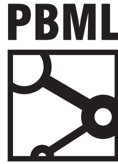

The Prague Bulletin of Mathematical Linguistics
NUMBER 94 SEPTEMBER 2010 57–66

MT Server Land: An Open-Source MT Architecure

Christian Federmann, Andreas Eisele

DFKI, German Research Center for Artificial Intelligence

Abstract

We describe the implementation of *MT Server Land*, an open-source architecture for machine translation that is developed by the MT group at DFKI. A *broker* server collects and distributes translation requests to several *worker* servers that create the actual translations. Users can access the system via a fast and easy-to-use web interface or use an XML-RPC-based API interface to integrate it into their applications. The source code is published under a BSD-style license and is freely available from GitHub[1].

1. Introduction

Easy-to-use machine translation (MT) services that are available via the internet are an important means to increase visibility of MT research and to help shaping the multi-lingual web. Applications such as *Google Translate* allow lay users to quickly and effortlessly create translations of texts or even complete web pages; the continued success of such services shows the potential that lies in *usable* machine translation, something both developers and researchers should strive for.

Despite impressive progress in recent times, MT can by far not be regarded as a solved problem, and the ongoing research on many levels requires careful analysis of existing systems that may vary along many dimensions or that may be hybrid solutions composed from building blocks taken from different paradigms. A significant number of existing systems from ongoing research projects should be made available to researchers from the field for a couple of reasons.

[1]You can download a copy of the code at `http://github.com/cfedermann/mt-serverland`.

© 2010 PBML. All rights reserved.
Corresponding author: cfedermann@dfki.de
Cite as: Christian Federmann, Andreas Eisele. MT Server Land: An Open-Source MT Architecure. The Prague Bulletin of Mathematical Linguistics No. 94, 2010, pp. 57–66. doi: 10.2478/v10108-010-0020-8.

For one, the ease of comparative evaluation would advance the understanding of merits and weaknesses and hence facilitate progress towards higher quality in MT. But the easy availability of systems would also allow researchers and developers from related areas to use MT functionality as building blocks in a larger context. Areas that would benefit most from this include efforts towards computer-aided translation (CAT) platforms, cross-lingual search and question answering, easy deployment of multilingual websites, knowledge acquisition from multilingual document repositories, and many more.

Beyond such groups, also decision makers from language industry and large organisation that are potential users of MT functionality should be given easy access to the existing functionality in order to allow them to judge the potential of such systems for specific applications.

Last not least, the general public, who often takes the offerings of service providers like Google or Microsoft to be representative of the current state of the art in MT, should be given a chance to compare these services against the functionality provided by ongoing research. In the context of ongoing MT research projects at DFKI's language technology lab, such as EuroMatrixPlus, ACCURAT or TaraXÜ, we have decided to design and implement such a translation application. We have published the source code as open-source and hope that it becomes a useful tool for the MT community.

2. Scope and Requirements

Considering some intended usages of the toolkit, we have collected a set of requirements our software should meet. We are planning for a staged delivery, where subsequent releases of the software will meet an increasing number of the requirements and where the priorities concerning the next round will be determined based on experience collected with active usage of the system as it was already delivered, in a set of realistic applications. The requirements can be grouped into core functionalities, important extensions, and features that would be useful in advanced applications.

Core Functionalities: A central requirement for the toolkit is to provide a single entry point to multiple MT engines for multiple users. The system should also support multiple language pairs and multiple MT engines per language pair, including different types of engines (SMT, RBMT, hybrid MT) and multi-engine setups, as well as variants of systems optimized for multiple application domains, text types, and styles. The system should provide access both via user-friendly, web-based interaction, as well as programmatically via a simple, yet powerful API such as a Web service.

Important Extensions: The system should allow to assign appropriate roles to each user (e.g. not every user should have access to every system, some user may have priority over others, etc.). The system should support many concurrent translation requests and multiple installations of the engines on different computers. It

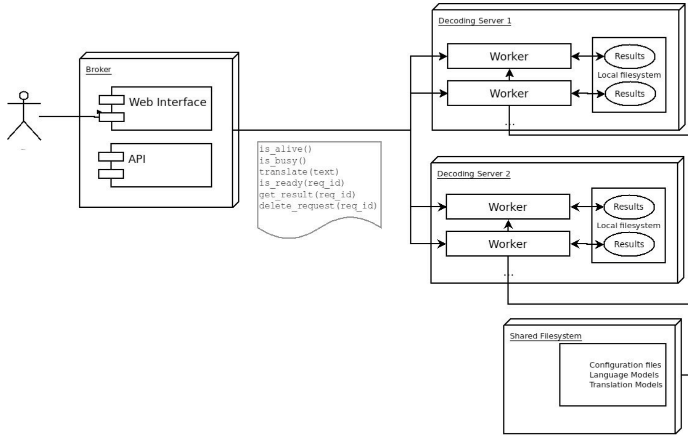

Figure 1. Overview of the System Architecture

should make sure that work is distributed over available resources via queuing and load balancing. The system should be able to recognize and handle exceptional circumstances caused by failure of engines and communication. The system should minimize the required administrative effort, even under heavy load.

Advanced Features: The sytem should be able to pass not only translation in- and output between users and MT engines, but also additional data generated by the engines, such as alignments, results of intermediate processing steps, as far as the engines are able to generate these. It should allow users to pass in additional information to the engines that will allow the engines to adapt to the needs of the each user (personalization, incremental training). Furthermore, it should provide auxiliary functionality, such as splitting of longer documents into paragraphs and sentences, tokenisation, case normalisation.

3. System Architecture

In this section, we will give an overview on the system's general architecture and the several components it is composed of. Figure 1 shows a bird's-eye view on the MT Server Land application. A similar application has been described in (Victor M. Sanchez-Cartagena, 2010).

3.1. Overview

The system consists of two different layers: first, we have the *broker* server that handles all direct requests from end users or API calls alike. Second, we have a layer of so-called *worker* servers, each implementing some sort of machine translation func-

```
package serverland;

message TranslationRequestMessage {
  required string request_id = 1;        // Random UUID-4 32-digit hex number
  required string source_language = 2;    // ISO 639-2 language codes
  required string target_language = 3;
  required string source_text = 4;        // UTF-8 encoded texts
  optional string target_text = 5;

  message KeyValuePair {
    required string key = 1;
    required string value = 2;
  }

  repeated KeyValuePair packet_data = 6;  // Contains additional request data
}
```

Figure 2. TranslationRequestMessage .proto definition

tionality. All communication between users and workers is channeled through the broker server which acts as a central "proxy" server. For users, both broker and workers "constitute" the MT Server Land application.

Human users connect to the system using any modern web browser, API access can be implemented using XML-RPC calls. It would be relatively easy to extend the API interface to support other protocols such as SOAP or REST. By design, all internal method calls that connect to the worker layer have to be implemented with XML-RPC. In order to prevent encoding problems with the input text, we send and receive all data encoded as Base64 Strings between broker and workers; the broker server takes care of the necessary conversion steps.

3.2. Broker Server

The broker server has been implemented using the *django web framework* which takes care of low-level tasks and allows for rapid development and clean design of components. We have used the framework for other project work before and think it is well suited to the task. More information on django can be found on the project website which is available at `http://www.djangoproject.com/`, the framework itself is available under an open-source BSD-license.

3.2.1. Translation Request Messages

Each translation request is defined by a unique *request id*, a *source* and *target* language as well as a *source text*. After the translation has been produced, the request will also contain the *target translation* and, for some worker implementations, *additional data* such as log files, alignment information or even parse trees that have been returned from the translation engine.

In order to allow flexible serialization of translation requests, we have implemented them using Google Protocol Buffers (Google, 2010b). Our `.proto` definition is shown in Figure 2, it can be compiled into Python code using the following command:

```
$ protoc --python_out=workers/ TranslationRequestMessage.proto
```

This will create a new Python file named `TranslationRequestMessage_pb2` inside the `workers/` folder of our MT Server Land application. Using protocol buffers allows to easily serialize Python instances to a binary representation and vice versa, something that has proven to be very useful during the development of the system.

3.2.2. Object Models

The broker server implements two main object *django models* which we describe below. Please note that we have also developed additional object models, e.g. for quota management or API access authentication. See the MT Server Land source code for more information.

A `WorkerServer` instance stores all information related to a remote worker server. This includes the respective hostname and port address as well as a name and a short description. In fact, this is just a shallow wrapper around the XML-RPC interface.

The `TranslationRequest` model represents an external translation job and related information such as the chosen worker server, the assigned request id and additional information about the creation date or the owner. We also prepare some fields for caching of translation request state. Please note that neither *source* nor *target* texts are stored within the django instance; instead they are kept in form of a serialized `TranslationRequestMessage` file which is named by the request id and stored in a configurable location on the broker server's hard disk.

3.2.3. User Interface

We developed a browser-based web interface to access and use the MT Server Land application. End users first have to authenticate before they can access their *dashboard* which lists all known translation requests for the current user and also allows to create new requests. Once a translation request has been completed by the chosen worker server, the result is transferred to the broker server's data storage, deleting the request data from the worker server. The user can view the result within the dashboard or download the file to a local hard disk. It is also possible to delete "pending" translation requests at any time, effectively terminating the corresponding thread within the connected worker server.

3.2.4. API Interface

In parallel to the browser interface, we have designed and started to implement an API that allows to connect applications to the MT functionality provided by our service using XML-RPC. Again, we first require authentication before any machine translation can be used. We plan to use so-called *auth tokens*, i.e. randomly generated 32-digit hexadecimal numbers which are bound to a certain user account, for this. We provide methods to list all requests for the current "user" (i.e. the application account) and to create, download, or delete translation requests. Extension to REST or SOAP protocols is possible. Again, serialized `TranslationRequestMessage` objects are used to exchange requests between the user's application and the MT Server Land.

3.2.5. Starting the Broker Server

Like any other django project, the broker server can be started in *debug mode* using the `python manage.py runserver` command. For internal deployment of the system, we have used the *lighttpd web server* which is a lightweight, fast and open-source web server that can be easily combined with a django application. More information can be found on the project website which is available at `http://www.lighttpd.net/`. We have configured the web server to serve all django media files and send all other requests to the django FCGI server that runs in a background process. A sample server configuration file `lighttpd-django.conf` and startup/stop scripts for django's FCGI mode are contained in the source code release package.

3.3. Worker Servers

Actual machine translation functionality is implemented by a layer of so-called worker servers that are connected to the central broker server. We have created a Python-based `AbstractWorkerServer` class which is the foundation for all worker implementations. The basic worker interface is described next.

Attributes: `finished`: Boolean that controls the main server loop. Defaults to `False`. `server`: The actual `SimpleXMLRPCServer` instance is bound here. `jobs`: Dictionary memorizing all translation requests the worker has accepted. Maps request ids as keys to `Process` objects that represent the actual worker threads. Request ids are random 32-digit hexadecimal UUID numbers.

General Methods: `__init__`: Constructor, takes care of setting up the logging and creates the actual XML-RPC server instance. `start_worker`: Starts the main server loop that handles requests. `stop_worker`: Sets `finished` to `True` and terminates all running translation processes. Intermediate results are lost, the file storage of the worker server should be cleaned afterwards to avoid keeping invalid requests.

Status Methods: `list_requests`: Returns a list of all registered translation request ids. `is_alive`: Returns `True` to signal that the worker server is up and running. `is_busy`: Checks whether the worker server is currently processing requests.

is_ready: Checks whether the request with the given request id is finished. is_valid: Checks whether the request id is valid, i.e. contained within jobs.

Translation Methods: language_pairs: Returns a read-only tuple containing tuples that encode the available language pairs which are supported by this translation engine. All languages are identified by ISO 639-2 codes[2]. language_code Converts the given ISO 639-2 code into the internal representation of language codes used by the worker's translation engine. start_translation: Takes the given serialized TranslationRequestMessage object, creates a local copy inside the worker server's /tmp/ folder and then starts a Process that calls the handle_translation handler. fetch_translation: Retrieves the translation result for the given request id if already available. Otherwise returns an empty String. delete_translation: Deletes the translation request with the given request id from the jobs dictionary, terminating the connected process if still running. handle_translation: Implements the actual translation functionality of a worker implementation. Custom worker servers need to overwrite this method.

3.3.1. Example: Implementing a Google Translate Worker

Worker servers can be implemented by subclassing AbstractWorkerServer and creating a custom handle_translation method. The listing in Figure 3 shows the actual code for a "Google worker" server that sends its input text to Google Translate and extracts the translation from the resulting website.

3.3.2. Worker Server Implementations

We have implemented worker servers for several MT systems:

- **Lucy RBMT**: our Lucy (Alonso and Thurmair, 2003) worker is implemented using an internal Lucy Server mode wrapper. Due to the system's architecture, this has to be run on a Windows machine. The actual worker code can be started on any platform.
- **Moses SMT**: a Moses (Koehn et al., 2007) worker is configured to serve exactly one language pair. We use the Moses Server mode to keep translation and language model in memory which helps to speed up the translation process.
- **Joshua SMT**: similar to the Moses worker, we have created a Joshua (Li et al., 2009) worker that works by creating a new Joshua instance for each translation request.

We have also created worker servers for popular online translation engines such as Google Translate, Microsoft Translator and Yahoo! Babel Fish which already makes available a huge number of language pairs for use in MT research contexts.

[2] See http://www.loc.gov/standards/iso639-2/ for more information.

```
import re, sys, urllib, urllib2
from worker import AbstractWorkerServer
from TranslationRequestMessage_pb2 import TranslationRequestMessage

class GoogleWorker(AbstractWorkerServer):
    """ Implementation of a worker server that connects to Google Translate. """
    __name__ = 'GoogleWorker'

    def language_pairs(self):
        """Returns a tuple of all supported language pairs for this worker."""
        languages = ('afr', 'alb', 'ara', ..., 'vie', 'wel', 'yid')
        return tuple([(a,b) for a in languages for b in languages if a != b])

    def language_code(self, iso639_2_code):
        """Converts a given ISO-639-2 code into the worker representation."""
        mapping = { 'afr': 'af', 'alb': 'sq', ... 'wel': 'cy', 'yid': 'yi' }
        return mapping.get(iso639_2_code)

    def handle_translation(self, request_id):
        """Translation handler that connects to Google Translate."""
        handle = open('/tmp/{0}.message'.format(request_id), 'r+b')
        message = TranslationRequestMessage()
        message.ParseFromString(handle.read())

        source = self.language_code(message.source_language)
        target = self.language_code(message.target_language)
        the_url = 'http://translate.google.com/translate_t'
        the_data = urllib.urlencode({'js': 'n', 'sl': source, 'tl': target,
          'text': message.source_text.encode('utf-8')})
        the_header = {'User-agent': 'Mozilla/5.0'}

        opener = urllib2.build_opener(urllib2.HTTPHandler)
        http_request = urllib2.Request(the_url, the_data, the_header)
        http_handle = opener.open(http_request)
        content = http_handle.read()
        http_handle.close()

        result_exp = re.compile('<textarea name=utrans wrap=SOFT ' \
          'dir="ltr" id=suggestion.*>(.*?)</textarea>', re.I|re.U)
        result = result_exp.search(content)

        if result:
            message.target_text = unicode(result.group(1), 'utf-8')
            handle.seek(0)
            handle.write(message.SerializeToString())

        handle.close()
```

Figure 3. Source code for the Google Translate worker

4. Basic Usage

The MT Server Land code can be obtained from GitHub and extracted to a local folder named `serverland/` using the following command:

```
$ git clone git://github.com/cfedermann/mt-serverland.git serverland
```

After downloading the source code, we need to create a database for the project. This can be done using the manage.py syncdb command, as shown below:

```
$ python manage.py syncdb
```

It is mandatory to create a superuser account during the syncdb step. We also provide a sample development.db file with a sample user admin:admin at the GitHub repository[3]. It is now possible to startup django in development using manage.py runserver, as we have already mentioned. However, before any translation work can be done, at least a single worker server instance has to be started and registered inside the django database.

The available worker server implementations can be found inside workers/. We also provide scripts to start and stop worker server instances. To startup the Google Translate worker server, we have to start it using the following command:

```
$ ./start_worker.py GoogleWorker localhost 1234
```

This will create a new GoogleWorker instance serving from http://localhost:1234/. In order to make this worker instance accessible from the MT Server Land system, we have to register it inside the broker server's database. For this, we access the django administration backend (which is available at http://127.0.0.1:8000/admin/) and create a WorkerServer object pointing to the correct host and port address. After the worker server has been created, authenticated users can create new translation requests which are then processed by the respective worker server.

5. Conclusion and Future Work

We have presented an open-source architecture for machine translation. The system can flexibly be extended and allows lay users to make use of MT technology within a web browser or by using XML-RPC method calls from custom applications. A central broker server receives requests from clients and dispatches them to a layer of worker servers that take care of the translation duties. We have used open-source software to build the system and have released the source code under a BSD-style license.

5.1. Open-Source Development

We hope that the MT Server Land software will benefit from and grow by being maintained as an open-source project. We have opted for hosting at the GitHub platform as this guarantees transparent development and ensures open access to the

[3] At http://github.com/downloads/cfedermann/mt-serverland/mt-serverland-development.db

source code. We continue to extend the MT Server Land code and available worker servers, possibly starting at the Machine Translation Marathon in Le Mans for which we are currently preparing project ideas related to the MT Server Land platform.

Acknowledgments

We would like to thank all members of the MT Group at DFKI for testing the MT Server Land prototype and for all their helpful feedback during the development of this software. This work was supported by the EuroMatrixPlus project (IST-231720) which is funded by the European Community under the Seventh Framework Programme for Research and Technological Development.

Bibliography

Alonso, Juan A. and Gregor Thurmair. The Comprendium Translator system. In *Proceedings of the Ninth Machine Translation Summit*, New Orleans, USA, 2003.

Google. Google Translate, 2010a. URL `http://translate.google.com/`.

Google. Google Protocol Buffers, 2010b. URL `http://protobuf.googlecode.com/`.

Koehn, Philipp, Hieu Hoang, Alexandra Birch, Chris Callison-Burch, Marcello Federico, Nicola Bertoldi, Brooke Cowan, Wade Shen, Christine Moran, Richard Zens, Chris Dyer, Ondrej Bojar, Alexandra Constantin, and Evan Herbst. Moses: Open source toolkit for statistical machine translation. In *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/P07-2045`.

Li, Zhifei, Chris Callison-Burch, Chris Dyer, Sanjeev Khudanpur, Lane Schwartz, Wren Thornton, Jonathan Weese, and Omar Zaidan. Joshua: An open source toolkit for parsing-based machine translation. In *Proceedings of the Fourth Workshop on Statistical Machine Translation*, pages 135–139, Athens, Greece, March 2009. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/W/W09/W09-0x24`.

Microsoft. bing Translator, 2010. URL `http://www.microsofttranslator.com/`.

Victor M. Sanchez-Cartagena, Juan Antonio Perez-Ortiz. ScaleMT: a Free/Open-Source Framework for Building Scalable Machine Translation Web Services. In *Open Source Tools for Machine Translation, MT Marathon 2010*, Dublin, Ireland, 2010.

Yahoo! Yahoo! Babel Fish, 2010. URL `http://babelfish.yahoo.com/`.

Address for correspondence:
Christian Federmann
`cfedermann@dfki.de`
Stuhlsatzenhausweg 3, D-66123 Saarbrücken, GERMANY