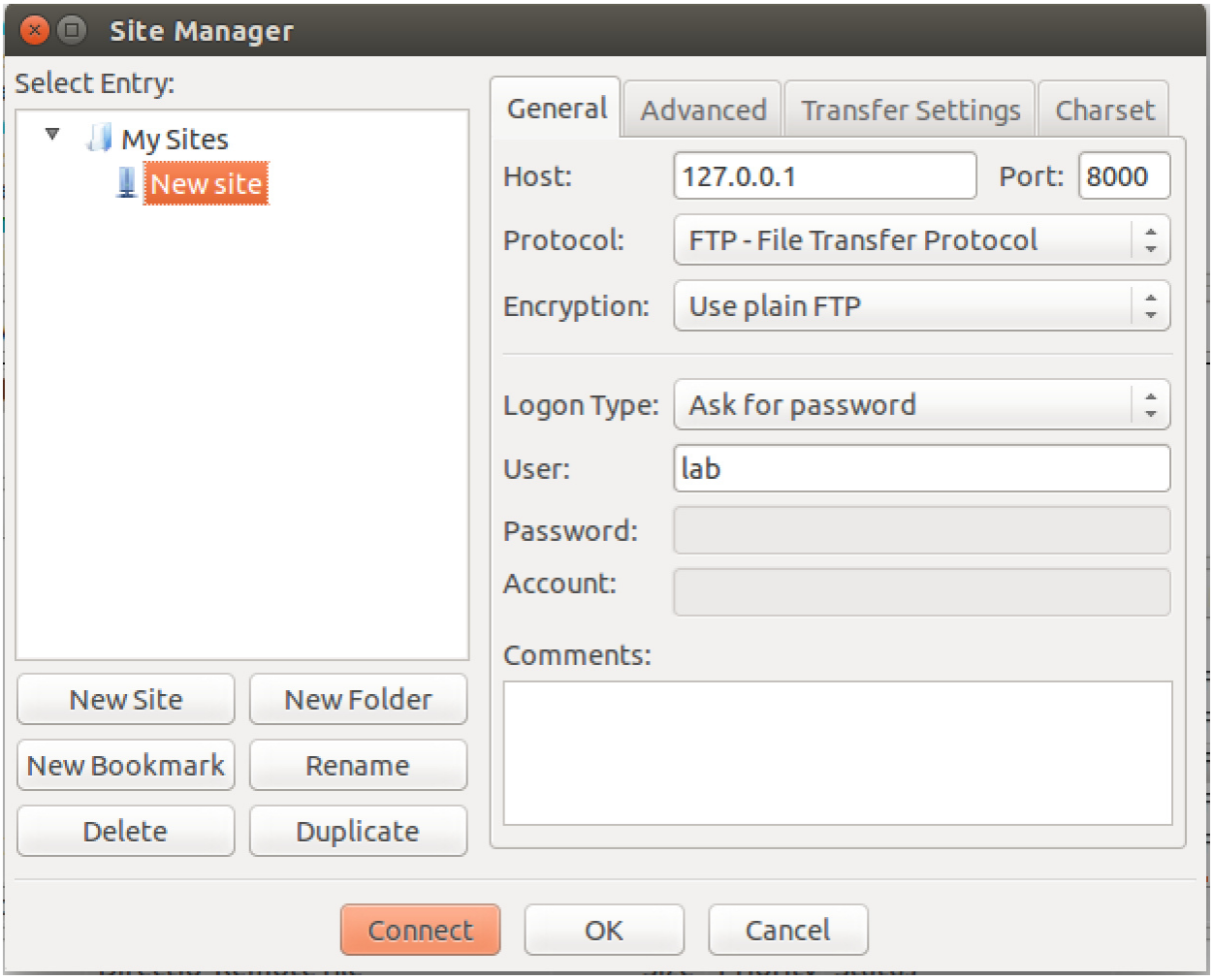
Team 14 FTP Proxy Report

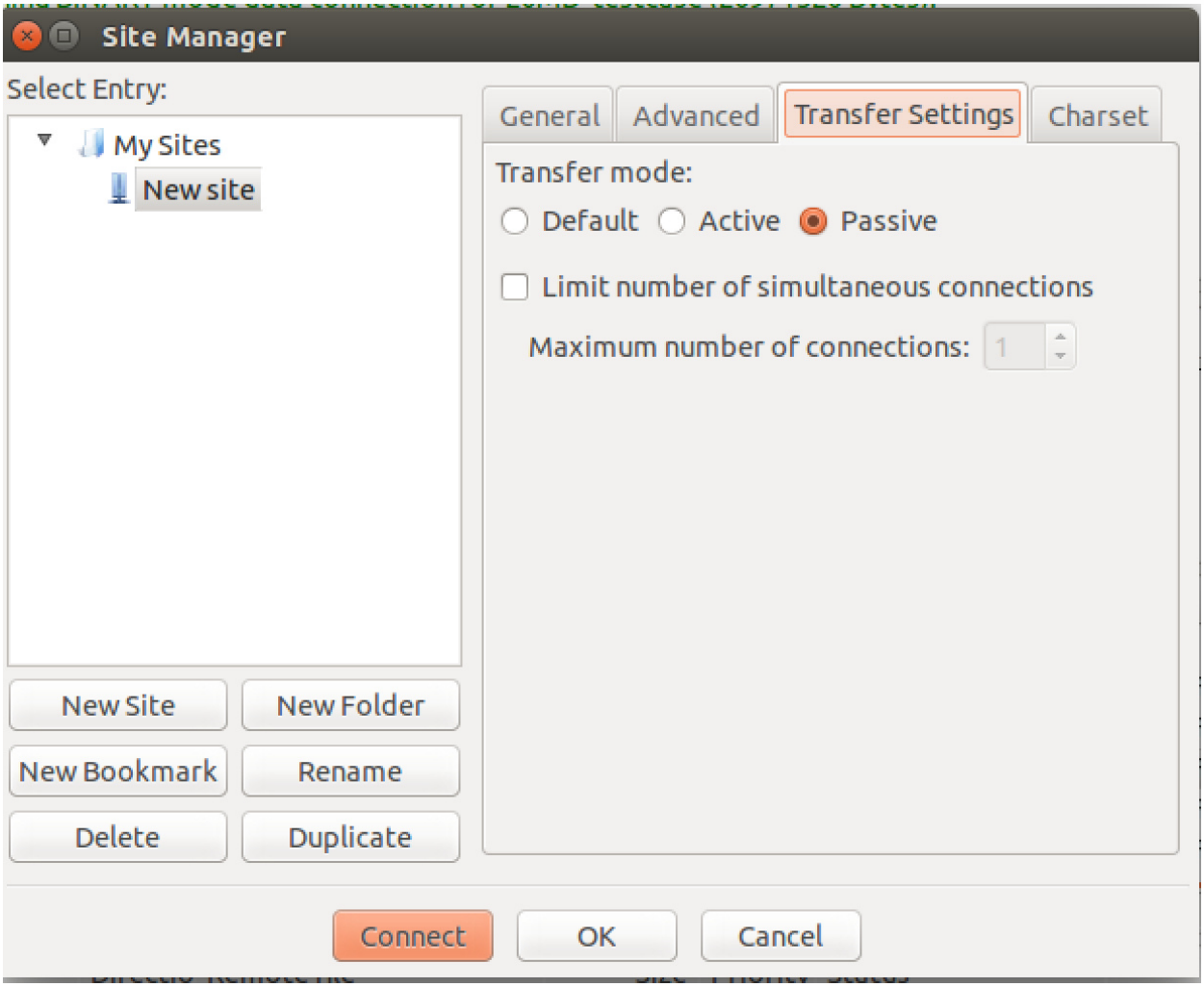
運行方法

```
$ make clean
$ make
$ ./proxy <ProxyIP> <ProxyPort> <Rate>
```

例如: `./proxy 127.0.0.1 8000 100` 可以將下載和上傳的速度限制為100KB/sec. 若需要更換限制的速度，則`control+c`終止程序之後，使用新的`Rate`重新運行proxy。

Filezilla 通過被動模式連接上`<ProxyIP>`和`<ProxyPort>`即可，連接所需要的賬號密碼是原FTP server所需要的賬號密碼。註意需要選擇不加密的plain FTP，以及被動模式。通過在filezilla客戶端點擊`control+s`可以進入站點管理，在`Encryption`選擇`Use plain FTP`在`Transfer Settings`設置為`Passive`





運行環境

適用於linux系統，本次測試環境使用ubuntu 14.04

所依賴的庫有linux下的`<sys/socket.h>`和`<pthread.h>`等。

註意：使用macOS進行測試可能會出現問題，因為socket的window size在macOS下使用setsockopt無法限制住，詳細可以參考我所記錄的issue#2

Proxy 理解

Proxy 作用

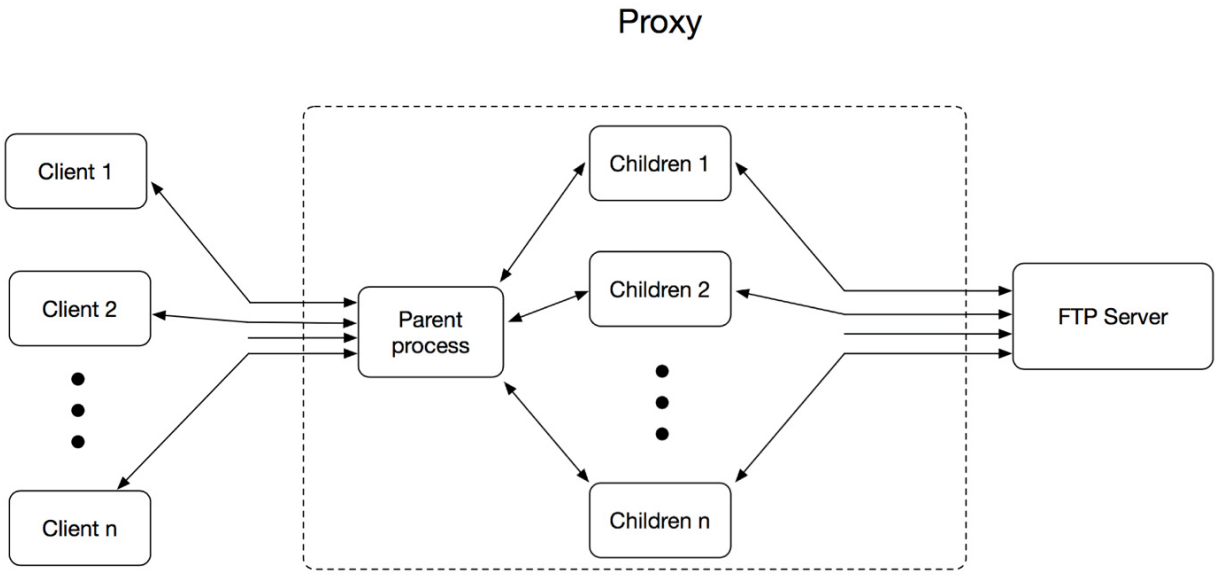
FTP proxy 所起的作用在FTP client與FTP server之間進行數據的轉送。本次的Proxy所起的作用就是從FTP server下載數據，以及從FTP client上傳數據到FTP server之間進行速度的限制。其實，proxy的作用還有很多，比如可以制作成應用層的防火墻，在封包傳到server/client之前，對封包的每壹層進行檢查，是壹種安全級別非常高的防火墻，但是速度相對比較慢。也可以通過proxy對封包數據進行過濾，速度進行限制，放置server接受太多的數據無法處理過來或者造成網絡擁塞。也可以當做壹種壹種代理上網的方式。但是FTP proxy的缺點就是，只能識別FTP協議，所能識別的協議比較有限。此處的FTP proxy只是針對未加密的FTP 被動模式所涉及。

程序基礎架構

整體的FTP proxy內部架構如下。可以允許同時有多個client與proxy進行連接。proxy的主程序會開啟壹個socket不斷進行listen，每次有client與proxy進行連接的時候，都會創建壹個子進程來處理。每壹個子進程都會與client建立壹條socket連線，再與server建立壹條socket連線。

此處的client是廣義而言的，並不是單指壹個主機，壹個進程都可以作為壹個client。在Filezilla中，每新建壹個下載或上傳進程，都會向proxy主程序發起壹個新的連接請求。所以每壹個任務都可以被視作是壹個新的client。

Proxy面向client就相當於server壹樣，面向server就相當於client壹樣。所以Proxy其實既要實現client的功能也要實現server的功能。



FTP 被動模式

在FTP 被動模式下client與server上傳數據的過程中使用的是建立連線所使用的`Port`，而需要從FTP下載數據時，client會開放壹個特定的`Port`然後將端口的信息以及被動模式信息發送給Server，server接受到消息會開放壹個端口來與client進行連接，並且回應client並且與client的這個端口建立連線。被動模式主要是為了避免因為cilent的端口沒有開放，而server發送了該端口的連接請求而被client的防火墻所阻擋而設計。

所以，在FTP的被動模式下每個client會有兩個連線，數據通路和命令通路。

- 數據通路，負責從server下載信息，如文件、目錄信息等。
- 命令通路，該通路負責從client上傳控制信息或者數據到server，以及server返回壹些相應的控制信息。

也正因為這兩個通路是分離開來的，以至於在過程中會出現兩個通路的信息不同步等壹些非常tricky的問題。在後面的問題解決，我有通過github的issue比較詳細地記錄下來。例如，從server下載完數據到prxoy，server會通過命令通路發送信息給proxy 226傳送完畢的信息，proxy直接將信息轉傳給client，client便認為數據已經傳送完畢，但是實際因為限速原因，數據還留在proxy沒有傳送完畢。亦或者是從client上傳數據到proxy，client上傳完畢，但是proxy數據還沒有轉送完畢，此時client過長時間沒有收到proxy返回的226信息，自動斷開連接等問題。

FTP 常見命令 & server 響應碼

在進行調試程序的時候，經常要通過判斷server的response code和client的命令來進行判斷。而且proxy也要有識別server response code的能力，比如在被動模式下建立數據通路就要識別server 返回的227信息。所以有壹些常見的命令和響應嗎需要了解。

Server Code	**Explanation**
`226`	Closing data connection. Requested file action successful (for example, file transfer or file abort).
`227`	Entering Passive Mode (h1,h2,h3,h4,p1,p2).
`150`	File status okay; about to open data connection.

命令	**解釋**
RETR	傳輸文件副本
STOR	接收數據並且在服務器站點保存為文件
LIST	如果指定了文件或目錄，返回其信息；否則返回當前工作目錄的信息

詳細地可以參考wikipedia:

- FTP命令列表,
- List of FTP server return codes

Linux Socket

此次proxy所使用的基礎就是linux下的socket，使用它，我們不必去實現壹些復雜的TCP協議的功能。只要通過其提供的API就能實現向讀寫文件壹樣控制TCP流了。

Linux Socket為網絡的低層協議提供了API(接口)。Socket 實現了IP的接口，在socket的基礎之上，我們可以建立網絡層的協議。比如建立TCP,建立UDP。通過Socket可以很容易地幫助我們實現這壹點。而且它提供了read/write這種功能的函數，讓我們可以像是處理文件讀寫壹樣來處理數據的傳輸。

通過Socket實現TCP

這裏簡單地介紹壹下，如果通過Socket實現TCP，由於Proxy同時面向Client和Server所以，在其內部會有client的功能也會有server的功能。

Server部分

在proxy內部，通過要與client進行連接，就要模擬server的功能。

1. 通過socket()函數創建壹個socket
2. 將這個server地址端口進行初始化，使用結構體sockaddr_in
3. 將socket於server的地址端口進行綁定，使用bind()
4. 對這個socket進行listen
5. 通過accpet來接受client的connect請求(此處在proxy的主程序中循環等待accept)
6. 連接完畢通過close關閉socket

如下方所示，這就是在proxy中創建壹個TCP連接server的server部分。

```
int create_server(int port) {
    int listenfd;
    struct sockaddr_in servaddr;

    listenfd = socket(AF_INET, SOCK_STREAM, 0);
    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port = htons(port);
    if (bind(listenfd, (struct sockaddr *)&servaddr ,
sizeof(servaddr)) < 0) {
        //print the error message
        perror("bind failed. Error");
        return -1;
    }

    listen(listenfd, 3);
    return listenfd;
}
```

client部分

同樣地，在proxy部分，要與server進行連接，就要模擬client的功能。

1. 創建壹個新的socket
2. 初始化server的地址信息
3. 通過connect連接到服務器
4. 連接完畢通過close關閉socket

```c
int connect_FTP(int ser_port, int clifd) {
    int sockfd;
    char addr[] = FTP_ADDR;
    int byte_num;
    char buffer[MAXSIZE];
    struct sockaddr_in servaddr;

    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        printf("[x] Create socket error");
        return -1;
    }

    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_port = htons(ser_port);

    if (inet_pton(AF_INET, addr, &servaddr.sin_addr) <= 0) {
        printf("[v] Inet_pton error for %s", addr);
        return -1;
    }

    if (connect(sockfd, (struct sockaddr *)&servaddr,
sizeof(servaddr)) < 0) {
        printf("[x] Connect error");
        return -1;
    }

    printf("[v] Connect to FTP server\n");
    if (ser_port == FTP_PORT) {
        if ((byte_num = read(sockfd, buffer, MAXSIZE)) <= 0) {
            printf("[x] Connection establish failed.\n");
        }

        if (write(clifd, buffer, byte_num) < 0) {
            printf("[x] Write to client failed.\n");
            return -1;
        }
    }

    return sockfd;
}
```

在連線建立完成之後就可以通過，server部分通過操作client的socket，client部分通過操作server的socket部分，就可以用read/write函數像文件讀寫壹樣來進行操作了。

速度限制功能

此處任務的核心目的，就是在proxy對上傳和下載速度進行控制。下面將介紹，用來進行限速的算法token bucket 令牌桶算法的基本原理，以及如何在proxy中實現這壹算法。

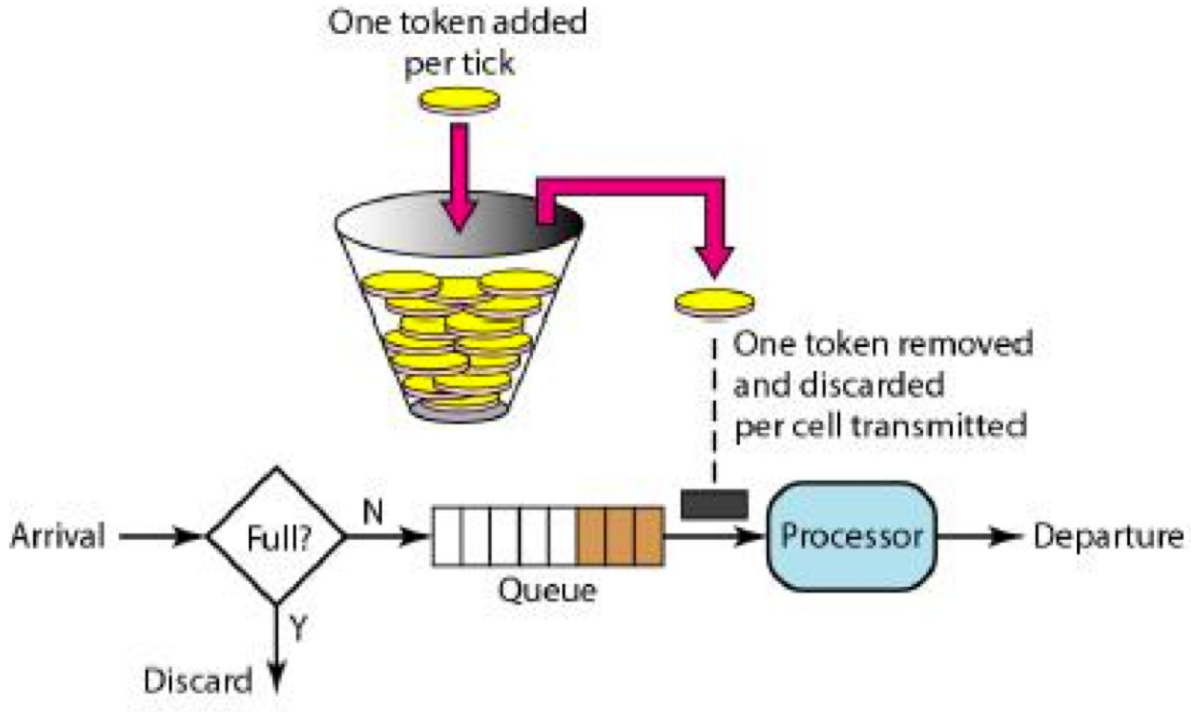
Token bucket(令牌桶算法)

介紹

令牌桶算法是網絡流量整形(Traffic Shaping)和速度限制(Rate Limiting)中所常用的壹種算法。它有兩個主要的特點，壹個就是能夠限制速度，控制封包發送的數量，另壹個就是允許妳突發的數據傳輸。相比於leaky bucket算法，它能更好地滿足QoS服務品質標準，允許壹段時間發送較多的封包，而之後不發送封包，但是保證封包發送的數目是在限制範圍之內的，既能夠保證限制這段時間內發送封包的平均速度，又能允許比較大的封包到來。而leaky bucket就是不能允許突發的流量的。

原理

下面用壹張圖來解釋壹下算法的具體原理:



令牌桶算法主要有三個主要部分:

1. token generation(令牌產生)：每個`1/Rate`時間往令牌桶中放入壹個令牌（相當於速度單位的最小單位）

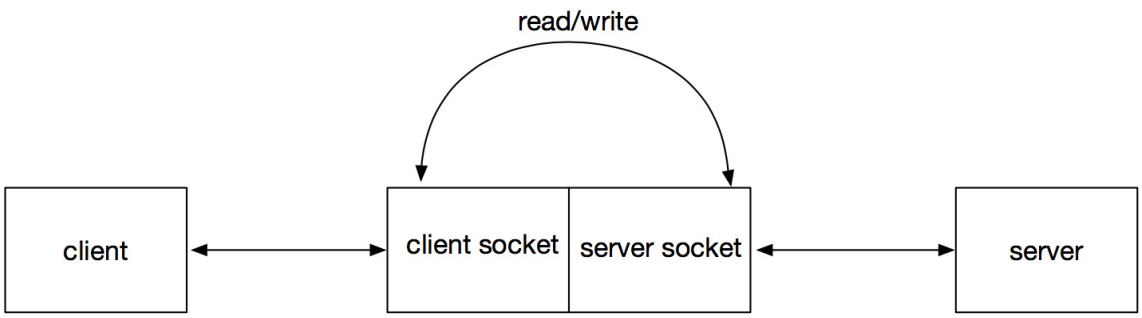
2. 封包隊列:建立壹個隊列來暫時存儲封包，當封包進入proxy之後，隊列未滿就暫存下來，若隊列已經滿了則不進行接收。
3. 消耗令牌，發送封包：每次從封包隊列的front取出封包，如果令牌桶中令牌的數量大於等於封包大小（取決於速度單位）則消耗相應的令牌，發送該封包，若沒有則繼續等待。

Linux下令牌桶具體實現方法

使用socket緩存區

核心

在`ftp_proxy.c`的`int proxy_func(int ser_port, int clifd, int rate)`就是在建立好兩邊的連線之後進行內部讀寫和限速功能的核心函數了，也相當於刪除結構圖中的每壹個單獨處理的子進程。



緩存區（充當封包隊列）

如圖所示中間部分，就是proxy的每壹個子進程所起的作用。每個socket部分，在建立完每TCP連線之後都會有兩個緩存區，壹個是發送緩存區，壹個是接受緩存區。也就是說proxy子進程上會有4個緩存區

- client socket的緩存區
  - 接受緩存區，read client socket 從client讀的數據，就是先緩存到這裏再從這裏面讀取數據的
  - 發送緩存區，write client socket 發送到client的數據，就是先緩存到這裏再進行發送的
- server socket的緩存區
  - 接受緩存區，read server socket 從server讀的數據，就是先緩存到這裏再從這裏面讀取數據的
  - 發送緩存區，write server socket 發送到server的數據，就是先緩存到這裏再

進行發送的

這裏緩存區的大小就是所建立的TCP連線的window size的大小。TCP的擁塞協議根據，網絡狀況丟包情況，會進行動態地調整。具體TCP的協議，在socket內部已經實現好了，了解了緩存區基礎已經足夠我們在這裏進行編程了，在這裏不需要自己去寫raw socket，來進行ACK等回復。只要把TCP連線當成stream(流)來處理，類似如文件流。

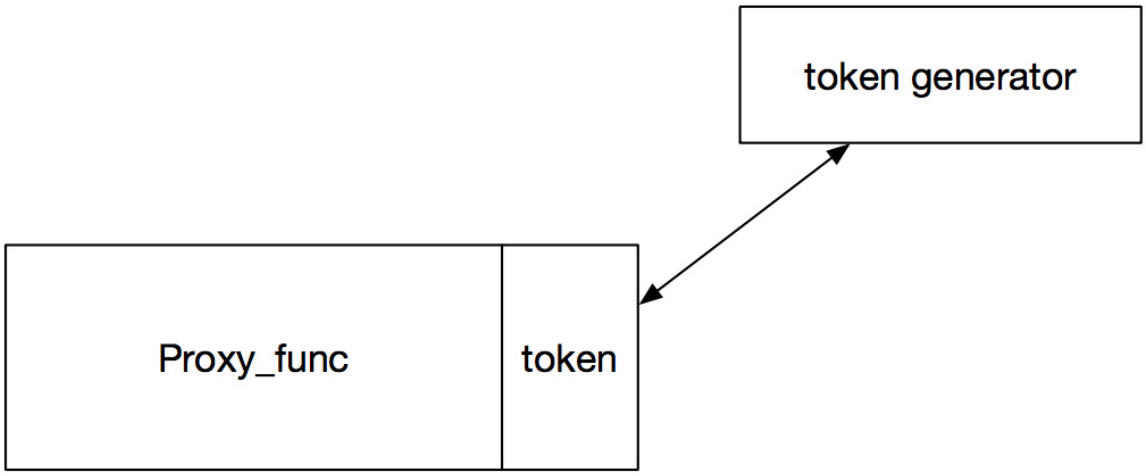
此處的緩存區，就可以充當我們令牌桶算法中**封包隊列**的職責了。所以之後，我們在實現令牌桶算法的時候，就只要從壹個緩存區的接受緩存區讀取數據->消耗令牌->發送到另壹個緩存區的發送緩存區即可。

令牌結構體

由於令牌的產生需要與其他任務運行是同步運行的，所以這裏就考慮使用Linux下的pthread多線程來進行設計。壹個proxy的子進程創建了壹個線程，負責產生令牌，且創建多線程與fork壹個子進程相比不會再次復制父進程的空間，因為那些數據大多數在這裏是多余的。進程與線程共享壹個共同的數據token。此處的token，因為在多線程裏共享多個變量，需要使用結構體。所以我就定義了壹個結構體token。在`token.h`可以詳細看到。

```
struct token {
    pthread_mutex_t mutex;
    int count;
    int rate;
    int token_per_time;
    double t; // delay time of generating and consuming token
};
```

這個結構體裏，包含了令牌的數目，當前所限制的速率。以及壹個互斥鎖，因為在訪問多線程臨界區資源的時候，需要通過呼互斥鎖來保證數據的正確性和完整性。



創建線程方法如下，proxy_func中創建

```
    // tokens not to tell upload and download
    struct token *proxy_token = (struct token *)malloc(sizeof(struct
token));
    proxy_token->rate = rate*1024;//Kbytes to bytes
    proxy_token->count = 0;
    // to balance consume and generate token frequency
    proxy_token->token_per_time = proxy_token->rate >= MAXSIZE*100 ?
MAXSIZE : proxy_token->rate / 100;
    proxy_token->t = ((double)(proxy_token-
>token_per_time)/proxy_token->rate);

    // mutex closer to critical source
    pthread_mutex_init(&proxy_token->mutex,NULL);
    // create a pthread to generate token
    pthread_t ptid;
    pthread_attr_t attr;
    pthread_attr_init(&attr);
    // thread token generator
    pthread_create(&ptid, &attr, token_generator, (void*)proxy_token);
    // char is 1 byte
```

產生令牌

這裏所用來產生令牌的函數，與token bucket算法描述上做了壹些修改。主要的修改就是不再是`1/rate`就產生壹個token，而是在壹定時間內產生壹些token，通過`token_per_time`我每次最少增加的token，以及產生這麼多token需要的時間`t`，此處我所采用每次產生token的數目為

```
proxy_token->token_per_time = proxy_token->rate >= MAXSIZE*100 ?
MAXSIZE : proxy_token->rate / 100;
```

大約是每秒所能傳送字節數的1/100。之所以這樣子做，是為了避免反復讀取臨界區變量，獲取互斥鎖所需過於頻繁，導致系統開銷過大，以及多余時間的花費。同時，在這裏我定義了計時函數，會把獲取互斥鎖所花費的時間，通過相應token的量補回去。這樣子修改之後，就可以降低訪問臨界區資源的次數，而且根據速度來設置每次放token的量，也會使傳輸更均勻，封包不用等很長時間才可以送壹次。

需要主要的壹點，在使用多線程的時候，需要將線程與線程進行手動的分離使用`pthread_detach`函數，因為默認進程產生了壹個線程與原始線程是阻塞關系的。

```
void *token_generator(void *data) {
    // detach thread from process
    pthread_detach(pthread_self());
    struct token* proxy_token;
    proxy_token = (struct token*)data;
    // two cancle point to avoid block by thread lock
    clock_t start, end;
    double dur = 0;
    int token_per_time = proxy_token->token_per_time;
    int usleep_time = (int)1000000*(proxy_token->t);

    while (1) {
        // delay
        usleep(usleep_time);
        // dur can not count sleep time into it
        start = clock();
        pthread_testcancel(); // cancle point
        pthread_mutex_lock(&proxy_token->mutex);
        if (proxy_token->count < 2*token_per_time) {
            proxy_token->count += token_per_time + (int)(dur *
proxy_token->rate);
            //printf("generate %d tokens\n", proxy_token->count);
        }
        // unlock
        pthread_mutex_unlock(&proxy_token->mutex);
        pthread_testcancel();
        end = clock();
        dur = (double)(end-start)/CLOCKS_PER_SEC;
    }
    return ((void*)0);
}
```

消耗令牌

在消耗令牌的過程中，我把它包裝成了壹個阻塞式的write function，當能夠拿到足夠的token的時候就進行write，拿不到的時候就進行等待。等待的時間，大致與產生足量的token的時間差不多。這樣子也避免了長時間占有互斥鎖的問題。原先，在設計的過程中，我沒有加入等待的時間，所以容易出現壹個問題，如果有壹方訪問互斥鎖的速度過快，另壹方就很難拿到互斥鎖，會導致阻塞而影響了準確的限速。

```
int rate_control_write(struct token* proxy_token, int fd, char*
buffer, int byte_num) {
    // seperate packet to get more fluent data flow
    int usleep_time = (int)1000000*(proxy_token->t);
    pthread_mutex_lock(&proxy_token->mutex);
    while (proxy_token->count < byte_num) {
        pthread_mutex_unlock(&proxy_token->mutex);
        usleep(usleep_time);
        pthread_mutex_lock(&proxy_token->mutex);
    }
    //printf("consume %d tokens %d\n", proxy_token->count,byte_num);
    proxy_token->count -= byte_num;
    pthread_mutex_unlock(&proxy_token->mutex);

    // write
    if (write(fd, buffer, byte_num) < 0) {
        printf("[x] Write to server failed.\n");
        return -1;
    }
    return 0;
}
```

應用令牌桶

在設計完成令牌桶之後，就可以在上傳和下載的過程中應用上這個算法了。

這裏所采用的方法是`selcect`函數，使用這個函數來復用I/O接口。當哪個client socket活躍就從那讀取數據，當server socket活躍就從那裏讀取數據。

初始化

```
// initialize select vars
FD_ZERO(&allset);
FD_SET(clifd, &allset);
FD_SET(serfd, &allset);
rset = allset;
maxfdp1 = max(clifd, serfd) + 1;

// select descriptor,I/O multiplexer
nready = select(maxfdp1, &rset, NULL, NULL, NULL);
```

下面以從client讀取數據到server為例：

```
if (FD_ISSET(clifd, &rset)) {
    memset(buffer, '\0', MAXSIZE);
    // read from TCP recv buffer
    if ((byte_num = read(clifd, buffer, proxy_token->token_per_time))
<= 0) {
        printf("[!] Client terminated the connection.\n");
        break;
    }

    // blocking write to TCP send buffer
    if (rate_control_write(proxy_token, serfd, buffer, byte_num) != 0)
{
        printf("[x] Write fail server in rate controling write\n");
        break;
    }
}
```

只要調用所設計的token.h中，消耗token的write funciton既可以像讀寫文件壹樣來進行處理了。使得框架的設計變得更加簡單。

框架修改

- 在原先基礎的框架之上，我修復了壹個原有的bug。詳細信息在github的issure#1可以看到分析過程和解決過程。主要解決了在通過fork創建數據通路的時候，由於原先框架會使得的連線未建立好就發送命令給client，client發起連接請求而導致錯誤。我在這裏重新調整了發送的順序，成功解決了這個問題。
- 增加多線程方式，如上述所描述，增加了多線程的方式來運作。
- 延遲發送226信息，更改了Proxy轉送226的時間，在傳輸完畢之後再傳送226，即修改相應速度，避免了數據在proxy還未傳完就發送226給client。issure#2

問題發現及解決

在這次實做的過程中，主要遇到了幾大困難。

第壹點，就是令牌桶多線程產生的問題，由於臨界區資源訪問速度不同，容易導致壹方饑餓，壹直無法消費令牌，或者產生令牌，就無從達到傳輸的目的了。解決辦法就是通過將他們的速度修改到相近。

第二點，就是緩存區隊列設計問題，原先只是把socket當成TCP的API來使用，沒有了解它其實是有自己的緩存區域的，通過自己再設計壹個隊列來暫存這些數據，其實遇到了很大的問題，儲存之後又如何消費令牌，涉及到了許多共享資源的問題。後來在了解了socket的設計之後發現，其實本身就自帶緩存區，解決了很多共享資源設計上的復雜度。

第三點，

- 如issure#1所描述的，框架本身存在的bug，導致客戶端連接失敗，需要進程重連的問題。也得到了修復。詳細可進issue查看解決過程，篇幅較長，這裏就不再壹壹贅述。
- 如issure#2所描述的，以25KB/sec上傳存在的響應超時的問題，以及如何通過`setsockopt`來進行調整緩存區的大小來解決這個問題。
- 如issure#3所描述的，以過快的速度下載壹個過小的文件，所存在的平均速度的問題。這主要是由於數據通路和命令通路不同步，以及如何通過延遲226 response code來解決這個問題。

測試結果

- 150KB/sec 上傳測試

```
Command:  STOR 1MB_testcasec7
Response:  150 Opening BINARY mode data connection for 1MB_testcasec7.
Response:  226 Transfer complete. 1,048,576 bytes transferred. 150.06 KB/sec.
Status:  File transfer successful, transferred 1.1 MB in 8 seconds
Status:  Retrieving directory listing...
```

- 100KB/sec 上傳測試

```
Command:  STOR 5MB_testcswcase
Response:  150 Opening BINARY mode data connection for 5MB_testcswcase.
Response:  226 Transfer complete. 5,242,880 bytes transferred. 100.06 KB/sec.
Status:  File transfer successful, transferred 5.3 MB in 55 seconds
```

- 100KB/sec 下載測試

```
Command:  RETR 5MB_testcase
Response:  150 Opening BINARY mode data connection for 5MB_testcase (5242880 Bytes).
Response:  226 Transfer complete. 5,242,880 bytes transferred. 100.03 KB/sec.
Status:  File transfer successful, transferred 5.3 MB in 54 seconds
Status:  Disconnected from server
```

- 75KB/sec 下載測試

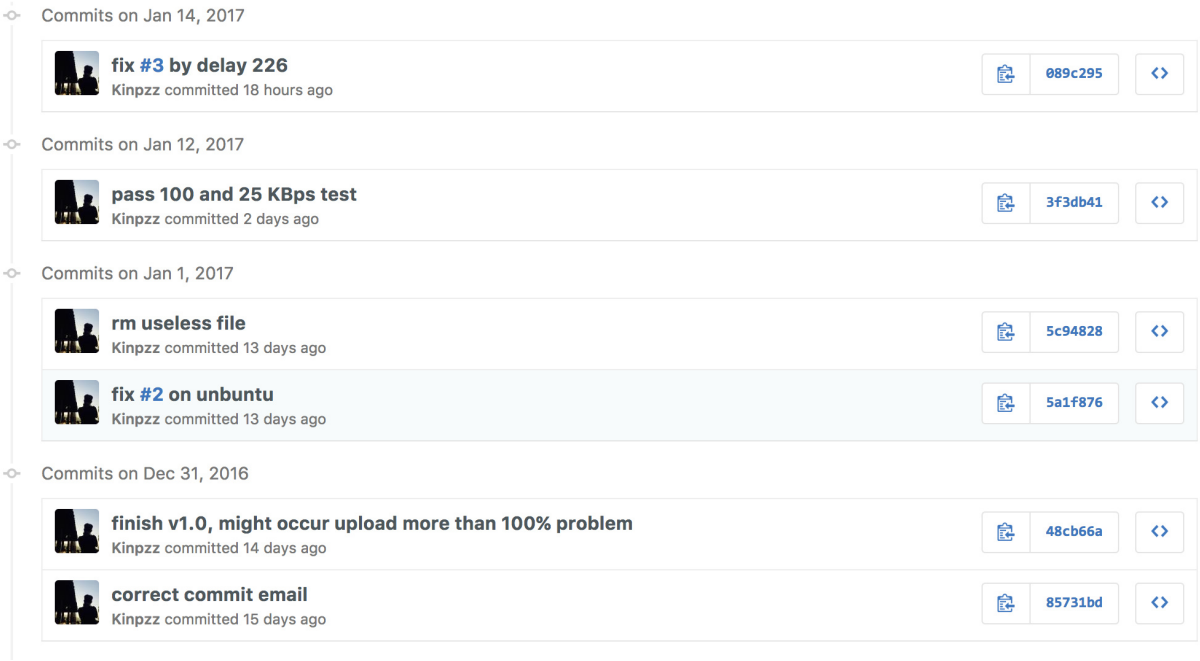
```
Response:  227 Entering Passive Mode (127,0,0,1,212,96)
Command:  RETR 10MB_testcase
Response:  150 Opening BINARY mode data connection for 10MB_testcase (10485760 Bytes).
Response:  226 Transfer complete. 10,485,760 bytes transferred. 75.40 KB/sec.
```

- 25KB/sec 上傳測試

```
Command:  STOR 10MB_testcase222c2
Response:  150 Opening BINARY mode data connection for 10MB_testcase222c2.
Response:  226 Transfer complete. 1,049,856 bytes transferred. 25.06 KB/sec.
Status:  File transfer successful, transferred 1.1 MB in 44 seconds
```

貢獻

- 顏鵬翔 x1057057
    - 寫proposal，制作投影片
    - 寫report，使用github記錄開發過程，進行demo
    - 實現token bucket限速功能，寫完代碼
    - 修復bug，解決壹些特殊情況下的問題，改進框架
    - 測試，根據測試結果修改代碼



- 朱占仕 x1052003
    - 測試