Server to parse documents in Lisa

Paul Huygen <paul.huygen@huygen.nl>

7th March 2013
12:48 h.

Abstract

This document describes and constructs a front-end for a system on supercomputer Lisa that parses documents with a parser like Alpino.

Contents

1 Introduction

Some academic research programs use a large quantity of documents as source-data. Because of the size of the document-set, the documents must be analysed automatically by computer. This computer-analysis involves a work-flow that contains resource-intensive "parsers".

The VU-university is co-owner of a supercomputer, Lisa, that could be used for the resource-intensive process-steps.

This document describes part of a system that transports documents to Lisa in order to have them processed and retrieves the parses. The advantages of this system are: 1) It hides the complexity of the Lisa supercomputer for users; 2) Users on the VU can use the computing power of Lisa without need of a Lisa account and 3) the documents of multiple users can be polled to use Lisa efficiently.

Processing documents with the described system involves the following steps:

1. The user request a unique identifier in order to generate unique filenames.
2. The user constructs a special document. The first line of the document contains a sha-bang (`#!`) followed by the command-line command to invoke the parser (with the arguments, without filenames). The next lines are the lines of the text to be parsed, formatted according to the requirements of the parser.
3. The user constructs a file-name that starts with the identifier, to ensure that the filename is unique.
4. The user sends the document to the intermediate server.
5. The user can request a list of the processing-states (waiting, being processed, ready) of the documents that she sent.
6. The user can download the parses of completed documents.

To enable this, the server has implemented the following four requests:

request	**argument**	**result**	**Description**
`getID`	–	integer	Get identifier to construct unique filenames (section 3.1)
`upload`	file	–	Upload a text (section 3.2)
`filstat`	filename	string	Check whether a file has been processed (section 3.3).
`getparse`	filename	parse	retrieve a parse (section 3.4)

This document implements the following:

- Server
- Python module for users
- demonstration script in Python
- demonstration script in Bash

2 Example scripts

2.1 Python module and demonstration script

The following script is a Python module that facilitates usage of the server in Python programs. The module contains a class `AlpinoLisa` with the following methods:

method	**arguments**	**result**	**Description**
`upload_text`	Alpino command, text, filename	request result	Upload a text for parsing.
`check_status`	filename	string	Check status of an uploaded text.
`retrieve_parse`	filename	parse	Retrieve the parse.

```
"../test/alpinolisa.py" 3a≡
      #!/usr/bin/python
      # alpinolisa -- interface to use VU-NLP service to parse texts with Lisa supercomputer
      ⟨ imports of alpinolisa.py 7f, ... ⟩
      ⟨ http requests in alpinolisa.py 7a, ... ⟩
      class AlpinoLisa():
        """
        have texts parsed with Alpino on Lisa supercomputer
        """

        def __init__(self):
          ⟨ initialise AlpinoLisa class 7e ⟩

        ⟨ methods in AlpinoLisa class 9e, ... ⟩

      ◇
```

To demonstrate how it works, the following python script submits all the files with extension `.txt` in it's own directory and retrieves the parses done by an english-text-to-parsed-KAF utility written by Ruben Izquierdo. The utility is started with the following command:

```
⟨ rubencommand 3b ⟩ ≡
      alpinocommand='/home/phuijgen/nlp/ruben/python/stanford.py'

      ◇
```
Fragment referenced in 3c.

Sorry for the strange nomenclature, mentioning everything "Alpino". This utility was originally developed for Alpino only and now it is extended to use all kind of processors. Furthermore, sorry fir the strange filepath. That will be made more general in the future.

```
"../test/alpinolisademo.py" 3c≡
      #!/usr/bin/python
      # alpinolisademo -- interface to use VU-NLP service to parse texts with Lisa supercomputer
      ⟨ imports of alpinolisademo 3d ⟩
      ⟨ rubencommand 3b ⟩
      ⟨ get list of text files 3e ⟩
      ⟨ open interface with the server 4a ⟩
      ⟨ submit texts 4c ⟩
      ⟨ poll and download 5a ⟩
      ◇
```

```
⟨ imports of alpinolisademo 3d ⟩ ≡
      import time
      ◇
```
Fragment referenced in 3c.

Make a list of the files in this directory with extension `.txt`. The texts in these files will be parsed.

```
⟨ get list of text files 3e ⟩ ≡
      import glob
      infils = glob.glob('*.txt')
      ◇
```
Fragment referenced in 3c.

Create an AlpinoLisa object to communicate with the server.

⟨ *open interface with the server* 4a ⟩ ≡
```
import alpinolisa
lisacon = alpinolisa.AlpinoLisa()
```
◇

Fragment referenced in 3c.
Defines: `Alpinolisa` Never used, `lisacon` 4fg, 6a.

The parse of a text will be stored in a file with the same name as the input file, but extension `.kaf`. This script considered a text as parsed when such a file exists. Therefore, we remove such files if they exist before parsing.

⟨ *remove file with kaf extension* 4b ⟩ ≡
```
import os
outfilnam = @1.replace('.txt', '.kaf')
if os.path.exists(outfilnam):
  os.remove(outfilnam)
```
◇

Fragment referenced in 4c.

Submit the texts:

⟨ *submit texts* 4c ⟩ ≡
```
⟨ status trick 4g ⟩
for infilnam in infils:
    ⟨ remove file with kaf extension (4d infilnam ) 4b ⟩
    ⟨ submit a single text (4e infilnam ) 4f ⟩
```
◇

Fragment referenced in 3c.

⟨ *submit a single text* 4f ⟩ ≡
```
lisacon.upload_text( alpinocommand, @1)
```
◇

Fragment referenced in 4c, 6a.
Defines: `upload_text` 9e.
Uses: `lisacon` 4a.

Note a dirty technical trick. When Lisa receives a single input file and then starts up a job, it will use only one core, even when later on much more files are submitted. When we first ask for a status, start-up of jobs is postponed, giving us time upload more files.

⟨ *status trick* 4g ⟩ ≡
```
status = lisacon.check_status("aap")
```
◇

Fragment referenced in 4c.
Uses: `lisacon` 4a.

When every textfile has been submitted, wait for the parses. Check every minute whether parses have been completed and download them. Present a summary of the nuber of retrieved parses and the number of files waiting on the screen.

Submitted files are in one of the following states:

ready: The parse has been downloaded and stored in a file.

being processed: Alpino is currently busy parsing the text.
waiting: The file is waiting in a queue.
unknown: Somehow the server does not know about the file.

The four variables `readyfils`, `waitingfils`, `lostfils` and `processedfils` keep track of the number of files that are in the four states. The script is ready when the values of `waitingfils`, `lostfils` and `processedfils` are zero.

⟨ *poll and download* 5a ⟩ ≡

```
#
# Poll and download
#
# Download the parses
ready = False
while ( not(ready) ):
 readyfils = 0; waitingfils = 0; lostfils = 0; processedfils = 0
 timeoutfils = 0
 for infilnam in infils:
   ⟨ poll, retrieve if ready and update status (5b infilnam ) 6a ⟩
 ⟨ print a summary 6c ⟩
 ready = (waitingfils + lostfils + processedfils == 0)
 if (not(ready)):
   time.sleep( 60 )

◇
```

Fragment referenced in 3c.

First look whether the parse of a file been retrieved. If this is not the case, ask the server what the status of the file is (method `AlpinoLisa.check_status()`). This method returns one of the strings `ready`, `waiting`, `being processed` or `unknown` to indicate the status of a given file. Retrieve the parse when it is ready (method `AlpinoLisa.retrieve_parse()`). When a file has been lost, resubmit it.

⟨ *poll, retrieve if ready and update status* 6a ⟩ ≡

```
pfil=@1
outfilnam = pfil.replace('.txt', '.kaf')
if os.path.exists(outfilnam):
   readyfils = readyfils + 1
else:
   status = lisacon.check_status(pfil)
   if ( 'ready' in status ):
     outfil = open(outfilnam, 'w')
     outfil.write(lisacon.retrieve_parse(pfil))
     outfil.close()
     readyfils = readyfils + 1
   elif 'unknown' in status:
     ⟨ submit a single text (6b pfil ) 4f ⟩
     lostfils = lostfils + 1
   elif 'waiting' in status:
     waitingfils = waitingfils + 1
   elif 'being processed' in status:
     processedfils = processedfils + 1
   elif 'timeout' in status:
     timeoutfils = timeoutfils + 1
```

◇

Fragment referenced in 5a.
Uses: pfil 6a.

Print a summary after each poll.

⟨ *print a summary* 6c ⟩ ≡

```
print "ready: %4d; processed: %4d; waiting: %4d; lost: %4d; timeout: %4d" % (
         readyfils, processedfils, waitingfils, lostfils, timeoutfils )
```

◇

Fragment referenced in 5a.

2.2 Bash example

The following bash script does the same as the the python example. It uses `curl` to perform the requests to the server.

"testscript" 6d≡

```
#!/bin/bash
⟨ variables of testscript 9a ⟩
⟨ http requests in testscript 7b, . . . ⟩
⟨ get a unique ID for filenames with curl 7d ⟩
echo "ID: " $ID
⟨ upload a file 9c ⟩
⟨ ask for the state of a file 13a ⟩
⟨ download a parse 14f ⟩
```

◇

3 The five server requests

The requests are sent to the url of the server: In the python script:

⟨ *http requests in alpinolisa.py* 7a ⟩ ≡
```
serverurl = 'http://nlp.labs.vu.nl/webservice/index.php'
```
◇

Fragment defined by 7a, 10d, 13e, 17f.
Fragment referenced in 3a.
Defines: `serverurl` 7e, 9e, 10d, 13e, 17f.

In Bash:

⟨ *http requests in testscript* 7b ⟩ ≡
```
SERVERURL=http://nlp.labs.vu.nl/webservice/index.php
```
◇

Fragment defined by 7b, 17g.
Fragment referenced in 6d.
Defines: `SERVERURL` 7d, 8g, 17g.

3.1 The unique identifier

The user prepends filenames of her documents with a unique identifier. In this way she can take care by herself that the documents have unique filenames and cannot be confused with files from other users. The following form constructs a request for a unique identifier.

⟨ *form to request a unique identifier* 7c ⟩ ≡
```
<form action="http://nlp.labs.vu.nl/webservice" method="get">
    <input type="submit" name="getID" value="getID">
</form>
```
◇

Fragment referenced in 17d.

The request is issued with the following HTTP request: `http://nlp.labs.vu.nl/webservice/index.php?getID`.

⟨ *get a unique ID for filenames with curl* 7d ⟩ ≡
```
ID=‘curl -s "$SERVERURL?getID=getID"‘
```
◇

Fragment referenced in 6d.
Uses: `SERVERURL` 7b, 17g.

In Python it goes as follows:

⟨ *initialise AlpinoLisa class* 7e ⟩ ≡
```
IDrequest=serverurl + "?getID"
resp, self.id = httplib2.Http().request(IDrequest)
self.id = self.id.replace("\n", "")
```
◇

Fragment referenced in 3a.
Uses: `httplib2` 7f, `serverurl` 7a, 17f.

We need to import `httplib2` for this:

⟨ *imports of alpinolisa.py* 7f ⟩ ≡
```
import httplib2
```
◇

Fragment defined by 7f, 9f.
Fragment referenced in 3a.
Defines: `httplib2` 7e, 11a, 13f.

The ID is a four-digit number. Assuming that the number of ID's in use in a couple of hours, this should be sufficient. Store the ID in a file.

Note that the HTTP server must be able to rewrite this file.

Processing the request for an ID involves 1) Read the number in the ID-file (set this number to zero if the ID-file does not exist); 2) increment the number in the ID-file modulo 9999 and 3) write the number to output.

⟨ *process the request for an ID* 8a ⟩ ≡
```
if(array_key_exists("getID", $_GET )){
  ⟨ read and update the content of the ID file 8b ⟩
  ⟨ write the ID 8e ⟩
};
```
◇

Fragment referenced in 17c.

⟨ *read and update the content of the ID file* 8b ⟩ ≡
```
$ID=file_get_contents('/usr/local/share/nlpservices/alpino/ID');
if($ID===FALSE){
  $ID=0;
};
⟨ write variable to new file (8c $ID+1,8d "/usr/local/share/nlpservices/alpino/ID" ) 19c ⟩
```
◇

Fragment referenced in 8a.

⟨ *write the ID* 8e ⟩ ≡
```
printf("%04d\n", $ID);
```
◇

Fragment referenced in 8a.

3.2 Upload a file

Uploading a file involves for the user 1) to create a file that can be parsed and that contains the parser-command and 2) upload that file. A suitable form to enable the user to upload a file would be:

⟨ *form to upload a file* 8f ⟩ ≡
```
<form action="m4_root_url" method="post" enc-type="multipart/form-data">
  <input type="file" name="infil" >
  <input type="submit" name="upload" value="upload">
</form>
```
◇

Fragment referenced in 17d.

A client could use this form, e.g. with a curl script as follows:

⟨ *upload the testfile* 8g ⟩ ≡
```
curl -s --form infil=@@1 --form upload=upload "$SERVERURL"
```
◇

Fragment referenced in 9c.
Uses: SERVERURL 7b, 17g.

In the python module the request is made as followa

As mentioned in section 1, the file to be uploaded has to begin with the Alpino command, followed by the text to be parsed. To be precise, the first line of the upload consists of a “sha-bang” followed by the alpino-command. So, the user has to construct such a file. Let us assume that the user only wants a simple parse of text in file `testtext.sts`:

⟨ *variables of testscript* 9a ⟩ ≡
```
ALPINOCOMMAND='#!Alpino -parse'
◇
```
Fragment referenced in 6d.

"testtext.sts" 9b≡
```
Dat is nog niet duidelijk .
We zien hier niet het breken van tandenstokers .
Dodelijke wapens worden gesloopt .
Slechts enkele nieuwe documenten zijn aan het licht gekomen .
Er is nog geen bewijs van verboden activiteiten gevonden .
◇
```

The testscript concatenates the the two file into a single one, prepends the name with the ID and uploads it.

⟨ *upload a file* 9c ⟩ ≡
```
TESTFILENAME=testtext.sts
UTESTFILENAME=$ID"."$TESTFILENAME
echo ALPINOCOMMAND >$UTESTFILENAME
cat $TESTFILENAME >>$UTESTFILENAME
⟨ upload the testfile (9d $UTESTFILENAME ) 8g ⟩
rm $UTESTFILENAME
◇
```
Fragment referenced in 6d.

The python library proceeds as follows:

⟨ *methods in AlpinoLisa class* 9e ⟩ ≡
```
def upload_text(self, alpinocommand, filnam):
  idfilnam =  self.id + filnam
  files = {'infil': (idfilnam, '#!' + alpinocommand + '\n' + open(filnam, 'r').read())}
  r = requests.post(serverurl, files=files)
  return r

◇
```
Fragment defined by 9e, 11a, 13f.
Fragment referenced in 3a.
Uses: `requests` 9f, `serverurl` 7a, 17f, `upload_text` 4f.

⟨ *imports of alpinolisa.py* 9f ⟩ ≡
```
import requests
◇
```
Fragment defined by 7f, 9f.
Fragment referenced in 3a.
Defines: `requests` 9e.

The server puts the uploaded file in the intray.

⟨ *process a file upload* 10a ⟩ ≡
```
if(array_key_exists('infil', $_FILES)){
  $uploaddir = '/usr/local/share/nlpservices/alpino/intray';
  $uploadfile = $uploaddir . '/' . basename($_FILES['infil']['name']);
  $result=move_uploaded_file($_FILES['infil']['tmp_name'], $uploadfile);
  if($result){
    printf("waiting\n");
  }else{
    printf("lost\n");
  };
}

```
◇

Fragment referenced in 17c.

3.3 Check status of files

The client may check the status of a file that she has sent. There are four possible states: "waiting", "being processed", "ready", "unknown". Furthermore, there will be a web page in which the user can see the status of every file that she has sent.

The status of a file can be retrieved with the following form:

⟨ *form to check the status of a file* 10b ⟩ ≡
```
<form action="m4_root_url" method="get">
  <input type="text" name="filnam">
  <input type="submit" name="filstat" value="status">
</form>
```
◇

Fragment referenced in 17d.

The curl script to retrieve the status will then be:

⟨ *ask status of file* 10c ⟩ ≡
```
FILSTAT=`curl -s "http://nlp.labs.vu.nl/webservice/index.php?filstat=status&filnam=@1"`
```
◇

Fragment referenced in 13a.
Defines: `FILSTAT` 13a.

In python it works as follows:

⟨ *http requests in alpinolisa.py* 10d ⟩ ≡
```
statusrequest=serverurl + "?filstat=status&filnam="
```
◇

Fragment defined by 7a, 10d, 13e, 17f.
Fragment referenced in 3a.
Defines: `statusrequest` 11a.
Uses: `serverurl` 7a, 17f.

⟨ *methods in AlpinoLisa class* 11a ⟩ ≡
```
def check_status(self, filnam):
  idfilnam = self.id + filnam
  resp, status = httplib2.Http().request(statusrequest + idfilnam)
  return status
```
◇
Fragment defined by 9e, 11a, 13f.
Fragment referenced in 3a.
Uses: `httplib2` 7f, `statusrequest` 10d.

Let us process this request. If the user supplied a filename, get the state from Lisa, put it in variable `$filstat` and write it out. Otherwise, write state "unknown".

⟨ *process the request for file-state* 11b ⟩ ≡
```
if(array_key_exists("filstat", $_GET )){
  if(!array_key_exists("filnam", $_GET )){
    $filstat="unknown";
  } else {
    $filename=$_GET['filnam'];
    ⟨ get the state of "filename" 11d ⟩
  };
  ⟨ write the state (11c $filstat ) 12m ⟩
};
```
◇
Fragment referenced in 17c.

Lisa sends at regular intervals reports of the files being processed. The report is printed in file `/usr/local/share/nlpservices/alpino/status`. Do as follows:

1. Check whether the parse is present in the outtray. If that is the case, report state "ready".
2. If that is not the case, check whether the file is still in the intray, waiting to be moved to Lisa. In that case, report state "waiting".
3. Get the status from the statusreport from Lisa.
4. report the status.

⟨ *get the state of "filename"* 11d ⟩ ≡
```
$filstat="false";
⟨ find out whether the file is in tray (11e outtray,11f ready ) 12a ⟩
if($filstat=="false"){
  ⟨ find out whether the file is in tray (11g intray,11h waiting ) 12a ⟩
};
if($filstat=="false"){
  ⟨ get the status from the statusfile 12b ⟩
};
if($filstat=="false") $filstat="unknown";
```
◇
Fragment referenced in 11b.

Check whether the file is still in the intray or already in the outtray. The first argument of the following macro is the name of the tray and the second argument is the state that belongs to that tray.

⟨ *find out whether the file is in tray* 12a ⟩ ≡
```
$outfil="/usr/local/share/nlpservices/alpino" . "/@1/" . $filename;
if(file_exists($outfil)){
  $filstat="@2";
};
```
◇

Fragment referenced in 11d.

If the statusfile is not obsolete, read from it until the filename has been found. The statefile is a text file with one word per line. Each line contains either the name of a "tray" in which files can reside, or the name of a file. The trays can be "intray:", "proctray:" or "outttray:". The intray contains the files that are waiting to be processed, the proctray contains the files that are being processed and the outtray contains the processed files.

Not that the files in the outtray of Lisa are not yet accessible for download to a client. Therefore, files in that list must be labeled as "being processed".

⟨ *get the status from the statusfile* 12b ⟩ ≡
```
$statusfile = '/usr/local/share/nlpservices/alpino/status';
$handle=fopen($statusfile, 'r');
$bufstat="unknown";
while(($buffer = fgets($handle, 4096)) !== false){
  ⟨ process status label (12c intray:,12d waiting ) 12k ⟩
  ⟨ process status label (12e proctray:,12f being processed ) 12k ⟩
  ⟨ process status label (12g outtray:,12h being processed ) 12k ⟩
  ⟨ process status label (12i toootray:,12j timeout ) 12k ⟩
  ⟨ process filename 12l ⟩
};
fclose($handle);
```
◇

Fragment referenced in 11d.

⟨ *process status label* 12k ⟩ ≡
```
if(preg_match("/^@1/", $buffer)==1){
  $bufstat="@2";
  continue;
}
```
◇

Fragment referenced in 12b.

⟨ *process filename* 12l ⟩ ≡
```
if(preg_match("/^" . $filename . "/", $buffer)==1){
  $filstat=$bufstat;
  break;
}
```
◇

Fragment referenced in 12b.

⟨ *write the state* 12m ⟩ ≡
```
printf("%s\n", @1);
```
◇

Fragment referenced in 11b.

In the testscript we are going to ask for the state of the file that we have sent.

⟨ *ask for the state of a file* 13a ⟩ ≡
```
⟨ ask status of file (13b $UTESTFILENAME ) 10c ⟩
echo File state: $FILSTAT;
◇
```
Fragment referenced in 6d.

3.4 Retrieve parses

To retrieve the parse of an input-file, the user could fill in a form like the following:

⟨ *form to retrieve a parse* 13c ⟩ ≡
```
<form action="m4_root_url" method="get">
  <input type="text" name="filnam">
  <input type="submit" name="getparse" value="getparse">
</form>
◇
```
Fragment referenced in 17d.

The curl script to print the retrieved file will then be:

⟨ *print parse of* 13d ⟩ ≡
```
echo ‘curl -s "http://nlp.labs.vu.nl/webservice/index.php?getparse=getparse&filnam=@1"‘ >@1
◇
```
Fragment referenced in 14f.
Defines: `PARSE` Never used.

In the python library:

⟨ *http requests in alpinolisa.py* 13e ⟩ ≡
```
retrieverequest = serverurl + ’?getparse=getparse&filnam=’
◇
```
Fragment defined by 7a, 10d, 13e, 17f.
Fragment referenced in 3a.
Defines: `retrieverequest` 13f.
Uses: `serverurl` 7a, 17f.

⟨ *methods in AlpinoLisa class* 13f ⟩ ≡
```
def retrieve_parse(self, filnam):
  idfilnam =  self.id + filnam
  resp, outfilcontent = httplib2.Http().request(retrieverequest + idfilnam)
  if outfilcontent == "notfound\n" :
    raise BaseException(’Could not retrieve ’ + filnam)
  return outfilcontent

◇
```
Fragment defined by 9e, 11a, 13f.
Fragment referenced in 3a.
Defines: `retrieve_parse` 6a.
Uses: `httplib2` 7f, `retrieverequest` 13e.

Note, that the above method assumes that, in response to this request, the server sends the file or it sends the word "notfound".

⟨ *process the request for a parse* 14a ⟩ ≡

```
if(array_key_exists("getparse", $_GET )){
  if(!array_key_exists("filnam", $_GET )){
    printf("notfound\n");
  } else {
    $filename= "/usr/local/share/nlpservices/alpino/outtray" . "/" . $_GET['filnam'];
    if(file_exists($filename)){
      ⟨ print file (14b $filename ) 14d ⟩
      ⟨ remove file (14c $filename ) 14e ⟩
    } else {
      printf("notfound\n");
    };
  };
};

◇
```

Fragment referenced in 17c.

Print the file in the argument. Note that opening and reading a file using PHP's `fopen()` instruction and the FEOF() test can lead to an infinite loop and an enormous logfile that takes up all the free discspace of the server. The `fopen()` instruction returns either a file handle or the boolean FALSE. In the latter case, `feof()` will print error messages tot the log file, but never return `true`. Therefore, if `fopen()` returns `false`, just print `notfound`.

⟨ *print file* 14d ⟩ ≡

```
$handle=fopen(@1, 'r');
if($handle===FALSE){
  printf("notfound\n");
}else{
  $bufsize=10000;
  while(!feof($handle)){
    print fread($handle, $bufsize);
  }
};
fclose($handle);
◇
```

Fragment referenced in 14a.

Remove a file:

⟨ *remove file* 14e ⟩ ≡

```
unlink(@1);
◇
```

Fragment referenced in 14a.
Defines: `unlink` 16ab.

The testfile downloads a parse:

⟨ *download a parse* 14f ⟩ ≡

```
⟨ print parse of (14g test01.sts.alp ) 13d ⟩
◇
```

Fragment referenced in 6d.

3.5 Upload a directory with software

Users can install a directory with their own scripts and programs and use these to process their texts. It works as follows: The user creates a directory tree with her own name as the root directory. In this directory she puts scripts and executables. Next, she packs the directory tree into a "gzipped" tar file. Then, she uploads the directory, with her own name as argument. If all goes well, the directory will be unpacked in Lisa as subdirectory of `nlp/service`.

⟨ *form to upload a directory* 15a ⟩ ≡
```
<form action="m4_root_url" method="post" enc-type="multipart/form-data">
  <input type="text" name="root" >
  <input type="file" name="tarball" >
  <input type="submit" name="upload" value="upload">
</form>
```
◇

Fragment referenced in 15b.

Process this form in a separate PHP file, because this upload is intended to be performed manually by the user.

"uploaddir.php" 15b≡
```
<html>
<head>
</head>
<body>
<?php
```
⟨ *process the request to upload a directory* 15e ⟩
```
?>
```
⟨ *form to upload a directory* 15a ⟩
```
</body>
</html>
```
◇

Don't forget to install the PHP script (put it in its place)

⟨ *expliciete make regels* 15c ⟩ ≡
⟨ *install a php file* (15d `uploaddir`) 21e ⟩
◇

Fragment defined by 15c, 17a, 21ab, 22d, 24.
Fragment referenced in 20.

⟨ *process the request to upload a directory* 15e ⟩ ≡
```
if(array_key_exists('tarball', $_FILES)){
```
⟨ *receive the tarball* 16a ⟩
⟨ *receive the name of the rootdir* 16c ⟩
⟨ *upload the tarball to Lisa* 16d ⟩
⟨ *remove the tarball* 16b ⟩
```
}
```
◇

Fragment referenced in 15b.

⟨ *receive the tarball* 16a ⟩ ≡

```
$uploadfile = tempnam();
unlink($uploadfile);
$result=move_uploaded_file($_FILES['infil']['tmp_name'], $uploadfile);
```

◇

Fragment referenced in 15e.
Uses: `unlink` 14e.

⟨ *remove the tarball* 16b ⟩ ≡

```
unlink($uploadfile);
```

◇

Fragment referenced in 15e.
Uses: `unlink` 14e.

⟨ *receive the name of the rootdir* 16c ⟩ ≡

```
$rootname=$_GET['root'];
```

◇

Fragment referenced in 15e.

⟨ *upload the tarball to Lisa* 16d ⟩ ≡

```
/usr/local/apache/sshcommand "nlp/service/bin/download_archive " . $rootname  <$uploadfile
```

◇

Fragment referenced in 15e.

4 Operations

4.1 Bookkeeping of texts

When the service is actually processing files, connection with the Lisa server is needed at regular intervals. However, when there are no clients, it is not necessary to repeatedly contact the server.

5 Overview of the scripts

5.1 Script that processes user requests

The interface functions are implemented in a single PHP script. The script processes HTTP requests and displays forms that can be used interactively.

The script is located in a directory that makes it accessible for the HTML server. Currently the URL for the interface-server is `m4_projurl` and the directory that belongs to this URL is `/srv/www/nlp.labs.vu.nl/public_html/webservice`.

"index.php" 16e≡

```
<?php
  ⟨ run pseudocron 18a ⟩
  ⟨ process requests 17c ⟩
?>
```

◇

Put the script in its proper place:

⟨ *expliciete make regels* 17a ⟩ ≡
⟨ *install a php file* (17b `index`) 21e ⟩
◇
Fragment defined by 15c, 17a, 21ab, 22d, 24.
Fragment referenced in 20.

The script processes the following requests:

- A request to get and ID. With the ID a user can generate unique filenames that cannot be generated by other users. This is needed because the files of all users will be pooled.
- Upload of a file that contains the text to be parsed.
- Request for information about the status of an upload. It tells the user whether the file is still waiting, being processed or ready.
- Request to download a parse.

⟨ *process requests* 17c ⟩ ≡
⟨ *process the request for an ID* 8a ⟩
⟨ *process a file upload* 10a ⟩
⟨ *process the request for file-state* 11b ⟩
⟨ *process the request for a parse* 14a ⟩
◇
Fragment referenced in 16e.

It is possible (mainly for testing purposes) to submit requests manually with forms.

`"forms"` 17d≡
⟨ *html header* (17e `Forms for Alpino interface`) 19a ⟩
⟨ *form to request a unique identifier* 7c ⟩
⟨ *form to upload a file* 8f ⟩
⟨ *form to check the status of a file* 10b ⟩
⟨ *form to retrieve a parse* 13c ⟩
◇

There is a Python library to help python users. It is used in the example of the previous section.

There is a demo script for Python. Put this script in a directory together with files that contain Dutch plain text and that have filenames that end with `.txt`. When the user runs the demo-script and all goes will, she will eventually get a file with the parse for each of the text files.

The scripts issue requests to the URL that belongs to this script, i.e. `http://nlp.labs.vu.nl/webservice/index.php`

⟨ *http requests in alpinolisa.py* 17f ⟩ ≡
```
serverurl = 'http://nlp.labs.vu.nl/webservice/index.php'
```
◇
Fragment defined by 7a, 10d, 13e, 17f.
Fragment referenced in 3a.
Defines: `serverurl` 7ae, 9e, 10d, 13e.

⟨ *http requests in testscript* 17g ⟩ ≡
```
SERVERURL=http://nlp.labs.vu.nl/webservice/index.php
```
◇
Fragment defined by 7b, 17g.
Fragment referenced in 6d.
Defines: `SERVERURL` 7bd, 8g.

5.2 Cron script

Communicate periodically with Lisa, to send files from the intray, to get files from the outtray and to get a list of files.

⟨ *run pseudocron* 18a ⟩ ≡

```
$indicatorfile="/usr/local/share/nlpservices/alpino/.userreq";
if (file_exists($indicatorfile)) {
  $runcron=(time()-fileatime($indicatorfile) >= 60);
} else {
  $runcron=(1==1);
};
if($runcron){
  shell_exec("/usr/local/share/nlpservices/alpino/bin/cronscript");
  $result=touch($indicatorfile);
};
◇
```

Fragment referenced in 16e.

"../bin/cronscript" 18b≡

```
#!/bin/bash
⟨ upload new texts 18c ⟩
⟨ download parses 18e ⟩
⟨ request filelist 18d ⟩
◇
```

⟨ *upload new texts* 18c ⟩ ≡

```
if [ "$(ls -A /usr/local/share/nlpservices/alpino/intray)" ]
then
 OLDD=‘pwd‘
 cd /usr/local/share/nlpservices/alpino/intray
 tar czf - * | /usr/local/apache/sshcommand "nlp/service/bin/download_archive $filename"
 rm *
fi
◇
```

Fragment referenced in 18b.

Request a list with the state of the uploaded texts. The script seems to expect some text, so give it something with the `yes` script.

⟨ *request filelist* 18d ⟩ ≡

```
yes | /usr/local/apache/sshcommand "nlp/service/bin/filstat \"aap\"" > /usr/local/share/nlpservices/al
◇
```

Fragment referenced in 18b.

⟨ *download parses* 18e ⟩ ≡

```
rsync -avz --remove-source-files  phuijgen@lisa.sara.nl:nlp/service/outtray /usr/local/share/nlpservic
◇
```

Fragment referenced in 18b.

5.3 HTML environment

Build the header of the HTML script with style instructions. This is all stolen from pipet (pipet.sourceforge.nl).

⟨ *html header* 19a ⟩ ≡
```
<!xml version="1.0" encoding="UTF-8">
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN" "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
  <title>@1</title>
  ⟨ html style 19b ⟩
</head>
◇
```
Fragment referenced in 17d.

⟨ *html style* 19b ⟩ ≡
```
<style type="text/css">
  body { background-color: #f4f4ff; }
  h1 { color: #444; }
  a { color: #44a; text-decoration: none; }
  a:hover { color: #4a4; }
  dt, label { color: #944; font-weight: bold; }
  input.button { background-color: #fff4f4; color: #944; border: 1px solid #944; }
  .poweredBy { font-size: small; }
</style>
◇
```
Fragment referenced in 19a.

6 Basic operations

⟨ *write variable to new file* 19c ⟩ ≡
```
$handle=fopen(@2, "w");
$result=fwrite($handle, @1);
$result=fclose($handle);
◇
```
Fragment referenced in 8b.

A How to read and translate this document

This document is an example of *literate programming* [1]. It contains the code of all sorts of scripts and programs, combined with explaining texts. In this document the literate programming tool `nuweb` is used, that is currently available from Sourceforge (URL:nuweb.sourceforge.net). The advantages of Nuweb are, that it can be used for every programming language and scripting language, that it can contain multiple program sources and that it is very simple.

A.1 Read this document

The document contains *code scraps* that are collected into output files. An output file (e.g. `output.fil`) shows up in the text as follows:

"output.fil" 4a ≡
```
# output.fil
< a macro 4b >
< another macro 4c >
◇
```

The above construction contains text for the file. It is labelled with a code (in this case 4a) The constructions between the < and > brackets are macro's, placeholders for texts that can be found in other places of the document. The test for a macro is found in constructions that look like:

< a macro 4b > ≡
```
This is a scrap of code inside the macro.
It is concatenated with other scraps inside the
macro. The concatenated scraps replace
the invocation of the macro.
```

Macro defined by 4b, 87e
Macro referenced in 4a

Macro's can be defined on different places. They can contain other macroÂ´s.

< a scrap 87e > ≡
```
This is another scrap in the macro. It is
concatenated to the text of scrap 4b.
This scrap contains another macro:
< another macro 45b >
```

Macro defined by 4b, 87e
Macro referenced in 4a

A.2 Process the document

The raw document is named `a_nlp_middle.w`.

A.3 Translate and run

This chapter assembles the Makefile for this project.

"Makefile" 20≡
⟨ *parameters in Makefile* 21c, ... ⟩
⟨ *impliciete make regels* 23a ⟩
⟨ *expliciete make regels* 15c, ... ⟩
⟨ *make targets* 22a, ... ⟩
◇

To begin with, the contents of the nuweb source must be unpacked and placed in the proper directories. To start simple, we put everything in a single directory. Unpacking involves the following steps:

1. Transform `a_nlp_middle` into another file, `m4_nlp_middle`, in which `\$` sequences have been replaced by `$` characters and comments (i.e. an `@%` character sequence, the remainder of the text-line including the end-of-line character) have been removed.
2. Run the m4 preprocessor[1] on `'m4_'nlp_middle` to obtain `nlp_middle`.
3. Unpack `nlp_middle` with nuweb.

The first step:

1. http://www.gnu.org/software/m4/

⟨ *expliciete make regels* 21a ⟩ ≡

```
m4_nlp_middle.w : a_nlp_middle.w
        gawk '{if(match($$0, "@%")) {printf("%s", substr($$0,1,RSTART-1))} else print}' a_nlp_middle.w
          | gawk '{gsub(/[\\][\$$]/, "$$");print}'  > m4_nlp_middle.w
```

◇

Fragment defined by 15c, 17a, 21ab, 22d, 24.
Fragment referenced in 20.

The second step:

⟨ *expliciete make regels* 21b ⟩ ≡

```
nlp_middle.w : m4_nlp_middle.w
        m4 -P m4_nlp_middle.w > nlp_middle.w
```

◇

Fragment defined by 15c, 17a, 21ab, 22d, 24.
Fragment referenced in 20.

The third step involves using the Nuweb program. Nuweb takes care of dependencies by itself.

⟨ *parameters in Makefile* 21c ⟩ ≡

```
NUWEB=/usr/local/bin/nuweb
```

◇

Fragment defined by 21cd, 22c, 23c.
Fragment referenced in 20.
Defines: NUWEB 21e, 22ab.

Currently, installing the software involves to unpack the nuweb source and move it into its proper location. This has to be done in two separate steps because later versions of Nuweb prepend the paths of outputfiles with `.\`.

⟨ *parameters in Makefile* 21d ⟩ ≡

```
WEBDIR=/srv/www/nlp.labs.vu.nl/public_html/webservice
```

◇

Fragment defined by 21cd, 22c, 23c.
Fragment referenced in 20.

⟨ *install a php file* 21e ⟩ ≡

```
$(WEBDIR)/@1.php : nlp_middle.w
        ${NUWEB} nlp_middle.w
        mv @1.php $(WEBDIR)
```

◇

Fragment referenced in 15c, 17a.
Uses: NUWEB 21c.

⟨ *make targets* 22a ⟩ ≡
```
install : $(WEBDIR)/index.php $(WEBDIR)/uploaddir.php
        ${NUWEB} nlp_middle.w
        mv index.php /srv/www/nlp.labs.vu.nl/public_html/webservice
        chmod 775 /usr/local/share/nlpservices/alpino/bin/cronscript
```

◇
Fragment defined by 22ab, 23bd.
Fragment referenced in 20.
Uses: NUWEB 21c.

To test the software, install it and run the testscript.

⟨ *make targets* 22b ⟩ ≡
```
test : nlp_middle.w inst.m4
        ${NUWEB} nlp_middle.w
        chmod 775 ./testscript
        ./testscript
```

◇
Fragment defined by 22ab, 23bd.
Fragment referenced in 20.
Uses: NUWEB 21c.

We use many suffixes that were not known by the C-programmers who constructed the `make` utility. Add these suffixes to the list.

⟨ *parameters in Makefile* 22c ⟩ ≡
```
.SUFFIXES: .pdf .w .tex .html .aux .log .php
```

◇
Fragment defined by 21cd, 22c, 23c.
Fragment referenced in 20.
Defines: SUFFIXES Never used.

A.4 Print the document

To print this document, unpack it, convert it into |pdf| and print. Conversion to |pdf| involves multiple steps with LaTeX and nuweb. This is performed with the `w2pdf` script, that can be obtained from http://nlp.labs.vu.nl/mytools/w2pdf:

⟨ *expliciete make regels* 22d ⟩ ≡
```
w2pdf :
        wget http://nlp.labs.vu.nl/mytools/w2pdf
        chmod 775 ./w2pdf
```

◇
Fragment defined by 15c, 17a, 21ab, 22d, 24.
Fragment referenced in 20.

Use the script to generate PDF

⟨ *impliciete make regels* 23a ⟩ ≡

```
%.pdf : %.w w2pdf  $(PDF_FIG_NAMES) $(PDFT_NAMES)
        ./w2pdf $*
```

◇

Fragment referenced in 20.

⟨ *make targets* 23b ⟩ ≡

```
pdf : nlp_middle.pdf

print : nlp_middle.pdf
        lpr nlp_middle.pdf
```

◇

Fragment defined by 22ab, 23bd.
Fragment referenced in 20.

A.5 Render as HTML

Render the document as HTML in a subdirectory.

⟨ *parameters in Makefile* 23c ⟩ ≡

```
HTMLDOCDIR=/usr/local/share/nlpservices/alpino/nuweb/html
```

◇

Fragment defined by 21cd, 22c, 23c.
Fragment referenced in 20.
Defines: `HTMLDOCDIR` 23d.

⟨ *make targets* 23d ⟩ ≡

```
$(HTMLDOCDIR) :
        mkdir $(HTMLDOCDIR)

html : $(HTMLDOCDIR) nlp_middle.w
        cp nlp_middle.w $(HTMLDOCDIR)/nlp_middle.nw
        cd $(HTMLDOCDIR) && w2html nlp_middle.nw
```

◇

Fragment defined by 22ab, 23bd.
Fragment referenced in 20.
Uses: `HTMLDOCDIR` 23c.

Bibliography To keep this document portable, create a portable bibliography file. It works as follows: This document refers in the |bibliography| statement to the local `bib`-file `nlp_middle.bib`. To create this file, copy the auxiliary file to another file `auxfil.aux`, but replace the argument of the command `\bibdata{nlp_middle}` to the names of the bibliography files that contain the actual references (they should exist on the computer on which you try this). This procedure should only be performed on the computer of the author. Therefore, it is dependent of a binary file on his computer.

⟨ *expliciete make regels* 24 ⟩ ≡
```
bibfile : nlp_middle.aux /home/paul/bin/mkportbib
        /home/paul/bin/mkportbib nlp_middle litprog

.PHONY : bibfile
```
◇
Fragment defined by 15c, 17a, 21ab, 22d, 24.
Fragment referenced in 20.

B References

B.1 Literature

References

[1] Donald E. Knuth. Literate programming. Technical report STAN-CS-83-981, Stanford University, Department of Computer Science, 1983.

B.2 URL's

Nuweb: nuweb.sourceforge.net

C Indexes

C.1 Filenames

C.2 Macro's

⟨ html style 19b ⟩ Referenced in 19a.
⟨ http requests in alpinolisa.py 7a, 10d, 13e, 17f ⟩ Referenced in 3a.
⟨ http requests in testscript 7b, 17g ⟩ Referenced in 6d.
⟨ impliciete make regels 23a ⟩ Referenced in 20.
⟨ imports of alpinolisa.py 7f, 9f ⟩ Referenced in 3a.
⟨ imports of alpinolisademo 3d ⟩ Referenced in 3c.
⟨ initialise AlpinoLisa class 7e ⟩ Referenced in 3a.
⟨ install a php file 21e ⟩ Referenced in 15c, 17a.
⟨ make targets 22ab, 23bd ⟩ Referenced in 20.
⟨ methods in AlpinoLisa class 9e, 11a, 13f ⟩ Referenced in 3a.
⟨ open interface with the server 4a ⟩ Referenced in 3c.
⟨ parameters in Makefile 21cd, 22c, 23c ⟩ Referenced in 20.
⟨ poll and download 5a ⟩ Referenced in 3c.
⟨ poll, retrieve if ready and update status 6a ⟩ Referenced in 5a.
⟨ print a summary 6c ⟩ Referenced in 5a.
⟨ print file 14d ⟩ Referenced in 14a.
⟨ print parse of 13d ⟩ Referenced in 14f.
⟨ process a file upload 10a ⟩ Referenced in 17c.
⟨ process filename 12l ⟩ Referenced in 12b.
⟨ process requests 17c ⟩ Referenced in 16e.
⟨ process status label 12k ⟩ Referenced in 12b.
⟨ process the request for a parse 14a ⟩ Referenced in 17c.
⟨ process the request for an ID 8a ⟩ Referenced in 17c.
⟨ process the request for file-state 11b ⟩ Referenced in 17c.
⟨ process the request to upload a directory 15e ⟩ Referenced in 15b.
⟨ read and update the content of the ID file 8b ⟩ Referenced in 8a.
⟨ receive the name of the rootdir 16c ⟩ Referenced in 15e.
⟨ receive the tarball 16a ⟩ Referenced in 15e.
⟨ remove file 14e ⟩ Referenced in 14a.
⟨ remove file with kaf extension 4b ⟩ Referenced in 4c.
⟨ remove the tarball 16b ⟩ Referenced in 15e.
⟨ request filelist 18d ⟩ Referenced in 18b.
⟨ rubencommand 3b ⟩ Referenced in 3c.
⟨ run pseudocron 18a ⟩ Referenced in 16e.
⟨ status trick 4g ⟩ Referenced in 4c.
⟨ submit a single text 4f ⟩ Referenced in 4c, 6a.
⟨ submit texts 4c ⟩ Referenced in 3c.
⟨ upload a file 9c ⟩ Referenced in 6d.
⟨ upload new texts 18c ⟩ Referenced in 18b.
⟨ upload the tarball to Lisa 16d ⟩ Referenced in 15e.
⟨ upload the testfile 8g ⟩ Referenced in 9c.
⟨ variables of testscript 9a ⟩ Referenced in 6d.
⟨ write the ID 8e ⟩ Referenced in 8a.
⟨ write the state 12m ⟩ Referenced in 11b.
⟨ write variable to new file 19c ⟩ Referenced in 8b.

C.3 Variables

`FILSTAT`: 10c, 13a.
`HTMLDOCDIR`: 23c, 23d.
`httplib2`: 7e, 7f, 11a, 13f.
`lisacon`: 4a, 4fg, 6a.
`NUWEB`: 21c, 21e, 22ab.
`pfil`: 6a, 6b.
`requests`: 9e, 9f.
`retrieverequest`: 13e, 13f.
`retrieve_parse`: 6a, 13f.
`SERVERURL`: 7b, 7d, 8g, 17g.