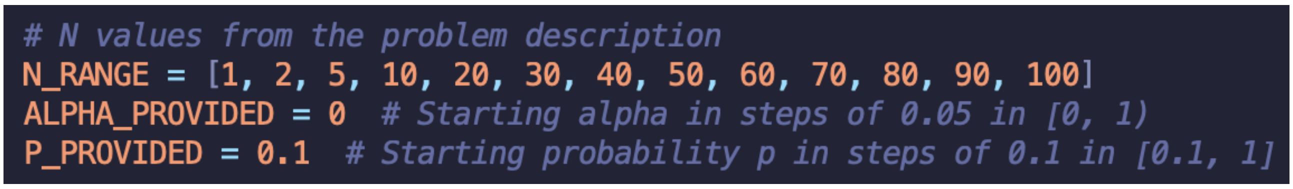
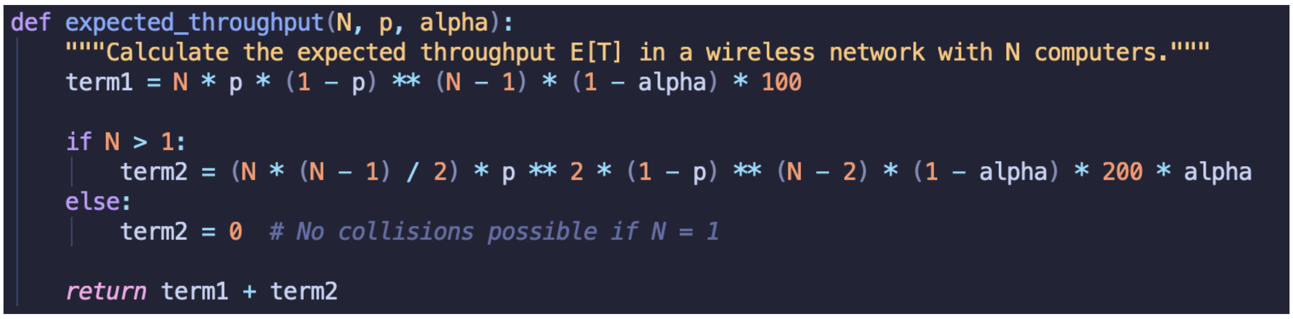
Andrei Ursachi 3351912
Milan Veul 3411389

The first part of the bonus:



```python
# N values from the problem description
N_RANGE = [1, 2, 5, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
ALPHA_PROVIDED = 0  # Starting alpha in steps of 0.05 in [0, 1)
P_PROVIDED = 0.1  # Starting probability p in steps of 0.1 in [0.1, 1]
```

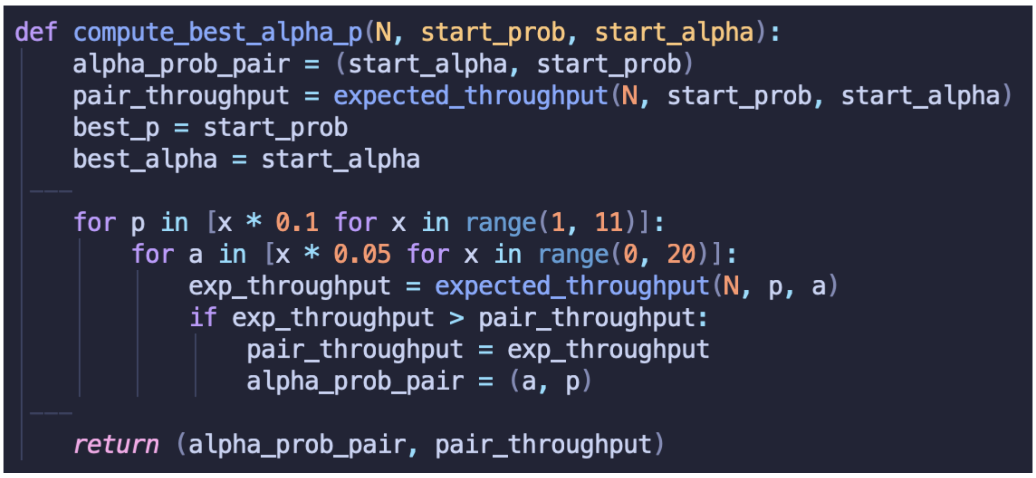
These are the initial conditions for the program:
- N_RANGE – are all the required nodes for test
- ALPHA_PROVIDED – this the first value for alpha – needed for resets
- P_PROVIDED – this is the first value for probability – needed for resets



```python
def expected_throughput(N, p, alpha):
    """Calculate the expected throughput E[T] in a wireless network with N computers."""
    term1 = N * p * (1 - p) ** (N - 1) * (1 - alpha) * 100

    if N > 1:
        term2 = (N * (N - 1) / 2) * p ** 2 * (1 - p) ** (N - 2) * (1 - alpha) * 200 * alpha
    else:
        term2 = 0  # No collisions possible if N = 1

    return term1 + term2
```

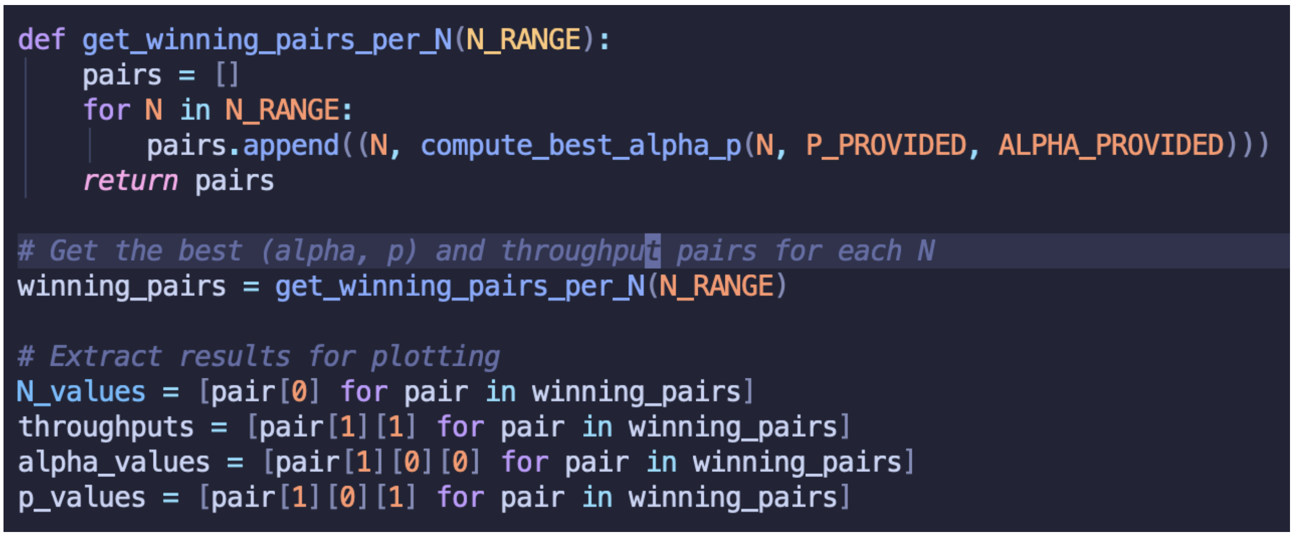
This function computes the expected throughput for a given N, probability and alpha (we need it like this because we generate for every N we want to compute all the possibilities with alpha and p and get the one with highest throughput).
This one uses Bernoulli Trial but modified to take in consideration multiple computers but also the case in which the data is corrupted.
If there are only 1 computer the probability is 100 for 100 packets (also we take alpha in consideration). But in case there are multiple Nodes we know that the total connection in a network with more nodes is N * (N – 1) / 2 but we only want 2 computers to send so we compute the probability for 2 computers sending at the same time so we have 200 because we have 200 packets now as we will have 2 computers sending.

We compute the best alpha,



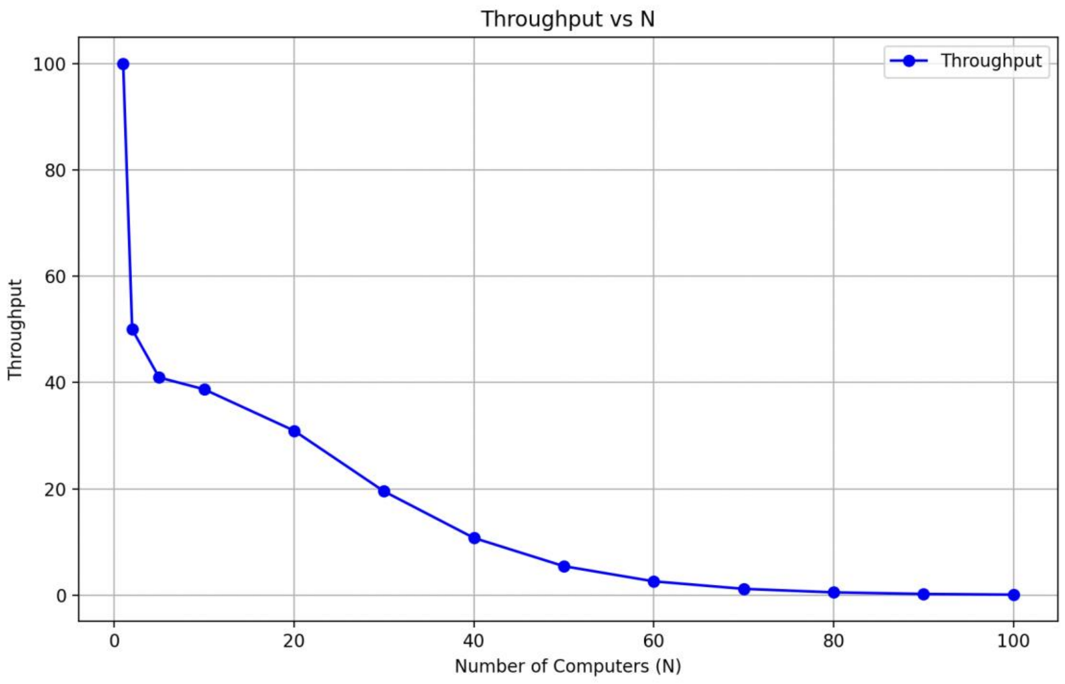
```python
def compute_best_alpha_p(N, start_prob, start_alpha):
    alpha_prob_pair = (start_alpha, start_prob)
    pair_throughput = expected_throughput(N, start_prob, start_alpha)
    best_p = start_prob
    best_alpha = start_alpha

    for p in [x * 0.1 for x in range(1, 11)]:
        for a in [x * 0.05 for x in range(0, 20)]:
            exp_throughput = expected_throughput(N, p, a)
            if exp_throughput > pair_throughput:
                pair_throughput = exp_throughput
                alpha_prob_pair = (a, p)

    return (alpha_prob_pair, pair_throughput)
```

probability for sending and throughput needed based on these and the N. The returned value is a tuple ((alpha, probability), pair_througput). We return the on with highest throughput.
The values for alpha are incremented in steps of 0.05 and probability in steps of 0.1.

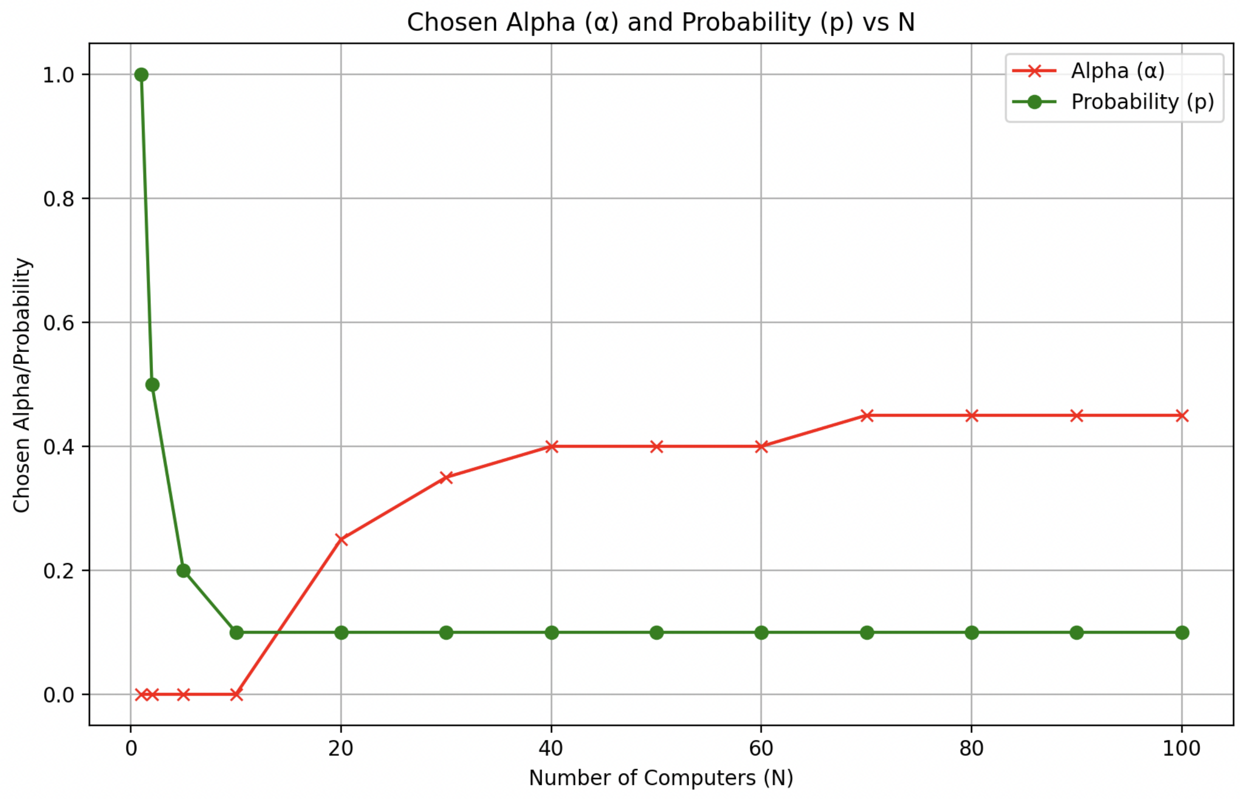


```python
def get_winning_pairs_per_N(N_RANGE):
    pairs = []
    for N in N_RANGE:
        pairs.append((N, compute_best_alpha_p(N, P_PROVIDED, ALPHA_PROVIDED)))
    return pairs

# Get the best (alpha, p) and throughput pairs for each N
winning_pairs = get_winning_pairs_per_N(N_RANGE)

# Extract results for plotting
N_values = [pair[0] for pair in winning_pairs]
throughputs = [pair[1][1] for pair in winning_pairs]
alpha_values = [pair[1][0][0] for pair in winning_pairs]
p_values = [pair[1][0][1] for pair in winning_pairs]
```

The function get_winning_pairs_per_N will return a list of all the pairs of (N, ((alpha, provided), throughput)) for each N in N_RANGE.
After that we create lists of values for N_values, throughputs, alpha_values and probability_values in order to plot them with matplotlib.

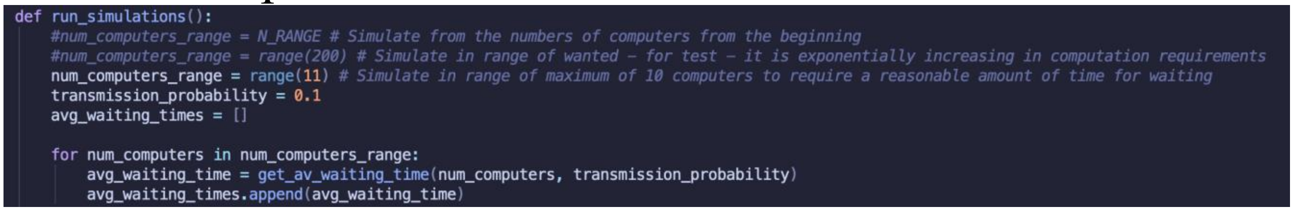


We can see that if we have 1 computer is 100% throughput but is slashed in half for just 1 node added. It is kept at around 35 - 45% for 2 – 10, but after is decreases at a pretty constant rate.



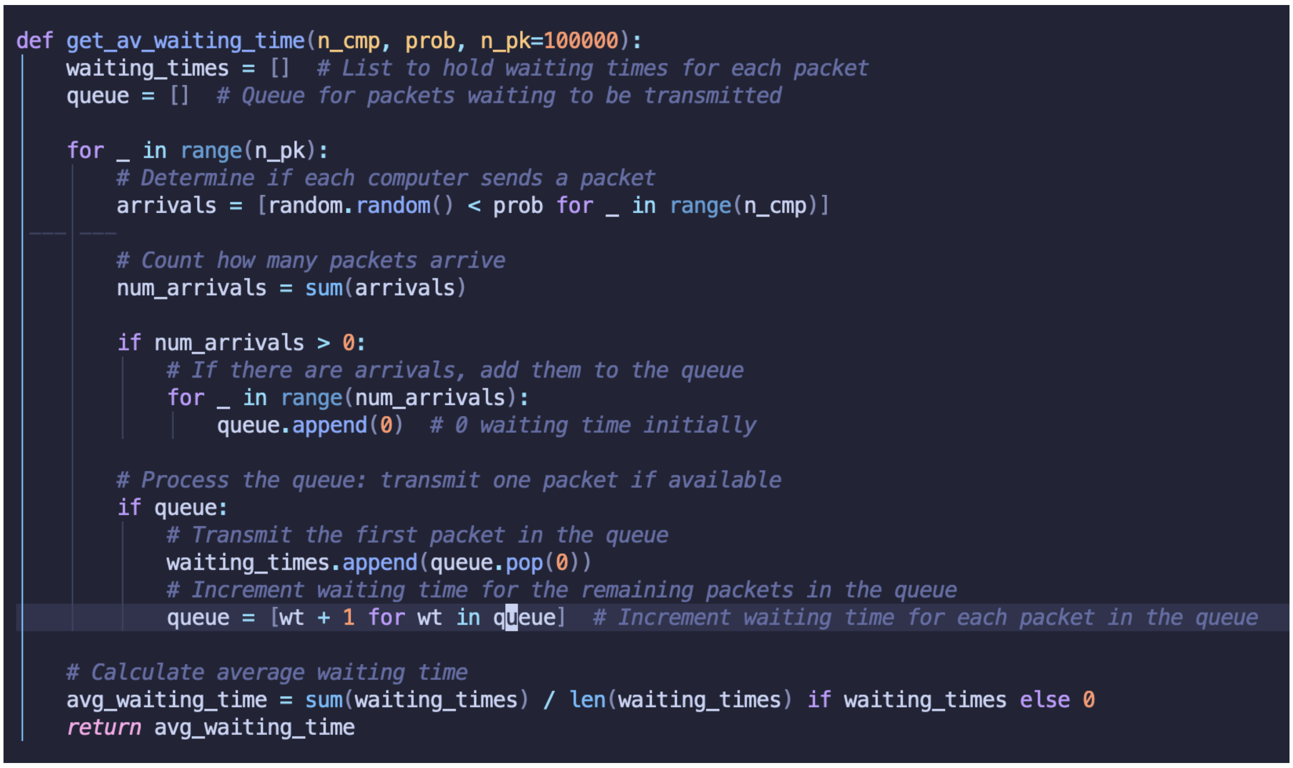
We can see that the probability follows a trend similar to the throughput which makes sense because with more nodes we want less and less probability for them to “fire” a packet as we want as few collisions as possible. But on the other half we can see that the redundancy efficiency settles at 0.45 which means that the more redundancy won’t help us get better throughput.

The second part of the bonus:



```python
def run_simulations():
    #num_computers_range = N_RANGE # Simulate from the numbers of computers from the beginning
    #num_computers_range = range(200) # Simulate in range of wanted - for test - it is exponentially increasing in computation requirements
    num_computers_range = range(11) # Simulate in range of maximum of 10 computers to require a reasonable amount of time for waiting
    transmission_probability = 0.1
    avg_waiting_times = []

    for num_computers in num_computers_range:
        avg_waiting_time = get_av_waiting_time(num_computers, transmission_probability)
        avg_waiting_times.append(avg_waiting_time)
```

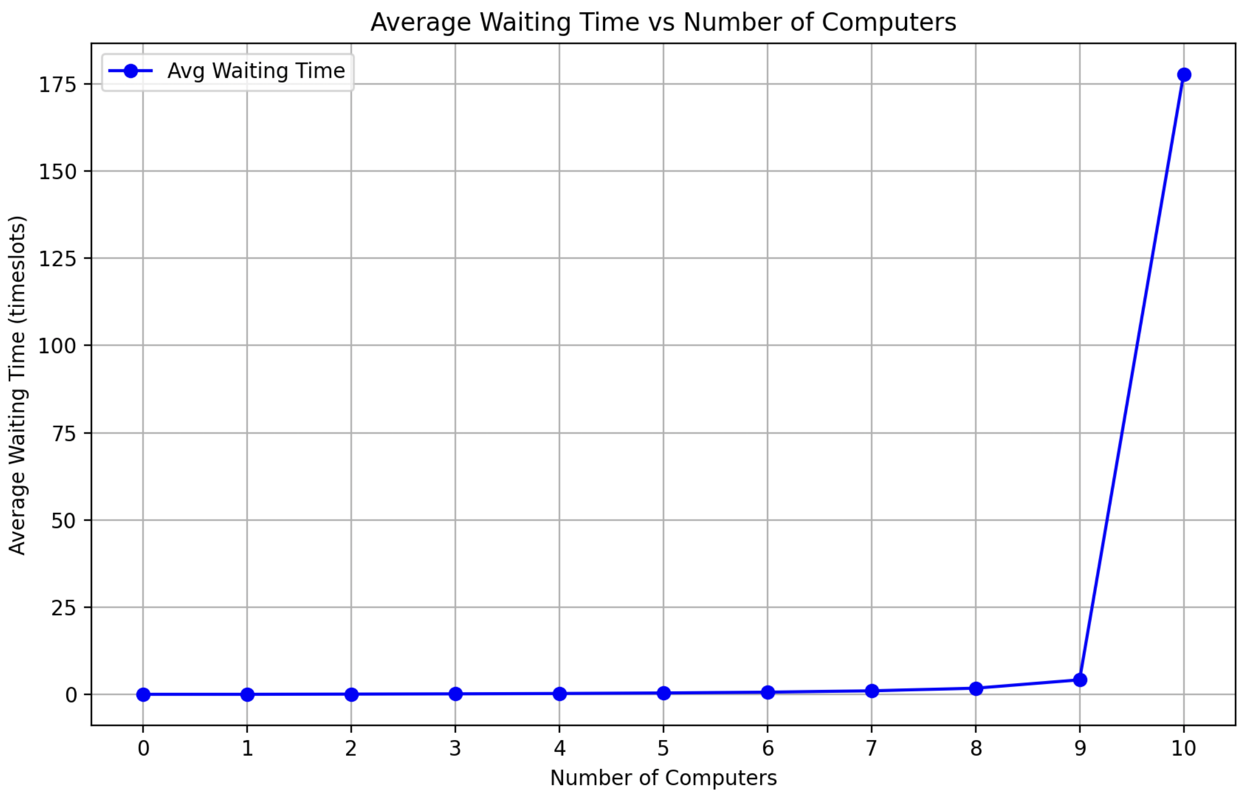
This function runs the simulations for a number of computers from 0 – 10 because the algorithm becomes very inefficient for high numbers.
The way it works it that for each number of computers in the network we compute the average wait time and we then plot it.



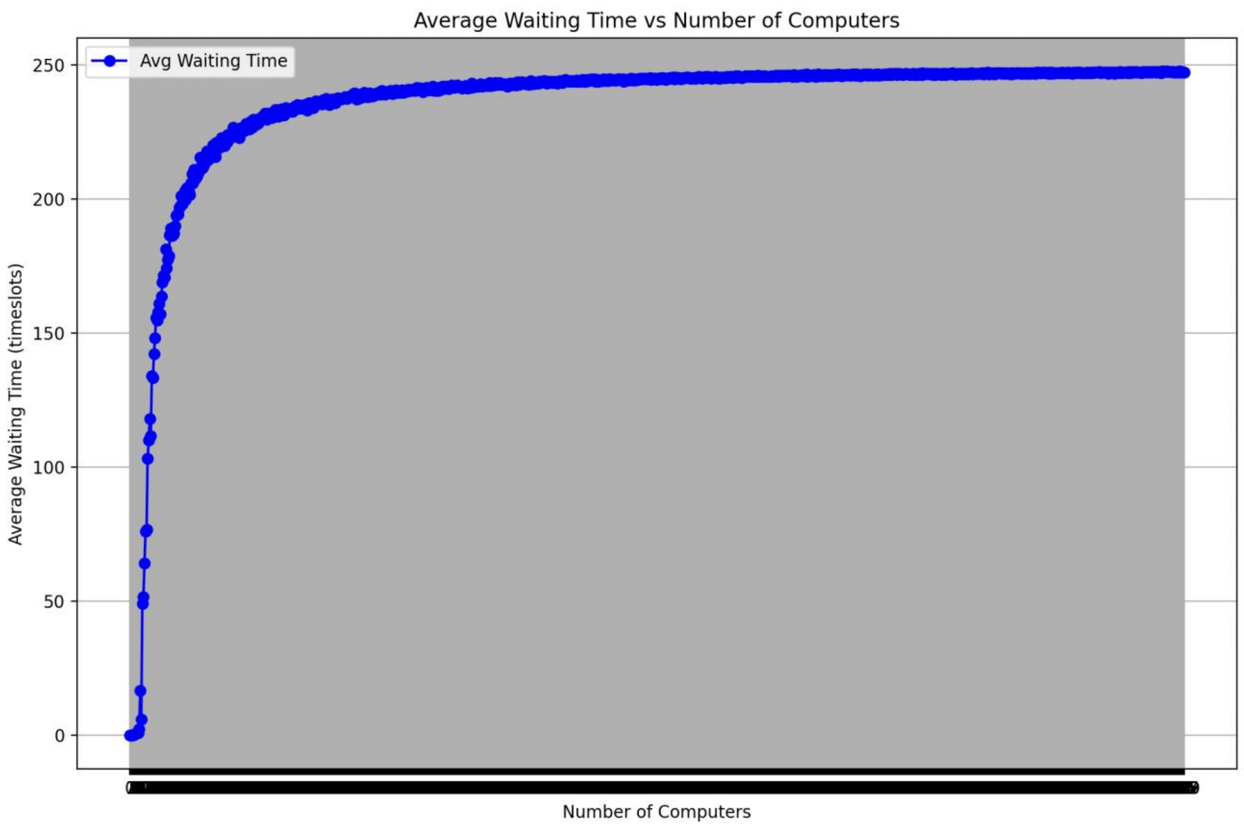
```python
def get_av_waiting_time(n_cmp, prob, n_pk=100000):
    waiting_times = []  # List to hold waiting times for each packet
    queue = []  # Queue for packets waiting to be transmitted

    for _ in range(n_pk):
        # Determine if each computer sends a packet
        arrivals = [random.random() < prob for _ in range(n_cmp)]

        # Count how many packets arrive
        num_arrivals = sum(arrivals)

        if num_arrivals > 0:
            # If there are arrivals, add them to the queue
            for _ in range(num_arrivals):
                queue.append(0)  # 0 waiting time initially

        # Process the queue: transmit one packet if available
        if queue:
            # Transmit the first packet in the queue
            waiting_times.append(queue.pop(0))
            # Increment waiting time for the remaining packets in the queue
            queue = [wt + 1 for wt in queue]  # Increment waiting time for each packet in the queue

    # Calculate average waiting time
    avg_waiting_time = sum(waiting_times) / len(waiting_times) if waiting_times else 0
    return avg_waiting_time
```

This function works by creating a list which contains the waiting time for all the packets in the network. For each computer we have a probability of 0.1 to send a packet so we create a list which contains all the computes that “fire” a packet. We assign a random probability from 0 to 1 and if it is smaller than p it means that the packet was sent from a computer. Then we append them at the end of the queue the packets with the time they were waiting in the queue. After that, if we have items in the queue we pop the first packet in the queue and we add it in the waiting_times to signal that we sent it. And we increment all the other packets as they are waiting another time.

At the end we compute the average time for the current N and we send it.



This graph shows that the average waiting time skyrockets from 10 computes onward as with more computers we will have more packets at the same time. (This is for when we have 100000 packets.) If we adjust the number of sent packets in the script to check for less packets (500 for example) we can notice the trend better for more nodes. We can see in the graph bellow (that checks for max N = 999) that the graph skyrockets pretty early but it has kind of a logarithmic curve and it looks like it reaches a maximum waiting time at around 250. In this case.


Average Waiting Time vs Number of Computers
Avg Waiting Time
Average Waiting Time (timeslots)
Number of Computers