The Prague Bulletin of Mathematical Linguistics
NUMBER 100 OCTOBER 2013 31–40

MTMonkey: A Scalable Infrastructure for a Machine Translation Web Service

Aleš Tamchyna, Ondřej Dušek, Rudolf Rosa, Pavel Pecina

Charles University in Prague, Faculty of Mathematics and Physics, Institute of Formal and Applied Linguistics

Abstract

We present a web service which handles and distributes JSON-encoded HTTP requests for machine translation (MT) among multiple machines running an MT system, including text pre- and post-processing. It is currently used to provide MT between several languages for cross-lingual information retrieval in the EU FP7 Khresmoi project. The software consists of an application server and remote workers which handle text processing and communicate translation requests to MT systems. The communication between the application server and the workers is based on the XML-RPC protocol. We present the overall design of the software and test results which document speed and scalability of our solution. Our software is licensed under the Apache 2.0 licence and is available for download from the Lindat-Clarin repository and Github.

1. Introduction

In this paper, we describe an infrastructure for a scalable machine translation web service capable of providing MT services among multiple languages to remote clients posting JSON-encoded requests.

The infrastructure was originally developed as a component of the EU FP7 Khresmoi project, a multilingual multimodal search and access system for biomedical information and documents (Aswani et al., 2012), to provide MT services for real-time translation of user queries and retrieved document summaries. The service is used with three language pairs (Czech–English, French–English, and German–English) in both directions within the Khresmoi project, but the system is designed to be language-independent and capable of serving multiple translation directions.

Corresponding author: tamchyna@ufal.mff.cuni.cz
Cite as: Aleš Tamchyna, Ondřej Dušek, Rudolf Rosa, Pavel Pecina. MTMonkey: A Scalable Infrastructure for a Machine Translation Web Service. The Prague Bulletin of Mathematical Linguistics No. 100, 2013, pp. 31–40.
doi: 10.2478/pralin-2013-0009.

For Khresmoi to run smoothly, the translation system must be able to quickly and reliably react to translation requests, typically with multiple requests arriving at the same time. Since machine translation is a highly computationally demanding task, solutions as efficient as possible must be sought. The system must also contain error detection and recovery mechanisms to ensure uninterrupted operation of the service. Moreover, the solution must be naturally scalable to allow for flexible increase of computational power to reach higher performance if required by its customers' demand.

In this paper, we describe the structure of our translation system, and detail the results of several performance tests. We make the system available as free software, licensed under the Apache 2.0 licence.[1] MTMonkey 1.0 is published via the Lindat-Clarin repository,[2] updated code is released on GitHub and open for comments and further contributions.[3]

2. Pre-considerations

We build upon Moses (Koehn et al., 2007), a statistical machine translation system. Koehn (2013, Section 3.3.22) explains how to operate Moses as *Moses Server* responding to translation requests on a given port. Support for using multiple translation directions was originally available as *Using Multiple Translation Systems in the Same Server* (Koehn, 2013, p. 121), later to be replaced by more general *Alternate Weight Settings* (Koehn, 2013, p. 135), which is still under development and currently does not work with multi-threaded decoding. We therefore decided to handle different translation directions using separate stand-alone Moses Server instances.

Moses does not provide any built-in support for load balancing, which is needed to distribute the translation requests evenly among the Moses instances. We therefore explored RabbitMQ,[4] a robust open-source messaging toolkit which can be used to implement even complex application communication scenarios. However, we concluded that for our relatively simple task where the main focus is on efficiency, its overhead is unpleasant while the benefits it brings are only moderate. We therefore decided to implement our own solution for request distribution and load balancing.

We implement our solution in Python, which was chosen due to its relatively high efficiency combined with the comfortable programming experience it offers.

There are several remote procedure call (RPC) protocols available that could be used in our system. For the public API, we use JSON-RPC,[5] which is simple and lightweight in comparison to other RPC protocols, making it highly suitable for RPC

[1] http://www.apache.org/licenses/LICENSE-2.0

[2] http://hdl.handle.net/11858/00-097C-0000-0022-AAF5-B

[3] https://github.com/ufal/mtmonkey

[4] http://www.rabbitmq.com/

[5] http://www.jsonrpc.org/

over the Internet (other formats could be easily added if needed). Moses Server implements XML-RPC,[6] which is similar to JSON-RPC, although not as lightweight. We employ XML-RPC for the internal API as well, since it has a native Python implementation, which is more efficient and seamless than JSON-RPC Python libraries.

MTMonkey is in its architecture very similar to the MT Server Land system (Federmann and Eisele, 2010), which uses XML-RPC as a response format and focuses more on the possibility of comparing different MT systems for the same translation direction than on low-latency processing of a large number of simultaneous requests. A similar approach to ours was also taken by Arcan et al. (2013), who built a multilingual financial term translation system on top of Moses.[7] They make their system freely available through both a web GUI and a RESTful service, using JSON as the response format. They provide lists of n-best translations and allow the users to upload their own dictionaries, which are used to override the SMT system-generated translations. The WebTranslation toolkit[8] for translating web pages which is built into Moses also supports distributing translation requests to multiple instances of Moses servers but this solution is a proof of concept only and not designed for production environments.

3. Implementation

MTMonkey consists of an *application server* and a set of *workers*. The application server handles translation request arriving through the public API and uses the internal API to distribute them to the workers, which perform the translations. The system is able to handle multiple incoming translation requests by load balancing and queuing. Self-check mechanisms are also included. The architecture of the system is visualized in Figure 1 and described in detail in Sections 3.1–3.6.

The application server and workers are implemented in Python and are compatible with Python versions 2.6 and 2.7. The installation and support scripts are written in Bash. In addition, we provide a very simple PHP-based web client that allows for an easy interactive testing of the service and serves as an example client implementation. We tested the whole system under Ubuntu 10.04, but it should be able to operate on any Unix-like system.

3.1. Public API

The application server provides a public API based on the REST[9] principles, accepting requests over HTTP in the JSON format as objects with the following keys:

[6] `http://www.xmlrpc.com/`

[7] `http://monnet01.sindice.net/monnet-translation/`

[8] `http://www.statmt.org/moses/?n=Moses.WebTranslation`

[9] `http://en.wikipedia.org/wiki/Representational_state_transfer`

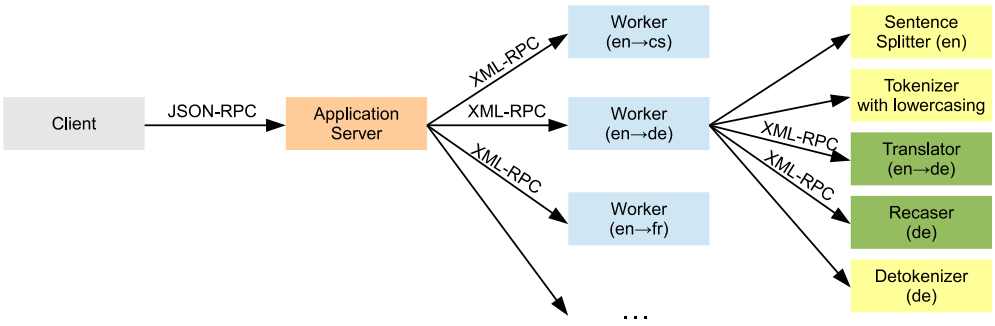

Figure 1. The overall architecture of the translation system. English-to-German translation is shown in detail.

sourceLang	the ISO 639-1 code of the source language (`cs`, `de`, `en`, `fr`);
targetLang	the ISO 639-1 code of the target language (`cs`, `de`, `en`, `fr`);
text	the text to be translated, in the UTF-8 character encoding;
detokenize	detokenize the translation (boolean);
alignmentInfo	request alignment information (boolean).

The response is a JSON object with the following keys:

errorCode	`0`, or error code;
translation	the translation, in the UTF-8 character encoding;
alignment-raw	alignment information (if requested by `alignmentInfo`) as a list of objects containing indexes of start- and end-tokens of corresponding source and target text chunks.

The only currently implemented advanced feature is the option to request alignment information, which can be used to determine which part of the input texts corresponds to which part of the translation. There are several other fields reserved for future use, such as `nBestSize` to request multiple translation options.[10] For simplicity, we omit description of parts of the API that are unused at the moment or that are only technical.

[10]Due to preparation for a future implementation of the `nBestSize` option, the actual structure of the response is more complicated than described, with the actual text of the translation being wrapped in an object that itself is a member of an array of translation options.

3.2. Application Server

The application server distributes incoming translation requests to individual workers. Available workers are listed in a simple configuration file – for each worker, its IP address, port, and translation direction (source and target language) are given. Because the workers are identified by a combination of the IP address and port number, there can be multiple workers on one machine listening on different ports.

If there are multiple workers available for a given translation direction, a simple round-robin load balancing is used. No other information, such as text length or worker configuration, is taken into account. However, we found that such a simple approach is sufficient for our needs, and at the same time it is fast enough not to unnecessarily increase the response time, making the application server lightweight enough to require only moderate computational resources. If more machines support several translation directions, a set of translation requests for that direction can be distributed relatively evenly among all the respective machines. The number of workers is potentially unlimited, i.e. the only limit is the available computational power.

3.3. Internal API

The application server communicates with workers through XML-RPC. A worker implements two XML-RPC methods:

process_task used to request a translation, returning the translated text (with additional information if requested, such as the alignment);

alive_check tests if the worker is running.

3.4. Workers

Each worker uses one instance of Moses providing translation in one direction and another instance of Moses that performs recasing. The only configuration parameters of a worker are the ports on which the Moses servers listen. The worker communicates with the Moses servers through XML-RPC. Workers run as multi-threaded XML-RPC servers which allows for transparent and light-weight asynchronous processing and parallelism. One physical machine may house multiple instances of a worker, each using its own MT system instance, providing translation in a different direction. Only the available RAM and hard drive space are the limits on the number of running worker instances.

3.5. Text Processing Tools

The input texts have to be preprocessed before translation. We use the usual pipeline of a sentence splitter and a lowercasing tokenizer. The sentence splitter is our reimplementation of the Moses sentence splitter in Python and uses the same nonbreaking prefixes definition files.

Due to our system being used as a component of a complex project, the sources of incoming translation requests are varied, and the texts to be translated can appear in various tokenizations. We therefore implemented our own language-independent tokenizer, which is robust with respect to possible pre-tokenization. We achieve this by "aggressive tokenization": splitting the text on any punctuation, including hyphens compounds and full stops in abbreviations (but keeping sequences of identical punctuation marks unsplit, as in "…"). Although such approach might hurt translation fluency, it helps prevent data sparsity. The same approach must be applied on the training data.

As a post-processing step, we use a Moses instance to perform recasing and a detokenizer, which is our reimplementation of the Moses detokenizer in Python.

3.6. Fault Recovery

To ensure uninterrupted operation, worker machines may be configured to perform scheduled self-tests and automatically restart the worker application as well as Moses servers in case of an error. We provide a testing script that may be added to the machines' *crontab*.

In addition, we run automatic external tests that are scheduled to translate a test sentence and notify the administrator of the service by e-mail in case of any error. These tests connect to the service in exactly the same way as other clients, i.e. they reflect the actual service state from the outside.

4. Evaluation

The evaluation presented in this section is focused on efficiency. We measure how fast the system is in serving various numbers of simultaneous requests.

4.1. System Configuration

We test the system using eight worker machines, each with four CPUs and 32 GB RAM. Each of the machines runs three worker instances (each for a different translation direction), i.e. there are four workers for each translation direction.

We use binarized models (for both the phrase-table and the language model) with lazy loading in Moses, which causes a slight decrease in translation speed.[11] However, this setup gives us more flexibility as it allows us to fit multiple instances of Moses into RAM on a single machine and begin translating almost instantly after starting the Moses servers. More details about the setup of the Moses translation system itself can be found in Pecina et al. (2012).

[11] The decrease in speed is noticeable even for batch translation using a single system.

4.2. Load Testing

To generate translation requests, we use two data sets, both created within the Khresmoi project. The first set consists of *sentences* from the medical domain with 16.2 words per sentence on average. The second set consists of medical search *queries* with an average length of 2.1 words per query.

In each of the tests, we run a number of clients simultaneously, either for one translation direction at a time, or for all six of them. Each of the clients sends 10 synchronous translation requests to the application server and reports the time elapsed for all of them to complete, which (divided by 10) gives the average response time. To test the scalability of our solution, we also run some of the tests with a reduced number of workers. The one-translation-direction tests were run separately for each of the six translation directions.[12] The tests were repeated 10 times with different parts of the test data.[13] The results were then averaged and the standard deviation was computed.

The results are shown in Table 1. We average the results over all translation directions since we observed that there are only little differences in performance with respect to the translation direction (less than 15% of the average response time). We can see that when moving from one client to 10 clients, the number of parallel requests rises faster than the average time needed to complete them. This indicates that the parallelization and load balancing function properly. However, the standard deviation is relatively large, which indicates that the load balancing probably could be improved. If we multiply the number of parallel requests by 10 one more time, the average request time gets also approximately multiplied by 10, indicating that the parallelization capacity has already been reached at that point.

The scalability tests revealed that with a large number of parallel requests, doubling the number of workers reduces the response time to approximately a half. This shows that the system scales well, with a possibility to reach low response times even under high load (the minimum average response time being around 550ms for sentence translations in our setup) provided that sufficient computational power is available.

In spite of the queries being more than seven times shorter than the sentences on average, the query translation was observed to be only up to five times faster than the sentence translation under low load, and becomes consistently only about twice as fast with higher numbers of parallel requests. This indicates that the length of the input texts is not as crucial for the system performance as other parameters.

[12] The 6-translation-directions tests were not run with 100 clients per direction since we are technically unable to run 600 clients in parallel.

[13] Except for the 100-client test which uses all of the data and was therefore run only once.

Data type	Translation directions	Clients per direction	Workers per direction	Response time [ms] avg	std dev
sentences	1	1	1	539	132
sentences	1	1	2	510	134
sentences	1	1	4	554	151
sentences	1	10	1	2,178	506
sentences	1	10	2	897	259
sentences	1	10	4	567	171
sentences	1	100	1	14,941	2,171
sentences	1	100	2	10,189	1,588
sentences	1	100	4	5,560	794
sentences	6	1	1	620	137
sentences	6	1	2	571	143
sentences	6	1	4	592	196
sentences	6	10	1	4,792	857
sentences	6	10	2	2,103	408
sentences	6	10	4	1,029	280
queries	1	1	4	112	29
queries	1	10	4	247	149
queries	1	100	4	2,593	526
queries	6	1	4	174	110
queries	6	10	4	545	91

Table 1. Load testing results.

5. Conclusion

We described a successful implementation of a machine translation web service that is sufficiently robust and fast enough to handle parallel translation requests in several translation directions at once and can be easily scaled to increase performance.

Our future plan is to implement worker hot-plugging for an even more flexible scalability, as currently adding or removing workers requires a restart of the application server. We also intend to add the drafted advanced features of the API, such as requesting and returning multiple translation options and their scores. We are also planning to develop a simple confidence-estimation module to assess the quality of produced translations.

We further plan to enrich the APIs with a method capable of retrieving diagnostic and statistical information, such as the list of supported translation directions, the number of workers for each translation direction, average response time or the number of requests served in the last hour. We would also like to add support for other MT decoders besides Moses.

Acknowledgements

The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement n° 257528 (KHRESMOI) and the project DF12P01OVV022 of the Ministry of Culture of the Czech Republic (NAKI – Amalach).

This work has been using language resources developed and/or stored and/or distributed by the LINDAT-Clarin project of the Ministry of Education of the Czech Republic (project LM2010013).

Bibliography

Arcan, Mihael, Susan Marie Thomas, Derek De Brandt, and Paul Buitelaar. Translating the FINREP taxonomy using a domain-specific corpus. In *Machine Translation Summit XIV*, Nice, France, 2013.

Aswani, Niraj, Thomas Beckers, Erich Birngruber, Célia Boyer, Andreas Burner, Jakub Bystroň, Khalid Choukri, Sarah Cruchet, Hamish Cunningham, Jan Dědek, Ljiljana Dolamic, René Donner, Sebastian Dungs, Ivan Eggel, Antonio Foncubierta-Rodríguez, Norbert Fuhr, Adam Funk, Alba García Seco de Herrera, Arnaud Gaudinat, Georgi Georgiev, Julien Gobeill, Lorraine Goeuriot, Paz Gómez, Mark Greenwood, Manfred Gschwandtner, Allan Hanbury, Jan Hajič, Jaroslava Hlaváčová, Markus Holzer, Gareth Jones, Blanca Jordan, Matthias Jordan, Klemens Kaderk, Franz Kainberger, Liadh Kelly, Sascha Kriewel, Marlene Kritz, Georg Langs, Nolan Lawson, Dimitrios Markonis, Ivan Martinez, Vassil Momtchev, Alexandre Masselot, Hélène Mazo, Henning Müller, Pavel Pecina, Konstantin Pentchev, Deyan Peychev, Natalia Pletneva, Diana Pottecherc, Angus Roberts, Patrick Ruch, Matthias Samwald, Priscille Schneller, Veronika Stefanov, Miguel A. Tinte, Zdeňka Urešová, Alejandro Vargas, and Dina Vishnyakova. Khresmoi: Multimodal multilingual medical information search. In *Proceedings of the 24th International Conference of the European Federation for Medical Informatics*, 2012. URL `http://publications.hevs.ch/index.php/attachments/single/458`.

Federmann, Christian and Andreas Eisele. MT Server Land: An open-source MT architecture. *Prague Bulletin of Mathematical Linguistics*, 94:57–66, 2010.

Koehn, Philipp. Moses, statistical machine translation system, user manual and code guide, July 2013. URL `http://www.statmt.org/moses/manual/manual.pdf`.

Koehn, Philipp, Hieu Hoang, Alexandra Birch, Chris Callison-Burch, Marcello Federico, Nicola Bertoldi, Brooke Cowan, Wade Shen, Christine Moran, Richard Zens, Chris Dyer, Ondřej Bojar, Alexandra Constantin, and Evan Herbst. Moses: Open Source Toolkit for Statistical Machine Translation. In *ACL 2007, Proceedings of the 45th Annual Meeting of the Association for Computational Linguistics Companion Volume Proceedings of the Demo and Poster Sessions*, pages 177–180, Prague, Czech Republic, June 2007. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/P/P07/P07-2045`.

Pecina, Pavel, Jakub Bystroň, Jan Hajič, Jaroslava Hlaváčová, and Zdeňka Urešová. Deliverable 4.3: Report on results of the WP4 first evaluation phase. Public deliverable, Khresmoi project, 2012. URL http://www.khresmoi.eu/assets/Deliverables/WP4/KhresmoiD43.pdf.

Address for correspondence:
Aleš Tamchyna
tamchyna@ufal.mff.cuni.cz
Institute of Formal and Applied Linguistics
Faculty of Mathematics and Physics,
Charles University in Prague
Malostranské náměstí 25
118 00 Praha 1, Czech Republic