Carrot^2: Design of a Flexible and Efficient Web Information Retrieval Framework

Stanisław Osiński and Dawid Weiss

Institute of Computing Science, Poznań University of Technology,
ul. Piotrowo 3A, 60–965 Poznań, Poland,
Dawid.Weiss@cs.put.poznan.pl

Abstract. In this paper we present the design goals and implementation outline of Carrot^2, an open source framework for rapid development of applications dealing with Web Information Retrieval and Web Mining. The framework has been written from scratch keeping in mind flexibility and efficiency of processing. We show two software architectures that meet the requirements of these two aspects and provide evidence of their use in clustering of search results.
We also discuss the importance and advantages of contributing and integrating the results of scientific projects with the open source community.

Keywords: Information Retrieval, Clustering, Systems Design.

1 Introduction

With a few notable exceptions, software projects rooting from academia are often perceived as useful prototypes, *spike-solutions* to use a software engineering term, that provide proofs for novel ideas but turn out to be unusable in production systems. In Carrot^2 we made an attempt to provide a useful, flexible and research-wise interesting system that can be efficient enough to satisfy real-life demands of commercial deployments. Two different software architectures coexist in the system: the XML-driven architecture is aimed at flexibility and ease of use, the local-interfaces architecture, developed later, targets the efficiency of processing.

Carrot^2 is mostly known for its Web search results clustering components, which successfully compete with commercial clustering solutions, such as Vivisimo or iBoogie (Carrot^2 was written at the same time Vivisimo was first released). The goal of this paper is to provide some insight into the internal architecture of Carrot^2 and to show that the applications of the framework are not limited to search results clustering only.

2 Goals, design assumptions and requirements

The primary goal of Carrot^2 was to enable rapid research experiments with novel text/web mining techniques. To minimize the effort involved in implementation and evaluation of a new algorithm, Carrot^2 provides ready-to-use implementations of the most common text processing tasks, such as:

– an efficient JFlex-based (http://jflex.de/) text tokenizer,
– trigram-based language identification [1]
– stopword filtering, stemming for 7 languages,
– search engine interfaces (HTML scraping, API access),
– access to test collections, e.g. Open Directory Project data (http://dmoz.org),
– presentation of results (HTML rendering) and automatic quality measurements.

Additionally, Carrot^2 contains implementations of a number of search results clustering algorithms, including classic agglomerative techniques (AHC), K-means, fuzzy clustering [2], biology-inspired clustering [3], Suffix Tree Clustering (STC) [4] and Lingo [5].

To be truly useful in both research and production settings, Carrot^2 had to meet a number of requirements:

Component architecture The project should be a library; a set of *components* with clearly established communication interfaces and all the infrastructure needed to combine them into useful applications. Some of these applications should be provided as demonstration and proof-of-concept.

Flexibility Components should be relatively autonomous and easy to reconfigure and customize. That is, components can be taken out of the project and put into other software easily.

Language and OS independence It should be possible to reuse components for systems written in any language and working on any operating system.

High performance The infrastructure in which the components cooperate should impose as little additional overhead as possible. In other words: efficient components combined together should produce an efficient system.

Permissive licensing options Certain open source licenses impose rigorous restrictions on derivative works. The framework and any third party libraries it includes, should be covered by a permissive license that lets everyone use the framework, or its subcomponents in other software (commercial or open source).

3 Overview of the design and implementation

3.1 Framework's fundamental elements: components

Central to the architecture of Carrot^2 is the notion of a *component*. The task of an *input component* is to acquire, or generate data for further processing based on some query (usually typed by a human). Examples of input components include search engine wrappers, test collections or even components returning random data. *Filter components* transform the data in some way. Examples include text segmentation, stemming, feature extraction, clustering or classification. *Output components* are responsible for consuming the result of previous components. Output components usually present the result to the human user, but may also

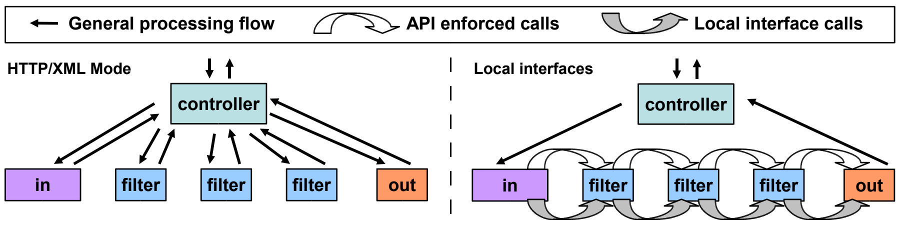

Fig. 1. Query processing in XML-based and local interfaces communication schemes.

process the result automatically as in benchmarking applications or tuning. A *controller component* combines other components into a *processing chain*: an ordered list of components where data obtained by an input component passes through a number of filtering components and is finally consumed by an output component.

3.2 Two architectures for component communication layers

Flexibility is usually achieved at the cost of performance and performance rarely goes along with flexibility. The mutually exclusive requirements are reflected in the framework's two different component communication layers: one design was aimed at language independence, component distribution and flexibility (*XML-based architecture*) the other was targeted at efficiency of processing (*local interfaces architecture*).

XML-based architecture In the XML-based architecture, components communicate solely using HTTP POST requests, exchanging custom XML messages (an approach similar to XML-RPC protocol). The communication is mediated by the controller component that knows the order of components to invoke from the current processing chain (see Figure 1). This communication scheme is characterized by the following features:

- components can be easily distributed – the controller component takes care of remote components' invocations, regardless of their physical location;
- components can be written in virtually any programming language that supports rudimentary elements of XML parsing and HTTP protocol;
- data-centric processing; components may not know how or where the data is produced. The only required information is the format of the input and output XML files;
- configuration and order of components in a processing chain takes place at the level of the controller component. This makes load balancing and component failover quite trivial to achieve.

Local interfaces architecture The XML-based architecture provides a great deal of flexibility with implementation and configuration of components. Alas, it also involves much cost in parsing/ serialization of XML files and network transfers. For production systems, an alternative solution had to be found.

We designed *local interfaces architecture* that stands for a very general concept of combining components using local method calls rather than network APIs. Note that from the viewpoint of the framework, nothing is known about these method calls – their signatures are not available for the framework until the components are assembled in a processing chain at runtime – this poses a very interesting design challenge. We have identified the following key criteria driving the local interfaces design:

Local method calls Local method calls are the key to achieving high performance. Data must not be passed via bounded buffers, but directly from component to component.

Memory/ object reuse Intense memory allocation/ garbage collection slows down any application by a factor of magnitude. The design must provide means to reuse intermediate component data from request to request.

Incremental pipelines Components may not need all of their predecessors' data at one time. Data should be passed between components as soon as possible.

Flexible data types Components should be able to exchange any data they need (using local method calls). The framework should provide means for this to happen with no extra overhead.

We separated the communication between components into *system* and *application-related* method calls. System-related calls are defined at the core Carrot^2 level. They include component lifecycle management methods and request-lifecycle methods; all components must implement these. Application-related method calls are unspecified, the components must perform an initial 'handshake' to establish their compatible methods.

At runtime, each processing chain is assembled dynamically by the controller component in the following way: each component knows its direct successor in the processing chain and itself expects data from its predecessor. Each component also declares its *capabilities* and capabilities it requires from the predecessor and successor component. A processing chain is successfully assembled only if capabilities of all components are pairwise compatible.

Capabilities are usually used to denote data-specific interfaces (with arbitrary method signatures) to perform a narrowing cast from an abstract component to a specific required type that lets the components communicate directly. For example, a component declaring `RawDocumentConsumer` capability may also declare a Java method `void nextDocument(RawDocument document)` that would accept a new document from its predecessor. The predecessor component, knowing its successor must be a raw document consumer, will simply cast the successor's object reference to a known interface and invoke the data-specific method `nextDocument` repeatedly for each new incoming document. This 'custom' communication between components is depicted as gray arrows in Fig. 1.

Each processing chain is assembled only once for all queries, so the casting and verification of capabilities overhead is minimal. After that, everything is already known and configured – almost no overhead at all is imposed by the framework at runtime. This makes local method calls extremely fast.

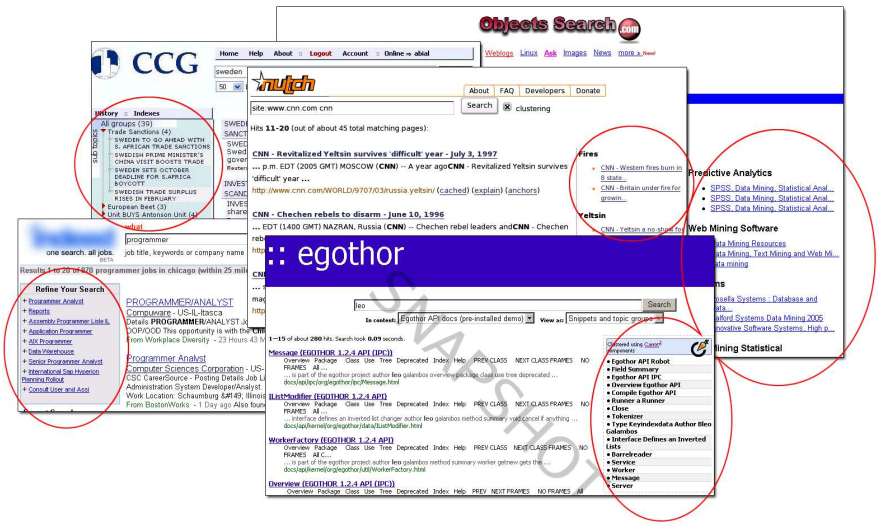

Fig. 2. Screenshots from commercial and open source software using Carrot^2 clustering and linguistic components (marked with red circles).

4 Examples of practical deployment and use

We developed Carrot^2 to be a generic framework, but we also provided several implementations of components serving for clustering of search results: clustering algorithms, input (search engine wrappers) and output (XML/XSLT generators) components. Shortly after publishing the framework, we received a great deal of positive feedback from the research community. New research projects and papers were based on the foundation of the Carrot^2 architecture: an ant-colony document clustering algorithm [3] or a rough set approach to clustering documents from the Web [2]. Applications reusing certain components of the system were presented [6]. The framework was used as a testbed for cross-comparison of existing algorithms [7].

We were equally pleased to observe substantial commercial interest in Carrot^2 and its selected components (see Figure 2 for screenshots of systems that somehow integrated Carrot^2 components). The project's clustering components (with local controller components) were also swiftly integrated in other open source projects: Lucene (http://jakarta.apache.org/lucene), Nutch (http://www.nutch.org) and Egothor (http://www.egothor.org).

5 'Open Sourcing' academic software

From the very beginning, development of Carrot^2 followed the principles of open source software. The project is licensed under very permissive BSD license and hosted at SourceForge (http://carrot2.sourceforge.net). Communication among the developers and support for users community is provided through a mailing list. Public CVS access to source code and continuous integration facility (nightly builds and a demo) are also provided (http://carrot.cs.put.poznan.pl).

Our experience with Open Sourcing the software has been very positive. We especially appreciate broad interest and support from the user community – both academic and commercial. Releasing academic software as open source helps to confront it with real requirements and expectations of Web users. It also helps to make the software last longer and gain a wider audience by integration with other Open Source products – something the community is more than willing to undertake if there are evident gains from such fusions.

6 Summary and conclusions

We have presented requirements and two different architectures for an efficient and flexible component-based software framework for simplifying the development of Web information retrieval and data mining applications. The presented ideas have been implemented and published as an open source project that spawned other research and commercial projects.

Acknowledgement. The authors would like to thank to all the committers, supporters and users of the Carrot^2 project. This research has been supported by internal university grant.

References

1. Grefenstette, G.: Comparing two language identification schemes. In: Proceedings of the 3rd International Conference on Statistical Analysis of Textual Data. (1995)
2. Lang, H.C.: A tolerance rough set approach to clustering web search results. Faculty of Mathematics, Informatics and Mechanics, Warsaw University (2004)
3. Schockaert, S.: Het clusteren van zoekresultaten met behulp van vaagmieren (clustering of search results using fuzzy ants). Master thesis, University of Ghent (2004)
4. Zamir, O.: Clustering Web Documents: A Phrase-Based Method for Grouping Search Engine Results. PhD thesis, University of Washington (1999)
5. Osiński, S., Stefanowski, J., Weiss, D.: Lingo: Search results clustering algorithm based on Singular Value Decomposition. In Kłopotek, M.A., Wierzchoń, S.T., Trojanowski, K., eds.: Proceedings of the International IIS: Intelligent Information Processing and Web Mining Conference. Advances in Soft Computing, Zakopane, Poland, Springer (2004) 359–368
6. Jensen, L.R.: A reuse repository with automated synonym support and cluster generation. Department of Computer Science at the Faculty of Science, University of Aarhus, Denmark (2004)
7. Osiński, S.: Dimensionality reduction techniques for search results clustering. MSc thesis, University of Sheffield, UK (2004)