Simulation Component and Data Coupling (SCDC)

User manual

Michael Hofmann

October 29, 2018

Contents

1 Introduction

The Simulation and Data Coupling (SCDC) library is a communication library for the technical coupling of independently developed software components in distributed computing environments. The following sections give an overview of the design of the SCDC library, the content of the software package, and its compilation and usage.

1.1 SCDC overview

A software component can be represented by any program or function available for the execution of a specific task. The library implements a service-oriented usage model where active client components access passive service components. Within this model, a software component might act in several roles, for example, by providing multiple different services or by being both a service and a client. The library design is application-independent and any interaction between a client and a services is considered as a data transfer.

To act as a service with a specific functionality, a software component provides so-called *datasets* which are managed by *data providers*. Data providers and datasets are identified with an URI-based addressing scheme as follows:

```
<access-method>://<component-id>/<data-provider>[/<dataset>]
```

`<access-method>` specifies how the service component, which provides the datasets, should be accessed and how data to and from the component should be transferred. Examples are `scdc` for accessing a local service component within the same process directly with function calls or `scdc+tcp` for accessing a remote service component in a distributed environment with TCP/IP network communication. Direct access is enabled by default for any component. Further access methods can be explicitly enabled by setting up so-called *node ports* using the corresponding library functions as described in Sect. 3.1.

`<component-id` identifies the service component to be accessed. The specific format of the identifier depends on the access method to be used. For example, direct access the local service component (i. e., method `scdc`) uses an empty identifier while accessing a remote service component with TCP/IP network communication (i. e., method `scdc+tcp`) uses the hostname (or IP address) of the computer where the remote service component is executed as identifier. Further details about specific access methods and their functionalities are described in Sect. 5.

`<data-provider>` identifies the data provider to be accessed on a service component. The identifier is an alphanumeric string (i. e., without further path hierarchy). A service component can contain several data providers distinguished by different identifiers. Data providers can be created by using the corresponding library functions as described in Sect. 3.2.

`<dataset>` identifies the dataset of a data provider to be accessed on a service component. The specific format of the identifier depends on the data provider of the dataset. Further details about specific data providers and their functionalities are described in Chap. 6.

To act as a client, a software component accesses the datasets of services by executing *commands*. Each execution of a commands can involve input data transferred from the client to the service and output data transferred back from the service to the client. Input and output data can be processed in continuous parts as a data stream, i. e. in both directions the data does not need to be processed at once in a single piece, but can be continuously provided or consumed. The corresponding library functions to be used by a client component are described in Chap. 4.

Additional information can be found in the following publications:

- M. Hofmann, F. Ospald, H. Schmidt, and R. Springer. Programming support for the flexible coupling of distributed software components for scientific simulations. In *Proceedings of the 9th International Conference on Software Engineering and Applications (ICSOFT-EA 2014)*, pages 506–511. SciTePress, August 2014.

- M. Hofmann and G. Rünger. Sustainability through flexibility: Building complex simulation programs for distributed computing systems. *Simulation Modelling Practice and Theory, Special Issue on Techniques And Applications For Sustainable Ultrascale Computing Systems*, 58(1):65–78, 2015.

1.2 Software package

The software package of the SCDC library contains the following parts:

`doc` Documentation of the library.

`doc/doxygen` Doxygen-based interface description to be generated for HTML and LaTeX.

`doc/manual/` LaTeXsources of this manual.

`src` Sources of the library.

`src/components` Internal library components.

`src/extra` External auxiliary components.

`src/include` Single directory of internal header files.

`src/lib` Library interfaces.

`src/lib/redirect` SCDC-based redirection of numerical computation and file access interfaces.

`src/lib/scdc` C interface of the SCDC library.

`src/lib/scdc.py` Python interface of the SCDC library.

`templates` Central Makefile templates.

`test` Test and demo programs.

`test/units` Tests for specific SCDC components and features.

`tools` SCDC-based tools.

1.3 Compilation and usage

Build system

The build system of the SCDC software package uses plain Makefiles. Configuration switches, paths, compilers, and flags are defined in file(s) `Makefile.in` and included in the regular Makefiles building the single parts. Makefiles are invoked recursively in the sub-directories defined in the file(s) `Makefile.local`. Building the whole library (including documentation, tests, and tools) is achieved by execution `make` in the main directory of the library.

```
$ make
```

Compiling and linking in C

All necessary files for compiling and linking against the C interface of the library are contained in the sub-directory `lib/scdc`. The sub-directory has to be added the include directories of the compiler, such that all header files within the sub-directory are available. Only header file `scdc.h` has to be included in the source files using the SCDC library, e. g.:

```
#include "scdc.h"

...

scdc_init(SCDC_INIT_DEFAULT);
...
scdc_release();
```

Compilation:

```
$ gcc -I<SCDC_DIR>/src/lib/scdc ...
```

The library file `libscdc.a` and the C++ linker (or the corresponding C++ library files) have to be used, e. g.:

```
$ g++ -L<SCDC_DIR>/src/lib/scdc ... -lscdc
```

1.3.1 Usage in Python

All necessary files for using the Python interface of the library are contained in the directory `<SCDC_DIR>/src/lib/scdc.py`. Either the directory is added to the module search path of the Python interpreter or the two files `scdc.py` and `scdcmod.so` have to be available to the Python interpreter. Importing the module `scdc` is sufficient for using all library functions, e. g.:

```
import scdc

scdc.init()
...
scdc.release()
```

2 General interface

2.1 Data types and constants

The SCDC library provides and uses definitions for the following data types and constants:

Type `scdcint_t`: Predefined data type used for (almost) all specifications of integer values (e. g., parameters, return values, . . .) throughout the entire library.

Constant `SCDC_SUCCESS`: Constant of data type `scdcint_t` signaling the success of an operation.

Constant `SCDC_FAILURE`: Constant of data type `scdcint_t` signaling the failure of an operation.

Type `scdc_nodeport_t`: Data type for storing a node port handle (see 3.1).

Constant `SCDC_NODEPORT_NULL`: Null constant for node port handles.

Type `scdc_dataprov_t`: Data type for storing a data provider handle (see 3.2).

Constant `SCDC_DATAPROV_NULL`: Null constant for data provider handles.

Type `scdc_dataset_t`: Data type for storing a dataset handle (see 4.1).

Constant `SCDC_DATASET_NULL`: Null constant for data provider handles.

2.2 Functions

Before using (almost) any library function (e. g., at the beginning of a program), the SCDC library has to be initialized using the function `scdc_init`. After the last usage of library functions (e. g., before the ending of a program), the SCDC library has to be released using the function `scdc_release`. The library should not be initialized or released multiple times in a row. (However, initializing the library again after it has been released should be possible.)

```
scdcint_t scdc_init(const char *conf, ...);
```

Parameter `conf`: String of colon-separated configuration parameters. The constant `SCDC_INIT_DEFAULT` or NULL can be used as default.

Parameter `...`: Varying number of additional parameters (see description of supported configuration parameters below).

Return: Whether the function was successful (`SCDC_SUCCESS`) or not (`SCDC_FAILURE`).

Description: Function for initializing the SCDC library. Must be called before (almost) all other library function call. Currently, the following configuration parameters are supported:

`no_direct` Disable the default direct access.

```
void scdc_release(void);
```

Description: Function for releasing the SCDC library. Must be called after (almost) all other library function calls.

Library functions usually return a value signaling their success or failure. Further (textual) information about the result of a function (e. g., a requested information or an error message) are stored internally in a static string variable. The variable can be accessed with the function `scdc_last_result` and contains the result of the last library function that was executed.

```
const char *scdc_last_result();
```

Return: Pointer to a static C string containing the last results message.

Description: Function for returning the result string of the last SCDC library function call.

The library can produce various logging information during its execution (if enabled at compile time, see configuration file `Makefile.in` of the build system described in Sect. 1.3). The default output is `stdout` for tracing information and `stderr` for error information. The output can be changed with the function `scdc_log_init` and reverted back to the default with the function `scdc_log_release`. These changes also can be made before the initialization and after the release of the library (i. e., with `scdc_init` and `scdc_release`).

```
scdcint_t scdc_log_init(const char *conf, ...);
```

Parameter `conf`: String of colon-separated configuration parameters.

Parameter `...`: Varying number of additional parameters (see description of supported configuration parameters below).

Return: Whether the function was successful (`SCDC_SUCCESS`) or not (`SCDC_FAILURE`).

Description: Function for initializing the logging. Can be called before `scdc_init`. Currently, the following configuration parameters are supported:

`log_handler` Two user-defined functions are used to output tracing and error information. For each user-defined function, a pointer to the function and a user-defined pointer (e. g., to some user data or NULL) have to be specified as additional parameters. The logging functions have to comply to the format defined by type `scdc_log_handler_f` and the corresponding pointer to the user data is provided to each function call (see description below).

`log_FILE` Two `FILE` streams are used to output tracing and error information. Pointers to these `FILE` streams have to be specified as additional parameters.

`log_filepath` Two files are used to store tracing and error information. A string `<logfilepath>` containing the path and basic file name as to be specified as additional parameter.

Tracing information are stored in file `<logfilepath>.out` and error information in file `<logfilepath>.err`.

`void scdc_log_release(void);`

Description: Function for releasing the logging. Can be called after `scdc_release`.

User-defined functions for the output of tracing and error information have to comply to the following format:
Format: `scdcint_t scdc_log_handler_f(void *data, const char *buf, scdcint_t buf_size);`

Parameter `data`: User-defined pointer previously specified together with the corresponding user-defined logging function.

Parameter `buf`: C string containing the logging information.

Parameter `buf_size`: Length of the logging information given in `buf` (without terminating null character).

Return: Number of characters written to the output.

3 Server-side functions

The functionalities provided by the SCDC library for setting up services on the server-side consist of two parts, functions for managing the node ports (i. e., the supported access methods) and the data providers (i. e., the provided services).

3.1 Node port management

Node ports represent the access methods through which a service component can be accessed by a client component. Direct access to a local service component within the same process with function calls is enabled by default. Further access methods can be enabled by opening new node ports with the function `scdc_nodeport_open`. An open node port is identified by a handle of type `scdc_nodeport_t`, which can be used for starting and stopping accepting connections with the functions `scdc_nodeport_start` and `scdc_nodeport_stop`, canceling a running node port (i. e., when started in a blocking mode) with the function `scdc_nodeport_cancel`, and to close a node port with the functions `scdc_nodeport_close`. A node port can be started and stopped multiple times as long as it is not closed. Furthermore, node ports should always be properly stopped (if started) and closed to allow for an appropriate release of utilized system resources. Details about specific access methods available are described in Chap. 5.

```
scdc_nodeport_t scdc_nodeport_open(const char *conf, ...);
```

Parameter `conf`: String of colon-separated configuration parameters.

Parameter `...`: Varying number of additional parameters.

Return: Handle of the created node port or `SCDC_NODEPORT_NULL` if the function failed.

Description: Function to open a new node port. The specific access method to be used has to be specified as the first configuration parameter given in parameter `conf`. Currently, the following access methods are supported:

`direct`: Direct function calls (see Sect. 5.2).

`uds`: Unix domain socket (see Sect. 5.3).

`tcp`: TCP/IP network communication (see Sect. 5.4).

`mpi`: Message Passing Interface (see Sect. 5.5).

```
void scdc_nodeport_close(scdc_nodeport_t nodeport);
```

Parameter `nodeport`: Handle of the node port.

Description: Function to close a node port.

`scdcint_t scdc_nodeport_start(scdc_nodeport_t nodeport, scdcint_t mode);`

Parameter `nodeport`: Handle of the node port.

Parameter `mode`: Specification of the mode for starting the node port (see description below).

Return: Whether the function was successful (`SCDC_SUCCESS`) or not (`SCDC_FAILURE`).

Description: Function to start accepting connections by a node port. The parameter `mode` has to be used to specify the mode for running the node port. Currently, the following modes specified with predefined constants are supported:

`SCDC_NODEPORT_START_NONE`: Do not run the node port.

`SCDC_NODEPORT_START_LOOP_UNTIL_CANCEL`: Run the node port in a blocking way (i. e., function blocks) until canceled with function `scdc_nodeport_cancel`.

`SCDC_NODEPORT_START_ASYNC_UNTIL_CANCEL`: Run the node port in a non-blocking way (i. e., function returns immediately) until canceled with function `scdc_nodeport_cancel`.

`scdcint_t scdc_nodeport_stop(scdc_nodeport_t nodeport);`

Parameter `nodeport`: Handle of the node port.

Return: Whether the function was successful (`SCDC_SUCCESS`) or not (`SCDC_FAILURE`).

Description: Function to stop accepting connections by a node port.

`scdcint_t scdc_nodeport_cancel(scdc_nodeport_t nodeport, scdcint_t interrupt);`

Parameter `nodeport`: Handle of the node port.

Parameter `interrupt`: Specifies whether the cancellation should be performed “hard” (1) or “soft” (0).

Return: Whether the function was successful (`SCDC_SUCCESS`) or not (`SCDC_FAILURE`).

Description: Function to cancel accepting connections by a node port (i. e., if the node port was started in in a blocking mode).

The following auxiliary functions are provided to ease the handling of node port-specific parts of URI addresses. The function `scdc_nodeport_authority` can be used to construct the authority part of an URI address for a specific access method. The function `scdc_nodeport_supported` can be used to test whether the library supports the specific access method given by an URI address.

`const char *scdc_nodeport_authority(const char *conf, ...);`

Parameter `conf`: String of colon-separated configuration parameters.

Parameter `...`: Varying number of additional parameters.

Return: Pointer to a static string containing the authority information or NULL if the function failed.

Description: Function to construct the authority information of an URI address for a given node port and its properties.

```
scdcint_t scdc_nodeport_supported(const char *uri, ...);
```

Parameter `uri`: String containing the pattern of the URI address. The string can include placeholders "`%s`" that are replaced with strings given as additional parameters.

Parameter `...`: Varying number of additional parameters.

Return: Whether the node port is supported (`SCDC_SUCCESS`) or not (`SCDC_FAILURE`).

Description: Function to determine whether a node port specified by the scheme of a given URI address is supported.

3.2 Data provider management

Data providers represent the services provided by a service component. A new service can be created by opening a new data provider with the function `scdc_dataprov_open`. An open data provider is identified by a handle of type `scdc_dataprov_t`, which can be used to close a data provider with the functions `scdc_dataprov_close`. A service component can provide multiple services through different data providers. Each data provider is identified by a unique base path within the URI-based addressing scheme. Data providers should always be properly closed to allow for an appropriate release of utilized system resources. Details about specific data providers available are described in Chap. 6.

```
scdc_dataprov_t scdc_dataprov_open(const char *base_path, const char *conf, ...);
```

Parameter `base_path`: String specifying the base path for the URI address of the data provider.

Parameter `conf`: String of colon-separated configuration parameters.

Parameter `...`: Varying number of additional parameters.

Return: Handle of the created data provider or `SCDC_DATAPROV_NULL` if the function failed.

Description: Function to open a new data provider. The specific data provider to be used has to be specified as the first configuration parameter given in parameter `conf`. Currently, the following data provider are supported:

`hook`: Arbitrary functionality with user-defined hook functions (see Sect. 6.1).

`access`: File system storage with different backends (see Sect. 6.2).

`store`: Non-hierarchical folder-oriented storage with different backends (see Sect. 6.3).

```
void scdc_dataprov_close(scdc_dataprov_t dataprov);
```

Parameter `nodeport`: Handle of the data provider.

Description: Function to close a data provider.

4 Client-side functions

4.1 Datasets

Datasets represent the basic elements which are provided by service components and can be individually accessed by client components. Specific datasets are identified by their URI address (see Sect. 1.1). A client can open a dataset with the function `scdc_dataset_open`, execute one or several commands on the dataset with the function `scdc_dataset_cmd`, and close the dataset with the function `scdc_dataset_close`. Each execution of a commands can involve input and output data transferred between the client and the service. The specification of the corresponding input and output data objects is described in Sect. 4.2. The specific commands that can be executed on the datasets are described for the different data providers in Chap. 6.

`scdc_dataset_t scdc_dataset_open(const char *uri, ...);`

Parameter `uri`: String containing the URI address.

Parameter `...`: Varying number of additional parameters.

Return: Handle of the opened dataset or `SCDC_DATASET_NULL` if the function failed.

Description: Function to the open a dataset with a given URI address. The authority of the URI address can be equal to the “%s” placeholder in which case it is replaced with the string given as additional parameter.

`void scdc_dataset_close(scdc_dataset_t dataset);`

Parameter `dataset`: Handle of the dataset.

Description: Function to close a dataset.

`scdcint_t scdc_dataset_cmd(scdc_dataset_t dataset, const char *cmd, scdc_dataset_input_t *input, scdc_dataset_output_t *output, ...)`

Parameter `dataset`: Handle of the dataset or `SCDC_DATASET_NULL` (see description below).

Parameter `cmd`: String containing the command to be executed.

Parameter `input`: Input object with data transferred from client to server (see Sect. 4.2).

Parameter `output`: Output object with data transferred from server to client (see Sect. 4.2).

Parameter `...`: Varying number of additional parameters.

Return: Whether the function was successful (`SCDC_SUCCESS`) or not (`SCDC_FAILURE`).

Description: Function to execute a command on a dataset. The dataset handle can be

`SCDC_DATASET_NULL` in which case the parameter `cmd` has to start with an URI address specifying the dataset to be used (i. e. to execute a single command without explicitly opening and closing a dataset). The authority of the URI address can be equal to the "%s" placeholder in which case it is replaced with the string given as additional parameter.

4.2 Dataset input/output

4.2.1 Input/output object structure

The input and output data of commands executed on datasets with the function `scdc_dataset_cmd` is represented by objects of type `scdc_dataset_input_t` and `scdc_dataset_output_t`. Both types are structures with the following fields:

`char format[SCDC_FORMAT_MAX_SIZE]`: String describing the format of the data (to be) transferred.

`scdc_buf_t buf`: Buffer containing the data (to be) transferred (see Sect. 4.2.2).

`scdcint_t total_size`: Currently know total size of the data (to be) transferred (bytes).

`char total_size_given`: Specifies how the given value in `total_size` should be interpreted. Predefined values are `SCDC_DATASET_INOUT_TOTAL_SIZE_GIVEN_EXACT`, `SCDC_DATASET_INOUT_TOTAL_SIZE_G`
`SCDC_DATASET_INOUT_TOTAL_SIZE_GIVEN_AT_MOST`, `SCDC_DATASET_INOUT_TOTAL_SIZE_GIVEN_NONE`.

`scdc_dataset_inout_next_f *next`: Pointer to a next function that produces or consumes more input/output. The next function reads or modifies the fields of the provided dataset input/output object.

`void *data`: Pointer to store arbitrary information within a dataset input/output object.

`scdc_dataset_inout_intern_t *intern`: Pointer to an internally used field of data. (Should only be initialized with `NULL`, but not further modified.)

`void *intern_data`: Pointer to internally used data. (Should only be initialized with NULL, but not further modified.)

The following auxiliary functions are provided to initialize all fields of an input or output object with safe "empty" values:

```
void scdc_dataset_input_unset(scdc_dataset_input_t *input);
```

Parameter `input`: Input object.

Description: Auxiliary function to reset all fields of an input object to a (null-like) value.

```
void scdc_dataset_output_unset(scdc_dataset_output_t *output);
```

Parameter `output`: Output object.

Description: Auxiliary function to reset all fields of an output object to a (null-like) value.

4.2.2 Data buffer specification

The data buffer of an input or output object is represented by an object of type `scdc_buf_t`, which is a structure with the following fields:

- **`void *ptr`:** Pointer to the memory location of the buffer.
- **`scdcint_t size`:** Size of the buffer (bytes).
- **`scdcint_t current`:** Current size of the data within the buffer (bytes).

The following preprocessor macros are provided to easy the access to the data buffer fields of an input or output object (i. e. parameter `_inout_` of the macros):

- **`SCDC_DATASET_INOUT_BUF_PTR(_inout_)`:** Return the memory location of a (single) data buffer in an input/output object.
- **`SCDC_DATASET_INOUT_BUF_SIZE(_inout_)`:** Return the size of the memory location of a (single) data buffer in an input/output object.
- **`SCDC_DATASET_INOUT_BUF_CURRENT(_inout_)`:** Return the current data size of a (single) data buffer in an input/output object.
- **`SCDC_DATASET_INOUT_BUF_SET_P(_inout_, _p_)`:** Set the memory location of a (single) data buffer in an input/output object to value `_p_`.
- **`SCDC_DATASET_INOUT_BUF_GET_P(_inout_)`:** Get the memory location of a (single) data buffer in a dataset input/output object (or NULL in case of a multiple data buffer).
- **`SCDC_DATASET_INOUT_BUF_SET_S(_inout_, _s_)`:** Set the size of the memory location of a (single) data buffer in a dataset input/output object to value `_s_`.
- **`SCDC_DATASET_INOUT_BUF_GET_S(_inout_)`:** Get the size of the memory location of a (single) data buffer in a dataset input/output object (or 0 in case of a multiple data buffer).
- **`SCDC_DATASET_INOUT_BUF_SET_C(_inout_, _c_)`:** Set the current data size of a (single) data buffer in a dataset input/output object to value `_c_`.
- **`SCDC_DATASET_INOUT_BUF_GET_C(_inout_)`:** Get the current data size of a (single) data buffer in a dataset input/output object (or 0 in case of a multiple data buffer).

4.2.3 Data streams

Input and output objects contain usually only a single data buffer that can store a fixed size input or output data. To provide a continuous stream of input and output data, the input/output objects can also contain a function pointer to a so-called next function (see field `next` Sect. 4.2.1). A next function has to comply to the following format:
Format: `scdcint_t scdc_dataset_inout_next_f(scdc_dataset_inout_t *inout, scdc_result_t *result);`

- **Parameter `inout`:** Dataset input or output that is or should be continued with the next function.

Parameter `result`: Structure for storing a result message of the next function.

Return: Whether the next function was successful (`SCDC_SUCCESS`) or not (`SCDC_FAILURE`).

The next function of an input object is responsible for creating further input data. The corresponding input object is given as the first parameter of the next function such that the object itself (i. e. its fields) can be modified by the next function. This includes, for example, the data buffer of the input object to provide further input data. To end the stream of input data, the function pointer of the next function within the input objects has to be set to NULL. The next function of an input object has to be specified by the user and the function is executed inside the call to `scdc_dataset_cmd` as long as the function pointer is not NULL.

The next function of an output object is responsible for creating further output data. The corresponding output object has to be specified as the first parameter of the next function such that the object itself (i. e. its fields) can be modified by the next function. This includes, for example, the data buffer of the output object to provide further output data. The end of the stream of output data is reached when the function pointer of the next function within the output objects is set to NULL. The next function of an output object is set after the call to `scdc_dataset_cmd` and has to be executed by the user as long as the function pointer is not NULL.

The following auxiliary macros are provided to ease the usage of next functions:
`scdc_dataset_input_next(_in_, _res_)`

Parameter `_in_`: Pointer to a input object that should be continued with the next function.

Parameter `_res_`: Pointer to a `scdc_result_t` structure for storing a result message of the next function.

Return: Whether the next function was successful (`SCDC_SUCCESS`) or not (`SCDC_FAILURE`).

Description: Macro to execute the next function of input object `_in_` one times.

`scdc_dataset_output_next(_out_, _res_)`

Parameter `_out_`: Pointer to a output object that should be continued with the next function.

Parameter `_res_`: Pointer to a `scdc_result_t` structure for storing a result message of the next function.

Return: Whether the next function was successful (`SCDC_SUCCESS`) or not (`SCDC_FAILURE`).

Description: Macro to execute the next function of output object `_out_` one times.

`scdc_dataset_input_next(_in_, _res_)`

Parameter `_in_`: Pointer to a input object that should be continued with the next function.

Parameter `_res_`: Pointer to a `scdc_result_t` structure for storing a result message of the next function.

Return: Whether the next function was successful (`SCDC_SUCCESS`) or not (`SCDC_FAILURE`).

Description: Macro to to execute the next function of input object `_in_` until it is finished or an error occurs.

`scdc_dataset_output_next(_out_, _res_)`

Parameter `_out_`: Pointer to a output object that should be continued with the next function.

Parameter `_res_`: Pointer to a `scdc_result_t` structure for storing a result message of the next function.

Return: Whether the next function was successful (`SCDC_SUCCESS`) or not (`SCDC_FAILURE`).

Description: Macro to execute the next function of output object `_out_` until it is finished or an error occurs.

4.2.4 Auxiliary functions

4.3 Configuration access

5 Nodeport reference

5.1 General functionalities

5.1.1 Server-side setup

The following configuration switches and parameters have to be provided for opening a new nodeport with the following function:

`scdc_nodeport_open("...[:<config_switches>]", <params>);`

Mandatory configuration switches: none

Mandatory parameters: none

Optional configuration switches and parameters:

`max_connections` Integer specifying the maximum number of concurrent connections the node port can handle.

5.1.2 Client-side usage

URI scheme: `scdc[+<scheme_suffix>]:[//<authority>]/<path>`

5.2 Direct: Function calls

5.2.1 Server-side setup

The following configuration switches and parameters have to be provided for opening a new nodeport with the following function:

`scdc_nodeport_open("direct[:<config_switches>]", <params>);`

Mandatory configuration switches: none

Mandatory parameters:

1. String containing an alphanumeric name identifying the node port. The identifier has to be used as the authority part of an URI to gain access through the corresponding node port.

Optional configuration switches and parameters: none

5.2.2 Client-side usage

URI scheme(s): `scdc:[//<id>]/<path>`

5.3 UDS: Unix domain socket

5.3.1 Server-side setup

The following configuration switches and parameters have to be provided for opening a new nodeport with the following function:

```
scdc_nodeport_open("uds[:<config_switches>]", <params>);
```

Mandatory configuration switches: none

Mandatory parameters: none

Optional configuration switches and parameters:

`socketname` String containing an alphanumeric name identifying the socket to be used for the node port. The identifier has to be used as the authority part of an URI to gain access through the corresponding node port.

`socketfile` String specifying the socket file within the local file system to be used for the node port. The identifier has to be used as the authority part of an URI to gain access through the corresponding node port.

5.3.2 Client-side usage

URI scheme(s): `scdc+uds:[//<authority>]/<path>`
`<authority> = [socketname:]<socketname> | socketfile:<socketfile>`

5.4 TCP: TCP/IP network socket

5.5 MPI: Message Passing Interface

6 Data provider reference

6.1 Hook: User-defined functionality

- data provider with arbitrary functionality defined by the user through hook functions

6.1.1 Server-side setup

hook functions that can be assigned to members of the structure `scdc_dataprov_hook_t`, format and role of the specific functions:

`open` Open the data provider (optional):
Format: `void *scdc_dataprov_hook_open_f(const char *conf, va_list ap)`
Python: `def scdc_dataprov_hook_open_f(conf, *args):`
`return dataprov|False`

`close` Close the data provider (optional):
Format: `scdcint_t scdc_dataprov_hook_close_f(void *dataprov)`
Python: `def scdc_dataprov_hook_close(dataprov):`
`return bool`

`config` Configure the data provider (optional):
Format: `scdcint_t scdc_dataprov_hook_config_f(void *dataprov,`
`const char *cmd, const char *param,`
`const char *val, scdcint_t val_size,`
`scdc_result_t *result)`
Python: `def scdc_dataprov_hook_config(dataprov, cmd, param, val):`
`return bool` *or* `(bool, result)`

`dataset_open` Open a dataset (optional):
Format: `void *scdc_dataprov_hook_dataset_open_f(void *dataprov,`
`const char *path,`
`scdc_result_t *result)`
Python: `def scdc_dataprov_hook_dataset_open(dataprov, path):`
`return dataset|False` *or* `(dataset|False, result)`

`dataset_close` Close a dataset (optional):

Format: `scdcint_t scdc_dataprov_hook_dataset_close_f(void *dataprov,`
`void *dataset,`
`scdc_result_t *result)`
Python: `def scdc_dataprov_hook_dataset_close(dataprov):`
`return bool` *or* `(bool, result)`

`dataset_close_write_state` Close a dataset and write its state to a string (optional):
Format: `scdcint_t scdc_dataprov_hook_dataset_close_write_state_f(void *dataprov,`
`void *dataset,`
`void *state, scdcint_t state_size,`
`scdc_result_t *result)`
Python: `def scdc_dataprov_hook_dataset_close_write_state(dataprov, dataset):`
`return str|False` *or* `(str|False, result)`

`dataset_open_read_state` Open a dataset by reading its state from a string (optional):
Format: `void *scdc_dataprov_hook_dataset_open_read_state_f(void *dataprov,`
`const void *state, scdcint_t state_size,`
`scdc_result_t *result)`
Python: `def scdc_dataprov_hook_dataset_open_read_state(dataprov, state):`
`return dataset|False` *or* `(dataset|False, result)`

`dataset_cmd` Execute a command on a dataset:
Format: `scdcint_t scdc_dataprov_hook_dataset_cmd_f(void *dataprov,`
`void *dataset,`
`const char *cmd, const char *params,`
`scdc_dataset_input_t *input, scdc_dataset_output_t *output,`
`scdc_result_t *result)`
Python: `def scdc_dataprov_hook_dataset_cmd(dataprov, dataset, cmd, params, input,`
`output):`
`return bool` *or* `(bool, result)`

6.1.2 Client-side usage

Command: `<cmd> [<param>]`

Usage: Execute command `<cmd>` with parameters `<param>`.

Input: Input of the command.

Output: Output of the command.

Failure: The command executed by the `dataset_cmd` hook function failed.

6.1.3 Configuration

Any configuration performed by the `config` hook function.

6.2 Access: File system storage

- file system storage
- backends: local file system, NFS, WebDAV

6.2.1 Server-side setup

`access:fs` Parameter: root directory

`access:nfs` Parameter: NFS export URL `nfs://<host>/<path>`

`access:webdav[:username:password]` Parameter: WebDAV server URL `http://<host>[:port]/<path>` if the optional config switches username and password are given, then the user name and the password have to given as separate string parameters

6.2.2 Client-side usage

Command: `cd [|<dir>]`

Usage: Change to directory given by `<dir>`.

Failure: Store `<dir>` does not exist.

Command: `ls`

Usage: List entries of the current directory.

Result: Number and list of entries separated by "|" character.

Command: `info [|<entry>]`

Usage: General information about the current directory or the given directory/file <entry>.

Result: Directory or file information.

Command: `mkd <dir>`

Usage: Create directory <dir> within the current directory.

Failure: Directory `<dir>` could not be created.

Command: `put <file> [<pos>][:<size>]`

Usage: Write data from the command input to the file `<file>` of the current directory. Data is written at the beginning of the file or (optionally) at position `<pos>`. Either all input or (optionally) at most `<size>` bytes are written. If the entry already exists, its data is overwritten. If the entry does not exists, it is created.

Input: Data to be written to the file.

Failure: File `<file>` could not be created or data write failed.

Command: `get <file> [<pos>][:<size>]`

Usage: Read data from the file `<file>` of the current directory into the command output. Data is read at the beginning of the file or (optionally) at position `<pos>`. Either all data or (optionally) at most `<size>` bytes are read. If the file does not exists, then the command fails.

Output: Data read from the file.

Failure: Entry `<entry>` does not exist or data read failed.

Command: `rm [<entry>]`

Usage: Remove the directory/file <entry> of the current directory.

Failure: Directory/file `<entry>` does not exist or could not be removed.

6.2.3 Configuration

`access:fs` none

6.3 Store: Non-hierarchical folder-oriented storage

- non-hierarchical folder-oriented storage
- backends: memory, local file system, MySQL, NFS, WebDAV

6.3.1 Server-side setup

`store:mem` Parameter: none

`store:fs` Parameter: root directory

`store:nfs` Parameter: NFS export URL `nfs://<host>/<path>`

`store:webdav[:username:password]` Parameter: WebDAV server URL `http://<host>[:port]/<path>`
if the optional config switches username and password are given, then the user name and the password have to given as separate string parameters

6.3.2 Client-side usage

Command: `cd [|ADMIN|<store>]`

Usage: Change to administration mode using no parameter or parameter `ADMIN`. Select a single store given by `<store>`.

Failure: Store `<store>` does not exist.

Administration mode

Command: `info`

Usage: General information about the stores.

Result: Store information.

Command: `ls`

Usage: List all available stores.

Result: Number of stores and list of store names separated by “|” character.

Command: `put <store>`

Usage: Create a store `<store>`.

Failure: Store `<store>` could not be created.

Command: `rm <store>`

Usage: Remove the store `<store>`.

Failure: Store `<store>` does not exist or could not be removed.

Single store selected

Command: `info`

Usage: General information about the selected store.

Result: Store information.

Command: `ls`

Usage: List all available entries of the selected store.

Result: Number of entries and list of entry names separated by “|” character.

Command: `put <entry> [<pos>][:<size>]`

Usage: Write data from the command input to the entry `<entry>` of the selected store. Data is written at the beginning of the entry or (optionally) at position `<pos>`. Either all input or (optionally) at most `<size>` bytes are written. If the entry already exists, its data is overwritten. If the entry does not exists, it is created.

Input: Data to be written to the entry.

Failure: Entry `<entry>` could not be created or data write failed.

Command: `get <entry> [<pos>][:<size>]`

Usage: Read data from the entry `<entry>` of the selected store into the command output. Data is read at the beginning of the entry or (optionally) at position `<pos>`. Either all data or (optionally) at most `<size>` bytes are read. If the entry does not exists, then the command fails.

Output: Data read from the entry.

Failure: Entry `<entry>` does not exist or data read failed.

Command: `rm <store>`

Usage: Remove the entry `<entry>` of the selected store.

Failure: Entry `<entry>` does not exist or could not be removed.

6.3.3 Configuration

`store:mem` none

`store:fs` none

7 Storage and compute redirection

7.1 POSIX file access and C++ file streams

7.2 Numerical libraries

8 Demonstration examples and tests

8.1 Demo: Three components