HOCHSCHULE RUHR WEST
UNIVERSITY OF APPLIED SCIENCES

Development and Implementation of a Streaming Protocol for Reliable and Efficient Data Transfer via a Lossy and Narrow Radio Channel

Bachelorarbeit
im Studiengang Angewandte Informatik
der Hochschule Ruhr West

Jan Dehlen
10006486

Erstprüferin:	Prof. Dr.-Ing. Gerd Bumiller
Zweitprüfer:	Philipp Horwat

Bochum, Oktober 2021

Abstract

In this document a reliable data streaming mechanism for a TDMA LPWAN application is developed by adapting a link layer solution for power line communication, published at the International Symposium on Power Line Communications and its Applications (ISPLC) 2015. A C++ implementation of the services link layer is provided and demonstrated working at a packet error rate of 50%.

Table of Contents

List of Figures

Table Directory

List of Abbreviations

HRW	Hochschule Ruhr West
LPWAN	Low Power Wide Area Network
ISPLC	International Symposium on Power Line Communications and its Applications
TDMA	Time Division Multiple Access
CSS	Chirp Spread Spectrum
SF	Spreading Factor
CR	Code Rate
OS	Operating System
MAC	Media Access Control
P2SP	Packet to Stream Protocol
PLC	Power Line Communication
PER	Packer Error Rate
FIFO	First In First Out

1 Introduction

1.1 Motivation

LPWAN is a wireless telecommunication wide area network. Devices communicate at long range with low bandwidth and a minute power demand. LPWAN is a cost effective, easy to deploy technology and easily added to preexisting infrastructures. The long-range connectivity allows power grid facilities 15 km apart from each other to exchange data wirelessly [1]. It gained in popularity especially for transferring short messages from e.g., sensors in IoT applications. Its limited bandwidth of typically several kbps and potentially high packet error rate (PER) make it rarely used for volume data transmissions. While those limitations cannot be overcome, the data streaming service described in this document aims to create a mechanism which makes sustained, reliable data streaming via LPWAN a viable use case.

1.2 Context

The streaming service described in this document is part of a larger project [2]. This surrounding project aims to develop a Power Quality and Grid Condition monitoring Application based on the proprietary LoRa PHY-layer [1] combined with a TDMA communication scheme. This LPWAN approach is explored as an additional option to the more commonly used PLC for grid communication [3]. In addition to meet strict time synchronization requirements, the project aims to utilize all scarce available channel resources. High priority services, carrying the time critical messages do not take up all channel capacity and are partly only sent if needed. To utilize the remaining capacity to transfer non-time critical data is the task of the streaming service described in this document.

The project builds a communication system of LoRa transceivers in fixed locations on the power grid. Nodes are collecting monitoring data at their deployment site. These nodes then send data via LoRa radio over a distance up to 15km to another device [4]. These receiving devices are called gateways and pass on the data through IP-based communication systems for further processing.

The LoRa modulation technique is based on based on Chirp Spread Spectrum modulation (CSS) which trades data rate for sensitivity within a fixed channel bandwidth [1]. One node is associated with a single gateway, while a gateway communicates to multiple nodes, so that a star architecture is constituted.

The data rate is typically limited to several kbps [2]. In case of transmission disturbances, it can be lowered further by altering the Spreading Factor (SF) and Coding Rate (CR) of the LoRa signal to gain sensitivity. In addition, LPWAN is typically restrictively

regulated. To efficiently utilize this severely restricted channel the project employs a cross layer design of MAC and Link-Layer [2]. A gateway as a master-device calculates the usage of available timeslots across nodes and types of data. This distribution of timeslots is sent to all nodes via a beacon broadcast. Time is divided into frames. Each frame starts with a Beacon distributing the necessary organizational data to all nodes. 2 – 32 Frames constitute a super-frame. The assigned timeslots include slots for static responses. Static responses are packets sent from a node to its gateway (uplink) and can be used for non-payload data like reception confirmation. Fixed timeslots for gateway to node communication (downlink) are called broadcasts. Non-fixed timeslots are referred to as dynamic slots.

The used LoRa devices are embedded-system units including a LoRa Chip by SEMTECH. Either the SX1268 or SX1262 chip is used, depending on the region in which the unit is supposed to operate. They are running a Linux Operating System (OS) [2].

The implementation of the Media Access Control (MAC) is split into one part being handled in the Linux kernel or driver space and another part in the user space. Within the driver space the RF-Chip control is handled to provide exact timings. The user space handles e.g., the preparation of the data to be sent and the processing of received data packets. The respective processes are called MAC-pre thread and MAC-post thread. These threads transfer and receive data from driver space via a serial interface. They will also access the streaming service to create a packet of streaming data to be sent within a dynamic slot at a transmitter or decode a received stream packet.

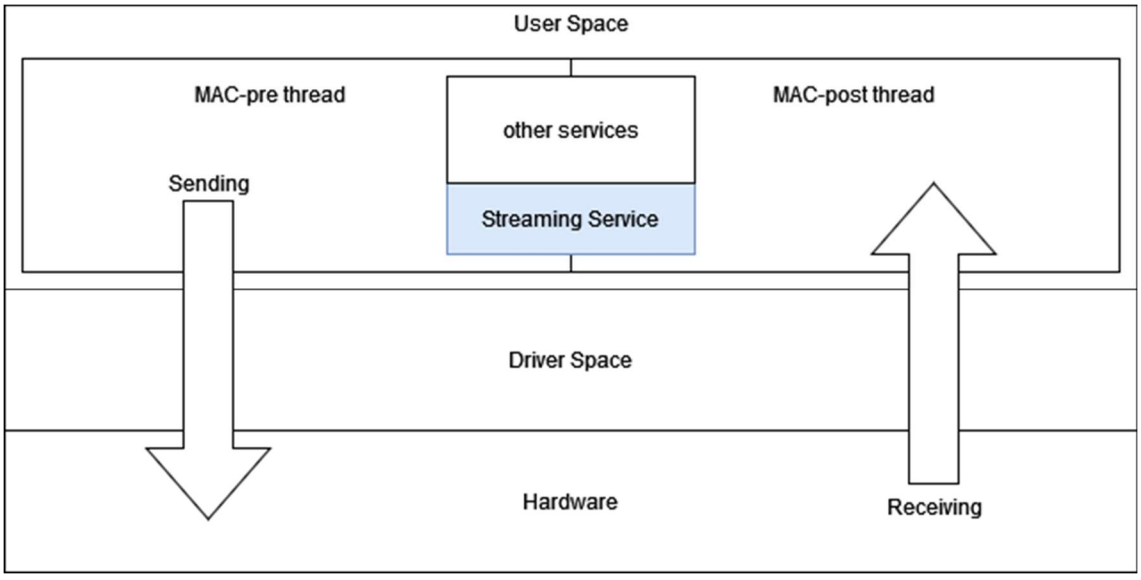

Figure 1: Location of the Streaming Service within the device

1.3 Study Objectives

The goal of the project documented here is firstly to develop a model for the streaming service of the *LPWAN Power Quality and Grid Condition monitoring Application*. It aims to provide reliable data streaming by applying the mechanisms of a MAC / Link Layer

crosslink solution for PLC communication [5] [6] [7] and adapting them to the given communication infrastructure of the monitoring application. This crosslink layer approach is explained, and the necessary alterations are pointed out. Also, a structure to manage multiple connections of a gateway is created.

Secondly a C++ implementation of the link layer according to the model is provided. It aims to be a base for a program executable on the hardware devices of the power monitoring application. The streaming service runs in the user space of the Linux OS of the LoRa devices. It interacts with the surrounding project by providing functions which can be called by the MAC-pre thread and MAC-post thread processes of the monitoring application. It queues and converts payload data of the system devices to a byte stream. From this byte stream packages are constructed that fit a given dynamic timeslot of variable size. The service keeps track of each data packet until reception is confirmed and will be responsible for reliability of the transmission including a retransmission mechanism. The goal is to utilize every available timeslot in each gateway-to-nodes setup. In addition, a mechanism to handle two levels of data priority is provided.

Finally, the implementation is tested for its capability to handle the packaging and reassembly of the data stream while operating with variably sized transmission windows and a PER of 50%.

This work does not include the overall scheduling of timeslots and time synchronization or other calculation in MAC-pre thread, MAC-post thread, or driver space. The description and implementation of the streaming service is a proof-of-concept work.t It aims to be applicable and runnable on the OS of the LPWAN devices but does not include runtime optimization, edge case handling and security mechanisms. The implementation covers the link layer functionality. Adjacent systems are not implemented or represented to a minimum extent if necessary for the implementation to run.

2 Concept of the Streaming Service

The challenge the system must tackle is to provide a reliable data transfer despite a high PER with minimal overhead to accommodate the efficiency goal. The reliability goal requires detection and retransmission of lost packets which in turn requires state awareness of the transmitter about the receiver and vice versa. Additionally, LoRa Radio is a time variant channel. Packet sizes must be adjustable on the fly according to current channel properties.

The strategy to achieve this is utilizing the MAC / Link Layer crosslink solution for PLC communication [5] [6] [7]. PLC communication also faces the challenges of high PER and time variance. The crosslink solution is designed to cater to these properties by performing transmissions of state reports between the communicating devices via header information within the data packets. In case of a packet loss further data is send carrying the updated states, so that lost packets can be identified and sent again. All packets sent are kept until reception is confirmed. An instance of such a link layer is held for each communication link [5].

While most of these mechanics are applicable to the LoRa WAN Streaming Service some major adjustments must be made. The overall communication between the LoRa gateway and nodes relies on the separation of payload data in dynamically used timeslots and management data in *broadcasts* in downlink direction and *static responses* in uplink direction. To reduce the overall overhead of the system the state information will not be transmitted within the header of each payload packet, but within the broadcast or static responses. Furthermore, those transmission types differ in size and structure and need to be processed differently. Consequently, downlink and uplink communication must be individually implemented.

The amount of data that can be transmitted within a single transmission slot is variable. Therefore, logically cohesive data packets which are to be transmitted, are converted into a byte stream. From this stream any number of bytes can be selected for a transmission. To reassemble the stream on the receiver side a Packet to Stream Protocol (P2SP) is required. This protocol adds header codes into the data stream which are used to reconstruct the original packets from the stream.

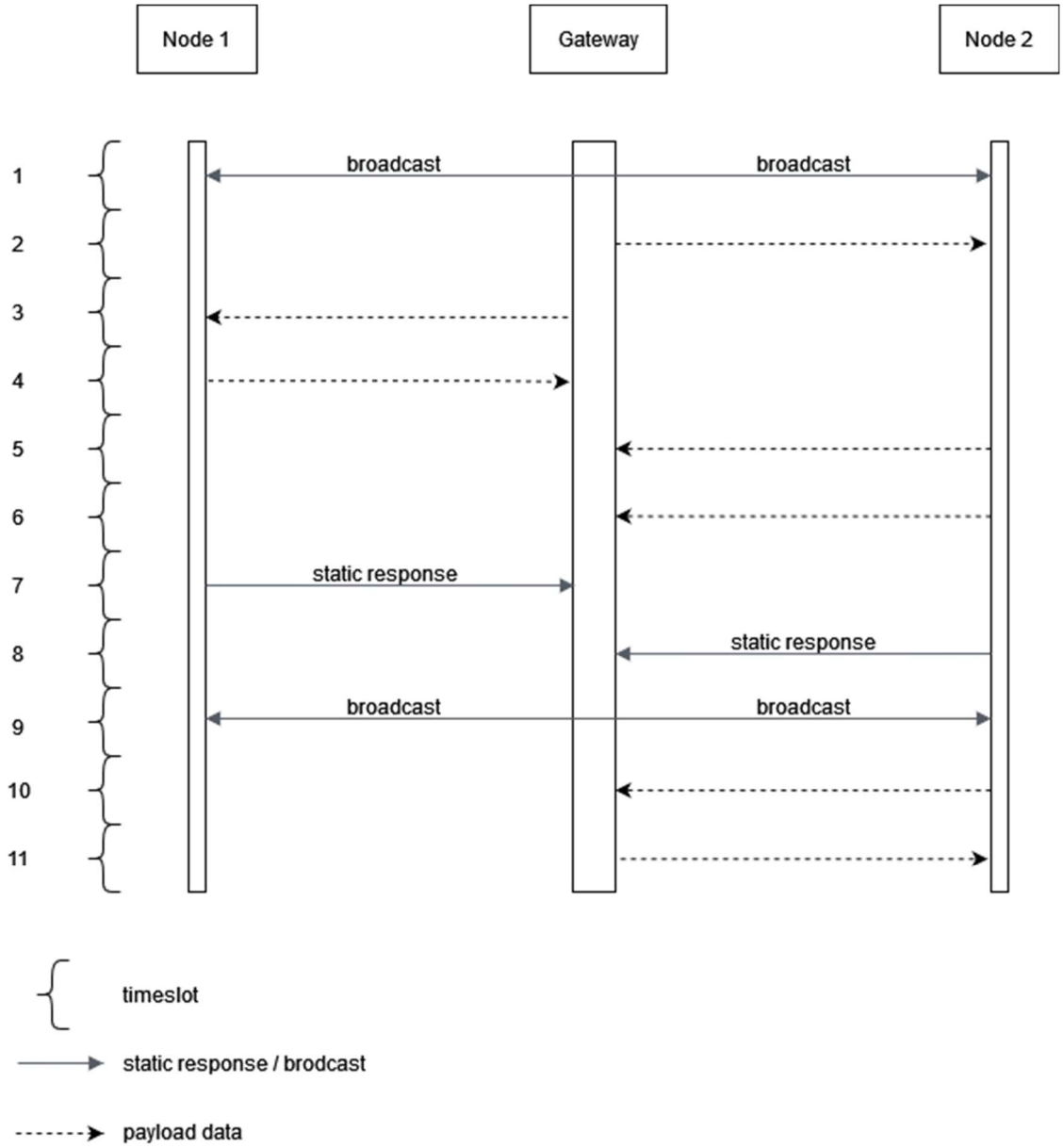

Figure 2: Broadcasts and static responses example, simplified

Aside from the communication between a gateway and a single node, a gateway must manage multiple nodes. It needs to maintain a logical connection to each one and keep track of their states. Moreover, the streaming service will provide information to decide which node will be assigned a timeslot for communication. Once a transmission window is scheduled the Streaming Service is called and operates in the following way:

The link layer prepares a set of bytes to fit the given transmission window on the LoRa radio channel. A set of bytes assembled in this way will be referred to as ***stream packet*** from here on. Each stream packet is tracked by the link layer until successful transmission is confirmed. The link layer is responsible for the reliability of the transmission and the correct reassembly of a data packet from multiple stream packets. If the transmission is not confirmed, a retransmit will be triggered. In that case the stream packet must likely be altered, either be divided or padded to accommodate the number of bytes the new transmission window can carry.

The system typically performs multiple transmits before a confirmation of successful transmission of an individual stream packet can be expected. Between static responses or broadcasts, multiple transmission windows will occur and must be utilized to make efficient use of the channel resources.

On the receiving side, the streaming service assembles the stream packets within its RX Buffer to rebuild the original packet. Once the packet is complete, it is passed to the surrounding system for further processing.

A node will communicate in this way with its associated gateway. A gateway, however, will communicate with multiple nodes. The logical connection between one node and a gateway will be referred to as a ***streamlink*** from here on. Therefore, a node is operating one streamlink, while a gateway is operating one streamlink for each connected node. To establish a fairness of channel usage between the streamlinks of a given gateway, the gateway schedules the usage of available timeslots. To do this, a gateway will operate a scheduler module in addition to the streamlinks. This leads to a model structure as shown below:

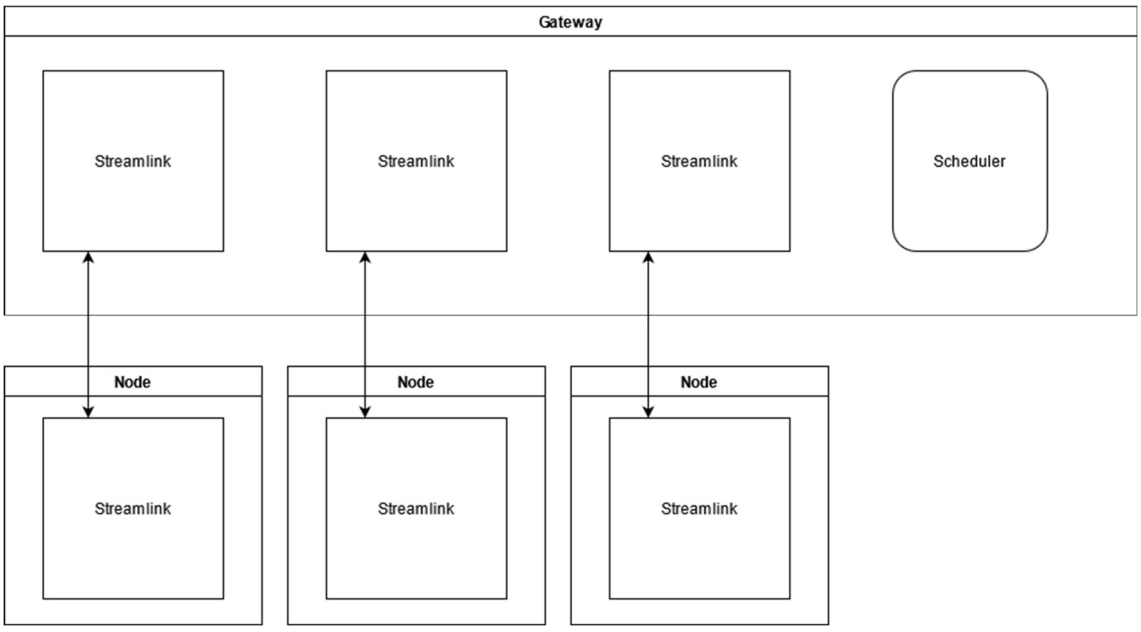

Figure 3: Distribution of Streamlinks

2.1 Packet to Stream Protocol

Before any transmission can be executed the transmitter needs data to be loaded into its transmission buffer. Since the streaming service is transferring a stream of data, any packet is converted into a byte stream by the P2SP. The protocol adds header codes into the byte stream. These codes contain the necessary information to find the beginning, end, and type of a data packet within the transmitted stream on the receiver side. The grid monitoring application [2] defines the following header:

data	Keyword (e.g., 0xAC5)	Code
Bit index	0-11	12-15

Table 1: P2SP header [2]

Code	Function
0000	Reserved for future use
0001	Start of IP-packet
0010	Start of PQMS packet
0011 - 0101	Reserved for future use
0110	Start of Security-Layer packet
0111	Start of Link-Layer packet
1000	Start of MAC-Layer packet
1001 – 1011	Reserved for future use
1100	Packet continued, following 2 bytes are data
1101	End of packet
1110	End of packet, one padding byte following
1111	P2SP header is used as padding, ignore on RX

Table 2: P2SP header codes [2]

On the receiving side the P2SP parses the header keyword and then uses the attached code to reconstruct the packet. For this to work the processed data stream needs to be reliably complete and in correct order. This is ensured by the link layer.

2.2 Streamlinks

A streamlink represents the logical connection between a gateway and a node. Connected devices store the device id of their counterpart within the streamlink. A node will usually operate just one streamlink for data transfer to its associated gateway. A gateway however operates as many streamlinks as nodes are connected. Via these streamlinks a gateway can distinguish between all connected nodes and choose the correct connection for sending data, adding data to the TX buffer, or deciding to which node an incoming transmission belongs. Through the streamlink, a device can access all storage buffers for communication with a certain link. Each link holds its own instance of the link layer to perform data transmissions with the connected device. Upon initialization a streamlink is inactive. In this state it is ready to accept a device id to establish a logical connection. This can be done by adding data to its TX buffer. Along-

side the data the destination device id must be provided, which is then used to setup the streamlink connection. Another way to activate a streamlink is to pass an incoming transmission to the streaming service, which is originating from an up to this point unknown device. In this case the streaming service assigns an inactive streamlink to maintain the connection. The number of available streamlinks is set upon initialization of the streaming service and does not change during operation, due to performance reasons. Therefore, if no inactive link is available, it is not possible to establish a new connection.

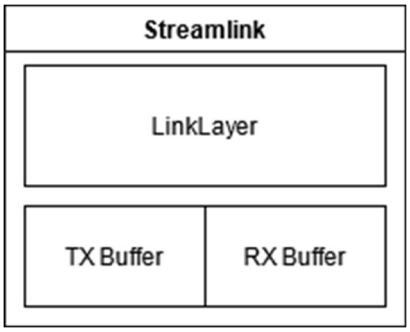

Figure 4: Streamlink with link layer and buffers

2.3 Buffers

The P2SP interacts with the stream links of a connection at the buffers. When converting a packet to a byte stream for transmission, the produced byte stream is written into the TX buffer of the streamlink. The bytes are written in the order in which they are to be sent. Likewise, a received byte stream is taken from an RX buffer to be processed by the P2SP. The reconstruction of the original packet requires the byte stream to be complete and in the exact order in which they were added to the TX buffer. The order in which the bytes were received might not match this, due to packet loss or corruption. In this case the received byte stream is missing bytes at random locations and cannot be processed by the P2SP. Therefore, the RX buffer is required to operate in a way that ensures that only complete and correctly reassembled parts of the byte stream are passed on.

To achieve this, byte stream *sequences* are introduced [7]. A sequence of the byte stream consists of its start address, meaning the index of the first byte within the buffer, and the number of bytes. With this information received bytes can be written to the RX buffer at the exact same location as they were in the TX buffer, even if previous sequences are delayed. This requires TX buffer and RX buffer to be the same in size and structure. Moreover, the RX buffer needs to be aware of incomplete parts of the received byte stream which cannot be passed on yet. In case of the arrival of a delayed sequence it needs to realize how many subsequent sequences are already present and can now be marked as completed.

An additional requirement by the LPWAN project for the streaming service is to distinguish between data of high and regular priority [2]. To ensure that high priority data can

be processed without having to wait on data of regular priority, both kinds of data are stored in dedicated buffers on either side of a streamlink. This enables the service to ensure that urgent data like necessary configuration information for future transmits can be prioritized over regular streaming data. The streamlink structure develops into a state as shown below:

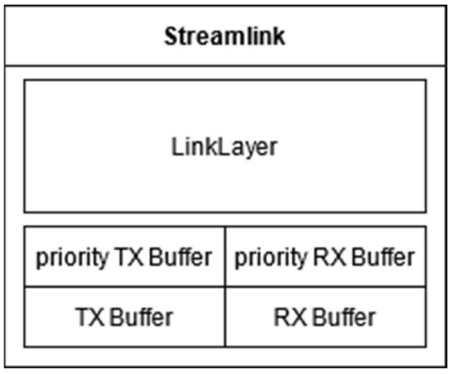

Figure 5: Streamlink with priority buffers

A packet which is to be transmitted, or is received, is stored within either the high or regular priority buffer depending on a parameter when sending or flag within the transmission on reception. Both types of buffers are identical in structure and functionality.

2.4 Link Layer

Byte stream sequences are carried over the radio channel in stream packets. The start address and length of a sequence are stored in the stream packet header. The link layer ensures the reliable transfer of those stream packets. This means that lost or corrupted stream packets are retransmitted. To accommodate the goal of high channel resource utilization, unnecessary retransmits must be avoided and waiting for a confirmation for each transmitted stream packet in a stop and wait manner is not viable. Therefore, the mechanism from PLC communication [5] [7] will be adapted to the given node/gateway architecture. This mechanism is based on the above-mentioned state awareness regarding a transmission on a given device and its connected counterpart. This requires each device to receive information from the connected device and to keep track of its own transmissions. This is done by **virtual links**.

2.4.1 Virtual Links

Within the device, byte sequence data, the state of the transmission and the payload itself is represented by a virtual link. On the transmitting side, the byte sequence is stored in the virtual link until reception is confirmed. On the receiving end, the sequence is kept until it is written successfully into the RX buffer. The structure of an individual streamlink holding several virtual links can be depicted like this:

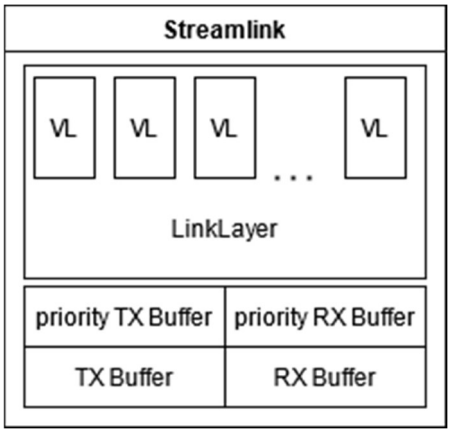

Figure 6: Streamlink with virtual links

A link layer will typically operate multiple virtual links in parallel and utilize information about their states to ensure a reliable transfer of the stream. Thus, virtual links are the base unit of the link layer. While the number of utilized virtual links of a streamlink might vary, the maximum number of virtual links per streamlink is fixed by the protocol for performance reasons. The implementation provided with this document uses eight virtual links per streamlink. If necessary, this number can be increased as the link layer header has unused reserved bits to support expanded addressing of virtual links. The figure below illustrates a gateway holding virtual links across four different streamlinks.

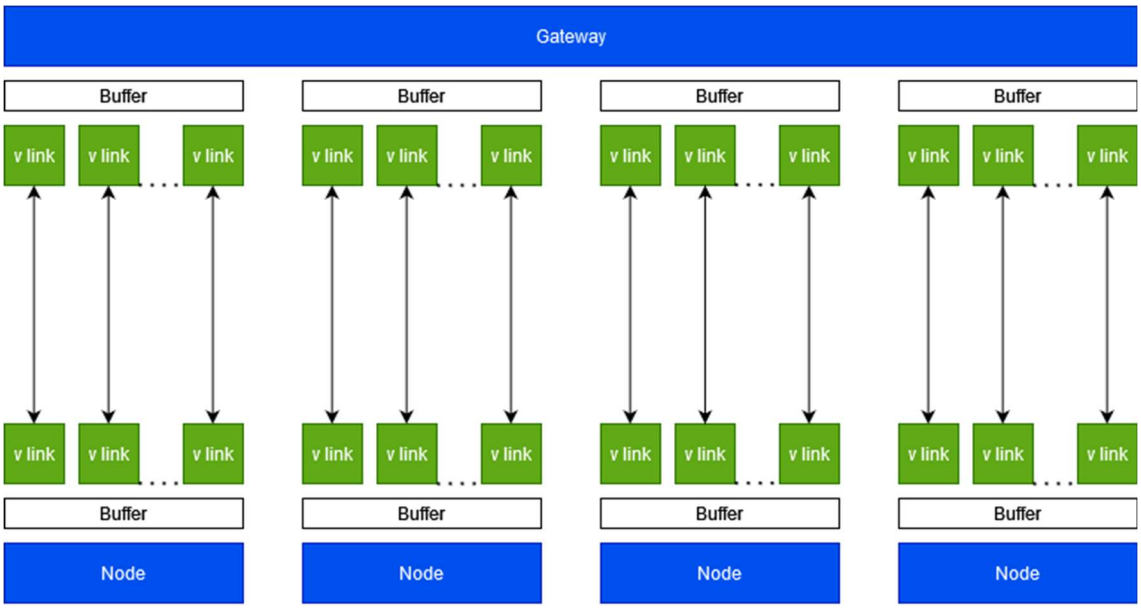

Figure 7: Virtual links across nodes

Virtual links adopt different states which are tracked and partly transmitted to the connected device to achieve the above-mentioned state awareness. This mechanism enables the reliable data transfer. These states are stored and transmitted as bit flags. The states on the transmitter differ from those on the receiver and are therefore described separately.

2.4.2 Sending

Once MAC-pre thread calls the streaming service to construct a stream packet for sending, the service determines the stream link for the desired destination device. Then the link layer selects the requested number of bytes from the TX buffer. If the priority TX buffer holds data, it is used over the regular buffer. The selected bytes are defined as a sequence by determination of sequence number and length. Together with the necessary header this constitutes the stream packet. Since the link layer needs to keep track of all its stream packets in transmission, each stream packet is kept in a virtual link. The states of a virtual link are depicted by the following state machine:

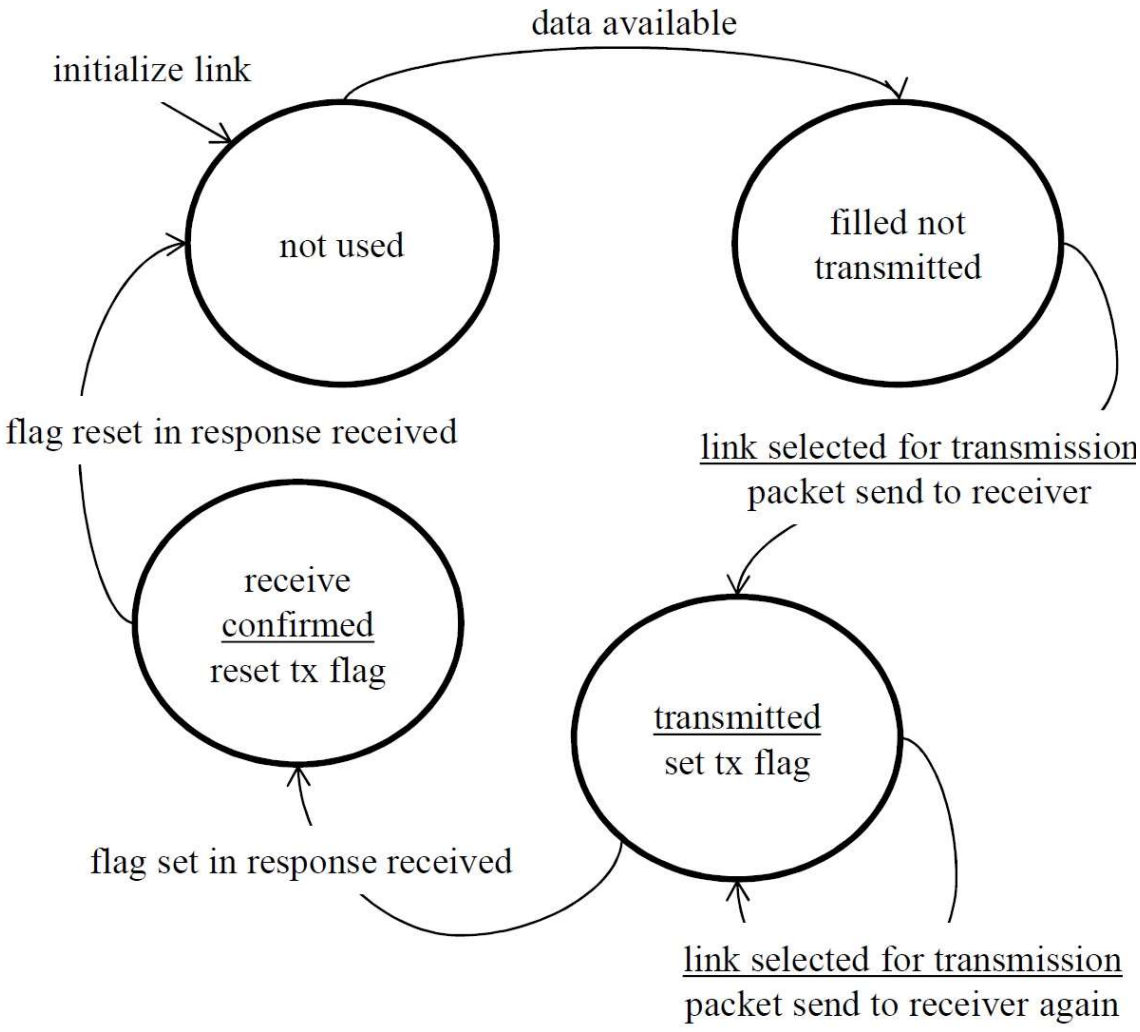

Figure 8: State machine TX virtual link [2]

This state machine from [2] is a slightly altered version from the PLC crosslink layer solution by [5]. But it is performing the same functionality. On initialization the virtual link is not used. When a stream packet is constructed and held by a virtual link it enters the state “filled not transmitted”. Frequently the packet will be provided immediately and the state “transmitted” is adopted directly after. In cases of retransmits, which will be elaborated on down below, the “filled not transmitted” state is kept for a longer duration.

At this stage the transmitting status is persisted as a flag. In case of a received confirmation flag from the receiver the virtual link changes to the “receive confirmed” state. If no confirmation is received with a response, the packet is assumed lost and retransmitted on the next TX call of the service. If reception is confirmed the virtual link is not ready to be used again. This is because the receiver might have received the packet but is not yet ready for receiving another packet. This will be explained in the description of the receiving process. The receiver signals to be ready for reception again, by transmitting the reset confirmation flag, which causes the virtual link to reenter the “not used” state, making it available for another transmission.

In case of a retransmit the available space might not be sufficient to hold the original sequence. In this case the exceeding part of the sequence is split and stored in another virtual link with its own sequence number and length information. While the first part is sent as usual the newly occupied virtual link assumes the state “filled not transmitted”. Already sent links, which need to be transmitted again are selected first. After that, filled links take priority over the construction of a new stream packet from the TX buffer data.

2.4.3 Receiving

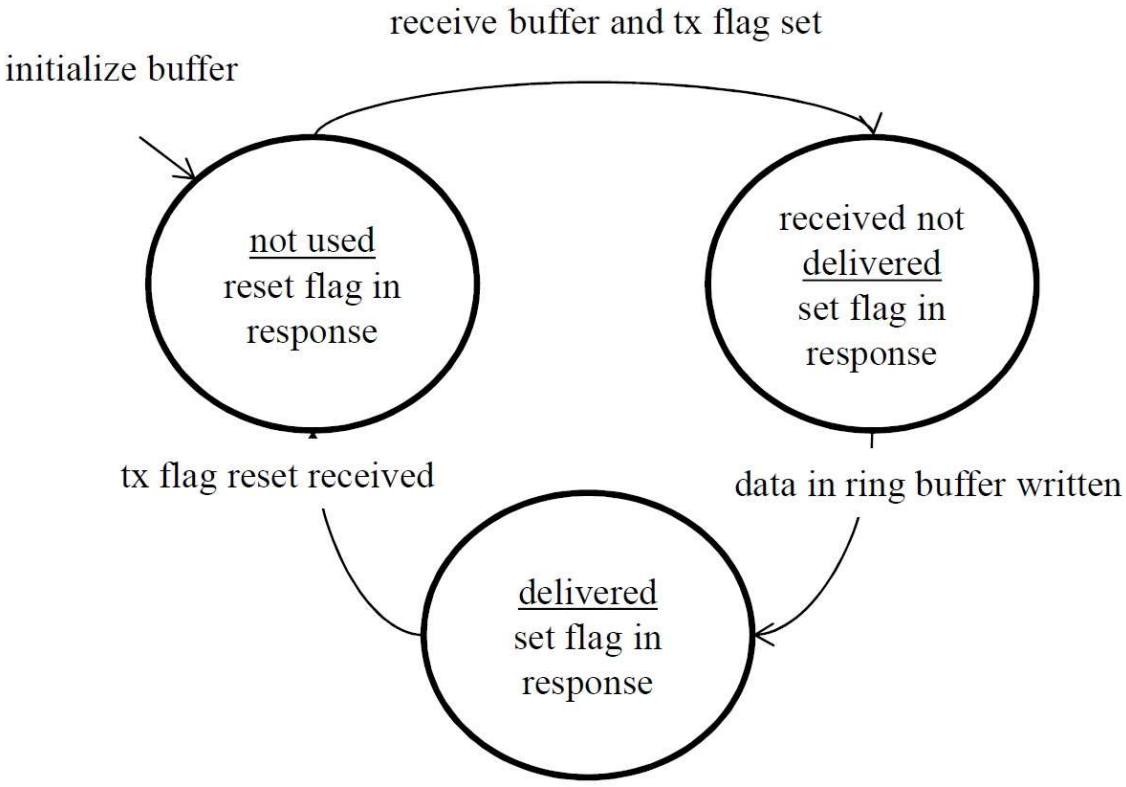

Figure 9: State machine RX virtual link [2]

The state machine on the receiving side is also initialized on the “not used” state. The link layer decodes the header of an incoming stream packet and determines the virtual link for the incoming stream packet and copies the payload data, sequence number

and number of bytes into the virtual link, which changes into the “received not delivered” state. The response flag is set, which is used to confirm the reception to the transmitter later. As described above, the received sequence must be written to a specific index within the RX buffer. If the required space is fully or partially occupied by data from a previous transmission, and not cleared by the next layer, the data is kept within the link. In this case the confirmation flag is not reset, causing the sender to remain in the “receive confirmed” state and not sending again on this specific virtual link. This way a flow control is enacted. Once the data is written successfully to the RX buffer and the confirmation is in turn confirmed by a reset TX flag of the transmitter, the receiving link re-enters the not used state and resets its response flag. Once this flag is communicated, the virtual link at the sender changes into the not used state and a new transmission can begin.

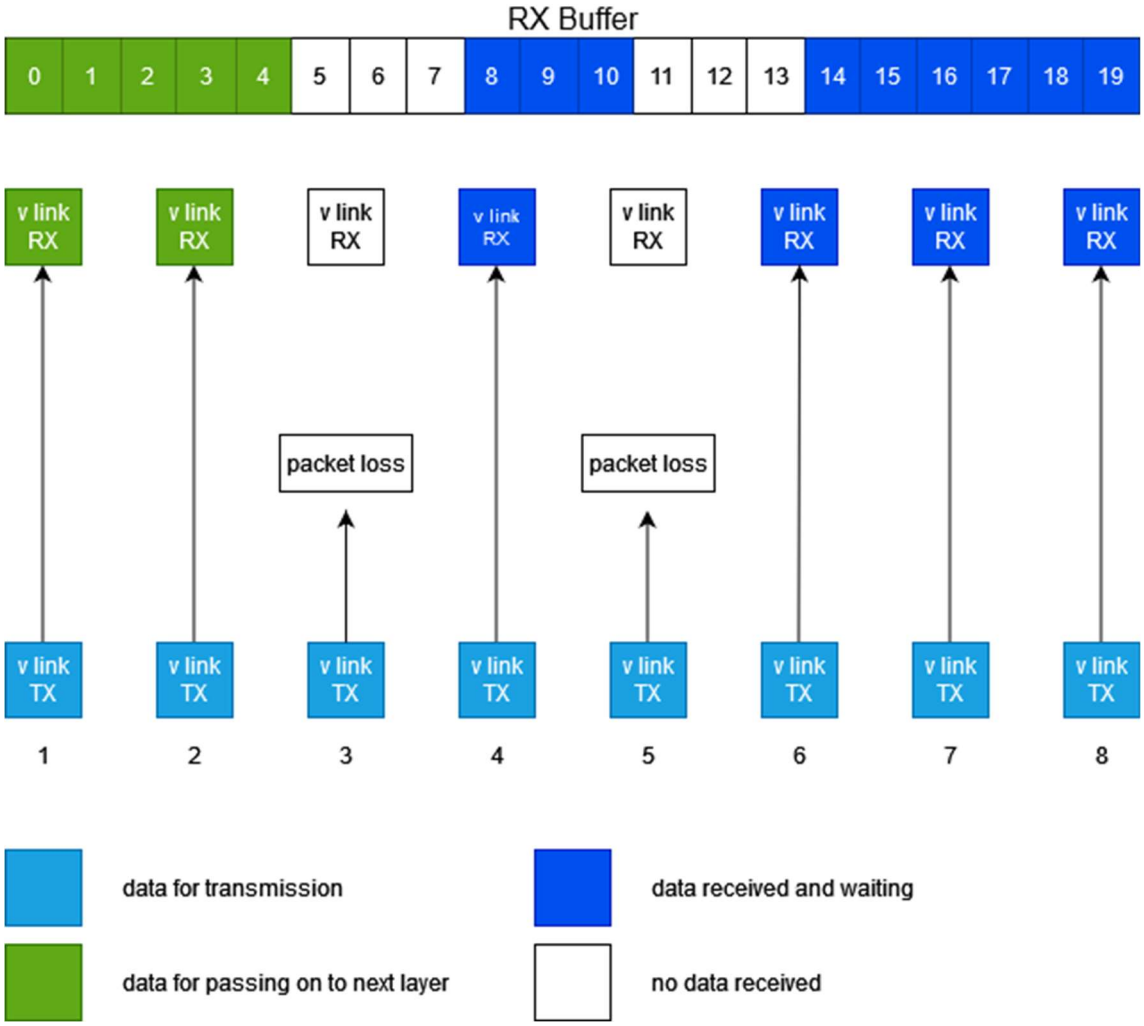

Figure 10: Virtual link transmission within a streamlink

The figure above shows a situation in which eight virtual links on the transmitter have sent data, but the stream packets of the TX links 3 and 5 were lost. The data of all other virtual links was received and written to the RX buffer. Only the sequences of the first two links are a complete part of the stream, represented by the buffer indices 0 - 4. Hence only those sequences are allowed to be passed to the next layer. The data se-

quence transmitted by virtual link 4 is now stored in the buffer indices 8 -10. This part of the stream can only be processed further, after the missing previous part of the stream has been added to the buffer. Once the sequence 5 to 7 is written to the buffer the stream is completed up to index 10 while the sequence 11 to 13 blocks the processing of subsequent data.

2.4.4 Stream Packet Header

To enable the functionality described above a streampacket needs to provide header information with each stream packet. The receiver mirrors the transmitter, meaning the RX buffer is the same in size and structure on both sides as already described and a streampacket is received on the same virtual link as it was sent. Consequently, the header must provide information about the virtual link, sequence number and length of the packet. Additionally, a priority flag is required to distinguish between the two priority levels described above.

data	virtual link id	reserved	priority	sequence number	length
Bit index	0-3	4-6	7	8-23 (16)	24 - 31

Table 3: Stream packet header

2.4.5 Broadcast and Static Responses

As described above, the state machines of a virtual link rely on receiving TX-flags and response-flags. Those are not included in the header information because the overall system does not acknowledge individual packets. Instead, dedicated timeslots exist for transmitting state information. A gateway sends a broadcast in downlink direction to all nodes, while each node gets timeslots assigned to send static responses in uplink direction. Those transmissions are not only used by the streaming service but carry additional data from other services. The gateway sends all state flags of all its streamlinks and all virtual links via a broadcast. A node adds its state flags to its static response.

The number of bytes necessary to contain all state flags of a streamlink depends on the number of virtual links. Furthermore, the size of the broadcast data of the streaming service depends on the number of streamlinks a gateway is operating. Since both values are variable the number of needed bytes is calculated dynamically.

The implementation provided with this document uses eight virtual links, eight TX flags and eight response flags, thus two bytes are sufficient per streamlink.

data	Response flags	TX flags
Bit index	0 - 7	7 - 15

Table 4: State flag bytes for up to 8 virtual links

For 9 to 16 virtual links 4 bytes would be necessary, since two bytes are required for both response and TX flags.

2.5 Scheduling

Static responses are also used to report status data beyond the state flags. That information can then be used by the Dynamic Channel Scheduler. The Dynamic Channel Scheduler is a module of the gateway. It allocates the resources within the dynamic channels of the frame structure [2]. This includes assigning timeslots for nodes to send stream packets. To enable the scheduler to prioritize nodes holding priority data in TX virtual links when assigning timeslots, this information must be reported. Since more than one timeslot might be assigned until the next status report multiple states must are distinguished for each priority level. With two bits for each buffer the following states are made known to the gateway via static responses.

data	regular buffer	priority buffer
bit index	0 - 1	2 - 3

Table 5: Priority data status in static response

code	data
00	no virtual link sending
01	One virtual link sending
10	Two virtual links sending
11	More than two links sending

Table 6: Priority data codes

2.6 Architecture

All functionalities described up to this point lead to an overall model:

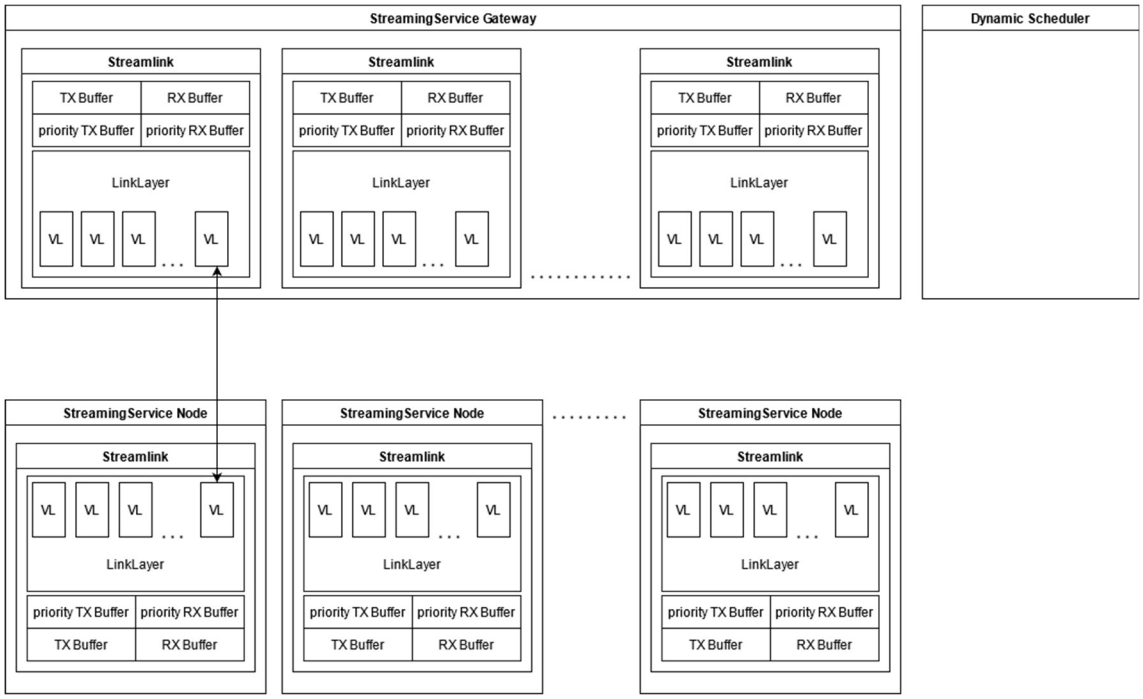

Figure 11: Streaming Service Architecture

This model constitutes the base architecture for the implementation of the streaming service. The central repeated element is the streamlink with all its contents. Beyond that, the gateway requires additional functionality to manage the multitude of streamlinks operated at the same time and the system needs to be able to handle a variable amount of streamlinks and virtual links. The distinct elements of this model have a directly corresponding class within the implementation, so that the source code can be easily related to the model.

3 Implementation

The implementation follows the architecture depicted above in an object orientated approach. It is realized in C++ and restricted to the core link layer functionality. The following sections describe the interfacing of the service implementation with the grid monitoring application and the classes with their operations.

3.1 Provided functions

The streaming service offers the following functions to the surrounding systems:

- **StreamingService** (Constructor)
  To initialize the service, a constructor is provided which expects a boolean parameter to specify if the service is acting as a gateway or not. In addition, the number of streamlinks and the id of the host device is required. If the service is not operating as a gateway the stream link number is ignored.
- **add_data_for_transfer**
  This function is used to add data to a TX buffer for transmission. Any byte sequence specified by a char pointer and length in bytes can be added if it is not exceeding available buffer storage. Further required parameters are the destination device and a boolean as to whether the data should be treated as regular or priority data.
- **build_streaming_packet**
  This function is performing the main service functionality. It is called for creating a stream packet for transmission and for decoding of a received stream packet. As a parameter a PacketControl object is expected which is holding the ids of source and destination device. Furthermore, the object is supposed to provide the length in bytes and a pointer to either allocated memory for the stream packet, which is to be constructed, or the received stream packet which is to be decoded.
- **get_stream_data**
  This function is used to fetch the received and reassembled byte stream from the RX buffer. Once bytes are collected from the RX buffer by this method, the corresponding buffer storage is marked as overridable. The id of the connected device must be specified to determine the corresponding streamlink. A boolean parameter specifies if the regular or priority buffer is called for.
- **get_stream_data_num_bytes**
  This function returns the number of bytes which are ready to be collected from the RX buffer. Id of the connected device and priority level are expected as parameters.
- **get_response_length**

Since the number of flag bytes is dependent on the number of used virtual links, this method provides the current number of bytes needed within a static response.

- **create_static_response**
  This function returns the bytes that are to be added to a static response. This implementation is returning the flag bytes.
- **process_response**
  This function is used to process received flag bytes.
- **get_broadcast_length**
  Since the length of a broadcast is dependent on the number of streamlinks and virtual links, this method provides the current number of bytes needed within a broadcast.
- **create_broadcast**
  This function returns the bytes that are to be added to a broadcast. This implementation is returning the flag bytes only.
- **process_broadcasts**
  This function is used to process a received broadcast.

3.2 Classes

3.2.1 StreamingService

StreamingService
+ m_device_id: uint_t - m_gateway: bool - m_num_links: uint8_t - m_streamlink: StreamLink* -NONE: uint8_t
+ StreamingService(is_gateway:bool, num_links:int, device_id:int):StreamingService + add_data_for_transfer(data:char*, num_bytes:int , target_device_id:uint8_t, priority:bool): void + build_streaming_packet(packet_control:PacketControl*):void + get_stream_data(device_id:uint8_t, data:char*, length:uint16_t, priority:bool):void + process_response(ptr_response:char*):void + get_response(ptr_response:char*):uint8_t + get_link_by_device_id(id:uint8_t):uint8_t + get_stream_data_num_bytes(device_id:uint8_t, priority, bool):uint16_t

The streaming service is a wrapper class exposing all functionality of the service to MAC-pre thread and MAC-post thread. Only one object of this class is required on each device. Like described above, a gateway differs from a node mainly in the number of streamlinks this object is referencing.

Aside from the exposed functions described above, this class manages access to the different streamlinks. Whether data is added, retrieved, send, or received, this class is responsible for operating the streamlink corresponding to the connected device or to provide an unused streamlink to establish a logical connection to a new device.

3.2.2 StreamLink

StreamLink
+ m_device_id: uint8_t + m_state: link_state - m_tx_buffer: Ringbuffer* - m_rx_buffer: Ringbuffer* - m_tx_buffer_priority: Ringbuffer* - m_rx_buffer_priority: Ringbuffer* - m_linklayer: Linklayer* - m_sequence_num: unsigned long - m_sequence_num_priority: unsigned long - M_RINGBUFFER_SIZE: uint16_t - M_VLINK_BUFFER_SIZE: uint16_t - M_NUM_VLINKS: uint8_t
+ build_streaming_packet_tx(ptr_payload:char*, num_bytes:int, priority:bool):void + build_streaming_packet_rx(control:PacketControl):void + add_to_tx_buffer(ptr_payload:char*, num_bytes:int, priority:bool):void + get_num_rx_bytes(data:char*, length:uint16_t, priority:bool):void

Objects of the streaming class are used to coordinate the different logical connections of a gateway. A *link_state* property shows if the link is engaged in a connection or not. If so, the id of the connected device is also stored here.

Each object contains four different buffer objects. For sending and receiving both priority- and regular data. Each buffer is of the class Ringbuffer and has transfer functions to pass buffer data to and from the buffers. This class ensures that the correct priority buffers and are used, and the sequence number of the transmission is maintained. Moreover, this class checks if a new streampacket must be constructed by the link layer or if a previously filled link is up for transmit or if a lost packet must be retransmitted. It tracks if a response or broadcast was received since the last transmission. If no response was received no new information about the receiver is available to decide about necessary retransmissions.

3.2.3 Ringbuffer

Ringbuffer
- m_buffer: char* - m_size:uint16_t - m_occupied: uint16_t - m_available: uint16_t - m_complete: uint16_t - m_tail: uint16_t - m_in: uint16_t - m_out: uint16_t - m_seq: vector<Sequence>
+ Ringbuffer(size: uint16_t):Ringbuffer + write_sequence(data: char*, length: uint8_t, sequence_num: uint16_t):bool + pop_stream(target: char*, length: uint16_t): bool + add_to_buffer(data: char*, length: uint32_t): void + pop_buffer(data: char*, length: int): bool + get_buffer_occupied(): uint16_t + get_free_bytes(): uint16_t + get_complete(): uint16_t - register_seq(seq: Sequence): void

The system operates multiple buffers. As described above a TX buffer and a RX buffer must be identical in size and structure to be able to reassemble the byte stream in correct order via the sequences. To ensure this, all buffer objects are of the same class and initialized with the same size parameter.

They are implemented as ring buffer. If a writing or reading process reaches the end of the fixed sized buffer, the exceeding bytes are written to the start or read from the start of the buffer. This behavior is referred to as a “wrap around” of a ring buffer. All objects of this Ringbuffer class store their size, available space, and occupied space in bytes. Those values are stored as “m_size”, “m_available” and “m_occupied”.

When sending, the TX buffer is passing the bytes to the virtual links in the order they were added to the buffer, following a “first in first out” (FIFO) policy. On the receiving side however, the reassembly of the byte stream requires different and more elaborate operations to handle byte sequences which do not arrive in their original order of transmission and ensure that only completed parts of the stream are passed on to the next layer. Therefore, the Ringbuffer class provides two sets of operations. One for the TX and one for the RX instances.

A TX instance uses the “add_to_buffer” function to load a byte stream for future transmission into the buffer storage and the “pop_buffer” function to take bytes for sending and freeing the buffer storage accordingly. To track the where to write new data and

which bytes are next for sending the storage indices “m_in” and “m_out” are maintained with each execution of the functions. Both functions perform the “wrap around” of a typical ring buffer. The “add_to_buffer” function checks if enough free storage is available and returns a false boolean value if not and aborts the operation.

An RX instance cannot write a received sequence to first index of the free space within the buffer. It is required to write the payload bytes to the exact start address transmitted with the stream packet. This operation is performed by the “write_sequence” function. This function is essential for the reassembly of the stream. To enable this functionality, indices denoting the start and end of the reassembled stream are maintained. These are the called “m_completed” and “m_tail”. Furthermore, beginning and end of sequences which are stored but not adjacent to the completed stream need to be remembered. To this end, a registry of those stored sequences is maintained. The registry is called “m_seq” and is a vector of “Sequence” structs. Each struct holding the start address and length of the sequence. With this data the function performs the following actions:

1. Check if the exact slot of the storage array matching the sequence is free
2. Copying the data to the buffer and performing the wrap around if needed
3. Updating the available and occupied values
4. Check if the added sequence is adjacent to the completed stream
5. If true:
    a. updating the m_complete index
    b. check if subsequent sequences are present by searching the registry
    c. updating the m_complete index for each includable sequence
    d. delete included sequences from the registry
6. If not: add the sequence to the registry

Processing of the registry related operations is mainly encapsulated within the private “register_seq” function. The required slot of the sequence might not be available if previous data was not yet collected by the next layer of the protocol.

To retrieve the received stream from the RX buffer the “pop_stream” function is provided. It returns the completed part of the stream to the caller. It performs the wrap around operation if needed, frees the according storage space by updating the “m_tail” index and updates the available and occupied values.

Additionally, getter functions provide the number of bytes of the completed stream as well as overall occupied and available space to the calling context.

3.2.4 VirtualLinks

VirtualLink
+ m_state: unsigned char + m_priority: bool + m_link_buffer: char* + m_buffer_length: uint8_t + m_buffer_occupied: uint8_t + m_start_address: uint16_t + m_timestamp: time_t
+ VirtualLink():VirtualLink + VirtualLink(uint8_t link_buffer_length):VirtualLink

Objects of the VirtualLink class hold the actual data sequences of the byte stream and all data to operate the link layer state machines. Firstly, this means the state of the link, which is used to determine which actions can be performed on the virtual link and its data. Secondly, it contains the start address and length of the sequence for reassembling the stream at the receiver. Additionally, the class contains a timestamp to track when this stream packet was transmitted. This is used to decide between multiple virtual links waiting to be retransmitted.

3.2.5 LinkLayer

LinkLayer
+ m_num_flag_bytes: uint8_t + m_error_tx: bool + m_error_rx: bool - m_num_links: uint8_t - m_tx_vlinks: VirtualLink* - m_rx_vlinks: VirtualLink* - m_link_states: unsigned char* - m_received_response_flags: unsigned char* - m_received_tx_flags: unsigned char* - m_response_flags: unsigned char* - m_tx_flags: unsigned char* - NONE: uint8_t
+ LinkLayer(buffer_size: uint16_t, num_links: uint8_t, link_buffer_size: uint16_t):LinkLayer + send_bytes(control: PacketControl*, source: Ringbuffer*, num_bytes: int, sequence_num: unsigned long, priority: bool): void + check_delivered(): void + check_receive_confirmed():void + LinkLayer():LinkLayer + receive_bytes(control: PacketControl*): void + get_vlink_states(states: char*): void + update_received_states(states: char*): void + write_to_rx_buffer(ringbuffer: Ringbuffer*, priority_buffer: Ringbuffer*): void + get_header_size(): uint8_t + find_retransmit(): uint8_t + find_filled_link(): uint8_t + retransmit(control, PacketControl*): bool + transmit_filled(control: PacketControl*): bool - find_free_link_tx(): uint8_t - find_free_link_rx(): uint8_t - calc_num_flag_bytes(): uint8_t - read_header(target_buffer: char*): uint8_t - create_header(target_buffer: char*, link_index: uint8_t): void - set_flag(states:unsigned char*, pos: uint8_t): void - reset_flag(states:unsigned char*, pos: uint8_t): void - read_flag(states: unsigned char*, pos: uint8_t): void

While a VirtualLink object contains data and states of individual transmissions of a stream packet, the LinkLayer class performs the necessary operations of these transmissions. An object of this class holds two arrays of eight virtual links. One array for

sending and one for receiving. The operations on each individual virtual link depend on the state of the virtual link and the state of the connected link as shown in the link layer state machines above. Therefore, the state of the connected link is stored in the flag bytes “m_received_response_flags” and “m_received_tx_flags”. These states are obtained by processed broadcasts or static responses which trigger the provided “update_received_states” function of this class. One flag byte contains up to 8 flag bits. The index of a flag bit within a flag byte array corresponds to the index of a virtual link object within an array of virtual links. The first bit in the received response byte and the first bit int the received TX byte is providing information regarding the counterpart of the first virtual link in the “m_rx_vlinks” array. Likewise, the connected device expects status data to be sent. Therefore, the necessary subset of states that is to be transmitted via a broadcast or static response is stored in the flag bytes “m_response_flags” and “m_tx_flags”. Those flag bytes are made accessible via the “get_vlink_states” function to include them in the construction of a broadcast or static response. Because this information is stored at bit level, the LinkLayer class provides functions to set, reset, and read bits by index within the flag byte arrays. Also, a function to calculate the size of the flag byte arrays is provided since it is dependent on the number of virtual links used.

The main functionality of this class is to construct stream packets at the transmitting device and decode incoming stream packets on the receiving device.

For sending the initiating PacketControl object is passed in as a parameter, together with the sequence number, length and priority which was determined by the stream link. The “send_bytes” function finds an unoccupied virtual link. The amount of data calculated by the stream link is transferred from the source buffer to the virtual link and a header for the stream packet is created. According to the header description above this includes the virtual link id, which is the index of the virtual link, the priority flag, start address and payload length in bytes. The packet is copied to the memory location indicated by the PacketControl object for sending. The sequence and state information are also stored in the VirtualLink object to be available in case of a necessary retransmission. The state of the link is updated, and the corresponding TX flag bit is set for a future broadcast or static response.

If the reception of the stream packet is not confirmed with the next incoming broadcast or static response via the received flag bytes, the stream link triggers a retransmit. The “retransmit” function accepts the PacketControl reference as a parameter wh. All other information is present in the streamlink. If multiple virtual links qualify for a retransmit the oldest one is selected for the current PacketControl object. If the sequence of the selected virtual link is not exceeding the number of bytes requested by the PacketControl object, the streampacket is created like during the original transmission attempt. If the current payload length is not sufficient to hold the original sequence, the sequence is split. The sequence of the current virtual link is shortened, a header with the new length is created and the modified stream packet is passed on for sending like during the original transmission attempt. The exceeding bytes are stored as a new sequence

in another virtual link. The new sequence number and length are calculated, and the state of the newly engaged link is set to “filled not transmitted”. The priority level of the original sequence is adopted. This split requires an unused virtual link available. If this is not the case the operation is aborted.

A virtual link engaged by the split operation is prioritized over the creation of a new stream packet on the next transmission. This operation is performed by the “transmit_filled” function. The same situation, that the current requested packet length is shorter than the actual length can occur here, just like on a retransmit. Therefore, this operation works completely analog to a retransmit. A split of virtual links is performed in this case which requires an unused link to manage the exceeding bytes of the original sequence.

When receiving the “receive_bytes” function evaluate the header of the received stream packet. The according virtual RX link is selected by the link id. Priority, start address, length and the payload data are stored in the virtual link. The state of the link is updated to “received not delivered” and the response flag corresponding to the link is set for a future broadcast or static response. After reception the “write_to_buffer” function is triggered. If the sequence is successfully written to the RX buffer the virtual link state changes to “delivered”.

To complete the full state machine cycle as described above, the received information from static responses and broadcasts must be processed. This is done by the “check_receive_confirmed” and “check_delivered” functions. These functions are executed when new flags are received. On a receive confirmed check the function alters the state of a TX virtual link from “transmitted” to “receive confirmed” if a response flag for the link is present and from “receive confirmed” to “not used” if the response flag is reset again. This makes the virtual link available for a new transmission.

The ”check_delivered” function updates the state of an RX link from “delivered” to “not used” if the received sequence is written to the RX buffer and the sender has reset its TX flag, making the link available for the next stream packet reception.

4 Validation

The behavior of the implementation is tested and observed under various conditions. Since the actual platform is not available at the time of development, this validation is restricted to calling the streaming service functions within the development environment. To imitate usage by the MAC-pre and MAC-post thread, reduced versions of PacketControl objects are used as parameters. Those objects were implemented according to the LPWAN project specification [2].

This is the description of the tests which are representative of functionality test during the implementation. The data of the transmission tests serve as proof that the service is working under the tested condition and give an impression of the impact of the altered condition. It is not meant to be a performance measurement.

4.1 Setup

Data packets, which are to be transmitted, stream packets and responses (static responses and broadcasts) outside of the streaming service are represented as structs. They only contain the payload data as a byte array and the length of the data array. Stream packets and responses additionally can store a source device id.

The imitation of sending and receiving of stream packets is encapsulated in functions, which take the references to device objects or ids of the communicating devices as a parameter. These functions initialize the structs described above and a PacketControl object. The sender fills the struct with data by using the according StreamingService function, taking the PacketControl object as a parameter. When this is done, the result is passed on to the receiving StreamingService function, along with a corresponding PacketControl object.

The payload to be streamed is taken from a file. It is converted to a byte stream and reassembled after all bytes have been processed by the service. A successful streaming operation can be easily confirmed if the output file is intact.

One StreamingService object representing a gateway and multiple ones representing nodes are initialized. File data is loaded into a packet struct, so it appears like a raw stream of bytes to the system. The file used for testing is 2510 bytes in size.

4.2 Variables

Streaming data can be done in multiple combinations of conditions. The different types of conditions which are tested are:

- Number of devices streaming
- Uplink streaming

- Downlink streaming
- Various or static stream packet size
- Packet error rate

The basic functionality is tested with one node streaming to a gateway and vice versa. If a test is done with variable stream packet size, the size is determined randomly between 6 and 255 bytes. A test performs a “dry run” at a PER of 0% to make it easier to determine the error source in case of failure.

4.3 Basic copy test

The first test is meant to stream a packet from one device to another using randomly sized stream packets without any packet loss. All virtual links are utilized between responses. After the transmits are done responses are exchanged to go through all stages of the state machines to make the virtual links available for the next iteration of transmissions.

1. Create byte stream from a source file
2. Add bytes to node TX buffer
3. Send eight randomly sized uplink stream packet to gateway
4. Send broadcast
5. Send static response
6. Send broadcast
7. Collect data from the gateway RX buffer and write to target file

Steps 3 to 7 are repeated until all bytes of the source file are written to the target file. This will be referred to as an *iteration*. Testing the downlink direction is very similar and will not be described separately as well as uplink and downlink transmissions being done in turns to test for interferences by the transmitting and receiving operations running simultaneously.

While not close to a real-world scenario this test helps to verify the following requirements:

- State machines operate correctly when stream packets arrive in order
- The byte stream is split up and reassembled without error
- Variable stream packet sizes can be handled
- Headers for the stream packets are created and read correctly
- Creation and processing of static responses and broadcasts is working

4.4 Basic PER Test

The next test is meant to show that lost stream packets are reliably retransmitted, and the stream is correctly reassembled despite packages not arriving in the order they were sent. To show comparable numbers of transmissions the stream packet size is fixed to 100 bytes in this test, including the header. The used data packet requires 27

streampackets to be successfully transmitted to complete the test. Four transmission attempts were used between responses.

The packet error rate does apply to broadcasts and static responses causing the state machines to halt and block transmissions even if the actual stream packets are successfully received. As a point of reference, the count of packets of the completed stream with a PER of 0:

PER %	Iterations	Lost transmissions	Static responses	Lost responses	Broadcasts	Lost broadcasts
0	7	0	7	0	14	0

Table 7: Transmissions with PER 0% and fixed packet size

The streaming service is meant to work under rough channel conditions, therefore a PER of 50% was imitated using pseudo randomly generated numbers to determine the success of a transmission.

PER %	Iterations	Retransmissions	Static responses	Lost responses	Broadcasts	Lost broadcasts
50	50	33	26	24	45	55
50	38	30	16	22	39	37
50	25	18	11	14	26	24
50	30	25	16	14	35	25
50	46	31	25	21	46	46
50	43	35	22	21	47	39

Table 8: Lost packets with PER 50% and fixed packet size

4.5 Full single connection test

Since retransmissions are working correctly, the next test reintroduces the variable stream packet size. Consequently, the number of necessary successful transmissions varies. The next table shows examples of stream packets needed to stream the test file.

PER %	Iterations	Transmissions	Static responses	Broadcasts
0	6	24	6	12
0	4	15	4	8
0	6	24	6	12
0	6	22	6	12

Table 9: Transmissions with PER 0% and variable packet size

Regarding retransmissions this means that the current available slot might not be large enough to carry the original sequence, causing the sequence to be split up into two separate stream packets. In addition to regular retransmits, the new sequence occupies a virtual link and another timeslot, which can cause the number of required transmission-attempts significantly.

PER %	Iterations	Transmissions	Retransmissions	Static responses	Lost responses	Broadcasts	Lost broadcasts	Splits
50	16	28	13	9	7	21	11	7
50	58	32	42	28	30	56	60	9
50	72	33	52	40	32	78	66	9
50	20	25	14	11	9	26	14	6
50	38	31	28	14	24	38	38	8
50	38	32	30	22	16	42	34	6

Table 10: Transmissions with PER 50% and variable packet size

4.6 Multi connection test

Finally, the system is required to maintain streams from multiple nodes to one gateway and from the gateway to nodes simultaneously. In this setup three node objects are streaming to the gateway, while the gateway itself is streaming to one of the nodes.

PER %	Iterations	Transmissions 3 nodes ang gateway	Static responses 3 nodes	Broadcasts
0	5	55	15	10
0	5	56	15	10
0	6	69	18	12
0	6	62	18	12

Table 11: Transmissions of 4 devices with PER 0% and variable packet size

PER %	Iterations	Transmissions	Retransmissions	Static responses	Lost responses	Broadcasts	Lost broadcasts	Splits
50	37	87	76	51	60	45	29	22
50	95	99	133	146	139	94	96	54

50	50	89	105	78	72	53	47	21
50	57	84	76	84	87	55	59	52

Table 12: Transmissions of 4 devices with PER 50% and variable packet size

The completion of this test shows that the system can operate sending and receiving from multiple devices in parallel while dealing with a variable packet size and a PER of 50%, which was the overall goal for the implementation to achieve.

The numbers of the multi connection test are not directly comparable to those of the other tests. The test cycles until all devices have successfully transmitted the test file. One single “unlucky” device regarding the packet loss can cause the test to continue and add transmission attempts while all other nodes might have completed the transmission way sooner. The test does show however, that the number of needed transmission attempts at PER 50% can vary significantly despite the testing conditions being unchanged.

5 Conclusion

Reliable data transfer is desired on any kind of communication channel. Many protocols achieve reliable data transfer but depending on the MAC layer and differences of channel usage and characteristics different mechanisms are required. In case of scarce channel resources efficiency is not a feature but a necessary requirement.

In this document a link layer solution for powerline communication [5] was adopted to suit a LPWAN LoRa radio system. Both PLC and LoRa radio are prone to suffer from high PER. The major difference being the LoRa system relying on dedicated response messages, instead of payload attached header information as the PLC system.

An object orientated C++ implementation of a service for streaming data between a gateway and multiple nodes could be provided. It manages the different connections of a gateway to multiple nodes enabling sending and receiving in both directions for each device. Its core functionality of delivering a byte stream correctly and reliable between two individual devices was verified. The implementation is ready to accommodate different configurations by dynamically adapting to the number of devices and virtual links between two individual devices. The service implementation provides a set of exposed for easy usage by the project context [2].

While more functionality regarding further interfacing with the context, security and optimization is desirable, the reliability of the data transfer across multiple streaming devices was achieved.

Attachment

File: “sourcecode_stream_service.zip”

Bibliography

[1] Semtech Corporation, "AN1200.22 LoRa™ Modulation Basics," 2015.

[2] G. Bumiller, "Communication and Interface Protocol for LPWAN for PQ and Grid Condition Monitoring Applications," 2021.

[3] S. Galli, A. Scaglione and Z. Wang, "For the Grid and Through the Grid: The Role of Power Line Communications in the Smart Grid," *Proceedings of the IEEE,* pp. vol. 99, no. 6, pp. 998-1027, June 2011.

[4] Semtech Corporation, "LoRa Developer Portal," Semtech, [Online]. Available: https://lora-developers.semtech.com/library/tech-papers-and-guides/lora-and-lorawan/. [Accessed 1 September 2021].

[5] G. Bumiller, "Powerline-Channel Adopted Layer-Design and Link-Layer for Reliable Data Transmission," *IEEE International Symposium on Power Line Communications and Its Applications (ISPLC),* 2015.

[6] J. Fey, G. Hallak and G. Bumiller, "Efficient Central Resource Management Algorithm," *International Symposium on Power Line Communications and its Applications (ISPLC),* 2016.

[7] J. Fey, Weiterentwicklung eines Power-line Communication MAC-Layer und Implementierung in einem Linux Netzwerktreiber, 2017.

Erklärung

Ich versichere an Eides statt durch meine Unterschrift, dass ich die vorliegende Arbeit selbständig und ohne fremde Hilfe angefertigt und an allen Stellen, die ich wörtlich oder annähernd wörtlich aus Veröffentlichungen entnommen habe, als solche kenntlich gemacht habe, mich auch keiner anderen als der angegebenen Literatur oder sonstiger Hilfsmittel bedient habe. Des Weiteren hat die Arbeit in dieser oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen.

Digital unterschrieben von
Jan Dehlen
Datum: 2021.10.22
'18:39:45 +02'00

Bochum, 22.10.2021

Ort, Datum

Unterschrift