1 Writing a simple concurrent caching http proxy

Solution The created proxy is a simple implementation of a forward proxy. It supports only HTTP GET request. But allow for multiple clients to be served concurrently. Also, it supports response caching with least recently used (LRU) cache policy. It uses two additional custom C modules sbuf and cache:

- The sbuf is a simple implementation of a concurrent producer/consumer buffer. It uses semaphores to limit the number of readers and writers based on available item slots. And an exclusive lock for accessing buffered items.
- The cache is an LRU cache, with linked list cache structure. LRU is provided by maintaining a linked list queue for access history. It moves cache item access record to the end of the queue each time the item is accessed. When total cache size becomes greater than the limit, it evicts item with LRU records located at the beginning of the LRU queue.

The source code of proxy is shown on listing 1, the source code of sbuf and cache is shown on listing 2 and listing 3 correspondingly.

```c
#include <stdio.h>
#include "csapp.h"
#include "sbuf.h"
#include "cache.h"

/* Max size of a cacheable request */
#define MAX_OBJECT_SIZE 102400

/* HTTP requset line limits */
#define MAX_HOSTNAME_LEN 256
#define MAX_PORT_LEN 6
#define MAX_QUERY_LEN 2048

/* HTTP max header limit */
#define MAX_HEADERS_NUMBER 200

/* Thread pool constants */
#define NTHREADS 4
#define SBUFSIZE 16

/* Task required proxy headers */
static const char *user_agent_hdr = "User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:10.0.3) Gecko/20120305 Firefox/10.0.3\r\n";
static const char *connection_hdr = "Connection: close\r\n";
static const char *proxy_connection_hdr = "Proxy-Connection: close\r\n";

typedef struct uri_info
{
    char hostname[MAX_HOSTNAME_LEN];
    char port[MAX_PORT_LEN];
    char query[MAX_QUERY_LEN];
} Uri_info;

typedef struct headers
{
    int cout;
    char *data[MAX_HEADERS_NUMBER];
} Headers;

static void proxy(int fd);
static int parse_headers(rio_t *rp, Headers *headers);
static int parse_uri(const char *uri, Uri_info *uri_info);
static void forward(int fd, Uri_info *uri_info, Headers *headers);
static void clear_headers(Headers *headers);
static void *thread(void *vargp);
static int build_cache_key(char key[], size_t length, Uri_info *uri_info);
static void clienterror(int fd, char *cause, char *errnum, char *shortmsg, char *longmsg);
static void servererror(int fd);

/* Thread safe buffer */
static sbuf_t sbuf;
```

Listing 1 (Cont.): proxy.c

```c
/* Local proxy cache */
static Cache cache;

int main(int argc, char **argv)
{
    if (argc != 2)
    {
        fprintf(stderr, "usage: %s <port>\n", argv[0]);
        exit(1);
    }

    /* Blocking SIGPIPE signal */
    /* SIGPIPE signal will be send by write() when
     *  the connection socket closes prematurely.
     * An unhandled SIGPIPE signal terminates the program */
    sigset_t mask;
    if (Sigemptyset(&mask) < 0)
        exit(1);

    if (Sigaddset(&mask, SIGPIPE) < 0)
        exit(1);

    if (Sigprocmask(SIG_BLOCK, &mask, NULL) < 0)
        exit(1);

    /* Creatin listening socket */
    int listenfd;
    if ((listenfd = Open_listenfd(argv[1])) < 0)
        exit(1);

    /* Creating thread pool */
    if (sbuf_init(&sbuf, SBUFSIZE) < 0)
        exit(1);

    if (cache_init(&cache) < 0)
        exit(1);

    int i;
    pthread_t tid;
    for (i = 0; i < NTHREADS; i++)
        if (Pthread_create(&tid, NULL, thread, NULL) != 0)
            exit(1);

    struct sockaddr_storage clientaddr;
    socklen_t clientlen = sizeof(clientaddr);
    int connfd;
    while (1)
    {
        if ((connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen)) < 0)
            continue;

        /* Logging client info */
        char client_host[MAXLINE];
        char port[MAXLINE];
        Getnameinfo((SA *)&clientaddr, clientlen, client_host, MAXLINE, port,
            MAXLINE, 0);
        printf("Accepted connection from (%s, %s)\n", client_host, port);
        /* Adding request to the buffer */
        if (sbuf_insert(&sbuf, connfd) < 0)
            exit(1);
    }

    return 0;
}

static void *thread(void *vargp)
{
    if (Pthread_detach(pthread_self()) != 0)
        exit(1);

    int connfd;
    while (1)
    {
```

Listing 1 (Cont.): proxy.c

```c
        /* Dequeuing a request from the buffer */
        if (sbuf_remove(&sbuf, &connfd) < 0)
            exit(1);

        /* Request pipeline */
        proxy(connfd);
        Close(connfd);
    }
}

/* This is the main request pipeline routine.
 * It gets a connected socket descriptor. Parses the request,
 * sends a proxy request to the requested server and
 * forwards the response back to the client. */
static void proxy(int fd)
{
    rio_t rio;
    Rio_readinitb(&rio, fd);
    char buf[MAXLINE];
    /* Reading an HTTP request line from the connected socket */
    if (Rio_readlineb(&rio, buf, MAXLINE) < 0)
    {
        servererror(fd);
        return;
    }

    char method[MAXLINE];
    char uri[MAXLINE];
    char version[MAXLINE];
    if (sscanf(buf, "%s %s %s", method, uri, version) == EOF)
    {
        clienterror(fd, "", "400", "Bad request",
                    "Request is empty");
        return;
    }

    printf("%s", buf);
    if (strcasecmp(method, "GET"))
    {
        clienterror(fd, method, "501", "Not implemented",
                    "Proxy does not implement this method");
        return;
    }

    if (strcasecmp(version, "HTTP/1.0") &&
        strcasecmp(version, "HTTP/1.1"))
    {
        clienterror(fd, version, "501", "Not implemented",
                    "Proxy supports only HTTP/1.0(1.1) protocol versions");
    }

    Uri_info reciever;
    if (parse_uri(uri, &reciever) < 0)
    {
        clienterror(fd, uri, "400", "Bad request",
                    "Invalid request URI. "
                    "URI must have the following structure: "
                    "http[s]://{hostname}[:{port}]/{location}");
        return;
    }

    Headers headers;
    int status;
    if ((status = parse_headers(&rio, &headers)) < 0)
    {
        switch (status)
        {
        case -1:
            clienterror(fd, "", "413", "Entity is too large",
                        "Maximum header count or maximum header length is
                            exceeded");
            break;
        case -2:
```

Listing 1 (Cont.): proxy.c

```c
            servererror(fd);
            break;
        }

        clear_headers(&headers);
        return;
    }

    forward(fd, &reciever, &headers);
    clear_headers(&headers);
}

/* This is the part of the request pipeline, where the request is being send to the server
 * and the response is being forwarded to the client */
static void forward(int fd, Uri_info *uri_info, Headers *headers)
{
    jmp_buf fd_closed;
    if (setjmp(fd_closed))
        return;

    /* Returning cached data if present */
    void *cached = NULL;
    size_t cached_length;
    char key[MAXLINE];
    if (build_cache_key(key, MAXLINE, uri_info))
    {
        if (cache_get(&cache, key, &cached, &cached_length) > 0)
        {
            Rio_writen(fd, cached, cached_length, fd_closed);
            return;
        }
    }

    rio_t rio;
    int clientfd;
    if ((clientfd = Open_clientfd(uri_info->hostname, uri_info->port)) < 0)
    {
        switch (clientfd)
        {
        case -1:
            servererror(fd);
            break;
        case -2:
            clienterror(fd, uri_info->hostname, "400", "Host not found",
                        "The DNS entry for the hostname was not resolved");
            break;
        }

        return;
    }

    /* jmp if clientfd closes connection */
    jmp_buf client_close_conn;
    if (setjmp(client_close_conn))
    {
        Close(clientfd);
        servererror(fd);
        return;
    }

    /* Sending the HTTP request line to the client socket */
    Rio_readinitb(&rio, clientfd);
    char buf[MAXLINE];
    sprintf(buf, "GET ");
    Rio_writen(clientfd, buf, strlen(buf), client_close_conn);
    sprintf(buf, "%s", uri_info->query);
    Rio_writen(clientfd, buf, strlen(buf), client_close_conn);
    sprintf(buf, " HTTP/1.0\r\n");
    Rio_writen(clientfd, buf, strlen(buf), client_close_conn);
    /* Sending the HTTP headers from the request to the client socket */
    int host_in_headers = 0;
    int i;
```

Listing 1 (Cont.): proxy.c

```c
    for (i = 0; i < headers->cout; i++)
    {
        if (!host_in_headers && strstr(headers->data[i], "Host:"))
            host_in_headers = 1;
        else if (strstr(headers->data[i], "User-Agent:") ||
                 strstr(headers->data[i], "Connection:") ||
                 strstr(headers->data[i], "Proxy-Connection"))
            continue;

        Rio_writen(clientfd, headers->data[i], strlen(headers->data[i]),
            client_close_conn);
    }

    if (!host_in_headers)
    {
        /* Sending a Host header */
        sprintf(buf, "Host: ");
        Rio_writen(clientfd, buf, strlen(buf), client_close_conn);
        sprintf(buf, "%s", uri_info->hostname);
        Rio_writen(clientfd, buf, strlen(buf), client_close_conn);
        sprintf(buf, "\r\n");
    }

    /* Sending a User-Agent header */
    sprintf(buf, "%s", user_agent_hdr);
    Rio_writen(clientfd, buf, strlen(buf), client_close_conn);
    /* Sending a Connection header */
    sprintf(buf, "%s", connection_hdr);
    Rio_writen(clientfd, buf, strlen(buf), client_close_conn);
    /* Sending a Proxy-Connection header */
    sprintf(buf, "%s", proxy_connection_hdr);
    Rio_writen(clientfd, buf, strlen(buf), client_close_conn);
    /* Sending the end of the request */
    sprintf(buf, "\r\n");
    Rio_writen(clientfd, buf, strlen(buf), client_close_conn);

    /* jmp is the client close connection while the request is being
        forwarded */
    jmp_buf fd_closed_and_client_opened;
    if (setjmp(fd_closed_and_client_opened))
    {
        Close(clientfd);
        return;
    }

    /* Creating a temp cache buffer */
    size_t cached_len = 0;
    int cacheable = 1;
    char cache_buffer[MAX_OBJECT_SIZE];
    char *cache_free = cache_buffer;
    /* Transmitting the response from the client socket to the listening
        socket*/
    Rio_readinitb(&rio, clientfd);
    size_t response_len = 0;
    while ((response_len = Rio_readnb(&rio, buf, MAXLINE)) > 0)
    {
        if (cacheable)
        {
            if (cached_len + response_len > MAX_OBJECT_SIZE)
            {
                cacheable = 0;
            }
            else
            {
                memcpy(cache_free, buf, response_len);
                cached_len += response_len;
                cache_free += response_len;
            }
        }

        Rio_writen(fd, buf, response_len, fd_closed_and_client_opened);
    }

```

Listing 1 (Cont.): proxy.c

```c
    Close(clientfd);
    /* Caching the respone */
    if (!(response_len < 0) && cacheable)
    {
        char key[MAXLINE];
        if (build_cache_key(key, MAXLINE, uri_info))
            cache_add(&cache, key, cache_buffer, cached_len);
    }
}

static int parse_uri(const char *uri, Uri_info *uri_info)
{
    char buf[MAXLINE];
    char *index = buf;
    /* Parsing a protocol */
    while (*uri != ':')
    {
        if (*uri == '\0')
            return -1;

        *index++ = *uri++;
    }

    *index = '\0';
    if (strcasecmp(buf, "http") && strcasecmp(buf, "https"))
        return -1;

    /* Omitting duplicated slashes */
    uri++;
    while (*uri == '/')
        uri++;

    /* Parsing a hostname */
    index = buf;
    while (*uri != ':' && *uri != '/')
    {
        if (*uri == '\0')
            return -1;

        *index++ = *uri++;
    }

    *index = '\0';
    /* Filling Uri_info with the hostname */
    size_t hostname_len = strlen(buf) + 1;
    if (hostname_len > MAX_HOSTNAME_LEN)
        return -1;

    memcpy(uri_info->hostname, buf, hostname_len);
    /* Parsing a port value */
    index = buf;
    if (*uri == ':')
    {
        uri++;
        while (*uri != '/')
        {
            if (*uri == '\0')
                return -1;

            *index++ = *uri++;
        }

        *index = '\0';
        /* Filling Uri_info with the port value*/
        size_t port_len = strlen(buf) + 1;
        if (port_len > MAX_PORT_LEN)
            return -1;

        memcpy(uri_info->port, buf, port_len);
    }
    else
    {
        char *default_port = "80";
```

Listing 1 (Cont.): proxy.c

```c
        strcpy(uri_info->port, default_port);
    }

    uri++;
    /* Omitting duplicated slashes*/
    while (*uri == '/')
        uri++;

    /* Parsing a query */
    uri--;
    index = buf;
    while (*uri != '\0')
        *index++ = *uri++;

    *index = '\0';
    /* Filling Uri_info with query data */
    size_t query_len = strlen(buf) + 1;
    if (query_len > MAX_QUERY_LEN)
        return -1;

    memcpy(uri_info->query, buf, query_len);

    return 0;
}

static int parse_headers(rio_t *rp, Headers *headers)
{
    headers->cout = 0;
    char buf[MAXLINE];
    ssize_t line_len;
    /* Reading first header line from the connected socket*/
    if ((line_len = Rio_readlineb(rp, buf, MAXLINE)) < 0)
        return -2;

    /* Checking if the header line length is less than MAXLINE */
    if (line_len == MAXLINE - 1)
        if (buf[line_len - 1] != '\n')
            return -1;

    /* While the line is not the end of the request */
    while (strcmp(buf, "\r\n"))
    {
        if (headers->cout > MAX_HEADERS_NUMBER)
            return -1;

        /* Copying the header to the Headers struct*/
        char *header;
        if ((header = Malloc(line_len + 1)) == NULL)
            return -2;

        memcpy(header, buf, line_len + 1);
        headers->data[headers->cout++] = header;

        /* Reading next line */
        if ((line_len = Rio_readlineb(rp, buf, MAXLINE)) < 0)
            return -2;

        /* Checking if the header line length is less than MAXLINE */
        if (line_len == MAXLINE - 1)
            if (buf[line_len - 1] != '\n')
                return -1;
    }

    return 0;
}

static void clienterror(int fd, char *cause, char *errnum,
                        char *shortmsg, char *longmsg)
{
    jmp_buf fd_closed;
    if (setjmp(fd_closed))
        return;

```

Listing 1 (Cont.): proxy.c

```c
    char buf[MAXLINE];
    /* Print the HTTP response headers */
    sprintf(buf, "HTTP/1.0 %s %s\r\n", errnum, shortmsg);
    Rio_writen(fd, buf, strlen(buf), fd_closed);
    sprintf(buf, "Content-type: text/html\r\n\r\n");
    Rio_writen(fd, buf, strlen(buf), fd_closed);

    /* Print the HTTP response body */
    sprintf(buf, "<html><title>Proxy Error</title>");
    Rio_writen(fd, buf, strlen(buf), fd_closed);
    sprintf(buf, "<body bgcolor="
                 "ffffff"
                 ">\r\n");
    Rio_writen(fd, buf, strlen(buf), fd_closed);
    sprintf(buf, "%s: %s\r\n", errnum, shortmsg);
    Rio_writen(fd, buf, strlen(buf), fd_closed);
    sprintf(buf, "<p>%s: %s\r\n", longmsg, cause);
    Rio_writen(fd, buf, strlen(buf), fd_closed);
    sprintf(buf, "<hr><em>The Proxy</em>\r\n");
    Rio_writen(fd, buf, strlen(buf), fd_closed);
}

static void servererror(int fd)
{
    clienterror(fd, "", "500", "Internal server error",
                "Something went wrong");
}

static void clear_headers(Headers *headers)
{
    int i;
    for (i = 0; i < headers->cout; i++)
        free(headers->data[i]);
}

static int build_cache_key(char key[], size_t length, Uri_info *uri_info)
{
    size_t key_length = strlen(uri_info->hostname) +
                        strlen(uri_info->port) +
                        strlen(uri_info->query);
    if (length < key_length + 1)
        return 0;

    memset(key, 0, length);
    strcat(key, uri_info->hostname);
    strcat(key, uri_info->port);
    strcat(key, uri_info->query);
    return 1;
}
```

Listing 1: proxy.c

```c
#include "sbuf.h"

int sbuf_init(sbuf_t *sp, int n)
{
    if ((sp->buf = Calloc(n, sizeof(int))) < 0)
        return -1;

    sp->n = n;
    sp->front = sp->rear = 0;
    if (Sem_init(&sp->mutex, 0, 1) < 0)
        return -1;

    if (Sem_init(&sp->slots, 0, n) < 0)
        return -1;

    if (Sem_init(&sp->items, 0, 0) < 0)
        return -1;

    return 0;
}

void sbuf_free(sbuf_t *sp)
{
    Free(sp->buf);
}

int sbuf_insert(sbuf_t *sp, int item)
{
    if (Sem_wait(&sp->slots) < 0)
        return -1;

    if (Sem_wait(&sp->mutex) < 0)
        return -1;

    sp->buf[(++sp->rear) % (sp->n)] = item;
    if (Sem_post(&sp->mutex) < 0)
        return -1;

    if (Sem_post(&sp->items) < 0)
        return -1;

    return 0;
}

int sbuf_remove(sbuf_t *sp, int *item)
{
    if (Sem_wait(&sp->items) < 0)
        return -1;

    if (Sem_wait(&sp->mutex) < 0)
        return -1;

    *item = sp->buf[(++sp->front) % (sp->n)];
    if (Sem_post(&sp->mutex) < 0)
        return -1;

    if (Sem_post(&sp->slots) < 0)
        return -1;

    return 0;
}
```

Listing 2: sbuf.c

```c
#include "cache.h"

/* Max size of a cacheable request */
#define MAX_OBJECT_SIZE 102400
#define MAX_CACHE_SIZE 1049000

#define PAYLOAD(node_p) ((char *)node_p + sizeof(CacheNode))
#define NODE_SIZE(payload_size) (sizeof(CacheNode) + payload_size)

static void prepend_item(Cache *cache, CacheNode *item);
static void remove_item(Cache *cache, CacheNode *item);
static void append_lru(Cache *cache, LruNode *item);
static void remove_lru(Cache *cache, LruNode *item);

int cache_init(Cache *cache)
{
    cache->total_size = 0;
    cache->head = NULL;
    if (Sem_init(&cache->readers_lock, 0, 1) < 0)
        return -1;

    if (Sem_init(&cache->writers_lock, 0, 1) < 0)
        return -1;

    return 0;
}

int cache_add(Cache *cache, const char *key, const void *buf, size_t length)
{
    if (length > MAX_OBJECT_SIZE)
        return 0;

    /* Holding writers lock */
    if (Sem_wait(&cache->writers_lock) < 0)
        return -1;

    /* LRU eviction */
    while (cache->total_size + length > MAX_CACHE_SIZE)
    {
        CacheNode *to_evict = cache->lru_head->item;
        cache->total_size -= to_evict->size;
        remove_item(cache, to_evict);
        remove_lru(cache, to_evict->lru);
        printf("Evicted fromt the cache: %s, freed %d bytes.\n",
            to_evict->key, (int)to_evict->size);
        free(to_evict->lru);
        free(to_evict);
    }

    /* Creating a new cache node */
    CacheNode *new_item;
    if ((new_item = Malloc(NODE_SIZE(length))) < 0)
        return -1;

    /* Creating a key copy */
    char *key_copy;
    if ((key_copy = Malloc(strlen(key) + 1)) < 0)
        return -1;

    memcpy(key_copy, key, strlen(key) + 1);
    /* Allocating a new lru node */
    LruNode *new_lru;
    if ((new_lru = Malloc(sizeof(LruNode))) < 0)
        return -1;

    new_lru->item = new_item;
    /* Updating item fields */
    new_item->key = key_copy;
    new_item->size = length;
    new_item->lru = new_lru;
    /* Copying data to the cache node */
    void *data = PAYLOAD(new_item);
    memcpy(data, buf, length);
```

Listing 3 (Cont.): cache.c

```c
    /* Updating linked lists */
    prepend_item(cache, new_item);
    append_lru(cache, new_lru);
    printf("Writen to the cache %s. %d bytes.\n", new_item->key,
        (int)new_item->size);
    if (Sem_post(&cache->writers_lock) < 0)
        return -1;

    return 0;
}

int cache_get(Cache *cache, const char *key, void **item, size_t *length)
{
    int is_cached = 0;
    /* Aquiring writers lock if it is the first reader */
    if (Sem_wait(&cache->readers_lock) < 0)
        return -1;

    cache->readers_count++;
    if (cache->readers_count == 1)
        if (Sem_wait(&cache->writers_lock) < 0)
            return -1;

    if (Sem_post(&cache->readers_lock) < 0)
        return -1;

    /* Key sequential lookup */
    CacheNode *current;
    for (current = cache->head; current != NULL; current = current->next)
    {
        if (!strcmp(current->key, key))
        {
            /* Moving the lru node to the end of the LRU list */
            if (Sem_wait(&cache->readers_lock) < 0)
                return -1;

            remove_lru(cache, current->lru);
            append_lru(cache, current->lru);
            if (Sem_post(&cache->readers_lock) < 0)
                return -1;

            /* Copying the cached value */
            if ((*item = Malloc(current->size)) < 0)
                return -1;

            memcpy(*item, PAYLOAD(current), current->size);
            *length = current->size;
            is_cached = 1;
            break;
        }
    }

    /* Releasing writers lock if it is the last reader */
    if (Sem_wait(&cache->readers_lock) < 0)
        return -1;

    cache->readers_count--;
    if (cache->readers_count == 0)
        if (Sem_post(&cache->writers_lock) < 0)
            return -1;

    if (Sem_post(&cache->readers_lock) < 0)
        return -1;

    if (is_cached)
        printf("Cache hit %s\n", key);
    else
        printf("Cache miss %s\n", key);

    return is_cached;
}

/* Appending an item to the end of the LRU linked list */
```

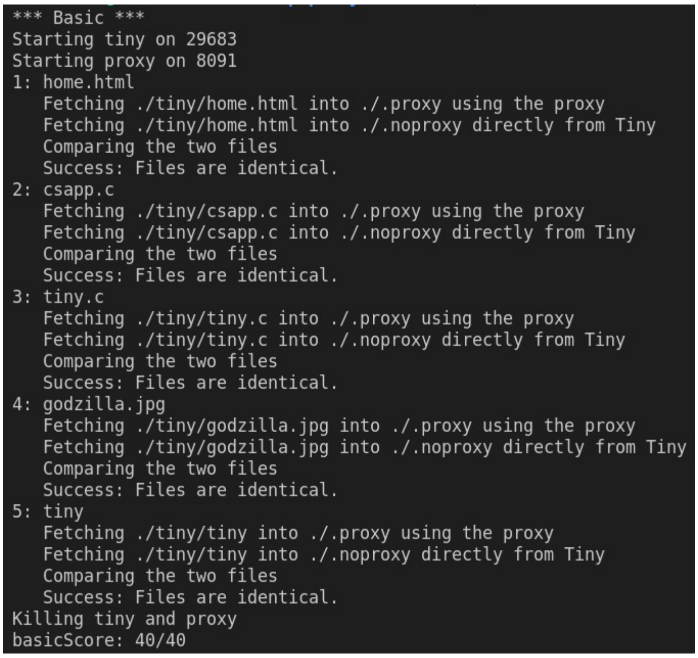
Listing 3 (Cont.): cache.c

```c
static void append_lru(Cache *cache, LruNode *item)
{
    item->next = NULL;
    item->prev = cache->lru_tail;
    if (cache->lru_tail != NULL)
        cache->lru_tail->next = item;
    else
        cache->lru_head = item;

    cache->lru_tail = item;
}

/* Removing an item from the LRU linked list*/
static void remove_lru(Cache *cache, LruNode *item)
{
    LruNode *prev = item->prev;
    LruNode *next = item->next;
    if (prev != NULL)
        prev->next = next;
    else
        cache->lru_head = next;

    if (next != NULL)
        next->prev = prev;
    else
        cache->lru_tail = prev;
}

/* Prepending an item to the start of the cached items linked list */
static void prepend_item(Cache *cache, CacheNode *item)
{
    item->prev = NULL;
    item->next = cache->head;
    if (cache->head != NULL)
        cache->head->prev = item;

    cache->head = item;
    cache->total_size += item->size;
}

/* Removing an item from the cached items linked list */
static void remove_item(Cache *cache, CacheNode *item)
{
    CacheNode *next = item->next;
    CacheNode *prev = item->prev;
    if (prev != NULL)
        prev->next = next;
    else
        cache->head = next;

    if (next != NULL)
        next->prev = prev;
}
```

Listing 3: cache.c

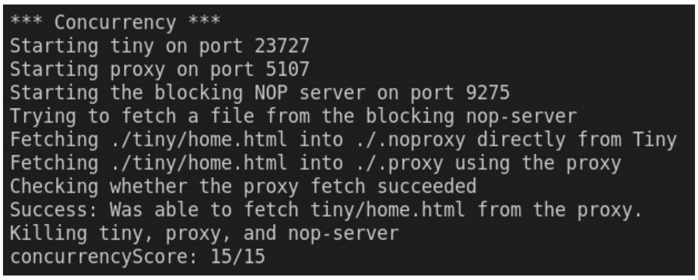
Results The proxy was run against the evaluator which verifies basic requirements.

- Validity of the proxy response. Figure 1.
- Support of concurrent requests. Figure 2.
- And functionality of the caching policy. Figure 3.



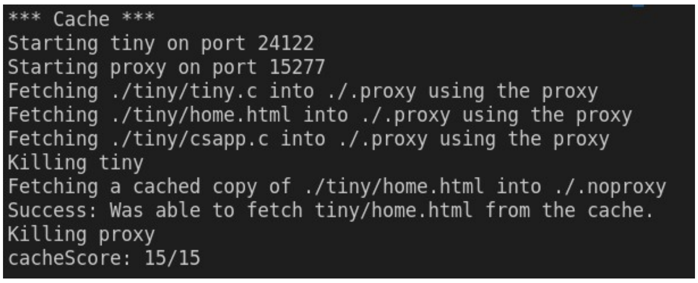
```
*** Basic ***
Starting tiny on 29683
Starting proxy on 8091
1: home.html
   Fetching ./tiny/home.html into ./.proxy using the proxy
   Fetching ./tiny/home.html into ./.noproxy directly from Tiny
   Comparing the two files
   Success: Files are identical.
2: csapp.c
   Fetching ./tiny/csapp.c into ./.proxy using the proxy
   Fetching ./tiny/csapp.c into ./.noproxy directly from Tiny
   Comparing the two files
   Success: Files are identical.
3: tiny.c
   Fetching ./tiny/tiny.c into ./.proxy using the proxy
   Fetching ./tiny/tiny.c into ./.noproxy directly from Tiny
   Comparing the two files
   Success: Files are identical.
4: godzilla.jpg
   Fetching ./tiny/godzilla.jpg into ./.proxy using the proxy
   Fetching ./tiny/godzilla.jpg into ./.noproxy directly from Tiny
   Comparing the two files
   Success: Files are identical.
5: tiny
   Fetching ./tiny/tiny into ./.proxy using the proxy
   Fetching ./tiny/tiny into ./.noproxy directly from Tiny
   Comparing the two files
   Success: Files are identical.
Killing tiny and proxy
basicScore: 40/40
```

Figure 1: Basic validity check



```
*** Concurrency ***
Starting tiny on port 23727
Starting proxy on port 5107
Starting the blocking NOP server on port 9275
Trying to fetch a file from the blocking nop-server
Fetching ./tiny/home.html into ./.noproxy directly from Tiny
Fetching ./tiny/home.html into ./.proxy using the proxy
Checking whether the proxy fetch succeeded
Success: Was able to fetch tiny/home.html from the proxy.
Killing tiny, proxy, and nop-server
concurrencyScore: 15/15
```

Figure 2: Concurrency check



```
*** Cache ***
Starting tiny on port 24122
Starting proxy on port 15277
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
```

Figure 3: Cache check