Universitá di Pisa



Master's Degree in Embedded Computing Systems

Component Based Software Design

# PC - Arduino
## *Relieable Data Transfer Protocol*

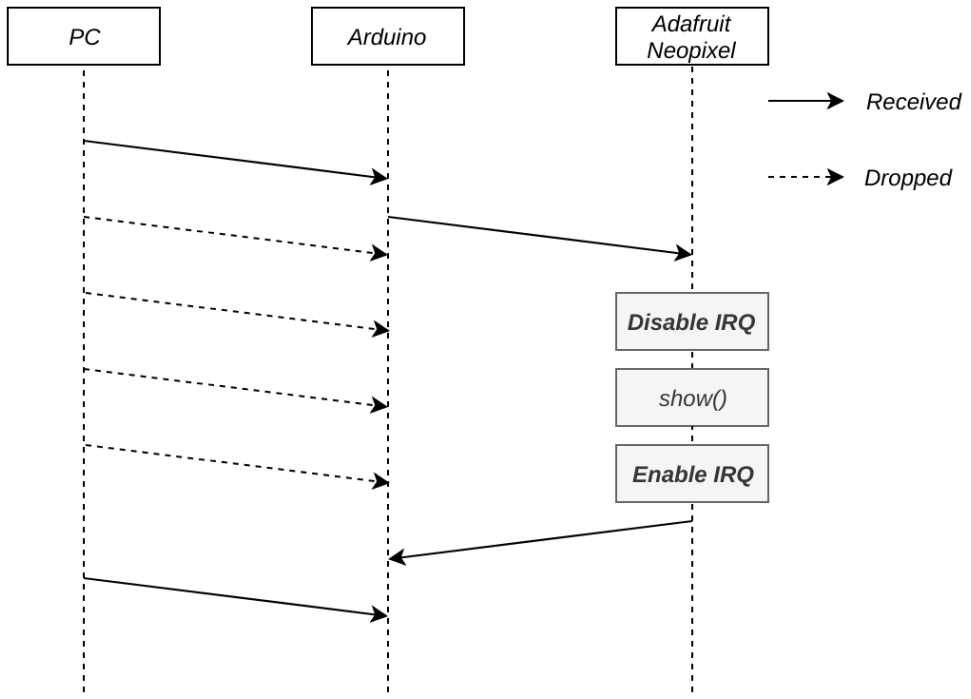
*A.Y. 2018/2019*

Michele Baldassini

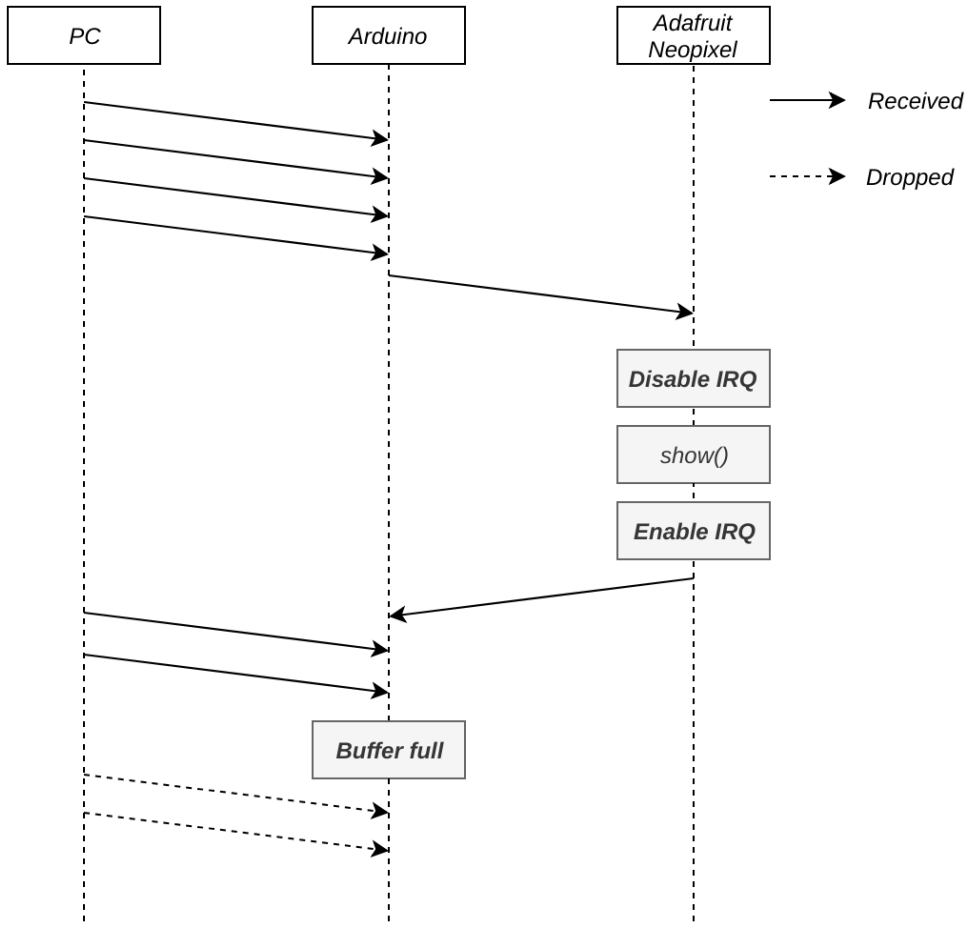
# Contents

# 1 Introduction

This document is a report for the project required for the course of Component-Based Software Design for Masters Degree in Embedded Computing Systems during A.Y. 2017/2018.
This project aims to design and implement a reliable communications protocol for the open-source Arduino-based tutor for MuseScore.
The need for a reliable protocol stems from the fact that certain tasks that a microcontroller must perform disable interrupt because of some sensitive timing required.
In our case, we had to light up a strip of LEDs (Adafruit Neopixel), whose function to show the color of the LED disables interrupts. Without a software side delay in sending packets (containing LED ids, and color in RGB format) many of the packets are discarded arriving in the time interval in which the interrupts are disabled.
Another problem that may arise is that of saturating the reception buffer of our Arduino, which, if it happens, would lead to a loss of the packets arrived at a full buffer.
To solve the problem of packet loss, a reliable protocol was therefore implemented, which allows the sender to detect when a packet has not been correctly received and send it again until confirmation of receipt is received.
Two possible scenarios are shown in the following figures.




(a)




(b)

Figure 1: Possible packet losses. In (a) it is possible to notice that the packets received when the interruptions are disabled are not inserted in the serial reception buffer. In (b) instead we note how, if the receiver does not empty the reception buffer quickly, this fills and subsequent data are discarded.

# 2 Protocol Design

The study of protocol design deals with algorithms for achieving reliable, efficient communication of whole units of information called frames (rather than individual bits, as in the physical layer) between two adjacent machines. By adjacent, we mean that the two machines are connected by a communication channel that acts conceptually like a wire. The essential property of a channel that makes it "wire-like" is that the bits are delivered in exactly the same order in which they are sent.
Unfortunately, communication channels make errors occasionally. Furthermore, they have only a finite data rate, and there is a nonzero propagation delay between the time a bit is sent and the time it is received. These limitations have important implications for the efficiency of the data transfer. The protocols used for communications must take all these factors into consideration.
In the following sections to explain the protocol, we will identify the *ReliabelDataTransfer* class as the *application layer*, the *Protocol* class as the *data link layer* and the *PhysicalLayer* class as the *physical layer*.

## 2.1 Protocol design issue

The data link layer uses the services of the physical layer to send and receive bits over communication channels. It has a number of functions, including:

1. Providing a well-defined service interface to the application.
2. Dealing with transmission errors.
3. Regulating the flow of data so that slow receivers are not swamped by fast senders.

To accomplish these goals, the data link layer takes the packets it gets from the application layer and encapsulates them into frames for transmission. Each frame contains a frame header, a payload field for holding the packet, and a frame trailer. Frame management forms the heart of what the data link layer does.
The principal service, that the data link layer provides to the application layer, is transferring data from the application layer on the source machine to the application layer on the destination machine. On the source machine is an entity, in the application layer that hands some bits to the data link layer for transmission to the destination. The job of the data link layer is to transmit the bits to the destination machine so they can be handed over to the application layer there.
The most sophisticated service the data link layer can provide to the application layer is connection-oriented service. With this service, the source and destination machines establish a connection before any data are transferred. Each frame sent over the connection is numbered, and the data link layer guarantees that each frame sent is indeed received. Furthermore, it guarantees that each frame is received exactly once and that all frames are received in the right order. Connection-oriented service thus provides the application layer processes with the equivalent of a reliable bit stream.

## 2.2 Framing

To provide service to the application layer, the data link layer must use the service provided to it by the physical layer. What the physical layer does is accept a raw bit stream and attempt to deliver it to the destination. If the channel is noisy, the physical layer will add some redundancy

to its signals to reduce the bit error rate to a tolerable level. However, the bit stream received by the data link layer is not guaranteed to be error free. Some bits may have different values and the number of bits received may be less than, equal to, or more than the number of bits transmitted. It is up to the data link layer to detect and, if necessary, correct errors.
The usual approach is for the data link layer to break up the bit stream into discrete frames, compute a short token called a checksum for each frame, and include the checksum in the frame when it is transmitted. When a frame arrives at the destination, the checksum is recomputed. If the newly computed checksum is different from the one contained in the frame, the data link layer knows that an error has occurred and takes steps to deal with it (e.g., discarding the bad frame and possibly also sending back an error report). Breaking up the bit stream into frames is more difficult than it at first appears. A good design must make it easy for a receiver to find the start of new frames. The chosen framing method plan to have a configurable number of bytes per frame known both the sender and the receiver. Thus, the receiver will always wait to have received an entire frame, knowing its a priori dimension.

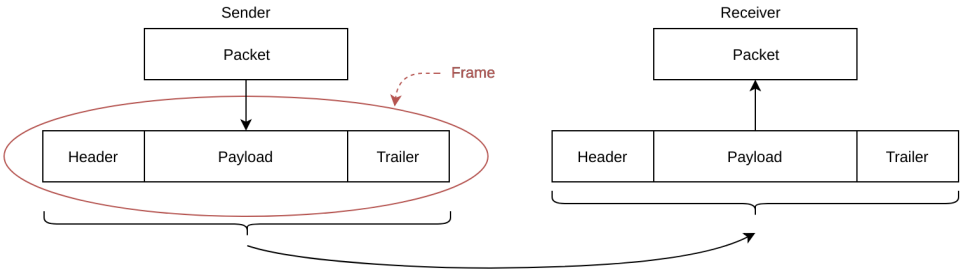



Figure 2: Relationship between packets and frames.

## 2.3 Flow Control

Another important design issue that occurs in the data link layer is what to do with a sender that systematically wants to transmit frames faster than the receiver can accept them. This situation can occur when the sender is running on a fast, powerful computer and the receiver is running on a slow, low-end machine.
Clearly, something has to be done to prevent this situation. The used approach is the feedback-based flow control in which the receiver sends information to the sender giving it permission to send more data.

# 3 Protocol Implementation

As far as the data link layer is concerned, the packet passed across the interface to it from the application layer is pure data, whose every bit is to be delivered to the destinations application layer. The fact that the destinations application layer may interpret part of the packet as a header is of no concern to the data link layer.

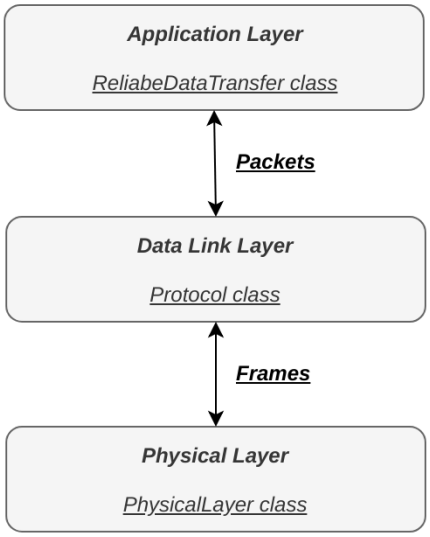



Figure 3: Implementation of the physical, data link, and application layers.

When the data link layer accepts a packet, it encapsulates the packet in a frame by adding a data link header and trailer to it. Thus, a frame consists of an embedded packet, some control information (in the header), and a checksum (in the trailer). The frame is then transmitted to the data link layer on the other machine. At lower level, there exist suitable library procedures *to_physical_layer* to send a frame and *from_physical_layer* to receive a frame. These procedures compute and append or check the checksum.
The receiver, after sending information to the sender giving it permission to send more data, has nothing to do. It just sits around waiting for something to happen. We will indicate that the data link layer is waiting for something to happen by the procedure call *wait_for_event(event)*. This procedure only returns when something has happened (e.g., a frame has arrived). Upon return, the variable event tells what happened.
When a frame arrives at the receiver, the checksum is recomputed. If the checksum in the frame is incorrect (i.e., there was a transmission error), the data link layer is so informed (*event* = *cksum_err*). If the inbound frame arrived undamaged, the data link layer is also informed (*event* = *frame_arrival*) so that it can acquire the frame for inspection using the function *from_physical_layer*.
As soon as the receiving data link layer has acquired an undamaged frame, it checks the control information in the header, and, if everything is all right, passes the packet portion to the application layer. Under no circumstances is a frame header ever given to a application layer.
A frame is composed of four fields: *kind*, *seq*, *ack*, and *info*, the first three of which contain control information and the last of which may contain actual data to be transferred. These control fields are collectively called the *frame header*.

The *kind* field tells whether there are any data in the frame, because data link layer distinguishes frames containing only control information from those containing data as well. The *seq* and *ack* fields are used for sequence numbers and acknowledgements, respectively. The *info* field of a data frame contains a single *packet* (which is the unit of information exchanged between the application layer and the data link layer on the same machine); the *info* field of a control frame is not used.
The application layer builds a packet by taking a message from user and adding the application layer header to it. This packet is passed to the data link layer for inclusion in the info field of an outgoing frame. When the frame arrives at the destination, the data link layer extracts the packet from the frame and passes the packet to the application layer.
The procedures *to_application_layer* and *from_application_layer* are used by the data link layer to pass packets to the application layer and accept packets from the application layer, respectively. Note that *from_physical_layer* and *to_physical_layer* pass frames between the data link layer and the physical layer. In other words, *to_application_layer* and *from_application_layer* deal with the interface between layers 2 and 3, whereas *from_physical_layer* and *to_physical_layer* deal with the interface between layers 1 and 2.
We assume that the channel is unreliable and loses entire frames upon occasion. To be able to recover from such calamities, the sending data link layer must start an internal timer or clock whenever it sends a frame. If no reply has been received within a certain predetermined time interval, the clock times out and the data link layer receives an interrupt signal. This is handled by allowing the procedure wait for event to return (*event = timeout*). The procedures *start_timer* and *stop_timer* turn the timer on and off, respectively. Timeout events are possible only when the timer is running and before *stop_timer* is called. It is explicitly permitted to call *start_timer* while the timer is running; such a call simply resets the clock to cause the next timeout after a full timer interval has elapsed (unless it is reset or turned off).
The procedures *start_ack_timer* and *stop_ack_timer* control an auxiliary timer used to generate acknowledgements under certain conditions. The procedures *enable_protocol* and *disable_protocol* are used where we no longer assume that the application layer always has packets to send. When the data link layer enables the protocol, the application layer is then permitted to interrupt when it has a packet to be sent. We indicate this with (*event = send_ready*). When the application layer is disabled, it may not cause such events. By being careful about when it enables and disables its application layer, the data link layer can prevent the application layer from swamping it with packets for which it has no buffer space. Frame sequence numbers are always in the range 0 to *MAX_SEQ* (inclusive). It is frequently necessary to advance a sequence number by 1 circularly (i.e., *MAX_SEQ* is followed by 0). The macro *inc* performs this incrementing.

## 3.1 Sliding Window Protocol

One way of achieving full-duplex data transmission is to use the same link for data in both directions. By looking at the *kind* field in the header of an incoming frame, the receiver can tell whether the frame is data or an acknowledgement.
When a data frame arrives, instead of immediately sending a separate control frame, the receiver restrains itself and waits until the application layer passes it the next packet. The acknowledgement is attached to the outgoing data frame (using the *ack* field in the frame header). In effect, the acknowledgement gets a free ride on the next outgoing data frame. The technique of temporarily delaying outgoing acknowledgements so that they can be hooked onto the next outgoing data frame is known as *piggybacking*.

The principal advantage of using piggybacking over having distinct acknowledgement frames is a better use of the available channel bandwidth. The *ack* field in the frame header costs only a few bits, whereas a separate frame would need a header, the acknowledgement, and a checksum. In addition, fewer frames sent generally means a lighter processing load at the receiver. The piggyback field costs only 1 bit in the frame header. It rarely costs more than a few bits.

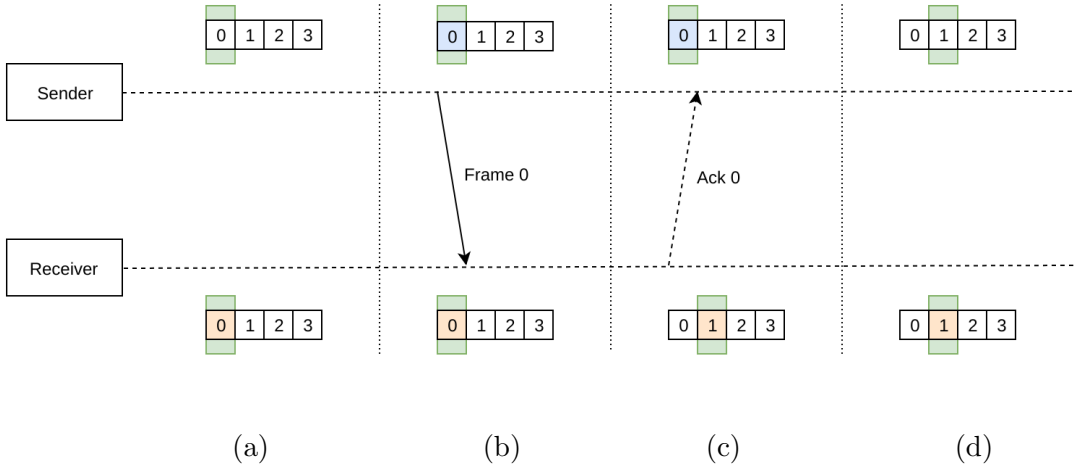


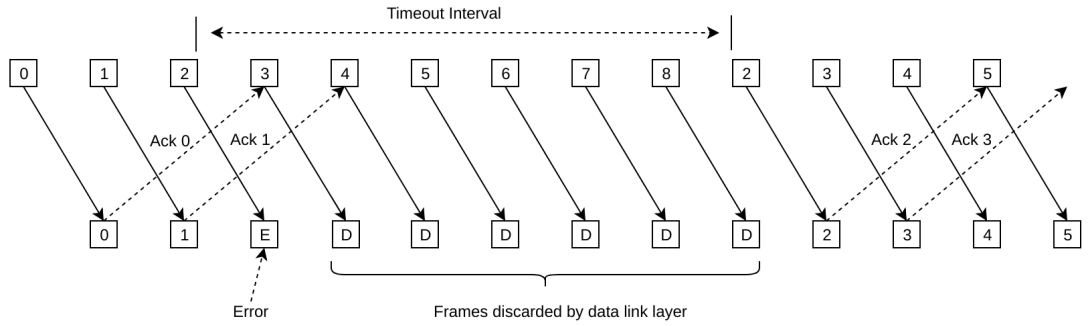

Figure 4: A sliding window of size 1, with a 2-bit sequence number. (a) Initially. (b) After the first frame has been sent. (c) After the first frame has been received. (d) After the first acknowledgement has been received.

However, piggybacking introduces a complication not present with separate acknowledgements. The receiver can jam the service if it has nothing to send. This can be solved by enabling a counter (*ack timeout*) when a data frame is received. If the count ends and there is no data frame to send, the receiver will send an *ack* control frame. The sender also adds a counter (*timeout*), if the counter ends without receiving confirmation, the sender assumes packet loss, and sends the frame again.
The essence of all sliding window protocols is that at any instant of time, the sender maintains a set of sequence numbers corresponding to frames it is permitted to send. These frames are said to fall within the sending window. Similarly, the receiver also maintains a receiving window corresponding to the set of frames it is permitted to accept. The senders window and the receivers window need not have the same lower and upper limits or even have the same size. In our implementation they are configurable in size.
The sequence numbers within the senders window represent frames that have been sent or can be sent but are as yet not acknowledged. Whenever a new packet arrives from the application layer, it is given the next highest sequence number, and the upper edge of the window is advanced by one. When an acknowledgement comes in, the lower edge is advanced by one. In this way the window continuously maintains a list of unacknowledged frames.
Since frames currently within the senders window may ultimately be lost or damaged in transit, the sender must keep all of these frames in its memory for possible retransmission. Thus, if the maximum window size is $N$, the sender needs $N$ buffers to hold the unacknowledged frames. If the window ever grows to its maximum size, the sending data link layer must forcibly shut off the application layer until another buffer becomes free.
The receiving data link layers window corresponds to the frames it may accept. Any frame falling within the window is put in the receivers buffer. When a frame whose sequence number

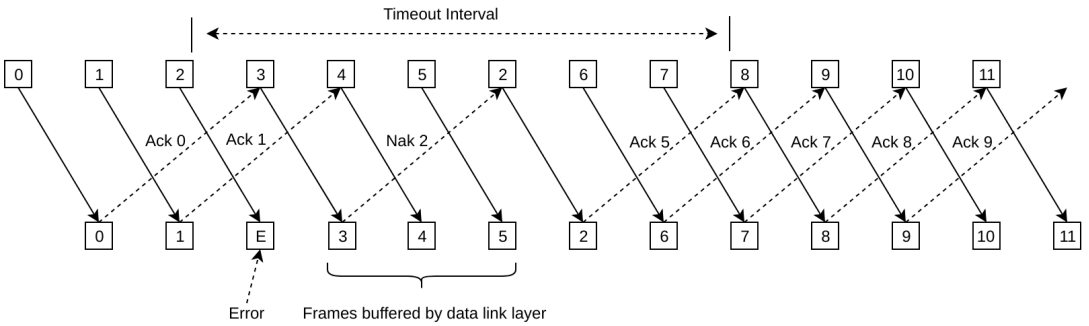
is equal to the lower edge of the window is received, it is passed to the application layer and the window is rotated by one. Any frame falling outside the window is discarded. In all of these cases, a subsequent acknowledgement is generated so that the sender may work out how to proceed. A window size of 1 means that the data link layer only accepts frames in order, but for larger windows this is not so. The application layer, in contrast, is always fed data in the proper order, regardless of the data link layers window size.

### 3.1.1 A Protocol using Go-Back-N

The transmission time required for a frame to arrive at the receiver plus the transmission time for the acknowledgement to come back is not negligible.
This problem can be viewed as a consequence of the rule requiring a sender to wait for an acknowledgement before sending another frame. If we relax that restriction, much better efficiency can be achieved. Basically, the solution lies in allowing the sender to transmit up to $N$ frames before blocking, instead of just 1.
This technique of keeping multiple frames in flight is an example of pipelining. Pipelining frames over an unreliable communication channel raises some serious issues. When a damaged frame arrives at the receiver, it obviously should be discarded, but what should the receiver do with all the correct frames following it? Remember that the receiving data link layer is obligated to hand packets to the application layer in sequence.




(a)




(b)

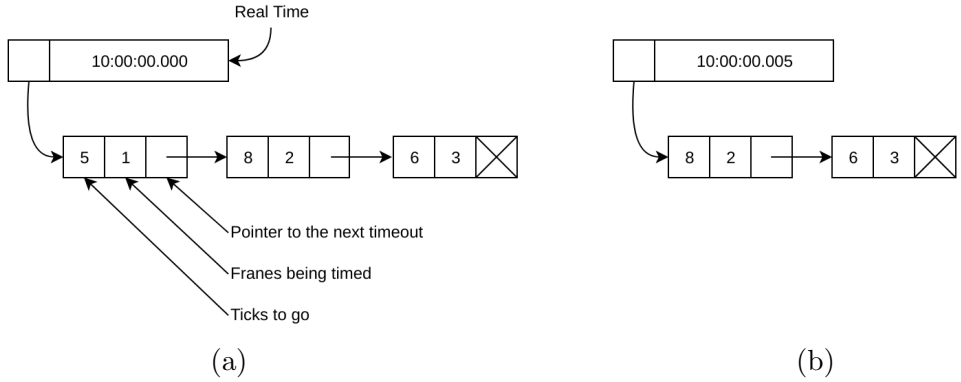
Figure 5: Pipelining and error recovery. Effect of an error when (a) receivers window size is 1 and (b) receivers window size is large.

One option, called **go-back-n**, is for the receiver simply to discard all subsequent frames, sending no acknowledgements for the discarded frames. This strategy corresponds to a receive window of size 1. In other words, the data link layer refuses to accept any frame except the next one it must give to the application layer.
If the senders window fills up before the timer runs out, the pipeline will begin to empty. Eventually, the sender will time out and retransmit all unacknowledged frames in order, starting with the damaged or lost one.
The other general strategy for handling errors when frames are pipelined is called **selective repeat**. When it is used, a bad frame that is received is discarded, but any good frames received after it are accepted and buffered. When the sender times out, only the oldest unacknowledged frame is retransmitted. If that frame arrives correctly, the receiver can deliver to the application layer, in sequence, all the frames it has buffered. Selective repeat corresponds to a receiver window larger than 1. This approach can require large amounts of data link layer memory if the window is large.
Selective repeat is often combined with having the receiver send a negative acknowledgement (*nak*) when it detects an error, for example, when it receives a checksum error or a frame out of sequence. *nak*s stimulate retransmission before the corresponding timer expires and thus improve performance.
If the *nak* should get lost, eventually the sender will time out for frame and send it (and only it) of its own accord, but that may be a quite a while later.
When the application layer has a packet it wants to send, it can cause a *send_ready* event to happen. However, to enforce the flow control limit on the sender window or the number of unacknowledged frames that may be outstanding at any time, the data link layer must be able to keep the application layer from bothering it with more work. The library procedures *enable_protocol* and *disable_protocol* do this job. The maximum number of frames that may be outstanding at any instant is not the same as the size of the sequence number space. For *go-back-n*, *MAX_SEQ* frames may be outstanding at any instant, even though there are (*MAX_SEQ* + 1 distinct sequence numbers (which are 0, 1, . . . , *MAX_SEQ*). To see why this restriction is required, consider the following scenario with (*MAX_SEQ = 7)*:

1. The sender sends frames 0 through 7.
2. A piggybacked acknowledgement for 7 comes back to the sender.
3. The sender sends another eight frames, again with sequence numbers 0 through 7.
4. Now another piggybacked acknowledgement for frame 7 comes in.

The sender has no way of telling. For this reason the maximum number of outstanding frames must be restricted to *MAX_SEQ*.
Since a sender may have to retransmit all the unacknowledged frames at a future time, it must hang on to all transmitted frames until it knows for sure that they have been accepted by the receiver. When an acknowledgement comes in for frame *n*, frames *n - 1*, *n - 2*, and so on are also automatically acknowledged. This type of acknowledgement is called a *cumulative acknowledgement*. This property is especially important when some of the previous acknowledgement-bearing frames were lost or garbled. Whenever any acknowledgement comes in, the data link layer checks to see if any buffers can now be released. If buffers can be released (i.e., there is some room available in the window), application layer can be allowed to cause more *send_ready* events.
Go-Back-N has multiple outstanding frames, it logically needs multiple timers, one per outstanding frame. Each frame times out independently of all the other ones. However, all of these

timers can easily be simulated in software using a single hardware clock that causes interrupts periodically. The pending timeouts form a linked list, with each node of the list containing the number of clock ticks until the timer expires, the frame being timed, and a pointer to the next node.




(a) (b)

Figure 6: Simulation of multiple timers in software. (a) The queued timeouts. (b) The situation after the first timeout has expired.

Every time the hardware clock ticks, the real time is updated and the tick counter at the head of the list is decremented. When the tick counter becomes zero, a timeout is caused and the node is removed from the list. Although this organization requires the list to be scanned when *start_timer* or *stop_timer* is called, it does not require much work per tick. Both of these routines have been given a parameter indicating which frame is to be timed.

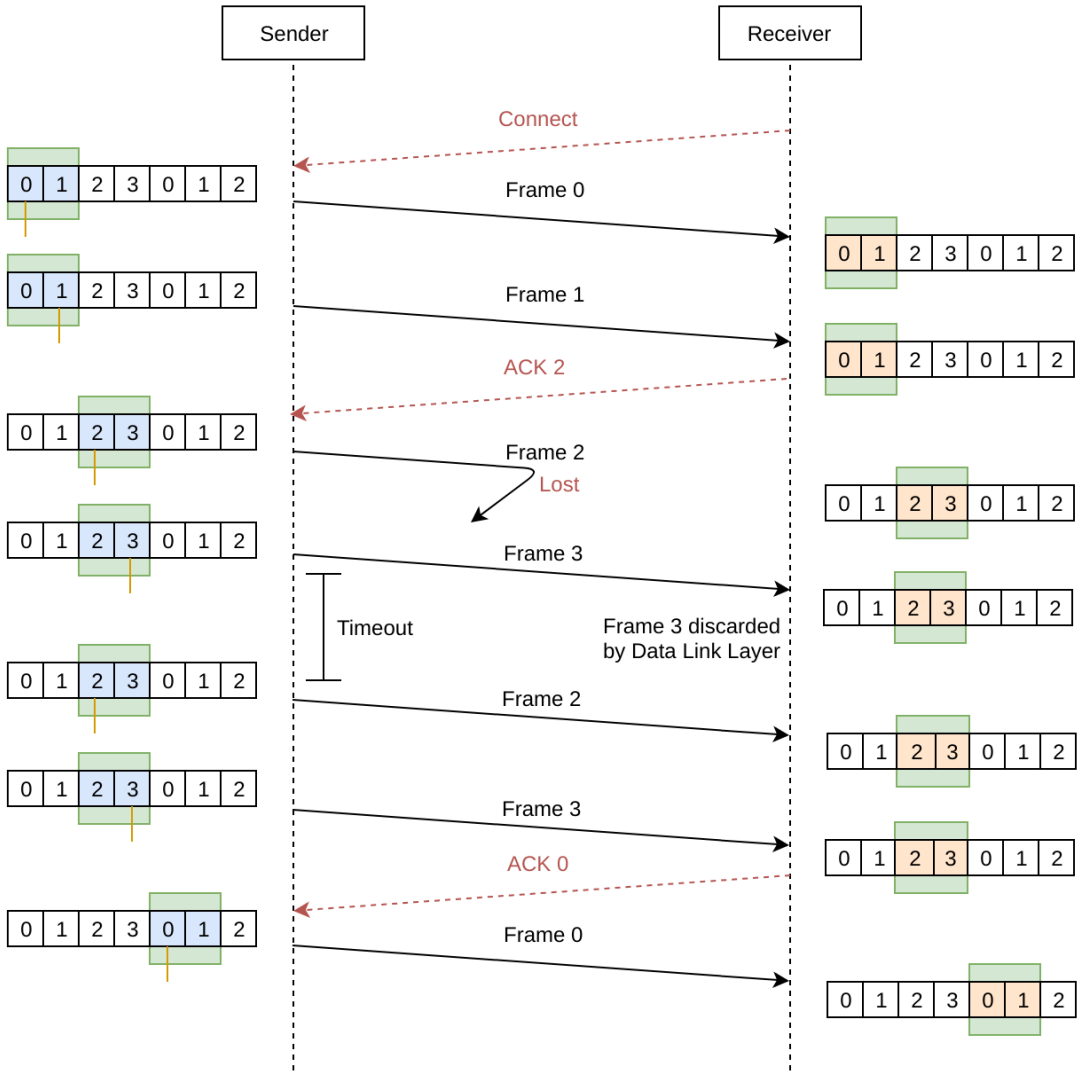



Figure 7: Go-Back-N implementation. The sender starts sending packets after receiving the connect signal from the receiver that communicates that it is ready to receive data. Having a window size of 1, the receiver discards any time it receives an out-of-sequence packet. The receiver expects to receive, in order, a number of frames equal to the size of the sequence.

### 3.1.2 A Protocol using Selective Repeat

In this protocol, both sender and receiver maintain a window of outstanding and acceptable sequence numbers, respectively. The senders window size starts out at 0 and grows to some predefined maximum. The receivers window, in contrast, is always fixed in size and equal to the predetermined maximum. The receiver has a buffer reserved for each sequence number within its fixed window. Associated with each buffer is a bit (*arrived*) telling whether the buffer is full or empty. Whenever a frame arrives, its sequence number is checked by the function *between* to see if it falls within the window. If so and if it has not already been received, it is accepted and stored. This action is taken without regard to whether or not the frame contains the next packet expected by the application layer. Of course, it must be kept within the data link layer and not passed to the application layer until all the lower-numbered frames have already been

delivered to the application layer in the correct order.
The problem is that after the receiver advanced its window, the new range of valid sequence numbers may overlap the old one. Consequently, the following batch of frames might be either duplicates (if all the acknowledgements were lost) or new ones (if all the acknowledgements were received). The receiver has no way of distinguishing these two cases. The solution to this problem lies in making sure that after the receiver has advanced its window there is no overlap with the original window. To ensure that there is no overlap, the maximum window size should be at most half the range of the sequence numbers.
Consequently, the number of buffers needed is equal to the window size, not to the range of sequence numbers. The number of timers needed is equal to the number of buffers, not to the size of the sequence space. Effectively, a timer is associated with each buffer. When the timer runs out, the contents of the buffer are retransmitted.
If the reverse traffic (frames being sent in the reverse direction on which to piggyback acknowledgements) is light, the acknowledgements may be held up for a long period of time, which can cause problems. In the extreme, if there is a lot of traffic in one direction and no traffic in the other direction, the protocol will block when the sender window reaches its maximum.

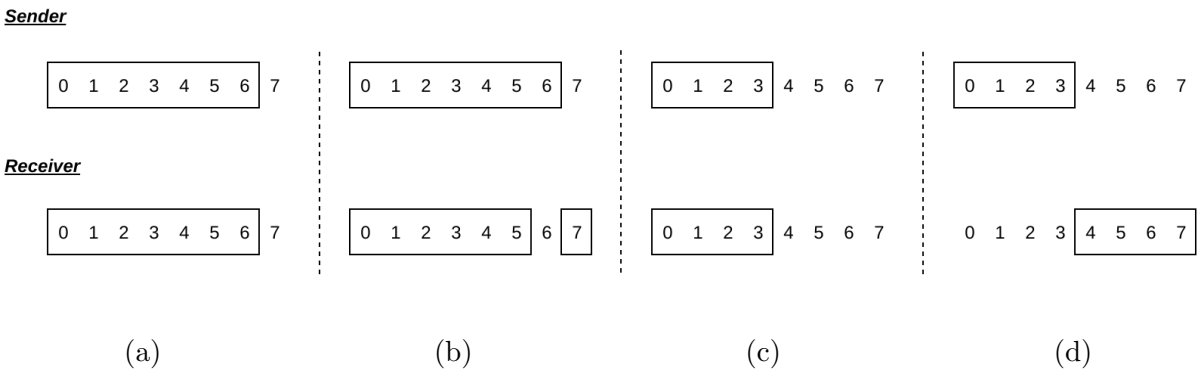



Figure 8: (a) Initial situation with a window of size 7. (b) After 7 frames have been sent and received but not acknowledged. (c) Initial situation with a window size of 4. (d) After 4 frames have been sent and received but not acknowledged.

To relax this assumption, an auxiliary timer is started by *start_ack_timer* after an in-sequence data frame arrives. If no reverse traffic has presented itself before this timer expires, a separate acknowledgement frame is sent. An interrupt due to the auxiliary timer is called an *ack_timeout* event. With this arrangement, traffic flow in only one direction is possible because the lack of reverse data frames onto which acknowledgements can be piggybacked is no longer an obstacle. Only one auxiliary timer exists, and if *start_ack_timer* is called while the timer is running, it has no effect. The timer is not reset or extended since its purpose is to provide some minimum rate of acknowledgements.
It is essential that the timeout associated with the auxiliary timer be appreciably shorter than the timeout used for timing out data frames. This condition is required to ensure that a correctly received frame is acknowledged early enough that the frames retransmission timer does not expire and retransmit the frame.
Whenever the receiver has reason to suspect that an error has occurred, it sends a negative acknowledgement (*nak*) frame back to the sender. Such a frame is a request for retransmission of the frame specified in the *nak*. In two cases, the receiver should be suspicious: when a damaged frame arrives or a frame other than the expected one arrives (potential lost frame). To avoid

making multiple requests for retransmission of the same lost frame, the receiver should keep track of whether a *nak* has already been sent for a given frame. The variable *no_nak* is true if no *nak* has been sent yet for *frame_expected*. If the *nak* gets mangled or lost, no real harm is done, since the sender will eventually timed out and retransmit the missing frame anyway. If the wrong frame arrives after a *nak* has been sent and lost, *no_nak* will be true and the auxiliary timer will be started. When it expires, an *ack* will be sent to resynchronize the sender to the receivers current status.
We assume also that the variable *oldest_frame* is set upon timeout to indicate which frame timed out.

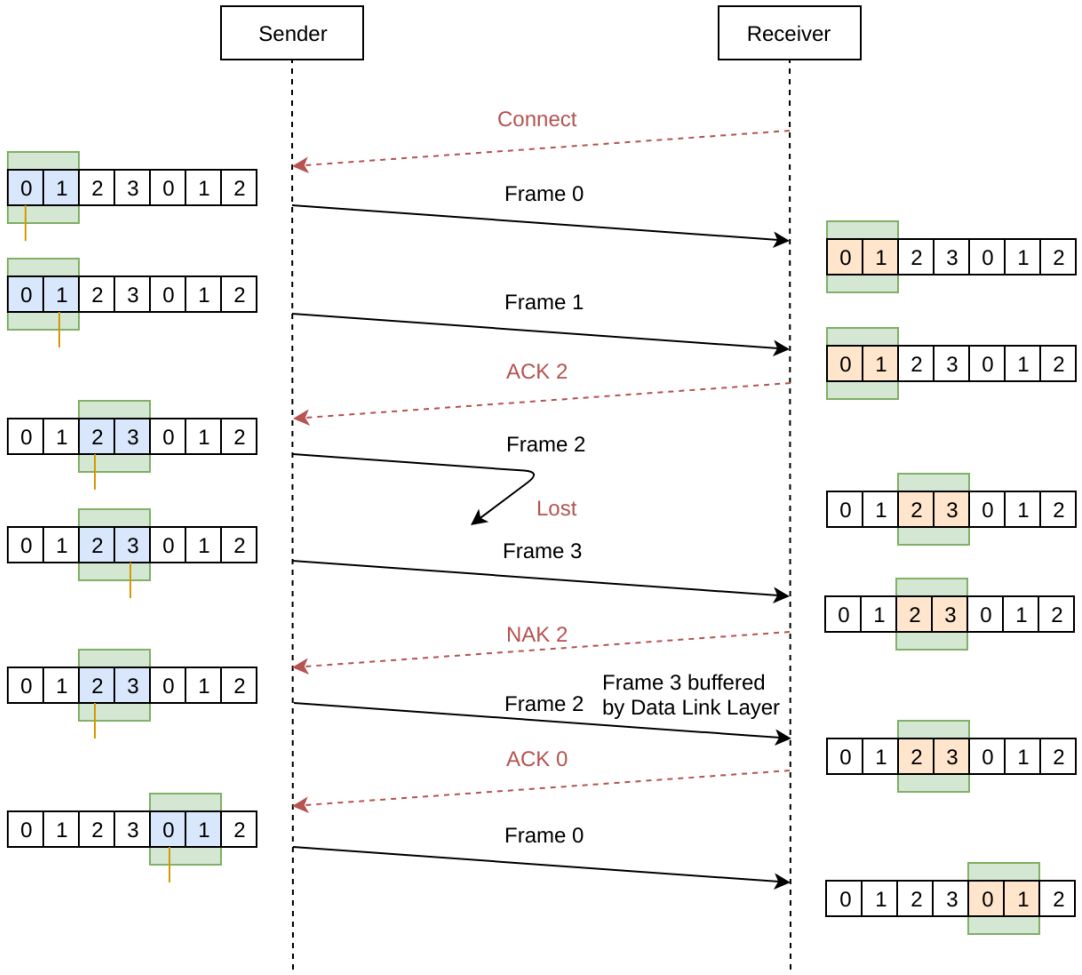



Figure 9: Selective Repeat implementation. The sender starts sending packets after receiving the connect signal from the receiver that communicates that it is ready to receive data. Having a window size greater than 1, the receiver accepts and buffers any out-of-sequence packet, whose sequence numbers are included in the current reception window.

# 4 Conclusions

Having performed several tests, using an Arduino Uno and a PC it is possible to conclude that the protocol is able to detect errors (simulating these randomly) and to deliver the message to the user without errors.
To allow a reliable data transfer an instance of the **ReliableDataTransfer** class must be created that provides four functions to initialize, send, receive and close a reliable connection-oriented transmission:

1. **int init(device, protocol, baudrate);** which passes the device, the protocol and the baudrate as arguments, sets the serial connection. Returns 1 if success, -1 if error.
2. **void send(buffer, buffer size, timeout);** which sends the message in buffer length buffer size. If it does not receive notification of receipt within the timeout time, the sender will return the previously sent packet.
3. **void recv(buffer, buffer size, timeout);** which receives the message of length buffer size in the buffer. The timeout is the time within which to send the ack from receipt of the message. It is important only in the case of selective Repeat.
4. **int close();** that closes the serial connection.

An example of how this library can be used is provided below.

```
#include "ReliableDataTransfer.h"

int main() {

    ReliableDataTransfer rdt;

    // Init
    int is_open = rdt.init("/dev/ttyACM0", "selective repeat", 115200);
    assert(is_open > 0);

    // Sender side
    unsigned char tx_buffer[16];
    for (int i = 0; i < sizeof(tx_buffer); i++)
        tx_buffer[i] = 'a' + i;
    rdt.send(tx_buffer, sizeof(tx_buffer), 1000);

    // Receiver side
    unsigned char rx_buffer[16];
    rdt.recv(rx_buffer, sizeof(rx_buffer), 1000);

    // Close
    int is_close = rdt.close();
    assert(is_close == 0);

    return 0;
}
```