ReLoC

Reinforcing Low Congestion

Computer Engineering - Spring 2024 - CE-NDS10 - 1047

Master's thesis

AALBORG UNIVERSITY
STUDENT REPORT

Study board of Electronics and IT
Fredrik Bajers Vej 7A
9000 Aalborg
http://www.tnb.aau.dk

Title:

ReLoC - Reinforcing Low Congestion

Project:

Master's thesis

Period:

February 2024 - June 2024

Group:

CE-NDS10 - Group number 1047

Members:

Jakob Lund Jensen
Peter Rudbæk Valentinussen

Supervisors:

Tatiana Kozlova Madsen
Mathias Drekjær Thorsager

Pages: 115
Handed in May 31, 2024

Abstract:

This project sets out to explore the possibilities of applying deep reinforcement learning to TCP congestion control. Throughout the project, a novel deep reinforcement learning-based congestion control algorithm was designed and implemented called Reinforcing Lower Congestion (ReLoC). Furthermore, three different variations of ReLoC were created and tested in a variety of network scenarios.
Four policies were created for the three variations of ReLoC. These four policies were the results of training in four different environments. The first policy was trained using a network environment that would always follow the same pattern of changing available capacity without any other TCP clients communicating over the channel. The second policy was trained using a network environment that would switch to random levels of available capacity, but still without any other TCP clients communicating over the channel. The final two policies were trained using the same network environments as the two prior respectively, but with another TCP client communicating over the channel.
Through testing it was shown that the ReLoC variants were able to outperform commonly used congestion control algorithms when operating alone on the channel, however, it would underperform when another TCP client would be utilizing the channel as well. Various suggestions for further improvements have been presented which may help ReLoC being able to perform well in scenarios where it is sharing the channel with another TCP client.

Preface

Reading guide
This report is split into three parts: analysis, system design and implementation, and evaluation. Furthermore, an appendix can be found after the bibliography.

Throughout the text, certain keywords will be followed by a reference number. These keywords will have a capitalized first letter. This indicates that the reference number is a hyperlink to the referenced content.

Jakob Lund Jensen
jjens19@student.aau.dk

Peter Rudbæk Valentinussen
pvalen19@student.aau.dk

Contents

Part I

Analysis

1 | Introduction

Data transfer over the internet has become commonplace for a majority of people, especially in the West, whether it is for communication, streaming, gaming, or other purposes. Over the past 10 years, internet accessibility worldwide has climbed linearly from 35% of the world's population to 67% [1]. As more people gain access to the internet yearly and the data-transferring demand of individuals increases, a need for better infrastructure is required. In recent years, Cellular networks saw the widespread launch of 5^{th} generation technology, which was a big upgrade for Cellular users. Similar upgrades are constantly being developed for the infrastructure across internet communication and telecommunication as a whole. Although many upgrades can be made to the physical infrastructure, there is also room for upgrades within the digital infrastructure.

A common protocol used for sending data is the Transmission Control Protocol (TCP). TCP ensures that all sent packets are received and ordered correctly. To ensure that packets are received, they resend packets they assume have been dropped. This, however, means that if a lot of packets are dropped, a lot of redundant packets are sent. Dropped packets are usually a result of a node sending more data than can fit through the bandwidth, leading to congesting the network. For this reason, users commonly use a Congestion Control Algorithm (CCA) for congestion avoidance. Such algorithms help increase the efficiency of TCP, giving its users higher throughput simply by sending less data. However, these algorithms have commonly been developed using hand-crafted heuristics. These CCAs are known as rule-based CCAs. Rule-based CCAs are made for general usage so that better TCP performance can be achieved regardless of the network structure. The problem with such CCAs is that they are usually not able to use the available capacity optimally.

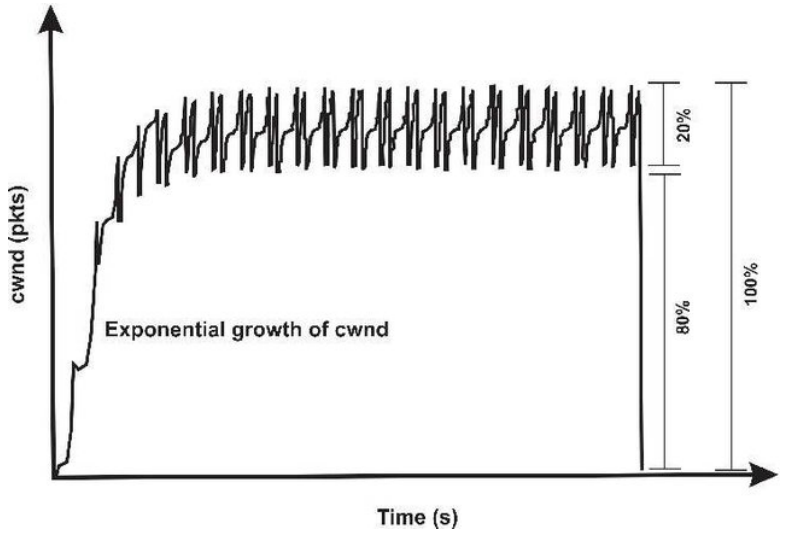


Figure 1.1. TCP CUBIC Congestion Window Growth Behavior [2]

TCP Cubic is a commonly used CCA. While it is fairly effective, it is not able to fully utilize all of the available capacity. In Figure 1.1, it can be seen how TCP Cubic constantly changes its congestion window (cWnd), instead of finding an optimal value. This type of behavior can lead to a loss in performance and is commonly seen in loss-based CCAs, such as TCP Cubic.

In recent years, the development of CCAs applying Machine Learning (ML) to replace hand-crafted heuristics has become an increasingly popular topic, as some see this as the next step in improving CCAs. As of the completion of this project, ML-based CCAs are still being developed as a proof of concept, with no implementations made for common usage yet. However, a variety of proof of concept implementations have been created and generally shown improvements over the classical rule-based CCAs [3]. One of the overlaying problems of creating ML-based CCAs is that they easily become specialists in the networks they are trained upon but can be difficult to generalize for general-purpose usage.

1.1 Initial problem formulation

How can machine learning be applied to TCP congestion control to improve its performance compared to rule-based TCP congestion control algorithms?

2 | Problem Analysis

This chapter seeks to explore existing CCAs, both rule-based and machine learning-based variants, and their available techniques, as well as their deficiencies. Moreover, the available machine learning methodologies and their tools will be explored for potential applicability.

2.1 TCP congestion control

To reduce congestion around bottlenecks in a network there are two possible solutions. The first is changing the network to lessen the effect of the bottleneck or removing it entirely. This solution only works if an alternative route is available on a network under your control. This solution is therefore insufficient in many scenarios, especially on the internet. The second option is to reduce the amount of sent data. This would lead to fewer packets being dropped due to queue overflow and should theoretically lead to a net increase in successfully transmitted data. This solution is the main idea behind TCP CCAs. [4]

The design problems of CCAs lie in sending as much data as possible without causing congestion. The optimal amount of data to have in-flight to fully utilize the available capacity is equal to the bandwidth-delay product (BDP). The BDP is, as its name implies, the product of the available bandwidth of the connection and the delay between sending a packet and receiving a response. This delay is also known as the round-trip time (RTT). While this information sounds simple to acquire, the sender is unaware of the state of both its sent packets as well as any intermediate nodes queues. The only indication of congestion is whether an acknowledgement packet (ACK) is successfully received or not. This problem becomes further complex when considering the network potentially changing and other nodes also communicating through the bottleneck. To control the amount of data being sent most CCAs makes use of a window. The window represents the number of unacknowledged bytes or segments that can be in transmission at once. The node will therefore stop sending if the next segment would make the current amount of segments or bytes in-flight exceed the window size. This window is the congestion window (cWnd).

2.1.1 Reactive congestion control algorithms

With the issue of preventing congestion at hand, one of the solutions proposed were reactive CCAs. These algorithms identify when congestion has occurred and respond by lowering the size of their cWnd. An example of this type of CCA is TCP Reno. TCP Reno detects congestion in the following two ways:

1. **Three duplicate ACKs**: When TCP Reno detects three ACK responses with the same sequence number, also called three duplicate ACKs, it assumes a packet was dropped. Take for example three received ACKs all with the sequence number response of 10, this means packet 11 has not arrived and is therefore considered dropped. Since a dropped packet may indicate congestion, this is taken as a sign of congestion.
2. **Timeout**: When no responses are received from the receiver within a predefined time period, a timeout occurs. A timeout can be a result of many packets being dropped and is therefore

considered a sign of heavier congestion than an instance of three duplicate ACKs. The predefined time period that defines when a timeout occurs is dependent on the RTT [4].

TCP Reno is designed around these two congestion indicators and is split into phases with unique roles to handle and/or prevent congestion. These phases can be seen in Figure 2.1. The first phase is called 'slow start'. In this phase, the cWnd starts out with a value of 1 or 2 segments. The cWnd is then used until an ACK is received, where the cWnd then is increased by 1 segment, which leads to the cWnd being doubled every RTT period, as long as no packets are dropped. This process is repeated until a threshold called 'slow start threshold' (ssthresh) is reached or congestion is detected. This phase can be seen in the start of Figure 2.1 until the first 'Threshold' label is reached. If the threshold is reached TCP Reno goes into its second phase, congestion avoidance. In this phase the cWnd is increased, however, instead of being doubled, it will now only be increased by one each RTT period. This allows TCP Reno to slowly probe for more available network capacity. This continues until congestion is detected. Congestion responses are identical for all phases. Whenever congestion occurs the ssthresh is set to be equal to half the value of the current cWnd. This allows TCP Reno to update the ssthresh to be more in line with the current network connection capacity. The ssthresh is utilized if the detected congestion is in the form of a timeout, then the cWnd is reset and TCP Reno enters the slow start phase anew, as seen on transmission 23 in Figure 2.1. If on the other hand, three duplicate ACKs were detected, then TCP Reno performs 'fast recovery'. Fast recovery is the act of setting the cWnd to be equal to the ssthresh value, instead of completely resetting it. This can be seen in transmission 17 in Figure 2.1. This concept of slowly probing by constant small increases but then decreasing quickly by a multiplicative factor is a concept called Additive Increase, Multiplicative Decrease (AIMD).

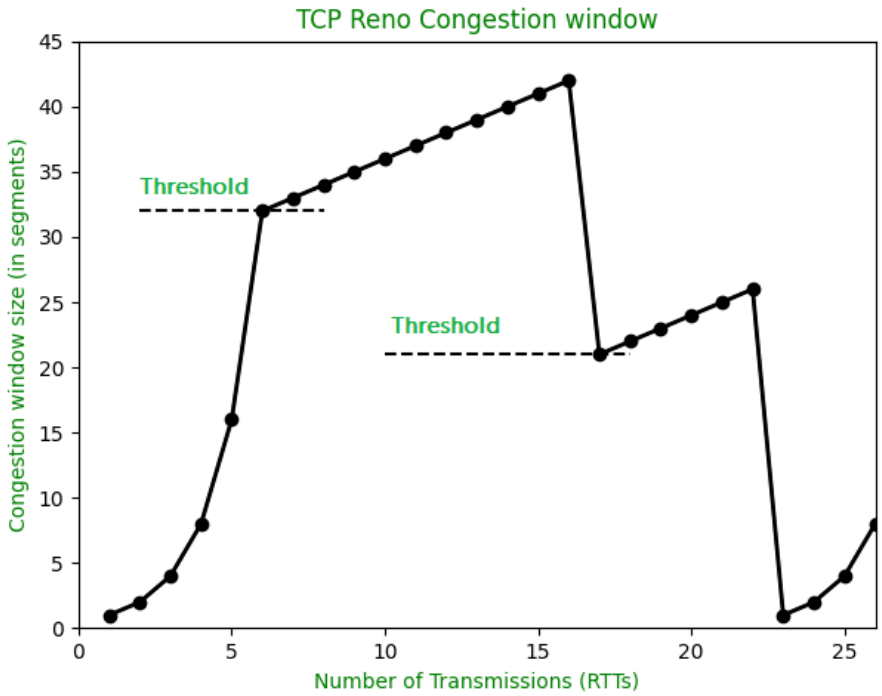


Figure 2.1. Example of cWnd over transmissions measured in RTT [5].

These phases and responses to congestion allow TCP Reno to change the cWnd to utilize the network's link capacity better, as well as adjust to changes in the network capacity. However, TCP Reno is not without its flaws. E.g. TCP Reno performs linear increase during congestion avoidance, thus it may take a significant time for the cWnd to reach the new upper bound. A congestion control algorithm made to excel in solving this task is the CCA called TCP Cubic[6].

TCP Cubic is another reactive CCA, which means that just like TCP Reno it handles congestion by reacting to duplicate ACKs and timeouts by reducing the cWnd. The big difference with TCP Cubic is its congestion avoidance phase. Here TCP Cubic utilizes the shape of a 3rd-degree polynomial as a guideline for its cWnd [4]. This polynomial can be seen in Figure 2.2. Here it is intended for the point of the polynomial with a slope value of 0, where 'a' and 'b' intersect, to correspond to the last known point of congestion so that the cWnd is quickly increased close to the previous value but then slows down. This allows TCP Cubic to reach a higher cWnd faster and also spend more time at that level. Furthermore, afterwards when the 'x' value becomes larger than 'a', as seen in Figure 2.2, then the cWnd starts to increase exponentially. This is to solve the issue of a potentially significantly higher link capacity so as not to waste too much time probing for capacity.

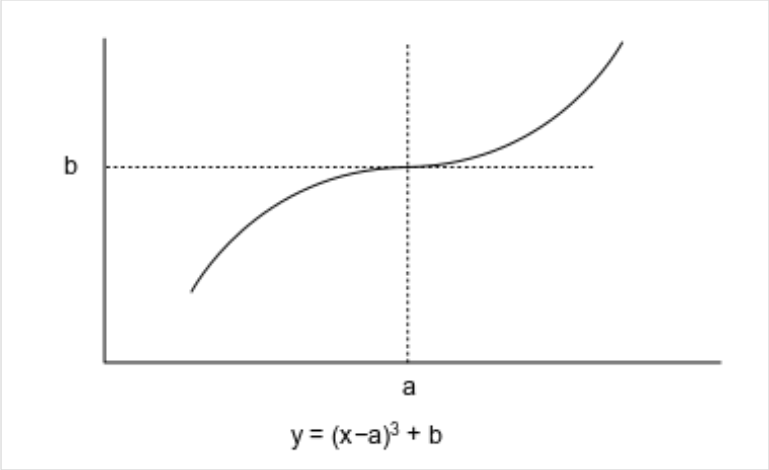


Figure 2.2. 3rd-degree polynomial shape used for controlling cWnd in TCP Cubic [4]

Another difference between TCP Reno and TCP Cubic is the decision of the ssthresh value. With TCP Reno, the value is half the cWnd in the instance congestion is detected, however, in the case of TCP Cubic the value is often set as 70% or 80% of the cWnd value. This difference lessens the immediate reduction in cWnd and therefore increases the overall sending rate under the assumption that the link capacity has not fallen dramatically.

2.1.2 Proactive congestion control algorithms

Another approach for handling congestion is a proactive approach. The concept is to identify when congestion is about to occur and reduce the cWnd beforehand, thus, avoiding packets dropping caused by congestion. This could significantly reduce, if not eliminate, the amount of retransmission needed as a result of congestion. The issue then becomes, how to identify congestion before it occurs.

A congestion control algorithm that attempts to solve this problem is TCP Vegas[7]. Unlike reactive

CCAs, TCP Vegas does not focus on duplicate ACKs and/or timeouts, but rather on the RTT of the ACKs. By looking for variations in RTT from the received ACKs it is possible to detect if the receiver's queue is starting to form a backlog. This works since packets waiting in the queue have a higher RTT value, meaning an increasing average RTT for received ACKs indicates that a queue is forming. Like the reactive CCAs mentioned before, TCP Vegas also makes use of a cWnd to control the number of packets in-flight, this means TCP Vegas can start reducing the cWnd value whenever congestion is imminent until it is stable, and then begin slowly increasing it again.

Fairness
The ability to avoid packet loss due to congestion is such an improvement that a proactive CCA seems to be a generally superior approach, however, while being proactive is effective in a vacuum it has issues when interacting with reactive CCAs. E.g. if one node using TCP Vegas and another using TCP Reno were to share a link, then they would both try to avoid congestion, but since TCP Vegas would detect a build-up in the receiver queue it would start reducing its cWnd before TCP Reno. As a result, TCP Reno would end up utilizing a vast majority of the link capacity. This is the issue of fairness. Two completely fair algorithms using the same link would have an equal split of the available link capacity. This means TCP Vegas cannot work effectively alongside most reactive CCAs.
Fairness, therefore, becomes a big consideration when designing a CCA, as a CCA, which aggressively uses link capacity becomes "unfair", and can be maliciously used by a single selfish actor to claim most of the available link capacity for themselves.

2.1.3 Congestion-based congestion control

While TCP Cubic and TCP Vegas use loss and RTT as congestion indicators respectively, they both have disadvantages that warrant the search for alternative solutions. One attempt at such a solution is another type of CCA called a congestion-based CCA. An example of such a CCA is Bottleneck Bandwidth and RTT (BBR)[8]. [4]

BBR operates by calculating an estimation of the BDP and then matches its in-flight data to this estimation. This calculation is the product of two estimated values, the Bandwidth Estimate (BWE) and the minimum RTT (RTT_{min}). First, the RTT_{min} is the RTT when not applying a load to the network. This is measured by sending individual packets with slight delays and measuring the RTT based on the responses. This value is not updated while normal transmissions are made, but is updated using a special 'PROBE_RTT' mode that performs the measurement. This mode is only entered once every 10 seconds to avoid significant throughput by measuring too often. The BWE is based on the maximum achieved throughput the node has achieved in the last 10 RTT periods. So, when sending packets the amount of data transferred is measured and based on the ACKs received, which is then used as an estimate of throughput for the network connection. With the two measurements made BBR is then able to use the product of the two as a reference for the amount of in-flight data the link can support. BBR then sends at a rate that fully utilizes this capacity, which is unlike the other CCAs mentioned previously that makes use of a cWnd. This is because BBR is a rate-based CCA and therefore sends at a specific rate instead of sending whenever ACKs arrive as is the case of cWnd-based CCAs.
Reductions in the available link capacity will implicitly be adjusted for, as this would cause a drop in measured throughput, which after 10 RTT periods would result in the BWE being lowered. To probe for an increase in available link capacity, BBR increases its sending rate to 1.25 times of the supposed

rate for one RTT period to generate more traffic. During the following RTT period, the rate is set to be 0.75 times the supposed rate, so the receiver buffer can be emptied for the additional packets. The throughput estimate of the first RTT period will be higher if there is more unutilized capacity, therefore the BWE will increase. If the link had no more capacity available, the BWE would not increase.

There are many considerations to make when choosing or creating a CCA. Whether it is loss, variance in RTT, or throughput, a CCA use some indication of congestion to adjust the amount of packets sent. This choice may both affect the effectiveness of the CCA's ability to maximize throughput and its ability to interact fairly with other CCAs. One should also consider whether the CCA should be window-based or rate-based. Furthermore, some CCAs are specifically designed for networks with large amounts of bandwidth, while others may be designed for networks with high loss rates[4].
Currently, full utilization of a network while ensuring complete fairness with other nodes present has not been achieved. This might be a lack of capability for rule-based CCAs, and if so, other more dynamic approaches might be needed.

2.2 Overview of machine learning paradigms

Machine Learning (ML) consists of three basic paradigms, Supervised Learning (SL), Unsupervised Learning (UL), and Reinforcement Learning (RL), along with various other paradigms that are currently gaining popularity [9]. When choosing a paradigm, one must consider the problem they set out to solve, as the different paradigms are specialized for specific problems.

Supervised learning requires training on labeled data. Through training, a model seeks to find patterns within the data that uniquely differentiate each label, allowing the model to appropriately output based on the input. SL is especially good for categorization problems, predictive analysis, anomaly detection, image processing, etc. [10]
Unsupervised learning uses unlabelled data. UL models are usually used for clustering and association problems, and can also be used for preprocessing data. [10]
Reinforcement learning does not commonly use training data, as the training is based on learning experiences from influencing the environment, this is called online training. However, RL can train on datasets as well, which is known as offline training [11]. RL requires an agent that can influence its environment, an environment that can be observed and influenced by the agent at various states, and a reward that can be calculated from the actions performed by the agent. In reinforcement learning a policy is learned. The policy is the set of instructions that the RL algorithm uses to decide what action to take. This policy is learned by performing a certain task repeatedly, where the learner will attempt to find the optimal actions based on the current state to maximize its reward. [12]

In order to understand what paradigm best fits the given problem, the process of congestion control is simplified as follows: A CCA is an algorithm that can perform actions to influence the network congestion, based on information gathered from the network. The CCA is able to perform a given action based on a list of hand-crafted rules. Using this simplified description, the concepts of RL can be applied as follows: The CCA will be an agent that performs actions to influence the environment, based on the current state of the environment. The CCA is able to perform a given action based on a learned policy. By applying the concepts of RL to the simplified description, it is shown how the process of a traditional CCA can be described using these concepts. Through this it is evident that

RL is a suitable paradigm for the solution, and will therefore be the focus of this project.

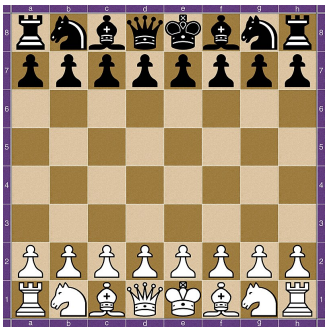
2.3 Reinforcement learning

Reinforcement learning sets out to create an optimal policy by reinforcing good actions taken by an agent. The agent is given rewards to signal whether the action was desired or not. This concept has been linked to animal behavior and the neurology behind learning. Furthermore, the studies of RL and animal learning have made great impacts on each other, inspiring new techniques and theories for both [13]. The concept of RL stems from reinforcing a desired behavior using rewards. This concept was first applied to animal learning, famously demonstrated by Pavlov when he conditioned a dog to salivate when hearing the sound of a specific bell [14]. Animals are typically rewarded with some form of treat or food during RL, whereas machines are rewarded with a numerical value that the machine aims to maximize.

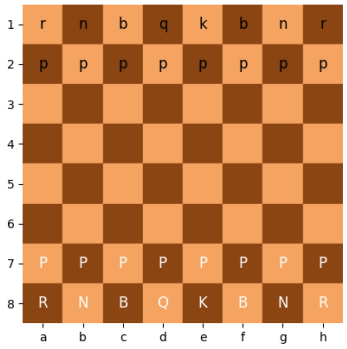
2.3.1 A reinforcement learning example

Reinforcement learning has been applied to many real-world scenarios. In 2017 RL was used to create a revolutionary chess-playing engine, AlphaZero, that could outperform all other engines and players at the time with less than 24 hours of training [15]. AlphaZero was also taught the games shogi and go, where it also achieved performances that outperformed human and artificial players within 24 hours of learning.

A lot of games, such as chess, can be seen as learning problems which RL is well suited to. The chess player is the agent, who can take actions by moving their pieces around on the board, within the legal restraints of the game. The chessboard is the environment, which contains all the information that is currently available and can possibly be available. This is to mean that there can never be any information outside of the environment, in the case of chess that would mean that pieces cannot move outside of the board. For the RL network, the chessboard will be a numerical representation, as can be seen in Figure 2.3. For the machine, the board is represented as a matrix, where each index is a tile on the board. The tiles can then be either vacant, represented with a dot, or have a piece on it. The pieces are usually represented either as a numerical value or a letter where capitalization denotes the color of the piece. The RL network constructs a policy to decide how to play the game. The policy acts as an instruction list for the agent. Initially, the agent performs seemingly random actions, moving a piece from one tile to another, as it does not have any understanding of the rules or objective of the game. However, throughout training, the RL network is given reward signals to indicate desired behavior. A reward, in the form of a numerical value, is given to the RL network once the game ends, based on the outcome of the game. As winning the game is the desired outcome, the RL network receives a reward if it wins, or a punishment if it loses. This encourages the RL network to attempt to win. This does however create a small obstacle for learning initially. A lot[1] of unique states, also called positions in chess, are possible. This means that it is difficult for the agent to win a game while still performing seemingly random actions. Reward shaping can be



(a) Starting position in a chess game shown on a digital chessboard [16]



(b) A graphical representation of the state

$$\begin{bmatrix} r & n & b & q & k & b & n & r \\ p & p & p & p & p & p & p & p \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ P & P & P & P & P & P & P & P \\ R & N & B & Q & K & B & N & R \end{bmatrix}$$

(c) The state represented as a matrix

Figure 2.3. Different representations of a starting position in chess

[1] Approximately $2 \cdot 10^{40}$ [17]

utilized to overcome this initial problem. Reward shaping is giving small rewards when the agent does something that helps it achieve its goal. In the case of chess, this could be capturing pieces, gaining control over the center tiles, castling into a strong defensive position, etc. Reward shaping can be utilized to speed up learning, but it comes with the cost of introducing bias into the learning since these sub-goals have been artificially created by the designer of the RL network. This bias comes as a byproduct of the human encouraging the machine to make specific types of plays that seem good to the human, instead of the machine learning everything from scratch.

2.3.2 An introduction to reinforcement learning

The RL framework, as can be seen in Figure 2.4, consists of an agent and an environment. The agent is able to observe the state $\mathbf{s_t}$ of and interact with the environment, such an interaction is called an action $\mathbf{a_t}$. Once an action has been performed, the agent receives a reward $\mathbf{r_t}$. This reward is determined by the current state of the environment. This reward is used as a signal for the agent to reinforce desired behavior.

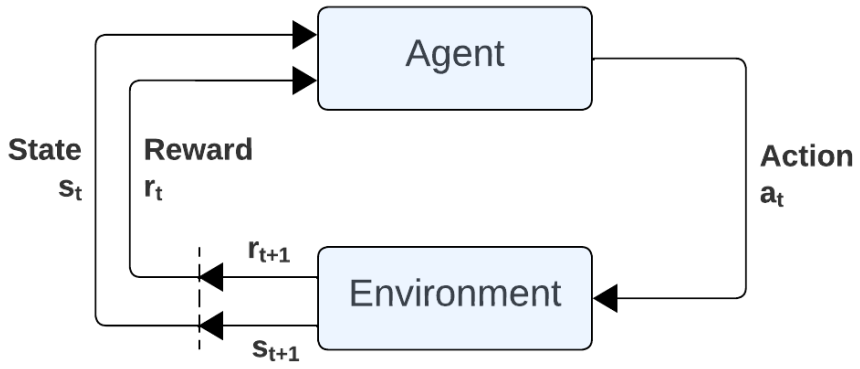


Figure 2.4. Interaction overview for the reinforcement learning framework [13]

The policy

The agent performs actions based on a policy π. The policy is learned in order to maximize the reward, while taking the future reward into account. Because of this, RL can be described as an optimization problem, in relation to optimizing the policy $\pi(\mathbf{s,a})$ [12],

$$\pi(S, A) = Pr(A = A|S = S) \qquad [12] \quad (2.1)$$

Equation 2.1 formalizes the probability of a policy taking an action given a state. In a simple case, an optimal policy could be a look-up table containing the optimal actions for each state-action pairing, in order to maximize the future reward. However, as the problem increases in complexity, the requirements for learning and representing an optimal policy become more expensive. In many cases, learning and representing an optimal policy becomes infeasible due to the large size of the state and/or action space, therefore the policy π is represented as an approximate function containing a lower dimensional argument θ,

$$\pi(s,a) \approx \pi(s,a,\theta) \quad [12] \quad (2.2)$$

The approximate policy can also be denoted as $\boldsymbol{\pi_\theta(s,a)}$.

The environment

The state of a system can often be assumed to be a Markov Decision Process (MDP), however, this assumption is a simplification, as the state of a system typically is a higher-dimensional environmental state based on a stochastic non-linear dynamical system [12]. Assuming the state evolves as an MDP and this MDP is known, a model of the process can be created and exploited by the learner.
Dynamic programming is often used to solve stochastic optimization problems, such as learning a policy. However, dynamic programming does have issues with scalability and is not well suited for problems with large state spaces. Dynamic programming is a mathematical framework that sets out to recursively break down a decision problem into smaller problems, based on available actions and their influence. Bellman's equation, seen in Equation 2.3, can be used if the sub-problems can be nested within a larger problem to find an optimal solution.

$$V(s) = max_\pi \mathbb{E}(r_0 + \gamma V(s')) \quad [12] \quad (2.3)$$

MDPs provide a framework for solving such optimization problems using dynamic programming. An MDP is an extension to Markov Chains, adding decisions and rewards, and is used in systems that are controlled partly by environmental randomness and partly by decisions taken by an actor. An MDP consists of a state-set $\mathcal{S}$, action-set $\mathcal{A}$, a set of rewards $\mathcal{R}$, a reward function R

$$R(s', s, a) = Pr(r_{k+1}|s_{k+1} = s', s_k = s, a_k = a) \quad [12] \quad (2.4)$$

and a chance to switch to a different state given the occurrence of an event, meaning an action taken at a given time in a given state.
While MDPs are a good framework for representing a model of the environment, not all environments can be represented using a model. These types of environments require model-free RL methods, which cannot utilize MDPs.

The value function

The value of a state quantifies the desirability, which is the cumulative reward, that can be gained from the current state in the long run, whereas the reward is a signal of desirability for the state in the short term. The value of the state can also be described as the expected cumulative reward. As states become more uncertain the further they are in the future, the more uncertain their expected reward becomes. Due to this reason, a discount rate γ is introduced to regulate the influence of the expected rewards,

$$V_\pi(s) = \mathbb{E}(\sum_k \gamma^k r_k | s_0 = s) \quad [12] \quad (2.5)$$

The subscript $\boldsymbol{\pi}$ indicates the value function for the specific policy. A more general variation of the value function for the best possible policy can be written as follows,

$$V(s) = \max_\pi \mathbb{E}(r_0 + \sum_{k=1}^{\infty} \gamma^k r_k | s_1 = s') \quad [12] \quad (2.6)$$

2.3.3 Categorization of reinforcement learning methods

Reinforcement learning problems come in a variety of types, based on the information that can be extracted from the environment. One can imagine a system that sets out to learn how to balance an inverted pendulum, such a task is completely deterministic based on the physics of the environment. Due to the deterministic nature of this task, a model of the environment can be established. On the other hand, a system that is learning to play chess will be faced with a non-deterministic adversary. While this adversary will be limited by the legal moves of the game, the moves chosen within this move-set could essentially be random, although some moves are more likely than others, as the adversary is expected to attempt to win the game. Due to this randomness from the adversary, an action taken by the agent from a specific state, may not always result in the same next state. This means that a model cannot be created for the task. In RL, two major categories are usually established, model-based RL and model-free RL [13]. Within these categories, sub-categories are present.

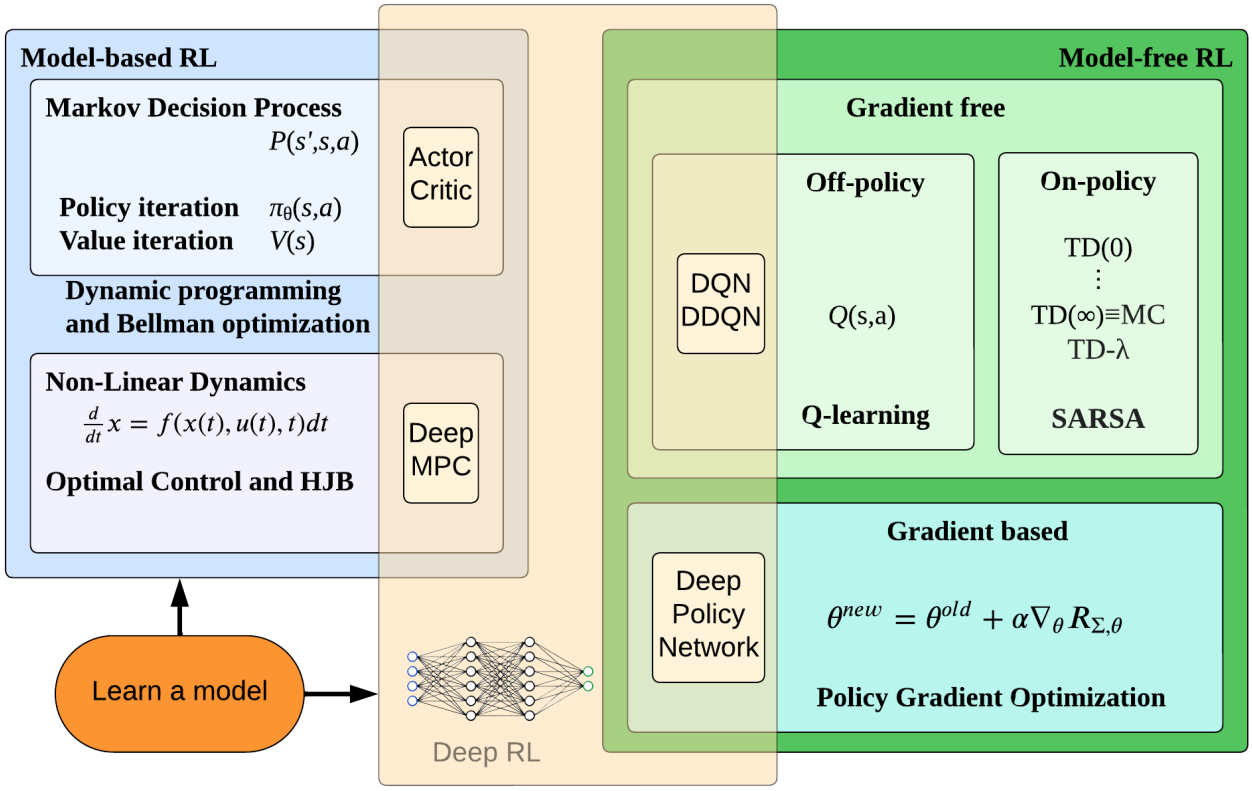


Figure 2.5. Simplified categorization of reinforcement learning techniques [12].

Figure 2.5 shows a simplification of the different categories and sub-categories. By analyzing the problem at hand using this categorization map, the type of RL algorithm that is most appropriate for the problem can be found.

2.3.4 Model-based vs. model-free

Model-based RL is deeply rooted in control theory and often utilizes control theory techniques alongside a model-based RL algorithm to further improve results [12]. In the case that a model

for a finite MDP is known for the environment, policy iteration or value iteration can be used to find the optimal policy or value respectively. These optimization algorithms are based on the dynamic programming framework mentioned in Subsection 2.3.2, which can reduce the computational complexity of learning to be linearly scaling with the number of sub-problems, while always finding the optimal policy or value function. Once the optimal value function has been found, it can be used to find the optimal policy,

$$\pi = \arg\max_{\pi} \mathbb{E}(r_0 + \gamma V(s')) \quad [12] \quad (2.7)$$

and the optimal value function can likewise be found by inverting the calculation.
Policy iteration and value iteration are both algorithms that exploit the knowledge of the model, by using the knowledge of the next state *s'*. While policy iteration and value iteration are powerful tools to optimize an RL algorithm, they are limited to model-based algorithms.

Model-free RL is used when there is no clear model that can be applied to the environment. This means that the RL algorithm is not capable of using the next state *s'* to make its decisions, however, a prediction of a likely state can be used. The prediction is made using trial and error throughout training. This can be seen in the case of a chess game. During a given state, the agent cannot know what the adversarial's next move will be, given the adversarial has more than a single legal move available. But the agent can try to predict how likely a move will be. Since the RL method does not know the next state *s'*, the optimal policy equation used for model-based RL methods, Equation 2.7, cannot be used without slight modification. Instead, model-free RL methods use their prediction of the next state *s'* as an approximation of *V(s')*.
Time-delayed rewards can be used to enforce behavior that may not necessarily be good at the time, but good in the future. This can again be seen in the case of chess. The agent may move a knight, which does not capture a piece on that specific turn, but the knight is now forking the king and a rook, creating a favorable position. Using a time-delayed reward, moving the knight to this position is shown to be a good move. This way a model-free RL algorithm can learn about what to expect from future states, without having a model to infer from.

When transferring data over a network, the variations experienced causing congestion which CCAs attempt to deal with, is often caused by stochastic environmental factors, such as other users. For this reason, a model cannot be created to accurately determine the future states of the network, leading an RL-based CCA to require a model-free approach.

2.3.5 Gradient-based vs. gradient-free

Gradient-free methods, also called action-value methods, create a policy based on the value of actions. The value of an action can be estimated in a variety of ways, one option is to estimate the cumulative reward for each action in each state, while another option is to find the quality of an action,

$$Q_t(a) = \frac{\sum_{i=1}^{t-1} R_i \cdot \mathbb{1}_{A_i=a}}{\sum_{i=1}^{t-1} \mathbb{1}_{A_i=a}} \quad [13] \quad (2.8)$$

In Equation 2.8, $\mathbb{1}_{A_i=a}$ is the random variable that is 1 if $A_i = a$ is true, and 0 if otherwise. In simpler words, the quality of action is the sum of rewards when action a taken prior to time t divided by the number of times action a was taken before time t, with some form of fail-safe for the denominator reaching zero [13].
The policy of a gradient-free method always takes the action with the highest value. This feature

allows the policy to be derived from the value function, and vice versa, as mentioned in Subsection 2.3.4. Due to the greedy nature of the value-based approach, the RL method can find itself stuck in a local maximum for optimality. For this reason, an off-policy method can be used, which introduces randomness into the training by forcing the RL algorithm to occasionally, decided by a parameter $\boldsymbol{\varepsilon}$, take random actions. By occasionally taking random actions, the RL algorithm will eventually have explored all actions for all states, finding the optimal solution for any problem, based on the definition chosen for the value of an action. By mapping the highest value of all actions in every state, and always choosing the action with the highest value, gradient-free methods are inherently deterministic. Due to the deterministic nature of these methods, gradient-free methods are mostly useful in problems with perfect information, such as chess, Super Mario Bros., energy management in smart grids, and robotics in manufacturing.

Gradient-based methods learn a parameterized policy,

$$\pi(a \mid s, \theta) = Pr\{A_t = a \mid S_t = s, \theta_t = \theta\} \quad \text{[13]} \quad (2.9)$$

Which uses the parameter vector $\theta \in \mathbb{R}^{d'}$ for decision-making of an action taken in a given state during a given time.
The key difference between a gradient-based method and a gradient-free method is that a gradient-based method does not need to pick the action with the highest value. However, a value function may still be used to learn policy parameters. Instead of using a value function to determine what action to take, the policy's performance is optimized using the gradient of the policy parameter for each action in each state using gradient ascent [13]. An actor-critic method can be used to learn approximates of both the policy and value function. In such a case, the actor refers to the policy, while the critic refers to the value function. This method is useful as the critic can learn from temporal difference signals, giving the ability to learn how actions influence later actions. A temporal difference signal is a delayed reward. This works by the agent taking an action followed by some amount of more actions, then a reward is given to the first action based on the later state. Using the chess example, an end-game scenario could require the actor to sacrifice a rook to get a checkmate. On the individual play, losing a rook would seem bad, but as it enables a checkmate it is the correct play to make and should be rewarded. To ensure exploration, a gradient-based method establishes action preferences for each state-action pair. These action preferences are parameterized numerical preferences that use a non-linear function, such as an exponential softmax distribution, to convert the parameter vector into a probability distribution. When taking an action, the policy chooses the action with the highest preference. This allows the policy to become robust in stochastic environments, while in a deterministic environment, the policy will learn to have a very high preference for a single action per state-action pair [12]. Due to this stochastic nature, gradient-based methods are preferred in problems with imperfect information, such as poker, social deduction/hidden role games, autonomous vehicles in dynamic environments, and financial market trading.

During online communication, congestion can happen abruptly depending on the network. In the common case of communicating over the World Wide Web, the communication link is commonly shared with other users. During communication, it is not known how many other users use the same link and to what extent and rate. For this reason, congestion control during online communication can be seen as a game with imperfect information, in which gradient-based RL methods are known to be preferred.

2.3.6 Deep reinforcement learning

A large problem for classical RL is known as the curse of dimensionality. Many problems that may be solved using RL techniques can suffer from having a very large state space or input space. This is seen in problems that require visual information. In the classic Atari game pong at a monochromatic standard resolution of 336 X 240, the number of discrete states would be over $10^{24,000}$ and have an input space of 80,640 [12]. Due to this problem, representing the policy and value function becomes an exceedingly more complex task. Deep learning models can find complex functions that would be virtually impossible for a person to craft.

Supervised learning was briefly introduced in Section 2.4. Typically the deep learning model used for deep reinforcement learning (DRL) will be a deep SL model. Many architectures exist within this category, created to solve different types of goals. For image processing, a typical architecture would be a Convolutional Neural Network (CNN), for problems with sequential data, a Recurrent Neural Network (RNN) along with its variations, is typically utilized, for problems with data represented as graphs, a Graph Neural Network (GNN) is typically utilized, etc. Due to the variance of architecture, it is important to identify what type of problem one is trying to solve so that a proper architecture can be implemented.

2.4 Literature review

Utilizing RL to improve CCAs is currently a hot topic of interest for researchers and tech companies, such as Microsoft [18] and Meta [19]. While implementations using various ML techniques for CCAs have been created [3], DRL-based CCAs have been shown to produce the best results.

Aurora [20] was created by Jay et al. in 2019 as a novel approach to congestion control. Until this point, research into congestion control using ML had not been attempted using DRL. This project uses a simple DRL implementation along with a simple linear reward function to show the potential of DRL within the application of CCA.
Aurora is a rate-based CCA, where the DRL network outputs a value that is used to set the sending rate. This value is then modified using Equation 2.10 to prevent the output from oscillating, with the final implementation having $\alpha = 0.025$.

$$x_t = \begin{cases} x_{t-1} * (1 + \alpha a_t) & a_t \geq 0 \\ x_{t-1}/(1 - \alpha a_t) & a_t < 0 \end{cases} \quad \text{[20]} \quad (2.10)$$

Aurora uses a statistical vector as the input. This vector consists of a latency gradient, latency ratio, and sending ratio. These terms are defined as follows, the latency gradient is the derivative of the latency in respect of time within a monitor interval of the size k, latency ratio is the ratio of the mean latency within the monitor interval and the lowest mean latency of any monitor interval recorded during the session, and sending ratio is the ratio of number of packets sent and ACKs received in the monitor interval.
The network architecture consists of a Proximate Policy Optimization (PPO) network as the RL method, with a Multi-Layer Perceptron (MLP) with two hidden layers as the neural network. The reward function used for Aurora was a linear function, which can be seen in Equation 2.11, utilizing throughput, latency, and loss.

$$reward = 10 * throughput - 1000 * latency - 2000 * loss \quad \text{[20]} \quad (2.11)$$

Despite the simple implementation, test results showed that Aurora was on par with or outperforming state-of-the-art rule-based and ML-based CCAs at the time of its release. Furthermore, Aurora was shown to be surprisingly robust in situations that it was not trained for.

R3Net [18] was developed by Microsoft as an initial attempt at implementing a DRL CCA and gauging its potential. Microsoft based its implementation on the usecase of real-time communication through video, which requires low latency but can be forgiving on packet drops and ordering mistakes to some extent, however, data cannot be prefetched.
Like Aurora, R3Net is a rate-based CCA that uses a DRL network to output a value that is used to set the sending rate. A sigmoid function was used to squish the output down to a normalized value of (0,1). This value was then mapped to a range of (0,8) Mb/s. R3Net used a vector consisting of the receive rate, average packet interval, packet loss rate, and average RTT as the input.
R3Net was implemented using a PPO network with a Gated-Recurrent Unit (GRU) variation of an RNN. The neural network consisted of four fully connected layers and two GRU layers. A reward was calculated, using the reward function seen in Equation 2.12, after every time step of 50ms based on the receive rate $\mathcal{R}$ within the time step, average RTT $\mathcal{D}$ within the time step, and packet loss rate $\mathcal{L}$ within the time step.

$$reward = 0.6\ln(4\mathcal{R} + 1) - \mathcal{D} - 10\mathcal{L} \quad \text{[18]} \quad (2.12)$$

While Microsoft concluded that R3Net shows promising results, their tests do not include any comparisons to state-of-the-art CCAs.

MVFST-RL [19] was developed by Meta for further optimization within their network communication. Meta has developed an alternative version of the QUIC protocol called MVFST which is used within their networks. MVFST-RL was Meta's initial attempt at creating a CCA using DRL. During the development of MVFST-RL, Meta focused on keeping the processing time of cWnd updates down. Meta argued that good results during event-based simulations that are paused between cWnd updates are not indicative of good performance in a real-world scenario, as the updates need to happen quickly in real-time to have any merit for proper usage.
Unlike the previously mentioned DRL-based CCAs, MVFST-RL is a window-based CCA. This means that instead of setting the sending rate to a specific value, MVFST-RL instead sets the cWnd value, dictating how many packets can be in-flight at a given time. While Meta does not describe every state feature used for the input, they do inform that the input vector consists of 20 features derived from acknowledgment packets from the receiver, such as RTT, queuing delay, packets sent, acknowledged, and lost. Instead of setting the cWnd to a specific value, MVFST-RL instead changes the value by performing one of the following actions: $cWnd : \{\, cWnd,\, cWnd/2,\, cWnd - 10,\, cWnd + 10,\, cWnd * 2\}$
The reward function consisted of the average throughput t and the maximum delay d during a 100ms window. These values were further influenced by a scalar parameter β and a small value to ensure stability ε.

$$reward = \log(t + \varepsilon) - \beta\log(d + \varepsilon) \quad \text{[19]} \quad (2.13)$$

MVFST-RL was developed using IMPALA as the RL method with the LSTM variant of RNNs. Meta reasoned the usage of IMPALA based on its asynchronicity. Using this method, the network established an actor and learner system, as the asynchronicity allows the network to update the gradient of the learner throughout the decision-making process while training without slowing down. The DL model consisted of two MLP layers and a single LSTM layer.
Meta concluded that their implementation showed promising results. MVFST-RL was able to retrieve a

cWnd update from the policy in approximately 3ms. Furthermore, MVFST-RL was able to outperform rule-based CCAs such as cubic. However, Meta found it difficult to train a model that could produce good results for a variety of network cases, instead their policy would become very specialized in specific cases.

Aurora, R3Net, and MVFST-RL are just a few of the DRL-based CCAs that have currently been published but are representative of the current state of the technology. As of now, it has been proven that DRL-based CCAs have great potential, although they can become too specialized for specific cases, leading them to not be good for general usage. Additionally, while Aurora and MVFST-RL compared performance against other CCAs, they did not consider the fairness of their DRL-based CCA. Some other DRL-based CCAs that were not explored in this section are Eagle [21], DRL-CC [22], Orca [23], and TCP-PPO$_2$ [24].

2.5 Delimitation

Based on the literature review in Section 2.4, it can be seen that examples of policies created using DRL-based networks can achieve better results in certain scenarios than standard rule-based CCAs. However, there is still room for improvement in terms of bandwidth utilization and fairness. The purpose of this project is therefore to further explore the area of DRL-based CCAs and to create a DRL-based CCA that can compete with existing CCAs.

As argued in Section 2.3, network communication is a stochastic process with imperfect information. Due to this type of environment, it is clear that a model-free gradient-based method is preferable. From this understanding, the choice of RL method can therefore be narrowed down to this category. The introduction of deep neural networks can further enhance the RL methods, as the policy and value functions are not limited by human-made heuristics, which is the limitation of the rule-based CCAs that ML-based CCAs are trying to overcome.

While it is important for a CCA to be effective in various network communication scenarios, this project will limit its scope to communication through a simulated dumbbell typology. The scope is limited to a simulated environment so that the network can be fully controlled and monitored without any environmental disturbances. By using a simulated environment, the results derived from this project will, however, not necessarily reflect the performance that may be achieved in a real-world scenario. Due to the simulation environment, the network will not experience any packet drops as a result of lossy connections or other environmental factors, ensuring all dropped packets are due to congestion within the network.

Despite all the network information being available through the simulation, the nodes on the simulated network will only be able to get information that would ordinarily be available for a node on a network in a real-world network. This is to ensure that the CCA, that is created throughout this project, will also be able to work in real network communication.

2.6 Problem formulation

How can deep reinforcement learning be used to learn a policy for TCP congestion control that is on par or outperforms classical congestion control algorithms based on hand-tuned heuristics while behaving fairly?

Part II

System Design and Implementation

3 | Design

This chapter will describe the considerations and decisions taken throughout designing a new DRL-based CCA which is proposed in this project named Reinforcing Lower Congestion (ReLoC).

3.1 Design overview

The system consists of two parts, the agent and the network, as can be seen in Figure 3.1. The network functions as the environment during training and testing of the agent.

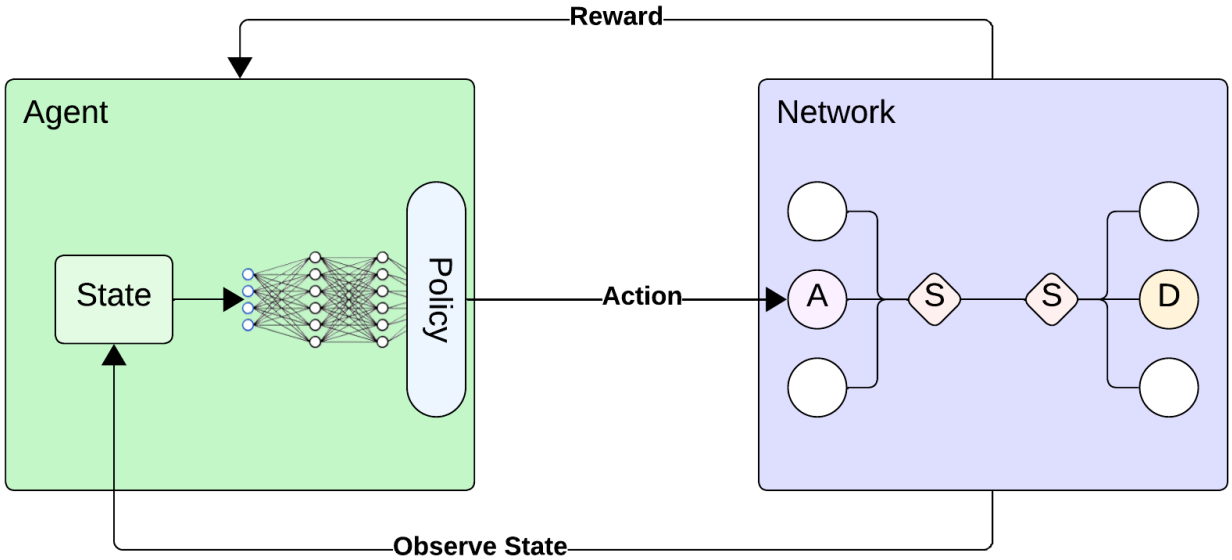


Figure 3.1. Simplified overview of the system

The agent node, denoted A, communicates with the destination node, denoted D, for a specified amount of time called the monitoring window. This communication is done over a dumbbell typology with two switches, denoted S, creating a bridge. On this network, other nodes can also be specified to communicate. After communication has been logged over the monitoring window, it is relayed to the agent, where the agent can observe the state of the environment. Once the state has been updated, it is passed through a deep learning model where the policy decides an action. This action is then applied by the agent node. In addition to observing the state of the environment, the agent also receives a reward once the communication has been logged.

3.2 Choice of reinforcement learning algorithm

In the delimitation, Section 2.5, the category of RL algorithms suited for the problem was narrowed down to model-free gradient-based methods. Furthermore, it was argued that introducing a deep neural network should be beneficial to achieving even greater results. This section will be used to

reason the choice of RL algorithm by looking into some popular RL methods, while the deep neural network will be discussed further in Section 3.3.

Deep deterministic policy gradient
Deep Deterministic Policy Gradient (DDPG) [25] is an off-policy RL algorithm that is inspired by Q-learning. The main philosophy behind the algorithm is that optimal action can be found given the optimal action-value function, that is the Q-function. Additionally to learning the Q-function, DDPG also learns a policy. This policy is used to approximate the optimal Q-function. Finding the optimal Q-function is trivial for problems with small action spaces, but as the action space increases in size, finding the optimal Q-function becomes exceedingly expensive, and finding the optimal Q-function for a continuous action space becomes impossible. However, the crucial point of DDPG's design is the capability to handle continuous action spaces [26]. DDPG is able to achieve this by approximating the optimal Q-function by using a gradient-based rule for learning the policy $\mu(s)$ to substitute for the action. This results in an approximation as follows,

$$\max_a Q(s,a) \approx Q(s,\mu(s)) \quad [26] \quad (3.1)$$

DDPG can be viewed as a variant of Q-learning specifically meant for continuous action spaces, however, it can only be used for problems with continuous action spaces. An expanded version of DDPG has also been developed, called Twin Delayed DDPG (TD3) [27]. This version is less sensitive to small adjustments of various hyperparameters that could cause overestimating Q-values while generally outperforming standard DDPG.

Proximal policy optimization
Proximal Policy Optimization (PPO) [28] is an on-policy gradient-based RL algorithm that handles both discrete and continuous action spaces. PPO sets out to take large steps away from the current policy without causing a performance collapse. In this context, a "step" refers to the extent to which the agent deviates from the current policy during training. By taking large steps, the algorithm becomes more sample efficient, meaning it is able to learn faster, thereby requiring less training. Two versions of PPO have been developed for this purpose, PPO-penalty and PPO-clip.
PPO-penalty uses a statistical distance called the Kullback-Leibler divergence, which is a measure of how one probability distribution diverges from a second probability distribution, in order to keep the steps from becoming too large. The size that the steps can be is called the Kullback-Leibler constraint.
PPO-clip uses a clip function for the objective to limit the size of the step. The objective in this case is to maximize the probability ratio of selecting a particular state-action pair, constrained by a clipping term. This clipping function directly modifies the advantage estimate, which quantifies how much better an action is compared to the average action at a given state. If the advantage for a state-action pair is positive, the minimum clipping term limits how much the objective for the state-action pair will increase, limiting the change of the policy. Likewise, if the advantage is negative, the maximum clipping term will limit how much the advantage will decrease.
PPO-penalty is not as commonly used as PPO-clip, due to it being more complex to implement than PPO-clip while seemingly not gaining any better performance. For this reason, PPO-clip is also the main algorithm used at OpenAI, the developers of PPO [29].

Alongside DDPG, PPO-penalty, and PPO-clip, other methods have also proven successful. While the following methods were considered, they will not be further discussed in this project: IMPALA [30], TRPO [31], and SAC [32].

Both DDPG and PPO have proven to be successful in the implementation of DRL-based CCAs [20][18][22][23]. For this project, PPO has been chosen as the RL algorithm due to its high sample efficiency while maintaining high performance. Furthermore, PPO-clip has been chosen over PPO-penalty due to it being simpler to implement while yielding equal or better results.

3.3 Reinforcement learning specifications

The performance of an RL algorithm is limited by the design of the network. To achieve high performance, it is important to carefully consider the state space, action space, and reward function. Furthermore, for a DRL algorithm, the design of the neural network used is also a big factor in achieving higher performance.

3.3.1 Deep learning model

As mentioned in Chapter 2, introducing a deep learning model to the RL network will free the learning process from the limitations of human-made heuristics.
When referring to the works mentioned in the literature review, Section 2.4, it can be seen that Aurora [20] used an MLP, TCP-PPO$_2$ [24] used an Artificial Neural Network (ANN), and the other works used an RNN or an RNN variation [18][19][21][22][23].

In the article *A Deep Reinforcement Learning Perspective on Internet Congestion Control* by Jay et al.. [20], they argue that "*Neural network architectures vary significantly and research suggests new architectures at an incredible rate, so choosing the optimal architecture is impractical.*" They present this argument to reason using a simple neural network for Aurora. With this, they argue that while even using a small fully connected neural network, a DRL-based CCA is capable of producing good results. This does, however, highlight that Jay et al.. did not attempt to implement the best-fitting DL architecture for the problem.

While Shi and Wang [24] mention the usage of an ANN, no details are given of its design or structure along with no argumentation of the choice of architecture. Even more troublesome for understanding the design is the usage of the umbrella term ANN. An ANN is widely used as a term for any neural network that aims to mimic the functionality of a biological neural network. This means that the term ANN may also cover networks using architectures such as MLPs, CNNs, RNNs, etc. Without any details of the ANN design, it is difficult to say what type of neural network Shi and Wang implemented.

As mentioned above, other related works used an RNN or an RNN variant. Abbasloo et al. [23] argues a need for introducing recurrency with the implementation of Orca. This argument claims that using only the current state space provides insufficient information about the network, due to the sporadic nature a network can have. While Abbasloo et al. do not go into detail about the design or implementation of their RNN, they do highlight the core problem when using RNNs, catastrophic forgetting. Catastrophic forgetting is one side of the problem found when using RNNs, called the exploding/vanishing gradient problem. This problem causes older information to either gain more or lose relevance for every new piece of information that is used. As long as the input sequence is short, this does not become problematic. But as the input sequence expands, performance degrades dramatically. This problem has however been solved with the LSTM variant and GRU variant. The

cell design of an LSTM is more complex than the GRU's cell design. This allows an LSTM to learn more complex solutions for long input sequences but also results in higher computational demands than a GRU. This trade-off between complexity and speed can generally be used as a deciding factor when choosing between an RNN variant.

For the implementation of ReLoC, the neural network will utilize LSTM layers along with fully connected layers. The LSTM layers are used to introduce recurrency into the decision-making, allowing the agent to gain a deeper understanding of the environment, by using information about how the environment has developed over time. The decision between using an LSTM or GRU depends on many factors. For general-purpose communication, it is a bit blurred which would be preferable. As this project aims to show a proof of concept, an LSTM was chosen to create a model that may find better solutions through higher complexity. Additionally to the LSTM layers, fully connected layers are used to further process a representation of the state and to process the outputs of the LSTM layers. Furthermore, it is common practice to have a fully connected layer after an LSTM layer that does not feed into another LSTM layer.

While the basic structure of ReLoC's neural network simply consists of some number of LSTM layers followed by some number of fully connected layers, this basic neural network structure will be referred to as the network base structure. However, some variations based on this structure are desired. Two variations of the base structure will be explored in this project.

The first variation introduces a time-distributed fully connected layer before the LSTM layers. The layer being time-distributed means the fully connected layer will be applied to each time-step of the input individually. This allows the layer to modify the input in a similar manner to preprocessing before the data is given to the LSTM layer while maintaining the time-distributed nature of the input. This variation can be seen in Figure 3.2.
The second variant of the base network architecture introduces a skip connection in the layering, which can be seen in Figure 3.3. This skip connection would take the initial input and pass it to the lower fully connected layers. This type of connection serves to reduce information loss throughout the layers by having un-processed data at lower layers of the network architecture.

These variations will be trained and tested against the base structure of ReLoC using the same state space, action space, and reward function, along with other relevant hyperparameters. This allows testing the performance of the three different network architectures directly.

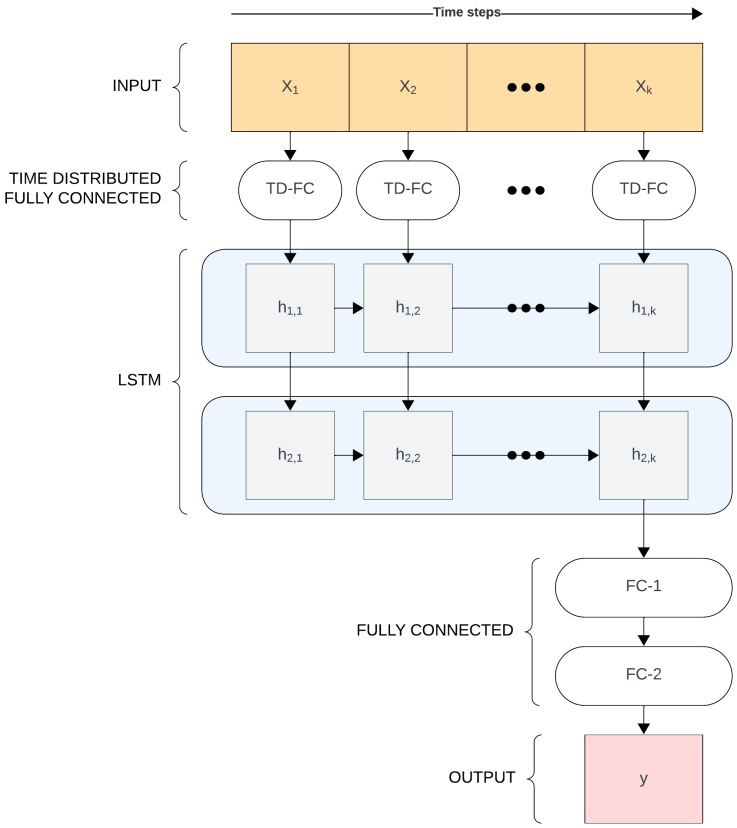


Figure 3.2. Network architecture of ReLoC's time distributed variant

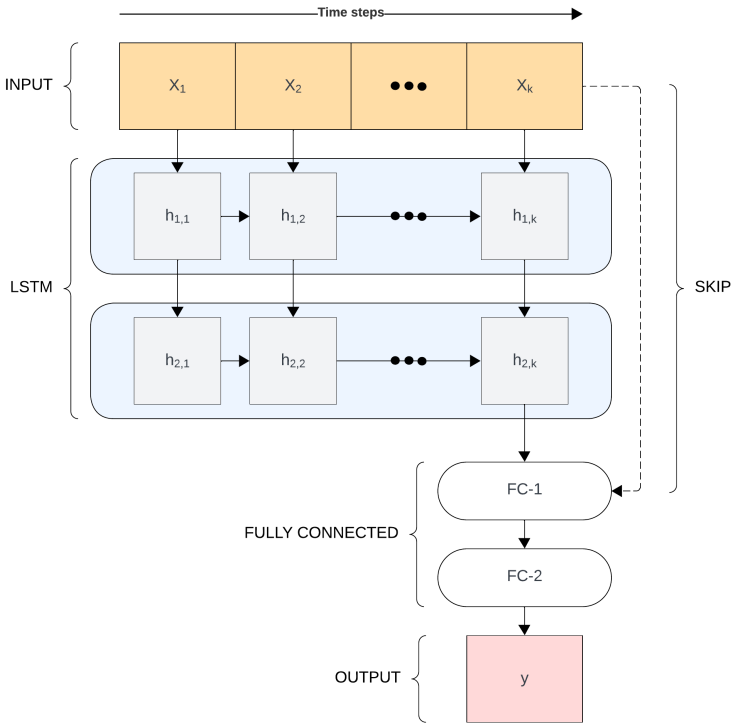


Figure 3.3. Network architecture of ReLoC's skip connection variant

3.3.2 State space

Since ReLoC is supposed to reflect the direction of the future generation of CCAs, it should be created such that it can be applied in general network scenarios. This means that ReLoC should not be reliant on network information that is not regularly available for nodes on networks.

The information that is used as the input for the network needs to reflect the state of the network. To utilize the available link capacity more optimally, the agent needs to recognize congestion on the channel. Since network congestion causes packets to drop and RTT to increase, these metrics can be used as indicators for congestion.

Despite dropped packets being an indicator of congestion, by itself, it does not relay any useful information. The amount of dropped packets must be in reference to the amount of sent packets. While dropping only a few packets during a transmission may sound good, it is not known whether those packets account for 1% of the transmissions or 50%. Furthermore, the definition of a dropped packet can vary depending on the source of information. For this project, dropped packets will be defined as *any sent packet that has not received an acknowledgment within the measuring window.*

For the implementation of ReLoC, the state space will include *sent packets* and *acknowledged packets.* While the amount of dropped packets, or a drop rate, could be included, it has been decided to disregard these metrics in order to minimize the complexity of the state space. Furthermore, these metrics are derivatives of *sent packets* and *acknowledged packets*, and for this reason, it is hoped that the agent will learn about dropped packets and their indication of congestion by itself.

The RTT is collected for a packet when the related acknowledgment is received. As various factors influence the RTT, it can be seen as a stochastic event. This means that throughout the measuring window, a probability distribution can be made for the RTT. At a constant link capacity, this distribution follows a normal distribution [33]. For this reason, the *average RTT* and *standard deviation of the RTT* is used to reflect the RTT on the network. It was chosen to include the *standard deviation of the RTT* in the state space, as using only the *average RTT* does not take into account the spread of the distribution.

Aurora [20] used a *latency gradient* for their state space. By using the gradient, Aurora is able to extract information that evolves over time. This approach has been deemed unnecessary for ReLoC, as ReLoC uses LSTM layers for the deep learning model, which introduces time-steps into the learning process. By using time steps, the deep learning model automatically extracts information regarding how the metrics evolve over time.

Both DRL-CC [22] and Orca [23] included the current *cWnd* in the state space. This has been deemed redundant for the implementation of ReLoC, as the amount of *sent packets* and their *RTT* directly reflects the current *cWnd*, when the node operates individually on the channel.

Based on these considerations and choices, the state space used for the implementation of ReLoC is as follows:
State Space = {*Sent packets, Acknowledged packets, Average RTT, standard deviation of the RTT*}

3.3.3 Action space

While increasing the action space with many different options can increase how effectively the agent can take actions, it will also increase the complexity of the action space, leading to more resource-heavy training. For this reason, the size of the action space should be as low as possible while still containing actions that can provide effective and desired changes to the environment.

For rate-based DRL-CCAs it is common to set the rate directly to some value determined by the agent, however, window-based algorithms typically change the cWnd using an arithmetic approach.
For this project, the action space has been narrowed down to five actions. This amount was speculated to be an appropriate amount of actions taking the complexity trade-off into consideration. As the optimal action space is virtually impossible to know without extensive trial and error, educated guesses are made to find an action space that can perform well. For this project, educated guesses were made based on a limited amount of trial and error. It is possible to further optimize the action space by automating tests running through different action spaces, this is however very computationally expensive and was decided not to be done for this project. For this implementation, the action spaces used by related works, mentioned in Section 2.4, were taken into consideration. While it can be seen that Aurora [20], R3Net [18], and DRL-CC [22] set the value of the rate or cWnd to a specific value, MVFST-RL [19] and Eagle [21] have a limited action space of five actions. These five actions consist of arithmetic operations to scale the value of the rate or cWnd. In both cases, the five actions give the agent the option to increase and decrease the rate or cWnd by a small and a large amount or to keep the value at the same amount. Using inspiration from these prior works, the action space for this project is as follows:

$$cWnd = \begin{cases} cWnd - 1 \\ cWnd * 0.75 \\ cWnd \\ cWnd * 1.25 \\ cWnd + 3 \end{cases} \tag{3.2}$$

It was decided that the agent will be able to lower the cWnd by subtracting a set amount of segments or by deducting a percentage value of the cWnd value. Likewise, it will be able to add a set amount of segments or multiply a percentage value. Additionally, it will also be able to keep the cWnd value at the same value, effectively taking no action.

3.3.4 Reward function

The reward function consists of two segments, a reward segment R_{pos} seen in Equation 3.5 and a punishment segment R_{neg} seen in Equation 3.6. The reward segment can further be split into two subsegments, which can be seen in Equation 3.3 and 3.4.
While it is important not to use any information that is not commonly available for the agent in a real-world scenario in the state space, such information can be used in the reward function. Since the reward function is only used during training and not during execution, using hidden information, such as network parameters, will not hinder the agent from executing on a real network. Once training is completed and the agent is deployed to a real network, the policy is not updated further. This means that the only necessary information, after training, is the information used in the state space, as the

agent will choose purely based on this information.

The ultimate goal of a CCA is to use the optimal amount of available link capacity. For this reason, the agent is rewarded by having a send rate S as close to the value of the available link capacity C_{avail} as possible. This subsegment of the reward function has been designed to be 1 if the available link capacity is used optimally, as follows,

$$C_{utility} = (1 - |1 - \frac{S}{C_{avail}}|) \tag{3.3}$$

As the agent can only send a non-negative integer amount of packets, *min(S)* will yield a reward of 0. It is, however, possible for the utilized capacity score $C_{utility}$ to become lower than 0, if the agent attempts to send more than 200% of the available link capacity, as can be seen in Figure 3.4.

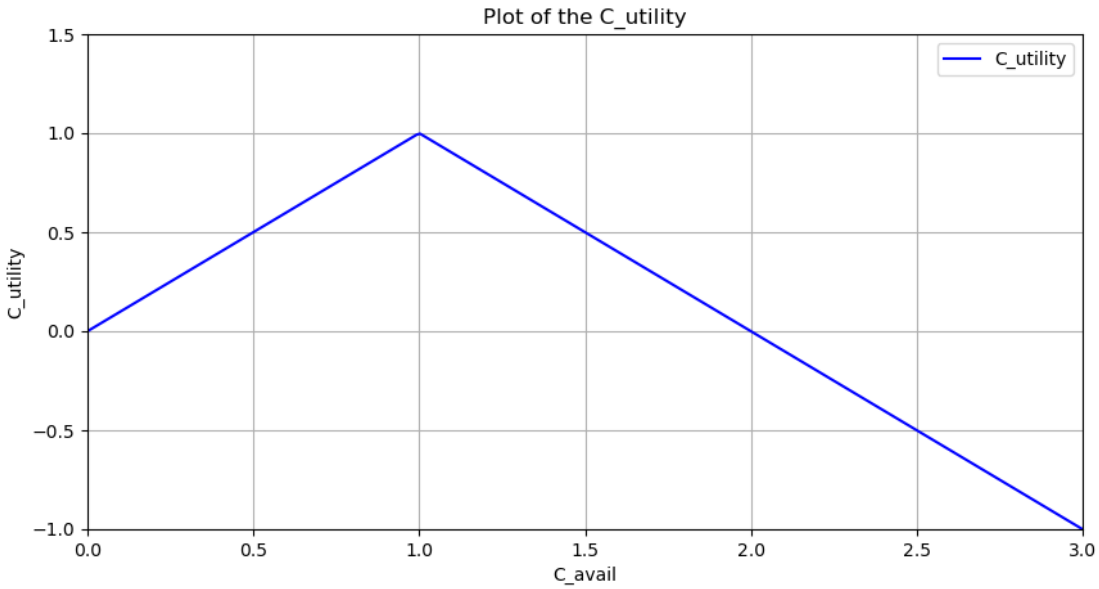


Figure 3.4. Graphical representation of $C_{utility}$

The inverse packet loss rate L is used to scale the positive reward. The inverse loss rate is calculated by dividing the received acknowledgement rate A with the send rate S. Both the received acknowledgement rate and send rate are measured as packets per second. While this variable is used in the reward segment, it is used to punish the agent when packets are dropped. This is done by scaling the positive part of the reward to be smaller in relation to how many packets are dropped.
The packet loss rate is in most cases a value between (0,1), however, in some instances it may be larger. This can occur in the case that a packet is sent by the end of the measuring window and the related acknowledgement is received at the beginning of the following measuring window.
If the agent attempts to send more than 200% of the available link capacity, then the agent should be heavily punished. To increase the punishment, if the utilized capacity score becomes negative then the inverse packet loss rate is set to 1. This will mean that the negative utilized capacity score is fully impactful. This results in the conditional function,

$$L = \begin{cases} 1, & C_{utility} < 0 \\ \frac{A}{S}, & Otherwise \end{cases} \tag{3.4}$$

To ensure that the reward can be calculated without any syntax errors, cases for error handling has been defined. These cases ensure that there are no division by zero errors. In the case that the sending rate and the available link capacity is 0, then the reward segment is set to 1. If either the sending rate or the available link capacity, but not both, are 0, then the reward segment is set to 0 as well. If neither the sending rate or available link capacity is 0, then the loss rate is multiplied with the utilized capacity score,

$$R_{pos} = \begin{cases} 1, & S = 0 \wedge C_{avail} = 0 \\ 0, & S = 0 \veebar C_{avail} = 0 \\ LC_{utility}, & Otherwise \end{cases} \tag{3.5}$$

By multiplying the inverse packet loss rate with the utilized capacity, the reward is lowered faster by surpassing capacity rather than staying within, which can be seen in Figure 3.5. This graph shows the positive reward value for a case where the dropped packet rate is equal to one over the available capacity once the maximum capacity has been breached.

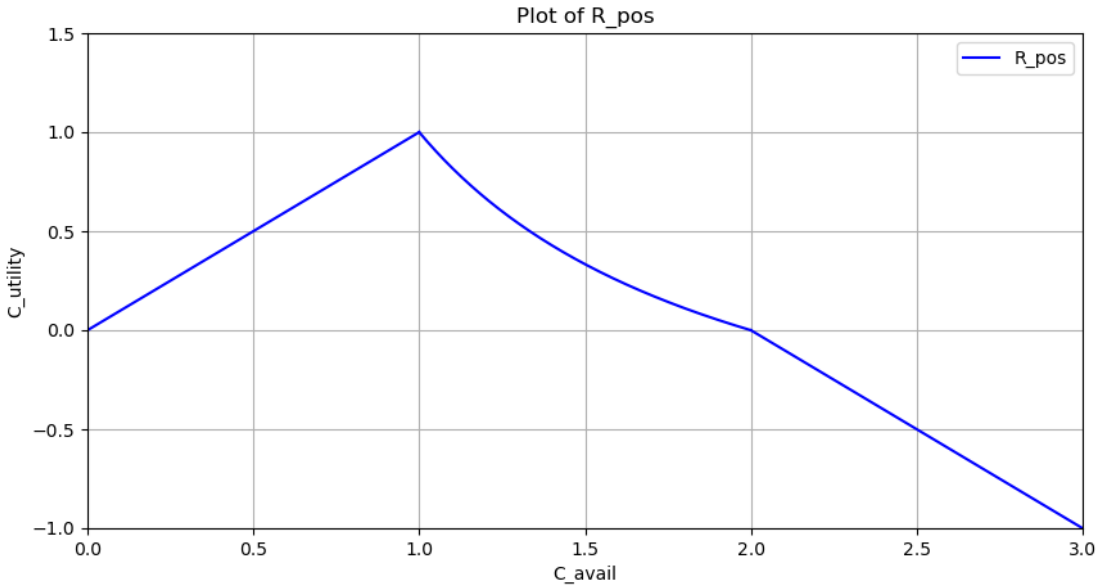


Figure 3.5. Graphical representation of R_{pos}

The punishment segment utilizes the average RTT measured over the measuring window as a metric for congestion. The average RTT is used to entice the agent to send at Kleinrock's optimal operating point. This is the point that achieves the lowest RTT with the highest delivery rate, as can be seen in Figure 3.6. This point exists as a side effect of packet queues on a network. Once a queue builds up, the RTT also increases, as the packets need to wait for their turn to be relayed. In the case that the queue is full and receives more packets, the packets are dropped.

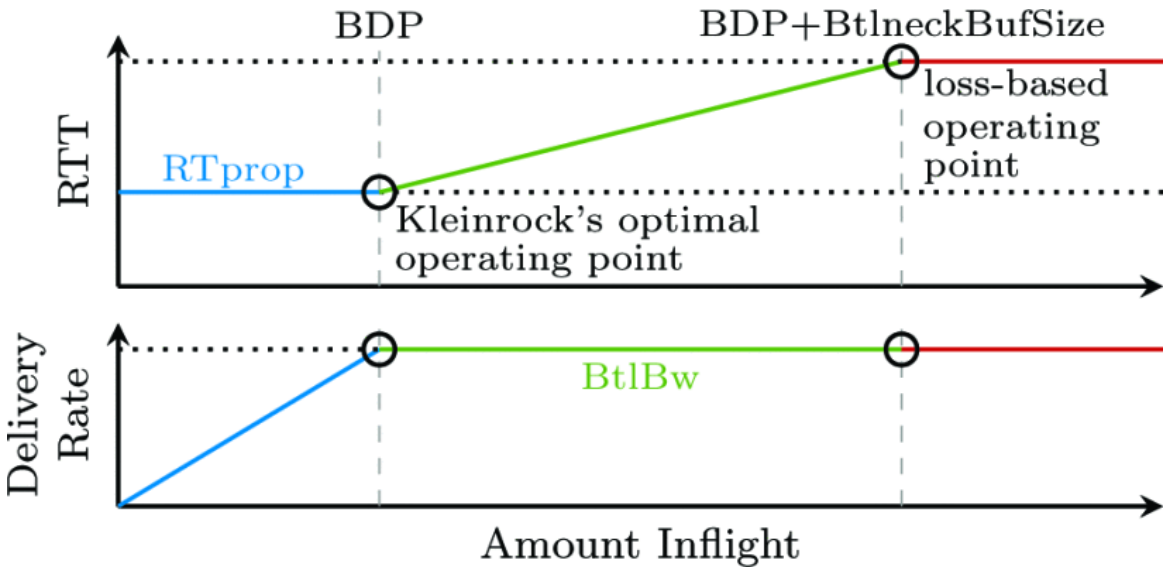


Figure 3.6. Effect of increasing the cWnd on the RTT and delivery rate [34]

From Figure 3.6 it can also be seen that once the delivery reaches the bottleneck bandwidth (BtlBw), the delivery rate is limited and the RTT starts increasing. From this, it can be understood that Kleinrock's optimal operating point is when a node is sending at a rate that reaches the BtlBw while keeping the RTT at a minimum point.

The square root is taken of the average RTT. This is done to throttle the punishment. For the purpose of throttling, a logarithm could have also been applied, however, a logarithmic approach would throttle more aggressively, making increasing RTT values less punishing. The punishment segment is also written as a conditional function to account for when the sending rate and available link capacity are equal to 0. This leads the punishment segment to be written as,

$$R_{neg} = \begin{cases} 0, & S = 0 \wedge C_{avail} = 0 \\ \sqrt{RTT_{avg}}, & Otherwise \end{cases} \tag{3.6}$$

To calculate the reward R, scalars α, β are used to further tune the rewards, as follows,

$$R = \alpha R_{pos} - \beta R_{neg} \tag{3.7}$$

The value of the scalars have been found through trial and error, as documented in Section 3.4.

3.4 Determining the scalar values for the reward function

A series of tests were performed to approximate appropriate values for the scalars alpha and beta in the reward function seen in Equation 3.7. These test use a trial and error approach on a simulated network which will be further discussed in Section 4.1. By training agents using different configurations for the scalar values, the resulting policies can be tested and their performance can be compared to determine what scalar values are the best. As it is not feasible to test every possible value for alpha and beta, the

test has been narrowed down to the following sets, $\alpha = \{2.5, 5.0\}$ and $\beta = \{0.0, 0.1, 0.2, 0.3, 0.4, 0.5\}$. Using the best performing combination, further tests will be performed with higher granularity, performing a step of 0.05 in each direction of the beta value.

3.4.1 Test setup

All the agents are trained using a deterministic network simulation and tested on a separate deterministic network simulation.
The agents are trained using a network simulation which changes the available link capacity throughout the simulation. These changes include both decreases and increases in the available capacity, to teach the agent both scenarios. The training simulation starts by offering 100% available capacity, this is then lowered to 50%, followed by 75%, then falling to 25%, and finally climbing back up to 100%, as can be seen in Figure 3.7. Each episode of training runs through this simulation, with 200 episodes of training per combination of scalar values. While this training does not encourage a robust policy, as the agent only gets familiar with those intervals, it should produce policies that can be compared and give a sufficient indication of the best combination of scalar values. For more robust training, the agents would be trained on networks that change the capacity to random values. However, this could potentially skew the results of this test, since the agents are not trained on the same simulations.

Figure 3.7. Changes in available link capacity through time during the training simulation for the alpha/beta test

Throughout the training, a rolling average of the reward for 30 episodes was taken, where the newest policy is saved when a new highest average is achieved. This policy is then used to evaluate the scalar value combination. The policies for each combination of scalar values are then used in a simulation that starts at 100% available link capacity, which then falls to 50% capacity, and then goes back to 75% capacity, as can be seen in Figure 3.8.

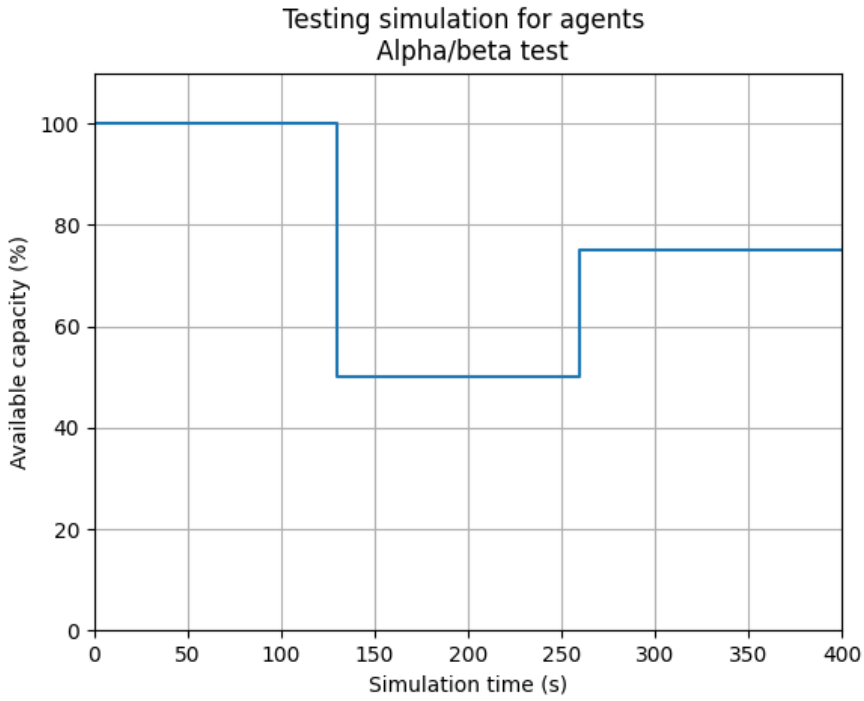


Figure 3.8. Changes in available link capacity through time during the test simulation for the alpha/beta test

To evaluate the performance of the policies, the cumulative amount of bytes sent, which has not been lost, over the simulation is compared.

3.4.2 Test results

As PPO is a gradient-based policy, its decisions are not deterministic. This means that using the same policy on the same test case can lead to some variance in the results. For this reason, each policy was tested five times, where the results of the five tests were averaged.

α\β	0.0	0.1	0.2	0.3	0.4	0.5
2.5	12,598,272	12,981,760	9,706,906	9,676,288	10,021,888	10,061,722
5.0	12,489,626	12,933,325	12,666,368	9,512,243	9,274,573	8,745,677

Table 3.1. Results of alpha/beta test. The best result is highlighted in green

In Table 3.1, it can be seen that a beta value of 0.1 outperforms the other beta values with a significant margin, however, at a beta value of 0.1 the change in alpha value does not lead to any significant change.

Using the results in Table 3.1, policies were trained using a beta value of 0.05 and 0.15 for both alpha values. These policies are used to test more granular steps.

α\β	0.05	0.1	0.15
2.5	12,871,270	12,981,760	10,617,242
5.0	13,336,883	12,933,325	11,917,414

Table 3.2. Results of alpha/beta test with short increments from 0.1. The best result is highlighted in green

Table 3.2 shows that lowering the beta value to 0.05 yields a significant improvement for an alpha value of 5.0, although it gives a slightly lower score for an alpha value of 2.5. From this, it is determined that the best combination of alpha/beta values is $\alpha = 5.0$ and $\beta = 0.05$. However, it is important to note that these results are only used to estimate the best combination. The following factors can be causes of inaccuracies for the estimate: The numbers tested and the granularity of the testing step size was based on hand-picked numbers gathered from some initial testing. The amount of tests taken may not have been sufficient to get a good estimate of their true average. Only one policy was trained for each combination. As PPO is an on-policy RL algorithm, it is prone to getting stuck in local maxima, this means that a combination may not show its true potential in this test. While these factors have been considered as potential causes of error, it was decided that attempting to improve on these aspects would be too time inefficient for achieving a better estimate.

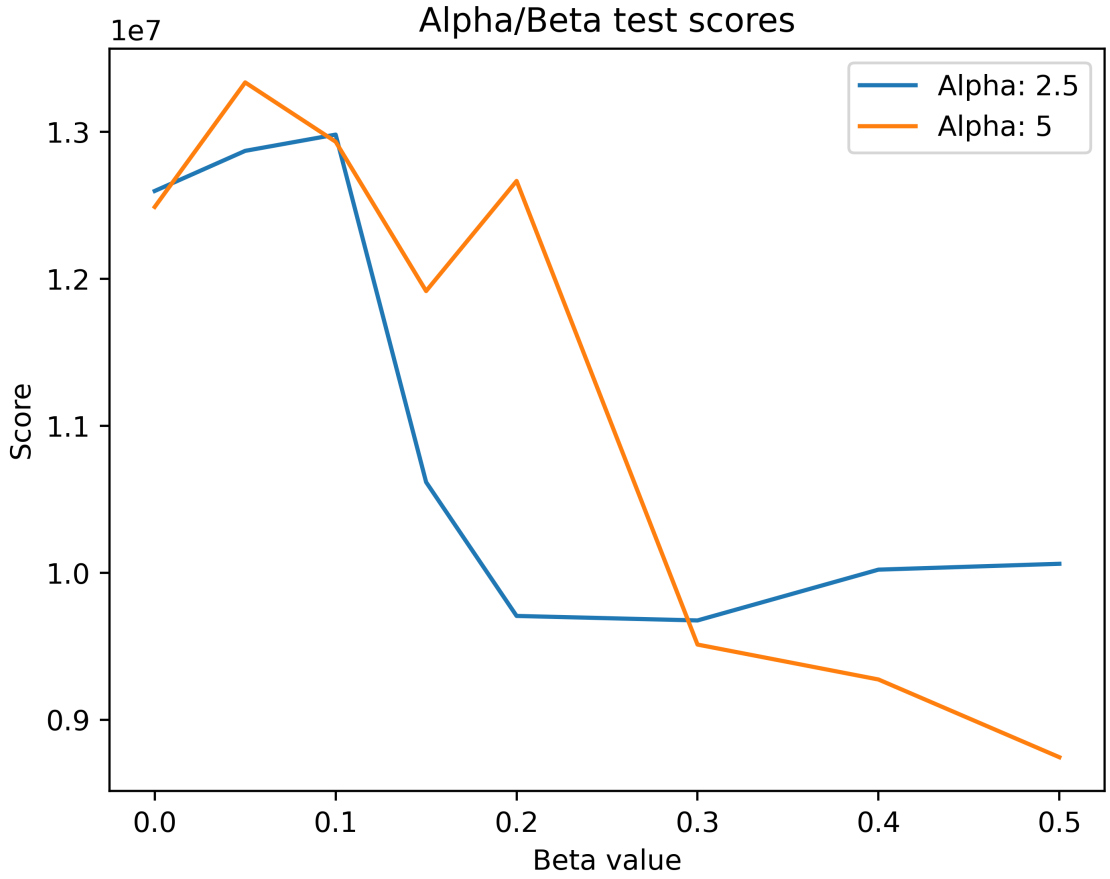


Figure 3.9. Results of the alpha/beta test

Once plotted out, the general trajectory can be seen for the plots. Both alpha values start out high, then increase until a beta value of 0.1 where they then decrease. From this plot, it can be concluded that by using the RTT as a negative segment for the reward function better performance can be achieved, however, the influence of this segment must be small. This behavior could be caused by the following reasons: It may be that a higher beta value makes the agent prioritize a low RTT value over a high throughput, leading the results in Tables 3.1 and 3.2 to show a lower amount of bytes sent that were acknowledged as it attempted to keep the RTT of the packets down. It may also be caused by higher beta values resulting in most actions yielding negative rewards during training. This would mean that the policy would rather take actions that it has never tried before, as the expected reward of those actions would be 0, leading to the policy taking rather random actions. Despite the reward graphs indicating that the policies had converged, it may be that with more training more actions can be explored and the randomness becomes less pronounced.

4 | Implementation

This chapter will contain descriptions of the implementations of the various elements used to create and train ReLoC. These elements have mainly been implemented in Python due to its fast implementation speed and the large amount of support for RL and ML from libraries. One such library is TensorFlow[35]. TensorFlow is a Python library and framework with the support necessary to create a DRL network and a supporting PPO implementation.
While the agent itself is implemented using the TensorFlow library, the implementation of the training loop and supporting functionalities such as observing the state and calculating a reward of the environment as well as performing actions is implemented independently as Python code, allowing full control over these elements of training and eventual usage of the agent.
The environment itself is implemented using a network simulation tool, NS-3, which, unlike the rest of the code, runs in C++, and will therefore need additional support for communication between itself and the main Python implementation.

4.1 Platform for training and validation of performance

As mentioned in Section 2.5, this project will be utilizing a simulated network. This network simulation is used for training the agent and validating its performance.

We created the platform used for training and testing an RL CCA in a previous semester. The key details about the design and implementation will be retold throughout this section. Further information regarding the platform and validation of its functionality can be found in our 9th semester project report [36].

Interface
For our 9th semester project [36] we created a system containing two subsystems. The system consists of an RL platform and a network simulator, visualized in Figure 4.1. The goal of that project was to create an interface between the two subsystems, that would allow an RL CCA to be trained using a network simulator.

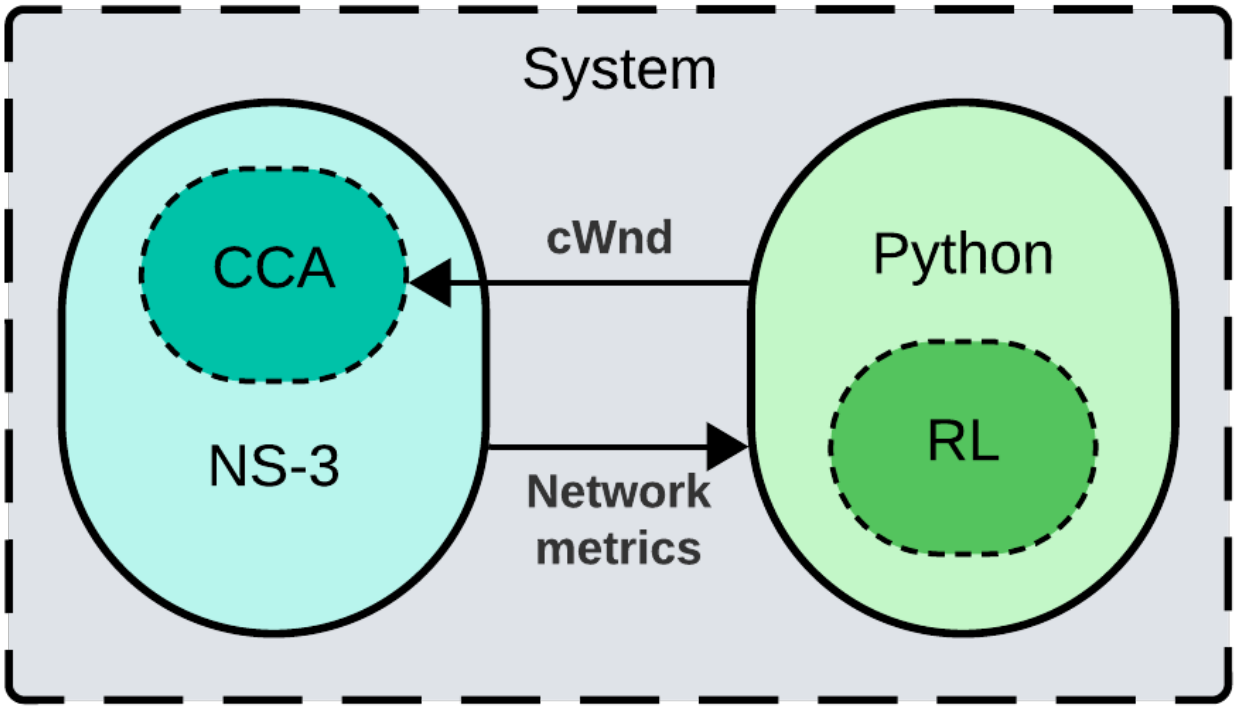


Figure 4.1. Design overview of the test and training platform [36]

For the implementation, we used NS-3, version 3.40 [37], as the network simulator and a Python environment for the RL platform. Through the interface, we are able to send relevant network metrics from NS-3 to the RL platform and change the cWnd in NS-3 from the RL platform. As the interfacing creates overhead in calculating and updating the cWnd, the simulation is paused during this process. This does, however, mean that our implementation assumes that the cWnd can be calculated and updated in an instant.

Platform

The RL platform was developed to be the main process. This process can then spawn an NS-3 simulation as a subprocess using the '*subprocess*' Python module [38].

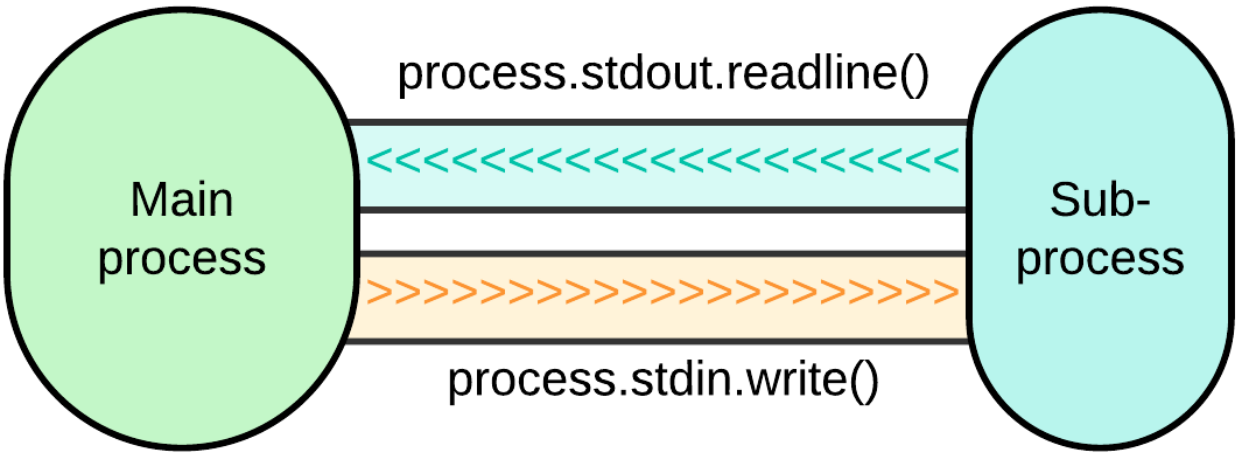


Figure 4.2. Information flow between the RL platform and the network simulator [36]

After spawning the subprocesses, the main process is able to communicate with the subprocess using the '*process.stdout.write()*' and '*process.stdout.readline()*' functions, as can be seen in Figure 4.2. With these functions, the main process can control the subprocess by sending commands and read the command-line outputs from the subprocess. Through these communication streams, the main process is able to retrieve the state of the network simulation for the agent and perform the action decided by the agent.

Network simulation

In order to create a CCA that, in theory, can run on a machine connected to a real network, it is important that the agent is not fed any information that is not available for a user outside of the simulated environment. In the simulated environment, there is full control over the network parameters and access to information throughout the entire network. A machine that is connected to a real network would not be able to access such information. For this reason, what information is gathered and fed to the agent during the development of ReLoC will need to reflect what can be accessed during normal communication on a real network. Therefore, the network simulator only prints data measured by the agent node. The hidden information used for the reward function is known values that are specified and handled on the RL platform, this means that testing does not require any difference in the simulation structure.

We created the network to be a simple dumbbell structure, as can be seen in Figure 4.3. This type of typology is limited by a bottleneck, which can be exploited to introduce congestion at a single point. This implementation introduces congestion by using the other nodes to send UDP packets. Since UDP packets do not have any form of congestion control and do not retransmit dropped packets, they are useful to introduce controlled bursts or continuous flows of congestion into the channels.

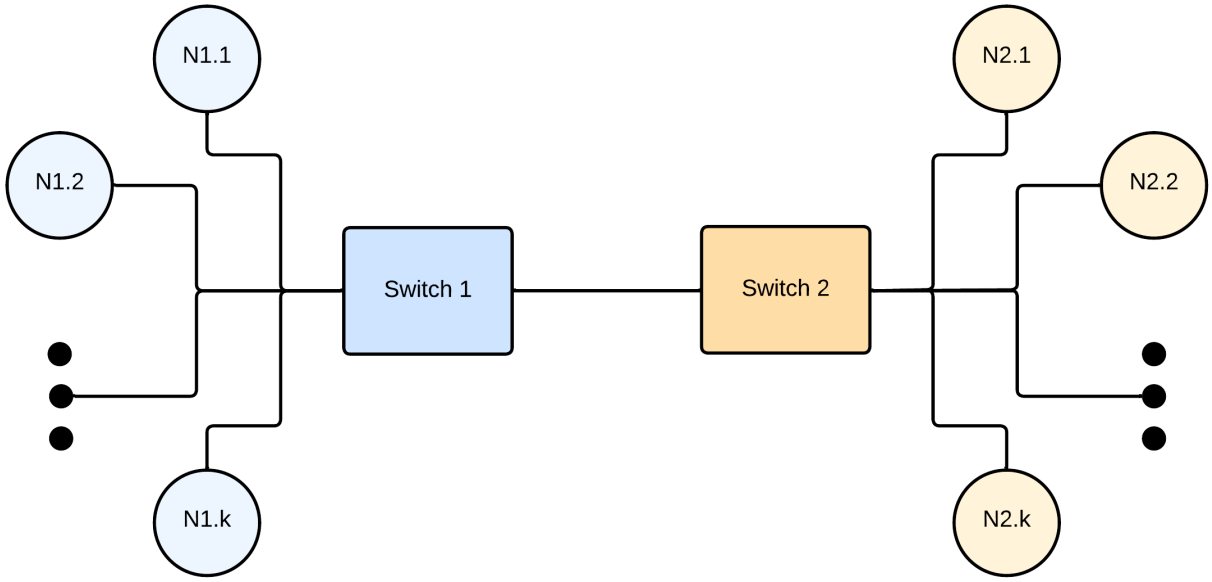


Figure 4.3. Dumbbell typology [36]

We created the simulation so that it is easy to add nodes on each side of the dumbbell. One node is selected as the agent node, which is then set to record the various network metrics that are used as the state. By measuring the network metrics locally from the node, it is guaranteed that it does not relay any information that would not be available on a real-world network.

The data link of the bottleneck was created as being full-duplex. While nodes are able to communicate freely, throughout training and testing, sender nodes will always be on the same side of the dumbbell, while the receivers will always be on the other side. This means that congestion will build in the channel dedicated for communication towards the receivers while the channel dedicated for communication towards the senders will effectively be reserved for ACK packets. While ACK packets do add to congestion during transmission, which may also add to the RTT of the original packets, their effect on congestion has been deemed insignificant in this project due to their small size leading the channel they are communicated over never becoming congested. By this, it is to be understood that introducing further congestion to the channel utilized by the ACKs will not be considered, furthermore, it is assumed that congestion would not significantly delay the ACKs.

4.2 Deep reinforcement learning agents

This section contains details on the implementation of the DRL agents that serve as the custom CCA. First, the general structure and elements of the agents will be outlined, whereafter the individual differences in network architecture will be specified.

There are numerous elements of the implemented agents, most of which are identical between the different implementation variations described in Section 3.3.1. As the agents are designed around the PPO algorithm, it includes both an actor and a critic function. These functions are implemented as DLSTM networks, which will be further described in Section 4.2.1. In addition to an actor and a

critic, an agent also contains a memory class. This memory class is used to store all learning-relevant information such as states, action probabilities, chosen actions, and rewards. This information is stored for multiple steps until enough for a batch is collected. The memory class can then gather all the stored data and give it to the agent in the form of a batch of training data that the agent can then learn from.

The primary function of the agent is the ability to use a policy to select an action when given the simulation state, also known as an observation, as input. This action selection is performed by the agent's method called '*choose_ action()*', which is seen below in Listing 4.1. In this listing, it can be seen that the parameter '*observation*' is first preprocessed after which the actor NN class is used to attain probabilities of each action being taken. Based on these probabilities a TensorFlow distribution is made from which a single sample is taken. This likelihood of the sample being each action is equal to the probability of the action used for the distribution. Using this, a random action is chosen. The last step, along with the logarithmic probability of the chosen action being fetched, is using the critic function NN class to predict the value of the state. The function then shapes the variables in the desired shapes and returns them as the end of the action selection process.

```
1     def choose_action(self, observation):
2         state = tf.convert_to_tensor([observation])
3 
4         probs = self.actor(state)
5         dist = tfp.distributions.Categorical(probs)
6         action = dist.sample()
7         log_prob = dist.log_prob(action)
8         value = self.critic(state)
9 
10        action = action.numpy()[0]
11        value = value.numpy()[0]
12        log_prob = log_prob.numpy()[0]
13        return action, log_prob, value
```

Listing 4.1. The '*choose_ action()*' method of the agent

The other functionality that the agent offers is the '*learn()*' method. When called this method makes the agent retrieve all the stored information in the memory class and use it as a batch for learning. The use of the loaded batch is to calculate the advantage of each time-step using the Generalized Advantage Estimation (GAE)[39]. Following this, the states, actions, and probabilities of taking said action at the time of selection are converted to tensors for further processing. The actor is then used to calculate the logarithmic probabilities of using any action using the same approach as used in the '*choose_ action()*' method. These probabilities are then used to calculate a ratio between new and old probabilities, which together with a clipped version of it is multiplied with the previously calculated advantage to yield weighted probabilities and clipped weighted probabilities. Both these weighted probabilities are then used to calculate the actor loss. The critic loss is calculated by first using the critic to estimate the critic value of the state. Then the return is calculated by adding the advantage and value together. The loss is then calculated using the Mean Squared Error (MSE) loss function found in TensorFlow on the critic value and the return. TensorFlow can use the losses and each of the actor's and critic's NNs to calculate the gradients for the network. An optimizer then applies these gradients to the individual networks as the last step of the learning.
This learning process will be repeated equal to the value of the '*n_ epochs*' variable, which means the

actor's and critic's predictions will vary more from the original predictions used in the batch.

4.2.1 Network architecture

This section describes the implementation of the NNs of the critic and actor for both the base implementation of ReLoC and its two variants.

The base structure of the NNs follows the designed structure in Section 3.3.1. The design specified multiple LSTM layers followed by fully connected layers, and two of each have been chosen to serve as the base implementation of ReLoC. There have been made no tests in regards to the ideal layer configuration or number of layers, excluding the two variants of ReLoC's base implementation. Instead, this number was chosen as it provides the minimum structure for a proof of concept implementation. This means that Figure 3.2 and Figure 3.3, while only being designed to show the general layering of the variants, have the same layers as the implemented versions. The implementation of ReLoC uses a total of 15 time-steps, limiting the memory to a 15-second time frame. A diagram of the base variant of ReLoC can be seen in Figure 4.4.

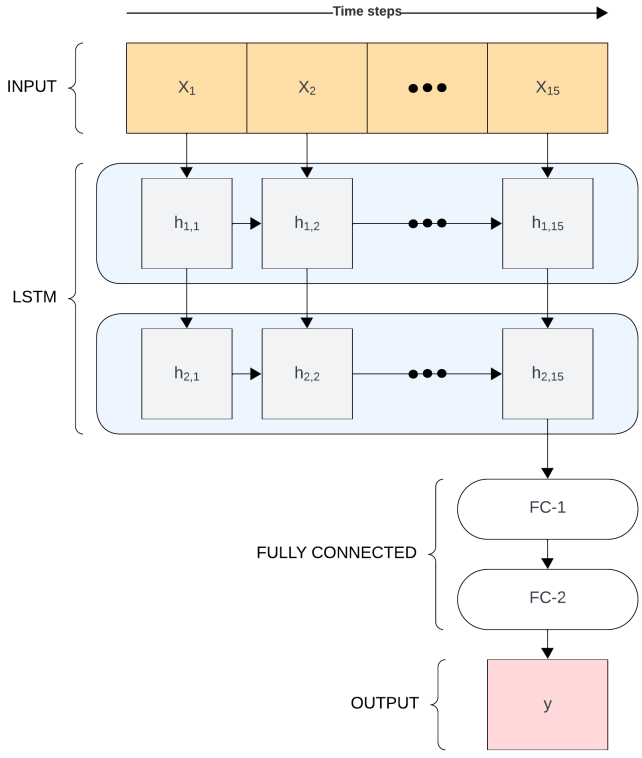


Figure 4.4. Network architecture of ReLoC-Base implementation

With the layers of ReLoC selected, the implementation of an NN can be made using TensorFlow's '*keras.Model*' class as a base for the actor and critic classes, and for the implementation of the individual layers using TensorFlow's '*keras.layers*' submodule. The usage of these can be seen in the implementation of the actor in Listing 4.2, where the '**Model**' class is used when defining the *ActorNetwork* class. To implement the layers, two instances of LSTM layers were defined on lines 7

and 8 in Listing 4.2. The first parameter given in the definition is the number of units in the layer, meaning that 64 units were used for both LSTM layers. Furthermore, an activation function in the form of ReLU was used to allow for non-linear relations to be created. However, only the first LSTM layer takes a '*return_sequences*' parameter different from the default, as it needs to return an output for every time-step for the following LSTM layer to work properly. The fully connected layers are implemented using the '*Dense*' layer. This layer also takes the parameter of units and activation function, and here the first layer receives the high value of 256 units and the ReLU activation function. The second fully connected layer, however, needs a number of units equal to the amount of desired outputs. The actor must return a probability for each action, therefore, the output is set as a variable of the number of actions. In addition, the activation function specified is the '*softmax*' function, which ensures the output is between 0 and 1.
With the layers defined, they can be applied in the mentioned order to any state input when the '*action*' method of the class is called, as seen on lines 12 through 18 in Listing 4.2.

```
1  class ActorNetwork(keras.Model):
2      def __init__(self, n_actions, **kwargs):
3          super(ActorNetwork, self).__init__()
4
5          self.n_actions = n_actions
6
7          self.lstm1 = layers.LSTM(64, activation='relu', return_sequences=True)
8          self.lstm2 = layers.LSTM(64, activation='relu')
9          self.fc1 = layers.Dense(256, activation='relu')
10         self.fc2 = layers.Dense(n_actions, activation='softmax')
11
12     def call(self, state):
13         x = self.lstm1(state)
14         x = self.lstm2(x)
15         x = self.fc1(x)
16         x = self.fc2(x)
17
18         return x
```

Listing 4.2. Implementation of the actor network of the agent

The implementation of the Critic class is almost identical to that of the Actor class. This is to ensure both Networks are able to identify the same level of relations in the input, which allows decisions to be made. The difference between them lies in the second fully connected layer, which is defined on Line 7 of Listing 4.3. In comparison to the Actor class implementation, seen in Listing 4.2, the critic requires only one output and therefore only has one unit. In addition to this, the output does not need to be in the form of a probability, so an activation function is not utilized.

```
1  class CriticNetwork(keras.Model):
2      def __init__(self, **kwargs):
3          super(CriticNetwork, self).__init__()
4          self.lstm1 = layers.LSTM(64, activation='relu', return_sequences=True)
5          self.lstm2 = layers.LSTM(64, activation='relu')
6          self.fc1 = layers.Dense(256, activation='relu')
7          self.q = layers.Dense(1, activation=None)
8
9      def call(self, state):
10         x = self.lstm1(state)
```

```
11          x = self.lstm2(x)
12          x = self.fc1(x)
13          q = self.q(x)
14          return q
```

Listing 4.3. Implementation of the critic network of the agent

For the implementation of the two ReLoC variants, ReLoC-TD and ReLoC-Skip, modifications based on the implementation of the Actor and Critic classes can be seen in Listing 4.2 and Listing 4.3.
For ReLoC-TD, the TensorFlow '*keras.layers.TimeDistributed*' layer was used before the LSTM layers in both the Actor and Critic models. This layer was given a '*Dense*' layer with 64 units and a ReLU activation function as its parameter, allowing it to apply the '*Dense*' layer to all time-steps individually before being used by the LSTM layers.
The ReLoC-Skip variant does not require the addition of an extra layer, but on the other hand simply an additional operation in the '*call*' method of both the Actor and Critic classes. This operation is TensorFlow's '*concat*' function, which is between the last LSTM layer and the first fully connected layer to concatenate the output of the LSTM layer with the last time-step's values of the original state. This allows the first fully connected layer to have an input consisting of the output of the last LSTM layer plus part of the original input.

With this, the agent is fully implemented with an Actor and Critic class, which has the ability to learn based on stored training data. This includes the base implementation of ReLoC as well as its two variants.

4.3 Reinforcement learning loop

This section describes the implementation of the iterating loop that runs over RL aspects such as state observations and actions application.

For RL to take place, the necessary variables and objects must first be initialized. This is first done by defining a large amount of variables in the form of hyperparameters used for the training. Not only that but the agent itself must also be initialized before the iteration begins, where some of the hyperparameters are utilized for the initialization of the agent. These hyperparameters include, but are not limited to, action space, state space, learning rate, and batch size.
Another object initialized pre-iteration, is the '*normalizer*', which serves to normalize the state observations made during training. This happens outside the episode loop to allow the normalizer to remember the limits of state values between episodes. Using a normalizer allows the agent to respond in a more general manner and avoid overfitting of the agent's actor and critic networks. This normalization uses min-max normalization and is done in pairs of two. As an example, the first pair in the state space is the number of sent and received packets. This means if the highest number of packets sent is 100, while the highest number of packets received is 80, both will be normalized based on the 100 packets sent. This is to maintain the relation between the two state variables. The second pair normalized is the average RTT and the standard deviation of the RTT.

When initializations are completed the episode training loop can be entered. This loop iterates through episodes of training and runs a total of 400 times when training a model. At the beginning of each episode, the network simulation is started as a subprocess and the cWnd is set as a random value

between 1 and 20 times the packet size. This randomized cWnd serves to randomize initial conditions, to avoid the agent from always taking the same actions throughout the start.
The network simulation used for training is the same as the network simulation seen during the training of the agent for the alpha/beta test. This training simulation can be seen in Figure 3.7 in Section 3.4. However, as the scenario in the simulation is deterministic, it makes the agent prone to over-specializing instead of learning in a generalized manner. To solve this, two training simulations will be used to train two different agents. This aforementioned simulation will be referred to as the deterministic training simulation. A second training simulation will be introduced and referred to as the random training simulation. These simulations will share the general structure of being split into five phases of 80 seconds each, where phases 1 and 5 are at 100% capacity, however, while the deterministic simulation is reduced by 25% capacity in the second phase and 50% capacity in the third. The random simulation will be done using two randomly chosen reductions in capacity. These two reductions will also be added together to determine the reduction of the capacity during the fourth phase, similar to how the fourth phase in the deterministic simulation is. E.g. a simulation could have a reduction in the available capacity of 62% in the second phase and 15% in the third phase. This would result in the fourth phase having a reduction in capacity of 77%.
The two agents trained from these simulations will be compared in the test section to evaluate the viability of each approach.
After starting the simulation an initial state will be extracted from the simulation, which is normalized by the '*normalizer*' object. With this, the training throughout time-steps can begin.

The loop iterating over time-steps will run through 400 evenly sized steps each episode, and given the simulation is 400 seconds, each time-step spans over 1 second. Because of the 1:1 ratio between time-steps and simulation time in seconds, the two concepts will be used interchangeably throughout the rest of the report.
The first event of each time-step is to calculate action probabilities and choose an action based on these probabilities along with the value of the state. These probabilities are calculated using the agent's '*choose_ action()*' method described in Section 4.2 with the last 15 time-steps normalized state information used as input parameter, as the time-step size used for the implementations is 15. The chosen action is then performed on the cWnd of the network simulation, which after one second of simulation time will return new state information and a reward score. The state, action, action probabilities, state value, and the reward for this step will then be stored in the agent's memory class, mentioned in Section 4.2.
The last part of the step-loop is checking if an entire batch of data has been collected in the agent's memory class. The batch size used for the implementation of the mentioned agents is 64. Thus, if 64 steps of data are in the memory class, the agent's '*learn()*' method will be called, which learns on the stored data as described in Section 4.2. If 64 steps of data are not available, the next iteration of the step-loop will begin.

Once all 400 steps of an episode have been completed the code exits the step loop and closes the subprocess running the network simulation. The rewards received over the entire episode are then summed and saved as a score for that episode. This score is used to calculate a moving average of the scores over the last 30 episodes. The episode with the highest moving average is considered the best-performing iteration of the agent and will have its actor and critic networks saved. This saved actor-critic pair will be used during the evaluation of the implementation's performance in the following test chapter.

With this, a fully functional RL agent capable of controlling the cWnd of a node can be trained using the implementation described in the sections above. The trained agent can be of any of the three described agent implementations, base ReLoC-Base, ReLoC-TD, or ReLoC-Skip. In addition, two versions of each agent can be trained, by using either the deterministic or random training simulation network. Then, the iteration of the agent, which has the best moving average score across episodes, will be saved for later evaluation.

Part III

Evaluation

5 | Tests and Results

Throughout this chapter, a variety of tests will be conducted and their results presented. The tests consist of various network scenarios meant to test certain interactions using a ReLoC agent. Each test and their purpose are presented in Section 5.1. All three versions of ReLoC are tested and compared to each other and TCP Cubic.

5.1 Test description

To evaluate the performance of ReLoC, various tests have been constructed, testing its capabilities on network scenarios for both being the only TCP connection on the network and sharing the network with another TCP connection. As a benchmark for performance, a node using TCP Cubic has also been tested in replacement of the ReLoC agent. TCP Cubic was chosen as the reference algorithm due to its popularity and high performance. To determine the performance between the two algorithms, the cumulative amount of acknowledged bytes sent[1] and average RTT are used as the performance metrics. The various tests will be carried out over 400 simulation seconds unless specified otherwise. Throughout this simulation, the available capacity can change to various degrees, as specified by the test descriptions. Furthermore, for all tests, the maximum available bandwidth over the bottleneck is set to 120kB/s, whereas the TCP sender node(s) are set to send with 200kB/s. By sending over capacity, these node(s) are only limited by their CCA and will congest the channel if they are not limited.

Throughout this section, the various tests will be presented with a test description along with a graph detailing the change in available link capacity throughout the simulation and the purpose of the test.

5.1.1 Test 1 - The baseline

For the baseline test, the available capacity will be set to 100% throughout the entirety of the test. This means that the ReLoC agent will have the entire bandwidth of the link available at all times.

Purpose of the test: This test is meant as a baseline test. During this test, it is expected that the ReLoC agent will be able to increase the cWnd until the limit is reached and then stabilize around the limit. This test will confirm whether ReLoC can detect congestion and react to it. Furthermore, as the available link capacity is static, the Kleinrock point of optimality will also remain static. For this reason, the ReLoC

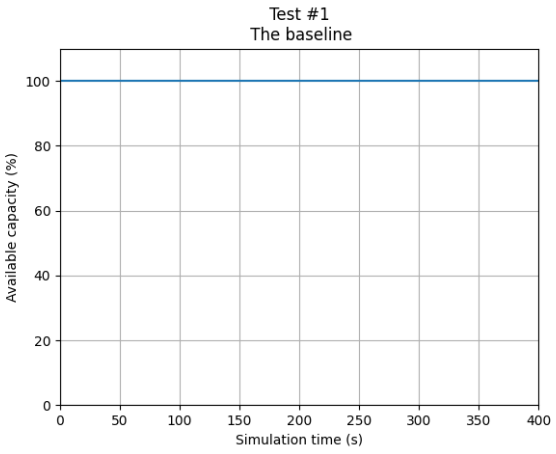


Figure 5.1. Graph showing the change in available link capacity for test 1

[1] Acknowledged bytes refers to the bytes from packets that have received an acknowledgment, i.e. packets that were not dropped.

agent should attempt to find and stabilize around this point of optimality once it has reached the capacity limit. Searching for and stabilizing around the Kleinrock point of optimality should, however, be expected for all tests, not only test 1.

5.1.2 Test 2 - Going down

For test 2, along with all following tests, the available link capacity will start at full capacity. Halfway through the simulation, at 200s, the available link capacity will drop to 50% where it will stay throughout the rest of the simulation.

Purpose of the test: This test is conducted to validate whether the ReLoC agent is able to react to a reduction in the available link capacity and act accordingly by lowering the cWnd to an appropriate size and then stabilizing to this new limit. This test gives insight into both how fast the agent is able to react to the new capacity limit and also how long it will take to reach this limit.

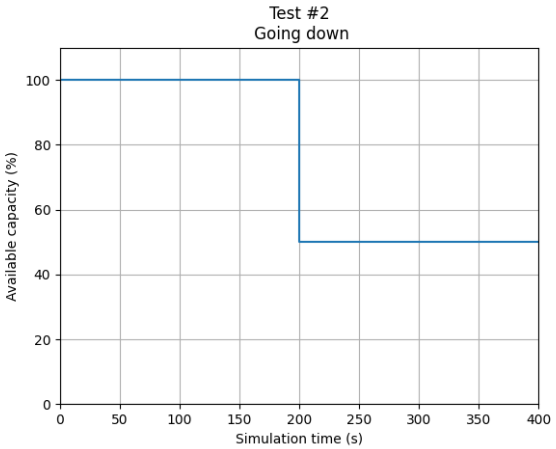


Figure 5.2. Graph showing the change in available link capacity for test 2

5.1.3 Test 3 - Coming back up

For test 3, the available capacity will drop and then increase again. At 130s into the simulation, the available link capacity will be lowered to 50%. After an additional 130s, at time-stamp 260, the available link capacity will be restored to 100%, allowing utilization of the entire bandwidth again.

Purpose of the test: This test is conducted to verify that the agent is not only able to reduce the cWnd when the available link capacity is lowered but that it can also increase the cWnd when the available link capacity is increased.

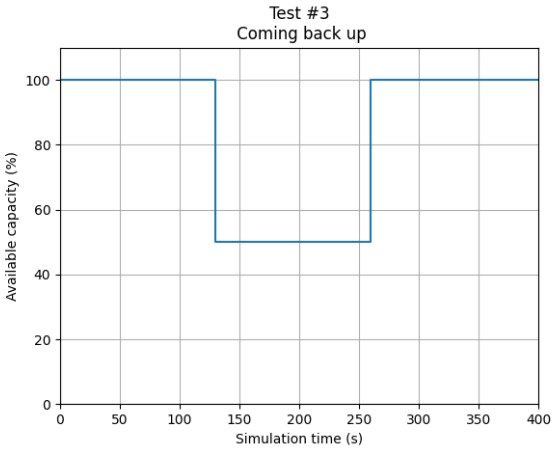


Figure 5.3. Graph showing the change in available link capacity for test 3

5.1.4 Test 4 - Varying capacity

Test 4 is similar to test 3, with the key difference being that the capacity is not restored to the starting value. At time-stamp 130 the available capacity is lowered to 50% like in test 3. At time-stamp 260 the available capacity is then increased to 75%.

Purpose of the test: This test is carried out to see how the ReLoC agent behaves when switching between various capacities. The capacity values for this test could be any different values, as the important part of this test is to confirm that the ReLoC agent is able to switch effectively between values.

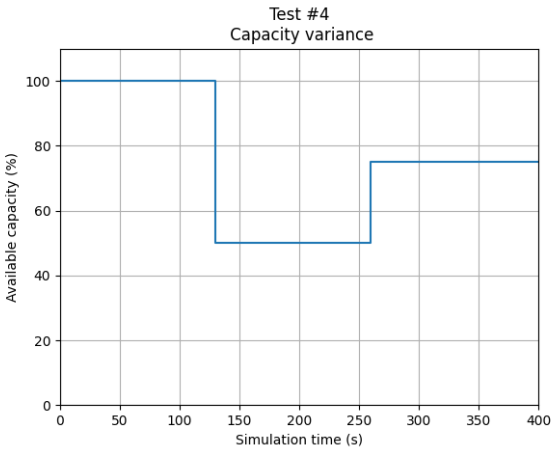


Figure 5.4. Graph showing the change in available link capacity for test 4

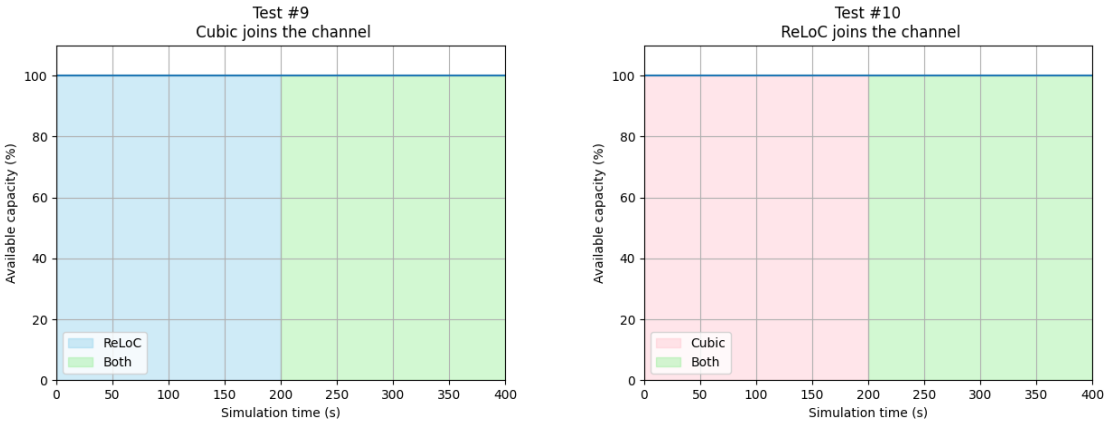
5.1.5 Test 5, 6, 7, and 8 - TCP tests

Tests five through eight are identical to tests one through four but with another node also communicating over the channel using TCP. This node will, however, be using TCP Cubic as its CCA. While the platform cannot interact with this node, it still records and relays data to the platform in the same manner as the ReLoC agent. This is done to gather the local network information from this node for the test results.

Purpose of the tests: These tests are performed to evaluate how the ReLoC agent performs with competing communication on the channel. Furthermore, it is used to determine how fair it behaves when sharing the bandwidth with another transmission.

5.1.6 Test 9 and 10 - Joining mid-transmission

Tests 9 and 10 are similar to each other. Both tests have 100% available capacity throughout the entire simulation, the same as test 1. For test 9 the ReLoC agent will be communicating throughout the entire simulation, where a node using TCP Cubic will join at time-stamp 200. For test 10 the roles are reversed.



(a) Test 9 (b) Test 10

Figure 5.5. Graphs showing the change in available link capacity for tests 9 and 10

The area under the capacity graphs in Figure 5.5a and 5.5b indicate what node is sending at what time. The blue area means that only the ReLoC agent is communicating, the pink area means that only the node using TCP Cubic is sending, and the green area means that both are sending at the same time.

Purpose of the tests: These tests are conducted to evaluate how the ReLoC agent behaves when another node enters communication over the channel and how the ReLoC node behaves when entering a channel where communication is already being conducted.

5.2 Test results - Proof of concept

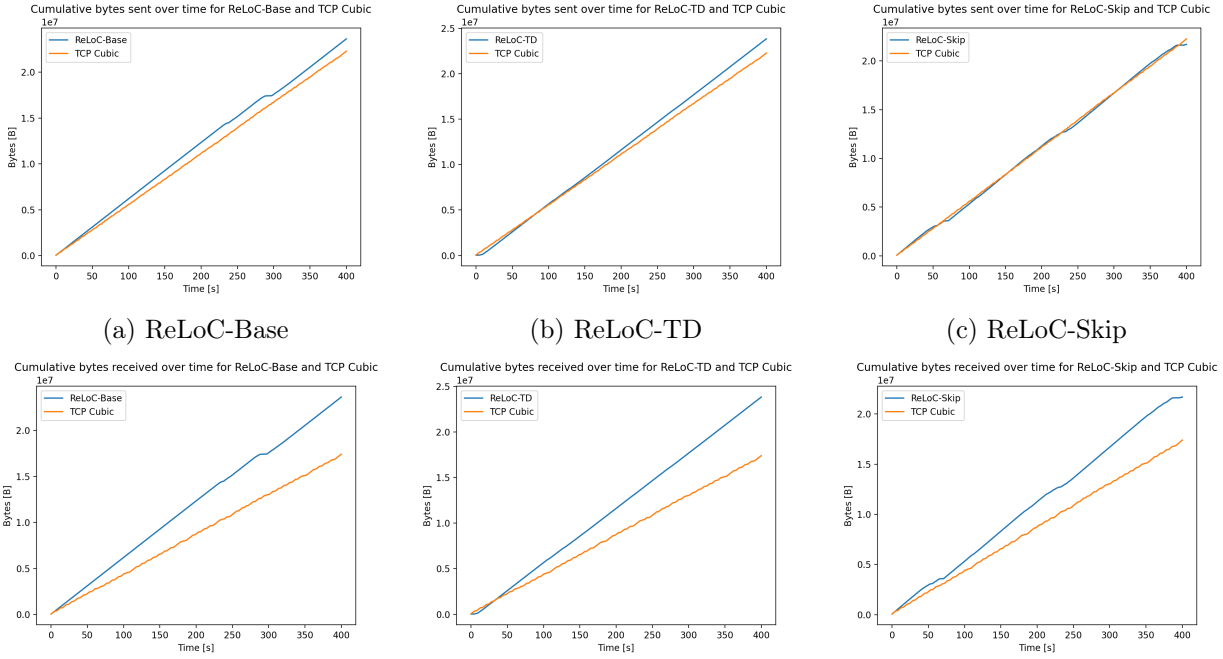
As mentioned in Section 4.3, at first an agent was trained exclusively on a specific network simulation. While this hindered the agent from generalizing, it was done to initially evaluate if the agent is capable of delivering promising results which would indicate that a proper DRL-based CCA can be created using ReLoC, that is to say, it serves as a proof of concept.
Do be aware that the limits of the axes are not confined to specific values between the testing of the different variants. The axes were chosen to be fit to the plot as some plots contain outliers that would cause the other plots to be harder to interpret through visual inspection. Furthermore, especially the plots overlaying RTT and cWnd would be harder to gather good intuition from, as by fitting the axes to the plots, the correlation between RTT and cWnd becomes more clear.

5.2.1 Test 1

Test 1 is the baseline test, where the link capacity is kept at 100% throughout the entire test. The test description can be found in Subsection 5.1.1.

Throughout all the tests, sent bytes will refer to the amount of bytes that are sent regardless if they are received by the receiver. Meanwhile, received bytes will refer to bytes sent by the sender that have been successfully received by the receiver.

Throughput results



(a) ReLoC-Base (b) ReLoC-TD (c) ReLoC-Skip

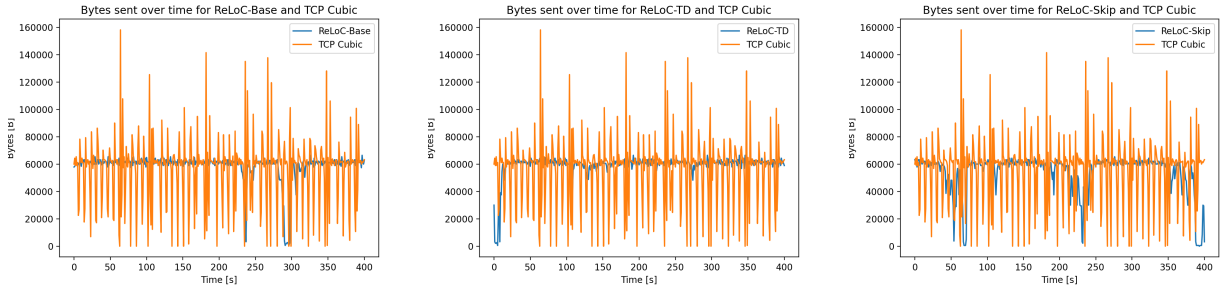
(d) ReLoC-Base (e) ReLoC-TD (f) ReLoC-Skip

Figure 5.6. Visualization of cumulative bytes sent and received throughout the test

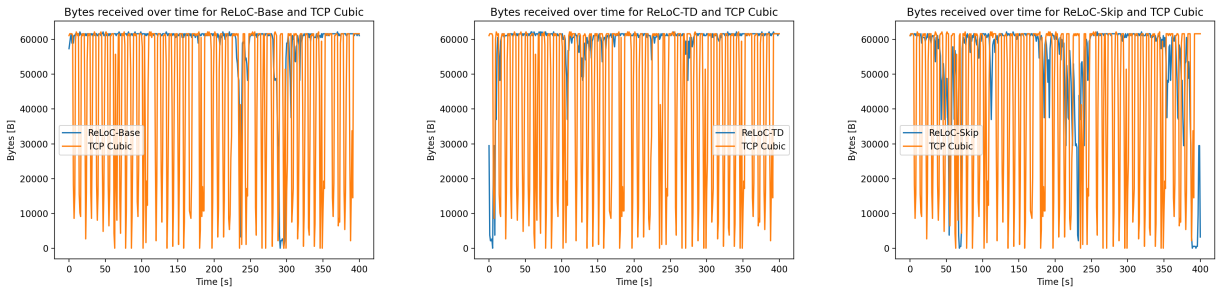
CCA	TCP Cubic	ReLoC-Base	ReLoC-TD	ReLoC-Skip
Cumulative bytes received	17,404,456	23,612,408	23,833,240	21,695,672

Table 5.1. Cumulative bytes received for each CCA

On Figures 5.6a, 5.6b, and 5.6c it can be seen that the different variations of ReLoC send approximately at the same rate as TCP Cubic, however, as can be seen on Figures 5.6d, 5.6e, and 5.6f, the number of bytes that are successfully received are significantly higher than that of TCP Cubic. This means that despite the sending rate being approximately the same, the achieved throughput is greater for ReLoC. When comparing the total amount of bytes received in Table 5.1, it can be seen that ReLoC-Base and ReLoC-TD are able to achieve similar results while ReLoC-Skip underperforms compared to the two other variants.



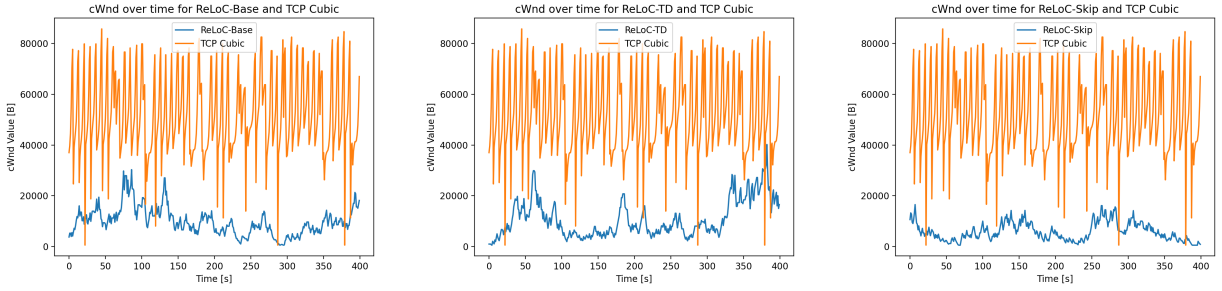
(a) ReLoC-Base (b) ReLoC-TD (c) ReLoC-Skip



(d) ReLoC-Base (e) ReLoC-TD (f) ReLoC-Skip

Figure 5.7. Visualization of sent and received bytes for each time-step

When inspecting the number of bytes sent and received per second, seen in Figure 5.7, some intuition can be gained regarding how the ReLoC CCAs can gain a lot more throughput while only sending a bit more data compared to TCP Cubic. In Figures 5.7a, 5.7b, and 5.7c it can be seen that the ReLoC variants mainly attempt to send at around 60kB/s, with a few drops in sending rate at moments. Meanwhile, TCP Cubic behaves more erratic, which is to be expected, as it probes for congestion. When inspecting Figures 5.7d, 5.7e, and 5.7f, it can be seen that the nodes hit a sending limit at 60kB/s. This is due to the total bandwidth being 120kB/s, but as the link is full duplex, the total bandwidth is split equally for each direction. This limit is what the ReLoC variants were observed to attempt to send at. So the ReLoC variants being able to achieve a significantly higher throughput despite only having a slightly higher sending rate can be explained by TCP Cubic's erratic behavior compared to the ReLoC variants. TCP Cubic often increases its sending rate above 60kB/s while probing for congestion, and below 60kB/s after resetting its cWnd. This leads TCP Cubic to under-utilize the available capacity.



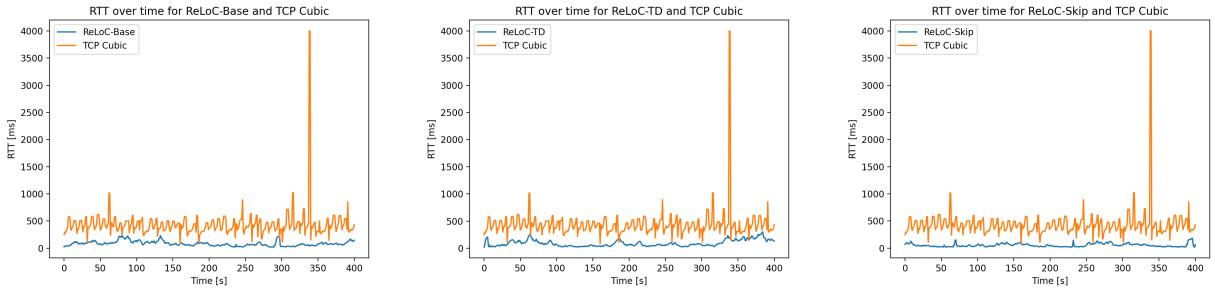
(a) ReLoC-Base (b) ReLoC-TD (c) ReLoC-Skip

Figure 5.8. Visualization of the cWnd value for each time-step

In Figure 5.8, it can be seen that the cWnd values are significantly lower for the ReLoC variants

compared to TCP Cubic. This is a result of the ReLoC variants attempting to hit Kleinrock's optimal operating point. TCP Cubic, however, continuously probes for congestion through loss. This means that TCP Cubic keeps increasing its cWnd beyond Kleinrock's optimal operating point until it reaches the loss-based operating point, as was shown in Figure 3.6. For this reason, TCP Cubic operates with a significantly higher cWnd than the ReLoC variants.

RTT results



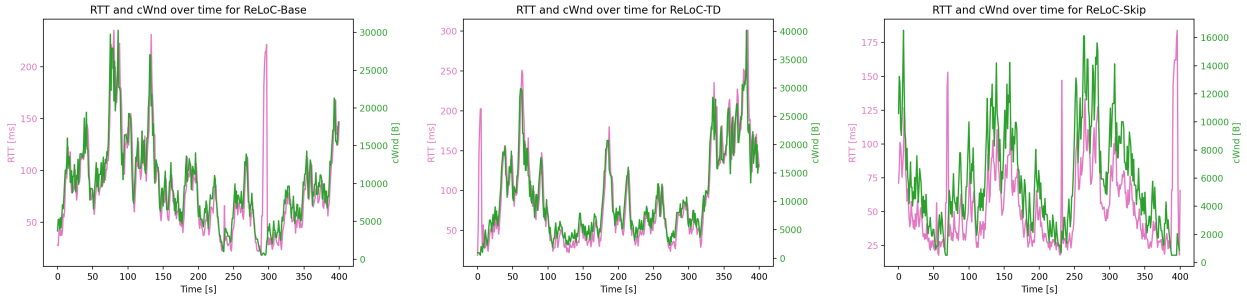
(a) ReLoC-Base (b) ReLoC-TD (c) ReLoC-Skip

Figure 5.9. Visualization of the average RTT in ms for each time-step

CCA	TCP Cubic	ReLoC-Base	ReLoC-TD	ReLoC-Skip
RTT(0,400)	438.47	83.51	90.51	54.55

Table 5.2. Average RTT in ms per segment for each CCA

Table 5.2 shows the average RTT for the different CCAs. The average RTT shown in each test will be split into segments. The segments are divided by the shifts in available capacity changes. As test 1 does not change the available capacity, only one segment is shown in Table 5.2. In the table, RTT(0,400) indicates that it is the segment that starts at time-stamp 0 and ends at time-stamp 400. It can be seen in Figure 5.9b that the RTT for all variants of ReLoC was consistently lower than the RTT using TCP Cubic. This is, however, expected considering TCP Cubic's loss-based probing nature, as discussed in the paragraph above. In Table 5.2, it can be seen that ReLoC-Skip has a significantly lower average RTT compared to the other two variants. However, as was seen before, ReLoC-Skip also has a significantly lower throughput than the other variants. This could be a sign that the agent prioritizes low RTT over high throughput.



(a) ReLoC-Base (b) ReLoC-TD (c) ReLoC-Skip

Figure 5.10. Visualization of the average RTT in ms and cWnd value plotted together for each time-step

When there is no packet loss, the curvature of an RTT plot is expected to reflect the curvature of its related cWnd plot, as these two values are inherently correlated. As can be seen in Figure 5.10, the RTT and cWnd show a high correlation, since there are very few packets dropped. In multiple instances, the plot of one value fits seamlessly over the plot of the other. But what is important is that when the curvature of one value changes between positive and negative, then the same thing happens for the other value, even if the peaks and valleys do not overlap. This is important, as it reflects how a change in the cWnd affects the RTT.

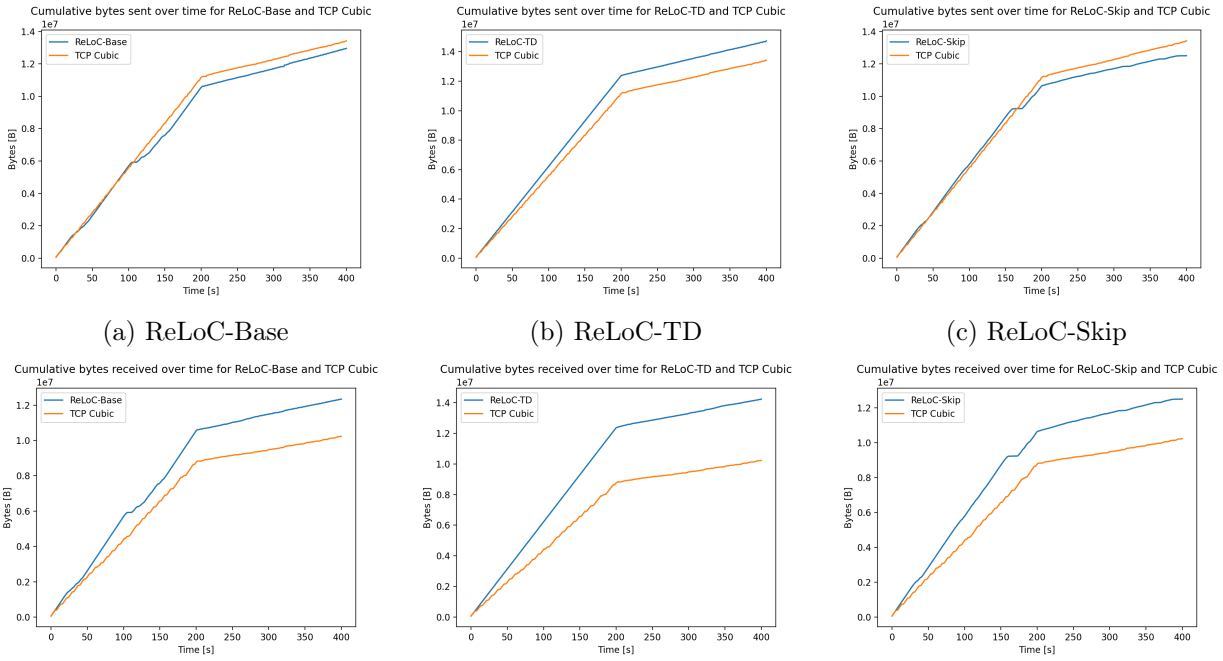
Test conclusion

By inspecting the results of the throughput, it can be determined that all ReLoC variants were able to find the maximum available link capacity and utilize it to a high degree. Furthermore, while utilizing the available capacity, the agents were able to keep the average RTT low, suggesting that they were operating closely to Klienrock's optimal operating point. All variants of ReLoC also proved to outperform TCP Cubic in both performance metrics.

5.2.2 Test 2

Test 2 aims to see whether the ReLoC variants are able to react to a change in capacity. At time-step 200 the available capacity is lowered to 50%. The test description can be found in Subsection 5.1.2.

Throughput results



(a) ReLoC-Base (b) ReLoC-TD (c) ReLoC-Skip

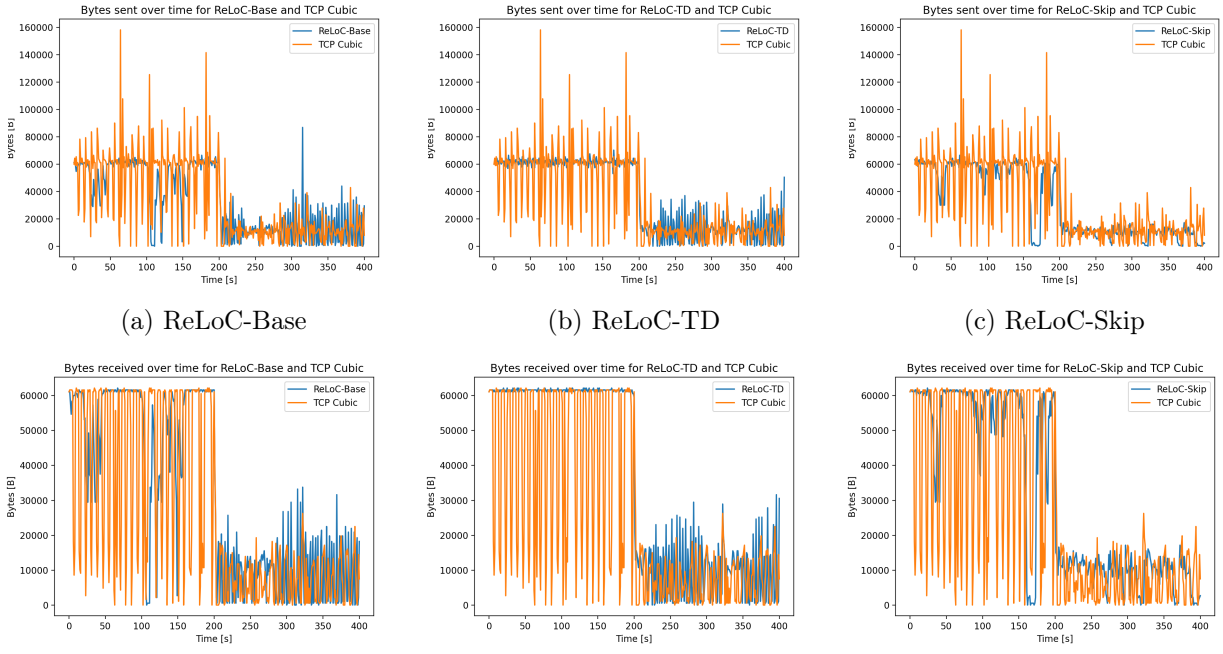
(d) ReLoC-Base (e) ReLoC-TD (f) ReLoC-Skip

Figure 5.11. Visualization of cumulative bytes sent and received throughout the test

CCA	TCP Cubic	ReLoC-Base	ReLoC-TD	ReLoC-Skip
Cumulative bytes received	10,234,920	12,343,008	14,243,128	12,499,520

Table 5.3. Cumulative bytes received for each CCA

Similar to test 1, it can be seen in Figures 5.11a, 5.11b, and 5.11c that the ReLoC variants attempt to send approximately the same amount of bytes as TCP Cubic. But once again, the ReLoC variants are shown to vastly outperform TCP Cubic in actual throughput. However, in difference to test 1, it can be seen in Table 5.3 that ReLoC-Base and ReLoC-Skip perform about equally, while ReLoC-TD significantly outperforms both.



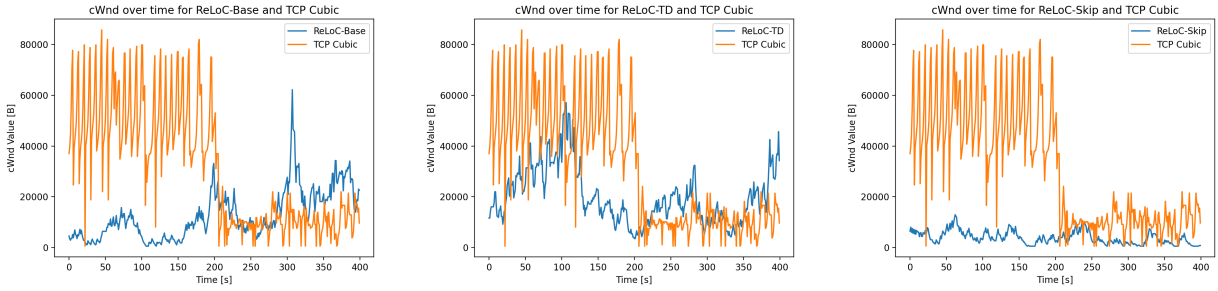
(a) ReLoC-Base (b) ReLoC-TD (c) ReLoC-Skip

(d) ReLoC-Base (e) ReLoC-TD (f) ReLoC-Skip

Figure 5.12. Visualization of sent and received bytes for each time-step

Once again, it can be observed that TCP Cubic behaves erratically and the ReLoC variants are consistently sending at approximately 60kB/s, with some occasional drops in send rate, while at 100% capacity. But once the available capacity is limited to 50%, the ReLoC variants also begin to express a similar erratic behavior as TCP Cubic. This is also visible in Figures 5.11a, 5.11b, and 5.11c, where the slope for each ReLoC variant during the initial segment is visibly steeper than that of TCP Cubic, it changes to be almost identical to the slope of TCP Cubic once the available capacity is lowered. Notably, ReLoC-Skip does not express the same erratic behavior as the other two variants, despite this, it does not significantly outperform the others in this section, as it spends too much time with its sending rate being considerably decreased.

It can be noticed that all variants of ReLoC and TCP Cubic send at most around 20kB/s, and ReLoC-Skip somewhat stabilizes around 15kB/s. This is significantly lower than 30kB/s, which would be expected when halving the available throughput. This problem seems to be a side effect of using UDP streams to limit the available capacity and will be discussed further in Chapter 6. For the test, the exact level the capacity is limited to will not cause any problem, as the limitation will still be equal for all nodes.

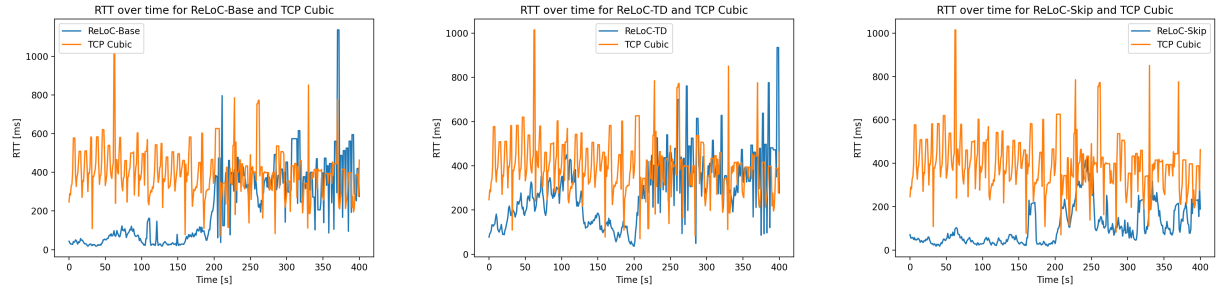


(a) ReLoC-Base (b) ReLoC-TD (c) ReLoC-Skip

Figure 5.13. Visualization of the cWnd value for each time-step

When looking at Figures 5.13a, 5.13b, and 5.13c, it can be seen that each variation of ReLoC behaved quite differently to the test. ReLoC-Base starts out with a relatively low and stable cWnd, as was seen in test 1. However, once the capacity is limited, ReLoC-Base increases the cWnd significantly, which is the opposite behavior of what would be expected. This explains the erratic behavior seen in Figure 5.12d. ReLoC-TD can be seen to have a quite high cWnd throughout the first half section with 100% available capacity, which then falls down to a level more akin to the cWnd seen in test 1. Once the available capacity is dropped, it can be seen that the cWnd is getting pushed up to similar values or above that of TCP Cubic's cWnd value. This once again explains the erratic behavior seen in Figure 5.12e. Finally, with ReLoC-Skip, it can be seen that the cWnd continuously stays low, dropping even further at extended points during the lowered capacity. During these points, the cWnd seems to occasionally drop too far down, leading to the valleys seen between time-stamp 250 to 400 in Figure 5.12f.

RTT results



(a) ReLoC-Base (b) ReLoC-TD (c) ReLoC-Skip

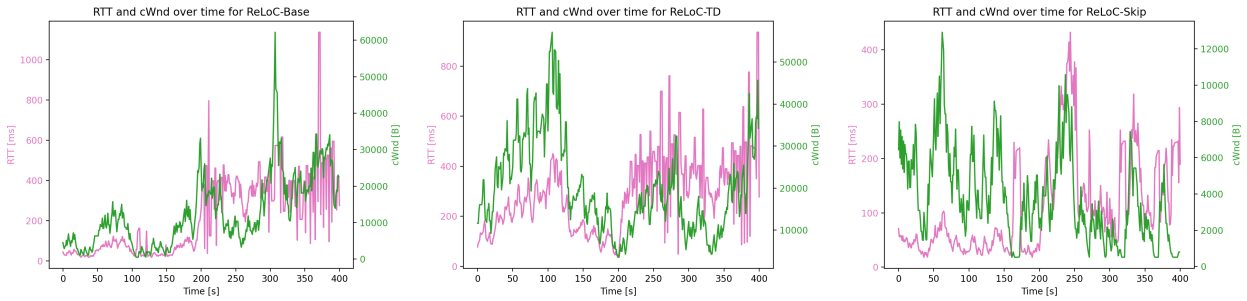
Figure 5.14. Visualization of the average RTT in ms for each time-step

CCA	TCP Cubic	ReLoC-Base	ReLoC-TD	ReLoC-Skip
RTT(0,199)	418.64	59.59	197.88	51.28
RTT(200,400)	384.91	384.00	386.51	167.11

Table 5.4. Average RTT in ms per segment for each CCA

Figures 5.14a, 5.14b, and 5.14c reflect the conclusion reached from the previous paragraph well. Both ReLoC-Base and ReLoC-TD suddenly get a very high RTT as they increase their cWnd dramatically in response to the lowered capacity. Meanwhile, ReLoC-Skip mostly keeps the RTT low. Once again, it

can be seen that ReLoC-Skip has the lowest average RTT out of the different variants, further proving the theory that ReLoC-Skip is more sensitive to high RTT.



(a) ReLoC-Base (b) ReLoC-TD (c) ReLoC-Skip

Figure 5.15. Visualization of the average RTT in ms and cWnd value plotted together for each time-step

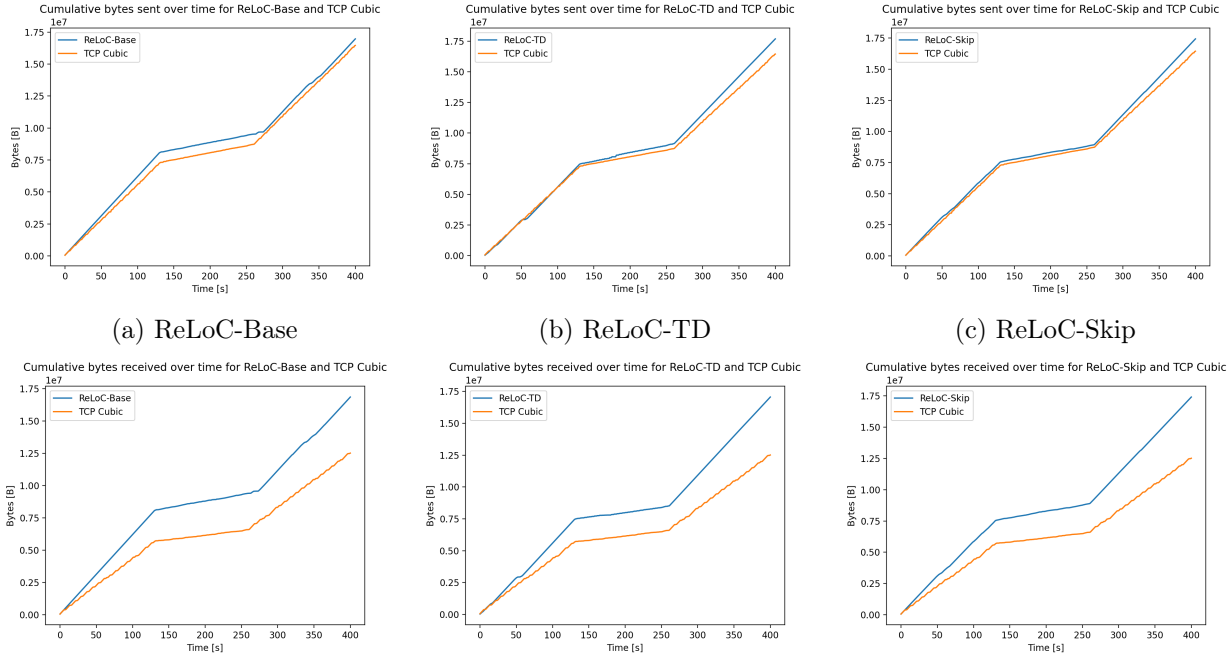
It can once again be seen how the RTT reflects the cWnd, although not as clearly as was shown in test 1. For all three variants, while the available capacity is at 100%, it is quite clear to see how the two plots are correlated to each other, when one value increases, so does the other, and likewise when a value decreases. Once the available capacity drops to 50%, this correlation becomes harder to see for ReLoC-Base, in Figure 5.15a, and for ReLoC-TD, in Figure 5.15b. This is due to the high congestion caused by the high cWnd. This means that the RTT experiences a lot more volatility.

Test conclusion
All variants were again able to outperform TCP Cubic in regards to throughput. They were also able to outperform TCP Cubic in RTT while there was 100% available capacity, however, once the capacity was lowered to 50%, only ReLoC-Skip was able to outperform TCP Cubic. From this test, it can be gathered that all variants are able to outperform TCP Cubic while the available capacity is at 100%, as was also seen in test 1. However, once the capacity is lowered to 50%, the variants are only able to compete approximately equal to TCP Cubic. Despite ReLoC-Skip having a lower RTT than TCP Cubic, it was still not able to significantly outperform TCP Cubic in regards to throughput while the available capacity was reduced to 50%.

5.2.3 Test 3

Test 3 is used to verify that the ReLoC variants are able to react to both decreases and increases in the available capacity. The available capacity is first lowered to 50% at time-step 130, then increased to 100% at time-step 260. The test description can be found in Subsection 5.1.3.

Throughput results



(a) ReLoC-Base (b) ReLoC-TD (c) ReLoC-Skip

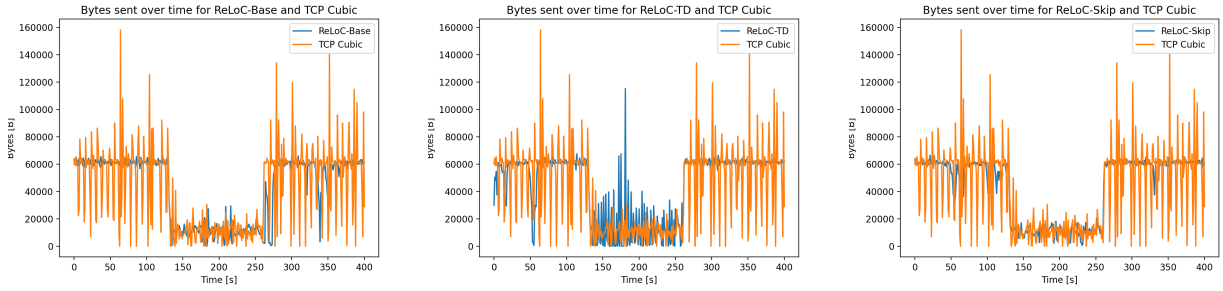
(d) ReLoC-Base (e) ReLoC-TD (f) ReLoC-Skip

Figure 5.16. Visualization of cumulative bytes sent and received throughout the test

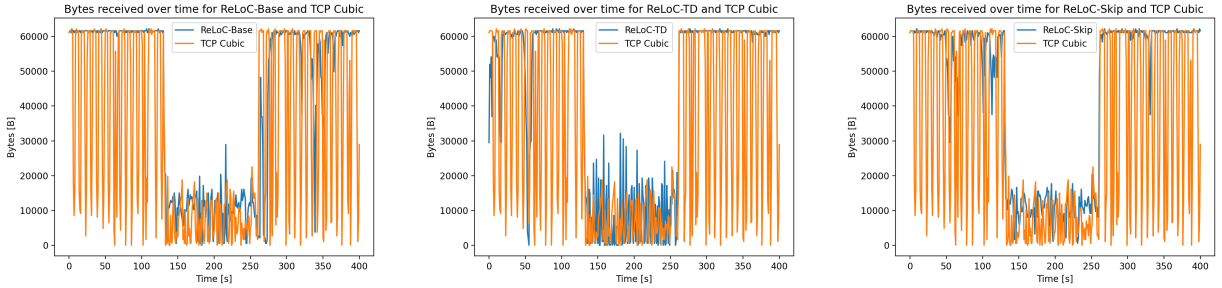
CCA	TCP Cubic	ReLoC-Base	ReLoC-TD	ReLoC-Skip
Cumulative bytes received	12,511,848	16,848,088	17,059,272	17,402,312

Table 5.5. Cumulative bytes received for each CCA

When inspecting the graphs in Figure 5.16, similar observations can be drawn as from test 1 and 2. However, in this test, it can be seen that the three variants perform very similarly in terms of throughput, with ReLoC-Skip being slightly better.



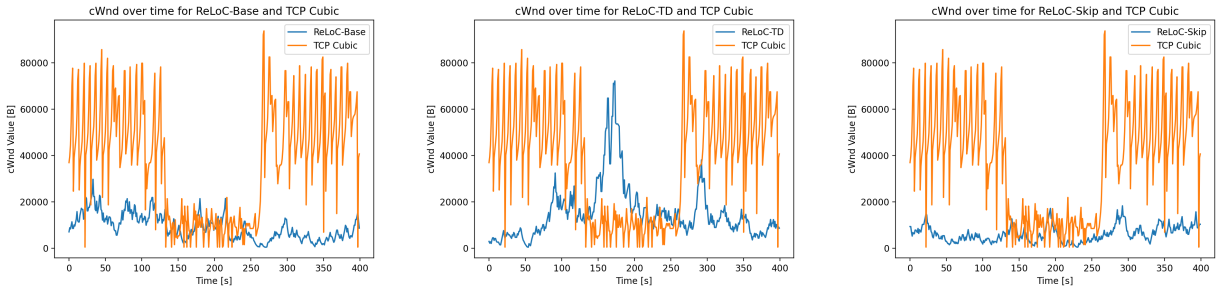
(a) ReLoC-Base (b) ReLoC-TD (c) ReLoC-Skip



(d) ReLoC-Base (e) ReLoC-TD (f) ReLoC-Skip

Figure 5.17. Visualization of sent and received bytes for each time-step

Compared to test 2, ReLoC-Base does not show the same erratic behavior while the capacity is lowered to 50%. However, ReLoC-TD continues to show very aggressive behavior, trying to push as many packets through as possible. When inspecting the Figures 5.17d, 5.17e, and 5.17f, it can be seen that once the available capacity is restored to 100%, all three variants are able to recover to and stabilize at 60kB/s again. This also shows that ReLoC-TD only expresses this aggressive behavior during reduced capacity.

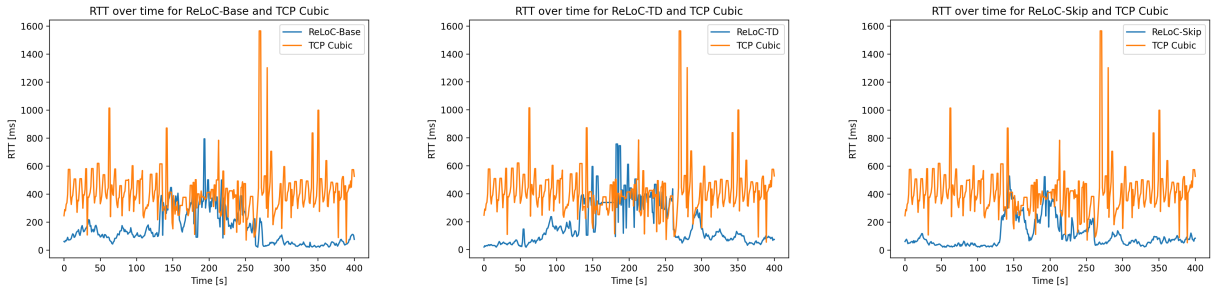


(a) ReLoC-Base (b) ReLoC-TD (c) ReLoC-Skip

Figure 5.18. Visualization of the cWnd value for each time-step

As has been the case in tests 1 and 2, ReLoC-Skip keeps a low cWnd throughout the entire test. ReLoC-Base also keeps its cWnd down, even during the lowered capacity, although the cWnd is on average higher than that of ReLoC-Skip. The behavioral change compared to test 2 is most likely a result of initial values being different and the capacity changing at a different time step. While the behavior in this instance was different than a similar case in test 2, this does not mean that ReLoC-Base behaves unpredictably, but rather that the states observed by the agent lead to action sequences that handled a similar situation differently.

RTT results



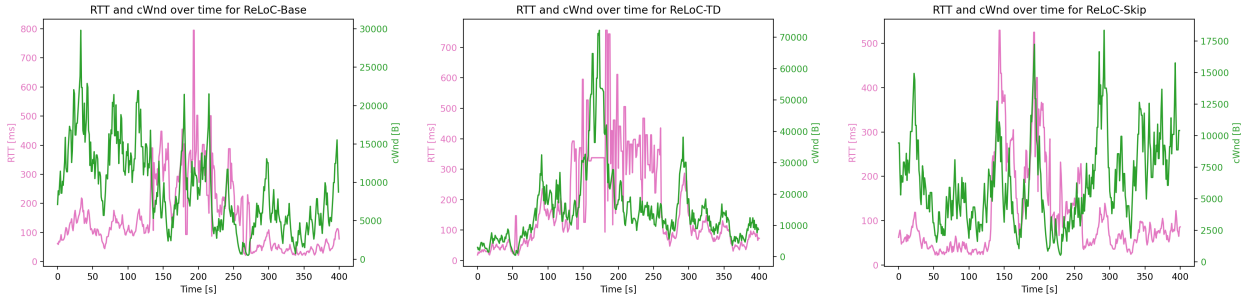
(a) ReLoC-Base (b) ReLoC-TD (c) ReLoC-Skip

Figure 5.19. Visualization of the average RTT in ms for each time-step

CCA	TCP Cubic	ReLoC-Base	ReLoC-TD	ReLoC-Skip
RTT(0,129)	434.13	112.16	85.71	45.47
RTT(130,259)	378.73	292.78	364.14	218.96
RTT(260,400)	449.47	56.91	101.02	69.12

Table 5.6. Average RTT in ms per segment for each CCA

As expected based on the earlier tests, at 100% capacity, all variations of ReLoC have a considerably lower RTT than TCP Cubic. In the case of both ReLoC-Base and ReLoC-Skip, the RTT during 50% capacity is also significantly lower than that of TCP Cubic, however, also significantly higher than during 100% available capacity. In the case of ReLoC-TD, due to the aggressive behavior, the average RTT during 50% capacity is similar to that of TCP Cubic.



(a) ReLoC-Base (b) ReLoC-TD (c) ReLoC-Skip

Figure 5.20. Visualization of the average RTT in ms and cWnd value plotted together for each time-step

Similar to test 2, it can be seen that the RTT reflects the cWnd. Further, it can once again be seen in Figures 5.20a and 5.20b, that during congestion the RTT becomes highly volatile.

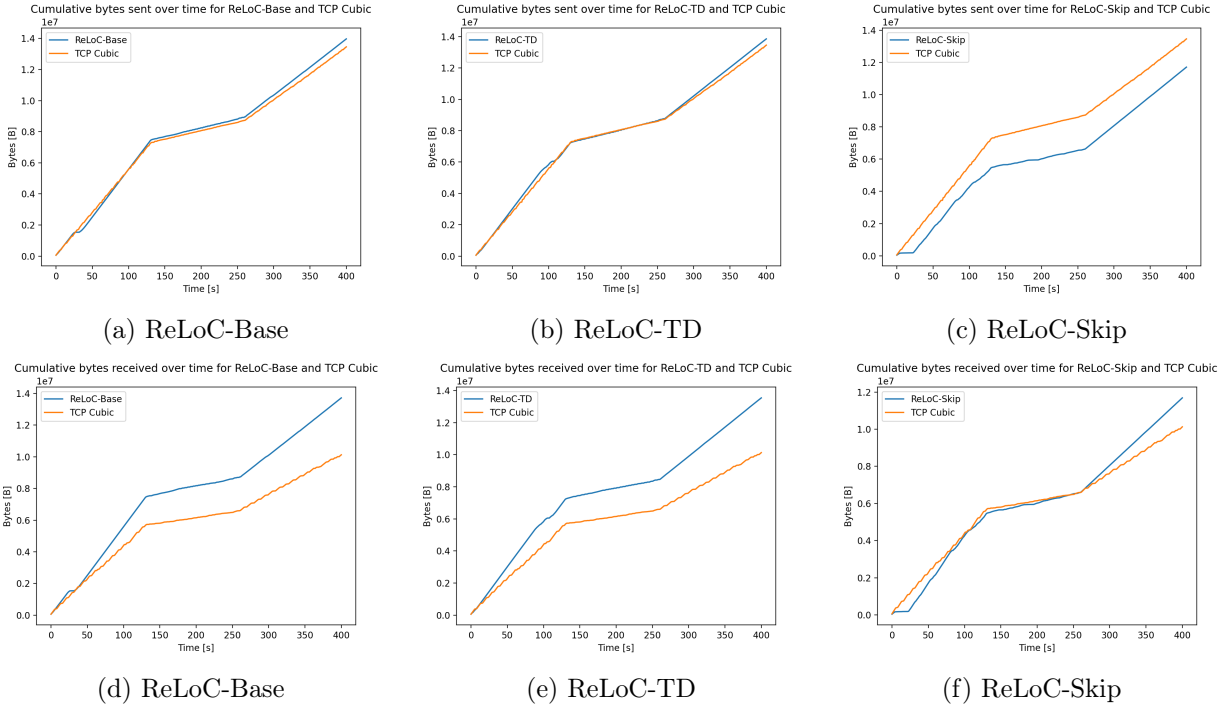
Test conclusion

Test 3 gave similar results to test 2. It was shown that all variants of ReLoC outperformed TCP Cubic during 100% capacity and performed equally in terms of throughput as TCP Cubic at 50% capacity, while both ReLoC-Base and ReLoC-Skip were able to keep a lower average RTT than TCP Cubic. Furthermore, it was shown that once the capacity was recovered back to 100%, all variants were able to increase back to this limit and find a stable sending rate with high throughput and low RTT.

5.2.4 Test 4

Test 4 is used to verify that the ReLoC variants are capable of operating at varying levels of capacity. The available capacity starts at 100%, where it decreases to 50% at time-step 130, and then increases to 75% at time-step 260. The test description can be found in Subsection 5.1.4.

Throughput results



(a) ReLoC-Base (b) ReLoC-TD (c) ReLoC-Skip

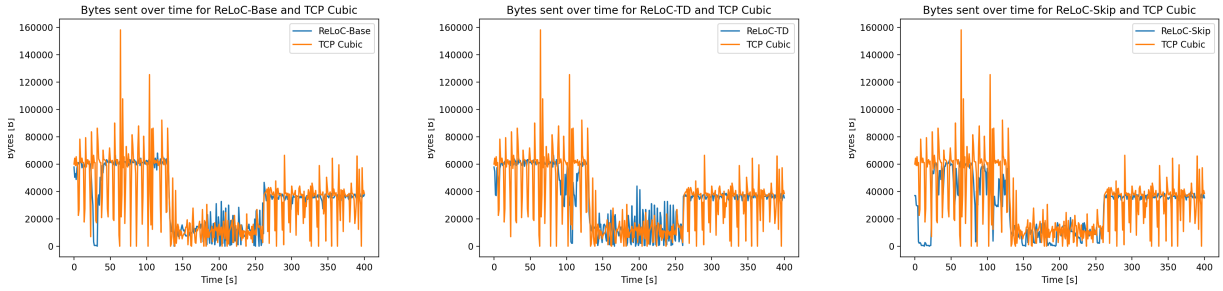
(d) ReLoC-Base (e) ReLoC-TD (f) ReLoC-Skip

Figure 5.21. Visualization of cumulative bytes sent and received throughout the test

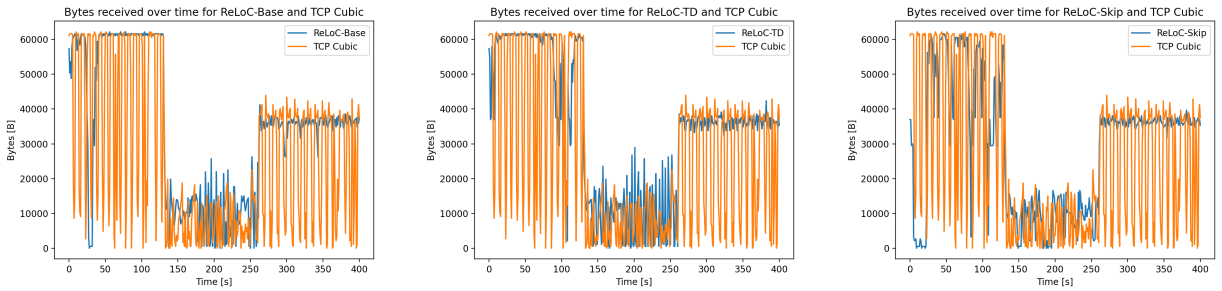
CCA	TCP Cubic	ReLoC-Base	ReLoC-TD	ReLoC-Skip
Cumulative bytes received	10,126,648	13,721,600	13,538,824	11,685,336

Table 5.7. Cumulative bytes received for each CCA

Once again, similar observations can be made regarding the throughput, with the exception of ReLoC-Skip, which for the first few moments did not send any packets. However, once ReLoC-Skip started sending packets, it quickly caught up to, and surpassed the total throughput of TCP Cubic.



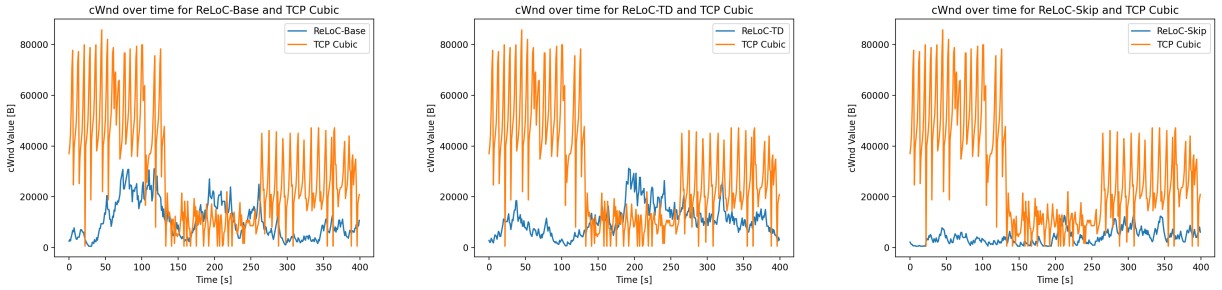
(a) ReLoC-Base (b) ReLoC-TD (c) ReLoC-Skip



(d) ReLoC-Base (e) ReLoC-TD (f) ReLoC-Skip

Figure 5.22. Visualization of sent and received bytes for each time-step

It can be seen in the graphs of Figure 5.22, that all the variants were able to increase up to 75% capacity and stabilize at this point. However, it can observed that the plots at 75% capacity are more volatile than that of the 100% capacity during the periods of stabilization. However, during 100% available capacity, the ReLoC variants are seen to be prone to make drastic drops seemingly at random.

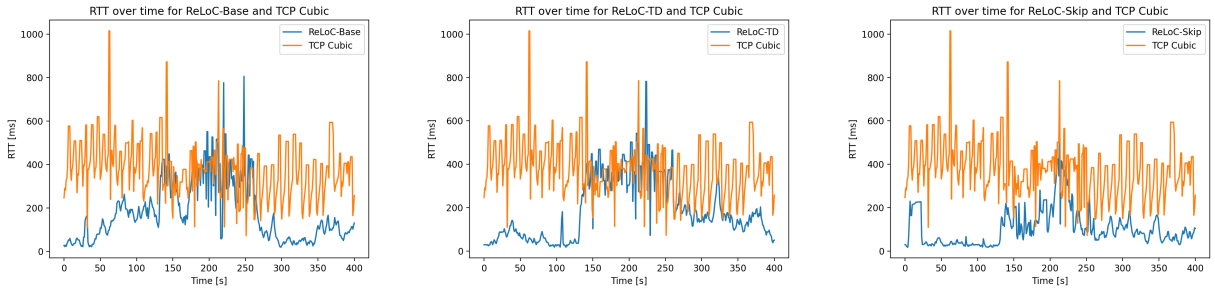


(a) ReLoC-Base (b) ReLoC-TD (c) ReLoC-Skip

Figure 5.23. Visualization of the cWnd value for each time-step

During this test, ReLoC-TD behaves less aggressively than during the previous tests. While ReLoC-TD does increase the cWnd, even above what TCP Cubic attempts to, it does not increase it to above 60,000, as was seen in the previous test. However, it is still raised above the loss-based operating point, causing high RTT and packets to be dropped. Otherwise, the test results are in line with the previous tests.

RTT results



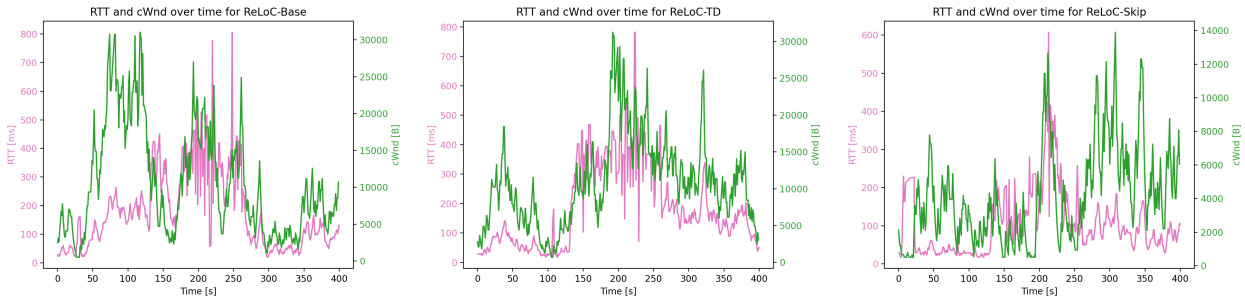
(a) ReLoC-Base (b) ReLoC-TD (c) ReLoC-Skip

Figure 5.24. Visualization of the average RTT in ms for each time-step

CCA	TCP Cubic	ReLoC-Base	ReLoC-TD	ReLoC-Skip
RTT(0,129)	434.13	125.53	80.71	55.17
RTT(130,259)	378.73	329.82	297.58	175.67
RTT(260,400)	342.46	88.19	131.40	86.28

Table 5.8. Average RTT in ms per segment for each CCA

The RTT reflects well the previous tests, showing that ReLoC-Skip tends to prioritize keeping the RTT low, while the two other variants tend to sacrifice some RTT by trying to push as many packets through when the capacity is lowered.



(a) ReLoC-Base (b) ReLoC-TD (c) ReLoC-Skip

Figure 5.25. Visualization of the average RTT in ms and cWnd value plotted together for each time-step

Once again, the results seen in the graphs of Figure 5.25 reflects the conclusions reached in the previous tests.

Test conclusion

In this test, as for all the previous tests, all variants of ReLoC were shown to outperform or match TCP Cubic in throughput and RTT. All variants were also able to show that they can perform at various levels of capacity. As all the variants were able to perform well throughout all tests, the proof of concept is proven to be successful.

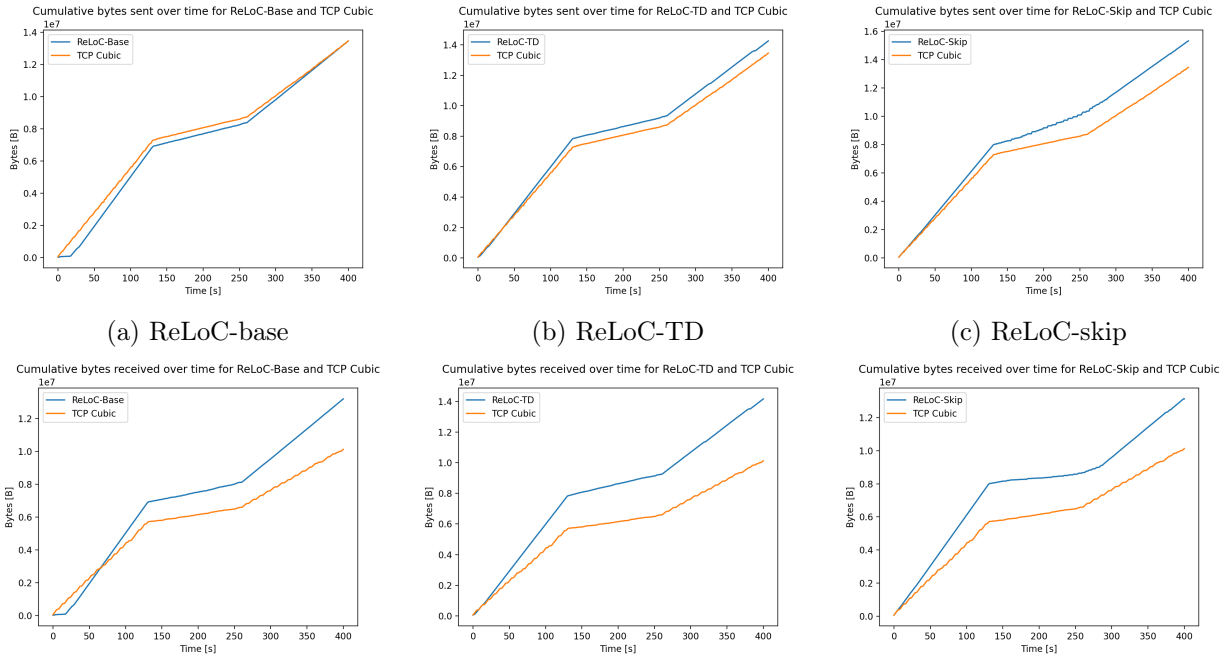
5.3 Test results - Generalizing training

Since the results presented in Section 5.2 showed that the proof of concept was successful, the next step was to diversify the training in order to generalize the agents. As mentioned prior, by diversifying the training, in this case by using randomly constructed simulations, the agents are able to develop policies that are better at handling a large variety of cases.
To avoid redundancy tests one through three for this generalized agent will not be presented in this chapter. Here it is assumed that getting good results in test four can only be achieved if the agent can get good results in tests one through three. However, graphs for tests one through three can be found in Appendix B.1.

5.3.1 Test 4

In this subsection, test 4 is performed again, but using the ReLoC variants that were trained to be more generalized. The test description can be found in Subsection 5.1.4.

Throughput results



(a) ReLoC-base (b) ReLoC-TD (c) ReLoC-skip

(d) ReLoC-base (e) ReLoC-TD (f) ReLoC-skip

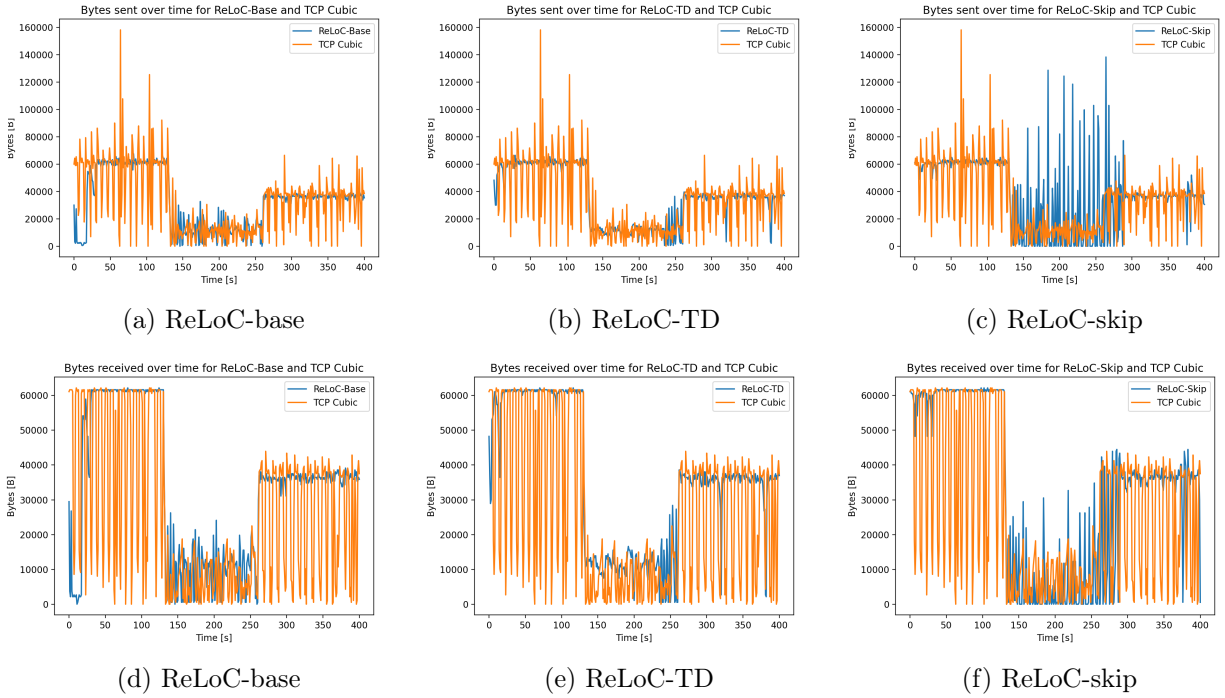
Figure 5.26. Visualization of cumulative bytes sent and received throughout the test

CCA	TCP Cubic	ReLoC-Base	ReLoC-TD	ReLoC-Skip
Cumulative bytes received	10,126,648	13,194,176	14,165,408	13,132,000

Table 5.9. Cumulative bytes received for each CCA

As seen with the deterministically trained variants, which were tested on the same test in Subsection 5.2.4, the performance of the randomly trained variants seen in Figures 5.26d, 5.26e, and 5.26f surpasses that of TCP Cubic in terms of total bytes received for the duration of the test. Based on the scores seen

in Table 5.9 this implementation of ReLoC-TD surpasses the best scores seen from the deterministically trained variants during this test.

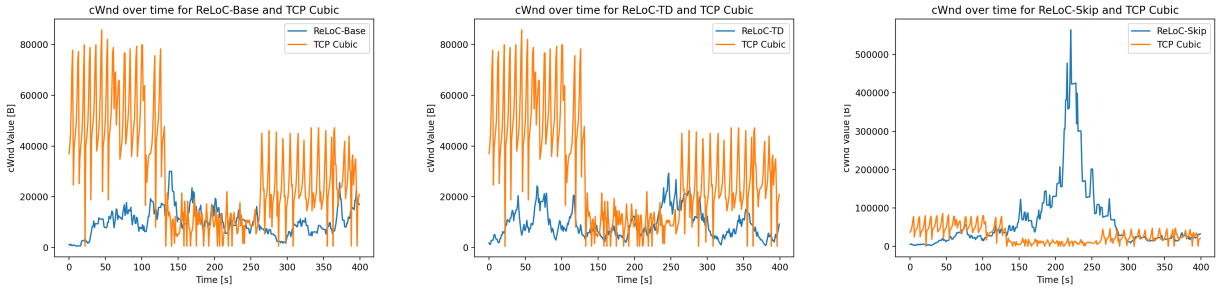


(a) ReLoC-base (b) ReLoC-TD (c) ReLoC-skip

(d) ReLoC-base (e) ReLoC-TD (f) ReLoC-skip

Figure 5.27. Visualization of sent and received bytes for each time-step

When inspecting the different graphs of Figure 5.27 a similar trend of accurately maintaining the sending rates when a larger amount of capacity is available, however, as is also seen with the deterministically trained ReLoC variants, the drop and available capacity from 130 seconds to 260 seconds into the simulation is still causing unstable sending and receiving rates. This can especially be identified in Figure 5.27c, where the sending rates violently fluctuate to values higher than when the full capacity was available. This may be an indication that this ReLoC-Skip policy can not handle the existence of high traffic from another node, in this case, UDP traffic.



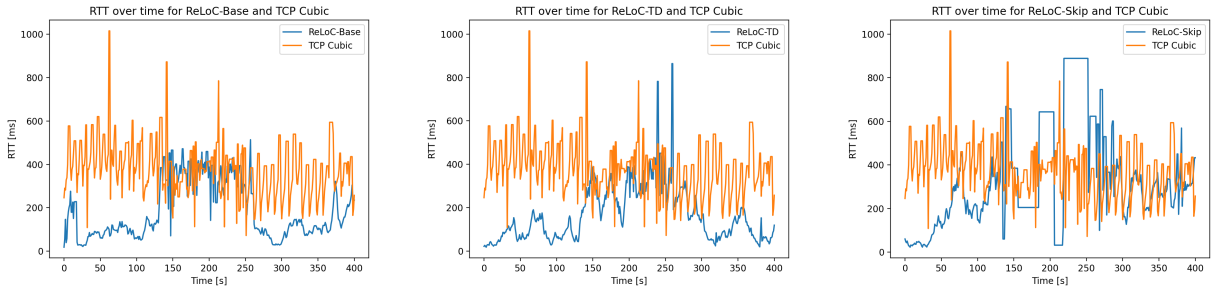
(a) ReLoC-base (b) ReLoC-TD (c) ReLoC-skip

Figure 5.28. Visualization of the cWnd value for each time-step

From previous tests it is observed that the cWnd for ReLoC variants generally is smaller than that of TCP Cubic, and while this is the case for ReLoC-Base and ReLoC-TD seen in Figures 5.28a and 5.28b, however, Figure 5.28c shows the cWnd of the ReLoC-Skip implementation is multiple times higher than that of the other variants during the 25% available capacity section from 130 seconds to

260 seconds into the simulation. This is a curious development as during tests of the deterministically trained variants, ReLoC-Skip was shown to be exceptionally good at keeping a low cWnd and RTT.

RTT results



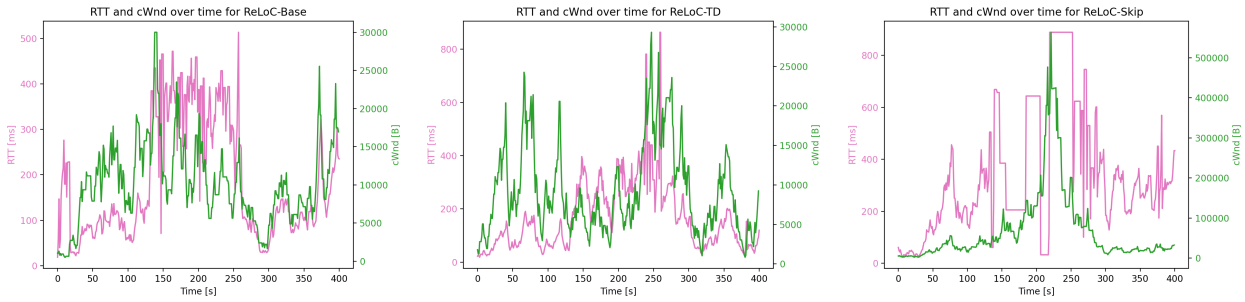
(a) ReLoC-base (b) ReLoC-TD (c) ReLoC-skip

Figure 5.29. Visualization of the average RTT in ms for each time-step

CCA	TCP Cubic	ReLoC-Base	ReLoC-TD	ReLoC-Skip
RTT(0,129)	434.13	97.52	80.71	173.78
RTT(130,259)	378.73	352.34	297.59	502.01
RTT(260,400)	342.46	118.19	131.40	333.69

Table 5.10. Average RTT in ms per segment for each CCA

The segment averages presented above in Table 5.10 are overall higher than the ones seen for the deterministically trained ReLoC variants seen in Table 5.8. This is especially the case for ReLoC-Skip, which when trained deterministically held especially low RTT averages. This development can indicate that training with random simulations may incentivize a prioritization of throughput compared to RTT.



(a) ReLoC-base (b) ReLoC-TD (c) ReLoC-skip

Figure 5.30. Visualization of the average RTT in ms and cWnd value plotted together for each time-step

On Figures 5.30a, 5.30b, and 5.30c it can be seen that the random training has not affected the relationship between cWnd and RTT. The two values can be seen to share the same shape. The biggest outlier is once again ReLoC-Skip in Figure 5.30c, where the lack of received packets generates a stagnant RTT curve, as the average RTT cannot be updated without packets being acknowledged.

Test conclusion

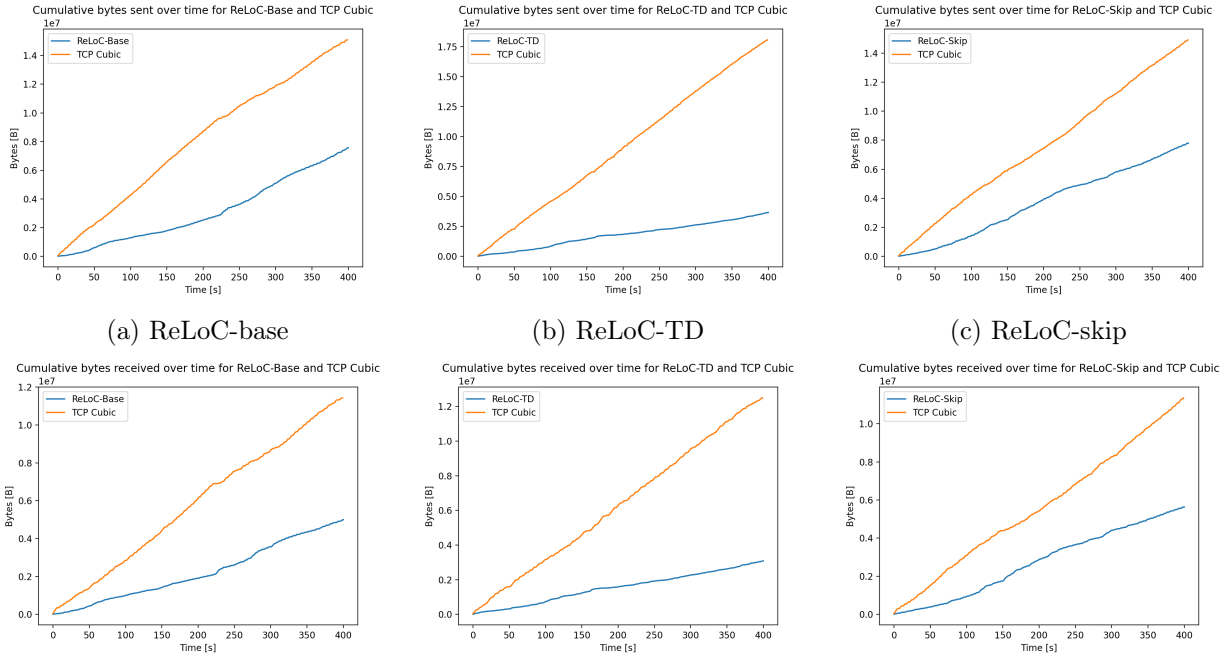
By training on a simulation of segments with randomly generated capacity, the ReLoC variations were

still capable of utilizing the bandwidth to a greater extent than TCP Cubic. This implementation, however, may prioritize bandwidth utilization at the cost of an increase in RTT. Furthermore, the variant ReLoC-skip showed a significant increase in cWnd when the available capacity was lowered. Whether this is the result of an unlikely series of action choices or an unintended habit from training is not known.

5.3.2 Test 5

Test 5 is performed as a baseline test for how ReLoC performs when there is another TCP connection on the same channel. Throughout the test the available capacity is at 100%, where the two TCP connections will have to share the capacity. The test description can be found in Subsection 5.1.5.

Throughput results



(a) ReLoC-base (b) ReLoC-TD (c) ReLoC-skip

(d) ReLoC-base (e) ReLoC-TD (f) ReLoC-skip

Figure 5.31. Visualization of cumulative bytes sent and received throughout the test

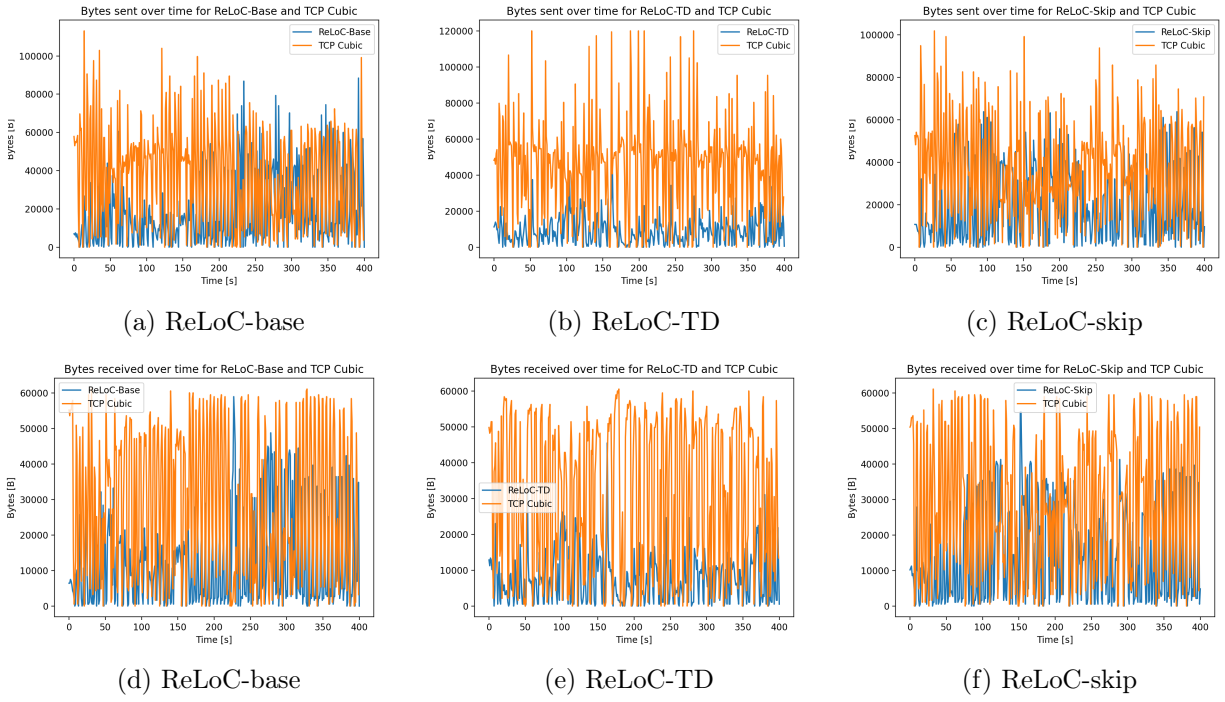
CCA	ReLoC-Base	ReLoC-TD	ReLoC-Skip
Cumulative bytes received: ReLoC	4,995,520	3,084,144	5,633,360
Cumulative bytes received: TCP Cubic	11,433,416	12,489,336	11,355,696

Table 5.11. Cumulative bytes received for each CCA

Figure 5.31 depicts the summed bytes sent and received for a ReLoC variant and a TCP Cubic node in the same simulation. Likewise, table 5.11 shows the cumulative sum of bytes received when testing

a ReLoC variant. The variant being used in the simulation is described in the column. On the rows, the results from either the ReLoC node or the TCP Cubic node can be found.

From Figure 5.31 it is apparent that TCP Cubic is more aggressive when compared with any ReLoC variation. This is known, as TCP Cubic attempts to send more packets than any ReLoC CCA. Looking at Table 5.11 it can be seen that the variant, which is able to compete the best against TCP Cubic, is ReLoC-Skip, which is capable of successfully transmitting nearly half the number of bytes as TCP Cubic. This distribution of usage of available network capacity is inefficient since using any ReLoC variant would be less efficient than using TCP Cubic whenever multiple TCP connections are competing for network capacity.

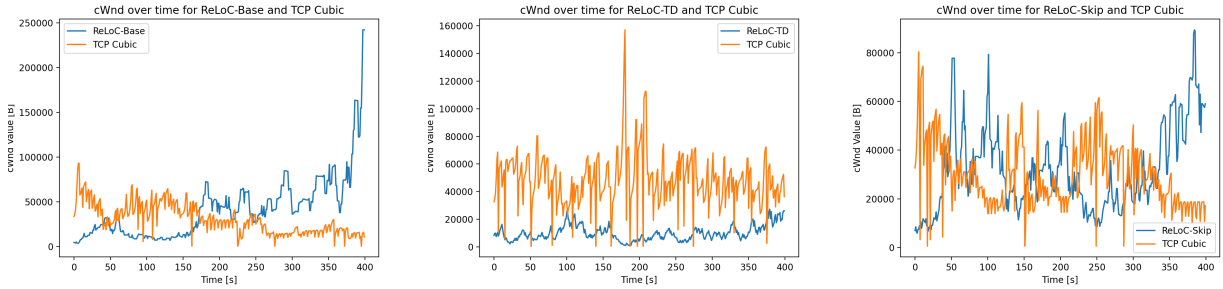


(a) ReLoC-base (b) ReLoC-TD (c) ReLoC-skip

(d) ReLoC-base (e) ReLoC-TD (f) ReLoC-skip

Figure 5.32. Visualization of sent and received bytes for each time-step

Figure 5.32 shows another angle of this issue, where it can be seen TCP Cubic is generally sending and receiving at a higher rate than the ReLoC variations. Some instances of a ReLoC variant sending at a rate equal to or higher than TCP Cubic can be seen. One such example is seen in Figure 5.32a, where ReLoC-Base is sending more packets than TCP Cubic at 240 seconds into the simulation. On the other hand, Figure 5.32b shows that ReLoC-TD never sends at a higher rate than TCP Cubic throughout the test.

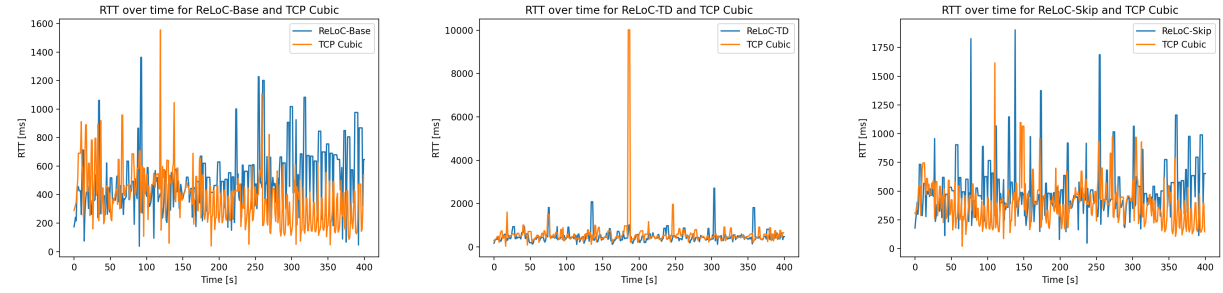


(a) ReLoC-base (b) ReLoC-TD (c) ReLoC-skip

Figure 5.33. Visualization of the cWnd value for each time-step

Comparing the cWnd of TCP Cubic and its ReLoC counterpart is an efficient way of comparing their ability to interact fairly. Starting with ReLoC-Base in Figure 5.33a it can be seen that the cWnd of TCP Cubic is higher during most of the first 170 seconds of the simulation, however, after this period the cWnd of ReLoC-Base becomes higher throughout the simulation, and at around 370 seconds into the simulation it starts increasing rapidly until the simulation ends. This behavior seems to indicate an ability to compete with TCP Cubic for the available bandwidth, however, comparing this information with the information shown in Figure 5.32d shows that even with a higher cWnd the ReLoC implementation does not outperform TCP Cubic's sending and receiving capabilities.
For ReLoC-TD, Figure 5.33b shows that unlike ReLoC-Base, which started competing with TCP Cubic for available capacity, ReLoC-TD stays at a low value during the entire duration of the simulation. This may be a result of ReLoC being designed to minimize the RTT, but as a side effect, it behaves more similarly to TCP Vegas in that it is unable to compete with aggressive CCAs.
The last variant, ReLoC-Skip, alternates in dominance with TCP Cubic in Figure 5.33c. Based on this, it would seem logical for the two to have a similar throughput throughout the simulation, but as seen in Table 5.11 TCP Cubic received more than double the number of bytes than ReLoC-Skip during the simulation.

RTT results



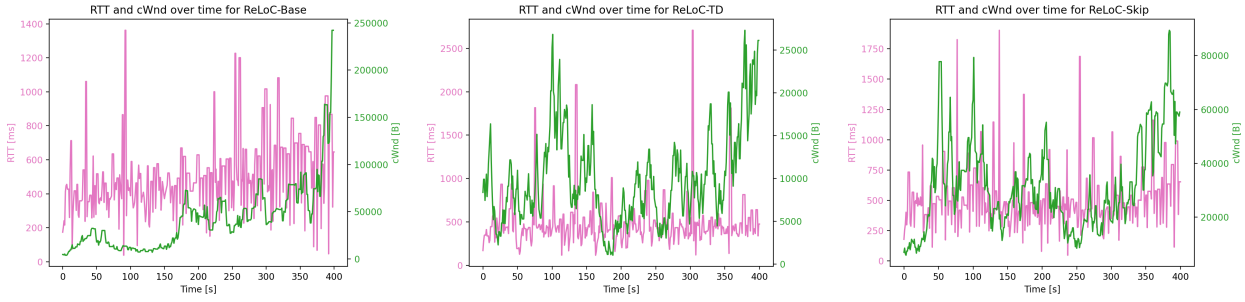
(a) ReLoC-base (b) ReLoC-TD (c) ReLoC-skip

Figure 5.34. Visualization of the average RTT in ms for each time-step

CCA	ReLoC-Base	ReLoC-TD	ReLoC-Skip
RTT(0,400) ReLoC	684.77	573.20	489.10
RTT(0,400) TCP Cubic	364.56	347.06	452.04

Table 5.12. Average RTT in ms per segment for each CCA

Table 5.12 shows the average RTT that each CCA experienced during the simulation. The table is separated in simulation by columns, meaning the averages in the same column are measured by different nodes in the same simulation.
Based on Figure 5.34 the RTT can be seen to correlate between the ReLoC variant and TCP Cubic for that simulation. This is a result of the two CCAs influencing the same bottleneck queue. This is further shown by the overall increase in average RTT for all ReLoC implementations in Table 5.12. This table informs that the only ReLoC implementation variation that has a lower average RTT than its TCP Cubic counterpart, is ReLoC-TD. This difference is the result of a spike in RTT over a few episodes seen in Figure 5.34b. As per the shown data, when competing with TCP Cubic, the ReLoC variations have higher average RTT.



(a) ReLoC-base (b) ReLoC-TD (c) ReLoC-skip

Figure 5.35. Visualization of the average RTT in ms and cWnd value plotted together for each time-step

The previous relation between RTT and cWnd was that whenever the cWnd would change, a similar change could be seen in the RTT. While this might still be the case to a certain extent, the addition of a new flow with auto-variation created the unpredictable behavior seen in Figure 5.35. While this change in relation makes it harder to identify an ideal cWnd value based on the RTT, it is still possible to detect queue buildup using the RTT, which then, depending on the learned policy, could result in a change in cWnd. Since having another node also communicating over the channel disrupts the relation between RTT and cWnd, the graphs plotting RTT and cWnd together will not be included for the remainder of the test results.

Test conclusion
The introduction of a non-static flow in the form of another TCP connection that will compete for the available capacity creates major challenges for the ReLoC implementations. None of the three ReLoC CCAs were able to compete at an equal level to TCP Cubic resulting in a significant loss in utilized capacity. The new flow also alters the relationship between cWnd and RTT, by introducing a new source of variation in RTT.
For ReLoC to work efficiently in a real scenario, it must be able to co-exist with more aggressive CCAs

such as TCP Cubic. However, based on the aforementioned test results, the current implementation of ReLoC would not be sufficiently capable in such a scenario.

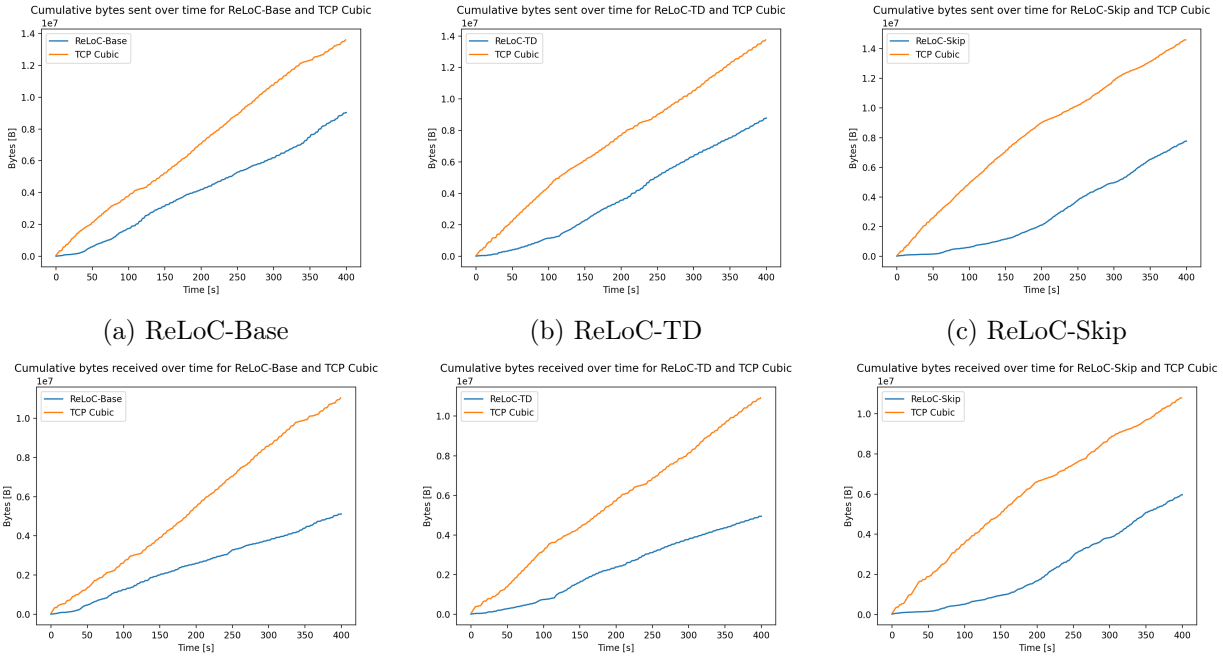
5.4 Test results - Making ReLoC competitive

The results presented in Section 5.3 showed that none of the ReLoC variants are able to produce competitive results while another node using TCP Cubic is also utilizing the channel. Due to TCP Cubic's aggressive nature, the ReLoC variants underperform as they try to find and stabilize at Klienrock's optimal operating point. In an attempt to make the ReLoC variants more competitive while sharing a channel with another node using TCP Cubic, the variants have been retrained. During this retraining, the only difference is that another node, using TCP Cubic, shares the channel. This retraining was performed in the hopes that the ReLoC variants would be able to find an effective strategy while coexisting with another node on a channel.

5.4.1 Test 5

In this subsection, test 5 is once again performed. However, here it is performed using variants of ReLoC that are trained with another TCP connection present on the channel. The test description can be found in Subsection 5.1.5.

Throughput results



(a) ReLoC-Base (b) ReLoC-TD (c) ReLoC-Skip

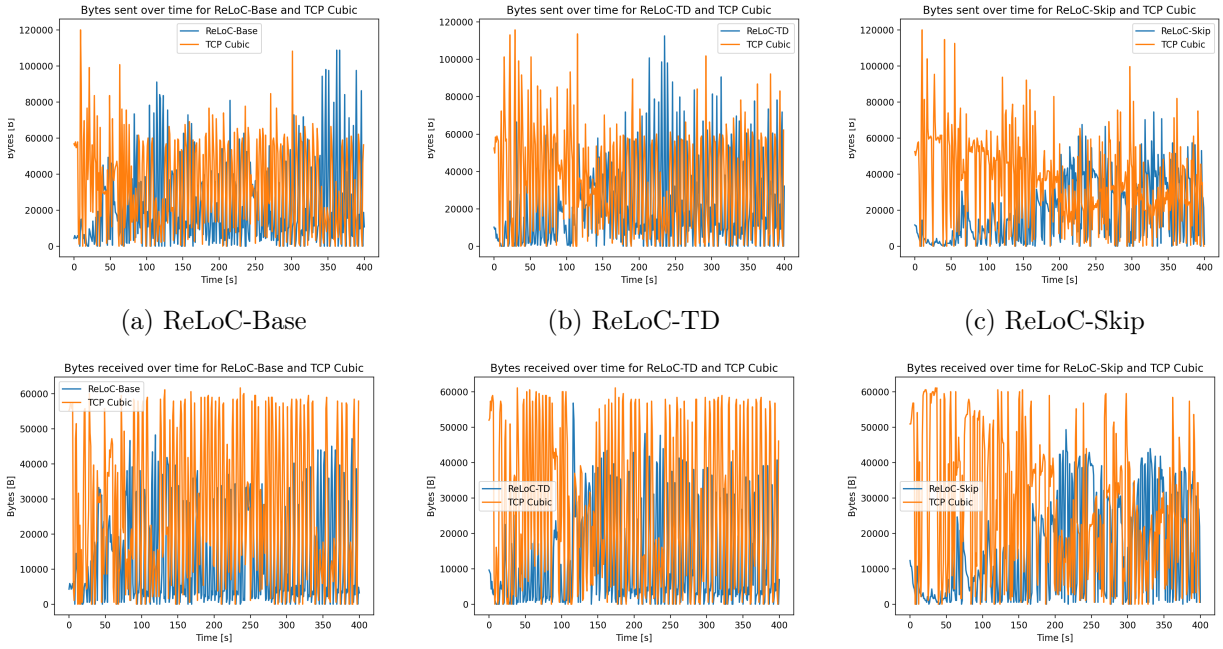
(d) ReLoC-Base (e) ReLoC-TD (f) ReLoC-Skip

Figure 5.36. Visualization of cumulative bytes sent and received throughout the test

CCA	ReLoC-Base	ReLoC-TD	ReLoC-Skip
Cumulative bytes received: ReLoC	5,115,048	4,952,104	5,968,360
Cumulative bytes received: TCP Cubic	11,037,312	10,917,248	10,797,184

Table 5.13. Cumulative bytes received for each CCA

Similar to test 5 in Section 5.3, all variants of ReLoC underperform compared to TCP Cubic, however, the variants trained with another TCP connection present on the channel are shown to score higher in this test than those that were not. While all three variants were outperformed by TCP Cubic, it can be seen that at around time-stamp 200, ReLoC-Skip increases its throughput and begins to approximately match the throughput of TCP Cubic.

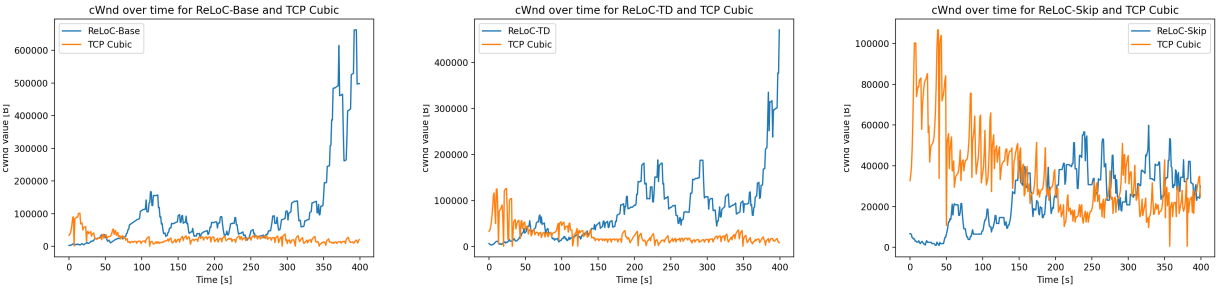


(a) ReLoC-Base (b) ReLoC-TD (c) ReLoC-Skip

(d) ReLoC-Base (e) ReLoC-TD (f) ReLoC-Skip

Figure 5.37. Visualization of sent and received bytes for each time-step

As 100% capacity is known to be 60kB/s, optimally the ReLoC variants, and TCP Cubic, should be sending at 30kB/s, as to maximize the throughput of all senders. This, however, can be seen to not be the case. In Figures 5.37d, 5.37e, and 5.37f, it can be seen that TCP Cubic tries to dominate the channel and is successful at sending at 60kB/s in many instances, before detecting congestion and lowering its cWnd again. In comparison, none of the ReLoC variants send at 60kB/s at any point but mostly lie near 30kB/s to 40kB/s. It can also be seen in Figure 5.37f that TCP Cubic becomes less dominant after time-step 200. This is in line with what can be seen in Figure 5.36f, where the slope representing the sending rate for ReLoC-Skip is approximately the same as the slope for TCP Cubic.

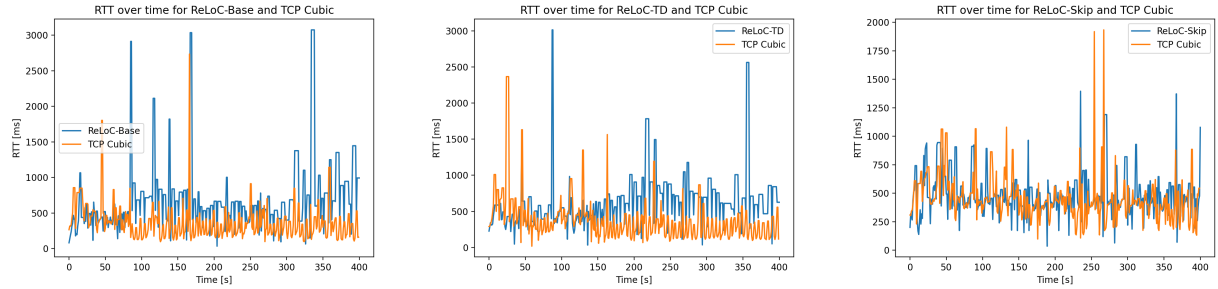


(a) ReLoC-Base (b) ReLoC-TD (c) ReLoC-Skip

Figure 5.38. Visualization of the cWnd value for each time-step

In Figures 5.38a and 5.38a it can be seen that ReLoC-Base and ReLoC-TD increase their cWnd significantly higher than TCP Cubic. If it is assumed that TCP Cubic finds the limit for congesting the network, as TCP Cubic is loss-based, it can be assumed that ReLoC-Base and ReLoC-TD, in a sense, chokes itself with congestion. It can be assumed that ReLoC-Base and ReLoC-TD tries to maximize throughput by sending as many packets through the network as possible, hoping that they are not dropped. Meanwhile, ReLoC-Skip can be seen, in Figure 5.38c, to steadily increase its cWnd until it is about equal to that of TCP Cubic. This, again, is in line with what was observed in the paragraphs above.

RTT results



(a) ReLoC-Base (b) ReLoC-TD (c) ReLoC-Skip

Figure 5.39. Visualization of the average RTT in ms for each time-step

CCA	ReLoC-Base	ReLoC-TD	ReLoC-Skip
RTT(0,400) ReLoC	684.77	573.20	489.10
RTT(0,400) TCP Cubic	364.56	347.06	452.04

Table 5.14. Average RTT in ms per segment for each CCA

Since ReLoC-Base and ReLoC-TD increased their cWnd to operate above the loss-based operating point, the RTT from their packets is also very high, both averaging over half a second per packet. However, ReLoC-Skip has an RTT close to that of TCP Cubic. This is expected as ReLoC-Skip mostly has its cWnd value within the same range as TCP Cubic.

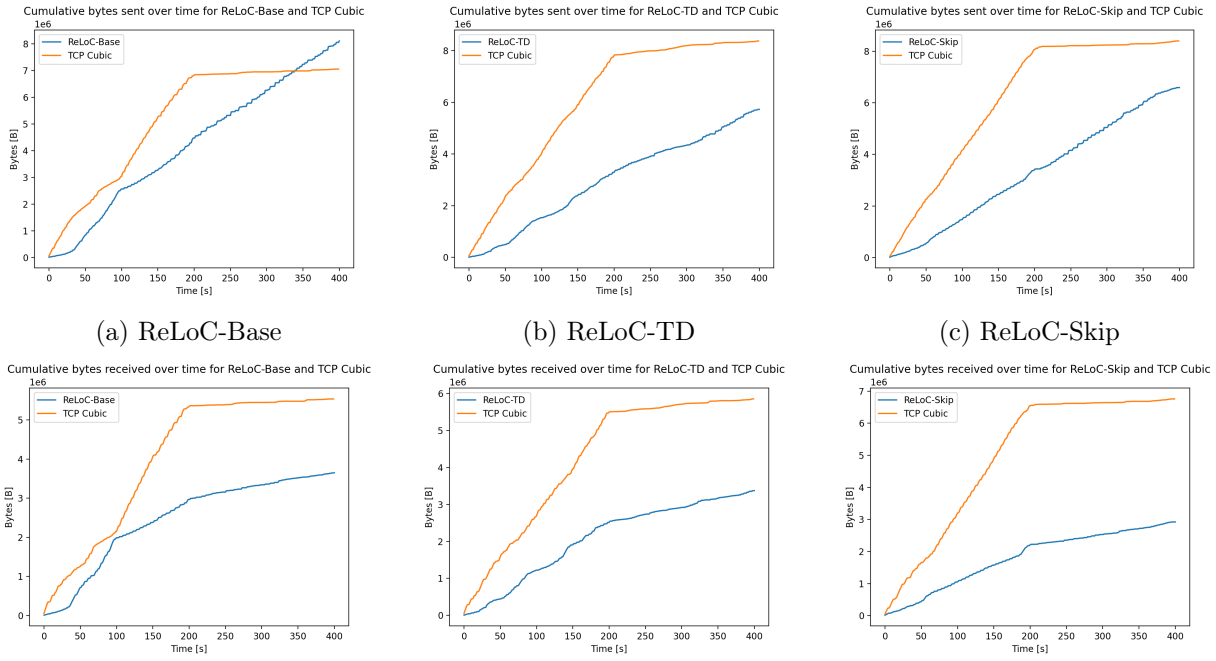
Test conclusion

Overall all variants of ReLoC showed worse performance than TCP Cubic, both in throughput and RTT. However, ReLoC-Skip showed that it was able to find a point where it could compete seemingly fairly with TCP Cubic to a point where both had approximately the same throughput and RTT.

5.4.2 Test 6

Test 6 is similar to test 2, where the available capacity drops to 50% at time-step 200, but another TCP connection is communicating over the channel. The test description can be found in Subsection 5.1.5.

Throughput results



(a) ReLoC-Base (b) ReLoC-TD (c) ReLoC-Skip

(d) ReLoC-Base (e) ReLoC-TD (f) ReLoC-Skip

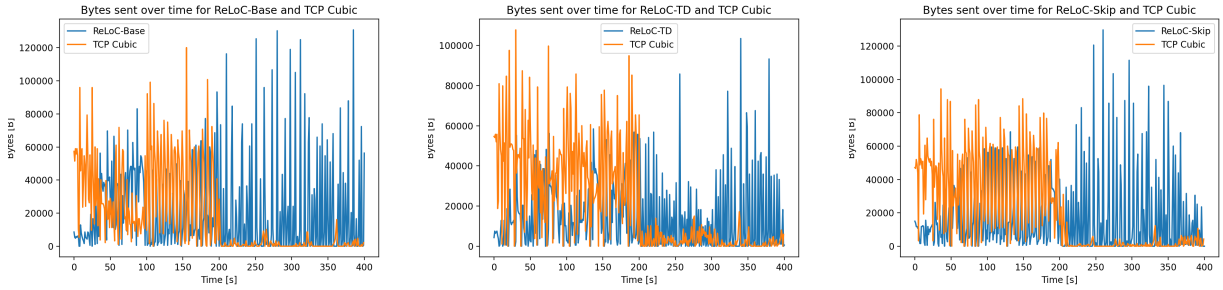
Figure 5.40. Visualization of cumulative bytes sent and received throughout the test

CCA	ReLoC-Base	ReLoC-TD	ReLoC-Skip
Cumulative bytes received: ReLoC	3,650,160	3,373,048	2,921,736
Cumulative bytes received: TCP Cubic	5,538,488	5,854,192	6,758,960

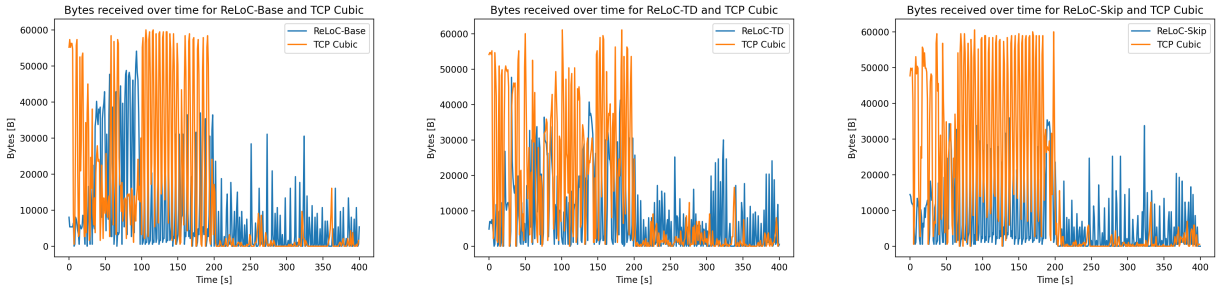
Table 5.15. Cumulative bytes received for each CCA

Despite the available capacity dropping to 50% at time-step 200, all variants can be seen to more or less keep the same sending rate as they had during 100% available capacity. This results in ReLoC-Base sending more packets than TCP Cubic throughout the test, although it still ends up with a lower overall throughput than TCP Cubic. It can be seen in Figure 5.40d that between time-step 50 and 100, ReLoC-Base actually gains a higher throughput than TCP-Cubic. Despite there being no

change in the network structure, after time-step 100, ReLoC-Base lowers its sending rate, resulting in lowering its throughput. Once the available capacity is lowered to 50%, ReLoC-Base does end up with slightly higher throughput than TCP Cubic. ReLoC-TD and ReLoC-Skip also manage to gain a higher throughput than TCP Cubic during the section with 50% available capacity.



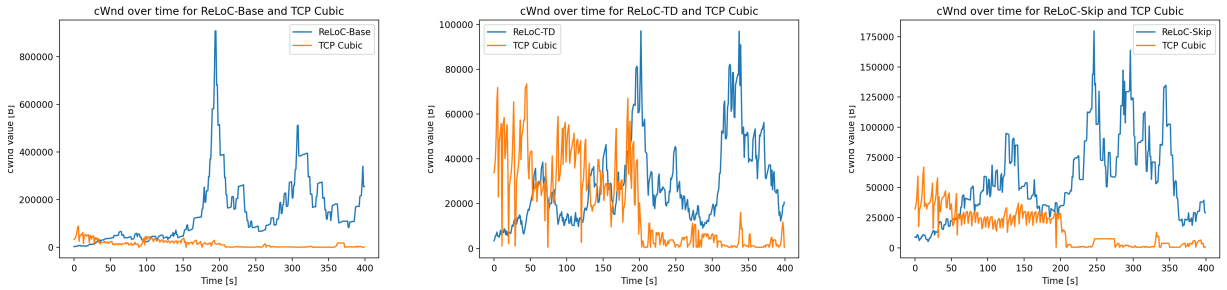
(a) ReLoC-Base (b) ReLoC-TD (c) ReLoC-Skip



(d) ReLoC-Base (e) ReLoC-TD (f) ReLoC-Skip

Figure 5.41. Visualization of sent and received bytes for each time-step

On the graphs in Figure 5.41, it can be seen that the ReLoC variants are able to gain higher throughput than TCP Cubic during the lowered available capacity by ignoring congestion. By sending a lot of packets. the network becomes highly congested and the RTT becomes very high as well, but the ReLoC variants seem to have gotten to the conclusion that it does not matter whether packets are dropped or the RTT as long as some packets make it to the destination. While this way of handling congestion makes sense to some degree, the underlying philosophy is wrong. As was proven in tests 1 through 4, operating near Klienrock's optimal operating point increases the overall throughput. Furthermore, while the ReLoC variants gained throughput from this aggressive behavior, TCP Cubic got choked by the congestion, which can be seen in Figures 5.41d, 5.41e, and 5.41f, as well as the slopes in Figures 5.40d, 5.40e, and 5.40f.

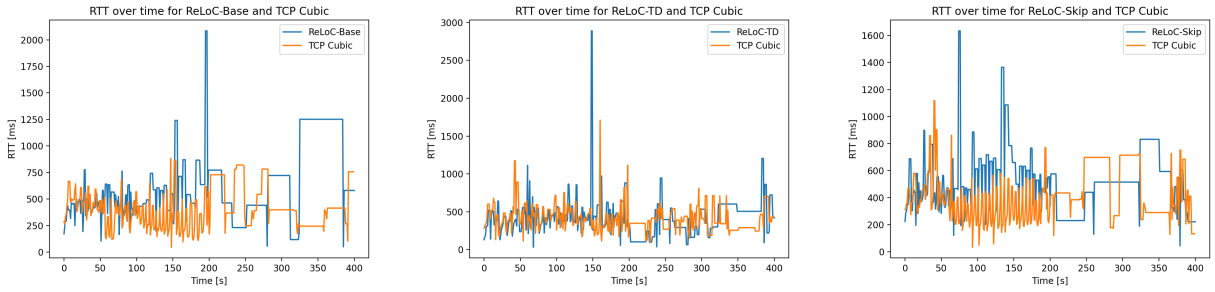


(a) ReLoC-Base (b) ReLoC-TD (c) ReLoC-Skip

Figure 5.42. Visualization of the cWnd value for each time-step

The graphs in Figure 5.42 support the observations made in the paragraphs above. In Figure 5.42a it can be seen that ReLoC-Base had its cWnd set slightly higher than TCP Cubic during the period of time-step 50 to 100. This is also the period that ReLoC-Base was able to outperform TCP Cubic in throughput. In test 5 using the ReLoC variants trained with a TCP node on the channel, in Subsection 5.4.1, ReLoC-Skip was shown to perform approximately evenly with TCP Cubic while keeping its cWnd at the same point or slightly above. This trend is the same as seen in this test for ReLoC-Base. This makes sense, as this would mean that ReLoC sends a similar amount of packets as TCP Cubic, but without drastically lowering its cWnd whenever loss is detected.

RTT results



(a) ReLoC-Base (b) ReLoC-TD (c) ReLoC-Skip

Figure 5.43. Visualization of the average RTT in ms for each time-step

CCA	ReLoC-Base	ReLoC-TD	ReLoC-Skip
RTT(0,199) ReLoC	544.37	679.54	569.78
RTT(0,199) TCP Cubic	368.54	383.59	362.65
RTT(200,400) ReLoC	723.54	550.59	474.41
RTT(200,400) TCP Cubic	452.03	486.17	465.64

Table 5.16. Average RTT in ms per segment for each CCA

By operating around the maximum point of TCP Cubic, a lot of congestion is generated. This means that while ReLoC-Base is able to compete with TCP Cubic for a period, it is at the cost of dropping packets and getting a high RTT. In Figure 5.43a and 5.43b it can be seen that for the first 200 seconds, ReLoC-Base and ReLoC-TD have a similar RTT as TCP Cubic, while ReLoC-Skip is shown to have a slightly higher RTT than TCP Cubic, as seen in Figure 5.43c. Once the available capacity is lowered to 50%, the ReLoC variants all congest the network so heavily that due to the large number of dropped packets, the graph gains a box-like appearance, since the RTT cannot be updated if a packet is dropped.

Test conclusion

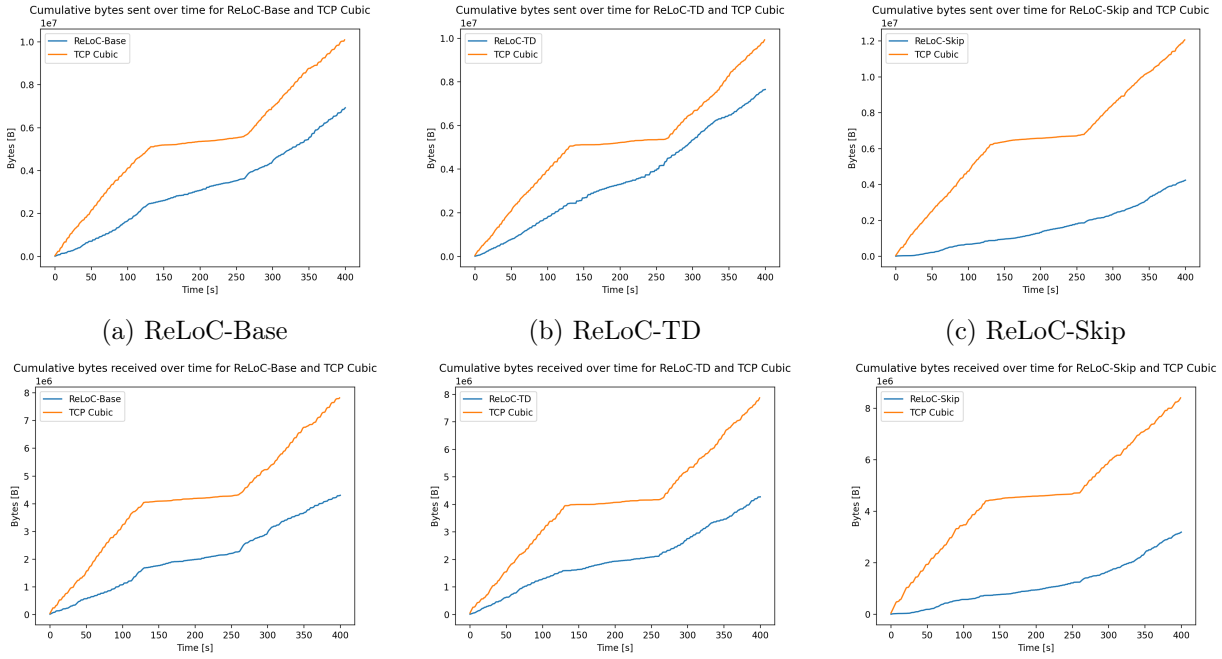
Throughout the test, the ReLoC variants were outperformed by TCP Cubic in both throughput and RTT. While ReLoC-Base showed promising results for a brief period of time, it reduced its cWnd, reducing its throughput. Once the available capacity was lowered, all variants were able to outperform

TCP Cubic, but they achieved this by showing extremely aggressive behavior, choking TCP Cubic. While this led to the ReLoC variants gaining higher throughput than TCP Cubic, it would likely have been more efficient to compete fairly with TCP Cubic, attempting to maximize the throughput of both nodes, not only would this likely increase the overall throughput between the two nodes, it could also lower the RTT.

5.4.3 Test 7

Test 7 is similar to test 3, where the available capacity drops to 50% at time-step 130 and then increases to 100% at time-step 260, but another TCP connection is communicating over the channel. The test description can be found in Subsection 5.1.5.

Throughput results



(a) ReLoC-Base (b) ReLoC-TD (c) ReLoC-Skip

(d) ReLoC-Base (e) ReLoC-TD (f) ReLoC-Skip

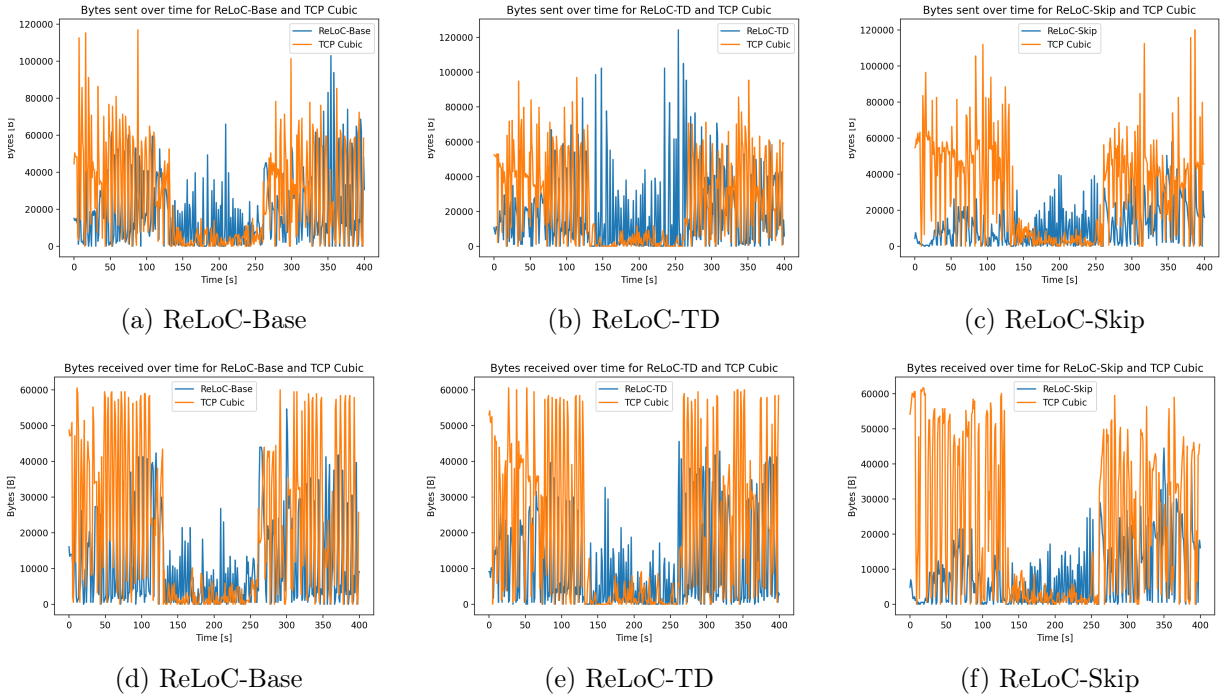
Figure 5.44. Visualization of cumulative bytes sent and received throughout the test

CCA	ReLoC-Base	ReLoC-TD	ReLoC-Skip
Cumulative bytes received: ReLoC	4,304,616	4,272,992	3,190,808
Cumulative bytes received: TCP Cubic	7,818,096	7,868,480	8,403,408

Table 5.17. Cumulative bytes received for each CCA

Once again, the ReLoC variants are shown to severely underperform compared to TCP Cubic. During this test, ReLoC-Skip especially underperformed compared to the other variants, having approximately

the same send rate throughout the entire test up until time-step 300, despite the changes in available capacity.

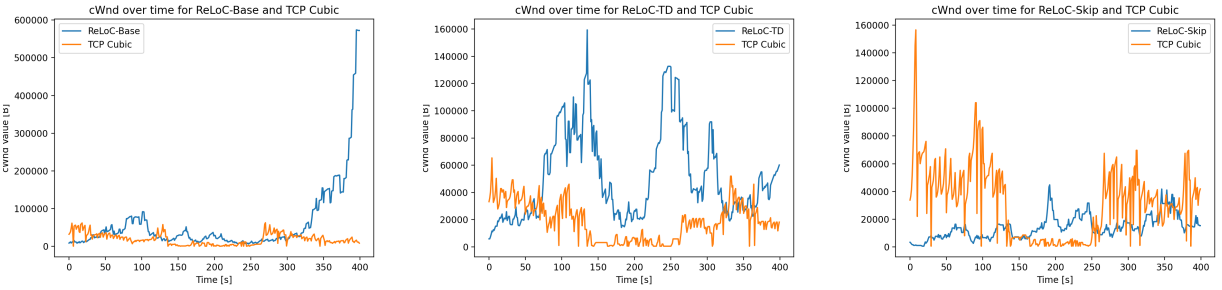


(a) ReLoC-Base (b) ReLoC-TD (c) ReLoC-Skip

(d) ReLoC-Base (e) ReLoC-TD (f) ReLoC-Skip

Figure 5.45. Visualization of sent and received bytes for each time-step

ReLoC-Base and ReLoC-TD show erratic behavior throughout the entirety of the test, as can be seen in Figures 5.45d and 5.45e. Meanwhile, although ReLoC-Skip underperforms compared to TCP-Cubic, it is shown to not behave erratically during the initial 129 seconds.

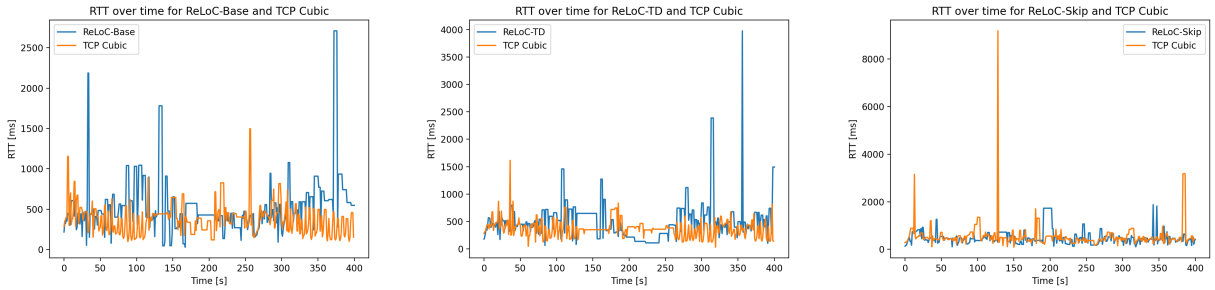


(a) ReLoC-Base (b) ReLoC-TD (c) ReLoC-Skip

Figure 5.46. Visualization of the cWnd value for each time-step

ReLoC-Base kept its cWnd around or a little above that of TCP Cubic throughout the majority of the test, except for the final 75 seconds, where it rapidly increased its cWnd. While keeping the cWnd at the same or a little above the cWnd of TCP Cubic had been shown to be a competitive strategy in the prior tests, it did not prove as successful in this test, however, this is likely why ReLoC-Base managed to get the highest throughput of the three ReLoC variants. ReLoC-Skip is shown to keep a very low cWnd throughout the first section of the test. While this caused it to have a low throughput as TCP Cubic's aggressiveness took most of the capacity, it is also the reason ReLoC-Skip did not show erratic behavior during this section.

RTT results



(a) ReLoC-Base (b) ReLoC-TD (c) ReLoC-Skip

Figure 5.47. Visualization of the average RTT in ms for each time-step

CCA	ReLoC-Base	ReLoC-TD	ReLoC-Skip
RTT(0,129) ReLoC	547.21	538.91	464.94
RTT(0,129) TCP Cubic	368.73	395.56	647.62
RTT(130,259) ReLoC	442.36	360.39	561.80
RTT(130,259) TCP Cubic	418.16	393.30	467.01
RTT(260,400) ReLoC	635.03	623.68	437.72
RTT(260,400) TCP Cubic	343.47	325.71	540.10

Table 5.18. Average RTT in ms per segment for each CCA

During this test, ReLoC-Skip is shown to have a lower RTT than TCP Cubic throughout sections one and three of the test. Although it has a lower RTT, it was shown to significantly lack throughput compared to TCP Cubic. In general, it can be seen in Figures 5.47a, 5.47b, and 5.47c, that the ReLoC variant were able to keep their RTT relatively close to that of TCP Cubic, although with some outliers making the overall average slower for ReLoC-Base and ReLoC-TD in all three sections except for ReLoC-TD when the available capacity was lowered to 50%. During the lowered capacity of 50%, ReLoC-TD and ReLoC-Skip had their cWnd increased significantly higher than TCP Cubic. For ReLoC-TD, this resulted in the box-shaped RTT graph which was also present during test 6, in Subsection 5.4.2. The same box-shaped peak can be seen around time-step 200 for ReLoC-Skip. It can be seen that ReLoC-Skip lowers its cWnd once the RTT peaks at around time-step 200 and the two peaks around time-step 350. This could signify that, like the ReLoC-skip which was trained as a proof of concept without another TCP connection on the channel, ReLoC-Step tries to prioritize RTT over high throughput.

Test conclusion

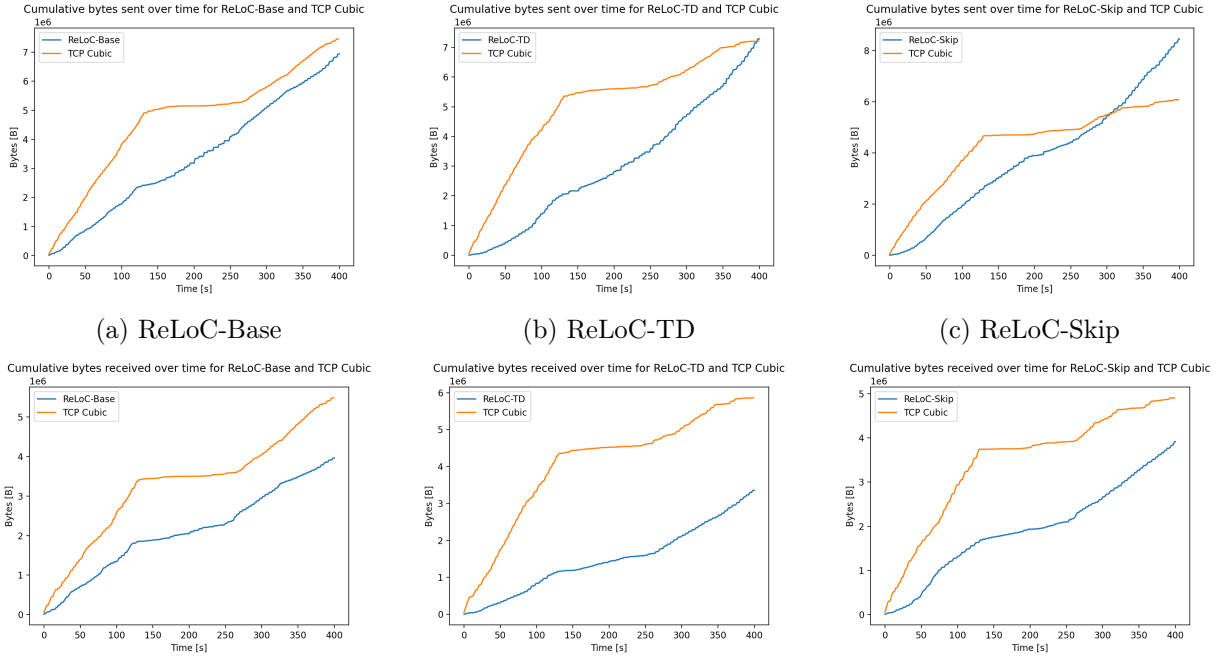
Throughout this test, and the prior tests, it has been shown that ReLoC-Base generally attempts to set its cWnd at around or a bit higher than that of TCP Cubic, ReLoC-TD has proven to be very aggressive, trying to choke TCP Cubic with congestion, while ReLoC-skip has proven to be more

sensitive toward RTT, generally trying to keep the RTT low, especially while there is 100% available capacity. Both ReLoC-Base and ReLoC-TD show a tendency to increase the cWnd explosively at around time-step 350, which has been witnessed in this test and test 5, in Subsection 5.4.1, which both ended the simulation at 100% capacity. It has also been shown that ReLoC-TD and ReLoC-Skip generally increase their cWnd during lowered capacity, choking TCP Cubic.

5.4.4 Test 8

Test 7 is similar to test 4, where the available capacity drops to 50% at time-step 130 and then increases to 75% at time-step 260, but another TCP connection is communicating over the channel. The test description can be found in Subsection 5.1.5.

Throughput results



(a) ReLoC-Base (b) ReLoC-TD (c) ReLoC-Skip

(d) ReLoC-Base (e) ReLoC-TD (f) ReLoC-Skip

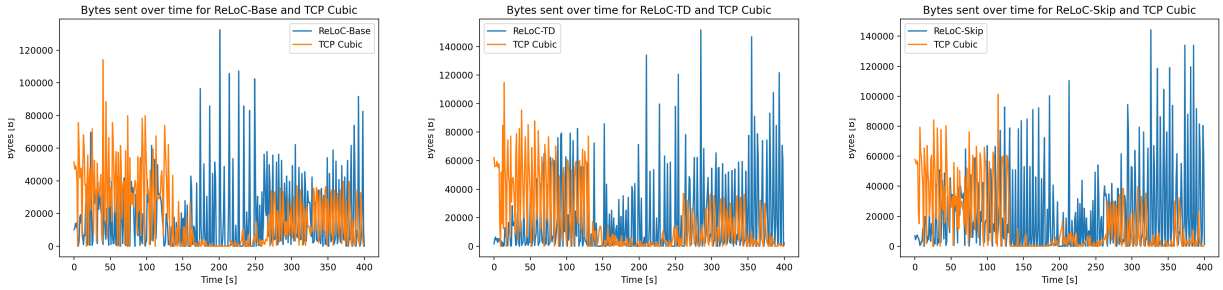
Figure 5.48. Visualization of cumulative bytes sent and received throughout the test

CCA	ReLoC-Base	ReLoC-TD	ReLoC-Skip
Cumulative bytes received: ReLoC	3,964,256	3,355,360	3,916,016
Cumulative bytes received: TCP Cubic	5,485,424	5,860,088	4,908,688

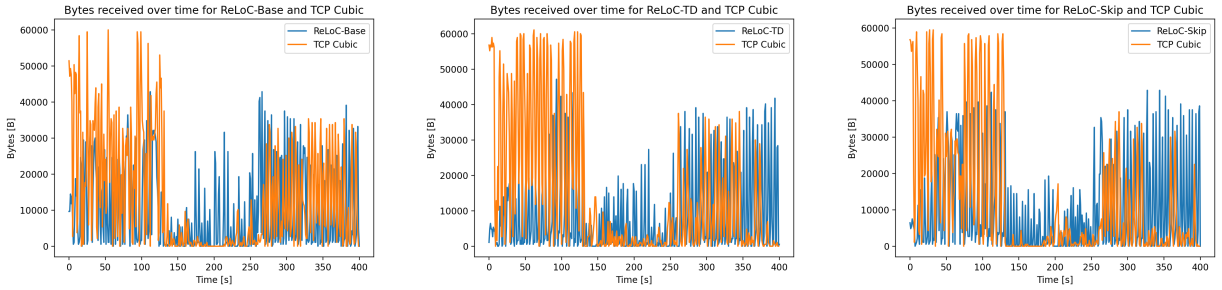
Table 5.19. Cumulative bytes received for each CCA

As expected, based on the prior test results, all variants of ReLoC underperformed compared to TCP Cubic. While ReLoC-Base loses out in throughput compared to TCP Cubic in the first segment, it gains higher throughput throughout the second segment and settles at approximately the same

throughput as TCP Cubic during the third segment. Both ReLoC-TD and ReLoC-Skip are shown to perform poorly, except for in the third section where they have higher throughput than TCP Cubic.



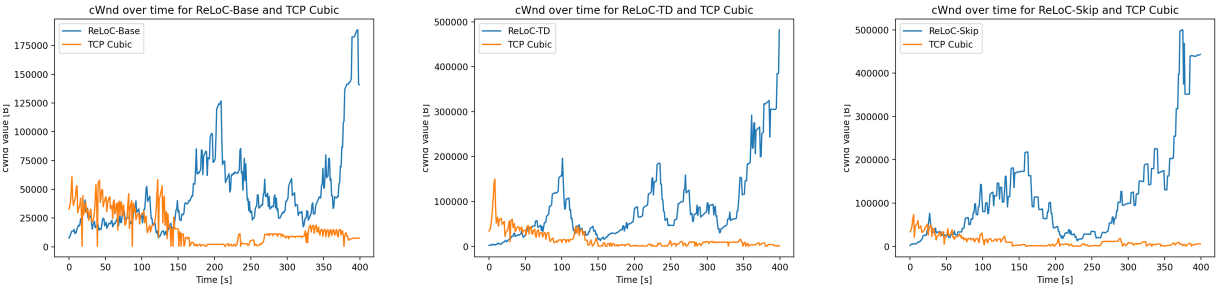
(a) ReLoC-Base (b) ReLoC-TD (c) ReLoC-Skip



(d) ReLoC-Base (e) ReLoC-TD (f) ReLoC-Skip

Figure 5.49. Visualization of sent and received bytes for each time-step

It can be seen towards the latter half the third section, around time-step 325, in Figure 5.49e and 5.49f, that both ReLoC-TD and ReLoC-Skip achieve higher throughput than TCP Cubic by, once again, using very aggressive tactics and choking TCP Cubic with congestion.

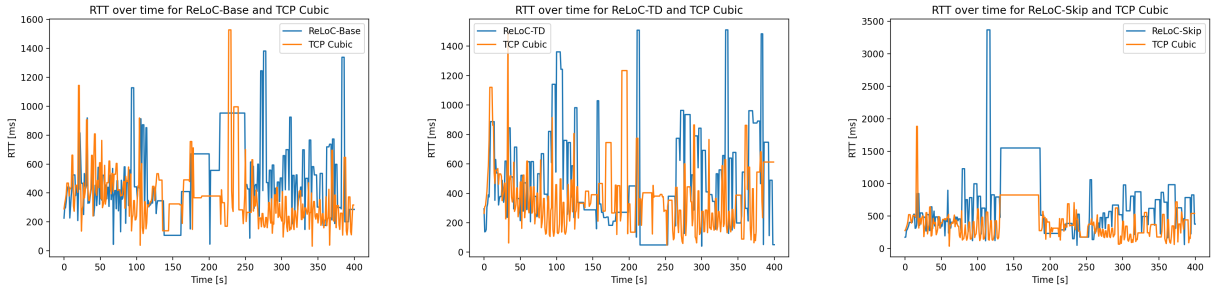


(a) ReLoC-Base (b) ReLoC-TD (c) ReLoC-Skip

Figure 5.50. Visualization of the cWnd value for each time-step

During this test, all variants can be seen to use very aggressive tactics during the reduced capacity and all dramatically increasing the cWnd towards the end. ReLoC-TD and ReLoC-Skip are shown to be a lot more aggressive than ReLoC-base, increasing the cWnd to way above 20,000. While ReLoC-Base once again is shown to keep its cWnd around the same as TCP Cubic during the first section and then enforce an aggressive strategy during the second section with the available capacity set to 50%, it can be seen that it reduces its cWnd down again once the available capacity is raised to 75%, but once again finishes off by dramatically increasing its cWnd around time-step 350.

RTT results



(a) ReLoC-Base (b) ReLoC-TD (c) ReLoC-Skip

Figure 5.51. Visualization of the average RTT in ms for each time-step

CCA	ReLoC-Base	ReLoC-TD	ReLoC-Skip
RTT(0,129) ReLoC	467.60	618.38	659.20
RTT(0,129) TCP Cubic	438.71	401.94	377.30
RTT(130,259) ReLoC	561.98	276.44	842.34
RTT(130,259) TCP Cubic	423.09	428.26	544.76
RTT(260,400) ReLoC	538.05	634.23	634.57
RTT(260,400) TCP Cubic	272.24	343.77	305.80

Table 5.20. Average RTT in ms per segment for each CCA

During sections one and three all ReLoC variations are shown to have much higher RTT than TCP Cubic. While ReLoC-TD shows to have a lower average RTT during section two, this result can be somewhat misleading due to the extremely high level of congestion.

Test conclusion

While ReLoC-Base and ReLoC-TD continued to show the behavior that was mentioned in the test conclusion of the prior test, in Subsection 5.4.3, ReLoC-Skip did not. During this test, ReLoC-Skip showed very aggressive behavior, akin to ReLoC-TD's expected behavior, completely ignoring keeping its RTT low. Throughout this test, and all tests including another TCP connection on the channel, the ReLoC variants have been shown to underperform. While the tests in Subsections 5.4.1, 5.4.2, 5.4.3, and 5.4.4, were meant as proof of concept to see if ReLoC would be useful on a network with other traffic, it may be that using the same approach of creating a proof of concept as for tests 1 through 4 was misguided. It may be that attempting to train the agents to be specialists for the networks in the tests, it would prove better to attempt training more generalized agents, as they may prove to be better at handling the various levels of capacity the extra traffic from TCP Cubic introduces.

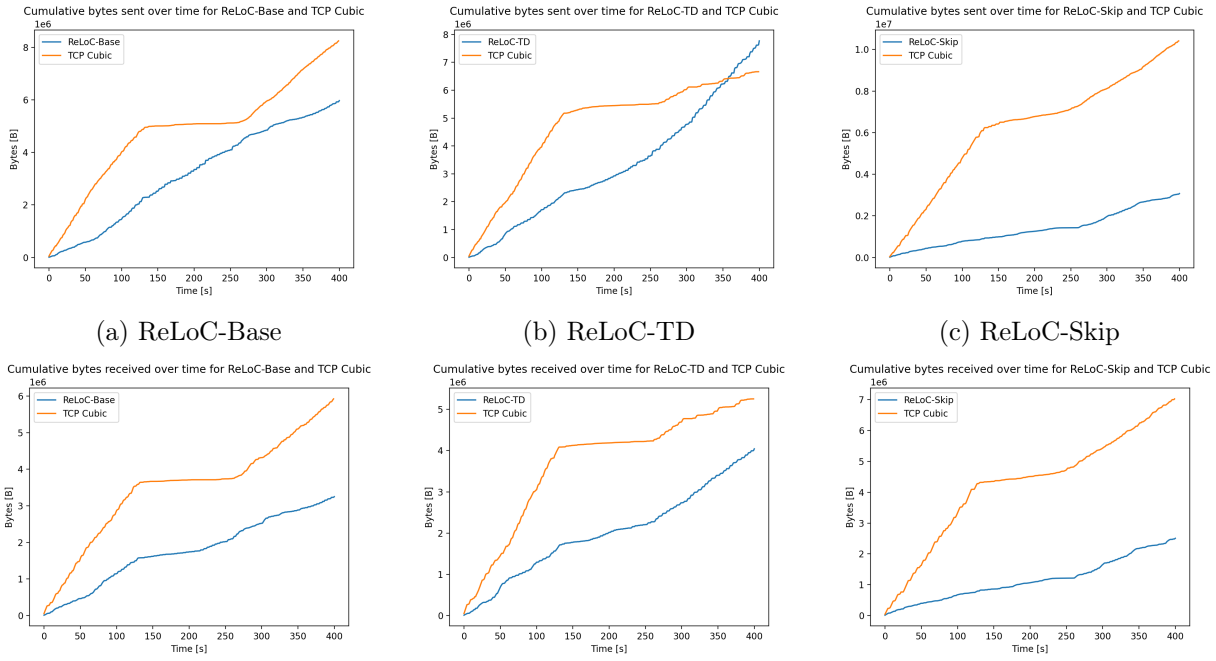
5.5 Test results - Improving through generalization

With the proof of concept agents being trained to specialize for the network scenarios present in the tests, their poor performance may be a sign that the current implementation of ReLoC cannot produce, or at least has not produced, a policy that can handle communication over a channel alongside another sender appropriately. However, it is believed that training a more generalized version of ReLoC may increase its performance as it will become better at handling the various levels of capacity caused by the extra traffic from the other node's communication.

5.5.1 Test 8

In this subsection, test 8 is performed again. Here the test is performed using the versions of ReLoC variations that are trained to be more generalized. The test description can be found in Subsection 5.1.5.

Throughput results



(a) ReLoC-Base (b) ReLoC-TD (c) ReLoC-Skip

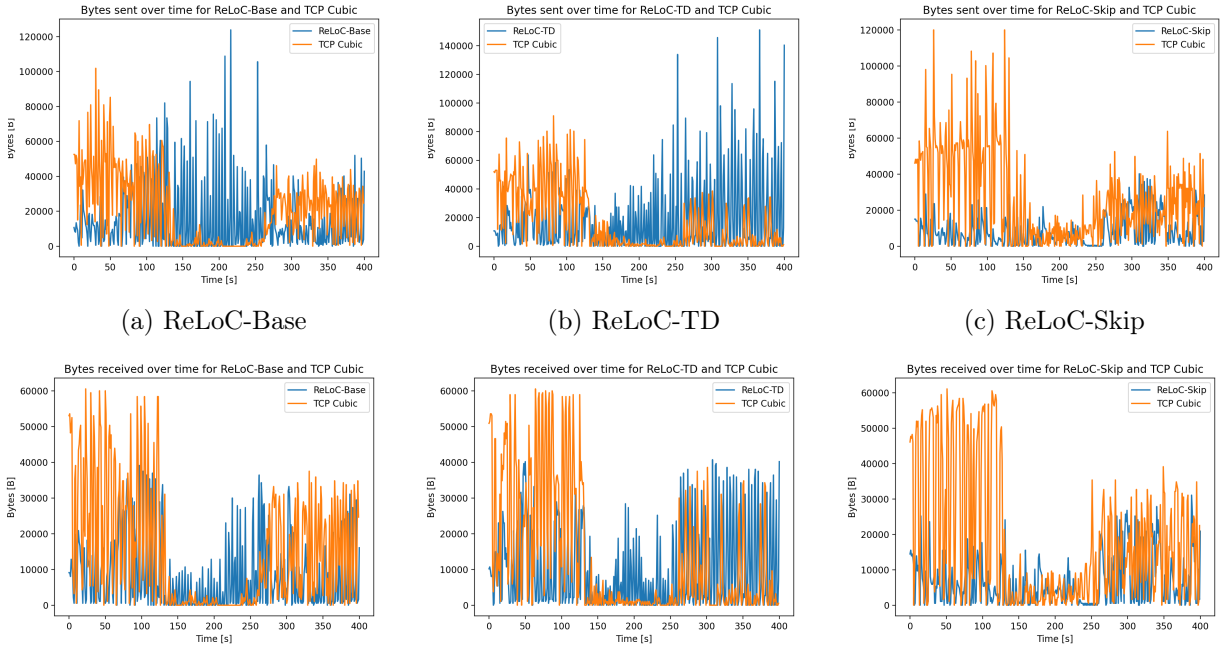
(d) ReLoC-Base (e) ReLoC-TD (f) ReLoC-Skip

Figure 5.52. Visualization of cumulative bytes sent and received throughout the test

CCA	ReLoC-Base	ReLoC-TD	ReLoC-Skip
Cumulative bytes received: ReLoC	3,248,696	4,041,976	2,506,872
Cumulative bytes received: TCP Cubic	5,922,264	5,252,800	7,023,744

Table 5.21. Cumulative bytes received for each CCA

Despite training the ReLoC variations to become more generalized, there seem to be no significant improvements. All variants continue to be outperformed by TCP Cubic, with ReLoC-TD outperforming TCP Cubic during the final section where the available capacity is limited to 75%.

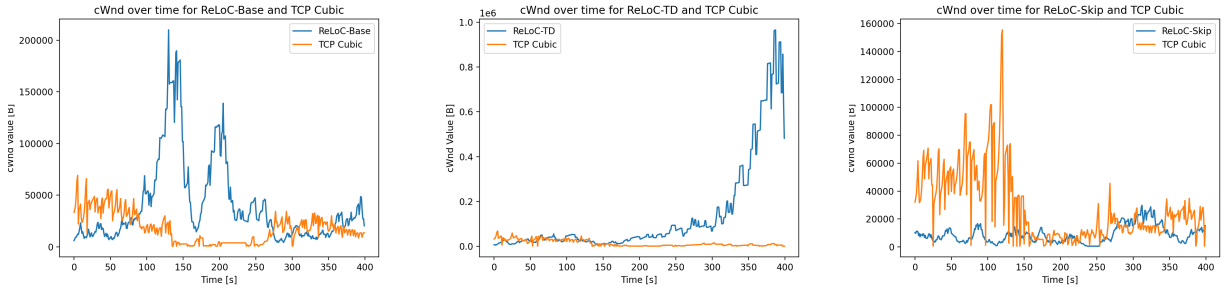


(a) ReLoC-Base (b) ReLoC-TD (c) ReLoC-Skip

(d) ReLoC-Base (e) ReLoC-TD (f) ReLoC-Skip

Figure 5.53. Visualization of sent and received bytes for each time-step

Once again it can be seen that ReLoC-Base and ReLoC-TD send a lot of packets during the section with lowered capacity at 50%, choking TCP Cubic with congestion. For ReLoC-TD this behavior continues as the capacity is raised to 75%, whereas ReLoC-Base lowers the send rate again. ReLoC-Skip is shown to have a very low send rate during the first section at 100% capacity, but during the following section at 50% capacity it seems to have approximately the same sending rate as TCP Cubic and during the final section at 75% it has a slightly lower sending rate than TCP Cubic.



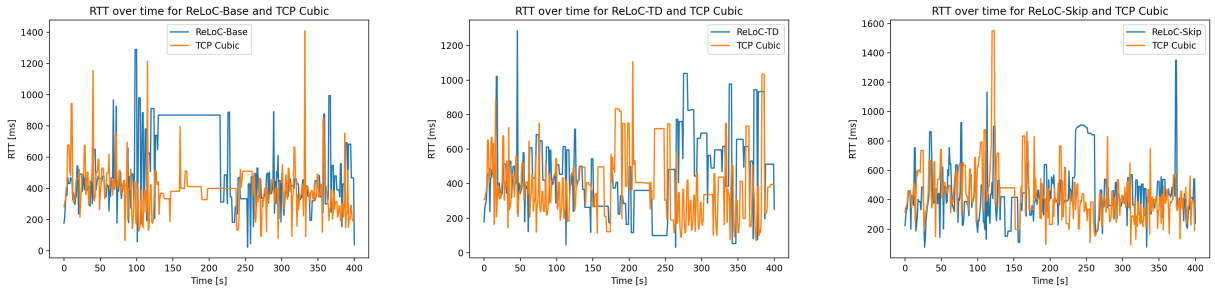
(a) ReLoC-Base (b) ReLoC-TD (c) ReLoC-Skip

Figure 5.54. Visualization of the cWnd value for each time-step

ReLoC-Base is seen in Figure 5.54a to have a slightly lower cWnd than TCP Cubic during the first section, then during the second section it increases the cWnd a lot to push through the limited capacity, and during the third section it mostly matches the cWnd of TCP Cubic. This gives the impression that this ReLoC-Base agent learned to match the cWnd of TCP Cubic to some degree while taking advantage

when capacity is low. ReLoC-TD matches the cWnd well for the first 150 seconds, whereafter it begins increasing its cWnd and at around time-step 300, it starts rapidly increasing the cWnd, creating a lot of congestion. ReLoC-Skip is once again shown to keep a low cWnd to prioritize RTT. During the first section of the test, ReLoC-Skip has a significantly lower cWnd than TCP Cubic, but during the second and third sections, ReLoC-Skip matches the cWnd of TCP Cubic well.

RTT results



(a) ReLoC-Base (b) ReLoC-TD (c) ReLoC-Skip

Figure 5.55. Visualization of the average RTT in ms for each time-step

CCA	ReLoC-Base	ReLoC-TD	ReLoC-Skip
RTT(0,129) ReLoC	501.51	472.76	442.22
RTT(0,129) TCP Cubic	422.39	383.27	550.77
RTT(130,259) ReLoC	701.19	292.55	495.32
RTT(130,259) TCP Cubic	398.12	479.46	413.38
RTT(260,400) ReLoC	433.68	624.45	428.84
RTT(260,400) TCP Cubic	358.53	232.32	361.69

Table 5.22. Average RTT in ms per segment for each CCA

During section two, ReLoC-Base and ReLoC-TD once again creates so much congestion that the average RTT does not give a clear picture, as was the case in precious tests in Subsections 5.4.2, 5.4.3, and 5.4.4. In the first and third sections, both ReLoC-Base and ReLoC-TD have significantly higher RTT than TCP Cubic. ReLoC-Skip shows once again that it attempts to keep its RTT down, having a lower RTT than TCP Cubic in section one, with the two other sections being relatively close. As a result of ReLoC-Skip keeping its RTT relatively low, it loses a lot of performance regarding throughput.

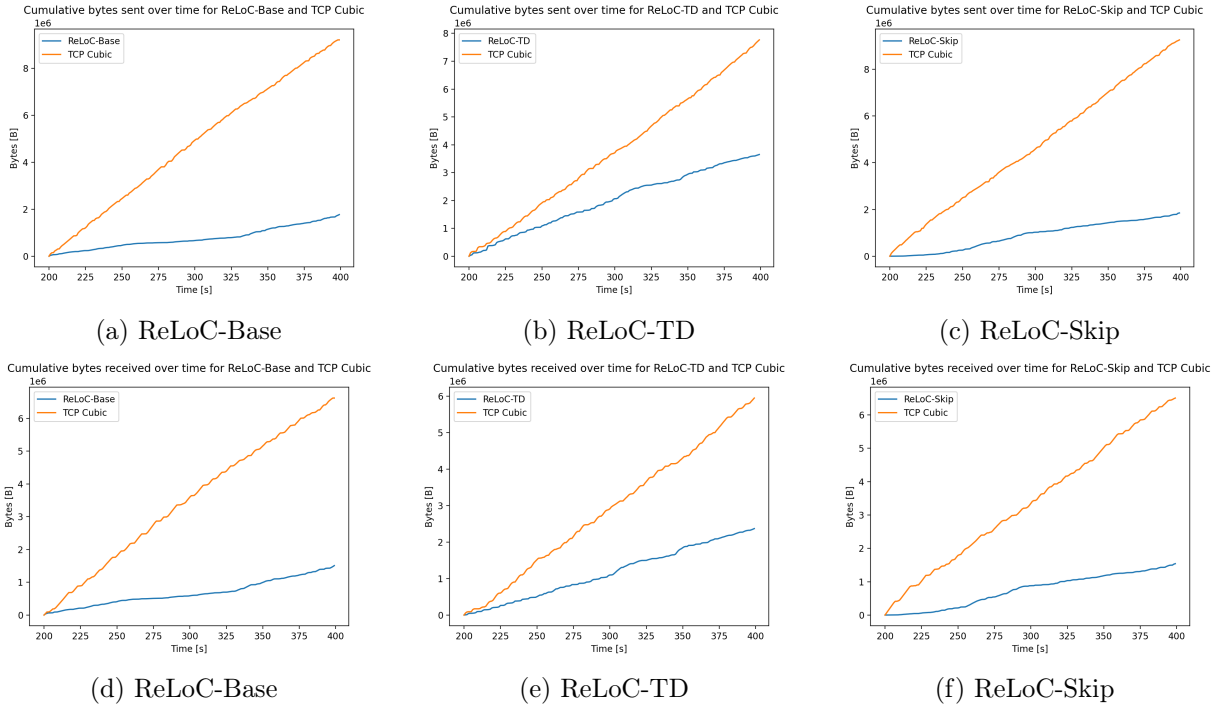
Test results

It was shown throughout this test that training the ReLoC variations to become more generalized did not significantly improve their performance.

5.5.2 Test 9

This test has a node using ReLoC running alone for 200 seconds. At the 200-second mark, a node using TCP Cubic begins transmitting. The capacity of the connection remains constant throughout the entire test. The full test description can be found in Section 5.1.6.

Throughput results



(a) ReLoC-Base (b) ReLoC-TD (c) ReLoC-Skip

(d) ReLoC-Base (e) ReLoC-TD (f) ReLoC-Skip

Figure 5.56. Visualization of cumulative bytes sent and received throughout the test after time-step 200 summed

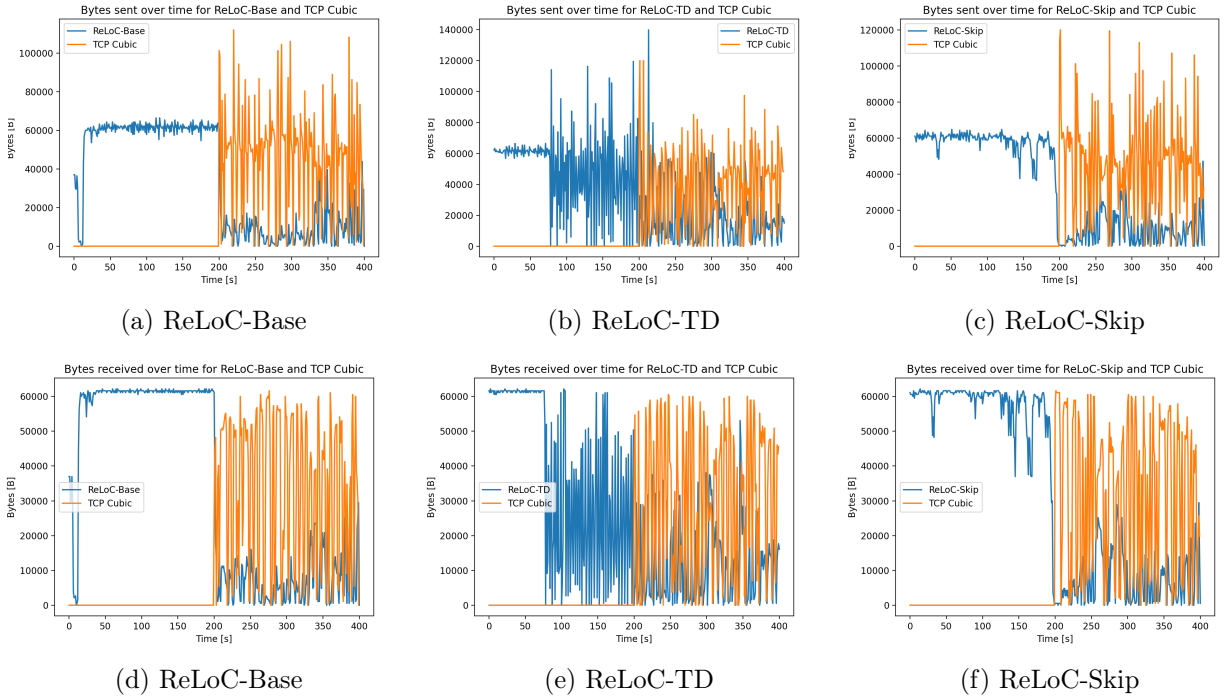
CCA	ReLoC-Base	ReLoC-TD	ReLoC-Skip
Cumulative bytes received: ReLoC	1,507,768	2,386,272	1,540,464
Cumulative bytes received: TCP Cubic	6,618,528	5,948,528	6,505,968

Table 5.23. Cumulative bytes received after time step 200 for each CCA

As this test revolves around the interaction of starting a TCP Cubic controlled flow while an already established ReLoC flow is present, the summed amount of data is relevant before the introduction of TCP Cubic takes place. This event happens at time-step 200, thus the plots in Figure 5.56 start 200 seconds into the simulation.

As is visualized in Figure 5.56 none of the ReLoC CCAs manage to sustain their hold on the available capacity. Immediately after 200 seconds of simulation time has passed, TCP Cubic takes a majority of the available link capacity. This means that even though TCP Cubic has only just been initialized, none of the ReLoC CCAs are capable of competing with TCP Cubic's aggressive nature. The data

provided in Table 5.23 shows that the best implementation at competing with TCP Cubic is ReLoC-TD, which is still unable to successfully receive data at even half the rate of TCP Cubic. This shows that even when ReLoC has full usage of the available capacity, it is a trivial task for TCP Cubic to overtake a majority of the capacity almost immediately.

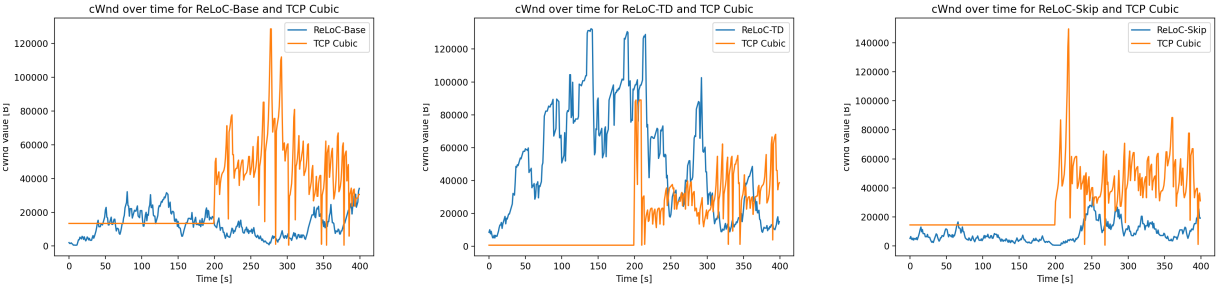


(a) ReLoC-Base (b) ReLoC-TD (c) ReLoC-Skip

(d) ReLoC-Base (e) ReLoC-TD (f) ReLoC-Skip

Figure 5.57. Visualization of sent and received bytes for each time-step

During the initial 200 seconds, where ReLoC has exclusive access to the available capacity, the ReLoC-TD implementation in Figure 5.57b shows a fluctuating sending rate, that goes above the efficient threshold, resulting in suboptimal receiving rate even before TCP Cubic is introduced. After the introduction of TCP Cubic, both the ReLoC-Base and ReLoC-Skip implementations have both their sending and receiving rates significantly lowered as TCP Cubic fills the available capacity. The outlier once again is ReLoC-TD, which loses significantly less of its sending and receiving rate, which is the reason it has the best overall result of the ReLoC CCAs as seen in Table 5.25.



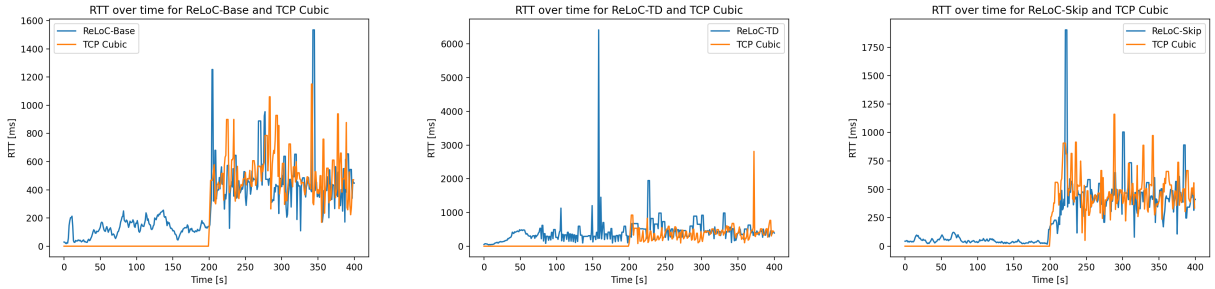
(a) ReLoC-Base (b) ReLoC-TD (c) ReLoC-Skip

Figure 5.58. Visualization of the cWnd value for each time-step

Figures 5.58a and 5.58c show the same cWnd lower than that of TCP Cubic even before it was introduced in the simulation. This allows TCP Cubic to aggressively probe for capacity without

opposition. In comparison, ReLoC-TD is more competitive in this regard, as Figure 5.58b shows that an early increase in cWnd for ReLoC-TD led to it remaining relatively competitive for a longer duration, however, the cWnd for ReLoC-TD can be seen slowly dropping as the simulation drags on, leading to TCP Cubic slowly taking control of more of the total capacity.

RTT results



(a) ReLoC-Base (b) ReLoC-TD (c) ReLoC-Skip

Figure 5.59. Visualization of the average RTT in ms for each time-step

CCA	ReLoC-Base	ReLoC-TD	ReLoC-Skip
RTT(0,199) ReLoC	124.59	343.82	48.43
RTT(200,400) ReLoC	469.75	517.57	456.90
RTT(200,400) TCP Cubic	513.05	395.85	489.15

Table 5.24. Average RTT in ms for each CCA

Based on both Figure 5.59 and Table 5.24 it can be assessed that the introduction of another flow makes the otherwise stable network begin to fluctuate because of the quickly changing nature of TCP Cubic. This introduction of TCP Cubic also results in an overall higher average RTT for all ReLoC CCAs. This is once again the results of TCP Cubic's aggressive probing for more capacity and its loss-based congestion control. It could therefore be theorized that the addition of a less aggressive CCA such as TCP Vegas would not increase the overall RTT to this extent.

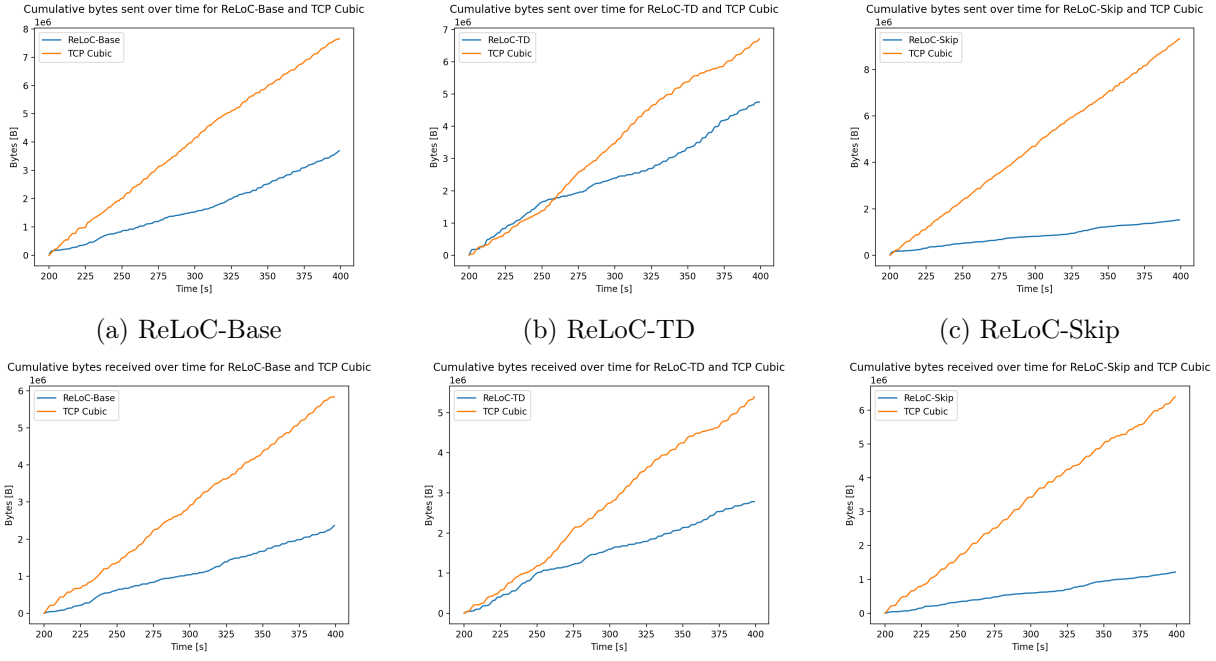
Test conclusion

Earlier tests have shown ReLoC CCAs to be less aggressive than TCP Cubic, this becomes even more apparent as the introduction of this CCA, in a network with an already stable ReLoC connection, completely outcompetes all ReLoC variants in the same fashion. This "takeover" was in one case slightly delayed by sacrificing some focus on maintaining lower RTT, however, this proved ineffective as the simulation continued.

5.5.3 Test 10

This test has a node using TCP Cubic running alone for 200 seconds. At the 200-second mark, a node using ReLoC begins transmitting. The capacity of the connection remains constant throughout the entire test. The full test description can be found in Section 5.1.6.

Throughput results



(a) ReLoC-Base (b) ReLoC-TD (c) ReLoC-Skip

(d) ReLoC-Base (e) ReLoC-TD (f) ReLoC-Skip

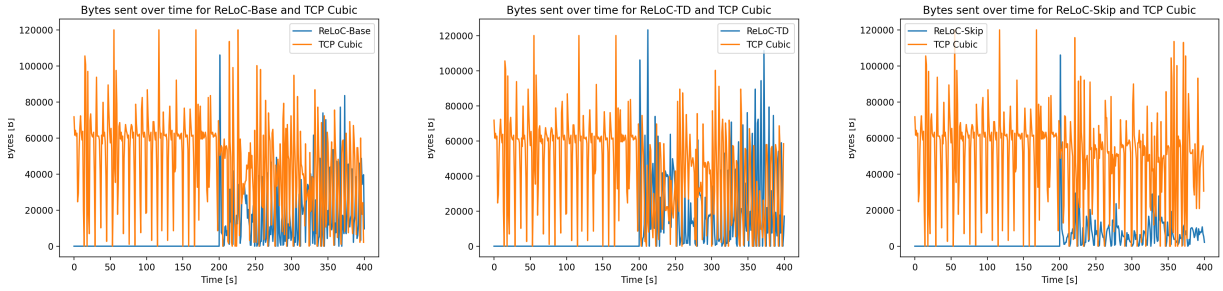
Figure 5.60. Visualization of cumulative bytes sent and received throughout the test after time-step 200 summed

CCA	ReLoC-Base	ReLoC-TD	ReLoC-Skip
Cumulative bytes received: ReLoC	2,375,016	2,785,592	1,216,184
Cumulative bytes received: TCP Cubic	5,837,576	5,392,160	6,388,048

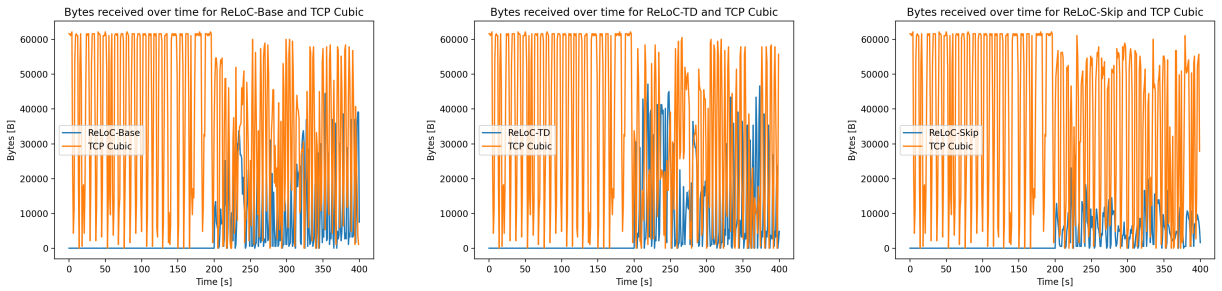
Table 5.25. Cumulative bytes received after time step 200 for each CCA

Like test 9 seen in Subsection 5.5.2 the interaction between ReLoC and TCP Cubic does not begin until 200 seconds into the simulation. As a result, the plots in Figure 5.60 start after 200 seconds of simulation time.

Figure 5.60 once again shows TCP Cubic taking a majority of the available capacity. However, in comparison to earlier tests, the results here are not as one-sided. Especially Figure 5.60e details that when ReLoC-TD first initiates it is able to compete at a fair level, being only slightly behind in bytes received, and as evident by Figure 5.60b, send more data at around time-step 250. While this fairness dwindles as the end simulation moves onward, the ability to compete with TCP Cubic's aggressive probing while it utilizes almost the entire capacity is far greater than what was shown in the previous test in Subsection 5.5.2. By comparing Table 5.25 with Table 5.23 it can be concluded that the implementation of ReLoC-Base and ReLoC-TD are both better at competing with TCP Cubic when they are the just introduced into the network, rather than if TCP Cubic is introduced at a later instance. This is likely a result of the trained policy's ability to receive time-distributed data.



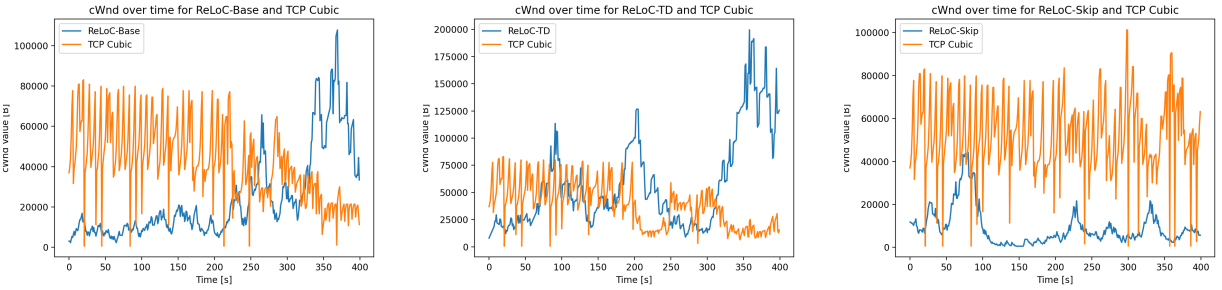
(a) ReLoC-Base (b) ReLoC-TD (c) ReLoC-Skip



(d) ReLoC-Base (e) ReLoC-TD (f) ReLoC-Skip

Figure 5.61. Visualization of sent and received bytes for each time-step

Once again Figure 5.61 shows the influence of the ReLoC CCAs being introduced in the simulation, where both the rate of sending and receiving is reduced after 200 seconds into the simulation. It is also reemphasized how by Figures 5.61c and 5.61f that the ReLoC-Skip implementation is the least efficient of the CCAs at competing with TCP Cubic for available capacity.

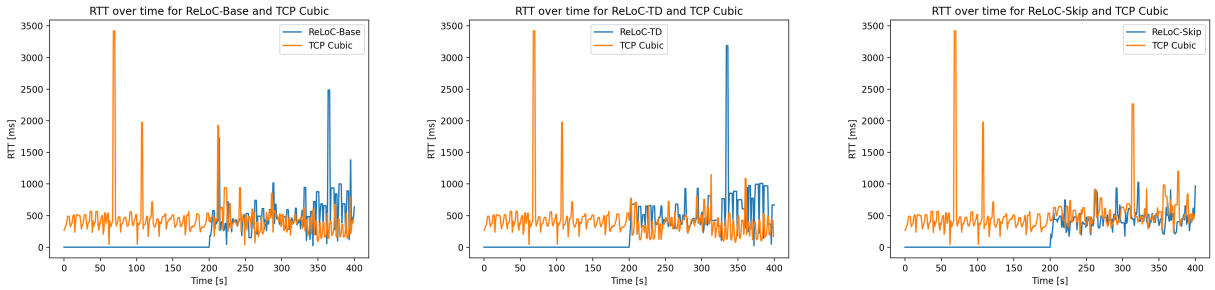


(a) ReLoC-Base (b) ReLoC-TD (c) ReLoC-Skip

Figure 5.62. Visualization of the cWnd value for each time-step

Figure 5.62 reveals a reason for the ReLoC CCAs ability to compete just after being introduced into the network. The cWnd for the ReLoC CCAs can be seen changing from the start of the simulation, where they have yet to be introduced into the network. This behavior is a mistake in the test implementation as the ReLoC CCA should not be active at this stage. Furthermore, since the state space only consists of information gained from sending packets and receiving corresponding ACKs, the ReLoC CCA is not able to get any state space information before the node using it is introduced 200 seconds into the simulation. The changes made the cWnd of the ReLoC CCA from time-step 0 to 200 can therefore be considered completely random, as the state space would consist of only zeroes.

RTT results



(a) ReLoC-Base (b) ReLoC-TD (c) ReLoC-Skip

Figure 5.63. Visualization of the average RTT in ms for each time-step

CCA	ReLoC-Base	ReLoC-TD	ReLoC-Skip
RTT(0,199) TCP Cubic	471.22	471.22	471.22
RTT(200,400) TCP Cubic	414.12	359.00	559.41
RTT(200,400) ReLoC	546.24	604.81	452.02

Table 5.26. Average RTT in ms for each CCA

From Table 5.26 it can be observed that the introduction of both ReLoC-Base and ReLoC-TD into the simulation reduces the average RTT that TCP Cubic's packets experience. As this is not the case for the ReLoC-Skip implementation it would imply, that the additional competition for capacity reduces the RTT of TCP Cubic. This interaction is likely as result of the queue being filled more quickly while TCP Cubic has an overall lower cWnd, but the exact reason for the phenomenon is uncertain.

Test conclusion

While ReLoC-Base and ReLoC-TD are able to compete reasonably well for some of the available capacity after being introduced, this is likely influenced by the cWnd of the CCAs being changeable before the CCA itself is introduced to the simulation. This test implementation oversight allowed the cWnd to start a higher value than otherwise possible. Whether this change amounted to the entire difference between the ReLoC CCAs ability to compete with TCP Cubic in comparison to the results seen in Test 8 in Section 5.5.1 and Test 9 in Section 5.5.2 is uncertain.

6 | Discussion

This chapter contains a discussion based on the implemented ReLoC CCAs and their designs. This discussion will also include eventual relation to the test results detailed in Chapter 5. In addition, a section dedicated to options for improving the current implementation through optimization or alternative methods is included.

The implementation of ReLoC is based on an extensive design phase. Part of this design was to isolate key features for identifying congestion. These features are used as the state variables and include packets sent, packets received, average RTT, and standard deviation of RTT. The action space was limited to only 5 actions to avoid redundant actions that would slow down learning. Two of which increase the cWnd, two which decrease it, and one which maintains its current value. Then a reward function, that rewards sending as close to the available capacity as possible while punishing increases in RTT. This was implemented together with the RL algorithm PPO to create the proof of concept RL-based CCA called ReLoC. Two variations of ReLoC were also implemented, ReLoC-TD and ReLoC-Skip, which respectively added a time-distributed layer and a skip connection to the agent network architecture.
four different versions were trained of each variant. These versions were based on training in a deterministic or randomized simulation and whether or not a different TCP connection would be present to compete for available capacity.

Throughout the test chapter, a lot of focus has been put on the performance of the various CCAs. But it may be worth to question what a good CCA is. Historically, selfish CCAs that seek to maximize the throughput for the individual has been the most commonly used algorithms. This was seen in the case with TCP Vegas, which showed great performance when used on a network alone or with other connections also using TCP Vegas. However, with a more aggressive CCA present, TCP Vegas gets dominated, losing a lot of performance to the more aggressive node. Due to examples like this, aggressive algorithms have been the standard for CCAs, as the alternative risks sacrificing throughput to someone who acts selfishly. However, the usage of aggressive CCAs means that the nodes will be competing for capacity, causing unoptimized usage of the available capacity. For better usage of the available capacity, an altruistic approach should be applied. Instead of competing for available capacity, the nodes should try to share the capacity evenly. While this may cause the individual to lose performance, it would increase the performance of the collective. For this reason, it can be gathered that a good CCA is not necessarily one that can simply maximize throughput, but rather one that can maximize its throughput without impeding the available capacity of other nodes. If such a CCA could operate at Kleinrock's point of optimality, then the nodes would be able to use the bandwidth optimally, with all nodes gaining high throughput along with low RTT.

Results
Testing the three ReLoC variations trained on a deterministic training simulation in a network without other TCP connections showed promising results. The ReLoC CCAs were all able to perform at a level above or equal to TCP Cubic. This allows the implementation to be a successful proof of concept for a DRL-based CCA.
In an attempt to get a more generalized solution, the variant with random training and no competing TCP connection was tested. While it, like the deterministically trained version, was able to perform

at a level equal to or above TCP Cubic in tests 1 to 4. The implementation performed very poorly in tests 1 to 4, where it was simulated in a network competing with TCP Cubic. Here it showed that ReLoC could not handle the aggressive nature of TCP Cubic.
The ReLoC implementations trained on a deterministic network with TCP Cubic present were next. These implementations were not able to compete evenly with TCP Cubic at most levels of available capacity. However, it was seen that while the variant ReLoC-Skip prioritized low RTT, the other variants ReLoC-Base and ReLoC-TD were competing more aggressively with TCP Cubic, while even managing to utilize more capacity than TCP Cubic when the available capacity was low. This meant the implementations had instances where competing with TCP Cubic was possible.
The last set of variants was trained on a simulation with random elements where a competing node utilizing TCP Cubic was connected. These versions only serve to emphasize the issues with the previous set. The ability to compete with TCP Cubic is only present when the available capacity is reduced. This behavior is undesirable, as ReLoC forces congestion during these segments, reducing the number of packets TCP Cubic can successfully send while sending at a considerable rate.

6.1 Future improvements

This section goes over multiple possible changes or additions to the current implementation of the ReLoC CCAs. Some of these improvements are based on the received test results, while others were considered outside the scope of the project.

An issue that appeared during the test results was the fact that the UDP connection, which was used to limit the available capacity, used more than the specified amount. E.g. when the UDP rate was set to 50% of the available capacity around 75% capacity was being used. The excess amount used was not linear nor did it match the number of bytes needed for the header of the UDP packets. As a result, the simulation network implementation is faulty, leading to less available capacity than expected. Since the intended available capacity is used in the reward function as a variable, this fault has likely affected the overall training of ReLoC. Any future work based on this project should seek to resolve or avoid this problem.

More training
When training RL agents it is important to train until the agent's training score converges. The speed at which the training score converges is dependent on both the specifics of the agent and the environment it is trained on. Thus, since the randomized simulation training has a large amount of possible states, that the agent can experience, it will take longer for the agent to fully explore and learn the correct action in each state of the environment. It is therefore possible that the agent's training score had not started converging, which would lead to a non-ideal policy.

Hyperparameter optimization
While most hyperparameter choices were made based on qualified guesses, previous experiences, or industry standards this does not guarantee these are optimal values for this specific problem. The time limitations for the project made it impossible to test for alternative values for all hyperparameters. One set of parameters, which were prioritized and tested for local maxima, was the alpha/beta scalars used in the reward function. Similar tests for further hyperparameters would lead to an overall increase in performance.

Realistic training simulaiton
In an attempt to create more generalized policies that would not over-specialize on the specific training simulation, variation was implemented into the training. This variation consisted of varying the size of the reduction to available capacity throughout the test. However, this meant every other aspect was stagnant. This training simulation therefore bore very little resemblance to communication in a real network. Since ReLoC is intended to work on a real network, training on a real network or a simulation that matches closely would lead to a more applicable implementation.

Specialized reward function
The reward function used for training the model was created to optimize the amount of data being received while maintaining low RTT. This worked well initially in tests 1 through 4, however, when introducing another TCP connection, which will compete for available network capacity, then the reward function should be redesigned to take concepts like fairness and other flows affecting the RTT and throughput into account.

Inverse reinforcement learning
While a lot of effort was put into creating a reward function, with empirical testing of the scalar values, other methods can be used to approximate optimal scalars for reward functions. Inverse Reinforcement Learning (IRL) is a method used for approximating optimal scalars for reward functions by using datasets of simulations where the optimal actions are performed. The computer then attempts to find the scalar values that can lead to a policy that takes these actions. By creating a large dataset, a reward function that is able to produce a policy with high performance and can generalize well can be achieved. However, creating such datasets is highly time-consuming. Additionally, as the supposedly optimal actions are hand-crafted, human bias can be a risk of lowering the potential of the RL process. The usage of IRL has been proven to work for DRL-based CCAs by Luo et al. [40]. Here Luo et al. reimplements Aurora [20] using IRL showing that this implementation was capable of improving the performance reached using the standard implementation of Aurora.

7 | Conclusion

CCAs are a vital tool for getting fast and reliable network communication when using a TCP connection. The classical CCAs are commonly created using hand-crafted heuristics, which lead to controlled systematic behavior that utilizes available network capacity at an efficient level but not optimally. In the search for a more optimal CCA, the concepts of ML presents a potential solution. As this problem is suited for the training of RL-based policies, this is an approach worth exploring. Furthermore, by using a DRL approach, more complex decisions can be made based on network conditions, which could improve the overall performance compared to standard hand-crafted CCAs. However, in the search for a better CCA the interaction between other CCAs should also be considered, as individual performance would be an overall loss if it came at the cost of another TCP client being unable to communicate effectively. As such, it is important that an ML-based CCA is capable of fairly competing for available bandwidth, so as to not be forced away by aggressive CCAs while also not forcing other common CCAs away. Through these considerations, a problem formulation has been constructed as follows:

> *How can deep reinforcement learning be used to learn a policy for TCP congestion control that is on par or outperforms classical congestion control algorithms based on hand-tuned heuristics while behaving fairly?*

In an attempt to create a DRL-based CCA, an RL algorithm was first selected in the form of PPO. A PPO agent makes use of an actor and a critic NN. These networks have been implemented as an NN with deep LSTM layers, which allows them to make use of information from previous states. This implementation was named ReLoC and trained in a network simulation with varying levels of available capacity. Three agents were created from the implementation, one base and two variations. These variations were created with additional features in the DLSTMs of the actor and critic. The first variation had a time-distributed fully connected layer before the LSTM layers and was named ReLoC-TD, the other variation had a skip-connection from the original input to after the LSTM layers, this variant is referred to as ReLoC-Skip. A total of four different iterations were trained for each of the three ReLoC variants based on four different training simulations. These iterations include:

1. UDP Deterministic: A deterministic simulation with no competing TCP connections.
2. UDP Random: A randomly generated simulation with no competing TCP connections.
3. TCP Deterministic: A deterministic simulation with a competing TCP Cubic connection.
4. TCP Random: A randomly generated simulation with a competing TCP Cubic connection.

During the tests, it became apparent that all the UDP-trained iterations of the ReLoC CCAs were able to effectively utilize the available network capacity in a network without a competing TCP connection. Especially the ReLoC-Skip variation achieved a particularly low RTT while still utilizing the full capacity. However, when introducing another TCP connection utilizing TCP Cubic, the test results showed very poor performance. This was a result of TCP Cubic utilizing almost the entirety of the available capacity alone. The TCP-trained iterations of ReLoC were tested against TCP Cubic, these

iterations showed both did better at competing with TCP Cubic, and at some stages of the tests, the ReLoC varaitons were able to utilize more capacity than TCP Cubic. However, when the ReLoC outperforms TCP Cubic, the total amount of available capacity utilized is lower than when TCP Cubic uses the majority. This is especially the case for ReLoC-TD, which puts a larger emphasis on competing for capacity by sending as many packets as possible causing heavy congestion. As a result, the ReLoC variations are inefficient when exposed to a TCP Cubic connection as both CCAs' overall performance will drop due to the competition.

From the test results, it can be concluded that the policies produced by ReLoC are capable of efficiently utilizing the available capacity in a network without a competing TCP connection. However, introducing another TCP flow reduces the performance of both flows. Despite this, the results still serve as a proof of concept that a DRL-based CCA has the potential to use more of the available network capacity than the classical hand-crafted CCAs.
This leaves openings for future research to be focused on training an agent, that can compete at a fair level with popular CCAs, such as TCP Cubic, without losing overall capacity utilization. Whether this is done by optimizing the parameters of the current implementation, fixing its shortcomings, or creating an entirely different implementation, are all possible options.

Bibliography

[1] International Telecommunication Union. Global offline population steadily declines to 2.6 billion people in 2023. https://www.itu.int/itu-d/reports/statistics/2023/10/10/ff23-internet-use/, November 2023. (Accessed on 30/05/2024).

[2] Mohd Mohamad, Mudassar Ahmad, and Md Ngadi. Experimental evaluation of tcp congestion contorl mechanisms in short and long distance networks. *Journal of Theoretical and Applied Information Technology*, 71:153–166, 01 2015.

[3] Wenting Wei, Huaxi Gu, and Baochun Li. Congestion control: A renaissance with machine learning. *IEEE Network*, 35(4):262–269, 2021. doi: 10.1109/MNET.011.2000603.

[4] Peter Babington. *Introduction to Computer Networks.* Peter L. Dordal, 1032 W Sheridan Rd, Chicago, IL 60660, United States, 2 edition, july 2020. This book was self-published and therefore does not have an ISBN/ISSN. Google Books entry: https://books.google.dk/books?id=i_owzwEACAAJ.

[5] Tcp tahoe and tcp reno - geeksforgeeks. https://www.geeksforgeeks.org/tcp-tahoe-and-tcp-reno/, march 2013. (Accessed on 03/12/2024).

[6] Sangtae Ha, Injong Rhee, and Lisong Xu. Cubic: a new tcp-friendly high-speed tcp variant. *ACM SIGOPS operating systems review*, 42(5):64–74, 2008.

[7] Lawrence S Brakmo, Sean W O'Malley, and Larry L Peterson. Tcp vegas: New techniques for congestion detection and avoidance. In *Proceedings of the conference on Communications architectures, protocols and applications*, pages 24–35, 1994.

[8] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. Bbr: Congestion-based congestion control: Measuring bottleneck bandwidth and round-trip propagation time. *Queue*, 14(5):20–53, oct 2016. ISSN 1542-7730. doi: 10.1145/3012426.3022184. URL https://doi.org/10.1145/3012426.3022184.

[9] F. Emmert-Streib and M. Dehmer. Taxonomy of machine learning paradigms: a data-centric perspective. *WIREs Data Mining and Knowledge Discovery*, 12, 2022. doi: 10.1002/widm.1470.

[10] Paul Fieguth. *An introduction to pattern recognition and machine learning.* Springer, Cham, Switzerland, 2022. ISBN 9783030959951.

[11] Rafael Figueiredo Prudencio, Marcos R. O. A. Maximo, and Esther Luna Colombini. A survey on offline reinforcement learning: Taxonomy, review, and open problems. *IEEE Transactions on Neural Networks and Learning Systems*, pages 1–0, 2023. doi: 10.1109/TNNLS.2023.3250269.

[12] S.L. Brunton and J.N. Kutz. *Data-Driven Science and Engineering: Machine Learning, Dynamical Systems, and Control.* Cambridge University Press, 2019. ISBN 9781108422093. URL https://books.google.dk/books?id=CYaEDwAAQBAJ.

[13] R.S. Sutton and A.G. Barto. *Reinforcement Learning, second edition: An Introduction*. Adaptive Computation and Machine Learning series. MIT Press, 2018. ISBN 9780262039246. URL https://books.google.dk/books?id=5s-MEAAAQBAJ.

[14] P Ivan Pavlov. Conditioned reflexes: an investigation of the physiological activity of the cerebral cortex. *Annals of neurosciences*, 17(3):136, 2010.

[15] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy P. Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *CoRR*, abs/1712.01815, 2017. URL http://arxiv.org/abs/1712.01815.

[16] Michael Volkov. File:starting position in a chess game.jpg - wikimedia commons. https://commons.wikimedia.org/wiki/File:Starting_position_in_a_chess_game.jpg, August 2011. (Accessed on 05/04/2024).

[17] Stefan Steinerberger. On the number of positions in chess without promotion. *International Journal of Game Theory*, 44(3):761–767, 2015.

[18] Joyce Fang, Martin Ellis, Bin Li, Siyao Liu, Yasaman Hosseinkashi, Michael Revow, Albert Sadovnikov, Ziyuan Liu, Peng Cheng, Sachin Ashok, et al. Reinforcement learning for bandwidth estimation and congestion control in real-time communications. *arXiv preprint arXiv:1912.02222*, 2019.

[19] Viswanath Sivakumar, Olivier Delalleau, Tim Rocktäschel, Alexander H Miller, Heinrich Küttler, Nantas Nardelli, Mike Rabbat, Joelle Pineau, and Sebastian Riedel. Mvfst-rl: An asynchronous rl framework for congestion control with delayed actions. *arXiv preprint arXiv:1910.04054*, 2019.

[20] Nathan Jay, Noga Rotman, Brighten Godfrey, Michael Schapira, and Aviv Tamar. A deep reinforcement learning perspective on internet congestion control. In *International Conference on Machine Learning*, pages 3050–3059. PMLR, 2019.

[21] Salma Emara, Baochun Li, and Yanjiao Chen. Eagle: Refining congestion control by learning from the experts. In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*, pages 676–685. IEEE, 2020.

[22] Zhiyuan Xu, Jian Tang, Chengxiang Yin, Yanzhi Wang, and Guoliang Xue. Experience-driven congestion control: When multi-path tcp meets deep reinforcement learning. *IEEE Journal on Selected Areas in Communications*, 37(6):1325–1336, 2019.

[23] Soheil Abbasloo, Chen-Yu Yen, and H Jonathan Chao. Classic meets modern: A pragmatic learning-based congestion control for the internet. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 632–647, 2020.

[24] Hanbing Shi and Juan Wang. Intelligent tcp congestion control policy optimization. *Applied Sciences*, 13:6644, 05 2023. doi: 10.3390/app13116644.

[25] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[26] OpenAI. Deep deterministic policy gradient. `https://spinningup.openai.com/en/latest/algorithms/ddpg.html`, July 2019. (Accessed on 10/05/2024).

[27] Scott Fujimoto, Herke Hoof, and David Meger. Addressing function approximation error in actor-critic methods. In *International conference on machine learning*, pages 1587–1596. PMLR, 2018.

[28] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[29] OpenAi. Proximal policy optimization — spinning up documentation. `https://spinningup.openai.com/en/latest/algorithms/ppo.html`, August 2017. (Accessed on 10/05/2024).

[30] Lasse Espeholt, Hubert Soyer, Remi Munos, Karen Simonyan, Vlad Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, et al. Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures. In *International conference on machine learning*, pages 1407–1416. PMLR, 2018.

[31] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *International conference on machine learning*, pages 1889–1897. PMLR, 2015.

[32] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International conference on machine learning*, pages 1861–1870. PMLR, 2018.

[33] Liang Cheng and Ivan Marsic. Fuzzy reasoning for wireless awareness. *International Journal of Wireless Information Networks*, 8, 09 2002. doi: 10.1023/A:1011329512150.

[34] Dominik Scholz, Benedikt Jaeger, Lukas Schwaighofer, Daniel Raumer, Fabien Geyer, and Georg Carle. Towards a deeper understanding of tcp bbr congestion control. In *2018 IFIP Networking Conference (IFIP Networking) and Workshops*, pages 1–9, 2018. doi: 10.23919/IFIPNetworking.2018.8696830.

[35] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL `https://www.tensorflow.org/`. Software available from tensorflow.org.

[36] Jakob Lund Jensen and Peter Rudbæk Valentinussen. Implementation platform for reinforcement learning-based congestion control algorithms. AAU projektbibliotek, December 2023. 9^{th} semester project by the authors of this project. Access to the related report can be requested from the authors.

[37] nsnam. ns-3.40 released | ns-3. https://www.nsnam.org/news/2023/09/27/ns-3-40-released.html, September 2023. (Accessed on 22/05/2024).

[38] Python. subprocess — subprocess management. https://docs.python.org/3/library/subprocess.html, May 2024. (Accessed on 22/05/2024).

[39] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*, 2015.

[40] Pengcheng Luo, Yuan Liu, Zekun Wang, Jian Chu, and Genke Yang. A novel congestion control algorithm based on inverse reinforcement learning with parallel training. *Computer Networks*, 237:110071, 2023. ISSN 1389-1286. doi: https://doi.org/10.1016/j.comnet.2023.110071. URL https://www.sciencedirect.com/science/article/pii/S1389128623005169.

Part IV

Appendices

A | Overview of other DRL CCAs

Algorithm	Tuning knobs	Input
Aurora	Rate-based	Latency gradient, latency ratio, and sending ratio
R3Net	Rate-based	Receive rate, average packet interval, packet loss rate, and average RTT
MVFST-RL	Window-based	RTT, queuing delay, packets sent, packets acknowledged, packets dropped, and 15 features that were not described
Eagle	Rate-based	T/F whether delay has been experienced, difference between number of increases and decreases in sending rate if a long delay was experienced, exponentially weighted moving average of queuing delay, exponentially weighted moving average of loss rate, and ratio of delivery rate
DRL-CC	Window-based	Sending rate, goodput, average RTT, mean deviation of RTT, and cWnd
Orca	Window-based	Throughput, average loss rate of packets, average delay of packets, number of acknowledged packets, time between last report and current report, smooth RTT, cWnd, maximum throughput so far, and minimum packet delay so far
TCP-PPO$_2$	Window-based	Current relative time, size of cWnd, number of bytes not acknowledged, quantity of ACKs obtained, RRT, throughput rate, and number of packet loss

Algorithm	Action space
Aurora	Set rate to specific number with a modifier for stabilizing
R3Net	Set rate to a value between 0 and 8 Mb/s
MVFST-RL	cWnd : { cWnd, cWnd/2, cWnd - 10, cWnd + 10, cWnd * 2}
Eagle	rate : { rate/2.89, rate/1.25, 1*rate, 1.25*rate, 2.89*rate}
DRL-CC	Set cWnd to a specific value
Orca	cWnd = 2^{alpha} *cWnd$_u$, (-2<alpha<2) cWnd is then used to calculate: rate = (cWnd)/(sRTT)
TCP-PPO$_2$	cWnd = cWnd$_{old}$ + a value derived from the observed state parameter information

Algorithm	Reward function
Aurora	R = 10*throughput - 1000*latency - 2000*loss
R3Net	R = 0.6ln(4*receive rate + 1) - avg RTT - 10*packet loss rate
MVFST-RL	R = log(avg throughput + stabilizer) - scalar*log(maximum delay + stabilizer)
Eagle	R = 5*ratio in delivery rate
DRL-CC	R = sum of log(avg goodput of MPTCP flow during the past epoch)
Orca	R = ((throughput - scalar*loss)/(delay))/((max throughput)/(min delay))
TCP-PPO$_2$	R = scalar(current throughput/max observed throughput) - (1 - scalar)(shortest observed delay/average delay)

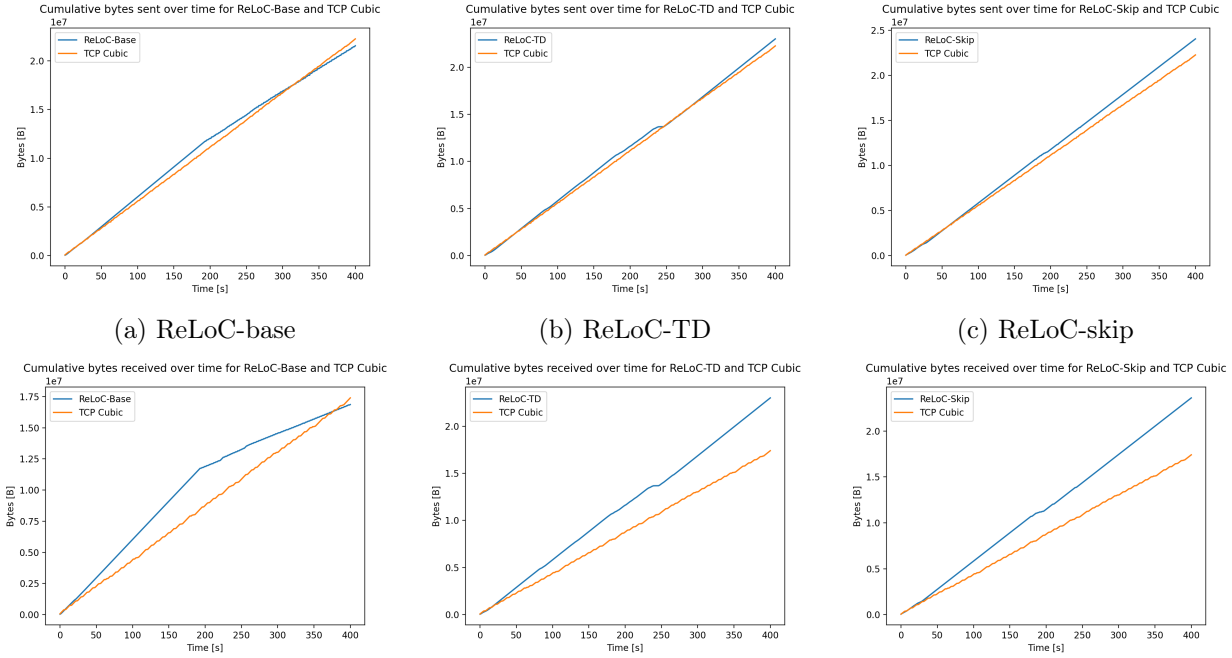
B | Test results exempted from the main text due to redundancy

This appendix contains test results from Section 5.3 and 5.5 which were not included in the main text.

B.1 Additional test results for the generalized ReLoC variants trained without another TCP connection on the channel

B.1.1 Test 1

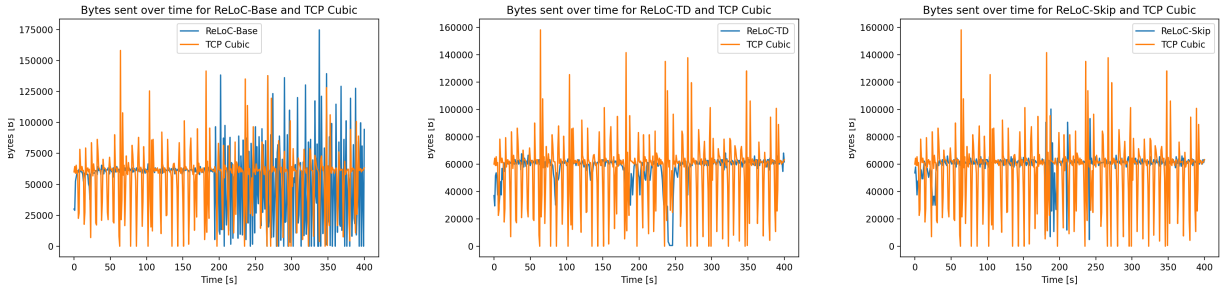
Throughput results



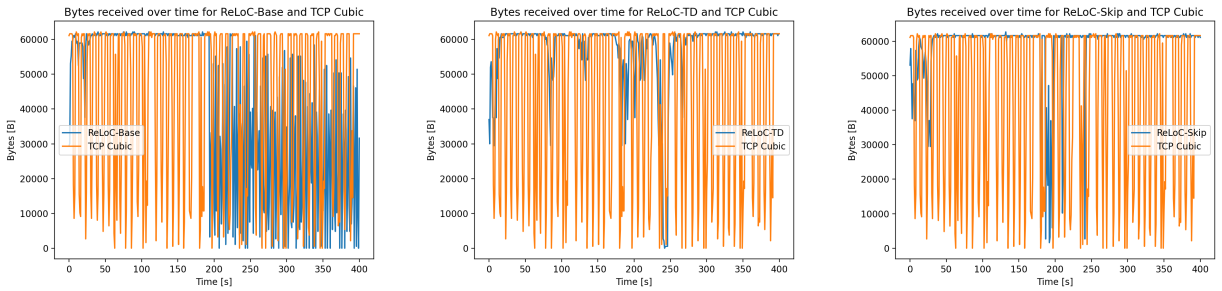
(a) ReLoC-base (b) ReLoC-TD (c) ReLoC-skip

(d) ReLoC-base (e) ReLoC-TD (f) ReLoC-skip

Figure B.1. Visualization of cumulative bytes sent and received throughout the test

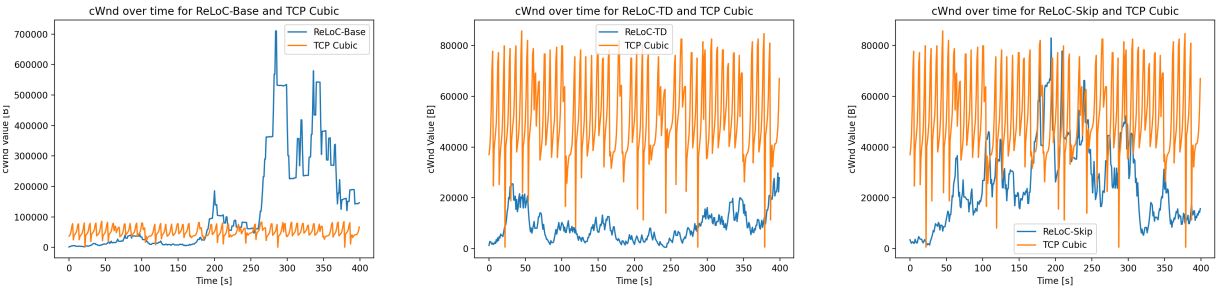


(a) ReLoC-base (b) ReLoC-TD (c) ReLoC-skip



(d) ReLoC-base (e) ReLoC-TD (f) ReLoC-skip

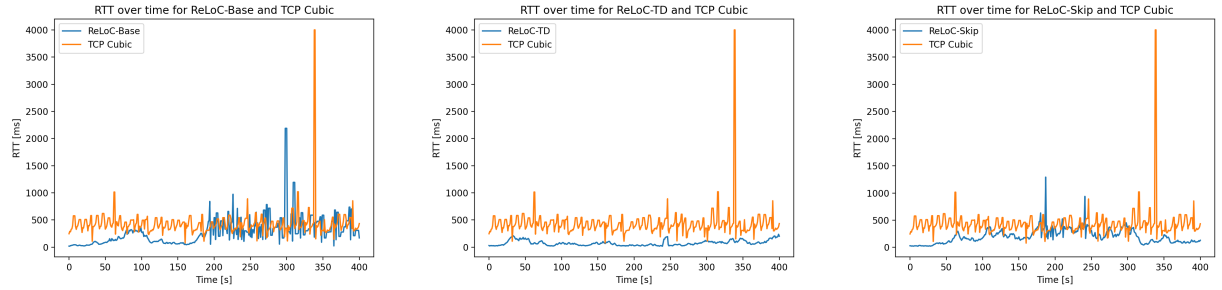
Figure B.2. Visualization of sent and received bytes for each time-step



(a) ReLoC-base (b) ReLoC-TD (c) ReLoC-skip

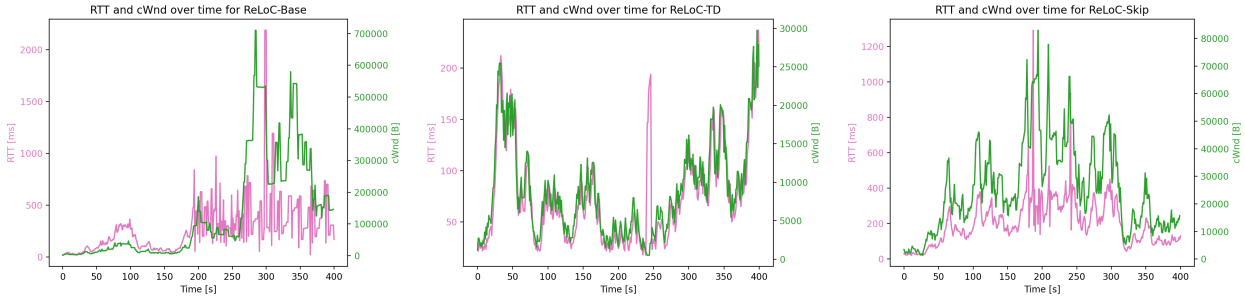
Figure B.3. Visualization of the cWnd value for each time-step

RTT results



(a) ReLoC-base (b) ReLoC-TD (c) ReLoC-skip

Figure B.4. Visualization of the average RTT in ms for each time-step

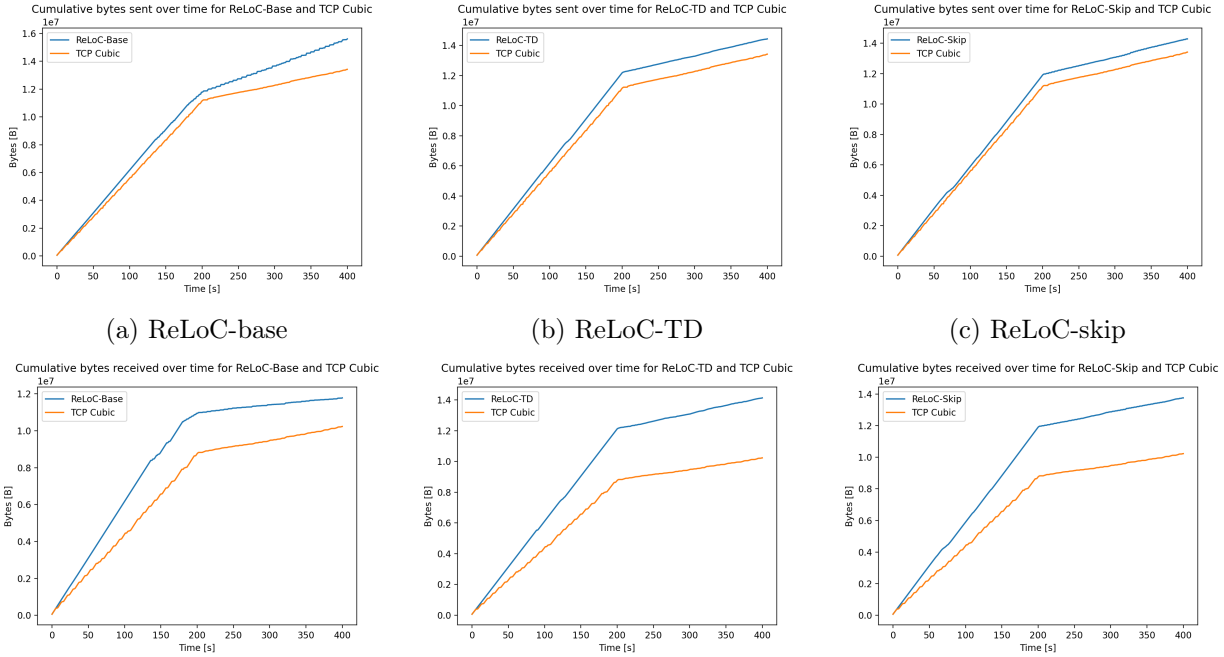


(a) ReLoC-base (b) ReLoC-TD (c) ReLoC-skip

Figure B.5. Visualization of the average RTT in ms and cWnd value plotted together for each time-step

B.1.2 Test 2

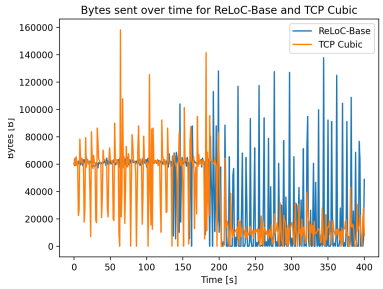
Throughput results



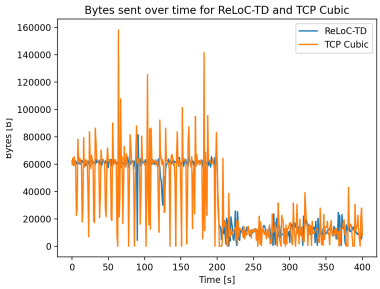
(a) ReLoC-base (b) ReLoC-TD (c) ReLoC-skip

(d) ReLoC-base (e) ReLoC-TD (f) ReLoC-skip

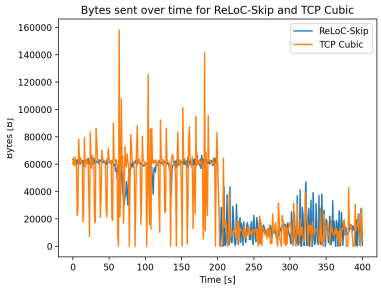
Figure B.6. Visualization of cumulative bytes sent and received throughout the test



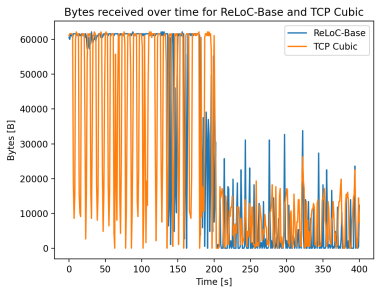
(a) ReLoC-base



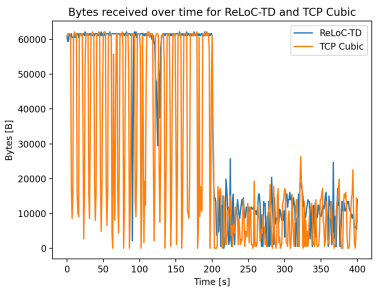
(b) ReLoC-TD



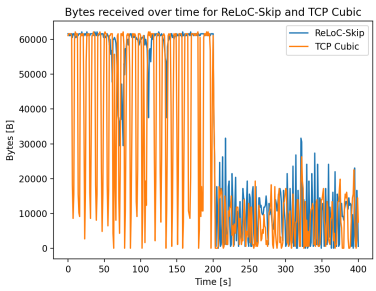
(c) ReLoC-skip



(d) ReLoC-base

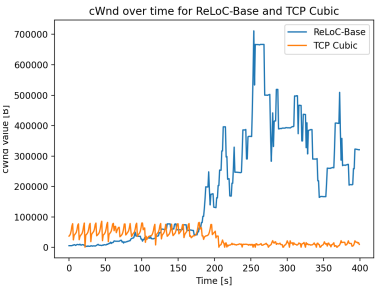


(e) ReLoC-TD

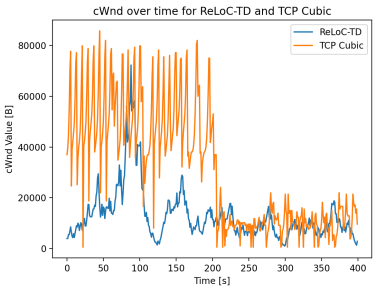


(f) ReLoC-skip

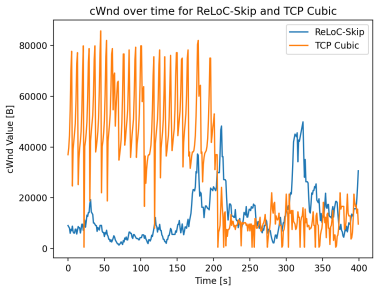
Figure B.7. Visualization of sent and received bytes for each time-step



(a) ReLoC-base



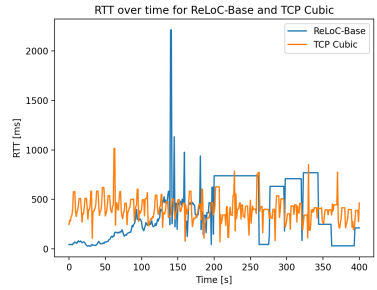
(b) ReLoC-TD



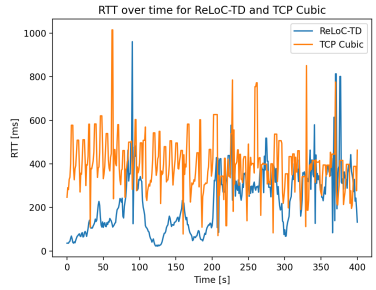
(c) ReLoC-skip

Figure B.8. Visualization of the cWnd value for each time-step

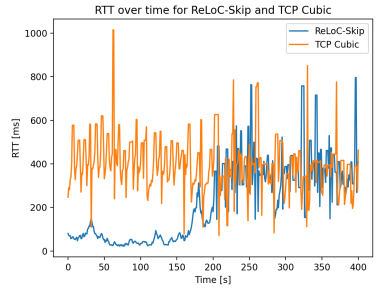
RTT results



(a) ReLoC-base

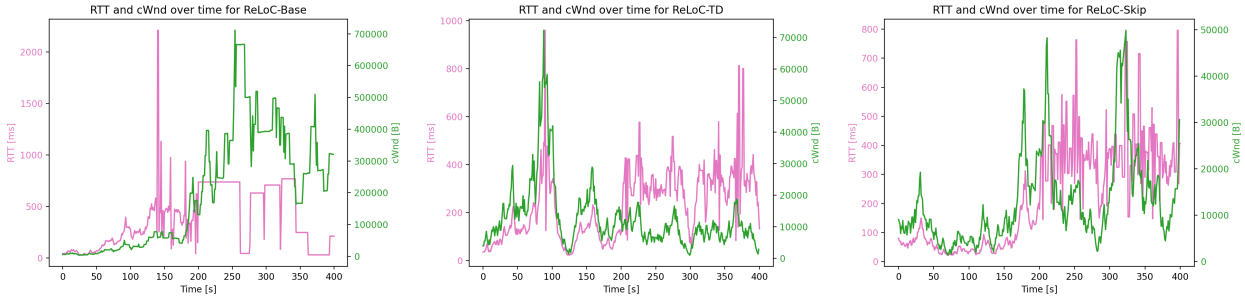


(b) ReLoC-TD



(c) ReLoC-skip

Figure B.9. Visualization of the average RTT in ms for each time-step

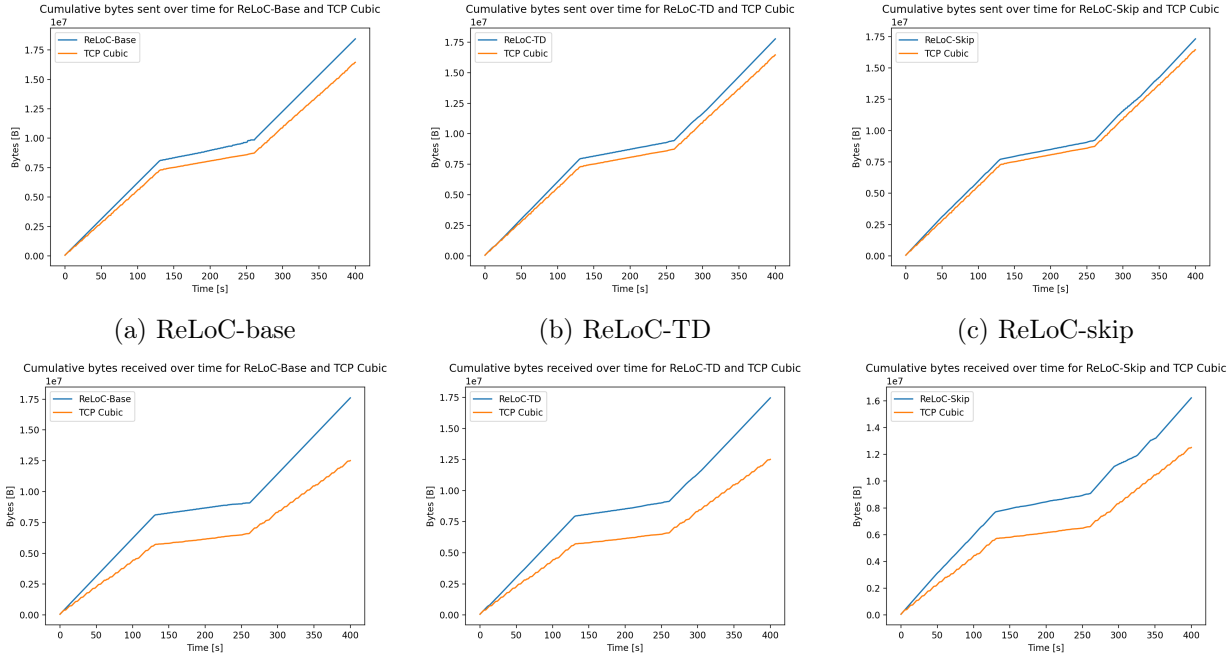


(a) ReLoC-base (b) ReLoC-TD (c) ReLoC-skip

Figure B.10. Visualization of the average RTT in ms and cWnd value plotted together for each time-step

B.1.3 Test 3

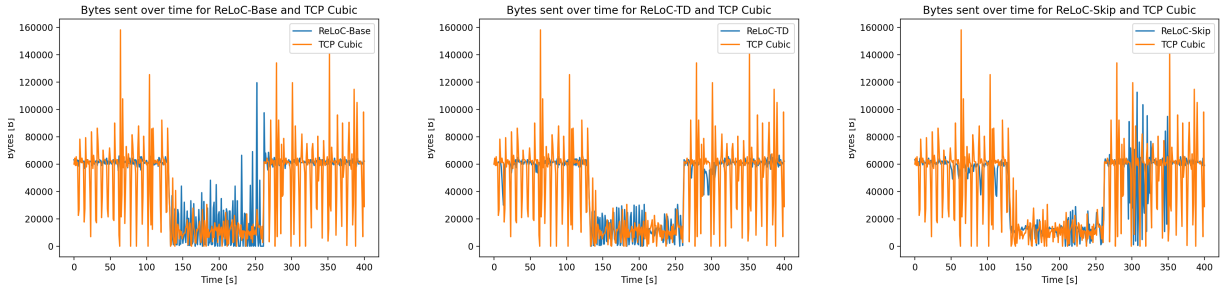
Throughput results



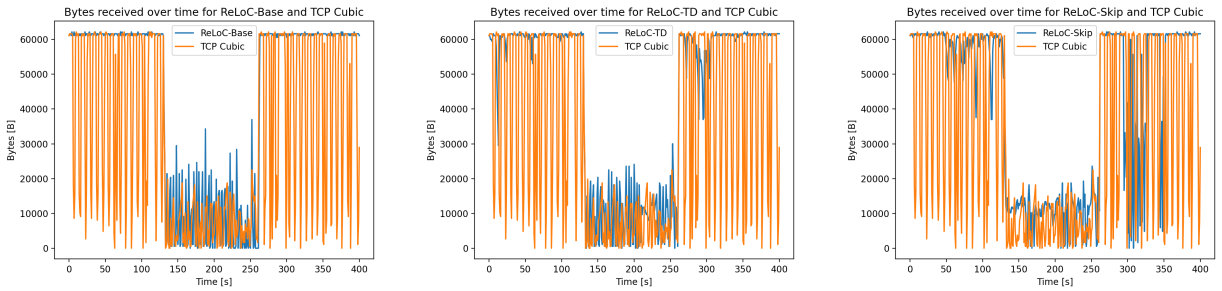
(a) ReLoC-base (b) ReLoC-TD (c) ReLoC-skip

(d) ReLoC-base (e) ReLoC-TD (f) ReLoC-skip

Figure B.11. Visualization of cumulative bytes sent and received throughout the test

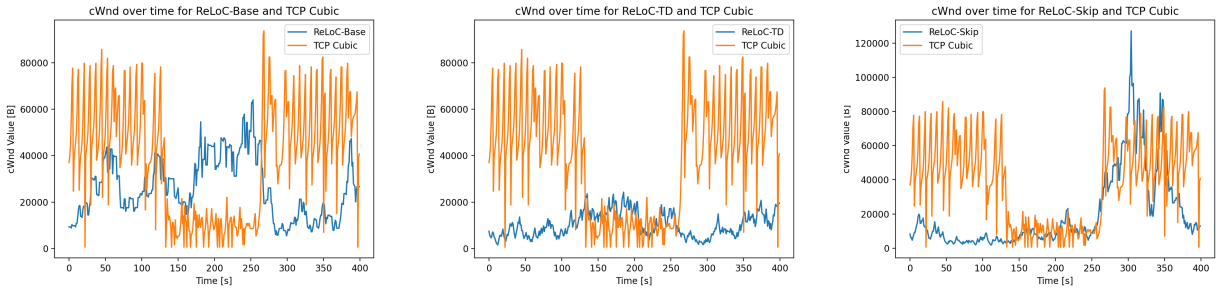


(a) ReLoC-base (b) ReLoC-TD (c) ReLoC-skip



(d) ReLoC-base (e) ReLoC-TD (f) ReLoC-skip

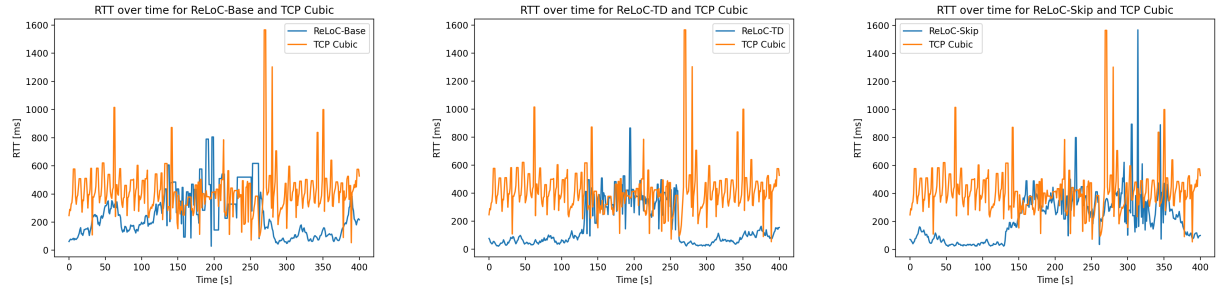
Figure B.12. Visualization of sent and received bytes for each time-step



(a) ReLoC-base (b) ReLoC-TD (c) ReLoC-skip

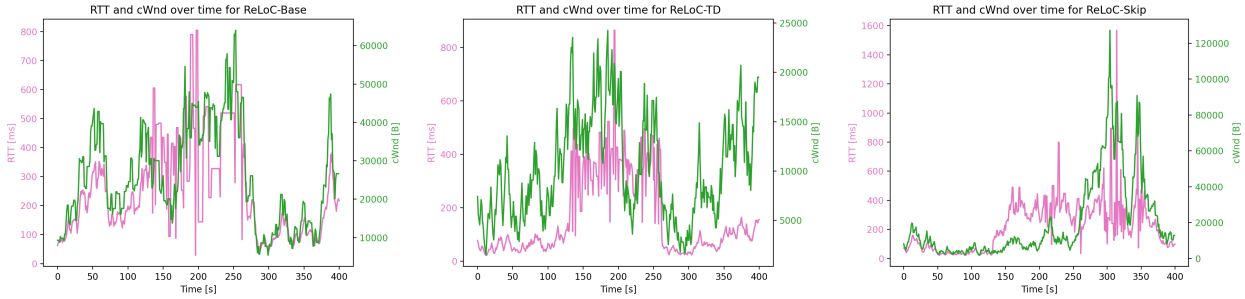
Figure B.13. Visualization of the cWnd value for each time-step

RTT results



(a) ReLoC-base (b) ReLoC-TD (c) ReLoC-skip

Figure B.14. Visualization of the average RTT in ms for each time-step



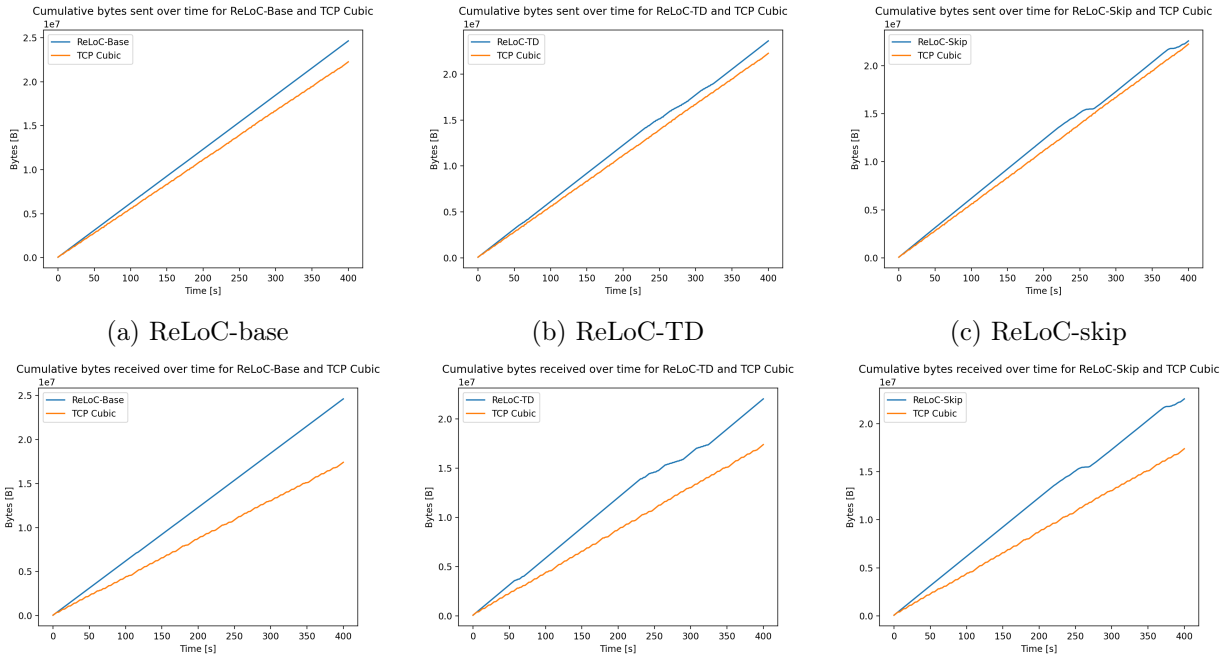
(a) ReLoC-base (b) ReLoC-TD (c) ReLoC-skip

Figure B.15. Visualization of the average RTT in ms and cWnd value plotted together for each time-step

B.2 Additional test results for the generalized ReLoC variants trained with another TCP connection on the channel

B.2.1 Test 5

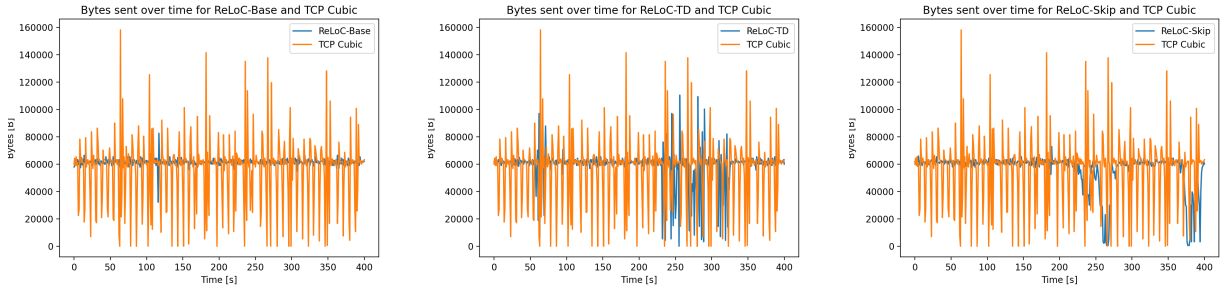
Throughput results



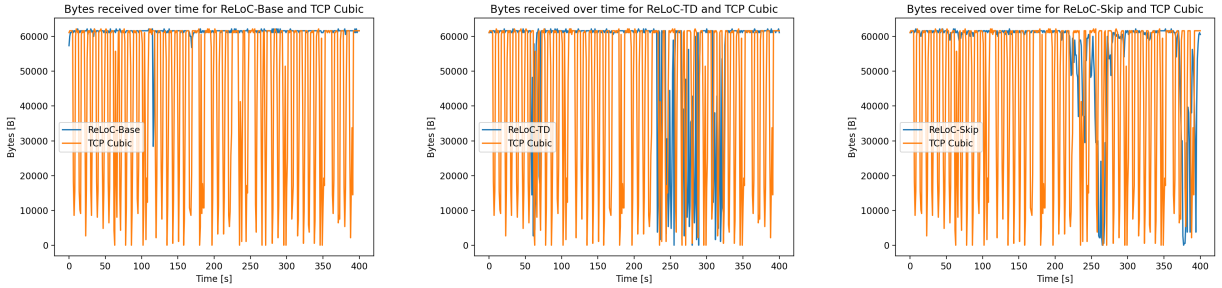
(a) ReLoC-base (b) ReLoC-TD (c) ReLoC-skip

(d) ReLoC-base (e) ReLoC-TD (f) ReLoC-skip

Figure B.16. Visualization of cumulative bytes sent and received throughout the test

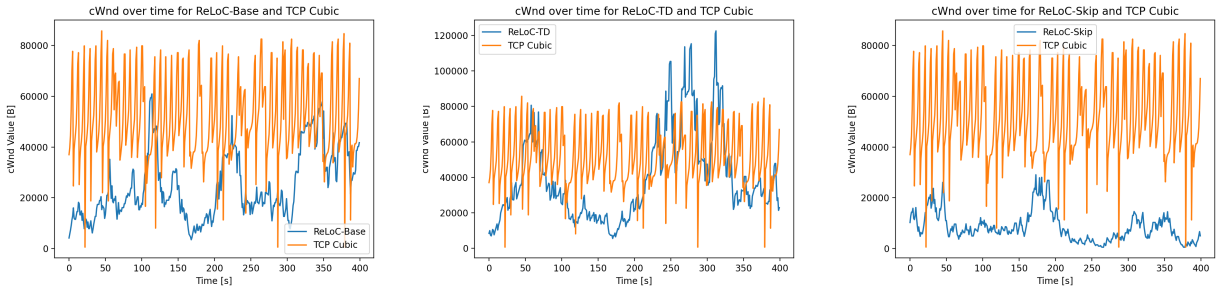


(a) ReLoC-base (b) ReLoC-TD (c) ReLoC-skip



(d) ReLoC-base (e) ReLoC-TD (f) ReLoC-skip

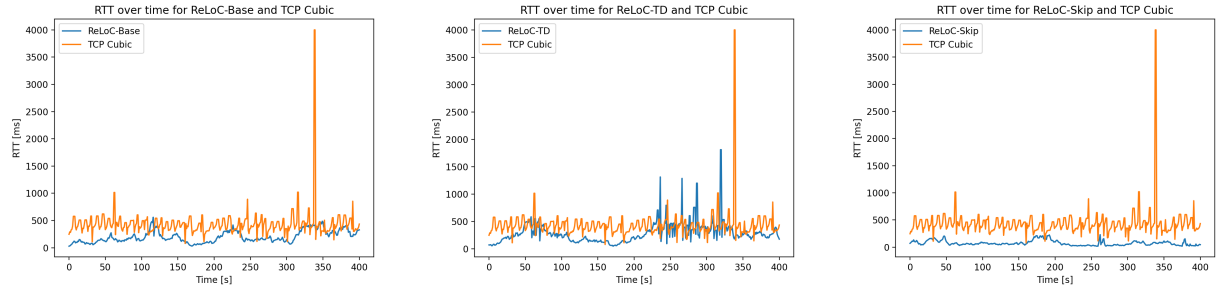
Figure B.17. Combined bytes sent



(a) ReLoC-base (b) ReLoC-TD (c) ReLoC-skip

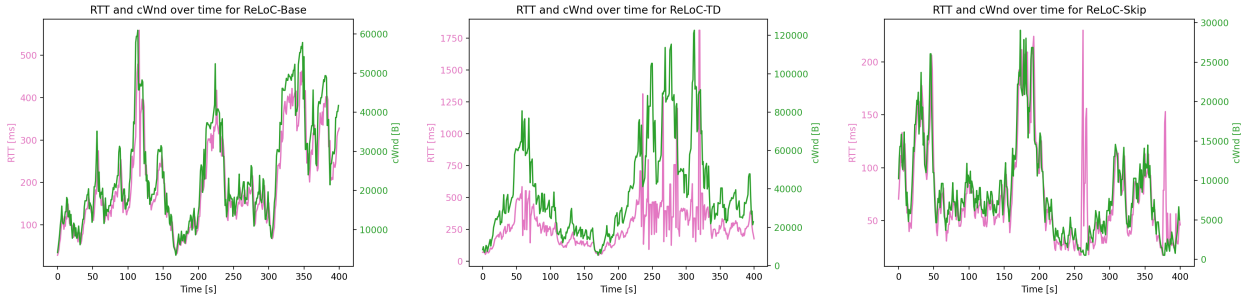
Figure B.18. Visualization of the cWnd value for each time-step

RTT results



(a) ReLoC-base (b) ReLoC-TD (c) ReLoC-skip

Figure B.19. Visualization of the average RTT in ms for each time-step

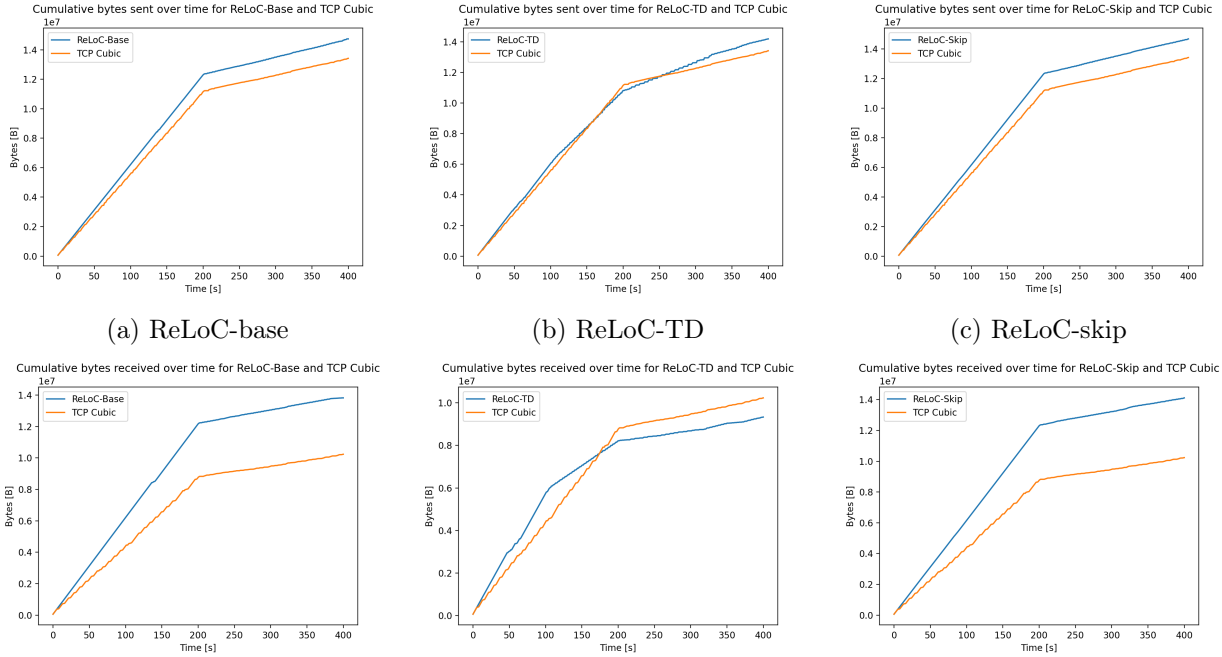


(a) ReLoC-base (b) ReLoC-TD (c) ReLoC-skip

Figure B.20. Visualization of the average RTT in ms and cWnd value plotted together for each time-step

B.2.2 Test 6

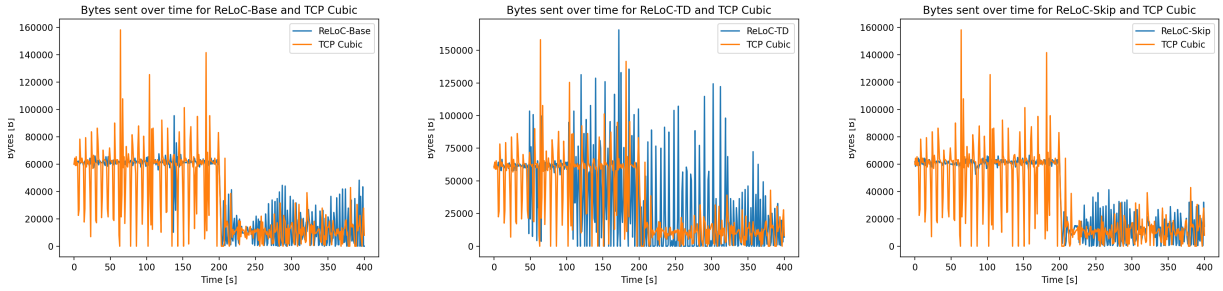
Throughput results



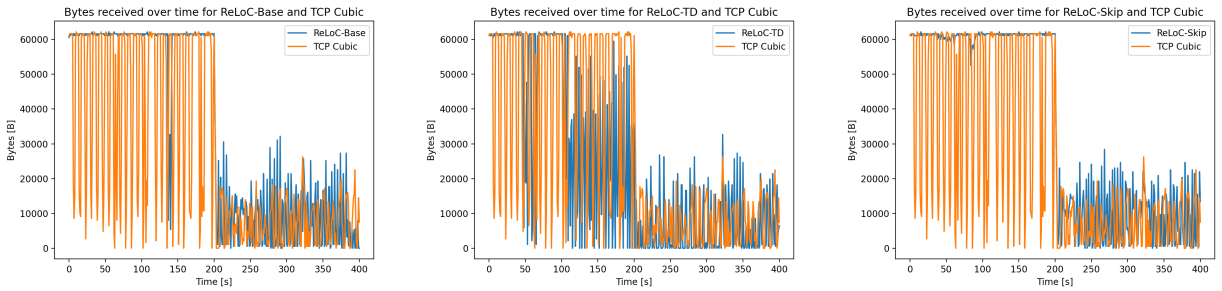
(a) ReLoC-base (b) ReLoC-TD (c) ReLoC-skip

(d) ReLoC-base (e) ReLoC-TD (f) ReLoC-skip

Figure B.21. Visualization of cumulative bytes sent and received throughout the test

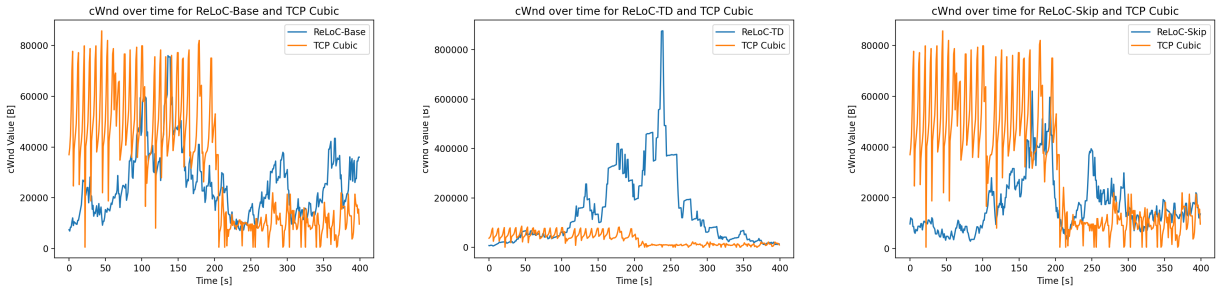


(a) ReLoC-base (b) ReLoC-TD (c) ReLoC-skip



(d) ReLoC-base (e) ReLoC-TD (f) ReLoC-skip

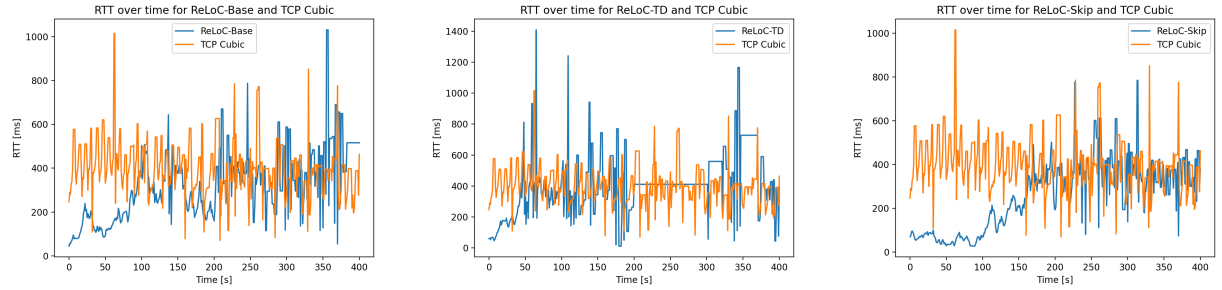
Figure B.22. Combined bytes sent



(a) ReLoC-base (b) ReLoC-TD (c) ReLoC-skip

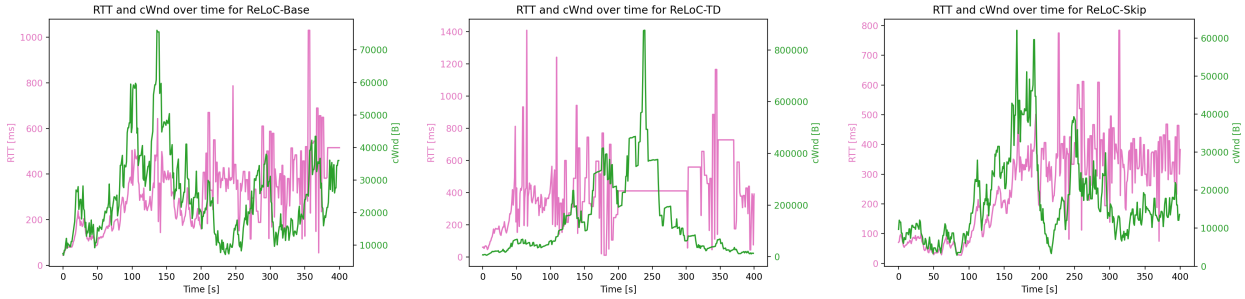
Figure B.23. Visualization of the cWnd value for each time-step

RTT results



(a) ReLoC-base (b) ReLoC-TD (c) ReLoC-skip

Figure B.24. Visualization of the average RTT in ms for each time-step

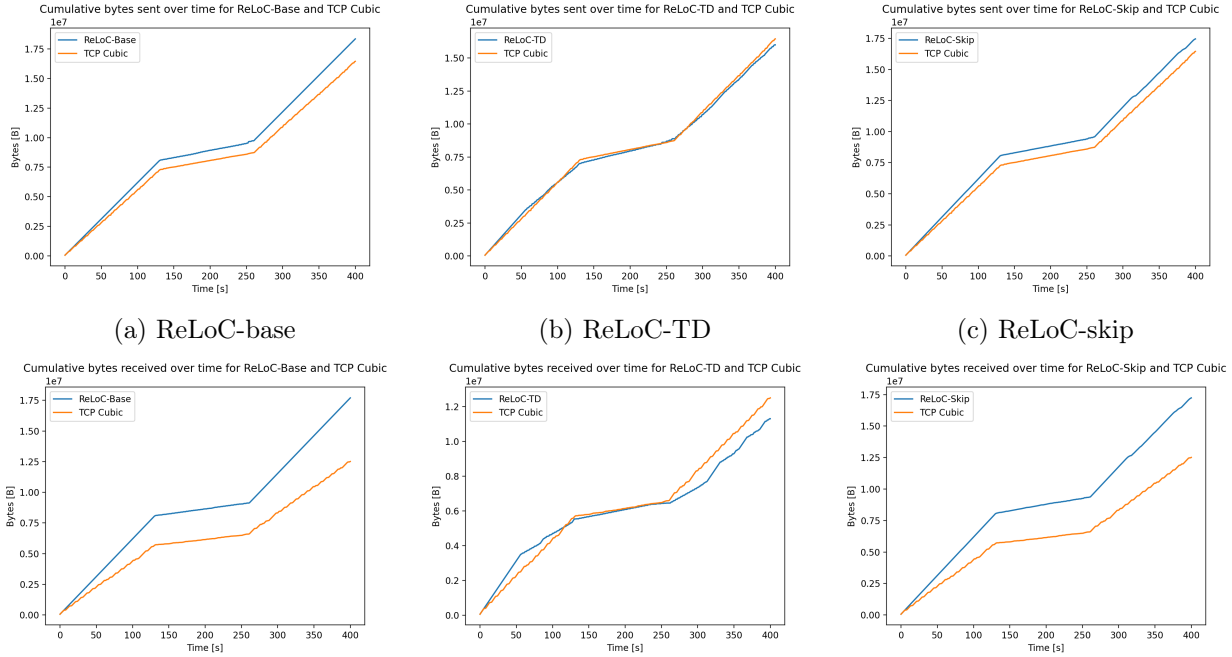


(a) ReLoC-base (b) ReLoC-TD (c) ReLoC-skip

Figure B.25. Visualization of the average RTT in ms and cWnd value plotted together for each time-step

B.2.3 Test 7

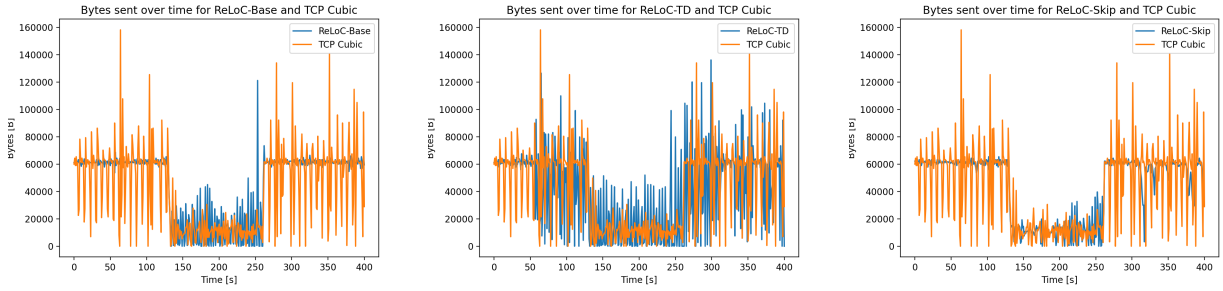
Throughput results



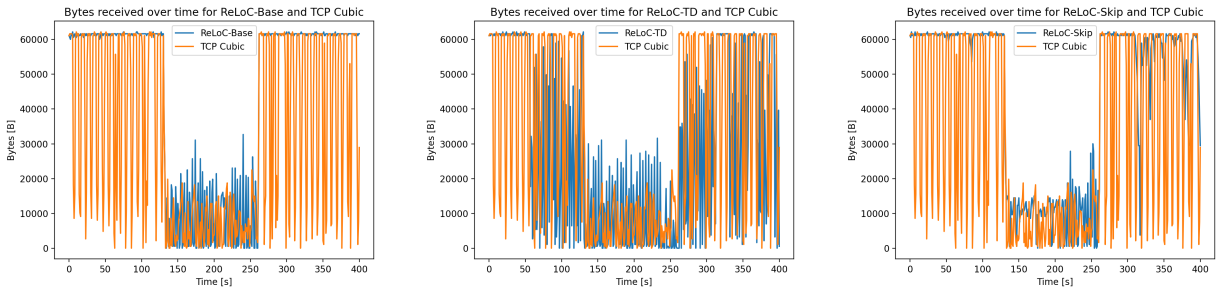
(a) ReLoC-base (b) ReLoC-TD (c) ReLoC-skip

(d) ReLoC-base (e) ReLoC-TD (f) ReLoC-skip

Figure B.26. Visualization of cumulative bytes sent and received throughout the test

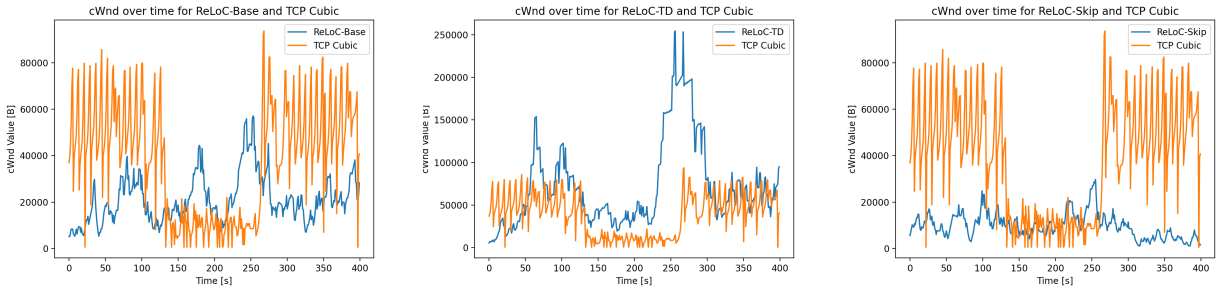


(a) ReLoC-base (b) ReLoC-TD (c) ReLoC-skip



(d) ReLoC-base (e) ReLoC-TD (f) ReLoC-skip

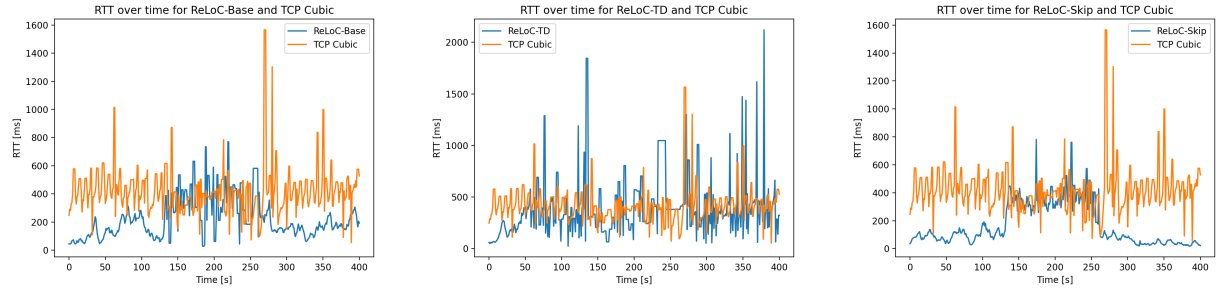
Figure B.27. Combined bytes sent



(a) ReLoC-base (b) ReLoC-TD (c) ReLoC-skip

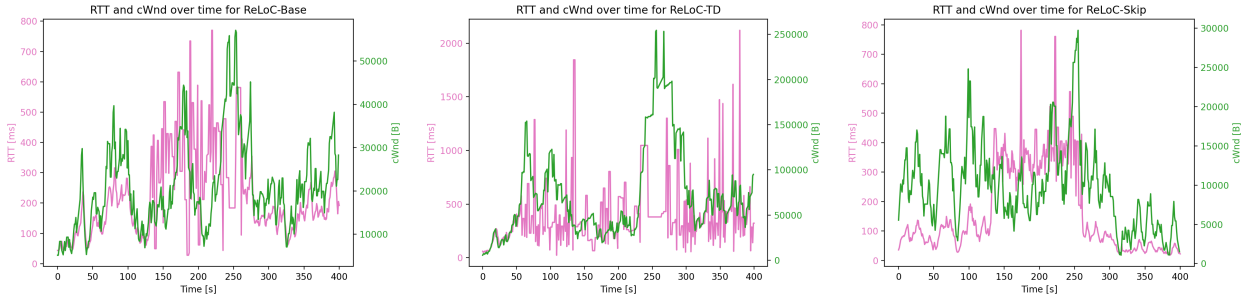
Figure B.28. Visualization of the cWnd value for each time-step

RTT results



(a) ReLoC-base (b) ReLoC-TD (c) ReLoC-skip

Figure B.29. Visualization of the average RTT in ms for each time-step



(a) ReLoC-base (b) ReLoC-TD (c) ReLoC-skip

Figure B.30. Visualization of the average RTT in ms and cWnd value plotted together for each time-step