Internet Congestion Control Research Group
Internet-Draft
Intended status: Experimental
Expires: 23 October 2025

J. Zheng
F. Liu
Y. Liu
Y. Lu
G. Chen
Nanjing University

21 April 2025

Pisces: Real-Time Video Transport Framework
draft-zheng-pisces-01

Abstract

This document specifies the Pisces, an ensemble video transport framework for real-time communication. Pisces complements the benefits of rule-based and learning-based approaches, without modifying the codec layer. Pisces uses an incremental and iterative reinforcement learning model to adapt to the unseen environment. When the real environment well matches the training environment, the learning-based approach is actively working. Otherwise, the rule-based approach is used to ensure transport safety. By simultaneously leveraging both rule-based logic and learning models to proactively probe network capacity, Pisces achieves high efficiency in both single-flow and cross-flow scenarios, while ensuring stable and fair cross-flow convergence. Pisces can be deployed in WebRTC, which replaces the default Google congestion control algorithm.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 23 October 2025.

Table of Contents

1. Introduction

Real-time video communication is dominating the Internet's traffic and has promoted the rapid growth of new businesses. However, technical challenges still remain to achieve consistent high QoE. On the one hand, the diverse and heterogeneous network environments, including cellular networks, WIFI, or optical fiber, make the bitrate inference harder at the sender side. On the other hand, the per-packet (millisecond) congestion control at the transport layer and the per-frame (second) rate control at the codec layer cannot perfectly cooperate with each other, leading to prolonged tail latency and further degrading QoE.

In order to achieve better QoE, two methods, rule-based and learning-based, are currently used. Although the rule-based bitrate adaptations have predictable behavior, they fail to achieve consistent high QoE in face of diverse and heterogeneous network environments. The emerging learning-based approaches using PPO [Sch17], DDPG [Lil15] or TD3 [Fuj18] can adapt such dynamics, while they perform poorly under the unseen network environment since a complete retraining procedure takes a long time.

This document describes the Pisces, an ensemble real-time video transport framework, where the rule-based and learning-based approaches run together in a cooperative manner, without modifying the codec layer. When the real environment deviates from the training environment, the rule-based approach is activated and the learning-based approach works as a backup. At the same time, our designed learning model performs incremental and iterative learning. Otherwise, the learning-based approach is activated to flexibly perform fine-grained bitrate adjustment.

The ensemble framework of Pisces is pluggable and complements the benefits of the rule-based and learning-based approach to provide consistent high QoE, while also ensuring **stable and cross-flow fair convergence. Pisces chooses modified GCC [draft-ietf-rmcat-gcc-02] as the rule-based counterpart, which can mitigate the performance degradation in presence of the unseen network environments. Besides, Pisces develop SPQL — an adaptive incremental and iterative learning model as the learning-based counterpart, which can quickly adapt to the unseen network environments with continuous state and action space. The core control logic of Pisces includes the queue draining

stage, exploration stage, evaluation stage, and exploitation stage, where they interact with each other to determine the final bitrate on the fly.

2. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT",

"SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

3. Design Overview

3.1. Adaptive Bitrate Framework Overview

Figure1 shows the main structure of an adaptive bitrate framework named Pisces . Pisces mainly includes an ensemble bitrate adaptation module and an incremental and iterative learning module. The bitrate adaptation leverages the wisdom of the rule-based and learning-based approaches, running both control logic to determine the final bitrate based on the collected network feedback. The learning module works in an incremental and iterative manner, providing consistent high performance by only relearning a part of the model.

3.1.1. Bitrate Adaptation Module

The monitor module collects the network information in the latest monitoring interval. The raw data can be transformed to historical states and rewards, feeding to the RL agent. The RTP/RTCP packets, including throughput and delay, are directly sent to the rule-based (RL-based) counterpart. Subsequently, the RL agent derives the historical state from the monitor module and generates one candidate bitrate x_rl. The transitions generated by the RL agent will be stored in the replay buffer of the learning module. The rule-based counterpart runs its fixed control logic to produce the other candidate bitrate x_cl according to the RTCP packet. Both x_rl and x_cl are calculated by the utility module and then evaluated by their corresponding utility values.

3.1.2. Incremental and Iterative Learning Module

The subspace partitioning module adaptively partitions the original state space into two subspaces and each subspace can be further partitioned. When the environment changes, the function refitting module can just refit the Q-value functions of the changed subspaces belonging to a variation state set and update the model to make RL Agent synchronize with the learning module, not involving the state in the unchanged subspaces. Partitioning a state space into subspaces allows Pisces to use a simple and practical fitting function and meanwhile guarantees the generalization ability.

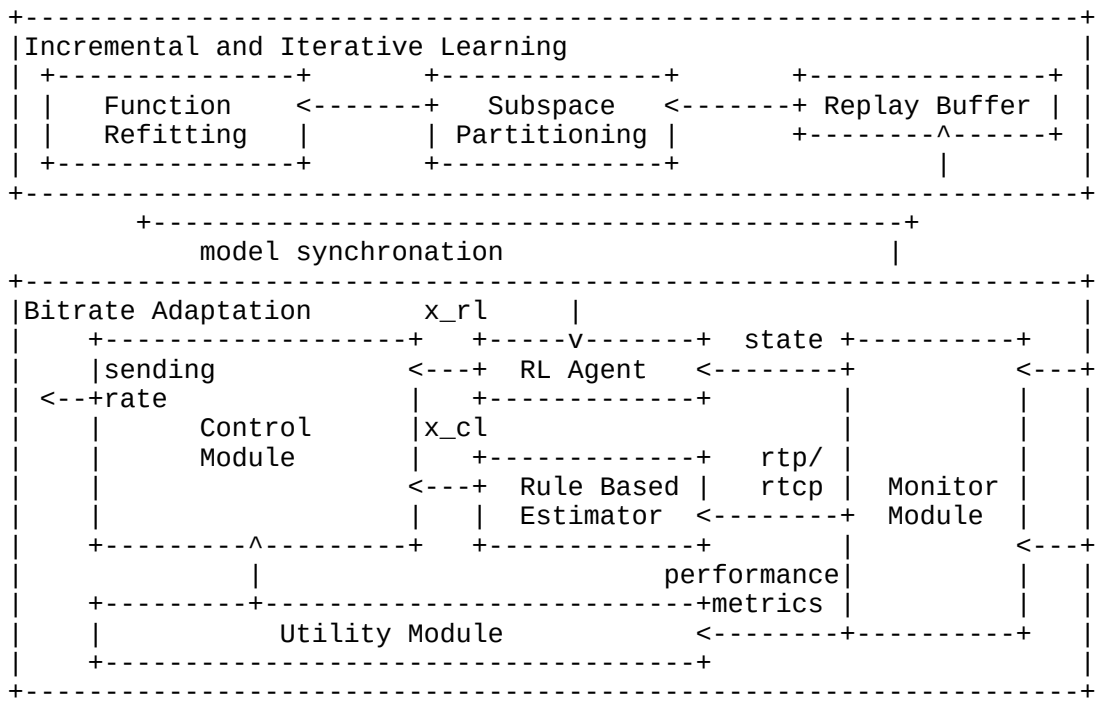

Figure 1: Main Structure of Pisces

3.2. State Machine Overview

The control module determines the switching logic among different stages and produces the running rate of each stage. The Figure 2 shows the overall control logic consists of four stages: queue draining, exploration, evaluation and exploitation.

At the start of each stage, Pisces estimate the congestion degree based on the measured current delay and the latest minimum delay at the sender side. Once the estimated delay exceeds a certain threshold and the current sending rate is higher than the receiving rate, indicating that the queue is continually building up, it enters the Drain Queue Stage and decreases the bitrate. The duration is one RTT.

Once the drain queue stage ends, it enters the exploration stage. The Exploration Stage takes an RTT for both rule-based counterpart and the RL agent to generate the candidate bitrates, based on the network state collected during this phase. If the difference between these two candidate bitrates is divergent enough, indicating that a disagreement happens, Pisces enters into the evaluation stage and judge which one is right; Otherwise, Pisces still keeps in the exploration stage.

In the Evaluation Stage, one RTT is divided into two evaluation intervals (EI), one for each candidate bitrate. To minimize the interference during the evaluation, Pisces try the smaller candidate bitrate in the first EI and the larger one in the second EI. This could avoid the side effect to some degree as a result of the queue accumulation if tried the larger one firstly. When the evaluation ends, it enters the exploitation stage.

The Exploitation Stage temporally takes the bitrate determined in the last control cycle as the sending rate. The main purpose of this stage is to quantify the performance of the two candidate bitrates obtained in the evaluation stage since the network feedback should have been received by the sender side at this moment. The collected feedback are fed into the utility function to calculate the corresponding utility values. The bitrate with higher utility value is preferred and will be set to the final sending rate. Finally, Pisces enters the next control cycle and starts with the exploration stage again. Note that the previous action -1, the last state , the reward and current state together form a sample and stored into the replay buffer used for the incremental and iterative learning. The detailed behavior for each state is described below.

4. Detailed Algorithm

4.1. State Machine

Pisces implements a state machine which keeps track of performance of both rule-based CC agent and learning-based CC agent. Utilization is then calculated and decisions are taken based on states.

4.1.1. State Transition Diagram

The following state transition diagram Figure2 determines the switching logic among different stages:

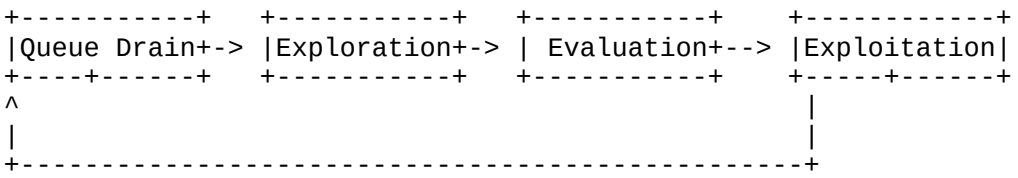

Figure 2: State Machine of Pisces

4.1.2. State Machine Operation Overview

When starting up, Pisces tries to ramp up sending quickly; to utilize the best of rule-based CC and learning-based CC, Pisces must continue to monitor the divergence of bitrate the two algorithm proposed, and then evaluate which one better suits current link capacity. If the rule-based agent performs better, Pisces must use the bitrate it given to guide later transmission, otherwise the bitrate learning-based agent given is picked to improve transmission quality. Pisces runs these measurements periodically to ensure best utilization among rule-based agent and learning-based agent. The frequency, duration of measurements differ depending on current link status. This state machine has several goals:

1. Achieve better utilization by learning from link states

2. Avoid security problem of learning-based methods by running rule-based model in parallel

3. Continous learning by sampling from link states and actions to keep model effective

4.1.3. State Machine Tactics

In Pisces, at any given time the sender can choose one of the following tactics:

1. Using rule-based model: When the rule-based method achieves a higher utility value during the evaluation phase, use this method to obtain improved performance.

2. Using learning-based model: When the learning-based method achieves a higher utility value during the evaluation phase, use this method to obtain improved performance.

3. Stay with previous selected bitrate: If there are no new comparative results between the rule-based and learning-based methods, continue using the previous bitrate.

4.2. Algorithm Organization

The Pisces algorithm is driven by both event and time. State transitions happen upon transport connection initialization, when a processInterval event is triggered, and when an event for receiving feedback messages is triggered. All of the sub-steps invoked referenced below are described below.

4.2.1. Initialization

Upon transport connection initialization, Pisces executes its initialization steps:

```
class Pisces:
  def __init__(self):
    self.state = PiscesState.START_UP
    self.feedback_processor = FeedbackProcessor()
    self.rl_agent = RLAgent()
    self.rule_agent = RuleAgent()
    self.last_bitrate = DEFAULT_START_BITRATE
    self.last_rtt = DEFAULT_START_RTT
    self.last_loss = 0
    self.last_receiving_rate = DEFAULT_START_BITRATE
    self.last_checkpoint = timeNow()
    self.smoothed_rtt = DEFAULT_START_RTT
    self.first_bitrate = DEFAULT_START_RTT
    self.second_bitrate = DEFAULT_START_RTT
    self.first_agent_utility = 0
    self.second_agent_utility = 0
    self.last_utility = 0
    self.feedbacks_received = []
```

The most important part of Pisces is rl_agent and rule_agent, which are responsible for making decisions based on predefined rules or learning models. FeedbackProcessor is meant to conduct link state information from TWCC feedbacks.

4.2.2. Per-ProcessInterval Steps

When processInterval event triggered, Pisces take further actions based on current state.

```
class Pisces:
  def onProcessInterval(self):
    # ...
```

4.2.3. Per-Feedback Steps

When the event of receiving feedback messages is triggered, Pisces does not process them immediately. Instead, it stores them in a buffer and processes them together later to match the RTT-based control cycle.

```
class Pisces:
    def onFeedbackMsg(self, msg):
        self.feedbacks_received.append(msg)
```

4.3. State Machine Operation

4.3.1. Startup Stage

When a Pisces flow starts up, it performs a exponential bitrate ramp up process to search the potential link capacity to later guide the learning-based model. This is done by double the bitrate each RTT until an observable delay rise happend. Pisces always start with Startup stage.

When initializing a connection, Pisces will set its state to START_UP. As Pisces exploring available bandwidth, it updates the last and max receiving rate achieved upon feedbacks. During the stage, Pisces keep track of receiving rate, current RTT and minimum RTT observed. When the sending rate goes beyond link capacity, a queue forms in the bottleneck link and then RTT rise will be observed. Upon the RTT signal, Pisces is able to probe current link capacity and exit START_UP stage. To mitigate the buffer filled by START_UP stage, Pisces enters DRAIN stage to prevent further congestion events.

```
class Pisces:
  # ...
  def onProcessInterval(self):
    if self.state == PiscesState.START_UP:
      if timeNow() < self.last_checkpoint + self.smoothed_rtt:
        return self.last_bitrate
      self.processFeedback()
      if self.last_rtt > self.min_rtt * DRAIN_ENTER_THRESHOLD_RTT:
        return self.enterDrain()
      self.last_checkpoint = timeNow()
      self.last_bitrate *= PROBE_BITRATE_GAIN
      return self.last_bitrate
    # ...
```

4.3.2. Drain Stage

Upon exiting Startup or a significant RTT grow happened on exploration stage, Pisces enters its Drain state. In Drain, Pisces aims to quickly reduce the queue length built during transmission by reducing current sending rate on each bitrate update event. The reduced bitrate is then sync to two agents.

```
class Pisces:
  # ...
  def enterDrain(self):
    self.state = PiscesState.DRAIN
    self.last_checkpoint = timeNow()
    self.last_bitrate *= DRAIN_BITRATE_GAIN
    self.rule_agent.setBitrate(self.last_bitrate)
    self.rl_agent.setBitrate(self.last_bitrate)
    return self.last_bitrate
```

Drain stage lasts one RTT, then Pisces exits Drain and enters Exploration.

```
class Pisces:
  # ...
  def onProcessInterval(self):
    # ...
    elif self.state == PiscesState.DRAIN:
      if timeNow() < self.last_checkpoint + self.smoothed_rtt:
        return self.last_bitrate
      self.processFeedback()
      return self.enterExploration()
    # ...
```

4.3.3. Exploration Stage

Pisces aims to select best performing algorithm with current network states. To achieve this, Pisces needs to run rule-based congestion control agent and learning-based counterpart simultaneously then collect feedbacks to determine which one is better performing. This stage takes one RTT to finish. During this stage, Pisces takes the bitrate chosen from last control loop to reset both agent. Then the bitrate given by rule-based agent is picked as final bitrate. Learning-based agent still monitors network feedbacks but only provide bitrate as a reference. The exploration stage ends once there is a gap larger than threshold between the bitrate given by rule based agent and learning based agent.

```
class Pisces:
  # ...
    def enterExploration(self):
    self.last_checkpoint = timeNow()
    self.rule_agent.setBitrate(self.last_bitrate)
    self.rl_agent.setBitrate(self.last_bitrate)
    return self.last_bitrate
```

```
class Pisces:
  # ...
  def onProcessInterval(self):
    # ...
    elif self.state == PiscesState.EXPLORATION:
      if timeNow() < self.last_checkpoint + self.smoothed_rtt:
        return self.last_bitrate
      self.last_checkpoint = timeNow()
      self.processFeedback()
      if (
        abs(self.rule_agent.getBitrate() - self.rl_agent.getBitrate())
        < BITRATE_VARIANCE_THRESHOLD * self.last_bitrate
      ):
        self.last_bitrate = self.rule_agent.getBitrate()
        return self.last_bitrate
      return self.enterEvaluationFirst()
    # ...
```

4.3.4. Evaluation Stage

To evaluate the performance of rule-based algorithm and learning-based algorithm, Pisces takes one RTT to try both bitrate given and then picks the better performing one. Evaluation stage is divided into two parts, one for each bitrate. Note that Pisces always starts evaluation with the smaller bitrate given to prevent potential congestion harmness.

```
class Pisces:
  # ...
  def enterEvaluationFirst(self):
    self.last_checkpoint = timeNow()
    self.state = PiscesState.EVALUATION_FIRST
    x_cl = self.rule_agent.getBitrate()
    x_rl = self.rl_agent.getBitrate()
    self.first_bitrate = min(x_cl, x_rl)
    self.second_bitrate = max(x_cl, x_rl)
    return self.first_bitrate

  def enterEvaluationSecond(self):
    self.last_checkpoint = timeNow()
    self.state = PiscesState.EVALUATION_SECOND
    return self.second_bitrate

class Pisces:
  # ...
  def onProcessInterval(self):
    # ...
    elif self.state == PiscesState.EVALUATION_FIRST:
      if timeNow() < self.last_checkpoint + self.smoothed_rtt / 2:
        return self.first_bitrate
      self.processFeedback()
      return self.enterExploitationSecond()
    elif self.state == PiscesState.EVALUATION_SECOND:
      if timeNow() < self.last_checkpoint + self.smoothed_rtt / 2:
        return self.second_bitrate
      self.processFeedback()
      return self.enterExploitationFirst()
    # ...
```

4.3.5. Exploitation Stage

As results of bitrate selection, feedbacks are expected to return one RTT later, during the exploitation stage. As Pisces chose two bitrate in the earlier half RTT and the latter half RTT separately, their feedbacks are returning in the earlier half RTT and the latter half RTT of Evaluation stage respectively. Through calculation of Utility function and several network-layer metrics collected from feedbacks, Pisces is able to measure the performance of rule-based agent and learning-based agent and pick the one behave better. After Pisces has done bitrate selecting, it will enter Exploration stage again.

```
class Pisces:
  # ...
  def enterExploitationFirst(self):
    self.last_checkpoint = timeNow()
    self.state = PiscesState.EXPLOITATION_FIRST
    return self.last_bitrate
  def enterExploitationSecond(self):
    self.last_checkpoint = timeNow()
    self.state = PiscesState.EXPLOITATION_SECOND
    return self.last_bitrate

class Pisces:
# ...
def onProcessInterval(self):
  # ...
  elif self.state == PiscesState.EXPLOITATION_FIRST:
    if timeNow() < self.last_checkpoint + self.smoothed_rtt / 2:
      return self.last_bitrate
    self.processFeedback()
    self.first_agent_utility = self.getUtility()
    return self.enterExploitationSecond()
  elif self.state == PiscesState.EXPLOITATION_SECOND:
    if timeNow() < self.last_checkpoint + self.smoothed_rtt / 2:
      return self.last_bitrate
    self.processFeedback()
    self.second_agent_utility = self.getUtility()
    if (
      max(self.first_agent_utility, self.second_agent_utility)
      > self.last_utility
    ):
      if self.first_agent_utility > self.second_agent_utility:
        self.last_bitrate = self.first_bitrate
      else:
        self.last_bitrate = self.second_bitrate
    return self.enterExploration()
  # ...
```

4.4. Feedback Process

Pisces uses transport-wide congestion control feedbacks to conduct link metrics. The specification of packet formats can be found in [draft-holmer-rmcat-transport-wide-cc-extensions-01]. RTT, receiving rate and loss calculation is based on google congestion control, which can be found in [draft-ietf-rmcat-gcc-02] in 5.Delay based control.

4.5. Utility Calculation

Pisces uses Utility function to evaluate performance of bitrate selection. The utility function can bound the bandwidth of a video flow into the range [MIN_BAND, MAX_BAND], which can meet the requirements of most industry scenarios. At the same time, it can speedup the training procedure. The utility function fully considers the impact of relative and absolute throughput, packet loss rate, delay, and delay jitter.

```
class Pisces:
# ...
def getUtility():
  delay_metric = self.min_rtt / (self.last_rtt * 2)
  return (
    self.last_receiving_rate - 10 * self.last_loss * self.last_receiving_rate
  ) / self.max_bw * delay_metric - 2 * delay_metric
```

The utility function integrates key performance indicators including throughput, packet loss rate, and latency. The delay_metric captures how far the current RTT deviates from the minimum observed RTT, serving as a dynamic penalty factor. The throughput component incorporates a loss penalty term to discourage high sending rates in high-loss conditions. The utility is normalized by max_bw to ensure comparability across flows, and the final term imposes an additional penalty for high delay, emphasizing latency sensitivity. The structure of this utility function is concave with respect to key variables such as receiving rate, loss rate, and delay, ensuring that the optimization process is stable and converges to a unique maximum. More importantly, when multiple concurrent flows adopt the same utility function, the aggregate utility forms a socially concave objective. According to the Social Concavity Theorem, this guarantees convergence to a unique Pareto-optimal point, where no individual flow's utility can be improved without degrading others. Therefore, the utility function of Pisces not only ensures desirable QoE by balancing throughput, loss, and latency, but also provides strong theoretical guarantees of cross-flow fairness. This enables distributed flows to reach a fair and efficient bandwidth allocation without centralized coordination, making Pisces particularly suitable for real-time video delivery in large-scale, multi-user, and heterogeneous edge environments.

4.6. Updating Control Parameters

Most of the algorithm deployed to process network signals are from the delay based estimator of GCC XX, including RTT, loss rate and receiving rate estimate. These are done in FeedbackProcessor. Feedbacks provided by WebRTC Transport-Wide CC extension are stored in Pisces.feedbacks_received to provide RTT-based network signal processing.

```
class Pisces:
  # ...
  def processFeedback(self):
    self.feedback_processor.process(self.feedbacks_received)
  # ...
```

After FeedbackProcessor has done processing network signals, callbacks of Pisces, rule-based agent and learning-based agent will be called. These metrics are used in later utility calculation and bitrate selection.

```
class Pisces:
  # ...
  def onFeedback(self, stats):
    self.last_rtt = stats.rtt
    self.last_loss = stats.loss
    self.last_receiving_rate = stats.receiving_rate
    self.smoothed_rtt = WEIGHT_RTT * self.smoothed_rtt + stats.rtt * (1 - WEIGHT_RTT)
    self.min_rtt = WEIGHT_MINRTT * self.min_rtt + stats.min_rtt * (1 - WEIGHT_MINRTT)
    self.max_bw = WEIGHT_MAXBW * self.max_bw + stats.max_bw * (1-WEIGHT_MAXBW)
    # ...
```

The max_bw and min_rtt are exceptions because we need to the two metrics stay stable but keep up with link state. Pisces uses an EWMA (exponential weighted moving average) filter when calculating smoothed_rtt, min_rtt and max_bw. The hyperparameter of EWMA can be fine-tuned to tradeoff sensitivity and robustness.

4.6.1. Updating Rule-based Agent

The rule-based agent is a modified version of GCC. The feedback processing part of GCC is extracted to provide network signals to learning-based agent as well (Feedback Processor). Delay-based bandwidth estimator and loss-based estimator are kept in rule-based agent and the decision procedure remain the same.

```
   class Pisces:
     # ...
     def onFeedback(self, stats):
       # ...
       self.rule_agent.onStats(stats)
```

4.6.2. Updating Learning-based Agent

Upon selection, the learning-based agent in Pisces selects a reinforcement learning-based model to produce a bitrate candidate. The model is built on PPO (Proximal Policy Optimization), containing features of loss rate, trendline, bitrate metric, delay metric and network state.

Among these features, loss rate, trendline and network state are obtianed directly from stats provided by FeedbackProcessor. Bitrate metric and delay metric need further calculations.

```
class Pisces:
  # ...
  def onFeedback(self, stats):
    # ...
    features = [self.loss, stats.trendline, self.last_bitrate/self.max_bw, self.min_rtt/s
elf.rtt, stat.network_state]
    self.rl_agent.onFeature(features)
class RLAgent:
  # ...
  def onFeature(self, features):
    self.history.append(features)
```

5. Implementation Status

5.1. Implementation on WebRTC

Pisces is designed to be compatible with WebRTC transport-wide congestion control [draft-holmer-rmcat-transport-wide-cc-extensions-01]. Like GCC [draft-ietf-rmcat-gcc-02], it takes Transport-wide CC feedbacks (e.g., RTT, loss rate) to estimate network capacity per process intervals, and then the results is sent to the codec as the target bitrate. Pisces can be implemented as an insertable congestion controller, which is injected into peer connection when creating peer connection factory. The whole process requires no modifications to the library code.

6. Security Considerations

This proposal makes no changes to the underlying security of transport protocols or congestion control algorithms. BBR shares the same security considerations as the existing standard congestion control algorithm [RFC5681].

7. IANA Considerations

This document has no IANA actions. Here we are using that phrase, suggested by [RFC5226], because Pisces does not modify or extend the wire format of any network protocol, nor does it add new dependencies on assigned numbers. BBR involves only a change to the congestion control algorithm of a transport sender, and does not involve changes in the network, the receiver, or any network protocol.

8. References

8.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", RFC 5226, DOI 10.17487/RFC5226, May 2008, <https://www.rfc-editor.org/info/rfc5226>.

[RFC5681] Allman, M., Paxson, V., and E. Blanton, "TCP Congestion Control", RFC 5681, DOI 10.17487/RFC5681, September 2009, <https://www.rfc-editor.org/info/rfc5681>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

8.2. Informative References

[Sch17] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and O. Klimov, "Proximal Policy Optimization Algorithms", July 2017, <https://arxiv.org/abs/1707.06347>.

[Lil15] Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., and D. Wierstra, "Continuous Control with Deep Reinforcement Learning", 2015, <https://arxiv.org/abs/1509.02971>.

[Fuj18] Fujimoto, S., van Hoof, H., and D. Meger, "Addressing Function Approximation Error in Actor-Critic Methods", 2018.

[draft-holmer-rmcat-transport-wide-cc-extensions-01] Holmer, S., Flodman, M., and E. Sprang, "RTP Extensions for Transport-wide Congestion Control", 19 October 2015, <https://datatracker.ietf.org/doc/html/draft-holmer-rmcat-transport-wide-cc-extensions-01>.

[draft-ietf-rmcat-gcc-02] Holmer, S., Lundin, H., Carlucci, G., De Cicco, L., and S. Mascolo, "A Google Congestion Control Algorithm for Real-Time Communication", 8 July 2016, <https://datatracker.ietf.org/doc/html/draft-ietf-rmcat-gcc-02>.

Appendix A. Change log

A.1. Version -00 to -01

* Added change log

* Made it clearer that we rewrite "Implementation on WebRTC".

* Added in Section 4.5, the utility function ensures cross-flow fairness by formally modeling the trade-offs among throughput, loss, and delay. The section provides a detailed discussion on how this concave utility design enables distributed homogeneous flows to achieve stable and fair bandwidth allocation without centralized coordination.

Authors' Addresses

Jiaqi Zheng
Nanjing University
Email: jzheng@nju.edu.cn

Feida Liu
Nanjing University
Email: 602023330020@smail.nju.edu.cn

Yu Liu
Nanjing University
Email: yliu@smail.nju.edu.cn

Yi Lu
Nanjing University
Email: Xavier4m@outlook.com

Guihai Chen
Nanjing University
Email: gchen@nju.edu.cn