TCP 拥塞控制和缓冲区膨胀

在本作业中，您将创建自己的网络模拟，以研究 TCP 的动态以及网络运营商做出的看似微小的配置决策如何对性能产生重大影响。

TCP 是一种用于在不可靠的数据包交换网络上获得可靠传输的协议。 TCP 的另一个重要组成部分是拥塞控制，即限制终端主机发送速率以防止网络基础设施因流量而不堪重负。

然而，即使终端主机使用 TCP，网络也可能会遇到与拥塞相关的性能问题。 当路由器和交换机上的数据包缓冲区太大时，可能会出现一种称为缓冲区膨胀的问题。

在本实验中，将使用 Mininet（一种用于网络实验的有用工具）来模拟小型网络并收集与 TCP 拥塞控制和缓冲区膨胀相关的各种性能统计数据。 这将使您能够推断 TCP 和路由器配置对网络性能的影响。

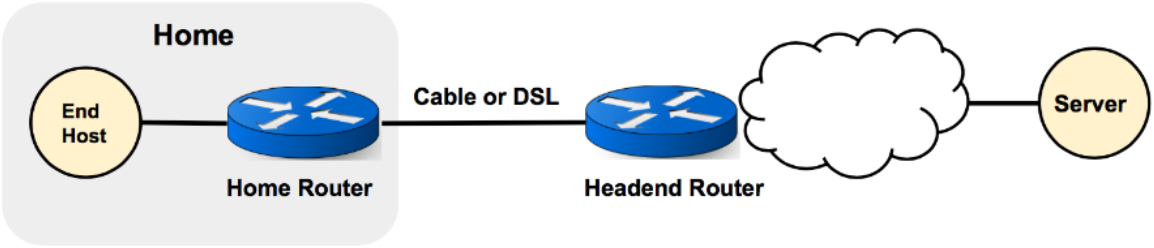
背景

TCP 拥塞窗口大小

TCP 拥塞窗口大小参数（congestion window size parameter，通常为“cwnd”）由发送方维护，并确定任何时候有多少流量可以未完成（已发送但未确认）。 有许多算法可用于在 TCP 连接期间控制 cwnd 的值，所有算法的目的都是在防止拥塞的同时最大化连接的吞吐量。

缓冲膨胀

缓冲区膨胀是当交换设备配置为使用过大的缓冲区时发生的一种现象，这反过来会导致高延迟和数据包延迟变化（抖动）。 即使在如下所示的典型家庭网络中也可能发生这种情况：



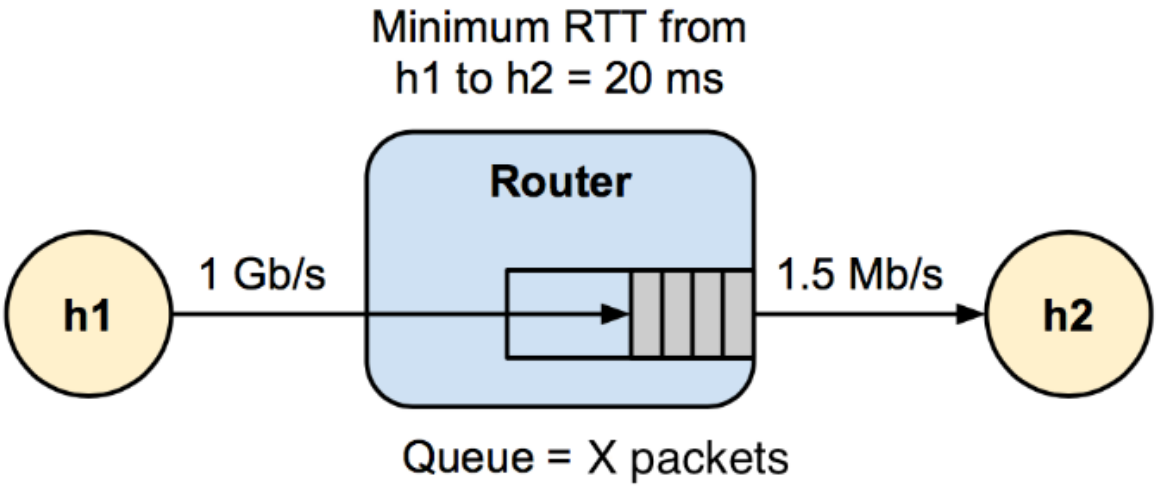
图示为家庭网络中的终端主机连接到家庭路由器。 然后，家庭路由器通过电缆或 DSL 连接到由互联网服务提供商 (ISP) 运行的头端路由器。 通过在 Mininet 中模拟和试验类似的网络，您将看到缓冲区膨胀如何导致性能不佳。

Mininet

Mininet 是一个网络模拟器，您可以使用它在一台计算机上创建虚拟主机（host）、交换机（switches）、控制器（controller）和链路（link）的自定义网络。 仿真网络中的虚拟设备可以运行真实的程序； 任何可以在 Linux 上运行的东西也可以在 Mininet 设备上运行。 这使得 Mininet 成为快速、轻松地模拟网络协议和测量的宝贵工具。 Mininet 简介是 Mininet 的 Python API 入门的有用指南。 如果您有兴趣，Mininet 网站还有其他资源。

A：网络仿真与测量

首先，您应该首先使用 Mininet 的 Python API 创建以下网络，该网络模拟典型的家庭网络：



这里 h1 是一个 Web 服务器，可以快速连接 (1Gb/s) 到您的家庭路由器。 家庭路由器与家庭计算机的下行链路连接速度较慢 (1.5Mb/s)。 h1 和 h2 之间的往返传播延迟或最小 RTT 为 20ms。 路由器缓冲区（队列）大小将是模拟中的自变量。
为了在 Mininet 中创建自定义拓扑，我们扩展了 mininet.topo.Topo 类。 我们已经为您将交换机（路由器）添加到拓扑中。 您需要添加 h1、h2 和具有适当特征的链接来创建上图中指定的设置。 Working with Mininet 的前几个小节描述了如何向拓扑添加元素并设置性能参数。

```
from mininet.topo import Topo

class BBTopo(Topo):
    "Simple topology for bufferbloat experiment."

    def __init__(self, queue_size):
        super(BBTopo, self).__init__()

        # Create switch s0 (the router)
        self.addSwitch('s0')

        # TODO: Create two hosts with names 'h1' and 'h2'


        # TODO: Add links with appropriate bandwidth, delay, and queue size
parameters.
        #       Set the router queue size using the queue_size argument
        #       Set bandwidths/latencies using the bandwidths and minimum RTT
given in the network diagram above

        return
```

接下来，我们需要一些辅助函数来生成两台主机之间的流量。 以下函数启动一个长期存在的 TCP 流，该流使用 iperf 将数据从 h1 发送到 h2。 Iperf是“用于主动测量 IP 网络上可实现的最大带宽的工具。”。 可以将此 iperf 流量视为单向视频通话。 它不断尝试从 Web 服务器 h1 向家庭计算机 h2 发送大量流量。

以下函数接收一个名为 `net` 的参数，该参数是我们在上面创建的具有 BBTopo 拓扑的 mininet 实例。我们已经为 iperf 服务器（h2）编写了部分。在iperf中，服务器是接收数据的服务器，应该是家用电脑 h2。 您需要完成该功能才能在 iperf 客户端 (h1) 上启动 iperf。 iperf 会话应运行 `experiment_time` 参数中指定的秒数。

您将需要使用 `popen` 函数在 mininet 主机上运行 shell 命令。 `popen` 的第一个参数是一个字符串命令，就像您在 shell 中运行的一样。 第二个参数应该是 `shell=True`。 您需要在文档https://iperf.fr/iperf-doc.php#3doc中查找适当的命令行选项，以在给定时间内作为客户端运行 iperf。 您还需要在 iperf 命令中包含 h2 的 IP 地址。 可以使用 `h2.IP()` 方法访问此 IP 地址。

```
def start_iperf(net, experiment_time):
    # Start a TCP server on host 'h2' using perf.
    # The -s parameter specifies server mode
    # The -w 16m parameter ensures that the TCP flow is not receiver window
limited (not necessary for client)
    print "Starting iperf server"
    h2 = net.get('h2')
    server = h2.popen("iperf -s -w 16m", shell=True)

    # TODO: Start an TCP client on host 'h1' using iperf.
    #       Ensure that the client runs for experiment_time seconds
    print "Starting iperf client"
```

接下来，您需要完成以下函数，启动从 h1 到 h2 的连续 ping 序列以测量 RTT。 应每 0.1 秒发送一次 ping。 结果应该从 stdout 重定向到 `outfile` 参数。

和前面一样， `net` 是具有 BBTopo 拓扑的 mininet 实例。需要使用 `popen`。 `popen` 的命令参数可以使用 `>` 重定向 stdout，就像普通的 shell 命令一样。 阅读 `ping` 的man page以获取有关可用命令行参数的详细信息。 确保 `popen` 的第二个参数是 `shell=True`。

```
def start_ping(net, outfile="pings.txt"):
    # TODO: Start a ping train from h1 to h2 with 0.1 seconds between pings,
redirecting stdout to outfile
    print "Starting ping train"
```

接下来，我们开发一些辅助函数来测量 TCP 流量的拥塞窗口。 这将使我们分析 mininet 网络中 TCP 连接的动态。 以下功能已经完成。

```
from subprocess import Popen
import os

def start_tcpprobe(outfile="cwnd.txt"):
    Popen("sudo cat /proc/net/tcpprobe > " + outfile, shell=True)

def stop_tcpprobe():
    Popen("killall -9 cat", shell=True).wait()
```

然后，我们创建一个辅助函数来监视给定接口上的队列长度。 这将使我们分析路由器缓冲区队列中的数据包数量如何影响性能。 这个功能已经完成。

```
from multiprocessing import Process
from monitor import monitor_qlen

def start_qmon(iface, interval_sec=0.1, outfile="q.txt"):
    monitor = Process(target=monitor_qlen,
                      args=(iface, interval_sec, outfile))
    monitor.start()
    return monitor
```

我们还需要一个在 h1 上启动 Web 服务器的辅助函数。 这个功能已经完成。

```
from time import sleep

def start_webserver(net):
    h1 = net.get('h1')
    proc = h1.popen("python http/webserver.py", shell=True)
    sleep(1)
    return [proc]
```

最后，我们需要一个在 h2 上运行的辅助函数，在`experiment_time`内每 3 秒从 h1 获取网站，并打印下载时间的平均值和标准差。 这个功能已经完成。

```
from time import time
from numpy import mean, std
from time import sleep

def fetch_webserver(net, experiment_time):
    h2 = net.get('h2')
    h1 = net.get('h1')
    download_times = []

    start_time = time()
    while True:
        sleep(3)
        now = time()
        if now - start_time > experiment_time:
            break
        fetch = h2.popen("curl -o /dev/null -s -w %{time_total} ", h1.IP(),
shell=True)
        download_time, _ = fetch.communicate()
        print "Download time: {0}, {1:.1f}s left...".format(download_time,
experiment_time - (now-start_time))
        download_times.append(float(download_time))

    average_time = mean(download_times)
    std_time = std(download_times)
    print "\nDownload Times: {}s average, {}s stddev\n".format(average_time,
std_time)
```

现在，我们需要将所有部分放在一起来创建网络，启动所有流量并进行测量。

`bufferbloat()` 函数应该：

- 创建 `BBTopo` 对象
- 启动 TCP 和队列监视器
- 使用iperf启动一个长时期的TCP流
- 启动 ping序列
- 启动网络服务器
- 定期从 h1 下载 index.html 网页并测量获取它需要多长时间

请注意，长期存在的流量、ping 序列和网络服务器下载应该同时发生。 完成此处之前的分配步骤后，请完成下面 `bufferbloat()` 函数中标记为 TODO 的部分。 每个 TODO 部分都需要添加一行来调用上面定义的函数。

```
from mininet.node import CPULimitedHost, OVSController
```

```
from mininet.link import TCLink
from mininet.net import Mininet
from mininet.log import lg, info
from mininet.util import dumpNodeConnections

from time import time
import os
from subprocess import call

def bufferbloat(queue_size, experiment_time, experiment_name):
    # Don't forget to use the arguments!

    # Set the cwnd control algorithm to "reno" (half cwnd on 3 duplicate acks)
    #    Modern Linux uses CUBIC-TCP by default that doesn't have the usual
sawtooth
    #    behaviour.  For those who are curious, replace reno with cubic
    #    see what happens...
    os.system("sysctl -w net.ipv4.tcp_congestion_control=reno")

    # create the topology and network
    topo = BBTopo(queue_size)
    net = Mininet(topo=topo, host=CPULimitedHost, link=TCLink,
                  controller= OVSController)
    net.start()

    # Print the network topology
    dumpNodeConnections(net.hosts)

    # Performs a basic all pairs ping test to ensure the network set up properly
    net.pingAll()

    # Start monitoring TCP cwnd size
    outfile = "{}_cwnd.txt".format(experiment_name)
    start_tcpprobe(outfile)

    # TODO: Start monitoring the queue sizes with the start_qmon() function.
    #       Fill in the iface argument with "s0-eth2" if the link from s0 to h2
    #       is added second in BBTopo or "s0-eth1" if the link from s0 to h2
    #       is added first in BBTopo. This is because we want to measure the
    #       number of packets in the outgoing queue from s0 to h2.
    outfile = "{}_qsize.txt".format(experiment_name)
    qmon = start_qmon(iface="TODO", outfile=outfile)

    # TODO: Start the long lived TCP connections with the start_iperf() function


    # TODO: Start pings with the start_ping() function
    outfile = "{}_pings.txt".format(experiment_name)


    # TODO: Start the webserver with the start_webserver() function


    # TODO: Measure and print website download times with the fetch_webserver()
function


    # Stop probing
```

```
    stop_tcpprobe()
    qmon.terminate()
    net.stop()

    # Ensure that all processes you create within Mininet are killed.
    Popen("pgrep -f webserver.py | xargs kill -9", shell=True).wait()
    call(["mn", "-c"])
```

完成上述所有步骤后，使用 `bufferbloat()` 函数运行实验两次，一次队列大小为 20 个数据包，第二次队列大小为 100 个数据包。 确保运行实验的时间足够长，以便在结果中看到 TCP 的动态，例如 cwnd 的锯齿行为（300 秒应该不错）。 选择反映队列大小的实验名称参数。

```
from subprocess import call
call(["mn", "-c"])

# TODO: call the bufferbloat function twice, once with queue size of 20 packets
and once with a queue size of 100.
```

B：绘制结果

在实验的这一部分中，将通过绘制拥塞窗口、队列长度和 ping RTT 随时间的变化来分析测量结果。 我们为每个测量提供了绘图函数，在下面已经完成的 `plot_measurements()` 函数中调用这些函数。

```
%matplotlib inline
from plot_cwnd import plot_congestion_window
from plot_qsize import plot_queue_length
from plot_ping import plot_ping_rtt

def plot_measurements(experiment_name_list, cwnd_histogram=False):

    # plot the congestion window over time
    for name in experiment_name_list:
        cwnd_file = "{}_cwnd.txt".format(name)
        plot_congestion_window(cwnd_file, histogram=cwnd_histogram)

    # plot the queue size over time
    for name in experiment_name_list:
        qsize_file = "{}_qsize.txt".format(name)
        plot_queue_length(qsize_file)

    # plot the ping RTT over time
    for name in experiment_name_list:
        ping_file = "{}_pings.txt".format(name)
        plot_ping_rtt(ping_file)
```

现在，您需要调用 `plot_measurements` 函数，其中 `experiment_name_list` 参数是您在上面运行 `bufferbloat()` 时使用的 `experiment_name` 参数的列表。这将生成6个图表，显示实验结果。

```
#TODO: Call plot_measurements() to plot your results
```

C：分析

在实验的这一部分中，您将使用上一节中的模拟和绘图来回答有关 TCP 和缓冲区膨胀的一些问题。 这些问题是开放式的，许多问题都有多个正确答案。 答案的长度没有要求，但尽量全面、简洁。 1-2 句话可能太短。 超过 2-3 段可能太长。

首先花一些时间思考一下您刚刚执行的模拟。 该模拟的设置就像家庭网络一样，其中一台家庭计算机通过路由器连接到远程服务器。 从路由器到服务器的链路的带宽比从家庭计算机到路由器的链路低得多。 模拟中的自变量是等待从路由器发送到服务器的数据包缓冲区的最大长度。

流量来源有3个：

1. 持久的 TCP 会话（使用 iperf 创建）将大量流量从家庭计算机发送到服务器。
2. 定期对家庭计算机和服务器进行 ping 和 ping 回复。
3. 定期尝试将网站（使用 HTTP over TCP）从家庭计算机下载到服务器。

正如您（希望）通过实验发现的那样，增加路由器上数据包缓冲区的长度会显着降低 ping RTT 和 HTTP 下载速率指标的性能。

问题一：除家庭网络之外，哪些计算机网络可能具有与您模拟的配置类似的配置？

问题二：写一个符号方程来描述 RTT 和队列大小之间的关系。符号方程应该推广到任何队列大小。 基本上，考虑系统在某一时间点的快照，并以参数方式使用队列大小和链路延迟来计算 RTT。

一个示例（不正确的）符号方程：

$$RTT = kq^2$$

其中 k是常数因子， q是队列中的数据包数量。 你给的方程不限于 k 和 q。

问题三：用技术术语描述为什么增加缓冲区大小会降低性能（RTT 和网页下载时间），从而导致缓冲区膨胀效应。 请务必明确地说明您生成的图以及 TCP 拥塞控制和缓冲区大小之间的关系。

问题四：请使用一些计算机网络以外的、非技术的类比来重新描述缓冲区膨胀效应的原因（即用外行人可以听懂的非技术性语言进行描述）。 能够描述外行人可以理解的技术内容非常重要，并且进行类比通常有助于您自己的推理。

问题五：缓冲区膨胀效应是特定于我们模拟的网络、流量和/或 TCP 拥塞控制算法的类型，还是普遍现象？是否有任何时候增加路由器缓冲区大小会提高性能的情况？ 如果有，请举个例子。 如果没有，请解释为什么。

问题六：确定并描述一种在不减少缓冲区大小的情况下缓解缓冲区膨胀问题的方法。