Cloud Computing Assignment 1

Hongpo Chen U2022097E

Background

In the world of computer networks, efficient data transmission is essential to ensure seamless flow of information across networks. A fundamental aspect of this that we have learnt in class is the process of congestion control, a process whereby the receiver controls the sender so that the receiver’s own buffer would not be overflowed by the sender sending too much information too fast. By including a rwnd value in the header, the receiver informs the sender of their available buffer space so that the sender can limit the amount of unacknowledged data to send. Without this process, we would likely have to deal with large amounts of packet loss and long delays when sending and receiving information on networks.

One approach to congestion control is the Additive-Increase Multiplicative-Decrease (AIMD) algorithm, a reliable and efficient feedback control algorithm. The sender increases sending rate to probe for unused bandwidth until loss occurs(additive increase), and it cuts the rate by half when loss is detected(multiplicative decrease).

This report delves into the implementation and design of the AIMD algorithm in the context of TCP congestion control, exploring the mechanisms and parameters that affect its behaviour. Specifically, by simulating the AIMD algorithm, we are able to visualize its behaviour and hence efficiency of TCP Reno, a popular TCP protocol, through various graphs on resource allocation, etc.

By adjusting the alpha and exponent values, as well as the functions(such as power and log) used in their calculations, we can see how the algorithm dynamically adjusts the congestion window sizes based on varying network conditions. With this we can gain insights into its efficacy in mitigating congestion and optimizing throughput.

With this, we can shed light on the practical implications of TCP Reno and the AIMD algorithm in various real- world scenarios such as data center environments with its integrated tiered architectures by investigating how AIMD parameters can be tailored to suit these diverse network structures and requirements. Computer networks are always evolving, and with this experiment we can glean valuable insights into their relevance.

Experiments

The code implemented iterates *ITERATESMAX* (set to 100) times, while checking if the 2 hypothetical TCP connections *x1* and *x2* exceed the custom threshold *C*.

The **first** run has alpha and exponent values as 1 and 0.5 respectively, with the difference being the multiplier for alpha being added to *x1* and *x2* values. *alpha1* is calculated as *alpha\*(x1^exponent 1)* while *alpha2* is calculated as *alpha\*(x2+1)*. The graphs for resource allocation and aimd window size are shown below.

A graph with a line

Description automatically generatedA graph with blue and orange lines

Description automatically generated

For the **second** run, *alpha2* was increased from 1 to 1.2 and *exponent2* was increased from 0.5 to 0.6. This leads to a larger increase and decrease in window size for *x2* as compared to *x1*. The multiplier for the alpha values remained as it was before.

A graph of a graph with lines and numbers

Description automatically generated with medium confidenceA graph with blue and orange lines

Description automatically generated

For the **third** run, *alpha2* and *exponent2* were reset back to 1 and 0.5 respectively. *alpha1* was calculated proportionally to the window size as *alpha\*x1* while *alpha2* was calculated inversely proportional to it instead as *alpha/x2.*

*A graph with blue lines and red lines

Description automatically generated*A graph with blue and orange lines

Description automatically generated

For the **fourth** run, all the parameters were reset to their first run states. However, the threshold *C* was changed from 10 to 3.

A graph with lines and numbers

Description automatically generatedA graph with blue and orange lines

Description automatically generated

For the **fifth** and final run, *C* was reverted to 10, and a linear regression model was implemented for predictive control of the *alpha2* value for *x2*'s window size. *alpha1* was still calculated as *alpha\*(x1^exponent 1)*. A random selection of a 1000 samples were generated to train the model, which was used to predict the future resource utilization, and dynamically adjust alpha based on the predicted deviation from the desired utilization level. This aims to optimize resource allocation for *x2*.

A graph with lines and a red line

Description automatically generated A graph with orange and blue lines

Description automatically generated

Discussion

**Experiment 1** sees *x1* getting more resources allocated to it than *x2*. This is due to the power function of *alpha1* increasing faster than the log function of *alpha2*. **Experiment 2** has a slightly higher *alpha2* value, giving it a higher increase but at the same time the decrease was also greater due to *exponent2* increasing. *x1* and *x2* had very similar window sizes and resource allocation. For **experiment 3**, *x1* increases at a much faster rate due to the calculation of the additive increase being multiplied instead of divided like in the case of *x2*. *x1* has significantly higher resource allocation than x2. **Experiment 4** only changed the threshold, which lead to a similar graph as the one in Experiment 1, except multiplicative decrease happens more often.

**Experiment 5** was the most significant experiment as the additive increase formula implemented was dynamic instead of a static formula or value. It was based on a predicted resource utilization value, and hence should be more efficient and optimal in terms of resource allocation. Even though *x1* in this case retained its additive increase formula calculated with power, *x2* received more resources for the first time.

**Alpha** determines the rate of additive increase in the congestion window size. By adjusting alpha, we can control how aggressively the congestion window expands during periods of low congestion. Higher values of alpha result in faster congestion window growth, leading to potentially higher throughput under the right network conditions. However, excessively high alpha values may cause congestion during periods of network congestion.

**Exponents** determine how the additive increase and multiplicative decrease phases affect the congestion window size. By adjusting the exponents, we can fine-tune the responsiveness of the congestion control algorithm to changes in network conditions. Lower values of the exponents result in more conservative congestion window adjustments, increasing stability and reducing a risk of network downtimes.

The **threshold C** represents the maximum allowable sum of congestion window sizes before congestion control measures are triggered. By varying the threshold C, we can explore how network performance is affected by different congestion avoidance strategies. Lower values of C may lead to more frequent congestion control actions, , which is a conservative strategy that reduces likelihood of congestion but potentially limits throughput. The inverse could be said for higher values of C.

Conclusion

By experimenting with these parameters, we gain insights into the dynamics of the AIMD algorithm and its role in TCP congestion control. Using these findings, network administrators and system designers can optimize their congestion control strategies for various networks and usage patterns.

Appendix

Code: all the different parameters for each experiment are written in comments with their respective Experiment numbers.

import numpy as np

import matplotlib.pyplot as plt

from sklearn.linear\_model import LinearRegression

ITERATESMAX = 100 # You can adjust this value as needed

C = 10 # Adjust C value as needed

exponent1 = 0.5 # Adjust exponent1 value as needed (should be less than 1 for decrease)

exponent2 = 0.5 # Adjust exponent2 value as needed (should be less than 1 for decrease)

ITERATESMAX = 100 # You can adjust this value as needed

C = 10 # Adjust C value as needed

# Experiment 4

# C = 3

alpha = 0.1 # Adjust alpha value as needed

exponent1 = 0.5 # Adjust exponent1 value as needed (beta)

exp= 0.5 # Adjust exponent2 onent2 value as needed (beta)

# Experiment 2

# exponent2 = 0.6

x1 = 1

x2 = 1

alpha1 = 1

alpha2 = 1

# Experiment 2

# alpha2 = 1.2

x1\_values = np.zeros(ITERATESMAX)

x2\_values = np.zeros(ITERATESMAX)

resource\_allocation = np.zeros(ITERATESMAX)

# Generate sample data for training the machine learning model

num\_samples = 1000

resource\_utilization\_history = np.random.rand(num\_samples, 1) # Example: resource utilization history (normalized)

# Predictive horizon for resource utilization

predictive\_horizon = 10

# Create a history matrix with columns representing past resource utilizations and rows representing different time steps

history\_matrix = np.zeros((num\_samples - predictive\_horizon, predictive\_horizon))

for i in range(predictive\_horizon):

history\_matrix[:, i] = resource\_utilization\_history[i:num\_samples - predictive\_horizon + i].flatten()

# Target values for alpha (for training the predictive model)

target\_alpha = resource\_utilization\_history[predictive\_horizon:]

# Train linear regression model

model = LinearRegression()

model.fit(history\_matrix, target\_alpha)

for i in range(ITERATESMAX):

# Use the trained machine learning model to predict future resource utilization

future\_resource\_utilization = np.zeros((1, predictive\_horizon))

future\_resource\_utilization[0] = resource\_utilization\_history[-predictive\_horizon:].flatten()

predicted\_resource\_utilization = model.predict(future\_resource\_utilization)

predicted\_alpha = max(0, min(1, predicted\_resource\_utilization[-1])) # Ensure predicted alpha is in the range [0, 1]

if (x1 + x2 <= C):

# Additive increase phase

print('Additive I')

# Experiment 1

# alpha1 = alpha \* np.power(x1, exponent1)

# alpha2 = alpha \* np.log(x2 + 1)

# Experiment 3

# alpha1 = alpha \* x1

# alpha2 = alpha / x2

# Experiment 5

alpha1 = alpha \* np.power(x1, exponent1)

alpha2 = predicted\_alpha

x1 = x1 + alpha1

x2 = x2 + alpha2

else:

# Simulate network condition (for example, congestion)

print('Multiplicative D')

beta1 = exponent1

beta2 = exponent2

x1 = x1 \* beta1

x2 = x2 \* beta2

# pause(1)

# Store values in arrays

x1\_values[i] = x1

x2\_values[i] = x2

# Display the final values

print("Final x1:", x1)

print("Final x2:", x2)

# Plotting the Resource Allocation graph

plt.figure(figsize=(10, 5))

plt.plot(x1\_values, x2\_values, label='Resource Allocation')

plt.plot(np.linspace(0, np.max(x1\_values), 100), np.linspace(0, np.max(x2\_values), 100), linestyle='--', color='r', label='Fairness Line')

plt.title('Resource Allocation: x1 vs x2')

plt.xlabel('x1 Allocation')

plt.ylabel('x2 Allocation')

plt.grid(True)

plt.legend()

plt.show()

# Plotting the AIMD Window Size graph

plt.figure(figsize=(10, 5))

plt.plot(range(1, ITERATESMAX + 1), x1\_values, label='AIMD Window Size for x1')

plt.plot(range(1, ITERATESMAX + 1), x2\_values, label='AIMD Window Size for x2')

plt.title('AIMD Window Size over Iterations')

plt.xlabel('Iterations')

plt.ylabel('Window Size')

plt.grid(True)

plt.legend()

plt.show()