Topic: TCP-Congestion Control Using Machine Learning

Members:

Gouri Verma(220CS01074)

Sayali Abhay Khamitkar(22CS01052)

Varsha Swaraj (22CS02005)

Sai Nikhita Palisetty (22CS01050)

TCP Tahoe

Objective:

We implemented and analyzed TCP Tahoe behavior in NS-3 using OpenGym, focusing on congestion window (cwnd), round-trip time (RTT), and throughput.

Methodology:

- Integrated TCP Tahoe logic into NS-3's TCP stack.
- Used ns3-OpenGym interface to extract cwnd, RTT, and throughput values during simulation.
- Simulated standard dumbbell topology.
- Monitored congestion events and recovery behavior.

```python
def get_action(self, obs, reward, done, info):
       socketUuid = obs[0]# unique socket ID
       envType = obs[1]  # TCP env type: event-based = 0 / time-based = 1
       simTime_us = obs[2] # sim time in us
       nodeId = obs[3] # unique node ID
       ssThresh = obs[4]# current ssThreshold
       cWnd = obs[5]# current contention window size
       segmentSize = obs[6]# segment size
       segmentsAcked = obs[7 # number of acked segments
       bytesInFlight  = obs[8] # estimated bytes in flight
       caState = obs[12]
       caEvent = obs[13]
       new_cWnd = 1
       new_ssThresh = 1
       if caEvent == 2:  # CA_EVENT_LOSS
           new_ssThresh = segmentSize
           new_cWnd = segmentSize  # Reset to 1 segment (slow start)
       elif (cWnd < ssThresh):      # IncreaseWindow
           # slow start
           if (segmentsAcked >= 1):

      elif (cWnd >= ssThresh):
            if (segmentsAcked > 0):# congestion avoidance
               adder = 1.0 * (segmentSize * segmentSize) / cWnd;
               adder = int(max (1.0, adder))
               new_cWnd = cWnd + adder
       new_ssThresh = int(max (2 * segmentSize, bytesInFlight / 2))
       actions = [new_ssThresh, new_cWnd]
       return actions
```

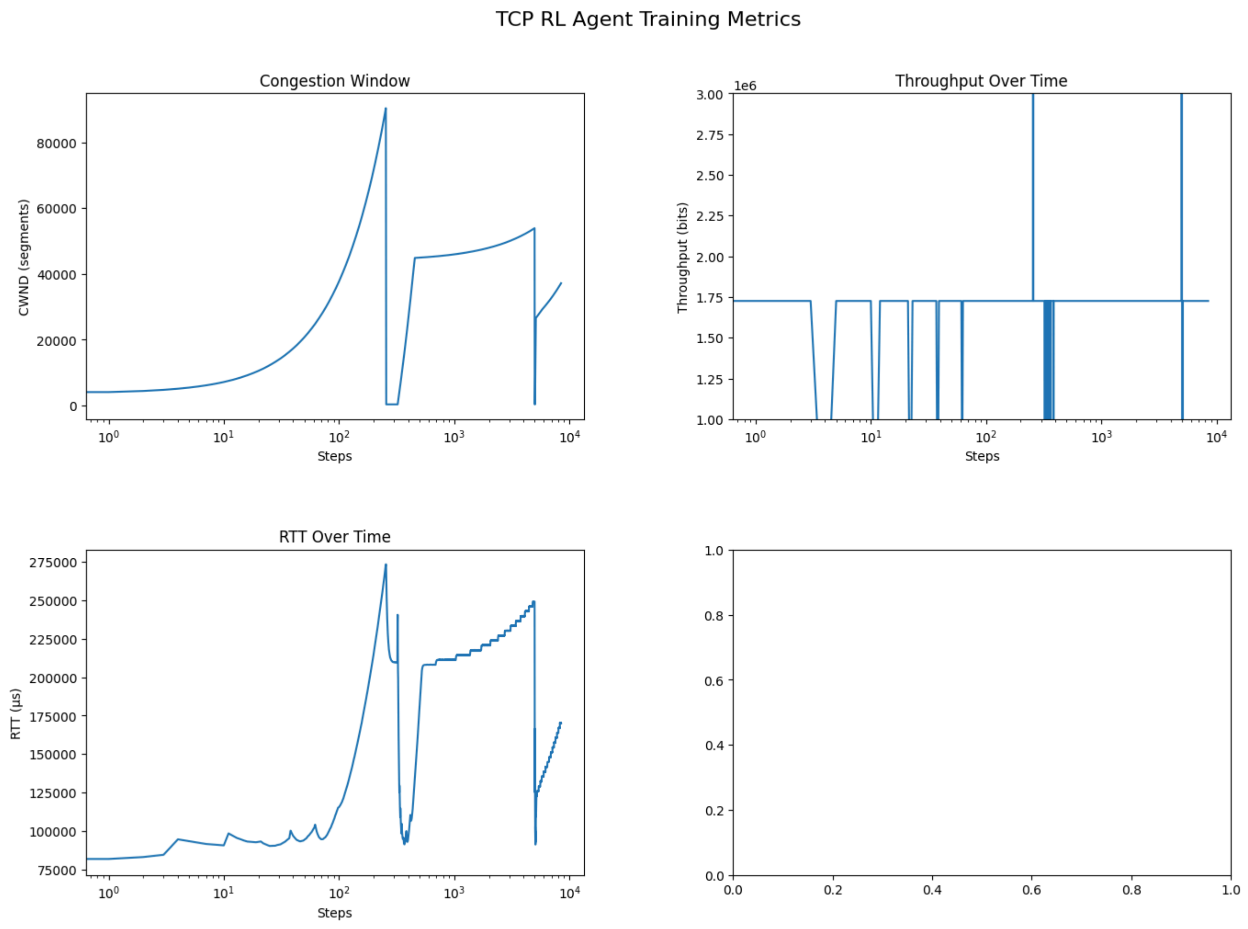
Here the code shared above, implements tcp tahoe in ns3 simulator.

Code Observation:

Implements TCP Tahoe logic in an RL setting using congestion window (CWND) and slow start threshold (ssThresh).

On **packet loss** (`caEvent == 2`), resets `cWnd` and `ssThresh` to one segment, following Tahoe's behavior.

In **slow start phase** (`cWnd < ssThresh`), increase `cWnd` exponentially.
In **congestion avoidance** (`cWnd >= ssThresh`), increases `cWnd` linearly based on additive increase formula.
Final `ssThresh` update uses a standard formula to halve `bytesInFlight`.



Graph observations:

Congestion Window (CWND):

- The CWND graph shows exponential growth initially, consistent with TCP Tahoe's **slow start** phase.
- Periodic sharp drops in CWND indicate **packet loss events**, after which CWND resets to 1 and enters the **congestion avoidance** phase.
- This sawtooth pattern is a characteristic behavior of TCP Tahoe.

Throughput Over Time:

- The throughput is relatively stable at around 1.75 Mbps, but there are occasional spikes and drops.
- These drops correspond to the times when CWND resets due to packet loss.
- The irregularities may suggest either high network variability

RTT (Round Trip Time) Over Time:

- RTT increases gradually with CWND growth, which is expected as higher CWND can lead to queue buildup.
- Significant spikes in RTT just before CWND drops indicate **congestion buildup**.
- After CWND resets, RTT drops sharply, showing network congestion has eased.

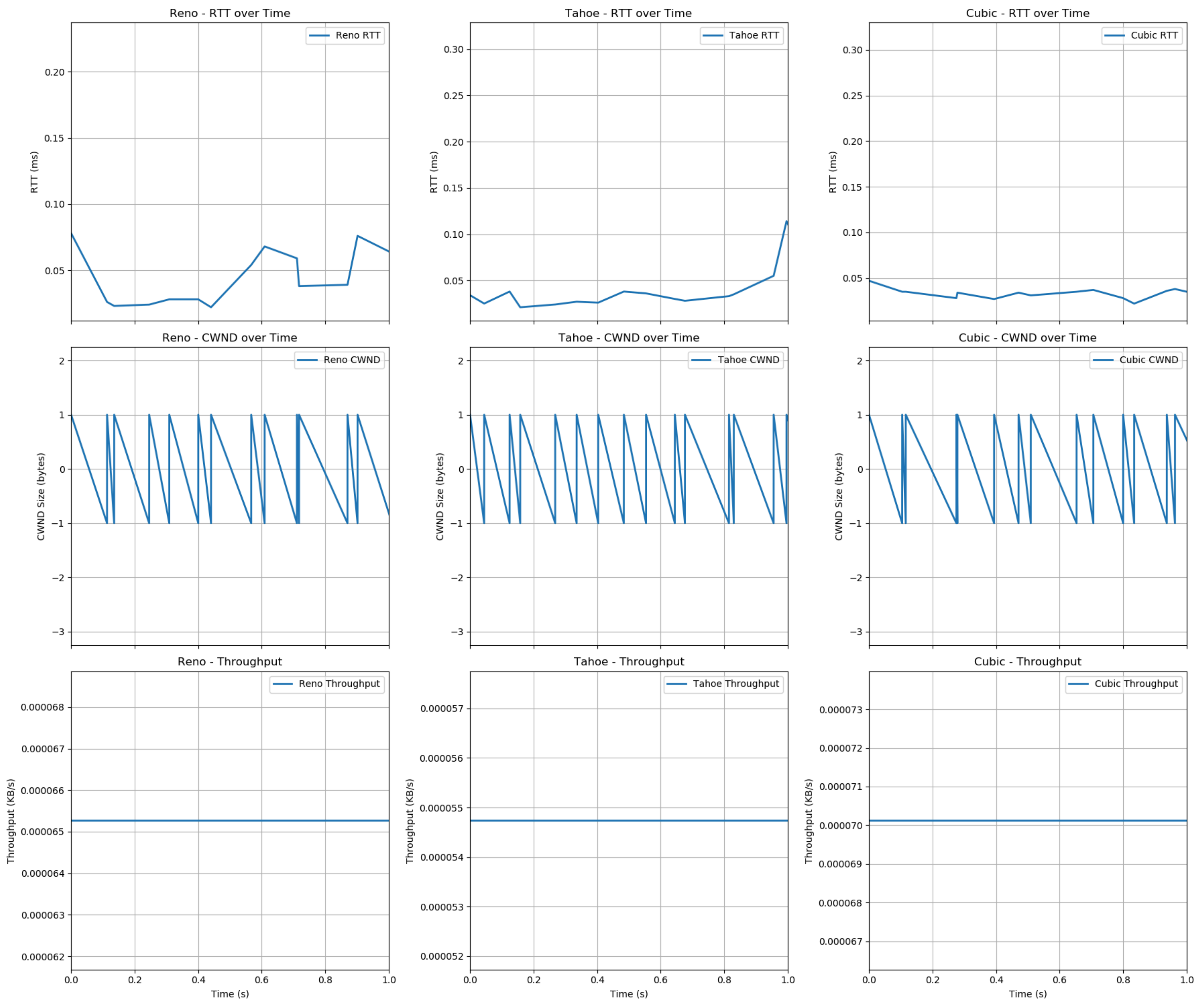
MahiMahi on Reno,Cubic,Tahoe

Mahimahi is a toolkit that records HTTP traffic from web applications and replays it under emulated network conditions. The goal was to compare the performance of different TCP congestion control algorithms — **Tahoe**, **Reno**, and **Cubic** — under identical realtime network conditions using the **Mahimahi** network emulation framework. Wireshark was used to capture packet traces and analyze network behavior.

One of the codes of cubic is given which captures the packets of network.

```
ALGO="cubic"
mkdir -p traces
# Set congestion control
echo "Setting TCP congestion control to $ALGO"
sudo sysctl -w net.ipv4.tcp_congestion_control="$ALGO"
# Capture duration
DURATION=15
# Start some traffic (replace with your real traffic pattern if needed)
mm-delay $DELAY_MS -- curl -s -o /dev/null http://amazon.com
echo "Starting tcpdump for $DURATION seconds..."
# sudo timeout $DURATION tcpdump -i lo -w "traces/${ALGO}.pcap" tcp
sudo timeout $DURATION tcpdump -s 96 -i lo -w "traces/${ALGO}.pcap" tcp



echo "Capture saved to traces/${ALGO}.pcap"
```

Graph observations:



1. RTT (Round-Trip Time) Over Time (Top Row)

- **Reno**: Consistently low RTT, relatively stable.
- **Tahoe**: Shows highest RTT as compared to others. Shows a lot of fluctuations
- **Cubic**: Shows less fluctuations and has relatively lesser RTT

Observation: Cubic shows lesser RTT as compared to the other two algorithms.

2. CWND (Congestion Window) Over Time (Middle Row)

- **All Three (Reno, Tahoe, Cubic)**: Share a **sawtooth pattern** — periodic increases followed by sharp drops.
  - This pattern is indicative of **congestion avoidance and packet loss reactions**.
  - All three keep hitting a CWND of just over **1.0 byte** before dropping to nearly 0 — suggesting that all are hitting congestion events regularly.

Observation:

Similar window behaviors suggest the network environment might be tightly constrained, leading all algorithms to behave similarly in terms of congestion response.

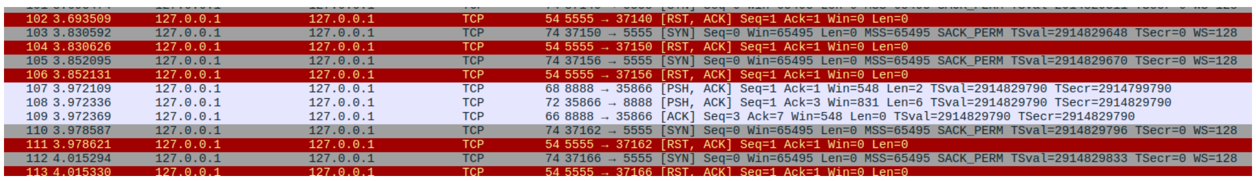
3. Throughput Over Time (Bottom Row)

- **Reno**: Slightly higher throughput (~0.00000065 Kbps).
- **Tahoe**: Moderate throughput (~0.00000055 Kbps).
- **Cubic**: Lowest throughput (~0.00000070 Kbps).

Observation:

Cubic has better throughput as compared to both Reno and Tahoe. Similarly, Reno has better throughput than Tahoe.

The .pcap file for reno is:



```
102 3.693509   127.0.0.1   127.0.0.1   TCP   54 5555 → 37140 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
103 3.830592   127.0.0.1   127.0.0.1   TCP   74 37150 → 5555 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM TSval=2914829648 TSecr=0 WS=128
104 3.830626   127.0.0.1   127.0.0.1   TCP   54 5555 → 37150 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
105 3.852095   127.0.0.1   127.0.0.1   TCP   74 37156 → 5555 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM TSval=2914829670 TSecr=0 WS=128
106 3.852131   127.0.0.1   127.0.0.1   TCP   54 5555 → 37156 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
107 3.972109   127.0.0.1   127.0.0.1   TCP   68 8888 → 35866 [PSH, ACK] Seq=1 Ack=1 Win=548 Len=2 TSval=2914829790 TSecr=2914799790
108 3.972336   127.0.0.1   127.0.0.1   TCP   72 35866 → 8888 [PSH, ACK] Seq=1 Ack=3 Win=831 Len=6 TSval=2914829790 TSecr=2914829790
109 3.972369   127.0.0.1   127.0.0.1   TCP   66 8888 → 35866 [ACK] Seq=3 Ack=7 Win=548 Len=0 TSval=2914829790 TSecr=2914829790
110 3.978587   127.0.0.1   127.0.0.1   TCP   74 37162 → 5555 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM TSval=2914829796 TSecr=0 WS=128
111 3.978621   127.0.0.1   127.0.0.1   TCP   54 5555 → 37162 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
112 4.015294   127.0.0.1   127.0.0.1   TCP   74 37166 → 5555 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM TSval=2914829833 TSecr=0 WS=128
113 4.015330   127.0.0.1   127.0.0.1   TCP   54 5555 → 37166 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
```

This shows the .pcap file which have details of captured file by wireshark and mahimahi.

TCP Cubic

Algorithm Implemented:

Initialization

- β- Controls how much the congestion window is reduced after a loss. ,
  C- Controls how fast the congestion window grows after a loss.
- `tcp_friendliness`, `fast_convergence`

ACK Handling

- If `cwnd ≤ ssthresh`: classic **slow start**, increase `cwnd` by 1.
- Else (**congestion avoidance**):
  - Calculate `target` congestion window by cubic function `wmax + C * (t - K)**3` where **w_max** is the window size where last congestion occurred, K is the time period that the above function takes to increase W to Wmax when there is no further loss event
  - Calculate window growth rate `cnt` i.e number of acks to increase `cWnd` by `segment_size`
    - If `target` > `cWnd`: `cnt = cWnd / (target - cWnd)`
    - Else: `cnt = 100 * cWnd` (very slow growth) It is set to set a large number of required acks to increase `cWnd` by `segment_size` because current `cWnd` is already greater than desired `target`
  - Update **Congestion Window:**
    - If `curr_cwnd_cnt`> `cnt`: Increase `cwnd` by `segment_size`, update `curr_cwnd_cnt` =0
    - Else: keep the same `cWnd`

Packet Loss

- Set `epoch_start = simTime_us, ack_cnt=0`
  `simTime_us` is the current simulation time in microseconds.
  `ack_cnt` is useful for Tcp Friendliness
- Set k by equation
  `K = ((self.w_max * (1 - self.beta)) / self.C) ** (1/3)`

$$K = \sqrt[3]{\frac{W_{max} - cwnd_{epoch}}{C}}$$

- If `fast_convergence` and `cwnd < W_last_max`:
  Implies this time loss occurred at a lower cwnd than last time
  - Set `W_last_max = cwnd * (2 - β)` (accelerates convergence- upper limit for loss in kept lower this time)
- Else: `W_last_max = cwnd`
- Apply **multiplicative decrease**:
  `ssthresh = cwnd = cwnd * (1 - β)`

TCP Friendliness

- Adjust growth based on TCP Reno formula:
  `W_tcp += (3β / (2-β)) * (ack_cnt / cwnd)`
- Ensures fairness with Reno flows.
- Reduces `cnt` if `W_tcp > cwnd`.

Core Mechanism

- Uses cubic function for window growth (concave when recovering, convex when exploring bandwidth)

CUBIC uses a **cubic function** for window growth with two phases:

1. **Concave phase**: After congestion, the window grows rapidly toward the previous maximum (*Wmax*), then slows to form a **plateau** near *Wmax*, stabilizing the network and avoiding overshoot.
2. **Convex phase**: Once past *Wmax*, growth starts slowly and accelerates to probe for new bandwidth, ensuring efficient exploration.

This design balances rapid recovery, stable utilization, and careful bandwidth probing.

Fast Convergence

Fast convergence speeds up bandwidth reallocation when network conditions change:

- When congestion occurs and cwnd < Wlast_max, algorithm assumes network capacity reduced
- Adjusts Wlast_max to cwnd × (2-β)/2 instead of just cwnd
- Allows existing flows to release bandwidth more quickly for new flows

TCP Friendliness

TCP friendliness ensures fair coexistence with standard TCP:

- Calculates what standard TCP would achieve (Wtcp)
- Uses whichever growth rate is more aggressive
- In networks with short RTTs where standard TCP performs well, CUBIC matches its behavior, In high-BDP networks, reverts to cubic growth for better performance

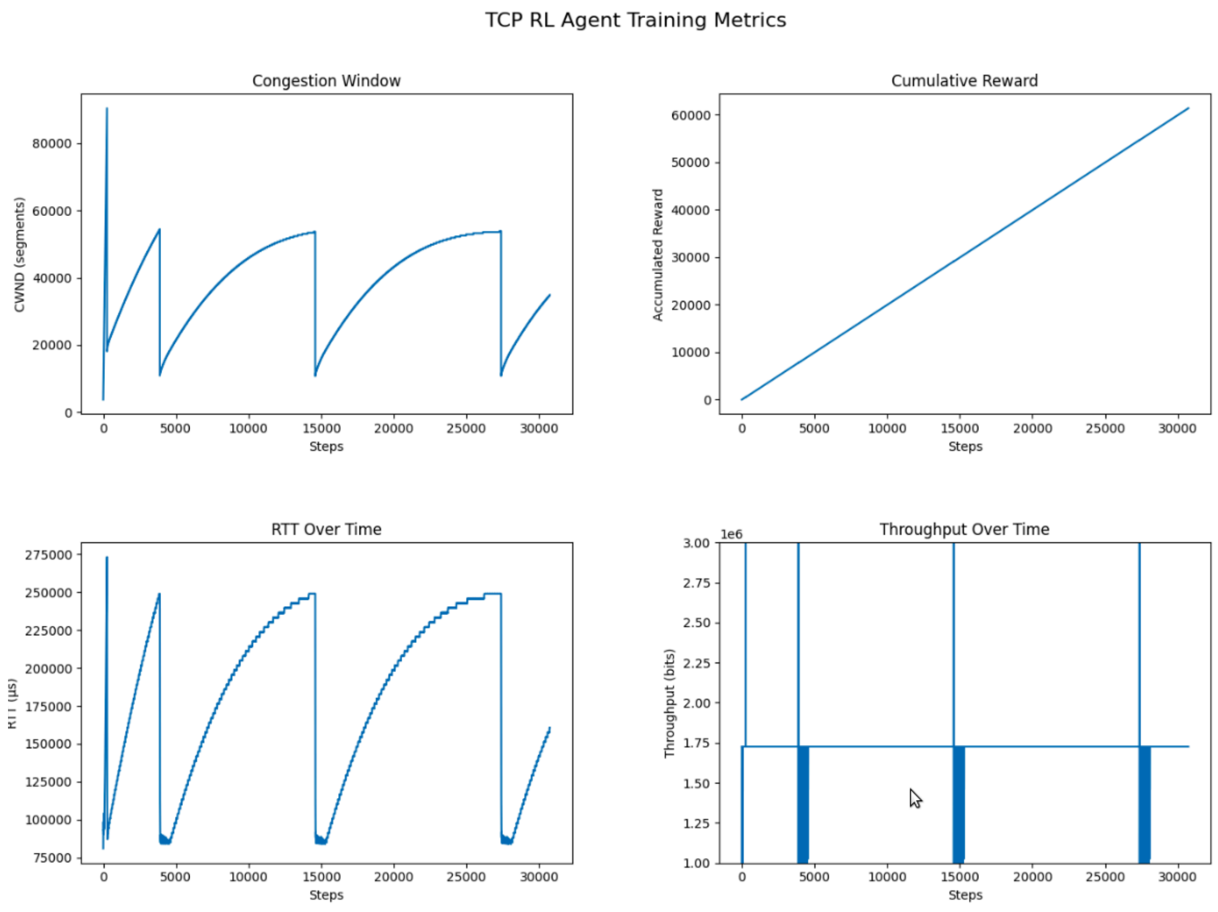
Window Adjustment

- During slow start: increments cwnd by 1 per ACK
- During congestion avoidance: uses cubic function to determine cnt value

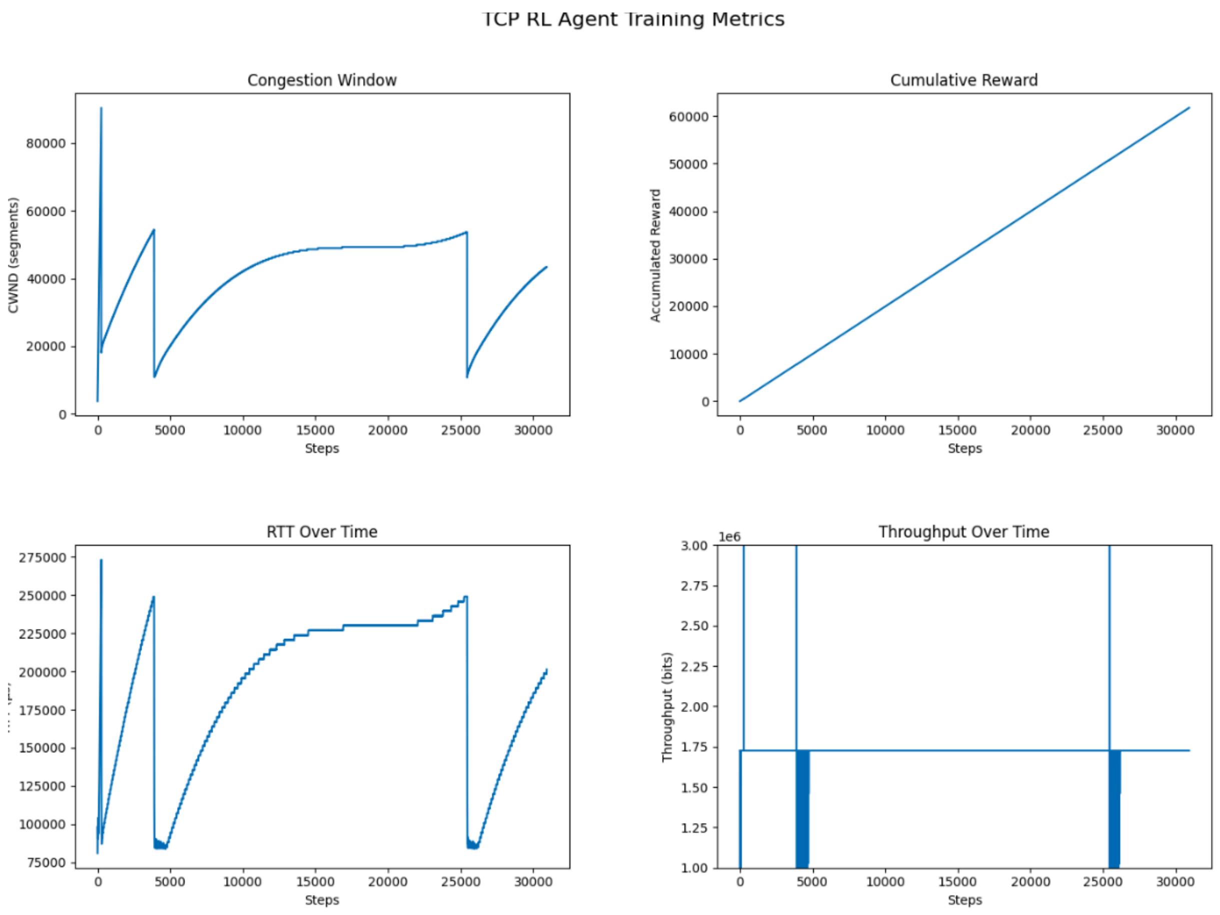
- Reduces window by factor β (0.2) on packet loss

Results

Without Fast Convergence

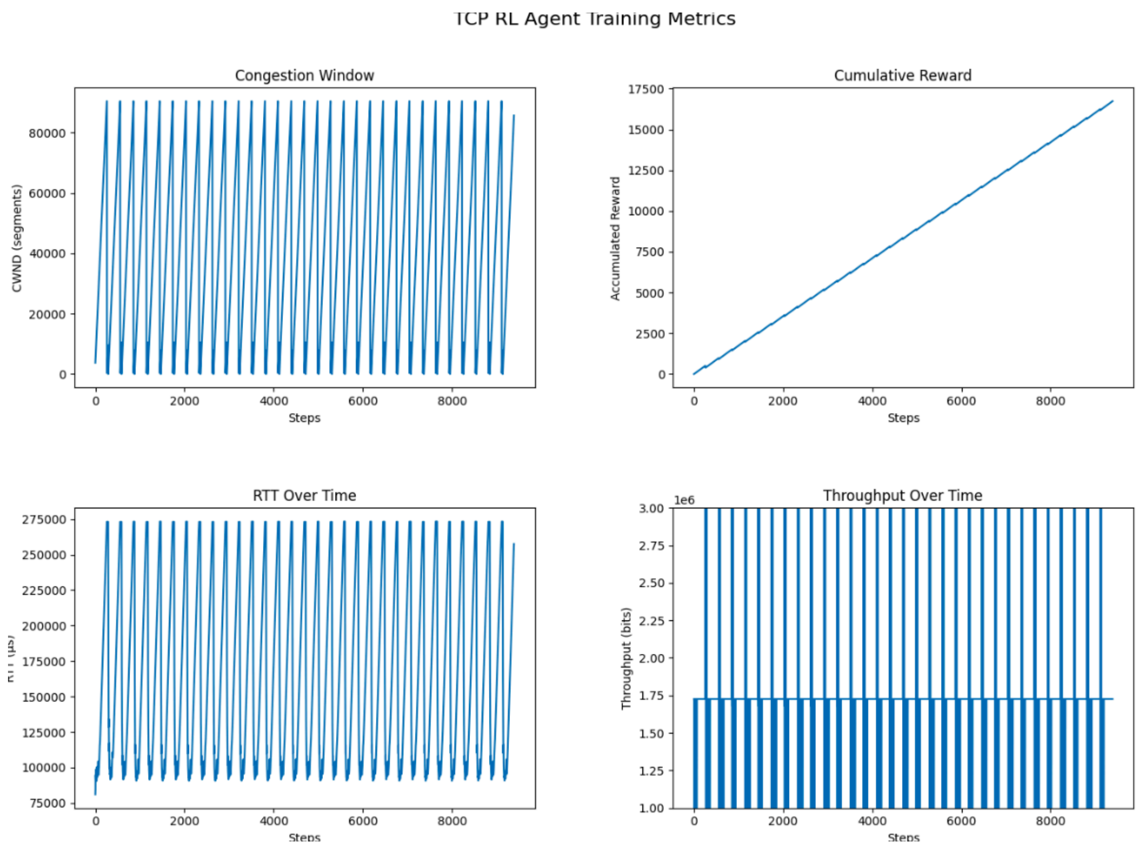


With Fast Convergence



There are more number of loss events in Algorithm implemented without fast convergence, adding fast convergence helps flows to release bandwidth more quickly for new flows

With TCP Friendliness



Here Congestion window fluctuates more as it adapts to Tcp Reno Behaviour

Reinforcement Learning

In reinforcement learning for TCP congestion control, predicting and responding to packet loss is vital. Loss prediction allows the agent to adaptively adjust the congestion window (`cWnd`) before the network experiences performance degradation due to congestion. This implementation uses a Deep Q-Network (DQN)-based approach to indirectly infer packet loss based on the observed reward, RTT, and throughput patterns, instead of explicit packet loss signals.

The logic integrates:

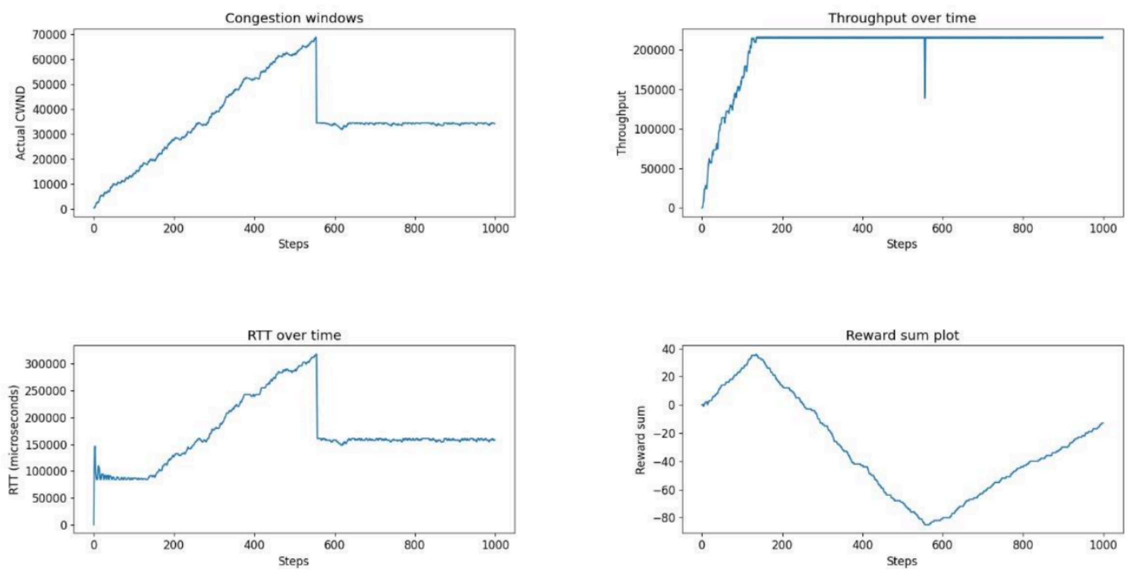
- A neural network to approximate Q-values.
- Epsilon-greedy strategy for exploration vs. exploitation.
- Reward feedback that encapsulates network performance—indirectly indicating losses.
- Adjustments in `cWnd` based on predicted Q-values or random exploration.

Loss (or congestion) is not predicted directly as a binary outcome but is inferred by the change in throughput, RTT, and reward signal—which drops during congestion episodes.

Algorithm

- **Predict Q-values** for all possible actions using the current state as input to the neural network.
- **Select the action** with the highest predicted Q-value (greedy action selection).
- **Execute the selected action** in the environment.
- **Observe the reward** received and the next state resulting from the action.
- **Update the neural network** by using the observed reward and the maximum Q-value of the next state to improve its predictions.

- **Repeat the process** so the network gradually learns which action is best for each state.



Here cwnd is stabilised eventually to select a cwnd which will not push network to congestion and RTT would be stabilised along with throughput being

Loss Predictor

Algorithm

The loss predictor is a machine learning component designed to forecast whether the next packet in transmission is likely to be lost, based on current network conditions. A **Random Classifier** is trained using simulation data derived from CUBIC TCP behavior. The **input features** for training include key network metrics such as:

- `ssThresh` (slow start threshold)
- `cWnd` (congestion window)
- `segmentsAcked`
- `segmentSize`
- `bytesInFlight`
- `caState` (congestion avoidance state)
- `caEvent` (congestion avoidance event)

The **target label** for training is binary—indicating whether the next packet was lost or not. This allows the model to learn patterns in network behavior that typically precede packet loss. Once trained, the model can generalize to unseen network states and provide real-time predictions during TCP operation.

Integration in CUBIC Algorithm

1. On each ACK or sending decision:

- Gather current network state
- Pass features to the trained random classifer model
- Receive **loss probability prediction**

2. Decision Logic:

- If **loss probability > threshold**:

        - Decrease `cWnd` by `1` proactively
    - Else:
        - Proceed with **normal CUBIC window update**

Current Limitations

This kind of random classifier doesn’t perform good if topology is changed, as it has been trained of states and loss predictions of a specific topology.

Further Explorations:

TCP Congestion Control with Reinforcement Learning: A Hybrid Approach

Hybrid TCP Cubic BBR Congestion Control Algorithm

Overview:

Implementation of a hybrid congestion control algorithm that combines the strengths of TCP CUBIC, delay-based control, and bandwidth-aware mechanisms, designed within the NS-3 simulation environment using the tcp event based reinforcement learning interface. The objective is to explore an adaptive congestion control strategy that is robust under varying network conditions.

Algorithm Design

The custom congestion control algorithm implemented in TcpHybridCubic is a hybrid TCP variant that blends multiple congestion control philosophies to achieve a balance between high throughput and low latency. It integrates the strengths of **CUBIC growth**, **delay sensitivity**, and **BBR-style bandwidth estimation** into a cohesive framework.

1. CUBIC Growth Function:

This component serves as the backbone of the congestion window growth mechanism. It follows the traditional CUBIC model:

- The congestion window (cWnd) increases **non-linearly** as a cubic function of time since the last congestion event.

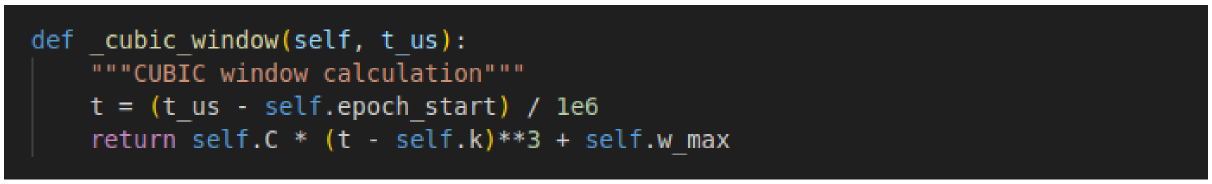
- The formula used is:
  $W(t) = C(t - K)^3 + W_{max}$
  where:

  C is a scaling constant (0.4 in this implementation),

  K is the time origin point derived from previous congestion,

  W_{max} is the congestion window before the last congestion event.

Code:



```
def _cubic_window(self, t_us):
    """CUBIC window calculation"""
    t = (t_us - self.epoch_start) / 1e6
    return self.C * (t - self.k)**3 + self.w_max
```

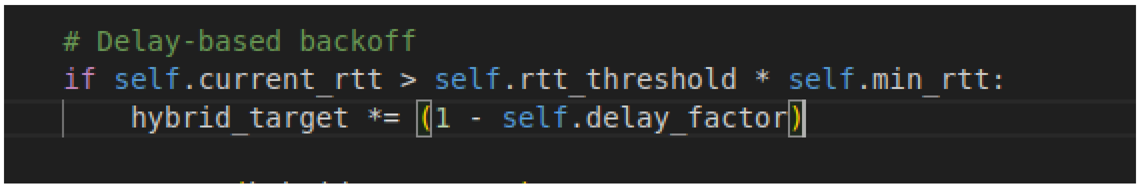
This design allows **aggressive probing** of available bandwidth after recovery, improving throughput in **high Bandwidth-Delay Product (BDP)** environments.

2. Delay-Based Adaptation

To mitigate **bufferbloat** (delays due to large queues), the algorithm continuously tracks RTT and adjusts growth accordingly.

- It keeps the **minimum observed RTT (min_rtt)**, and compares it to the current RTT.
- If the current RTT exceeds the minimum by more than **20%**, it's interpreted as **queue buildup**.
- In this case, it **scales down** the growth target to slow down sending.

Code:



```python
# Delay-based backoff
if self.current_rtt > self.rtt_threshold * self.min_rtt:
    hybrid_target *= (1 - self.delay_factor)
```

This makes the algorithm more **responsive to congestion signals** before packet loss occurs, helping to **preserve low latency**.

3. BBR-Inspired Bandwidth Estimation

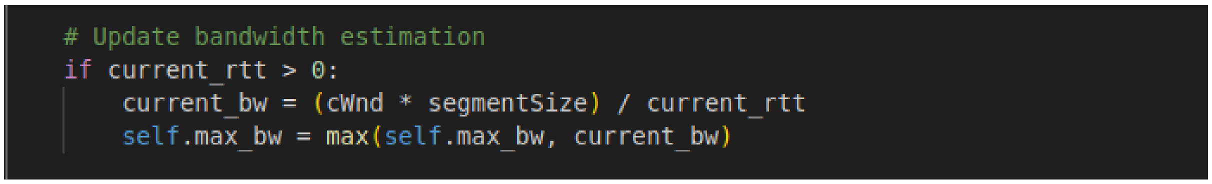
Inspired by the **BBR (Bottleneck Bandwidth and RTT)** algorithm, this component estimates the **maximum sustainable bandwidth** as:

$$Bandwidth \ = \frac{cWnd \cdot SegmentSize}{RTT}$$

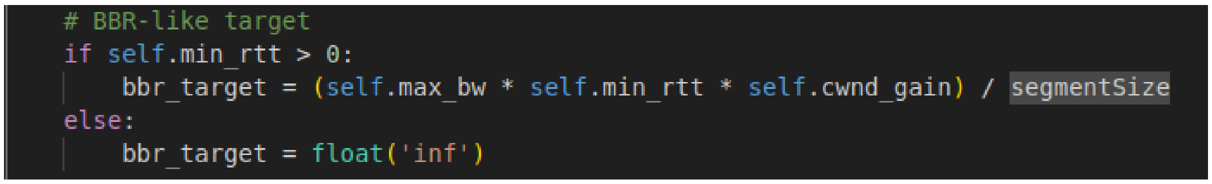
- The maximum observed bandwidth (max_bw) is tracked over time.
- A **bandwidth-delay product (BDP)**-based target is calculated:

$$BDP\ Target \ = \frac{(max\_bw \times min\_rtt \times cwnd\_gain)}{segment\ size})$$

Code:



```python
# Update bandwidth estimation
if current_rtt > 0:
    current_bw = (cWnd * segmentSize) / current_rtt
    self.max_bw = max(self.max_bw, current_bw)
```



```python
# BBR-like target
if self.min_rtt > 0:
    bbr_target = (self.max_bw * self.min_rtt * self.cwnd_gain) / segmentSize
else:
    bbr_target = float('inf')
```

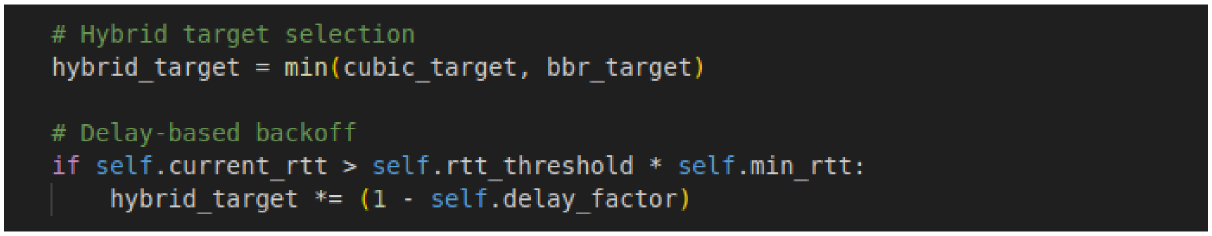
This makes congestion window adaptation more **realistic**, reflecting actual **delivery capacity**, rather than only reacting to loss or delay.

4. **Hybrid Target Window Calculation**

This stage combines all metrics to compute a **hybrid target window**:

- The algorithm selects the **minimum** of the CUBIC and BBR targets to avoid **overestimating bandwidth**.
- If **RTT has spiked**, it further reduces the window using a **delay penalty**.

Code:



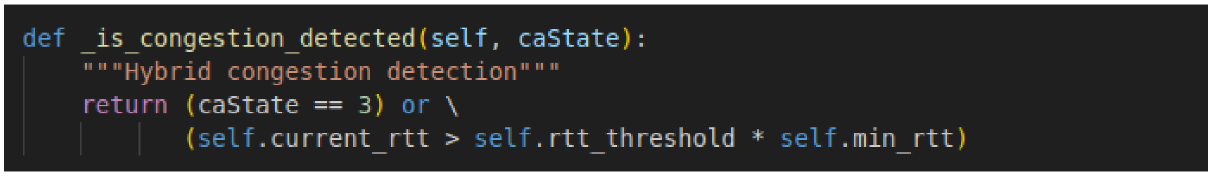
```
# Hybrid target selection
hybrid_target = min(cubic_target, bbr_target)

# Delay-based backoff
if self.current_rtt > self.rtt_threshold * self.min_rtt:
    hybrid_target *= (1 - self.delay_factor)
```

This hybrid mechanism enables **optimistic probing** with a **conservative fallback** during congestion.

5. Congestion Detection and Response

The algorithm determines congestion based on **either loss-based events or excessive RTT**.

Detection:



```
def _is_congestion_detected(self, caState):
    """Hybrid congestion detection"""
    return (caState == 3) or \
           (self.current_rtt > self.rtt_threshold * self.min_rtt)
```

Response:

```
# CUBIC response
self.w_max = cWnd
self.epoch_start = simTime_us
self.k = ((self.w_max * (1 - self.beta)) / self.C) ** (1/3)

# BBR-like response
self.max_bw *= 0.85
self.min_rtt = float('inf')
```

Optionally, if fast_convergence is enabled and cwnd dropped quickly:

```
# Fast convergence
if self.fast_convergence and cWnd < self.w_max:
    self.w_max = cWnd * (2 - self.beta) / 2
```

This allows faster recovery and quicker adaptation to network changes.

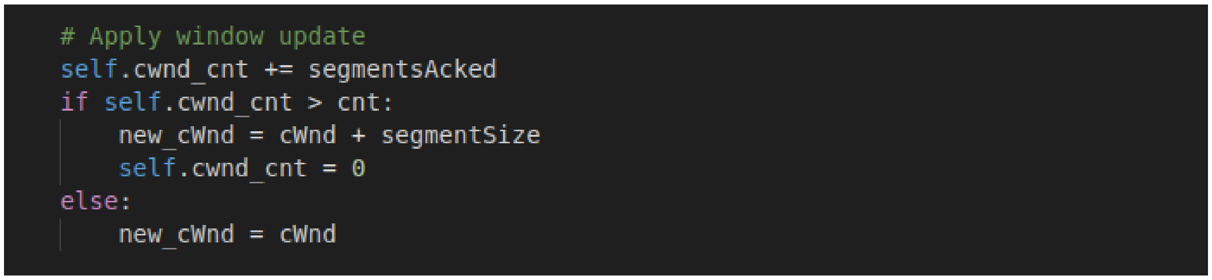
6. Adaptive Growth & cwnd Update

Window growth pacing is carefully tuned using a dynamic counter:

Code:

```
# Window growth calculation
if hybrid_target > cWnd:
    cnt = cWnd / (hybrid_target - cWnd)
else:
    cnt = 100 * cWnd # conservative fallback
```

The actual cwnd increase only occurs when enough ACKs arrive:



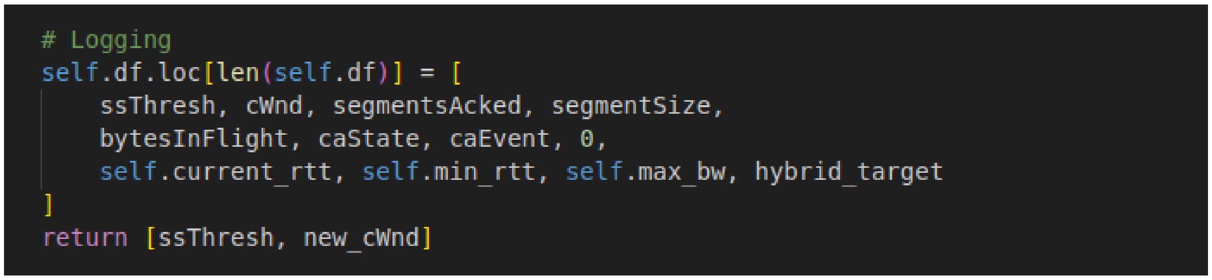
```python
# Apply window update
self.cwnd_cnt += segmentsAcked
if self.cwnd_cnt > cnt:
    new_cWnd = cWnd + segmentSize
    self.cwnd_cnt = 0
else:
    new_cWnd = cWnd
```

This keeps the algorithm **TCP-friendly**, avoiding aggressive cwnd jumps.

7. Logging for Analysis

Every step logs metrics for post-analysis and plotting:

Code:



```python
# Logging
self.df.loc[len(self.df)] = [
    ssThresh, cWnd, segmentsAcked, segmentSize,
    bytesInFlight, caState, caEvent, 0,
    self.current_rtt, self.min_rtt, self.max_bw, hybrid_target
]
return [ssThresh, new_cWnd]
```

This supports **debugging**, **visualization**, and **research evaluation**.

Combined Impact

This hybrid congestion control strategy strikes a **balance** between competing goals:

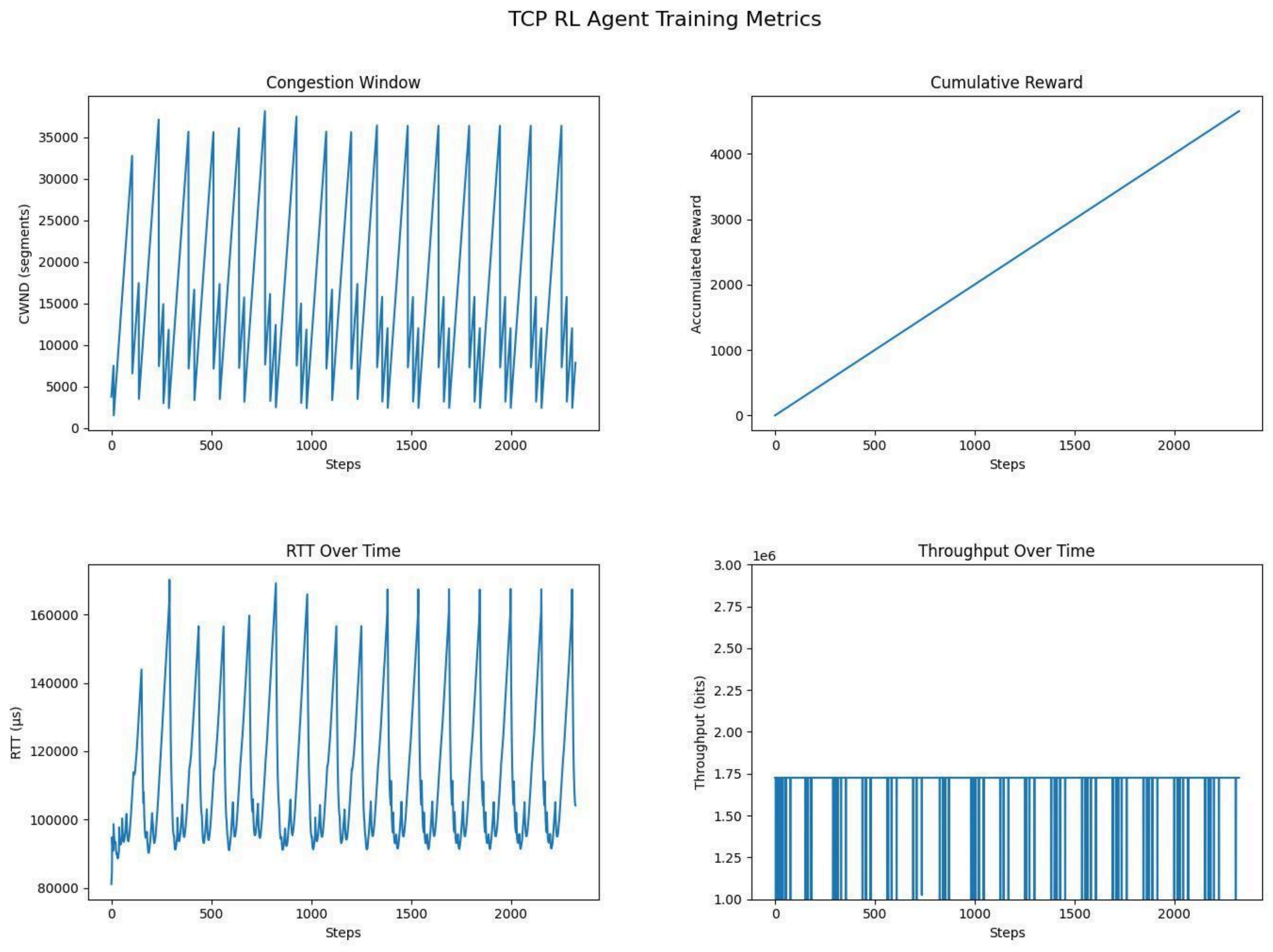
Goal	**Technique Used**
High Throughput	CUBIC’s aggressive non-linear growth

Low Latency	Delay-aware backoff based on RTT trends
Bandwidth Efficiency	BBR-style max bandwidth estimation
Responsiveness	Adaptive hybrid target with fast convergence

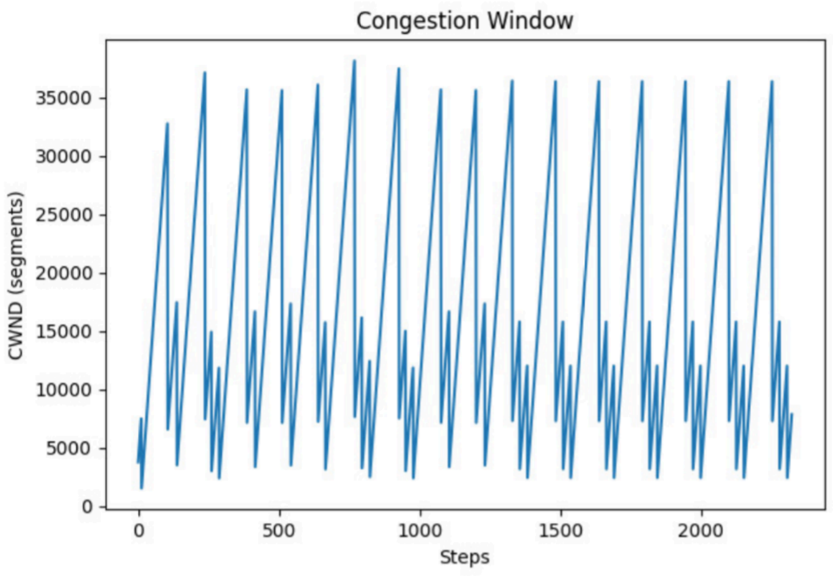
By merging these elements, TcpHybridCubic aims to **outperform traditional TCP variants** in modern, dynamic networks — particularly in wireless, 5G, or high-latency environments.

Training Metrics Analysis

The performance of the RL agent was visualized using the following plots:

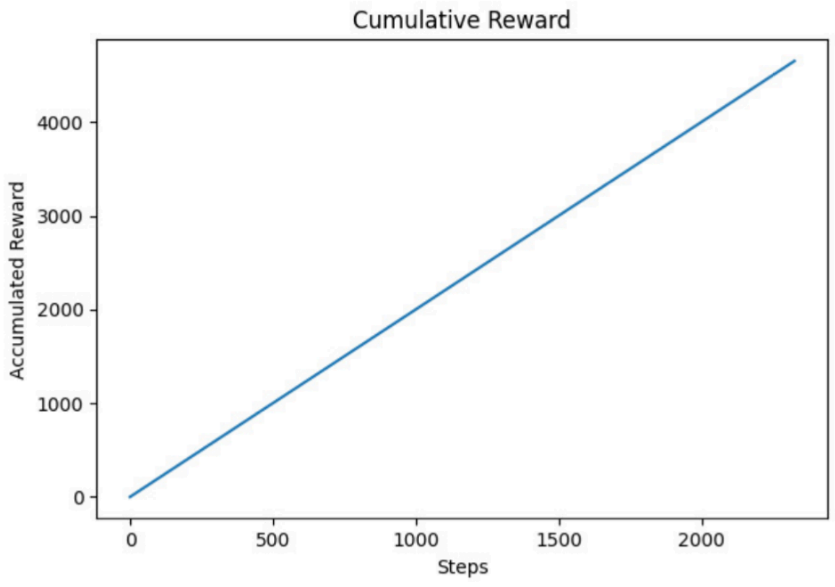


1. Congestion Window Evolution



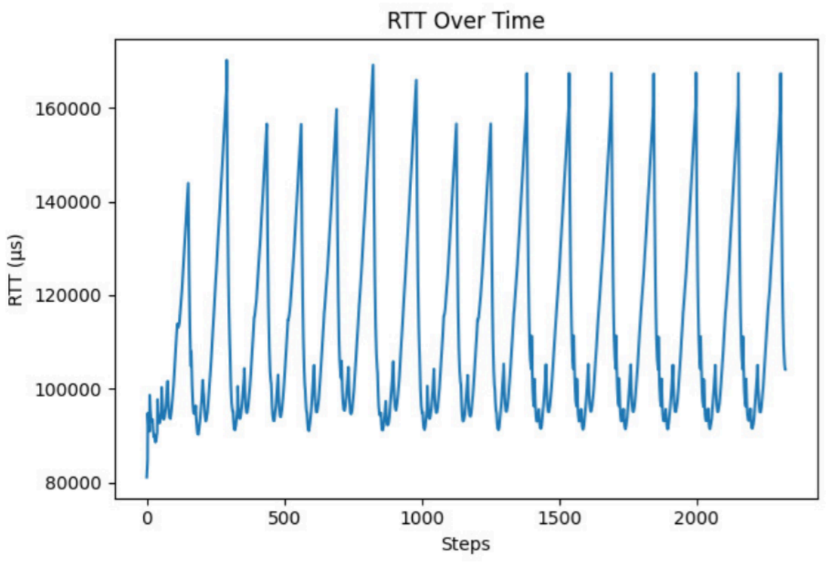
- The congestion window (cWnd) exhibits a sawtooth pattern, typical of congestion avoidance behavior.
- Peaks and drops reflect congestion events followed by recovery via CUBIC probing.
- The magnitude of oscillations demonstrates responsiveness to bandwidth availability and RTT conditions.

2. Cumulative Reward



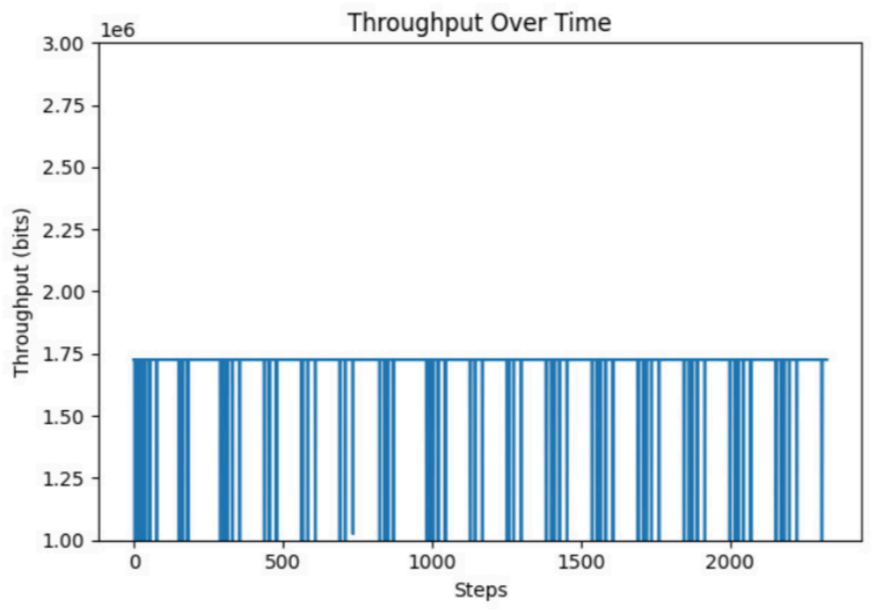
- The reward signal increases steadily, indicating consistent learning and performance improvement by the RL agent.
- The linear growth confirms that the agent's actions are successfully aligned with the environment's performance goals.

3. RTT Over Time



- RTT values fluctuate within a controlled range, with brief spikes indicating network congestion or queuing delays.
- The algorithm responds to these spikes by adapting the window, preventing sustained high-latency conditions.

4. Throughput Over Time



- Throughput remains relatively stable with minor fluctuations, demonstrating efficient bandwidth utilization.

- Periodic drops coincide with congestion responses, followed by swift recovery, showcasing the hybrid algorithm's adaptability.

Future Work

- Incorporate fairness assessment with coexisting TCP variants.
- Evaluate performance over wireless and lossy links.
- Optimize reward shaping to balance latency and throughput more finely.

Hybrid TCP Tahoe RL Congestion Control Algorithm

A hybrid congestion control algorithm is implemented by integrating the traditional TCP Tahoe mechanism with a reinforcement learning-based approach. The system intelligently balances between rule-based decisions and learned behavior through adaptive weighting, depending on current network stability. A Deep Q-Network (DQN) agent is designed with conservative exploration parameters and uses a normalized state representation to ensure stable training. The reward function considers multiple factors such as throughput, RTT, loss, and alignment with Tahoe during unstable phases. Enhancements like outlier-resistant RTT estimation, bounded exploration, and dynamic stability detection contribute to robust and efficient congestion handling. This design aims to preserve TCP reliability while introducing learning-driven adaptability.

Core Components

1. **Adaptive weighting system that dynamically balances traditional and RL approaches:**

```
hybrid_cwnd = int(rl_weight * rl_cwnd + tahoe_weight * tahoe_cwnd)
```

2. **Conservative exploration with reduced epsilon (0.3) and slower decay (0.99)**
3. **Outlier-resistant RTT estimation:**

```
if abs(measured_rtt - self.srtt) > 3 * self.rttvar:
    # Outlier detection with smaller weight for anomalies
    weight = 0.05
```

4. Multi-objective reward function:

```
reward = 2.0 * throughput_reward + 0.8 * stability_reward + 0.5 * rtt_penalty +
          1.0 * loss_penalty + 1.0 * tahoe_alignment_reward
```

5. Network stability detection to govern decision-making:

```
def check_network_stability(self, current_time, cwnd, caEvent):
    """Determine if the network is currently stable"""
    # Track cwnd for stability analysis
    self.cwnd_stability_window.append(cwnd)

    # Check for recent packet loss
    if caEvent == 2:  # CA_EVENT_LOSS
        self.last_loss_time = current_time
        self.stable_network = False
        return False

    # If loss was recent (within last 2 seconds), consider network unstable
    if current_time - self.last_loss_time < 2.0:
        self.stable_network = False
        return False

    # Check cwnd stability if we have enough history
    if len(self.cwnd_stability_window) >= 3:
        cwnd_values = list(self.cwnd_stability_window)
        cwnd_variance = np.var(cwnd_values) / (np.mean(cwnd_values) + 1e-6)  # Normali

        # High variance indicates instability
        if cwnd_variance > 0.2:
            self.stable_network = False
            return False

    # Network appears stable
    self.stable_network = True
```

6. **Bounded exploration that limits random actions to sensible ranges near Tahoe's recommendations**

```
def act(self, state, tahoe_action_idx):
    """Choose action using dynamic epsilon-greedy policy that favors TCP Tahoe in unst
    # Check if rewards are stable or positive
    avg_reward = np.mean(self.reward_history) if self.reward_history else 0

    # Default to TCP Tahoe-like actions when rewards are negative or unstable
    if avg_reward < 0 and len(self.reward_history) >= 5:
        # 80% chance to follow Tahoe in bad conditions
        if np.random.rand() < 0.8:
            return tahoe_action_idx

    # Standard epsilon-greedy exploration
    if np.random.rand() <= self.epsilon:
        # Even during exploration, bias toward actions close to Tahoe's action
        # This creates a bounded exploration around the Tahoe action
        action_range = max(1, self.action_size // 3)  # Limit action range
        lower_bound = max(0, tahoe_action_idx - action_range)
        upper_bound = min(self.action_size - 1, tahoe_action_idx + action_range)
        return random.randint(lower_bound, upper_bound)

    # Let the model decide
    act_values = self.model.predict(state)
    return np.argmax(act_values[0])
```

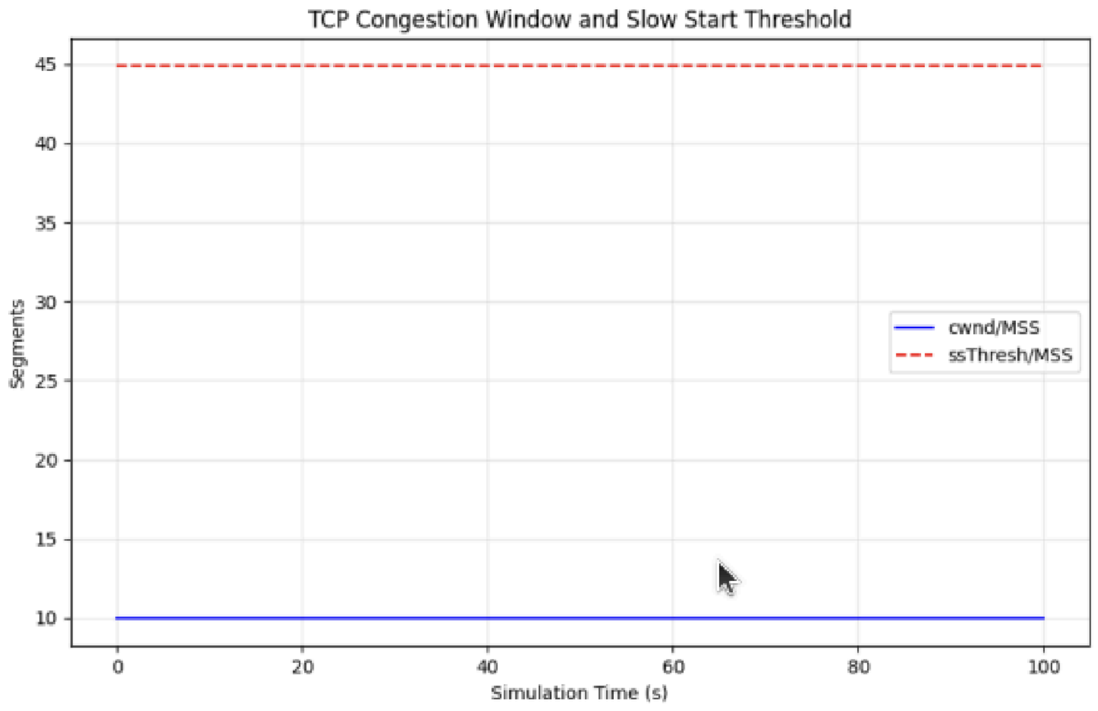
Summary:

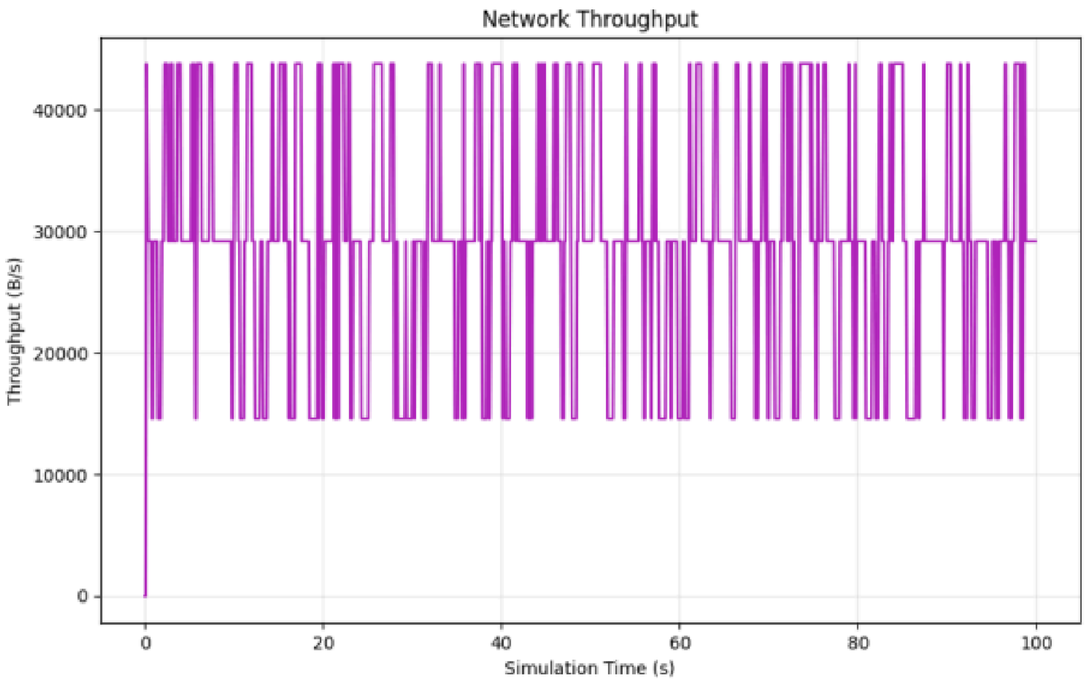
This algorithm blends the reliability of TCP Tahoe with the adaptability of a reinforcement learning (RL) agent. It uses a dual-decision system, dynamically weighting Tahoe and RL based on network stability. In unstable or loss-heavy conditions, it relies more on Tahoe for safety, while in stable conditions, it allows RL to optimize congestion control.

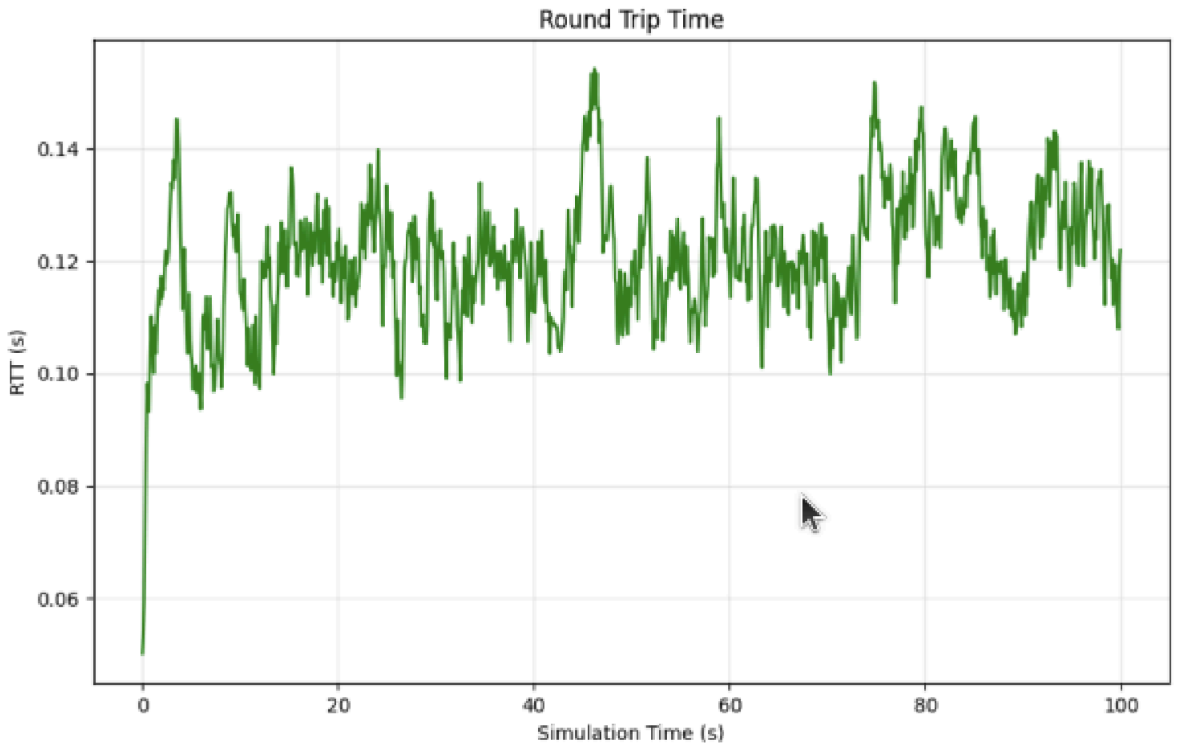
The RL agent observes a 6D state (including cwnd, RTT, and congestion state) and chooses from 3 actions (decrease/maintain/increase cwnd) using a modified epsilon-greedy policy. The reward function prioritizes throughput while penalizing instability, high RTT, and packet loss.

Observations:

Though this Hybrid model is expected to give better results than the original Tahoe implemented above there are some shortcomings.







Possible Causes for Hybrid Implementation Underperformance

1. Reward function miscalibration:

The reward function may be overvaluing throughput (weighted at 2.0) relative to stability (0.8) and RTT penalties (0.5)

```python
reward = (
    2.0 * throughput_reward +  # Higher weight on throughput
    0.8 * stability_reward +   # Moderate weight on stability
    0.5 * rtt_penalty +        # Lower weight on RTT
    1.0 * loss_penalty +       # Moderate weight on loss
    1.0 * tahoe_alignment_reward  # High weight on Tahoe alignment when unstable
)
```

2. Dynamic weighting ineffectiveness:

The dynamic weighting mechanism may not be responding fast enough to network conditions

```
if not self.stable_network or avg_reward < 0:
    tahoe_weight = 0.9  # Higher reliance on Tahoe in unstable conditions
    rl_weight = 0.1
else:
    tahoe_weight = self.tahoe_weight  # Normal weighting in stable conditions
    rl_weight = self.rl_weight
```

3. Network stability detection sensitivity:

The stability detection mechanism might not be correctly identifying unstable periods

4. Model architecture:

Model architecture might fail to learn the network dynamics.

Future Scope

1. Stronger Stability Penalties:

The reward function can be refined to penalize RTT increases and CWND fluctuations more significantly, encouraging smoother behavior.

2. Faster Instability Response:

The algorithm can be made more responsive to unstable network conditions by increasing the Tahoe weighting more aggressively upon instability detection.

3. Enhanced RL Architecture:

Utilizing more advanced models such as RNNs or LSTMs would enable the RL agent to better capture and leverage temporal patterns in the network behavior.

4. **Simulated Pre-Training:**

   Pre-training the RL agent in a simulated environment before real-world deployment can reduce initial instability and accelerate performance convergence.

Analysis:

Under varying network conditions, **Cubic** consistently delivered the best performance, with **higher throughput** and **lower RTT**. **Reno** performed similarly in terms of throughput but had slightly higher RTT. **Tahoe**, on the other hand, exhibited the **highest RTT** and overall **lower performance**, making it less suitable for dynamic environments.

In the **Hybrid model**, we integrated **Reinforcement Learning with TCP Tahoe**, using adaptive weighting based on network stability. This approach led to an improvement in **RTT**, though **throughput gains were limited**, possibly due to the unpredictable nature of the network.

For the **Loss Predictor**, we used a **random classifier** to anticipate upcoming packet losses. If a loss was predicted, the congestion window was proactively reduced. While experimental, this method shows potential for smarter congestion control strategies.

References:

1. Introductory Article: https://par.nsf.gov/servlets/purl/10179370
2. For Reinforcement Learning: https://github.com/20kaushik02/TCP-RL
3. Aurora: https://arxiv.org/pdf/1810.03259
   https://github.com/PCCproject/PCC-RL
4. Random Forest Based Loss Predictor:
   https://dl.acm.org/doi/pdf/10.1145/3229543.3229550
5. TCP Cubic:
   https://www.cs.princeton.edu/courses/archive/fall16/cos561/papers/Cubic08.pdf

https://www.rfc-editor.org/rfc/rfc9438.html#name-constants-of-interest

https://github.com/nikhil286/Analyzing-TCP-NewReno-Veno-Westwood-BIC-CUBIC-using-Network-simulator-NS3/blob/master/TcpCubic.c
https://elixir.bootlin.com/linux/v4.4/source/net/ipv4/tcp_cubic.c

6. Mahimahi: **http://mahimahi.mit.edu/**
   https://squidarth.com/demonstrating-congestion-control