Technical University Berlin

Telecommunication Networks Group

Remote Socket Architecture Service and Interface of the Last Hop Protocol for TCP

Morten Schläger and Tobias Poschwatta

{morten,posch}@ee.tu-berlin.de

Berlin, October 2001

TKN Technical Report TKN-01-016

TKN Technical Reports Series

Editor: Prof. Dr.-Ing. Adam Wolisz

Abstract

The Remote Socket Architecture is a RPC-like implementation of the well known BSD socket interface for sockets of the AF_INET family. Its purpose is to improve the quality (e.g.: performance) of wireless Internet access. It is composed of two layer hierarchy. The higher layer is technology independent and implements the export of the socket interface while maintaining its syntax and semantic. The lower layer is technology dependent and is responsible to provide the required transmission service and addressing functionality. The service of the lower layer depends on the requirements of the application. In case of TCP based application the lower layer must implement a reliable service while it can implement a semi-reliable or unreliable service in case of UDP based applications.

This documents describes the service which is expected from the lower layer in case TCP sockets are considered as well as the interface between these two layers.

Contents

1 Introduction **2**

2 Communication Subsystem **4**

2.1 Service of the CS 4
2.2 Addressing 5
2.3 Abstract Interface Definition 6
2.3.1 Registration 7
2.3.2 Connection Management 8
2.3.3 Data Transfer 8
2.3.4 Miscellaneous 9
2.4 CS API Definition 10
2.4.1 Implementation of the interface 10
2.4.2 ReSoA runtime environment 12
2.4.3 Functions implemented by the service provider 13
2.4.4 Usage of the API 17

A CS API Headerfile **19**

B Error Codes (excerpt from include/asm-i386/errno.h) **25**

Copyright at Technical University Berlin. All Rights reserved.

Chapter 1

Introduction

The *Remote Socket Architecture* is a RPC-like implementation of the Berkeley socket interface, supporting sockets of the AF_INET family. Its purpose is to improve the quality of wireless Internet access, especially regarding performance in case of TCP. In case of UDP the desired improvement is depended on the application.

ReSoA separates the operating system functionality into a communication stub at the end-system and the actual Internet protocol stack within an access-point or edge router. The end-system is called *Remote Socket Client (RSC)* and the access point or edge router is called *Remote Socket Server (RSS)*. The architecture is divided into two layers as shown in figure 1.1. The higher layer, called the *Socket Export Layer (SEL)*, implements the functionality of the socket interface. For this purpose it holds instances of *Local Socket Modules (LSM)* on the RSC and *Remote Socket Modules (RSM)* on the RSS. Each BSD socket is implemented by a LSM/RSM pair. LSM and RSM communicate with each user using the *Export Protocol (EP)*.

The EP relies on the service provided by the *Communication Subsystem (CS)* to exchange messages between each other. The CS is ReSoA's second layer. Although from ReSoA's point of view the CS is a single layer, it will mostly be composed out of several layers. It contains technology dependent protocols (e.g.: in case of 802.11 WLAN a MAC protocol) and a *Last Hop Protocol (LHP)*. The LHP is responsible to adapt the service of the applied technology to the demands of ReSoA. The LHP is not only dependent from the used technology but also from the application. As an example figure 1.1 shows three application. The FTP-application requires a reliable service, hence uses a TCP socket, while the VoIP und MPEG 4 application are delay-sensetive and therefore are using a UDP socket.

To support this service requirements different LHPs are needed. In case of a TCP socket the LHP must provide a reliabe service. In case of a UDP socket a reliable LHP could reduce the quality of service seen by the application. As illustraded by the figure flow-specific LHPs could be deployed.

However, this document deals only with TCP support. The purpose of this document is to describe the service which the Socket Export Layer expects form the CS as well as to describe the interface between these two layers in case the application uses TCP sockets.

Copyright at Technical University Berlin. All Rights reserved.

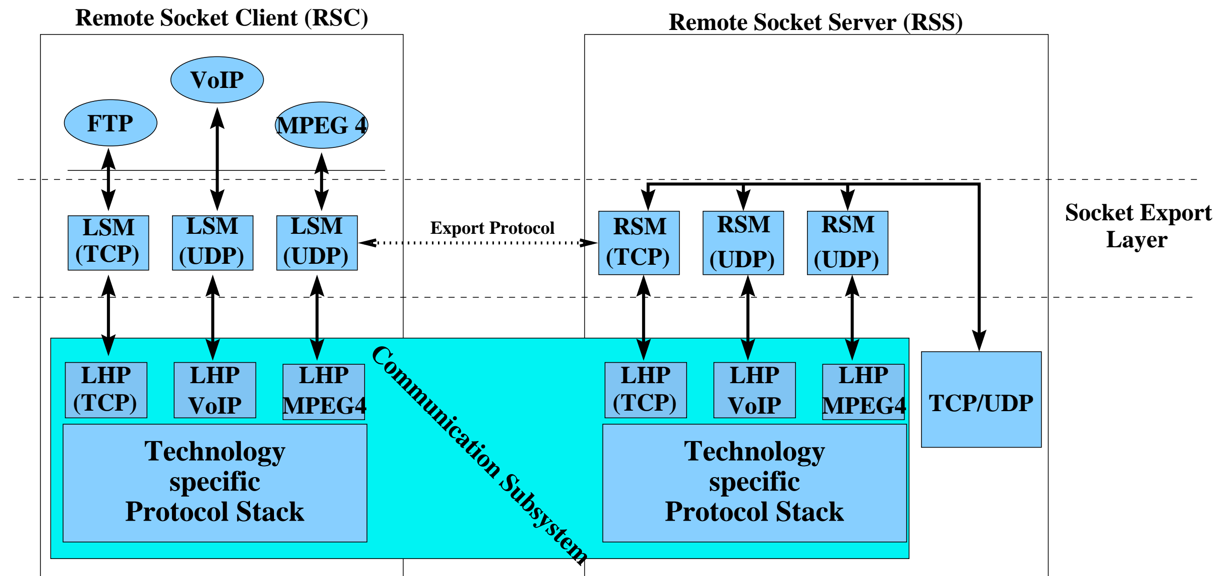

Figure 1.1: Basic components of ReSoA

In this document the term *service user* always refers to either LSM or RSM. Only in case a distinction is needed we use LSM or RSM instead of service user. The term *service provider* always refers to a CS entity or to the implementor of the CS. Finally the term *interface* always means the interface between the service user and the service provider, thus between the two layers of ReSoA.

Copyright at Technical University Berlin. All Rights reserved.

Chapter 2

Communication Subsystem

The communication subsystem is responsible for delivery of messages between RSC and RSS. It is technology and application dependent and is composed out of a technology specific protocol stack and an adaption layer which is called *Last Hop Protocol (LHP)*. The LHP is responsible to implement the interface between LSM/RSM and CS as well as to improve the service of the technology specific protocol stack to meet the requirements of ReSoA. In case a specific technology already meets the requirements of ReSoA the LHP must only implement the interface. Although the Export Socket Layer demands a specific service from the CS, the CS provider is free to implement an arbitrary protocol stack as long as the chosen protocol stack meets the service specification given in section 2.1.

Since the service and not the protocols of the CS are important for ReSoA this chapter describes only the service and the interface to the CS. However, please note that the design of the CS decides about the achievable performance. The chapter is divided into three parts. We start with the service description. Then we give an abstract definition of the service interface and finally we present an API.

2.1 Service of the CS

The service provided by the CS is connection oriented and accessible through a single service access point as shown in figure 2.1. The service user (either LSM or RSM) creates a local communication end-point upon initialization and closes this communication endpoint at the end of the session (e.g. when the computer is shutdown, or when the ReSoA module is removed). The RSM, in addition, creates a new communication endpoint for every client it support. The shutdown of a communication endpoint can either be graceful or abrupt. In the former case the CS must make sure that both the local send and local receive queues are empty before the communication endpoint is released. In case of a abrupt connection termination the CS must discard all queued messages.

The core functionality of the CS is reliable transport of messages (CS-SDUs) between RSC and RSS as well as addressing. Reliability here means that every message passed to the CS is delivered exactly once in the correct order to the service user at the peer node. Further, the CS is responsible for the detection of transmission errors. All messages delivered to the receiving service user should be error free (at least with a high (state of the art) probability).

Copyright at Technical University Berlin. All Rights reserved.

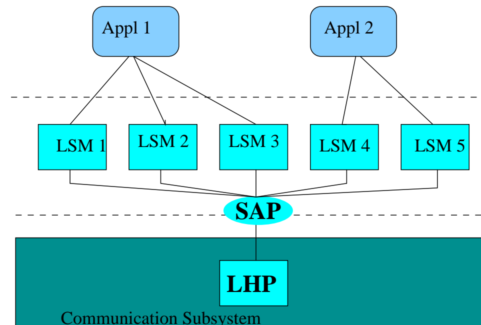

Figure 2.1: One to one relation between LSM and CS instances

It is CS's responsibility to chose and implement the protocol mechanisms like error control, flow control, which are appropriate to offer this service in an efficient manner. The design of the CS has a high impact on the performance of the whole system.

Beside the reliable service the CS should support an un- or semi reliable service. This will be used for sockets which are attached to unreliable protocols like UDP sockets.

The CS must prevent message boundaries and must be able to accept CS_SDUs of a length up to 65600 bytes [1][2]. The LHP must support segmentation and reassembly in case the deployed technology is not able to carry messages of such length.

Since the service user needs to protect some of her requests by timers to avoid deadlocks but is not able to measure the RTT between the RSC and RSS due to a very limited number of bidirectional data traffic the CS is responsible for estimating the RTT of CS-SDUs. The measured value must be offered to the service user on request.

Further, it is CS's task to note when the peer(s) are not longer available/reachable. In case that the CS detects an interception in the connectivity of certain length it has to inform the service user. Thus, the CS must implement some kind of keep-alive functionality.

Finally, the CS has to inform the service user about any kind of errors like that it was not able to deliver a message or that the peer entity has reseted a connection.

2.2 Addressing

The CS must provide an addressing scheme which allows the RSC and RSS to address each other. For example, IP addresses could be used if the CS were IP based. However, the implementation of the RSM and RSS are address format independent, which opens the question how can they address each other when they do not know the addressing format.

[1] The size of data chunks passed via the socket interface is limited by the length parameter of the socket interface. This parameter allows for passing data chunks of up to 2^{32} bytes. However, most protocols limit this size to 65535 bytes. Therefore we decided to support only the latter size. Since the LHP-PDU must carry the data of the socket call as well as the header of the Export Protocol the maximal size of a LHP-SDU must be larger than 65535.

[2] This figure is of theoretical interest only, as long as one considers only prototypes. In case of a smaller SDU size one must make sure that test applications do not try to send too much data with a single call.

Copyright at Technical University Berlin. All Rights reserved.

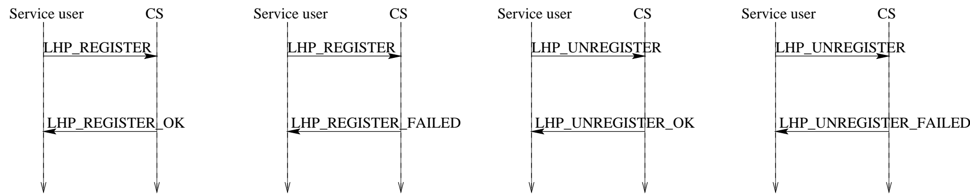

Figure 2.2: LHP service primitives for registering

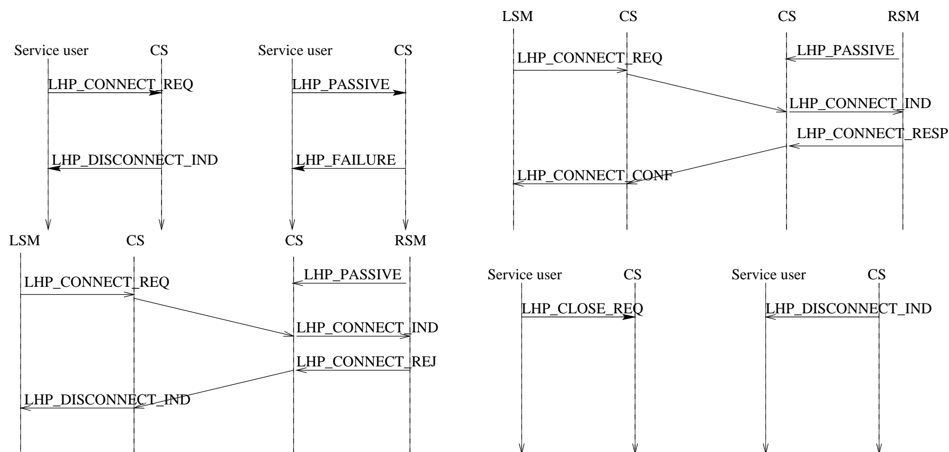

Figure 2.3: LHP service primitives for connection management

To solve this problem the LSM and RSM read their local addresses in a coded representation from some configuration file. The exact procedure how to obtain the addresses is implementation dependent and not part of the abstract interface description. In addition to this the LSM must also read the address of an available RSS from a second file. In order to allow for an (relative) address format independent design the LSM and RSM just read the binary coded address without interpreting them. For example in case of an IP based CS the LSM would read two 4 byte integers in network byte order.

The maximum size of such an encoded address is defined by the interface.

2.3 Abstract Interface Definition

Figure 2.3 through 2.5 show the available service primitives. Beside the classical primitives for connection management and data communication some special primitives are defined to collect statistics or to implement an interface flow control. An important property of the interface is that no message may get lost, since the service user does not implement an own error control. To achieve this an interface flow control is needed.

The abstract interface specification does not define whether the interface is synchronous or asynchronous. However, some of the request need some kind of confirmation as indicated in figure 2.3 through 2.5. This is described in more detail below.

Copyright at Technical University Berlin. All Rights reserved.

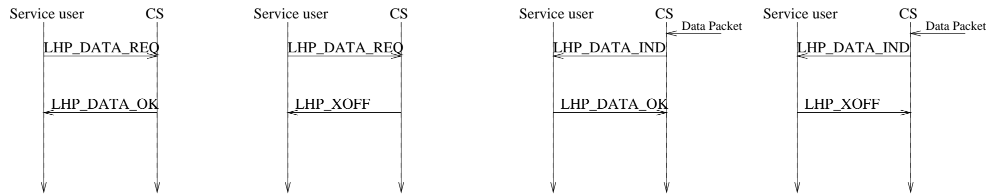

Figure 2.4: LHP service primitives for data exchange

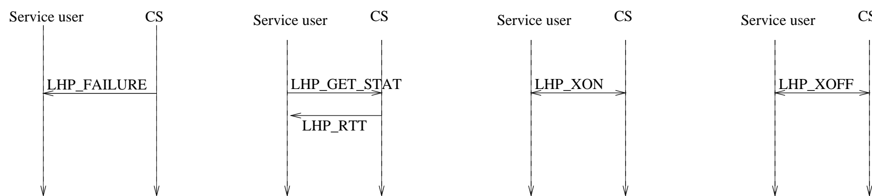

Figure 2.5: LHP service primitives - Miscellaneous

The different primitives can be classified into four categories, namely communication endpoint management, connection-management, data transfer and miscellaneous.

2.3.1 Registration

LHP_REGISTER

The `LHP_REGISTER` primitive is used to create a new local communication end-point by a service user. This primitive should not trigger any data exchange between the peer hosts. In case the primitive succeeds then a connection identifier is returned using the `LHP_REGISTER_OK`-primitive. This identifier must be used for all later requests belonging to this communication endpoint. In case the communication end-point could not be created an error reason is returned using the `LHP_REGISTER_FAILED`. With the `LHP_REGISTER` primitive the service user must specify which service (e.g. reliable,semi-, un- reliable) it requests and which provider is asked to provider the service[3]. In case either the service or the provider are unknown the creation of the new endpoint fails.

LHP_UNREGISTER

The `LHP_UNREGISTER` primitive is used to a release a communication endpoint. If the communication endpoint is valid then the service provider confirms the request using the `LHP_UNREGISTER_OK` primitive, otherwise the service user answers with the `LHP_UNREGISTER_FAILED` primitive.

[3]Currently, only a reliable service must be supported.

Copyright at Technical University Berlin. All Rights reserved.

2.3.2 Connection Management

Connection Establishment

A connection is either active or passive established. The active end initiates the establishment of a new connection using the LHP_CONNECT_REQ primitive while the passive end (the RSS is waiting for incoming connections) of a connection uses the LHP_PASSIVE-primitive to inform the service provider that it should accept incoming connection requests.

When a CS-entity receives a connection request (initial packet of a new connection) and when it is set to accept incoming connections it informs its service user about the new connection using the LHP_CONNECT_IND-primitive. If the CS was not configured to accept new connections it must deal with the request locally, without notifying the service user (e.g.: send a reset packet).

After the passive end of a CS connection has received a connection request from its peer entity, it must create a new communication end-point for this new connection. The old communication end-point must stay ready to process further incoming connection requests.

The service user (at the passive end) can either accept a new connection using the LHP_CONNECT_RESP primitive or reject it using the LHP_CONNECT_REJ primitive. In the former case the CS sends a connection confirmation to its peer entity. In the latter case it sends a disconnect indication. The initiator of a CS connection is informed with LHP_CONNECT_CONF-primitive when the connection establishment is completed. If the LHP could not fulfill a connect request, then it informs its service user using the LHP_DISCONNECT_IND-primitive. This can either mean that the peer CS entity has rejected the connection (the local CS has received something like a reset packet) or that the initiating CS has received no response at all after a number of retransmissions.

Connection Release

To release a connection three primitives are available. The LHP_CLOSE_REQ-primitive is used to initiate a connection release, while the LHP_DISCONNECT_IND primitive is used to inform the service user about the fact that a CS connection was terminated. In general, only the LSM (RSC) should use this primitive.

The LHP_RESET-primitive should be used by the service user to trigger an abrupt connection release. Upon this request the CS should send some kind of reset packet rather then to start a normal connection termination sequence.

2.3.3 Data Transfer

LHP_DATA_REQUEST

The LHP_DATA_REQUEST primitive is used to request the transparent delivery of a CS_SDU. The SDU can have an arbitrary length up to 65600 octets.

The CS responds to a data request with one of the following primitives

- LHP_DATA_OK: This response indicates success. A positive result, however, does not mean that the message was delivered to the peer nor that the request was consumed

Copyright at Technical University Berlin. All Rights reserved.

by the peer service user but only that the request was accepted. The service user is not informed when the message was successfully delivered.

- `LHP_XOFF`: This response tells the service user that the CS currently is not able to accept further requests. However, this is only temporary. The CS will inform the service user when it is able to process further requests using the `LHP_XON` primitive.

LHP_DATA_INDICATION

The `LHP_DATA_INDICATION` primitive is used to deliver messages to the service user.

All messages for which the service user on the peer host has requested reliable delivery must be delivered in the same order as they have been sent. It is the task of the CS to preserve the message boundaries. A message must not be removed from the receive buffer before it was successfully delivered to the service user. Thus no messages may be lost at the interface.

The `LHP_DATA_INDICATION` primitive is acknowledged with one of the following primitives.

- LHP_DATA_OK to indicate success.
- LHP_XOFF to signal that the service user is at the moment not able to accept the message. In this case the service user will later trigger the service provider to deliver all queued messages.

2.3.4 Miscellaneous

LHP_GET_STATS

With this primitive the service user can request some of the statistics of the underlying communication system, like the round trip time (RTT). Such information might be required to set private timers or to learn about the state of the CS.

Currently only a single return value is specified, namely the measured RTT of a SDU.

LHP_XON

This service primitive can be used by both the service user and the service provider. It signals that originator of this signal is ready to accept further data.

LHP_XOFF

This service primitive is used to stop the data exchange via the local interface until new resources become available.

LHP_FAILURE_IND

When the communication subsystem detects an error situation it must inform its service users. Triggering errors are for example that a message could not be delivered to the peer (after a certain number of attempts), that the keep-alive function signals communication problems or when the peer resets a connection.

Copyright at Technical University Berlin. All Rights reserved.

The indication should inform the service user about the error reason and whether it affects all entities or only a specific entity, e.g.: when the link between RSC and RSS is intercepted all entities are affected but the failure of delivery of a single message affects only the origin of this message. The latter is especially important on the RSS when the communication to a single RSC is not possible but all other RSC are still reachable.

2.4 CS API Definition

This section describes one specific Linux kernel implementation of the interface in C-language notation. This API is used by the current ReSoA implementation.

The main idea behind choosing a two layered approach for ReSoA was to separate technology dependent operation from technology independent one. With such a design it is possible to use the same implementation of the LSM and RSM on top of different technologies as long as the API is protocol and address format independent.

In user space, the socket interface meets these requirements. Although it would be possible to have a socket based implementation of the ReSoA/CS API implementation, such an interface would come along with too much overhead. The socket interface was designed to be used from the user space and not from inside the kernel. Therefore we decided to design a new socket-like data structure which is tailored for our needs. This new object is named *LHP_s*. Appendix A gives all the details about the new object.

Table 2.1 shows the association behind the abstract interface given section 2.3 and the API defined in this chapter. The abstract interface is implemented using function calls. Thus, the implementation of the interface is synchronous. When a call to a function of the API returns, however, this does not mean that the call was completed but only that the service provider has either accepted the request or rejected it. The usage of function calls allow for realizing some of the service primitives by return values of function calls. For example in case the service user calls a function to transmit data (`LHP_DATA_REQ`) the return value of the function indicates whether the service provider could accept the send request or not.

The service provider is responsible to implement all the functions which are not marked as callback, while the service user must implement the callbacks. The service user must call the callbacks when appropriate as described in section 2.4.3. However, please note that the table shows only function pointers. This means that the CS designer is free to chose names for the different functions.

2.4.1 Implementation of the interface

The interface should be implemented for Linux kernel version 2.4.x as loadable kernel module. It is only intended to use this interface from inside the kernel. The service provider is responsible to implement the functions referenced in the `struct LHP_provider_s` and to call the appropriate callbacks as explained in section 2.4.3.

Operation inside the kernel requires very precise declaration about the execution context of every function call. In particular, distinction between process context and interrupt time has to be done. Generally, a task that is executable in interrupt handlers can be safely performed with process context (e.g.: in a schedule_task function or kernel thread). Knowing

Copyright at Technical University Berlin. All Rights reserved.

Abstract Interface	API	Callback
LHP_REGISTER	int (*init) (struct LHP_s*)	-
LHP_REGISTER_OK	return value of (*init)() function is zero	-
LHP_REGISTER_FAILED	return value of (*init)() function is negativ	-
LHP_UNREGISTER	void (*cleanup) (struct LHP_s*)	-
LHP_UNREGISTER_OK	return value of `cleanup` function is zero	-
LHP_UNREGISTER_FAILED	return value of `cleanup` function is negativ	-
LHP_CONNECT_REQ	int (*connect) (struct LHP_s*,struct sockaddr*)	-
LHP_PASSIVE	int (*passive) (struct LHP_s *,struct sockaddr*)	-
LHP_CONNECT_IND	int(*connect_indication)(struct LHP_s*,struct LHP_s*)	X
LHP_CONNECT_RESP	return value of `connect_indication` callback is zero	-
LHP_CONNECT_REJ	return value of `connect_indication` is negative	-
LHP_CONNECT_CONF	int (*connect_confirm)(struct LHP_s*)	X
LHP_CLOSE_REQ	void (*close)(struct LHP_s*,int)	-
LHP_DISCONNECT_IND	void (*disconnect_indication)(struct LHP_s*,int)	X
LHP_DATA_REQ	int (*send)(struct LHP_s*, struct sk_buf*)	-
LHP_DATA_OK	return value of `LHP_DATA_REQ` is zero	-
LHP_XOFF	return value of `LHP_DATA_REQ` is `EAGAIN`	-
LHP_DATA_IND	int (*receive)(struct LHP_s,struct sb_buf**)	-
LHP_GET_STATS	int (*getoption)(struct LHP_s*,int, void*,int); with option = LHP_OPT_RTT	-
LHP_RTT	return value of `getoption` function	-
LHP_XON	void (*send_ready)(struct LHP_s*) and void (*receive_ready)(struct LHP_s*)	X
LHP_XOFF	Return message of send call	-
LHP_FAILURE	void (*disconnect_indication)(struct LHP_s*,int)	X

Table 2.1: Association between abstract interface and API

Copyright at Technical University Berlin. All Rights reserved.

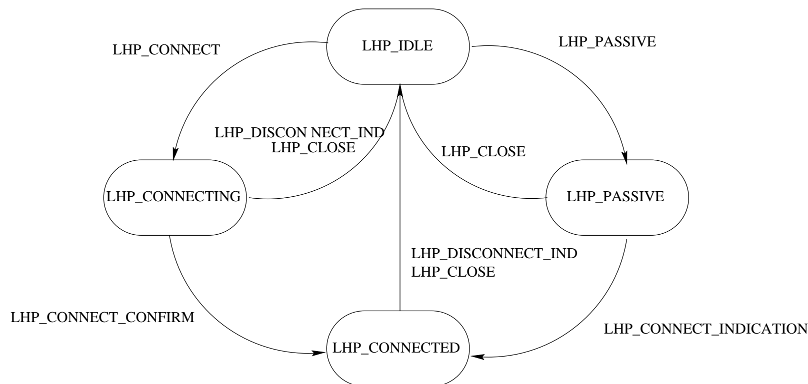

Figure 2.6: State machine of the LHP interface

that, it is only necessary to mention those functions that must have a process context for their execution.

All functions and callbacks are expected to be executable during interrupt time. Provider implementations may use `in_interrupt()` kernel function to schedule the work, that cannot be executed within the interrupt handler.

A communication end-point is implemented by the data structure `struct LHP_s`. This data structure includes references to two data structures, namely `LHP_provider_s` and `LHP_user_s` which glue together the service user and service provider as explained in the chapter ReSoA run time environment (see figure 2.7). The `LHP_user_s` object holds pointers to the required callbacks and some management information. This datatype is maintained by the service user. The `LHP_provider_s` object holds references of the primitives and some management information. This object is maintained by the service provider.

Interface state machine

The service provider is responsible implement the state machine shown in figure 2.4.1. The state `LHP_UNCONNECTED` is the initial state.

2.4.2 ReSoA runtime environment

One of the features of ReSoA is that the CS should be chose-able form a set of CSes. To allow for this ReSoA provides functionality to dynamically register new communication subsystems. Further it provides wrapper function that hide all the function pointers. This wrapper functions for example test whether a function pointer is not null before it is called.

In order to register a new communication subsystem the service provider must implement a protocol according to the service specification. Further the protocol implementation must support the function as shown in table 2.1 and must call the described callbacks. Next a `struct LHP_provider_s` data structure must be created and initialized with references to the functions. Beside the function pointers, this data structure holds a unique identifier for the provider (e.g.: LHP_PROVIDER_SIEMENS or LHP_PROVIDER_TKN), a type (e.g.: RELIABLE) and the amount of private data space required by the CS.

Copyright at Technical University Berlin. All Rights reserved.

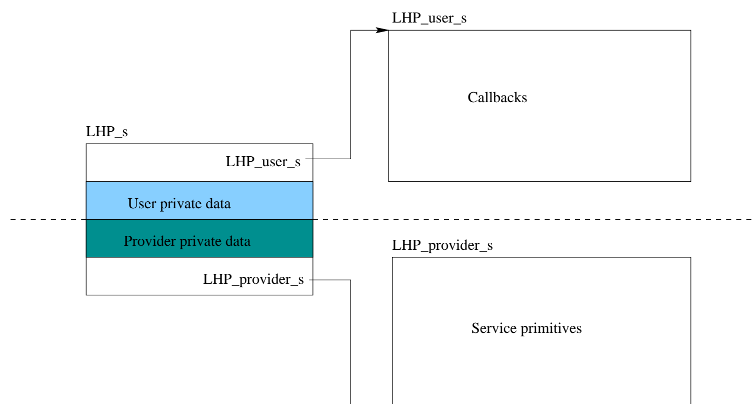

Figure 2.7: LHP-interface objects

After the service provider has set-up this data structure he must register his CS with ReSoA. For this purpose the runtime environment provides the following two functions.

- `int LHP_register_provider(struct LHP_provider_s *lhp)` function is used to register a new CS. The lhp parameter must point to a initialized `struct LHP_provider_` data structure. The (provider,type) tuple of the referenced data structure must be unique. The service provider must call this function for every LHP he wants to make available to ReSoA.
- `int LHP_unregister_provider(struct LHP_provider_s *)` function must be used to remove a CS from the ReSoA runtime environment.

2.4.3 Functions implemented by the service provider

Registration

The `init`-function is called to create a new communication endpoint. It must follow the following prototype:

`int (*init) (struct LHP_s* lhp)`.

The `lhp` parameter is initialized to point to a valid `LHP_if` object which in turn references a `LHP_provider_s`-object.

The function must return `0` on success and a negative number otherwise. The number of a negative return value should be inline with the system defined errno value (see Appendix B).

The following error codes are defined:

- **ENOMEM**: Insufficient memory is available.
- **EINVAL**: Invalid argument.

However, the service user does not call this function directly but uses the `int LHP_create(int provider, int type, struct LHP_user_s *user, int allocation, struct LHP_s **lhp)`

Copyright at Technical University Berlin. All Rights reserved.

wrapper function. This function locates the appropriate LHP_provider_s data type using the provider and type parameter. Further it creates the LHP_s object and initializes it. The wrapper function alloctes the memory required for the service user private area as well as the memory needed for the service provider private are. After initialization the `void *provider_private` member points to valid memory area of the size as requested by the service provider when the LH_provider_s object was set-up. Finally, the wrapper function calls the init function of the selected LHP_provider_s object. The idea behind allocating the memory for the service user and service provider private data in a single chunk is to save resources.

The `cleanup`-function is called to release a communication endpoint and must follow the following prototype:

```
void (*cleanup)(struct LHP_s *lhp)
```

Again, it is not directly called but using the wrapper function `void LHP_relase(struct LHP_s*)`.

Connection Management

- `int (*connect)(struct LHP_s *lhp, struct sockaddr *remote_addr)`

  This function corresponds to the `LHP_CONNECT_REQ`-primitive of the generic interface. It must implement an active open. The service user specifies the address of the peer end-system as well as the connection endpoint which is asked to establish a new connection. The function call should return immediately (non-blocking behavior) with either a return value 0 to indicate that the CS establishes the connection or with a negative result indicating an error (according to the system wide errno values).

  The following return values are defined:

  - **EAFNOTSUPPORT** : The used address format was not valid.
  - **EINPROGRESS**: The call is non-blocking and the connection cannot be completed immediately.
  - **EALREADY:** The **L**HP object is non-blocking and a previous connection attempt has not yet been completed.
  - **EINVAL**: Invalid argument.

  The service user must be informed about the success of the connection establishment by calling the *int (*connect_confirm)(struct LHP_s *lhp)* callback or about a failure by calling the *int (*disconnect_indication)(struct LHP_s *lhp, int reason)* callback.

  The CS must not accept any data requests before the CS connection is established.

- `int (*passive)(struct LHP_s *lhp, struct sockaddr *local_addr)`

  This function corresponds to the `LHP_CONNECT_P`-primitive. The service user must specify on which local address it wants to receive new connections as well as the communication end-point which should be set to the passive open mode.

  Since the service user has no idea about the address format which is used by the CS it reads the address from some configuration file without interpreting it.

Copyright at Technical University Berlin. All Rights reserved.

This function is non-blocking. Its purpose is to just configure the attached protocol to accept new connections. This function returns zero on success and a negative value otherwise. The error code is defined by the system defined errno file.

The following error codes are defined:

- **EINVAL**: Invalid argument.

- `void (*close) (struct LHP_s *, int reason)`

  This function corresponds to the `LHP_DISCONNECT_REQ`-primitive of the generic interface. The service user must specify the communication endpoint for which it requests to close the connection.

  This function has no return value, since success is always assumed.

Data exchange

- `int (*send)(struct LHP_s* lhp, struct sk_buff *skb)`

  This function corresponds to the data request primitive of the abstract interface. The service user must specify the communication endpoint and a sk_buf which holds the data to be transmitted.

  The function is non-blocking. The return value is positive (and equals the length of the transmitted data chunk) when the CS was able to accept the message for delivery otherwise a negative value is returned. The return values should follow the system defined errno file. In case of an temporarily lack of resources `EAGAIN` should be returned. This corresponds to the interface flow control described by the abstract interface.

  In case the attached protocol is not connected the CS should return an error.

  The following error codes are defined:

  - **EMSGSIZE**: The size of the EP-SDU is not supported.
  - **EAGAIN**: The outgoing queue is full. The service user is informed when resources become available.
  - **EINVAL**: Invalid argument passed.
  - **ENOTCONN**: Data request called for a communication endpoint which is not connected.

- `int (*receive)(struct LHP_s *lhp, struct sk_buff **skb)`

  This function is used to consume data by the service user. It has no direct correspondent function in the abstract interface. It is called by the service user after he was informed by the `LHP_receive_ready` callback that new data is pending. The function is non-blocking. In case no data are waiting it should return **ENODATA**.

  The service user must provide an address at which the CS can store a reference to the sk_buff that holds the data.

Copyright at Technical University Berlin. All Rights reserved.

On success the receive function returns a positive number indicating how many byte were received otherwise a negative number is returned which must be inline with the following error codes.

- **EINVAL**: Invalid argument
- **ENOTCONN**: Communication endpoint was not connected yet.
- **ENODATA**: The received function was called but no data were available.

Miscellaneous

- `int (*setoption)(struct LHP_s *lhp, int name, void *optval, int optlen))`

  This function is used to configure the communication endpoint.

  This function is not used at the moment.

- `int (*getoption)(struct LHP_s *lhp, int name, void *optval, int *optlen)`

  This function is used to read information about the configuration of the communication end-point or to receive statistics. The following information are available

  - **RTT** : To get the RTT of a CS connection.
  - **VENDOR_STRING** : To get information about the provider.
  - **...**.

Callbacks

Callbacks are used to inform the service user about the occurrence of asynchronous events like the the reception of new data. The callbacks are implemented by the service user.

The CS never directly invokes any callback but uses a wrapper function. This wrapper functions are implemented by the ReSoA environment. In following we discuss which callback is triggered by which event and which wrapper function should be called.

- `int LHP_connect_indication(struct LHP_s *lhp, LHP_s *new)` This callback must be called by CS when it has received a connect request and has been configured to accept incoming connection requests using the `passive` function call. The parameter references a new communication endpoint. The service user can either accept this connection by returning 0 or refusing the connection by returning a negative value.

  The service user must set this callback when it performs a passive open but can leave it uninitialized when it is the active end.

- `void LHP_connect_confirm(struct LHP_s *lhp)`

  With this callback the CS informs the active end of a connection that its connection request invoked with the `connect` function has succeeded.

Copyright at Technical University Berlin. All Rights reserved.

- `void LHP_disconnect_indication(struct LHP_s *lhp, int reason)`

  This callback informs the service user that a connection request has failed or when the peer entity has closed a connection. The reason parameter should show the reason. The following reasons are defined: ...

- `void LHP_receive_ready(struct LHP_s * lhp)`

  The callback is used to indicate that the CS has received some data which can be read by the service user using the `receive`-primitive.

- `void LHP_write_ready(struct LHP_s *lhp)`

  This callback should be called by the CS when space becomes available in its send queue.

2.4.4 Usage of the API

The service user (LSM or RSM) use this API to access the service of the CS without knowing who (which provider) implements the CS or what protocol or addressing format is used.

To allow for such an independent implementation the service user must achieve the required parameters from some management interface. The service provider is responsible to provide these information. For example after the service provider has registered its CS using the `register_provider` function it can set-up a database (file) with the provider identifier and the service type. Further the address of the server is required.

The service user does not directly call the functions implemented by the service provider but uses some wrapper functions provided by the ReSoA runtime environment.

When a new service user is created it reads this configuration and creates the required communication endpoints. Since the operation differ on the Remote Socket Client and the Remote Socket Server we consider both separately.

The Remote Socket Server creates a single communication endpoint upon start-up using the `create` function. After the communication endpoint was successfully created it initializes the required callback functions and uses the `passive` function to inform the CS that it is ready to accept new connections. Then it remains idle until it is triggered by the `connect_ind` callback.

When the RSS has received the connect_ind callback it either accepts the new connection or refuses it. In the former case it returns zero and in the latter case it returns a negative value. The connect_ind callback comes along with a reference to a new created communication end-point. If the RSS accepts the new connection it uses this communication end-point to exchange message with the new client. The initial communications endpoint stays unconnected and is ready to accept further connections.

After the connection is accepted the RSS can send and receive data on this communication end-point. There will be a single communication end-point for every client. This communication end-point is used for all communication between the RSS and RSC. When the client closes a session the RSS is informed by the `disconnect_ind callback`.

The Remote Socket Client also creates a communication endpoint upon start-up. After the endpoint was successfully created it uses the `connect` function to establish a connection

Copyright at Technical University Berlin. All Rights reserved.

to the RSS. The address of the RSS is taken form some configuration file. After initiating the connection the RSC is idle until it is triggered either by the `connect_conf` or the `disconnect_ind` callback. In the former case the RSC is allowed to send and receive data from this communication endpoint. At the end of the session the client releases the communication end-point with the `release` function.

Copyright at Technical University Berlin. All Rights reserved.

Appendix A

CS API Headerfile

```
/*
 * ReSoA
 * Last Hop Protocol Interface
 *
 * Copyright (c) 2001 Telecommunication Network Group, Technical University of Berlin
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 * 3. All advertising materials mentioning features or use of this software
 *    must display the following acknowledgment:
 *      This product includes software developed by the Telecommunication Network
 *      Group of Technical University Berlin.
 * 4. Neither the name of the University nor of the Laboratory may be used
 *    to endorse or promote products derived from this software without
 *    specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS ``AS IS'' AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
```

Copyright at Technical University Berlin. All Rights reserved.

```
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 *
 *
 * Tobias Poschwatta <posch@ft.ee.tu-berlin.de>
 * Morten Schläger   <morten@ee.tu-berlin.de>
 *
 * Technische Universität Berlin
 * Telecommunication Network Group
 * Berlin, 2001
 *
 * Version:
 *
 */

#ifndef _RESOA_LHP_H
#define _RESOA_LHP_H

#include <linux/errno.h>
#include <linux/skbuff.h>

enum {
  LHP_IDLE = 0,
  LHP_CONNECTING = 1,
  LHP_CONNECTED = 2,
  LHP_PASSIVE = 3
};

struct LHP_s;

struct LHP_user_s {

  int private_size;

  /* connection management */
  void (*connect_confirm)(struct LHP_s *lhp);
  int (*connect_indication)(struct LHP_s *lhp, struct LHP_s *new);
  void (*disconnect_indication)(struct LHP_s *lhp, int reason);

  /* data transfer */
  void (*send_ready)(struct LHP_s *lhp);
  void (*receive_ready)(struct LHP_s *lhp);

};
```

Copyright at Technical University Berlin. All Rights reserved.

```
struct LHP_provider_s {

  int provider;
  int type;

  int private_size;

  int headroom;

  /* service instances */
  int (*init)(struct LHP_s *lhp);
  void (*cleanup)(struct LHP_s *lhp);

  /* connection management */
  int (*connect)(struct LHP_s *lhp, struct sockaddr *remote_addr);
  int (*passive)(struct LHP_s *lhp, struct sockaddr *local_addr);
  void (*close)(struct LHP_s *lhp, int reason);

  /* data transfer */
  int (*send)(struct LHP_s *lhp, struct sk_buff *skb);
  int (*receive)(struct LHP_s *lhp, struct sk_buff **skb);

  /* options */
  int (*setoption)(struct LHP_s *lhp, int name, void *optval, int optlen);
  int (*getoption)(struct LHP_s *lhp, int name, void *optval, int *optlen);

  /* used by LHP registry */
  struct LHP_provider_s *next;
  int refcount;

};

struct LHP_s {

  int state;

  struct LHP_provider_s *provider;
  void *provider_private;

  struct LHP_user_s *user;
  void *user_private;

  /* used by passive instances */
  struct LHP_user_s *child_user;
  int backlog;
```

Copyright at Technical University Berlin. All Rights reserved.

```
};



/*
 * provider registry
 *
 */

enum {
  LHP_PROVIDER_TKN = 1,
  LHP_PROVIDER_SIEMENS = 2,
  LHP_N_PROVIDERS
};


extern int LHP_register_provider(struct LHP_provider_s *provider);
extern void LHP_unregister_provider(struct LHP_provider_s *provider);
extern struct LHP_provider_s *LHP_lookup_provider(int provider, int type);



/*
 * service instances
 *
 */

extern int LHP_create(int provider, int type, struct LHP_user_s *user, int allocation,
                      struct LHP_s **lhp);
extern void LHP_release(struct LHP_s *lhp);



/*
 * connection management
 *
 */

extern int LHP_connect(struct LHP_s *lhp, struct sockaddr *remote_addr);
extern void LHP_connect_confirm(struct LHP_s *lhp);
extern int LHP_passive(struct LHP_s *lhp, struct sockaddr *local_addr, int backlog,
                       struct LHP_user_s *child_user);
extern int LHP_connect_indication(struct LHP_s *lhp, struct LHP_s *new);
extern void LHP_disconnect_indication(struct LHP_s *lhp, int reason);
```

Copyright at Technical University Berlin. All Rights reserved.

```
extern void LHP_close(struct LHP_s *lhp, int reason);



/*
 * data transfer
 *
 */

static inline void LHP_send_ready(struct LHP_s *lhp)
{
  if (lhp->user && lhp->user->send_ready)
    lhp->user->send_ready(lhp);
}

static inline void LHP_receive_ready(struct LHP_s *lhp)
{
  if (lhp->user && lhp->user->receive_ready)
    lhp->user->receive_ready(lhp);
}

extern int LHP_send(struct LHP_s *lhp, struct sk_buff *skb);
extern int LHP_receive(struct LHP_s *lhp, struct sk_buff **skb);



/*
 * options
 *
 */

enum {
  LHP_OPT_VENDOR_STRING = 1,
  LHP_OPT_RTT = 2
};

static inline int LHP_setoption(struct LHP_s *lhp, int name, void *optval, int optlen)
{
  if (lhp == NULL)
    return -EINVAL;
  return lhp->provider->setoption(lhp, name, optval, optlen);
}

static inline int LHP_getoption(struct LHP_s *lhp, int name, void *optval, int *optlen)
{
```

Copyright at Technical University Berlin. All Rights reserved.

```
  if (lhp == NULL)
    return -EINVAL;
  return lhp->provider->getoption(lhp, name, optval, optlen);
}



/*
 * skb allocation
 *
 */

static inline struct sk_buff *LHP_alloc_skb(struct LHP_s *lhp,
                                     unsigned int size, int allocation)
{
  struct sk_buff *skb;

  if (lhp == NULL)
    return NULL;

  skb = alloc_skb(size + lhp->provider->headroom, allocation);

  if (skb != NULL)
    skb_reserve(skb, lhp->provider->headroom);

  return skb;
}

#endif
```

Copyright at Technical University Berlin. All Rights reserved.

Appendix B

Error Codes (excerpt from include/asm-i386/errno.h)

```
#ifndef _I386_ERRNO_H
#define _I386_ERRNO_H

#define EPERM            1      /* Operation not permitted */
#define ESRCH            3      /* No such process */
#define EINTR            4      /* Interrupted system call */
#define EIO              5      /* I/O error */
#define ENXIO            6      /* No such device or address */
#define E2BIG            7      /* Arg list too long */
#define ENOEXEC          8      /* Exec format error */
#define EBADF            9      /* Bad file number */
#define EAGAIN          11      /* Try again */
#define ENOMEM          12      /* Out of memory */
#define EACCES          13      /* Permission denied */
#define EFAULT          14      /* Bad address */
#define EBUSY           16      /* Device or resource busy */
#define ENODEV          19      /* No such device */
#define EINVAL          22      /* Invalid argument */
#define ENFILE          23      /* File table overflow */
#define EMFILE          24      /* Too many open files */
#define EFBIG           27      /* File too large */
#define ENOSPC          28      /* No space left on device */
#define EDEADLK         35      /* Resource deadlock would occur */
#define ENOLCK          37      /* No record locks available */
#define ENOSYS          38      /* Function not implemented */
#define EWOULDBLOCK     EAGAIN  /* Operation would block */
#define ENOMSG          42      /* No message of desired type */
#define EIDRM           43      /* Identifier removed */
#define ECHRNG          44      /* Channel number out of range */
```

Copyright at Technical University Berlin. All Rights reserved.

```
#define ELNRNG          48      /* Link number out of range */
#define EUNATCH         49      /* Protocol driver not attached */
#define EBADE           52      /* Invalid exchange */
#define EBADR           53      /* Invalid request descriptor */
#define EBADRQC         56      /* Invalid request code */

#define ENODATA         61      /* No data available */
#define ETIME           62      /* Timer expired */
#define ENONET          64      /* Machine is not on the network */
#define EREMOTE         66      /* Object is remote */
#define ENOLINK         67      /* Link has been severed */
#define EADV            68      /* Advertise error */
#define ECOMM           70      /* Communication error on send */
#define EPROTO          71      /* Protocol error */
#define EBADMSG         74      /* Not a data message */
#define EOVERFLOW       75      /* Value too large for defined data type */
#define ENOTUNIQ        76      /* Name not unique on network */
#define EREMCHG         78      /* Remote address changed */
#define EUSERS          87      /* Too many users */
#define ENOTSOCK        88      /* Socket operation on non-socket */
#define EDESTADDRREQ    89      /* Destination address required */
#define EMSGSIZE        90      /* Message too long */
#define EPROTOTYPE      91      /* Protocol wrong type for socket */
#define ENOPROTOOPT     92      /* Protocol not available */
#define EPROTONOSUPPORT 93      /* Protocol not supported */
#define ESOCKTNOSUPPORT 94      /* Socket type not supported */
#define EOPNOTSUPP      95      /* Operation not supported on transport endpoint */
#define EPFNOSUPPORT    96      /* Protocol family not supported */
#define EAFNOSUPPORT    97      /* Address family not supported by protocol */
#define EADDRINUSE      98      /* Address already in use */
#define EADDRNOTAVAIL   99      /* Cannot assign requested address */
#define ENETDOWN        100     /* Network is down */
#define ENETUNREACH     101     /* Network is unreachable */
#define ENETRESET       102     /* Network dropped connection because of reset */
#define ECONNABORTED    103     /* Software caused connection abort */
#define ECONNRESET      104     /* Connection reset by peer */
#define ENOBUFS         105     /* No buffer space available */
#define EISCONN         106     /* Transport endpoint is already connected */
#define ENOTCONN        107     /* Transport endpoint is not connected */
#define ESHUTDOWN       108     /* Cannot send after transport endpoint shutdown */
#define ETIMEDOUT       110     /* Connection timed out */
#define ECONNREFUSED    111     /* Connection refused */
#define EHOSTDOWN       112     /* Host is down */
#define EHOSTUNREACH    113     /* No route to host */
#define EALREADY        114     /* Operation already in progress */
```

Copyright at Technical University Berlin. All Rights reserved.

```
#define EINPROGRESS     115     /* Operation now in progress */
#define EREMOTEIO       121     /* Remote I/O error */

#define ENOMEDIUM       123     /* No medium found */
#define EMEDIUMTYPE     124     /* Wrong medium type */

#endif
```

Copyright at Technical University Berlin. All Rights reserved.