**Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH**

**Technical
Memo**

TM-99-01

SMESPR/1.0

Matthias Fischmann

January 1999

Deutsches Forschungszentrum für Künstliche Intelligenz GmbH

Postfach 20 80
67608 Kaiserslautern, FRG
Tel.: + 49 (631) 205-3211
Fax: + 49 (631) 205-3210

Stuhlsatzenhausweg 3
66123 Saarbrücken, FRG
Tel.: + 49 (681) 302-5252
Fax: + 49 (681) 302-5341

The Smes Client/Server Protocol (SMESPR/1.0)

Matthias Fischmann

DFKI-TM-99-01

This work has been supported by a grant from The Federal Ministry of Education, Science, Research and Technology (FKZ ITWM-01IW809).

ISSN 0946-0071

The Smes Client/Server Protocol (SMESPR/1.0)

Matthias Fischmann <fischman@dfki.de>

January 21, 1999

Abstract

SMESPR is a robust, efficient, flexible and platform/language independent protocol that allows the smes kernel to export its functionality over any sort of computer networks (in particular TCP/IP). It allows for results in different formats (HTML/XML, SQL and plain feature structures are available, others may be easily added) as well as for creating and controlling graphical user interfaces from the interior of the smes world. Furthermore, any sort of access or privacy policy can be plugged into the existing protocol very easily. A prototype implementation is in use at the DFKI in several projects.

This article contains the documentation of SMESPR/1.0 and out prototype implementation.

1 Introduction

The smes system developed in the PARADIME project [1][2] is a framework providing solutions for a heterogenous set of linguistic analysis tasks. To make it available on multiple platforms and in different environments with minimal amount of changes in the kernel, it is convenient to export the functionality over a network interface. This allows us to use the prototype written in allegro common lisp[1] and incrementally recode it in C/C++, while the client software works independently in perl/Tk[2] and Java[3].

SMESPR/1.0 is a protocol designed for an arbitrary text processing system[4] that wants to provide its functionality over a network. This suggests that a connected client is to submit a query consisting of a text together with information about the desired kind of computation and about the format of the result. For instance, a query could contain the information "invoke the function `parse-from-string` on the sentence 'Der Umsatz von Siemens wuchs im letzten Quartal um 13 Prozent.' and return the result as an HTML fragment to be browsed in a web client.".

A *session* (the time between connection and disconnection of a client) basically consists of a query-response loop: after the client has submitted a query, the server computes it, returns the result, and then waits for the next query.

This paper documents the protocol and its prototype implementation consisting of a server package for the franz lisp version of smes and a client with both shell facilities and a graphical user interface (written in perl/Tk).[5] We provide a programmer's manual, an abstract specification.

1.1 Contents of this paper

In section 2, we will describe how our libaries are used to write perl clients and to build new lisp server images that extend the basic functionality. To keep the system portable and language independent, we provide an abstract specification of the underlying protocol in section 3 that is based on abstract network streams. Section 4 concludes the paper.

[1] Allegro CL is a product of Franz Inc.

[2] perl is a scripting language available from http://www.perl.org/

[3] Java is a registered trademark of Sun Microsystems, Inc.

[4] Although it is particularly designed for the smes framework, other related projects may benefit from the work we present here as well.

[5] The alpha-version of a java-client is provided with the prototype distribution as well, but it is still experimental and we will not consider it here.

1.2 Acknowledgements

This paper and the corresponding implementation would not have been possible without the efforts of Peter Rullmann, who wrote most of the parac code, and the feedback of Thierry Declerck, Judith Klein, Sven Schmeier, Markus Becker, Feiyu Xu, Christian Braun and others. Most importantly, Günter Neumann has come up with the idea of providing internet access to the smes system in the first place, and was responsible for coordinating the design phase and the integration of the network libraries into the smes kernel.
The `ParaClient` object makes use of the `RecDescent` Parser package by Damian Conway to process the configuration data.

2 Using the paraserv libraries

Before we give a formal description, we will approach the problem from two sides. The next section contains a tutorial on how to write perl scripts that contact a running smes server using the `ParaClient` perl module[6]. Then, we give an introduction to setting up a new server with the lisp package `paraserv`.

2.1 Writing Client Scripts in Perl

Before we start, make sure that you have the SMESPR/1.0 prototype distribution installed on your system and a public server is running somewhere in the net.
`ParaClient` is a perl class that encapsulates the job of maintaining the connection, generating query strings and parsing the response to obtain the result. Initialization and opening the connection is straight forward:

```
use ParaClient;

my $verbose = 1000;
  # this is extremely noisy. 5 is a little more moderate.

my ($host, $port) = ('limit.dfki.uni-sb.de', 23440);
  # or wherever your server is.

my $timeout = 30;
  # seconds. this is the default value, so you could
  # skip this. on slow connections, it may be necessary
  # to increase this to 60 or even higher.

my $client = ParaClient->new($verbose, $host, $port, $timeout);

if ($client->open()) {

  # ... this your program...

} else {
  die "connection to $host:$port failed";
}
```

Next, we need to encode everything we know about the server method we want to invoke in a method specification string. In particular, the name of the method, the names and values of the arguments, and the result type need to be fixed. Have a look at the following example:

[6]This module is used in the parac program coming with the distribution as well.

```
my $method = '(smes::fst-from-string
                 ((:fst . :np-star) (:trace . t))
                 ("text/html")
               )';
```

The second list element represents the arguments passed to the method using a lisp-style association list: every argument is represented by a dotted lisp list mapping the keyword argument name to its value (see [3] to learn more about dotted lists).
Taken a list of method specifications `ms` (every method will be invoked, in order of appearance in the list) and a string `$s` containing the text in question, a query is submitted as follows.

```
if ($client->query($s, @ms) > 0) {

  # success! the server will now respond.
  # ...

} else {
  die "failed to submit query";
}
```

There are two levels of abstraction the response can be retrieved on. (i) if interested in the pure result only, use `response`; (ii) in need of sophisticated control mechanism to obtain debug and progress reports from the server during the computation, use the primitive commands `readline` and `readpackage`.
To understand (ii), it is necessary to know more about the protocol first. Therefore, we do not talk about this option here. (i), however, is powerful enough to obtain the result; all other output coming from the server will be dumped to stderr (where you could read it by rebinding `STDERR`). This is how it works:

```
print STDERR "\n[reading response]\n\n";
my $pkg = $client->response();

if (defined $pkg) {
  print STDERR "\n[done]\n\n";
  my $code = $pkg->{'code'};
  my $contents = $pkg->{'contents'};
  print STDOUT $contents;
} else {
  die "could not read package";
}
```

The code cited in this section is contained in the sample program parascript coming with the distribution.

2.2 Customizing the Lisp Server

To learn how to use the smes system, see the manual coming with the smes distribution.
The package `paraserv` (nickname `ps`) provides SMESPR/1.0 compliant server functionality to the allegro common lisp system (using the `defsystem` package developed by Mark Kantrowitz in 1991). It contains a function `dump-server` that creates a lisp image that can be run as a server in the background. This means that creating a new server image is as easy as typing the following line to the prompt of a lisp image containing the desired functionality:

```
(setf ps::*specs*
  `((smes::morph-from-string "Morphology"
      ((:trace "Debug Mode"
        ,#'(lambda (x) (not (member x '(t nil))))
        (:boolean)
       )
      )
      (("text/plain" . nil))
    )
    (smes::fst-from-string "Phrasal Chunker"
      ((:fst "Select Subgrammar"
pp          ,#'(lambda (x) (not (member x
                                  (list :all-frags
                                        :firmen-treiber
                                        :main
                                        :np-star))))
          (:menu (:all-frags . "All Subgrammars")
                 (:firmen-treiber . "Company names")
                 (:main . "NP and PP grammar")
                 (:np-star . "NP*"))
        )
      )
      (("text/plain" . nil)
       ("text/html" . ,#'smes::format-fst-from-x-to-html-string))
    )

    ...

  )
)
```

Figure 1: Description of the server methods of the prototype server.

```
(load-system ''paraserv'') (dump-server)
```

However, it is necessary to calibrate `paraserv` before we get there. The smes methods (or lisp functions exported to the network) must be explicitely defined, and the overall behaviour of the server (locations of logfiles, verbosity, network ports etc.) might have to be adapted to individual needs.

2.2.1 Describing Lisp Functions

`ps::*specs*` describes the behaviour of the server. Before we define the structure of this rather complicated variable, have a look at a part of the standard server's `*specs*` variable in figure 1. There are two methods, one called `morph-from-string` and the other one `fst-from-string`. For each method, a real name ("Morphology" or "Phrasal Chunker", respectively), description of accepted parameters and the set of supported result types are mentioned.
In general, `ps::*specs*` contains a list of arbitrary many method descriptions. Each description is in turn a list that contains the following pieces of information (in this order).

- The name of the function to be declared as a smes method. This is a lisp symbol, the package name defaults to `ps`.
- A string representing the external name of the method. This keeps the user away from names like `smes::scan-from-string`, and gives him something like "Tokenizer" instead.

- A list of four element lists of the form (⟨key⟩ ⟨name⟩ ⟨test⟩ ⟨kind⟩). Each such list represents a keyword argument that is passed to the function when it is called. It consists of
  - A keyword ⟨key⟩.
  - Some string ⟨name⟩ containing the name passed to the user. (As with the user name of the method, we want to hide unintuitive internal names from the smes user.)
  - A function ⟨test⟩ that checks whether the value provided to the keyword is acceptable. Returns nil if an argument is acceptable, and some text or arbitrary lisp expression that is interpreted as an error message otherwise.
  - A list ⟨kind⟩ containing information to allow the client to build specific input widgets. It may be one of the following.
    * `(:string` ⟨width⟩ ⟨height⟩`)`. ⟨width⟩ and ⟨heigth⟩ describe the size a text field should have that reads the argument value (this is a guideline and may be ignored by the client; it just makes it easier to guess an appropriate size).
    * `(:menu (`⟨lisp-item⟩ `.` ⟨name⟩`) ...)`. Again, we export a lisp object together with ist user-visible name.
    * `(:boolean)`. Does not need any arguments. It is syntactic sugar for `(:menu (t . ''on'') (nil . ''off''))`.
    * `(:buttons (`⟨lisp-item⟩ `.` ⟨name⟩`) ...)`. Same as `:menu`, but allows for multiple choices.
- An assoc list mapping the name of every legal result type to the function that generates it. For instance, `(''text/html'' . #'convert)` attaches the output type "text/html" to the lisp function `convert`. This function will be called with `(convert` ⟨raw-values⟩ ⟨user-args⟩`)`, where ⟨raw-values⟩ is the list of values returned by the smes method, and ⟨user-args⟩ is extracted from the query to allow the user calibrate the output function depending on a particular needs arising during the session (see there).

  If the function is set to nil, a `(format nil ''~S'' (car` ⟨result⟩`))` will be used instead (this is generally used for "text/plain" type results).

2.2.2 Other Server Switches

Server switches allow calibration of the server behaviour concerning logfiles, verbosity etc. They are implemented using special variables. In particular, the following variables may be changed by the user:

- `*logfile*`. Path and filename of the (only) server logfile. Defaults to "~/log/paraserv.log".
- `*server-image-name*`. The name of the server image to be created by `dump-server`. Defaults to "~/bin/paraserv".
- `*verbosity*`. The size of the debug trace produced in the logfile. 0 means silent (only hard error messages are still dumped), 3 dumps connection data and the current session phase, 5 is moderately talkative, 8 quite unreasonable and 10 dumps every query in full and a lot more.
- `*default-port*` The socket port the server is listening on. `23440` by default.
- `*mail-errors*`. List of error codes that cause the server to send a mail when occuring. Defaults to `nil`.
- `*mail-to*`, `*mail-from*`, `*mail-reply-to*`. If a mail is sent, these header field entries are used. They all default to `''smes@dfki.de''` and should probably be changed if you change `*mail-errors*`.
- `*mail-subject*`. Defaults to `''<!>''`.
- `*protocol-version*` `''SMESPR/1.0''` by default. This is interesting for future extensions only.

2.3 Writing lisp functions that are smes methods

So far, we have have assumed that the smes methods already exist as lisp functions, and we described how to export them. In this section, we define the constraints such a lisp function must satisfy to be sucessfully called by the server. Furthermore, we document the concepts of sending output and debug information to the client.

2.3.1 Invocation

A kernel function that may be called by the server module must have the following form:

`(name` ⟨text⟩ `:key`$_1$ ⟨val$_1$⟩ `:key`$_2$ ⟨val$_2$⟩ `...)`

⟨text⟩ is the corpus string extracted from the query, while ⟨val$_n$⟩ are arbitrary lisp expressions. All keywords are optional. If a keyword is not mentioned in the query, an apropriate default value must be available.

2.3.2 Producing Output

SMESPR/1.0 offers three ways for a smes kernel function to produce output:

- Write on stdout with `#'format`. This kind of output is directly sent to the client and may appear in a window called "standard output", or on the terminal.
- Return one or more values. These values are converted to the desired result type automatically and sent to the client as the final result of a computation.
- Use `#'send-package` to send a user type package:

  `(send-package` ⟨number⟩ ⟨contents⟩`)`

  ⟨number⟩ tags the string ⟨contents⟩ to be of some particular meaning. Customized client software may react by interpreting this string in the desired way (pop up or update a progress bar, visualize partial results or the like). It is an error for ⟨number⟩ to be 0, 1 or 2 (these package numbers are hard wired in the protocoll). See section 3 for more information about packages.

3 The Protocol

This section contains a quasi-formal SMESPR/1.0 specification. We assume that, two parties, namely the client **C** and the server **S**, already have initialized a bidirectional network stream that can be used by both parties both for reading and writing. We regard the process of establishing this stream as atomic, i. e. we do not care about the implementation here.
A session starts with the stream being established and ends with it being closed. It is split into three phases, namely the *handshake phase*, the *query phase*, and the *response phase*, the latter two subsumed under the name *main loop*.

3.1 Handshake and Entering the Main Loop

After the stream has been established, **S** initializes the session by generating a unique ID and sending it over the stream as a line (a `0x0a`-terminated string). This ID may be used for matching **S**' logfile entries to error reports from the user for trouble shooting.
C sends an acknowledgement that contains three lines:

- An echo of the ID just received.
- The version of SMESPR she wishes to use for communication. This is currently `SMESPR/1.0`.
- A name of the authentication and/or encryptions protocol. This is `NONE` by default, since we don't care about security at the moment.

```
!query! 21 !query!
(
  (text . "bla grahg.
may be more than one line.
")
  (methods .
    ((:fst-from-string
       (
         (:fst . :main)
         (:trace . t)
       )
       ("text/html" . nil)
     )
     (:fst-from-string
       (
         (:fst . :firmen-treiber)
       )
       ("text/plain" . nil)
     )
    )
  )
)
```

Figure 2: An example query.

If these three lines have been sent, **S** sends an **ack** package or an error (see section 3.3.1 for more information on packages). After this initial acknowledgement, **C** and **S** may enter some cryptographical conversation if desired. (By default, nothing happens here.)
Finally, the session enters the mainloop. **C** submits a query (*query phase*) and receives a response (*response phase*). This may be iterated arbitrarily often. The session is terminated by **C** closing the stream.

3.2 The Query Phase

Every query consists of a one-line header and a body. The header contains the number of lines of the complete query. (See figure 2 for an example.)

```
QRY_HEADER ::= '!query! ' NUM_OF_LINES ' !query!' 0x0a
```

We define the strings going over the stream by means of BNF fragments. `'quoted'` words are terminals (or just strings to be copied literally), CAPITALIZED words are non-terminals to be defined in more detail (if the definition is obvious as in the case of `NUM_OF_LINES`, it is omitted). '+' and '*' are used as usual. `STRING` is a text enclosed in double quotes (""). Whitespaces outside strings and Lisp syntax comments are ignored. All characters outside strings are case insensitive. Furthermore, definitions from the common lisp standard [3] hold if this document is ambigous or silent.
The body of a query contains a list of two-element dotted lists in lisp syntax (sometimes referred to as assoc list) that are interpreted as key-value pairs.

```
QUERY       ::= '(' ENTRY+ ')'

ENTRY       ::= '(' 'text' '.' STRING ')'
              | '(' 'methods' '.' '(' EXP+ ')' ')'
```

There are only two valid `ENTRY` types, namely `'text'` and `'methods'`. The `'method'` key contains a non-empty list of `EXP`s each of which describes smes method invokation and contains information about the output type.

```
EXP   ::= '(' NAME '(' ARG* ')' TYP ')'
        | '(' NAME NIL TYP ')'

NIL   ::= 'NIL' | '()'

TYP   ::= '(' T_NAM ')'
        | '(' T_NAM '.' THING ')'

T_NAM ::= STRING

ARG   ::= '(' KEY '.' VALUE ')'
```

For each `EXP` in a query, the server calls the lisp function `NAME` with the specified `ARG`s as keyword arguments. The syntax will be transformed into `(SMES:NAME ''some text'' :key`$_1$ `val`$_1$ `:key`$_2$ `val`$_2$ `...)`.
CAUTION: arbitrary lisp expressions present a severe security gap, since lisp is powerful enough to execute arbitrary code on the server host machine even when only parsing them. Implementors are encouraged to introduce proprietary restrictions that fit their server methods as tight as possible.
`T_NAM` contains the type of the response and `THING` may contain additional information in future extensions. At the moment, it is always `NIL`, i. e. the first case of `TYP` always holds.
Supported result types are:

- `''text/plain''`: yields the output of the kernel functions without any postprocessing. This is obviously supported by all methods and should be the default value.
- `''text/html''`: yields an HTML fragment that could be browsed when wrapped in a header and a footer (the fragment starts and ends in the body of a well-formed HTML document). Might be undefined on some of the smes methods.
- `''application/x-sql''`: this is a more sophisticated result-type, but it is still hard-wired in the protocol to demonstrate the flexibility of the system. It returns a sequence of SQL `insert` statements that can be fed into an SQL database. The types of the tables used in the statements depend on the specified method.

Implementors are free to introduce further result types. If **C** uses an extension **S** does not know of, an error 3 is returned (see section 3.3.1 below).
The query phase ends with **S** analysing the query. If satisfied, an **ack** package is launched to initialize the response phase. Otherwise, an **error** is signalled to reject the query. In this case, it is **C**'s turn to submit a new query or to disconnect.

3.3 The Response Phase

What now follows is a sequence of `stdout` lines and packages that come from **S** to **C**. For each method invokation, exactly one package of type 0 or 1 is launched to indicate its termination. The response phase is terminated by an additional **ack** package. (This is redundant, but useful to deal with internal server errors that make **S** miscount the number of method calls.)
Standard output of smes methods is passed to **C** line by line as it comes into existence. It may be forwarded to the client's terminal, or browsed in a window. It usually contains progress reports, incrementally obtained results, or debugging information.
Additionally, **S** can launch a package to provide more specific information like final results, control expressions for a graphical user interface, or error messages that must be parsed by the client.

The notion of packages has already been used extensively up to here. We will now define it properly.

3.3.1 Packages

Like a query, a *package* consists of a header and a body containing `NUM_OF_LINES` further lines. Unlike a query, the header contains the package type in addition to the size of the body. There are three predefined package types: **error** (0), **result** (1), and **ack** (2). Packages of higher type are called *user packages*.

```
PKG_HEADER ::= '!package! ' ID ' ' NUM_OF_LINES ' !package!' 0x0a
```

All other output that is sent over the connection from **S** to **C** must be blocked and buffered until the package transmission is completed. Also, **S** must ensure that all lines not contained in a package that match `PKG_HEADER` actually *are* package headers. Standard output may have to be filtered not to contain such lines.
A detailed description of the three predefined package types follows.

- **error: 0**

  The first line of an error package starts with an error number, followed by a colon (':', or `0x3a`), a space (`0x20`), and a plain error message. There may be more lines containing arbitrary text.

  Predefined errors are

  1 unknown method `METHOD`
  2 illegal arguments for method `METHOD`: `ARGS`
  4 result type `T_NAM` not supported by method `METHOD`
  5 internal error
  6 internal error in method `METHOD`
  7 internal error while converting result to type `T_NAM`
  9 parse error (illegal query)
  14 `ID` code `YOURS` differs from `MINE`
  15 illegal protocol version `VERSION`
  16 illegal authentication protocol `PROTOCOL`
  18 unknown package type `NUMBER`
  19 no connection

  18 and 19 are exceptions in that they are not producded by **S** but locally by **C**. Anyway, they are useful enough to require every client to support them.

  User defined errors have numbers greater than 255, and may be freely introduced by the implementor. **C** has to handle unknown error messages in some reasonable way (return the description, or the like).

- **result: 1**

  This package contains the final result of one method call. The value of the executed lisp function is feed into the pretty printer function and the output (that is always a string and of the chosen result type) is delivered by means of the result package.

  Result packages are always returned in the order the method invokation requests are listed in the query.

- **ack: 2**

  Usually, type 2 packages do not contain any particular information. They end initial handshake and query/response phases. However, as they are ordinary packages, they may very well be used for proprietary extensions (which are then simply ignored by standard clients).

- **Defining new package types**

  New package types may be freely introduced at any time. If **C** does not know the type of a received package, an error 18 occurs.

3.4 Transmission of Service Information

The `*specs*` variable introduced in section 2 contains a certain amount of information that is interesting for the client only. This information can be requested by **C** sending a special one-line query consisting of the string `'!configure!'`. This query causes the server to send a single result package containing the information represented by `*specs*` in a slightly modified way.
This package is a proprietary extension of the prototype implementation, but might be added to the standard in the future. For more details, consult the source code or just dump a configuration package to a file and look at it by hand (the differences to `*specs*` are rather superficial).

4 Conclusion

SMESPR is defined on a level of abstraction that leaves room for efficient implementations, by allowing to choose an arbitrary stream model (TCP/IP sockets, the package oriented UDP/IP protocol, etc.). This hopefully makes it useful not only for the prototype in lisp, but also for industrial quality versions to come.
The stream model of our prototype is TCP/IP, although SMESPR is strictly package oriented, and the package oriented UDP protocol is in general more efficient. This is because sockets are more easy to write and maintain and (most importantly) available in virtually every programming language that exists, and thus more flexible in the design and development phase. On the other hand, this slows things down a little. We still don't have any reliable information on the efficiency of the system. We expect it to be easy to write fast implementations using efficient network technology and programming languages, but the acl server seems to have a constant connection and handshake cost of a few seconds. This is not as severe as one might think, since smes is designed to perform expensive computations that take a much longer time. However extensive benchmarks and optimizations must be considered next.

References

[1] G. Neumann, R. Backofen, J. Baur, M. Becker, C. Braun, *An information extraction core system for real worl german text processing*, in *5th International Conference of Applied Natural Language*, p. 208–215, Washington, USA, 1997.

[2] G. Neumann, G. Mazzini, *Domain-adaptive Information Extraction*, DFKI Report, to appear.

[3] Guy L. Steele jr., *Common Lisp – The Language*, 2nd edition, digital press.

SMESPR/1.0

Matthias Fischmann

TM-99-0
Technical Mem