计算机网络实验：一组

组长: 胡啸。组员：唐帅, 毕益绅, 陈月如, 曹潇

2020 年 12 月 18 日

目录

1 摘要

公平和效率是计算机网络中的两个主要问题。

在多队列调度问题中，本文实现了多种算法以尽可能公平地服务每条队列，共 Deficit Round Robin[4],Weighted Fair Queueing[3],Self-clocked Fair Queueing[2] 三种调度算法。在此基础上理论上分析了实验结果以及上述三种算法的公平性。

在多路访问协议研究中,本文实现了 Pure Aloha,Slot Aloha,Persistent CSMA,non-Persistent CSMA 四种算法，并在此基础上模拟了 802.11 的 MAC 子层协议。最后在多个角度上对实验结果进行了分析。

2 多队列公平调度

2.1 MM1

单服务台等待模型 M/M/1/∞ 是指：顾客的相继到达时间服从参数为 λ 的负指数分布，服务台的个数为 1，服务时间 V 服从参数为 μ 的负指数分布，系统空间无限，允许永远排队，

这是一类最简单的排队系统。

指数分布（也称为负指数分布）是描述泊松过程中的事件之间的时间的概率分布，即事件以恒定平均速率连续且独立地发生的过程。它具有无记忆的关键性质，即两次相邻到达的顾客的时间间隔：

$$f(x) = \lambda e^{-\lambda x} x > 0 \tag{1}$$

$$P(X < t) = F(t) = \int_0^t f(x)dx = 1 - e^{-\lambda x} x > 0 \tag{2}$$

$$F(t) \in [0, 1] \tag{3}$$

在程序中通过随机出 $F(t)$ 来逆向计算出下一个顾客的到达间隔：

$$F(t) = random(0, 1) \tag{4}$$

$$t = -\frac{ln(1 - random(0, 1))}{\lambda} \tag{5}$$

表 1 为平均队列长度模拟值和平均等待时间与理论值的比较，其中参数为 $\lambda = 0.75\mu = 1.0 endtime = 1e9$:

指标	计算值	理论值
等待时间	3.9997	4
队列长度	2.25	2.24998

表 1: MM1 计算值与理论值

2.1.1 队长结果分析

记 $p_n = P\{N = n\} (n = 0, 1, 2...)$ 为系统达到平稳状态后队长 N 的概率分布，令 $\rho = \frac{\lambda}{\mu}$，并保证 $\rho < 1$，平均队列长度和队长的分布图如图 1 所示。

队长为 n 的概率为 $p_n = (1 - \rho)\rho^n$，对该式积分得到队列长度分布：

$$F(n) = P(X > n) = 1 - P(X <= n) = 1 - \Sigma_0^n (1 - \rho)\rho^n = \rho^{n+1}$$

图 1 中红色曲线为理论结果：$F(n) = \rho^{n+1}$. 蓝色曲线为实验结果，通过对比可以发现，实验结果和理论的结果基本吻合，只有在 $y < 10^{-6}$ 时曲线在开始抖动。对于单服务台等待制排队

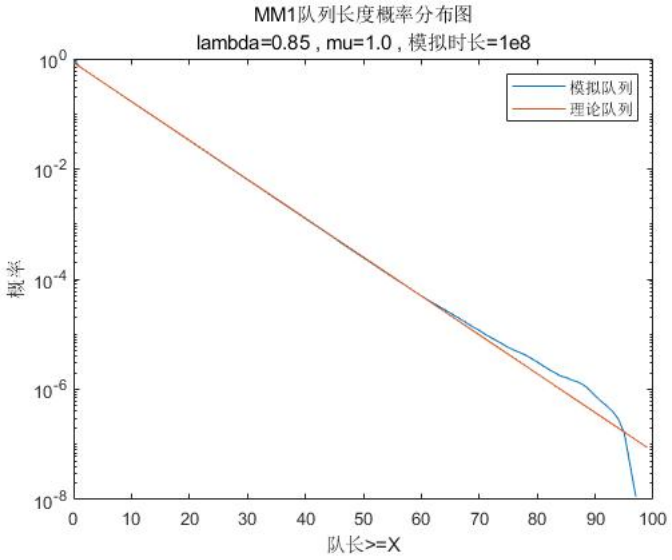

图 1: 队列长度概率分布

系统，由已得到平稳状态下的队长分布，可以得到平均队长 L 为:

$$\begin{aligned} L &= \Sigma_{n=0}^{\infty} n p_n = \Sigma_{n=0}^{\infty} n(1-\rho)\rho^n \\ &= (\rho + 2\rho^2 + 3\rho^3 + ...) - (\rho^2 + 2\rho^3 + 3\rho^4 + ...) \\ &= \rho + \rho^2 + \rho^3 + ... = \frac{\rho}{1-\rho} = \frac{\lambda}{\mu - \lambda} \end{aligned} \tag{6}$$

平均排队长 L_q 为:

$$\begin{aligned} L_q &= \Sigma_{n=0}^{\infty}(n-1)p_n = L - (1-p_0) \\ L - \rho &= \frac{\rho^2}{1-\rho} = \frac{\lambda^2}{\mu(\mu-\lambda)} \end{aligned} \tag{7}$$

在参数为 $(\lambda = 0.7, \mu = 1.0, endtime = 1e9)$ 的情况下，实验计算结果为 2.25，而理论结果为 2.24998。平均队长的计算精度达到了千分之一。

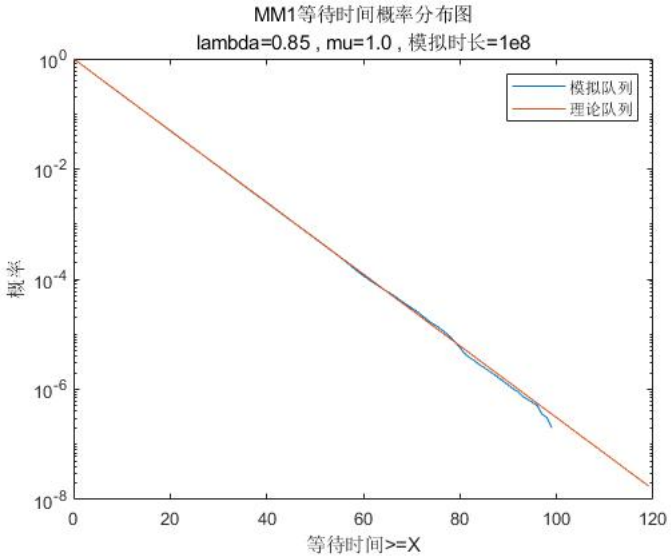

图 2: 等待时间概率分布

2.1.2 等待时间结果分析

设顾客在系统中的逗留时间为 T，可说明它服从参数为 $\mu-\lambda$ 的负指数分布:

$$\begin{aligned} f(t) &= \Sigma_{n=0}^{\infty}(1-\rho)\rho^n \frac{\mu(\mu t)^n}{n!}e^{-(\mu-\lambda)} \\ &\quad -(1-\rho)\mu e^{-(\mu t)}\Sigma_{n=0}^{\infty}\frac{\mu(\rho\mu t)^n}{n!} \\ &= (\mu-\lambda)e^{-(\mu-\lambda)t} \end{aligned} \tag{8}$$

对该式积分，得到系统的逗留时间分布为:

$$\begin{aligned} F(t) &= P(X>t) = 1-P(X<t) \\ &= 1-\int_0^t f(t)dt \\ &= e^{-(\mu-\lambda)t} \end{aligned} \tag{9}$$

图二中红色曲线为理论结果 $F(t)=e^{-(\mu-\lambda)t}$，蓝色曲线为实验结果，实验结果和理论结果基本吻合，只有在 $y<10^{-6}$ 时曲线在开始抖动。通过（8）式可以计算出平均逗留时间:

$$W = E(T) = \frac{1}{\mu-\lambda} \tag{10}$$

因为顾客在系统中的逗留时间等于等待时间和接受服务时间之和，所以可以得到平均等待时间 W_q 为

$$W_q = W - \frac{1}{\mu} = \frac{\lambda}{\mu(\mu - \lambda)} \tag{11}$$

在参数为 $(\lambda = 0.75, \mu = 1.0, endtime = 1e9)$ 的情况下，实验计算结果为 3.9997，而理论结果为 4。平均等待时间的计算精度达到了千分之一。

2.2 公平性度量

定义 $W_f(t_1, t_2)$ 为在时间区间 $[t_1, t_2]$ 内对队列 f 的服务量，根据 $W_f(t_1, t_2)$，可得到公平性度量公式：

$$|W_f(t_1, t_2) - W_g(t_1, t_2)| \tag{12}$$

其中 $W_f(t_1, t_2), W_g(t_1, t_2)$ 分别指的是：时间区间 $[t_1, t_2]$ 内对队列 f 的服务量和时间区间 $[t_1, t_2]$ 内对队列 g 的服务量。

当一个调度算法不偏向任何一个特定的队列时，该方法是公平的，当调度算法偏向某种特定的队列时，队列 f 会受到更多服务，此算法则相对不公平。所以当调度算法在时间区间 $[t_1, t_2]$ 内更偏向 f 时 $|W_f(t_1, t_2) - W_g(t_1, t_2)|$ 将会增大。

当一个调度算法的结果满足下式时：

$$|W_f(t_1, t_2) - W_g(t_1, t_2)| < c \tag{13}$$

称该算法是公平的。其中 c 为一个常数，该值越小则说明算法越公平。

定义 $w_f(t_1, t_2) = \frac{W_f(t_1, t_2)}{r_f}$ 为队列 f 归一化的被服务量，其中 r_f 为队列 f 的带宽。在此基础上，拓展上述对算法公平的定义为：当一个调度算法满足 $|w_f(t_1, t_2) - w_g(t_1, t_2)| < c$ 时，称该算法是公平的，即：

$$|\frac{W_f(t_1, t_2)}{r_f} - \frac{W_g(t_1, t_2)}{r_g}| < c \tag{14}$$

2.3 赤字轮询算法

队列	平均等待时间	平均队列长度
1	8.72853	2.62668
2	8.29376	2.787828
3	7.9375	2.94592

表 2: DRR 队列结果

当两个队列的带宽、权重和包大小都相同时，可以证明简单的 RR 轮询算法是公平的，此时两队列满足下面的式子：

$$|W_f(t_1, t_2) - W_g(t_1, t_2)| < 1 * packet \tag{15}$$

当包大小和权重不同时，RR 算法不能满足公平，为了解决该问题 Sheerhar 和 Varghese 提出了 Deficit Round Robin 算法 [4] ，该算法可在队列权重不同且包大小不同时保证公平性，其论文中的算法流程如算法 1 所示，可以证明 DRR 算法公平性满足下面的式子（Q 为该队列的赤字量）：

$$|\frac{W_f(t_1, t_2)}{Q_f} - \frac{W_g(t_1, t_2)}{Q_g}| < c \tag{16}$$

Algorithm 1 Deficit Round Robin

parameters: const integer N :Nb of queues
const integer Q[1..N]:Per queue quantum
integer DC[1..N]: Per queue deficit counter
queue queue[1..N]:The queues

procedure:

```
1:  while true do
2:      for i in 1..N do
3:          if not queue[i].empty() then
4:              DC[i]:= DC[i] + Q[i]
5:              while not queue[i].empty() and DC[i] ≥ queue[0].size() do
6:                  DC[i] := DC[i] − queue[i].head().size()
7:                  send( queue[i].head() )
8:                  queue[i].dequeue()
9:              end while
10:             if queue[i].empty() then
11:                 DC[i] := 0
12:             end if
13:         end if
14:     end for
15: end while
```

在参数 $\lambda = (0.27, 0.30, 0.33)$，$\mu = 1.0$, 赤字 $= (5.6, 6.0, 6.6)$，模拟时长 =1e8 的情况下，得到了图 3、4。每个队列的平均队列长度和平均等待时间如表 2 所示。

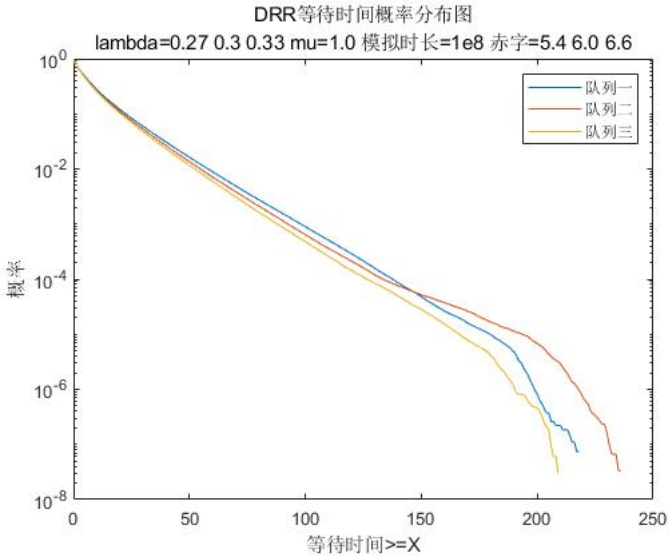

图 3: DRR 等待时间概率分布

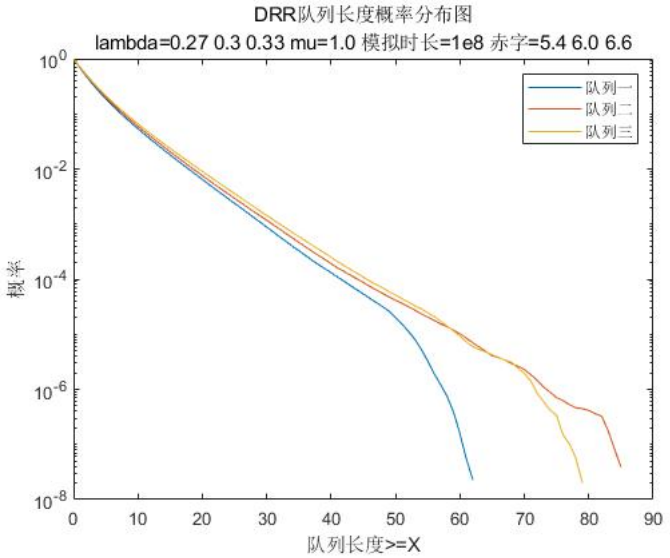

图 4: DRR 队列长度概率分布

由 16 式可知，当各队列的赤字量和 λ 成正比的时候，可以最大限度地保证 DRR 算法的公平性，此时各条队列的等待时间分布曲线和队长分布曲线接近重合。

2.4 最理想的公平调度算法

Fluid Flow Server (FFS) 算法被认为是理想上最公平的算法，它的思想为轮询地服务每条队列，但每次轮询是每条队列只服务 1 个比特。FFS 的公平性程度如下：

$$|w_f(t_1, t_2) - w_g(t_1, t_2)| < 1bit \tag{17}$$

也就是说对两个队列服务的最大差别至多为 1 比特，即理想的公平，但是每次只服务 1 比特不仅效率较低，还破坏了包的完整性。Packet-by-packet GPS[3]（也被人称为 WFQ 算法）和其简化版 SCFQ[2] 算法则通过虚拟时间来近似地模拟 FFS 算法。在赤字轮询算法中，本文通过 2.2 节的公平性度量方法对 DRR 进行了分析。而在后面的 WFQ 和 SCFQ 算法中，本文则从对 FFS 算法的近似程度来对 DRR 进行分析。

2.5 WFQ(Packet-by-packet GPS)

WFQ 的基本思想为：选择在 FFS 算法下最先完成的包进行服务。由于服务不可被中止，当前 FFS 算法下最先完成的包不一定是整个模拟中最先完成的，所以 WFQ 并不能完全模拟 FFS。

可以证明使用 WFQ 调度的包的完成时间不会晚于使用 FFS 调度的包的时间加上最大包长。为了模拟 FFS，Parekh[3] 在论文中提出了虚拟时间来计算 FFS 中最先完成的包。虚拟时间系统可概括为以下几点：

- 虚拟时间和真实时间存在一个对应函数
- 虚拟时间的流速是可变的，虚拟时间的流速等于正常时间流速/W，权重 W 则取决于此时的空队列数
- 虚拟包长决定了包在虚拟时间下的流速，它的值等于真实包长/队列权重
- 第 p^{j+1} 个包的离开时间 $V(p^{j+1}) = max(V(p^j), V(A^{j+1})) + \frac{len(p^{j+1})}{w_j}$。也就是说如果当某个包到达时，如果前一个包已被服务完，那么虚拟离开时间等于这个包的虚拟到达时间加上虚拟包长。如果前一个包未被服务完，那么虚拟离开时间等于上个包的虚拟离开时间加上虚拟包长
- 虚拟时钟只会在新包到达，或者在 FFS 模拟离开时改变

当一个新包到达时，需要的操作如下：

- 计算该包的虚拟完成时间。

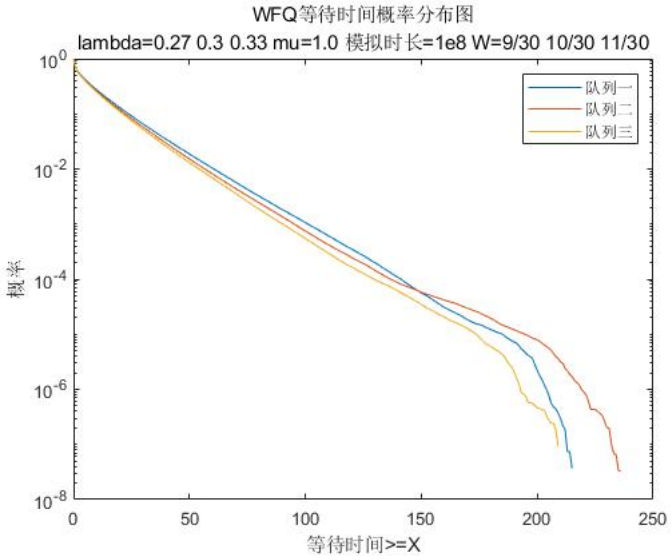

图 5: WFQ 等待时间概率分布

- 若 WFQ 队列为空，进行服务
- 更新虚拟时钟
- 将该包的虚拟开始服务时间作为事件点放入 FFS 事件队列。

当 FFS 完成一个包的服务时: 重新计算虚拟时钟速度并计算 FFS 的下一个事件点。当一个 WFQ 完成一个包的服务时：选择并调度所有队列中虚拟完成时间最短的包进行调度。

在参数 $\lambda = (0.27, 0.30, 0.33)$，$\mu = 1.0$,w= $(2.7/9, 3/9, 3.3/9)$，模拟时长 =1e8 的情况下，我们得到了图 5、6。得到的个队列的平均队列长度和平均等待时间如表 3 所示。

当各队列的权重和 λ 成正比的时候，可以最大限度地保证 WFQ 算法的公平性，此时各条队列的等待时间分布曲线和队长分布曲线接近重合。在后面的对比分析中，当权重和各队列的 λ 不成正比的时候，三条曲线则变成了分开状态。

队列	平均等待时间	平均队列长度
1	8.38861	2.5349
2	7.8591	2.65744
3	7.46534	2.7937

表 3: WFQ 队列结果

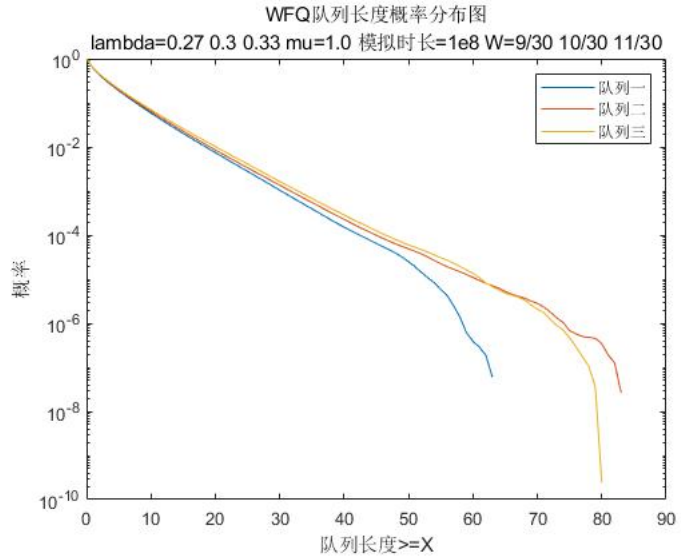

图 6: WFQ 队列长度概率分布

2.6 SCFQ(Self-clocked fair queueing)

Golestan[2] 提出了一个 WFQ 的一个简化算法，相较于 WFQ，SCFQ 在对 FFS 的模拟程度上差于 WFQ，但是时间复杂度则低于 WFQ。WFQ 的时间复杂度为 O(n)，SCFQ 为 O(logn)，DRR 为 O(1)。

队列	平均等待时间	平均队列长度
1	8.41517	2.54207
2	7.98261	2.69449
3	7.65693	2.85693

表 4: SCFQ 队列结果

在 WFQ 中，虚拟时钟是在 FFS 中更新的。SCFQ 算法做了下面两个简化：

- 仅跟踪 SCFQ 中包的到达和离开
- SCFQ 中每个包只需要更新一次虚拟时钟

SCFQ 的参数和 2.5 节中 WFQ 算法取值相同，图 7 图 8 为 SCFQ 的结果图，可以看到它和 WFQ 的算法结果相似，但是 SCFQ 的时间复杂度更低。本实验在同一机器上，统计了

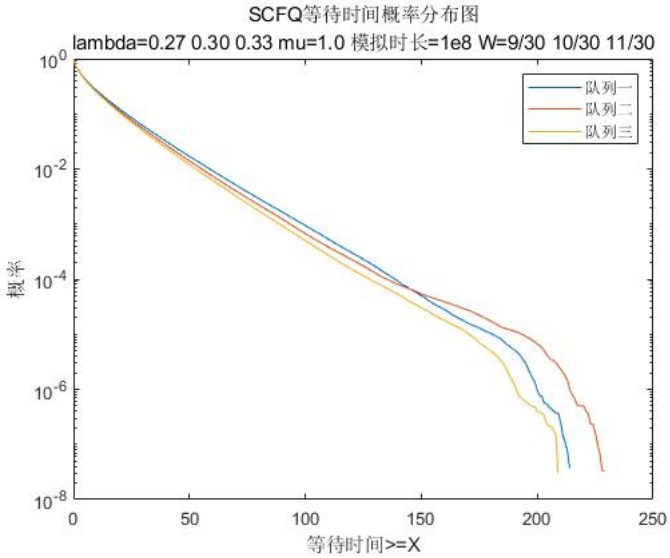

图 7: SCFQ 等待时间概率分布

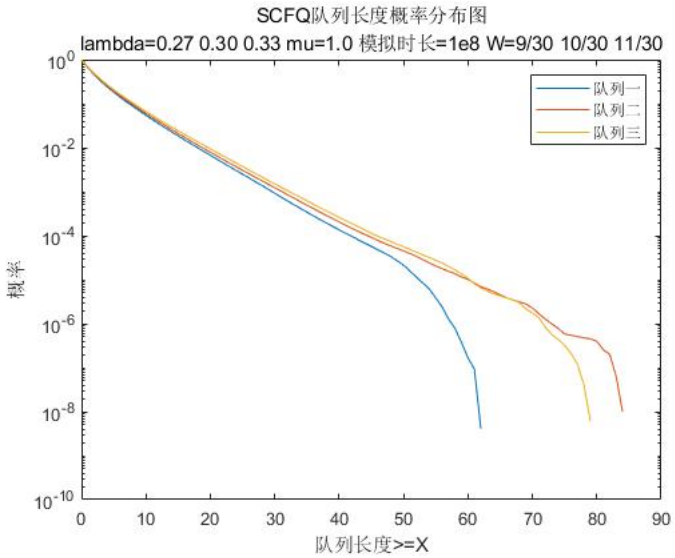

图 8: SCFQ 队列长度概率分布

WFQ 和 SCFQ 在 1e7 的情况下运行时间的差别，以此来粗略地展示两者时间复杂度的差别，此结果会在后文的对比分析一节中展示。

2.7 实验结果对比及分析

2.7.1 MM1 运行时长与精度关系

本实验中,MM1 的模拟时长分别取 1e7,1e8,1e9，其他参数保持不变。模拟仿真后得到图 9 和图 10。其中的红色曲线为 2.1 节中提到的理论结果曲线。通过图 9 和图 10 可以看到，当模拟时长为 1e7 的 $y < 10^{-5}$ 时候曲线在左右开始出现波动；当模拟时长为 1e8 的 $y < 10^{-6}$ 时候曲线在左右开始出现波动；当模拟时长为 1e9 的时候曲线在 $y < 10^{-7}$ 左右开始出现波动。

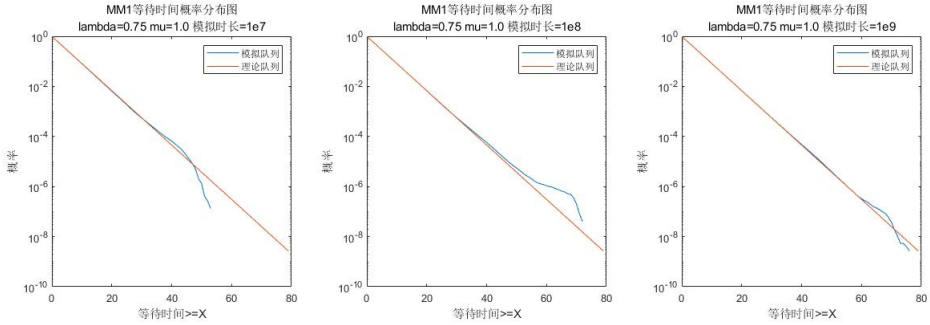

图 9: 不同模拟时长下 MM1 等待时间比较图

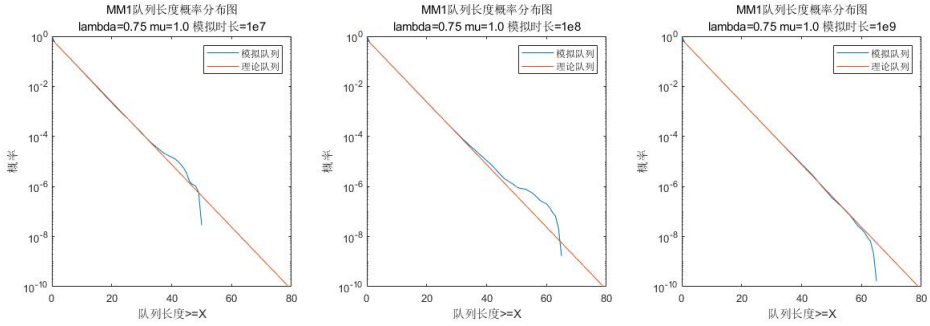

图 10: 不同模拟时长下 MM1 队列长度比较图

通过对比可知，在到达率 λ 和服务率 μ 保持不变的情况下，模拟时长越长，MM1 模拟结果的稳定性越高，包可能达到的等待时间最大值和队列长度最大值就会更大，但每个等待时间和队列长度出现的概率不变。

2.7.2 λ 与赤字量和权重的关系

图 11 和图 12 中 λ 参数分别为 (0.27,0.3,0.33)，模拟时长为 1e8。赤字量为 (5.4,6,6.6)，权重为 (2.7/9，3/9,3.3/9)，左中右分别为 DRR、WFQ、SCFQ 结果图。从三张对比图中可以看出，当赤字量和队列权重与 λ 成比例时，三条曲线接近重合，此时三条队列是公平的。

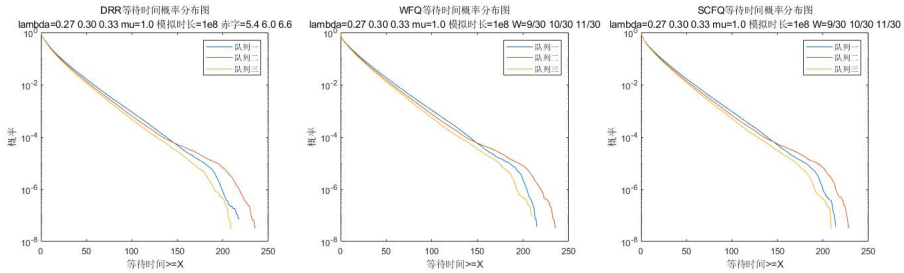

图 11: 等待时间比较图

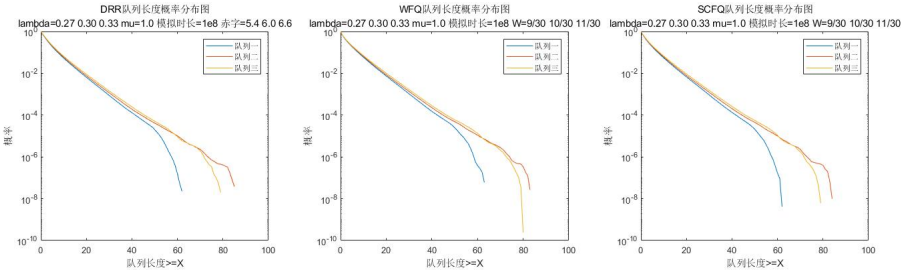

图 12: 队列长度比较图

2.7.3 λ 相同赤字不同结果分析

这里 DRR 的赤字分别为 (5.4 6.0 6.6)，(6.0 6.0 6.0)，(6.6 6.0 5.4)，其他参数保持不变。模拟仿真后得到图 13 和图 14。通过图 13 和图 14 可以看到，三组数据曲线都是在左右出现轻微抖动，说明赤字对队列的稳定性无影响。

从图 14 可以看出，当一个队列的 λ 越大，它的等待时间越长。对于等待时间分布图，当赤字量相同的时候，某队列 λ 越大，它的等待时间越长。但是，从图 13 中左图可以看出，当赤字量与 λ 成比例时，此时处在公平调度的状态，赤字量越大，该队列某一个包前面的包接受服务的速率越快，因而等待时间越短。

通过对比三组数据，可以得到在队列间赤字与 λ 呈现正相关的情况下，三队列的等待时间和队列长度的出现概率更接近。

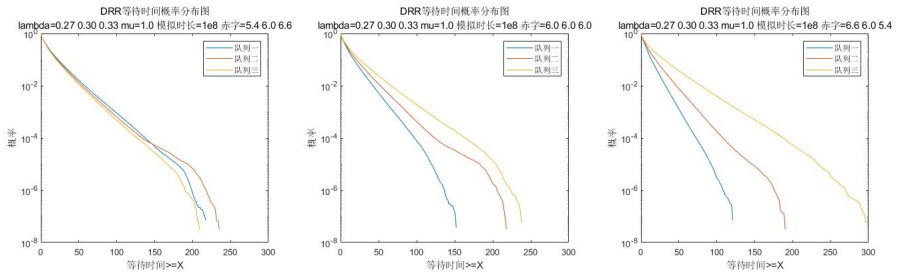

图 13: 不同赤字比例下 DRR 等待时间比较图

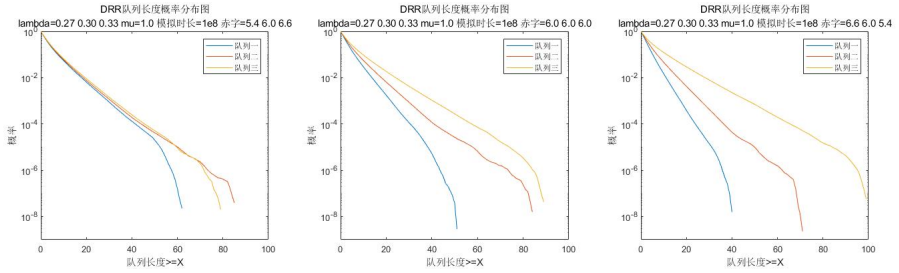

图 14: 不同赤字比例下 DRR 队列长度比较图

2.7.4 WFQ 不同权重 w 对比分析

这里 WFQ 的权重 w 分别为 (2.7/9 3/9 3.3/9)，(1/3 1/3 1/3)，(1/1 1/3 1/5)，其他参数保持不变。模拟仿真后得到图 15 和图 16。从图 15 和图 16 可以看出，当权重 w 和 λ 成正比的适合，三条曲线成重合状态。

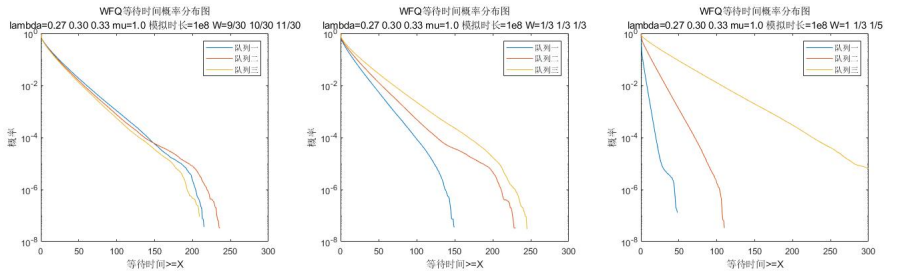

图 15: 不同 w 比例下 WFQ 等待时间比较图

当两图的中间结果可以看出，当队列权重相同的情况下，λ 越大，该队列受到的服务越大。如果对 λ 不同的队列设置了不合适的权重，会得到极端不公平的情况，如两图右边结果。

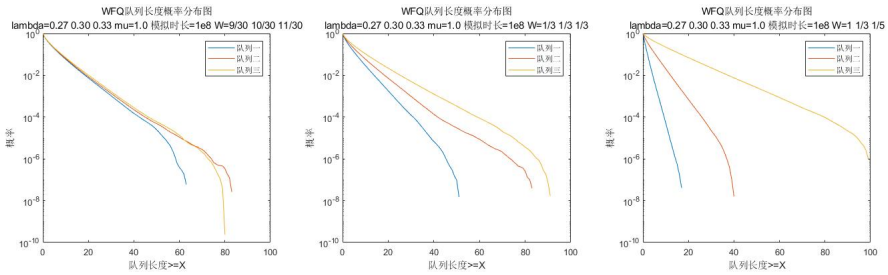

图 16: 不同 w 比例下 WFQ 队列长度比较图

2.7.5 SCFQ 不同 λ 对比分析

这置 SCFQ 的到达率 lambda 分别为 (0.27 0.30 0.33), (0.25 0.28 0.31), (0.23 0.26 0.29), 其他参数保持不变，权重均为 1/3。

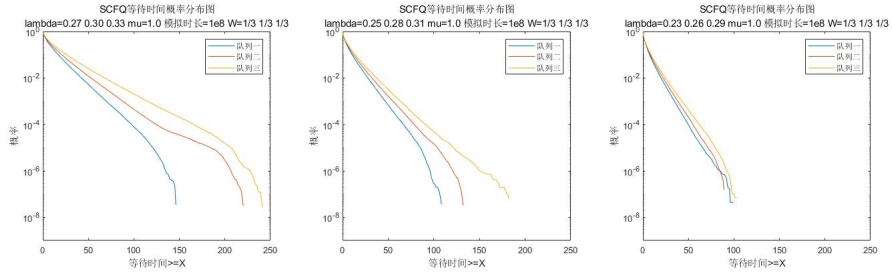

图 17: 不同 λ 下 SCFQ 等待时间比较图

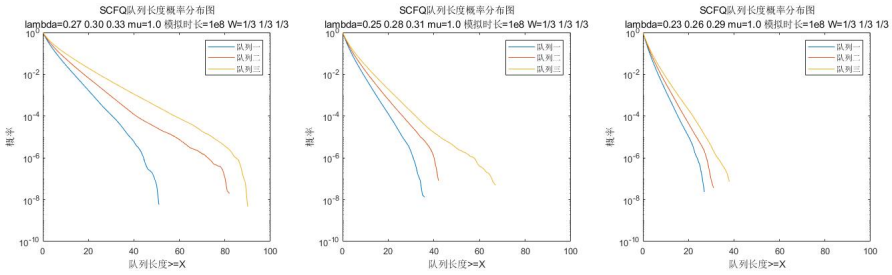

图 18: 不同 λ 下 SCFQ 队列长度比较图

通过图 17 和图 18 可知，在同组数据中，λ 越小的队列斜率越小，其等待时间和队列长度都相对较小。对于同队列在不同数据组中的表现，以队列一为例，随着 λ 的减小其斜率逐渐减小，说明对于不同数据下的同队列，其等待时间和队列长度也会受到 λ 的影响。

通过以上分析可以得出结论：在相同权重下，SCFQ 算法中队列到达率 λ 越小，队列的等待时间和队列长度都会相对呈现降低趋势，相应的，它们受到的服务量也会减少。

2.7.6 1e7 模拟时间下程序运行时间比较

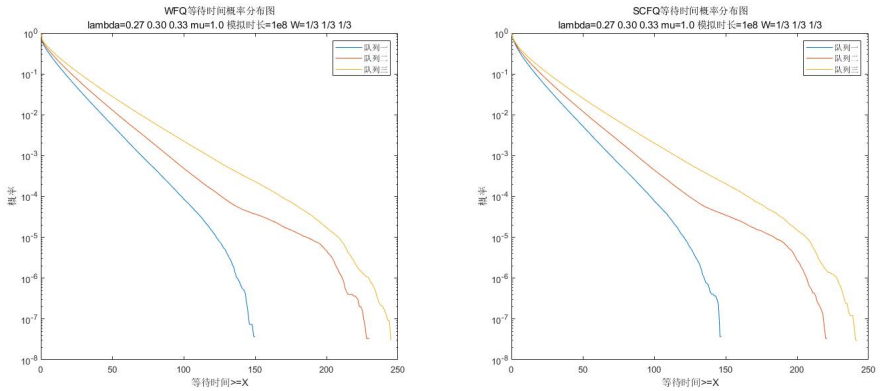

图 19: 相同参数下 DRR、WFQ、SCFQ 等待时间分布

WFQ 运行时间	SCFQ 运行时间
471.561s	135.026s

表 5: WFQ 与 SCFQ 运行时间比较

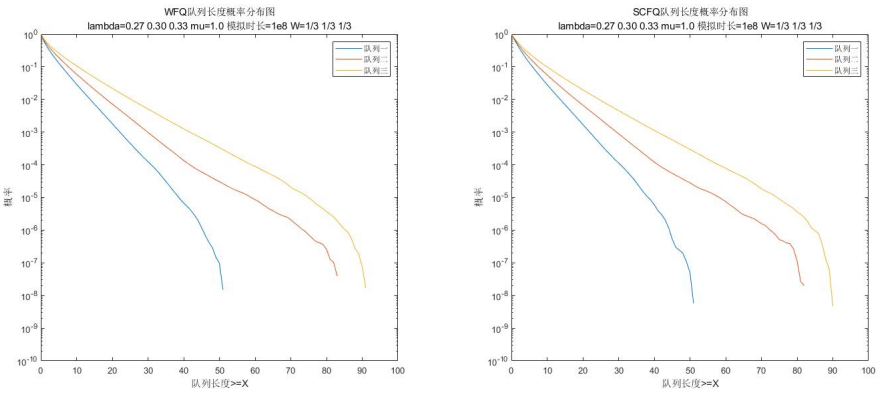

图 20: 相同参数下 WFQ、SCFQ 队列长度分布

通过对 WFQ 和 SCFQ 设置相同参数比较其在相似情况下的运行状况。从图 19 和图 20 可以看到，两种算法在数据基本保持一致的情况下其趋势大致相同，曲线斜率、曲线长度和波动情况都大致类似，说明这两种调度算法在参数一致的情况下对两队列的公平性影响大致相同。

为了尽快统计出两者运行时间，本节实验在 1e7 的情况下进行了统计。如果在模拟时间为 1e8 的情况下运行程序则需要 30-80 分钟。通过表 5 比较两种算法的运行时间，可以看到 CFQ 运行时间相比于 WFQ 有明显降低。这说明相近的参数下 SCFQ 运行时间更短，运行效率更高。

2.7.7 3 队列 λ 的和与 μ 关系

结果如图 21 所示，其中横轴为队长而纵轴为各队长的持续时间。由于程序无法模拟完成，故只取了部分前期数据。三个 λ 的和超过 0.97 倍 μ，产生拥塞现象，队列长度最大可达到 278 个，平均队列长度超过了 200 个包。由于队列模拟时，队列长度太大，需要更新的每个队列每个包数据持续增加，即便总模拟时间设为 1e6，经过 5h，也只达到了约 3e5 时长的队列模拟结果。

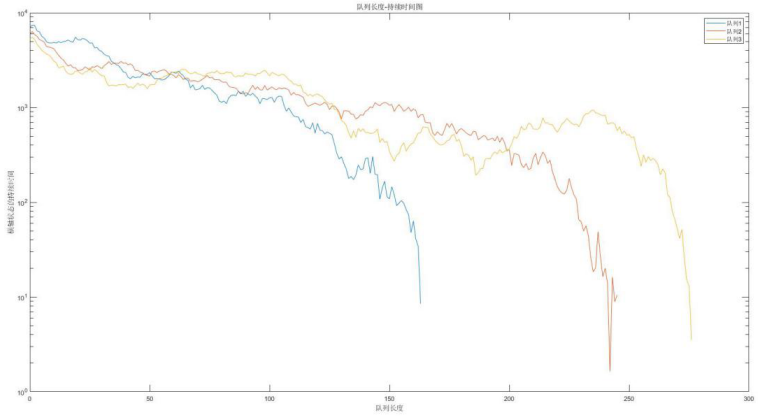

图 21: 各队队长的持续时间图

2.8 总结

WFQ、SCFQ、DRR 算法依次提出于 1993、1994、1995 年，顺着算法的发展过程，本小组对多队列公平调度问题在理论和程序上有了更深刻的理解。

多队列单服务台问题在日常生活中应用广泛，且有很好的应用前景。此类问题的关键是要保证接受服务地公平性, 本文所用度量标准能够解决这一关键问题。因此本文的多队列调度算法对解决现实生活中的非计算机网络问题同样具有参考价值。

3 多路访问协议研究

3.1 Slot Aloha

slot aloha 是在 pure Aloha 的基础上进行改进。纯 Aloha 较为简单：当用户有数据发送时就传输。但是纯 Aloha 的效果并不好，其最佳信道利用率仅为 18% 左右。

此后 Roberts 提出的 slot aloha 则可以将系统容量增加一倍。pure aloha 和 slot aloha 的主要区别为：pure aloha 的时间是连续的，而 slot aloha 的时间被分成时间槽，所有帧都必须同步到时间槽中，并且必须等到下一个时间槽的开始时刻才可以发送帧。

计算吞吐率, 效率，碰撞率的方法如下所示：吞吐量 = 成功发送包的数目/总尝试发送次数，效率 = 成功发送包的数/(槽大小 * 槽数目)，碰撞率 = 碰撞的次数/总尝试发送次数。通过程序模拟之后得到的各个站点及全网的指标如表 6:

队列	吞吐率	效率	碰撞率
1	0.983769	0.00197526	0.0162307
2	0.984452	0.00197556	0.0155485
3	0.984707	0.00197715	0.0152934
4	0.984324	0.00197784	0.0156764
5	0.984559	0.00197685	0.0154414
all	0.984155	0.00990554	0.0158447

表 6: Slot Aloha 结果

假定站产生的新包数可以模型化为一个平均每 slot time 产生 N 个包的泊松分布。如果 N>1, 则用户群生成包的速率大于信道的处理速率，因此，几乎每个帧都要经受冲突，为了取得合理的吞吐量，应该期望 0<N<1。

除了新生成的包，每个站还会产生由于先前遭受冲突而重传的包。在此基础上做进一步假设：在每个 slot 中，尝试发送的包和因碰撞未发送的包合起来也符合泊松分布，定义每 slot 平均发送的包的数目为 G，每 slot 成功发送的包的数目为 S，并且两者满足下面的式子：

$$S = Ge^{-G} \tag{18}$$

在代码中，我们采用下面的公式计算 S 和 G:

$$G = \frac{packets\_send\_times}{number\_of\_slot} \tag{19}$$

$$S = \frac{packets\_send\_sucess}{number\_of\_slot} \tag{20}$$

程序结果如表 7 所示，从该表可知，在站点数为 5 时每个 slot 平均发包数目为 G=0.02013，每个 slot 成功发包数目为 0.019811，根据第 18 式，可得到理论的 S 值为 0.0197289。两者的误差小于 0.0001，可认为该程序结果基本正确。并且从第 19 式和第 20 式可以得出 $throughput = \frac{S}{G}$。

G 值	S 值	理论 S 值	S/G
0.02013	0.0198111	0.0197289	0.98415

表 7: 站点数为 5 时的 S-G 值

3.2 Slot Aloha S-G 曲线研究

从 3.1 节的第 20 式得出 S 与效率的关系为，S 值 = 效率 * 时槽大小，在时槽大小不变的情况下，效率和 S 值成正比，而 S 值取决于 G 值，G 值和每 slot 的发包数有关。当站点数增加的时候，每个 slot 的发包数会上升。故本实验统计了站点数从 2 到 1000 的 S-G 值，得到如图 22 所示曲线。

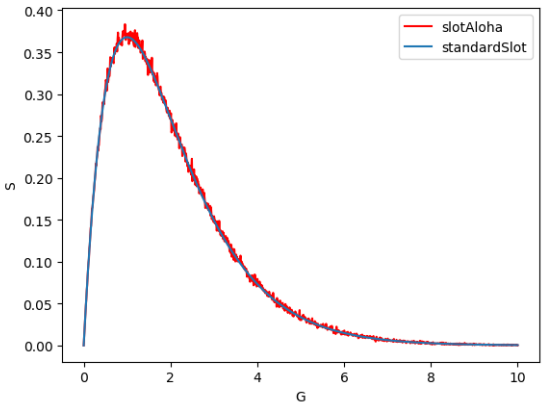

图 22: S-G 曲线

图 22 中蓝色曲线为理论结果，即 $S = Ge^{-G}$，红色曲线为程序运行结果，可以发现与理论值基本吻合。

slot aloha 的尖峰在 G=1 处，此时 S=1/e，大约为 0.368，是 pure aloha 的两倍。如果系统运行在 G=1 处，则空时间槽的概率为 0.368。使用 slot aloha，我们期望的最好结果是 37% 为空时间槽，37% 成功，剩下 26% 冲突。如果在更高的 G 值上运行，则空时间槽数会降低，但冲突时间槽数会成指数增长。

3.3 Slot Aloha 与其他算法比较

3.3.1 Slot Aloha 与 Pure Aloha

图 23 为 slot aloha 与 pure aloha 结果对比，其中蓝色曲线为理论的 pure aloha 的结果，即 $S = Ge^{-2G}$。pure aLoha 最大 S 值出现在当 G=0.5 时，S=1/2e, 大约等于 0.184。换句话说，最好信道利用率为 18%，而对于 slot aloha 来说，当 G=1.0 时，$S_{slot} = \frac{1}{e} = 2S_{pure} = 2\frac{1}{2e}$。

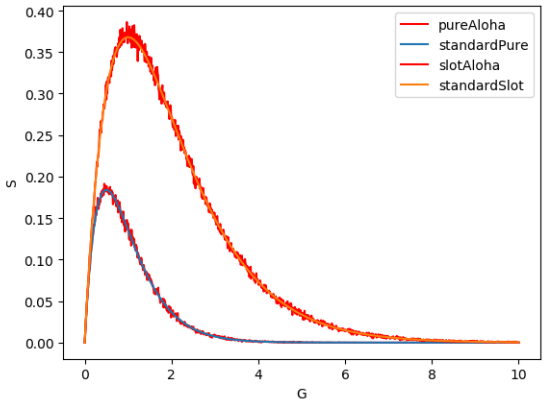

图 23: Pure ALOHA 与 Slot ALoha

3.3.2 Aloha 与坚持和非坚持 CSMA

1 坚持与非坚持结果如图 24 所示。在 CSMA 中，站点可以监听是否有传输，并据此采取相应动作。对于 1 坚持 CSMA，当一个站有数据要发送时，它首先侦听信道，确定当时是否有其他站正在发送数据。如果信道空闲，它就发送数据。否则，如果信道忙，该站等待直至信

道变成空闲；然后，站发送一帧，如果发生冲突，该站等待一段随机时间，然后从头开始上述过程。

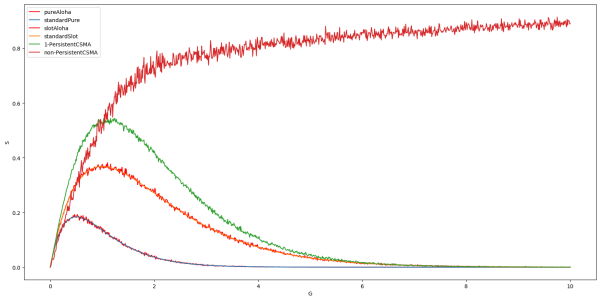

图 24: 不同随机访问协议曲线对比

对于 CSMA，如果信道当前正在使用中，则该站并不持续对信道进行监听，相反，它会等待一段随机时间，然后重复上述算法。因此，该算法会导致更好的信道利用率，但比起 1 坚持，也带来了更大的延迟。

3.4 802.11 协议模拟及分析

队列	吞吐率	效率	碰撞率
1	0.857706	0.00044147	0.142294
2	0.858234	0.00044387	0.141747
3	0.858881	0.00044253	0.141119
4	0.855901	0.00043704	0.144099
5	0.857569	0.00044254	0.142431
all	0.857662	0.00220745	0.142334

表 8: 协议仿真结果

802.11 协议仿真结果如表 8, 仿真步骤如下:

- 当某一站点需要竞争信道进行发送时，其首先需要等待 DIFS 时间，若 DIFS 时间内，信道保持空闲状态，那么就可以进行 backoff 过程
- 进入 backoff 过程时，其首先需要从竞争窗口选择一个随机数

- 在 backoff 过程中，每经过一个 slot time，节点会“监听”一次信道，若信道空闲，则相应的随机回退计数器的值减 1
- 当节点的随机倒数计数器倒数至 0 时，节点竞争获得信道，从而可以发送数据
- 当成功发送完数据，节点需要再次等待”DIFS 的时间后，重新开始 backoff 过程
- 如果发生碰撞，采用二进制指数退避的方法对竞争窗口 CW 进行扩展。在冲突之后，从扩大的竞争窗口中重新选择随机数，并进入 back off 过程。

3.4.1 回退次数与初始窗口大小对吞吐率的影响

图 25 为初始竞争窗口大小相同的情况下，回退次数对吞吐率的影响。图 26 为回退次数相同的情况下，初始竞争窗口对吞吐率的影响。图 27 为两者的相互作用对吞吐率的影响。

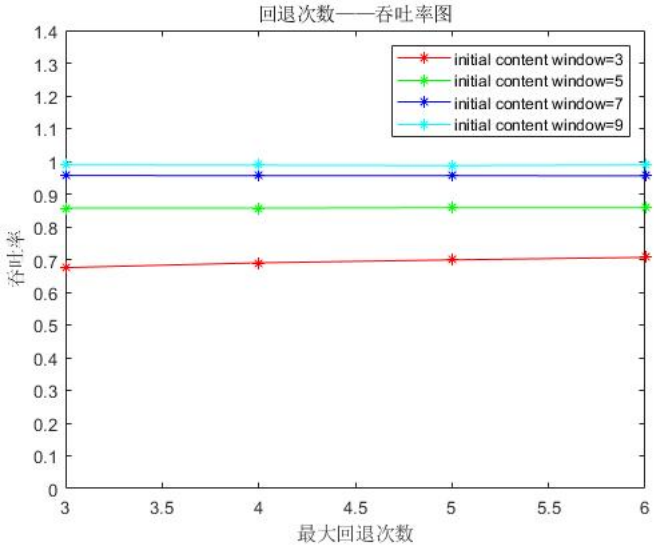

图 25: 回退次数对吞吐率的影响

从结果可以看出，初始竞争窗口比回退次数对吞吐率的影响更大，在冲突损失较小的情况下，没有必要设置更大的 CW 窗口大小，因为更大的窗口大小就意味着更多的回退次数，从而消耗更多的时间。并且，从图 26 中可知，随着初始竞争窗口的增大，曲线的斜率也逐渐降低，这意味着增加初始竞争窗口对吞吐率的影响逐渐降低。

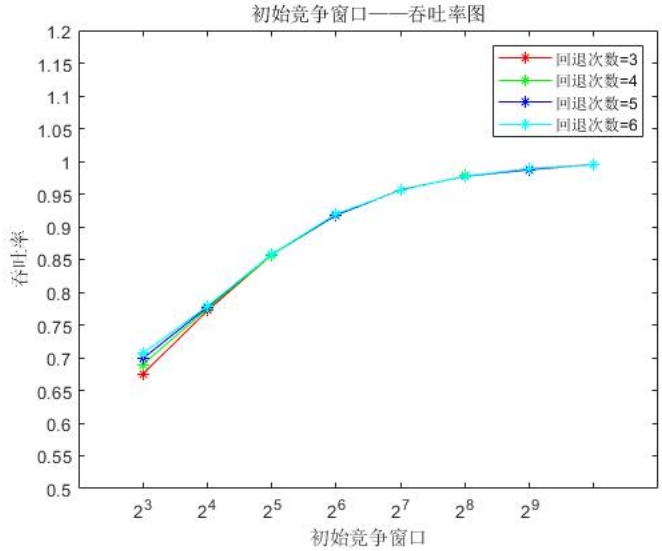

图 26: 初始窗口大小对吞吐率的影响

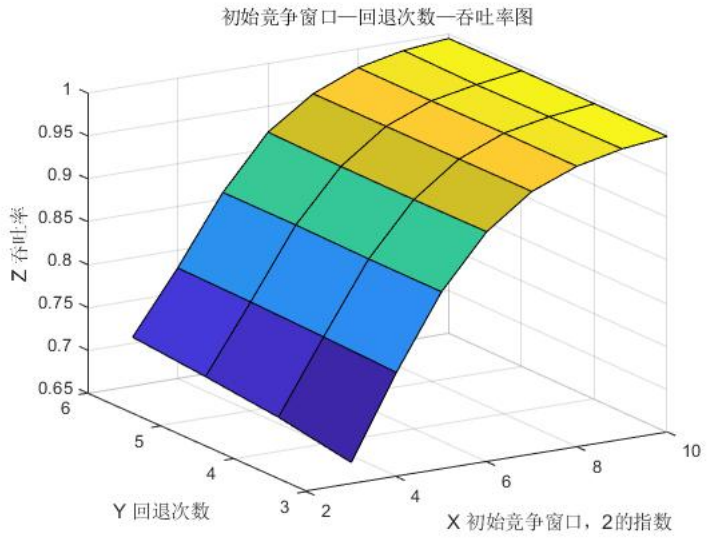

图 27: 回退次数与初始窗口大小对吞吐率的影响

3.4.2 回退次数与初始窗口大小对效率的影响

图 28 为初始竞争窗口大小相同的情况下，回退次数对效率的影响。图 28 为回退次数相同的情况下，初始竞争窗口对效率的影响。图 30 为两者的共同作用对效率的影响。

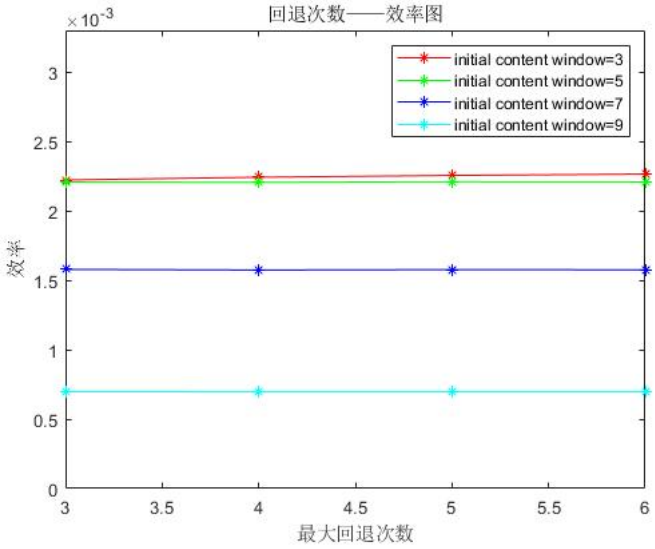

图 28: 回退次数对效率的影响

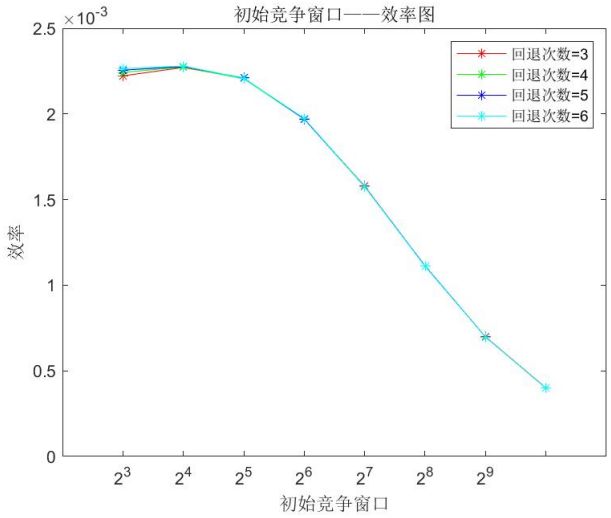

图 29: 初始窗口大小对效率的影响

从结果可以看出，当初始竞争窗口增大的时候，用在 back off 上的时间也就会更多，从而造成效率的下降，并且初始竞争窗口对效率和吞吐率的作用效果相反。

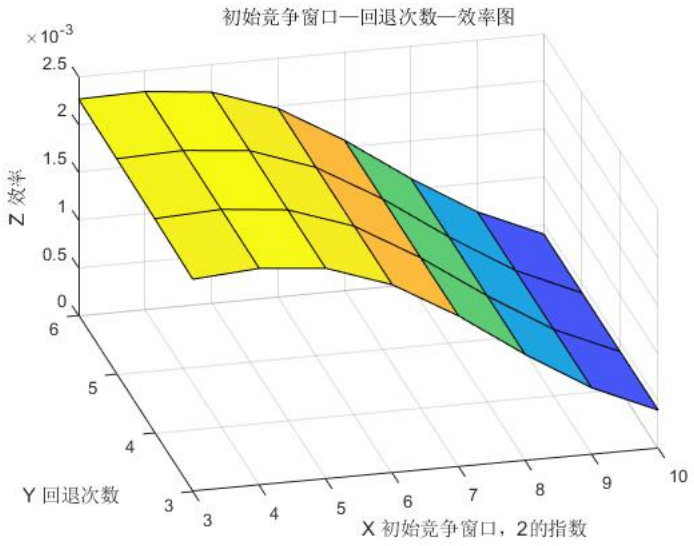

图 30: 回退次数与初始窗口大小对效率的影响

3.4.3 协议中的其他指标

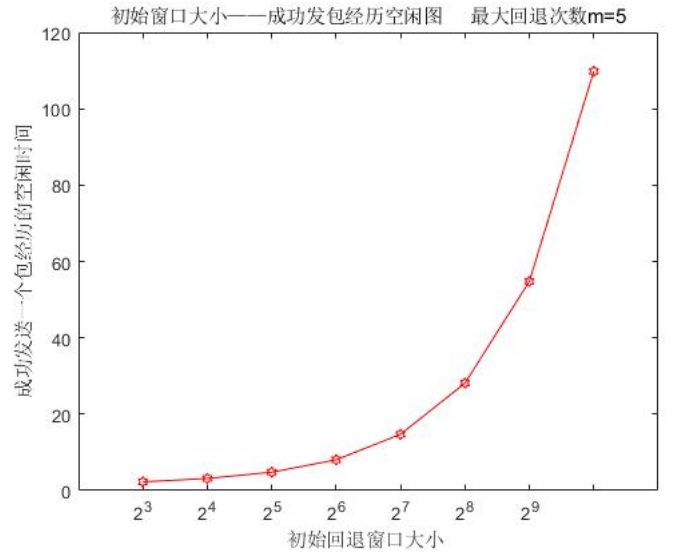

图 31: 站点数为 5 时每成功发送一个包空闲信道的时间与 CW 的关系

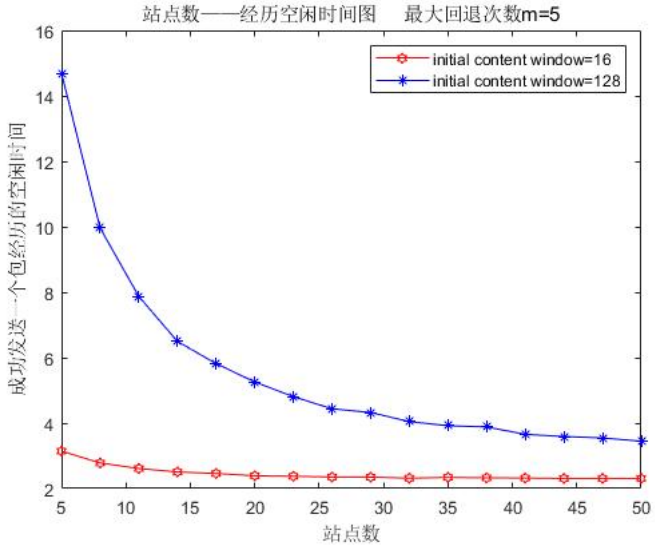

图 32: 站点数不同时每成功发送一个包空闲信道的时间与 CW 的关系

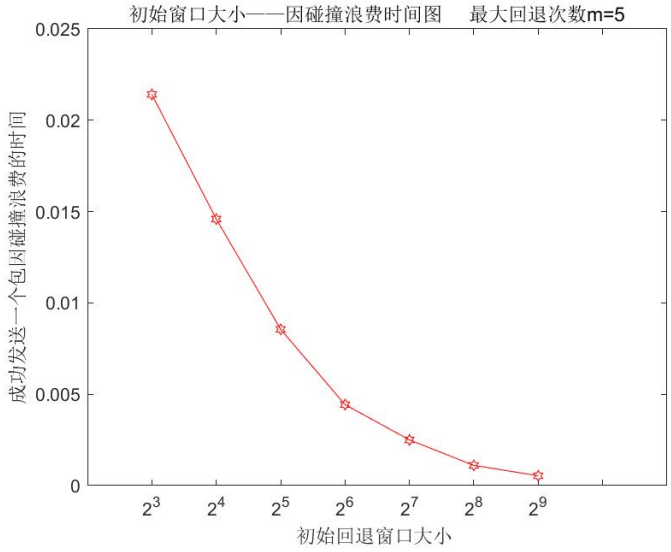

图 33: 站点数为 5 时浪费在碰撞上的时间与 CW 的关系

Giuseppe Bianchi[1] 在论文中使用了下式来分析协议的性能:

$$S = E[P]/[T_s + \sigma\frac{1 - P_t r}{P_s P_{tr}} + T_c(\frac{1}{P_s} - 1)] \tag{21}$$

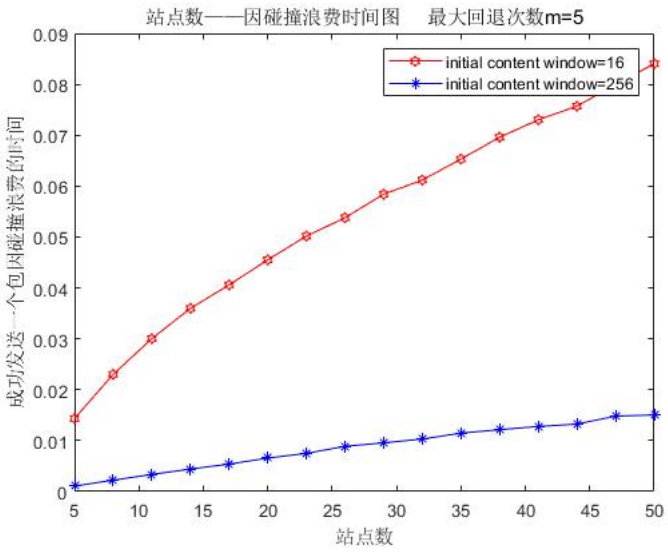

图 34: 站点数不同时浪费在碰撞上的时间与 CW 的关系

各参数意义如下:

- S 为归一化的吞吐量，具体定义如下:

$$S = \frac{E[payload \quad information \quad transmitted \quad in \quad a \quad slot \quad time]}{E[length \quad of \quad a \quad slot \quad time]} \tag{22}$$

- E[P]:payload 的期望, 代表数据包传输时间的均值。
- Ts: 若节点成功发送该数据，则损耗时间为 Ts。
- σ: 时槽大小
- P_{tr}: 发送概率
- P_s: 在有包发送的前提下发送成功的概率

式 21 的分母表示成功传输一个包所需要的信道时间，这个事件可以分为三个部分。

式 21 分母的第二项是每成功发送一个包花费在空闲信道上的时间。$1/(P_{tr}P_s)$ 是每成功发送一个包花费在信道上的时间。而在这 slot times 中,(1-Ptr) 则是空闲时间。

图 31、32 显示了每成功发送一个包所经历的空闲信道时间 (单位为 Slot time)，其中图 31 为站点数为 5 的情况下花费在空闲信道的时间与 CW 的关系，图 32 为站点数不同的情况下花费在空闲信道的时间与 CW 的关系，从图中可以看出初始 CW 越大，那么 back off 的时

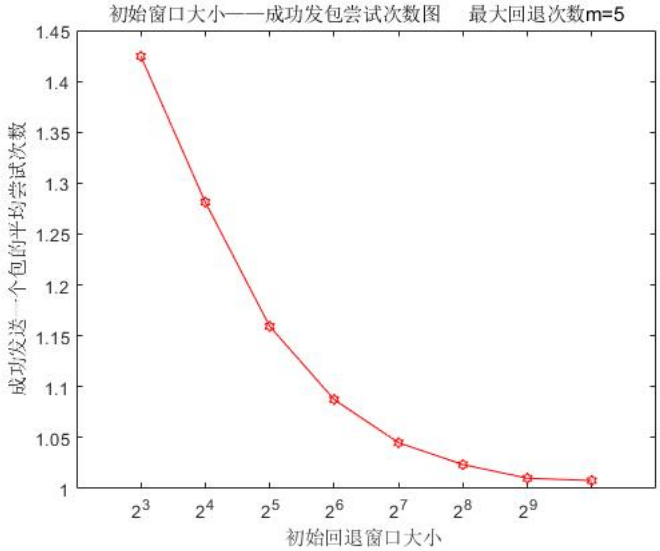

图 35: 每成功发送一个包所尝试的次数与初始 CW 的关系

间越长，每成功发送一个包花费在空闲信道上的时间槽数量越大。式 21 分母的第三项表示每成功传送一个包因为碰撞而在信道上浪费的时间，$1/P_s - 1$ 是每成功发送一个包的平均碰撞次数，乘以 Tc 就是浪费在碰撞上的时间。本实验中以 Tc 为单位绘制结果曲线。图 33 为站点数为 5 的情况下浪费在碰撞上的时间与 CW 的关系，图 34 为站点数不同的情况下浪费在碰撞上的时间与 CW 的关系。事实上，如果以 Tc 为单位的话，该曲线则可看做 (碰撞率-CW) 曲线。

图 35 显示了每成功发送一个包所尝试的次数与初始 CW 的关系，初始 CW 越小，尝试发送时更容易碰撞，尝试次数越多。

3.5 总结

在多路访问协议研究中,本文实现了 Pure Aloha,Slot Aloha,Persistent CSMA,non-Persistent CSMA 四种算法，并在此基础上模拟了 802.11 的 MAC 子层协议。本节实验参考了 Giuseppe Bianchi[1] 中的模型和评价方法分析了各参数对协议的影响

参考文献

[1] Giuseppe Bianchi. Performance analysis of the ieee 802.11 distributed coordination function. *IEEE Journal on selected areas in communications*, 18(3):535–547, 2000.

[2] S Jamaloddin Golestani. A self-clocked fair queueing scheme for broadband applications. In *Proceedings of INFOCOM'94 Conference on Computer Communications*, pages 636–646. IEEE, 1994.

[3] Abhay K Parekh. A generalized processor sharing approack to flow control in integrated services network. *IEEE/ACM TRANSACTIONS ON NETWORKING*, 1(3), 1993.

[4] Madhavapeddi Shreedhar and George Varghese. Efficient fair queueing using deficit round robin. In *Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*, pages 231–242, 1995.

A MM1 代码

```
//MM1算法实现
//author: 唐帅
#include <iostream>
#include <fstream>
#include <stdlib.h>
#include <math.h>
#include<random>
#include<algorithm>
#include <string>
#include<queue>
using namespace std;

uniform_real_distribution<float> randomFloats(0.0, 1.0);
default_random_engine generator;
double expon(double lambda)
{
        double u;
        do
        {
                u = randomFloats(generator);
        }
```

```
        while ((u == 0) || (u == 1));

        return -log(1-u)/lambda;
}

struct pack
{
        double size;
        double arrive_time;
        double leave_time;
        pack(double s, double a_t, double l_t)
        {
                size = s; arrive_time = a_t; leave_time = l_t;
        }
};

std::queue<pack>m_queue;

double lambda0, mu0;
double sim_time = 0;
double end_time = 1e8;

int next_event_type;

double time_next_event[2];

int num_events = 2;

double wait_time = 0;
double wait_gap = 1;

const int x_max = 1000;//横坐标数组大小

double q_lenth[x_max];
```

```
double q_wait[x_max];

double last_event_time = 0;
int num_of_come = 0;
void init()
{
        next_event_type = 0;
        time_next_event[0] = expon(lambda0);
        time_next_event[1] = time_next_event[0]+expon(mu0);

        m_queue.push(pack(1, time_next_event[0], time_next_event
            [1]));

        for (int i = 0; i < x_max; i++)
        {
                q_lenth[i] = 0;
                q_wait[i] = 0;
        }
}
void timing()
{
        double min_time_next_event = DBL_MAX;
        for (int i = 0; i < num_events; i++)
        {
                if (time_next_event[i] < min_time_next_event)
                {
                        min_time_next_event = time_next_event[i];
                        next_event_type = i;
                }
                sim_time = min_time_next_event;
        }
}
void switch_q()
{
```

```
        //单队列，无调度
}
void Arrive()
{
        if (m_queue.size() == 1)
                q_lenth[0] += sim_time - last_event_time;
        else
                q_lenth[m_queue.size() - 2] += sim_time -
                   last_event_time;

        last_event_time = sim_time;

        //此处注意是根据当前到达时间随机出下一个到达
        num_of_come++;
        time_next_event[next_event_type] = sim_time + expon(lambda0
           );
        m_queue.push(pack(1, time_next_event[next_event_type],
           DBL_MAX));
}
void Depart()
{
        q_lenth[m_queue.size()-2] += sim_time - last_event_time;
        last_event_time = sim_time;

        double time = m_queue.front().leave_time - m_queue.front().
           arrive_time;
        wait_time += time;
        q_wait[int(time / wait_gap)]+=1;
        m_queue.pop();

        //队列非空
        if (m_queue.front().arrive_time < time_next_event[
           next_event_type])
        {
```

```
                time_next_event[1] = time_next_event[1] + expon(mu0
                    );
        }
        else//队列空
        {
                time_next_event[1] = m_queue.front().arrive_time +
                    expon(mu0);
        }
        m_queue.front().leave_time = time_next_event[1];
}
int main()
{
        cout << "input lambda and mu" << endl;
        cin >> lambda0 >> mu0;

        cout << "运行中" << endl;

        init();

        while (sim_time < end_time)
        {
                timing();
                switch (next_event_type)
                {
                case 0:
                        Arrive();
                        break;
                case 1:
                        Depart();
                        break;
                }
        }

        double rho = lambda0 / mu0;
```

```
        cout << "等待时间理论值:" << 1 / (mu0 - lambda0) << endl;
        cout << "等待时间实际值:" << (wait_time / (float)
           num_of_come) << endl;

        double all = 0;
        double all_time = 0;

        for (int i = 0; i < x_max; i++)
        {
                all += q_lenth[i] * i;
                all_time += q_lenth[i];
        }

        cout << "队长理论值:" << all / all_time << endl;
        cout << "队长实际值:" <<(lambda0*lambda0) / (mu0*(mu0 -
           lambda0)) << endl;

        ofstream wait_time_distribution("wait_time_distribution.txt
           ");
        ofstream queue_length_distribution("
           queue_length_distribution.txt");

        for (int j = 0; j < x_max; j++)
        {
                wait_time_distribution << q_wait[j]/num_of_come <<
                   ' ';
                queue_length_distribution << q_lenth[j] / all_time
                   << ' ';
        }

        wait_time_distribution.close();
        queue_length_distribution.close();
}
```

B DRR 代码

```
//Deficit Round Robin算法实现
//author: 唐帅
#include <iostream>
#include <fstream>
#include <stdlib.h>
#include <math.h>
#include<random>
#include<algorithm>
#include <string>
#include<queue>
using namespace std;

uniform_real_distribution<float> randomFloats(0.0, 1.0);
default_random_engine generator;
double expon(double lambda)
{
        double u;
        do
        {
                u = randomFloats(generator);
        }
        while ((u == 0) || (u == 1));

        return -log(1-u)/lambda;
}

struct pack
{
        double size;
```

```
        double wait_time;
        pack(double s, double wait_t=0)
        {
                size = s;  wait_time = wait_t;
        }
        void add_wait_time(double t)
        {
                wait_time += t;
        }
};
vector<queue<pack>>m_queues;

double lambda[3];
double mu;

double sim_time = 0;
double end_time = 1e7;

//queue arrive:0 1 2 depature:3
int next_event_type;

double time_next_event[4];
int num_events = 4;

double all_wait_time[3];
double wait_gap = 1;

const int wait_x_max = 1000;
const int length_x_max = 100;

double queue_lenth[3][length_x_max];
double queue_wait[3][wait_x_max];

double last_event_time = 0;
```

```
int num_of_come[3];
int num_of_sent[3];

double quantum[3];

double deficit_counter[3];

bool isServerBusy = false;

int now_queue = 0;
int last_queue = 5;

double mina = 0;
void init()
{
        for (int i = 0; i < 3; i++)
        {
                for (int j = 0; j < wait_x_max; j++)
                {
                        queue_wait[i][j] = 0;
                }
                for (int j = 0; j < length_x_max; j++)
                {
                        queue_lenth[i][j] = 0;
                }

                all_wait_time[i] = 0;
                num_of_come[i] = 0;
                num_of_sent[i] = 0;

                deficit_counter[i] = 0;

                time_next_event[i]=expon(lambda[i]);
```

```
                queue<pack> q_temp;
                m_queues.push_back(q_temp);
        }
        time_next_event[3] = DBL_MAX;
}

void timing()
{

        double min_time_next_event = DBL_MAX;
        for (int i = 0; i < num_events; i++)
        {
                if (time_next_event[i] < min_time_next_event)
                {
                        min_time_next_event = time_next_event[i];
                        next_event_type = i;
                }
        }
        sim_time = min_time_next_event;
}

void switch_queue()
{
        bool isEmpty = true;//是否全部为空队列
        for (int i = 0; i < 3; i++)
        {
                if (!m_queues[i].empty())
                {
                        isEmpty = false;
                        break;
                }
        }
```

```
if (isEmpty)
        return;

while (!isServerBusy)
{
        if (now_queue != last_queue)
        {
                deficit_counter[now_queue] += quantum[
                   now_queue];
        }

        last_queue = now_queue;

        if (!m_queues[now_queue].empty())
        {
                double pack_size = m_queues[now_queue].
                   front().size;
                if (pack_size > mina)
                {
                        mina = pack_size;
                }
                if (pack_size <= deficit_counter[now_queue
                   ])
                {
                        deficit_counter[now_queue] -=
                           pack_size;

                        double wait_time = m_queues[
                           now_queue].front().wait_time;

                        queue_wait[now_queue][int(wait_time
                            / wait_gap)]++;

                        all_wait_time[now_queue] +=
```

```
                                wait_time;

                        time_next_event[3] = sim_time +
                                pack_size;

                        isServerBusy = true;

                        num_of_sent[now_queue]++;

                        return;
                }
        }
        else
        {
                deficit_counter[now_queue] = 0;
        }

        now_queue++;
        now_queue = now_queue % 3;

    }

}
void Arrive()
{

    for (int i = 0; i < 3; i++)
    {
            int queue_size = m_queues[i].size();

            for (int j = 0; j < queue_size; j++)
            {
                    pack pack_temp = m_queues[i].front();
```

```
                        m_queues[i].pop();

                        pack_temp.add_wait_time(sim_time -
                           last_event_time);

                        m_queues[i].push(pack_temp);
                }

                queue_lenth[i][m_queues[i].size()] += sim_time -
                   last_event_time;
        }

        last_event_time = sim_time;

        num_of_come[next_event_type]++;

        m_queues[next_event_type].push(pack(expon(mu)));

        time_next_event[next_event_type] = sim_time + expon(lambda[
           next_event_type]);

        switch_queue();

}
void Depart()
{
        for (int i = 0; i < 3; i++)
        {
                int queue_size = m_queues[i].size();

                for (int j = 0; j < queue_size; j++)
                {
                        pack pack_temp = m_queues[i].front();
```

```
                        m_queues[i].pop();

                        pack_temp.add_wait_time(sim_time -
                           last_event_time);

                        m_queues[i].push(pack_temp);
                }

                queue_lenth[i][m_queues[i].size()] += sim_time -
                   last_event_time;
        }

        m_queues[now_queue].pop();

        last_event_time = sim_time;

        isServerBusy = false;

        time_next_event[3] = DBL_MAX;

        switch_queue();
}
int main()
{
        cout << "input lambda 0 1 2" << endl;

        cin >> lambda[0] >> lambda[1] >> lambda[2];

        cout << "input mu" << endl;

        cin >> mu;

        cout << endl << "input quantum 0 1 2" << endl;
```

```
cin >> quantum[0]>> quantum[1]>> quantum[2];

if (lambda[0] + lambda[1] + lambda[2] > mu*0.95)
{
        cout << "wrong: lambda[0] + lambda[1] + lambda[2] >
            mu" << endl;
        cout << "please check Max-Min fairness" << endl;
}
cout << "running..." << endl;
init();
while (sim_time < end_time)
{
        timing();

        switch (next_event_type)
        {
        case 0:
        case 1:
        case 2:
                Arrive();
                break;
        case 3:
                Depart();
                break;
        }
}
cout << mina << endl;
ofstream wait_time_distribution;
ofstream queue_length_distribution;


for (int i = 0; i < 3; i++)
{
        cout << "queue " << i << " :" << endl;
```

```
cout << "num_of_sent:" << num_of_sent[i] << endl;

cout << "mean wait time:" << all_wait_time[i] /
   num_of_come[i]<< endl;

double all = 0;
double all_time = 0;

for (int j = 0; j < length_x_max; j++)
{
        all += queue_lenth[i][j] * j;

        all_time += queue_lenth[i][j];
}

cout << "mean queue length:" << all / all_time<<
   endl;

string file_name= "wait_time_distribution"+
   to_string(i)+".txt";;

wait_time_distribution.open(file_name);

for (int j = 0; j < wait_x_max; j++)
{
        wait_time_distribution << queue_wait[i][j]/
           num_of_sent[i] << ' ';
}
cout << queue_wait[i][0] << endl;

wait_time_distribution.close();

file_name= "queue_length_distribution"+to_string(i)
```

```
                +".txt";

            queue_length_distribution.open(file_name);

            for (int j = 0; j < length_x_max; j++)
            {
                    queue_length_distribution << queue_lenth[i
                        ][j] / all_time << ' ';
            }

            queue_length_distribution.close();
        }



        double res[3];
        cout << endl;
        for (int i = 0; i < 3; i++)
        {
                res[i] = num_of_sent[i] / lambda[i];
                cout << "w" << i << ": " << res[i] << endl;
        }

        double max_diff = -DBL_MAX;
        max_diff = (abs(res[0] - res[1]) > max_diff ? abs(res[0] -
            res[1]) : max_diff);
        max_diff = (abs(res[1] - res[2]) > max_diff ? abs(res[1] -
            res[2]) : max_diff);
        max_diff = (abs(res[2] - res[0]) > max_diff ? abs(res[2] -
            res[0]) : max_diff);
        cout << "max difference:" << max_diff << endl;
        cout << "max difference unit time:" << max_diff/end_time <<
             endl;
}
```

C WFQ 代码

```
//Weighted Fair Queueing算法实现
//author: 唐帅

#include <iostream>
#include <fstream>
#include <stdlib.h>
#include <math.h>
#include<random>
#include<algorithm>
#include <string>
#include<queue>
using namespace std;

uniform_real_distribution<float> randomFloats(0.0, 1.0);
default_random_engine generator;
double expon(double lambda)
{
        double u;
        do
        {
                u = randomFloats(generator);
        }
        while ((u == 0) || (u == 1));

        return -log(1-u)/lambda;
}

struct pack
{
        double packLen;
        double timestamp;
        double RemainingVirlength;
```

```
        double remainTtransmitTime;

        int flowID;

        double arrive_time;

        bool operator<(const pack &b)const
        {
                return timestamp > b.timestamp;
        }
};

vector<queue<pack>>FFS_queues;
priority_queue<pack>WFQ_queue;

double lambda[3];
double mu;

double end_time = 1e8;

//queue arrive:0 1 2 wfq depature:3 ffs depature:4
int next_event_type;

double time_next_event[5];
int num_events = 5;

double all_wait_time[3];
double wait_gap = 1;

const int wait_x_max = 1000;
const int length_x_max = 100;

double queue_lenth[3][length_x_max];
double queue_wait[3][wait_x_max];
```

```
double last_arrive_or_depart_time = 0;

int num_of_come[3];
int num_of_sent[3];

double w[3];

double sim_time = 0, prev_sim_time = 0, virtual_time = 0,
   prev_virtual_time = 0;

double virtual_time_rate=0;

double lastTimeStamp[3];

int real_time_queue_length[3];
double compute_VT_Rate()
{
        double r = 0.0;

        for (int i = 0; i < 3; i++)
                if ( FFS_queues[i].size() > 0 )
                        r += w[i];

        if ( r > 0 )
                return(1/r);
        else
                return(0.0);
}

void init()
{
        for (int i = 0; i < 3; i++)
        {
```

```
                for (int j = 0; j < wait_x_max; j++)
                {
                        queue_wait[i][j] = 0;
                }
                for (int j = 0; j < length_x_max; j++)
                {
                        queue_lenth[i][j] = 0;
                }

                all_wait_time[i] = 0;
                num_of_come[i] = 0;
                num_of_sent[i] = 0;

                time_next_event[i]=expon(lambda[i]);

                queue<pack> q_temp;
                FFS_queues.push_back(q_temp);

                lastTimeStamp[i] = 0;

                real_time_queue_length[i] = 0;
        }
        time_next_event[3] = DBL_MAX;
}

void update_process()
{
        for (int i = 0; i < 3; i++)
        {
                if (!FFS_queues[i].empty() && FFS_queues[i].front()
                   .RemainingVirlength > 0)
                {
                        FFS_queues[i].front().RemainingVirlength =
                           FFS_queues[i].front().RemainingVirlength
```

```
                                - virtual_time_rate * (sim_time -
                                    prev_sim_time);
                }
        }

        if (!WFQ_queue.empty())
        {
                pack temp=WFQ_queue.top();
                WFQ_queue.pop();
                temp.remainTtransmitTime -= (sim_time -
                   prev_sim_time);
                WFQ_queue.push(temp);
        }
}

void timing()
{
        prev_virtual_time = virtual_time;
        prev_sim_time = sim_time;

        virtual_time_rate = compute_VT_Rate();

        double wfq_departure = 0;

        if (!WFQ_queue.empty())
                wfq_departure = sim_time + WFQ_queue.top().
                   remainTtransmitTime;
        else
                wfq_departure = DBL_MAX;

        time_next_event[3] = wfq_departure;

        double ffs_departure=DBL_MAX;
```

```
for (int i = 0; i < 3; i++)
{

        if (!FFS_queues[i].empty())
        {
                double h = FFS_queues[i].front().
                   RemainingVirlength;

                h = h / virtual_time_rate;

                if (h < ffs_departure)
                        ffs_departure = h;
        }
}
ffs_departure += sim_time;

time_next_event[4] = ffs_departure;

double min_time_next_event = DBL_MAX;
for (int i = 0; i < num_events; i++)
{
        if (time_next_event[i] < min_time_next_event)
        {
                min_time_next_event = time_next_event[i];
                next_event_type = i;
        }
}

sim_time = min_time_next_event;

virtual_time = prev_virtual_time + virtual_time_rate * (
   sim_time - prev_sim_time);

update_process();
```

```
}


void Arrive()
{
        for (int i = 0; i < 3; i++)
        {
                queue_lenth[i][real_time_queue_length[i]] +=
                   sim_time - last_arrive_or_depart_time;
        }
        last_arrive_or_depart_time = sim_time;

        num_of_come[next_event_type]++;
        real_time_queue_length[next_event_type]++;

        pack pack_arrive;
        pack_arrive.packLen = expon(mu);
        pack_arrive.RemainingVirlength = pack_arrive.packLen / w[
           next_event_type];
        pack_arrive.timestamp = max(virtual_time, lastTimeStamp[
           next_event_type]) + pack_arrive.RemainingVirlength;
        pack_arrive.flowID = next_event_type;
        pack_arrive.remainTtransmitTime = pack_arrive.packLen;
        pack_arrive.arrive_time = sim_time;

        FFS_queues[next_event_type].push(pack_arrive);

        WFQ_queue.push(pack_arrive);

        lastTimeStamp[next_event_type] = pack_arrive.timestamp;

        time_next_event[next_event_type] = sim_time + expon(lambda[
           next_event_type]);
}
```

```
void FFS_Depart()
{
        for (int i = 0; i < 3; i++)
        {
                if (!FFS_queues[i].empty())
                {
                        if (FFS_queues[i].front().
                           RemainingVirlength<0.0000001)
                        {
                                FFS_queues[i].pop();
                        }
                }
        }
}
void WFQ_Depart()
{
        int queue_id = WFQ_queue.top().flowID;

        double temp_wait_time = sim_time - WFQ_queue.top().
           arrive_time-WFQ_queue.top().packLen;
        all_wait_time[queue_id] += temp_wait_time;

        queue_wait[queue_id][int(temp_wait_time / wait_gap)]++;

        num_of_sent[queue_id]++;

        for (int i = 0; i < 3; i++)
        {
                queue_lenth[i][real_time_queue_length[i]] +=
                   sim_time - last_arrive_or_depart_time;
        }

        last_arrive_or_depart_time = sim_time;
```

```
        real_time_queue_length[queue_id]--;

        WFQ_queue.pop();
}
int main()
{
        cout << "input lambda 0 1 2" << endl;

        for (int i = 0; i < 3; i++)
                cin >> lambda[i];

        cout << "input mu" << endl;

        cin >> mu;

        cout << "输入队列权重 w 0 1 2的分母(如输入3会转化为1/3):"
           << endl;

        double a;
        for (int i = 0; i < 3; i++)
        {
                cin >> a;
                w[i] = 1 / a;
        }

        if (lambda[0] + lambda[1] + lambda[2] > mu*0.95)
        {
                cout << "wrong: lambda[0] + lambda[1] + lambda[2] >
                    mu" << endl;
                cout << "please check Max-Min fairness" << endl;
        }
        cout << "running..." << endl;
        init();
        while (sim_time < end_time)
```

```
{
        timing();
        switch (next_event_type)
        {
        case 0:
        case 1:
        case 2:
                Arrive();
                break;
        case 3:
                WFQ_Depart();
                break;
        case 4:
                FFS_Depart();
        }
}
ofstream wait_time_distribution;
ofstream queue_length_distribution;

for (int i = 0; i < 3; i++)
{
        cout << "queue " << i << " :" << endl;

        cout << "num_of_sent:" << num_of_sent[i] << endl;

        cout << "mean wait time:" << all_wait_time[i] /
           num_of_come[i]<< endl;

        double all = 0;
        double all_time = 0;

        for (int j = 0; j < length_x_max; j++)
        {
                all += queue_lenth[i][j] * j;
```

```
                all_time += queue_lenth[i][j];
        }

        cout << "mean queue length:" << all / all_time<<
           endl;

        string file_name= "wait_time_distribution"+
           to_string(i)+".txt";;

        wait_time_distribution.open(file_name);

        for (int j = 0; j < wait_x_max; j++)
        {
                wait_time_distribution << queue_wait[i][j]/
                   num_of_sent[i] << ' ';
        }
        cout << queue_wait[i][0] << endl;

        wait_time_distribution.close();

        file_name= "queue_length_distribution"+to_string(i)
           +".txt";

        queue_length_distribution.open(file_name);

        for (int j = 0; j < length_x_max; j++)
        {
                queue_length_distribution << queue_lenth[i
                   ][j] / all_time << ' ';
        }

        queue_length_distribution.close();
}
```

```
        double res[3];
        cout << endl;
        for (int i = 0; i < 3; i++)
        {
                res[i] = num_of_sent[i] / lambda[i];
                cout << "w" << i << ": " << res[i] << endl;
        }

        double max_diff = -DBL_MAX;
        max_diff = (abs(res[0] - res[1]) > max_diff ? abs(res[0] -
           res[1]) : max_diff);
        max_diff = (abs(res[1] - res[2]) > max_diff ? abs(res[1] -
           res[2]) : max_diff);
        max_diff = (abs(res[2] - res[0]) > max_diff ? abs(res[2] -
           res[0]) : max_diff);
        cout << "max difference:" << max_diff << endl;
        cout << "max difference unit time:" << max_diff/end_time <<
            endl;
}
```

D SCFQ 代码

```
//Self-clocked Fair Queueing算法实现
//author: 唐帅
#include <iostream>
#include <fstream>
#include <stdlib.h>
#include <math.h>
#include<random>
#include<algorithm>
#include <string>
#include<queue>
```

```
using namespace std;

uniform_real_distribution<float> randomFloats(0.0, 1.0);
default_random_engine generator;
double expon(double lambda)
{
        double u;
        do
        {
                u = randomFloats(generator);
        }
        while ((u == 0) || (u == 1));

        return -log(1-u)/lambda;
}

struct pack
{
        double packLen;
        double timestamp;

        int flowID;

        double arrive_time;
        double remainTtransmitTime;
        bool operator<(const pack &b)const
        {
                if (timestamp == b.timestamp)
                {
                        return arrive_time > b.arrive_time;
                }

                return timestamp > b.timestamp;
        }
```

```
};

priority_queue<pack>SCFQ_queue;

double lambda[3];
double mu;

double end_time = 1e8;

//queue arrive:0 1 2 wfq depature:3 ffs depature:4
int next_event_type;

double time_next_event[4];
int num_events = 4;

double all_wait_time[3];
double wait_gap = 1;

const int wait_x_max = 1000;
const int length_x_max = 100;

double queue_lenth[3][length_x_max];
double queue_wait[3][wait_x_max];

double last_arrive_or_depart_time = 0;

int num_of_come[3];
int num_of_sent[3];

double w[3];

double sim_time = 0, virtual_time = 0, pre_sim_time = 0;

double lastTimeStamp[3];
```

```
int real_time_queue_length[3];
void init()
{
        for (int i = 0; i < 3; i++)
        {
                for (int j = 0; j < wait_x_max; j++)
                {
                        queue_wait[i][j] = 0;
                }
                for (int j = 0; j < length_x_max; j++)
                {
                        queue_lenth[i][j] = 0;
                }

                all_wait_time[i] = 0;
                num_of_come[i] = 0;
                num_of_sent[i] = 0;

                time_next_event[i]=expon(lambda[i]);

                lastTimeStamp[i] = 0;

                real_time_queue_length[i] = 0;
        }
        time_next_event[3] = DBL_MAX;
}
void update_process()
{
        if (!SCFQ_queue.empty())
        {
                pack temp=SCFQ_queue.top();
                SCFQ_queue.pop();
                temp.remainTtransmitTime -= (sim_time -
```

```
                    pre_sim_time);
                SCFQ_queue.push(temp);
        }
}
void timing()
{
        pre_sim_time = sim_time;

        double scfq_departure = 0;

        if (!SCFQ_queue.empty())
                scfq_departure = sim_time + SCFQ_queue.top().
                    remainTtransmitTime;
        else
                scfq_departure = DBL_MAX;

        time_next_event[3] = scfq_departure;


        double min_time_next_event = DBL_MAX;
        for (int i = 0; i < num_events; i++)
        {
                if (time_next_event[i] < min_time_next_event)
                {
                        min_time_next_event = time_next_event[i];
                        next_event_type = i;
                }
        }

        sim_time = min_time_next_event;
        update_process();
}

double S(double t)
```

```
{
        double r = 0.0;

        for (int i = 0; i < 3; i++)
                if ( real_time_queue_length[i] > 0 )
                        r += w[i];

        if ( r > 0 )
                return(1/r);
        else
                return(0.0);
}

void Arrive()
{
        if(!SCFQ_queue.empty())
                virtual_time = SCFQ_queue.top().timestamp;

        for (int i = 0; i < 3; i++)
        {
                queue_lenth[i][real_time_queue_length[i]] +=
                   sim_time - last_arrive_or_depart_time;
        }
        last_arrive_or_depart_time = sim_time;

        num_of_come[next_event_type]++;
        real_time_queue_length[next_event_type]++;

        pack pack_arrive;
        pack_arrive.packLen = expon(mu);
        pack_arrive.timestamp = max(virtual_time, lastTimeStamp[
           next_event_type]) + pack_arrive.packLen/w[
           next_event_type];
        pack_arrive.flowID = next_event_type;
```

```
        pack_arrive.arrive_time = sim_time;
        pack_arrive.remainTtransmitTime = pack_arrive.packLen;

        SCFQ_queue.push(pack_arrive);

        lastTimeStamp[next_event_type] = pack_arrive.timestamp;

        time_next_event[next_event_type] = sim_time + expon(lambda[
           next_event_type]);
}
void SCFQ_Depart()
{
        int queue_id = SCFQ_queue.top().flowID;

        double temp_wait_time = sim_time - SCFQ_queue.top().
           arrive_time-SCFQ_queue.top().packLen;
        all_wait_time[queue_id] += temp_wait_time;

        queue_wait[queue_id][int(temp_wait_time / wait_gap)]++;

        num_of_sent[queue_id]++;

        for (int i = 0; i < 3; i++)
        {
                queue_lenth[i][real_time_queue_length[i]] +=
                   sim_time - last_arrive_or_depart_time;
        }

        last_arrive_or_depart_time = sim_time;

        real_time_queue_length[queue_id]--;

        SCFQ_queue.pop();
```

```
        time_next_event[3] = DBL_MAX;
}
int main()
{
        cout << "input lambda 0 1 2" << endl;

        for (int i = 0; i < 3; i++)
                cin >> lambda[i];

        cout << "input mu" << endl;

        cin >> mu;

        cout << "输入队列权重 w 0 1 2的分母(如输入3会转化为1/3):"
           << endl;

        double a;
        for (int i = 0; i < 3; i++)
        {
                cin >> a;
                w[i] = 1 / a;
        }
        if (lambda[0] + lambda[1] + lambda[2] > mu*0.95)
        {
                cout << "wrong: lambda[0] + lambda[1] + lambda[2] >
                    mu" << endl;
                cout << "please check Max-Min fairness" << endl;
        }
        cout << "running..." << endl;

        init();
        while (sim_time < end_time)
        {
                timing();
```

```
        switch (next_event_type)
        {
        case 0:
        case 1:
        case 2:
                Arrive();
                break;
        case 3:
                SCFQ_Depart();
                break;
        }
}
ofstream wait_time_distribution;
ofstream queue_length_distribution;

for (int i = 0; i < 3; i++)
{
        cout << "queue " << i << " :" << endl;

        cout << "num_of_sent:" << num_of_sent[i] << endl;

        cout << "mean wait time:" << all_wait_time[i] /
           num_of_come[i]<< endl;

        double all = 0;
        double all_time = 0;

        for (int j = 0; j < length_x_max; j++)
        {
                all += queue_lenth[i][j] * j;

                all_time += queue_lenth[i][j];
        }
```

```
            cout << "mean queue length:" << all / all_time<<
               endl;

            string file_name= "wait_time_distribution"+
               to_string(i)+".txt";;

            wait_time_distribution.open(file_name);

            for (int j = 0; j < wait_x_max; j++)
            {
                    wait_time_distribution << queue_wait[i][j]/
                       num_of_sent[i] << ' ';
            }
            cout << queue_wait[i][0] << endl;

            wait_time_distribution.close();

            file_name= "queue_length_distribution"+to_string(i)
               +".txt";

            queue_length_distribution.open(file_name);

            for (int j = 0; j < length_x_max; j++)
            {
                    queue_length_distribution << queue_lenth[i
                       ][j] / all_time << ' ';
            }

            queue_length_distribution.close();
    }

    double res[3];
    cout << endl;
    for (int i = 0; i < 3; i++)
```

```
        {
                res[i] = num_of_sent[i] / lambda[i];
                cout << "w" << i << ": " << res[i] << endl;
        }

        double max_diff = -DBL_MAX;
        max_diff = (abs(res[0] - res[1]) > max_diff ? abs(res[0] -
           res[1]) : max_diff);
        max_diff = (abs(res[1] - res[2]) > max_diff ? abs(res[1] -
           res[2]) : max_diff);
        max_diff = (abs(res[2] - res[0]) > max_diff ? abs(res[2] -
           res[0]) : max_diff);
        cout << "max difference:" << max_diff << endl;
        cout << "max difference unit time:" << max_diff/end_time <<
            endl;
}
```

E 协议仿真代码

```
#include <string>
#include <vector>
#include<iostream>
using namespace std;

#ifndef SIMNODE_H_
#define SIMNODE_H_

#define ulong unsigned long
#define uint  unsigned int

#define DOWN_TIMER              0
#define UP_TIMER                1
```

```
#define IDLE                    0
#define BUSY                    1

#define IDLE_STATE              0
#define BACKOFF_STATE           1
#define SEND_STATE              2
#define RECEIVE_STATE           3
#define COLLISION_STATE         4
#define ACKTIMEOUT_STATE        5

#define Halt                    0
#define Not_Halt                1

#define PHY                     (208)
#define MAC                     (224)
#define ACK                     (112)

#define PAYLOAD (1500*8)
#define MRate (54)

#define ulRAND_MAX              4294967295

class SimNode {

public:
        ulong  CURRENT_TIME;
        ulong  ALARM_TIME;

        uint   TIMER_MODE;
        uint   WORK_MODE;

        uint   IP;

        uint   CWmin;
```

```
        uint   CWmax;
        uint   CW;
        uint   CW_COUNTER;

        uint   SLOT;
        uint   SIFS;
        uint   DIFS;
        uint   PHY_TIME;
        uint   TRANS;

        uint   PACKET_COUNTER;
        uint   DROP_COUNTER;

        double RATE;

        uint send_sucess;
        uint send_all_num;
        uint coli_times;

        uint Halt_flag;
        SimNode(int min,int max);
        virtual ~SimNode();

        unsigned long ulrand(void);

        void inital(int ip);
        void run(int clock_in,int flag_channel);
        void timer(int flag_channel);
        void set_alarm(ulong alarm_time);

};

#include "SimNode.h"
#include "stdio.h"
```

```
#include "stdlib.h"
#include "time.h"
#include "math.h"
#include "assert.h"

#include <string>
using namespace std;

#define uint  unsigned int
#define ulong unsigned long

SimNode::SimNode(int min,int max) {

        CURRENT_TIME    = 0;
        ALARM_TIME      = 0;

        TIMER_MODE      = 0;
        WORK_MODE       = 0;

        IP              = 0;

        //CWmin           = 31;
        //CWmax          = 1024;
        CWmin = pow(2, min) - 1;
        CWmax = pow(2, max);
        CW              = CWmin;
        CW_COUNTER      = 0;

        SLOT            = 20;
        SIFS            = 10;
        DIFS            = SIFS + 2*SLOT;
        PHY_TIME        = 20;
        TRANS               = 0;
```

```
        RATE            = MRate;

        PACKET_COUNTER  = 0;
        DROP_COUNTER    = 0;

        Halt_flag       = Not_Halt;


        send_sucess = 0;
        send_all_num = 0;
        coli_times = 0;
}

SimNode::~SimNode(){

}

void SimNode::inital(int ip){
        IP       = ip;
}

void SimNode::run(int clock_in,int flag_channel){

        CURRENT_TIME = clock_in;

        switch(TIMER_MODE){
        case DOWN_TIMER:
        {
                switch(WORK_MODE){
                case IDLE_STATE:
                        assert(CW_COUNTER <= 0);
                        CW_COUNTER = ulrand() % CW;
                        set_alarm(DIFS);
                        WORK_MODE = BACKOFF_STATE;
```

```
        break;
case BACKOFF_STATE:
        if(CW_COUNTER <= 0){
                WORK_MODE = SEND_STATE;
                send_all_num++;
        }
        else if(CW_COUNTER > 0){
                if(Halt_flag == Halt){
                        set_alarm(10);
                        Halt_flag = Not_Halt;
                }
                else if(Halt_flag == Not_Halt){
                        CW_COUNTER = CW_COUNTER -
                           1;
                        set_alarm(SLOT);
                }
        }
        break;
case SEND_STATE:
        TRANS = DIFS + 2 * PHY_TIME + double(MAC +
           PAYLOAD + ACK) / RATE + SIFS;
        set_alarm(TRANS);
        WORK_MODE = RECEIVE_STATE;
        break;
case RECEIVE_STATE:
        PACKET_COUNTER = PACKET_COUNTER + 1;
        send_sucess++;
        CW              = CWmin;
        WORK_MODE       = IDLE_STATE;
        break;
case COLLISION_STATE:
        TRANS = DIFS + 2 * PHY_TIME + double(MAC +
           PAYLOAD + ACK) / RATE + SIFS;
        set_alarm(TRANS);
```

```
                        WORK_MODE = ACKTIMEOUT_STATE;
                        break;
                case ACKTIMEOUT_STATE:
                        WORK_MODE = IDLE_STATE;
                        break;
                }
                break;
        }
        case UP_TIMER:
        {
                this->timer(flag_channel);
                break;
        }
        }
}

unsigned long SimNode::ulrand(void) {
        return (
                (((unsigned long)rand()<<24)&0xFF000000ul)
                |(((unsigned long)rand()<<12)&0x00FFF000ul)
                |(((unsigned long)rand()    )&0x00000FFFul));
}

void SimNode::timer(int flag_channel){
        if(WORK_MODE==BACKOFF_STATE || WORK_MODE==IDLE_STATE)
                if(flag_channel==BUSY){
                        ALARM_TIME = 1 + CURRENT_TIME;
                        Halt_flag = Halt;
                }
        if(CURRENT_TIME >= ALARM_TIME){
                TIMER_MODE = DOWN_TIMER;
        }
}
```

```
void SimNode::set_alarm(ulong alarm_time){
        ALARM_TIME = alarm_time + CURRENT_TIME;
        TIMER_MODE = UP_TIMER;
}



#include <vector>

#define ulong unsigned long

#include "SimSchedule.h"
#include<iostream>
using namespace std;
int main()
{

        int    allNum=5;
        const ulong  sim_time  = 1e7;
        //cin >> allNum;

        sim_main(allNum,sim_time,5,10);
        //trace.close();
        return EXIT_SUCCESS;
}
```