Ellogon and the challenge of threads

Georgios Petasis

Software and Knowledge Engineering Laboratory,
Institute of Informatics and Telecommunications,
National Centre for Scientific Research "Demokritos",
Athens, Greece
petasis@iit.demokritos.gr

Abstract

This paper is about the Ellogon language engineering platform, and the challenges faced in modernising it, in order to better exploit contemporary hardware. Ellogon is an open-source infrastructure, specialised in natural language processing. Following a data model that closely resembles TIPSTER, Ellogon can be used either as an autonomous application, offering a graphical user interface, or it can be embedded in a C/C++ application as a library. Ellogon has been implemented in C/C++ and Tcl/Tk: in fact Ellogon is a vanilla Tcl interpreter, with the Ellogon core loaded as a Tcl extension, and a set of Tcl/Tk scripts that implement the GUI. The core component of Ellogon, being a Tcl extension, heavily relies on Tcl objects to implement its data model, a decision made more than a decade ago, which poses difficulties into making Ellogon a multi-threaded application.

1 Introduction

In this paper we describe Ellogon [Petasis et al., 2002; 2003], a multi-lingual, cross-platform, general-purpose language engineering environment, developed in order to aid a wide range of users – from researchers in the natural language field or computational linguistics, to companies that produce and deliver language engineering systems. Being in constant development since 1998, Ellogon is a mature and well tested infrastructure, as it has been used in many Greek and European funded projects. Its facilities have been used for creating a wide range of applications, from multilingual information extraction systems to controlled language checkers and ontology-based information extraction systems.

Ellogon as a text engineering platform offers an extensive set of facilities, including tools for visualising textual/HTML/XML data and the associated linguistic information, support for lexical resources (like creating and embedding lexicons), tools for creating annotated corpora, accessing databases, comparing annotated data, or transforming linguistic information into vectors for use with various machine learning algorithms. Additionally, Ellogon offers some unique features, like the ability to freely modify annotated textual data (with Ellogon automatically applying the required transformations on the associated linguistic information), and the ability to create stand-alone applications with customised user interfaces that perform specific tasks.

Ellogon was amongst the first platforms to offer complete multi-lingual support, both in its processing core, and its GUI. The main reason behind this feature was, of course, the Unicode support offered by Tcl 8.1, which was available in a beta form when Ellogon development started, along with the decision to use Tcl objects as a basis for the data model of Ellogon. The idea for relying the data model of Ellogon on Tcl objects was a promising approach, about a decade ago, when Ellogon development was starting. The data model could be implemented by objects holding Tcl lists, indexed by Tcl Hash tables. The modularity and reference counting of Tcl objects ensured reusability, and no conversions were needed in order to expose the data model to Tcl. However, modularity and reusability comes with a small cost: each Tcl object has an overhead in the used memory, and processing a document can produce tenths of thousands of objects. But again a solution was found, based on the observation that objects tend to contain the same information: build a cache of objects, and reuse objects whenever appropriate. This approach worked well for many years, but recent hardware brings a new challenge: how can the data model meet multiple threads?

The rest of the paper is organised as follows: in chapter two, the architecture and data model of Ellogon are described, along with some implementation details. Chapter three presents the objects cache, the motivation behind this cache, the advantages the cache offers with respect to memory usage, and the issues raised by this cache in a multi-threaded environment. Finally chapter four concludes this document.

2 The Architecture and Data Model of Ellogon

2.1 Ellogon Architecture

Ellogon aims to be a versatile infrastructure that can be used either as an NLP integrated development environment (IDE), or as a library that can be embedded to foreign applications. As a result, Ellogon proposes a modular architecture with four independent subsystems (Figure 1):

- A highly efficient core developed in C, which implements an extended version of the TIPSTER data model. Its main responsibility is to manage the storage of the textual data and the associated linguistic information, and to provide a well-defined programming interface (API) that can be used in order to retrieve/modify the stored information.
- An object oriented C++ API, which increases the usability of the C core API. This object oriented API is exposed to a wide range of programming languages, including C++, Java, Tcl, Perl and Python.
- An extensive and easy to use graphical user interface (GUI). This interface can be easily tailored to the needs of the end user.
- A modular pluggable component system. All linguistic processing within the platform is performed with the help of external, loaded at run-time, components. These components can be implemented in a wide range of programming languages, including C, C++, Java, Tcl, Perl and Python.

2.2 Ellogon Data Model

Ellogon shares the same data model as the TIPSTER architecture (Figure 2). The central element for storing data in Ellogon is the Collection. A collection is a finite set of documents. An Ellogon document consists of textual data, as well as linguistic information about the textual data. This linguistic information is stored in the form of attributes and annotations.

An attribute associates a specific type of information with a typed value. An annotation, on the other hand, associates arbitrary information (in the form of attributes) with portions of the textual data. Each such portion, named span, consists of two character offsets denoting the start and the end characters of the portion, as measured from the first character of some textual data. Annotations typically consist of four elements:

- A numeric identifier. This identifier is unique for every annotation within a document, and can be used to unambiguously identify the annotation.
- A type. Annotation types are textual values that are used to classify annotations into categories.

- A set of spans, each of which denotes a range of annotated textual data.
- A set of attributes. These attributes usually encode the necessary linguistic information.

A visual representation of Ellogon's data model can be seen in Figure 2.

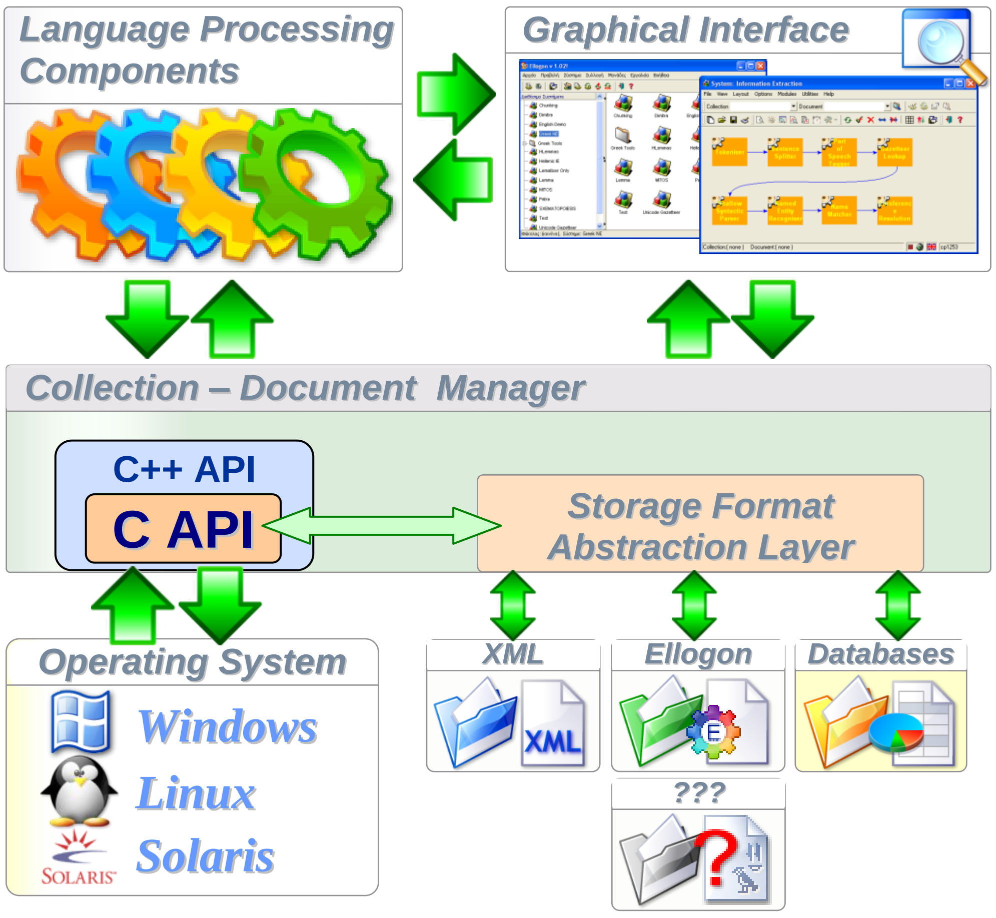
Language Processing Components
Graphical Interface
Collection – Document Manager
C++ API
C API
Storage Format Abstraction Layer
Operating System
Windows
Linux
Solaris
XML
Ellogon
Databases
???

Figure 1: Ellogon Architecture.

2.3 Implementing the Data Model

The top-level object of Ellogon's data model is the collection. The collection has been implemented as a C structure, which contains a set of Tcl objects among other elements:

- A Tcl list object, containing the documents to be deleted (if any).
- A Tcl command token, holding the Tcl command that represents a collection at the Tcl level.
- A Tcl Hash table that contains the attributes of the collection. Each attribute is a Tcl list object.
- Two Tcl objects that can hold arbitrary information, such as notes and associated information.

Additionally, the collection maintains a list of documents. A document is implemented as a C structure, and has similar features to the collection, like a Tcl command token, a Tcl object holding the text of the document, and two Tcl Hash tables, storing the attributes and annotations of the document. Both attributes and annotations are implemented as Tcl objects. Each attribute is a Tcl list object, which contains three elements:

- The attribute name: the name can be an arbitrary string.

- The type of the attribute value: this can be an item from a predefined set of value types.
- The value of the attribute, which can be an arbitrary (even binary) string.

An annotation is a slightly more complex type, again represented by Tcl object of custom type. It can be roughly seen as a list of four elements:

- The annotation id: an integer, which uniquely identifies the annotation inside a document.
- The annotation type: an arbitrary string that classifies the annotation into a category.
- A list of spans: each span is a Tcl list object, holding two integers, the start/end character offsets of the text annotated by the span.
- A list of attributes: a Tcl list object, whose elements are attributes.

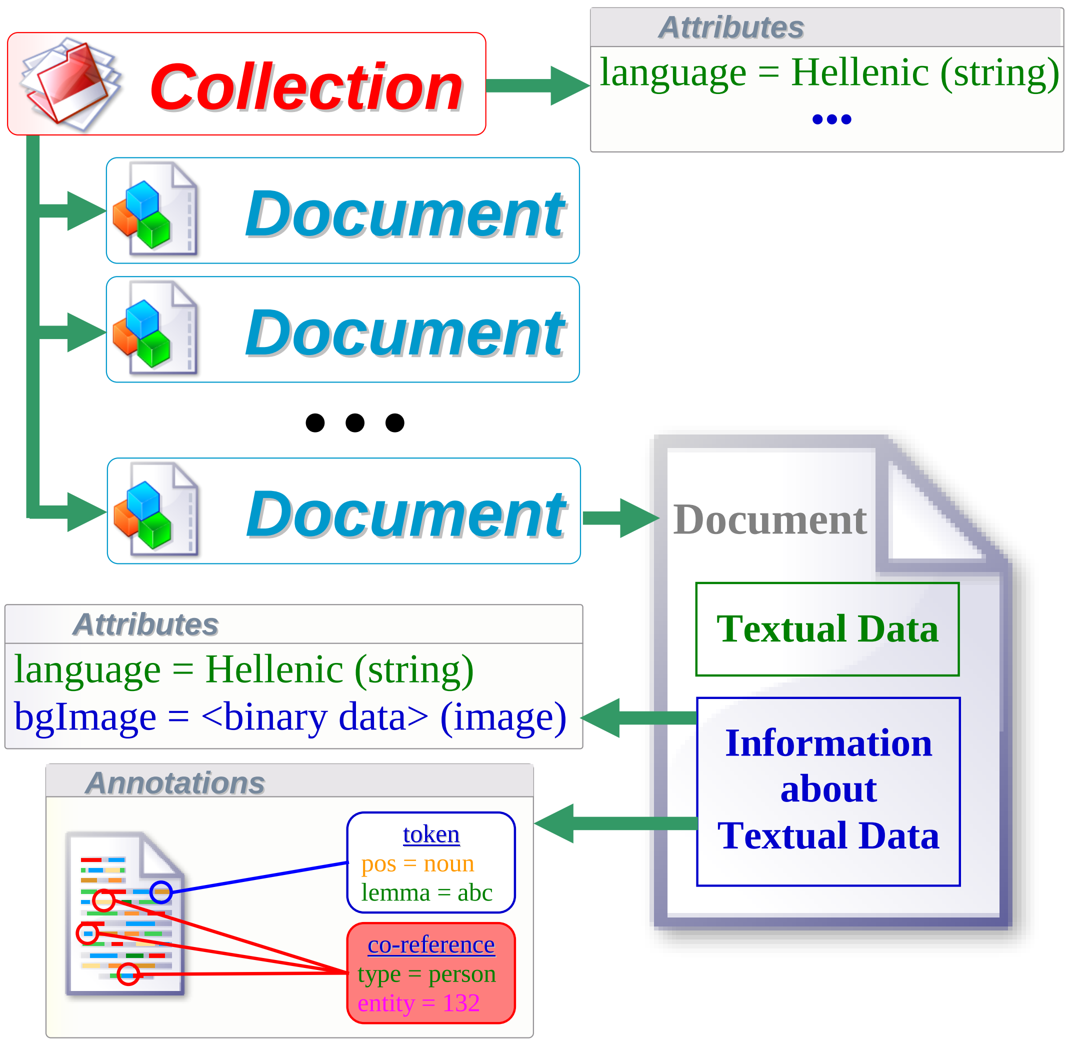

Figure 2: The Data Model of Ellogon.

3 Caching Tcl objects

Memory consumption is always an issue in Ellogon, as it directly limits the size and the number of documents that can be processed. Ellogon has chosen to implement a global memory cache for Tcl objects, containing information from all opened Collections and Documents. The motivation behind the creation of this cache was a simple observation that linguistic information tends to repeat a lot. Consider for example a part-of-speech tagger, analysing a document containing 10.000 words. Each word will be assigned an annotation of type “token” (so the annotation type “token” will be repeated 10.000 times), and each annotation will contain at least an attribute, named “pos”, that will hold the part-of-speech tag of the word. Assuming that a tag set of 10 possible part-of-speech categories is used, each tag has a potential repetition in the thousands. Keeping a cache, so as not to repeat “token’ and “pos” 10.000 times, but share 10.000 times two Tcl objects, makes sense. And caching and reusing larger clusters of Tcl objects makes even more sense, as the memory reduction can be drastically reduced. Memory reduction is even greater as more documents are added, and information is shared among collections. The current implementation of the object cache contains many heuristics that can cache and share a range of objects, from attribute names, annotation types or objects holding integers (i.e. used as identifiers or spans) to more complex constructs like spans or whole attributes.

3.1 Thread safety

Of course, a major concern involving a global caching mechanism, spanning the data model of a whole application, is how thread safety is ensured. Tcl objects cannot be shared across threads, constituting the approach followed so far thread unfriendly. This is an open question for the Ellogon platform, which may not have an easy answer, despite the fact that parallel processing of documents in different threads is a highly desirable feature.

The core component of Ellogon (Collection and Document Manager – CMD) that implements the data model (and the object cache) is already thread-safe:

- The global variables/objects are few, and their access is protected by mutexes.
- The object cache is global, and protected again with a mutex.
- Ellogon plug-in components use thread-specific storage for storing their “global” variables (through special pre-processor definitions for C/C++ components).

However, the fact that the core is thread safe does not mean that threads can be used: as long as a cached object is passed back to Tcl, it is no longer protected by the mutexes the core uses: Tcl will assume that the object is local to the current thread the interpreter is running on, and will modify its reference count (at least) without any restriction.

3.2 So, can Ellogon become multi-threaded?

This is a difficult question to be answered, and the motivation for this paper. A multi-threaded Ellogon must fulfil the following requirements:

- The graphical user interface must not block during component execution, suggesting that it is running on its own thread.
- Each execution chain (a set of loaded at runtime components named as “System” in Ellogon terminology) must run on its own thread.
- Since each “System” executes all components on a single document before advancing into the next document, the documents of a collection can be distributed into N threads, and can be processed in parallel. This is a highly desired feature.

However, this threading design poses some difficulties in its implementation:

- The object cache: the object cache limits memory usage both within a single document, but also across documents. In the current implementation, Tcl objects are managed by the cache, which suggests that they are created in the “main” thread of the application. In case documents are created and used on different threads, the cached Tcl objects will be accessed by different threads, an operation that is not allowed by the Tcl threading model.

- The Graphical User Interface: the GUI is not limited to instrumentation of the component execution, but offers many operations on the linguistic data produced during component execution, including a wide range of viewers for browsing this information. Since the GUI runs on a separate thread than the documents were processed, again Tcl objects will be shared by multiple threads.

The Data Model of Ellogon has been designed to rely heavily on Tcl objects, as this seemed a good idea back in 1998, for the following reasons:

- Tcl objects are reusable bits, due to the reference counting and the well defined C API for their management. Structuring the data model upon them, allows parts of the model to be cached and shared in multiple documents, as it has been implemented by the object cache.
- The GUI of the application is in Tcl/Tk, and the linguistic data does not need to be converted to Tcl objects, in order to be displayed/managed by the GUI, if it is already stored as Tcl objects.
- Despite that there is an extensive API for managing the various elements of the data model, it is far more convenient to use the Tcl commands to manipulate them, especially for plug-in components written in Tcl. For example, it is easier to get the annotations contained in an “annotation set” through “foreach”, than through a “for” with a counter, which calls “tip_Nth” to get the Nth element from the annotation set.

However, nowadays computers with multiple processing CPU cores are common, and supporting multiple threads seems a natural extension. But, it is unclear if such an extension is feasible without a major re-implementation of the core component and without sacrificing the significant memory reduction offered by the object cache.

4 Conclusions

This paper describes a language engineering platform known as “Ellogon”, one of the first general-purpose language engineering infrastructures to offer complete Unicode support. Being in constant usage for more than a decade, the platform has to continuously evolve in order to better exploit a constantly improving hardware, without introducing major and incompatible changes to the infrastructure. Since CPUs with multiple processing cores are becoming more affordable and thus wide spread, the introduction of multiple processing threads in the Ellogon platform is a major concern. However, it is currently unclear if multiple threads can be introduced without a significant reorganisation of the platform, a question that is the main motivation for this paper.

5 References

[Grishman, 1996]: Grishman, R. 1996. “TIPSTER Architecture Design Document Version 2.2”. *Technical Report, DARPA*. Available at http://www.tipster.org

[Petasis et al., 2002]: G. Petasis, V. Karkaletsis, G. Paliouras, I. Androutsopoulos and C. D. Spyropoulos, “Ellogon: A New Text Engineering Platform”. In Proceedings of the 3rd International Conference on Language Resources and Evaluation (LREC 2002), Las Palmas, Canary Islands, Spain, vol. I, pp. 72 – 78, May 2002.

[Petasis et al., 2003]: G. Petasis, V. Karkaletsis, G. Paliouras and C. D. Spyropoulos, “Using the Ellogon Natural Language Engineering Infrastructure”. In Proceedings of the Workshop on Balkan Language Resources and Tools, 1st Balkan Conference in Informatics (BCI’2003), Thessaloniki, Greece, November 21, 2003.