Distributed Systems Project

EE 597 Fall 2019

Arshine Kingsley -- 3940957181
Santiago Zapata -- 3189193726

Problem description

The project provides us with a scenario where UAVs have the mission of selecting targets within a common range. Our objective was to write an algorithm such that these targets were unique among the nodes.

The initial scenario has 8 UAVs numbered 1 through 9 (skipping 5). Similarly, we have 8 targets numbered 11 through 19 skipping 15. Once the targets are in range, they need to be tracked by a UAV. The node will keep tracking the target as long as it still in range.

Base code provided

We started by analyzing the provided UDP code. It has two threads running on parallel, with one of these threads overseeing the communication over UDP and the other one tracking the targets.

The UDP thread is used to keep the data about other nodes updated, as all the nodes are continuously providing information about the target they are tracking. The intention of running them in parallel is that the node has this information up to date and at hand when it is required.

The tracking thread is a little more elaborated. It starts by checking if it is already tracking a target. If it is still in range, then it keeps tracking it. If the UAV is not tracking anything and something comes in range, then it checks if it is already being tracked and if not, it selects it to track. This is done within a cycle, so the node will constantly overwrite the target that wants to track until the cycle is done, and it comes out of it with the information of the last one. This means that if a node saw, for example, targets 11, 12 and 13 and they are not previously taken, it will get out of this cycle with the information to track 13. With this information, the node takes the target. A message is sent over UDP to inform the other UAVs about the target that is being tracked.
This is running constantly, so the UAVs are constantly checking for the targets. One important detail is that the code implements a thread lock, because both threads are accessing the same data structures (the information of all nodes) to read and write, so we need to make sure that only one of the threads can make changes at a time.

Our proposed algorithm

The nodes start once again by checking if they already have a target and if it is still in range. If they don't have one and start seeing potential targets, they first evaluate if these targets are already being tracked by other nodes. Once a UAV has a list of the available targets, it evaluates the distance to each one and selects the closest one (not the last one in the cycle like in the original code). When the node has defined a target, it marks itself in selecting mode and sends a message to the other nodes letting them know:

- That it is selecting
- The target it wants to track
- The distance to that target.

We then release the thread lock, go to sleep for 50 milliseconds, and grab the thread lock again. This will allow the UDP thread to check for any other nodes that may be also selecting. Once we update the state about the rest of the UAVs, the node checks if someone else wants the same target and if so, it calculates priority. The closer node will have a higher priority and if they happen to have the same distance, then the node with the lower number will get the tiebreaker. If a node loses the bid for a competed target, then it resets its information and goes back to find potential targets.

In summary, we are improving the original code in two ways. By adding the selecting mode and evaluating priorities, we are able to avoid duplicate targets. Also, the process to select the objective is now based on distance, rather than just the last number on the list.

Handling losses in the environment

The main challenge once we solved that all the nodes get a different target is how to handle losses in the network. We have two types of messages going through the network. If a node has a target and it keeps tracking it, it sends a message to the rest of the UAVs stating the tracked id (an update message). The other message (a control message) is sent when a node is in selecting mode and it is trying to get a target. The update message is being continuously sent, so it is not a big issue if one of them gets lost (up to certain degree as we will see on the results).

Losses can have a great impact on our algorithm. If a control message gets lost, the rest of the nodes are not aware and the algorithm will assume no one else is in this mode, so we have the risk of taking duplicate targets. Our solution was to reduce the loss probability by sending the message multiple times. This generates an overhead in the network, but improves our probability of getting the message.

If we analyze the probabilities, we can see that if we sent the message three times, we can handle up to 30% loss with less than 1% probability that the three messages will get lost.

The following table shows the probability of losing all messages depending on how many repeated messages are sent and the loss probability:

	Number of messages being sent						
	2	**3**	**4**	**5**	**6**	**7**	**8**
10% loss	1.00%	0.10%	0.01%	-	-	-	-
20% loss	4.00%	0.80%	0.16%	-	-	-	-
30% loss	9.00%	2.70%	0.81%	-	-	-	-
50% loss	25.00%	12.50%	6.25%	3.13%	1.56%	-	-
80% loss	64.00%	51.20%	40.96%	32.77%	26.21%	20.97%	16.78%

As we can see, if we keep sending three messages with 50% loss, we would have over 10% loss probability, so the solution starts to fail. This is solved by sending more messages, being five enough in this case to put us below 5%, but this comes with the burden of generating a bigger overhead.

Results

In order to evaluate our algorithm, we ran several simulations changing the loss probability. Tests with no loss, 10%, 20% and 30% loss were successful on selecting unique targets by sending three messages. When we compare this with the results of the original program, we can see that we experienced an increase on the latency and the rate. This is explained because now we have the control messages flowing through the network.

We started having issues when we moved the loss probability to 50%. With a higher chance of losing packets, we noticed that the number of times node-target pairings had all unique targets were only 4, with 2 having duplicate targets. Once we adjusted the number of messages to five, we were able to run clean again, going the 6 rounds without duplicate targets. Once again, this comes at the cost of a higher rate as we are generating more overhead in the network.

Once we raised the loss probability to 80%, we were not able to make it work properly. In order to get it working, we would need to send about seventeen messages, and we would also need to start checking for the update messages.

It is also worth mentioning that the original program was never able to have a clean round, as it had duplicates on every round with all the different losses. However, with the lower loss probabilities, more nodes were covered. It also showed a better latency every time, this is explained by the fact that it is a simpler program.

	Original program		Proposed algorithm		
Loss probability	**Average Latency (secs)**	**Unique targets on all rounds**	**Average Latency (secs)**	**Unique targets on all rounds**	**Messages sent**
No loss	3.498	No	4.461	Yes	3
10% loss	3.565	No	4.158	Yes	3
20% loss	3.814	No	4.03	Yes	3
30% loss	3.69	No	4.009	Yes	3
50% loss	3.772	No	4.146	No	3
			4.155	Yes	5
80% loss	3.844	No	4.062	No	7

Next, we show the resulting graphs for every test we ran.

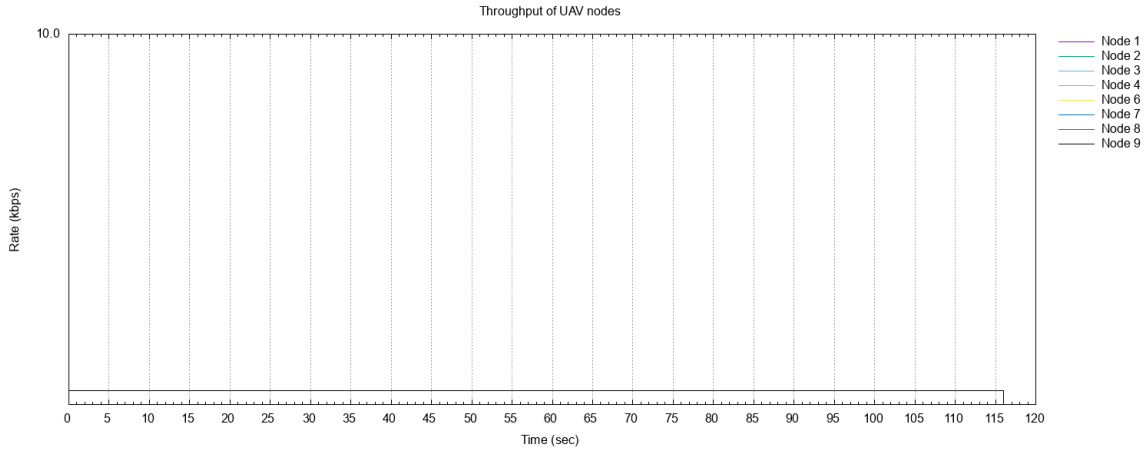

Figure 1. Original program (same result every time)

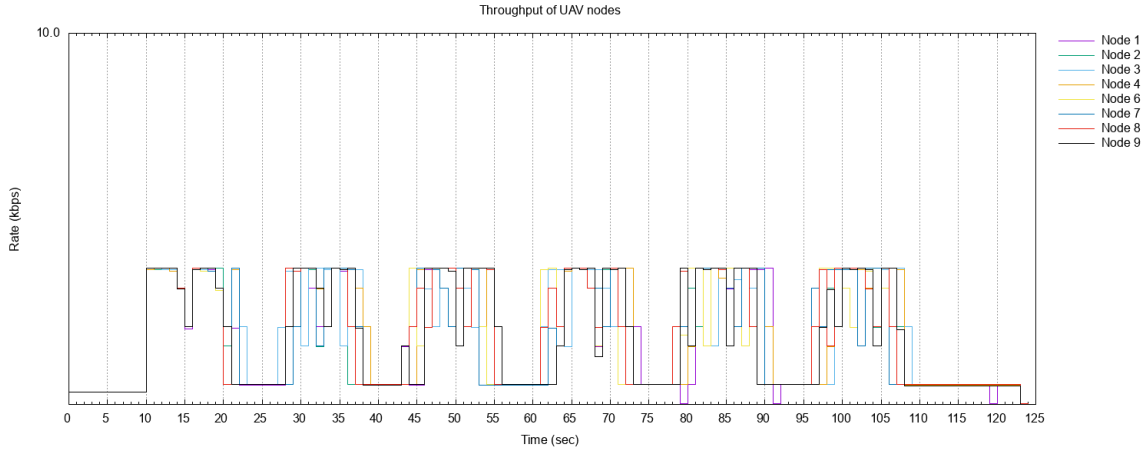

Figure 2. 10% loss

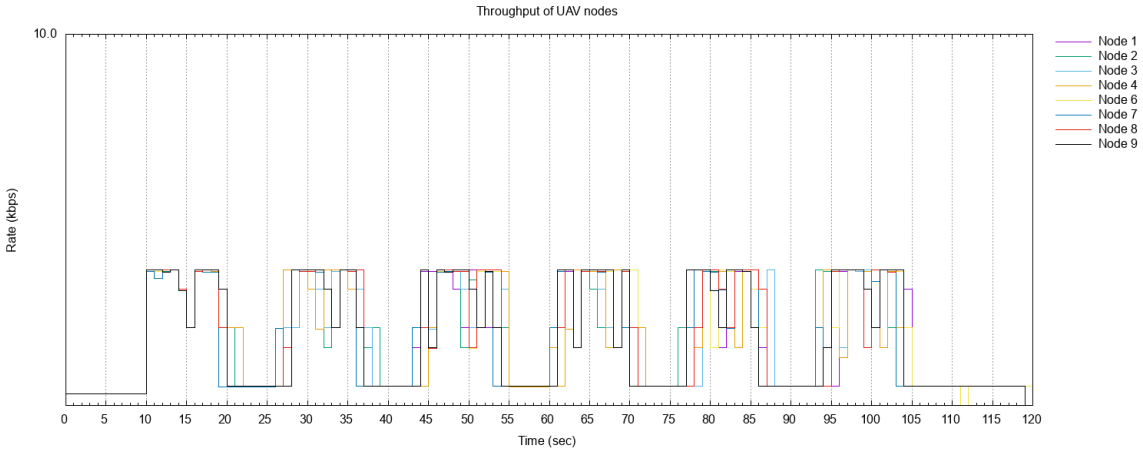

Figure 3. 20% loss

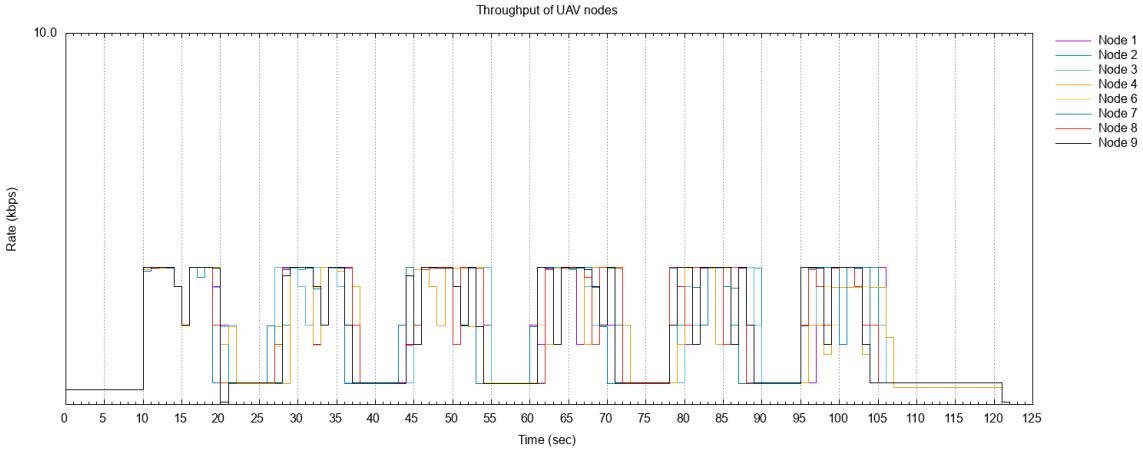

Figure 4. 30% loss

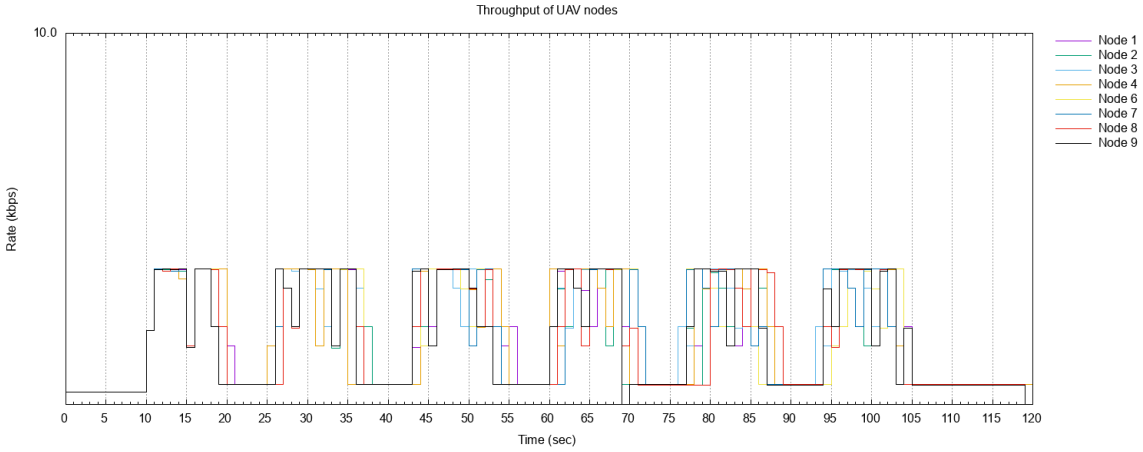

Figure 5. 50% loss (3 messages)

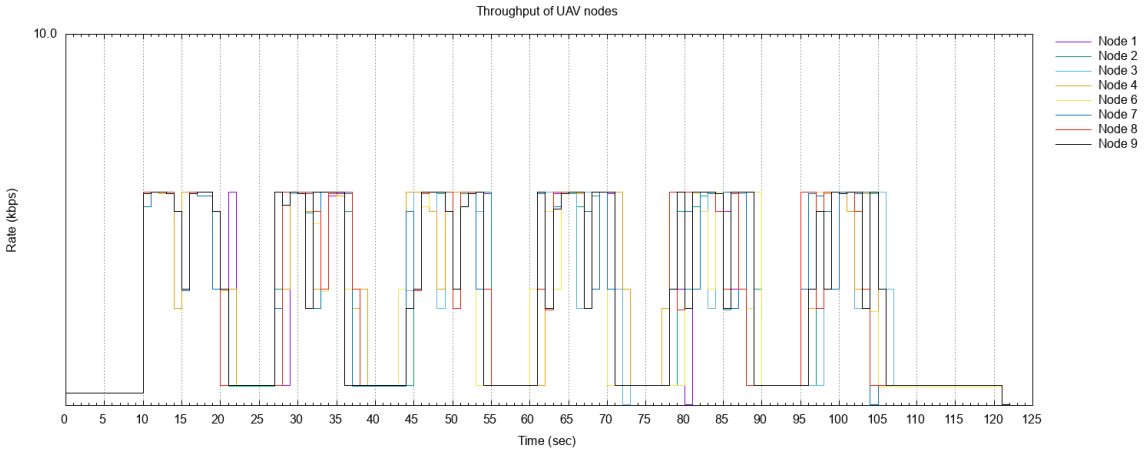

Figure 6. 50% loss (5 messages)

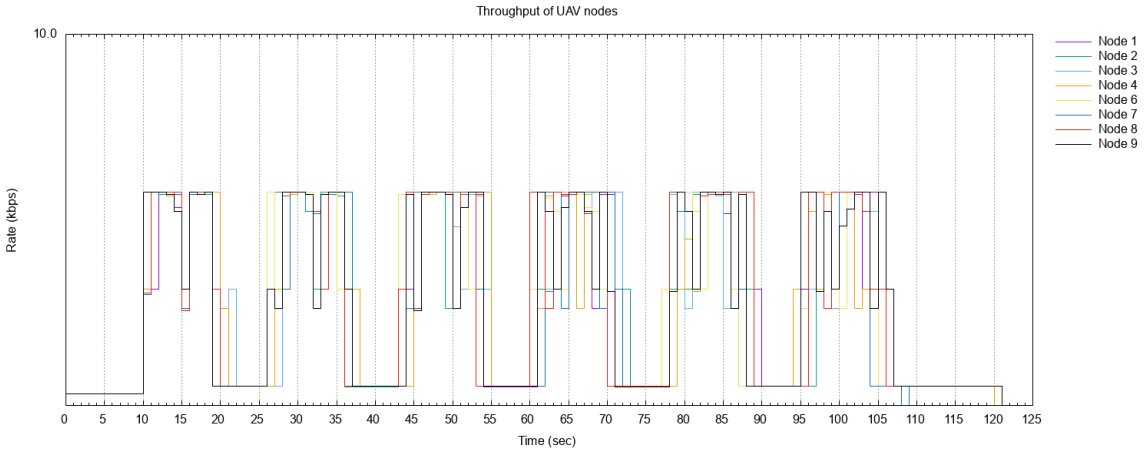

Figure 7. 80% loss (7 messages)

The following table has the values of the average rate (in kbps) for the nodes through the different simulations. As we can see, it is relatively stable, and we get a higher value when we are generating more overhead packets.

	Node 1	**Node 2**	**Node 3**	**Node 4**	**Node 6**	**Node 7**	**Node 8**	**Node 9**
No Loss	2.2628	2.2342	2.2799	2.2521	2.1844	2.2447	2.2212	2.2249
10% Loss	2.2279	2.2388	2.2307	2.2028	2.2066	2.1528	2.2138	2.1884
20% Loss	2.2091	2.2130	2.2296	2.2367	2.1964	2.1512	2.1828	2.1884
30% Loss	2.2173	2.2144	2.1882	2.2121	2.1715	2.1457	2.2167	2.2085
50% (3 packets)	2.2250	2.2904	2.2656	2.1930	2.1803	2.1732	2.2058	2.2153
50% (5 packets)	3.1315	3.0941	3.0526	3.0679	3.0148	3.0181	3.0759	3.0799

Discussion

A flaw in our reasoning are the repeated messages. Even if we reduce the probability by sending the message several times, we are not guaranteeing that the message will get delivered, so we can't have the certainty that our data is always up to date. A possible solution to this would be to implement TCP, but this can potentially slow down our communications, so an adjustment to the way the protocol handles the back off and the window sizes would be required. If we decided to stay with UDP, we could develop an acknowledgment method at the application layer, like QUIC does.