5. Results

In the previous section I explained that I will test the algorithm in two manners. First I'll let the simulations run in different configuratons. The goal is to develop a measure for the `CAPACITY` and `SENDNODES` variables in a way that will optimally benefit the performance of the algorithm.

5.1 Simulations

Using the various scripts I designed to automate the simulations, I'll let it run for an extended period of time. Specifically, I've designed the sender algorithm in a way that it transmits an incrementing counter value as the data in the packets. When the counter hits 10000, meaning that many packets have been sent, I'll stop it. By capturing the output data on the serial port of the virtual receiver mote in Cooja, I'll have a very easy and yet accurate way of detecting when errors occur. I simply use a small shell-script to run through the output, and if comes across a sudden 'jump' in values, meaning a value is missing, I'll have detected an error.

For each version of the algorithm I'll execute this 10 times. That way I'll have a collection of (50 x 10000) = 50000 packets that have crossed the communication channel. This may seem like a big number, but remember that the simulator doesn't run in real-time but in 'computer' time. Sending 10000 packets only take around 45 seconds in the single-hop setup, because it's so simple. For the advanced set-ups the simulator will run a lot slower and I won't send that many packets.

5.1.1 Single-hop setup

In the previous chapter we discussed how we'll increase the the `CAPACITY` variable every new simulation cycle. The variable `SENDNODES` stays at a value of 10, meaning 20 nodess in total in the network. This is a good value to represent a realistic situation.

5.1.1 Multi-hop setup

Now in the multi-hop setup, we'll probably see a decrease of performance in all simulations, since it requires a lot more resources from the nodes.

Before we continue to the LibReplay simulations and test the validity of the algorithm, we first need to establish a optimal working value for the `CAPACITY` variable in the algorithm.

Looking at the graphs it's quite easy to see where they start decreasing in performance. The first half of the graphs is on the lower side because of the

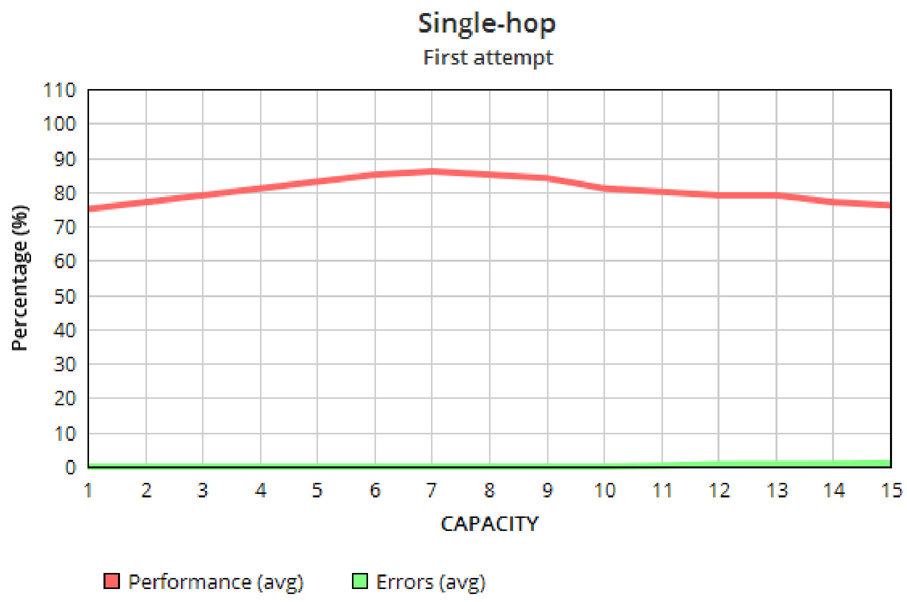

Figure 1: First thing you notice is that the performance (speed) drops when the capacity is going to it's maximum value. A very good thing we notice is that there are almost no erorrs over the whole simulation. The lower performance for low values of `CAPACITY` is understandable. Every time the node successfully sends a packet it has to go fetch a new message. This creates a lot of overhead.

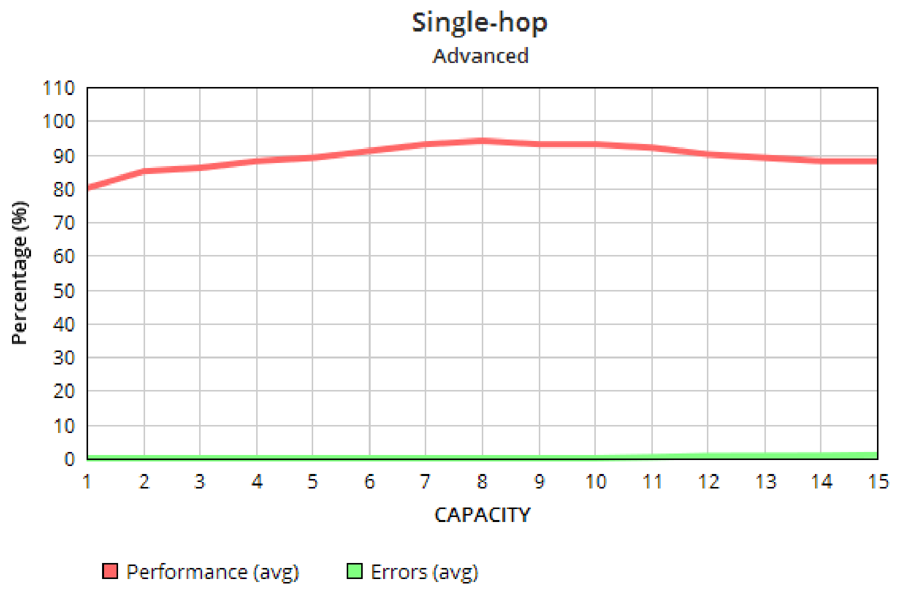

Figure 2: Again we notice almost no errors over the whole simulation. It's a pleasant surprise to see that the performance of the advanced algorithm is much better than the first attempt.

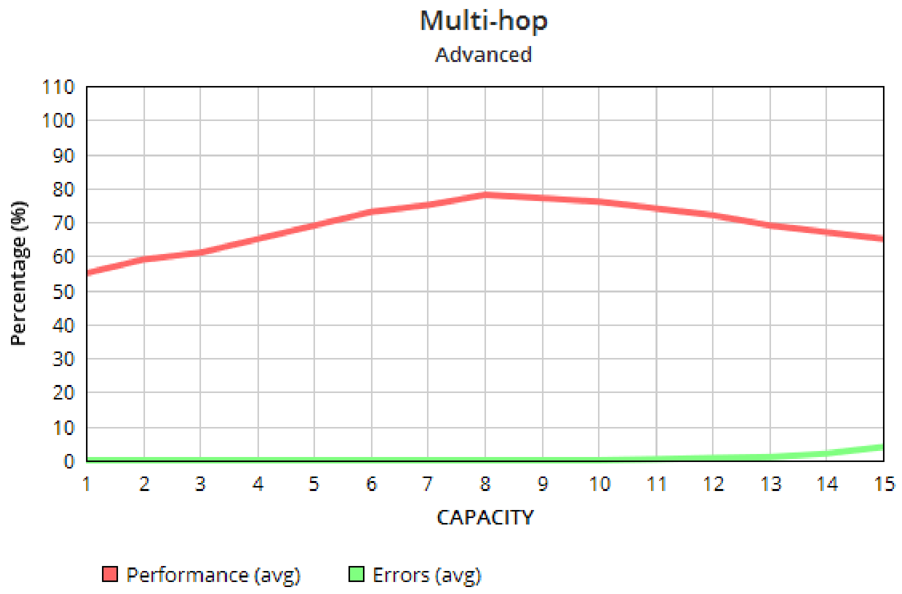

Figure 3: The performance of the algorithm drops significantly in comparison to the single-hop situation. This is perfectly normal though and lies within the expectations. The error percentage is what we truly care about and this stays nice and low. Even at the end where it rises a bit, it stays within the acceptable range.

high overhead of fetching a new value from the application layer every time you send one. With the single-hop, first attempt version simulations the optimal performing value is at `CAPACITY = 7`. For the single-hop, advanced version simulations it's at the 8 mark. For the multi-hop version the peak lies at 8 as well. The average of these three values is 7.6, but we make it 8 since we can't make arrays with decimals behind the comma.

5.2 LibReplay

For the LibReplay simulations, we will insert the logging components, save the execution state, and replay the execution state. When replaying the execution state we will test the three elements we needed, to prove self-stabilization in this algorithm. To test the algorithms we will use either of these elements repeatedly in random order.

1. Insertion of packets
2. Deletion of packets
3. Reordering of packets

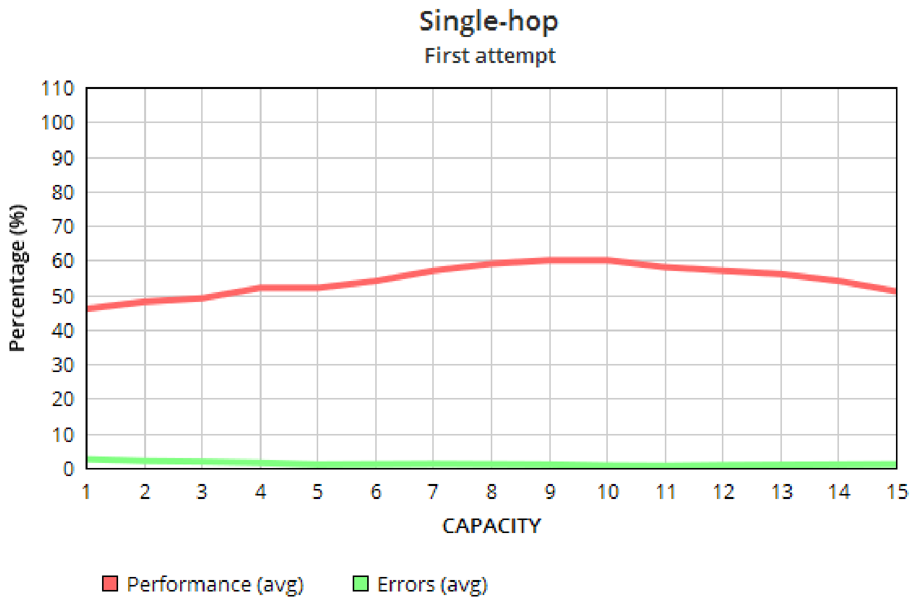

Figure 4: It is clear to see the first attempt algorithm is able to keep the errors at a minimum, but it pays a price on the performance scale. The large overhead of this algorithm prevents it from working more effectively.

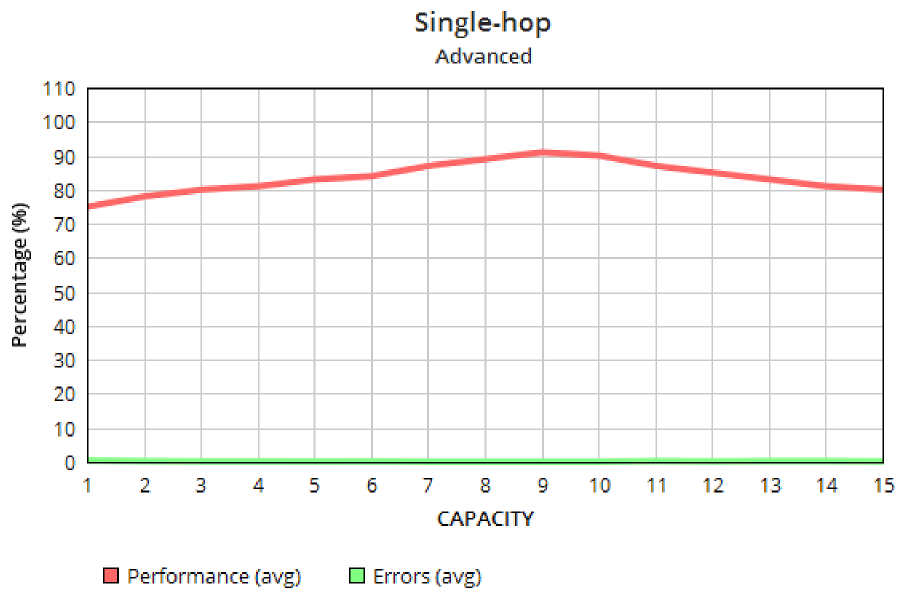

Figure 5: Our algorithm with Error Correction Coding does very well, as expected. It doesn't suffer from the overhead that the first algorithm has, and can recover from errors in packets.

5.1.1 Single-hop setup

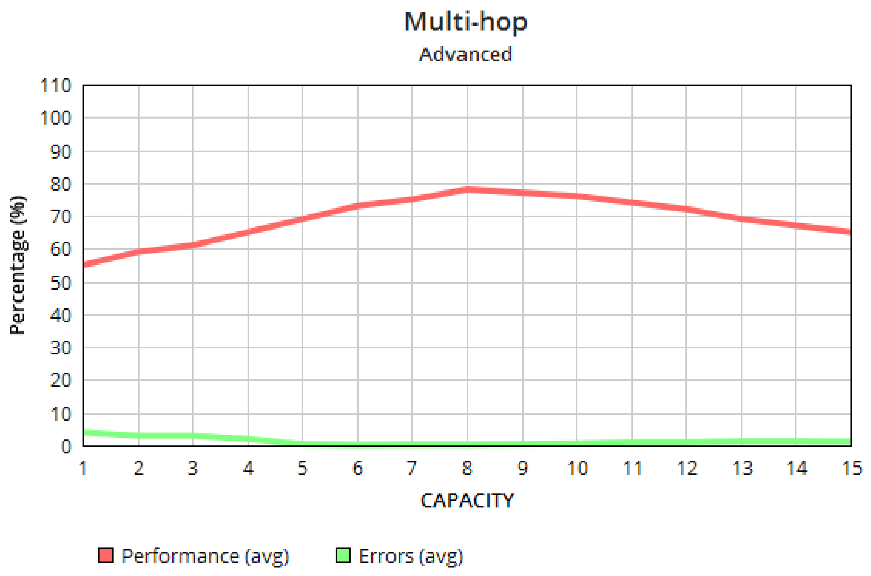

Figure 6: Although this algorithm clearly suffers from the overhead created by the routing algorithm, it is nice to see that the errors are kept at a minimum and that the performance isn't as low as we had expected.

5.1.1 Multi-hop setup

5.3 Interpretation: analysis of results

After getting the data together and looking at the graphs, it is clear to see the algorithms works very well under pressure from errors. Even the first attempt algorithm performs well by keeping the amount of errors low, but it is clear to see that it suffers from the overhead. Not surprisingly, the single-hop advanced algorithm performs the best out of the three. This is because on one hand, it has the Error Correction Codes that help keep errors from reaching the application layer at the receiver side, and on the other hand it doesn't have to work with the larger overhead of the first attempt algorithm as well as not having to worry about routing packets along the network.