VIETNAM NATIONAL UNIVERSITY, HANOI

UNIVERSITY OF ENGINEERING AND TECHNOLOGY

Hoang Thanh Tam

AN IMPLEMENTATION OF RECEIVER-SIDE REAL-TIME CONGESTION CONTROL FOR SIMULATIONS USING SIMPY

Major: Computer Science

HA NOI - 2017

VIETNAM NATIONAL UNIVERSITY, HANOI

UNIVERSITY OF ENGINEERING AND TECHNOLOGY

Hoang Thanh Tam

AN IMPLEMENTATION OF RECEIVER-SIDE REAL-TIME CONGESTION CONTROL FOR SIMULATIONS USING SIMPY

Major: Computer Science

Supervisor: Dr. Hoang Xuan Tung

HA NOI - 2017

AUTHORSHIP

"I hereby declare that the work contained in this thesis is of my own and has not been previously submitted for a degree or diploma at this or any other higher education institution. To the best of my knowledge and belief, the thesis contains no materials previously published or written by another person except where due reference or acknowledgement is made."

Signature:……

SUPERVISOR'S APPROVAL

"I hereby approve that the thesis in its current form is ready for committee examination as a requirement for the Bachelor of Computer Science degree at the University of Engineering and Technology."

Signature:……

ACKNOWLEDGEMENT

I would like to express my sincere gratitude to Doctor Hoang Xuan Tung to instruct me to complete this thesis.

ABSTRACT

In today's technology era, real-time communication has a significant meaning in our daily life [1]. Particularly, the Internet has enabled people and information systems to communicate more rapidly across greater distances than ever before, and the Internet of Things (IoT) is expanding communications not only between computers but also across objects in our physical environments. As a result, nowadays any device can be part of real-time and peer-to-peer communications.

With the popularity of browsers nowadays [2], enabling real-time communications capability natively in browsers is becoming more important than ever before. WebRTC is developed to meet that need. Supported by Google, Mozilla and Opera, and others, WebRTC is free, open source but plays an essential role in bringing Real-Time Communications (RTC) capabilities into browsers and mobile applications with via simple APIs. The mission of WebRTC is to enable rich, high-quality RTC applications to be developed for the browser, mobile platforms, and IoT devices, and allow them all to communicate via a common set of protocols, as claim in [3].

One of the most important parts in WebRTC is congestion control algorithm. Real-time communication has to face so many challenges in congestion control and the development team has deployed a method named Receiver-side Real-time Congestion Control [3] (RRTCC) to handle this problem. This thesis is an implementation of this algorithm for simulation using SimPy [4] – a process-based discrete-event simulation framework based on standard Python.

TABLE OF CONTENTS

List of Figures

List of Tables

ABBREVATIONS

API	Application Program Interface
IoT	Internet of Things
TCP	Transmission Control Protocol
RRTCC	Receiver-side Real-time Congestion Control
RTC	Real-Time Communications
RTCP	Real-Time Control Protocol
RTP	Real-time Transport Protocol
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
WebRTC	Web Real-Time Communication

Chapter 1

INTRODUCTION

1.1. Motivation

As defined in [5], real-time communication is the mode of telecommunications in which all users can exchange information instantly or with negligible latency.

Real-time communication is not just the passing of the information; it is one-to-one or one-to-many or many-to-many communication. It is the instant transmission of the message where the source and the destination both are present at the same time.

Businesses need to understand the urgency of improving the real-time communication and adapt to it. This is not only going to increase the productivity but also save time and resources. The real-time communication is becoming sophisticated with the coming of the modern techniques, available for the businesses to make better and quicker decisions.

Web Real-Time Communication (WebRTC) is a new standard that lets browsers communicate in real time using a peer-to-peer architecture. It is about secure, consent-based, audio/video (and data) peer-to-peer communication between HTML5 browsers, according to [6].

Latency is the very challenging problem we need to deal with when we want to have great performance real-time applications. Congestion control contributes an undeniable part of handling latency solution. In WebRTC, there is an algorithm called Receiver-side Real-time Congestion Control (RRTCC [3]) which is developed to deal with congestion control problem.

1.2. Contributions and thesis overview

The purpose of this thesis is to propose an implementation of RRTCC for simulations using SimPy. Contribution of this thesis is twofold. Firstly, we present how to use SimPy to simulate real-time communication systems that use RRTCC for multimedia data transmission. Secondly, we use our simulations to show the benefits of RRTCC for RTP multimedia sessions in real-time transmission comparing with no congestion control algorithm.

The rest of this thesis is organized as follows.

Chapter 2 provides theoretical background about RRTCC algorithm, a simulation framework named SimPy, and other libraries like NumPy, PyKalman. Besides is an implementation of RRTCC that are performed by other researchers. In chapter 3 the simulation we have implemented will be discussed in detailed. Chapter 4 is the evaluations of our RRTCC implementation presented in previous chapter, which is the comparison between using RRTCC and using no congestion control algorithm. The final chapter is the conclusion and the future improvements for our simulation.

Chapter 2

RELATED WORK

We dedicate this chapter for theoretical backgrounds and related work. Particularly, we present a brief introduction about Real-time Transport Protocol, Real-Time Control Protocol, WebRTC and Receiver-side Real-time Congestion Control algorithm. We also briefly mention existing implementations of RRTCC that are performed by other researchers. And finally, we introduce softwares and tools that are used to accomplish this project, especially SimPy and its Python based ecosystem.

2.1. Real-time Transport Protocol (RTP) and Real-Time Control Protocol (RTCP)

According to [7], RTP provides end-to-end network transport functions suitable for applications transmitting real-time data, such as audio, video or simulation data, over multicast or unicast network services. The data transport is augmented by a control protocol (RTCP) to allow monitoring of the data delivery in a manner scalable to large multicast networks, and to provide minimal control and identification functionality.

There are five types of RTCP packet which carry a variety of control information:

- SR: Sender report, for transmission and reception statistics from participants that are active senders.
- RR: Receiver report, for reception statistics from participants that are not active senders.
- SDES: Source description items.
- BYE: Indicates end of participation.
- APP: Application-specific functions.

2.2. Congestion Control in WebRTC

2.2.1. WebRTC (Web Real-Time Communication)

WebRTC offers web application developers the ability to write rich, real-time multimedia applications (such as voice call, video call, TV conference) on the web, without requiring plugins, downloads or installs. As written in [8], the purpose of WebRTC is to help build a strong RTC platform that works across multiple web browsers, across multiple platforms. This is a disruptive evolution in the web applications world, since it enables, for the very first time, web developers to build real-time multimedia applications with no need for proprietary plug-ins, as claim by [6].

2.2.1.1. Architecture of WebRTC

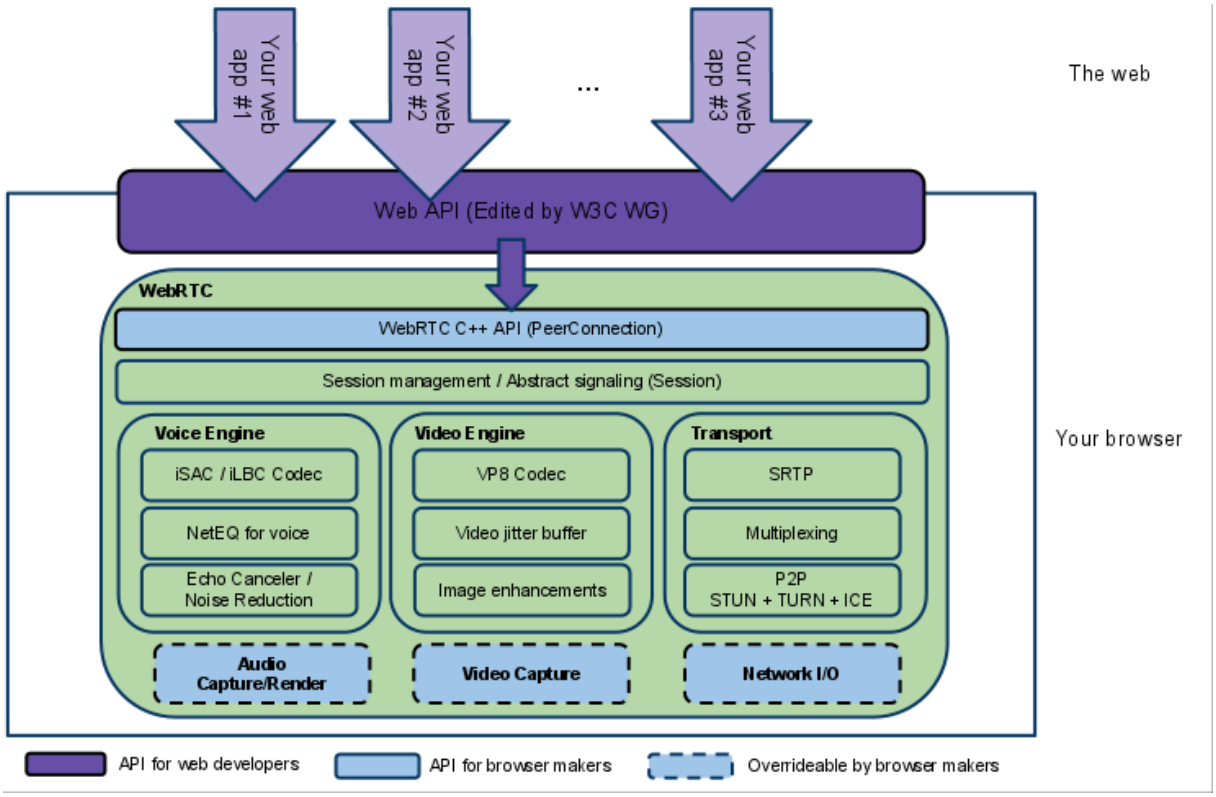

Figure 1 Overall architecture of WebRTC

This diagram is published in [3]. It describes the overall architecture of WebRTC. Browser developers will be interested in the WebRTC C++ API and the capture/render hooks at their disposal to make their browsers support WebRTC while web application developers may be more interested in the Web API.

2.2.1.2. Web APIs

According to [9], WebRTC implements three APIs:

- MediaStream (aka getUserMedia)
- RTCPeerConnection
- RTCDataChannel

MediaStream: Get access to data streams, such as from the user's camera and microphone.

RTCPeerConnection: Audio or video calling, with facilities for encryption and bandwidth management.

RTCDataChannel: Peer-to-peer communication of generic data.

2.2.2. Receiver-side Real-time Congestion Control

In this part, we will discuss about RRTCC algorithm. We focus on its main approach, system model, architecture, deployed scenarios, and implementation experience.

2.2.2.1. Introduction

According to [10], there are so many reasons that congestion control for real-time is facing:

- The media is usually encoded in forms that cannot be quickly changed to accommodate varying bandwidth, and bandwidth requirements can often be changed only in discrete, rather large steps.
- The participants may have certain specific wishes on how to respond - which may not be reducing the bandwidth required by the flow on which congestion is discovered.
- The encodings are usually sensitive to packet loss, while the real-time requirement precludes the repair of packet loss by retransmission.

On the other hand, [11] announced that congestion control is a requirement for all applications sharing the Internet resources. So the WebRTC development team develops RRTCC algorithm (which includes two congestion control algorithms) and confidently said in [10] that they together are able to provide good performance and reasonable bandwidth sharing with other video flows using the same congestion control and with TCP flows that share the same links.

2.2.2.2. System model

The system contains several elements:

- RTP packet - an RTP packet containing media data.

- RTP sender - sends the RTP stream over the network to the RTP receiver. It generates the RTP timestamp.
- RTP receiver - receives the RTP stream, marks the time of arrival.
- RTCP sender at RTP receiver - sends receiver reports, REMB messages and transport-wide RTCP feedback messages.
- RTCP receiver at RTP sender - receives receiver reports and REMB messages and transport-wide RTCP feedback messages, reports these to the sender side controller.
- RTCP receiver at RTP receiver, receives sender reports from the sender.
- Loss-based controller - takes loss rate measurement, round trip time measurement and REMB messages, and computes a target sending bitrate.
- Delay-based controller - takes the packet arrival info, either at the RTP receiver, or from the feedback received by the RTP sender, and computes a maximum bitrate which it passes to the loss-based controller.

There are two ways to implement Receiver-side Real-time Congestion Control. The difference is the positions of loss-based controller and delay-based controller. Both of them are described in [10].

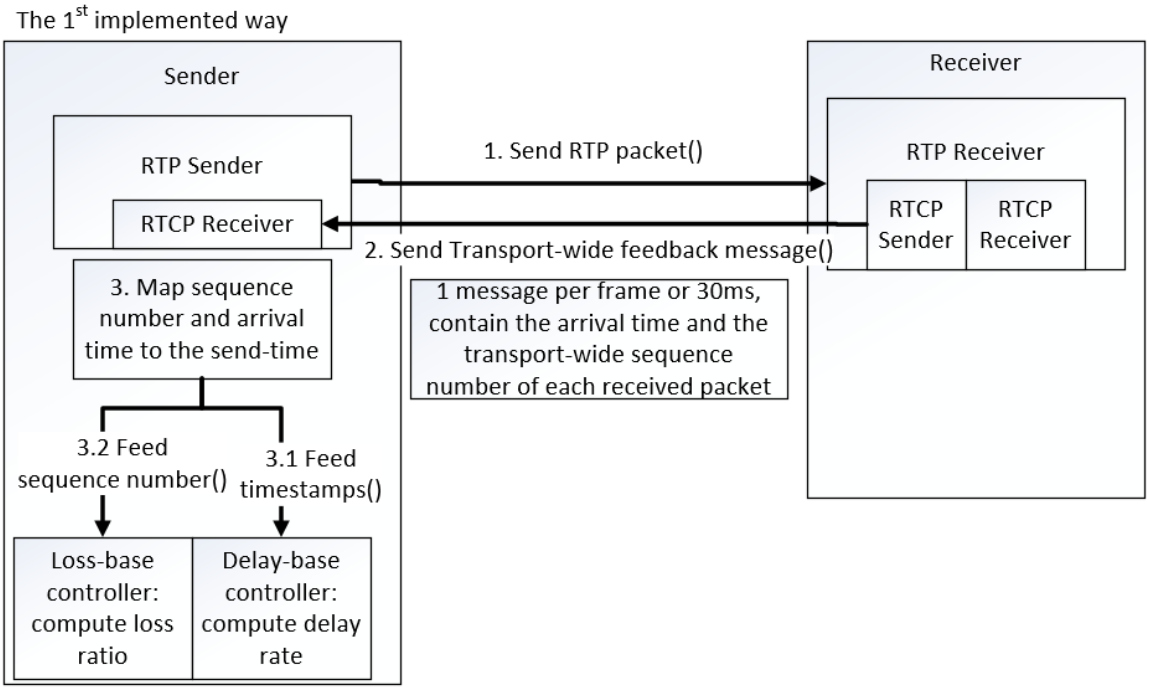

Figure 2 The first version of RRTCC implementation

In the first way, both loss-based controller and delay-based controller locate at the sender. The first version can be realized by using a per-packet feedback protocol. Here, the RTP receiver will record the arrival time and the transport-wide sequence number of each received packet, which will be sent back to the sender periodically using the transport-wide feedback message. The recommended feedback interval is once per received video frame or at least once every 30ms if audio-only or multi-stream. If the feedback overhead needs to be limited this interval can be increased to 100ms. The sender will map the received {sequence number, arrival time} pairs to the send-time of each packet covered by the feedback report, and feed those timestamps to the delay-based controller. It will also compute a loss ratio based on the sequence numbers in the feedback message.

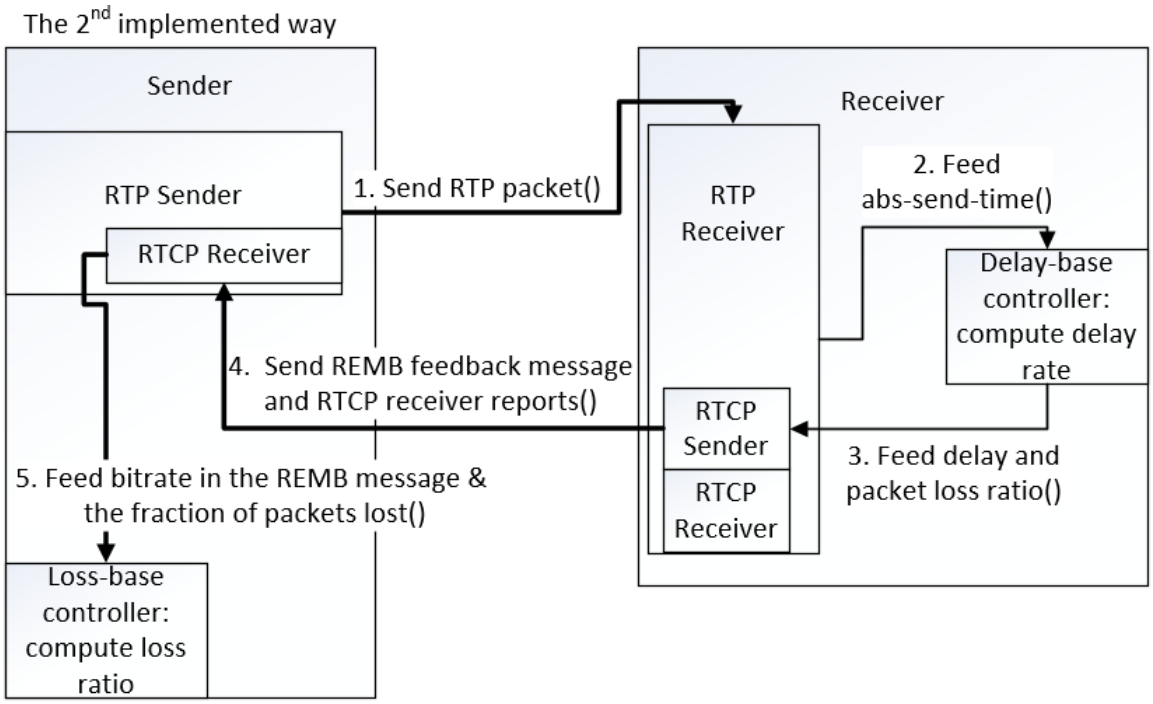

Figure 3 The second version of RRTCC implementation

In the second way, the loss-based controller is put at sender side, while the delay-based controller is put at receiver side. The delay-based controller monitors and processes the arrival time and size of incoming packets. The output from the delay-based controller will be a bitrate, which will be sent back to the sender using the REMB feedback message. The packet loss ratio is sent back via RTCP receiver reports. At the sender the bitrate in the REMB message and the fraction of packets lost are fed into the loss-based controller, which outputs a final target bitrate. It is RECOMMENDED to send the REMB message as soon as congestion is detected, and otherwise at least once every second.

Two parts (loss-based controller and delay-based controller), both are designed to increase the estimate of the available bandwidth (or sending rate). The loss-based estimate As is compared with the delay-based estimate Ar. The actual sending rate is set as the minimum between As and Ar.

2.2.2.3. Delay-based control

The delay-based control algorithm estimates the overuse or underuse of the bottleneck link based on the timestamps. The delay-based control algorithm can be further decomposed into three parts: an arrival-time filter, an over-use detector, and a rate controller.

Firstly, inter-group delay variation is calculated as follows:

$$d(i) = t(i) - t(i-1) - \left(T(i) - T(i-1)\right) = \frac{L(i)}{C(i)} - \frac{L(i-1)}{C(i-1)} + w(i)$$
$$= \frac{L(i) - L(i-1)}{C(i)} + w(i) = \frac{dL(i)}{C(i)} + w(i) = \frac{dL(i)}{C(i)} + m(i) + v(i)$$

Where:

- i is denoted as the i^{th} packet or group packet.
- t(i) is the arrival-time of i^{th} packet or group packet.
- T(i) is the departure-time of i^{th} packet or group packet.
- L(i) is the size of i^{th} packet or group packet.
- d(i): inter-group delay variation.
- C(i): the capacity of the path.
- w(i) = m(i) + v(i) : is a sample from a stochastic process W, which is a function of the capacity C(i), the current cross traffic, and the current sent bitrate.

The receiver tracks d(i) and frame size L, and use Kalman filter to estimate C(i) and m(i). Then it uses those estimates to detect whether or not the bottleneck link is over-used. m(i) is fed to the over-use detector. Overuse is triggered only when m(i) exceeds a threshold value. Underuse is signaled when m(i) falls below a certain threshold value. When m(i) is zero, it is considered a stable situation and the old rate is kept. Then the signal is fed to the rate control subsystem. The rate control subsystem has 3 states: Increase, Decrease and Hold. "Increase" is the state when no congestion is detected. "Decrease" is the state when congestion is detected. "Hold" is a state that waits until built-up queues have drained before going to "increase" state. The state transition is described as in the table below:

Table 1 How status is updated based on signal

	Hold	Increase	Decrease

Over-use	Decrease	Decrease	
Normal	Increase		Hold
Under-use		Hold	Hold

There are two types of Increase state: multiplicative and additive. Multiplicative increase happens if the current bandwidth estimate appears to be far from convergence. Additive increase happens if the current bandwidth estimate appears to be closer to convergence. The notation of estimated bandwidth is $\hat{A}$. $\hat{A}$ is calculated as follows:

- During multiplicative increase, the estimate is increased by at most 8% per second:
  - eta = $1.08^{\min(\text{time\_since\_last\_update\_ms} / 1000, 1.0)}$
  - $\hat{A}$(i) = eta * $\hat{A}$ (i-1)
- During the additive increase the estimate is increased with at most half a packet per response_time interval:
  - response_time_ms = 100 + rtt_ms
  - alpha = 0.5 * min(time_since_last_update_ms / response_time_ms, 1.0)
  - $\hat{A}$(i) = $\hat{A}$ (i-1) + max(1000, alpha * expected_packet_size_bits)
- When an over-use is detected: $\hat{A}$(i) = beta * $\hat{R}$ (i)
  - beta is typically chosen to be in the interval [0.8, 0.95], 0.85 is the RECOMMENDED value.
  - $\hat{R}$ (i) is the currently incoming bitrate which is calculated by receiver.

2.2.2.4. Loss-based control

The loss-based controller should run every time feedback from the receiver is received:

$$A_s = \begin{cases} A_s(i-1)\ x\ (1\ -0.5p)\ if\ p > 0.10 \\ A_s(i-1)\ if\ 0.02 \le p \le 0.1 \\ 1.05\ x\ A_s(i-1) + 1000\ if\ p < 0.02 \end{cases}$$

Where:

- $A_s(i)$ means the sender available bandwidth estimate at time i.
- p is the packet loss ratio which is calculated by sender.

The actual sending rate is set as the minimum between As and Ar.

2.3. An implementation of RRTCC using ns-2

Fairness of RRTCC [12] is a research which show the preliminary results of the fairness of RRTCC when RRTCC and TCP flows share a 5Mbps bottleneck link with 50 or 100ms link delay.

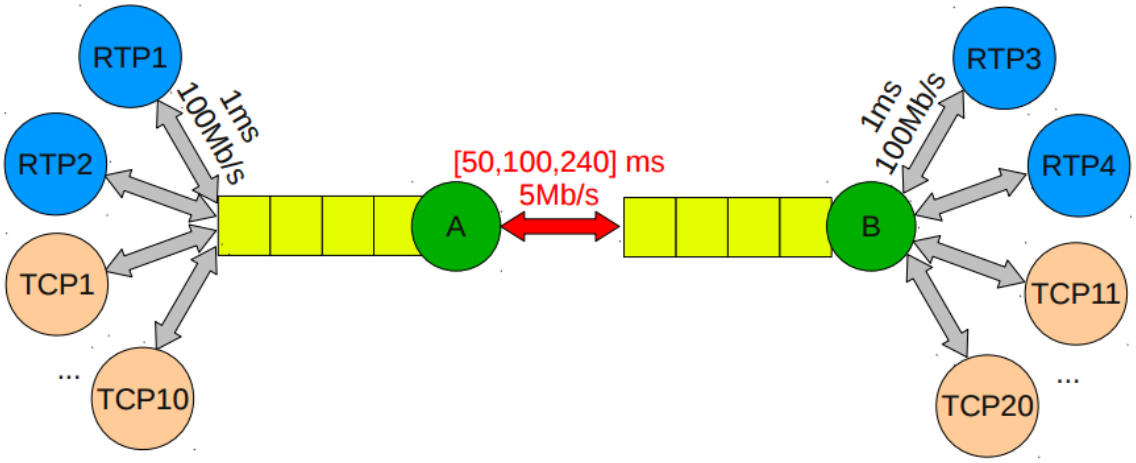

Figure 4 NS-2 implementation setup topology

Here is the configuration:

Table 2 Configuration of RRTCC implementation in ns-2

Parameter	**Value**
Default queue size	50
Codec start rate	128kps
Max. end-to-end delay	400ms
RTCP Interval	1 second

In the scenario when 1 RRTCC and 5 TCP flows share a same bottleneck link, the result shows that RRTCC and TCP collaborate very friendly.

Table 3 The results when 1 RRTCC and 5 TCP flows share a same bottleneck link

	50ms	**100ms**
Goodput	3593 kbps (σ =279)	3700 kbps (σ =420)
Packet Loss Rate	3.68% (σ = 0.26)	4% (σ = 0.19)
Average TCP Throughput (during ON time)	688.5 kbps (σ = 53)	323 kbps (σ = 111)
Average TFS	151% (σ = 8)	72% (σ = 0.21)

where $TFS = \frac{TCP\ Throughput}{\frac{Total\ Throughput}{no.\ of\ flow}}$.

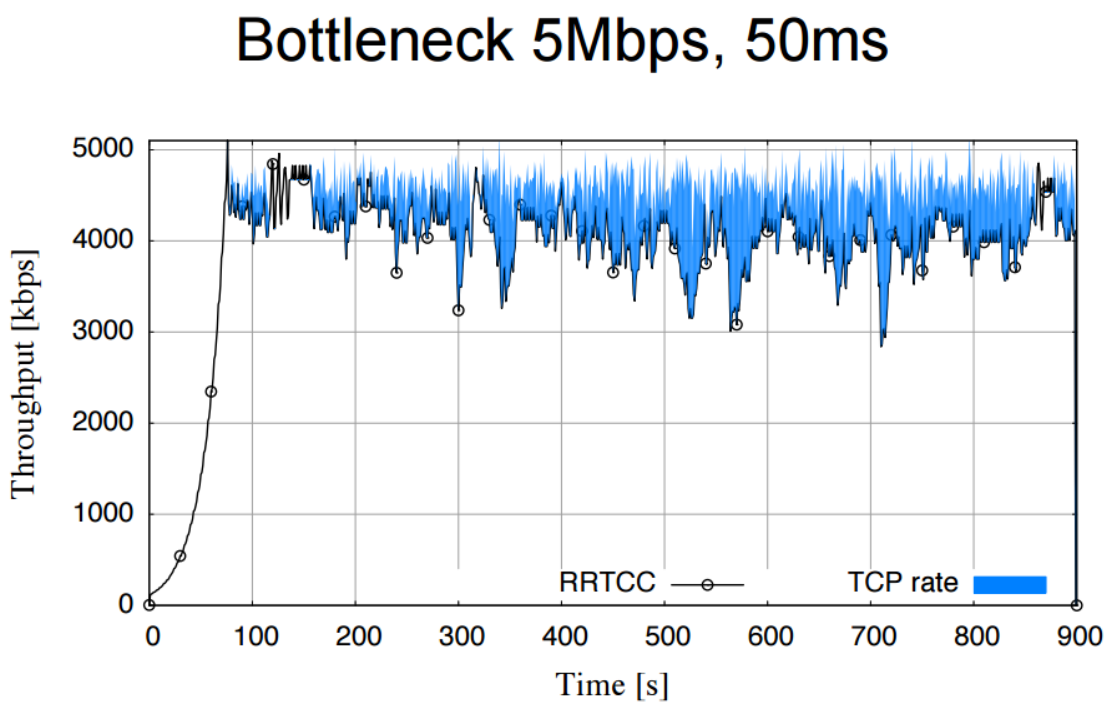

Figure 5 Throughput by time of RRTCC and TCP flows working together on the same bottleneck link

Through this research, we can conclude that RRTCC is promising congestion control algorithm. It proves that RRTCC has great efficiency when it runs with other TCP flows in the internet.

2.4. Tools

2.4.1. SimPy

There are many simulation tools in the market which support network simulation such as ns-2, ns-3, SimPy library… They are designed for different purposes. We decided to use SimPy for our simulation because it is easy to scale and to use.

According to [4], SimPy is a process-based discrete-event simulation framework based on standard Python. Simulations can be performed "as fast as possible", in real time or by manually stepping through the events.

2.4.2. NumPy

NumPy is the fundamental package for scientific computing with Python. As listed in [13], it contains among other things:

- A powerful N-dimensional array object.
- Sophisticated (broadcasting) functions.
- Tools for integrating C/C++ and Fortran code.
- Useful linear algebra, Fourier transform, and random number capabilities.

In our simulation, NumPy is used to generate some random variables by some distributions in some stochastic processes.

2.4.3. PyKalman

As said in [14] , the Kalman Filter is an unsupervised algorithm for tracking a single object in a continuous state space. Given a sequence of noisy measurements, the Kalman Filter is able to recover the "true state" of the underling object being tracked. Common uses for the

Kalman Filter includes radar and sonar tracking and state estimation in robotics. PyKalman is the dead-simple Kalman Filter, Kalman Smoother, and EM library for Python.

In our simulation, PyKalman is used to estimate the mean and covariance of network noise.

Chapter 3

THE METHOD

In this chapter, the simulation we have implemented is discussed in details. It includes:

- How to use SimPy and how we apply SimPy to our simulation.
- How we simulate a real-time system.
- How we implement Receiver-side Real-time Congestion Controller in the first way and UDP Congestion Controller.

3.1. SimPy

A simulation in SimPy can be created by 3 simple steps:

- Step 1: Create an `Environment` instance.
- Step 2: Add a process to the environment instance (a process usually is a function).
- Step 3: Run process.

Now we will analysis a simple example from SimPy documentation [4] in order to know it better. Firstly, we need to understand some basic concepts in SimPy:

Concept	**Description**
Environment (env)	A simulation instance which control simulation scenarios. It contains one or many processes.

Process	An execution thread which simulates something in physical world.
Event	It is something which may happen, is going to happen or has happened in the environment.
Yield	A command. This command triggers an event which is the result of course which processes run.
Timeout	An event type. Events of this type are triggered after a certain amount of (simulated) time has passed.
Resource	Shared resources between processes.
Request	A process needs to request the usage right to a resource.
Preemptive Resource	An important request can take is even when it is being used by other process.

This example models a bank counter and customers arriving at random times. Each customer has a certain patience. It waits to get to the counter until he's at the end of his tether. If he gets to the counter, he uses it for a while before releasing it.

```
"""
Bank renege example
Scenario:
  A counter with a random service time and customers who renege.
"""
import random
import simpy

RANDOM_SEED = 42
NEW_CUSTOMERS = 5  # Total number of customers
INTERVAL_CUSTOMERS = 10.0  # Generate new customers every x seconds
MIN_PATIENCE = 1  # Min. customer patience
MAX_PATIENCE = 3  # Max. customer patience

def source(env, number, interval, counter):
    """Source generates customers randomly"""
```

```
    for i in range(number):
        try:
          c = customer(env, 'Customer%02d' % i, counter,
time_in_bank=12.0)
          env.process(c)
        except simpy.Interrupt:
          print('Was interrupted.")
        t = random.expovariate(1.0 / interval)
        yield env.timeout(t)


def customer(env, name, counter, time_in_bank):
    """Customer arrives, is served and leaves."""
    arrive = env.now
    print('%7.4f %s: Here I am' % (arrive, name))

    with counter.request() as req:
        patience = random.uniform(MIN_PATIENCE, MAX_PATIENCE)
        # Wait for the counter or abort at the end of our tether
        results = yield req | env.timeout(patience)
        wait = env.now - arrive
        if req in results:
            # We got to the counter
            print('%7.4f %s: Waited %6.3f' % (env.now, name, wait))
            tib = random.expovariate(1.0 / time_in_bank)
            yield env.timeout(tib)
            print('%7.4f %s: Finished' % (env.now, name))
        else:
            # We reneged
            print('%7.4f %s: RENEGED after %6.3f' % (env.now, name,
wait))

# Setup and start the simulation
print('Bank renege')
random.seed(RANDOM_SEED)
env = simpy.Environment()

# Start processes and run
counter = simpy.Resource(env, capacity=1)
env.process(source(env, NEW_CUSTOMERS, INTERVAL_CUSTOMERS, counter))
env.run()
```

The program start at line `print('Bank renege')`. It seeds for random library, create an `Environment` instance, create a `Resource` named counter (it is the ATM machine) with the quality of 1. Then it add a process to the environment and run.

Now we take a closer look to the process named `source`. `source` takes four arguments which are `env`, `NEW_CUSTOMERS`, `INTERVAL_CUSTOMERS` and `counter` corresponding with fours parameters which are `env`, `number`, `interval`, `counter` respectively. `source` creates a number of "*number*" customer, next customer shows up after a random interval time. While creating customer, `source` listens for interruption from outside. If an interruption occurs, `source` will stop.

Function `customer` takes four parameters named `env`, `name`, `counter`, `time_in_bank`. The time when customer arrives at the ATM machine is when the function `customer` is called. Customer requests the ATM machine by `with counter.request() as req:` Patience of a customer is measure by his waiting time which is initiated randomly between MIN_PATIENCE and MAX_PATIENCE. A customer waits until his request is responded or his patience runs out `results = yield req | env.timeout(patience)`. If customer gets the counter first, the system will print out the time he waited, and the time he finishes using the ATM machine. If the patience runs out first, he will leave and the system will print out the time he waits before reneges.

An output of the above program:

```
Bank renege
 0.0000 Customer00: Here I am
 0.0000 Customer00: Waited  0.000
 3.8595 Customer00: Finished
10.2006 Customer01: Here I am
10.2006 Customer01: Waited  0.000
12.7265 Customer02: Here I am
13.9003 Customer02: RENEGED after  1.174
23.7507 Customer01: Finished
34.9993 Customer03: Here I am
34.9993 Customer03: Waited  0.000
37.9599 Customer03: Finished
40.4798 Customer04: Here I am
40.4798 Customer04: Waited  0.000
43.1401 Customer04: Finished
```

In our implementation, we created an `Environment` instance, then added few process in order to send and forward RTP and RTCP packet. We will talk about these processes in details later.

3.2. Architecture of our simulation

We need three main components to create a transmission session: sender, receiver and general manager. All nodes is known by the manager. Inside the manager, there is a network which play the role as a transmission environment between sender and receiver.

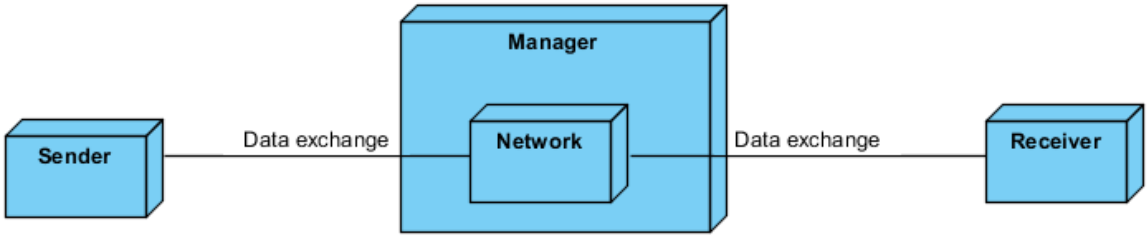

Figure 6 Simple architecture of our simulation

Network in real life is a set of computers, routers, switches… and it is very complicated to implement in details. Our network simply is a combination of some probabilistic models which decide whether a packet is lost or not. If loss does not happen, the network will create delay time for this packet randomly. Packets are transmitted from sender to receiver through the network. The main workflow of our simulation is described as the chart below:

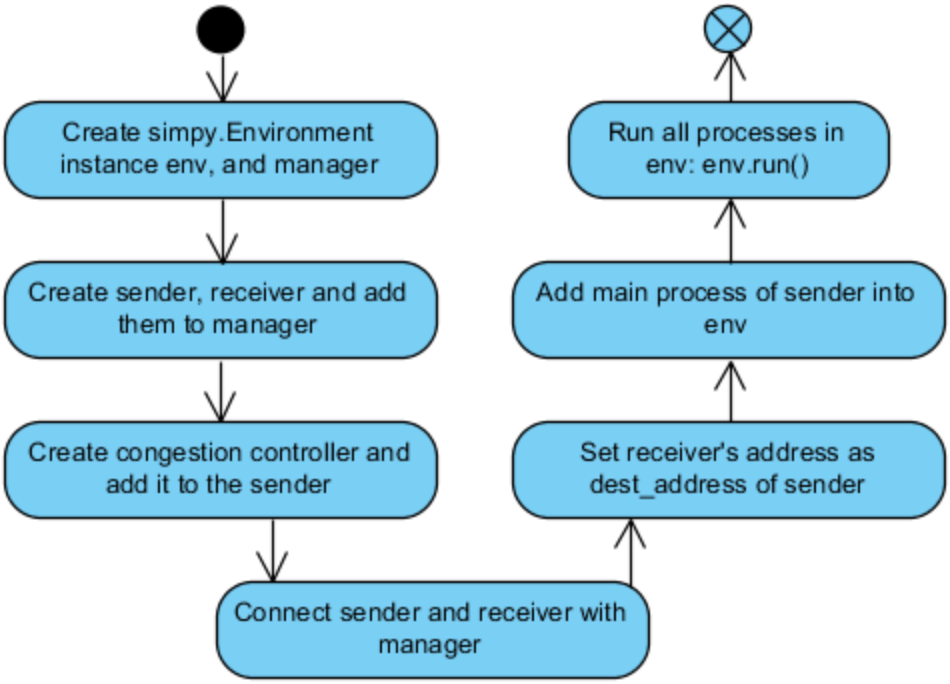

Figure 7 Main program of our simulation

As we mentioned before, all processes of a SimPy simulation live in a `simpy.Environment` instance. So we create an `Environment` instance first, then the manager. After that, sender and receiver application are created and added to the manager. Then we create a congestion controller and plug it into the sender. Next, we set the destination address for sender to send multimedia data. Then main process (sending data process) is added to the simulation environment. This process will send multimedia data (RTP packet) to receiver and start several other process to support sending data (such as RTCP sending process). Now all things have been set up. The last thing to do is call `env.run()`, which will run all processes added in the environment before.

In the next parts, we will talk about each module in details.

3.3. Simulation scenario

Here are 4 basic steps of our simulation scenario:

- Step 1: Sender sends a number of *RTP_packets_num* RTP packets with the same size to receiver in total. After a RTP packet is sent, sender waits *RTP_interval* seconds until it sends the next RTP packet.
- Step 2: After received the first RTP packet, receiver starts RTCP process (a SimPy procees), which sends RTCP report back to sender.
- Step 3: When the sender have just sent all RTP packets, it sends a BYE RTCP packet to receiver.
- Step 4: When receiver get the BYE packet, it interrupts the RTCP process, the session is over.

We make a few assumptions for our simulation:

- Data (RTP packet) is generated as requested.
- All the RTP packets have the same size.
- For each packet loss, the inter-group delay variation of this packet is assigned with the value of the previous packet (except the first one).
- The RTP sending rate is upper bounded and lower bounded by the maximum and minimum sending bitrate.
- The RTP sending rate does not need to be in some specific values (quality levels like 144p or 320p…) because it can contain redundant data which supports to Forward Error Correction (FEC).
- The RTP sending rate is upper bounded and lower bound by a maximum and minimum value respectively because it depends on maximum and minimum quality of the transmission.

3.4. Modules in details

3.4.1. RTP and RTCP packet

RTP and RTCP packet are simulated as two separated classes, each class contains several necessary properties that supported for our simulation. Both of them inherit from a general class named `Packet`. The properties of this two classes are based on RFC3550 [7].

Sender sends multimedia data to receiver by RTP packets.

RTP packet class contains several properties:

Table 4 RTP packet class's properties

	Property's name	**Meaning**
1	source_address	The source address of the packet
2	dest_address	The destination address of the packet
3	sq_num	The sequence number of the packet
4	timestamp	The timestamp of the packet

Sender and receiver control transmission session by notifying their partner some necessary information by RTCP packet.

RTCP packet class contains several properties:

Table 5 RTCP packet class's properties

	Property's name	**Meaning**
1	source_address	The source address of the packet
2	dest_address	The destination address of the packet
3	type	Indicate the type of the RTCP packet (There are five types of RTCP packets as we said before)
4	timestamp	The timestamp of the packet
5	fb_sq_num	The sequence number of the feedback packet

6	sq_num_vector	The vector that contains the packet's information which receiver received and send back to sender
7	base_transport_sq_num	The first sequence number of packets whose information is in the sq_num_vector

3.4.2. `RTPApplication` class: Sender and Receiver.

To simulate sender and receiver, we design a class named `RTPApplication`. This class is the blueprint for sender and receiver. RTP application is general real-time communication entity.

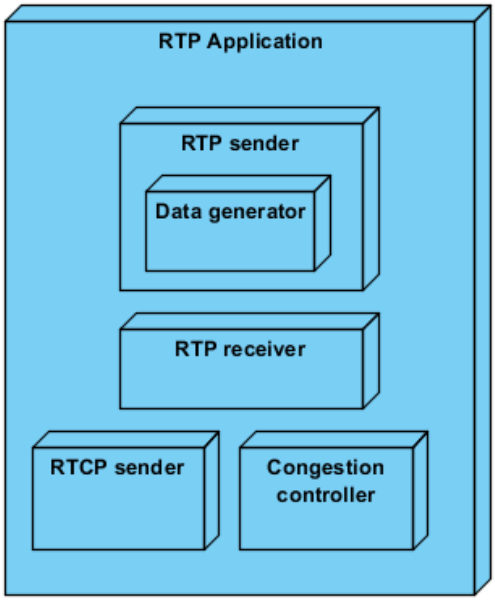

Figure 8 Full-duplex RTP application architecture

An RTP application has several main components, each component plays a specific role:

- RTP sender: sends the RTP stream over the network to the RTP receiver.
- Data generator: is a part of RTP sender, which generates RTP packet.
- RTP receiver: receives the RTP stream, marks the arrival time.
- Congestion controller: controls sending rate of RTP packet.

- RTCP sender: sends RTCP reports.

Multimedia data will be transferred as flowchart below:

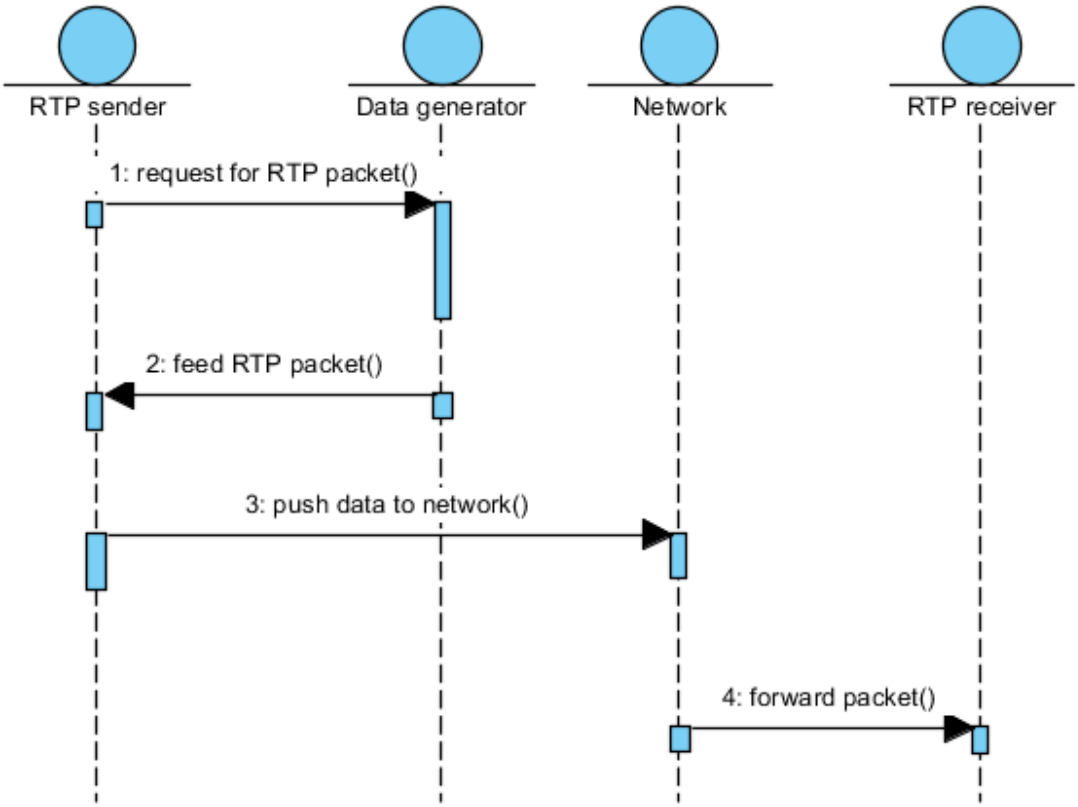

Figure 9 RTP packet flow

Firstly, RTP sender is started by the main process. When it requests data generator for a RTP packet, a RTP packet is created and fed back to RTP sender. Then it pushes data to `network` in order to forward this packet to receiver. `Network`, when receives this packet, checks for its destination address and forward it to the corresponding RTP receiver. But in reality, packet loss can happen due to many reasons. The following chart describes the behavior of `network` in our simulation.

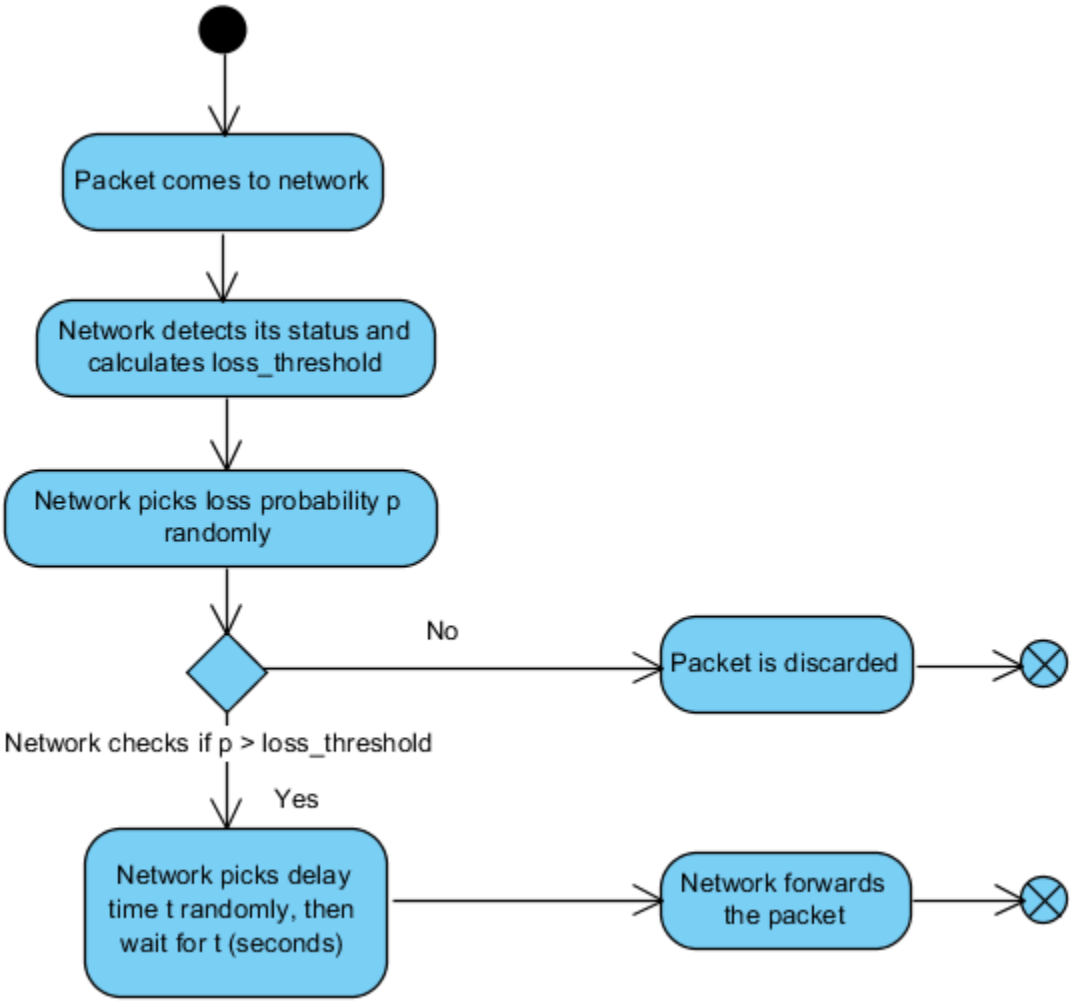

Figure 10 The behavior of `network`

Our network simply is a combination of some probabilistic models. It has some statuses:

- The first one is when the bandwidth is not used by anyone. We call it “Idle”.
- The second one is when the network link is used by some flows, the available bandwidth is smaller than when network is Idle. We call it “Cross-traffic”.
- The last one is when congestion happens and the available bandwidth is zero. We call it “Saturated”.

Each status occurs with different probability, where “Saturated" has the smallest probability and “Idle" has the highest one. `Network` updates its status each an interval time.

`Network` can handle `lambda_out` bits per second at a time. `lambda_out` is different for each `network`’s status. Combine with the incoming bitrate, network calculates loss threshold of a packet when it comes to `network`.

For each packet come to `network`, it choose a real number between 0 and 1 and compare it with loss threshold. By this action, it will decide whether a packet is lost or not, and if lost does not happen, it creates delay time (transmission time) for this packet randomly.

When a RTP packet comes to receiver, RTP receiver will be responsible to handle this packet.

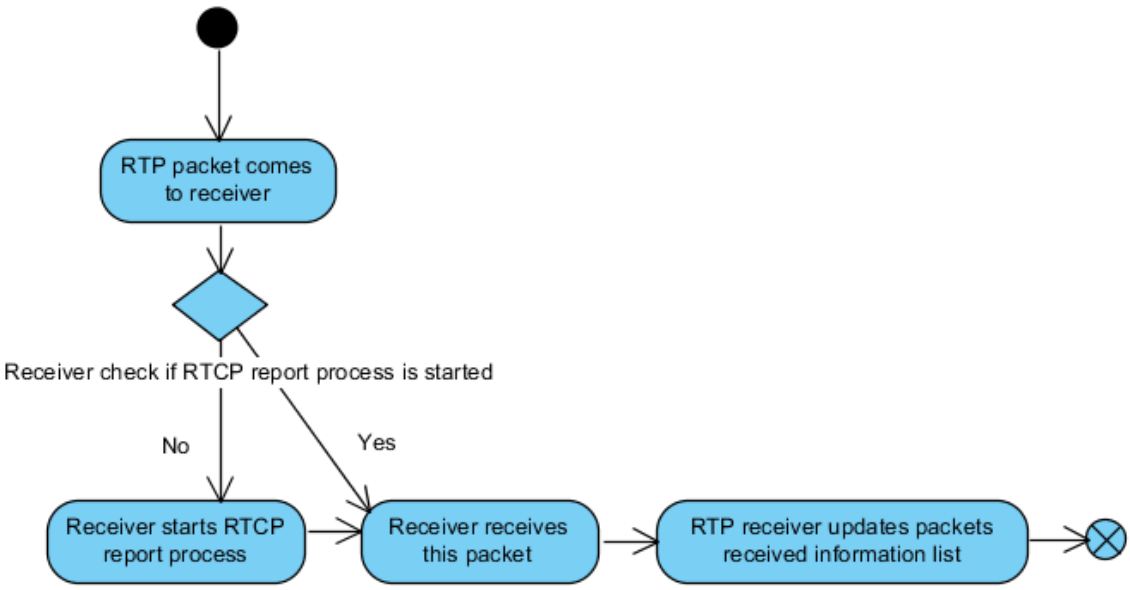

Figure 11 RTP receiver handles RTP packet

In a real-time transmission session, receiver needs to send some information (likes packet arrival status, delay time…) to sender through RTCP report packets. So when a RTP packet come, receiver checks whether RTCP report process is started. If not, the RTCP report

process will be started. Then receiver will receive this packet and update some information about this packet.

This updating action records two kinds of information. The first thing is the arrival time of RTP packet. The second thing is the arrival status. There are 3 kinds of status: 1 for arriving on time, 2 for arriving late, 0 for missing. The default value is 0.

RTCP report process sends RTCP report packet back to sender frequently each `RTCP_interval` seconds excepts when an interruption happens. An interruption occurs when receiver receives a RTCP BYE packet from sender. When receiver get this packet, it understands the session is over and stops its RTCP report process. RTCP contains arrival status, latency and delay information of packets which receiver receives in an amount of time.

When sender gets a RTCP report packet, it will use information in this packet combined with data it has received before to adjust the RTP sending rate. And this is where RRTCC shows its power to provide a smooth transmission session.

In the next part, we will talk more specific about how RRTCC is implemented to uses loss and delay information to adjust multimedia sending rate. Besides is the UDP Congestion Controller, which is implemented to be compared with RRTCC.

3.4.3. `Congestion controller`

RTP application has a `congestion controller` to avoid congestion by adjusting sending rate. In our simulation, we implement two versions of congestion controller: Receiver-Side Real-Time Congestion Controller and UDP congestion controller.

3.4.3.1. Receiver-Side Real-Time Congestion Controller

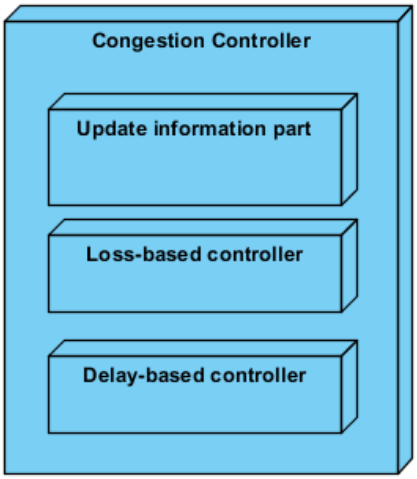

Figure 12 Receiver-Side Real-Time Congestion Controller structure

Update packet's information part

Sender updates information about RTP packets that it sent when it receives a RTCP report packet.

As you have known, a RTCP report packet contains packet's arrival status (on-time/late/missing) and arrival time of RTP packets which receiver received. When this information is transferred back to sender, it is stored in a multidimensional array. Then bases on it, sender calculates inter-group delay variation d(i):

$$d(i) = t(i) - t(i-1) - \left(T(i) - T(i-1)\right)$$

Where:

- i is denoted as the i^{th} packet or group packet.
- t(i) is the arrival-time of i^{th} packet or group packet.
- T(i) is the departure-time of i^{th} packet or group packet.

Then it uses Kalman filter to find out the mean m(i) and covariance of network noise. Network noise is a function of the link capacity, the current cross traffic, the current sent bitrate, and it is modeled as a white Gaussian process [10].

Adjust RTP sending rate

`As` and `Ar` are the estimated bandwidth which are calculated by loss-based controller and delay-based controller. The minimum of them will be chosen as the current estimated bandwidth. Sender will adjust the RTP sending rate based on it. Then the `RTP_sending_rate` will be adjusted to be more proportional. Now we will talk about loss-based controller and delay-based controller in details, how they estimates the current bandwidth.

Loss-based controller

Loss-based controller suggests the estimated bandwidth through loss information.

It counts the number of packets which are lost in last `RTCP_limit` packets was sent (except un-reported packets). Then it calculate the loss probability `p` by divide number of lost packet to `RTCP_limit`. Then `p` is compared to some thresholds which is specify in [10]. Depends on p and the last bandwidth estimated by loss-based controller, loss-based controller will predict the next bandwidth:

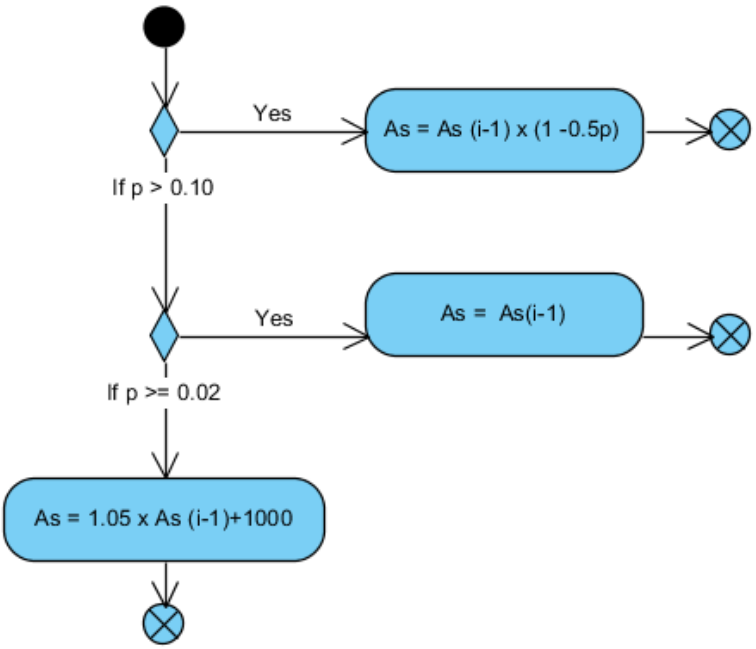

Figure 13 Update A_s based of lost rate

Delay-based controller

Delay-based controller suggests the estimated bandwidth through delay information.

Firstly, the delay-based controller update a threshold named `del_var_th` and the state of itself. `del_var_th` is calculated from last `del_var_th`, arrival times `t(i)` and `t(i-1)`, the inter-group delay variation estimate `m(i)`, and `K(i)`, with `K(i)=K_d` if $|m(i)| <$ `gamma_1(i-1)` or `K(i)=K_u` otherwise. `K_d` and `K_u` are constants, whose recommended values are 0.01 and 0.00018 respectively.

$$\text{del\_var\_th}(i) = \text{del\_var\_th}(i-1) + (t(i)-t(i-1)) * K(i) * (|m(i)| - \text{del\_var\_th}\_1(i-1))$$

Secondly, sender updates the RTP sending session state. There are three kinds of state: increase (sending rate), decrease (sending rate), and hold (sending rate). There are three kinds of signals: over-use, under-use, normal. Bases on the current state and the signal, the state transition is made. The inter-group delay variation estimate `m(i)`, obtained as the output of the arrival-time filter, is compared with a threshold `del_var_th(i)`.

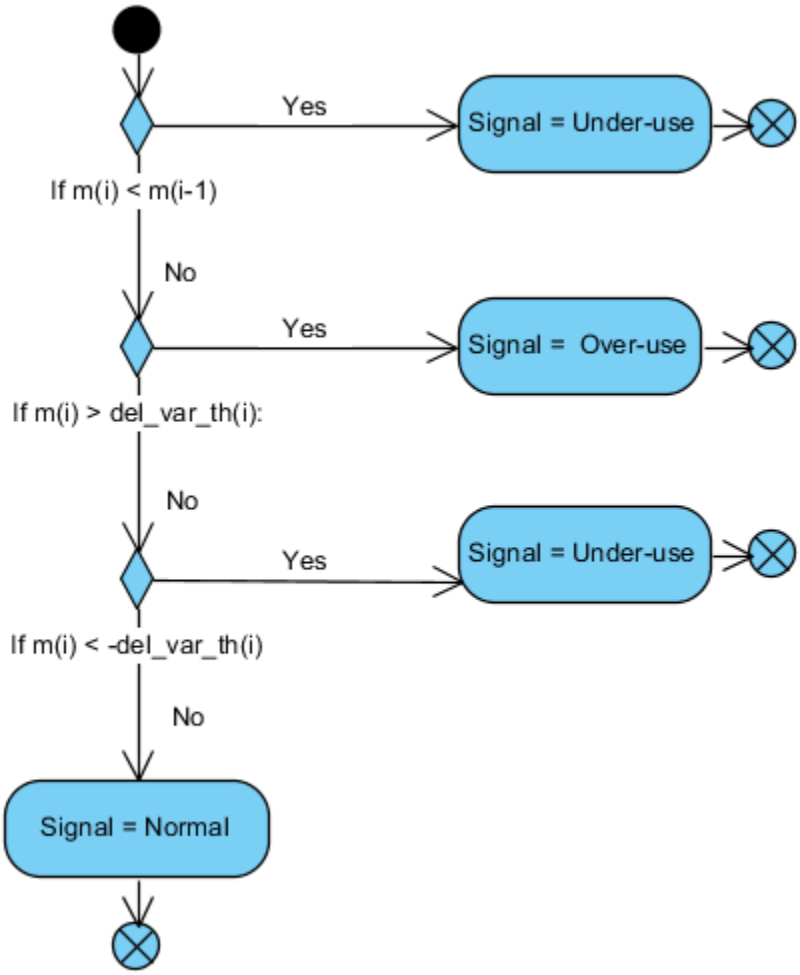

Figure 14 Create transition signal based on m(i)

An estimate above the threshold is considered as an indication of over-use. However, if `m(i) < m(i-1)`, over-use will not be signaled even if all the above conditions are met. Similarly, the opposite state, under-use, is detected when `m(i) < -del_val_th(i)`. If neither over-use nor under-use is detected, the detector will be in the normal state [10]. After achieving signal, sender will use it to determine the next RTP sending session state. How the state is updated is described in the finite state machine below:

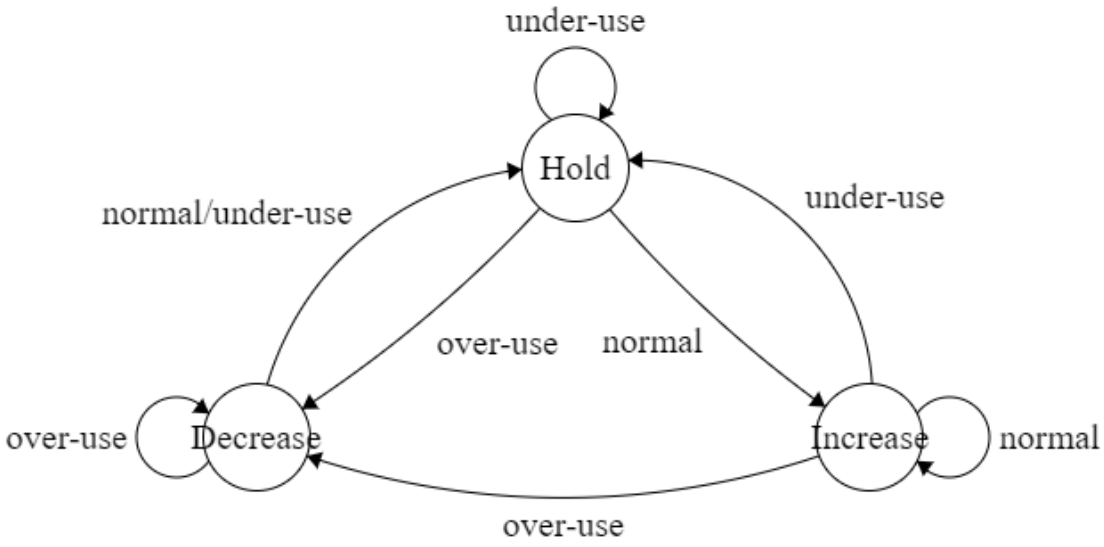

Figure 15 The finite state machine of RTP sending session state

Moreover, there are two types of increase state: additive increase and multiplicative increase The system does a multiplicative increase if the current bandwidth estimate appears to be far from convergence, while it does an additive increase if it appears to be closer to convergence (if the currently incoming bitrate `R(i)`, is close to an average of the incoming bitrates).

Thirdly, delay-based controller will estimates incoming bitrate `R(i)` based on data receiver received. `R(i)` is measured by the delay-based controller over a `T` seconds window:

```
R(i) = 1/T * sum(RTP_packet_size * N(i))
```

Where `N(i)` is the number of packets receiver receives over T seconds.

Finally, delay-based controller will estimate the bandwidth Ar.

3.4.3.2. UDP congestion controller

In our simulation, UDP congestion controller means no congestion controller at all. Sender sends RTP packet in a constant bitrate and do not care about how the network is, whether the loss ratio is high. It still handle the RTCP packet sent from receiver and use the information inside to compute the loss ratio. This congestion controller is implemented in order to evaluate the efficiency of Receiver-side Real-time Congestion Control algorithm

more exactly by the comparison between Receiver-side Congestion Controller and UDP Congestion Controller.

RESULTS AND DISCUSSIONS

This chapter shows the evaluations of our RRTCC implementation presented in previous chapter. Firstly, we describe our simulation setting including simulation times and parameters related to bandwidth estimation and loss ratios. Secondly, simulation results of RRTCC performance are shown in comparison with UDP, the de facto transport protocol for real-time multimedia applications.

In our simulation, there is one sender, one receiver and one manager. The manager contains a network. Sender is connected with receiver by the network. Both sender and receiver are RTP applications. Sender sends RTP packets to receiver and receiver sends RTCP packets back to sender to help sender adjust sending rate, enhance the quality of the transmission session.

We ran our simulation a number of times with different configurations. All of them follow the simulation scenario and assumptions that we said in Chapter 3. For each configuration, we ran our simulation several times in two case: when the sender uses Receiver-side Real-time Congestion Controller, and when the sender uses UDP Congestion Controller.

4.1. Results in a specific configuration

Below are two remarkable results with the same configuration. Here is the configuration:

Table 6 RRTCC configuration

Parameter	**Value**	**Measurement unit**
del_val_th[1]	12.5	Millisecond
K_u	0.01	
K_d	0.00018	
Beta	0.95	
T (Time window for measuring the loss ratio)	0.5	Second

Table 7 `RTP Application` and `Network` configuration

Parameter	**Value**	**Measurement**
Initial sending rate	7200000	Bits/second
Maximum sending rate	10000000	Bits/second
Minimum sending rate	3000000	Bits/second
Numbers of RTP packets	1000	Packet
RTCP interval	3	Second
Mean of network delay	0.01	Second
Variance of network delay	0.001	Second
Network updating interval time	2	Second

Table 8 `Network`'s bandwidth and the probabilities corresponding with its status for RRTCC simulation

Network status	Bandwidth (bits/second)	Probability
Idle	5760000	70%
Cross-traffic 1	4320000	25%
Cross-traffic 2	2880000	
Saturated	0	5%

The chart bellow shows the estimated bandwidth of loss-based controller, delay-based controller, and final estimated bandwidth according time when the sender uses Receiver-side Real-time Congestion Controller:

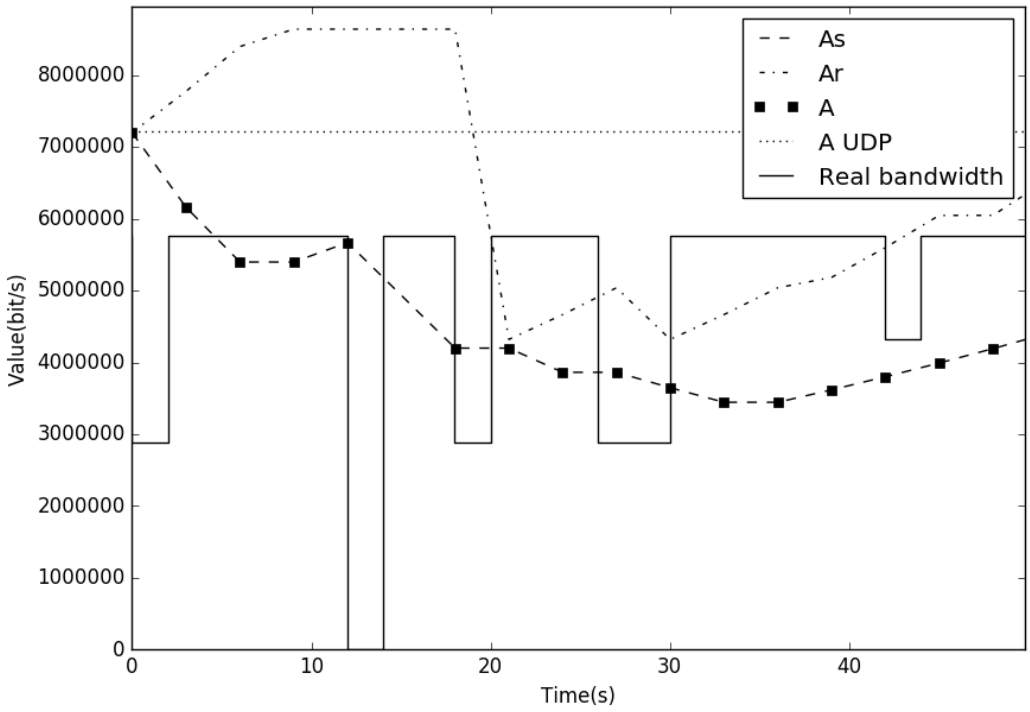

Figure 16 Estimated bandwidths of the sender having Receiver-sider Real-time Congestion Controller

Meanwhile:

- Ar: Estimated bandwidth of delay-based controller.
- As: Estimated bandwidth of loss-based controller.
- A: Final estimated bandwidth (It is also the sending rate of sender using Receiver-side Real-time Congestion Control).
- A UDP: The constant sending rate of sender using UDP congestion control, which is kept at 7200000 bits per second.
- Real bandwidth: The real bandwidth of `network.`

The average final estimated bandwidth is 7082516 bits per second.

The loss ratio calculated by sender in two cases are presented in the chart below:

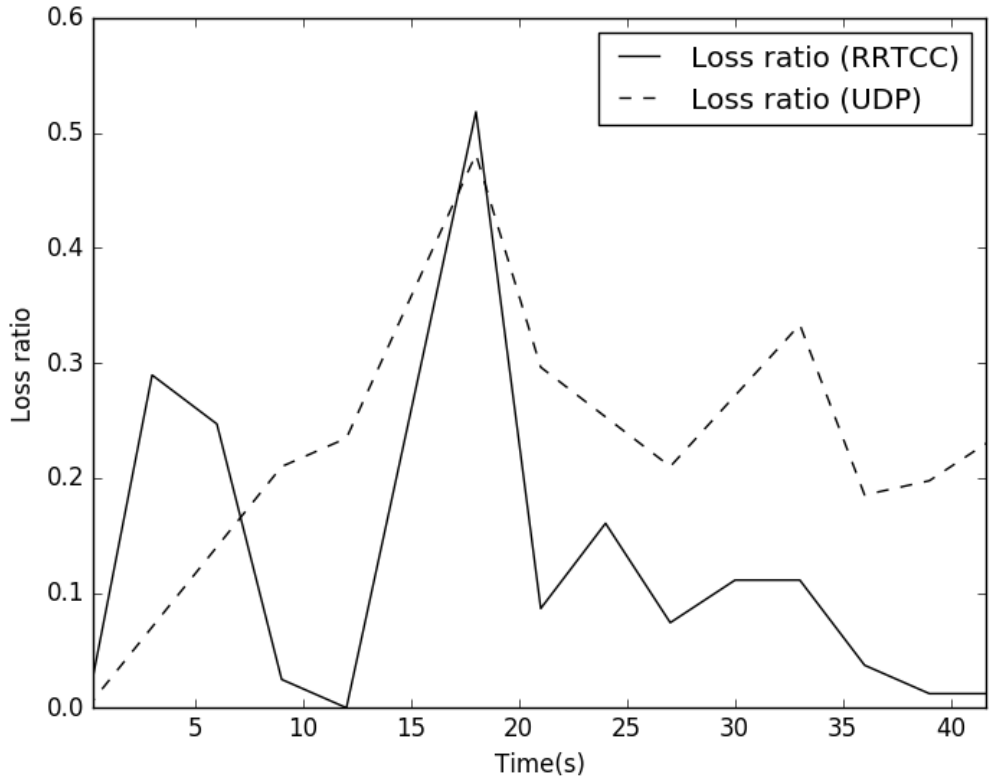

Figure 17 Loss ratio calculated by sender in both cases

Meanwhile:

- Loss ratio (RRTCC): The loss ratio calculated by sender using Receiver-side Real-time Congestion Controller.
- Loss ratio (UDP): The loss ratio calculated by sender using UDP Congestion Controller.

Those charts above represent the trend of estimated bandwidth of sender which uses Receiver-side Real-time Congestion Controller, and the loss ratio of packets in both case of a specific run time.

As you can see, Receiver-side Real-time Congestion Control has done a great job to keep the loss ratio of the transmission low. The loss ratio of the sender which implements RRTCC fluctuates at first, then becomes stable. It is kept lower than 20 percent most of time. It is about 2 to 3 times lower than the loss ratio of the sender which just implements UDP congestion control.

The estimated bandwidth of RRTCC is stable enough to be suitable with a real-time system. In this situation, Receiver-side Real-time Congestion Controller helps sender sending its data with just a little bit lower sending rate than UDP Congestion Controller but with much more lower loss ratio.

4.2. The comparison between RRTCC and UDP congestion control

With the same configuration as the run above, we changed the `Network` configuration and ran our simulation several times in two cases: when the sender uses UDP Congestion Controller, and when the sender uses Receiver-side Real-time Congestion Controller. Because `network`'s bandwidth according time is generated randomly, we recorded it in the first case and used it in the second case every time. Then we calculated the average of loss ratio, RTP sending rate and goodput. Here are `network` configurations:

Table 9 `Network` configurations

	Bandwidths for `network`'s statuses (bits per second)

Configuration	Idle	Cross-Traffic 1	Cross-Traffic 2	Saturated
1	4320000	2880000	1440000	0
2	5760000	4320000	2880000	0
3	7200000	5760000	4320000	0
4	10080000	7200000	5760000	0

Here are the results - the average of loss ratio and RTP sending rate:

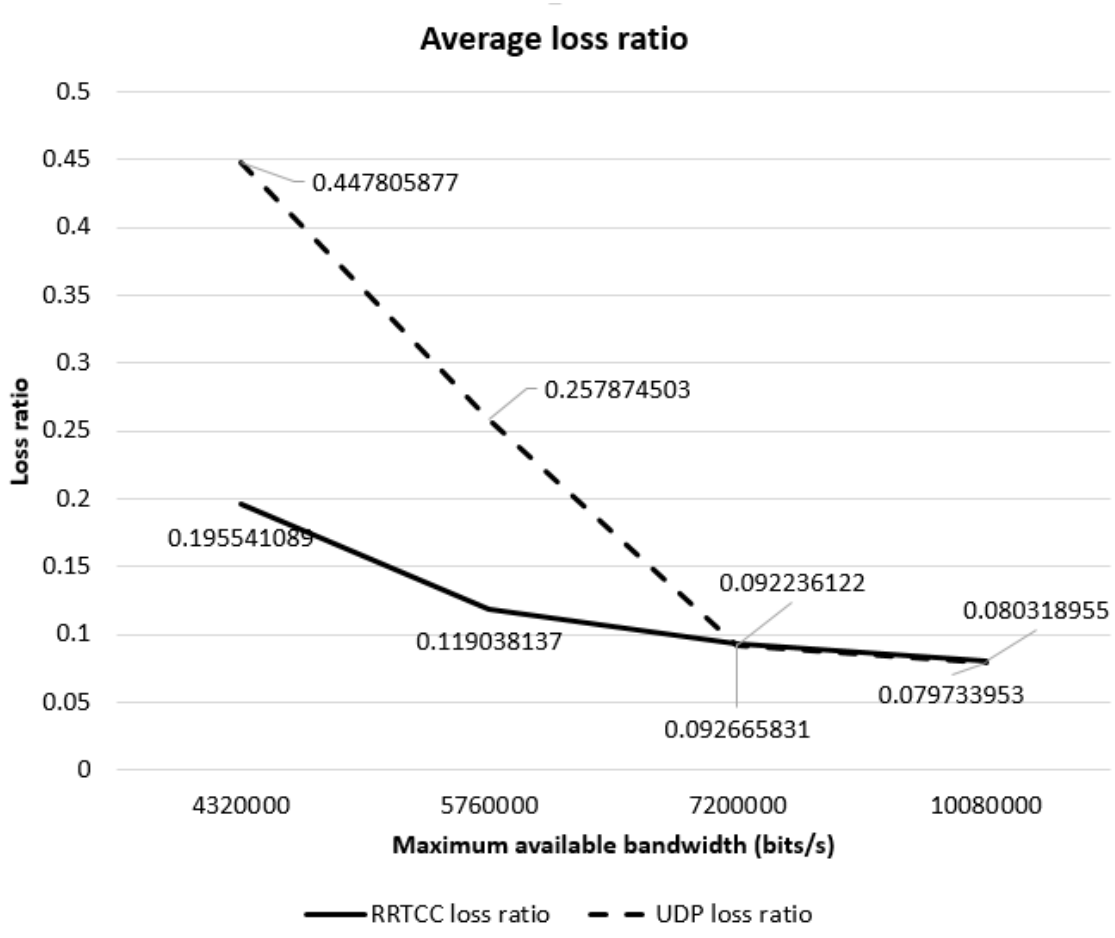

Figure 18 Average loss ratio for different `network` configurations

Meanwhile:

- RRTCC loss ratio: The loss ratio calculated by sender using Receiver-side Real-time Congestion Controller.
- UDP loss ratio: The loss ratio calculated by sender using UDP Congestion Controller.

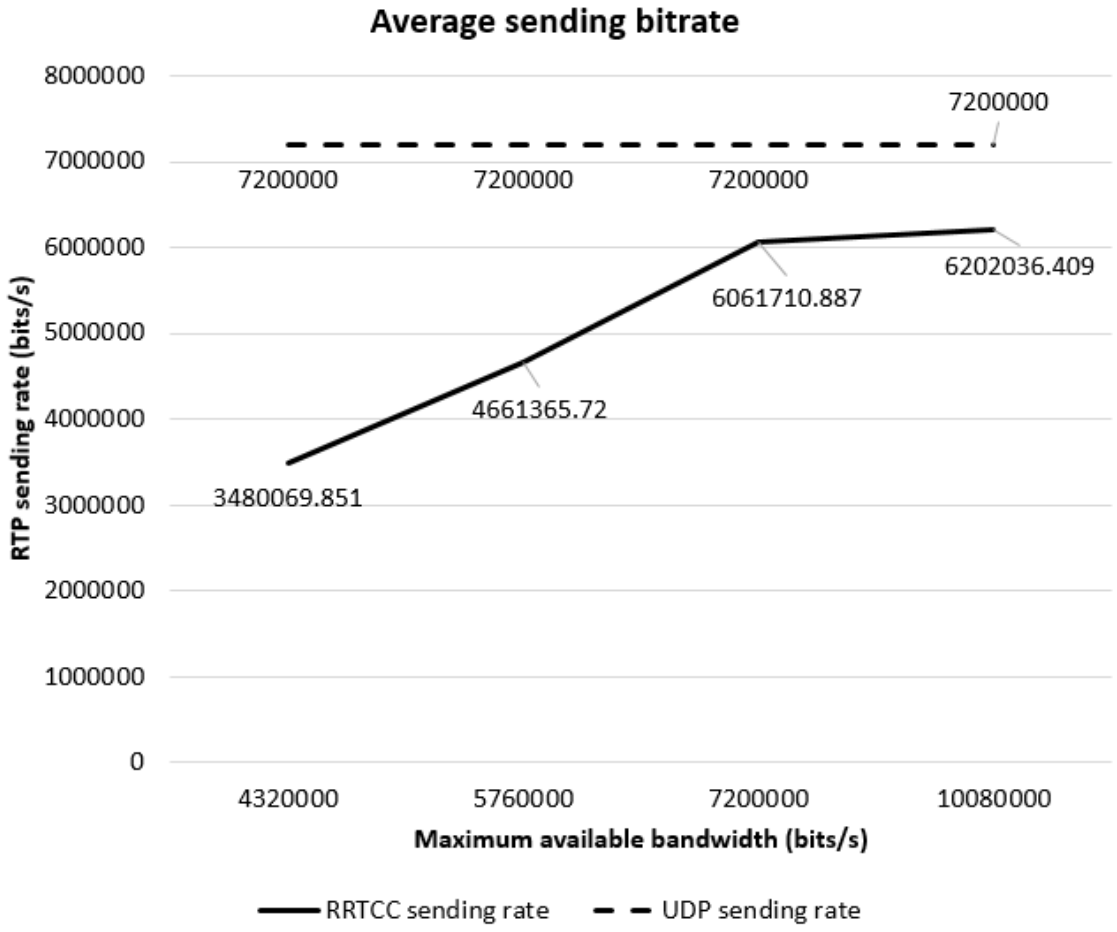

Figure 19 Average sending bitrate for different `network` configurations

Meanwhile:

- RRTCC sending rate: The average sending bitrate of sender using Receiver-side Real-time Congestion Controller.
- UDP sending rate: The average sending bitrate of sender using UDP Congestion Controller.

Through those charts above, we see that Receiver-side Real-time Congestion Control shows a better performance than UDP Congestion Control when the network bandwidth is lower than UDP congestion controller's sending bitrate. Receiver-side Real-time Congestion Controller helps sender sending its data with just a little bit lower sending rate than UDP Congestion Controller but with much more lower loss ratio. But when network bandwidth is quite the same of higher than UDP congestion controller's sending bitrate, the sending bitrate in both cases is approximate and the loss ratio of sender using UDP Congestion Control is small enough to ensure a good real-time transmission.

To be more specific, let's look at the average goodput (the real bitrate that receiver receives) in both cases.

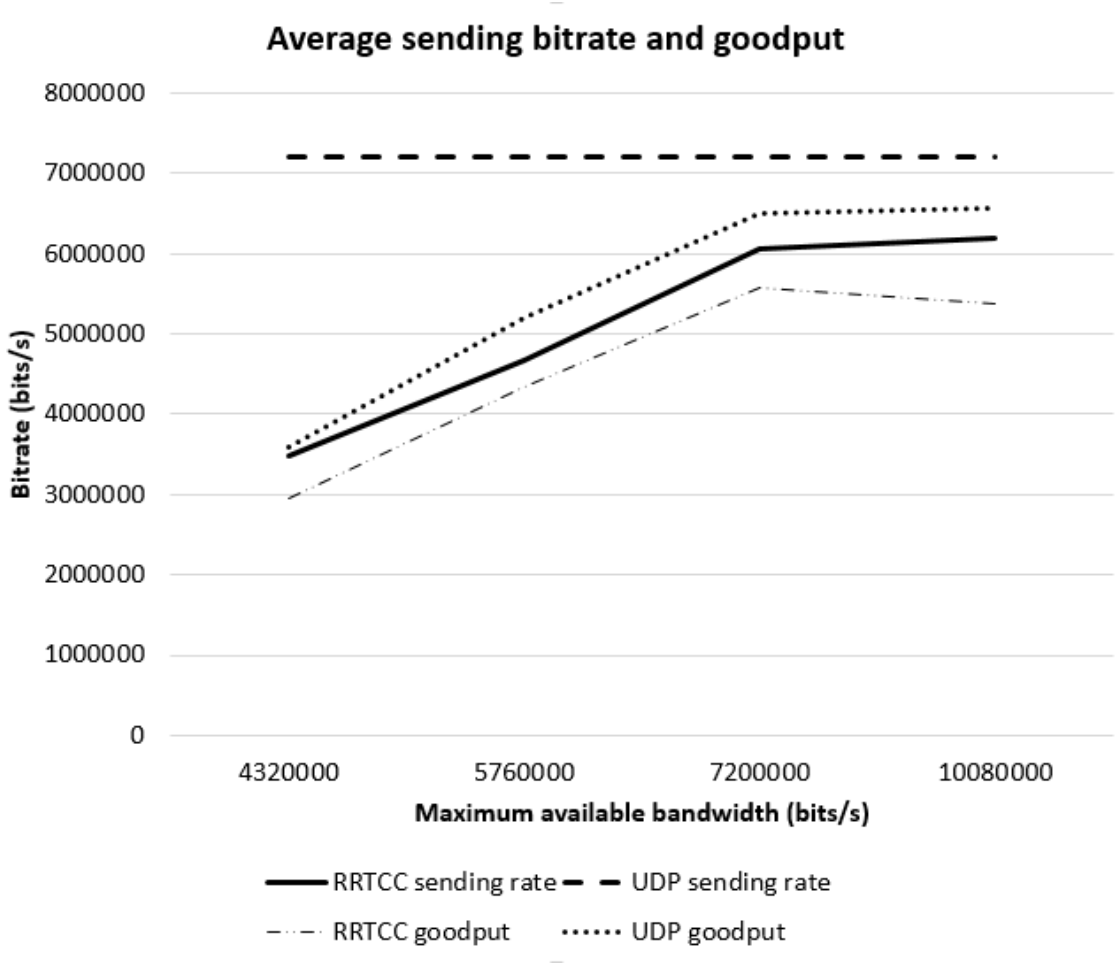

Figure 20 Average sending bitrate and goodput for different `network` configurations

Meanwhile:

- RRTCC sending rate: The average sending bitrate of sender using Receiver-side Real-time Congestion Controller.
- UDP sending rate: The average sending bitrate of sender using UDP Congestion Controller.
- RRTCC goodput: The average real bitrate that receiver receives when sender uses Receiver-side Real-time Congestion Controller.
- UDP goodput: The average real bitrate that receiver receives when sender uses UDP Congestion Controller.

As you can see from the chart above, the goodput lines in both cases are closer to the RRTCC sending rate line than the UDP sending rate line. And two goodput lines and the RRTCC sending rate line have the same trend according time. So sender using UDP congestion control sends data very fast to the `network` but the real bitrate that receiver receives is much lower than the sending rate. The goodput when sender using UDP congestion control is quite the same (a little bit higher) in comparison with RRTCC.

CONCLUSIONS

The final chapter is the conclusion and the future improvements for our simulation of Receiver-side Real-time Congestion Control algorithm.

5.1. Conclusions

The Receiver-side Real-time Congestion Control is a great algorithm for real-time transmission.

Our simulation has implemented RRTCC in a quite correct way and it shows the reasonable estimated bandwidth (sending rate) and loss ratio chart for real-time transmission. And its efficiency is quite good when it is compared with UDP Congestion Controller in small network bandwidth condition.

Our simulation is not only aiming at evaluating RRTCC but can hopefully be used as a framework for other congestion control algorithms. For example, ones can extend our simulation by adding TCP congestion controller or other mechanisms to inspect how they work with various scenarios and applications.

In addition, through our work process, we see SimPy is a good simulation framework to simulate a discrete-event network. It is really easy to use. On another hand, SimPy is written in Python, and Python is a clear and powerful object-oriented programming

language, so SimPy is very easy to understand and to apply to any discrete-event simulation.

5.2. Future Works

Our simulation only focuses on the first implemented way of RRTCC. In the future, we will try to deploy the second implemented way of RRTCC. Also, we believe that more thorough simulations and evaluations can be done and they may reveal interesting aspects of RRTCC in various situations. It is also our plan for future works.

References

[1] "How Real-Time Communications Is Empowering Business," [Online]. Available: https://www.tagove.com/real-time-communications-empowering-business/.

[2] "Net market share," [Online]. Available: https://netmarketshare.com/.

[3] Varun Singh, Albert Abello Lozano, Jorg Ott, "Performance Analysis of Receive-Side Real-Time Congestion Control for WebRTC".

[4] "SimPy Documentation," [Online]. Available: https://simpy.readthedocs.io/en/latest/.

[5] "Definition of real-time communication," [Online]. Available: http://searchunifiedcommunications.techtarget.com/definition/real-time-communications.

[6] S. Loreto and S. P. Romano, Real-Time Communication with WebRTC.

[7] Schulzrinne, H.; Casner, S.; Frederick, R.; Jacobson, V., "RTP: A Transport Protocol for Real-Time Applications," [Online]. Available: www.ietf.org_rfc_rfc3550.

[8] "WebRTC," [Online]. Available: https://webrtc.org.

[9] S. Dutton, "Getting Started with WebRTC," [Online]. Available: https://www.html5rocks.com/en/tutorials/webrtc/basics/.

[10] S. Holmer, H. Ludin, G. Carlucci, L. D. Cicco and S. Mascolo, "A Google Congestion Control Algorithm for Real-Time Communication," [Online]. Available: https://tools.ietf.org/html/draft-ietf-rmcat-gcc-02.

[11] S. Floyd, "Congestion Control Principles," [Online]. Available: https://tools.ietf.org/html/rfc2914.

[12] V. Singh, "Fairness of RRTCC".

[13] "NumPy," [Online]. Available: http://www.numpy.org/.

[14] "PyKalman," [Online]. Available: https://pykalman.github.io/.

Appendix A