CLAM: Quickly deploy NLP command-line tools on the web

Maarten van Gompel
Centre for Language Studies (CLS)
Radboud University Nijmegen
proycon@anaproy.nl

Martin Reynaert
CLS, Radboud University Nijmegen
TiCC, Tilburg University
reynaert@uvt.nl

http://proycon.github.io/clam

Abstract

In this paper we present the software CLAM; the Computational Linguistics Application Mediator. CLAM is a tool that allows you to quickly and transparently transform command-line NLP tools into fully-fledged *RESTful* webservices with which automated clients can communicate, as well as a generic webapplication interface for human end-users.

1 Introduction

In the field of Natural Language Processing, tools often come in the form of command-line tools aimed at UNIX-derived systems. We consider this good practice in line with the UNIX philosophy (McIlroy et al., 1978) which states, amongst others, that programs should 1) do one thing and do it well, and 2) expect the output of one program to be the input of another. This can be rephrased as the *Rule of Modularity*: write programs consisting of simple parts, connected by well-defined interfaces (Raymond, 2004).

Programs operating at the command-line offer such modularity, making them ideally suitable for integration in a wide variety of workflows. However, the command-line may not be the most suitable interface for non-specialised human end-users. Neither does it by itself facilitate usage over network unless explicit server functionality has been programmed into the application. Human end-users often want a Graphical User Interface (GUI), a special instance of which is a Web User Interface. Yet for automated clients operating over a network, such an interface is a cumbersome barrier, and these instead prefer a properly formalised webservice interface. CLAM offers a solution to this problem, when all there is is a simple NLP command-line tool.

CLAM finds application in areas where people want to make their software available to a larger public, but a command-line interface is not sufficient. Setting up your tool may be complicated, especially if there are many dependencies or the target audience does not use Linux machines. CLAM is ideally suited for quick demo purposes, or for integration into larger workflow systems. It removes the burden from the software developer (you) to have to implement a server mode and build a GUI or web-interface, thus saving precious time.

2 System architecture

The Computational Linguistics Application Mediator (CLAM) is a tool that wraps around your command-line interface and allows you to very quickly and transparently turn your program into **1)** a *RESTful* (Fielding, 2000) webservice with which automated clients can communicate, as well as **2)** a generic web user interface for human end-users. Just like an actual clam is a shell around the animal that inhabits it, which most onlookers never see directly, CLAM wraps around your sofware, providing extra functionality and hardening it through its built-in security mechanism. You do not need to modify your original software in any way, it is always taken as a given, you merely need to describe it.

An NLP command-line tool can usually be described in terms of *input files*, *output files* and parameters influencing its run. Parameters may either be *global parameters*, pertaining to the system as a whole, or *local parameters* which act as *metadata* for specific input files. File formats are never dictated by CLAM itself, but are up to the service provider to define.

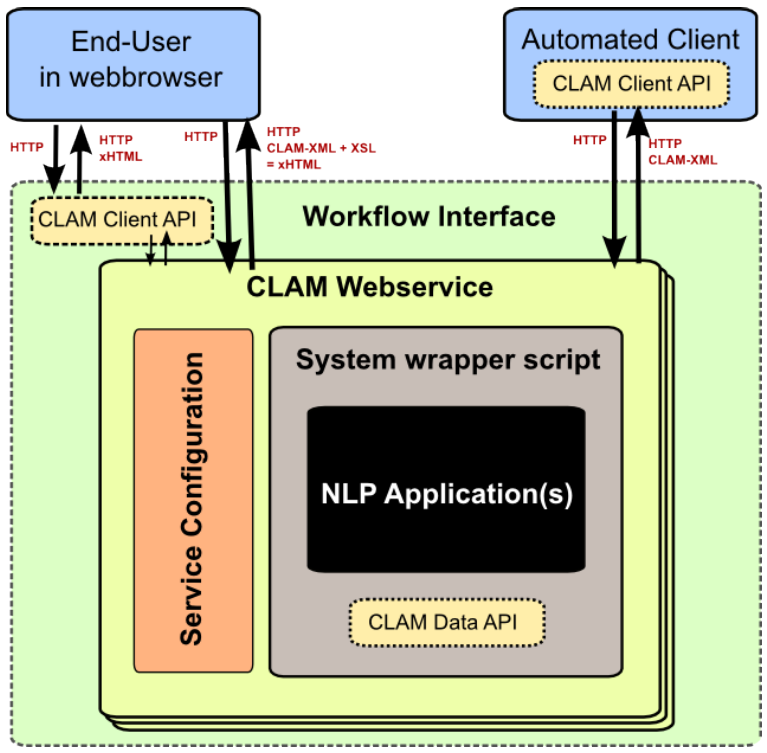

Figure 1: Schematic overview of the CLAM architecture

CLAM discerns three *states*, which also reflect the stages in which the end-user or automated client interacts with the system

1. The system is ready to accept files for input and input parameters
2. The system is running
3. The system is done and the output files are offered for presentation/download.

Any tool that can be described in these terms can be used with CLAM. The system has been designed specifically to work with software that may take quite some time to process or runs large batches. Stage two therefore is not confined to lasting mere seconds as is custom in web-based applications, but may last as long as hours, days, or any duration that the end-user is willing to wait. Also, end-users need not maintain a connection to the server. Human end-users may close their browser and return at will, and automated clients simply poll the system's status with a certain interval.

You are not limited to just a single run of your system; you may set it up to allow upload and processing of multiple files and run them in batch fashion. This approach is common in processing text files for purposes such as tokenisation or any form of tagging.

In order for CLAM to turn a command-line tool into a webservice, developers are expected to provide two things in addition to the actual tool:

1. **Service configuration** - This specifies everything there is to know about your application, it defines what the input will be, what the output will be, and what parameters the system may take. Input and output are always in the form of files, adhering to whatever format you desire. The web user interface, however, also optionally offers a text field for users to create files on the fly.

2. **System wrapper script** - This is a small script that CLAM will invoke to start your system. It acts as the glue between CLAM and your actual application and may do some necessary interpretation and transformation of parameters to suit the command-line interface of your application.

A generic client for communicating with the webservice is already provided, more specific clients can be written using the CLAM API (Python) to greatly facilitate development. The architecture of CLAM is schematically visualised in Figure 1.

CLAM is a multi-user system, although out-of-the-box it simply uses an "anonymous" user and requires no authentication. Each user can create an arbitrary number of *projects*. One project corresponds to one run of the system, which may be one large batch depending on how you configure your service. Users can always return to earlier projects and inspect input files and output files, until they explicitly delete the project.

2.1 Service Configuration

In the service configuration file, you specify precisely what kind of input goes into the system, and what kind of output goes out: this results in a deterministic and thus predictable webservice. With any input and output files, arbitrary metadata can be associated. For input files, metadata is created from parameters that can be set by users, these are rendered as input fields in the web interface. You can specify how this metadata is carried over to output files. Additionally, as part of the metadata, provenance data is generated for all output files. These are both stored in a simple and straightforward XML format.

All these definitions are specified in so-called *profiles*. A profile defines *input templates* and *output templates*. These can be seen as "slots" for certain filetypes and their metadata. A small excerpt of a profile for a simple translation system with some associated metadata is shown in Figure 2. A full discussion of its syntax goes beyond the scope of this paper, but is explained at length in the manual.

```
Profile(InputTemplate('maininput', PlainTextFormat,
  "Translator input: Plain-text document",
  StaticParameter(
    id='encoding',name='Encoding',description='The character encoding of the file',
    value='utf-8'
  ),
  ChoiceParameter(
    id='language',name='Language',description='The language the text is in',
    choices=[('en','English'),('nl','Dutch'),('fr','French')]),
  ),
  extension='.txt',
  multi=True
), OutputTemplate('translationoutput', PlainTextFormat,
    "Translator output: Plain-text document",
    CopyMetaField('encoding','maininput.encoding')
    SetMetaField('language','de'),
    removeextension='.txt',
    extension='.translation',
    multi=True
))
```

Figure 2: An excerpt of a fictitious *profile* for a simple translation system from English, Dutch or French to German. The attribute `multi=True` states that multiple files of this type may be submitted during a single run

Global parameters to the system are specified independently of any profiles. Consider a global parameter that would indicate whether or not want the fictitious translation system seen in Figure 2 to be case-sensitive, and take a look at the following example[1]:

```
PARAMETERS =  [
  ('Translation parameters', [
       BooleanParameter(id='casesensitive',name='Case Sensitivity',
       description='Enable case sensitive behaviour?')
])]
```

[1]Parameters are always grouped into named groups, "Translation parameters" is just the label of the group here

2.2 System Wrapper

Communication between CLAM and your command-line tool proceeds through a system wrapper script. The service configuration file defines what script to call and what variables, pre-defined by CLAM, to pass to it:

```
COMMAND = "mywrapper.py $DATAFILE $OUTPUTDIRECTORY"
```

This is then executed whenever a user runs a project. It is the job of the system wrapper script to invoke your actual application.

There are two main means of communicating the parameters to the system wrapper: one is to make use of the data file (`$DATAFILE`), which is an XML file that contains all input parameters. It can be parsed and queried effortlessly using the CLAM API, provided you write your wrapper script in Python. The other way, more limited, is to specify parameter flags for your global parameters[2] in the service configuration, and simply let CLAM pass all global parameters as arguments on the command line:

```
COMMAND = "mywrapper.pl $INPUTDIRECTORY $OUTPUTDIRECTORY $PARAMETERS"
```

By passing the input directory, the system wrapper script can simply look for its input files there.

3 Extensions

CLAM can be extended by developers in several ways. One is to write *viewers*, which take care of the visualisation of output files for a specific file format, and are used by the web user interface. Viewers may be implemented as internal Python modules, or you can link to any external URL which takes care of the visualisation. Another extension is *converters*, these allow users to upload an input file in one file type and have it automatically converted to another. Converters for PDF and Word to plain text are already provided through third party tools.

4 Technical Details

CLAM is written in Python (2.6 or 2.7), (van Rossum, 2007). It comes with a built-in HTTP server for development purposes, allowing you to quickly test and adjust your service. Final deployment can be made on common webservers such as Apache, Nginx or lighthttpd through the WSGI mechanism. The service configuration file itself is by definition a Python file calling specific configuration directives in the CLAM API. The system wrapper script may be written in any language, but Python users benefit as they can use the CLAM API which makes the job easier. Projects and input files are stored in a simple directory structure on disk, allowing your tool easy access. No database server is required.

The webservice offers a *RESTful* interface (Fielding, 2000), meaning that the HTTP verbs `GET`, `POST`, `PUT` and `DELETE` are used on URLs that represent resources such as projects, input files, output files. The web application is implemented as a client-side layer on the webservice. It is presented through XSL transformation (Clark, 1999) of the webservice XML output.

User authentication is implemented in the form of HTTP Digest Authentication, which ensures that the password is sent in encrypted form over the network even with servers where HTTPS is not used. HTTPS support is not present in CLAM itself but can be configured in the encompassing webserver. The underlying user database can be specified either directly in the service configuration file or in a table in a Mysql database, but it is fairly easy to replace this and communicate with another external database of your choice instead. There is also support for propagating credentials from another authentication source such as Shibboleth[3], allowing for integrating with single-sign-on scenarios. Implementation of OAuth2[4] will follow in a later version.

CLAM is open-source software licensed under the GNU Public License v3. Both the software as well as the documentation can be obtained through the CLAM website at github: `http://proycon.github.io/clam`.

[2]caveat: this does not work for local parameters, i.e. parameters pertaining to files
[3]`http://shibboleth.net`
[4]`http://oauth.net/2/`

5 Related Work

As far as we know, the only tool comparable to CLAM is Weblicht (Hinrichs et al., 2010). Both tools are specifically designed for an NLP context. CLAM, however, is of a more generic and flexible nature and may also find easier adoption in other fields.

When it comes to data formats, Weblicht commits to a specific file format for corpus data. CLAM leaves file formats completely up to the service providers, although it does come, as a bonus, with a viewer for users of FoLiA (van Gompel and Reynaert, 2013).

Weblicht is Java-based whereas CLAM is Python-based, which tends to be less verbose and more easily accessible. System wrapper scripts can be written in any language, and service configuration files simply consist of directives that require virtually no Python knowledge.

All in all CLAM offers a more lightweight solution than Weblicht, allowing webservices to be set up more easily and quicker. Nevertheless, CLAM offers more power and flexibility in doing what it does: wrapping around command-line tools, its webservice specification is more elaborate than that of Weblicht. On the other hand, CLAM deliberately does not go as far as Weblicht and does not offer a complete chaining environment, which is what Weblicht is. In this we follow the aforementioned UNIX philosophy of doing one thing well and one thing only. Service chaining certainly remains possible and CLAM provides all the information to facilitate it, but it is left to other tools designed for the task. CLAM has been successfully used with Taverna (Hull et al., 2006) in the scope of the CLARIN-NL project "TST Tools for Dutch as Webservices in a Workflow" (Kemps-Snijders et al., 2012).

Acknowledgements

CLAM support and development is generously funded by CLARIN-NL (Odijk, 2010), and is being used by various projects in the Dutch & Flemish NLP communities, whose feedback and support have contributed to its success.

References

J. Clark. 1999. XSL transformations (XSLT) version 1.0. Technical report, 11.

R. T. Fielding. 2000. *Architectural Styles and the Design of Network-based Software Architectures. Doctoral dissertation*. University of California, Irvine.

M. Hinrichs, T. Zastrow, and E. W. Hinrichs. 2010. Weblicht: Web-based LRT services in a Distributed eScience Infrastructure. In Nicoletta Calzolari, Khalid Choukri, Bente Maegaard, Joseph Mariani, Jan Odijk, Stelios Piperidis, Mike Rosner, and Daniel Tapias, editors, *LREC*. European Language Resources Association.

D. Hull, K. Wolstencroft, R. Stevens, C. Goble, M. R. Pocock, P. Li, and T. Oinn. 2006. Taverna: a tool for building and running workflows of services. *Nucleic Acids Res*, 34(Web Server issue):729–732, July.

M. Kemps-Snijders, M. Brouwer, J.P. Kunst, and T. Visser. 2012. Dynamic web service deployment in a cloud environment. In Nicoletta Calzolari, Khalid Choukri, Thierry Declerck, Mehmet Ugur Dogan, Bente Maegaard, Joseph Mariani, Jan Odijk, and Stelios Piperidis, editors, *LREC*, pages 2941–2944. European Language Resources Association (ELRA).

M. D. McIlroy, E. N. Pinson, and B. A. Tague. 1978. Unix time-sharing system forward. *The Bell System Technical Journal*, 57(6, part 2):p.1902.

J. Odijk. 2010. The CLARIN-NL project. In *Proceedings of the Seventh International Conference on Language Resources and Evaluation, LREC-2010*, pages 48–53, Valletta, Malta.

E. S. Raymond. 2004. The Art of Unix Programming.

M. van Gompel and M. Reynaert. 2013. FoLiA: A practical XML Format for Linguistic Annotation - a descriptive and comparative study. *Computational Linguistics in the Netherlands Journal*, 3.

G. van Rossum. 2007. Python programming language. In *USENIX Annual Technical Conference*. USENIX.