Implementation of WebLicht Webservices in Perl

Helmut Schmid
Institut für maschinelle Sprachverarbeitung
Universität Stuttgart

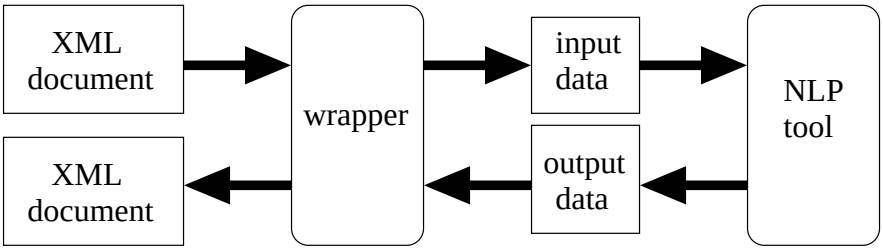
1 Introduction

This document describes the implementation of WebLicht webservices in Perl on Linux systems using a DOM parser. Familiarity with the Perl programming language, XML documents, and WebLicht is assumed.

A WebLicht webservice typically reads an XML document[1] and returns another XML document.

Both documents conform to the TCF format. TCF is an XML-based format for the encoding of text corpora and corpus annotations.

WebLicht webservices are usually based on an existing NLP tool such as a tokenizer, POS tagger, or parser with idiosyncratic input and output formats. These tools are turned into webservices by means of a wrapper program. The wrapper translates between TCF and the tool-specific formats and implements the webservice functionality. Hence there is no need to modify the NLP tool itself.



[1]This is the normal case. There are also some special webservices which read, for instance, a plain text document and return an XML document.

The processing of a webservice request by a wrapper typically comprises the following steps:

1. reading of the input document
2. extraction of data from the input document
3. sending of the data to the tool in its proprietary format
4. reading of the output data of the NLP tool
5. generation of the output XML document

We will now show step by step how wrappers can be implemented in Perl. The rest of this tutorial is organized as follows: Section 2 presents the TCF format. Sections 3 and 4 describe the implementation of two wrapper examples, and Sections 5 and 6 deal with the testing and installation of webservice wrappers.

2 The TCF Format

The TCF format is an XML-based format for the encoding of text corpora and their annotations. The top-most element has the name **D-Spin**. It dominates a **MetaData** element and a **TextCorpus** element[2]. The **MetaData** element is currently empty. In the future it will be used to store metadata information about the document. There should be no need for webservices to read or modify the **MetaData** element.

The example TCF document in Appendix A contains a **TextCorpus** with a **text** element which encloses the plain text of the corpus[3]. The different annotations of the corpus are encoded by means of the elements **tokens**, **sentences**, **POStags**, **lemmas**, **parsing**, and **depparsing**. The example document could have been created by a typical webservice pipeline which starts with a plain text file which is first converted to a TCF document with a **text** element. This document is sent to a tokenizer which adds a **tokens** element, a POS tagger which adds a **POStags** element and a **lemmas** element, a constituent parser which adds a **parsing** element, and a dependency parser which adds the **depparsing** element.

The **tokens** element encodes the tokenization of the corpus by means of a sequence of **token** elements. Each **token** has an **ID** attribute which uniquely identifies it, and optional **start** and **end** attributes which refer to the position of the first/last character of the token in the plain **text** element. The **tokens** attribute **charOffsets="true"** indicates that all **token** elements have **start** and **end** attributes. The **token** XML tag

[2]The TCF format also allows a **Lexicon** element instead of the **TextCorpus** element. **Lexicon** elements are used to encode lexicons but this part of the TCF format is not well developed yet.

[3]This is somewhat problematic because some XML parsers have difficulties to deal with huge text strings.

encloses the token string. This particular representation is redundant because the token string can also be inferred from the **start** and **end** attributes.

The **sentences** element encodes the sentence boundaries by means of a sequence of **sentence** elements. Each **sentence** has a **tokenIDs** attribute. Its value is the sequence of IDs of the **token**s which belong to this sentence. the **sentence** elements have optional **start** and **end** attributes which specify the boundaries of the sentence within the **text** element.

The **POStags** element encodes part-of-speech information. Its **tagset** attribute specifies the tagset. Each **tag** inside of a **POStags** element refers to one or more tokens by means of the **tokenIDs** attribute and encloses the corresponding part-of-speech label. Since the **tokenIDs** attribute may specify a sequence of token IDs, it is possible to encode words consisting of multiples tokens and even discontinuous words.

The representation of **lemmas** is analogous to that of **POStags**.

parsing elements encode constituent parse trees. The **tagset** attribute specifies the set of grammatical labels used. Each **parse** consists of one **constituent** element. Each **constituent** bears a **cat**egory attribute and either encloses a set of other **constituent**s or it has a **tokenIDs** attribute which specifies the token(s) of the respective terminal node of the parse tree.

depparsing elements represent dependency parses. Each **parse** consists of a sequence of **dependency** elements which represent the word-governor dependencies. The grammatical function of the dependent is encoded in the **func** attribute of the **dependency** element.

A formal specification of the TCF format by means of RelaxNG schemas can be found in Appendix D.

3 Wrapper Example 1: txt2tcf.perl

We will start with a first example of a wrapper program, namely the **txt2tcf.perl** webservice implemented at the University of Stuttgart. It converts a plain text document into a TCF document with UTF-8 encoding. Using the **wget** program which is available on most Linux systems, the webservice can be called as follows to convert the plain text file **test.txt** into the TCF document **test.xml**:

```
> wget --post-file=test.txt http://gelbaugenpinguin.ims.uni-stuttgart.de/
  cgi-bin/dspin/txt2tcf3.perl --output-document=test.xml
```

The wrapper reads the input file, guesses the character encoding, converts to Unicode, generates an XML document, and prints it.

We will now look line by line at the actual code in order to understand how the program works. The complete code is listed in Appendix B.

The first line of the wrapper program tells the operating system the location of the Perl interpreter which is to be used to execute this script:

```
#!/usr/bin/perl
```

The following lines load several Perl modules which the wrapper uses.

```
use Encode;
use Encode::Guess;
use XML::LibXML;
```

The **Encode** module is needed to convert strings between different character sets. **Encode::Guess** provides a character set guesser, and the **XML::LibXML** module implements a Perl interface to the efficient XML parser **libxml2** which is implemented in C++. A documentation of the **XML::LibXML** module is available at http://search.cpan.org/dist/XML-LibXML.

libxml2 is a DOM parser which reads an XML file into an internal data structure for further processing and manipulation. DOM parsers are more comfortable to work with than SAX parsers which process the XML file without building an internal data structure by calling user-defined processing function via hooks whenever a predefined XML element is parsed. The disadvantage of DOM parsers is their bigger demand for memory. **libxml2** is an efficient DOM parser which is able to process large XML files.

The next two lines define variables for the name of the top element of the TCF document (D-Spin) and the encoding (UTF-8) of the XML file which will be produced as output.

```
my $header = 'D-Spin';
my $encoding = 'UTF-8';
```

Now the namespaces used in the output document are defined.

```
my $def = 'http://www.dspin.de/data';
my $tc = 'http://www.dspin.de/data/textcorpus';
my $md = 'http://www.dspin.de/data/metadata';
```

The whole input file is read into the variable **$in**.

```
my $in;
{ local $/; $in = <>; }
```

The command `local $/;` temporarily undefines the input record separator in order to allow the following command `$in = <>` to read in the whole file at once.

Now the guesser is called to determine the encoding of the file.

```
my $enc = guess_encoding($in);
$enc = guess_encoding($in, 'latin1') unless ref($enc);
```

By default, it considers ASCII and the different Unicode variants. If none of these applies, the guesser is called again to check whether the input is compatible with a Latin1 encoding. If not, an error is reported by calling the function **report_error**:

```
report_error(1,"invalid input","unable to guess encoding") unless ref($enc);
```

The next command decodes the input string into Perl's internal string format:

```
$in = $enc->decode($in);
```

A new XML document with the root element **D-Spin** is created with the namespace stored in $def.

```
my $doc = XML::LibXML->createDocument('1.0', $encoding);
my $root = $doc->createElementNS( $def, "D-Spin" );
```

A version attribute, an empty **MetaData** child element, and a **TextCorpus** child element are added:

```
$root->addChild($doc->createAttribute('version', '0.3'));
$root->addNewChild( $md, 'MetaData' );
my $corpus = $root->addNewChild( $tc, 'TextCorpus' );
```

Finally, a child element **text** is added to the **TextCorpus** element and the input file is stored as its content.

```
$doc->setDocumentElement( $root );
my $text = $corpus->addNewChild($tc, 'text');
$text->addChild($doc->createTextNode($in));
```

Now we are ready to print the HTTP header and the resulting XML document:

```
print "Content-Type: text/xml; charset=$encoding\n\n";
print $doc->toString(1);
```

The argument of the libxml2 serialisation function **toString** turns on the pretty print mode.

Whenever the wrapper encounters an error, it calls the function **report_error** which prints an HTTP header and an XML document with an error report and terminates.

```
sub report_error {
    my $id = shift;
    my $code = shift;
    my $message = shift;
    print "Content-Type: text/xml; charset=utf-8
Status: 400 bad request

<?xml version=\"1.0\" encoding=\"utf-8\"?>
<$header xmlns=\"http://www.dspin.de/data\" version=\"0.3\">
 <error xmlns=\"http://www.dspin.de/data/error\">
  <type code=\"$id\">$code</code>
  <message>$message</message>
 </error>
</$header>
";
    die "Error: $message!";
}
```

4 Wrapper Example 2: tree-tagger.perl

The **tree-tagger.perl** wrapper implements a POS tagging webservice for different languages based on the TreeTagger software. The input document is a TCF document with **tokens**. The wrapper returns the same document with the two additional elements **POStags** and **lemmas**. The TreeTagger has to read a large parameter file when it is started. This takes some time. Therefore the wrapper connects to a TreeTagger **daemon** which resides in memory. The wrapper and the tagger daemon communicate via sockets. The complete wrapper code is found in Appendix C.

The TreeTagger wrapper uses the following Perl modules:

```
use Encode;
use IO::Socket;
use XML::LibXML;
use XML::LibXML::XPathContext;
```

The **IO::Socket** module is required to create a connection to the TreeTagger daemon. **XML::LibXML::XPathContext** is needed to execute XPath expressions which extract the **token** elements from the TCF document.

The next commands define the name of the header element and the required name spaces.

```
my $header = 'D-Spin';
```

```
my $def = 'http://www.dspin.de/data';
my $tc = 'http://www.dspin.de/data/textcorpus';
```

Now we define mappings between languages and port numbers and between languages and tagsets. There is a separate tagger daemon for each language and each daemon is associated with a port.

```
my($in,%port,%tagset);
$port{'en'} = 7070;
$port{'de'} = 7071;
$port{'fr'} = 7072;
$port{'it'} = 7073;

$tagset{'en'} = "PennTB";
$tagset{'de'} = "STTS";
$tagset{'fr'} = "SteinFR";
$tagset{'it'} = "SteinIT";
```

The input XML file is read and parsed:

```
{ local $/; $in = <>; }

my $parser = XML::LibXML->new();
my $dom = $parser->parse_string( $in );
```

An error is reported if the parsing failed:

```
report_error(1, "invalid input", "unable to parse XML document")
    unless defined $dom;
```

The encoding of the input document is stored in a variable.

```
my $encoding = $dom->encoding();
```

An **XPathContext** is created for the document and the two namespaces are registered.

```
my $xc = XML::LibXML::XPathContext->new( $dom->documentElement() );
$xc->registerNs( 'def', $def );
$xc->registerNs( 'tc', $tc );
```

The following loop finds all TextCorpus elements (although currently only one TextCorpus element is allowed in TCF documents). The XPath expression `/*/tc:TextCorpus` matches any **TextCorpus** child elements of the top-most node (which is named D-Spin).

```
my $corpus;
foreach $corpus ($xc->findnodes( '/*/tc:TextCorpus' )) {
```

The language attribute of the **TextCorpus** is retrieved. If it is not one of the covered languages, nothing is done.

```
    my $lang = $corpus->getAttribute( 'lang' );
    next unless exists $port{$lang};
```

The **tokens** element is searched inside the **TextCorpus** element. If there is none, the wrapper reports an error and terminates.

```
    report_error(1,"invalid input","no input tokens")
        unless $xc->findnodes( 'tc:tokens' ,$corpus);
```

Now the list of tokens is extracted.

```
    my @tokens = $xc->findnodes( 'tc:tokens/tc:token' ,$corpus);
```

A connection to the socket of the tagger daemon for the respective language is created. An error is reported if the connection could not be established.

```
    my $sock = new IO::Socket::INET( PeerAddr => 'localhost',
                                     PeerPort => $port{$lang},
                                     Proto => 'tcp');
    report_error(2, "internal error", "unable to connect to socket")
        unless $sock;
```

The token strings are extracted and sent to the tagger in UTF-8 encoding with one token per line. Empty tokens are reported as an error.

```
    my $token;
    foreach $token (@tokens) {
        my $tok = $token->textContent;
        report_error(1, "invalid input", "empty token in XML document")
            if $tok eq '';
        print $sock encode("utf-8", $tok),"\n";
    }
```

The input connection to the tagger daemon is closed.

```
    shutdown($sock, 1);
```

Two new XML elements **POStags** and **lemmas** are added to the **TextCorpus** element. and the tagset attribute of the **POStags** element is properly defined.

```
    my $POStags = $corpus->addNewChild($tc, 'POStags');
    my $lemmas = $corpus->addNewChild($tc, 'lemmas');
    $POStags->addChild($dom->createAttribute('tagset', $tagset{$lang}));
```

Now the output of the tagger daemon is read. There is one line of output for each input token. We iterate over the input tokens, read one line for each token, and decode it from UTF-8 format. An error is reported if nothing was read.

```
    foreach $token (@tokens) {
        $_ = decode("utf-8", <$sock>);
        report_error(2, "internal error", "erroneous socket output")
            unless defined $_;
```

The newline character is stripped and the input is split at tab character positions. The elements are stored in the variables **$w**, **$t**, and **$l**.

```
        chomp;
        my($w,$t,$l) = split(/\t/);
```

The token ID is retrieved and an error is reported if it cannot be found.

```
        my $id = $token->getAttribute('ID');
        report_error(1, "invalid input", "missing token ID in XML document")
            unless defined $id;
```

A **tag** element is added to **POStags** for each **token**.

```
        add_child( $POStags, 'tag', $id, $t);
```

Similarly, a **lemma** element is added to **lemmas** for each **token**.

```
        add_child( $lemmas, 'lemma', $id, $l);
```

The socket connection is closed.

```
    close $sock;
```

Finally the result document is printed.

```
print "Content-Type: text/xml; charset=$encoding\n\n";
print $dom->toString(1);
```

The following auxiliary function adds an element whose name is the second argument to the node transmitted as the first argument, and stores the token ID as the value of the attribute **tokenIDs**, and the text string as the text content of the element.

```
sub add_child {
    my $node = shift;
    my $name = shift;
    my $id = shift;
    my $text = shift;

    my $child = $node->addNewChild($tc, $name);
    $child->addChild($dom->createAttribute('tokenIDs', $id));
    $child->addChild($dom->createTextNode($text));
}
```

5 Testing

The above wrappers can be tested by calling them with an appropriate argument file. The input can also be "piped" into the program as shown below:

```
> echo "This is a test." | ./txt2tcf.perl
Content-Type: text/xml; charset=UTF-8

<?xml version="1.0" encoding="UTF-8"?>
<D-Spin xmlns="http://www.dspin.de/data" version="0.3">
  <MetaData xmlns="http://www.dspin.de/data/metadata"/>
  <TextCorpus xmlns="http://www.dspin.de/data/textcorpus">
    <text>This is a test.</text>
  </TextCorpus>
</D-Spin>
```

The tagger webservice can be tested with the command

```
> ./tree-tagger.perl test.xml
```

where test.xml contains the following XML document:

```
<?xml version="1.0" encoding="UTF-8"?>
<D-Spin xmlns="http://www.dspin.de/data" version="0.3">
  <MetaData xmlns="http://www.dspin.de/data/metadata"/>
  <TextCorpus xmlns="http://www.dspin.de/data/textcorpus" lang="en">
    <text>This is a test.</text>
```

```
    <tokens>
      <token ID="t1">This</token>
      <token ID="t2">is</token>
      <token ID="t3">a</token>
      <token ID="t4">test</token>
      <token ID="t5">.</token>
    </tokens>
    </sentences>
  </TextCorpus>
</D-Spin>
```

6 Installation

The installation depends on the type of webserver. In case of an Apache server, the wrapper programs need to be copied to the directory **/www/cgi-bin**.

In order to be able to use the webservice in WebLicht, you have to register it in the webservice registry in Leipzig.

A An Example TCF Document

```
<?xml version="1.0" encoding="UTF-8"?>
<D-Spin xmlns="http://www.dspin.de/data" version="0.3">
  <MetaData xmlns="http://www.dspin.de/data/metadata"/>
  <TextCorpus xmlns="http://www.dspin.de/data/textcorpus" lang="en">
    <text>Federal pay czar tries again to trim A.I.G. bonuses.</text>
    <tokens charOffsets="true">
      <token ID="t1" start="0" end="6">Federal</token>
      <token ID="t2" start="8" end="10">pay</token>
      <token ID="t3" start="12" end="15">czar</token>
      <token ID="t4" start="17" end="21">tries</token>
      <token ID="t5" start="23" end="27">again</token>
      <token ID="t6" start="29" end="30">to</token>
      <token ID="t7" start="32" end="35">trim</token>
      <token ID="t8" start="37" end="42">A.I.G.</token>
      <token ID="t9" start="44" end="50">bonuses</token>
      <token ID="t10" start="51" end="51">.</token>
    </tokens>
    <sentences tokRef="true">
      <sentence ID="s1" tokenIDs="t1 t2 t3 t4 t5 t6 t7 t8 t9 t10/>
    </sentences>
    <POStags tagset="PennTB">
      <tag tokenIDs="t1">JJ</tag>
      <tag tokenIDs="t2">NN</tag>
      <tag tokenIDs="t3">NN</tag>
```

```
  <tag tokenIDs="t4">VBZ</tag>
  <tag tokenIDs="t5">RB</tag>
  <tag tokenIDs="t6">TO</tag>
  <tag tokenIDs="t7">VB</tag>
  <tag tokenIDs="t8">NP</tag>
  <tag tokenIDs="t9">NNS</tag>
  <tag tokenIDs="t10">.</tag>
</POStags>
<lemmas>
  <lemma tokenIDs="t1">federal</lemma>
  <lemma tokenIDs="t2">pay</lemma>
  <lemma tokenIDs="t3">czar</lemma>
  <lemma tokenIDs="t4">try</lemma>
  <lemma tokenIDs="t5">again</lemma>
  <lemma tokenIDs="t6">to</lemma>
  <lemma tokenIDs="t7">trim</lemma>
  <lemma tokenIDs="t8">A.I.G.</lemma>
  <lemma tokenIDs="t9">bonus</lemma>
  <lemma tokenIDs="t10">.</lemma>
</lemmas>
<parsing tagset="PennTB">
  <parse>
    <constituent cat="TOP">
      <constituent cat="S">
        <constituent cat="NP-SBJ-1">
          <constituent cat="JJ" tokenIDs="t1"/>
          <constituent cat="NN" tokenIDs="t2"/>
          <constituent cat="NN" tokenIDs="t3"/>
        </constituent>
        <constituent cat="VP">
          <constituent cat="VVZ" tokenIDs="t4"/>
          <constituent cat="ADVP">
            <constituent cat="RB" tokenIDs="t5"/>
          </constituent>
          <constituent cat="S">
            <constituent cat="NP-SBJ">
              <constituent cat="-NONE-">
                <constituent cat="*-1"/>
              </constituent>
            </constituent>
            <constituent cat="VP">
              <constituent cat="TO" tokenIDs="t6"/>
              <constituent cat="VP">
                <constituent cat="VV" tokenIDs="t7"/>
                <constituent cat="NP">
                  <constituent cat="NNP" tokenIDs="t8"/>
                  <constituent cat="NNS" tokenIDs="t9"/>
                </constituent>
              </constituent>
```

```
              </constituent>
            </constituent>
          </constituent>
          <constituent cat="." tokenIDs="t10"/>
        </constituent>
      </constituent>
    </parse>
  </parsing>
  <depparsing tagset="PennTB">
    <parse>
      <dependency func="MOD" depID="t1" govID="t3"/>
      <dependency func="MOD" depID="t2" govID="t3"/>
      <dependency func="SUBJ" depID="t3" govID="t4"/>
      <dependency func="ROOT" depID="t4"/>
      <dependency func="MOD" depID="t5" govID="t4"/>
      <dependency func="TO" depID="t6" govID="t7"/>
      <dependency func="SCOMP" depID="t7" govID="t4"/>
      <dependency func="MOD" depID="t8" govID="t9"/>
      <dependency func="OBJ" depID="t9" govID="t4"/>
    </parse>
  </parsing>
 </TextCorpus>
</D-Spin>
```

B Wrapper Example1: txt2tcf.perl

```
#!/usr/bin/perl -w

use Encode;
use Encode::Guess;
use XML::LibXML;

my $header = 'D-Spin';
my $encoding = 'UTF-8';

my $def = 'http://www.dspin.de/data';
my $tc = 'http://www.dspin.de/data/textcorpus';
my $md = 'http://www.dspin.de/data/metadata';

my $in;
{ local $/; $in = <>; }

my $enc = guess_encoding($in);
$enc = guess_encoding($in, 'latin1') unless ref($enc);
report_error(1,"invalid input","unable to guess encoding") unless ref($enc);
```

```
$in = $enc->decode($in);

my $doc = XML::LibXML->createDocument('1.0', $encoding);
my $root = $doc->createElementNS( $def, "D-Spin" );
$root->addChild($doc->createAttribute('version', '0.3'));
$root->addNewChild( $md, 'MetaData' );
my $corpus = $root->addNewChild( $tc, 'TextCorpus' );
$doc->setDocumentElement( $root );
my $text = $corpus->addNewChild($tc, 'text');
$text->addChild($doc->createTextNode($in));

# generate the output
print "Content-Type: text/xml; charset=$encoding\n\n";
print $doc->toString(1);


sub report_error {
    my $id = shift;
    my $code = shift;
    my $message = shift;
    print "Content-Type: text/xml; charset=utf-8
Status: 400 bad request

<?xml version=\"1.0\" encoding=\"utf-8\"?>
<$header xmlns=\"http://www.dspin.de/data\" version=\"0.3\">
 <error xmlns=\"http://www.dspin.de/data/error\">
  <type code=\"$id\">$code</code>
  <message>$message</message>
 </error>
</$header>
";
    die "Error: $message!";
}
```

C Wrapper Example2: tree-tagger.perl

```
#!/usr/bin/perl -w

use Encode;
use IO::Socket;
use XML::LibXML;
use XML::LibXML::XPathContext;
```

```
my $header = 'D-Spin';

my $def = 'http://www.dspin.de/data';
my $tc = 'http://www.dspin.de/data/textcorpus';

# define the port numbers of the tagger daemons for the different languages
my($in,%port,%tagset);
$port{'en'} = 7070;
$port{'de'} = 7071;
$port{'fr'} = 7072;
$port{'it'} = 7073;

$tagset{'en'} = "PennTB";
$tagset{'de'} = "STTS";
$tagset{'fr'} = "SteinFR";
$tagset{'it'} = "SteinIT";

# read the whole input at once and then restore line-by-line reading
{ local $/; $in = <>; }

# parse the XML input
my $parser = XML::LibXML->new();
my $dom = $parser->parse_string( $in );
report_error(1, "invalid input", "unable to parse XML document")
    unless defined $dom;

my $encoding = $dom->encoding();

my $xc = XML::LibXML::XPathContext->new( $dom->documentElement() );
$xc->registerNs( 'def', $def );
$xc->registerNs( 'tc', $tc );

# process all input corpora
my $corpus;
foreach $corpus ($xc->findnodes( '/*/tc:TextCorpus' )) {
    my $lang = $corpus->getAttribute( 'lang' );
    next unless exists $port{$lang};

    report_error(1,"invalid input","no input tokens")
        unless $xc->findnodes( 'tc:tokens' ,$corpus);

    # extract the tokens which are to be annotated
    my @tokens = $xc->findnodes( 'tc:tokens/tc:token' ,$corpus);
```

```
    # create a socket connection to the tagger daemon
    my $sock = new IO::Socket::INET( PeerAddr => 'localhost',
                                     PeerPort => $port{$lang},
                                     Proto => 'tcp');
    report_error(2, "internal error", "unable to connect to socket")
        unless $sock;

    # send the tokens to the tagger daemon
    my $token;
    foreach $token (@tokens) {
        my $tok = $token->textContent;
        report_error(1, "invalid input", "empty token in XML document")
            if $tok eq '';
        print $sock encode("utf-8", $tok),"\n";
    }
    shutdown($sock, 1);

    # add new XML elements for the tags and the lemmata
    my $POStags = $corpus->addNewChild($tc, 'POStags');
    my $lemmas = $corpus->addNewChild($tc, 'lemmas');
    $POStags->addChild($dom->createAttribute('tagset', $tagset{$lang}));

    # read the POS tags and lemmata from the tagger daemon
    foreach $token (@tokens) {
        $_ = decode("utf-8", <$sock>);
        report_error(2, "internal error", "erroneous socket output")
            unless defined $_;
        chomp;
        my($w,$t,$l) = split(/\t/);
        my $id = $token->getAttribute('ID');
        report_error(1, "invalid input", "missing token ID in XML document")
            unless defined $id;

        # Add a tag element to the document
        add_child( $POStags, 'tag', $id, $t);

        # Add a lemma element to the document
        add_child( $lemmas, 'lemma', $id, $l);
    }
    close $sock;
}

# generate the output
print "Content-Type: text/xml; charset=$encoding\n\n";
```

```
print $dom->toString(1);

sub add_child {
    my $node = shift;
    my $name = shift;
    my $id = shift;
    my $text = shift;

    my $child = $node->addNewChild($tc, $name);
    $child->addChild($dom->createAttribute('tokenIDs', $id));
    $child->addChild($dom->createTextNode($text));
}

sub report_error {
    my $id = shift;
    my $code = shift;
    my $message = shift;
    print "Content-Type: text/xml; charset=utf-8
Status: 400 bad request

<?xml version=\"1.0\" encoding=\"utf-8\"?>
<$header xmlns=\"http://www.dspin.de/data\" version=\"0.3\">
 <error xmlns=\"http://www.dspin.de/data/error\">
  <type code=\"$id\">$code</code>
  <message>$message</message>
 </error>
</$header>
";
    die "Error: $message!";
}
```

D RelaxNG Schema

D-Spin.rnc

```
default namespace = "http://www.dspin.de/data"

element D-Spin {

  attribute version { "0.4" },

  external "MetaData.rnc",
  (external "TextCorpus.rnc" |
```

```
   external "Lexicon.rnc" |
   external "Error.rnc")

}
default namespace = "http://www.dspin.de/data"

element D-Spin {

    attribute version { xsd:string },

  grammar { include "MetaData.rnc" },
  (grammar { include "TextCorpus.rnc" } |
   grammar { include "Lexicon.rnc" } |
   grammar { include "error.rnc" })
}
```

MetaData.rnc

```
default namespace = "http://www.dspin.de/data/metadata"

start =
  element MetaData {
    element source { xsd:string }?
  }
```

TextCorpus.rnc

```
default namespace = "http://www.dspin.de/data/textcorpus"

start =
  element TextCorpus {

    attribute source { xsd:anyURI }?,
    attribute lang { xsd:language }?,

    (\text | tokens | sentences | POStags | lemmas | parsing | depparsing |
     coreferences | namedEntities | sem_lex_rels)+
  }

\text = element \text {
  (attribute location { xsd:anyURI } |
   xsd:string)
}
```

```
tokens =
  element tokens {
    attribute charOffsets { "true" }?,
    element \token {
      attribute ID    { xsd:ID }?,
      (attribute start { xsd:int },
       attribute end   { xsd:int })?,
      xsd:string
    }+
  }

sentences =
  element sentences {
    attribute tokRef { "true" }?,
    attribute charOffsets { "true" }?,
    element sentence {
      attribute ID    { xsd:ID }?,
      (attribute start { xsd:int },
       attribute end   { xsd:int })?,
      attribute tokenIDs   { xsd:IDREFS }
    }+
  }

POStags =
  element POStags {
    attribute tagset { xsd:string },
    element tag {
      attribute ID    { xsd:ID }?,
      attribute tokenIDs { xsd:IDREFS },
      xsd:string
    }+
  }

lemmas =
  element lemmas {
    element lemma {
      attribute ID    { xsd:ID }?,
      (attribute tokenIDs { xsd:IDREFS } |
       attribute tagID { xsd:IDREF }),
      xsd:string
    }+
  }
```

```
parsing =
  element parsing {
    attribute tagset { xsd:string },
    element parse {
      attribute ID    { xsd:ID }?,
      constituent
    }+
  }

constituent =
  element constituent {
    attribute cat { xsd:string },
    attribute ID { xsd:ID }?,
    (attribute tokenIDs { xsd:IDREFS } | constituent*)
  }


depparsing =
  element depparsing {
    attribute tagset { xsd:string }?,
    element parse {
      attribute ID    { xsd:ID }?,
      element dependency {
        attribute func { xsd:string }?,
        attribute depID { xsd:IDREF },
        attribute govID { xsd:IDREF }?
      }+
    }+
  }

relations =
  element relations {
    attribute type { xsd:string },
    element relation {
      attribute ID   { xsd:ID }?,
      attribute func { xsd:string }?,
      attribute refIDs { xsd:IDREFS }
    }+
  }

namedEntities =
  element namedEntities {
    attribute tokRef { "true" }?,
```

```
    attribute charOffsets { "true" }?,
    element entity {
     attribute class { xsd:string },
     attribute ID    { xsd:ID }?,
     (attribute start { xsd:int },
      attribute end   { xsd:int })?,
     attribute tokenIDs { xsd:IDREFS }
    }+
  }


# The remaining declarations are needed for representing the
# semantic relations from GermanNet

  sem_lex_rels =
  element sem_lex_rels {
      attribute src { xsd:string },
      (synonymy | antonymy | pertonymy | arg1 | arg1_pred | arg2 | arg2_pred |
       hyponymy | hyperonymy | association| holonymy | meronymy | entailed |
       entailment | caused | causation)+
  }



  orthform =
  element orthform {
          attribute ID    { xsd:ID }?,
          attribute refID { xsd:IDREF },
          xsd:string
      }

  synonymy =
  element synonymy {
      orthform+
  }

  antonymy =
  element antonymy {
      orthform+
  }

  pertonymy =
  element pertonymy {
      orthform+
```

```
}

hyponymy =
element hyponymy {
    orthform+
}

hyperonymy =
element hyperonymy {
    orthform+
}

association =
element association {
    orthform+
}

holonymy =
element holonymy {
    orthform+
}

meronymy =
element meronymy {
    orthform+
}

entailed =
element entailed {
    orthform+
}

entailment =
element entailment {
    orthform+
}

arg1 =
element arg1 {
    orthform+
}

arg1_pred =
element arg1_pred {
```

```
      orthform+
  }

   arg2 =
  element arg2 {
      orthform+
  }

  arg2_pred =
  element arg2_pred {
      orthform+
  }

  caused =
  element caused {
      orthform+
  }

  causation =
  element causation {
      orthform+
  }
```

Lexicon.rnc

```
default namespace = "http://www.dspin.de/data/lexicon"

start =
  element Lexicon {
    attribute lang { xsd:language }?,
    lemmas, (POStags | frequencies | word-relations)*
  }

lemmas =
  element lemmas {
    element lemma {
      attribute ID { xsd:ID }?,
      xsd:string
    }+
  }

POStags =
  element POStags {
```

```
    attribute tagset { xsd:string }?,
    element tag {
      attribute ID    { xsd:ID }?,
      attribute lemID { xsd:IDREF },
      text
    }+
  }

frequencies =
  element frequencies {
    element frequency {
      attribute lemID { xsd:IDREF },
      xsd:int
    }+
  }

word-relations =
  element word-relations {
    attribute type { xsd:string }?,
    element word-relation {
      attribute func { xsd:string }?,
      attribute freq { xsd:int }?,
      sig*,
      term, term+
    }+
  }

term =
  element term {
    attribute lemID { xsd:IDREF }?,
    xsd:string?
  }

sig =
  element sig {
    attribute measure { xsd:string }?,
    xsd:float
  }
```

Error.rnc

```
default namespace = "http://www.dspin.de/data/error"
```

```
start =
  element error {
    element type {
      attribute code { xsd:int },
      xsd:string
    },
    element message { xsd:string }
  }
```