SCSCP

Symbolic Computation Software Composability Protocol

Version 1.1.4

31 May 2009

Alexander Konovalov
Steve Linton

Alexander Konovalov — Email: `alexk at mcs dot st-andrews dot ac dot uk`

— Homepage: `http://www.cs.st-andrews.ac.uk/˜alexk/`

— Address: School of Computer Science
University of St Andrews
Jack Cole Building, North Haugh,
St Andrews, Fife, KY16 9SX, Scotland

Steve Linton — Email: `sal at cs dot st-andrews dot ac dot uk`

— Homepage: `http://www.cs.st-andrews.ac.uk/˜sal/`

— Address: School of Computer Science
University of St Andrews
Jack Cole Building, North Haugh,
St Andrews, Fife, KY16 9SX, Scotland

Abstract

The GAP package SCSCP implements the Symbolic Computation Software Composability protocol (http://www.symbolic-computation.org/scscp) for the computational algebra system GAP.

Copyright

Acknowledgements

Colophon

Versions history:

- Version 0.1 - first half of 2007;
- Version 0.2 - December 2007;
- Version 0.3 - May 2008;
- Version 0.4 - August 2008;
- Version 1.0 - March 2009;
- Version 1.1 - May 2009.

Contents

Chapter 1

Preface

The GAP package SCSCP implements the Symbolic Computation Software Composability protocol [FHK$^+$b]. This protocol specifies an OpenMath-based remote procedure call framework, in which all messages (procedure calls and returns of results of successful computation or error messages) are encoded in OpenMath using content dictionaries scscp1 and scscp2 ([FHK$^+$a], [FHK$^+$c]). Using the SCSCP package, GAP can communicate locally or remotely with any other OpenMath-enabled SCSCP-compliant application which may be not only another computer algebra system but also another instance of the GAP system or even, for example, an external Java or C/C++ application via libraries http://java.symcomp.org/ or http://www.imcce.fr/Equipes/ASD/trip/scscp/ providing an SCSCP API. Such communication will go into seamless manner for the GAP user, since all conversions from GAP to OpenMath and vice versa will be performed in the background. See the SCI-Ence project homepage http://www.symbolic-computation.org/ for the details about computer algebra systems and other sotware supporting SCSCP

The SCSCP package for GAP has two main components:

- SCSCP server;

- SCSCP client.

There are several ways to start GAP SCSCP server:

- call `RunSCSCPserver` (5.2.1) from the GAP session specifying the server name and the port number from the GAP session;

- start GAP as `gap myserver.g`, where `myserver.g` is the server configuration file with the last command being the call of `RunSCSCPserver` (5.2.1) (an example of such configuration file is given in `scscp/example/myserver.g`);

- start GAP as a daemon using the script `gapd.sh` which is supplied in the root directory of the package (for the description of all available options see comments in `gapd.sh`).

During startup the server installs all procedures that it will provide and loads their lookup mechanisms, and then begins to listen to the specified port. The recommended port number is 26133 which has been assigned to SCSCP by the Internet Assigned Numbers Authority (IANA) in November 2007, see http://www.iana.org/assignments/port-numbers.

When the server accepts a connection from client, it starts the "accept-evaluate-return" loop:

- accepts the `"procedure_call"`; message;

5

- performs lookup of the appropriate GAP function;

- evaluates the result (or produces a side-effect);

- returns the result in the `"procedure_completed"` message or returns an error in the `"procedure_terminated"` message.

The server works in a "multi-user" mode. When one client is connected, the server is busy for other clients. As soon as the computation is finished and the client is disconnected, the server is waiting for the next connection, and normally it never stops until it will be terminated by the service provider. The server maintain a queue of five incoming connections (this parameter can be easily modified), and on each iteration evaluates the next request from the queue.

There is an SCSCP server accessible at `chrystal.mcs.st-andrews.ac.uk`, port 26133. It is running under development versions of the GAP system and a selection of currently distributed packages. The reader is encouraged to try to use examples from the manual to access this service, replacing `"localhost"` by its address, where appropriate. Please report to Alexander Konovalov if you will discover any bugs or if the server seems not available.

The SCSCP client:

- establishes connection with the specified server at the specified port;

- sends the `"procedure_call"` message to the server;

- waits for the result of the computation or returns to pick it up later;

- fetches the response, extracting the result from the `"procedure_completed"` message or entering the break loop in the case of the `"procedure_terminated"` message.

On the top of this functionality we built a set of instructions for simple parallel computations framework using the SCSCP protocol, which allows to send several procedure calls in parallel and then collect all results or pick up the first available result, and implements the master-worker skeleton. These tools are presented in the Chapter 8.

The package also implements a new kind of GAP input-output streams, namely input-output TCP streams (see Chapter 3), based on the functionality for TCP/IP protocol usage provided by the GAP package IO. Such streams may constitute an independent interest for adapting streams-using GAP code to use streams across the network.

Finally, the manual describes how the communication by SCSCP goes between several instances of the GAP system, but the same behaviour is expected from any SCSCP-compliant application: the set of supported OpenMath symbols clearly will be different, but the rules of communication are precisely specified in the SCSCP specification [FHK$^+$b]. See the homepage of the SCIEnce project `http://www.symbolic-computation.org/` for the information about SCSCP-compliant computer algebra systems and other tools developed in the project.

Chapter 2

Installation

2.1 Installation and system requirements

The SCSCP client for GAP is fully functional under GAP 4.4 and works in UNIX/Linux environments, Mac OS X (UNIX installation) and MS Windows.

The SCSCP server for GAP is fully functional with the GAP development version and goes automatically into the compatibility mode to work with GAP 4.4.12 and earlier versions. The only limitation of this compatibility mode is that in the case of an error the break loop occurs on the server and can not be transmitted to the client (however, if the service consumer is the service provider himself/herself, then this is not as crucial as it might be in the general case). After the GAP 4.5 release the package will fully be compatible with the official GAP releases.

To use the SCSCP package it is necessary to install recent versions of GAP4 packages IO [Neu], GAPDoc [LN] and OpenMath [CKS].

The SCSCP package is distributed in standard formats (`tar.gz`, `tar.bz2`) and can be obtained from http://www.cs.st-andrews.ac.uk/~alexk/scscp.htm or from the GAP web site (the latter also offers `zoo`- and `win.zip`-archives. To unpack the `zoo`-archive the program `unzoo` is needed, which can be obtained from the GAP homepage http://www.gap-system.org/ (see section 'Distribution'). To install SCSCP package, put its `zoo`-archive into the `pkg` subdirectory of your GAP4.4 installation and enter the command `unzoo -x scscp-X.X.X.zoo`, then the subdirectory `scscp` (containing subdirectories `doc`, `lib` etc.) will be created in the `pkg` subdirectory. Installation using other archive formats is performed in a similar way.

When there are no access rights to the root directory of the main GAP installation, it is also possible to install the package *outside the GAP main directory* by unpacking it inside a directory `MYGAPDIR/pkg`. Then to load the package GAP should be started with `-l ";MYGAPDIR"` option.

2.2 Configuration files

There are four files in the package which may need to be modified to setup and customise the package. The first three files are related with the server's functionality:

- `scscp/config.g` specifies:

    - default `InfoLevel` for the `InfoSCSCP` (9.2.3) class;
    - default SCSCP server name and port to be used by `RunSCSCPserver` (5.2.1) if GAP is started with the `scscp/example/myserver.g` file;

- – whether the server accepts calls to procedures which are standard OpenMath symbols, or only procedures installed in the transient content dictionary (see `InstallSCSCPprocedure` (5.1.1));

- – service description to be returned to the client by `GetServiceDescription` (5.3.1).

- `scscp/gapd.sh` is the script to start the GAP SCSCP server as a daemon. To use it, adjust the local call of GAP and, if necessary, call options (for example, memory usage, startup from the workspace etc.) and the location of the root directory of the SCSCP package in section 1 of this script.

- `scscp/example/myserver.g` is an example of the server configuration file which loads all necessary packages, reads all needed code, installs all procedures which will be exposed to the client and finally starts the SCSCP server (see Chapter 5).

The fourth file is related with the client's functionality for parallel computations:

- The file `scscp/configpar.g` assigns the global variable `SCSCPservers` which specifies a list of hosts and ports to search for SCSCP services (which may be not only represented by GAP services, but also by another SCSCP-compliant systems). It will be used to run parallel computations with the SCSCP package (see Chapter 8).

See comments in these configuration files for further details and examples.

Chapter 3

Using streams

The package implements new kind of GAP input-output streams, called input-output TCP streams. Such streams are based on the functionality for the TCP/IP protocol usage provided by the GAP package IO, and may constitute an independent interest for GAP users.

Input-output TCP streams are intended to support all operations, implemented for streams in GAP. It is assumed that all existing code using streams should work with this kind of streams as well (please let us know, if you will notice that this is not the case!). We installed methods for input-output TCP streams to support the following operations: `ViewObj`, `PrintObj`, `ReadByte`, `ReadLine`, `ReadAll`, `WriteByte`, `WriteLine`, `WriteAll`, `IsEndOfStream`, `CloseStream`, `FileDescriptorOfStream`, `UNIXSelect`.

3.1 Input-output TCP streams

3.1.1 IsInputOutputTCPStream

◇ `IsInputOutputTCPStream`   (filter)

`IsInputOutputTCPStream` is a subcategory of `IsInputOutputStream`. Streams in the category `IsInputOutputTCPStream` are created with the help of the function `InputOutputTCPStream` (3.1.4) with one or two arguments dependently on whether they will be used in the client or server mode. Examples of their creation and usage will be given in subsequent sections.

3.1.2 IsInputOutputTCPStreamRep

◇ `IsInputOutputTCPStreamRep`   (filter)

This is the representation that is used for streams in the category `IsInputOutputTCPStream`.

3.1.3 InputOutputTCPStream (for server)

◇ `InputOutputTCPStream(socket_descriptor)`   (function)

Returns: stream

Returns a stream in the category `IsInputOutputTCPStream` for the given socket descriptor that will be used to accept incoming connections. An example demonstrating this will be given after the next section.

However, the service provider much likely will create such stream automatically inside the function `RunSCSCPserver` (5.2.1) rather then manually.

3.1.4 InputOutputTCPStream (for client)

◊ `InputOutputTCPStream(`*hostname, port*`)` (function)

    Returns: stream

Returns a stream in the category `IsInputOutputTCPStream` that will be used by the client for communication with the specified server and port.

3.2 Example of client-server communication via input-output TCP streams

The following example demonstrates an interaction between client and server using input-output TCP stream, and shows how such streams are created in the function `RunSCSCPserver` (5.2.1). It uses some functions from the IO package, see the IO manual for their description. Now we will show step by step what is happens on server and client (of course, if you will try this example, the numbers denoting descriptors may be different).

    Firts, we will start two GAP sessions, one for the server, another one for the client. Now we enter the following commands on the server's side:

```
——————————————— Example ———————————————
gap> sock := IO_socket( IO.PF_INET, IO.SOCK_STREAM, "tcp" );
3
gap> lookup := IO_gethostbyname( "localhost" );
rec( name := "localhost", aliases := [  ], addrtype := 2, length := 4,
  addr := [ "\177\000\000\>" ] )
gap> port:=26133;
26133
gap> res := IO_bind( sock, IO_make_sockaddr_in( lookup.addr[1], port ) );
true
gap> IO_listen( sock, 5 );
true
gap> socket_descriptor := IO_accept( sock, IO_MakeIPAddressPort("0.0.0.0",0) );
```

After the last command you will not see the GAP prompt because the server starts to wait for an incoming connection. Now we go to the client's side and create an input-output TCP stream to the server. Here it can be created in one step:

```
——————————————— Example ———————————————
gap> clientstream:=InputOutputTCPStream( "localhost", 26133 );
Creating a socket...
Connecting to a remote socket via TCP/IP...
```

Now we are trying to connect to the server, and as soon as the connection will be established, the stream will be created at the client side, and we will see the output and the new GAP prompt:

```
 ──────────────── Example ────────────────
 < input/output TCP stream to localhost >
 gap>
```

On the server you will get the socket descriptor and then you will be able to create a stream from it:

```
 ──────────────── Example ────────────────
 4
 gap> serverstream := InputOutputTCPStream( socket_descriptor );
 < input/output TCP stream to socket >
```

Now we can write to this stream on the client side and then read from it on the server side and backwards. First, write on the client:

```
 ──────────────── Example ────────────────
 gap> WriteLine( clientstream, "12345" );
 true
```

Now read and write on the server:

```
 ──────────────── Example ────────────────
 gap> ReadLine( serverstream );
 "12345\n"
 gap> WriteLine( serverstream, "54321" );
 true
```

And finally we read on the client and close the stream:

```
 ──────────────── Example ────────────────
 gap> ReadLine( clientstream );
 "54321\n"
 gap> CloseStream( clientstream );
```

and similarly close the stream on the server:

```
 ──────────────── Example ────────────────
 gap> CloseStream( serverstream );
```

Chapter 4

Message exchange by SCSCP

To ensure the message exchange as required by SCSCP specification, the SCSCP package extends the global record `OMsymRecord` from the OpenMath package with new entries to support scscp1 and scscp2 content dictionaries ([FHK⁺a], [FHK⁺c]), and also service-dependent transient private content dictionaries (see 5 for details about transient content dictionaries). It also overwrites some OpenMath functions by their extended (but backwards compatible) versions, and adds some new OpenMath-related functions to send and receive SCSCP messages, documented below.

Note that functions documented in this chapter belong to the middle-level interface, and the user may find it more convenient to use functions developed on top of them and explained in next chapters.

4.1   Communication with the SCSCP server

4.1.1   StartSCSCPsession

◊ StartSCSCPsession(*stream*)   (function)
    Returns: string

Initialises SCSCP session and negotiates with the server about the version of the protocol. Returns the string with the service_id (which may be used later as a part of the call identifier) or causes an error message if can not perform these tasks.

```
─────────────────────── Example ───────────────────────
 gap> s := InputOutputTCPStream("localhost",26133);
 #I  Creating a socket ...
 #I  Connecting to a remote socket via TCP/IP ...
 < input/output TCP stream to localhost:26133 >
 gap> StartSCSCPsession(s);
 #I  Got connection initiation message
 #I  <?scscp service_name="GAP" service_version="4.dev" service_id="localhost:2\
 6133:5541" scscp_versions="1.0 1.1 1.2 1.3" ?>
 #I  Requesting version 1.3 from the server ...
 #I  Server confirmed version 1.3 to the client ...
 "localhost:26133:5541"
```

After the call to `StartSCSCPsession` the SCSCP server is ready to accept procedure calls.

4.1.2 OMPutProcedureCall

◊ OMPutProcedureCall(*stream, proc_name, objrec*)   (function)

Returns: Nothing

Takes a stream *stream*, the string *proc_name* and a record *objrec*, and writes to *stream* an OpenMath object `procedure_call` for the procedure *proc_name* with arguments given by the list `objrec.object` and procedure call options (which should be encoded as OpenMath attributes) given in the list `objrec.attributes`. This function accepts the option `cd:="cdname"` to specify the name of the content dictionary if the procedure is actually a standard OpenMath symbol note that the server may reject such a call if it accepts only calls of procedures from the transient content dictionary, see `InstallSCSCPprocedure` (5.1.1) for explanation). If the `cdname` is not specified, `scscp_transient_1` content dictionary will be assumed by default.

```
──────────────────────── Example ────────────────────────

gap> t:="";; stream:=OutputTextString(t,true);;
gap> OMPutProcedureCall( stream, "WS_Factorial", rec( object:= [ 5 ],
>       attributes:=[ [ "call_id", "user007" ],
>                      ["option_runtime",1000],
>                      ["option_min_memory",1024],
>                      ["option_max_memory",2048],
>                      ["option_debuglevel",1],
>                      ["option_return_object"] ] ) );;
gap> Print(t);
<?scscp start ?>
<OMOBJ>
        <OMATTR>
                <OMATP>
                        <OMS cd="scscp1" name="call_id"/>
                        <OMSTR>user007</OMSTR>
                        <OMS cd="scscp1" name="option_runtime"/>
                        <OMI>1000</OMI>
                        <OMS cd="scscp1" name="option_min_memory"/>
                        <OMI>1024</OMI>
                        <OMS cd="scscp1" name="option_max_memory"/>
                        <OMI>2048</OMI>
                        <OMS cd="scscp1" name="option_debuglevel"/>
                        <OMI>1</OMI>
                        <OMS cd="scscp1" name="option_return_object"/>
                        <OMSTR></OMSTR>
                </OMATP>
                <OMA>
                        <OMS cd="scscp1" name="procedure_call"/>
                        <OMA>
                                <OMS cd="scscp_transient_1" name="WS_Factorial"/>
                                <OMI> 5</OMI>
                        </OMA>
                </OMA>
        </OMATTR>
</OMOBJ>
<?scscp end ?>
```

4.1.3 SCSCPwait

◇ SCSCPwait(*stream[, timeout]*)   (function)

   Returns: nothing

   This function may be used by the SCSCP client to wait (using `IO_select` (**IO: IO_select**)) until the result of the procedure call will be available from *stream*. By default the timeout is one hour, to specify another value give it as the optional second argument in seconds. See the end of this chapter for the example.

4.1.4 OMGetObjectWithAttributes

◇ OMGetObjectWithAttributes(*stream*)   (function)

   Returns: record with components `object` and `attributes`, or `fail`

   This function is similar to the function `OMGetObject` from the OpenMath package, and the main difference is that it is able to understand OpenMath attribution pairs. It retrieves exactly one OpenMath object from the stream *stream*, and stores it in the `object` component of the returned record. If the OpenMath object has no attributes, the `attributes` component of the returned record will be an empty list, otherwise it will contain pairs [`attribute_name`,`attribute_value`], where `attribute_name` is a string, and `attribute_value` is a GAP object, whose type is determined by the kind of an attribute. Only attributes, defined by the SCSCP are allowed, otherwise an error message will be displayed.

   If the procedure was not successful, the function returns `fail` instead of an error message like the function `OMGetObject` (**OpenMath: OMGetObject**) does. Returning `fail` is useful when `OMGetObjectWithAttributes` is used inside accept-evaluate-return loop.

   As an example, the file `scscp/tst/omdemo.om` contains some OpenMath objects, including those from the SCSCP Specification [FHK⁺b]. We can retrieve them from this file, preliminary installing some SCSCP procedures using the function `InstallSCSCPprocedure` (5.1.1):

```
——————————————————————— Example ———————————————————————
gap> InstallSCSCPprocedure("WS_Factorial", Factorial );
gap> InstallSCSCPprocedure("GroupIdentificationService", IdGroup );
gap> InstallSCSCPprocedure("GroupByIdNumber", SmallGroup );
gap> test:=Filename( Directory( Concatenation(
>          GAPInfo.PackagesInfo.("scscp")[1].InstallationPath,"/tst/" ) ),
>          "omdemo.om" );;
gap> stream:=InputTextFile(test);;
gap> OMGetObjectWithAttributes(stream);
rec( object := 1, attributes := [  ] )
gap> OMGetObjectWithAttributes(stream);
rec( object := "Primes@localhost:26133",
  attributes := [ [ "call_id", "call_identifier" ] ] )
gap> OMGetObjectWithAttributes(stream);
rec( object := 120, attributes := [  ] )
gap> OMGetObjectWithAttributes(stream);
rec( object := [ 24, 12 ],
  attributes := [ [ "call_id", "alexk_9053" ], [ "option_runtime", 300000 ],
       [ "option_min_memory", 40964 ], [ "option_max_memory", 134217728 ],
       [ "option_debuglevel", 2 ], [ "option_return_object", "" ] ] )
gap> CloseStream( str );
```

4.2 Communication with the SCSCP client

4.2.1 OMPutProcedureCompleted

◊ `OMPutProcedureCompleted(`*stream, objrec*`)` (function)

> **Returns:** nothing

Takes a stream *stream*, and a record *objrec*, and writes to *stream* an OpenMath object `procedure_completed` with the result being `objrec.object` and information messages (as Open-Math attributes) given in the list `objrec.attributes`.

```
――――――――――――――――――――――――――――― Example ―――――――――――――――――――――――――――――
 gap> t:="";; stream:=OutputTextString(t,true);;
 gap> OMPutProcedureCompleted( stream,
 >      rec(object:=120,
 >      attributes:=[ [ "call_id", "user007" ] ] ));
 gap> Print(t);
 <?scscp start ?>
 <OMOBJ>
         <OMATTR>
                 <OMATP>
                         <OMS cd="scscp1" name="call_id"/>
                         <OMSTR>user007</OMSTR>
                 </OMATP>
                 <OMA>
                         <OMS cd="scscp1" name="procedure_completed"/>
                         <OMI> 120</OMI>
                 </OMA>
         </OMATTR>
 </OMOBJ>
 <?scscp end ?>
```

4.2.2 OMPutProcedureTerminated

◊ `OMPutProcedureTerminated(`*stream, objrec, error_cd, error_type*`)` (function)

> **Returns:** nothing

Takes a stream *stream*, and a record with an error message *objrec* (for example `rec( attributes := [ [ "call_id", "localhost:26133:87643:gcX33cCf" ] ], object := "localhost:26133 reports : Rational operations: <divisor> must not be zero")` and writes to the *stream* an OpenMath object `procedure_terminated` containing an error determined by the symbol *error_type* from the content dictionary *error_cd* (for example, `error_memory`, `error_runtime` or `error_system_specific` from the scscp1 content dictionary ([FHK⁺a]).

This is the internal function of the package which is used only in the code for the SCSCP server to return the error message to the client.

4.3 Example: SCSCP session

In the following example we start an SCSCP session and perform ten procedure calls in a loop before closing that session. Note that we demonstrate the usage of the session ID `sid` and the function `RandomString` from the OpenMath package to produce some unique call identifier. The call ID is a mandatory attribute for any procedure call, however, it is not nesessarily random; for example, it may be just a string with the number of the procedure call.

```
                                  ── Example ──
  gap> stream:=InputOutputTCPStream( "localhost", 26133 );
  #I  Creating a socket ...
  #I  Connecting to a remote socket via TCP/IP ...
  < input/output TCP stream to localhost:26133 >
  gap> sid := StartSCSCPsession( stream );
  #I  Got connection initiation message
  #I  Requesting version 1.3 from the server ...
  #I  Server confirmed version 1.3 to the client ...
  "localhost:26133:5541"
  gap> res:=[];
  [  ]
  gap> for i in [1..10] do
  >     OMPutProcedureCall( stream, "WS_Factorial",
  >       rec( object := [ i ],
  >        attributes := [ [ "call_id",
  >          Concatenation( sid, ":", RandomString(8) ) ] ] ) );
  >     SCSCPwait( stream );
  >     res[i]:=OMGetObjectWithAttributes( stream ).object;
  > od;
  gap> CloseStream(stream);
  gap> res;
  [ 1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800 ]
```

Also note the usage of `SCSCPwait` (4.1.3) to wait until the result of the computation will be available from `stream`.

Chapter 5

Running SCSCP server

5.1 Installation of SCSCP procedures

There may various ways to run SCSCP server, for example:

- allowing generic services like evaluation of arbitrary OpenMath code;

- offering highly specialized procedures like identification of groups of order 512;

- providing access to a database of mathematical objects.

Each of these use cases requires certain control over the level of functionality exposed to the client. To achieve this, before starting SCSCP service its provider must call the function `InstallSCSCPprocedure` (5.1.1) to make required procedures "visible" for the client.

Additionally, the service can be made made accessible only for clients running on the same computer, or accessible only through a particular network interface, or generally accessible. This customization is made at the stage of starting the SCSCP server with the function `RunSCSCPserver` (5.2.1).

5.1.1 InstallSCSCPprocedure

◇   `InstallSCSCPprocedure(procname, procfunc[, description][, narg1[, narg2][, signature]])`   (function)

     Returns: Nothing

     For a string `procname` and a function `procfunc`, `InstallSCSCPprocedure` makes the `procfunc` available as SCSCP procedure under the name `procname`, adding it to the transient OpenMath content dictionary `scscp_transient_1` that will exist during the service lifetime.

     The second argument `procfunc` may be either a standard or user-defined GAP function (procedure, operation, etc.).

     The rest of arguments are optional and may be used in a number of combinations:

- `description` is a string with the description of the procedure. It may be used by the help system. If it is omitted, the procedure will be reported as undocumented.

- `narg1` is a non-negative integer, specifying the minimal number of arguments, and `narg2` is a non-negative integer or infinity, specifying the maximal number of arguments. If `narg2` is omitted then the maximal number of arguments will be set to `narg1`. If both `narg1` and

narg2 are omitted then the minimal number of arguments will be set to zero and their maximal number will be set to infinity.

- *signature* is the signature record of the procedure. If the *signature* is given, then the number of arguments must be explicitly specified (by *narg1* with or without *narg2*) at least to zero and infinity respectively (to ensure proper matching of arguments). Note that it is completely acceptable for a symbol from a transient content dictionary to overstate the set of symbols which may occur in its children using the scscp2.symbol_set_all symbol, and to use standard OpenMath errors to reject requests later at the stage of their evaluation. For example, using such approach, we will define the procedure WS_Factorial accepting not only immediate <OMI> objects but anything which could be evaluated to an integer.

. The signature must be either a list of records, where *i*-th record corresponds to the *i*-th argument, or a record itself meaning that it specifies the signature for all arguments. In the latter case the record may be rec( ) corresponding to the scscp2.symbol_set_all symbol (this will be assumed by default if the signature will be omitted).

If more detailed description of allowed arguments is needed, the signature record (one for all arguments or a specific one) may have components CDgroups, CDs and Symbols. The first two are lists of names of content dictionary groups and content dictionaries, and the third is a record whose components are names of content dictionaries, containing lists of names of allowed symbols from these dictionaries, for example:

```
————————————————————————— Example —————————————————————————

  signature := rec( CDgroups := [ "scscp" ],
                    CDs := [ "arith1", "linalg1" ],
                    Symbols := rec( polyd1 := [ "DMP", "term", "SDMP" ],
                                    polyu := [ "poly_u_rep", "term" ] ) );

```

In the following example we define the function WS_Factorial that takes an integers and returns its factorial, using only mandatory arguments of InstallSCSCPprocedure (5.1.1):

```
————————————————————————— Example —————————————————————————

 gap> InstallSCSCPprocedure( "WS_Factorial", Factorial );
 InstallSCSCPprocedure : procedure WS_Factorial installed.

```

In the following example we install the procedure that will accept a list of permutations and return the number in the GAP Small Groups library of the group they generate (for the sake of simplicity we omit tests of validity of arguments, availability of IdGroup for groups of given order etc.)

```
————————————————————————— Example —————————————————————————

 gap> IdGroupByGenerators:=function( permlist )
 > return IdGroup( Group( permlist ) );
 > end;
 function( permlist ) ... end
 gap> InstallSCSCPprocedure( "GroupIdentificationService", IdGroupByGenerators );
 InstallSCSCPprocedure : procedure GroupIdentificationService installed.

```

After installation, the procedure may be reinstalled, if necessary:

```
──────── Example ────────
gap> InstallSCSCPprocedure( "WS_Factorial", Factorial );
WS_Factorial is already installed. Do you want to reinstall it [y/n]? y
InstallSCSCPprocedure : procedure WS_Factorial reinstalled.
```

Finally, some examples of various combinations of optional arguments:

```
──────── Example ────────
InstallSCSCPprocedure( "WS_Phi", Phi,
                       "Euler's totient function, see ?Phi in GAP", 1, 1 );
InstallSCSCPprocedure( "GroupIdentificationService",
                        IdGroupByGenerators, 1, infinity, rec() );
InstallSCSCPprocedure( "IdGroup512ByCode", IdGroup512ByCode, 1 );
InstallSCSCPprocedure( "WS_IdGroup", IdGroup, "See ?IdGroup in GAP" );
```

Note that it is quite acceptable to overstate the signature of the procedure and use only mandatory arguments in a call to `InstallSCSCPprocedure` (5.1.1), which will be installed then as a procedure that can accept arbitrary number of arguments encoded without any restrictions on OpenMath symbols used, because anyway the GAP system will return an error in case of the wrong number or type of arguments, though it might be a good practice to give a way to the client to get more precise procedure description a priori, that is before sending request. See 5.3 about utilities for obtaining such information about the SCSCP service. Some more examples of installation of SCSCP procedures are given in the file `scscp/example/myserver.g`.

5.1.2 OMsymRecord

◇ `OMsymRecord` (global variable)

This is the global record from the OpenMath package used for the conversion from OpenMath to GAP. It is extended in the SCSCP package by adding support for symbols from `scscp1` and `scscp2` content dictionaries ([FHK+a], [FHK+c]). Additionally, `InstallSCSCPprocedure` (5.1.1) adds to this record a component corresponding to the appropriate transient content dictionary (by default, `scscp_transient_1`) defining mappings between OpenMath symbols from this content dictionary and installed SCSCP procedures.

5.2 Starting SCSCP server

5.2.1 RunSCSCPserver

◇ `RunSCSCPserver(servertype, port)` (function)

Returns: nothing

Will start the SCSCP server at port given by the integer `port`. The first parameter `servertype` is either `true`, `false` or a string containing the server hostname:

- when `servertype` is `true`, the server will be started in a "universal" mode and will accept all incoming connections;

- when *servertype* is false, the server will be started at localhost and will not accept any incoming connections from outside;

- when *servertype* is a string, for example, "scscp.symbolic-computation.org", the server will be accessible only by specified server name (this may be useful to manage accessibility if, for example, the hardware has several network interfaces).

```
――――――――――――――――― Example ―――――――――――――――――
gap> RunSCSCPserver( "localhost", 26133 );
Ready to accept TCP/IP connections at localhost:26133 ...
Waiting for new client connection at localhost:26133 ...
```

Actually, there are several ways to start GAP SCSCP server:

- from the GAP session as shown in the example above;

- starting GAP as gap myserver.g, where myserver.g is the server configuration file with the last command being the call RunSCSCPserver, which may take its arguments from the configuration file scscp/config.g (an example of such configuration file is given in scscp/example/myserver.g);

- start GAP as a daemon using the script gapd.sh which is supplied in the root directory of the package (for the description of all available options see comments in gapd.sh) and may overwrite parameters from scscp/config.g.

See Section 2.2 about configuring files config.g and gapd.sh.

5.3 Procedures to get information about the SCSCP server

5.3.1 GetServiceDescription

◊ GetServiceDescription(*server, port*)   (function)
    Returns: record

Returns the record with three components containing strings with the name, version and description of the service as specified by the service provider in the scscp/config.g (for details about configuration files, see 2.2).

```
――――――――――――――――― Example ―――――――――――――――――
gap> GetServiceDescription( "localhost", 26133 );
rec( service_name := "GAP SCSCP service",
  version := "GAP 4.4.12 + SCSCP 1.1 started on Tue 21 Apr 2009 00:35:38 BST",
  description := "Started with scscp/example/myserver.g from SCSCP 1.1" )
```

5.3.2 GetAllowedHeads

◊ GetAllowedHeads(*server, port*)   (function)
    Returns: record

Returns the record with components corresponding to content dictionaries. Each component is a list of names of symbols from the corresponding content dictionary which are allowed to appear as a "head" symbol (i.e. the first child of the outermost <OMA>) in an SCSCP procedure call to the SCSCP server running at `server`:`port`.

Note that it is acceptable (although not quite desirable) for a server to "overstate" the set of symbols it accepts and use standard OpenMath errors to reject requests later.

```
──── Example ────
gap> GetAllowedHeads("localhost",26133);
rec( scscp_transient_1 := [ "GroupIdentificationService",
     "IO_UnpickleStringAndPickleItBack", "IdGroup512ByCode", "PointImages",
     "QuillenSeriesByIdGroup", "SCSCPStartTracing", "SCSCPStopTracing",
     "WS_ConwayPolynomial", "WS_Factorial", "WS_FactorsCFRAC", "WS_FactorsECM",
     "WS_FactorsMPQS", "WS_FactorsPminus1", "WS_FactorsPplus1", "WS_FactorsTD",
     "WS_IdGroup", "WS_Karatsuba", "WS_Phi" ] )
```

5.3.3 IsAllowedHead

◊ IsAllowedHead(*cd, symbol, server, port*)   (function)

Returns: `true` or `false`

Checks whether the OpenMath symbol `cd.symbol`, which may be a symbol from a standard or transient OpenMath content dictionary, is allowed to appear as "head" symbol (i.e. the first child of the outermost <OMA> in an SCSCP procedure call to the SCSCP server running at `server`:`port`. This enables the client to check whether a particular symbol is allowed without requesting the full list of symbols.

Also, it is acceptable (although not necessarily desirable) for a server to "overstate" the set of symbols it accepts and use standard OpenMath errors to reject requests later.

```
──── Example ────
gap> IsAllowedHead( "permgp1", "group", "localhost", 26133 );
true
gap> IsAllowedHead( "nums1", "pi", "localhost", 26133 );
false
```

5.3.4 GetTransientCD

◊ GetTransientCD(*transient_cd, server, port*)   (function)

Returns: record

Returns a record with the transient content dictionary `transient_cd` from the SCSCP server running at `server`:`port`. Names of components of this record correspond to symbols from the `meta` content dictionary.

By default, the name of the transient content dictionary for the GAP SCSCP server is `scscp_transient_1`. Other systems may use transient content dictionaries with another names, which, however, must always begin with `scscp_transient_` and may be guessed from the output of `GetAllowedHeads` (5.3.2).

─────────────────── Example ───────────────────

```
gap> GetTransientCD( "scscp_transient_1", "localhost", 26133 );
rec(
  Description := "This is a transient CD for the GAP SCSCP service",
  CDName := "scscp_transient_1",
  CDReviewDate := "2009-04-21",
  CDDate := "2009-04-21",
  CDVersion := "0",
  CDRevision := "0",
  CDStatus := "private",
  CDDefinitions :=
    [ rec( Name := "WS_Factorial",
           Description := "See ?Factorial in GAP" ),
      rec( Name := "WS_Phi",
           Description := "Euler's totient function, see ?Phi in GAP" ),
 ...
      rec( Name := "SCSCPStartTracing",
           Description := "To turn on tracing using given filename on the server" ),
      rec( Name := "SCSCPStopTracing",
           Description := "To turn off tracing on the server" ) ] )
```

5.3.5 GetSignature

◊ GetSignature(*transientcd, symbol, server, port*) (function)

Returns: record

Returns a record with the signature of the OpenMath symbol *transientcd.symbol* from a transient OpenMath content dictionary. This record contains components corresponding to the Open-Math symbol whose signature is described, the minimal and maximal number of its children (that is, of its arguments), and symbols which may be used in the OpenMath encoding of its children. Note that it is acceptable for a symbol from a transient content dictionary to overstate the set of symbols which may occur in its children using the scscp2.symbol_set_all symbol, and use standard Open-Math errors to reject requests later, like in the example below: using such approach, the procedure WS_Factorial is defined to accept not only immediate <OMI> objects but anything which could be evaluated to an integer.

─────────────────── Example ───────────────────

```
gap> GetSignature("scscp_transient_1","WS_Factorial","localhost",26133);
rec( symbol := rec( name := "WS_Factorial", cd := "scscp_transient_1" ),
     minarg := 1, maxarg := 1,
     symbolargs := rec( name := "symbol_set_all", cd := "scscp2" ) )
```

Chapter 6

Client's functionality

Sending and getting requests to the SCSCP server(s), the client operates with processes. Process is an abstraction which encapsulates an `InputOutputTCPStream` (see `IsInputOutputTCPStream` (3.1.1)) from a client to the server and the process ID of the CAS running as a server (deduced from the connection initiation message; may be unassigned, if the server CAS did not communicate it).

6.1 Sending request and getting result

6.1.1 IsProcess

◇ `IsProcess` (filter)

This is the category of processes. Processes in this category are created using the function `NewProcess` (6.1.2).

6.1.2 NewProcess

◇ `NewProcess(command, listargs, server, port)` (function)

Returns: object in the category `IsProcess`

command and `server` are strings, *listargs* is a list of GAP objects and `port` is an integer. Calls the SCSCP procedure with the name *command* and the list of arguments *listargs* at the server and port given by `server` and `port`. Returns an object in the category `IsProcess` for the subsequent waiting the result from its underlying stream.

It accepts the following options:

- `output:="object"` is used to specify that the server must return the actual object evaluated as a result of the procedure call. This is the default action requested by the client if the `output` option is omitted.

- `output:="cookie"` is used to specify that the result of the procedure call should be stored on the server, and the server should return a remote object (see 6.2) pointing to that result (that is, a cookie);

- `output:="nothing"` is used to specify that the server is supposed to reply with a `procedure_completed` message carrying no object just to signal that the call was completed

successfully (for the compatibility, this will be evaluated to a `"procedure completed"` string on the client's side);

- `cd:="cdname"` is used to specify that the OpenMath symbol corresponding to the first argument *command* should be looked up in the particular content dictionary `cdname`. Otherwise, it will be looked for in the default content dictionary (`scscp_transient_1` for the GAP SCSCP server);

- `debuglevel:=N` is used to obtain additional information attributes together with the result. The GAP SCSCP server does the following: if `N=1`, it will report about the CPU time in milliseconds required to compute the result; if `N=2` it will additionally report about the amount of memory used by GAP in bytes will be returned (using the output of `MemoryUsageByGAPinKbytes` (9.2.7) converted to bytes); if `N=3` it will additionally report the amount of memory in bytes used by the resulting object and its subobjects (using the output of `MemoryUsage` (**Reference: MemoryUsage**)).

See `CompleteProcess` (6.1.3) and `EvaluateBySCSCP` (6.1.4) for examples.

6.1.3 CompleteProcess

◇ `CompleteProcess(process)`   (function)

    Returns: record with components `object` and `attributes`

    The function waits, if necessary, until the underlying stream of the process will contain some data, then reads the appropriate OpenMath object from this stream and closes it.

    It has the option `output` which may have two values:

- `output:="cookie"` has the same meaning as for the `NewProcess` (6.1.2)

- `output:="tree"` is used to specify that the result obtained from the server should be returned as an XML parsed tree without its evaluation.

In the following example we demonstrate combination of the two previous functions to send request and get result, calling the procedure `WS_Factorial`, installed in the previous chapter:

```
                              ──── Example ────
 gap> s := NewProcess( "WS_Factorial", [10], "localhost", 26133 );
 #I  Creating a socket ...
 #I  Connecting to a remote socket via TCP/IP ...
 #I  Got connection initiation message
 #I  Request sent ...
 < process at localhost:26133 pid=6543 >
 gap> x := CompleteProcess(s);
 rec( object := "3628800", attributes := [ [ "call_id", "localhost:26133:6543" ] ] )
```

See more examples in the description of the function `EvaluateBySCSCP` (6.1.4), which combines the two previous functions by sending request and getting result in one call.

6.1.4 EvaluateBySCSCP

◇ `EvaluateBySCSCP(command, listargs, server, port)`   (function)

    Returns: record with components `object` and `attributes`

command and `server` are strings, *listargs* is a list of **GAP** objects and `port` is an integer. Calls the SCSCP procedure with the name *command* and the list of arguments *listargs* at the server and port given by `server` and `port`.

Since `EvaluateBySCSCP` combines `NewProcess` (6.1.2) and `CompleteProcess` (6.1.3), it accepts all options which may be used by that functions (`output`, `cd` and `debuglevel`) with the same meanings.

──────── Example ────────

```
gap> EvaluateBySCSCP( "WS_Factorial",[10],"localhost",26133);
#I  Creating a socket ...
#I  Connecting to a remote socket via TCP/IP ...
#I  Got connection initiation message
#I  Requesting version 1.3 from the server ...
#I  Server confirmed version 1.3 to the client ...
#I  Request sent ...
#I  Waiting for reply ...
rec( attributes := [ [ "call_id", "localhost:26133:2442:6hMEN40d" ] ],
     object := 3628800 )
gap> SetInfoLevel(InfoSCSCP,0);
gap> EvaluateBySCSCP( "WS_Factorial",[10],"localhost",26133 : output:="cookie" );
rec( attributes := [ [ "call_id", "localhost:26133:2442:jNQG6rml" ] ],
  object := < remote object TEMPVarSCSCP@localhost:26133 > )
gap> EvaluateBySCSCP( "WS_Factorial",[10],"localhost",26133 : output:="nothing" );
rec( attributes := [ [ "call_id", "localhost:26133:2442:9QHQrCjv" ] ],
  object := "procedure completed" )
```

Now we demonstrate the procedure `GroupIdentificationService`, also given in the previous chapter:

──────── Example ────────

```
gap> G:=SymmetricGroup(4);
Sym( [ 1 .. 4 ] )
gap> gens:=GeneratorsOfGroup(G);
[ (1,2,3,4), (1,2) ]
gap> EvaluateBySCSCP( "GroupIdentificationService", [ gens ],
>                     "localhost", 26133 : debuglevel:=3 );
#I  Creating a socket ...
#I  Connecting to a remote socket via TCP/IP ...
#I  Got connection initiation message
#I  Requesting version 1.3 from the server ...
#I  Server confirmed version 1.3 to the client ...
#I  Request sent ...
#I  Waiting for reply ...
rec( attributes := [ [ "call_id", "localhost:26133:2442:xOilXtnw" ],
      [ "info_runtime", 6 ], [ "info_memory", 670175232 ],
      [ "info_message", "Memory usage for the result is 24 bytes" ] ],
  object := [ 24, 12 ] )
```

Service provider may suggest to the client to use a counterpart function

─────────────── Example ───────────────

```
IdGroupWS:=function( G )
local H, result;
if not IsPermGroup(G) then
  H:= Image( IsomorphismPermGroup( G ) );
else
  H := G;
fi;
result := EvaluateBySCSCP ( "GroupIdentificationService",
            [ GeneratorsOfGroup(H) ], "localhost", 26133 );
return result.object;
end;
```

which works exactly like `IdGroup` (**Reference: IdGroup**):

─────────────── Example ───────────────

```
gap> G:=DihedralGroup(64);
<pc group of size 64 with 6 generators>
gap> IdGroupWS(G);
#I  Creating a socket ...
#I  Connecting to a remote socket via TCP/IP ...
#I  Got connection initiation message
#I  Request sent ...
#I  Waiting for reply ...
[ 64, 52 ]
```

6.1.5 TerminateProcess

◊ TerminateProcess(*process*)   (function)

The function is supposed to send an "out-of-band" interrupt signal to the server. Current implementation works only when the server is running as "localhost" by sending a SIGINT to the server using its PID contained in the *process*. It will do nothing if the server is running remotely, as the SCSCP specification allows the server to ignore interrupt messages. Remote interrupts will be introduced in one of the next versions of the package.

6.2 Remote objects

The SCSCP package introduces new kind of objects - *remote objects*. They provide an opportunity to manipulate with objects on remote services without their actual transmitting over the network. Remote objects store the information that allows to access the original object: the server name and the port number through which the object can be accessed, and the variable name under which it is stored in the remote system. Two remote objects are equal if and only if all these three parameters coincide.

There are two types of remote object which differ by their lifetime:

- temporary remote objects which exist only within a single session;

- persistent remote objects which stay alive across multiple sessions.

First we show the example of the temporary remote object in a session. The procedure `PointImages` returns the set of images of a point i under the generators of the group G. First we create the symmetric group S_3 on the client and store it remotely on the server (call 1), then we compute set of images for $i = 1, 2$ (calls 2,3) and finally demonstrate that we may retrieve the group from the server (call 4):

```
                    ───── Example ─────
gap> stream:=InputOutputTCPStream( "localhost", 26133 );
#I  Creating a socket ...
#I  Connecting to a remote socket via TCP/IP ...
< input/output TCP stream to localhost:26133 >
gap> StartSCSCPsession(stream);
#I  Got connection initiation message
#I  Requesting version 1.3 from the server ...
#I  Server confirmed version 1.3 to the client ...
"localhost:26133:6184"
gap> OMPutProcedureCall( stream, "store_session",
>       rec( object := [ SymmetricGroup(3) ],
>        attributes := [ [ "call_id", "1" ],
>                        ["option_return_cookie"] ] ) );
true
gap> SCSCPwait( stream );
gap> G:=OMGetObjectWithAttributes( stream ).object;
< remote object TEMPVarSCSCPoS5nhta3@localhost:26133 >
gap> OMPutProcedureCall( stream, "PointImages",
>       rec( object := [ G, 1 ],
>        attributes := [ [ "call_id", "2" ] ] ) );
true
gap> SCSCPwait( stream );
gap> OMGetObjectWithAttributes( stream );
rec( attributes := [ [ "call_id", "2" ] ], object := [ 2 ] )
gap> OMPutProcedureCall( stream, "PointImages",
>       rec( object := [ G, 2 ],
>        attributes := [ [ "call_id", "3" ] ] ) );
true
gap> SCSCPwait( stream );
gap> OMGetObjectWithAttributes( stream );
rec( attributes := [ [ "call_id", "3" ] ], object := [ 1, 3 ] )
gap> OMPutProcedureCall( stream, "retrieve",
>       rec( object := [ G ],
>        attributes := [ [ "call_id", "4" ] ] ) );
true
gap> SCSCPwait( stream );
gap> OMGetObjectWithAttributes( stream );
rec( attributes := [ [ "call_id", "4" ] ],
  object := Group([ (1,2,3), (1,2) ]) )
gap> CloseStream(stream);
```

After the stream is closed, it is no longer possible to retrieve the group G again or use it as an argument.

Thus, the usage of remote objects existing during a session reduces the network traffic, since we pass only references instead of actual OpenMath representation of an object. Also, the remote object

on the server may accumulate certain information in its properties and attributes, which may not be included in it default OpenMath representation.

Now we show remote objects which remain alive after the session is closed. Such remote objects may be accessed later, for example, by:

- subsequent procedure calls from the same instance of GAP or another system;

- other instances of GAP or another systems (if the identifier of an object is known)

- another SCSCP servers which obtained a reference to such object as an argument of a procedure call.

6.2.1 StoreAsRemoteObjectPersistently

◊ StoreAsRemoteObjectPersistently(*obj, server, port*)   (function)
◊ StoreAsRemoteObject(*obj, server, port*)   (function)
    Returns: remote object

Returns the remote object corresponding to the object created at *server*:*port* from the Open-Math representation of the first argument *obj*. The second form is just a synonym.

```
————————————————————————— Example —————————————————————————
gap> s:=StoreAsRemoteObject( SymmetricGroup(3), "localhost", 26133 );
#I  Creating a socket ...
#I  Connecting to a remote socket via TCP/IP ...
#I  Got connection initiation message
#I  Request sent ...
#I  Waiting for reply ...
< remote object TEMPVarSCSCP@localhost:26133 >
```

Internally, the remote object carries all the information which is required to get access to the original object: its identifier, server and port:

```
————————————————————————— Example —————————————————————————
gap> s![1];
"TEMPVarSCSCP"
gap> s![2];
"localhost"
gap> s![3];
26133
```

When the remote object is printed in the OpenMath format, we use symbols @ and : to combine these parameters in the OpenMath reference:

```
————————————————————————— Example —————————————————————————
gap> OMPrint(s);
<OMOBJ>
        <OMR xref="TEMPVarSCSCP@localhost:26133" />
</OMOBJ>
```

This allows substitution of remote object as arguments into procedure calls in the same manner like we do this with usual objects:

```
—————————————————————— Example ——————————————————————
gap> EvaluateBySCSCP("WS_IdGroup",[s],"localhost",26133);
#I  Creating a socket ...
#I  Connecting to a remote socket via TCP/IP ...
#I  Got connection initiation message
#I  Request sent ...
#I  Waiting for reply ...
rec( object := [ 6, 1 ], attributes := [ [ "call_id", "localhost:26133:6726" ] ] )
```

6.2.2 IsRemoteObject

◊ IsRemoteObject   (filter)

This is the category of remote objects.

6.2.3 RemoteObjectsFamily

◊ RemoteObjectsFamily   (family)

This is the familty of remote objects.

6.2.4 RetrieveRemoteObject

◊ RetrieveRemoteObject(*remoteobject*)   (function)
   Returns: object

This function retrieves the remote object from the remote service in the OpenMath format and constructs it locally. Note, however, that for a complex mathematical object its default OpenMath representation may not contain all information about it which was accumulated during its lifetime on the SCSCP server.

```
—————————————————————— Example ——————————————————————
gap> RetrieveRemoteObject(s);
#I  Creating a socket ...
#I  Connecting to a remote socket via TCP/IP ...
#I  Got connection initiation message
#I  Request sent ...
#I  Waiting for reply ...
Group([ (1,2,3), (1,2) ])
```

6.2.5 UnbindRemoteObject

◊ UnbindRemoteObject(*remoteobject*)   (function)
   Returns: true or false

Removes any value currently bound to the global variable determined by *remoteobject* at the SCSCP server, and returns true or false dependently on whether this action was successful or not.

──────────── Example ────────────

```
gap> UnbindRemoteObject(s);
#I  Creating a socket ...
#I  Connecting to a remote socket via TCP/IP ...
#I  Got connection initiation message
#I  Request sent ...
#I  Waiting for reply ...
true
```

Finally, we show an example when first we create a group on the service running on port 26133, and then identify it on the service running on port 26134. Instead of transmitting the group to the client and then sending it as an argument to the second service, the latter service directly retrieves the group from the first service.

──────────── Example ────────────

```
gap> s:=StoreAsRemoteObject( SymmetricGroup(3), "localhost", 26133 );
< remote object TEMPVarSCSCP@localhost:26133 >
gap> EvaluateBySCSCP( "WS_IdGroup", [ s ], "localhost", 26134 );
rec( object := [ 6, 1 ], attributes := [ [ "call_id", "localhost:26134:7414" ] ] )
```

──────────── Example ────────────

```
gap> EvaluateBySCSCP("WS_IdGroup",[s],"localhost",26133 : output:="cookie" );
#I  Creating a socket ...
#I  Connecting to a remote socket via TCP/IP ...
#I  Got connection initiation message
#I  Request sent ...
#I  Waiting for reply ...
rec( object := < remote object TEMPVarSCSCP2@localhost:26133 >,
  attributes := [ [ "call_id", "localhost:26133:7328" ] ] )
```

Chapter 7

Examples of **SCSCP** usage

In this chapter we are going to demonstrate some examples of communication between client and server using the SCSCP.

7.1 Providing services with the SCSCP package

You can try to run the SCSCP server with the configuration file `scscp/example/myserver.g`. To do this, go to that directory and enter `gap myserver.g`. After this you will see some information messages and finally the server will start to wait for the connection. The final part of the startup screen may look as follows:

```
 ────────── Example ──────────

  #I  Installed SCSCP procedure Factorial
  #I  Installed SCSCP procedure WS_Factorial
  #I  Installed SCSCP procedure GroupIdentificationService
  #I  Installed SCSCP procedure IdGroup512ByCode
  #I  Installed SCSCP procedure WS_IdGroup
  #I  Installed SCSCP procedure WS_Karatsuba
  #I  Installed SCSCP procedure EvaluateOpenMathCode
  #I  Ready to accept TCP/IP connections at localhost:26133 ...
  #I  Waiting for new client connection at localhost:26133 ...

```

See further self-explanatory comments in the file `scscp/example/myserver.g`. There also some test files in the directory `scscp/tst/` supplied with detailed comments. First, you may use demonstration files, preliminary turning on the demonstration mode as it is explained in these files, or just executing step by step each command from `scscp/tst/demo.g` and `scscp/tst/omdemo.g`. Then you can try to use files `scscp/tst/id512.g`, `scscp/tst/idperm.g` and `scscp/tst/factor.g` for further tests of SCSCP services.

7.2 Identifying groups of order 512

We will give an example guiding you through all steps of creation of your own SCSCP service.

The GAP Small Group Library does not provide identification for groups of order 512 using the function `IdGroup`:

```
──────────────────────── Example ────────────────────────
 gap> IdGroup( DihedralGroup( 256 ) );
 [ 256, 539 ]
 gap> IdGroup(DihedralGroup(512));
 Error, the group identification for groups of size
 512 is not available called from
 <function>( <arguments> ) called from read-eval-loop
 Entering break read-eval-print loop ...
 you can 'quit;' to quit to outer loop, or
 you can 'return;' to continue
 brk>
```

However, the GAP package ANUPQ [GNO] has a function `IdStandardPresented512Group` that does this work as demonstrated below:

```
──────────────────────── Example ────────────────────────
 gap> LoadPackage("anupq");
 -------------------------------------------------------------
 Loading ANUPQ 3.0 (ANU p-Quotient package)
 C code by  Eamonn O'Brien <obrien@math.auckland.ac.nz>
            (ANU pq binary version: 1.8)
 GAP code by Werner Nickel <nickel@mathematik.tu-darmstadt.de>
         and   Greg Gamble  <gregg@math.rwth-aachen.de>


            For help, type: ?ANUPQ
 -------------------------------------------------------------
 true
 gap> G := DihedralGroup( 512 );
 <pc group of size 512 with 9 generators>
 gap> F := PqStandardPresentation( G );
 <fp group on the generators [ f1, f2, f3, f4, f5, f6, f7, f8, f9 ]>
 gap> H := PcGroupFpGroup( F );
 <pc group of size 512 with 9 generators>
 gap> IdStandardPresented512Group( H );
 [ 512, 2042 ]
```

The package ANUPQ requires UNIX environment and it is natural to provide an identification service for groups of order 512 to make it available for other platforms.

Now we need to decide how the client will transmit a group to the server. Can we encode this group in OpenMath? But there is no content dictionary for PcGroups. Should we convert it to a permutation representation to be able to use existing content dictionaries? But then the resulting OpenMath code will be not compact. However, the SCSCP protocol provides enough freedom for the user to select its own data representation, and since we are linking together two copies of the same system, we may use the *pcgs code* to pass the data to the server (see `CodePcGroup` (**Reference: CodePcGroup**).

First we create a function which accepts the integer number that is the code for pcgs of a group of order 512 and returns the number of this group in the GAP Small Groups library:

```
──────────────────────── Example ────────────────────────
 IdGroup512ByCode := function( code )
```

```
  local G, F, H;
  G := PcGroupCode( code, 512 );
  F := PqStandardPresentation( G );
H := PcGroupFpGroup( F );
return IdStandardPresented512Group( H );
  end;
```

After such function was created on the server, we need to make it "visible" as an SCSCP procedure:

```
─── Example ───
gap> InstallSCSCPprocedure("IdGroup512", IdGroup512ByCode );
InstallSCSCPprocedure : procedure IdGroup512 installed.
```

Note that this function assumes that the argument is a valid code for some group of order 512, and we wish the client to make it sure that this is the case. To do this, and also for the client's convenience, we provide the client's counterpart for this service. Here the group must be a pc-group of order 512, otherwise an error message will appear.

```
─── Example ───
IdGroup512:=function( G )
local code, result;
if Size( G ) <> 512 then
  Error( "G must be a group of order 512 \n" );
fi;
code := CodePcGroup( G );
result := EvaluateBySCSCP( "IdGroup512ByCode", [ code ],
                        "localhost", 26133 );
return result.object;
end;
```

Now the client can call the function `IdGroup512`, and the procedure of getting result is as much straightforward as using `IdGroup` for those groups where it works:

```
─── Example ───
gap> IdGroup512(DihedralGroup(512));
#I  Creating a socket ...
#I  Connecting to a remote socket via TCP/IP ...
#I  Got connection initiation message
#I  Request sent ...
#I  Waiting for reply ...
[ 512, 2042 ]
```

Chapter 8

Parallel computing with SCSCP

8.1 Managing multiple requests

Using procedure calls explained in the previous section, the user can create several requests to multiple services to execute them in parallel, or to wait until the fastest result will be available.

8.1.1 SynchronizeProcesses

◊ SynchronizeProcesses(*process1, process2, ..., processN*)   (function)
◊ SynchronizeProcesses(*proclist*)   (function)
   Returns: list of records with components `object` and `attributes`
   The function collects results of from each process given in the argument, and returns the list, i-th entry of which is the result obtained from the i-th process. The function accepts both one argument that is a list of processes, and arbitrary number of arguments, each of them being a process.

```
 ────────────────────────────── Example ──────────────────────────────

  gap> a:=NewProcess( "WS_Factorial", [10], "localhost", 26133 );
  #I  Creating a socket ...
  #I  Connecting to a remote socket via TCP/IP ...
  #I  Got connection initiation message
  #I  Requesting version 1.3 from the server ...
  #I  Server confirmed version 1.3 to the client ...
  #I  Request sent ...
  < process at localhost:26133 pid=2064 >
  gap> b:=NewProcess( "WS_Factorial", [20], "localhost", 26134 );
  #I  Creating a socket ...
  #I  Connecting to a remote socket via TCP/IP ...
  #I  Got connection initiation message
  #I  Requesting version 1.3 from the server ...
  #I  Server confirmed version 1.3 to the client ...
  #I  Request sent ...
  < process at localhost:26134 pid=1975 >
  gap> SynchronizeProcesses(a,b);
  #I  Process number 1 is ready
  #I  Closed 1st process, waiting for 2nd ...
  [ rec( attributes := [ [ "call_id", "localhost:26133:2064:yCWBGYFO" ] ],
        object := 3628800 ),
```

```
    rec( attributes := [ [ "call_id", "localhost:26134:1975:yAAWvGTL" ] ],
        object := 2432902008176640000 ) ]
```

8.1.2 FirstProcess

◊ FirstProcess(*process1, process2, ..., processN*)   (function)
◊ FirstProcess(*proclist*)   (function)

    Returns: records with components `object` and `attributes`

The function waits for the result from each process given in the argument, and returns the result coming first, terminating all remaining processes at the same time. The function accepts both one argument that is a list of processes, and arbitrary number of arguments, each of them being a process.

─────────────────────────────── Example ───────────────────────────────

```
gap> a:=NewProcess( "WS_Factorial", [10], "localhost", 26133 );
#I  Creating a socket ...
#I  Connecting to a remote socket via TCP/IP ...
#I  Got connection initiation message
#I  Requesting version 1.3 from the server ...
#I  Server confirmed version 1.3 to the client ...
#I  Request sent ...
< process at localhost:26133 pid=2064 >
gap> b:=NewProcess( "WS_Factorial", [20], "localhost", 26134 );
#I  Creating a socket ...
#I  Connecting to a remote socket via TCP/IP ...
#I  Got connection initiation message
#I  Requesting version 1.3 from the server ...
#I  Server confirmed version 1.3 to the client ...
#I  Request sent ...
< process at localhost:26134 pid=1975 >
gap>  FirstProcess(a,b);
#I  Process number 1 is ready
rec( attributes := [ [ "call_id", "localhost:26133:2064:mdb8RaO2" ] ],
  object := 3628800 )
```

8.1.3 SCSCPservers

◊ SCSCPservers   (global variable)

    `SCSCPservers` is a list of hosts and ports to search for SCSCP services (which may be not only represented by GAP services, but also by another SCSCP-compliant systems).

It is used by parallel skeletons `ParQuickWithSCSCP` (8.1.4) and `ParListWithSCSCP` (8.2.1).

The initial value of this variable is specified in the file `scscp/configpar.g` and may be reassigned later.

8.1.4 ParQuickWithSCSCP

◇ `ParQuickWithSCSCP(`*commands, listargs*`)`   (function)

Returns: record with components `object` and `attributes`

This function is constructed using the `FirstProcess` (8.1.2). It is useful when it is not known which partcular method is more efficient, because it allows to call in parallel several procedures (given by the list of their names *commands*) with the same list of arguments *listargs* (having the same meaning as in `EvaluateBySCSCP` (6.1.4)) and obtain the result of that procedure call which will be computed faster.

In the example below we call two factorisation methods from the GAP package FactInt to factorise $2^{150}+1$. The example is selected in such a way that the runtime of these two methods is approximately the same, so you should expect results from both methods in some random order from repeated calls.

```
————————————————————— Example —————————————————————
gap> ParQuickWithSCSCP( [ "WS_FactorsECM", "WS_FactorsMPQS" ], [ 2^150+1 ] );
#I  Creating a socket ...
#I  Connecting to a remote socket via TCP/IP ...
#I  Got connection initiation message
#I  Request sent ...
#I  Creating a socket ...
#I  Connecting to a remote socket via TCP/IP ...
#I  Got connection initiation message
#I  Request sent ...
#I  Process number 2 is ready
rec( object := [ 5, 5, 5, 13, 41, 61, 101, 1201, 1321, 8101, 63901, 268501,
      13334701, 1182468601 ],
 attributes := [ [ "call_id", "localhost:26134:6731" ] ] )
```

8.2 MasterWorker skeleton

In this section we will present more general framework to run parallel computations, which has a number of useful features:

- it is implemented purely in GAP;

- the client (i.e. master, which orchestrates the computation) will work in UNIX/Linux, Mac OS X and MS Windows;

- it may orchestrate both GAP and non-GAP SCSCP servers;

- if one of servers (i.e. workers) will be lost, it will retry the computation on another available server;

- it allows to add dynamically new workers during the computation on hostnames and ports from a range perviously declared in `SCSCPservers` (8.1.3).

To configure this functionality, the file `scscp/configpar.g` assigns the global variable `SCSCPservers` which specifies a list of hosts and ports to search for SCSCP services (which may be not only represented by GAP services, but also by another SCSCP-compliant systems). See comments in this file for further instructions.

8.2.1 ParListWithSCSCP

◊ `ParListWithSCSCP(`*listargs, procname*`)`   (function)

> **Returns:** list
>
> `ParListWithSCSCP` implements the well-known master-worker skeleton: we have a master (SCSCP client) and a number of workers (SCSCP servers) which obtain pieces of work from the client, perform the required job and report back with the result, waiting for the next job.
>
> It returns the list of the same length as *listargs*, *i*-th element of which is the result of calling the procedure *procname* with the argument *listargs[i]*.
>
> It accepts two options which should be given as non-negative integers: `timeout` which specifies how long the client will wait for the result (if not given, the default value is one hour) and `recallfrequency` which specifies the number of iterations after which the search for new services will be performed (if not given the default value is zero meaning no such search at all).

```
─────────── Example ───────────
gap> ParListWithSCSCP( List( [2..6], n -> SymmetricGroup(n)), "WS_IdGroup" );
#I  master -> [ "localhost", 26133 ] : SymmetricGroup( [ 1 .. 2 ] )
#I  master -> [ "localhost", 26134 ] : SymmetricGroup( [ 1 .. 3 ] )
#I  [ "localhost", 26133 ] --> master : [ 2, 1 ]
#I  master -> [ "localhost", 26133 ] : SymmetricGroup( [ 1 .. 4 ] )
#I  [ "localhost", 26134 ] --> master : [ 6, 1 ]
#I  master -> [ "localhost", 26134 ] : SymmetricGroup( [ 1 .. 5 ] )
#I  [ "localhost", 26133 ] --> master : [ 24, 12 ]
#I  master -> [ "localhost", 26133 ] : SymmetricGroup( [ 1 .. 6 ] )
#I  [ "localhost", 26133 ] --> master : [ 720, 763 ]
#I  [ "localhost", 26134 ] --> master : [ 120, 34 ]
[ [ 2, 1 ], [ 6, 1 ], [ 24, 12 ], [ 120, 34 ], [ 720, 763 ] ]
```

8.2.2 SCSCPreset

◊ `SCSCPreset()`   (function)

> **Returns:** nothing
>
> If an error occurs during a call of `ParQuickWithSCSCP` (8.1.4) and `ParListWithSCSCP` (8.2.1), some of parallel requests may be still running at the remaining services, making them inaccessible for further procedure calls. `SCSCPreset` resets them by closing all open streams to SCSCP servers.

8.2.3 SCSCPLogTracesToGlobal

◊ `SCSCPLogTracesToGlobal(`*testname*`)`   (function)
◊ `SCSCPLogTracesToGlobal()`   (function)

To analyse the performance of parallel SCSCP framework, we make use of the EdenTV program [BL07] developed initially to visualize the performance of parallel programs written in functional programming language Eden, and now distributed under the GNU Public License and available from http://www.mathematik.uni-marburg.de/~eden/?content=EdenTV.

Called with the string containing the name of the test, this functions turns on writing information about key activity events into trace files in current directories for the client and servers listed `SCSCPservers` (8.1.3). The trace file will have the name of the format *testname*`.client.tr` for

the client and $testname.<hostname>.<port>.tr$ for the server. After the test these files should be collected from remote servers and concatenated (e.g. using `cat`) together with the standard preamble from the file `scscp/tracing/stdhead.txt` (we recommend to put after the preamble first all traces from servers and then the client's traces to have nicer diagrams). The resulting file then may be opened with EdenTV.

In the following example we use a dual core MacBook laptop to generate trace files for two tests and then show their corresponding trace diagrams:

```
————————————————— Example —————————————————

SCSCPLogTracesToGlobal("quillen100");
ParListWithSCSCP( List( [1..100], i->[512,i]), "QuillenSeriesByIdGroup" );
SCSCPLogTracesToGlobal();
SCSCPLogTracesToGlobal( "euler" );
ParListWithSCSCP( [1..1000], "WS_Phi" );
SCSCPLogTracesToGlobal();
```
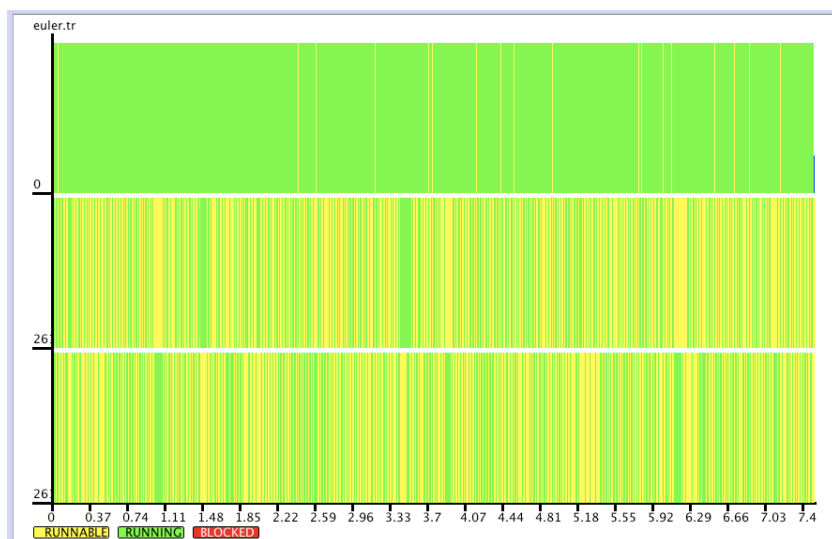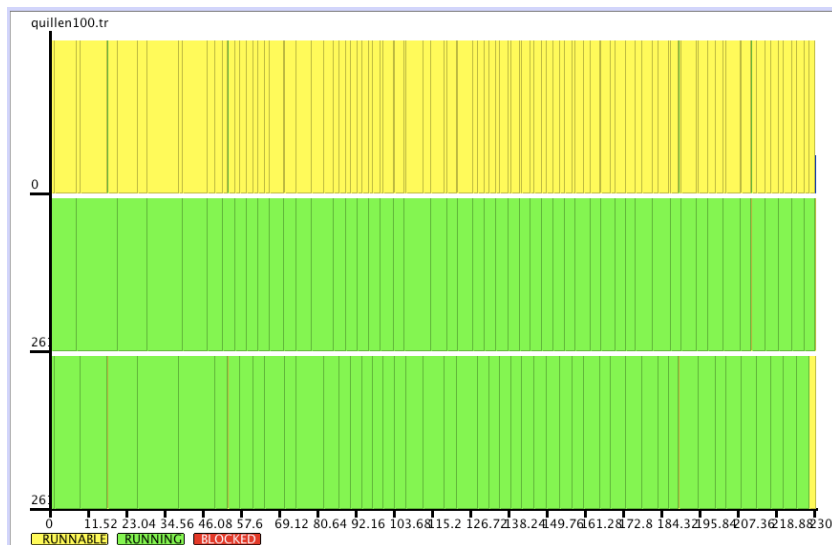
The diagrams (made on an dual core MacBook laptop), shows that in the first case parallelising is efficient and master successfully distributes load to workers, while in the second case a single computation is just too short, so most of the time is spent on communication. To parallelize the Euler's function example efficiently, tasks must rather be grouped in chunks, which should be enough large to reduce the communication overload, but enough small to ensure that tasks are evenly distributed.

Of course, tracing can be used to investigate communication between a client and a single server in a non-parallel context as well. For this purpose, `SCSCPservers` (8.1.3) must be modified to contain only one server.

`ParListWithSCSCP` (8.2.1) can be easily modified to have parallel versions of other list operations like `ForAll` (**Reference: ForAll**), `ForAny` (**Reference: ForAny**), `First` (**Reference: First**), `Number` (**Reference: Number**), `Filtered` (**Reference: Filtered**), and also to have the skeleton in which the queue may be modified during the computation (for example, to compute orbits). We plan to provide such tools in one of the next versions of the package.

8.3   Example: parallelising Karatsuba multiplication for polynomials

The file `scscp/example/karatsuba.g` contains an implementation of the Karatsuba multiplication algorithm for polynomials. This algorithm can be easily parallelized since each recursive step creates three recursive calls of the same function for other polynomials. *We will not parallelize each* recursive call, since this will create enormous data flow. Instead of this we parallelize only the top-level function. For our experiments with parallelising Karatsuba multiplication for polynomials with integer coefficients we used the multi-core workstation, on which we started one SCSCP client and two SCSCP servers. After reading the file `scscp/example/karatsuba.g`, the following function was created on the server

```
                           ──── Example ────
 KaratsubaPolynomialMultiplicationExtRepByString:=function(s1,s2)
 return String( KaratsubaPolynomialMultiplicationExtRep(
                EvalString(s1), EvalString(s2) ) );
 end;
```

and then was made available as an SCSCP procedure under the name `WS_Karatsuba` by the command

```
                           ──── Example ────
 InstallSCSCPprocedure( "WS_Karatsuba",
                        KaratsubaPolynomialMultiplicationExtRepByString);
```

This function provides a "bridge" between the client's function `KaratsubaPolynomialMultiplicationWS` and the server's function `KaratsubaPolynomialMultiplicationExtRep`, which performs the actual work on the server. `WS_Karatsuba` converts its string arguments into internal representation of univariate polynomials (basically, lists of integers) and then converts the result back into string (since such data exchange format was chosen).

We are going to parallelize the following part of the client's code:

```
———————————————————— Example ————————————————————

...
u := KaratsubaPolynomialMultiplicationExtRep(f1,g1);
v := KaratsubaPolynomialMultiplicationExtRep(f0,g0);
w := KaratsubaPolynomialMultiplicationExtRep(
      PlusLaurentPolynomialsExtRep(f1,f0),
      PlusLaurentPolynomialsExtRep(g1,g0) );
...
```

and this can be done straightforwardly - we replace two first calls by calls of the appropriate SCSCP services, then perform the 3rd call locally and then collect the results from the two remote calls:

```
———————————————————— Example ————————————————————

...
u := NewProcess( "WS_Karatsuba",[ String(f1), String(g1) ],"localhost", 26133);
v := NewProcess( "WS_Karatsuba",[ String(f0), String(g0) ],"localhost", 26134);
w := KaratsubaPolynomialMultiplicationExtRep(
      PlusLaurentPolynomialsExtRep(f1,f0),
      PlusLaurentPolynomialsExtRep(g1,g0) );
wsresult:=SynchronizeProcesses2( u,v );
u := EvalString( wsresult[1].object );
v := EvalString( wsresult[2].object );
...
```

We obtain almost double speedup on three cores on randomly generated polynomials of degree 32000:

```
———————————————————— Example ————————————————————

gap> fam:=FamilyObj(1);;
gap> f:=LaurentPolynomialByCoefficients( fam,
>        List([1..32000],i->Random(Integers)), 0, 1 );;
gap> g:=LaurentPolynomialByCoefficients( fam,
>        List([1..32000],i->Random(Integers)), 0, 1 );;
gap> t2:=KaratsubaPolynomialMultiplication(f,g);;time;
5892
gap> t3:=KaratsubaPolynomialMultiplicationWS(f,g);;time;
2974
```

Chapter 9

Service functions

9.1 Tools for making documentation

9.1.1 SCSCPBuildManual

◇ `SCSCPBuildManual()` (function)

This function is used to build the manual in PDF and HTML formats and also in the plain text format for the online help. To build the PDF version, a recent and fairly complete TEX distribution is recommended.

Since the SCSCP manual is distributed together with the package, it is not necessary for the user to use this function, which is normally used only by developers.

9.1.2 SCSCPBuildManualHTML

◇ `SCSCPBuildManualHTML()` (function)

This function is used to build the manual only in HTML format. This does not require TEX installation and works in any operating system.

Since the SCSCP manual is distributed together with the package, it is not necessary for the user to use this function, which is normally used only by developers.

9.2 Other commands

9.2.1 PingSCSCPservice

◇ `PingSCSCPservice(hostname, portnumber)` (function)

Returns: `true` or `fail`

This function returns `true` if the client can establish connection with the SCSCP server at *hostname*:*portnumber*. Otherwise, it returns `fail`.

```
 ─────────────────────── Example ───────────────────────

 gap> PingSCSCPservice("localhost",26133);
 #I  Creating a socket ...
 #I  Connecting to a remote socket via TCP/IP ...
 #I  Got connection initiation message
```

```
  true
  gap> PingSCSCPservice("localhost",26134);
  #I  Creating a socket ...
  #I  Connecting to a remote socket via TCP/IP ...
  Error: rec(
    number := 61,
    message := "Connection refused" )
  fail
```

9.2.2 PingStatistic

◇ PingStatistic(*hostname, portnumber, n*)   (function)
  Returns: nothing

The function is similar to the UNIX `ping`. It tries n times to establish connection with the SCSCP server at *hostname*:*portnumber*, and then displays statistical information.

——————— Example ———————
```
  gap> PingStatistic("localhost",26133,5);
  #I  Creating a socket ...
  #I  Connecting to a remote socket via TCP/IP ...
  #I  Got connection initiation message nr 1
  ...
  #I  Creating a socket ...
  #I  Connecting to a remote socket via TCP/IP ...
  #I  Got connection initiation message nr 5
  100 packets transmitted, 100 received, 0% packet loss, time 26ms
  min/avg/max = [ 0, 13/50, 2 ]
```

9.2.3 InfoSCSCP

◇ InfoSCSCP   (info class)

InfoSCSCP is a special Info class for the SCSCP package. The amount of information to be displayed can be specified by the user by setting InfoLevel for this class from 0 to 4, and the default value of InfoLevel for the package is specified in the file `scscp/config.g`. The higher the level is, the more information will be displayed. To change the InfoLevel to `k`, use the command `SetInfoLevel(InfoSCSCP, k)`. In the following examples we demonstrate various degrees of output details using Info messages.

——————— Example ———————
```
  gap> EvaluateBySCSCP( "WS_Factorial",[10],"localhost",26133);
  #I  Creating a socket ...
  #I  Connecting to a remote socket via TCP/IP ...
  #I  Got connection initiation message
  #I  Request sent ...
  #I  Waiting for reply ...
  rec( object := "3628800", attributes := [ [ "call_id", "localhost:26133:38" ] ] )
```

─── Example ───

```
gap> SetInfoLevel(InfoSCSCP,0);
gap> EvaluateBySCSCP( "WS_Factorial",[10],"localhost",26133);
rec( object := "3628800", attributes := [ [ "call_id", "localhost:26133:41" ] ] )
```

─── Example ───

```
gap> SetInfoLevel(InfoSCSCP,3);
gap> G:=SymmetricGroup(4);;
gap> gens:=GeneratorsOfGroup(G);
[ (1,2,3,4), (1,2) ]
gap> EvaluateBySCSCP( "GroupIdentificationService",[gens],"localhost",26133);
#I  Creating a socket ...
#I  Connecting to a remote socket via TCP/IP ...
#I  Got connection initiation message
#I Composing procedure_call message:
<?scscp start ?>
<OMOBJ>
    <OMATTR>
                <OMATP>
                        <OMS cd="scscp1" name="call_id"/>
                        <OMSTR>localhost:26133:1212</OMSTR>
                </OMATP>
                <OMA>
                        <OMS cd="scscp1" name="procedure_call"/>
                        <OMA>
                                <OMS cd="scscp_transient_1" name="GroupIdentificationService"/>
                                <OMA>
                                        <OMS cd="list1" name="list"/>
                                        <OMA>
                                                <OMS cd="permut1" name="permutation"/>
                                                <OMI> 2</OMI>
                                                <OMI> 3</OMI>
                                                <OMI> 4</OMI>
                                                <OMI> 1</OMI>
                                        </OMA>
                                        <OMA>
                                                <OMS cd="permut1" name="permutation"/>
                                                <OMI> 2</OMI>
                                                <OMI> 1</OMI>
                                        </OMA>
                                </OMA>
                        </OMA>
                </OMA>
        </OMATTR>
</OMOBJ>
<?scscp end ?>
#I  Request sent ...
#I  Waiting for reply ...
#I  <?scscp start ?>
#I Received message:
<OMOBJ>
```

```
                <OMATTR>
                        <OMATP>
                                <OMS cd="scscp1" name="call_id"/>
                                <OMSTR>localhost:26133:1212</OMSTR>
                        </OMATP>
                        <OMA>
                                <OMS cd="scscp1" name="procedure_completed"/>
                                <OMA>
                                        <OMS cd="list1" name="list"/>
                                        <OMI> 24</OMI>
                                        <OMI> 12</OMI>
                                </OMA>
                        </OMA>
                </OMATTR>
</OMOBJ>
#I  <?scscp end ?>
#I  Got back: object [ 24, 12 ] with attributes [ [ "call_id", "localhost:26133:1212" ] ]
rec( object := [ 24, 12 ], attributes := [ [ "call_id", "localhost:26133:1212" ] ] )
```

9.2.4 DateISO8601

◇ `DateISO8601()` (function)

Returns: string

Returns the current date in the ISO-8601 YYYY-MM-DD format. This is an internal function of the package which is used by the SCSCP server to generate the transient content dictionary, accordingly to the definition of the OpenMath symbol `meta.CDDate`.

———— Example ————

```
gap> DateISO8601();
"2009-03-31"
```

9.2.5 CurrentTimestamp

◇ `CurrentTimestamp()` (function)

Returns: string

Returns the result of the call to `date`. This is an internal function of the package which is used to add the timestamp to the SCSCP service description.

———— Example ————

```
gap> CurrentTimestamp();
"Tue 31 Mar 2009 11:19:38 BST"
```

9.2.6 Hostname

◇ `Hostname()` (function)

Returns: string

Returns the result of the call to `hostname`. This function may be used in the configuration file `scscp/config.g` to specify that the default hostname which will be used by the SCSCP server will be detected automatically using `hostname`.

```
 ─────────────────────────── Example ───────────────────────────

 gap> Hostname();
 "scscp.symbolic-computation.co.uk"
```

9.2.7 MemoryUsageByGAPinKbytes

◊ `MemoryUsageByGAPinKbytes()` (function)

   Returns: integer

   Returns the current volume of the memory used by GAP in kylobytes. This is equivalent to calling `ps -p <PID> -o vsz`, where `<PID>` is the process ID of the GAP process. This is an internal function of the package which is used by the SCSCP server to report its memory usage in the `info_memory` attribute when being called with the option `debuglevel=2` (see options in `EvaluateBySCSCP` (6.1.4) and `NewProcess` (6.1.2)).

```
 ─────────────────────────── Example ───────────────────────────

 gap> MemoryUsageByGAPinKbytes();
 649848
```

9.2.8 LastReceivedCallID

◊ `LastReceivedCallID()` (function)

   Returns: string

   Returns the call ID contained in the most recently received message. It may contain some useful debugging information; in particular, the call ID for the GAP SCSCP client and server contains colon-separated server name, port number, process ID and a random string.

```
 ─────────────────────────── Example ───────────────────────────

 gap> LastReceivedCallID();
 "scscp.symbolic-computation.co.uk:26133:77372:choDZBgA"
```

9.2.9 IO_PickleToString

◊ `IO_PickleToString(obj)` (function)

   Returns: string containing "pickled" object

   This function "pickles" or "serialises" the object `obj` using the operation `IO_Pickle` (**IO: IO_Pickle**) from the IO package, and writes it to a string, from which it could be later restored using `IO_UnpickleFromString` (9.2.10). This provides a way to design SCSCP procedures which transmit GAP objects in the "pickled" format as OpenMath strings, which may be useful for objects which may be "pickled" by the IO package but can not be converted to OpenMath or for which the "pickled" representation is more compact or can be encoded/decoded much faster.

   See `IO_Pickle` (**IO: IO_Pickle**) and `IO_Unpickle` (**IO: IO_Unpickle**) for more details.

────────────────────────── Example ──────────────────────────

```
gap> f := IO_PickleToString( GF( 125 ) );
"FFIEINTG\>15INTG\>13FAIL"
```

9.2.10 IO_UnpickleFromString

◇ IO_UnpickleFromString(*s*)   (function)

Returns: "unpickled" GAP object

This function "unpickles" the string *s* which was created using the function IO_PickleToString (9.2.9), using the operation IO_Unpickle (**IO: IO_Unpickle**) from the IO package. See IO_PickleToString (9.2.9) for more details and suggestions about its usage.

────────────────────────── Example ──────────────────────────

```
gap> IO_UnpickleFromString( f );
GF(5^3)
gap> f = IO_UnpickleFromString( IO_PickleToString( f ) );
true
```

References

[BL07]   Jost Berthold and Rita Loogen. Visualizing Parallel Functional Program Runs — Case
           Studies with the Eden Trace Viewer. In *Parallel Computing: Architectures, Algorithms
           and Applications. Proceedings of the International Conference ParCo 2007*, volume 15 of
           Advances in Parallel Computing. IOS Press, 2007. 37

[CKS]   Marco Costantini, Alexander Konovalov, and Andrew Solomon. Openmath — OpenMath
           functionality in GAP. GAP4 package (http://www.cs.st-andrews.ac.uk/~alexk/
           openmath.htm). 7

[FHK⁺a]   Sebastian Freundt, Peter Horn, Alexander Konovalov, Sylla Lesseni, Steve Linton, and
           Dan Roozemond. OpenMath content dictionary scscp1. (http://www.win.tue.nl/
           SCIEnce/cds/scscp1.html). 5, 12, 15, 19

[FHK⁺b]   Sebastian Freundt, Peter Horn, Alexander Konovalov, Sylla Lesseni, Steve Linton, and Dan
           Roozemond. Symbolic Computation Software Composability Protocol (SCSCP) specifica-
           tion, version 1.3, 2009. (http://www.symbolic-computation.org/scscp). 5, 6, 14

[FHK⁺c]   Sebastian Freundt, Peter Horn, Alexander Konovalov, Steve Linton, and Dan Rooze-
           mond. OpenMath content dictionary scscp2. (http://www.win.tue.nl/SCIEnce/cds/
           scscp2.html). 5, 12, 19

[GNO]   Greg Gamble, Werner Nickel, and Eamonn O'Brien. ANUPQ — ANU p-Quotient. GAP4
           package (http://www.math.rwth-aachen.de/~Greg.Gamble/ANUPQ/). 32

[LN]   Frank Lübeck and Max Neunhöffer. GAPDoc — A Meta Package for GAP Documentation.
           GAP4 package (http://www.math.rwth-aachen.de/~Frank.Luebeck/GAPDoc). 7

[Neu]   Max Neunhöffer. IO — Bindings for low level C library IO. GAP4 package (http://
           www-groups.mcs.st-and.ac.uk/~neunhoef/Computer/Software/Gap/io.html). 7

Index