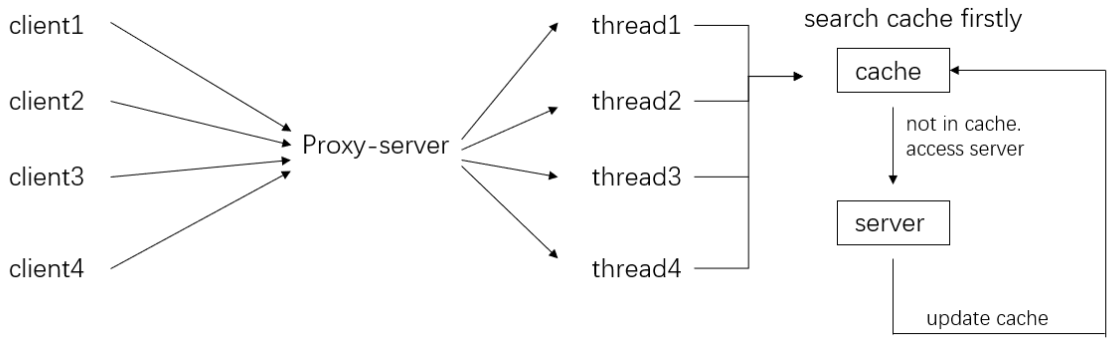
Proxy-lab

18300240005 罗铭源

实现思路

代理服务器架构



client向Proxy-server发起http请求，proxy为每一个client开启一个线程来并发地处理请求

处理一个请求时，proxy先在自身的cache中查找请求的资源，如果找到，就直接返回给client；否则，proxy访问server获得请求对应的http response，随后将资源返回给client，并且更新自身cache。如上图所示。

proxy main loop

代理服务器的主循环。开启一个 listen socket 监听由 argv[1] 指定的端口，当client与proxy建立连接时，为client新开一个连接用connection socket，将该socket作为参数传给 proxy_doit() 函数，并以该函数运行新线程来处理请求

同时还要初始化用于cache的互斥锁cache_lock

```
int main(int argc, char* argv[])
{
    int listenfd;
    char hostname[MAXLINE], port[MAXLINE];
    socklen_t clientlen;
    struct sockaddr_storage clientaddr;

    /* initialize the lock */
    Sem_init(&cache_lock, 0, 1);

    if( argc != 2 ){
        printf("usage:\n"
               "     proxy [portnumber]\n");
        exit(1);
    }

    listenfd = Open_listenfd(argv[1]);
    while (1) {
        clientlen = sizeof(clientaddr);
        int * connfdp = (int *)malloc(sizeof(int));
```

```
        *connfdp = Accept(listenfd, (SA *)&clientaddr, &clientlen);
            Getnameinfo((SA *) &clientaddr, clientlen, hostname, MAXLINE,
                        port, MAXLINE, 0);
            printf("Accepted connection from (%s, %s)\n", hostname, port);
        /* create the thread */
        pthread_t tid;
        Pthread_create(&tid, NULL, proxy_doit, connfdp);

    }
    return 0;
}
```

thread start function proxy_doit()

该函数分为以下几个部分：

- 分离线程，以便线程结束自动回收资源
- 解析http request的请求行
  - 先将请求行分离为 method, uri, version 三个字段
  - 调用parse_uri()解析uri，将uri分离为 host, port, path 三个字段
- 根据解析出的host和path，在cache中进行查找（访问cache时要通过cache_lock给cache上锁）
  - 若cache命中，将cache中的内容直接发送给client，并结束线程
  - 否则，进入下一步
- 与server建立连接
- 构造http request，这一步需要对client原有的request中的version以及部分header进行替换
  - 替换version为HTTP/1.0
  - 替换为Proxy-connection: close
  - 替换为Connection: close
  - 替换User-Agent
- 发送request，上一步中每完成一行header的构造，就直接发送给server，以节省proxy空间开销；最后发送 \r\n 作为结尾
- 从server接收响应并更新cache
  - 先将response转发给client，再更新cache
- 关闭使用的套接字，结束线程

```
void *proxy_doit(void * pfd){
    /* detach the thread itself */
    Pthread_detach(Pthread_self());

    char buf[MAXLINE], method[MAXLINE], uri[MAXLINE], version[MAXLINE];
    rio_t rio;
    int fd = *(int *)pfd;

    char * resp;
    size_t content_sz;

    Rio_readinitb(&rio, fd);
    if (!Rio_readlineb(&rio, buf, MAXLINE)){
        return NULL;
    }
    sscanf(buf, "%s %s %s", method, uri, version);
```

```
    if (strcmp(method, "GET")) {
        //clienterror(fd, method, "501", "Not Implemented", "Not Implemented");
        return NULL;
    }
    char host[MAXLINE], path[MAXLINE];
    int port;

    /* parse the request line */
    int r;
    if((r = parse_uri(uri,host,path,&port)) == -1){
        return NULL;
    }
    char cport[16];
    sprintf(cport, "%d", port);

    /*
     * find if the object is in the cache
     * need to lock the cache before searching
     */
    P(&cache_lock);
    cte_t *ctep = cache_hit(host,path);
    if(ctep != NULL){
        printf("cache hit. %s %s\n",host,path);
        resp = ctep->content ;
        content_sz = ctep->content_size;
        /* return the object to the client */
        Rio_writen(fd, resp, content_sz);
        Close(fd);
        V(&cache_lock);
        return NULL;
    }
    V(&cache_lock);

    /* not in the cache , connect with server */
    int sockfd = open_clientfd(host,cport);
    if(sockfd == -1){
        close(sockfd);
        return NULL;
    }
    /* request head line */
    sprintf(buf, "%s %s HTTP/1.0\r\n", method, path);
    Rio_writen(sockfd, buf, strlen(buf));
    /* generate the http request. read request from client and parse it to http
1.0 */
    while (Rio_readlineb(&rio,buf,MAXLINE) > 0){
        if(! strcmp(buf,"\r\n")) break;

        /* whether to reserve or change some headers of http */
        if(strstr(buf, "Proxy-Connection"))
            strcpy(buf, "Proxy-Connextion: close\r\n");
        else
        if(strstr(buf, "Connection"))
            strcpy(buf, "Connection: close\r\n");
        else
        if(strstr(buf, "User-Agent"))
            strcpy(buf, user_agent_hdr);

        /* send the request to the server line by line */
```

```
        printf("%s",buf);
        Rio_writen(sockfd, buf, strlen(buf));
    }
    /* send the end line */
    Rio_writen(sockfd, "\r\n", 2);

    /* recv from server */
    resp = calloc(MAX_OBJECT_SIZE, sizeof(char));
    content_sz = Rio_readn(sockfd, resp, MAX_OBJECT_SIZE);

    /* return the object to the client */
    Rio_writen(fd, resp, content_sz);
    /* update the cache with the new object */
    /* add lock before access cache */
    P(&cache_lock);
    cache_put(ctep, host, path, resp, content_sz);
    V(&cache_lock);
    /* close the socket */
    Close(sockfd);
    Close(fd);
    return NULL;
}
```

cache

见 `cache.h` `cache.c`

一个cache项由四个部分组成：

- hostname: 主机域名
- filepath: 资源路径
- content: 资源
- content_size: 资源大小

设计cache为全相联，含16个cache项

几个用于cache操作的函数:

- cte_flush() : 刷新一个cache项，即将其全部字段置0
- cte_match() : 判断一个cache项是否满足一个request
    - 通过比较
- cache_hit() : 针对一个request，判断cache是否命中
- cache_put() : 将一个object加入cache，或者更新某个cache项中object的内容

```
/*
 * cte_flush - flush a cache entry
 */
void cte_flush(cte_t * ctep){
    memset(ctep,0,sizeof(cte_t));
}

/*
 * cte_match - check if a cache entry matchs a request
 */
int cte_match(cte_t * ctep, char * host, char * path){
```

```
    return !strcmp(host,ctep->host) && !strcmp(path, ctep->path) ;
}

/*
 * cache_hit - search the cache to find a hit of request
 */
cte_t * cache_hit(char * host, char * path){
    int i;
    for(i = 0; i< MAX_CACHE_LINE; i++){
        if ( cte_match(cache + i, host, path) ){
            return cache + i;
        }
    }
    return NULL;
}

/*
 * cache_put - put or update an object into cache
 */
void cache_put(cte_t * oldctep, char * host, char * path, char * content, size_t
content_size){
    cte_t * dst = oldctep;
    if( !dst ){
        for(dst = cache; dst - cache < MAX_CACHE_LINE; dst++){
            if (dst->content_size == 0){
                break ;
            }
        }
        if(dst - cache == 16 )
            dst = cache;
    }
    cte_flush(dst);
    strcpy(dst->host, host);
    strcpy(dst->path, path);
    memcpy(dst->content, content, content_size);
    dst->content_size = content_size;
    printf("cache put: %s %s\n", host, path);
    //Write(1, dst->content, dst->content_size);
}
```

其它辅助函数

- parse_uri() : 将uri解析为 host, path, port 三个字段

```
int parse_uri(char * uri, char * host, char * path, int * port){
    char * pc;

    /* judge if uri is legal (begining with "http://") */
    if( strstr(uri,"http://") != uri ){
        fprintf(stderr, "Error: parse_uri() , illegal uri\n");
        return -1;
    }
    /* parse host , port number(if there is) */
    uri += strlen("http://");
    if((pc = strstr(uri, ":"))){
```

```
        *pc = 0;
        strcpy(host,uri);
        /* parse port number */
        *port = atoi(pc+1);
        uri  = pc + 1;
    }
    else
    if((pc = strstr(uri, "/"))){
        *pc = 0;
        strcpy(host,uri);
        uri = pc + 1;
    }
    /* parse path */
    if((pc = strstr(uri, "/"))){
        strcpy(path,pc);
    }
    else{
        path[0] = '/';
    }
    return 0;
}
```

io操作

使用csapp.h中定义的robust io

结果

./driver.sh

```
*** Cache ***
Starting tiny on port 14123
Starting proxy on port 27383
Fetching ./tiny/tiny.c into ./.proxy using the proxy
Fetching ./tiny/home.html into ./.proxy using the proxy
Fetching ./tiny/csapp.c into ./.proxy using the proxy
Killing tiny
Fetching a cached copy of ./tiny/home.html into ./.noproxy
Success: Was able to fetch tiny/home.html from the cache.
Killing proxy
cacheScore: 15/15

totalScore: 70/70
rivership@ubuntu:~/share/share/CSAPP_lab/proxylab-handout$
```