An open-source highly scalable web service architecture for the Apertium machine translation engine

Víctor M. Sánchez-Cartagena
Department de Llenguatges i Sistemes
Informàtics, Universitat d'Alacant,
Spain, vmsanchez@dlsi.ua.es

Juan Antonio Pérez-Ortiz
Department de Llenguatges i Sistemes
Informàtics, Universitat d'Alacant,
Spain, japerez@dlsi.ua.es

Abstract

Some machine translation services like Google Ajax Language API have become very popular as they make the collaboratively created contents of the web 2.0 available to speakers of many languages. One of the keys of its success is its clear and easy-to-use application programming interface (API) and a scalable and reliable service. This paper describes a highly scalable implementation of an Apertium-based translation web service, that aims to make contents available to speakers of lesser resourced languages. The API of this service is compatible with Google's one, and the scalability of the system is achieved by a new architecture that allows adding or removing new servers at any time; for that, an application placement algorithm which decides which language pairs should be translated on which servers is designed. Our experiments show how the resulting architecture improves the translation rate in comparison to existing Apertium-based servers.

1 Introduction

Although Apertium (Armentano-Oller et al., 2005) is a popular machine translation toolbox, Apertium-based machine translators are not massively used over the Internet. The reason for this low usage is that the only way to access the translators is querying a server through some of the existing HTTP web forms, like the one located at apertium.org. This method has two disadvantages: it makes difficult for external developers to easily grab the results of the translations, and it is not scalable as it needs to launch an Apertium instance and load all the linguistic data for each request.

Our proposal consists of creating the right framework for the Apertium engine to support a massive access. We have designed a clean and intuitive application programming interface (API), and created a scalable and reliable open-source architecture implementing its services.

We will start off by describing the relationship between web 2.0 applications and machine translation (section 2). In section 3 we will list different approaches to solve the Apertium limitations related to massive access. Then, in section 4 we will explain how our API works. Later, in section 5 our scalable architecture will be described. Section 6 contains the experiments we made to test our system. Finally, sections 7 and 8 show a list of future works and the conclusions that can be drawn from the development of our system.

2 Machine Translation in Web 2.0

Machine translation services are becoming very useful in the web 2.0 era. One of the key features of web 2.0 applications is that they profit from the contributions of users collaborating in content creation. But linguistic barriers make the massive collaboration and understanding of the contents very difficult. Web applications which inte-

J.A. Pérez-Ortiz, F. Sánchez-Martínez, F.M. Tyers (eds.)
Proceedings of the First International Workshop on Free/Open-Source Rule-Based Machine Translation, p. 51–58
Alacant, Spain, November 2009

grate 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 which rely on the Google Ajax Language API[1].

However, in spite of the evident benefits, a very reduced number of web applications using Apertium exist. There is, for example, a Wordpress plug-in[2] that translates blog posts into any of the languages supported by Apertium; the plug-in is successfully used by some bloggers, but it cannot be massively installed since it sends POST requests to the non-scalable web form at apertium.org. There are three main limitations behind this lack of web applications integrating Apertium:

- There is not a clear, well-documented API, that allows web applications to access Apertium from JavaScript code loaded from different domains. The JSONP technique (see Section 4) is not supported and, therefore, there is no simple mechanism to get around the JavaScript *same origin policy*[3].

- Every Apertium process has to load the linguistic data of the corresponding language pair every time it is invoked. Boot time is considerably long (see the experiments in section 6.1). Until now, there was no way of keeping data loaded in memory and use it to translate different input texts, that is, working in *daemon* mode.

- Finally, and as a consequence of the previous item, there is not a server or cluster of servers capable of processing massive translation requests.

3 Alternative approaches

We know of at least one other approach, apart from ours, suggested in order to solve the aforementioned Apertium limitations. The approach by Minervini (2009) consists of rewriting Apertium to keep linguistic data in memory and add multithreading support to it. This way, the Apertium daemon only loads data once, saving a lot of CPU cycles, and then processes concurrent requests thanks to multithreading.

Minervini's approach has many advantages. It introduces less overhead, because threads are more efficient than processes (Wagner and Towsley, 1995). It also needs less memory than our approach when running on multiprocessor machines since, under certain circumstances, we launch more than one Apertium process for the same language pair in order to use all the CPU capacity of the machine.

We decided not to follow the multithreading approach for one main reason: it heavily depends on Apertium's internal structure. Although updates in the linguistic data do not affect the multithreaded version, updates in the Apertium core libraries interface will make it difficult to maintain. Moreover, adding to the pipeline extra modules not included in the Apertium libraries can be difficult. Our approach (see section 5), considers Apertium as a *black box*, so modes files can still be easily edited to add new modules whose design does not follow any special guideline. New modules must only meet a single condition: flush their output when they receive a null character from the standard input.

A different alternative would be delegating scaling to existing open-source middleware systems. This alternative is unfeasable since it requires Apertium to be modified to add communication with the middleware and we want our system not to be affected by changes in Apertium.

4 A New API for Apertium

We have designed a simple JSONP REST API compatible with Google Ajax Language API that simplifies the task of creating Apertium-based web applications. Developers used to work with Google Ajax Language API will easily adapt to it and applications using our API will be more scalable, since the task of getting translations from the Apertium server will be possible for any JavaScript client, like regular web browsers.

Our API has two available operations: listing supported language pairs, and translating. The listing operation is different from the Google's API one. Google provides a JavaScript library where the supported languages are hardcoded, but since we have not written a JavaScript library yet,

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

[2] http://xavi.infobenissa.com/utilitats/wp-apertium/

[3] http://en.wikipedia.org/wiki/Same_origin_policy

a web service operation is provided to list the available language pairs. The listing operation returns an array of JSON objects containing the supported language pairs. A HTTP GET or POST request to the URL

```
http://ApertiumServerInstallationHost/
    ApertiumServerRouter/resources/listPairs
```

will return the list of supported language pairs as, for instance:

```
{”responseData”:[{”sourceLanguage”:”ca”,”
    targetLanguage”:”oc”},{”sourceLanguage”:”en”,”
    targetLanguage”:”es”}],”responseDetails”:null,”
    responseStatus”:200}
```

The translating operation works exactly like the Google’s API one. For example, to translate the text “hello world” from English to Spanish we have to make a HTTP GET or POST request to the following URL:

```
http://ApertiumServerInstallationHost/
    ApertiumServerRouter/resources/translate?q=hello
    %20world&langpair=en%7Ces
```

The server will return a JSON object with the response status and the translated text, if any:

```
{”responseData”:{”translatedText”:”hola Mundo”},”
    responseDetails”:null,”responseStatus”:200}
```

The *same origin policy* will not allow using XMLHttpRequest to access our web service from third-party client-side JavaScript applications, but there are some exceptions: one of them are `<script>` elements, which can point to any domain. So, one way of calling the Apertium web service from any JavaScript application is by dynamically adding a `<script>` element to the DOM pointing to the URL of the web service. Both operations support a parameter called *callback*. When this parameter is provided, the operation returns a call to a function whose name is the value of the callback parameter with one argument: the JSON object with the status code and the result of the operation. This way, the callback function is called as soon as the result is available. This method is called JSONP and is currently used by many web 2.0 applications. The following function is an example of how to use our API from JavaScript:

```
function translateJSONP(url,text,pair,callback)
{
  url+=”?”;
  url += ”callback=”+encodeURIComponent(callback)+
      ”&q=”+encodeURIComponent(text)+”&langpair
      =”+encodeURIComponent(pair);
  var script = document.createElement(”script”);
  script.setAttribute(”src”,url);
  script.setAttribute(”type”,”text/javascript”);
  document.getElementsByTagName(”body”)[0].
      appendChild(script);
}
```

5 A New Web Service Architecture for Apertium

In this section we introduce the architecture that allows our system to respond to massive translation requests. It consists of two open-source applications:

- ApertiumSlave runs on a machine with Apertium installed and manages a set of Apertium daemons; it performs the requested translations by sending them to the right daemon.
- ApertiumRouter (*request router*) runs on a web server; it processes the translation requests and sends them to the right ApertiumSlave instance.

Since servers usually do not have enough memory to run an Apertium daemon for every supported language pair, there is a *placement algorithm* which is executed periodically on ApertiumRouter to decide which daemons to run on each server so that all translation requirements are met. Additionally, it tries to minimize the number of daemon creations and terminations, because each time a daemon is started it loads its language pair linguistic data. The *placement algorithm* is executed by a component called *placement controller*.

ApertiumRouter has a queue for each supported language pair. When a translation request arrives, it is inserted in the corresponding queue. When a request is taken from the queue, it is sent to a server running a daemon with the corresponding language pair. The server is chosen trying to balance the load between the different servers. These queues are part of a component called *load balancer*.

The whole system is able to scale by adding new servers running ApertiumSlave. 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[4]. This component decides when the system needs more servers according to demand based on the *placement algorithm* output. If the algorithm detects that the total CPU demand is higher than the CPU capacity for all the servers, or that there is not enough free memory, then new servers are added. The *dynamic server manager* is part of ApertiumRouter.

All these components have been designed to be as much independent as possible, to allow easier future modifications and their distribution over different machines.

5.1 Daemonizing Apertium

As stated before, one of the reasons behind Apertium's limited scalability is the high CPU cost of loading linguistic data, and the fact that data is loaded every time Apertium is launched to perform a translation. Experiments in section 6.1 show that, in this case, translating a text, split in many fragments, is much slower than translating the whole text, because linguistic data has to be repeatedly loaded in memory for every fragment.

The idea behind our daemon implementation is very simple. Apertium can be considered as a standard UNIX process that reads from standard input and writes to standard output, and as long as its standard input is not closed, the process will never end. Therefore, we have built an Apertium daemon by queueing translation requests, sequentially writing source texts from the requests to the process standard input, and not closing it when there are not requests in the queue. The different translations are separated in Apertium's standard output flow by *superblanks* (text fragments that are not translated by Apertium) that encapsulate a unique number identifying each translation. For avoiding that finished translations remain temporarily stored in the buffers of the pipeline modules, we send a null character after the text of each translation requests. The different modules flush their output when this null character is read. Some modules of the Apertium pipeline were slightly modified to meet the previous requirements.

5.2 Measuring and Predicting Load

The *placement algorithm* takes as input the total CPU and memory capacity of each server (Ω_s and Γ_s, respectively), the maximum load that can be assigned to a single daemon on each server (Ω'_s), the amount of CPU cycles (load) needed by each language pair (ω_p), the memory needed by a daemon of each language pair (γ_p), a matrix representing which daemons can run on each server (R), and another matrix representing which daemons are currently running on each server (I_{t-1}).

Memory is measured in megabytes. In the Linux operating system Γ_s can be obtained by looking at the information in /proc/meminfo, and γ_p by starting the daemon and measuring the memory consumption of all the pipeline processes with the command top-b.

CPU capacities and demands are expressed as number of translated characters per second. We calculate Ω'_s by translating a long text from Spanish to Catalan with a single daemon on server s and dividing its number of characters by the translation total execution time. Ω_s is calculated by following a similar process but with several daemons and texts. Parameter ω_p for a pair p is predicted by adding up the computational cost of all the requests of that pair in the last period (note that the placement algorithm is executed periodically), and dividing the result by the period duration. The computational cost of a request depends on its number of characters and on the language pair.

5.3 Load Balancing and Scheduling

One of the most important tasks of the *request router*, apart from executing the placement algorithm, is sending each translation request to a server that runs a daemon for the involved language pair and balancing the load between all the servers.

The request router 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

[4] http://aws.amazon.com/ec2/

have been sent to each server, but have not been completed yet. This parameter is called *server load*. Dispatching works as follows:

1. The dispatcher thread checks whether the lowest load in the set of servers running a daemon with the dispatcher's 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 load, and returns to step 1. Obviously, server's load measure is updated accordingly.

This way, although queues are independent, load is balanced globally. 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 because it will be more often the server with less load.

5.4 Application Placement

As we stated before, the most important element of this highly scalable architecture is the *placement algorithm*. It is based on the solution proposed by Tang et al. (2007), but with some minor modifications that allow it to run more than one instance of the same daemon (application) on a server.

The original algorithm takes as input Ω_s, Γ_s, ω_p, γ_p, R and I_{t-1}, as described in 5.2. It tries to maximize the amount of satisfied load, and outputs a matrix indicating which daemons should run on each server (I_t), and the load satisfied by that placement (L). It repeatedly and incrementally optimizes the placement solution in multiple rounds. In each round, it first computes the maximum total application demand that can be satisfied by the current placement solution. The algorithm quits if all the application demands can be already satisfied. Otherwise, it shifts load across machines (without placement changes), and then considers stopping unproductive application instances and starting more useful ones in order to increase the total satisfied application demand. The load shifting step before the placement changing step is critical as it dramatically simplifies subsequent placement changes. Note that, in the algorithm description, *placement change*, *application start/stop*, and *load shifting* are all hypothetical. The real placement changes are executed after the placement algorithm terminates.

The load shifting step involves solving the *maximum flow* and the *minimum cost–maximum flow* problems on a graph. In our preliminary implementation, we have solved these problems with algorithms that are easy to implement, but which are not the fastest ones. We have chosen two special variations of the Ford–Fulkerson algorithm (Ford and Fulkerson, 1962). Maximum flow problem has been solved with Edmonds–Karp algorithm (Edmonds and Karp, 1972). Minimum cost–maximum flow problem has been solved with another variation optimized specifically for the structure of our graph.

The algorithm proposed by Tang et al. (2007) assumes that no more than one instance of each daemon can run on the same server. But, on servers with many CPUs, it may happen that a single daemon does not consume all the server's capacity. So, if only one language pair is assigned to this kind of servers, more than one daemon for that pair will be needed to satisfy the required load. We deal with this problem by adding an additional parameter to the server's information: the capacity of a single daemon (Ω'_s). We have modified two steps of the algorithm to include the new parameter:

- When building the flow graph, the edges between application nodes and server nodes change their capacities. According to the original algorithm, an edge from application node N_{ai} to server node N_{sj} must have infinity capacity if an instance of application i is running on server j, and zero capacity otherwise. In our modification, if the capacity of a single daemon of server j is C_{sj} and there are N_{ij} daemons of application i running on server j, the capacity of the edge from N_{ai} to N_{sj} must be $C_{sj} * N_{ij}$.

- In the placement changing inner loop, when taking an application from the *residual app tree* and starting an instance on server j, no

	Full	Split	Slave	Router
Time (s)	73	2002.39	219	321
Ratio	1	27.43	3	4.40

Table 1: Execution times (s) for different web service implementations.

more than C_{sj} can be assigned to it. If the application still has some residual demand, it is put again in the residual app tree.

6 Experiments and Results

6.1 Comparing with standard Apertium implementation

One interesting way of evaluating the usefulness and efficiency of our system is by comparing the scalable architecture to a standard Apertium setup by measuring the time needed for translating a set of input texts.

Table 1 shows the time needed to translate a text version of Miguel de Cervantes' Don Quijote[5], (383 184 words) from Spanish to Catalan with different configurations. The first column shows the time needed to translate it by launching the standard Apertium process and sending a file with the whole text to it. The second column shows the time needed to translate sequentially the same text split in 1 858 fragments by launching the standard Apertium process for each fragment. The third one shows the time needed to translate sequentially those fragments by sending them directly to an instance of ApertiumSlave with a Spanish-Catalan daemon already loaded. And the fourth one shows the time needed to translate sequentially the 1 858 fragments by sending them to ApertiumRouter with only an associated instance of ApertiumSlave running on the same machine. The second row represents the ratio between each implementation and the first one. All the tests have been made on the same machine[6].

As we can see in the table, the ratio obtained by sending the translation requests to ApertiumSlave is much lower than the one obtained launching a different Apertium instance to perform each translation. However, it is higher than 1 because of the time spent by the requests waiting in different queues. When the fragments are sent sequentially to ApertiumRouter the ratio is a bit higher than the previous one, but still much lower than the one shown in the second column, which indicates that ApertiumRouter is not introducing a big overhead and our system accomplishes its objectives.

6.2 Placement algorithm execution time

One of the facts that can limit the system's scalability is the execution time of the placement algorithm. This algorithm should be executed frequently to adapt the number and type of running daemons to the changing load requirements. An experiment was carried out in order to measure the execution time of the placement algorithm for different number of servers in a realistic environment. To do so, we executed it with all the stable language pairs available for Apertium. For each number of servers, we executed the algorithm 10 times. The first time, the CPU demand for each language pair was chosen randomly and there were not running daemons. The following times the running daemons were the result of the previous step, and the CPU demand of each language pair was calculated by multiplying the previous one by a factor $f = 1 + \mathrm{rand}(-0.5, 0.5)$. The experiment is conditioned by the parameter L_{cpu}, which is the factor between the total CPU demand on the first time the algorithm is executed and the total CPU capacity. The randomly chosen CPU demands are normalized to fit this factor.

Server CPU capacities are chosen randomly from the set $\{40\,000,\ 27\,000,\ 30\,000\}$, server memory capacities are chosen randomly from $\{1\,000,\ 500,\ 800,\ 1\,400\}$ and language pairs memory demands from $\{64,\ 124,\ 69,\ 28,\ 97\}$. The plot in figure 1 shows the average execution time with different values of L_{cpu}.

In the worst case ($L_{cpu} = 1$) the algorithm needs about 30 seconds to finish with 1 500 servers. It is important to have a small interval of time between placement executions to be able to react to sudden changes on load demands. And it is also important to keep placement algorithm execution time lower than its execution period. If not, the placement changes to satisfy the demands will be made after the demands have changed. Defining which is the maximum acceptable execution time will need a deeper research. For

[5] http://www.gutenberg.org/dirs/etext99/2donq10.txt

[6] A desktop PC with a Pentium IV 3.0 GHz

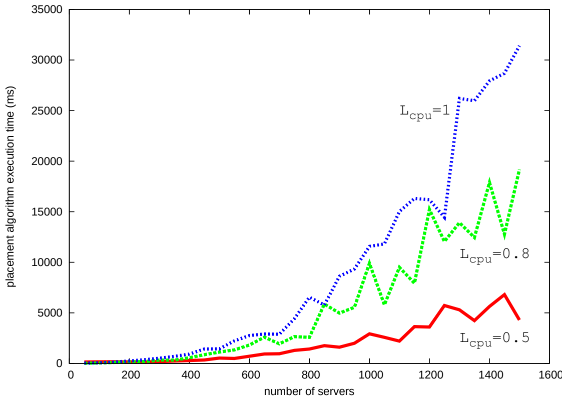

Figure 1: Average execution time (ms) with different values of L_{cpu}.

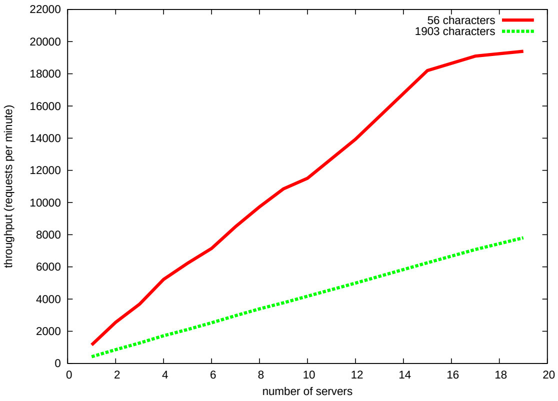

Figure 2: Throughput obtained making requests with two texts of 1 903 and 56 characters.

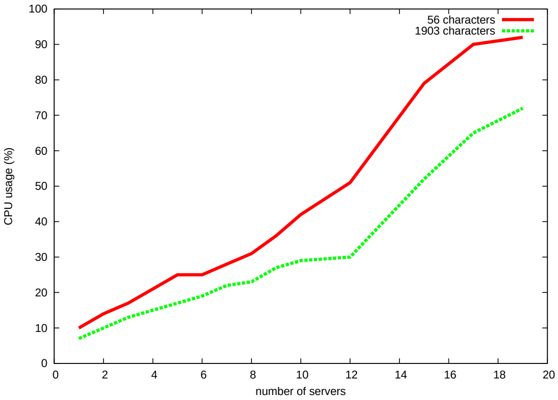

Figure 3: CPU usage of the machine running ApertiumRouter making requests with two texts of 1 903 and 56 characters.

instance, if we decide to execute the placement algorithm every minute, 20 seconds could be a reasonable execution time. Figure 1 shows that the algorithm would need 20 seconds with around 1 200 servers; therefore, in that case our system could manage around 1 200 servers.

Tang et al. (2007) state that their algorithm can run with about 7 000 servers and 17 000 applications in 30 seconds. Their implementation is much more faster than ours, specially if we take into account that our experiments have been made with only 40 applications.

6.3 Is the router a bottleneck?

One of the biggest shortcomings of our architecture is that all the translation requests are processed by a single instance of ApertiumRouter. Although processing a request does not have a high CPU cost, there is a limit on the number of requests per second it can process, and, consequently, if this limit is reached adding more servers will not improve system's capacity. With the following experiment we tried to determine the maximum amount of servers that an instance of ApertiumRouter can manage.

In our experiment, we deployed the router on Apache Tomcat[7] running on an Amazon EC2 small server instance, and added small servers running ApertiumSlave sequentially. For each number of them, we calculated the number of requests per minute the system could process (throughput) using Apache JMeter[8]. We also measured CPU usage of the machine running ApertiumRouter with htop. Due to limitations of our testing environment, we could only launch 19 servers running ApertiumSlave. The language pair of all the requests was Spanish-Catalan, and the experiments were repeated with different input text lengths. Figures 2 and 3 show the results.

The results show that the CPU usage of ApertiumRouter depends on the request rate (throughput). For the same server capacity, the request rate is higher with shorter texts, because each request needs less CPU cycles to be processed. When the number of servers is near 20, if we translate texts of 56 characters the CPU usage is over 90% and throughput does not grow lin-

[7] http://tomcat.apache.org/

[8] http://jakarta.apache.org/jmeter/

early with the number of servers. That means that the limit has almost been reached. With longer texts, the request rate is lower, so the CPU usage is lower too and the limit is not reached with 20 servers. It can be concluded that ApertiumRouter acts as the bottleneck of our system because it cannot work with the number of servers allowed by the placement algorithm.

7 Future Work

As we demonstrated in section 6.3, ApertiumRouter acts as a bottleneck. Consequently, increasing the number of requests it can process would improve scalability. A possible solution would be having more than one running instance of ApertiumRouter. We should study how to balance the load between them and how to share the placement information.

In 6.2 we stated that our implementation of the placement algorithm was much slower than the one by Tang et al. (2007). One of the causes of the efficiency gap is that we have used different implementations of the flow algorithms. Implementing the highest-label preflow-push algorithm and the enhanced capacity scaling algorithm proposed by Ahuja et al. (1993) would probably improve its efficiency.

Although we have shown in 6.3 that a single instance of ApertiumRouter can manage around 20 servers, this number can decrease when the input load for a language pair rises abruptly. Under this situation, the threads created to process incoming requests have to wait until the placement controller launches new daemons, and the thread limit of the web server could be reached.

Finally, our system could be easily adapted to work with other machine translation engines.

8 Conclusions

This paper describes a scalable system that makes Apertium-based machine translators available over a well defined web service API. The presented JSON API is compatible with Google Ajax Language API and scalability is provided by an open-source architecture and a placement algorithm based on the proposal by Tang et al. (2007). Our implementation is not as scalable as the one by Tang et al. (2007) because it only supports around 20 translation servers and its capacity is reduced when running many different language pairs. However, it can fit the needs of small and medium organizations. For instance, the translation services on apertium.org or softcatala.cat could take advantage of this new architecture. We plan to deploy it as a public service in a few months.

The source code of this project can be downloaded from http://apertium.svn.sourceforge.net/svnroot/apertium/branches/gsoc2009/vitaka/.

9 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.

References

Ahuja, R., Magnanti, T., and Orlin, J. (1993). *Network flows: theory, algorithms, and applications*. Prentice Hall.

Armentano-Oller, C. et al. (2005). An open-source shallow-transfer machine translation toolbox: consequences of its release and availability. In *OSMaTran: Open-Source Machine Translation, A Workshop at Machine Translation Summit X*, pages 23–30.

Edmonds, J. and Karp, R. M. (1972). Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM*, 19(2):248–264.

Ford, L. and Fulkerson, D. (1962). Flow in networks. *Princeton University Press*.

Minervini, P. (2009). 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*.

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

Wagner, T. and Towsley, D. (1995). Getting started with posix threads. Technical Report, Department of Computer Science, University of Massachusetts.