REMEDI

Robust and Efficient Machine Translation in a Distributed Infrastructure

Bi-Annual Report: Month 18

Principal Investigator:

Dr. Christof Monz
Informatics Institute
University of Amsterdam
Science Park 904
1098 XH Amsterdam

Phone: +39 (0)20 525 8676
Fax: +39 (0)20 525 7940
E-mail: c.monz@uva.nl

Scientific Programmer:

Dr. Ivan S. Zapreev
Informatics Institute
University of Amsterdam
Science Park 904
1098 XH Amsterdam

E-mail: i.zapreev@uva.nl

Month	Deliverable	Progress
6	Implementation of language model data structure; implementation of multi-threading with large shared data structures; month 6 report	✓
12	Implementation of translation and reordering model data structures; implementation of decoder search and pruning strategies; month 12 report	✓
18	Implementation of search lattice data structure and feature passing; implementation of server front end; implementation of robustness preserving load balancing and restarting; complete software deliverable of decoder infrastructure; month 18 report	✓
24	Implementation of language model server in distributed environment; implementation of distributed translation model service; month 24 report	
30	Implementation of distributed reordering model services; complete software deliverable of distributed translation infrastructure; Implementation of k-best hypothesis extraction; month 30 report	
36	Implementation of Margin Infused Relaxed Algorithm for parameter tuning; implementation of distributed optimization infrastructure, including hyper-parameter estimation; final report; complete manual	

Contents

1 Introduction

This REMEDI deliverable extends the previous deliverable, described in [ISZ16a], by introducing the following new functionalities:

- **Search lattice generation** - storing the decoding data needed for translation tuning;
- **Load balancing** - distributing the translation load between multiple decoder instances;
- **Web client** - a web client application for the translation infrastructure;
- **Text pre/post-processing** - sending source and target texts for pre/post- processing;
- **Translation priorities** - allowing to give translation jobs different priorities;

As before, our software keeps following the client/server architecture based on the WebSockets protocol [Web11]. For this deliverable, our software package was completed with the following applications, turning it into a complete translation system infrastructure:

- **bpbd-balancer** - the load balancer that has the same WebSockets interface as the **bpbd-server**[1] and is supposed to distribute load between multiple translation server instances;
- **bpbd-processor** - the text pre/post-processing server that allows to employ various third party text processing tools for different languages. Its purpose is to prepare text for translation and to post process the translated text to make it look more like the original;
- **translate.html** - a light web client to communicate with the translation infrastructure for performing the translation jobs;

The rest of the report is organized as follows: Section 2 describes the structure of the deliverable. Section 3 provides all necessary information on the new software pieces. Section 4 concludes and indicates some future work directions.

[1]The translation server application was delivered previously.

2 Deliverables

This deliverable consists of this report as well as software deliverables. Both are distributed as a downloadable archive with the following standard structure that will be re-used for the subsequent deliveries:

- **`REMEDI/month-18/data/`** - stores example models and query files
- **`REMEDI/month-18/software/`** - stores the software project
- **`REMEDI/month-18/report/`** - stores this document

Next, we provide some details on the structure of the software part of the delivery. The software components are located in `REMEDI/month-18/software/`, and are accompanied by an extended markdown [Gru15] document file `README.md` providing all the necessary software details. Similar, but shorter, information is also provided in Section 3 of this document. The delivered software is a standard Netbeans 8.0.2 C++ project, that can also be build using cmake and make, and its top-level structure is as follows:
[Project-Folder]/

`doc/` - project-related documentation

`ext/` - external libraries used in the project

`inc/` - C++ header files

`src/` - C++ source files

`script/` - stores various scripts

`script/web/` - web client for translation system

`script/text/` - dummy pre/post-processing scripts

`script/test/` - scripts used for testing

`nbproject/` - Netbeans project data

`server.cfg` - example server configuration file

`balancer.cfg` - example load balancer configuration file

`processor.cfg` - example processor configuration file

`LICENSE` - code license (GPL 2.0)

`CMakeLists.txt` - the cmake build script

`README.md` - project information document

`Doxyfile` - Doxygen configuration file

3 Software Details

In this section, we discuss the main aspects of the delivered software. This is a cumulative release, so for the information on licensing, software structure, building, and running[2] we refer to the previous delivery report [ISZ16a] and the software documentation [ISZ16b]. In this section we will concentrate on the new functionality and provide a few details of the implementation aspects. As before, we produce an open-source and freely-distributed GPL2.0 (http://www.gnu.org/licenses/) licensed software that is available as a github project:

https://github.com/ivan-zapreev/Basic-Translation-Infrastructure

The rest of this section is organized as follows: Section 3.1 lists the new software functionality. Section 3.2 describes how the software can be run. Section 3.3 talks about the main elements of the code design.

3.1 Functionality

In this section we first report on the project-imposed functional and non-functional requirements of the deliverable and indicate to what extent they have been achieved. Further, we talk about additional points of this delivery.

Page 5 of the REMEDI proposal [Mon14] contains the "Bi-Annual Deliverable-Payment Chart" table storing the required elements of the 18-month's delivery. Below, we match these points with this delivery content to show that we successfully fulfilled all of them and more.

Implementation of search lattice data structure and feature passing (S1, WP1): In order to obtain the best performance of the translation system one can employ Discriminative Training, see Chapter 9 of [Koe10]. The latter uses generated word lattice, c.f. Chapter 9.1.2 of [Koe10], to optimize translation performance by reducing some measure of translation error. This is done by tuning the translation parameters such as feature lambda values of the model feature weights. Our software allows for word lattice generation, as explained in Section 3.2.1 of this document.

Implementation of server front end (S1, WP1): Previously, we have delivered the **bpbd-client** application whose purpose is to send translation requests to the translation infrastructure. This is a command-line Linux application and thus its usability is limited. With the global availability of the Internet and intranet infrastructures it has been decided to provide the translation system with a light-weight web interface. This interface has been implemented in a form of thin client based on the modern web technologies, such as: HTML5 [HTM14], CSS [CSS16], JavaScript [Jav15], Bootstrap [Boo16], JQuery [jQu16], and WebSockets [Web11]. For the sake of better usability, this client was made to be able to communicate to the translation infrastructure directly without using any web server. For more details on the delivered Web client application see Section 3.2.3.

[2]Running of the previously delivered software components.

Implementation of robustness preserving load balancing and restarting (S1, W1): We deliver a developed load-balancing server application that can be used for the following purposes:

- Distribute load between multiple, same source-target language pair, translation servers;
- Aggregate multiple translation servers for different source-target language pairs;
- Aggregate balancer and translation servers providing a single infrastructure entry point;

For information on configuring and running the balancer application see Section 3.2.4. Note that the *restarting* is ensured by the following: *(i)* the software is ensured to be robust and thus does not require restarts, see Section 3.5; *(ii)* each of the delivered server applications can be safely stopped through its server console; *(iii)* in case a translation server goes offline, the load balancer automatically tries to reconnect to it with a given/configurable re-connection timeout.

Complete software deliverable of decoder infrastructure (S1, W1): The complete decoder infrastructure consists of a decoder server[3], pre/post- processing server, and tuning capable software. For this delivery, we have developed tuning script wrappers which allow to trivially integrate out software into the Oyster tuning infrastructure[4]. We briefly describe this in Section 3.2.1. In addition to that, we have created a text processing server capable of incorporating one or more third party text pre/post-processing scripts. Details on the text processing server can be found in Section 3.2.5.

Month 18 report (R3): This is the document you are reading now.

Additional points: Despite the ambitious aims, we did our best and extended this delivery with supplementary, yet very important, features such as:

- Translation job priorities, described in Sections 3.2.2 and 3.2.3;
- Software design diagrams, see Section 3.3;
- Software deployment diagrams, see Section 3.4;
- Extended validation and testing, see Section 3.5:
  - Valgrind [ABBF$^+$00] profiled source code;
  - Server stress testing with multiple clients;
- Extended tool documentation, see the README.md [ISZ16b];

It is important to note that, the translation job priorities apply to the entire translation infrastructure and not just to the translation server instances. In other words, these priorities are also taken into account by the load balancer and the text processor servers.

[3]The translation server application was delivered previously.

[4]Oyster is a home-brewed python-based translation infrastructure that is meant to be substituted with the infrastructure developed within the REMEDI project.

3.2 Running software

This section briefly covers how the provided software can be used for performing text translations. We begin with the changes made to the **bpbd-server** and **bpbd-client** applications. Further we introduce new software pieces: **bpbd-balancer**, **bpbd-processor**, and **translate.html**. For more details on how the software is to be configures and run, see the *"Using software"* section of README.md or follow the URL:

https://github.com/ivan-zapreev/Basic-Translation-Infrastructure#using-software

3.2.1 Translation server: bpbd-server

The translation server application has been extended with two main features:

1. The translation request priorities;
2. Word lattice generation for the tuning mode;

The former is simply introduced as a priority parameter in a translation job request message received by the server and is further seamless in the server interface, configuration parameters or the server console. The translation job priority is defined by an integer value. Clearly, the higher the value is the sooner the job is to be processed by the server.[5] The default priority value is zero - indicating normal or neutral priority. Jobs with equal priorities are handled at the first-come-first-serve basis. The translation jobs of a given priority are not served until all the jobs of the higher priorities are taken care of.

The latter is only available if the software is compiled in the tuning mode. The reason for that is that if the software is compiled in the tuning mode, it is a number of times slower than in the regular, i.e. production, mode. Enabling of the tuning mode can be done by setting the value of the `IS_SERVER_TUNING_MODE` macro, in the `./inc/server/server_configs.hpp` file, to *true* and then re-compiling the software.[6] The tuning mode only has impact on the bpbd-server executable.

If the server is compiled in the tuning mode, then the word lattice generation can be enabled through the options in the server's configuration file. The options influencing the lattice files generation are as follows:

```
[Decoding Options]
   #The the tuning word lattice generation flag; <bool>:
   #This flag only works if the server is compiled with
   #the IS_SERVER_TUNING_MODE macro flag set to true,
   #otherwise it is ignored, i.e. is internally re-set
   #to false.
   de_is_gen_lattice=true
```

[5]The same priority rule applies to the load balancer, and the text processor servers.

[6]It is recommended to `make clean` the project first in order to avoid possible caching issues of the object files.

```
#Stores the lattice data folder location for where the
#generated lattice information is to be stored.
de_lattices_folder=./lattices

#The file name extension for the feature-id-to-name
#mapping file needed for tuning. The file will be
#generated if the lattice generation is enabled. It
#will have the same name as the config file plus this
#extension.
de_lattice_id2name_file_ext=feature_id2name

#The file name extension for the feature-scores file
#needed for tuning. The file will be generated if the
#lattice generation is enabled. It will have the same
#name as the session id plus the translation job id
#plus this extension.
de_feature_scores_file_ext=feature_scores

#The file name extension for the lattice file needed
#for tuning. The file will be generated if the lattice
#generation is enabled. It will have the same name as
#the session id plus the translation job id plus this
#extension.
de_lattice_file_ext=lattice
```

The lattice generation will be enabled if the value of the `de_is_gen_lattice` parameter is set to *true*. The word lattice is generated per source sentence and consists of a translation hypothesis graph and employed feature weights. The word lattice format is conform to that of the Oister translation system. The lattice files, are stored in the folder specified by the `de_lattices_folder` parameter.

The lattice files employ internal feature ids to identify the features used to compute a score of each hypothesis. To map these ids to the feature names, found in the server config file, a global id-to-feature-name file mapping can be generated. This file is placed in the same folder where the translation server is being run. Also, the mapping is only generated if the server is started with the `-f` command line option/flag. The latter is exclusively enabled if the server is compiled in the tuning mode. If started with this flag, the server exits right after the mapping is generated and does not load the models or attempts to start the WebSockets server. The name of the id-to-feature-name file is defined by the server config file name, with the file-name extension defined by the config file parameter: `de_lattice_id2name_file_ext`.

In the lattice files, each sentence gets a unique sentence-id, corresponding to its position in the test source file. The sentence ids start from 1. Each sentence's lattice consists of two files with the name being the sentence id and the extensions defined by the `de_feature_scores_file_ext` and `de_lattice_file_ext` parameter values:

- `<sentence-id>.de_lattice_file_ext` - stores the graph of the translation process: the partial hypothesis and the transitions between them attributed with source and target phrases and added costs.
- `<sentence-id.de_feature_scores_file_ext` - stores raw feature weight values (i.e. without lambdas coefficients) used during the hypothesis expansion process.

For additional information on the lattice file formats see Section "Word lattice files" of [ISZ16b].

Once the translation process, with word lattice generation, is finished `de_lattices_folder` folder stores the lattice information files for each of the translated sentences. In order to combine them together into just two larger files, storing lattice graphs and feature scores for all sentences, one can to use the provided `./script/combine-lattices.sh` script. The synopsis of this script is self explanatory:

```
$ combine-lattices.sh
Usage: ./combine-lattices.sh <lattice-dir> <result-file-name>
                             <sent-lattice-ext> <set-scores-ext>
    <lattice-dir> - the directory with the lattice files
    <result-file-name> - the file name to be used for the
                         combined lattice data
    <sent-lattice-ext> - the lattice file extension for a
                         sentence, default is 'lattice'
    <set-scores-ext> - the feature scores file extension for
                       a sentence, default is 'feature_scores'
```

3.2.2 Command-line client: bpbd-client

The client application was delivered before and despite multiple internal changes and improvements, it got just a few interface changes, clearly visible from its synopsis:

```
$bpbd-client
<...>
PARSE ERROR:
             Required arguments missing:
             output-file, input-lang, input-file

Brief USAGE:
   bpbd-client  [-d <error|warn|usage|result|info|
                 info1|info2|info3>] [-c]
                [-s <the translation priority>]
                [-l <min #sentences per request>]
                [-u <max #sentences per request>]
                [-p <post-processor uri>]
                [-t <server uri>]
                [-r <pre-processor uri>]
```

```
                  [-o <target language>]
                  -O <target file name>
                  -i <source language>
                  -I <source file name>
                  [--] [--version] [-h]

For complete USAGE and HELP type:
   bpbd-client --help
```

Below we consider the new and changed options of the client application:

- `[-s <the translation priority>]` - the translation priority is an optional integer parameter with a default value 0. The higher the priority the sooner the job will be considered by the balancer, text processor and translation servers.
- `[-t <server uri>]` - the server URI of the translation system, here only the shot parameter name is changed, from "`-s`" to "`-t`".
- `[-r <pre-processor uri>]` - the parameter allowing to indicate the pre-processor server URI, if not specified, no pre-processing is done.
- `[-p <post-processor uri>]` - the parameter allowing to indicate the post-processor server URI, if not specified, no pre-processing is done.
- `-i <source language>` - in case text pre-processing is enabled, the source language parameter value can be set to `auto`, indicating that the source language is to be detected.

3.2.3 Web client: translate.html

The web client for the translation system is just a web application that uses the WebSockets API of HTML5 to sent JSON [JAS06] requests to the text processor, translation server or to a load balancer. The web client can be activated by opening:

`./script/web/translate.html`

in any modern web browser. At the moment, the client was tested and proven to work in the following 64-bit browsers under `OS X EI Capitan`:

- Opera 38.0.2220.41,
- Safari 9.1.1 (11601.6.17)
- Google Chrome 51.0.2704.103
- Firefox 47.0.

The web client interface is shown in Figure 1. As one can see, it is simple and intuitive, serving the main purpose of connecting to the translation infrastructure to perform translations. The input (source) text can be typed into the text area on the left by hand or loaded from a file. The translated (target) text can be found in the text area on the right and can be downloaded to the hard drive. The resulting text is annotated with a pop-up information, visible by hovering over the text with a mouse pointer. Note that, the translation priorities in the web interface have the same meaning and functionality as in the command line translation client, see Section 3.2.2.

Most of the interface controls have tool tips. Yet for the sake of completeness below, in Figure 2, we provide an annotated screen shot of the interface.

3.2.4 Load Balancer: bpbd-balancer

The load balancer server is meant to be used for the following purposes:

1. Distribute load between translation servers (same source-target language pairs);
2. Aggregate multiple translation servers (different source-target language pairs);
3. Create different topology of balancer and translation server deployments;

The balancer server is a multi-threaded application supporting translation job priorities. The use of the balancer server is straightforward. When started from a command line with no arguments, **bpbd-balancer** reports on the available command-line options:

```
$ bpbd-balancer
<...>
PARSE ERROR:
            Required argument missing: config

Brief USAGE:
  bpbd-balancer  [-d <error|warn|usage|result|info|
                      info1|info2|info3>] -c
                  <balancer configuration file>
                  [--] [--version] [-h]

For complete USAGE and HELP type:
  bpbd-balancer --help
```

As one can see the only required command-line parameter of the server is a configuration file. The latter shall contain the necessary information for running the balancer server and connecting to translation servers. Once the load balancer is started there is still a way to change some of its run-time parameters. The balancer configuration file and the server console are discussed below.

Balancer config file: In order to start the load balancer one must have a valid configuration file for it. The latter stores the minimum set of parameter values needed to run the balancer server. Among other things, this config file specifies the location of the translation servers to connect to.

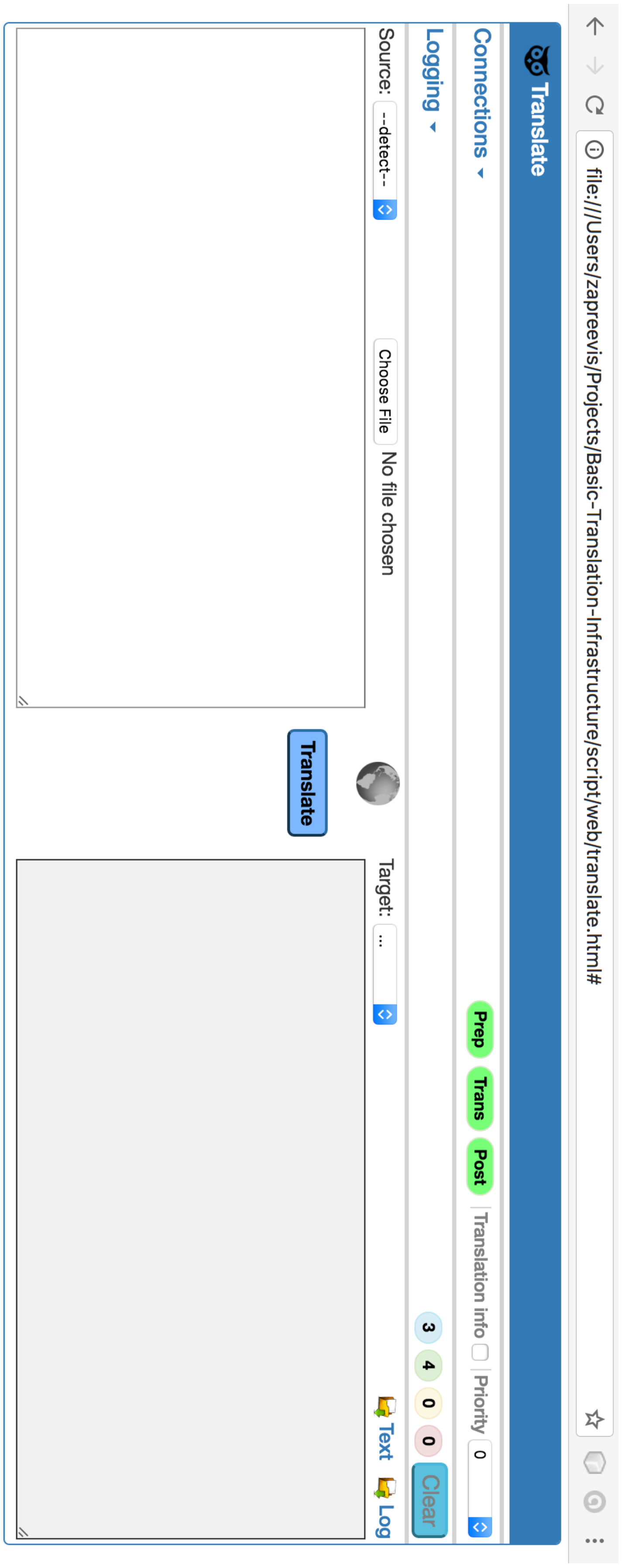
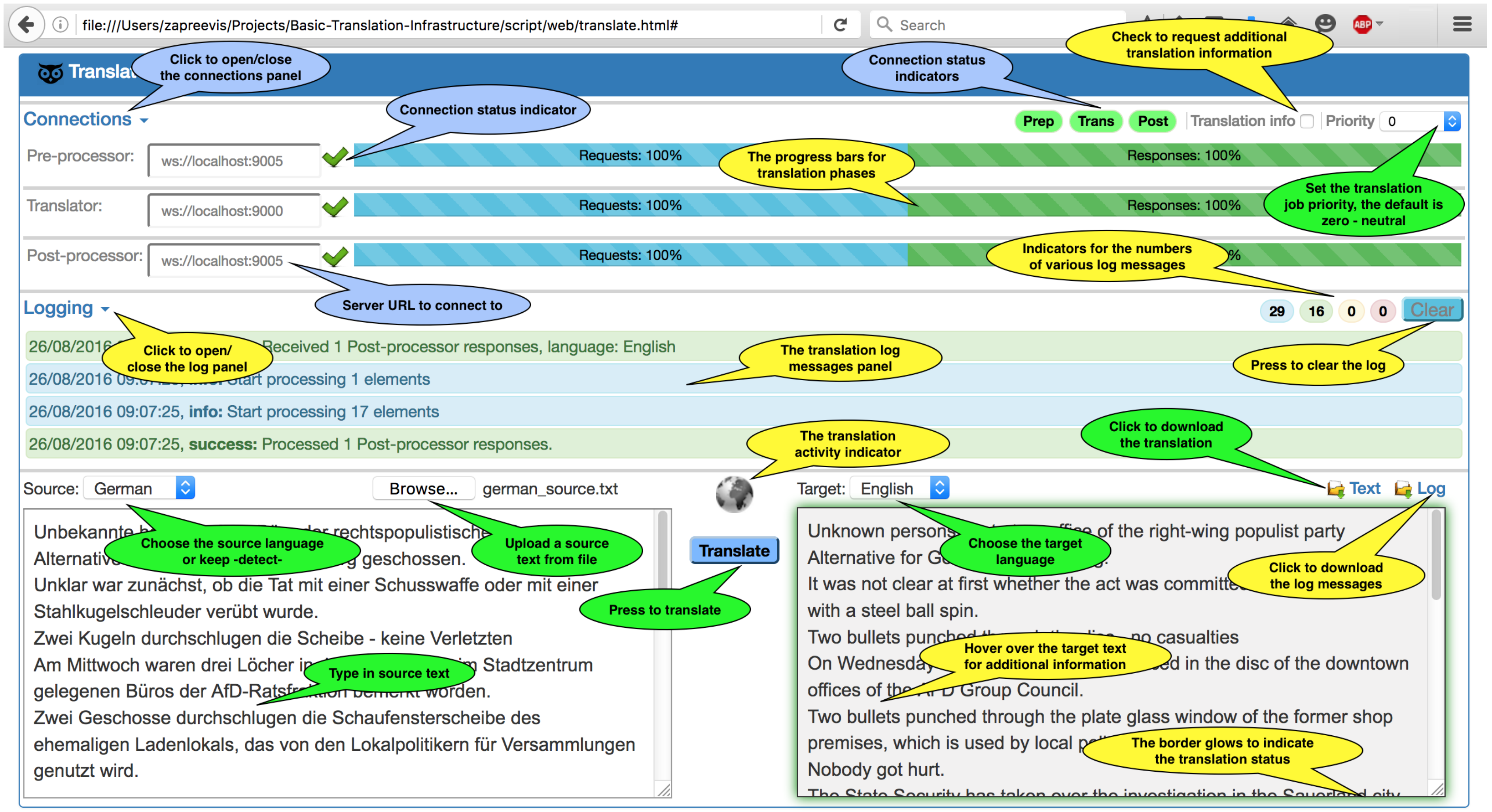


Figure 1: Web Client Translation System UI


file:///Users/zapreevis/Projects/Basic-Translation-Infrastructure/script/web/translate.html#
Search
Translat
Click to open/close the connections panel
Connection status indicators
Check to request additional translation information
Connections
Connection status indicator
Prep
Trans
Post
Translation info
Priority
Set the translation job priority, the default is zero - neutral
The progress bars for translation phases
Indicators for the numbers of various log messages
Server URL to connect to
Logging
Click to open/ close the log panel
The translation log messages panel
Clear
Press to clear the log
The translation activity indicator
Click to download the translation
Source:
Browse...
Choose the source language or keep -detect-
Upload a source text from file
Translate
Press to translate
Type in source text
Target:
Choose the target language
Text
Log
Click to download the log messages
Hover over the target text for additional information
The border glows to indicate the translation status

Pre-processor:	ws://localhost:9005	Requests: 100%	Responses: 100%
Translator:	ws://localhost:9000	Requests: 100%	Responses: 100%
Post-processor:	ws://localhost:9005	Requests: 100%	%

29 16 0 0

```
26/08/2016 Received 1 Post-processor responses, language: English
26/08/2016 09: Start processing 1 elements
26/08/2016 09:07:25, info: Start processing 17 elements
26/08/2016 09:07:25, success: Processed 1 Post-processor responses.
```

Source: German — german_source.txt

Unbekannte ... der rechtspopulistische
Alternativ ... g geschossen.
Unklar war zunächst, ob die Tat mit einer Schusswaffe oder mit einer
Stahlkugelschleuder verübt wurde.
Zwei Kugeln durchschlugen die Scheibe - keine Verletzten
Am Mittwoch waren drei Löcher i ... im Stadtzentrum
gelegenen Büros der AfD-Ratsfr... worden.
Zwei Geschosse durchschlugen die Schaufensterscheibe des
ehemaligen Ladenlokals, das von den Lokalpolitikern für Versammlungen
genutzt wird.

Target: English

Unknown persons ... ffice of the right-wing populist party
Alternative for G... g.
It was not clear at first whether the act was committe...
with a steel ball spin.
Two bullets punched ... no casualties
On Wednesda... d in the disc of the downtown
offices of the ... D Group Council.
Two bullets punched through the plate glass window of the former shop
premises, which is used by local p...
Nobody got hurt.

Figure 2: Annotated Web Client Translation System UI

An example configuration file can be found in: `[Project-Folder]/balancer.cfg`. The content of this file is self explanatory and has a significant amount of comments.

An important thing to note is that, each translation server specified in the configuration file is given a load weight value. It is a positive integer indicating the work capacity of the server relative to the other ones. If there are three translation servers for the same source-target language pair specified, accordingly, in the config file with the load weights, e.g., 1, 2, and 7, then 10% of translation requests will be sent to the first server, 20% to the second one and 70% to the last one. In other words for the same source-target language pair, the normalized weights define the percent of translation requests that will be sent to this or that translation server. If a translation server goes offline then the weights of the remaining servers will get re-normalized. Also the load balancer will monitor the missing server availability by trying to periodically connect to it. If the connection is re-established then the things will automatically get back to normal.

When run with a properly formed configuration file, **bpbd-balancer** gives the following output. Note the `-d info3` option ensuring additional information output during starting up and connecting to translation servers.

```
$ bpbd-balancer -d info3 -c ../balancer.cfg
<...>
USAGE: The requested debug level is: 'INFO3', the maximum
       build level is 'INFO3' the set level is 'INFO3'
USAGE: Loading the server configuration option from:
         ../balancer.cfg
INFO: The configuration file has been parsed!
INFO: Balancer parameters: {server_port = 9000,
      num_req_threads = 10, num_resp_threads = 10,
      translation servers:
      [{SERVER_NAME_01, ws://localhost:9001, load weight=1},
       {SERVER_NAME_02, ws://localhost:9002, load weight=2},
       {SERVER_NAME_03, ws://localhost:9003, load weight=1},
       {SERVER_NAME_04, ws://localhost:9004, load weight=1}, ]}
INFO3: Sanity checks are: OFF !
INFO3: Configuring the translation servers' manager
INFO3: Configuring 'SERVER_NAME_01' adapter...
INFO2: 'SERVER_NAME_01' adapter is configured
INFO3: Configuring 'SERVER_NAME_02' adapter...
INFO2: 'SERVER_NAME_02' adapter is configured
INFO3: Configuring 'SERVER_NAME_03' adapter...
INFO2: 'SERVER_NAME_03' adapter is configured
INFO3: Configuring 'SERVER_NAME_04' adapter...
INFO2: 'SERVER_NAME_04' adapter is configured
INFO2: The translation servers are configured
USAGE: Running the balancer server ...
USAGE: The balancer is started!
```

```
USAGE: ---------------------------------------------------------
<...>
```

For a less informative output one can reduce the log level or simply run:

`bpbd-balancer -c ../balancer.cfg`.

Balancer console: Once the balancer is started it is not run as a Linux daemon but a simple multi-threaded application that has its own interactive console allowing to manage some of the configuration file parameters and obtain some run-time information about the load balancer. The list of available console commands is given in the listing below:

```
$ bpbd-balancer -d info3 -c ../balancer.cfg
<...>
USAGE: The balancer is started!
USAGE: ---------------------------------------------------------
USAGE: General console commands:
USAGE:  'q & <enter>'  - to exit.
USAGE:  'h & <enter>'  - print HELP info.
USAGE:  'r & <enter>'  - run-time statistics.
USAGE:  'p & <enter>'  - print program parameters.
USAGE:  'set ll  <level> & <enter>'  - set log level.
USAGE: Specific console commands:
USAGE:  'set int  <positive integer> & <enter>'  - set the
                               number of incoming pool threads.
USAGE:  'set ont  <positive integer> & <enter>'  - set the
                               number of outgoing pool threads.
>>
```

3.2.5 Text Processor: bpbd-processor

The text processor server is meant to be used for the following purposes:

1. Pre-process text for translation;
2. Detect the source language;
3. Post-process text after translation;

The text processor server is a multi-threaded application supporting translation job priorities. Note that currently the text processor does not have any load balancing capability. The reason for that is that text processing is expected to be significantly less computation intensive, when compared to the text translation task itself. Also, text pre/post-processing is an important, but optional step in the translation process.

The use of the **bpbd-processor** executable is straightforward. When started from a command line with no arguments, *bpbd-processor* reports on the available command-line options:

```
$ bpbd-processor
<...>
PARSE ERROR:
            Required argument missing: config

Brief USAGE:
  bpbd-processor  [-d <error|warn|usage|result|info|
                       info1|info2|info3>] -c
                  <processor configuration file>
                  [--] [--version] [-h]

For complete USAGE and HELP type:
  bpbd-processor --help
```

As one can see the only required command-line parameter of the server is a configuration file. The latter shall contain the necessary information for running the processor server. Once the processor server is started there is still a way to change some of its run-time parameters. The text processor configuration file and the server console are discussed below.

Processor config file: In order to start the processor server one must have a valid configuration file for it. The latter stores the minimum set of parameter values needed to run the server. Among other things, this config file specifies the pre/post-processor scripts to be used. An example configuration file is:

`[Project-Folder]/processor.cfg`

The dummy/example versions of the pre/post-processor scripts are located in:

`[Project-Folder]/script/text/`

These scripts are:

- `pre_process.sh` - the dummy/example pre-processor script;
- `post_process.sh` - the dummy/example post-processor script;

When setting up a text processor server, we expect third-party language-detection, pre/post-processing scripts to be used. The only requirement to them, or their wrappers, is that they fulfill the same input/output interface as the provided `pre_process.sh`, and `post_process.sh` scripts. Note that the pre/post- processor scripts do not need to be bash scripts. They can be any command-line executable applications that satisfy the scripts' interface. Run the provided dummy/example scripts to get more details on their expected interface and functionality.

Further, the content of the provided example text processor configuration file is self explanatory and contains a significant amount of comments. When run with a properly formed configuration file, **bpbd-processor** gives the following output. Note the `-d info3` option ensuring additional information output during starting up and connecting to translation servers.

```
$ bpbd-processor -d info3 -c ../processor.cfg
<...>
USAGE: The requested debug level is: 'INFO3', the maximum
       build level is 'INFO3' the set level is 'INFO3'
USAGE: Loading the processor configuration option from:
         ../processor.cfg
INFO: The configuration file has been parsed!
USAGE: The lattice file folder is: ./proc_text
WARN: The directory: ./proc_text does not exist, creating!
INFO: Processor parameters: {server_port = 9000,
       num_threads = 20, work_dir = ./proc_text,
       pre_script_conf =
       {call_templ = ../script/text/pre_process.sh
                                 <WORK_DIR> <JOB_UID>
                                 <LANGUAGE>},
       post_script_conf =
       {call_templ = ../script/text/post_process.sh
                                 <WORK_DIR> <JOB_UID>
                                 <LANGUAGE>}}
INFO3: Sanity checks are: OFF !
USAGE: Running the processor server ...
USAGE: The processor is started!
USAGE: ---------------------------------------------------------
<...>
```

For less output reduce the log level or simply run:

```
bpbd-processor -c ../balancer.cfg.
```

Processor console: Once the processor is started it is not run as a Linux daemon but as a simple multi-threaded application that has its own interactive console allowing to manage some of the configuration file parameters and obtain some run-time information about the text processor. The list of available console commands is given in the listing below:

```
$ bpbd-processor -d info3 -c ../balancer.cfg
<...>
USAGE: The processor is started!
USAGE: ---------------------------------------------------------
USAGE: General console commands:
USAGE:  'q & <enter>'  - to exit.
USAGE:  'h & <enter>'  - print HELP info.
USAGE:  'r & <enter>'  - run-time statistics.
USAGE:  'p & <enter>'  - print program parameters.
USAGE:  'set ll  <level> & <enter>'  - set log level.
```

```
USAGE: Specific console commands:
USAGE:  'set nt  <positive integer> & <enter>'  - set the
                                       number of processor threads.
>>
```

3.3 Code Design

In this section we provide some insights into the most critical parts of the new code base. We do that by providing the UML [OMG16] class and sequence diagrams for the developed sources. Our purpose is to help understand the software design, and the ideas and motivation behind.

First of all, we consider the translation client-server communication protocol which was changed from a plain-text proprietary format to a JSON [JAS06] based one. The main motivation for switching to JSON was a necessity to communicate to the web client application written in JavaScript. In addition, JSON is a well-established text-based format for storing data. Using JSON allows for a flexible, easily extendible, and human readable protocol with a wide variety of existing open source libraries to work with. More information on the messaging system design and implementation will be presented in Section 3.3.1.

Secondly, we discuss the newly added applications: **bpbd-balancer**, and **bpbd-processor**. Both of them follow a very similar internal design pattern. Therefore for the sake of brevity in Section 3.3.2, we will present the design of the most intricate applications of these two: *bpbd-balancer*.

3.3.1 Translation server messaging

All communications between the translation clients, text processor, translation server, and load balancer applications are based on an internal protocol using JSON data format over WebSockets. The advantages of this approach were mentioned earlier. In this section, c.f. Figure 3, we provide a class diagram for a part of this protocol related to the translation job and supported language request and response messages. Remember that for any translation job we have:

- **request** - contains a number of sentences in a source language to be translated;
- **response** - contains a number of translated sentences in a target language;

For communication over translation/balancer server's supported language pairs we have:

- **request** - asks the server for a list of supported source-target language pairs;
- **response** - contains the list of supported source-target language pairs;

As one can see from Figure 3 we use multiple inheritance of C++ to implement the messaging classes. First of all we distinguish between incoming (*incoming_msg*) and outgoing (*outgoing_msg*) messages, which both have the same base class *msg_base*, storing the main protocol constants and values common to both of them. Another part of the class tree arises from distinguishing between the request (*request_msg*) and response (*response_msg*) message types.

Further, let us consider two types of messages: *supported language*, and *translation job*. Each of these has associated request and response classes. For example, *supp_lang_req* is the class storing data and functionality common for all supported language request classes. On the other hand, *trans_job_resp* is the class that stores data and functionality common for all translation job responses.

Clearly, a request sent by the client application is to be received by the server. Similarly, each response sent by the server is to be received by the client. Therefore each request and response can be of two types: an incoming and an outgoing message. This is where the multiple inheritance comes into place. For example *trans_job_req_out* is the concrete class that is used on the client to represent the outgoing translation job request, so this class inherits from *trans_job_req* and *outgoing_msg*. The same request on the server will be represented by the *trans_job_req_in* class that also inherits from *trans_job_req* but its another base class is the incoming message class *incoming_msg*.

In addition it is important to note that the classes considered in Figure 3 are indeed used for communicating between the translation clients and the translation severs or load balancers. To communicate with the text processor server another set of message classes is to be used. These classes also inherit from the same base classes, such as: *request_msg*, *response_msg*, *incoming_msg*, *outgoing_msg*, and *msg_base* and follows the same design patterns as discussed above. We omit these additional message classes from the diagram for the sake of brevity.

3.3.2 Load balancer design

The internal **bpbd-balancer** application design is depicted in Figure 4. The purpose of this figure is to introduce the implementation details of the load balancer with its main entities and explain their roles. Further, we will consider the main classes of Figure 4 one by one.

bpbd_balancer: is the entry point of the balancer application. Its main responsibilities are to read the configuration file and to set up, configure, and start the other two main entities: *balancer_server* and *balancer_console*.

balancer_console: is responsible for implementing the interactive shell between the command line and the balancer server. It allows to e.g. change the number of threads serving the internal queues, report on the run-time server information, stop the server, and etc.

balancer_server: is a WebSocket server class that is responsible for receiving the supported language and translation job requests from the client applications and sending corresponding responses. When instantiated, this class brings to life and keeps reference to two other important entities: *adapters_manager*, and *balancer_manager*. The former is used to obtain the list of supported source-target language pairs[7] and manage the connections to the translation servers. The latter is needed to keep track of open client sessions and to dispatch the translation job requests into.

adapters_manager: has the following main purposes: *(i)* to keep information of the source-target language pairs supported by the online translation servers; *(ii)* to instantiate and manage

[7]These are needed to reply the supported language requests

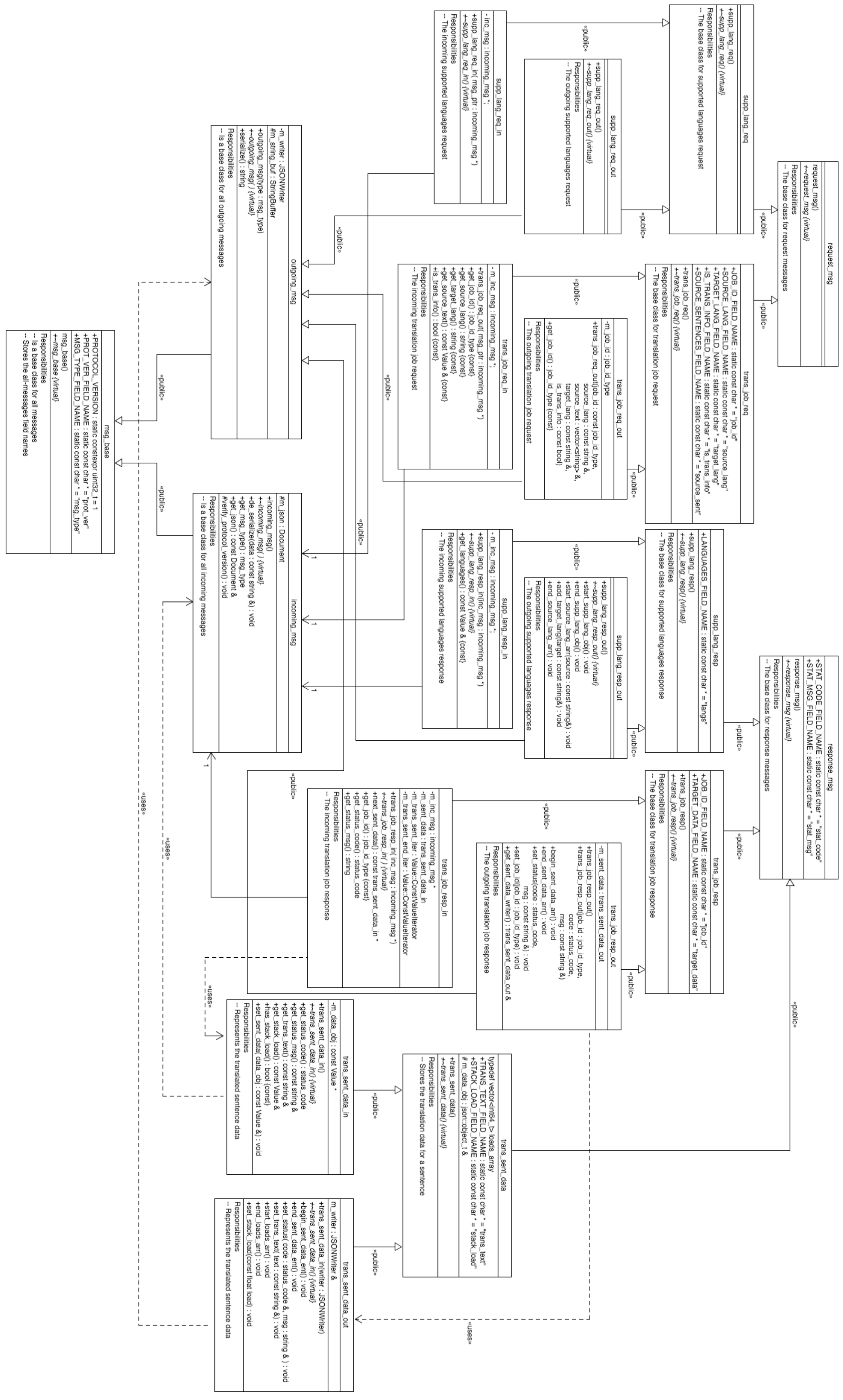


Figure 3: Messaging classes

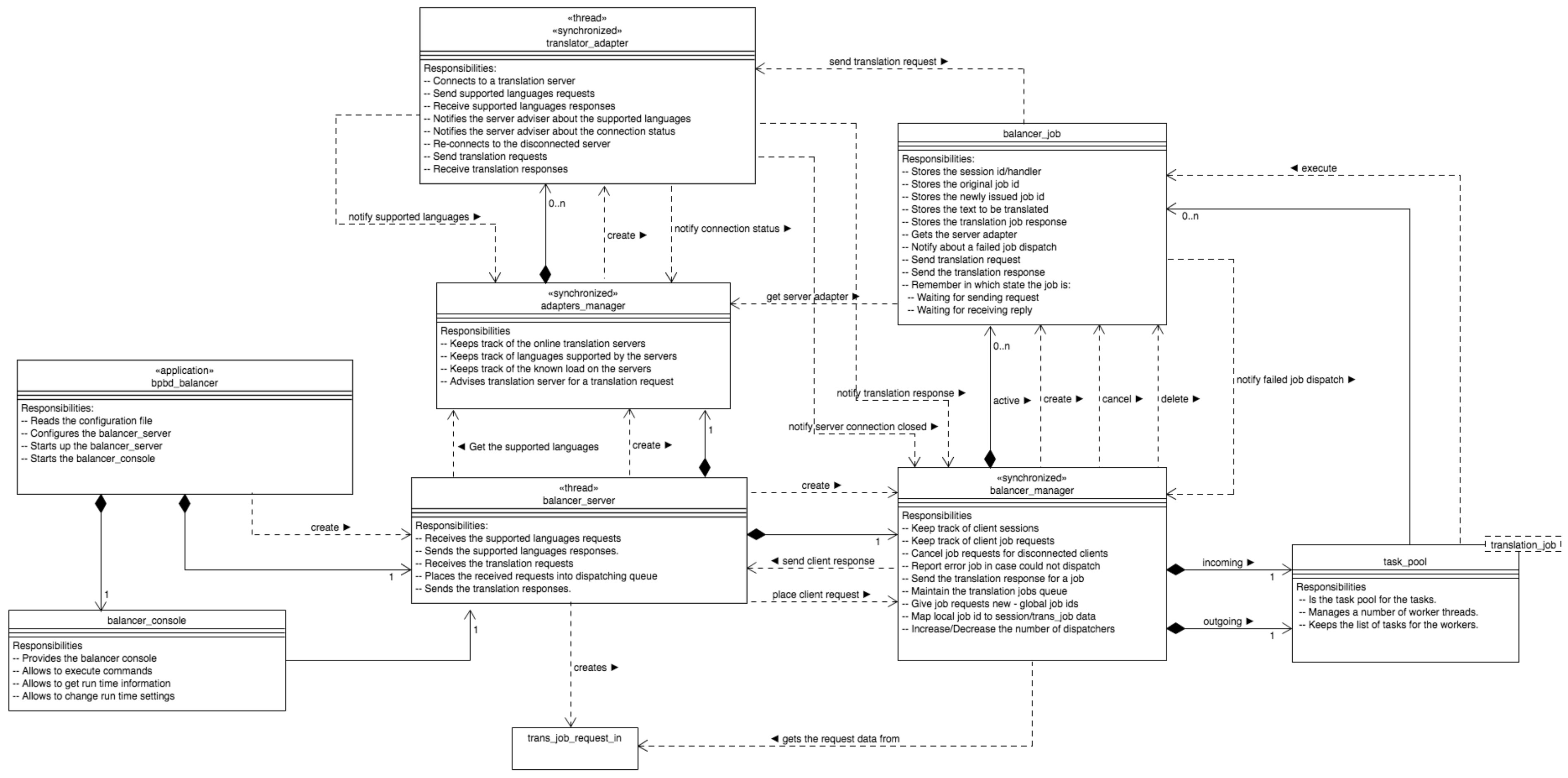

«thread»
«synchronized»
translator_adapter
Responsibilities:
-- Connects to a translation server
-- Send supported languages requests
-- Receive supported languages responses
-- Notifies the server adviser about the supported languages
-- Notifies the server adviser about the connection status
-- Re-connects to the disconnected server
-- Send translation requests
-- Receive translation responses
send translation request ▶
balancer_job
Responsibilities:
-- Stores the session id/handler
-- Stores the original job id
-- Stores the newly issued job id
-- Stores the text to be translated
-- Stores the translation job response
-- Gets the server adapter
-- Notify about a failed job dispatch
-- Send translation request
-- Send the translation response
-- Remember in which state the job is:
-- Waiting for sending request
-- Waiting for receiving reply
◀ execute
0..n
notify supported languages ▶
0..n
create ▶
notify connection status ▶
«synchronized»
adapters_manager
Responsibilities
-- Keeps track of the online translation servers
-- Keeps track of languages supported by the servers
-- Keeps track of the known load on the servers
-- Advises translation server for a translation request
get server adapter ▶
0..n
notify translation response ▶
active ▶
create ▶
cancel ▶
delete ▶
notify failed job dispatch ▶
notify server connection closed ▶
«application»
bpbd_balancer
Responsibilities:
-- Reads the configuration file
-- Configures the balancer_server
-- Starts up the balancer_server
-- Starts the balancer_console
◀ Get the supported languages
create ▶
1
«thread»
balancer_server
Responsibilities:
-- Receives the supported languages requests
-- Sends the supported languages responses.
-- Receives the translation requests
-- Places the received requests into dispatching queue
-- Sends the translation responses.
create ▶
create ▶
1
1
◀ send client response
place client request ▶
«synchronized»
balancer_manager
Responsibilities
-- Keep track of client sessions
-- Keep track of client job requests
-- Cancel job requests for disconnected clients
-- Report error job in case could not dispatch
-- Send the translation response for a job
-- Maintain the translation jobs queue
-- Give job requests new - global job ids
-- Map local job id to session/trans_job data
-- Increase/Decrease the number of dispatchers
incoming ▶
1
outgoing ▶
1
task_pool
Responsibilities
-- Is the task pool for the tasks.
-- Manages a number of worker threads.
-- Keeps the list of tasks for the workers.
translation_job
1
balancer_console
Responsibilities
-- Provides the balancer console
-- Allows to execute commands
-- Allows to get run time information
-- Allows to change run time settings
1
creates ▶
trans_job_request_in
◀ gets the request data from

Figure 4: Load Balancer: The internal design

the translation server adapter class objects; *(iii)* issue adapter objects to handle concrete translation job requests and responses. In other words, for each translation server from a configuration file the manager instantiates and configures a translation server adapter object. The online translation servers are monitored and are requested to provide information on their supported source-target language pairs. This information, including the load weight information, is stored in the *adapters_manager* and is used to issue translation adapters for concrete translation jobs. So this class implements the load balancing strategy by assigning a translation adapter class instance to each translation job request. For each source-target language pair, this is done by following a random distribution defined by the corresponding translation server load weights.

translator_adapter: is a WebSocket client class responsible for managing the connection to a concrete translation server.[8] This class is used to send (forward) the translation requests to translation servers and receive the translation responses.

balancer_manager: keeps track of client sessions and translation job requests/responses. Contains two types of task pools: *one* for the incoming translation job requests - to be forwarded to appropriate translation servers; *another* for the outgoing translation job responses - to be forwarded to the appropriate translation clients. Clearly, the load balancer can be seen as a single client for the translation servers behind it. Remember that each client must issue unique translation job ids to the jobs it is sending to the translation infrastructure. Due to that requirement, the translation job ids issued by the translation clients, and send through the load balancer, are to be re-assigned to become unique. This is also one of the many responsibilities of the *balancer_manager* class.

balancer_job: is the class representing a balancer job. Its main purpose is to encapsulate the translation job request and eventually its corresponding response and track and monitor the job's internal state.

task_pool: is a generic template class used through out the implementation to realize a multi-threaded prioritized task pool. Each task-pool instance has one or more worker threads associated with it and stores a priority queue of tasks which are to be consequently executed by these workers. In case of the load balancer, this template is instantiated with the *balancer_job* as the template parameter value.

Execution examples: In order to illustrate some of the execution scenarios involving the entities described above, we provide Figure 5 storing combined sequence diagram(s) thereof. These diagrams are self-explanatory and are provides solely for the concept-illustration purposes.

[8]It can indeed be another instance of the load balancer application as well.

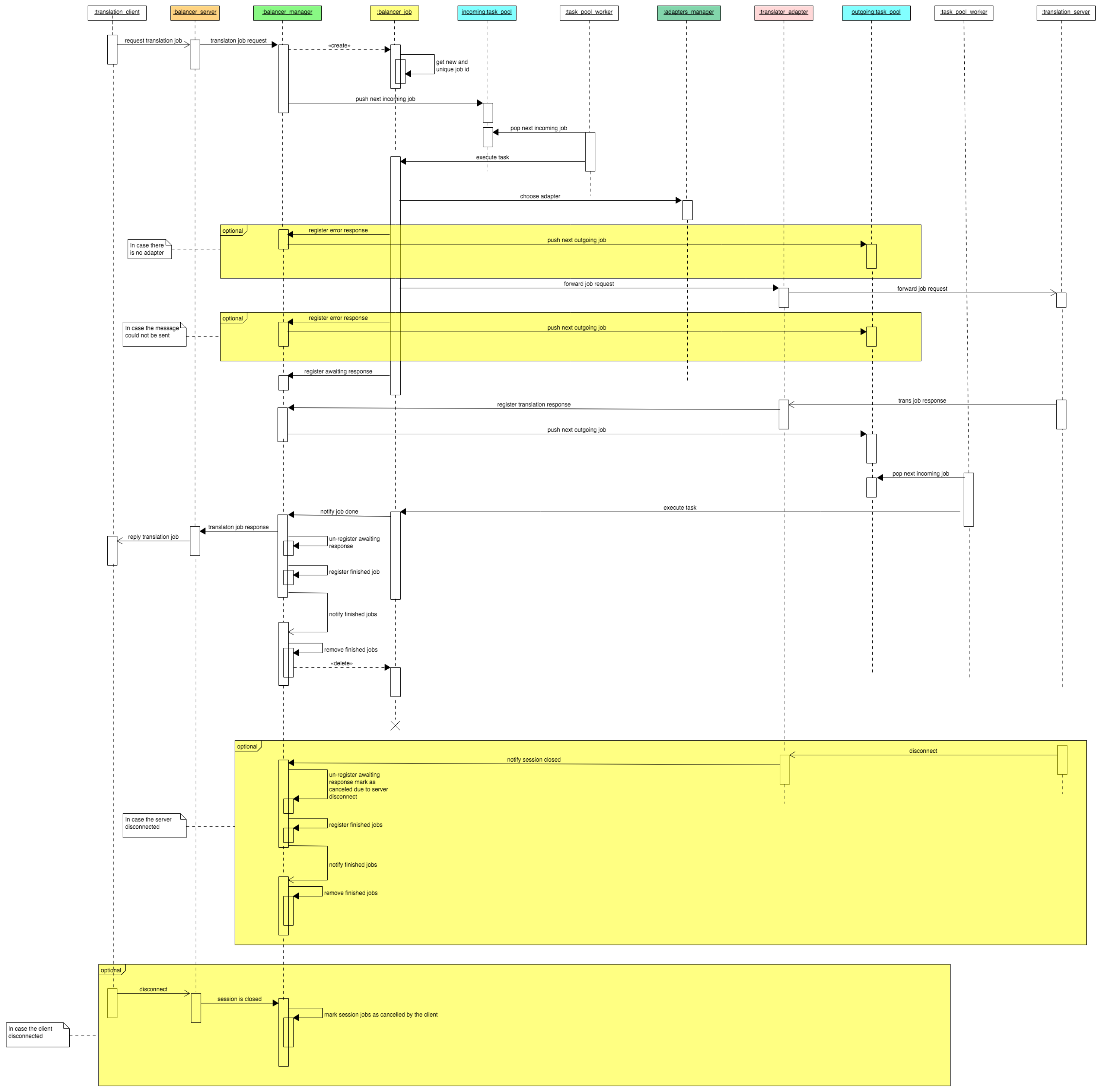


Figure 5: Load Balancer: The main sequences

3.4 System deployment

In this section we are discussing the possible system deployment configurations. Recall that in Section 3.6.1 of [ISZ16a] we presented the ultimate system design given in Figure 6. In that design we assumed that each model could have its own server accessible from and shared by multiple remote decoder servers hidden behind a load balancer.

Such an approach promises gains due to high data sharing and task isolation capabilities. Yet it is potentially prone to network latency issues: when times required to execute the model queries are becoming much smaller than the times needed for network communications to the models. To figure out if the latter is a real risk the system prototype would have to be implemented and stress tested on large-scare real-life examples. Unfortunately, in this project we do not have sufficient resources for that. Therefore, it has been decided to eliminate, or at least significantly reduce, the network latency risks by keeping the models together with the decoder. Taking these considerations into account, the current decode design is shown in Figure 7.

In the sense of the system's distributive power, our approach to implementing the decoder server is not limiting at all. Consider Figure 9 showing an example system deployment configuration. As one can see, there can be multiple servers for the same source-target language pairs, and the load balancer server can distribute the load between them in the desired way. Also a load balancer can have other load balancers behind and/or can aggregate translation servers for various source/target language pairs. All of these can be placed on different physical servers connected by a network infrastructure.

Up until now, the shown deployments did not employ text processing server(s). The reason for that is that this task, although being an important part of the translation chain, is considered to be optional. Yet, to present the overall picture, Figure 9 shows some of the possible system deployments with text processing servers. Note that any pre/post-processing server in the figure might support multiple languages and even language detection for pre-processing. Also, the decoders can be substituted with load balancer instances, spreading the translation load between multiple decoders. Further, we shall briefly describe the possible translation system deployment configurations given in Figure 9.

Type - 01: Only pre-processing, different physical servers. This is the situation when only pre-processing is enabled. Whether the pre-processing script supports language identification or not will depend on the concrete pre-processing script implementation. If not, and it is requested, then an error is to be reported by the processing server. Since there is no post-processor, the resulting target text will be output by the translation system 'as is', i.e., it will be tokenized and lower-cased, and with just one sentence per line. *This configuration can be recommended for systems with large load from the pre-processing script.*

Type - 02: Only post-processing, different physical servers. This is the situation when only post-processing is enabled. In this case the provided source text is supposed to be tokenized[9],

[9]Words and punctuation marks are to be separated with single ASCII spaces.

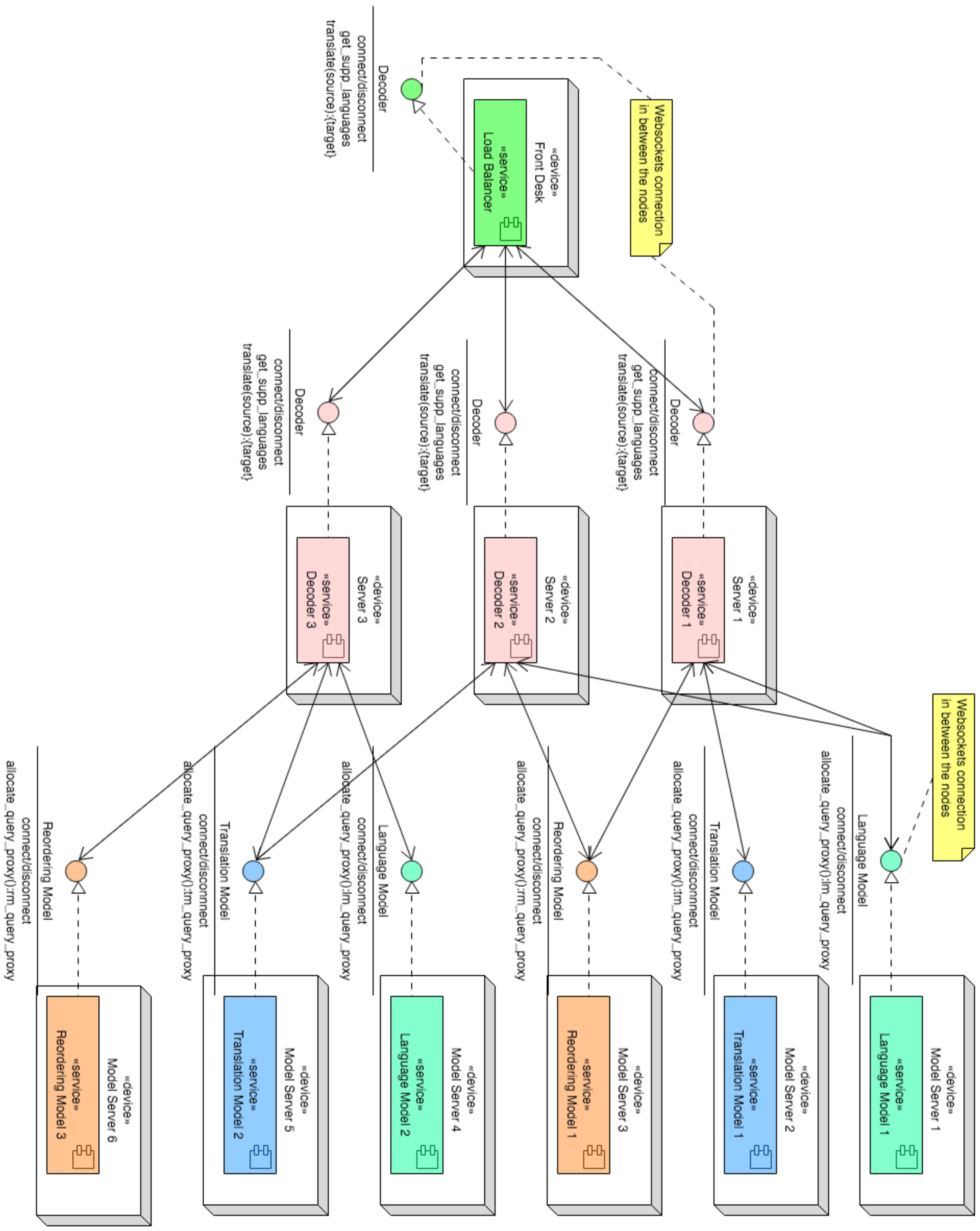


Figure 6: The initially proposed ideal system design

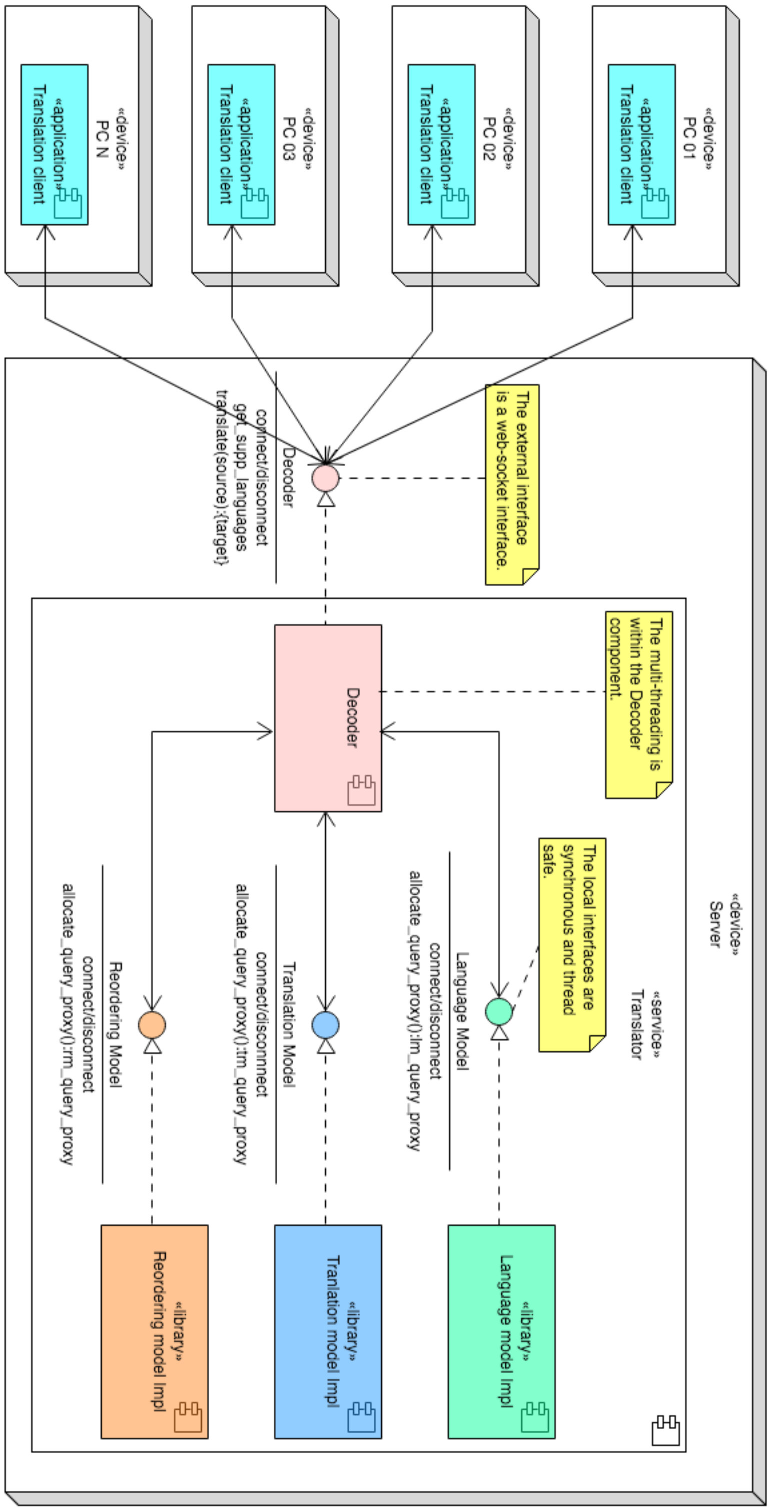


Figure 7: The current decoder server design

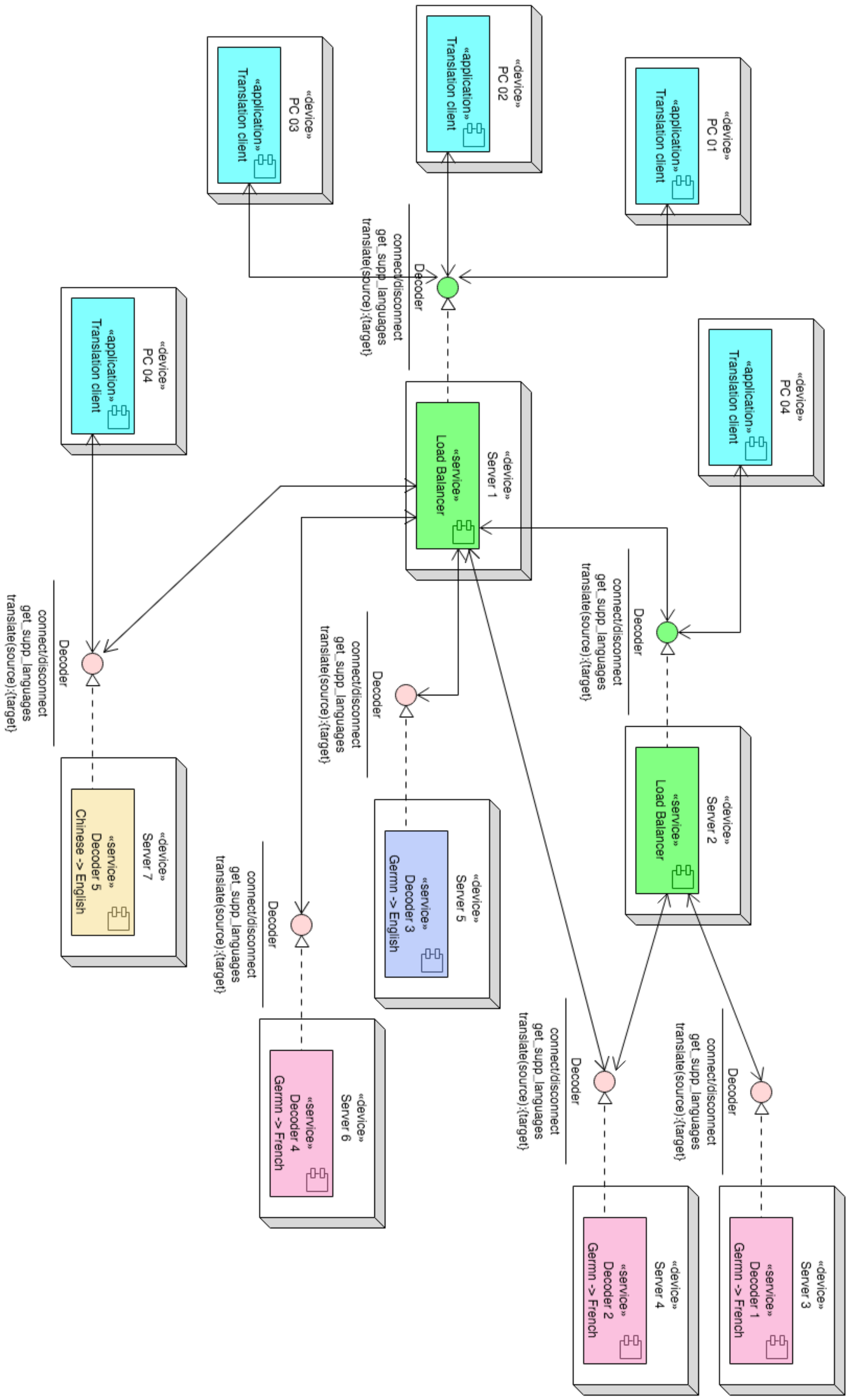


Figure 8: An example system deployment

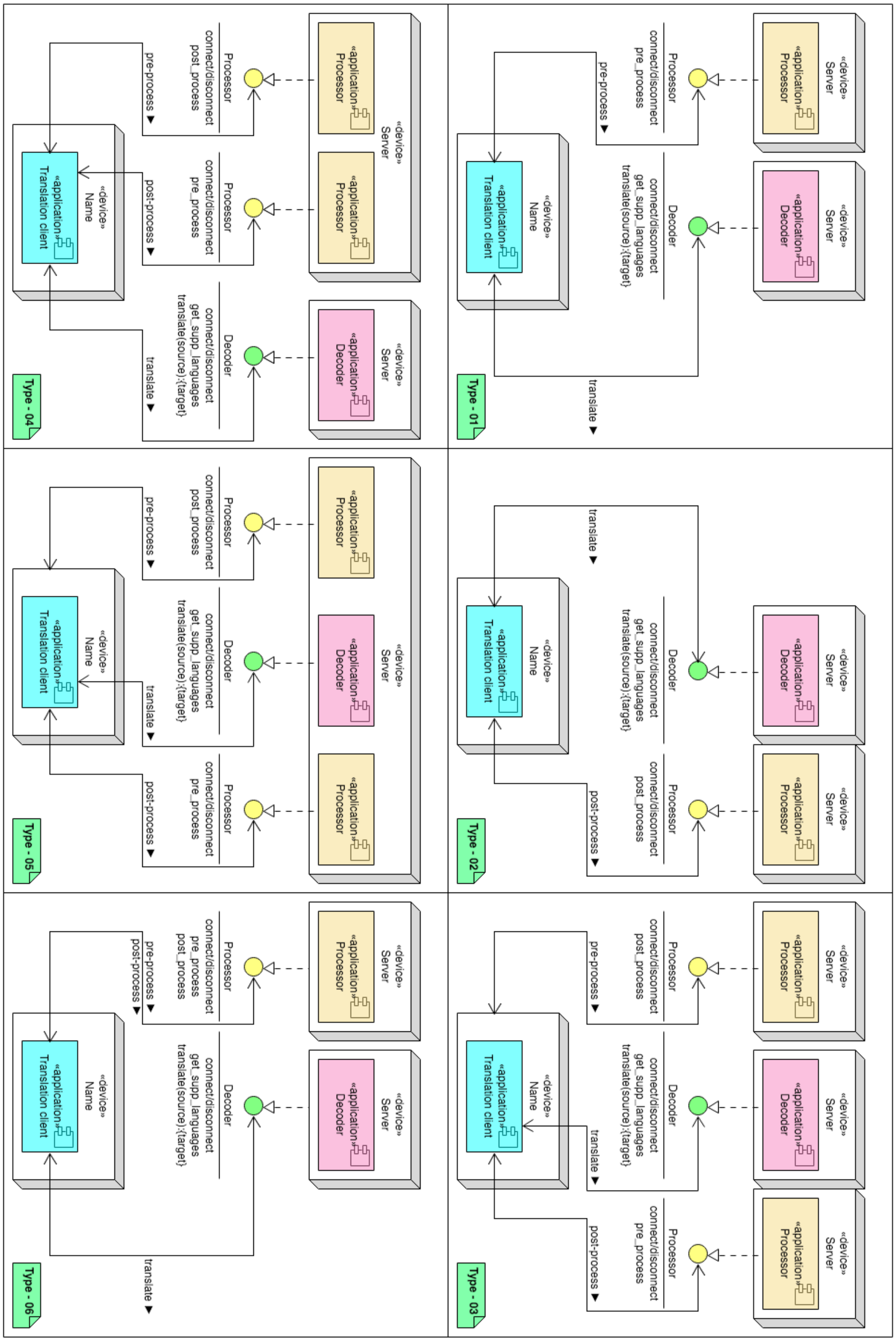


Figure 9: Deployments with text processing servers

lower-cased[10], mapped to unicode[11], and there must be just one sentence per line in the source text file. The resulting target text will de untokenized and upper-cased but one can still expect to get one target sentence per line in the output as there is no source text available to restore the original text structure. *This configuration can be recommended for systems with large load from the post-processing scripts.*

Type - 03: Pre- and post-processing, different physical servers. This is the situation when both pre- and post-processing are enabled. Yet, all of the servers are run on different physical computation nodes. Please note that this renders the situation more complex when the post-processor script needs the source text for restoring the text structure. The complication is due to the fact that, even if the pre-processor script makes and stores a copy of the source text file, it is yet to be communicated to the server doing post-processing. *This configuration can be recommended for systems with large load from the pre/post-processing scripts.*

Type - 04: Pre- and post-processing, one physical server. This is the situation when both pre- and post-processing are enabled and are run together on one physical server. In this case, internal - temporary file sharing between the pre- and post-processing scripts becomes simpler. *This configuration can be recommended for servers with multiple processors and shared hard drives or if the pre- and post- processing have low performance impact on the system. In the latter case, it might be easier to just run the configuration of Type - 06.*

Type - 05: Separate applications on one physical server. This is the situation when both pre- and post-processing are enabled and are run together on one physical server with the translation system. In this case, internal, temporary file sharing between the pre- and post-processing scripts becomes simpler. *This configuration can be recommended for servers with multiple processors and multiple shared hard drives or if the pre- and post- processing have low performance impact on the system or as a test configuration.*

Type - 06: Pre- and post-processing, one application, one physical server. This is the situation when both pre- and post-processing are enabled and are run together within one application on one physical server. In this case, internal, temporary file sharing between the pre- and post-processing scripts becomes simpler. *This configuration can be recommended for servers with multiple processors and multiple shared hard drives or if the pre- and post- processing have low performance impact on the system.*

3.5 Testing and robustness

In this delivery, along with the regular Valgrind [ABBF+00] testing, we have performed large-scale stress testing for the entire system infrastructure. The reason for that was adding two multi-threaded server applications: **bpbd-balancer** and **bpbd-processor** into the ecosystem, and also making some vital code unifications to the **bpbd-server** and **bpbd-client** applications.

[10]All letters must be in lower case.

[11]The longer UTF-8 character sequences are to be substituted with the equivalent but shorter ones.

Currently, our software is highly parallelized: uses multiple threads, which work with shared resources, and has multiple synchronization points. All this can lead to various concurrency issues such as deadlocks, livelocks, starvation problems and etc, see [Tai94] for definitions and more details. So the important software aspects, such as quality and robustness, became easy to compromise and required additional attention.

There are two well-known and commonly accepted ways to ensure the stability and robustness of a system like ours: *(i)* formal system verification; *(ii)* extended system testing. The first approach can be done in many ways, e.g., by building the formal model of the employed multi-threaded algorithms and then verifying them for robustness with dedicated model checking tools, or by using static analysis software to verify the system's source code, and etc. Despite such an approach being highly accurate, we had to neglect it due to our resource and time constraints. The second approach, even though is does not provide 100% accuracy, is much easier and faster to implement and provides sufficient confidence in the software quality.[12] Therefore, our pragmatic choice had to be made for an extended stress testing of the created infrastructure. Such a testing was implemented and performed and allowed to detect and fix several concurrency issues. After that the test runs went smooth and did not reveal any additional issues, giving us a high confidence in a good quality of the delivered software. For the sake of completeness, let us describe the way the stress testing was performed.

3.5.1 Test target and configuration

The purpose of the stress testing we did was to try to break the translation infrastructure software by flooding it with the concurrent translation requests from multiple clients. To make testing more realistic, we distributed our setup over 8 multi-core computation machines: smt3 to smt10. All of the machine run CentOS 6 and have slightly different hardware configurations. Yet, they all posses multiple cores and dozens Gb of RAM, see Appendix A for more details. In our experimental setup we used smt3 to smt6 to run translation clients; smt7 to run text processing server; smt8 to run load balancer; smt9, smt10 to run translation servers. The client applications would connect to the text processing server for text pre/post-processing and the load balancer for text translation. Let us consider each of the applications' configuration in more details.

Client applications on smt3 to smt6: Each machine would run 400 instances of the **bpbd-client** application in parallel. Each of the client applications would take a $3,000$ sentences long German text and send it first to pre-processing then to translation and then to post-processing. All of the *bpbd-client* applications on a machine were started with the delivered test script:

```
./script/test/stress_test.sh
```

which has a self explanatory synopsis, to be seen when started from the command line with no arguments:

```
$stress_test.sh
ERROR in stress_test.sh: Improper number of arguments!
```

[12]As it allows to stress test the main system use cases insuring their robust implementation.

```
------
SHORT:
------
  This is a stress testing script for the translation
  system. It's purpose is to run multiple translation
  clients in parallel.
------
USAGE:
------
stress_test.sh <num-processes> <trans-uri> <proc-uri>
                <source-file> <source-lang> <target-lang>
    <num-proc> - the number of bpbd-clients to run in
                 parallel
    <trans-uri> - the URL of the translation server,
                  with the port
    <proc-uri> - the URL of the text processor server,
                 with the port
    <source-file> - the source text file
    <source-lang> - the source language or auto
    <target-lang> - the target language
------
PURPOSE:
------
  The main purpose of the script is to stress load the
  infrastructure  and see if any errors or deadlocks
  or alike are occurring.
```

This scripts first starts a control run, and logs its results, then it starts the requested number of client applications one by one and waits until all of them finish their execution. Next it compares the outputs of these clients with the control run, trying to identify any differences in translation results, such as reported errors. Since we used dummy pre/post-processing scripts our source German text was lower-cased, and tokenized, with individual sentences put on separate lines. Finally it is important to note that, 4 machines running 400 translation client each, where every client needs to translate $3,000$ sentences, means $4 * 400 * 3,000 = 4,800,000$ translation tasks. All of them had to be handled by the load balancer located on smt8 and the two translation servers running on smt9 and smt10. This is quite significant, also considering that the number of clients using the infrastructure at the same time is then equal to $4 * 400 = 1.600$. Waiting for all of the translation jobs to finish took from 36 to 48 hours[13].

Text processing server on smt7: The text pre/post-processing was done by a single instance of the **bpbd-processor** application, using 20 worker threads to manage pre- and post-processing requests. The reason we did not use two independent instances on two different machines is that

[13]The exact numbers are not important as we did not have a goal of performance testing.

doing both pre- and post-processing within the same application increases its load. Also note that we used the provided dummy scripts for pre/post-processing which merely copy files and do not do actual language detection. This however is not a limitation, as having actual scripts means longer execution times for processor jobs. The latter only decreases the concurrency within the *bpbd-processor* application, resulting in less stress.

Load balancing server on smt8: The load balancing server was configured to use 10 worker threads for the incoming translation requests pool and 10 worker threads for the outgoing translation responses pool. It was also configured to evenly distribute the work load between the two German $\rightarrow$ English translation servers located on smt9 and smt10. The latter was achieved by setting the both servers' `load_weight` parameter values to 1. Other parameters, such as `reconnect_time_out` are not important, as they had no influence on the server performance.

Translation servers on smt9 and smt10: Both translation servers were configured to translate from German into English. We used some minimum size models for these experiments, as the smaller the model the faster the translation task. The latter ensures tighter timing and thus more stress in concurrency aspects of the system. Yet, the system was tuned in such a way that it would give 15 BLEU points [PRWZ02] on the German text that had to be translated. This is sufficient to say that the translation system does perform some low-quality translation and is not completely dummy. The last thing to mention about the translation servers is that each of them was configured to use 40 worker threads to perform parallel translations.

Results. The first few runs of the experimental set-up described above revealed several concurrency issues in the software which were successfully fixed and thus eliminated. The subsequent runs did not reveal any new issues. This does not guarantee that the system is completely bug free, yet it gives a high level of confidence, considering that the number of parallel clients was $1,600$ and the number of translation tasks was $4,800,000$. This is far beyond what the system is meant to be capable of. After all, this system is developed for being used in the research environment as an experimental platform and for smaller companies with up to 200 simultaneously active translations clients.

4 Conclusions

In this report we discussed the content of the month 18 REMEDI project delivery. As required and specified in the REMEDI project proposal [Mon14] this delivery contains:

1. **Implementation of search lattice data structure and feature passing (S1, WP1)** - see Section 3.2.1;
2. **Implementation of server front end (S1, WP1)** - see Section 3.2.3;
3. **Implementation of robustness preserving load balancing and restarting (S1, W1)** - see Section 3.2.4;
4. **Complete software deliverable of decoder infrastructure (S1, W1)** - see Sections 3.2.1 and 3.2.5;

5. **Month 18 report (R3)** - this document;

All of these bullets were discussed in more detail in Section 3.1. It remains to add that in addition to the required part, as usual, we have extended our delivery with supplementary but highly useful items, such as: *(i)* global support for translation job priorities, c.f. Sections 3.2.2 and 3.2.3; *(ii)* software design diagrams, c.f. Section 3.3; *(iii)* software deployment diagrams, and advises on useful configurations, c.f. Section 3.4; *(iv)* Extended validation and testing, $1,600$ simultaneous clients stress-loaded the software with almost $5,000,000$ sentences to translate, c.f. Section 3.5; *(v)* extended and up to date infrastructure documentation, c.f. [ISZ16b];

In the next 6 month period of the project, we shall concentrate on the 24th month REMEDI delivery. The requirements thereof are described in the “Bi-Annual Deliverable-Payment Chart” table on page 5 of the REMEDI proposal [Mon14].

References

[ABBF$^+$00] Cerion Armour-Brown, Christian Borntraeger, Jeremy Fitzhardinge, Tom Hughes, Petar Jovanovic, Dejan Jevtic, Florian Krohm, Carl Love, Maynard Johnson, Paul Mackerras, Dirk Mueller, Nicholas Nethercote, Julian Seward, Bart Van Assche, Robert Walsh, Philippe Waroquiers, and Josef Weidendorfer. Valgrind, 2000. http://valgrind.org/.

[Boo16] Bootstrap. https://bootstrapdocs.com/v3.3.6/docs/, 2016. Accessed: 2016-09-02.

[CSS16] Cascading style sheets. https://www.w3.org/Style/CSS/Overview.en.html, 2016. Accessed: 2016-09-02.

[Gru15] John Gruber. Markdown: A plain text formatting syntax. https://daringfireball.net/projects/markdown/, 2015.

[HTM14] Html5 reference. https://www.w3.org/TR/html5/, 2014. Accessed: 2016-09-02.

[ISZ16a] Christof Monz Ivan S. Zapreev. 12 month project report, 2016.

[ISZ16b] Christof Monz Ivan S. Zapreev. The software documentation, 2016.

[JAS06] Javascript object notation. https://www.ietf.org/rfc/rfc4627.txt, 2006. Accessed: 2016-09-06.

[Jav15] Javascript reference. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference, 2015. Accessed: 2016-09-02.

[jQu16] jQuery. http://learn.jquery.com/, 2016. Accessed: 2016-09-02.

[Koe10] Philipp Koehn. *Statistical Machine Translation*. Cambridge University Press, New York, NY, USA, 1st edition, 2010.

[Mon14] Christof Monz. Robust and efficient machine translation in a distributed infrastructure, 2014.

[OMG16] Inc. Object Management Group. Unified modeling language, 2016. http://www.uml.org/.

[PRWZ02] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: A method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, ACL '02, pages 311–318, Stroudsburg, PA, USA, 2002. Association for Computational Linguistics.

[Tai94] Kuo-Chung Tai. Definitions and detection of deadlock, livelock, and starvation in concurrent programs. In *Proceedings of the 1994 International Conference on Parallel Processing - Volume 02*, ICPP '94, pages 69–72, Washington, DC, USA, 1994. IEEE Computer Society.

[Web11] The websocket protocol. https://www.rfc-editor.org/info/rfc6455, 2011. Accessed: 2016-09-02.

Appendices

A Test machine configurations

smt3:

```
[izapree1@smt3 ~]$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                48
On-line CPU(s) list:   0-47
Thread(s) per core:    1
Core(s) per socket:    12
Socket(s):             4
NUMA node(s):          8
Vendor ID:             AuthenticAMD
CPU family:            16
Model:                 9
Model name:            AMD Opteron(tm) Processor 6174
Stepping:              1
CPU MHz:               2199.944
BogoMIPS:              4400.10
Virtualization:        AMD-V
L1d cache:             64K
L1i cache:             64K
L2 cache:              512K
L3 cache:              5118K
...
```

smt4, smt5, smt6:

```
[izapree1@smt4 ~]$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                64
On-line CPU(s) list:   0-63
Thread(s) per core:    2
```

```
Core(s) per socket:    8
Socket(s):             4
NUMA node(s):          8
Vendor ID:             AuthenticAMD
CPU family:            21
Model:                 1
Model name:            AMD Opteron(TM) Processor 6276
Stepping:              2
CPU MHz:               2300.006
BogoMIPS:              4599.73
Virtualization:        AMD-V
L1d cache:             16K
L1i cache:             64K
L2 cache:              2048K
L3 cache:              6144K
...
```

smt7, smt8, smt9, smt10:

```
[izapree1@smt7 ~]$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                40
On-line CPU(s) list:   0-39
Thread(s) per core:    2
Core(s) per socket:    10
Socket(s):             2
NUMA node(s):          2
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 62
Model name:            Intel(R) Xeon(R) CPU E5-2670 v2 @ 2.50GHz
Stepping:              4
CPU MHz:               1200.000
BogoMIPS:              4999.27
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              25600K
...
```