

# 天津大学

## 《计算机网络》课程设计 周进度报告

题目：第三周 实现 HTTP 的并发请求

学 号：	3020202184 3020244344
姓 名：	刘锦帆 李镇州
学 院：	智能与计算学部
专 业：	计算机科学与技术
年 级：	2020 级
任课教师：	石高涛

2022 年 3 月 31 日

# 目 录

第一章	协议设计	1
1.1	协议设计	1
1.2	HTTP pipeline 设计	1
第二章	协议实现	3
2.1	Pipelining 实现	3
第三章	实验结果及分析	4
3.1	测试样例	4
第四章	进度总结及项目分工	5
4.1	本周进度情况	5
4.2	人员分工	5

## 一 协议设计

### 1.1 协议设计

- 首先要求服务器能连续响应客户端使用同一个 TCP 连接同时发送多个请求，即可以实现 http 管线化。
- HTTP/1.1 中单个 TCP 连接在同一时刻只能处理一个请求，为了解决这个问题，HTTP/1.1 在 RFC 2616 中规定了 Pipelining。因此 RFC 2616 中规定：一个支持持久连接的客户端可以在一个连接中发送多个请求（不需要等待任意请求的响应）。收到请求的服务器必须按照请求收到的顺序发送响应。
- 对于 HTTP 的并发请求，如果服务器认为其中一个请求是错误的，为了避免服务器把它和其他并发的请求全部拒绝了，服务器需要设计可以识别这条错误的请求的下一条请求，使得尽量多的正确并发请求得到满足。

### 1.2 HTTP pipeline 设计

1. 什么是 HTTP pipelining: http 管线化是一项实现了多个 http 请求但不需要等待响应就能够写进同一个 socket 的技术，采用管线化的请求会对页面载入时间产生动态的提高，尤其是当通过高延迟的网络，例如通过卫星网络连接；普通情况下通过同一个 tcp 数据包发送多个 http 请求，而 http 管线化向网络上发送更少的 tcp 数据包，大幅减轻网络负载；只有幂等的请求能够被管线化，例如 get 和 head 请求；post 请求不应该被管线化；新建立连接的请求因为无法判断源服务器(代理服务器)是否支持 http1.1 协议，也不应该被管线化处理。所以，仅在重用已经成功建立的持久化连接的情况下，才可以使用管线化。http 管线化需要客户端和服务端双方都能够支持，http1.1 规定服务器必须支持管线化，但并未提及服务器必须管线化响应信息，但如果客户端选择管线化的通信方式，服务器必须能够支持和受理。
2. HTTP pipelining 的优势：减少 cpu 和内存占用，减轻网络堵塞，减轻后续请求的延迟。不采用管道化意味着每次请求必须被应答之后，它的连接才能空闲以便发送下一次请求；不采用管道化会导致平均每个连接带来额外的延迟，或者如果服务器不支持 http 长连接，进行其他的 tcp 三次握手增加了额外的请求往返，双倍延迟；不需要牺牲当前的 tcp 连接，就能够报告错误。一个单用户客户端对于任何一台服务器或者代理服务器都可以

维护不多于两个的连接数. 在当前由  $n$  台服务器组成的网络中, 任意一台代理服务器对另外的服务器或者代理服务器应该维护  $2*n$  个连接. 这些指南目的在于提升 http 响应性能, 避免网络堵塞。

## 二 协议实现

### 2.1 Pipelining 实现

本周的实现较为简单。由于在 Lab2 中我们实现了缓冲区的自动化更新、防溢出。可以支持任意大小的文件传入。所以本周的任务中，我们只需要关注如何将收到的信息拆分为单个单个的报文。然后处理之即可。

我们的实现方式，是通过函数 `strstr()` 获得拆分每个报文。`strstr(str,dest)` 函数得到 `dest` 在 `str` 第一次出现的地址。于是就能通过 `strstr()` 得到报文结束前 `"\r\n\r\n"` 的位置。然后通过简单的字符串处理，就能开始处理单个报文了。部分代码如图 2-1。

```
147 // s/liso_server.c
148
149 /**
150  * @brief Deal with buffer --> which contains (multiple) requests
151  * if it's a pipeline request
152  * @param dbuf --> dynamic_buffer, requests
153  * @param readret --> size of buffer
154  * @param client_sock --> client's sock
155  * @param sock --> server's sock
156  * @param cli_addr --> client's address
157  * @return
158  * --> PERSISTENT : Go on waiting for msg
159  * --> EXIT_FAILURE : End this connection
160  * --> CLOSE : Close current connection ( Which can be ignored in our lab )
161  */
162
163 int deal_buf(dynamic_buffer *dbuf, size_t readret, int client_sock, int sock, struct sockaddr_in cli_addr)
164 {
165     print_dynamic_buffer(dbuf);
166     char *t = temp=dbuf->buf;
167     int cnt = 0;
168     /* deal pipeline */
169     dynamic_buffer *return_buffer = (dynamic_buffer*) malloc(sizeof(dynamic_buffer));
170     while(1)
171     {
172         while((t=strstr(temp, dest))!=NULL)
173         {
174             int len = t - temp;
175             PRINTF("=====CURRENT CNT: %d=====\\n", cnt++);
176             dynamic_buffer *each = (dynamic_buffer *) malloc(sizeof(dynamic_buffer));
177             init_dynamic_buffer(each);
178             append_dynamic_buffer(each, temp, len);
179             append_dynamic_buffer(each, dest, strlen(dest));
180             temp = t + strlen(dest);
181             LOG("Starting dealing with msg\\n-----\\n\\n\\n-----\\n", each->current);
182             Return_value result = handle_request(client_sock, sock, each, cli_addr);
183             append_dynamic_buffer(return_buffer, each->buf, each->current);
184             LOG("MSG Appended\\n");
185             if(result==CLOSE)
186             {
187                 return CLOSE;
188                 free_dynamic_buffer(each);
189             }
190             update_dynamic_buffer(dbuf, temp);
191             LOG("msg to be sent\\n===== Sending =====\\n\\n\\n" return_buffer->buf);
192             if(!return_buffer->current)
193             {
194                 LOG("Not complete\\n");
195                 free_dynamic_buffer(return_buffer);
196                 return PERSISTENT;
197             }
198             if (send(client_sock, return_buffer->buf, return_buffer->current, 0) != return_buffer->current)
199             {
200                 close_socket(client_sock);
201                 close_socket(sock);
202                 free_dynamic_buffer(return_buffer);
203             }
204         }
205     }
206 }
```

图 2-1 Liso Server Pipelining

### 三 实验结果及分析

### 3.1 测试样例

我们通过 client 端发送测试样例 `sample/request_pipelining`，得到如图3-1 的结果，表明我们的实现是正确的。

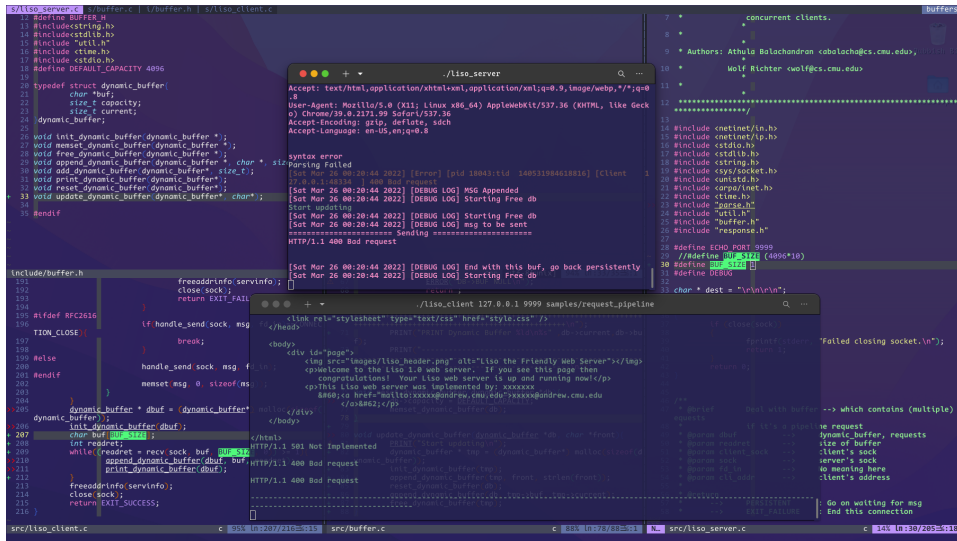


图 3-1      pipelining test

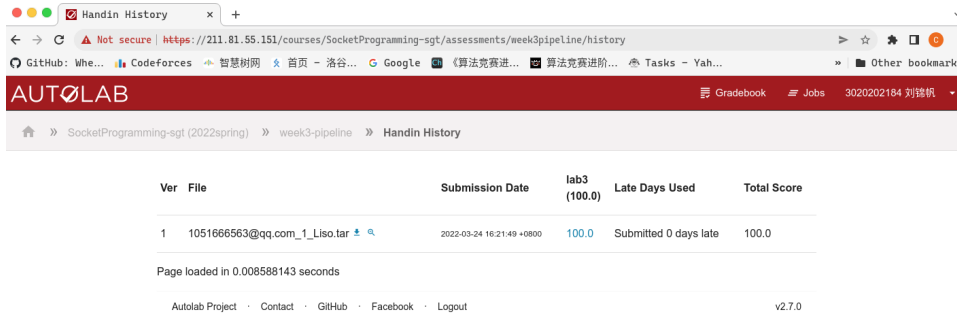


图 3-2 Autolab Test Result

## 四 进度总结及项目分工

### 4.1 本周进度情况

本周对 pipelining 进行了实现。

	本周任务要求	完成	备注
1	实现 pipelining	✓	无

表 4-1 本周进度完成表

### 4.2 人员分工

人员分工如表4-2所示。

人员	项目分工
刘锦帆	完成大部分代码工作，以及协议实现部分
李镇州	完成 client 端的处理以及协议设计部分的写作

表 4-2 人员分工表