A Flexible Distributed Architecture for Natural Language Analyzers

Xavier Carreras & Lluís Padró

TALP Research Center
Departament de Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya
Barcelona, Spain.
{carreras,padro}@lsi.upc.es

Abstract

Many modern NLP applications require basic language processors such as POS taggers, parsers, etc. All these tools are usually pre-existing, and must be adapted to fit in the requirements of the application to be developed. This adaptation procedure is usually time consuming and increases the application development cost. Our proposal to minimize this effort is to use standard engineering solutions for software reusability. In that sense, we converted all our language processors to classes which may be instantiated and accessed from any application via a CORBA broker. Reusability is not the only advantatge, since the distributed CORBA approach also makes it possible to access the analyzers from any remote application, developed in any language, and running on any operating system.

1. Introduction

Modern NL Application such as Machine Translation, Summarizing, Dialog systems, etc. are very likely to require basic language processors such as tokenizers, morphological analyzers, lemmatizers, POS taggers, syntactic parsers, etc. A significant share of the effort required to develop NLP applications is devoted to the adaptation of existing software resources to the platform, format or API of the final system.

Some initiatives have been taken aiming to facilitate the integration of existing resources, such as GATE (Cunningham et al., 1996), based on the Tipster architecture (Grishman, 1996). Nevertheless, since they were more interested in the development of research prototypes than final applications, those systems were far away from being efficient in processing time and in storage space.

The approach of GATE is to develop a wrapper for each linguistic tool to be used, and to define which are the dependencies between tools. The resulting system is a powerful prototyping environment in which one can test and select the processing chain that best matches one's needs.

Nevertheless, the system is based on a pipelined schema, thus, any piece of text is run through all the selected processors, and for each of them, is annotated and stored before it is passed to the next step. The prototyping power of the system, which enables to create and test any combination of existing processors, has the price of a very large space and time cost, which makes it unsuitable for industrial development purposes.

In this paper we present the architecture we are currently using, which enables the quick and easy integration of basic language analyzers in any NLP application. It is based on a distributed object representation of language analyzers which makes possible a client-server architecture to combine the different tools, enabling a quick integration of linguistic processors into an application, and the development of efficient NLP applications.

Xavier Carreras holds a grant by the Catalan Research Department. This research has been partially funded by EC (NAMIC IST-1999-12392) and the Spanish Research Department (TIC2000-0335-C03-02, TIC2000-1735-C02-02)

2. Requirements

There are two crucial requirements to integrate language analyzers in NLP applications: Reusability and efficiency, which are highly related.

Obviously, the ideal situation is to completely reuse pre-existing language analyzers inside any new NLP application. The usual technique consists of wrapping the pre-existing analyzer with format converters that translate from the application data exchange format to the format required by the analyzer and viceversa.

This has the drawback that each piece of text to be analyzed must be translated to the appropriate format, the analyzer must be initialized (which may be costly), and the results must be converted back and stored. This procedure must be followed *for each* analyzer one wants to run.

This solution may be fast and cheap to develop, since the pre-existing analyzers are used as black boxes and don't need to be re-engineered, but since those analyzers are usually designed as stand-alone programs receiving an input and producing an output, this approach implies a large overhead in the execution time and storage space of the resulting system.

Since most nowadays NLP applications are critical in time and/or storage space, either because they process huge amounts of text (as in the case of IR indexing engines) or because they are interactive (i.e. dialog systems, online translation, ...), language analyzers must perform only the minimum number of format conversions, disk I/O operations, and costly initialization procedures.

This may be achieved if pre-existing analyzers are (re)designed as software objects (functions, methods, servers, ...) which may be called or invoked from inside other software, instead of as stand-alone black-box programs.

Summarizing, a trade-off between reusability and efficiency must be reached, and we think that a reasonable investment in re-engineering linguistic processors as *embeedable* software components enables a faster development of efficient NLP applications based on those analyzers and eases the maintenance of the analyzers.

3. A Client-Server vision

In this section, the solution we have adopted after several years of experience developing and using language analyzers is presented. We think that it reduces the cost required to adapt or integrate the analyzers in new NLP applications, keeping a good efficiency level.

Our current system is an evolved version of the analyzers presented in (Carmona et al., 1998). We use a client-server architecture, in which NLP applications are seen as having two layers: A basic linguistic service layer which provides analysis services (morphological, tagging, parsing, ...), and an application layer which, acting as a client, requests services from the analyzers.

In this scenario, integrating the basic analyzers in a new NLP application is reduced to three simple steps:

- Convert the data from application internal representation to the data structures required by the service interface.
- Call the service and obtain the results.
- Convert the results to the application internal representation.

The advantages of this architecture are:

- It enables to use the analyzer as a function call from any NLP application, not as a separate software package.
- The clients requesting analysis services may be not only NLP applications, but also other service-providing modules (e.g. a parsing module might request a PoS tagging service). This enables the construction of increasingly more complex language analysis servers.
- It becomes unnecessary to define data interchange formats between analyzers. Each application can choose its own representation, provided it knows how to map it to the necessary data structures or parameters when requesting a service.
- Conversions are performed between application data structures and servers data structures, dramatically reducing the overhead caused by the reading, writing, parsing, and transmitting of text-based representations such as XML, SGML, etc. (note that this doesn't mean than the NLP application can not use XML, only that it doesn't need to internally work with it)
- The linguistic processors do not need to be initialized for each piece of text to be analyzed.
- The application may decide how and when to invoke each analyzer, and on which text segment (i.e. there is no need of a whole-text pipelined processing).

This Client-Server approach could be implemented in any Object-Oriented language, but if pre-existing processors are developed in different languages, the re-engineering cost would soon become to high. Fortunately, current software engineering technology provides us with means to integrate software objects developed in different languages, and even, running in different computers. This is further discussed in the next section.

4. Distributed architecture

The next step after the client-server vision of NLP presented above consists of having our clients and servers interact through some standard distributed object middleware, which will allow a client object to access a server object developed in any other language and running on any other computer or operating system.

There are several such engineering tools, such as SOAP or CORBA, and although the most suitable may depend on the kind of analysis services to be offered, the kind of applications to be developed, and the exploitation environment of these applications, any of them should serve the purpose of having a comfortable and efficient development and exploitation environment.

In our case, we relied on CORBA (Common Object Request Broker Architecture) standard (OMG, 1991) to manage that interaction. The architecture is briefly described in the next section.

4.1. CORBA

As detailed in the OMG web page[1], the *Common Object Request Broker Architecture*, is the Object Management Group's answer to the need for interoperability among the rapidly proliferating number of hardware and software products available today. CORBA allows applications to communicate with one another no matter where they are located or who has designed them.

The *Object Request Broker* (ORB) is the middleware that establishes the client-server relationships between objects. Using an ORB, a client can transparently invoke a method on a server object, which can be on the same machine or across a network. As seen in Figure 1, the ORB intercepts the call and is responsible for finding an object that can implement the request, pass it the parameters, invoke its method, and return the results. The client does not have to be aware of where the object is located, its programming language, its operating system, or any other system aspects that are not part of an object's interface. In so doing, the ORB provides interoperability between applications on different machines in heterogeneous distributed environments and seamlessly interconnects multiple object systems.

In fielding typical client/server applications, developers use their own design or a recognized standard to define the protocol to be used between the devices. Protocol definition depends on the implementation language, network transport and a dozen other factors. ORBs simplify this process since the protocol is defined through the application interfaces via a single implementation language-independent specification, the *Interface Definition Language* (IDL).

ORBs allow the integration of existing components. Developers simply model the legacy component using the same IDL they use for creating new objects, then write code that translates between the standardized and the legacy interfaces.

[1]See http://www.omg.org

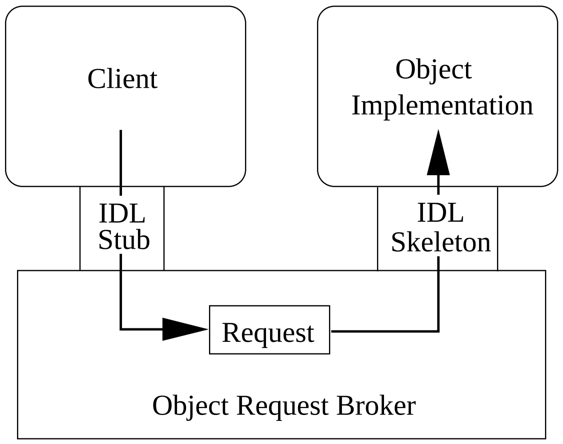

Figure 1: Client-server interaction via CORBA

4.2. Advantages of a standard

The advantages of using a standard architecture for the middleware of NLP applications are:

- Applications may be distributed over a network, increasing their efficiency, since several tasks may be performed in parallel.
- Several instances of the same service may be activated if necessary.
- Clients and servers may be written in different programming languages and run on different machines, which may run under different operating systems,
- CORBA specifications are open standards, and many free implementations can be found. We used ORBacus[2] and COPE[3].

The described advantages have a price that must be paid for. The main drawback is that the API of each analysis server are predefined, that is, the clients must adapt to the server, and the provided service may not cover exactly the client's needs. Nevertheless, this is a drawback that is also present in the wrapper approach, where the application must use the analyzer as a black box. In addition, the object oriented architecture makes it possible to build new servers via inheritance of existing ones, easing the development of modules which provide the exact kind of service required by the application.

5. The TALP-CLiC Language Processors

In this section we describe the particular framework we have adopted for the development of natural language analyzers and applications. Our system is developed by reseachers from *TALP research center* (Universitat Politècnica de Catalunya) and from *Centre de Llenguatge i Computació* (Universitat de Barcelona).

The engineering methodology followed is that of the Object Oriented design. Following some basic principles of Object Oriented methodology the analyzers become flexible, reusable and easy to distribute.

[2]http://www.orbacus.com

[3]http://www2.lunatech.com/research/corba/cope/

5.1. The Analyzer Abstraction

An analyzer can be seen as a software component -or object- that performs a language processing task at some level. Our architecture is based on the design of data classes which define the objects that hold and structure the information involved in the language processing. Up to now, our architecture includes two processing levels, defined by the processed data object:

Word. Basic object representing a word in a text. It holds the word from, its morphological information (type of word, tokens, etc.) and morhposyntactic information, namely the list of possible morphological analysis ([lemma, POS tag] pairs) and the disambiguated candidate.

Sentence. Object representing a sequence of words which form a sentence. Holds the sequence of words and a syntactic tree.

As usual, a class hierarchy defines the classes in the architecture (e.g.. words, sentences, syntactic trees) and the relations which structure the objects (e.g. a syntactic tree is linked to a sentence). Each data class provides an interface for accessing and modifying the data held by the objects of the class.

Under this framework, an analyzer is seen as an object that provides a task on a processing level. For example, a POS tagger disambiguates the morphological analysis of a word, or a parser generates the syntactic tree of a sentence. Therefore, in this setting, an analyzer is an object which provides methods to analyze data objects through the interface of an analyzer class.

Since data classes provide the linguistic structure to be held, and the analyzer objects use this structure to get and set the data, it is crucial that the design of data classes is general and rich enough to suit the needs of the analyzers in the architecture. A proper design of the data classes, with abstract access to the data, is required to ensure the incorporation of further language structure without affecting the design of the analyzers depending on such data classes.

While most of the language tasks may be specified with a simple interface, few of them require a simple computation on the data. Usually, the output of an analyzer is the result of a complex process of computation and reasoning which requires internal functionalities, the use knowledge and heuristics, and non-trivial parameter settings.

On the one hand, the functionalities that an analyzer requires can often be useful not only to the main task provided by the analyzer but also to other components in the architecture (e.g. a specific wrapper may require the use of the temporal expressions patterns of a morphological analyzer, or the mechanism of a chart parser may be used with arbitrary grammars, rather than specific syntactic grammars). On the other hand, the processing of an analyzer usually can be abstracted from the knowledge, and processign parameters may be set up under a general criterion (e.g. an analyzer abstracted from the language-dependent knowledge may be used for processing several languages or restricted subsets of a language considering specific terminology, or statistical-based analyzers can usually be tuned to meet an optimal criterion in the precision/coverage trade-off).

For the sake of flexibility and reusability, it is crucial that the interface of an analyzer class provides access to the internals of the analysis process. We distinct three types of methods an analyzer should provide:

- Analysis. Provide the task of the analyzer as a black-box, in a simple way.
- Setting. Allow to change the default behaviour of the analyzer.
- Functionalities. Provide access to the specific internal functionalities that are present in the task of the analyzer. Their use may be complex and may require knowledge on the internal data structures of the object.

Our general approach consists in putting special effort in the design of the class hierarchy which defines both the analyzer classes and the data classes which hold the data across the language processing. Analyzer objects must rely only on the language data objects, and the processing functionalities must be independent of any external processing and accessible from the outside through a rich interface. Standard software engineering must be used to achieve this goal.

5.2. Particular Analyzers

In this section we briefly describe the analyzers in our architecture. We distinguish two types of analyzers, according to their nature: primitive and shell analyzers.

Primitive Analyzers. Low-level analyzers, which perform the basic language tasks.

Tokenizer. Receives raw text and returns a sequence of words.

Morphological analyzer (MACO). Given a sequence of words, adds to each one a list of morphological analysis. It also recognizes and joins multi-word forms in a single word. It is composed by specific analyzers -such as a form dictionary search, recognizers for dates, numbers, monetary and percent quantities, suffixed forms, multiword compounds, etc. Details can be found in (Carmona et al., 1998)

POS *Taggers (Relax, TreeTagger)*. Receive word sequences and disambiguate its POS, based on the context of the word. See (Padró, 1998; Màrquez, 1999) for details.

Named Entity recognition (NAME). Given a sequence of words, recognizes and classifies name entities.

Sentence Splitter. Receives a sequence of words and returns a sequence of sentences.

Full parsing (TACAT). Chart parser for full syntactic parsing. Details in (Atserias and Rodríguez, 1998).

Partial Parser. Identifies partial parsing structures, namely chunks and clauses of a sentence.

Shell Analyzers. A shell analyzer offers high level functionalities, and, rather than performing the tasks, composes the functionalities of low-level analyzers of the architecture.

The main purpose of shell analyzers is the encapsulation of the standard processing involving several primitive analyzers into class methods. In this way, standard processing becomes easy and both applications or higher level analyzers can make use of it as a black-box.

In our system, we have designed only a shell analyzer which carries all the standard high-level processing, namely the *MorphoSyntax* analyzer. It provides analysis methods for tokenizing raw text and processing language both at word and sentence levels. The analyzer is parametrizable for choosing language, skipping processing levels, or selecting the method for a particular task, when several analyzers are available.

5.3. Applications

Final applications which require language processing are built on the top of analyzers, requesting their services and making particular use of the processed material.

Since analyzers are defined in classes, an application can follow two main trends for their explotation, depending on the needs:

- Direct instantiation, as with arbitrary software libraries. The application may instance primitive analyzers (Figure 2) or a shell analyzer (Figure 3) depending on its needs.
- Client-Server scheme, building a separate CORBA server which provides the analyzers functionalities, as ilustrated in Figure 4

In all cases, standard software methodologies are followed.

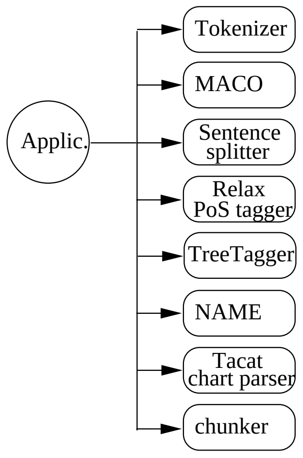

Figure 2: Direct use of the primitive analyzers from an application

The easiness of adaptation of analyzers into an application depends again on the particular needs of the application. Standard processing is direct through shell analyzers. Non-conventional processing will require direct access to low-level analyzers and finer control of the process.

As an example, our basic application is the *Standard Language Processor*. It is a distributed client-server application providing standard language processing facilities at different contexts. The server is just a CORBA interface to the *MorphoSyntax* analyzer. Up to now, four clients use this server:

- Basic *ms-analyze*, a command-line tool which processes plain text and produces a simple verticalized output.

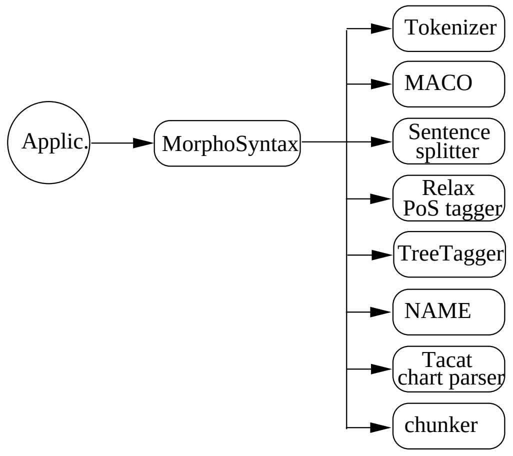

Figure 3: Use of the shell analyzer from an application

- *xms-analyze* deals with XML encoded text, using our particular DTD for basic language structure
- *NAMIC SpLP* deals with XML encoded text, using the specific DTD for NAMIC project (Basili et al., 2001).
- A CGI program provides a demo of our analyzers through the web[4].

In each case, the client is just a simple program which extracts the data from its corresponding source of data into the language data objects, requests an analysis service to the server and returns the result back to the corresponding output. The architecture is presented in Figure 4.

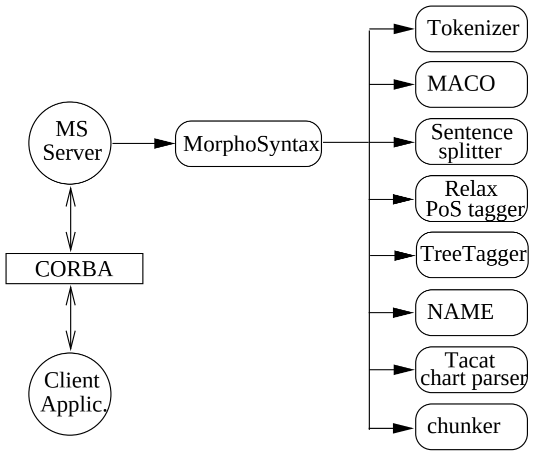

Figure 4: Use of the shell analyzer from an application via corba

But the availability of the analyzers as reusable software components enables the development of higher-level applications, as for instance an authoring program for writing news in a web-based environment, which may require language services for automatically detecting and linking named entities. Moreover, another sophisticated service may be the automatic detection of relations between entities, which would require partial syntactic parsing as a preprocess to the inference of entity relations. The application could build a sever offering the requested services, just by selecting those analyzers involved in the tasks, and providing analysis services as compositions of the analyzers methods. Moreover, a data object in the authoring could link language data objects and formatting information of the text in an inherited class.

[4]http://www.lsi.upc.es/˜nlp

6. Conclusions

We have presented a client-server architecture to develop NLP systems which require basic language analysis services.

An Object Oriented paradigm is used to design and implement the language analyzers, obtaining great felixibility to reuse the NLP processors from any application. When managed through some standard middleware such as CORBA, this approach enables to distribute NLP applications on a network.

A further step may include having permanently running language analysis servers that may be called by any client application. This is client-server interaction may be moved up to the Internet, providing linguistic analisys services to any NLP application running in the net and constituting a useful mechanism to share resources between reseach groups, or even becoming a commercial service.

7. References

J. Atserias and H. Rodríguez. 1998. TACAT: TAgged Corpus Analizer Tool. Technical Report LSI-98-2-T, Departament de LSI. Universitat Politècnica de Catalunya.

R. Basili, R. Catizone, L. Padró, M.T. Pazienza, G. Rigau, A. Setzer, N. Webb N., Y. Wilks, and F. Zanzotto. 2001. Multilingual Authoring: the NAMIC Approach. In *Proceedings of the ACL Workshop "Human Language Technology and Knowledge Management"*.

J. Carmona, S. Cervell, L. Màrquez, M.A. Martí, L. Padró, R. Placer, H. Rodríguez, M. Taulé, and J. Turmo. 1998. An Environment for Morphosyntactic Processing of Unrestricted Spanish Text. In *Proceedings of the 1st International Conference on Language Resources and Evaluation, LREC*, pages 915–922, Granada, Spain, May.

H. Cunningham, Y. Wilks, and R. Gaizauskas. 1996. GATE - a General Architecture for Text Engineering. In *Proceedings of 16th International Conference on Computational Linguistics, COLING*, Copenhagen, Denmark.

R. Grishman. 1996. The TIPSTER Text Phase II Architecture Design, Version 2.2. Technical Report, New York University.

L. Màrquez. 1999. *Part–of–Speech Tagging: A Machine–Learning Approach based on Decision Trees*. Phd. Thesis, Dep. Llenguatges i Sistemes Informàtics. Universitat Politècnica de Catalunya.

OMG. 1991. Common Object Request Broker Architecture. Technical Document, Object Management Group. http://www.omg.org, http://www.corba.org.

L. Padró. 1998. *A Hybrid Environment for Syntax–Semantic Tagging*. Phd. Thesis, Dep. Llenguatges i Sistemes Informàtics. Universitat Politècnica de Catalunya, February. http://www.lsi.upc.es/˜padro.