SC4052 Assignment 1 Report

Name: Nigel Chen Chin Hao

Student ID: N2300146E

Introduction

The Transmission Control Protocol (TCP) is a cornerstone of modern network communication, ensuring reliable data transfer across the internet. A key mechanism within TCP that aids in managing congestion and maintaining network stability is the Additive Increase Multiplicative Decrease (AIMD) algorithm. The main objective of this report is to investigate the impact of different AIMD parameter configurations, namely alpha (α) and beta (β).

To anchor the investigation in practical outcomes, the report includes a section dedicated to numerical examples and experiments. These experiments are designed to illustrate how different numbers of TCP users or flows, sharing a single bottleneck, influence the dynamics of TCP.

Experiments

To delve deeper into the effects of various Additive Increase Multiplicative Decrease (AIMD) mechanisms on datacenter performance, a series of simulations was designed and executed. These simulations, crafted in Python, serve to illuminate the intricacies of AIMD behaviour under different configurations. The accompanying appendix provides access to the Python code used for these simulations.

The outcomes of these simulations are presented in the form of graphs, which were carefully constructed to illustrate the convergence rate to fairness and the operational efficiency of each tested AIMD mechanism. These graphical illustrations not only enhance the accessibility of the data but also provide unequivocal visual evidence of the varying impacts that different configurations have on network performance. This methodical approach is intended to provide a comprehensive and intelligible analysis of the critical role AIMD mechanisms play in optimising data centre operations, thereby contributing valuable insights into the field of network management.

Linear AIMD

$$cwnd_{new} = \begin{cases} cwnd_{curr} + \alpha & : cwnd_{curr} \leq cwnd_{max} \Rightarrow \text{AI} \\ cwnd_{curr} \cdot \beta & : cwnd_{curr} > cwnd_{max} \Rightarrow \text{MD} \end{cases}$$

$$\text{where } \alpha > 0 \text{ and } 0 \leq \beta < 1$$

Simulation Statistics

Figure #	# of flows	Alpha (α)	Beta (β)	Approx. # of iterations to fairness	# of windows sent after 20k iterations
1	2	1	0.5	14,000	150,013,640
2	2	1000	0.5	19	169,992,753
3	2	0.1	0.5	115,000	157,252,450
4	2	1	0.2	9,000	120,000,742
5	2	1	0.8	17,000	178,566,420
6	2	1000	0.2	13	121,081,638
7	2	1	0	2,000	100,033,812
8	10	1	0.5	2,900	150,101,255
9	100	1	0.5	290	150,079,650
10	10	1	0.2	1,600	120,124,370
11	10	1	0.8	3,800	179,970,344

Table 1

Observations

In our investigation into network protocols, we explored how the parameter β influences the rate at which fairness is achieved within the context of Transmission Control Protocol (TCP). We found that a lower β value accelerates the attainment of fairness among data flows, but this benefit is accompanied by increased variability in the congestion window (CWND), potentially reducing TCP's overall efficiency.

Through an analysis of Figure 4 (β=0.2) and Figure 5 (β=0.8) presented in the appendix, it becomes clear that a lower β value leads to faster convergence towards fairness—about 47% quicker over 8000 iterations. However, this rapid convergence is offset by greater fluctuations in CWND sizes, as evidenced by Figure 4's CWND varying significantly more than that of Figure 5. Specifically, Figure 4 resulted in 32% fewer windows sent compared to Figure 5 after 20,000 iterations.

Table 1 introduces another critical factor, α, showing that higher values of α significantly hasten the convergence to fairness, often in fewer than 20 iterations. This effect is attributed to the additive increase (AI) component, which boosts efficiency, and the multiplicative decrease (MD) component, which enhances fairness. For instance, comparing Figure 1 (α=1) and Figure 2 (α=1000), the CWND in Figure 2 fluctuates far more frequently than in Figure 1, raising concerns about its suitability in data centre environments, where stability is often preferred.

Additionally, the study reveals an inverse relationship between the number of data flows and the iterations required to achieve fairness. For example, with 10 flows, approximately 2900 iterations are needed for convergence, whereas 100 flows can achieve fairness in roughly 290 iterations, indicating a tenfold increase in speed.

These findings underscore the necessity of carefully adjusting the parameters α and β to strike an optimal balance between fairness and efficiency in network protocols. Understanding and manipulating these variables can significantly impact the performance and reliability of data transmission, particularly in environments like data centre where optimal resource allocation is crucial.

Linear AIMD with Slow Start

$$cwnd_{new} = \begin{cases} cwnd_{curr} + {SS_\alpha}^n & : cwnd_{curr} \leq cwnd_{max} \cdot SS_{threshold} \Rightarrow \text{SS} \\ cwnd_{curr} + \alpha & : cwnd_{max} \cdot SS_{threshold} < cwnd_{curr} \leq cwnd_{max} \Rightarrow \text{AI} \\ cwnd_{curr} \cdot \beta & : cwnd_{curr} > cwnd_{max} \Rightarrow \text{MD} \end{cases}$$

$$\text{where } \alpha > 0 \text{ , } 0 \leq \beta < 1 \text{ , } 0 \leq SS_{threshold} \leq 1 \text{ , } SS_\alpha > 1$$
$$\text{and } n \text{ is the } \# \text{ of successive slow start}$$

Statistics

Figure #	# of flows	Alpha (α)	Beta (β)	SS Threshold	Approx. # of iterations to fairness	# of windows sent after 20k iterations
12	10	1	0.2	0.3	1,500	131,691,338
13	10	1	0.2	0.5	600	167,452,842
14	10	1	0.2	0.8	180	86,488,041
15	10	1	0	0.5	<100	148,879,600
16	10	1	0	0.6	<10	40,080,000
17	100	1	0	0.6	<10	147,119,500

Table 2

Observations

The Slow Start (SS) mechanism is designed to enhance the performance of network communication by rapidly increasing the size of the congestion window (CWND) to an optimal level and then maintaining that level to ensure efficiency. This method not only accelerates the growth of CWND but also leads to a quicker adjustment towards fair bandwidth allocation among users.

In the study, illustrated by Figure 10 (using Linear AIMD alone) and Figure 13 (combining Linear AIMD with SS), it is evident that implementing SS, particularly with a threshold value set at 0.5, results in a threefold faster convergence to fair bandwidth distribution compared to Linear AIMD without SS. Furthermore, Figure 13 demonstrates a notable increase in the number of windows transmitted after 20,000 iterations—about 40% more than that observed in Figure 10.

Adding SS to the Linear AIMD framework introduces additional variables for consideration, such as the SS threshold and SS_alpha, requiring careful calibration to avoid negatively impacting network efficiency. This complexity is further illustrated in the analysis of Figures 14 and 16, where improper parameter settings can lead to decreased efficiency. The number of data flows, a variable element in network environments, also plays a crucial role in the efficiency of SS. For instance, despite having identical settings, a scenario with 10 flows (Figure 16) versus 100 flows (Figure 17) shows a dramatic increase in efficiency in the latter, underscoring the sensitivity of SS to flow volume.

Given the variable nature of data flows within datacenter, the findings suggest that while AIMD enhanced with SS can offer significant benefits in certain contexts, it may not universally be the best approach due to its complexity and the unpredictability of network conditions.

Simple Adaptive AIMD

$$cwnd_{new} = \begin{cases} cwnd_{curr} + \alpha(\frac{cwnd_{max}}{cwnd_{curr}}) & : cwnd_{curr} \leq cwnd_{max} \Rightarrow \text{AI} \\ cwnd_{curr} \cdot (\beta - \delta(\frac{cwnd_{max}}{cwnd_{curr}})) & : cwnd_{curr} > cwnd_{max} \Rightarrow \text{MD} \end{cases}$$

$$\text{where } \alpha > 0 \text{ , } 0 \leq \beta < 1 \text{ and } 0 < \delta < \beta$$

Statistics

Figure #	# of flows	Alpha (α)	Beta (β)	Delta (*δ*)	Approx. # of iterations to fairness	# of windows sent after 20k iterations
18	10	1	0.5	0.1	1,600	148,458,803
19	100	1	0.5	0.1	170	149,445,943
20	10	1	0.2	0.1	600	134,229,612
21	10	1	0.8	0.1	3100	171,870,583
22	10	1	0.5	0.01	1900	154,936,206
23	10	1	0.8	0.01	3800	179,900,779

Table 3

Observation

In the context of data centre operations, an algorithm that dynamically adjusts the parameters α and β in response to the current size of the congestion window (CWND) presents a highly effective strategy. This adaptive approach is designed to quickly achieve and then maintain optimal efficiency levels. It adjusts the weight of additive increase (AI) after each multiplicative decrease (MD) event to expedite the return to peak efficiency, subsequently moderating the increase rate to make full use of the large CWND size before it decreases due to another MD.

Distinct from the Slow Start (SS) method previously discussed, this adaptive Additive Increase Multiplicative Decrease (AIMD) framework only applies aggressive MD when the CWND size substantially surpasses its maximum threshold. The level of this aggressive response is modifiable through the δ parameter, offering fine-tuned control over the algorithm's behaviour. Analysis of Figure 18 from our study illustrates that as the system approaches a fair distribution of network resources, the intensity of MD actions decreases, indicating a more stabilised network condition.

This adaptive mechanism's ability to tailor its response based on real-time network conditions makes it exceptionally suitable for datacenter environments, where traffic patterns can be highly variable and unpredictable. By ensuring that efficiency is rapidly achieved and sustained, this approach not only optimises network performance but also enhances the overall reliability and stability of datacenter operations.

Conclusion

The comprehensive examination of Linear, Slow Start, and Adaptive AIMD within TCP underscores the intricate balance between achieving fairness and maximising efficiency in datacenter environments.

The introduction of SS into Linear AIMD illustrates a method to enhance efficiency by allowing exponential CWND growth initially, followed by maintenance of optimal efficiency levels. This approach, while beneficial in certain contexts, introduces additional complexity and sensitivity to flow variability, potentially limiting its applicability in dynamic datacenter environments.

Adaptive AIMD emerges as a promising solution, dynamically adjusting α and β values in response to current network conditions, thus aiming for a swift attainment and maintenance of efficiency without the harsh penalties of significant CWND reductions. This method's flexibility and responsiveness to changing network conditions make it a potentially superior choice for datacenter applications.

In conclusion, our findings emphasise the necessity of fine-tuning AIMD parameters to navigate the trade-offs between fairness and efficiency. While SS offers a method to rapidly achieve optimal conditions, its practicality is constrained by the variability of datacenter traffic. Adaptive AIMD, with its dynamic adjustments and tailored approach, appears to be the most viable strategy for ensuring robust and efficient data transmission in these complex network environments. Future work should focus on refining these adaptive mechanisms, with an emphasis on real-world applicability and the ability to anticipate and react to the ever-evolving demands of datacenter networks.

Appendix

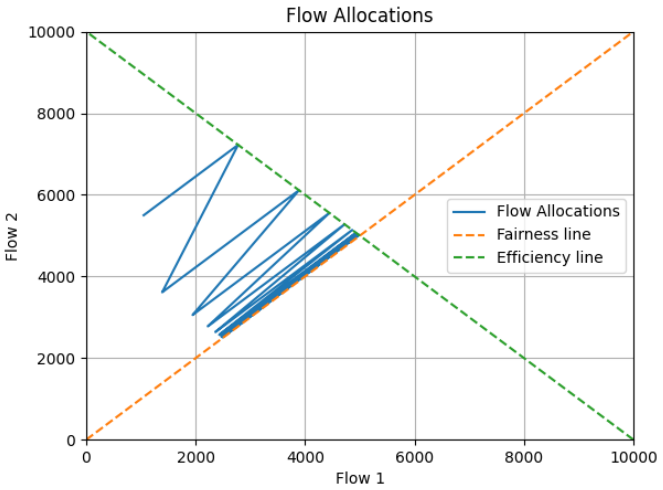

Figure 1.1
NUM_OF_FLOW = 2
ALPHA = 1, BETA = 0.5

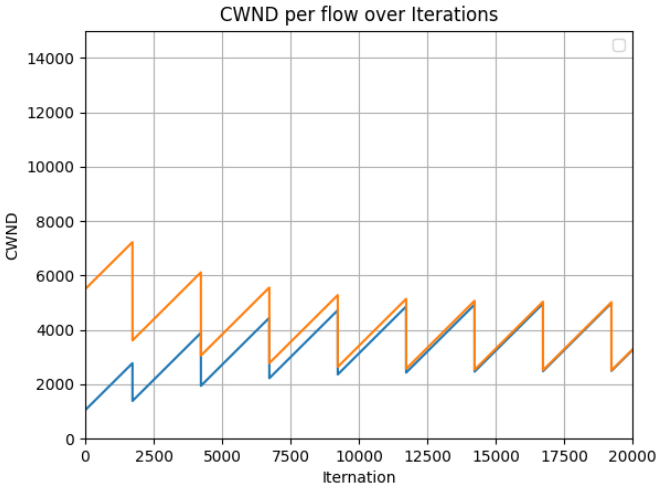

Figure 1.2
NUM_OF_FLOW = 2
ALPHA = 1, BETA = 0.5

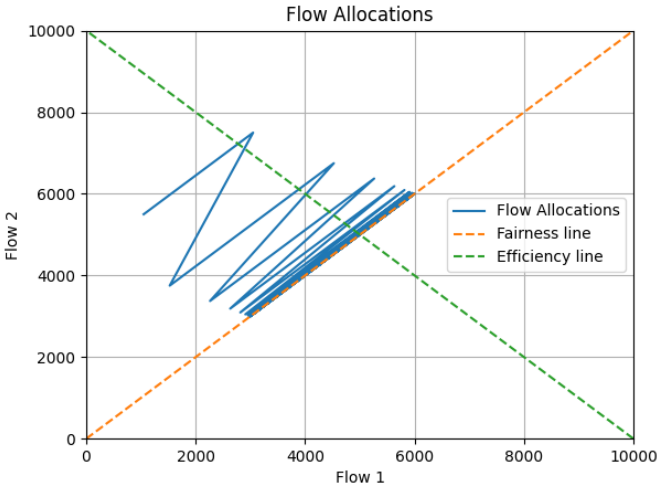

Figure 2.1
NUM_OF_FLOW = 2
ALPHA = 1000, BETA = 0.5

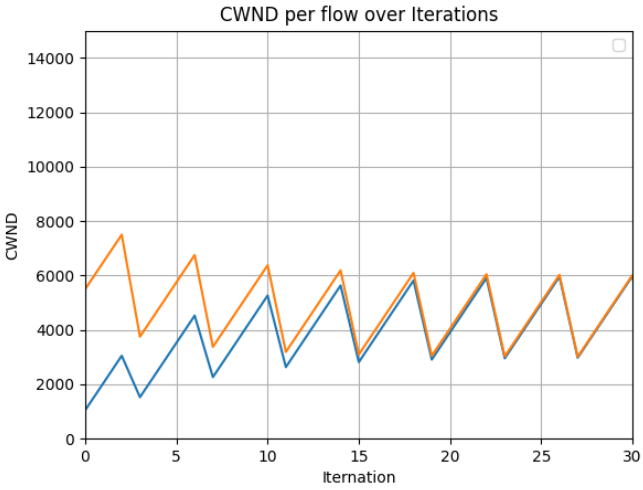

Figure 2.2
NUM_OF_FLOW = 2
ALPHA = 1000, BETA = 0.5

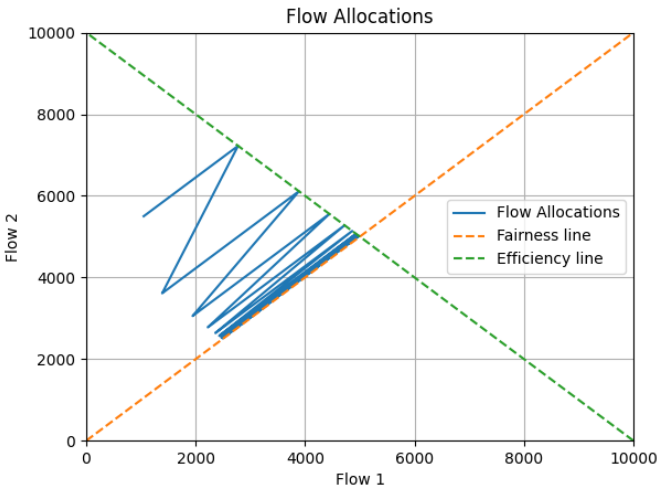

Figure 3.1
NUM_OF_FLOW = 2
ALPHA = 0.1, BETA = 0.5

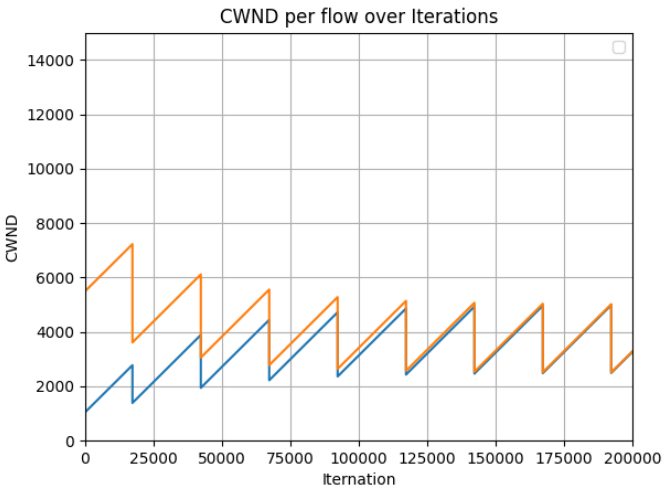

Figure 3.2
NUM_OF_FLOW = 2
ALPHA = 0.1, BETA = 0.5

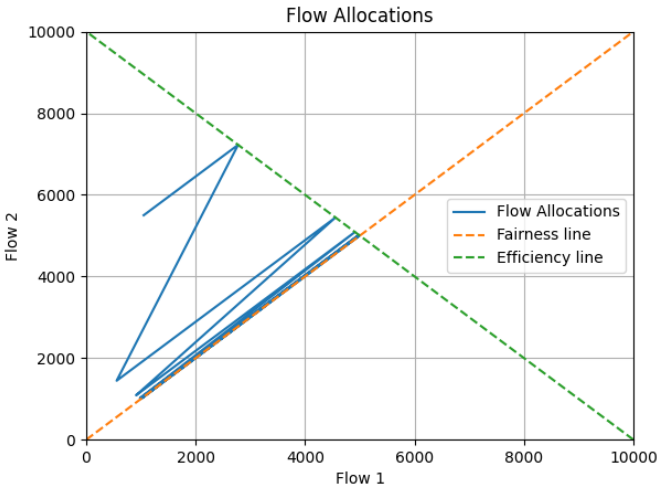

Figure 4.1
NUM_OF_FLOW = 2
ALPHA = 1, BETA = 0.2

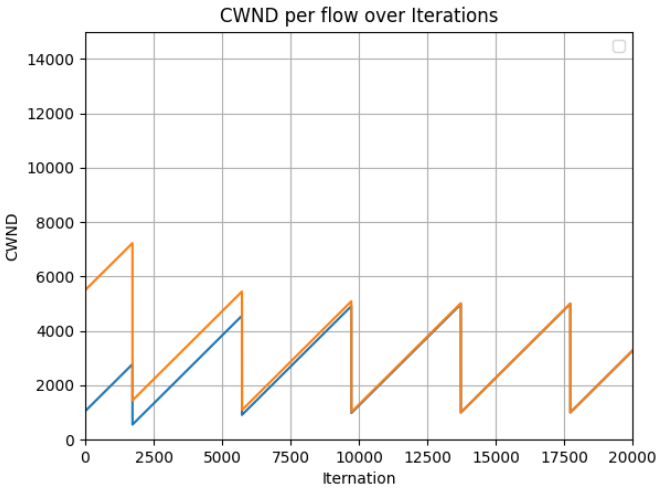

Figure 4.2
NUM_OF_FLOW = 2
ALPHA = 1, BETA = 0.2

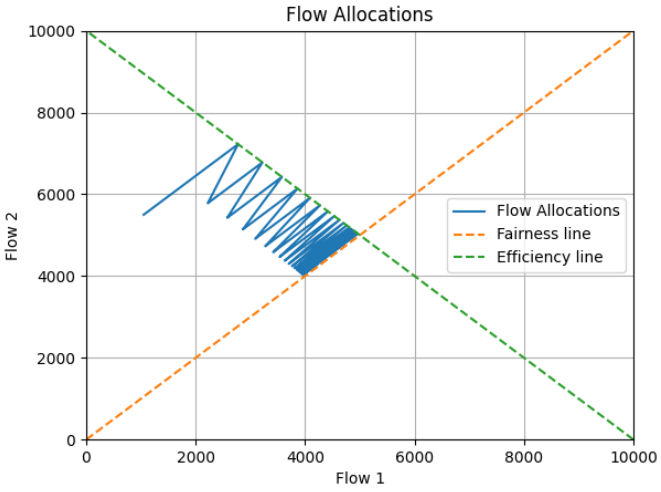

Figure 5.1
NUM_OF_FLOW = 2
ALPHA = 1, BETA = 0.8

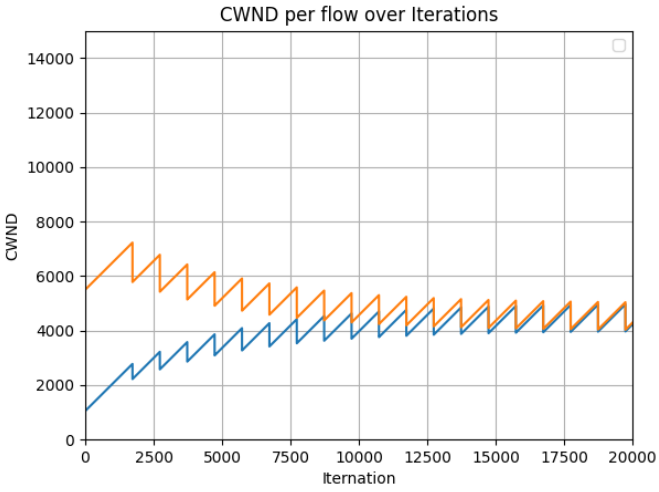

Figure 5.2
NUM_OF_FLOW = 2
ALPHA = 1, BETA = 0.8

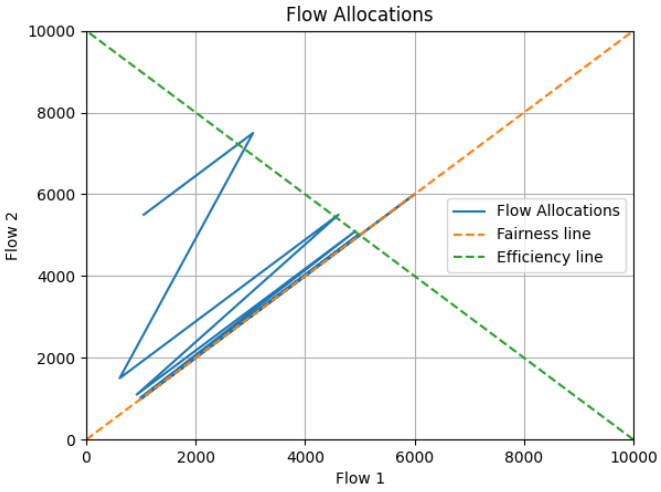

Figure 6.1
NUM_OF_FLOW = 2
ALPHA = 1000, BETA = 0.2

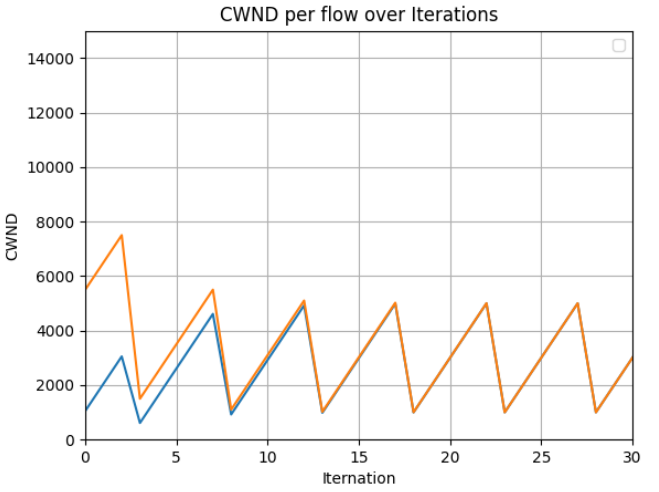

Figure 6.2
NUM_OF_FLOW = 2
ALPHA = 1000, BETA = 0.2

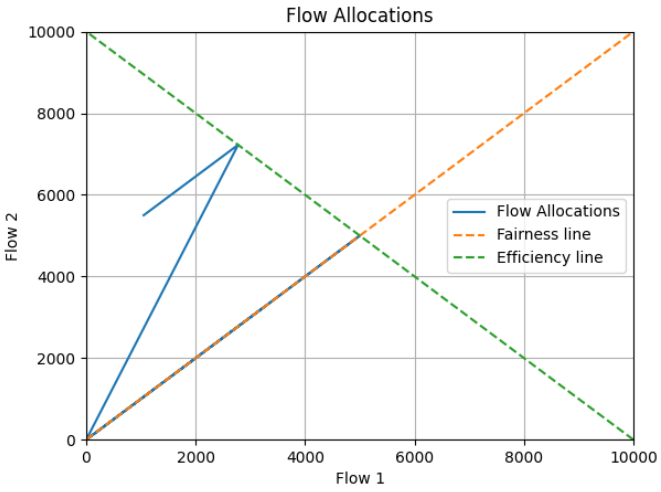

Figure 7.1
NUM_OF_FLOW = 2
ALPHA =1, BETA = 0

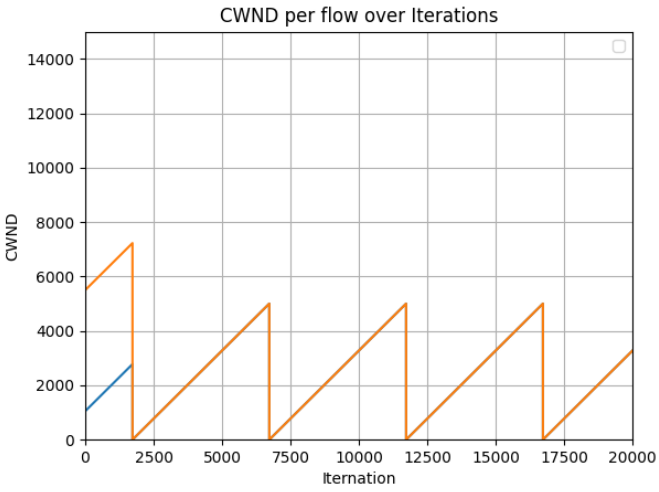

Figure 7.2
NUM_OF_FLOW = 2
ALPHA = 1, BETA = 0

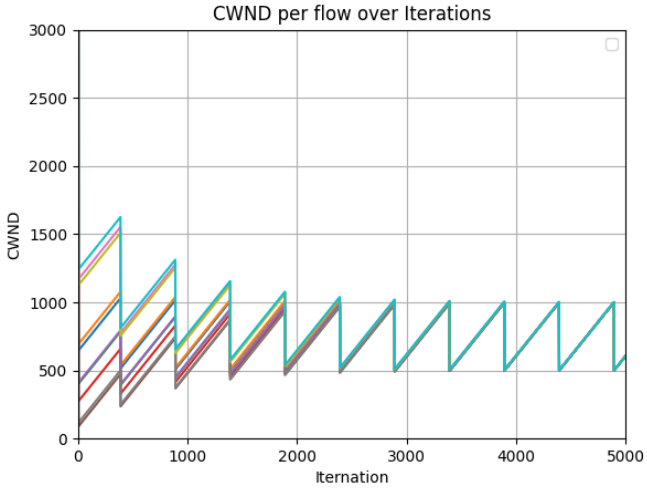

Figure 8
NUM_OF_FLOW = 10
ALPHA = 1, BETA = 0.5

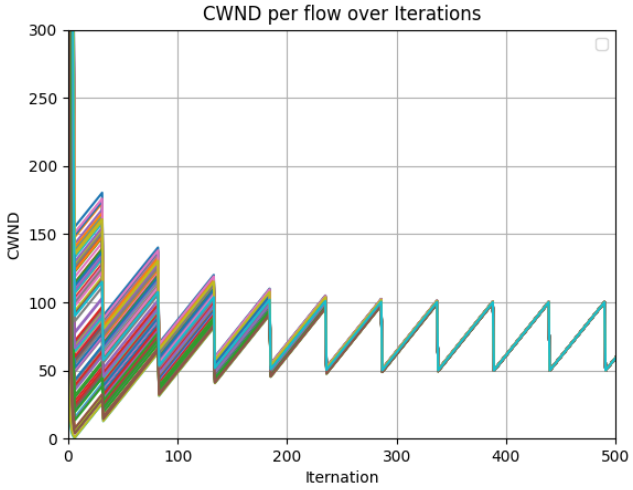

Figure 9
NUM_OF_FLOW = 100
ALPHA = 1, BETA = 0.5

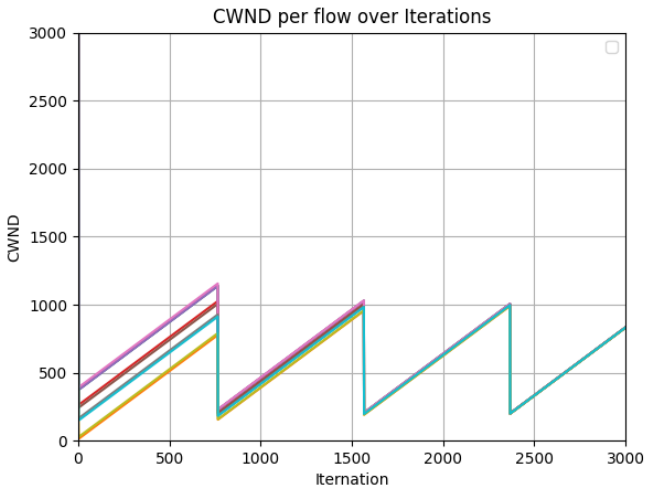

Figure 10
NUM_OF_FLOW = 10
ALPHA = 1, BETA = 0.2

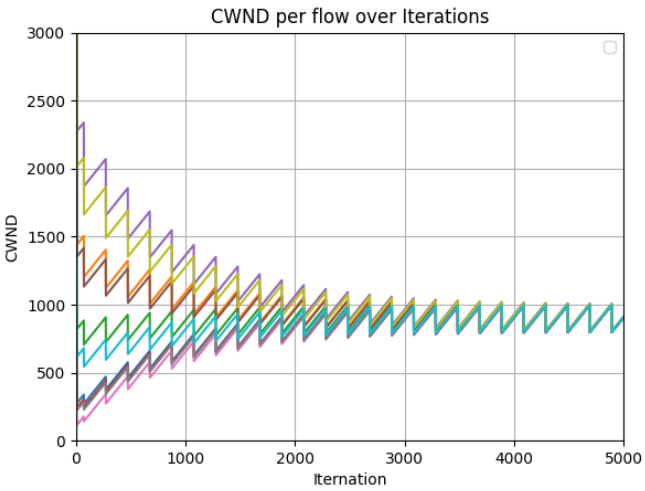

Figure 11
NUM_OF_FLOW = 10
ALPHA = 1, BETA = 0.8

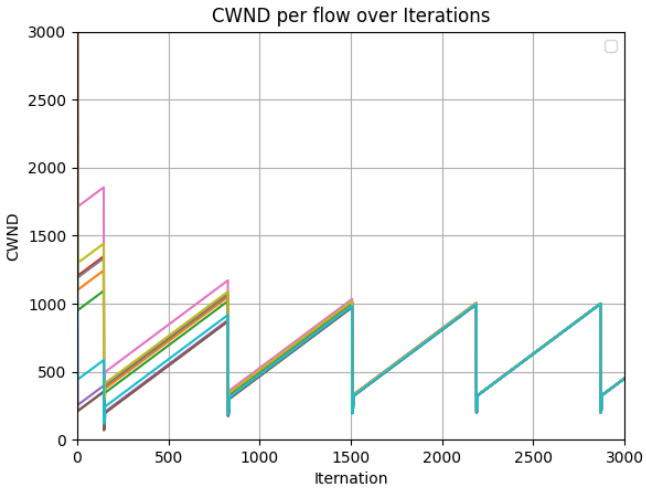

Figure 12
NUM_OF_FLOW = 10
ALPHA = 1, BETA = 0.2
SS_THRESH = 0.3

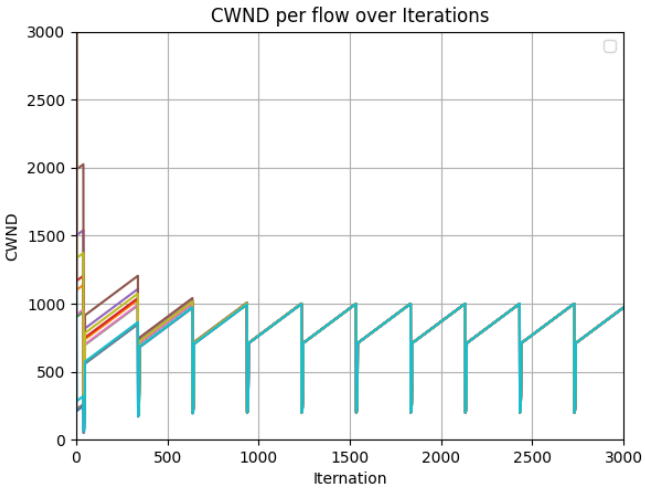

Figure 13
NUM_OF_FLOW = 10
ALPHA = 1, BETA = 0.2
SS_THRESH = 0.5

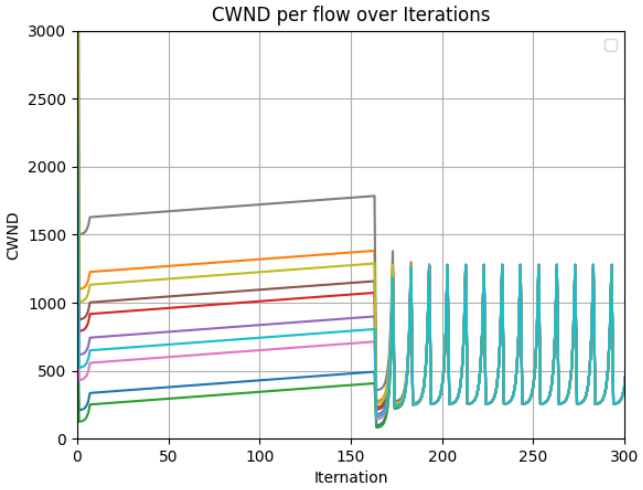

Figure 14
NUM_OF_FLOW = 10
ALPHA = 1, BETA = 0.2
SS_THRESH = 0.8

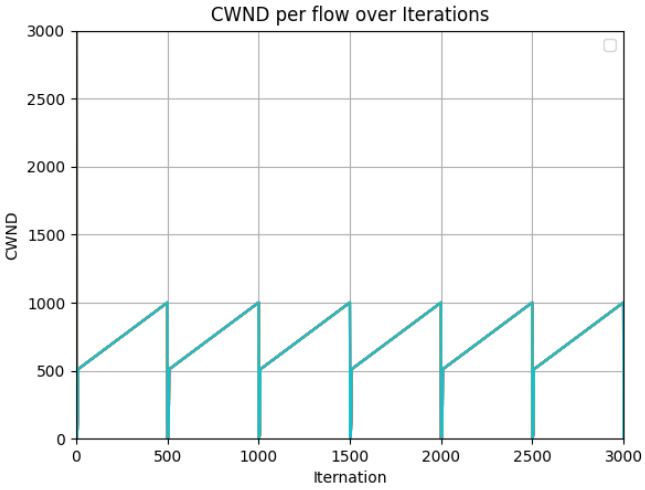

Figure 15
NUM_OF_FLOW = 10
ALPHA =1, BETA = 0
SS_THRESH = 0.5

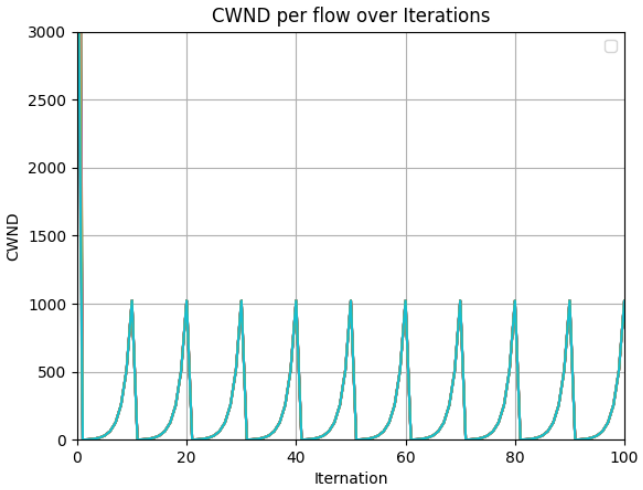

Figure 16
NUM_OF_FLOW = 10
ALPHA = 1, BETA = 0
SS_THRESH = 0.6

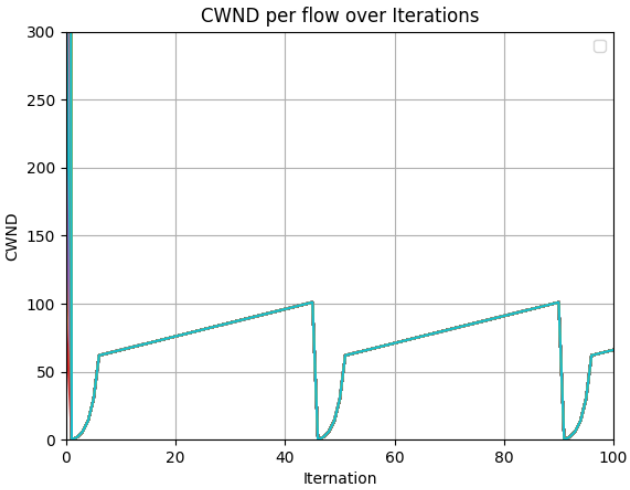

Figure 17
NUM_OF_FLOW = 100
ALPHA = 1, BETA = 0
SS_THRESH = 0.6

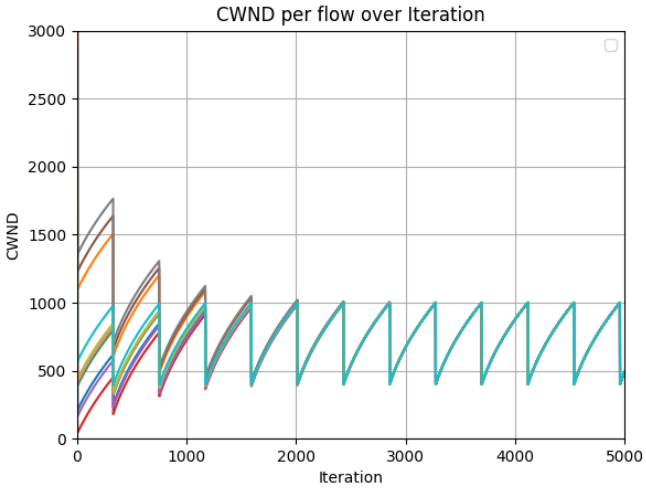

Figure 18
NUM_OF_FLOW = 10
ALPHA = 1, BETA = 0.5
DELTA = 0.1

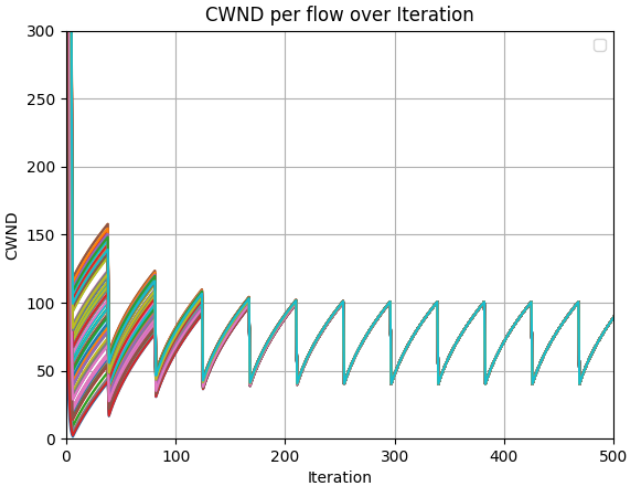

Figure 19
NUM_OF_FLOW = 100
ALPHA = 1, BETA = 0.5
DELTA = 0.1

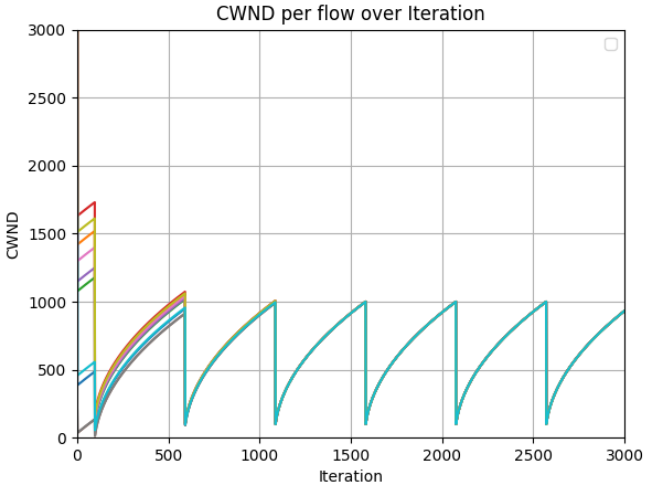

Figure 20
NUM_OF_FLOW = 10
ALPHA = 1, BETA = 0.2
DELTA = 0.1

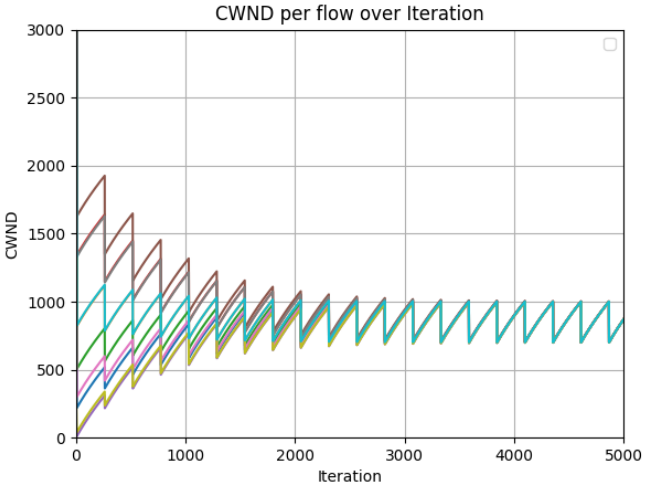

Figure 21
NUM_OF_FLOW = 10
ALPHA = 1, BETA = 0.8
DELTA = 0.1

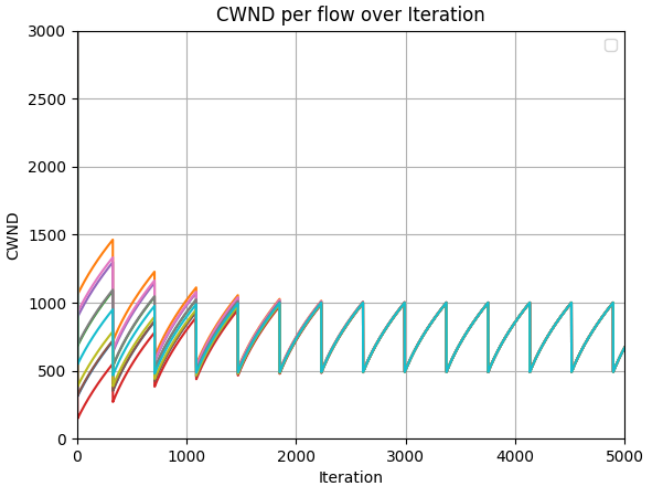

Figure 22
NUM_OF_FLOW = 10
ALPHA = 1, BETA = 0.5
DELTA = 0.01

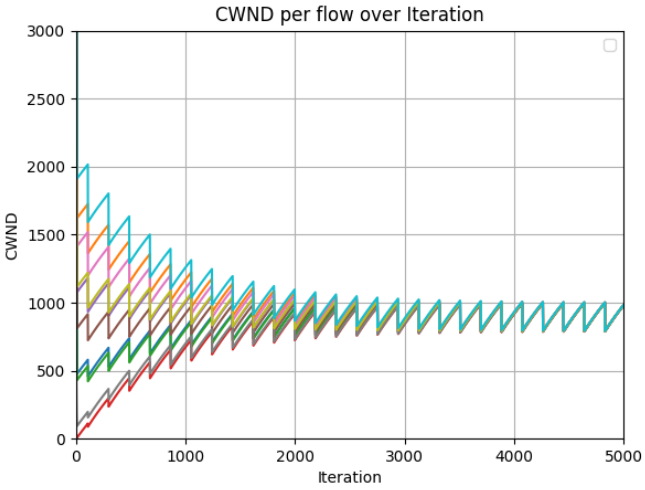

Figure 23
NUM_OF_FLOW = 10
ALPHA = 1, BETA = 0.8
DELTA = 0.01

Github repo URL: https://github.com/nigelhao/sc4052-assignments/blob/main/assignment1/aimd-simulator.py

```python
import numpy as np
import matplotlib.pyplot as plt

NUM_OF_FLOW = 10
CWND_SIZE = 10000

ALPHA = 1
BETA = 0.2

SS_THRESH = 0.5
SS_ALPHA = 2

DELTA = 0.01

ITER_LIMIT = CWND_SIZE * 2


def linear_AI(flow_allocations):
    flow_allocations += ALPHA
    return flow_allocations


def linear_MD(flow_allocations):
    flow_allocations *= BETA
    return flow_allocations


def nonlinear_SS(flow_allocations, ss_n):
    flow_allocations += SS_ALPHA**ss_n
    return flow_allocations


def adaptive_AI(flow_allocations):
    flow_allocations += ALPHA * (CWND_SIZE / round(np.sum(flow_allocations)))
    return flow_allocations


def adaptive_MD(flow_allocations):
    flow_allocations *= BETA - DELTA * (CWND_SIZE / round(np.sum(flow_allocations)))
    return flow_allocations


def graph_flow_allocation(allocation_history):

    x = y = np.linspace(0, CWND_SIZE, 400)
    w = np.linspace(0, CWND_SIZE, 400)
    v = -w + CWND_SIZE
    plt.figure(2)
    plt.plot(
        allocation_history[:, 0], allocation_history[:, 1], label="Flow Allocations"
    )
    plt.plot(x, y, label="Fairness line", linestyle="dashed")
    plt.plot(v, w, label="Efficiency line", linestyle="dashed")
    plt.xlabel("Flow 1")
    plt.ylabel("Flow 2")
    plt.title("Flow Allocations")
    plt.legend()
    plt.grid(True)
    plt.xlim(0, CWND_SIZE)
    plt.ylim(0, CWND_SIZE)
    plt.show()


def graph_cwnd_iternation(allocation_history):
    plt.figure(1)
    for i in range(NUM_OF_FLOW):
        plt.plot(allocation_history[:, i])

    plt.xlabel("Iteration")
    plt.ylabel("CWND")
```

```
    plt.title("CWND per flow over Iteration")
    plt.legend()
    plt.grid(True)
    plt.xlim(0, 5000)
    plt.ylim(0, 3 * (CWND_SIZE / NUM_OF_FLOW))
    plt.show()


def calculate_valid_windows(allocation_history):
    total = 0
    for flows in allocation_history:
        i_total = np.sum(flows)
        total += round(i_total) if i_total < CWND_SIZE else CWND_SIZE

    return total


def simulate_linear(flow_allocations):

    allocation_history = []
    for _ in range(ITER_LIMIT):

        allocation_history.append(flow_allocations.copy())
        if np.sum(flow_allocations) <= CWND_SIZE:
            flow_allocations = linear_AI(flow_allocations)
        else:
            flow_allocations = linear_MD(flow_allocations)

    return np.array(allocation_history)


def simulate_simplified_tcp_reno(flow_allocations):

    allocation_history = []
    ss_n = 0
    for _ in range(ITER_LIMIT):

        allocation_history.append(flow_allocations.copy())
        if np.sum(flow_allocations) <= (CWND_SIZE * SS_THRESH):
            ss_n += 1
            flow_allocations = nonlinear_SS(flow_allocations, ss_n)
        elif np.sum(flow_allocations) <= CWND_SIZE:
            ss_n = 0
            flow_allocations = linear_AI(flow_allocations)
        else:
            ss_n = 0
            flow_allocations = linear_MD(flow_allocations)

    return np.array(allocation_history)


def simulate_adaptive(flow_allocations):

    allocation_history = []
    for _ in range(ITER_LIMIT):

        allocation_history.append(flow_allocations.copy())

        if np.sum(flow_allocations) <= CWND_SIZE:
            flow_allocations = adaptive_AI(flow_allocations)
        else:
            flow_allocations = adaptive_MD(flow_allocations)

    return np.array(allocation_history)


def main():
    np.set_printoptions(precision=10)
    flow_allocations = np.random.randint(0, CWND_SIZE, size=NUM_OF_FLOW).astype(float)

    # flow_allocations[0] = 1050
    # flow_allocations[1] = 5500
```

```
143
144     # allocation_history = simulate_linear(flow_allocations)
145     # allocation_history = simulate_simplified_tcp_reno(flow_allocations)
146     allocation_history = simulate_adaptive(flow_allocations)
147
148     print("Total window(s) sent:", calculate_valid_windows(allocation_history))
149
150     if NUM_OF_FLOW == 2:
151         graph_flow_allocation(allocation_history)
152
153     graph_cwnd_iternation(allocation_history)
154
155
156 main()
157
```