PBML

The Prague Bulletin of Mathematical Linguistics
NUMBER 93 JANUARY 2010 97–106

ScaleMT: a Free/Open-Source Framework for Building Scalable Machine Translation Web Services

Víctor M. Sánchez-Cartagena, Juan Antonio Pérez-Ortiz

Departament de Llenguatges i Sistemes Informàtics, Universitat d'Alacant, Spain

Abstract

Machine translation web services usage is growing amazingly mainly because of the translation quality and reliability of the service provided by the Google Ajax Language API. To allow the open-source machine translation projects to compete with Google's one and gain visibility on the internet, we have developed ScaleMT: a free/open-source framework that exposes existing machine translation engines as public web services. This framework is highly scalable as it can run coordinately on many servers, efficiently managing the resources needed by the engines, and its API is compatible with Google's one. ScaleMT is based on previous efforts to build a web service for the Apertium machine translation toolbox, but we have also tested it with Matxin, another free/open-source transfer-based machine translation engine. Additionally, we have compared ScaleMT to an alternative web service implementation for Apertium, obtaining satisfactory results.

1. Introduction

Machine translation (MT) web services are becoming very useful in the web 2.0 era. One of the key features of web 2.0 applications (O'Reilly, 2005) is that they profit from the contributions of users collaborating in the creation of content. However, linguistic barriers make the massive collaboration and understanding of the contents very difficult. Web applications which integrate machine translation services usually attract users speaking different languages and therefore receive more contributions, as can be seen by the increasing number of web applications relying on the Google Ajax Language API[1]. As a result, the MT engine that provides the service receives a

[1] http://code.google.com/apis/ajaxlanguage/

Corresponding author: vmsanchez@dlsi.ua.es
Cite as: Víctor M. Sánchez-Cartagena, Juan Antonio Pérez-Ortiz. ScaleMT: a Free/Open-Source Framework for Building Scalable Machine Translation Web Services. The Prague Bulletin of Mathematical Linguistics No. 93, 2010, pp. 97–106. ISBN 978-80-904175-4-0. doi: 10.2478/v10108-010-0011-9.

high amount of useful feedback and its visibility is increased. In the case of an open-source project, being popular would make people feel more interested in it and even join the community of developers.

Unfortunately, open-source MT projects, such as Apertium (Forcada et al., 2009) or Matxin (Alegria et al., 2007), are usually not designed to act as web services: they are not scalable, since they cannot run coordinately on many computers, and spend many CPU cycles loading resources; and they do not have an easy-to-use and internet-friendly API (Application Programming Interface).

With the aim of overcoming these problems, we introduce ScaleMT, a free/open-source framework that exposes existing machine translation engines as public web services, with an API compatible with Google Ajax Language API. Additionally, it allows the MT engines to be deployed on multiple servers in order to achieve high scalability. ScaleMT is based on a previously developed scalable web service architecture for Apertium (Sánchez-Cartagena and Pérez-Ortiz, 2009), that have been generalised to work with different MT engines. The main advantage of ScaleMT is that the architecture of the engines to be exposed does not need to be changed, although it must meet some requirements explained in section 2.

As the API of the web service has already been described previously (Sánchez-Cartagena and Pérez-Ortiz, 2009), this paper will focus on the ScaleMT architecture and how it can run with different MT engines. Firstly, section 2 will explain which kind of translation engines this framework is focused on. Later, in section 3 we will describe the ScaleMT architecture. After that, section 4 explains the steps needed to add a new translation engine to the service. Section 5 contains the description and results of two different experiments: a comparison between the different MT engines that can run with ScaleMT, and a comparison between ScaleMT and another efficient MT web service that can be found in the literature (Minervini, 2009). Finally, the paper ends with some conclusions that can be drawn from the development of the system and from the experiments, and with a list of future tasks.

2. Translation Engines that Can Profit from this Architecture

ScaleMT has been designed to work with MT engines which have these two features:

1. The translation engine is a process that reads the input text from its standard input and writes the translation to its standard output. It starts to translate before reading the full text from the input and dies when the standard input is closed.
2. Every time the process is launched, it needs to perform some start-up operations that require many CPU cycles before it can translate.

The second feature is very common on transfer-based MT systems such as Apertium or Matxin: rules and dictionaries need to be loaded before the engines can use them to translate the input text; therefore, translating many small texts is extremely inefficient. Our framework reuses the processes of the translation engine, translating

many source texts with the same ones, thus avoiding loading rules and dictionaries time after time. The reuse is possible thanks to the first feature, as will be explained with more detail in the next section.

3. System Architecture

In this section we present the architecture of ScaleMT. Our proposal makes the translation engines more efficient by turning them into *daemons* (that is, processes running in the *background* rather than under the interaction of a user). Besides that, it is able to run on multiple servers thanks to an algorithm which decides which daemons should run on each server and a load balancing method that decides which server should process each request. ScaleMT consists of two main Java applications:

ScaleMTSlave runs on a machine with the translation engine installed and manages a set of running translation engine instances (daemons); it performs the requested translations by sending them to the right daemon.

ScaleMTRouter (*request router*) runs on a web server; it processes the translation requests and sends them to the right `ScaleMTSlave` instance.

The different components of the architecture are explained in detail next.

3.1. Daemonising Engines

As we stated before (see section 2), our framework is designed to work with translation engines that spend many CPU cycles performing start-up operations when they are launched. Since the start-up cost is so high, a daemon to reuse translation engine processes and minimise the amount of start-ups must be found. A daemon is a process that is launched once and can perform many translations. Taking advantage from the first feature described in section 2, we have built a daemon by queueing translation requests, sequentially writing source texts from the requests to the standard input of the translation engine process, and not closing it when there are not requests in the queue. This approach has to deal with two issues that are solved in different ways depending on the translation engine (see section 4):

- Separating the different translations: the process behaves as if it was translating a long text but we need the different translations to be easily isolated from the output of the process.
- Making the daemon translate immediately: input/output implementations in many operating systems and programming languages use buffers for efficiency reasons. It can happen that a translated text is stored in a buffer and not returned until the buffer is completely filled.

A daemon can only translate with the language pair for which it has loaded the data.

3.2. Load Balancing

The *request router* sends each translation request to a `ScaleMTSlave` instance running a daemon for the involved language pair. Choosing that server and fairly distributing the work between all the available ones is called *load balancing*.

`ScaleMTRouter` manages one queue for each language pair. When a request arrives, it is put on the queue corresponding to its language pair. For each queue, there is a *dispatcher thread* that consumes requests from it independently from the other queues, and sends them to the most suitable server. Each request in the queue has an associated CPU cost. The dispatcher thread keeps track of the sum of the CPU costs of the requests that have been sent to each server, but have not been completed yet, namely *waiting rate*. Dispatching works as follows:

1. The dispatcher thread checks whether the lowest waiting rate in the set of servers with a daemon for the associated pair is lower than a particular threshold. If this condition is not held, it waits a short time and executes this step again.
2. It takes the first request from the queue, sends it to the server with the lowest waiting rate, and returns to step 1. Server's waiting rate is updated accordingly.

This way, although queues are independent, work is fairly distributed. If a server is processing many requests for a language pair A, requests for language pair B will take a long time to be processed because both need to share CPU. If there is another server that is not processing A requests, it will translate B requests faster and, consequently, receive more B requests as it will be more often the server with less waiting rate.

3.3. Application Placement

Servers usually do not have enough memory to run a daemon for every supported language pair. Consequently, we should run the daemons which receive more translation requests, and adapt the number of daemons and the power of the machines where they run to the amount of work they have to perform. Additionally, load changes throughout time, so the running daemons should change too. To deal with these problems we have developed a *placement algorithm* based on the work by Tang et al. (2007) that is executed periodically and decides which daemons should run on each server.

The algorithm is widely based on the concept of load. The load is an estimation of the CPU power needed to perform translations measured in *translated characters per second*. Each server has a *load capacity*, estimated by translating a set of texts on it, and a *memory capacity*. Each language pair has a *memory requirement*, estimated once by simply running its daemon and measuring the memory it needs, and a *load requirement*, estimated periodically from the requests received by the service. After new load requirements are estimated, the algorithm is executed to decide how many daemons of each language pair should run on each server, following these guidelines:

- Satisfied load must be maximised, since the load capacity of a server is distributed between the language pairs of the daemons running on it.

- The number of daemons to be started or stopped must be minimised.
- The summation of the memory requirements of the daemons running on a server must not be higher than its memory capacity.

3.4. Scaling

The whole system is able to scale by adding new servers running `ScaleMTSlave`. These servers can be added manually, or we can let a *dynamic server manager* decide when to add or remove them. The servers added by the *dynamic server manager* can be machines from a local network (with SSH access enabled) or Amazon EC2 instances.[2] This component decides when the system needs more servers based on the *placement algorithm* output: if the amount of load satisfied by the placement proposed by the algorithm is lower than the total load demand (because the servers do not have enough CPU capacity or enough memory), then new servers are added.

4. Adding a New Translation Engine

With the aim of achieving engine-independence, ScaleMT uses an XML configuration file for defining relevant information about the translation engines without changing a single line of Java code nor recompiling the project.

Firstly, we must specify the language pairs and formats supported by the engine. Then, the commands which run the translation engine need to be defined. Most translation engines are made of a deformatter module, that removes the format information from the input text; a translation core, that translates the text; and a reformatter, that restores the format information. Therefore, a pipeline of programs can be defined. One of the components of the pipeline (the one with the highest start-up time, usually the core) should be chosen to be kept in execution, and the other components allowed to be executed once for every translation, even with different parameters depending on the translation type. The configuration format allows a high flexibility on the communication between the pipeline components.

The most important part of the configuration file contains the solution to the problems defined in section 3.1:

- The strategy to isolate the different translations in the engine flow consists of adding extra sentences containing an unique identifier before and after the text to be translated. Then, their translation and the unique identifier of the text are searched in the output.
- Ensuring that the translation engine returns the translation immediately and it does not remain stored in buffers involves sending some extra text (*padding sentences*) to the daemon after each translation to completely fill the buffers. A *padding sentence* should be included in the configuration file and the system em-

[2] `http://aws.amazon.com/ec2/`

pirically estimates how many units of that sentence has to send after each translation source to completely fill the buffers and get the translation. To avoid overloading the system, *padding* is not sent unless the daemon local queue is empty, which means that it will not receive more requests soon.

Sample configuration files with detailed comments about each section can be found with the source code of ScaleMT[3]. As a proof of concept, we have evaluated our framework with Apertium and Matxin.

4.1. Apertium

In the Apertium pipeline (Forcada et al., 2009), text is first deformatted, and the format information is put into *superblanks*, special blocks between square brackets that are not translated. The output of the deformatter is then translated by the core, and finally the format is restored by the reformatter. We decided to keep in execution only the core, a decision which has two advantages. First, we can use *superblanks* to separate the different translations, and forget to worry about how the separation sentences will be translated. And second, since the deformatter and reformatter are executed for each translation, we can use the same daemon to translate different formats.

With Apertium there is no need of sending *padding* after each source text to be translated as all the modules in the Apertium pipeline have an mechanism called *null flush* that makes them flush their buffers when they receive a null character in their input. Consequently, the configuration XML includes and option to enable the sending a null character after each translation request.

4.2. Matxin

Matxin pipeline architecture (Alegria et al., 2007) is different from the Apertium one. Matxin deformatter has two outputs: the text without format and the format information. The unformatted text is translated by the core and later joint with the format information by the reformatter. As a consequence of that, we have to keep in execution only the core because a pipeline cannot be built with two flows. We have used as separator sentences unknown words since they do not fire any transfer rule that could waste CPU. As Matxin does not support null flush, we use the *padding* mechanism to ensure that the translation engine returns the translations immediately. We have chosen a long sequence of random characters as *padding* because a single unknown word consumes residual CPU time, and a long word fills the buffers faster.

Additionally, two more problems arose during the adaptation process. First, one of the components of the Matxin pipeline is `iconv`, a process that changes text encoding. It does not write the text with the new encoding until it reads the end-of-file (EOF) character. So, we wrote a modified version of `iconv` that processes input file line

[3] `http://apertium.svn.sourceforge.net/svnroot/apertium/trunk/scaleMT/ScaleMTSlave/sampleconfs`

per line. Second, the Matxin process crashes with some inputs making the daemon running on the top of that process to die. To deal with this problem we have created a mechanism which detects if a process has died and launches it again.

5. Experiments and Results

Note that in order to ensure reproducibility, scripts that automatically perform the following experiments are available[4].

5.1. Efficiency Gain for Apertium and Matxin

First, we will estimate the performance improvement in Apertium and Matxin when translating a text with ScaleMT. The main idea under Amdahl's law (Amdahl, 1967) is that the performance improvement obtained by using some *faster mode* of execution in a program is limited by the fraction of the time the *faster mode* can be used. Analysing the time needed to perform a translation with Apertium or Matxin, we can split it in the time needed to load resources (*start-up time*) and the time needed to perform the actual translation (*translation time*). ScaleMT reduces the start-up time because the time needed to start the whole MT engine process is replaced by a little overhead caused by launching the deformatter and the reformatter, and by translating the separation sentences; but translation time is unchanged. Therefore, the fraction of the time the faster mode can be used corresponds to the start-up time.

The plot on the left of figure 1 shows the time needed to translate input texts of different lengths by four different systems: Apertium (Spanish–English), Matxin (Spanish–Basque), ScaleMT running Apertium (Spanish–English) and ScaleMT running Matxin (Spanish–Basque). The time needed to translate a 0-length text is the start-up time. The plot on the right shows the performance gain in Apertium and Matxin. The experiments have been run with an instance of `ScaleMTRouter` and and a single instance of `ScaleMTSlave` running on the same machine.[5] Following Amdahl's law, Apertium's gain is greater than Matxin's because, for the same source text length, the percentage of time spent in the start-up operations is greater too.

5.2. Comparing with Other Scalable Apertium Web Service Implementations

Different approaches to build scalable MT web services have been proposed. For instance, Minervini (2009) designed `Apertium-service`, an efficient Apertium-based web service, by completely changing Apertium architecture. The key idea under his approach is replacing the original pipeline-based architecture with a multithreaded

[4] `http://apertium.svn.sourceforge.net/svnroot/apertium/trunk/scaleMT/ScaleMTRouter/experiments`

[5] An AMD Turion TL-56 with 2 GB of RAM memory

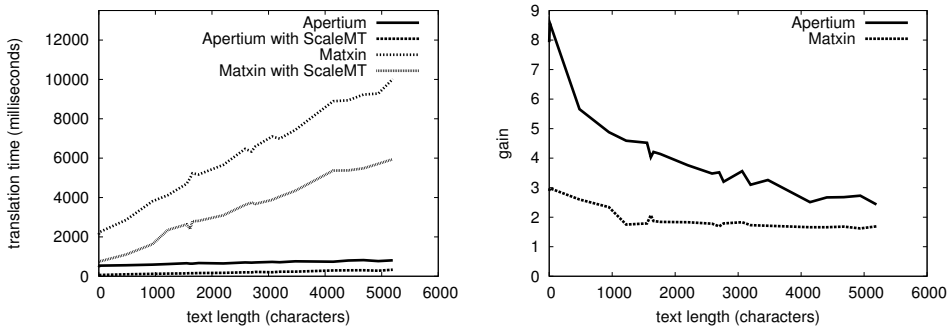

Figure 1. Translation time and efficiency gain with the use of ScaleMT.

resource pool in which linguistic data is kept loaded. We have performed some experiments comparing ScaleMT with `Apertium-service`. They are similar to the experiments originally performed by Minervini (2009) comparing his `Apertium-service` with the predecessor of ScaleMT, but they cover a wider data range.

The plot on the left of figure 2 shows the time needed to translate (English–Spanish) fragments of different lengths of the GPL license text[6] by a single client. The plot on the right shows the average time needed to translate (again from English to Spanish) the preamble of the GPL license text (2531 characters) when requested by different amounts of concurrent clients. Both experiments have been performed on an Amazon EC2 small instance. In the case of ScaleMT, there is only one slave running on the same machine as the router.

5.3. Other Interesting Results

The previous experiments have targeted only a single slave instance. However, it is also important to check if the architecture is able to scale to a high number of slaves. The experiments performed by Sánchez-Cartagena and Pérez-Ortiz (2009) about this topic show that, running on a not very powerful machine (an Amazon EC2 small instance), the router can process up to 19 000 requests per minute. With input texts that do not need a high CPU capacity to be processed (56 characters, Apertium Spanish–Catalan), 20 slaves are needed to perform the work needed by such a high request rate. Since the changes made to generalise the architecture have been mainly focused on the structure of the slaves, that maximum request rate is still valid. However, since translating a text with Matxin needs more CPU capacity, the number of servers needed to manage this rate could be even higher.

[6] `http://www.gnu.org/licenses/gpl.html`

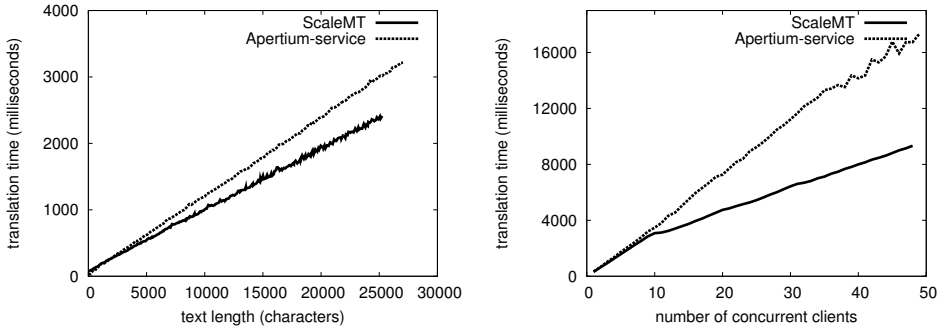

Figure 2. Comparison between ScaleMT and `Apertium-service`.

6. Conclusions

We have developed ScaleMT, a free/open-source framework to automatically create web services from existing MT engines. According to our experiments, ScaleMT is more suitable to work with Apertium than with Matxin. When translating texts of around 500 characters, Apertium performance gain is 5.6, and Matxin's is only 2.6, mainly because the start-up time of Apertium is bigger than the start-up time of Matxin, compared with the time needed to translate a typical text. Additionally, Matxin, as of revision 248, is unstable and unpredictably crashes with some inputs. Consequently, the daemon is started more times than necessary, causing performance loss.

When comparing with `Apertium-service`, the time needed to translate individual texts is similar in both architectures, although ScaleMT performs better with longer texts. This is not a clear advantage because translations requested from web pages to a JSON API are not usually very long. However, when testing the systems in a more realistic scenario (many concurrent clients), ScaleMT outperforms `Apertium-service`. Furthermore, ScaleMT is engine-independent and, consequently, adaptable to future changes in Apertium, while `Apertium-service` is an ad-hoc solution.

ScaleMT is currently under evaluation to be the *official* Apertium web service. Source code for ScaleMT can be downloaded from `http://apertium.svn.sourceforge.net/svnroot/apertium/trunk/scaleMT`.

7. Future Work

It would be interesting to consider ScaleMT to build a web service over the Moses decoder (Koehn et al., 2007). Firstly, it should be checked whether it meets the requirements explained in section 2, and then find the appropriate sentences to separate the

different requests in the pipeline. We originally did not address Moses because there is already a web service implementation for it. However, we could draw interesting conclusions by comparing ScaleMT with the Moses web service.

It is worth improving scalability by increasing the maximum request rate supported by the router. This is not an easy task because it probably would involve having many router instances and synchronising them.

8. Acknowledgements

This work has been partially funded by Google through the Google Summer of Code program and by Spanish Ministerio de Ciencia e Innovación through project TIN2009-14009-C02-01.

Bibliography

Alegria, I., A.D. de Ilarraza, G. Labaka, M. Lersundi, A. Mayor, and K. Sarasola. Transfer-based MT from Spanish into Basque: reusability, standardization and open source. *Lecture Notes in Computer Science*, 4394:374–384, 2007.

Amdahl, G.M. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485. ACM New York, NY, USA, 1967.

Forcada, M.L., F.M. Tyers, and G. Ramírez-Sánchez. The Apertium machine translation platform: five years on. In *Proceedings of the First International Workshop on Free/Open-Source Rule-Based Machine Translation*, pages 3–10, 2009.

Koehn, P., H. Hoang, A. Birch, C. Callison-Burch, M. Federico, N. Bertoldi, B. Cowan, W. Shen, C. Moran, R. Zens, et al. Moses: Open source toolkit for statistical machine translation. In *Annual Meeting of the Association for Computational Linguistics: Demonstration Session*, 2007.

Minervini, P. Apertium goes SOA: an efficient and scalable service based on the Apertium rule-based machine translation platform. In *Proceedings of the First International Workshop on Free/Open-Source Rule-Based Machine Translation*, pages 59–66, 2009.

O'Reilly, T. What is web 2.0. In *Design Patterns and Business Models for the Next Generation of Software*, `http://www.oreillynet.com/pub/a/oreilly/tim/news/2005/09/30/what-is-web-20.html`, 2005.

Sánchez-Cartagena, V.M. and J.A. Pérez-Ortiz. An open-source highly scalable web service architecture for the Apertium machine translation engine. In *Proceedings of the First International Workshop on Free/Open-Source Rule-Based Machine Translation*, pages 51–58, 2009.

Tang, C., M. Steinder, M. Spreitzer, and G. Pacifici. A scalable application placement controller for enterprise data centers. In *Proceedings of the 16th international conference on World Wide Web*, pages 331–340. ACM, 2007.