ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

TIK Institut für Technische Informatik und Kommunikationsnetze

Remo Balaguer

TCP Congestion Control on RTP Media Streams

Semester Thesis
March 2014 to June 2014

Tutor: Brian Trammell
Supervisor: Prof. Dr. Bernhard Plattner

Abstract

Media streams are becoming increasingly more important in the everyday use of the Internet. The amount of real-time based services is growing while the underlying technology is still packet switched.

TCP congestion control in the network layer can suffer from imperfections like delay and loss in the OSI Layers 1 and 2. These can lead to severe performance decrease or even failure of media streaming services over TCP. The reason for the suboptimal performance of TCP congestion control lies in the fact that assumptions for the algorithms are violated frequently.

This thesis considers RTP media streams over TCP without RTCP, although RTP usually runs over UDP. The uncommon protocol stack with RTP over TCP without RTCP is motivated by the IPTV system of Swisscom which uses this exotic combination of protocols and shows some performance issues due to the TCP congestion control.

The result of this thesis is a simulation environment for RTP media streams over TCP without RTCP and the evaluation of different TCP congestion algorithms under various delay and loss conditions for that specific situation.

Acknowledgments

In the first place I would like to thank Prof. Dr. Bernhard Plattner for providing the academic environment for this work. Also, I wish to express great thanks to my tutor Brian Trammell for his constant support.

I express my gratitude to Remo Niedermann, ICT Security consultant at Swisscom, who helped me finding the project and the specific contact person Dario Vieceli, Lead of IPTV and OTT Software Development at Swisscom, whom I'd like to thank for the idea and support of this thesis.

Contents

Chapter 1

Introduction

The Internet technology has revolutionized society and science during the past decades. The ability to send datagrams from one host to another has led to an enormous amount of useful services. Most of which did not have rigorous timing requirements. But as the bandwidth increased and the machines got faster, people also started to use the Internet for real time media transport, which essentially is not what a packet switched network was designed for in the first place. Nevertheless, the Internet has become a very important infrastructure for the transport of media streams (e.g. VoIP, Videoconferencing and (High Definition) IPTV) because of the economic advantages. The universal utilization of the network prevents the need for secondary infrastructure but at the same time gives rise to new challenges.

The requirements for a media transport network seem to be simple at first sight. Since good audio and video quality is very much appreciated, the bandwidth of a network, that should be able to transport media streams, has to be large enough. A typical bandwidth of 10Mbit/s or more is essential. Also, low delay is crucial for human conversations. A typical one-way latency of 150ms or less is needed in VoIP. But these two things are not sufficient.

Since there are usually multiple nodes between two machines, a packet switched computer network can suffer congestion collapse. A congestion collapse is the state of the network when communication is barely possible due to overfilled buffers in the nodes. This happens when incoming traffic exceeds the outgoing bandwidth for a specific node. The consequence is bufferbloat or the drop of packets. Without congestion control the sender would just retransmit the packets that are dropped which unfortunately would overload the node even more. In the end, the network would collapse due to the positive feedback between packet loss and retransmission. In order to prevent situations like that, TCP comprises congestion control that runs an algorithm on the server-side and pursues the objective to find the optimal sending rate without overloading nodes between the sender and the receiver.

In order for the sender to find out whether a TCP packet has reached the destination, the receiver sends an acknowledgement (ACK) back to the server. Of course, the server could wait for an ACK after each packet that has been sent, but this would be terribly inefficient. So as to use the capacity of the transmission line (delay is non-zero) more efficiently, the server sends multiple packets at once.

The amount of unacknowledged packets on the sender side is called sending window. The sender takes the minimum between the advertised window and the congestion window as a sending window. The TCP congestion control algorithm determines the size of the congestion window (cwnd). For the sake of completeness, it has to be mentioned that the receiver advertises the so-called receiver window back to the sender. This mechanism enables the receiver to inform the sender when it is overloaded by incoming data.

So as to adjust the congestion window, the sender has to probe the transmission line. The size of cwnd is being increased until congestion is detected. Knowing the size of the cwnd when congestion is detected enables a congestion control algorithm to act in a smart way (e.g. decrease cwnd in a certain way for a specific amount of time). But first it has to be able to find out whether congestion happens. In general, there are two ways to detect congestion when explicit congestion notification (ECN) is not available. The first class of algorithms assumes that packets are dropped due to overfilled queues when congestion happens, therefore "loss-based" algorithms like *New Reno* take packet losses as an indicator for congestion. The second class

of algorithms assumes that the delay of a line changes when congestion is about to happen because of buffers getting filled up. Therefore "delay-based" algorithms like *Vegas* take delay variations as an indicator or predictor for congestion. The third class of algorithms is "hybrid" like *Illinois* because they make use of loss and delay information at the same time.
The difficulty with all of those algorithms lies in the fact that losses and delay variations can not only originate from congestion but also from imperfections in physical layers (e.g. DSL lines, wireless and mobile networks). That is why assumptions of the congestion control algorithms are frequently violated and the sender may misinterpret a loss or delay signal as a sign for congestion. This leads to an unneeded decrease of the cwnd and causes a performance loss. If the physical layer for whatever reasons drops some packets or produces variations in delay while congestion is not the origin, the congestion algorithm cannot necessarily improve the situation by decreasing the sending rate. Therefore, current congestion control can under some circumstances unnecessarily reduce the efficiency of a transmission line.
It should be mentioned that, additionally to the performance, also fairness between different TCP sessions and different congestion control algorithms should be preserved, which is a challenge when designing new algorithms.
A very nice overview over the common congestion algorithms is the IEEE Survey: "Host-to-Host Congestion Control for TCP" [1]

Chapter 2

The Problem

2.1 Motivation

The motivation to write this thesis lies in several meetings with Dario Vieceli, Swisscom IPTV Lead for IPTV/OTT Software Development. He points out the potential for improvement in the area of TCP congestion control.
Swisscom IPTV with over 1 million customers is wide-spread in Switzerland and runs over the same DSL (just recently also over fiber) as the non-IPTV traffic. The IPTV media streams are RTP over TCP (without RTCP). TCP was given preference over UDP for reliability reasons meaning that no router or middlebox of the end user network blocks or impedes the media stream traffic.
Also RTCP was not implemented in order to avoid the presence of two congestion control mechanisms at the same time. Hence RTP is used for synchronization and framing while the congestion control part is taken over by TCP. It has to be mentioned that this combination of protocols is rather exotic and is not the standard for RTP. Normally RTP runs over UDP and lets RTCP control the transmission speed which is aware of the incoming media stream bitrate and can adapt to it.
Dario Vieceli mentions the suboptimal situation of the congestion algorithms struggling with last mile loss characteristics like random loss and burst loss. Performance issues like stalls in the video playback are observed. These problems originate in the fact that the physical link is not perfect. Current TCP congestion control algorithms do not seem to be able to cope well with this. Therefore, some further investigations on this problem are advisable.

2.2 The Task

This semester thesis comprises the creation of a simulation environment with the purpose of understanding the specific problem. Therefore, a test network has to be set up in order to evaluate a couple of currently-implemented CC algorithms under a set of loss and delay profiles that are close to the situation in the above mentioned case. The emulation tool *Mininet* is used as an emulation environment and also an appropriate lab setup with real machines should be built up. Different TCP congestion control algorithms should then be tested at different random loss and delay conditions. Additionally a suitable media server has to be found and configured since the original server is an appliance and is not available.
The outcomes are instructions of how to build the lab setup and a comparison between the different congestion control algorithms based on measurements in the lab. A short insight into the analysis of the real IPTV system stream is provided as well as a comparison between TCP bulk data transfer and media stream transfer.

Chapter 3

Simulation Methodology

In the interest of understanding the problem of TCP congestion control on RTP media streams and evaluating different congestion control algorithms, a simulation/emulation environment is inevitable.
The access to the IPTV network of Swisscom was obviously not possible during this project. There are basically three different approaches when trying to get an insight into networks that cannot actually be accessed. The first option is to simulate the network, the second one is to emulate the network on one machine and the third alternative is to actually connect a couple of real machines and setup a network in the lab. A disadvantage of all these techniques is that the networks built are just models of reality and may not always reflect it perfectly.
In this thesis there is an attempt to emulate the network but performance issues lead to the third alternative with real machines.

3.1 Emulation

This section presents the installation and use of network emulation software that is utilized during this project. Network emulation provides a neat way to experiment with various network topologies and different lines between nodes. The ability to control the whole emulated network centrally is a huge advantage in the process of experimenting. A well known emulation environment is called *Mininet*.

3.1.1 Mininet

What is it?

Mininet is a network emulator that can run multiple end-hosts on a single Linux kernel. Various network topologies with different switches, routers and links can be created. Once a topology is set up, each element of it runs the same kernel. Links can be set up at arbitrary bandwidth, delay and loss. Furthermore each host in *Mininet* behaves just like a real machine. One can log in via *ssh* or run any programs of the underlying Linux system. Also the network interfaces look like a real ones for the host. [2]

Installation

The easiest way to use *Mininet* is the VM Installation. [3]

- Install Virtual Box (or another virtualization software)
- Download the *Mininet* VM Image from `https://bitbucket.org/mininet/mininet-vm-images/downloads`
- Create a new Linux VM in *VirtualBox* and choose the downloaded .vmdk image as the virtual hard disk
- Start the VM and log in (login and password are `mininet`)

- In order to see if both interfaces eth0 and eth1 have IP addresses assigned run:
  `$ ifconfig -a`
  If there are interfaces without any IP assigned run:
  `$ sudo dhclient ethX`
  for the interfaces in question. Remember the assigned IP for the next step.
- From your host OS log into the *Mininet* VM with:
  `$ ssh -X mininet@[IP of VM]`
  and enter the password `mininet`
- A simple topology with two nodes can be created via:
  `$ sudo mn`
- You should now be able to open a new terminal via SSH:
  `$ xterm`
- Every time you have finished the experiments run the following to exit and clean up:
  `mininet> exit`
  `$ sudo mn -c`

Run a simple emulation [4]

The following steps demonstrate how to set up a simple network with two end hosts and one interconnecting line. The line can be specified by bandwidth, delay and loss. A network like this can serve as the base of experiments with media streams.

- Create a network with two nodes and a link (bandwidth 10Mbps and a delay of 10ms):
  `sudo mn --link tc,bw=10,delay=10ms`
- You can now open terminals for the two nodes using:
  `$ xterm h1 h2`
  and run some programs
- You can also use iperf or ping in the *Mininet* console to test the network:
  `mininet> iperf`
  `mininet> h1 ping -c10 h2`
- A nice overview of the *Mininet* usage can be found on
  `http://mininet.org/walkthrough`

For the purpose of investigating on the network traffic *Wireshark* or *tcpdump* can be started from the *Mininet* console at any time.

Run a simple RTP over UDP streaming session in *Mininet*

A first step towards streaming over the network is the application of *ffserver* which is part of the *ffmpeg Project* [12] and provides a great number of different streaming functionalities. Although it does not run over TCP without RTCP, it produces traffic that is more than just bulk data transfer. The following steps explain how to establish such a stream.

- Install *ffserver* on *Mininet*
- Change directory to `/home/mininet/mininet/custom/`
- Put a MPEG-1 file *my_movie.mpg* to `/home/mininet/mininet/custom/`
- Create a simple topology:
  `$ sudo mn`
- Start a network capture in a *Mininet* console for later evaluation:
  `$ sudo tcpdump -ni s1-eth1 -w filmcap.pcap -s 128`
- Open two xterm terminals for the two nodes:
  `$ xterm h1 h2`
- On host 1 run:
  `ffserver -f ffserver-example.conf` with the content:

```
#################################################
Port 8090
BindAddress 0.0.0.0
MaxHTTPConnections 2000
MaxClients 1000
MaxBandwidth 500000
CustomLog -
NoDaemon

RTSPPort 7654
RTSPBindAddress 0.0.0.0

<Stream test1-rtsp>
    Format rtp
    File "/home/mininet/mininet/custom/my_movie.mpg"
</Stream>
#################################################
```

- On host 2 run:
  ffplay rtsp://[IP of host 1]:7654/test1-rtsp

Mininet Timing

Since delay is crucial for TCP throughput, because it determines how fast the sender can react to the network condition, it is obvious that an emulation environment for TCP congestion control evaluation should be stable with respect to timing. *Mininet* is often mentioned in the context of Software Defined Networks and not in relation to TCP congestion control performance. This leads to the idea of checking the timing performance of *Mininet* with a simple file transfer experiment. It should show if the simulated line and the hosts act realistically by looking at the behavior of RTT over time. As performance may be limited due to the hardware in use, the specs of the test machine (TIK Laptop) are indicated:

```
Manufacturer: LENOVO
Version: ThinkPad T410
CPU: Intel(R) Core(TM) i7 CPU M 620 @ 2.67GHz
Memory: DDR3 4GiB
OS: Ubuntu 12.04 LTS 64-bit with 3.2.0-61 kernel
Virtualization software: Virtual Box
Mininet Version: 2.1.0 with 3.8.0-19 kernel
```

The following python script for *Mininet* (motivated from the *Mininet* Introduction [2]) is used to produce RTT measurements:

```
#!/usr/bin/python

from mininet.topo import Topo
from mininet.net import Mininet
from mininet.node import CPULimitedHost
from mininet.link import TCLink
from mininet.util import dumpNodeConnections
from mininet.log import setLogLevel
import os
from time import sleep

class SingleSwitchTopo(Topo):
    "Single␣switch␣connected␣to␣n␣hosts."
    def __init__(self, n=2, **opts):
        Topo.__init__(self, **opts)
        switch = self.addSwitch('s1')
        """print 'n  esch: ' + str(n)
        print 'range esch: ' + str(range(n))"""
        for h in range(n):
            # Each host gets 50%/n of system CPU
            host = self.addHost('h%s' % (h + 1),
               cpu=.5/n)
            # Choose in bw Mbps, delay ms delay, loss% loss, 1000 packet queue
            self.addLink(host, switch,
               bw=10, delay='10ms', loss=0, use_htb=True)

def perfTest():
    "Create␣network␣and␣run␣simple␣performance␣test"
    os.system( "sudo␣mn␣-c")
    # set tcp congestion control
    os.system( "sudo␣/sbin/sysctl␣-w␣net.ipv4.tcp_congestion_control=reno")
    # also cubic or illinois possible
    topo = SingleSwitchTopo(n=2)
    # create network topology
    net = Mininet(topo=topo, host=CPULimitedHost, link=TCLink)
    net.start()
    print "Dumping␣host␣connections"
    dumpNodeConnections(net.hosts)
    print "Testing␣network␣connectivity"
    net.pingAll()
    print "Do␣the␣http␣file␣transfer"
    h1, h2 = net.get('h1', 'h2')
    # start the server
    re=h1.cmd('cd␣randdata;␣nohup␣python␣-m␣SimpleHTTPServer␣80␣&')
    sleep(1)
    print re
    # start the capture
    h2.cmd('sudo␣tcpdump␣-ni␣h2-eth0␣-w␣h2-bw50-100ms.pcap␣-s␣128␣&')
    sleep(1)
    # start the download
    ri=h2.cmd('wget␣-r␣10.0.0.1')
    print ri
    h2.cmd('rm␣-r␣10.0.0.1')
    # exit experiment
    net.stop()


if __name__ == '__main__':
    setLogLevel('info')
    perfTest()
```

With the help of *tcptrace* which is introduced in subsection 4.1.2 the RTT behavior over time can be obtained from the network capture file:

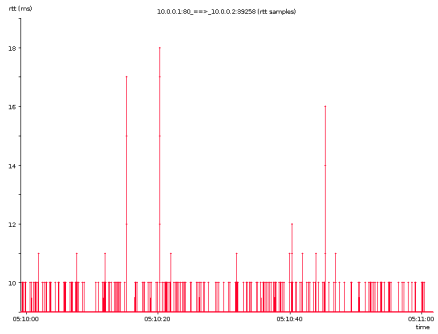

Fig. 1) RTT behavior for a TCP bulk data transfer on a line with 10ms delay and 10Mbps bandwidth

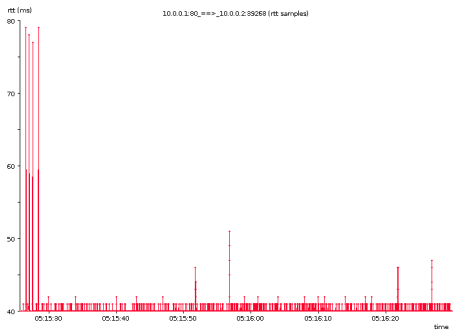

Fig. 2) RTT behavior for a TCP bulk data transfer on a line with 40ms delay and 10Mbps bandwidth

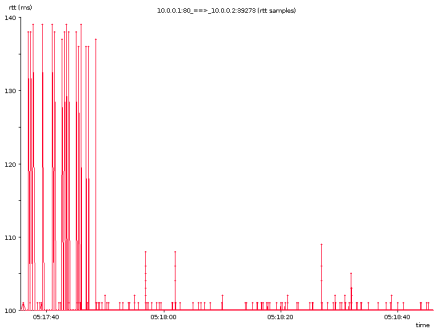

Fig. 3) RTT behavior for a TCP bulk data transfer on a line with 100ms delay and 10Mbps bandwidth

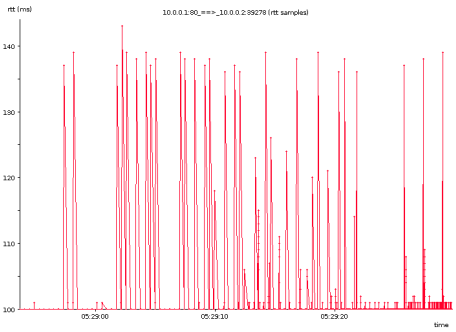

Fig. 4) RTT behavior for a TCP bulk data transfer on a line with 100ms delay and 50Mbps bandwidth

The plots show that RTT is not stable at all. There are periods of time where the RTT is much larger than it should be. Especially for larger delays and bandwidths, the variation of RTT becomes striking. Also a repetition of the experiment and turning of the HTB (Hierarchical Token Bucket) in *Mininet* did not change the unpleasant situation.
For this thesis it is considered as insufficient for experiments with media streams over TCP because TCP performance depends a lot on RTT. The causes lie probably in some context switches in *Mininet* or the OS of the host machine.
In order to have a more realistic network, a lab setup with real machines needs to be created.

3.2 Lab setup

This section presents the installation and use of a test network in the lab. The reason for using separate Linux Boxes is the need to have hosts that act as realistic as possible with respect to timing and TCP performance. Although the media server and client can be the same as in the emulation setup, one Linux Box has to act as a line simulator. The fact that three machines have to be set up and configured causes more effort than a simple emulation on one single machine. However, an advantage of a setup like this is that a real IPTV Box can also be tested under additional delay and loss introduced by the line simulator. Furthermore it can be used to capture the traffic between the box and the LAN.

3.2.1 Topology overview

The lab setup comprises 3 Linux boxes, the first of which acts as a media server, the second as a line simulator (delay and loss) and the third as a client. The following sections present a possible implementation and configuration of such a network using *WANem* as a line simulator.

3.2.2 Line Simulator with WANem

What is WANem?

WANem (Wide Area Network Emulator) is an Open Source software licensed under the GNU General Public License. It can be used to simulate WAN characteristics like delay, packet loss, packet corruption, disconnections, packet re-ordering, jitter and so on.
WANem consists of a re-mastered Knoppix and hooks into the Linux kernel towards provisioning the network emulation characteristics. Additionally, it extends the functionality with additional modules. An intuitive web interface allows configuring. [5]

Why WANem?

It may be asked why *WANem* is preferred over the *netem* tool in Linux that can also introduce delay and loss. The reason for using *WANem* in this thesis is that it shows a very low minimal delay (1ms) while *netem* on the specific Ubuntu test machine with two bridged interfaces shows varying minimal delays up to 100ms. The reason for this strange behavior can not be found during weeks. Also *WANem* shows a very high maximal bandwidth at 500Mbps while *netem* shows only bandwidths up to 10Mbps. It may be problems of the specific hardware and OS version of the test machine that lead to the issues with *netem*. But there are also other reasons for using *WANem*. It is capable of introducing burst losses and has a very nice configuration interface that allows for complicated configuration of line simulation in short time.

Installation and configuration

WANem can be booted from a live CD. The Iso file is available on `http://wanem.sourceforge.net`.
In order to simulate a line, two interfaces of the machine have to be bridged:

- Open a LXTerminal and enter:
  `$ exit2shell`
- Open a the interfaces configuration file:
  `$ leafpad /etc/network/interfaces`
- Change file to:

```
auto br0
iface br0 inet static
    address 10.0.0.8
    netmask 255.255.255.0
    gateway 10.0.0.1
    bridge_ports ethX ethY
    bridge_fd 0
    bridge_stp off
```

- Restart networking using:
  `$ /etc/init.d/networking restart`

Configure the line characteristics

A web-browser with a configuration GUI is started by default in *WANem*. There you can set up the line characteristics for the two interfaces that are bridged.

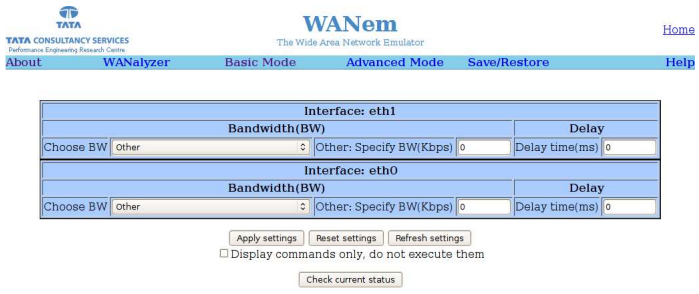

Fig. 1) *WANem* Basic Configuration Web-interface GUI

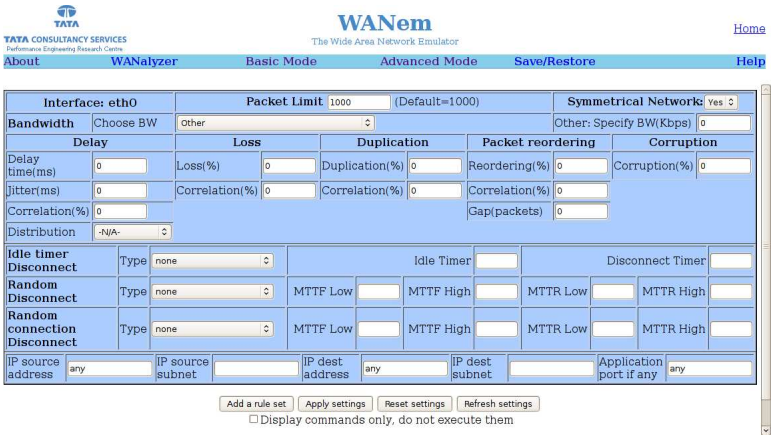

Fig. 2) *WANem* Advanced Configuration Web-interface GUI

3.2.3 Host configuration

Before the server and client software can be started, the interfaces have to be configured to provide network connectivity. The following steps describe the configuration of the server:

- Stop the network-manager:
  ```
  $ sudo service network-manager stop
  ```
- Open a the interfaces configuration file:
  ```
  $ gedit /etc/network/interfaces
  ```
- Change file to:
  ```
  auto lo
  iface lo inet loopback
  ```
- Restart networking:
  ```
  $ /etc/init.d/networking restart
  ```
- Configure the interface as follows:
  ```
  $ ifconfig eth0 10.0.0.1
  $ ifconfig eth0 netmask 255.255.255.0
  $ route add default gw 10.0.0.8
  ```

The interfaces of the client can be configured in the same manner but with another IP:

- Configure the interface as follows:
  ```
  $ ifconfig eth0 10.0.0.2
  $ ifconfig eth0 netmask 255.255.255.0
  $ route add default gw 10.0.0.8
  ```

3.3 Streaming

The following section presents steps and options to perform media streaming with different TCP congestion control algorithms over the network.

3.3.1 Selecting the TCP Congestion Control Algorithm

No matter which streaming server is used, on Linux the TCP congestion control algorithm can be set in the system as follows:

```
sudo /sbin/sysctl -w net.ipv4.tcp_congestion_control=[algorithm]
```

where `[algorithm]` stands for `reno` , `cubic` , `illinois` , etc.

3.3.2 ffserver

ffserver is a streaming server for audio and video. It supports a variety of formats and protocols.

Installation

Since ffserver is part of ffmpeg the latter has to be installed. On Ubuntu for example this can be done by the following command:

```
$ sudo apt-get install ffmpeg
```

Run a stream

The following command starts a streaming server based on the configuration file below:
`$ ffserver -f ffserver-example.conf`

with a configuration file like:

```
Port 8090
BindAddress 0.0.0.0
MaxHTTPConnections 2000
MaxClients 1000
MaxBandwidth 500000
CustomLog -
NoDaemon

RTSPPort 7654
RTSPBindAddress 0.0.0.0

<Stream test1-rtsp>
    Format rtp
    File "/home/mininet/mininet/custom/[your file].mpg"
</Stream>
```

The stream is now accessible via:
`$ ffplay rtsp://[IP of server]:7654/test1-rtsp` (RTP over UDP)
`$ ffplay rtsp://[IP of server]:7654/test1-rtsp?tcp` (for pure TCP)

Issues

Since RTP over TCP is needed and the request for TCP by ffplay does not initiate a RTP over TCP but a pure TCP transmission, ffserver does not seem to be the appropriate tool for this project.

3.3.3 VLC

The second tool used for streaming is VLC.

Installation

The VLC media player is preinstalled in many Linux distributions and can otherwise easily be installed using the package manager.

Send a stream

- Open *VLC media player* on Streaming Host
- *Media -> Stream...*
- *Add...* -> (select the ts (MPEG transport stream) file)
- *Stream*
- *Next*
- *New destination -> RTSP*
- *Add* and enter the <name of the media> and remember the port
- Deactivate "Active Transcoding"
- *Next*
- Select "Stream all elementary streams"
- *Stream*

Receive a stream

The stream can be played by ffplay:
`$ rtsp://<sender ip>:<port>/<name of the media>`

or by VLC:

- Open VLC on Client
- *Media -> Open Network Stream...*
- Enter the URL: `rtsp://<sender ip>:<port>/<name of the media>`

VLC as root issue

There might be an issue in *Mininet* because the *VLC media player* opens under root there, which causes the following message:
`VLC is not supposed to be run as root. Sorry.`
An easy way to solve that problem is:

- Open the file:
  `$ vi /usr/bin/vlc`
- replace `geteuid` with `getppid` and save

Issues

VLC is not able to send the stream over TCP when a client requests that. When requesting TCP instead of UDP as the transport protocol with "`?tcp`" at the end of the URL for ffplay it says:
`method SETUP failed: 461 Unsupported transport, Protocol not supported`
Also with openRTSP as client and the option "`-t`" it says that this transport protocol is unsupported.

3.3.4 live555 Media Server

For the purpose of using the exotic protocol stack with RTP over TCP without RTCP, a custom server is necessary. *live555 Media Server* comprises C++ libraries for multimedia streaming, using open standard protocols (RTP/RTCP, RTSP, SIP, etc.). [6]
In order to turn off the RTCP, a patch is necessary.

Patch and Compilation

- Get source code *live.2014.04.23.tar.gz* from /urlhttp://www.live555.com/liveMedia/public/
- Unpack the tar file
- Open the *RTSPClient.cpp* in the folder *live/liveMedia*
- Since the sender should not be able to react to the networks condition via RTCP, the *receiver reports* should be disabled completely. In fact all appearances of `"enableRTCPReports() = ...;"` on the code lines 1107, 1154 and 1176 should be set to `"False"` which can be done manually or with the following commands:

  – Make a copy of RTSPClient.cpp
  `$ cp RTSPClient.cpp RTSPClientOriginal.cpp;`
  – Patch the file
  `$ patch RTSPClientOriginal.cpp -i turnoffRR.patch -o RTSPClient.cpp;`
  with the turnoffRR.patch file:

```
1154c1154
<       if (subsession->rtpSource() != NULL) subsession->rtpSource()->enableRTCPReports
     () = True; // start sending RTCP "RR"s now
---
>       if (subsession->rtpSource() != NULL) subsession->rtpSource()->enableRTCPReports
     () = False; // start sending RTCP "RR"s now
1176c1176
<       if (subsession.rtpSource() != NULL) subsession.rtpSource()->enableRTCPReports()
     = True; // start sending RTCP "RR"s now
---
>       if (subsession.rtpSource() != NULL) subsession.rtpSource()->enableRTCPReports()
     = False; // start sending RTCP "RR"s now
```

- Go back to the top directory called *live* and run `./genMakefiles linux`
- Run `make`
- Now the binaries can be used

Send a stream

In the *live/testProgs* folder there are a lot of test applications. The program *testOnDemandRTSPServer* can stream all sorts of media files that are in the same directory. Therefore the media file just as to be moved into the same directory as the *testOnDemandRTSPServer* and the server can be started. The application shows the URL under which the stream can be received:

```
$ ./testOnDemandRTSPServer

"mpeg4ESVideoTest" stream, from the file "test.m4e"
Play this stream using the URL "rtsp://10.0.0.1:8554/mpeg4ESVideoTest"

"h264ESVideoTest" stream, from the file "test.264"
Play this stream using the URL "rtsp://10.0.0.1:8554/h264ESVideoTest"

"h265ESVideoTest" stream, from the file "test.265"
Play this stream using the URL "rtsp://10.0.0.1:8554/h265ESVideoTest"

"mpeg1or2AudioVideoTest" stream, from the file "test.mpg"
Play this stream using the URL "rtsp://10.0.0.1:8554/mpeg1or2AudioVideoTest"

"mpeg1or2ESVideoTest" stream, from the file "testv.mpg"
Play this stream using the URL "rtsp://10.0.0.1:8554/mpeg1or2ESVideoTest"

"mp3AudioTest" stream, from the file "test.mp3"
Play this stream using the URL "rtsp://10.0.0.1:8554/mp3AudioTest"

and so on...
```

Receive a stream

A neat tool to receive the stream without wasting CPU power is openRTSP since it does not display the movie on the screen. This tool is included within *live555* and should be used because of the disabled receiver reports.
Further it can request TCP with the option "`-t`":
`$ ./openRTSP -t [URL indicated by testOnDemandRTSPServer]`

Issues

The advantage of this media server is that it can be tweaked to use the special protocol stack. The downside is the inability to get stall information of the video playback by the client *openRTSP*.

3.4 How to rip a movie file from the IPTV Box to a ts file

The following steps describe how to extract a MPEG Transport Stream (*.ts file) from the IPTV Box out of a network capture file.

- Start capturing the traffic with a bridge that is connected to the IPTV Box before (!) starting the playback on the Box
- Open the capture file in *Wireshark*
- *Telephony -> RTP -> Stream Analysis*
- Select the stream and save it as *.ts file

Hint: `$ ffprobe [file]` is a handy tool to find out details about a media file.

Chapter 4

Analysis of Measurements

4.1 Analysis Tools

4.1.1 Wireshark

Besides *tcpdump* also *Wireshark* [7] can be used to capture network traffic. Further *Wireshark* provides a lot analysis. For this thesis the following parts were important:

- Filter for different protocols (e.g. RTP, RTCP, TCP,...)
- Extract the movie file from a network traffic capture
- Plot I/O statistics for a rough picture of the traffic

4.1.2 Tcptrace

tcptrace [8] can be downloaded at `http://www.tcptrace.org/download.html`. It is well suited to get plots and statistics about TCP traffic from a capture file. In this thesis the following functionalities is used:

- Time Sequence Graph:
  `tcptrace -S [capture file]`
  (jPlot [10] is recommended to export the plots)
- General statistics like the number of *SACK* (selective acknowledgement) packets seen or the average outstanding window:
  `$ tcptrace -lW [capture file]`.

4.1.3 Tcpcsm

tcpcsm [9] can be downloaded at `http://www.wand.net.nz/~salcock/tcpcsm/`. It is well suited for estimating the cwnd of TCP traffic from a capture file. Since the a TCP congestion control algorithm does essentially control the size of cwnd, it makes sense to get this quantity when trying to evaluate the performance of a specific algorithm. The following command estimates the size of cwnd:

```
flight_cwnd [capture file] -R -S -q -o [output file]
```

A self-made python script is used to plot the cwnd from the output text file.

```
#!/usr/bin/python

import pylab
import numpy as np
import matplotlib.pyplot as plt
import os

# get the pcap file name, run filght_cwnd
txtfile = str.format(raw_input("Please␣enter␣a␣file:␣\n"))
os.system("flight_cwnd␣"+txtfile+".pcap␣-R␣-S␣-q␣-o␣"+txtfile+".txt")

# import to python
fh  = open(txtfile+'.txt').read()

# allocate
the_list = []
x=[]
y=[]

# data import similar to http://stackoverflow.com/questions/9746927/python-data-import
for line in fh.split('\n'):
    print line.strip()
    splits = line.split()
    if  len(splits) ==1 and splits[0]== line.strip():
        splits = line.strip().split(',')
    if splits:the_list.append(splits)

for i in range(len(the_list)):
    print the_list[i]
    if  the_list[i][-1]=='':
        the_list[i].pop(-1)
        the_list[i].extend(the_list[i+1])
        i += 1

# get the start time
starttime=the_list[1][7]

# cut out cwnd data (each row is another timestamp
# ignore the last 10 rows in order to not mess up the plot with data that does not belong to the cwnd information
for k in range(len(the_list)-10):
        x.append(float(the_list[k][7])-float(starttime))
        y.append(float(the_list[k][10]))

fig = plt.figure()
ax1 = fig.add_subplot(111)
ax1.set_title("Wand␣flight␣cwnd␣estimation")
ax1.set_xlabel('Time␣Stamp')
ax1.plot(x,y, c='r')
leg = ax1.legend()
plt.show()
```

4.2 User experience performance of the IPTV Box

The stalls visible to the unaided eye during one minute of TV watching are counted. This is done for 15 different loss and delay profiles. The results are shown in the table below. The numbers indicate the minimum of noticed stalls.

0.5% loss	0	0	2	3	3
0.1% loss	0	0	2	2	3
0% loss	0	0	0	0	0
	0ms	40ms	80ms	120ms	160ms

Table 4.1: Minimum number of Stalls

It would be nice to have a comparison to the lab setup. However the *openRTSP* client does not display the video, hence stalls cannot be seen by the unaided eye. Although there is the `"-Q"` option that provides some QOS information, it unfortunately does not report video playback stalls. Another approach was to rip the received stream again with openRTSP using the options `"-i"` for a AVI file or `"-4"` for a MP4 file. Sadly neither *VLC* nor *ffplay* nor the *Totem Movie Player* could display the movie. Therefore it was impossible with the current setup to get a comparison.

4.3 IPTV box congestion analysis

The following analysis plots are based upon one-minute long TV streaming sessions. The *WANem* Simulator is placed between the TIK Internet Gateway and the IPTV Box.
The Time Sequence Graph plots the sequence number in black and the receive window advertised from the other endpoint in blue. The X-axis represents time whereas the Y-axis depicts the sequence number. The Time Sequence Graph therefore shows the progression of the transmission where the slope is related to the transmission speed.
With long delays the *slow start* of the congestion control algorithm can be clearly seen. It causes a relatively long time to speed up the transmission.

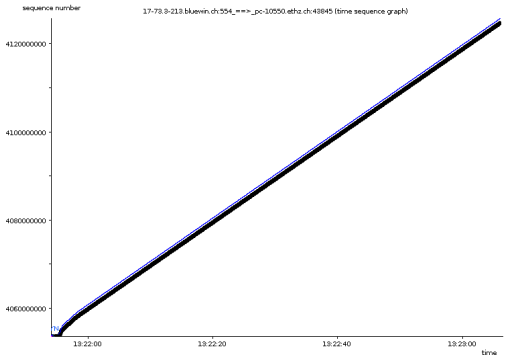

Fig. 1) IPTV Box with 120ms delay and 0% loss

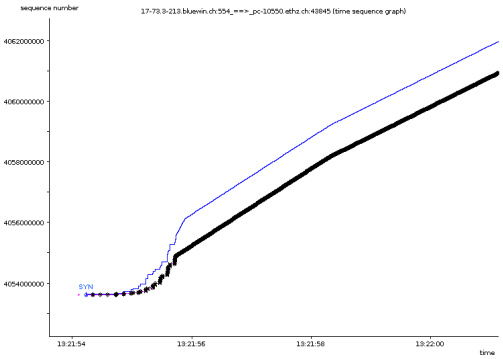

Fig. 2) A close-up view of Fig. 1

It can be seen that SACK blocks, which are marked in green, are found more often as the loss increases.

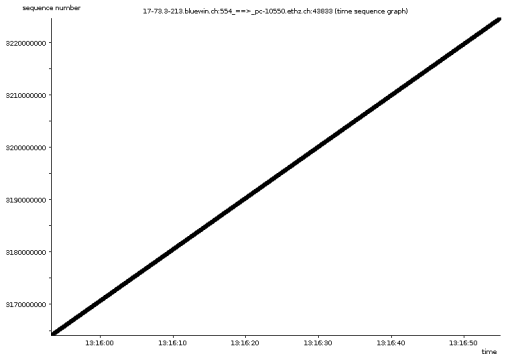

Fig. 3) IPTV Box with 0ms delay and 0% loss

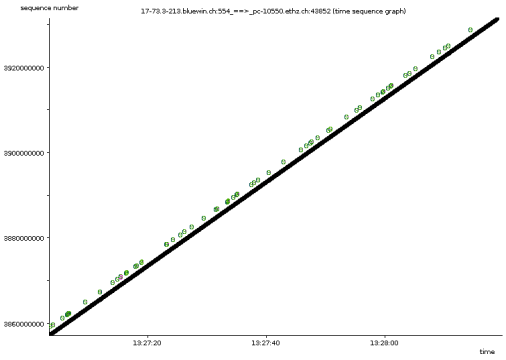

Fig. 4) IPTV Box with 0ms delay and 0.1% loss

When zooming in, the flights can be seen. The black up and down arrows represent the sequence numbers of the last and first bytes of the segment respectively. The blue line represents the advertised window and the magenta line keeps track of the ACK values.

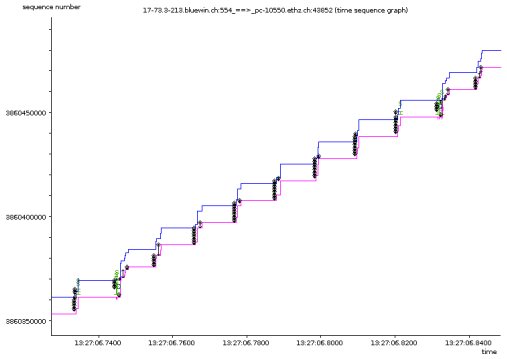

Fig. 5) A close-up view of Fig. 4

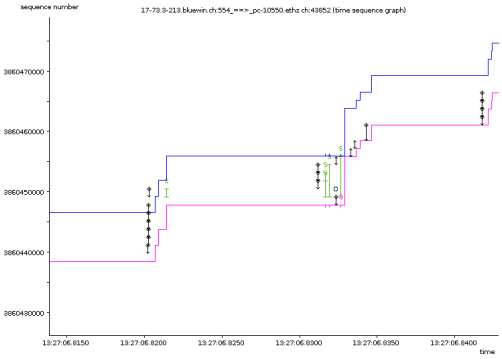

Fig. 6) A close-up view of Fig. 5

When delay and loss increase, dips can be seen in the Time Sequence Graph as well as variable flight durations that look congestion-like.
Generally, it looks like *Illinois* because it seems that there is a lot of feedback. This can be seen since it finds an optimal sending rate (straight line in the Time Sequence Graph) after some time and it does not increase the rate very fast after a loss (as for example *Cubic* would do).

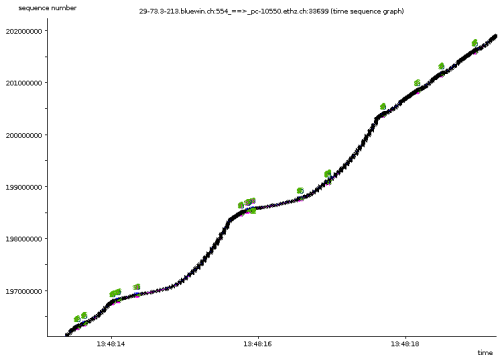

Fig. 7) IPTV Box with 40ms delay and 0.5% loss

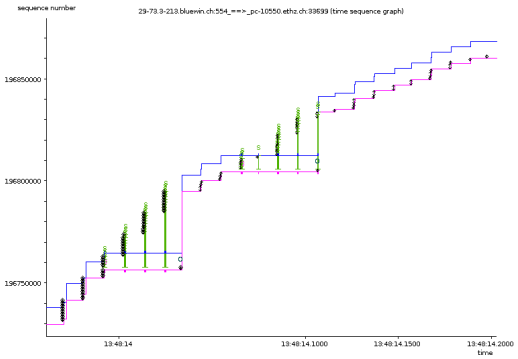

Fig. 8) A close-up view of the Fig. 7

The knowledge of cwnd would provide an even better insight into the way TCP congestion control performs since it is the manipulated variable of the algorithm. However the measurements can only be taken next to the client and it does not make sense to use *tcpcsm* in that case, because the distance to the sender would be too large for a reasonable estimation. If there was access to the server, it would be feasible.

4.4 Different Congestion Control on RTP over TCP

The following analysis plots are based upon a network capture between a machine with the *live555 Media Server* and a machine with the openRTSP client. The *WANem* line simulator machine is placed between the two. An approximately one-minute long video, that was originally captured on the IPTV box, was streamed.

4.4.1 New Reno characteristics

Without any additional delay or loss the Time Sequence Graph looks perfectly straight. The maximum data rate is found very quickly. As soon as there is delay (80ms here) the *slow start* is clearly visible.

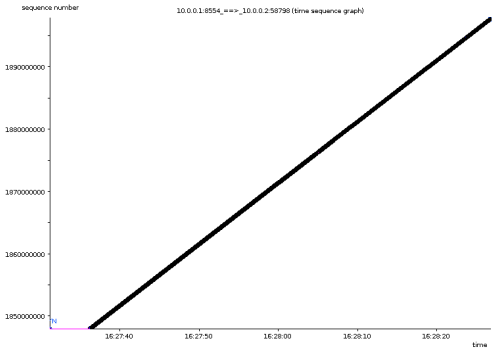

Fig. 1) live555 Server with 0ms delay and 0% loss

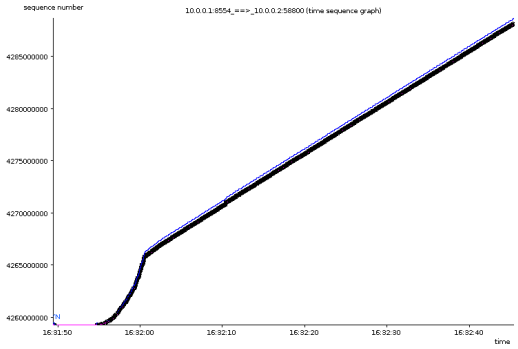

Fig. 2) live555 Server with 80ms delay and 0% loss

When delay and loss increase SACK blocks and dips in the sequence number line can be seen.

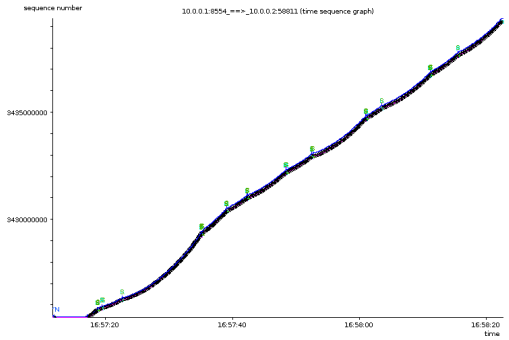

Fig. 3) live555 Server with 120ms delay and 0.5% loss

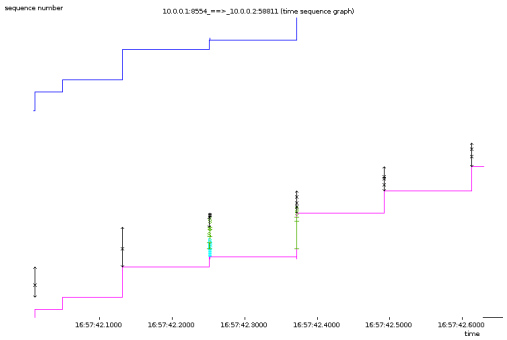

Fig. 4) A close-up view of the Fig. 3

Cyan arrows represent retransmitted segments with the up and down arrows representing the sequence numbers of the last and first bytes of the segment.

4.4.2 Comparison of different congestion control algorithms on live555 RTP over TCP without RTCP media streams

For the comparison of different TCP congestion control algorithms in the case of media streaming via RTP over TCP without RTCP, there was a movie file captured on the real IPTV Box and play backed in the lab setup for different TCP congestion control algorithms. Additional delay and loss were introduced to simulate a line. Many measurements were taken and one example (120ms delay and 0.1% loss) is presented here. The movie file has a length of 50 sec and a bitrate of 7575 kb/s according to *ffprobe*. Additionally to the Time Sequence Graph there is the flight_cwnd graph by *tcpcsm* which shows the estimated size of the cwnd in bytes at the sender whereas the X-axis represents time.

New Reno

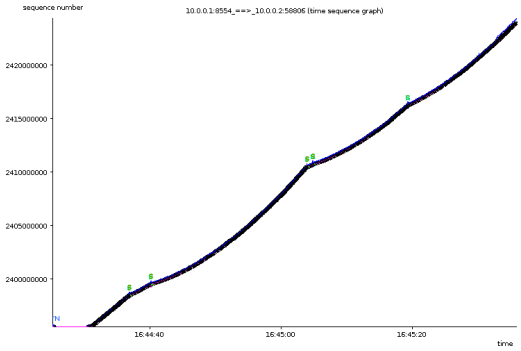

Fig. 5) live555 Server with 120ms delay and 0.1% loss

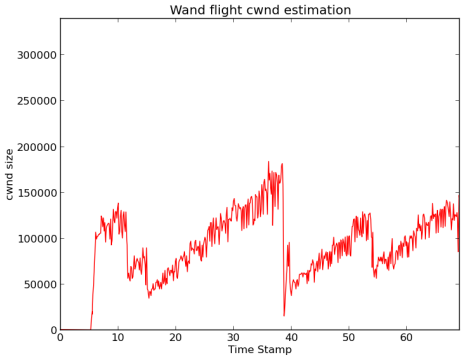

Fig. 6) flight_cwnd estimation related to Fig. 5

throughput	401’702 Bps
max owin	105’705 bytes
avg owin	56’656 bytes

It can be seen that *New Reno* is quite conservative and since it has only loss as a feedback signal, it does not speed up very fast. It never reaches big cwnd sizes which limits the throughput.

Cubic

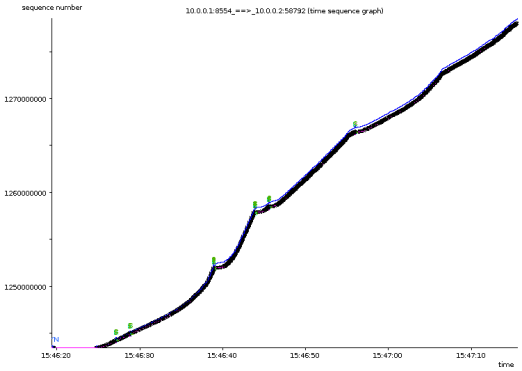

Fig. 7) live555 Server with 120ms delay and 0.1% loss

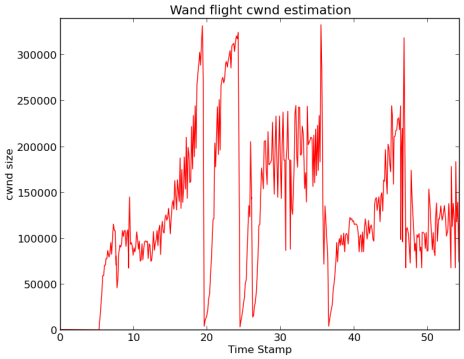

Fig. 8) flight_cwnd estimation related to Fig. 7

throughput	615’280 Bps
max owin	268’545 bytes
avg owin	78’637 bytes

Cubic is much more aggressive than *New Reno* due its cubic cwnd function that depends on the time since the last congestion event. The effect of that can be seen in the large second derivative of the congestion window (the rapid speedup) after a loss. Also it is quite obvious that *Cubic* is much more jittery compared to *New Reno* but it gets more bandwidth faster. It is quite impressive how much higher the throughput with *Illinois* is compared to *New Reno*.

Illinois

As a comparison to the *New Reno* and *Cubic* case above there is also the same analysis for the "hybrid" algorithm *Illinois*:

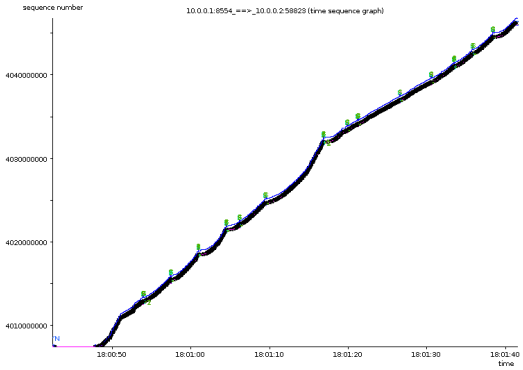

Fig. 9) live555 Server with 120ms delay and 0.1% loss

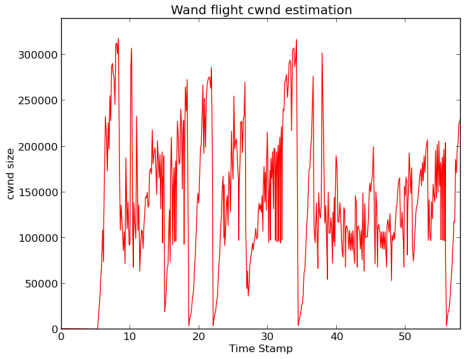

Fig. 10) flight_cwnd estimation related to Fig. 9

throughput	651’377 Bps
max owin	254’381 bytes
avg owin	75’707 bytes

Illinois seems to find an optimal sending rate after some time due to the big amount of feedback (*Illinois* is a loss-delay based algorithm). The result is a much straighter line than *Cubic* in the Time Sequence Graph. The fact that it is less jittery than *Cubic* and has an even higher throughput makes it the best congestion control algorithm of the three evaluated ones (with respect to media streaming since this requires a constant and high data rate).

4.4.3 Comparison of different congestion control algorithms on TCP bulk data transfer as a reference

This section shows that for the evaluation of TCP congestion control algorithms for media streaming, it is not sufficient to perform TCP bulk data transfers, as this results in a different behavior.
With the purpose of having a reference to the RTP media stream experiments, the same movie file is transferred by file transfer (over the same simulated line) instead of streamed. The movie file with the size 49.1 MB was served by the Python *simpleHTTPServer* and received by *wget*.

There is a difference to the last section. The scale of the Y-Axis in the flight_cwnd plot is changed for this section because of the much higher values appearing.

New Reno

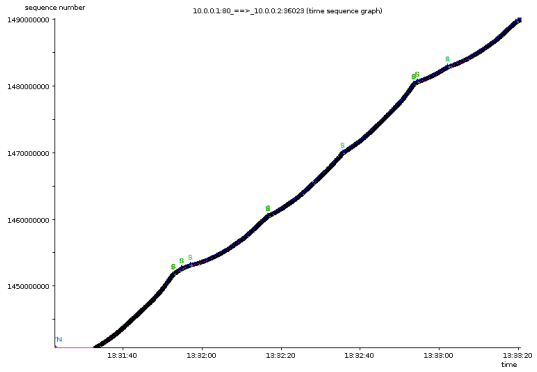

Fig. 11) simpleHTTPServer with 120ms delay and 0.1% loss

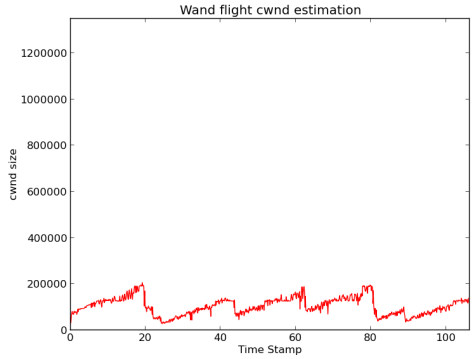

Fig. 12) flight_cwnd estimation related to Fig. 11

throughput	416’434 Bps
max owin	146’249 bytes
avg owin	65’534 bytes

Like in the streaming scenario *New Reno* is quite conservative and never reaches big cwnd sizes. Generally it looks very similar to the streaming scenario and never reaches a straight line in the Time Sequence Graph.

Cubic

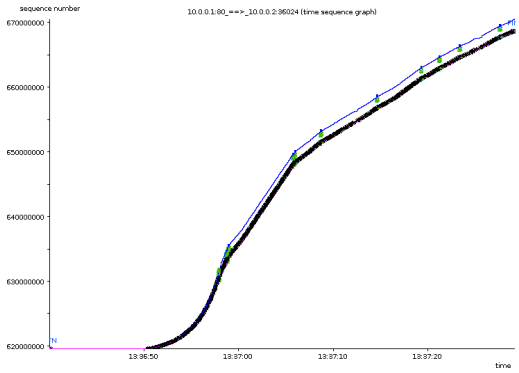

Fig. 13) simpleHTTPServer with 120ms delay and 0.1% loss

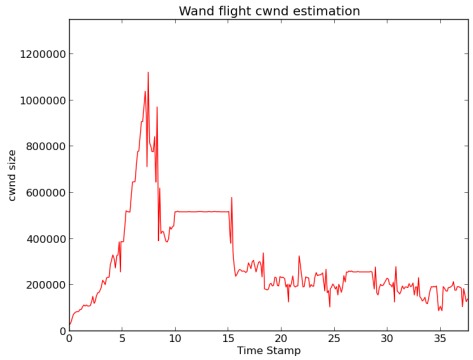

Fig. 14) flight_cwnd estimation related to Fig. 13

throughput	995’953 Bps
max owin	964’369 bytes
avg owin	254’803 bytes

Cubic performed much better in the TCP bulk data transfer than in the streaming experiment. I assume that the *simpleHTTPServer* tries to send as fast as possible, whereas a streaming server tries to keep the sending rate in a certain range (constant in the optimal case).

Illinois

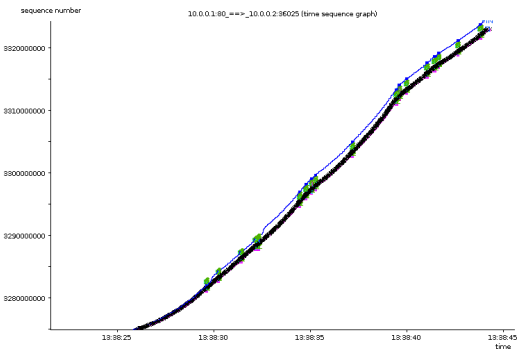

Fig. 15) simpleHTTPServer with 120ms delay and 0.1% loss

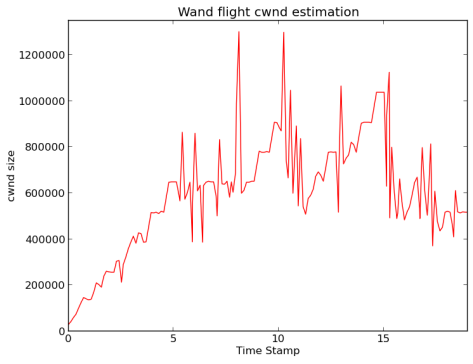

Fig. 16) flight_cwnd estimation related to Fig. 15

throughput	1’621’072 Bps
max owin	1’068’625 bytes
avg owin	417’036 bytes

Also *Illinois* performed much better in the TCP bulk data transfer than in the streaming experiment. Again I assume that the *simpleHTTPServer* tries to send as fast as possible whereas a streaming server tries to keep the sending rate in a certain range (constant in the optimal case). It is remarkable how big the cwnd size gets during the TCP bulk data transfer. Since the line has a rather high delay, there can be a lot of data in flight and the size of cwnd grows. This also leads to more buffering and therefore again to a higher delay which is kind of a positive feedback loop that can become a problem.

Chapter 5

Outlook

Since a lot of work for this thesis went into finding a way to model the server and the line, further work could be done regarding the given problem. Possible components of future work may include:

5.1 Improvement of the lab setup

Since there is no way to use the real IPTV Server (that runs on the RTP over TCP without RTCP stack) for experiments, a custom model of the situation has been built. It has to be assumed that the custom server and client described by this thesis do not exactly behave like the real server. A comparison with the real IPTV system could indicate areas of potential improvement of the lab setup.

5.2 Experiments with burst losses

Dario from Swisscom mentions not only random loss (and delay) as imperfect characteristics of the last mile but also burst losses up to 150ms. Burst losses can easily be simulated using *WANem*. Further experiments for the evaluation of different TCP congestion control algorithms could be done with burst losses to have a more realistic view on the performance of the algorithms.

5.3 More Algorithms

There is a large number of TCP congestion control algorithms. This thesis compares *New Reno*, *Cubic* and *Illinois*. The performance comparison of different TCP congestion control algorithms could be extended to a larger set of algorithms such as Vegas, Compound, STCP, etc.

5.4 Web100

Network captures can only help to estimate the size of cwnd with tools, which is not perfectly accurate. The *Web100 Project* [11] could be used to get the real size of cwnd directly from the kernel, because this reflects the reactions of the algorithm best.

5.5 Middleboxes

Middleboxes in the end user network can influence the performance of TCP significantly. Future research could cover the effect of middleboxes on TCP congestion control performance over lossy lines in the case of media streaming.

5.6 SIAD

The recently published TCP congestion control algorithm SIAD by Mirja Kuehlewind could be evaluated for media streaming over TCP.

5.7 Mininet performance

In this thesis *Mininet* shows large variations in RTT which makes it necessary to have real machines interconnected for TCP performance experiments. Further work could investigate the accuracy of *Mininet* with respect to timing and TCP performance.

5.8 TCP Congestion Control specifications

A continuative thesis could specify the optimal TCP congestion control for media streaming based upon measurement and analysis conclusions.

5.9 Lossy last mile

In general the characteristics and influences of imperfections in the physical layers like WLAN or LTE could be investigated. Consequences for TCP congestion control could then be found and recommendations for future algorithms could be formulated.

5.10 Quality of Experience

The relation between the performance of different TCP congestion control algorithms and the Quality of Experience of media streams over TCP could be explored.

Chapter 6

Summary

The main goal of this thesis is to investigate the effect of TCP congestion control on RTP media streams in the context of IPTV. The case is motivated by issues in the Swisscom IPTV system, where customers can experience stalls in the stream playback although there is enough bandwidth available. Swisscom uses RTP over TCP without RTCP instead of RTP over UDP to bypass problems with middleboxes, that could arise when using UDP.
Unfortunately assumptions of the TCP congestion control algorithms can be violated with the presence of lossy last miles, which avoids a stable allocation of bandwidth that would be needed for an optimal performance of media streaming. Since lossy last miles are not rare with modern technologies like DSL or WLAN, the problem is quite widespread.
The task of this project is to build a model network with a line simulator (for delay and loss) and take measurements for TCP congestion control algorithm evaluation regarding the specific problem. *Mininet* shows unstable RTT timing which leads to the necessity to have a lab setup with real machines. Eventually the setup comprises the installation and configuration of a media server and a client that run the exotic protocol stack of RTP over TCP without RTCP.
Measurements of network traffic captures are analyzed with the help of *tcptrace* for Time Sequence Graphs and *tcpcsm* for cwnd estimations. A short evaluation of the algorithms is presented which makes clear that *Illinois* performs better than *New Reno* or *Cubic* in the specific environment. Also a comparison to TCP bulk data transfer illustrates that the behavior of TCP is not the same for RTP media streaming (over TCP) and bulk data transfer.
Finally some suggestions for further work are proposed.

Bibliography

[1] Host-to-Host Congestion Control for TCP,
Afanasyev, A. and Tilley, N. and Reiher, P. and Kleinrock, L., Communications Surveys Tutorials, IEEE, third quarter 2010, pages 304-342,
http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=05462976,
Date of my last visit: 20th of May 2014

[2] Introduction to Mininet,
Github Wiki,
https://github.com/mininet/mininet/wiki/Introduction-to-Mininet,
Date of my last visit: 22th of May 2014

[3] Download/Get Started With Mininet,
mininet.org,
http://mininet.org/download,
Date of my last visit: 22th of May 2014

[4] Mininet Walkthrough,
mininet.org,
http://mininet.org/walkthrough,
Date of my last visit: 22th of May 2014

[5] WANem
The Wide Area Network emulator,
http://wanem.sourceforge.net,
Date of my last visit: 26th of May 2014

[6] LIVE555 Streaming Media,
http://www.live555.com/liveMedia,
Date of my last visit: 27th of May 2014

[7] Wireshark,
http://www.wireshark.org,
Date of my last visit: 27th of May 2014

[8] tcptrace,
http://www.tcptrace.org,
Date of my last visit: 27th of May 2014

[9] tcpcsm,
http://www.wand.net.nz/ salcock/tcpcsm/,
Date of my last visit: 27th of May 2014

[10] jPlot,
http://tcptrace.org/jPlot/#Download,
Date of my last visit: 27th of May 2014

[11] The web100 Project,
http://www.web100.org,
Date of my last visit: 12th of June 2014

[12] FFmpeg,
http://www.ffmpeg.org,
Date of my last visit: 12th of June 2014

Appendix A

Task description

A.1 The central question

What's the effect of server-side TCP congestion control on RTP over TCP without RTCP media streams?

A.2 Background

Swisscom IPTV application layer people observed performance issues like stalls in the media stream playback. These problems originate in the fact that the physical link is not perfect and introduces random loss and burst losses. TCP congestion control does not seem to cope well with it.

A.3 Tasks

An appropriate lab setup comprising a media server, a line simulator and a client should be installed and different TCP congestion control algorithms should be tested at different random loss and delay conditions. Further a short analysis of the measurements for different TCP congestion control algorithms has to be done.

A.4 Annotation

At the beginning of the project it was not clear to me that there is no test server application or a lab that I could use. Therefore, a simulation/emulation environment had to be built first. Also *Mininet* performed worse than I expected on the test machine, which led to the necessity to have a lab setup with real machines. These two things then occupied very much time. As a consequence the extent, to which the different TCP congestion control algorithms could be evaluated, decreased. However a lab setup was created, that was able to show the problem and the characteristics of the different TCP congestion control algorithms very well.