Tuning TCP AIMD Parameters & TCP Users

1 Introduction

TCP (Transmission Control Protocol) is a widely used transport layer protocol in computer networks. One of the key mechanisms in TCP is the Additive Increase Multiplicative Decrease (AIMD) algorithm, which regulates the congestion window size to achieve fair and efficient network utilization. This report explores different variants of tuning AIMD parameters and investigates their impact on TCP dynamics with varying numbers of users sharing a single bottleneck.

2 Methodology: Tuning AIMD Parameters

The AIMD mechanism in TCP involves two parameters: alpha and beta. The alpha parameter controls the additive increase in the congestion window size, while the beta parameter controls the multiplicative decrease. Different functions can be used to design these parameters, affecting how TCP responds to network congestion.

2.1 Fixed Predefined Alpha and Beta Values

We used conventional alpha and beta parameters for the predefined values, setting alpha to 1 and beta to 0.5. These values represent the standard AIMD parameters commonly used in TCP implementations. This allows for comparison against custom functions and serves as a baseline for evaluation.

2.2 Custom Alpha and Beta Functions

Alternatively, custom alpha and beta functions can be tailored to specific network conditions. These functions can be exponential, logarithmic, polynomial, or sigmoid, each offering different behaviors in response to changes in the congestion window size. To ensure adherence to their respective purposes, we ensured that the alpha and beta functions were monotonically increasing and decreasing, respectively.

1. ***Exponential Functions:*** They exhibit rapid growth and decay, respectively, leading to aggressive window adjustments that can be suitable for dynamic network environments.
   1.1. Alpha Function: $\alpha(window) = e^{0.05\times window}$
   1.2. Beta Function: $\beta(window) = e^{-window}$
2. ***Logarithmic Functions:*** They result in slower adjustments, making it potentially suitable for stable networks with gradual changes.
   2.1. Alpha Function: $\alpha(window) = \log(window + 1)$
   2.2. Beta Function: $\beta(window) = \dfrac{1}{\log(window + 5)}$

3. ***Polynomial Functions:*** They offer a balance between rapid and gradual adjustments, depending on the degree of the polynomial. We chose different degrees to test out to observe how the degree affects the response of the TCP dynamics to network congestion.
   3.1. Alpha Function:
   $$\alpha(window) = 0.1 \times window^3 + 0.5 \times window^2 + 0.2 \times window + 1$$
   3.2. Beta Function: $\beta(window) = -0.01 \times window^2 + 0.05 \times window + 0.1$
4. ***Sigmoid Functions:*** They introduce a nonlinear response to congestion, with a gradual increase followed by a steep decrease.
   4.1. Alpha Function: $\alpha(window) = \dfrac{1}{1 + e^{-0.1 \times (window-5)}}$
   4.2. Beta Function: $\beta(window) = \dfrac{1}{1 + e^{-0.05 \times (window-7)}}$

2.3 Varying Number of Users and Scalability of Functions

We also explore the impact of varying the number of users sharing a single bottleneck in the network and assess the scalability of the system when employing a single function, specifically the exponential function, known for its quick convergence. We experimented with different user counts in multiples of 3 (i.e. 3, 6, 9) utilizing the same bottleneck. This was done to preserve the initial unfairness in the congestion windows.

2.4 Experiment Setup

To evaluate the performance of different combinations of alpha and beta functions in the AIMD mechanism of TCP (Sections 2.1-2.2), we conducted simulations with a TCP_Simulator class. Each simulation involved three users sharing a single bottleneck. The initial congestion window sizes for the users were set arbitrarily as [10, 1, 4] for users 1, 2, and 3, respectively. This initialization was chosen to intentionally create a scenario with low fairness, allowing us to observe how well each function redistributes fairness and how fast they are able to converge.

Using the TCP_Simulator class, we can specify fixed predefined alpha and beta values, or custom alpha and beta functions. The simulation runs for a maximum number of iterations (50 iterations) or until convergence is achieved (convergence threshold = 1×10^{-5}). During the simulations, we collected several metrics to assess the performance of the AIMD parameters:

1. **Number of Iterations until Converged:** This metric indicates the convergence speed of the TCP dynamics, i.e., the number of iterations required until the congestion windows stabilize. It is calculated by counting the iterations until the absolute difference between consecutive congestion window (cwnd) sizes falls below a predefined threshold.

   We count the number of iterations until $|\text{cwnd}_{\text{current}} - \text{cwnd}_{\text{previous}}| < \text{Convergence Threshold}$

2. **Final Congestion Windows:** These are the congestion window sizes achieved by each user at convergence. They provide insights into how well the TCP algorithm adjusts the window sizes to optimize network utilization.
3. **Throughputs:** Throughput represents the amount of data successfully transmitted per unit time by each user. It is calculated as the product of the congestion window size and the corresponding alpha value. Throughput = Congestion Window * Alpha.
4. **Network Throughput:** This metric represents the total amount of data transmitted per unit time across all users. It provides an overall measure of network efficiency and utilization. Network Throughput = $\sum_{\text{all users}}$ Throughput.
5. **Jain's Fairness Index (JFI):** JFI quantifies the fairness of throughput distribution among users. A value close to 1 indicates perfect fairness, while lower values signify increasing levels of unfairness. $\text{Jain's Fairness Index} = \frac{(\sum \text{Throughput})^2}{N \sum (\text{Throughput}^2)}$.
6. **Throughput Fairness:** This metric compares the throughput achieved by the user with the highest throughput to that of the user with the lowest throughput. A higher throughput fairness value signifies a more balanced distribution of network resources. $\text{Throughput Fairness} = \frac{\text{Minimum Throughput}}{\text{Maximum Throughput}}$.

3 Results and Discussion

Function	Predefined Fixed	Exponential	Logarithmic	Polynomial	Sigmoid
Number of iterations until converged	18	10	19	6	18
Final Congestion Windows (in integers)	[5 5 5]	[5 5 4]	[7 3 5]	[13 0 2]	[7 4 5]
Throughputs	[5.00 5.00 5.00]	[7.12 7.20 5.03]	[14.11 3.79 9.82]	[3.55×10^3 0.26 10.43]	[3.77 1.66 2.21]
Network Throughput	15.00	19.34	27.73	3564.21	7.64
Jain's Fairness Index	1.00	0.98	0.83	0.34	0.89
Throughput Fairness	1.00	0.70	0.27	7.32×10^{-5}	0.44

Table 1: Custom Alpha and Beta Functions

From Table 1, we observe that the predefined alpha and beta values led to equal congestion window sizes and high fairness, aligning with conventional TCP behavior. In contrast, using exponential functions resulted in comparable convergence speeds but slightly reduced fairness, indicating a less equitable distribution of throughput among users while maintaining reasonable network utilization. Logarithmic functions achieved fast convergence but exhibited lower fairness metrics, indicating a more pronounced imbalance in throughput allocation despite high network utilization. Polynomial functions showed rapid convergence and high network throughput but significantly lower fairness metrics, suggesting poor fairness in throughput allocation. Lastly, sigmoid functions achieved a relatively fair distribution of throughput among users but lower network throughput, indicating suboptimal network utilization despite fairness.

The findings underscore the critical impact of alpha and beta function selection on TCP dynamics and network performance. The exponential function, despite slightly lower fairness, offers fast convergence and high network throughput, making it suitable for scenarios prioritizing a balance between fairness and efficiency. Conversely, the polynomial function, while sacrificing fairness, excels in maximizing network throughput, making it valuable in contexts where throughput optimization is paramount, such as financial trading or real-time data processing.

In industries like telecommunications, where network performance is crucial, these findings provide valuable insights for optimizing resource allocation and overall system performance. Whether prioritizing fairness or maximizing throughput, the choice of alpha and beta functions can significantly influence network dynamics and efficiency, offering tailored solutions to meet diverse industry requirements.

While our experiments have provided valuable insights into the impact of different alpha and beta functions on TCP dynamics and network performance, it is crucial to note that these results are based on a single set of hyperparameters for each function. Further optimization through hyperparameter tuning could potentially yield better results. Techniques like GridSearch can systematically explore a range of hyperparameter values to achieve faster convergence, higher network throughput, or improved fairness metrics, depending on the specific use case. For instance, adjusting the parameters of the exponential function might lead to even faster convergence without sacrificing fairness excessively. Similarly, fine-tuning the parameters of other functions could strike a better balance between fairness and network efficiency.

Users	**3**	**6**	**9**
Number of iterations until converged	10	10	10
Final Congestion Windows (in integers)	[5 5 4]	[5 5 4 5 5 4]	[5 5 4 5 5 4 5 5 4]
Throughputs	[7.12 7.20 5.03]	[7.12 7.20 5.03 7.12 7.20 5.03]	[7.12 7.20 5.03 7.12 7.20 5.03 7.12 7.20 5.03]

Users	3	6	9
Network Throughput	19.34	38.70	58.04
Jain's Fairness Index	0.98	0.98	0.98
Throughput Fairness	0.70	0.70	0.70

Table 2: Varying number of users

As we vary the number of users from 3 to 6 and then to 9 while retaining the exponential function, we observe consistent behavior in terms of convergence and fairness metrics (Table 2). Despite the increased number of users, the exponential function maintains fast convergence and high network throughput. The final congestion window sizes and throughputs for each user remain consistent, indicating the scalability of the exponential function in regulating TCP dynamics across different network sizes.

Interestingly, as the number of users increases, the network throughput also increases proportionally, highlighting the efficient utilization of network resources facilitated by the exponential function. Additionally, the fairness metrics, including JFI and throughput fairness, remain stable across different user counts, indicating a fair distribution of throughput among users even in larger networks.

4 Conclusion

In conclusion, our study underscores the pivotal role of alpha and beta functions in shaping TCP dynamics and network performance. While fixed predefined values uphold stability and fairness with efficient network utilization, alternative functions offer trade-offs between convergence speed, fairness, and efficiency.

Selecting the most suitable alpha and beta functions requires careful consideration of specific network conditions and requirements. Our findings shed light on the relative performance of different functions, providing valuable insights for network optimization.

However, there remains ample opportunity for further optimization and refinement through hyperparameter tuning. This iterative process ensures continuous enhancement of network performance to meet evolving industry demands and application requirements.

Moreover, our experiments highlight the scalability of the exponential function in regulating TCP dynamics across varying user numbers. Its ability to ensure quick convergence and fair resource allocation underscores its robustness in scalable network environments.

In summary, our research contributes to advancing the understanding of TCP dynamics and lays the groundwork for future studies to explore novel approaches for optimizing network performance in diverse contexts.¶

5 Appendix

Code below can also be found on: https://github.com/googlercolin/TuningAIMD

5.1 Code to Compare Custom Alpha and Beta Functions

```
import numpy as np

class TCP_Simulator:
    def __init__(self, alphas=None, betas=None, alpha_function=None, beta_func-
tion=None, initial_window=[10, 1, 4]):
        if alphas is not None and betas is not None:
            self.alphas = alphas
            self.betas = betas
        elif alpha_function is not None and beta_function is not None:
            self.alpha_function = alpha_function
            self.beta_function = beta_function
        else:
            raise ValueError("Please provide either alphas and betas or alpha_-
function and beta_function.")

        self.window = np.array(initial_window, dtype=int)
        self.congestion_state = np.zeros(len(self.window), dtype=int)  # 0: Con-
gestion Avoidance, 1: Fast Recovery
        self.ssthresh = np.inf
        self.throughputs = np.zeros(len(self.window))  # Store throughput for
each user

    def update_alpha(self):
        if hasattr(self, 'alphas'):
            return self.alphas
        else:
            return self.alpha_function(self.window)

    def update_beta(self):
        if hasattr(self, 'betas'):
            return self.betas
        else:
            return self.beta_function(self.window)

    def construct_transition_matrix(self):
        n = len(self.window)
        A = np.diag(self.update_beta()) + np.outer(self.update_alpha(),
(np.ones(n) - self.update_beta())) / sum(self.update_alpha())
        return A

    def update_congestion_window(self, A):
        if np.any(self.congestion_state == 1):  # In fast recovery phase
            self.window += 1  # Increase window by 1 for each ACK received
            lost_packet_idx = np.argmax(self.congestion_state == 1)  # Identify
the packet loss
            self.window[lost_packet_idx] = self.ssthresh  # Set window to
ssthresh
            self.congestion_state[self.congestion_state == 1] = 0  # Exit fast
recovery
```

```
        else:  # In congestion avoidance or slow start phase
            self.window = np.dot(A, self.window)  # AIMD: Increase window by 1
for each RTT
            if np.any(self.window >= self.ssthresh):  # Enter fast recovery upon
detecting packet loss
                self.ssthresh = np.max(self.window)  # Set ssthresh to the maxi-
mum window size achieved
                self.window = np.floor(self.ssthresh / 2)  # Reduce window to
half of ssthresh
                self.congestion_state[self.window < self.ssthresh] = 1  # Enter
fast recovery

    def simulate(self, max_iterations=50, convergence_threshold=1e-5):
        A = self.construct_transition_matrix()

        for i in range(max_iterations):
            prev_window = np.copy(self.window)
            self.update_congestion_window(A)

            # Calculate throughput for each user
            self.throughputs = self.window * self.update_alpha()

            if np.all(np.abs(self.window - prev_window) <
convergence_threshold):
                print("Convergence achieved.")
                break
            print(f"Iteration {i+1}: Congestion window = {self.window}")

        print("Final congestion windows (rounded to integers):", np.round(self-
.window).astype(int))

        # Calculate fairness and efficiency metrics
        self.calculate_metrics()

    def calculate_metrics(self):
        print("Throughputs:", self.throughputs)
        # Efficiency Metrics
        total_throughput = np.sum(self.throughputs)
        print("Network Throughput:", total_throughput)

        # Jain's Fairness Index
        jfi = np.sum(self.throughputs)**2 / (len(self.window) * np.sum(self-
.throughputs**2))
        print("Jain's Fairness Index:", jfi)

        # Throughput fairness
        max_throughput = np.max(self.throughputs)
        min_throughput = np.min(self.throughputs)
        throughput_fairness = min_throughput / max_throughput
        print("Throughput Fairness:", throughput_fairness)


def exponential_alpha(window):
    return np.exp(0.05*window)
```

```python
def exponential_beta(window):
    return np.exp(-window)

def logarithmic_alpha(window):
    return np.log(window + 1)

def logarithmic_beta(window):
    return 1 / (np.log(window + 5))

def polynomial_alpha(window):
    return 0.1 * window**3 + 0.5 * window**2 + 0.2 * window + 1

def polynomial_beta(window):
    return -0.01 * window**2 + 0.05 * window + 0.1

def sigmoid_alpha(window):
    return 1 / (1 + np.exp(-0.1 * (window - 5)))

def sigmoid_beta(window):
    return 1 / (1 + np.exp(-0.05 * (window - 7)))

def main():
    # Initial window sizes for multiple users
    initial_window = [10, 1, 4]

    # Predefined alpha and beta values
    alphas = np.array([1, 1, 1])  # List of alpha values for each user
    betas = np.array([0.5, 0.5, 0.5])   # List of beta values for each user
    tcp_simulator = TCP_Simulator(alphas=alphas, betas=betas,
initial_window=initial_window)
    print("\nFixed Predefined Alphas and Betas:")
    print("Alphas:", alphas)
    print("Betas:", betas)
    tcp_simulator.simulate()

    # Create TCP simulator instances with custom alpha and beta functions
    # Exponential functions
    tcp_simulator_exp = TCP_Simulator(alpha_function=exponential_alpha, beta_-
function=exponential_beta, initial_window=initial_window)
    print("\nExponential Functions:")
    tcp_simulator_exp.simulate()

    # Logarithmic functions
    tcp_simulator_log = TCP_Simulator(alpha_function=logarithmic_alpha, beta_-
function=logarithmic_beta, initial_window=initial_window)
    print("\nLogarithmic Functions:")
    tcp_simulator_log.simulate()

    # Polynomial functions
    tcp_simulator_poly = TCP_Simulator(alpha_function=polynomial_alpha, beta_-
function=polynomial_beta, initial_window=initial_window)
    print("\nPolynomial Functions:")
    tcp_simulator_poly.simulate()
    # Sigmoid functions
    tcp_simulator_sig = TCP_Simulator(alpha_function=sigmoid_alpha, beta_func-
tion=sigmoid_beta, initial_window=initial_window)
    print("\nSigmoid Functions:")
    tcp_simulator_sig.simulate()

if __name__ == "__main__":
    main()
```

5.2 Code to Compare Varying Number of Users and Scalability of Functions

```
import numpy as np

class TCP_Simulator:
    def __init__(self, alpha_function=None, beta_function=None,
initial_window=[10, 1, 4]):
        self.alpha_function = alpha_function
        self.beta_function = beta_function

        self.window = np.array(initial_window, dtype=int)
        self.congestion_state = np.zeros(len(self.window), dtype=int)  # 0: Con-
gestion Avoidance, 1: Fast Recovery
        self.ssthresh = np.inf
        self.throughputs = np.zeros(len(self.window))  # Store throughput for
each user

    def update_alpha(self):
        return self.alpha_function(self.window)

    def update_beta(self):
        return self.beta_function(self.window)

    def construct_transition_matrix(self):
        n = len(self.window)
        A = np.diag(self.update_beta()) + np.outer(self.update_alpha(),
(np.ones(n) - self.update_beta())) / sum(self.update_alpha())
        return A

    def update_congestion_window(self, A):
        if np.any(self.congestion_state == 1):  # In fast recovery phase
            self.window += 1  # Increase window by 1 for each ACK received
            lost_packet_idx = np.argmax(self.congestion_state == 1)  # Identify
the packet loss
            self.window[lost_packet_idx] = self.ssthresh  # Set window to
ssthresh
            self.congestion_state[self.congestion_state == 1] = 0  # Exit fast
recovery
        else:  # In congestion avoidance or slow start phase
            self.window = np.dot(A, self.window)  # AIMD: Increase window by 1
for each RTT
            if np.any(self.window >= self.ssthresh):  # Enter fast recovery upon
detecting packet loss
```

```
                self.ssthresh = np.max(self.window)  # Set ssthresh to the maxi-
mum window size achieved
                self.window = np.floor(self.ssthresh / 2)  # Reduce window to
half of ssthresh
                self.congestion_state[self.window < self.ssthresh] = 1  # Enter
fast recovery

    def simulate(self, max_iterations=50, convergence_threshold=1e-5):
        A = self.construct_transition_matrix()

        for i in range(max_iterations):
            prev_window = np.copy(self.window)
            self.update_congestion_window(A)

            # Calculate throughput for each user
            self.throughputs = self.window * self.update_alpha()

            if np.all(np.abs(self.window - prev_window) <
convergence_threshold):
                print("Convergence achieved.")
                break
            print(f"Iteration {i+1}: Congestion window = {self.window}")

        print("Final congestion windows (rounded to integers):", np.round(self-
.window).astype(int))

        # Calculate fairness and efficiency metrics
        self.calculate_metrics()

    def calculate_metrics(self):
        print("Throughputs:", self.throughputs)
        # Efficiency Metrics
        total_throughput = np.sum(self.throughputs)
        print("Network Throughput:", total_throughput)

        # Jain's Fairness Index
        jfi = np.sum(self.throughputs)**2 / (len(self.window) * np.sum(self-
.throughputs**2))
        print("Jain's Fairness Index:", jfi)

        # Throughput fairness
        max_throughput = np.max(self.throughputs)
        min_throughput = np.min(self.throughputs)
        throughput_fairness = min_throughput / max_throughput
        print("Throughput Fairness:", throughput_fairness)


def exponential_alpha(window):
    return np.exp(0.05*window)

def exponential_beta(window):
    return np.exp(-window)

def main():
    # Initial window sizes for multiple users (example)
    initial_window = [10, 1, 4]

    # Scale initial window sizes for 5 and 10 users
    initial_window_6_users = [10, 1, 4, 10, 1, 4]
    initial_window_9_users = [10, 1, 4, 10, 1, 4, 10, 1, 4]

    # Exponential functions for alpha and beta
    alpha_function = exponential_alpha
    beta_function = exponential_beta

    print("\nEvaluation with 3 users:")
    tcp_simulator = TCP_Simulator(alpha_function=alpha_function,
beta_function=beta_function, initial_window=initial_window)
    tcp_simulator.simulate()

    print("\nEvaluation with 6 users:")
    tcp_simulator_6_users = TCP_Simulator(alpha_function=alpha_function, beta_-
function=beta_function, initial_window=initial_window_6_users)
    tcp_simulator_6_users.simulate()

    print("\nEvaluation with 9 users:")
    tcp_simulator_9_users = TCP_Simulator(alpha_function=alpha_function, beta_-
function=beta_function, initial_window=initial_window_9_users)
    tcp_simulator_9_users.simulate()

if __name__ == "__main__":
    main()
```