拥塞算法设计说明书

任务目标

本设计的目标是实现一个基于RTT的动态拥塞控制算法。该算法旨在通过调整拥塞窗口（cwnd）和超时时间（RTO）来优化网络传输效率，并避免网络拥塞。本文详细介绍了`datagram_was_sent` 和`ack_received` 函数中的拥塞算法的设计思路和机制

算法信息

名称：murasame_2022201459.cc

最优得分：4.22

设计思路来源

算法的设计的设计融合了多种经典的拥塞控制算法，尤其是TCP Vegas的思想。TCP Vegas通过RTT的测量和比较来调整拥塞窗口，以实现更精细的拥塞控制。以下是设计中的主要思想来源：

1. **TCP Vegas**：通过估计预期吞吐量和实际吞吐量，检测网络中的拥塞情况，并根据差异调整拥塞窗口。
2. **慢启动（Slow Start）**：在连接初期和发生超时重传后，拥塞窗口以指数方式增长。
3. **拥塞避免（Congestion Avoidance）**：在拥塞窗口达到慢启动阈值后，拥塞窗口以加性增大（AIMD）的方式增长。
4. **TCP Reno的超时**：在发生超时重传后，将慢启动阙值调整为当前拥塞窗口的一半，然后将cwnd设置为初始值。

机制说明

常规拥塞控制（当收到ack时）

1. 采样RTT

在接收到ACK后，首先计算当前的RTT（Round-Trip Time）：

```
    sample_rtt = timestamp_ack_received - send_timestamp_acked;
```

本来准备使用TCP经典的RTO计算方法在此步中计算RTO的，但是最后发现效果不如固定RTO为固定值，故而放弃。

2. 更新基础RTT（base RTT）

通过使用一个全局min_rtt记录全局最小的RTT值，更新 `last_base` 和 `base_rtt`：

```
    if(min_rtt>sample_rtt)
        min_rtt = sample_rtt;

    if (min_rtt != base_rtt) {
        last_base = base_rtt;
        base_rtt = min_rtt;
    }
```

3. 计算预期和实际吞吐量以及吞吐量差异

根据当前的 `cwnd` 和 `RTT`，计算预期和实际的吞吐量以及吞吐量差异：

```
    double expected = cwnd / base_rtt;
    double actual = cwnd / sample_rtt
    double diff = base_rtt * (expected - actual);
```

这里参考了TCP Vegas中的思想，即

- **expected**：

$$expected = \frac{cwnd}{base\_rtt}$$

  - 预期吞吐量，基于最小RTT（`base_rtt`）计算，即假设网络中没有排队延迟时的吞吐量。

- **actual**：

$$actual = \frac{cwnd}{sample\_rtt}$$

  - 实际吞吐量，基于当前的RTT（`sample_rtt`）计算，即当前网络状况下的实际吞吐量。

- **diff**：

$$diff = base\_rtt \times (expected - actual)$$

  - 是预期吞吐量和实际吞吐量之间的差异，乘以 `base_rtt` 可以将这个差异量化为时延的形式。这个差异值用于判断当前网络状况：
  - 当 `diff` 小于某个阈值（`alpha`）时，认为没有拥塞，可以增加拥塞窗口。
  - 当 `diff` 大于某个阈值（`betas`）时，认为存在拥塞，需要减少拥塞窗口。

4.计算目标拥塞窗口

根据 base_rtt 和 last_base 计算目标拥塞窗口大小：

```
double target = cwnd * base_rtt / last_base;
```

这里的计算方法参考了RTT比例法，即根据最小RTT和之前的RTT值来调整拥塞窗口：

- base_rtt 代表当前测得的最小RTT，反映了当前网络的最小延迟。
- last_base 代表之前测得的最小RTT，反映了之前网络的最小延迟。

这种方法使得控制器能够根据当前和之前的网络状况动态调整拥塞窗口，从而更灵活地应对网络变化。

5.慢启动（Slow Start）

在 cwnd 小于慢启动阈值 ssthresh 时，采用慢启动算法进行拥塞窗口的调整：

```
    if (cwnd < ssthresh) { // 慢启动
        if (static_cast<uint64_t>(diff) > gamma) {
            cwnd += min(static_cast<uint64_t>(cwnd), static_cast<uint64_t>(target) + 1);
        } else {
            cwnd += min(max(2 * cwnd, 2.0), SLOW_START_INCREMENT * (ssthresh - cwnd));
        }
```

当差异大于gamma 时，认为存在拥塞，需要保守的增长，所以最多也才增长一倍的cwnd

反之，则可以快速增长，拥塞窗口 cwnd 按照慢启动增量（SLOW_START_INCREMENT）和当前窗口大小进行指数增长。

6.拥塞避免（Congestion Avoidance）

在 cwnd 大于慢启动阈值 ssthresh 时，采用拥塞避免算法进行调整：

```
else { // 拥塞避免
    if (diff < alpha) {
        increase_times++;
        decrease_times = 0;
        cwnd += CONGESTION_AVOIDANCE_INCREMENT;
    } else if (diff > betas) {
        decrease_times++;
        increase_times = 0;
        cwnd = max(1.0, cwnd - 0.1);
    }
}
```

- 当差异小于 alpha 时，增加 cwnd 并记录增加次数。
- 当差异大于 betas 时，减少 cwnd 并记录减少次数。

7. 动态调整参数

根据连续的增加或减少次数，动态调整 alpha、betas 和 ssthresh 的值：

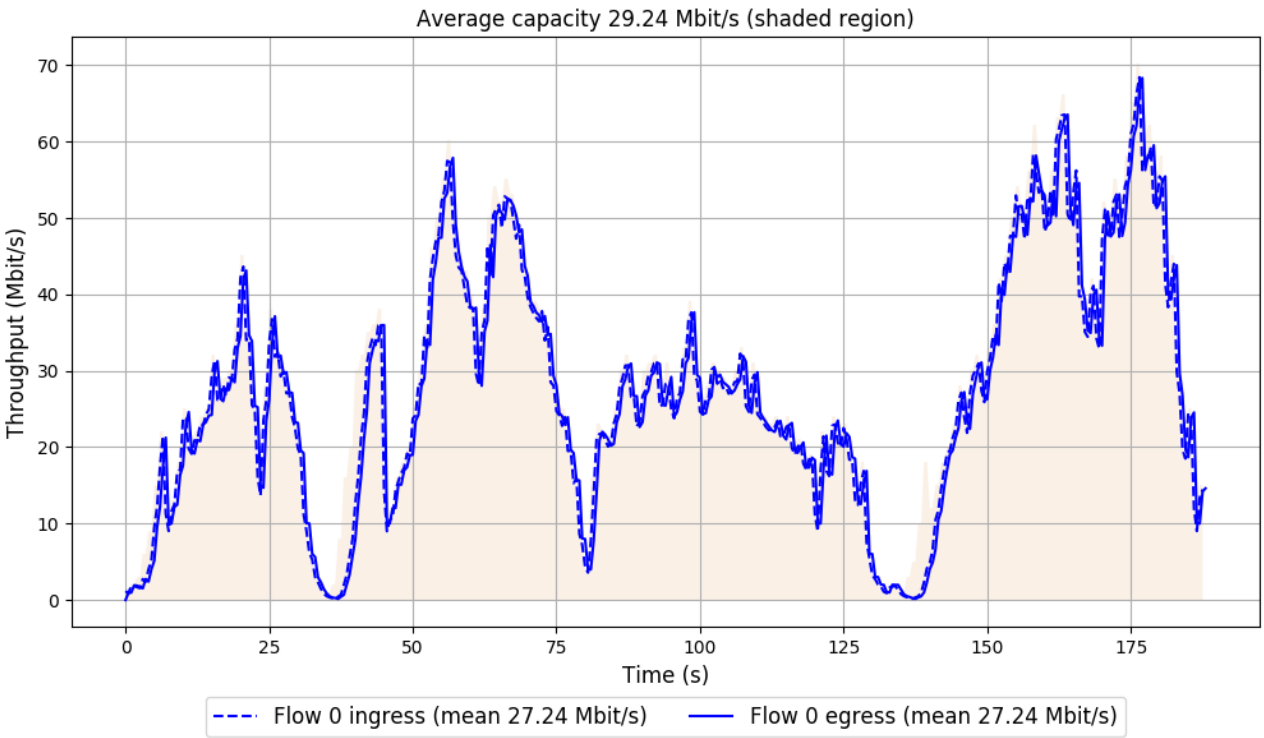
```
    if (increase_times >= 2) {
        alpha *= ALPHA_INCREMENT;
        betas *= ALPHA_INCREMENT;
        increase_times = 0;
        ssthresh *= 1.4;
    }
    if (decrease_times >= 2) {
        alpha *= ALPHA_DECREMENT;
        betas *= ALPHA_DECREMENT;
        decrease_times = 0;
        ssthresh *= 0.714;
    }
```

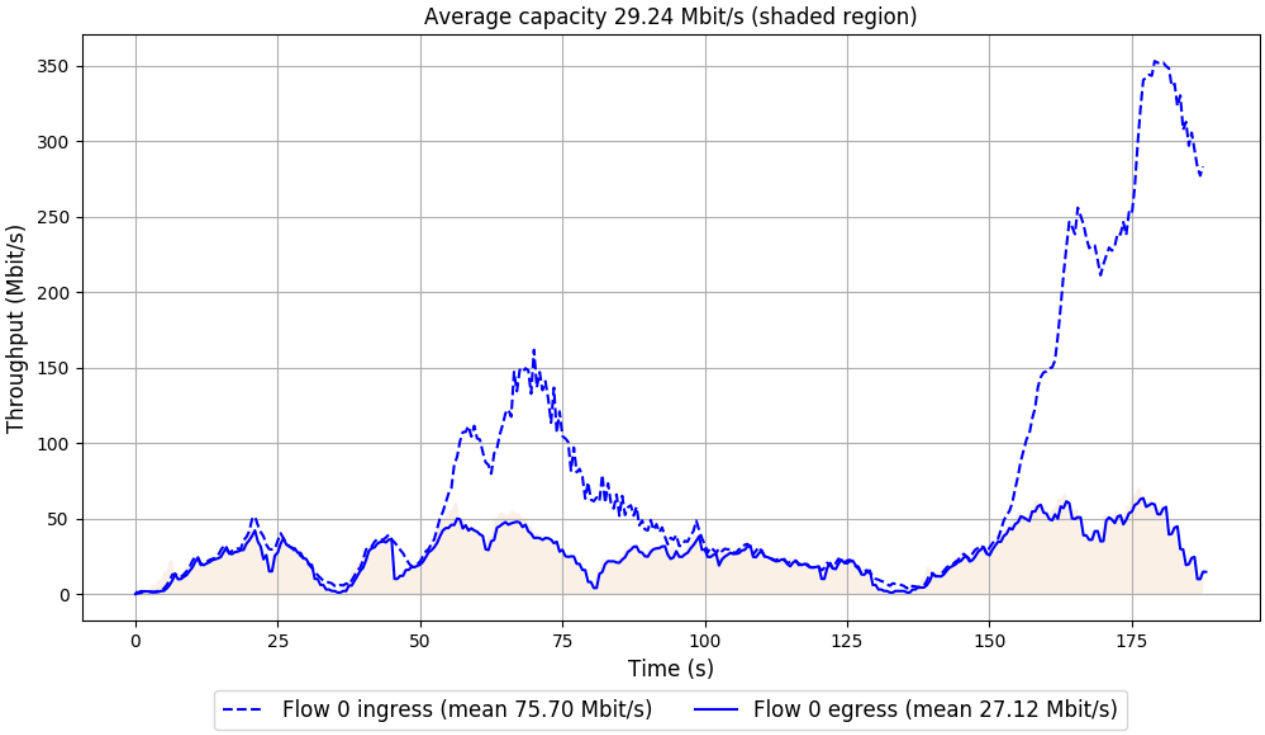
超时拥塞控制

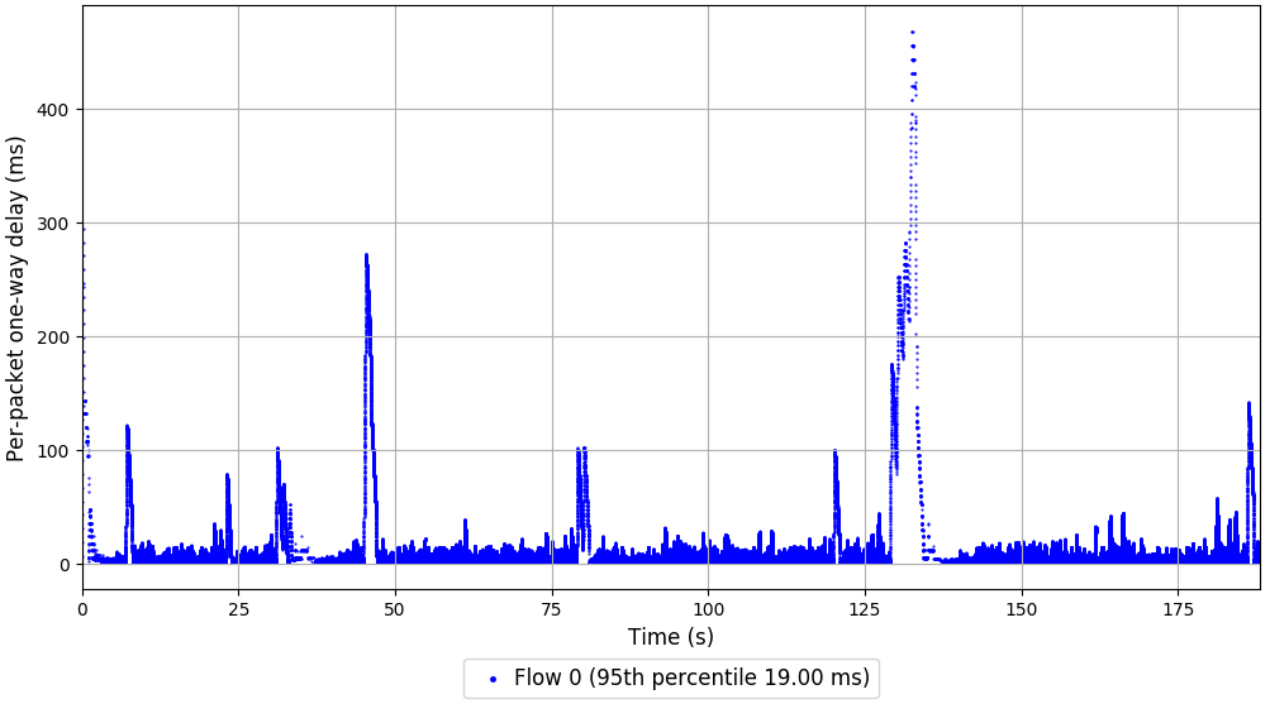
在datagram_was_sent函数中接受到超时信号后，使用经典的Reno方法，更新ssthresh为cwnd的一半，并将cwnd初始化

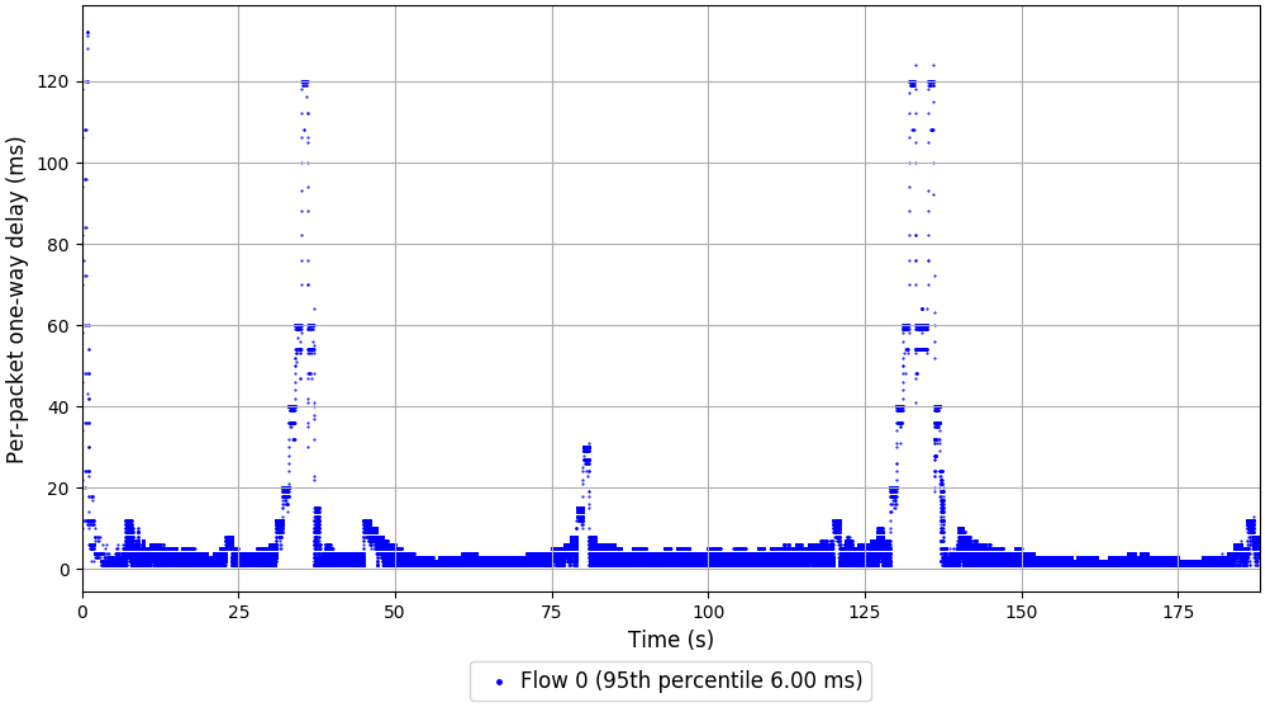
```
  if(after_timeout){
    ssthresh = cwnd/2 + 1;
    cwnd = min(25.0,ssthresh/2);
  }
```

性能图片




Average capacity 29.24 Mbit/s (shaded region)
Throughput (Mbit/s)
Time (s)
Flow 0 ingress (mean 75.70 Mbit/s)
Flow 0 egress (mean 27.12 Mbit/s)


Per-packet one-way delay (ms)
Time (s)
Flow 0 (95th percentile 19.00 ms)


Per-packet one-way delay (ms)
0
20
40
60
80
100
120
0
25
50
75
100
125
150
175
Time (s)
Flow 0 (95th percentile 6.00 ms)