2. Theory

There can be no practice without knowledge coming first, so I will use this chapter to explain the original algorithm, and how I'll use to implement the practical version.

I divided the theoretical algorithm in two parts. This is in accordance with the way the algorithm is build up, and maps nicely on the actual development cycle I went through when implementing the algorithm.

1. We start with the first attempt algorithm. This forms the foundation for the other, more advanced versions. Just like in a house, you want a strong foundation.
2. Next is the packet formation algorithm. This is a relatively small part, but it adds an important feature. To reach the fully advanced sender/receiver algorithm as described in the paper, we add Error Correction Coding.

First attempt algorithm

We start with this first algorithm before moving on to the advanced versions. It is a self-stabilizing, large overhead end-to-end communication algorithm for coping with packet omissions, duplications, and reordering. We have two sides, a sender algorithm and a receiver algorithm. The sender starts by fetching a batch of messages from the application layer. Each data packet is put in message of the form , where *dat* represents the data, or the actual message packet. The sender then sends the packet to the receiver. The two other variables are the Alternating Index, and a label. The label increments until every time a new message is send. The Alternating Index will only increment in modulo 3 when a new message batch is fetched. This means that for every value of the Ai, labels map on to it.

The receiver gets the message delivered, reads the values and puts it in a position in a *packet_set* array. The positions in the array is not randomly chosen, but depends on the *label* value. The Receiver then sends an ACK (acknowledgement) message back to the sender. When the sender receives this message, it knows the current message was delivered at the destination and it puts the incoming acknowledgement in the **ACK_set** array. The sender keeps the Alternating Index the same, but increments the label value. During this process where the receiver gets the message, puts it away, and sends an acknowledgement message, the sender keeps transmitting the same message. It only stops when the acknowledgement arrives. This is the principle of the **Alternating Bit Protocol** or ABP[**?**], and it is just slightly different from the well know Stop-and-Wait ARQ protocol[**?**], which waits with sending new messages until an acknowledgement arrives.

Once the receiver has received the full batch of messages and the incoming label reaches , the receiver sets it's **Last Delivered Alternating Index** value to the incoming *Ai*, and delivers the messages in *packet_set* to the application layer on the receiver side.

At this point both sides reset their arrays and the cycle start again with new messages.

Packet formation from messages

To add a form of redundancy to the algorithm, the packet_set() function is added. It takes a batch of messages of length *pl* and size *ml* (per message), regards this batch of messages as a 2D bit-matrix, and tranposes this whole matrix. Instead of a pl x ml size matrix we now have a matrix with n amount of rows, where $n > ml$ because of the added redundancy bits from the Error Correction Coding.