深圳大学实验报告

课程名称		计算机网络 			
实验名称		实验 2: 码率自适应传输			
学	院 _	计算机与软件学院			
专		软件工程(腾班)			
指导教师 _		张磊			
报告	人 _	黄亮铭	学 号 	2022155028	
实验时间 2024年3月15日~2024年4月10			24年4月10日		
提交时间 2024年4月10			10 日		

教务处制

一、 实验目的与要求

- 1. 阅读并理解 ABR 算法。
- 2. 复现一个 ABR 算法,根据网络带宽、缓冲区容量等因素,自适应选择视频段的码率并请求。
- 3. 完成试验任务:引入网络波动、理解简单 ABR 算法和改进 ABR 算法。

二、 实验过程

- 1. **引入网络波动:** 下载实验所需要的库, 引入 fluctuation.h 头文件, 根据功能解析模块中的 Network Fluctuation, 修改源代码, 模拟网络传输中的带宽波动。
 - a) 在服务端引入 fluctuation.h 头文件;

```
1 #include "fluctuation.h"
2
```

图 1: 引入头文件

b) 在代码初始化阶段调用 load_fl()函数初始化网络波动相关信息;

```
33 /***初始化阶段***/
34 load_fl();
```

图 2: 调用 load fl()函数

c) 更改原文件发送函数 send()为 send_fl(),参数一致;

```
bytes_sent = send_fl(new_socket, (char *)&file_size_buf, INT_SIZE, 0);

int tmp_send_count = send_fl(new_socket, buffer, buf_len, 0);

bytes_sent = send_fl(new_socket, &s_stop_byte, sizeof(s_stop_byte), 0);
```

图 3: 更改服务端文件函数名

d) 在结束阶段调用 release_fl()函数释放为网络波动模拟模块所分配的系统资源;

```
164 release_fl();
165
166 return 0;
```

图 3: 调用 release fl()函数

2. 简单 ABR 算法理解

这个简单的自适应比特率(ABR)算法的主要目标是根据当前网络状况和播放器缓冲情况,动态地选择适当的视频码率,以提供最佳的观看体验

在这个简单 ABR 算法中,核心是动态调整视频码率,以平衡视频质量和网络带宽。当网络条件较好时,选择较高分辨率以提供更好的观看体验;而当网络条件较差时,选择较低分辨率以避免缓冲和卡顿现象。同时,通过等待一段时间来为下一个视频段制定码率决策,可以更好地适应网络状况的变化,提高整体的观看体验。

但是,在当前版本的代码框架中,需要等待一段时间才能为下一个视频段制定码率决策,这可能会导致不必要的等待时间。要优化这一点,可以考通过**异步处理或并行处理**来实现这一优化,以确保在等待视频段下载的同时,仍可以进行码率决策的制定。

ABR 算法的主要步骤为:初始化下载队列,然后循坏请求视频段,同时统计每个视频段的下载时间,将视频段交付播放器并进行 QoE 记录和码率自适应决策。

3. 改进 ABR 算法

a) 简单的 ABR 算法

简单的码率自适应决策为:如果当前视频段的下载时间小于200ms,则以1080p的分辨率下载新的视频段;如果当前视频段下载时间不小于200ms但小于600ms,则以480p的分辨率下载新的视频段;否则以360p的分辨率下载新的视频段。

根据上述的决策描述,我实现的代码如下图所示。

strcat(temp, suffix);

//简单的码率自适应决策

```
int next_id=video_id+2; //间隔一个chunk进行下载
if(next id<=VIDEO LEN-1){</pre>
    if(download_time<200){</pre>
       char temp[REQUEST SIZE]="ocean-1080p-8000k";
       char suffix[6]="'
       sprintf(suffix, "-%d.ts", next id);
       strcat( temp, suffix);
        QueuePush(&download_queue,temp);
                                           //将决策结果入队
    }else if(download_time<600){</pre>
        char temp[REQUEST_SIZE]="ocean-480p-2500k";
        char suffix[6]="
        sprintf(suffix, "-%d.ts", next_id);
        strcat( temp, suffix);
        QueuePush(&download queue,temp); //将决策结果入队
    }else{
        char temp[REQUEST_SIZE]="ocean-360p-1000k";
        char suffix[6]=""
        sprintf(suffix, "-%d.ts", next_id);
```

图 4: 简单 ABR 算法

QueuePush(&download_queue,temp); //将决策结果入队

b) 改进的 ABR 算法 I

通过查看我们需要下载的视频的文件,我们发现视频源中存在720p分辨率的视频文件。而简单的码率自适应决策中忽略了720p分辨率的视频文件,客户端只有1080p、480p和360p三种分辨率的视频文件可以选择。因此,在改进ABR算法I中,我们加入了720p分辨率的视频文件供客户端选择。

改进的 ABR 算法 I 的码率自适应决策为:如果当前视频段的下载时间小于 200ms,则以 1080p 的分辨率下载新的视频段;如果当前视频段下载时间不小于 200ms 但小于 400ms,则以 720p 的分辨率下载新的视频段;如果当前视频段下载时间不小于 400ms 但小于 600ms,则以 480p 的分辨率下载新的视频段;否则以 360p 的分辨率下载新的视频段。

根据上述的决策描述, 我修改了简单 ABR 算法的代码, 新增了720p 分辨率的选择分支。

```
//简单的码率自适应决策
int next_id=video_id+2; //间隔一个chunk进行下载
if(next id<=VIDEO LEN-1){</pre>
    if(download_time<200){</pre>
        char temp[REQUEST_SIZE]="ocean-1080p-8000k";
        char suffix[6]="
        char suffix[6]="";
sprintf(suffix, "-%d.ts", next_id); |
        strcat( temp, suffix);
        QueuePush(&download_queue, temp);
                                             //将决策结果入队
    }else if (download time<400){</pre>
        char temp[REQUEST_SIZE]="ocean-720p-5000k";
        char suffix[6]="'
        sprintf(suffix, "-%d.ts", next_id);
        strcat( temp, suffix);
        QueuePush(&download queue, temp);
                                             //将决策结果入队
    }else if(download_time<600){</pre>
        char temp[REQUEST_SIZE]="ocean-480p-2500k";
        char suffix[6]=""
        char suffix[6]="";
sprintf(suffix, "-%d.ts", next_id);
        strcat( temp, suffix);
        QueuePush(&download_queue,temp);
                                             //将决策结果入队
    }else{
        char temp[REQUEST_SIZE]="ocean-360p-1000k";
        char suffix[6]=""
        sprintf(suffix, "-%d.ts", next id);
        strcat( temp, suffix);
        QueuePush(&download_queue,temp);
                                             //将决策结果入队
```

图 5: 改进 ABR 算法 I

c) 改进的 ABR 算法 II (基于缓存补偿的视频码率自适应算法)

算法策略: 在初始阶段, 客户端始终请求低于当前带宽值的码率等级; 在一段时间后, 视频缓存时长累积至上切阈值, 码率决策模块请求高于当 前带宽值的码率级别; 随时间推移, 缓存时长消耗至下切 阈值以下, 码 率决策模块逐级切换码率等级并重新累积缓存时长。

不妨设视频缓存时长最大为 q_max,显然最小值为 0。进一步在视频缓存时长内划分缓存阈值,当缓存时长低于 q_min 时,客户端将请求最低码率以快速积累缓存市场;当缓存时长高于 q_up 时,客户端将以最高码率请求视频分片,提高画面质量。以最高码率请求视频分片时缓存进入消耗状态并向动态下切阈值 q_down 运动,当缓存时长 低于 q_down 时,码率决策模块将逐级切换至当前带宽下的最高 码率,缓存再次进入累积状态。



图 6: 视频缓存时长

算法设计:

1) 带宽预估: 取前 M (代码实现中为 4) 个实际下载速率的均值为下一时刻的带宽预估值(除去当前下载速率)记为

$$\begin{cases} \hat{b} = \frac{1}{M-1} \sum_{i=2}^{M} b_{k-i} \\ b_{\text{down}} = \hat{b} \times \theta \end{cases}, b_{\text{down}}$$
 , b_{down} 为带宽衰减阈值。若当前分片下载速率低于 b_{down} ,则以 b_{down} 作为下一时刻带宽预估值。若当前分片下

低于 b_{down} ,则以 b_{down} 作为下一时刻带宽预估值。若当前分片下 载速率大于 \hat{b} ,则以前 M (代码实现中为 4) 个实际下载速率的均 值为下一时刻的带宽预估值。否则, 我们对前 M 个视频分片的实 际下载速率赋权重,然后计算得出下一时刻带宽预估值。权重计

 $w_{{\scriptscriptstyle k-i}} = \frac{w(1-w)^i}{1-(1-w)^{{\scriptscriptstyle M}}}; \ i \in \{1,2,\cdots,M\}$, 下一时刻带 算公式如下:

宽预估值为: $\tilde{b} = \sum_{i=1}^{M} b_{k-i} w_{k-i}$ 。综上所述,带宽预估值定义如下:

$$\tilde{b} = \begin{cases} \frac{1}{M} \sum_{i=1}^{M} b_{k-i}, & b_{k-1} \ge \hat{b} \\ \sum_{i=1}^{M} b_{k-i} w_{k-i}, & b_{\text{down}} \le b_{k-1} < \hat{b} \\ b_{k-1}, & b_{k-1} < b_{\text{down}} \end{cases}$$

2) 码率自适应:首先我们设置 q min 和 B max(q min 定义同前文, B_max 定义为缓存上限),然后计算出切换码率所需要的时间:

$$t = \frac{1}{\tilde{b}} \sum_{j=tar}^{cur} R_j \times \tau \tag{6}$$

其中: \tilde{b} 是带宽预估值, R_{cur} 为当前请求的码率, R_{tur} 为要切换 到的目标码率,τ是每个视频分片的播放时长。在计算出所 用时间后,进一步定义码率下切阈值。

然后依次计算出 q_{down} 和 q_{up} :

$$\begin{cases} \alpha = \frac{R_{\text{cur}}}{R_{\text{max}}} \\ q_{\text{down}} = q_{\text{min}} + (1 + \alpha)t \end{cases} \qquad q_{\text{up}} = B_{\text{max}} \times \frac{cur}{L+1} \times \beta$$

β为上切阈值调整参数,取值为 0.85 时效果最好。

算法实现:

因为代码较长,这里只截图核心部分的代码: 预测带宽值得 getNextDownLoadRate()函数和根据预测带宽值选择不同码率的视频 分片的 getNextBitRate()函数。其余函数见代码压缩包或实验报告附页。

```
double getNextDownloadRate()

44□ {

45□ double next_download_rate

46□ long double sum = 0;

47□ int cnt = 0;

48□ for (int i = rear - 1; i
         double next_download_rate_hat = 0;
         int cnt = 0;
for (int i = rear - 1; i > font && i > rear - 5; i--) {
    sum += download_rates[i];
    cnt += 1;
double mean_download_rate = 0;
if (cnt == 0) {
    mean_download_rate = 8000;
         else {
    mean_download_rate = sum / cnt;
         double down_download_rate = mean_download_rate * 0.4;
         if (download_rates[rear] >= mean_download_rate) {
    next_download_rate_hat = (mean_download_rate * cnt + download_rates[rear]) / (cnt + 1) + 0.5;
         for (int i = 0, j = rear; i < MAX_LEN && j > font; i++, j--) {
    tmp_download_rate += w[i] * download_rates[j];
             next_download_rate_hat = tmp_download_rate + 0.5;
            next_download_rate_hat = down_download_rate;
         return next_download_rate_hat;
                            6: getNextDownLoadRate()
125 int getNextBitRate(double buffer_time)
126⊟ {
127
              int now_grade = getNowGrade();
128
             int result_grade = now_grade + 1 > 3 ? 3 : now_grade + 1;
             double next_download_rate = getNextDownloadRate();
//printf("PREDICT RATE: %lf BUFFER_TIME: %lf\n", next_do
//printf("BUFFER_TIME: %lf\n", buffer_time);
129
130
131
             predict_rate[p_idx++] = next_download_rate;
132
             if (buffer_time < getQMin()) {</pre>
133<u>-</u>
134
135
                   cur_grade = 0;
136
                   return R[0];
137
138⊟
              else if (buffer_time > getQUp(now_grade)) {
139
                   cur_grade += 1;
140
                   if (cur_grade > 3) cur_grade = 3;
141
                   return R[cur_grade];
142
143 🗐
             else if (next_download_rate > R[result_grade]) {
                   cur_grade += 1;
if (cur_grade > 3) cur_grade = 3;
return R[cur_grade];
144
145
146
147
             else if (next_download_rate < R[now_grade]) {</pre>
148
                   if (buffer_time > getQDown()) {
149⊟
150
                         return R[now_grade];
151
152□
                    else {
153
                         cur_grade -= 1;
154
                         if (cur_grade < 0) cur_grade = 0;</pre>
155
                         return R[cur_grade];
156
157
158
             else {
                   return R[now_grade];
159
160
161 }
```

图 7: getNextBitRate()

三、实验结果

1. 简单的 ABR 算法

运行简单的 ABR 算法进行测试,评分结果如下图。

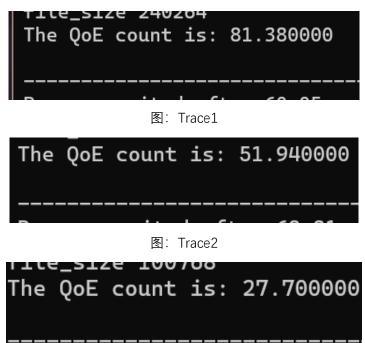


图: Trace3

2. 改进的 ABR 算法 I

运行改进 ABR 算法 I 进行 QoE 指标结算, 评分结果如下图:

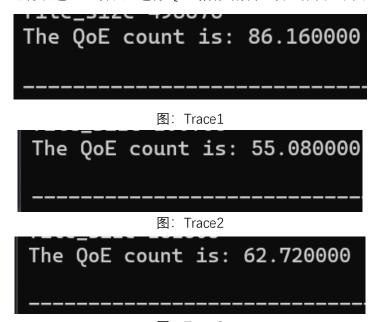


图: Trace3

3. 改进的 ABR 算法 Ⅱ

如下图所示(左:视频分片序号,中:实际带宽,右:预估带宽),改进 ABR 算法 || 对视频带宽值的预估非常准确,即使是在带宽突变的情况下,改进 ABR 算法 || 对带宽的预估情况仍然较好。

01	TIMES:	1276.56	510.62
02	TIMES:	423.53	344.89
03	TIMES:	411.33	302.94
04	TIMES:	421.85	227.69
05	TIMES:	331.26	180.38
96	TIMES:	212.89	153.91
97	TIMES:	218.24	107.83
80	TIMES:	110.13	81.83
09	TIMES:	104.32	71.57
10	TIMES:	108.38	108.19
11	TIMES:	107.94	64.59
12	TIMES:	106.10	119.44
13	TIMES:	153.32	145.81
14	TIMES:	213.87	173.06
15	TIMES:	216.95	200.13
16	TIMES:	214.38	126.29
17	TIMES:	203.26	258.43
18	TIMES:	397.11	312.47
19	TIMES:	433.12	467.48
20	TIMES:	834.44	622.49
21	TIMES:	823.31	723.45
22	TIMES:	800.93	825.51
23	TIMES:	841.36	1028.59
24	TIMES:	1646.74	1603.04
25	TIMES:	3121.14	1963.34
26	TIMES:	2242.11	1046.71
27	TIMES:	969.68	844.39
28	TIMES:	347.96	474.63

图: 带宽预估 (Trace1)

运行改进 ABR 算法 I 进行 QoE 指标结算,评分结果如下图:

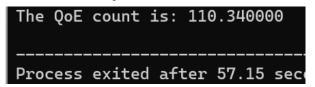


图: Trace1 (平均)

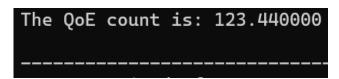


图: Trace1 (最佳)

如下图所示(左:视频分片序号,中:实际带宽,右:预估带宽),改进ABR 算法 || 对视频带宽值的预估非常准确,即使是在带宽突变的情况下,

改进 ABR 算法 II 对带宽的预估情况仍然较好。

20	TIMES:	223.76	208.76
21	TIMES:	222.84	228.88
22	TIMES:	272.66	258.31
23	TIMES:	311.96	289.97
24	TIMES:	350.43	351.09
25	TIMES:	467.31	448.53
26	TIMES:	662.44	911.43
27	TIMES:	2163.56	3306.02
28	TIMES:	9928.75	1700.63
29	TIMES:	1320.51	1788.38
30	TIMES:	1086.39	1644.75
31	TIMES:	550.74	394.35
32	TIMES:	315.51	260.35
33	TIMES:	212.33	153.52
34	TIMES:	195.02	121.21
35	TIMES:	176.18	108.55
36	TIMES:	170.07	100.70
37	TIMES:	155.78	89.16
38	TIMES:	128.80	77.85
39	TIMES:	110.43	71.48
40	TIMES:	111.13	119.63
41	TIMES:	126.14	121.31
42	TIMES:	135.55	128.83
43	TIMES:	140.50	143.72
44	TIMES:	170.70	154.63
45	TIMES:	169.75	167.67
46	TIMES:	187.74	180.68

图: 带宽预估 (Trace2)

运行改进 ABR 算法 I 进行 QoE 指标结算,评分结果如下图:

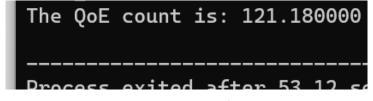


图: Trace2 (平均)

The QoE count is: 274.410000 01 TIMES: 508.09 201.7 02 TIMES: 231.89 172.4

图: Trace2 (最佳)

如下图所示 (左: 视频分片序号,中: 实际带宽,右: 预估带宽),改进 ABR 算法 || 对视频带宽值的预估非常准确,即使是在带宽突变的情况下,

改进 ABR 算法 II 对带宽的预估情况仍然较好。

32	TIMES:	489.74	293.44
33	TIMES:	484.24	294.36
34	TIMES:	490.66	247.45
35	TIMES:	321.09	172.80
36	TIMES:	170.01	136.31
37	TIMES:	168.95	110.91
38	TIMES:	168.48	169.98
39	TIMES:	170.48	170.06
40	TIMES:	170.33	100.91
41	TIMES:	164.19	169.00
42	TIMES:	169.01	101.06
43	TIMES:	167.41	168.66
44	TIMES:	172.03	101.99
45	TIMES:	168.09	101.14
46	TIMES:	166.03	101.38
47	TIMES:	168.55	171.97
48	TIMES:	183.19	6440.48
49	TIMES:	25242.13	18010.91
50	TIMES:	46447.75	27473.38
51	TIMES:	38018.45	36685.07
52	TIMES:	37029.94	23017.96
53	TIMES:	37539.35	22465.29
54	TIMES:	37405.94	21592.12
55	TIMES:	34349.42	14572.63
56	TIMES:	502.38	9634.36
57	TIMES:	97.82	4659.95
58	TIMES:	97.67	83.24
59	TIMES:	97.73	58.73

图: 带宽预估 (Trace3)

运行改进 ABR 算法 I 进行 QoE 指标结算,评分结果如下图:

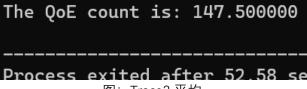


图: Trace3 平均

The QoE count is: 350.250000

图: Trace3 最佳

四、 实验分析

- 1. 简单的 ABR 算法实现非常简单,代码结构清晰易懂。但是存在不能适应 网络环境的动态变化的缺点,在网络波动较大的时候尤其明显。实验结果 表明简单的 ABR 算法并不能很好地适应网络波动。
- 2. 改进的 ABR 算法 I 只在简单的 ABR 算法之上增加了一个 720P 的选择分支, 让客户端在 480P 和 1080P 之间增加了一个选项。该选项使得客户端在网络波动时可以选择更加合适的码率播放视频, 提高了视频播放质量和用户体验。实验结果表明增加的 720P 分支在一定程度上改善了简单 ABR 算法存在的缺点。
- 3. 改进的 ABR 算法 II 是基于缓存补偿的视频码率自适应算法,能预测下一时刻的网络带宽,同时也能根据视频缓存时长动态调整请求的码率等级。实验结果表明该算法对网络带宽的预测比较准确,能适应网络环境的动态变化,即使是网络波动较大的时候也能有良好的表现。QoE 评分也比上述两者高,但是并不是很明显。原因可能是因为算法复杂度较前两者更高,而视频总时长较短,无法体现绝对优势。
- 4. 通过对比三种不同的 ABR 算法,可以发现改进的 ABR 算法 II 在提高用户 观看体验方面具有明显优势。然而,实际应用中需要考虑算法的复杂度、 实现成本以及调整参数的难度等因素,选择最适合具体场景和需求的算法。

五、 实验总结

- 1. 本次实验我实现了<u>基于缓存补偿的视频码率自适应算法</u>这篇论文里面的 ABR 算法。
- 2. 我查阅了 fluctuation.h 中的源码, 了解了如何通过基础延迟和随机延迟组合模拟网络波动。
- 3. 通过本次实验,我了解了经典 ABR 算法的分类:基于带宽预测、基于缓冲区容量和基于神经网络学习等。
- 4. 通过本次实验, 我了解了 ABR 算法的基本原理和优化方法。

六、 思考题

1 如果有某一用户群体,相较于其他人,对视频质量的波动没有那么在意,而较低的视频质量回事他们更加恼火。针对该用户群体,当前QoE评分方式是否合适?如果不合适,如何调整?

当前 QoE 评分方式可能不太合适。因为它对视频卡顿的惩罚程度较高,而对高视频质量的奖励程度较低,同时对视频质量波动的惩罚也较高。这会导致客户端更加倾向播放低质量视频以维持视频不卡顿。

调整方法: 1) 增加对较低视频质量的惩罚; 2) 适当降低对卡顿时间的惩罚; 3) 适当提高对高质量视频的奖励; 4) 适当降低视频质量波动的惩罚程度。

例如设置参数 (参照实验平台): alpha=0.005, beta=0.001,gamma=0.005,新增参数 c=0.001。

- 2 用户在观看常规视频时可以完整观看到视频的全部内容,而在沉浸式视频 (360°视频、点云视频等)中,用户会选择性的观看部分视频内容。对于沉浸式视频,如何评估用户的视频观看体验?
- 1) 考虑视频的加载速度、流畅度、缓冲时间等技术性能指标,这些指标直接影响到用户的观看体验。
- 2) 监测用户在沉浸式视频中的交互行为,包括观看方向、转头频率、放大缩小等操作。这些交互行为可以提供关于用户兴趣和注意力焦点的重要信息。
- 3) 考虑用户在不同部分视频内容上的观看时长和观看频率。通过分析用户对不同 内容区域的观看持续时间和频率,可以了解用户对视频中各个部分的关注程度。
- 4) 收集用户的反馈和评价,了解他们对沉浸式视频观看体验的主观感受。这可以通过用户调查、问卷调查、用户评论等方式进行收集。

七、附页

```
改进 ABR 算法 II 头文件全部代码如下所示
#include <math.h>
#define MAX LEN 4
#define N 100000
#define W 0.4
#define Qmin 2000
#define Bmax 10000
//const int MAX LEN = 4, N = 1e5 + 10;
//const double W = 0.6;
const int R[] = \{ 1000, 2500, 5000, 8000 \};
double download rates[N];
double w[MAX LEN];
int font = 0, rear = 0;
int cur grade = 3;
int st = 1;
double predict rate[N];
int p idx;
void init(int len)
    double beta = 1, alpha = W;
    for (int i = 0; i < len; i++) {
         beta *= (1 - W);
    beta = 1 - beta;
    for (int i = 0; i < len; i++) {
         alpha *= (1 - W);
         w[i] = alpha / beta;
}
```

```
void updateDownQueue(int file size, long download time)
     {
         double alpha = 1;
         if (download time \leq 0) {
              download time = 1;
              alpha = 0.85;
         double download_rate = (double)file_size / download_time / alpha;
         download rates[++rear] = download rate;
         if (rear - font > 4) font++;
     }
     double getNextDownloadRate()
         double next download rate hat = 0;
         long double sum = 0;
         int cnt = 0;
         for (int i = rear - 1; i > font && i > rear - 5; i--) {
              sum += download rates[i];
              cnt += 1;
         double mean download rate = 0;
         if (cnt == 0) {
              mean download rate = 8000;
         }
         else {
              mean download rate = sum / cnt;
         double down download rate = mean download rate * 0.4;
         if (download rates[rear] >= mean download rate) {
              next download rate hat = (mean download rate * cnt + download rates[rear]) /
(cnt + 1) + 0.5;
         }
         else if (download_rates[rear] >= down_download_rate){
              double tmp download rate = 0;
              if (st && rear - font <= MAX LEN) {
                   init(rear - font);
                   if (rear - font == MAX LEN)
                        st = 0;
              for (int i = 0, j = rear; i < MAX\_LEN \&\& j > font; i++, j--) {
                   tmp download rate += w[i] * download rates[j];
```

```
}
              next download rate hat = tmp download rate + 0.5;
         }
         else {
              next download rate hat = down download rate;
         return next_download_rate_hat;
    }
    int getSwitchGrade()
         double max_rate = getNextDownloadRate();
         int tar grade = 0;
         for (int i = 0; i < 4; i++) {
              if (max_rate > R[i]) {
                   tar grade = i - 1;
              }
         }
         return tar grade;
    }
    int getNowGrade()
         return cur_grade;
    double getSwitchingTime(double now_bit_rate_idx)
         double t = 1 / getNextDownloadRate();
         int tmp = 0;
         for (int i = max(getSwitchGrade(), now_bit_rate_idx); i >= min(getSwitchGrade(),
now_bit_rate_idx); i--) {
              tmp += R[i];
         return t * tmp;
    }
    double getQMin()
         return Qmin;
    }
    double getQDown()
     {
         double alpha = R[getNowGrade()] / R[0];
```

```
return Qmin + (1 + alpha) * getSwitchingTime(getNowGrade());
}
double getQUp(int now_bit_rate_idx)
{
    return Bmax * 0.85 * (now bit rate idx + 1) / 4;
}
int getNextBitRate(double buffer_time)
{
    int now grade = getNowGrade();
    int result grade = now grade + 1 > 3 ? 3 : now grade + 1;
    double next download rate = getNextDownloadRate();
    if (buffer_time < getQMin()) {</pre>
         cur_grade = 0;
         return R[0];
    else if (buffer time > getQUp(now grade)) {
         cur grade += 1;
         if (cur_grade > 3) cur_grade = 3;
         return R[cur grade];
    }
    else if (next download rate > R[result grade]) {
         cur grade += 1;
         if (cur grade > 3) cur grade = 3;
         return R[cur grade];
    }
    else if (next download rate < R[now grade]) {
         if (buffer_time > getQDown()) {
              return R[now grade];
         }
         else {
              cur grade -= 1;
              if (cur_grade < 0) cur_grade = 0;
              return R[cur grade];
         }
    }
    else {
         return R[now grade];
}
```

指导教师批阅意见

成绩评定

指导教师签字:

年 月 日

注: 1、报告内的项目或内容设置,可根据实际情况加以调整和补充。

2、教师批改学生实验报告时间应在学生提交实验报告时间后10日内。