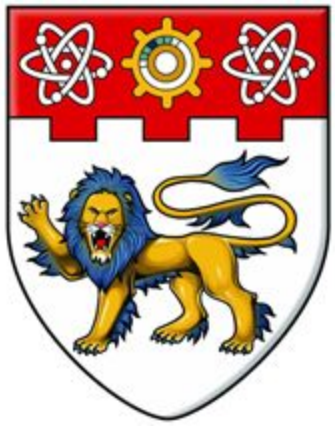


NANYANG TECHNOLOGICAL UNIVERSITY

Assignment 1 :
Packet Switching in Radio Channels

CZ3006/CE3005 – COMPUTER NETWORKS
(Semester 2, AY 2019/2020)

WILSON THURMAN TENG
U1820540H

Assignment 1 : Packet Switching in Radio Channels

Description

In this assignment, we are required to reproduce the Figure 9 of the Kleinrock-Tobagi using suitable software. This should be done using different values for a, the normalised propagation delay and p, the probability that the transmission medium is idle. The protocols to reproduce are as follows:

- **A.** Slotted ALOHA (Dependent on G)
- **B.** Pure ALOHA (Dependent on G)
- **C.** Non-Persistent CSMA (Dependent on G & a)
- **D.** 1-Persistent CSMA (Dependent on G & a)
- **E.** P-Persistent CSMA (Dependent on G, a & p)

Organisation

This Python implementation and presentation of assignment 1 was separated into 2 files:

I. *"Assignment 1_protocol_data_generator.ipynb"*
II. *"Assignment 1_interactive_graphs.ipynb"* OR *"Assignment 1_interactive_graphs.html"*

I. helps to generate the data points for the different protocols in a comma separated values file. **II.** utilizes the comma separated values file which are stored in the "Generated Data" folder to generate interactive graphs.

Program Flow

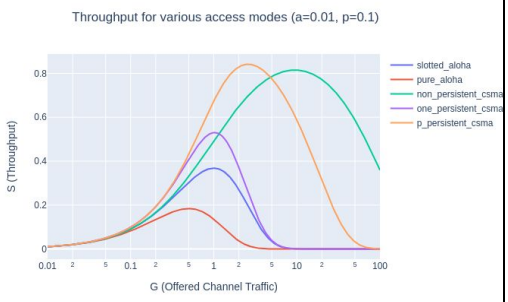
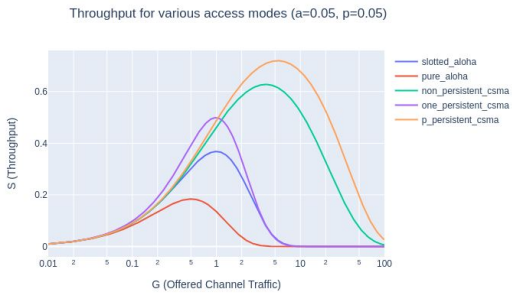
To generate the data, an array of points in log scale are generated to represent G, the packets per transmission time. This is accomplished by first generating an array of float values from -2 to 2, *x*. Each element in the array is then calculated by 10******x* to get a range from 0.01 to 100.

For each protocol, the G value is then used to calculate the corresponding S values, which represents the throughput of the protocol. A higher throughput would imply that more data is transmitted between two transmission points in a unit time.

Protocol Implementation

The formulas for protocols **A**, **B**, **C** & **D** are relatively straight-forward to implement. However, protocol **E** is slightly challenging in nature due to the need to calculate limits to infinity for certain variables. Although this is a generally bad programming practice, that issue was overcome by the usage of *try* and *except* statements (as shown in Fig. 1 in Appendix). Those help to stop the calculation when Python encounters an overflow error. This will allow Python to calculate the most accurate result as the sum to infinity would then stop only when necessary.

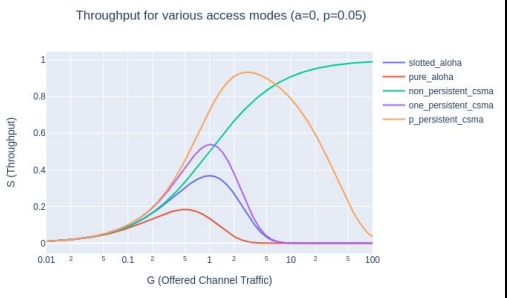
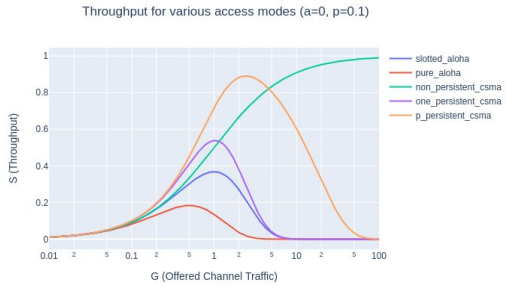
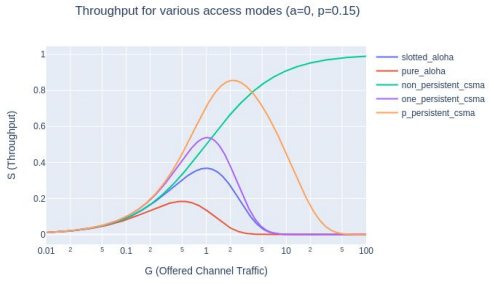
Observations

<table>
<tr>
<td> 
</td>
<td> 
</td>
<td rowspan="2">We can observe that the 2 aloha protocol graphs are the same as these 2 protocols are unaffected by the value of a.

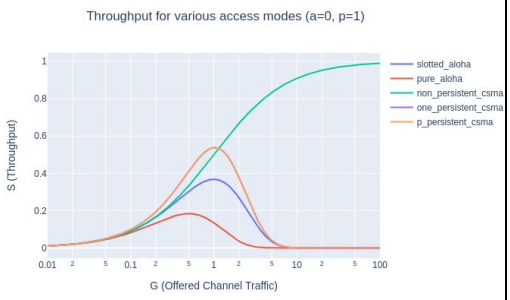
On the other hand, as a increases, the peak of the other graphs decrease. Intuitively, this is true as the throughput should decrease if the normalised propagation delay increases.

Another observation is that the peak of these graphs move towards the left when a increases, implying that the protocol becomes less efficient for higher traffic loads.</td>
</tr>
<tr>
<td>a=0.01, p=0.05</td>
<td>a=0.05, p=0.05</td>
</tr>
</table>

Special Cases (a=0)

<table>
<tr>
<td> 
</td>
<td> 
</td>
<td> 
</td>
</tr>
<tr>
<td>a=0, p=0.05</td>
<td>a=0, p=0.10</td>
<td>a=0, p=0.15</td>
</tr>
</table>

We can observe that in the ideal scenario when a=0, non-persistent protocol produces an exponential increase in throughput. We can also observe that as p increases, the peak of the p-persistent protocol graph moves towards the left and the maximum value decreases.

<table>
<tr>
<td> 
</td>
<td>There is another special instance when a=0 and p=1. P-persistent protocol is supposed to combine the best of both worlds and obtain the benefits of both 1-persistent as well as non-persistent protocols. However, when a=0 and p=1, we can observe that the p-persistent protocol is similar to the 1-persistent protocol. This concludes that the advantages of p-persistent protocol can only be reaped if 0<p<1.</td>
</tr>
<tr>
<td>a=0, p=1</td>
<td></td>
</tr>
</table>

Appendix

```
# (34) Prob of successful transmission | After removing condition N from (33)
def p_s(G, a, p):
    n = 1
    result = 0
    pi_n0 = pi_n(G, 0, a)
    nxt_term = p_s_n(G, n, a, p) * pi_n(G, n, a)/(1-pi_n0)
    while (nxt_term > ERROR_MARGIN):
        try:
            nxt_term = p_s_n(G, n, a, p) * pi_n(G, n, a)/(1-pi_n0)
            result += nxt_term
            n += 1
        except:
            break
    return result
```

(Fig. 1) An example of the implementation of Probability of Success, Ps