The dialogue situation is as follows: Two business persons, speaking different languages, are involved in a face-to-face dialogue trying to schedule an appointment. They both have at least some knowledge of English and use English as a common language. In case one of the dialogue partners runs into problems, he or she activates the interpretation system by pressing a button and switches back to his or her mother-tongue. The system interprets the respective utterances into English. Therefore, it interprets the dialogue on demand in certain situations.

The Verbmobil system consists of a large number of components, each of them designed to cope with specific aspects of the interpretation process. Among them are a recorder for speech signals, a HMM-based word recognizer, modules for prosodic, syntactic and semantic analysis, dialogue processing, semantic evaluation as well as components for both german and english synthesis. There are several interfaces between the individual parts of the application which are used to forward results or to realize question-answering behavior.

The interchanged data between components (a component normally corresponds to a unique software module) is very heterogeneous with regard to both type and quantity: Speech information as it is sent from the recorder to the speech recognizer consists of a stream of short integer values which may amount to several megabytes. The objects exchanged between semantics construction and transfer are relatively small, but highly structured: Semantic representations with several embedded layers.

3 ICE: Design and structrue

As briefly noted above, we are using a channel abstraction to model communication between components. The model is largely oriented at the approach of CSP (*C*ommunicating *S*equential *P*rocesses, Hoare (1978)), mainly for two reasons:

- We decided to use a message-passing approach to communication. The two other kinds of process communication largely available, namely shared memory and remote procedure calls are disadvantegous for our purposes: The employment of shared memory may lead to memory or bus contention when several processors are simultaneously attached to the same physical memory segment. Furthermore, multiple concurrent write attempts have to be synchronized. Remote procedure calls did not seem to be the right choice either since their use implies a rendez-vous-synchronization which slows down a system due to network latencies[1].
- Making the objects involved in communication explicit, offers several ways to manipulate them. Without too much effort, we were able to introduce split channels in order to incorporate visualization tools or introduce different modes of communication depending on the type of data to be exchanged.

The low level basis of ICE is realized by PVM (Geist et al., 1994), which is a message passing system for multiple hardware architectures. It has been developed and extended for almost seven years now and is very reliable. It allows a net of Unix workstations to behave like a single large parallel computer. PVM supplies each message with a tag which simplified the introduction of channels to a large extent (roughly, a message is tagged uniquely to identify the channel it is sent on. This enables a receiving component to select messages on individual channels).

3.1 System structure

The architecture of a system using ICE as communication framework is depicted in Fig. 1. Before describing in detail the structure of a component, we will point out the overall layout of an application.

We assume that an application consists of a number of components. We could have adopted the notion of agents cooperating to a certain degree while carrying out a certain task cooperatively, but this would have meant to mix up different conceptual levels of a system: The communication facilities we are describing here establish the means by which pieces of software may communicate with each other. They do not prescribe the engineering approaches used to implement the individual software components themselves. We do not state that agent architectures

[1] The channels of CSP and Occam both use rendez-vous-synchronization. In this respect we deviated from the original model.

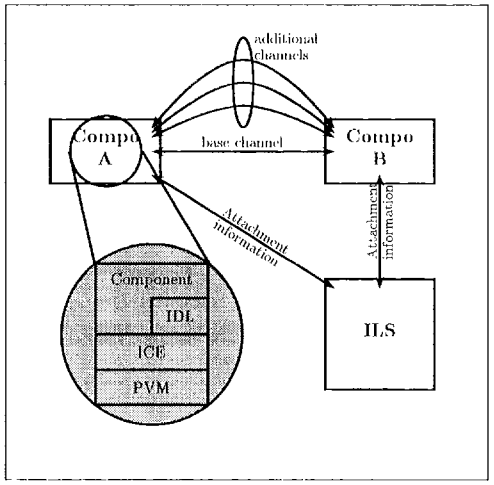

Figure 1: Principle component layout

(e.g. Cohen et al. (1994)) can not be realized with our mechanism[2], but the range of cases where ICE can be applied is broader than this.

All communication is done by the means of channels, as set out above. We distinguish two types of channels:

- *Base channels* are the primary facilities of communication. They are configured in a way guaranteeing that each component is able to interact with each other component it wishes to, regardless of programming languages, hardware architectures, or system software being used. This is achieved by using the standard communication mode of PVM, which supports XDR[3]. Message passing is done asynchronously.

- *Additional channels* were added in order to satisfy some needs that frequently arise during the design and implementation of large AI-systems with heavy use of communication. They can be used to separate data streams from control messages or may be configured in various ways, e.g. by switching off the XDR encoding to speed up message passing.

3.2 Split channels

Both types of channels can be configured in an additional way. Beyond being bidirectional communication devices between two components, other modules can be attached to listen to data transported on a channel or to inject messages. These *split channels* are achieved by dividing a channel into two endpoints, one at each side of the channel.

Both ends are described using a configuration file that is read by the ILS (see below) upon startup. In this file, for each endpoint a list of real channels is defined, each of which points to a component and is equipped with a name, configuration flags and its purpose (which can be sending or receiving). Any number of real channels may be marked sending or receiving. The behavior of the components affected by split channels does not have to be changed, since splitting occurs transparently for them.

Consider Fig. 2 as an example for what purpose split channels were used.

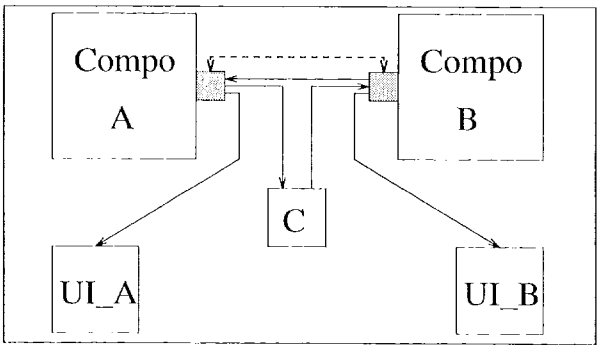

Figure 2: Split channel configuration

Two components, **A** and **B**, are connected using a channel which is depicted by a dashed line. The channel endpoints are split up to allow visualization of message data sent by either component. The visualization is performed by two additional components labelled **UI_A** and **UI_B**. Furthermore, the data sent by component **A** must undergo some modification while being transported to component **B**. Thus, another component **C** is configured capable of transforming the data. It is spliced into the data path between **A** and **B**. Note that data sent by component **B** arrives at **A** unaffected from modification by component **C**.

3.3 ILS: Information Service

Channels can be established by any component. There is no need for synchronization between components during the configuration of the communication system. To support this schema, a dedicated component named ILS (*I*ntarc *L*icense *S*erver) was introduced. It stores information about the actual structure of the application system. This information includes names and locations of all components participating in the sys-

[2] Indeed, distributed blackboards as used in Cohen et al. (1994) can easily be modelled using a channel-based approach.

[3] e*X*ternal *D*ata *R*epresentation, see Corbin (1990), an encoding schema for data objects independent of the current programming environment.

tem as well as an overview about all channels currently established between components. The actions performed by the ILS include:

- *Attachment and Detachment of components.* A component desiring to take part in the communication activities of the application has to identify and register itself at the ILS. This is done by sending a message containing the name of the component to the ILS. Analogously, a component should detach itself from the ILS by sending an appropriate message before leaving the application. In case of a program failure resulting in the inability of a component to detach the ILS is capable of handling the detachment autonomously.

- *Configuration of channels.* Each creation and destruction of a channel is done by interacting with the ILS in order to notify the ILS of the request and to get back information about the necessary data structures. The creation of a channel is done in two phases: First, any of the endpoint components sends a channel creation request to the ILS. The ILS updates its internal configuration map taking care that split channel definitions are taken into account; it then answers to the requesting component the individual tag used for this channel and the process identity of the target component[4]. If the target component has not yet registered within the application, this fact is acknowledged to the source component. The only point at which this matters is the time of the first message sending attempt which will be blocked until the target component registers at the ILS. In that case, the ILS notifies the source component of the event and communication can take place normally.

  The second phase handles the notification of the target component. As just described, this component need not be present by the time of the channel creation request. In this case the notification is simply delayed. The notification consists of the necessary data to create the intended channel within the component. The implementor need not track those configuration messages, the communication layer handles this transparently. Furthermore, concurring channel requests do not interfer.

[4]PVM addresses components — which are identical to processes for it — by a task id that is assigned by the pvm daemon. The ILS maintains a mapping from component names to those task ids. This mapping need not be bijective, since we allow multiple components within one process (see below).

3.4 Component structure

The interior structure of a component (see Fig. 1) is layered as far as the communication parts of the software are concerned. The low level communication routines are provided by PVM (see above). Next, a software layer defines the functions of ICE. This is comprised of the basic functionality of ICE itself and a set of interface functions for different programming languages. We currently support C, C++, Lisp (Allegro Common Lisp, Lucid Common Lisp and CLISP), Prolog (Quintus Prolog and Sicstus Prolog) and Tcl/Tk.

These software layers suffice to communicate basic data types like numbers and strings. Additionally, a separate layer (IDL) is present to allow the exchange of more compex data types. One may specify routines to encode and decode user-defined data types which can then be transmitted just as the predefined scalar types. At the moment, this schema is used for a few dedicated data structures, e.g. for speech data or arbitrary prolog terms, which may be even cyclic.

4 Experiences with the application

Verbmobil is built up by two sorts of components. The "core" components are used to transform the input data into the output data (e.g. recording, speech recognizer etc.). These NLP-components are embedded in the so called "testbed" that serves as an application framework. The testbed is designed as an experimental environment that provides all the features required to test the core components and to study the operation of the whole application. The testbed consists mainly of the following parts:

- The graphical user interface (GUI) provides a comfortable frontend to the application. Using the GUI the user can watch the operation of the whole system, control the behavior of the components and monitor the dataflow between the components.

- The testbed manager (TBM) is used to start up the whole application and to distribute the processes of the application to the hosts of the network. Further, the testbed manager collects data about the operation of the components and visualizes this information using the GUI.

- The visualisation manager (VIM) collects all the data transferred between any of the components using ICE channels.

If one wants to study only some parts of the system, it is possible to start the application containing only a subset of the existing components (e.g. only the speech recording module and some speech recognizers). The testbed provides the facility to choose in an offline process the components that are desired to be executed. This configuration is done by simply editing a configuration file and selecting the keywords "yes" or "no" for each component. All the components not selected are automatically replaced by "stub-modules," so there is no need to change source code and recompile the components, even if data is sent to a non-existent component. On the other side it is possible to configure the usage of alternative components (e.g. two german speech recognizers). In this case both components are started and we are able to select from the GUI which of both components we actually want to use.

Currently there are 32 existing components that contribute to roughly 650 MB disk space (the executables, libraries and data files required at runtime use up 380 MB). Some of the components are structured using subcomponents that are implemented in different programming languages and are executed in own processes. The 32 main components are implemented using the following programming languages: C (10 components), Lisp (7 components), Prolog (5 components), C++ (5 components), Fortran (4 components), Tcl/Tk (1 component).

Starting a heavy weight system containing all the currently existing components, we get about 95 UNIX processes requiring 520 MB memory. In this configuration we are using 52 base channels and 24 additional channels (76 ICE channels in total). Six of these 24 additional channels are configured not to use the XDR coding, because they are used to transfer high volume data (e.g. audio data).

Because the communication is built up by strictly using the features of ICE and the underlaying PVM, the application can run on a single host as well as distributed to the hosts of an local area network. The decision which component will run on which host of the network is configurable. Each component can be assigned to a specific host, or we can leave the assignment of an adequate host to PVM.

5 Experiences with an architectural experiment

In addition to the employment within the Verbmobil prototype, we used ICE as communication device for some experiments in the framework of the

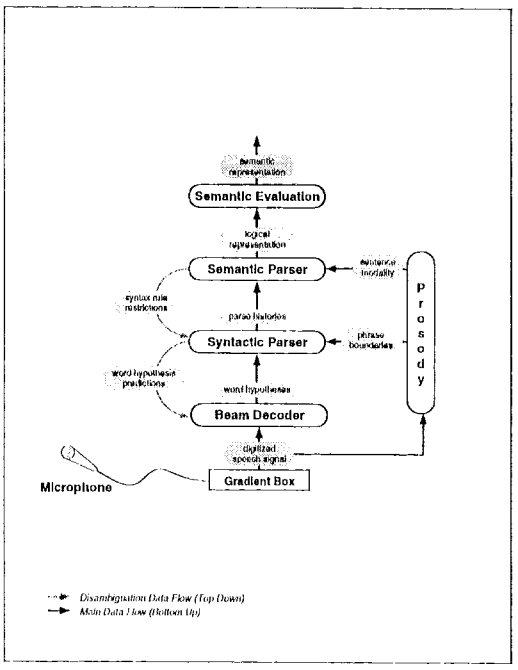

Figure 3: The experimental system architecture.

architectural branch of the project. The approach here is to develop a speech translation system obeying design principles that have their origin in the goal of constructing a system reflecting some of the assumed properties of human speech processing, namely working incrementally from left to right and exploring the effects of interaction between different levels of speech recognition and understanding. These two principles have serious implications for the design of individual components and the complete system. To give a concrete example, consider the interface between a speech recognizer and a syntactic parser. The recognizer produces a connected graph where each edge denotes a word hypothesis. Due to the inability to remove paths in advance that can not be pursued further at a later stage of operation, the input to the syntactic parser grows enormously. We noticed that wordgraphs produced incrementally may be ten times larger than conventionally constructed graphs (resulting in over 2000 word hypotheses for an utterance of 4.7 seconds).

The experimental system architecture is shown in Fig. 3. It consists of several modules interconnected by a main data path that delivers results according to the "standard" linguistic hierarchy, viz. from word recognition to syntax, semantics and finally transfer[5]. Besides this mainstream data path we set up several interaction facilities that are used to propagate information

[5]The transfer component ist not shown in Fig. 3.

backwards, which may consist of binary judgements about the applicability of a hypothesis, a ranking among different possible analyses or even predictions about what might be expected in the future.

These methods were for example examined at the crucial interface between a HMM-based speech recognition device and a syntactic parser (Hauenstein and Weber, 1994). A tight interaction between these two components was created which was used to model a synchronization point at every frame in the speech input (i.e. every 10 ms). At each of these points a set of word hypotheses is sent to the parser. The parser then tries to integrate the new hypotheses into existing partial analyses constructed so far. The feedback loop to the speech recognizer consists of information about the syntactic ranking of the parse each word is integrated into. If a word can not be used in any way, it is simply rejected. In the case of integration of a word into a parse a ranking is produced which incorporates values from a statistical n-gram language model and a stochastic unification grammar which models the probability of a syntactic derivation.

To realize a prediction mode in this interaction, a different schema was used: At each frame the parser computes a set of possible continuations for each word, i.e. it restricts the language model to pairs of words (in case of a bigram model) which are syntacticallly plausible and could be integrated into a currently existing syntactic derivation. By doing so, the search space of the speech recognizer is restricted.

6 Conclusion

We have presented the concepts and implementation of a communication system designed for use in large AI systems which nowadays are typically built to operate in a distributed manner within local networks of workstations. We argued that the adaptation of sound theoretical concepts which for example can be found in Hoare (1978) lead to solutions that have considerably more power that ad-hoc communication devices implemented as the need to communicate arises. The channel model was slightly modified and realized on top of PVM, a de facto standard for communication in distributed systems. The system structure reflects a set of components that communicate bilaterally without the involvement of a central mechanism or data structure that participates in every communication event. Instead, once the identity of the communication partners is established, communication between them is strictly local.

We introduced a central name server in order to store the components acting in an application and to be able to service requests for the creation of channels and such. Channels come in two flavors what on the one hand guarantees succesful communication between any two partners and on the other hand leaves room for tailoring properties of message channels to certain preferences. Furthermore, split channels allow for the easy configuration of a system with respect to interchangeable parts of the system and attached visualization.

We showed that the communication system realized using this methods is advantegeous in several situations and system contexts, ranging from strictly sequential systems over intermediary forms to highly interactive systems.

References

Jan W. Amtrup. 1995. ICE–Intarc Communication Environment: User's Guide and Reference Manual. Version 1.4. Verbmobil Technical Document 14, Univ. of Hamburg, December.

P.R. Cohen, A. Cheyer, M. Wang, and S.C. Baeg. 1994. An open agent architecture. In *Proc. of AAAI-94*, pages 1–8, Stanford, CA.

John R. Corbin. 1990. *The Art of Distributed Applications.* Sun Technical Reference Library. Springer-Verlag, New York.

Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. 1994. PVM3 User's Guide and Reference Manual. Technical Report ORNL/TM-12187, Oak Ridge National Laboratory, Oak Ridge, Te., May.

Ian Graham and Tim King. 1990. *The Transputer Handbook.* Prentice Hall, New York, London et al.

Andreas Hauenstein and Hans Weber. 1994. An Investigation of Tightly Coupled Speech Language Interfaces Using an Unification Grammar. In *Proceedings of the Workshop on Integration of Natural Language and Speech Processing at AAAI '94*, pages 42–50, Seattle, WA.

Charles A. Richard Hoare. 1978. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, August.

Wolfgang Wahlster. 1993. Translation of face-to-face-dialogs. In *Proc. MT Summit IV*, pages 127–135, Kobe, Japan.