SC4052 Assignment 1

Lim Zhe Xun (U2121481J)

Introduction

The Transmission Control Protocol (TCP) is a fundamental protocol that ensures reliable data transmission across the Internet. However, TCP faces challenges in the data center environment, due to high variations in workloads requiring predictable latency and large sustained throughput [2]. This report aims to shed light on how TCP algorithms can be designed for data centers and conduct analysis on their performance.

Literature Review

Homa

Homa is a new network transport system designed specifically for data centers, addressing issues with TCP while incorporating lessons from technologies like Infiniband and RDMA [1].

Design

1. **Messages** – Unlike TCP, which relies on streams, Homa is message-based. It operates using remote procedure calls (RPCs), where a client sends a request message to a server and receives a response. This design improves efficiency by allowing multiple threads to read from a single socket and enables direct dispatching of messages to worker threads.
2. **Connectionless** – Homa is also connectionless, meaning it does not require a setup phase before communication, unlike TCP. This eliminates overhead and allows a single socket to handle multiple RPCs with different peers simultaneously. Each RPC operates independently, without enforcing a strict order between them, making communication more flexible and efficient.
3. **Network States** – Additionally, Homa manages network state more efficiently than TCP. It maintains minimal state per socket, RPC, and peer, reducing memory usage and improving scalability. Despite being connectionless, it ensures reliable communication by handling flow control, retries, and congestion control at the RPC level.

Overall, Homa offers a more scalable, efficient, and flexible approach to network transport in data centers compared to traditional TCP.

DCTCP

Data Center TCP (DCTCP) is a modified version of TCP designed specifically for data centers [2]. It improves efficiency by using an existing network feature called Explicit Congestion Notification (ECN) [3]. ECN allows network switches to notify computers when congestion is about to occur, so they can slow down data transfer before packets are lost.

Design

The DCTCP functions on 3 key components:

1. **Simple Marking at the Switch** – Using a single parameter K as marking threshold, marks arriving packets with the CE codepoint if the queue is greater than K.
2. **ECN-Echo at Receiver** – Compared to TCP receivers, DCTCP receivers convey the exact sequence of packets that have been marked with the CE codepoint. To reduce communication overhead, DCTCP groups multiple acknowledgments while still accurately tracking congestion.
3. **Controller at Sender** – The sender first estimates the ratio of packets that are marked using the formula:

   $$a \leftarrow (1 - g) * a + g * F$$

   where 0<g<1 and F is the fraction of packets marked in the last window. In contrast to TCP which halves its window size per marked packet, DCTCP uses the following formula to readjust its window size:

   $$cwnd \leftarrow cwnd * (1 - {}^{\alpha}/_{2})$$

   When congestion is high (a=1), DCTCP halves its window size like TCP. But when congestion is low (a=0), DCTCP reduces its window size inversely proportional to a.

Unlike standard TCP, which reduces its speed drastically after detecting congestion, DCTCP adjusts more smoothly based on the level of congestion, maintaining efficiency while keeping delays low.

TCP CUBIC

CUBIC is an improved version of standard TCP designed for high-speed and long-distance networks [4]. Traditional TCP struggles with low performance in such environments due to its linear congestion control, so CUBIC replaces this with a cubic function to improve scalability and stability.

One of CUBIC's key features is that its window growth is based on real-time calculations rather than being dependent on round-trip time (RTT). This allows it to be more stable and fair while maintaining the scalability benefits of its predecessor, BIC. The protocol aims to balance congestion window size before and after congestion occurs, ensuring smoother performance.

Design

1. **Slow Start** – TCP CUBIC employs a modified slow start behaviour. Once the window size rises above the preset threshold, a less aggressive exponential increase is used instead.
2. **Clamp on Additive Increase** – During AIMD operation, the rate at which the congestion window increases is limited to a maximum of 20 times the minimum round-trip delay in packets per RTT. This effectively caps the increase rate at around 20 packets per RTT.
3. **Cubic Function** – The window size is adjusted according to the formula below:

$$K = \sqrt[3]{(Wmax * (1 - \beta)/C)}$$

$$cwnd = Wmax + C\ (t - K)$$

Where Wmax is the window size before congestion, and B and C are algorithm parameters.

TCP-LLM

The emergence of Large Language Models (LLMs) such as GPT-4 opens up new possibilities for TCP optimization. LLMs excel in reasoning, adaptability, and generalization, which make LLMs suitable for addressing TCP challenges in dynamic networks. TCP-LLM proposes a framework consisting of three main components: the Integrated Encoder, the TCP-LLM Head, and Low-Rank TCP Adaptation [5]. Together, these components facilitate efficient processing of TCP-specific data and improve the model's adaptability.

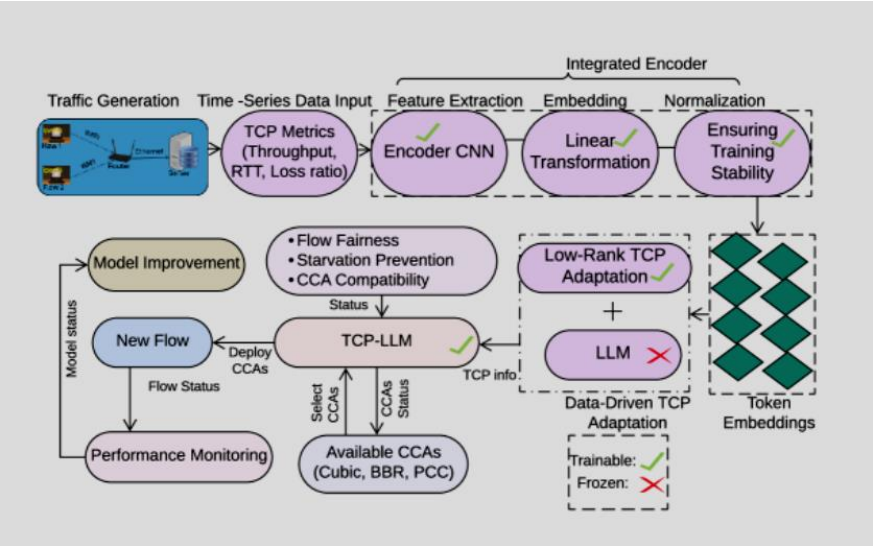


Fig. 1. TCP-LLM architecture [5]

Design

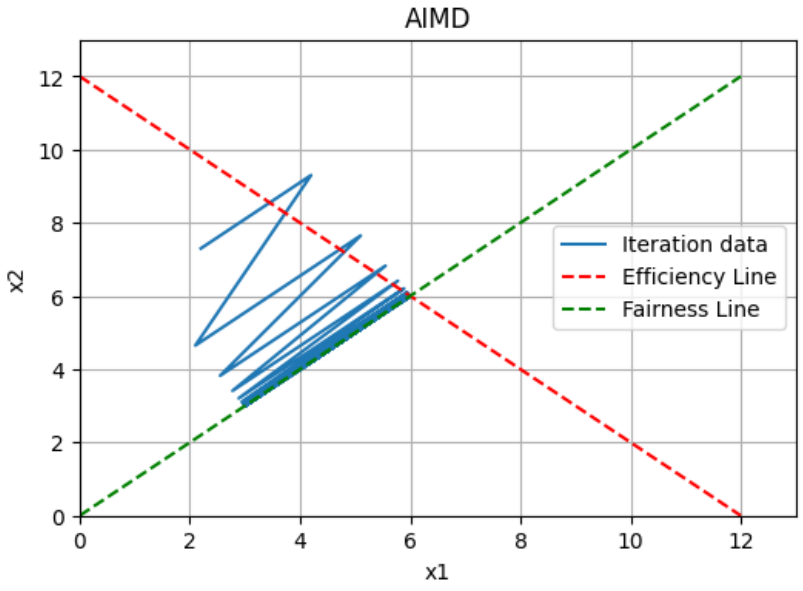
1. **Integrated Encoder** – The design incorporates a 1-dimensional convolutional neural network to process TCP time-series data and capture temporal dependencies. At the same time, scalar metrics like throughput and latency are encoded into embeddings for LLM downstream tasks.
2. **TCP-LLM Head** – This module is designed to predict the optimal CCA based on TCP metrics. It starts by initializing the model with relevant input parameters, including the number of input features and available CCAs. A forward pass is then performed to generate predictions.
3. **Low-Rank TCP Adaptation** – To reduce the computational costs of finetuning the entire LLM, this module utilizes Low-Rank Adaptation (LoRA) which focuses on training only on the parameter efficient matrices of the model. This enables the model to address key issues in TCP congestion control, namely flow unfairness, TCP starvation, and CCA incompatibility

AIMD Tuning

This section explores and visualizes how the AIMD parameters can be innovated. Following tutorial, I implemented the code for the AIMD, DCTCP and CUBIC algorithms and analyze their graphs on convergence, efficiency and fairness. For all experiments, the variables were kept constant at x1=1.2, x2=6.3, threshold=12, max iterations= 100. All code was written in Python 3 and can be referenced in the appendix.

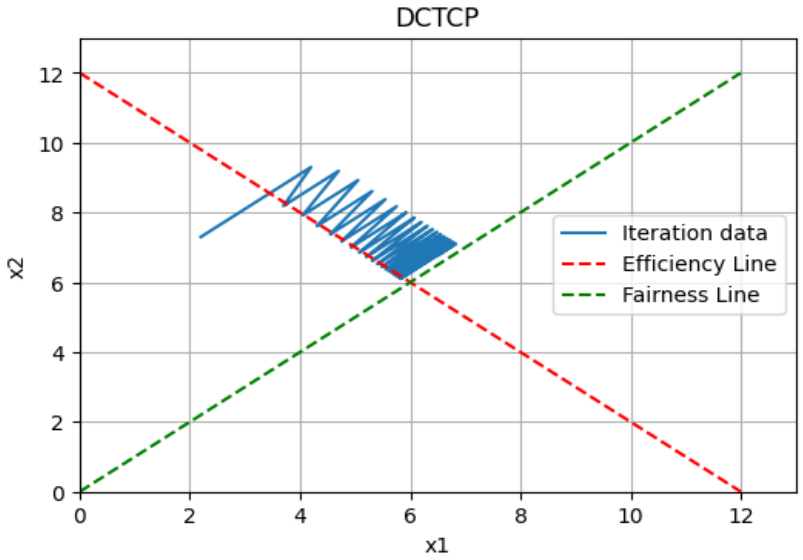
AIMD (Baseline)

Using the default values of a=1,b=0.5, it is observed that AIMD quickly converges to the fairness line, but below the efficiency line. This observation suggests that AIMD prioritizes fairness over efficiency in its congestion control mechanism.



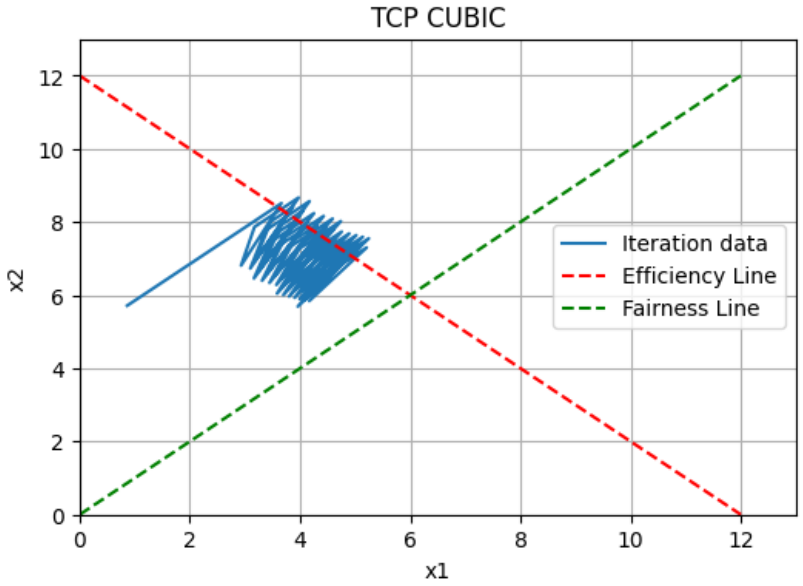
DCTCP

DCTCP replaces AIMD's multiplicative function with one that adjusts the window size inversely proportional to the ratio of previously marked packets. From the graph below, it is observed that DCTCP converges quickly towards the efficiency and fairness lines. However, this algorithm converges above the efficiency line, indicating that the network would always experience a small amount of congestion.



TCP CUBIC

TCP CUBIC replaces the additive function with a cubic function corresponding to the time elapsed since the last congestion. This algorithm quickly converges to the efficiency line while avoiding overshooting. However, it takes significantly longer to converge to the fairness line, suggesting that it does not effectively balance bandwidth between competing flows in the short run.



References

[1] B. Montazeri, Y. Li, M. Alizadeh, and J. Ousterhout. Homa: A Receiver-Driven Low-Latency Transport Protocol Using Net work Priorities. In Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, SIG COMM '18, pages 221—-235, New York, NY, USA, 2018. Association for Computing Machinery.

[2] Alizadeh, M., Greenberg, A., Maltz, D. A., Padhye, J., Patel, P., Prabhakar, B., ... & Sridharan, M. (2010, August). Data center tcp (dctcp). In *Proceedings of the ACM SIGCOMM 2010 Conference* (pp. 63-74).

[3] Ramakrishnan, K., Floyd, S., & Black, D. (2001). Rfc3168: The addition of explicit congestion notification (ecn) to ip.

[4] Gilkar, G. A., & Sahdad, Y. (2014). TCP CUBIC-congestion control transport protocol. *International Journal of in Multidisciplinary and Academic Research (SSIJMAR)*, *3*(5), 116-120.

Appendix

Graph Visualization Code

```
def plot(x1_values, x2_values, threshold, chart_title='x2 vs x1'):
    plt.figure(figsize=(6, 4))
    plt.plot(x1_values, x2_values, label="Iteration data")
    plt.plot([0, threshold], [threshold, 0], label='Efficiency Line',
linestyle='--', color='red')
    plt.plot([0, threshold], [0, threshold], label='Fairness Line',
linestyle='--', color='green')


    plt.xlabel('x1')
    plt.ylabel('x2')
    plt.title(chart_title)
    upper_limit = max([x1_values.max(), x2_values.max(), threshold])
    plt.xlim(0, int(upper_limit)+1)
    plt.ylim(0, int(upper_limit)+1)
    plt.legend()
    plt.grid(True)
    plt.show()
```

AIMD Algorithm

```
def aimd(x1:float, x2:float, alpha:float=1, beta:float=0.5,
threshold:int=12, iterations:int=100):
    x1_values = np.zeros(iterations)
    x2_values = np.zeros(iterations)

    for i in tqdm(range(iterations)):
        if x1+x2 <= threshold:
            x1 += alpha
            x2 += alpha

        else:
            x1 *= beta
            x2 *=beta

        x1_values[i] = x1
        x2_values[i] = x2

    return x1_values, x2_values
```

DCTCP Algorithm

```
def dctcp(x1:float, x2:float, alpha:float=1, g:float=0.1, threshold:int=12,
iterations:int=100):
    x1_values = np.zeros(iterations)
    x2_values = np.zeros(iterations)

    a = 0

    for i in tqdm(range(iterations)):
        if x1+x2 <= threshold:
            # Follow additive increment from AIMD
            x1 += alpha
            x2 += alpha

        else:
            # Get fraction of marked packets
            marked = x1+x2-threshold
            f = marked/(x1+x2)

            # Apply formula
            a = (1-g) * a + g * f
            x1 *= (1-a/2)
            x2 *= (1-a/2)

        x1_values[i] = x1
        x2_values[i] = x2

    return x1_values, x2_values
```

TCP CUBIC Algorithm

```
def cubic(x1: float, x2: float, C: float = 0.4, beta: float = 0.8,
threshold: int = 12, iterations: int = 100):
    x1_values = np.zeros(iterations)
    x2_values = np.zeros(iterations)

    # Initialize W_max and time elapsed since last congestion
    x1_max, x2_max = x1, x2
    t = 0

    for i in tqdm(range(iterations)):
        if x1 + x2 <= threshold:
            # Compute K
            k1 = (x1_max * (1-beta) / C) ** (1/3)
            k2 = (x2_max * (1-beta) / C) ** (1/3)

            # Apply cubic formula
```

```
            x1 = x1_max + C * (t-k1)
            x2 = x2_max + C * (t-k2)
            t += 1
        else:
            # If congested, set new W_max values before decreasing value
            x1_max = x1
            x2_max = x2
            x1 *= beta
            x2 *= beta
            t = 0

        x1_values[i] = x1
        x2_values[i] = x2

    return x1_values, x2_values
```