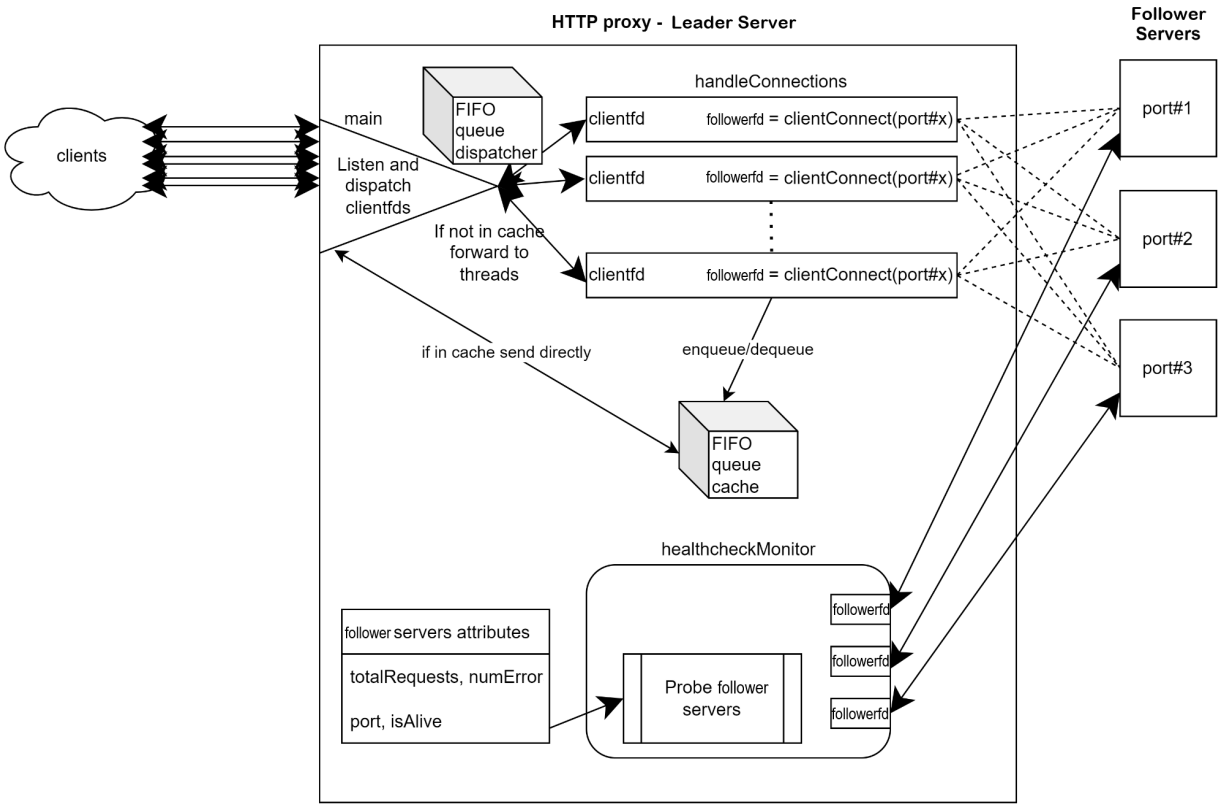
Andy Choi
Achoi15@ucsc.edu
Assignment 3: HTTP proxy
CSE 130: Principles of Computer System Design, Fall 2021

Objective:

The objective of this project is to create an HTTP reverse proxy that will support GET requests, persistent connections, load balancing, and caching. Concurrency is needed for servicing multiple client requests at the same time. The proxy server will be the manager server which distributes the client's file descriptors to one of the prioritized follower servers for handling. The proxy acts as a client for the follower servers and acts as a server for incoming request connections. Load balancing requires probing the "healthcheck" of the followers to prioritize the follower server that has the least amount of requests. For caching, if the GET request resource is in cache the server would send a HEAD request to the follower server holding the resource and parse through the Last_Modified header line to determine if the stored resource is newer or the same age.

Implementation:



Structures:

circQueue - a data structure for dispatching client file descriptors to handleConnection

(from asgn2) and for caching.
fileInfo - contains variables of a file.
httpServerInfo - contains variables necessary for processing received requests (from asgn2)
httpProxy - contains a struct pointer to followerServerInfo.
thread - contains thread IDs and variables necessary for handleConnection
thread. followerServerInfo - contains attributes of a follower server.

Threads:

1. Main - dispatcher thread in charge of adding requests to the queue and incrementing the number of connections for health checking after R requests. (The idea to do incrementation in dispatcher came from Christian Norris)
2. healthcheck - thread in charge of probing the follower servers which updates our follower servers' variables of total entries, number of errors, and status.
3. handleConnection - a pool of worker threads in charge of obtaining client file descriptor, establishing a connection between client and follower servers until the connection is interrupted.

Connection Forwarding:

handleConnection is similar to asgn2, which follows a dispatcher-worker model. Bridge_loop uses select() to monitor multiple file descriptors which it waits until one or more of the file descriptors become ready for forwarding.

Params: threadObject for incrementing countConnectionsHC for health checking and sending header lines to the priority server.
Returns: returns when the proxy server terminates.
void *handleConnection(void *arg):
After unlocking queueMut and obtaining a client file descriptor.

Read request
Process 400 and 501 requests.
Set priority port.
Create a client socket from the priority follower's port.
Send a header request to the server.
Call bridge_loop to forward all messages between client and follower server until connection is interrupted.

Functions:
Functions from asgn2 are omitted.

int create_client_socket(uint16_t port):
Starter code given.
Creates a socket for connecting to a server running on the same computer, listening on the specified port number.

https://git.ucsc.edu/cse130/spring20-palvaro/achoi15/-/blob/master/asgn3/loadbalancer.c

int bridge_connections(int fromfd, int tofd):
Send up to BUFFER_SIZE bytes from fromfd to tofd.
Params: fromfd, tofd are valid sockets.
Returns: number of bytes sent, 0 if connection closed, -1 on error.

https://git.ucsc.edu/cse130/spring20-palvaro/achoi15/-/blob/master/asgn3/loadbalancer.c
Forwards all messages between both sockets until the connection is interrupted.
Params: sockfd1, sockfd2 are valid sockets.
void bridge_loop(int sockfd1, int sockfd2):
    While 1
        Set file descriptors sets for select
        Call select to return the number of fds ready for reading
        Call bridge_connections to forward connection
        If bridge_connections returns -1,
        Either the client didn't send anything or the server didn't send anything.

void send500(int fd):
Sends 500 internal server errors to a given file descriptor socket.
Params: file descriptor of client.

Load Balancing:

Healthcheck thread requires a health mutex and a health condition variable to work in parallel with the dispatcher thread and worker threads.
In the dispatch thread in main, after every R request and when the proxy first starts, the dispatcher will signal the healthcheck thread to start probing the follower servers to determine which servers are alive and their entries.
Then getFollowerPort() determines the priority server and returns its port.

Functions:

Params: proxyObject for follower servers.
Returns: returns when the proxy server terminates.
void *healthcheck(void *arg):
    Probe follower servers.
    While 1
        Wait on health condition in main
        Probe follower servers.

Params: proxyObject for follower servers.
Returns: returns after every follower servers have been probed.
void probeFollowerServers(proxyObj *proxyPtr):
    Lock health mutex.
    For 0 to the number of follower servers
        Create a client socket from the server port. If fail set isAlive to false and continue.

Create a health check message and send. If fail set isAlive to false and continue.
Process health check response by:
Receive response from server. If fail set isAlive to false and continue.
Checking if the response contains 200 OK. If fail set isAlive to false and continue.
Check if there exists a double carriage return. If fail set isAlive to false and continue.
Sscanf response until sscanf gets the correct contentLength amount.
Set follower server number of errors and total requests as well as isAlive to true. Close client socket.

Unlock health mutex.

Params: proxyObject for follower servers.
Returns: returns port of proirity follower server or -1 if no priority exists.
Int getFollowerPort(proxyObj *proxyPtr):

For 0 to the number of follower servers

Check if isAlive

Then find the index of the server with the least amount of total requests. If tied to total request or tied to number errors, compare the number of errors and choose the index of the least amount of errors.

Check if index is not -1, increment total requests by 1 for health check entry
Else return -1.

Return port of index.

Caching:

A file struct and a circular queue are used for caching. It follows a FIFO implementation. Enqueueing and dequeuing are done in handleConnections, more specifically, in bridge_connections when sending bytes from the follower servers to the client. Bridge_loop will continue to call bridge_connections until everything is sent from the server to the client. This is where the file contents are set using memset().

```
typedef struct circQueue {
  int *buff; // for dispatcher and consumer queue
  int size;
  int head;
  int tail;
  int count;
  struct fileInfo *file;
} queue;

struct fileInfo {
  char *filename;
  bool needsUpdate;
```

```
  int maxFileSizeCache;
  ssize_t contentLength;
  int headerSize;
  uint8_t *buff; // file's content
  char *age; // file's last modified age
};
```

circularQueue.c:
initCache initialize fileInfo and cacheQueue
queue *initCache(int size, int maxFileSizeCache)

Checks if file is in cache
bool isInCache(queue *q, char *filename)

Returns the index of filename in cache. -1 if not
int getIndex(queue *q, char *filename)

Enqueues file to cache, sets filename and sets the age when the file is enqueued.
void enqueueCache(queue *q, char *filename)

Removes file from cache and resets it's buffer.
Returns the filename that was removed.
char *dequeueCache(queue *q)

Clear entry's file contents and last modified age.
void clearFileEntry(queue *q, int index)

lib.c:
Params: proxyDate is the date of the cached file and followerDate is the date of the stored file. Returns: true if proxyDate is the same age of newer than stored file.
bool fileIsNewer(char *proxyDate, char *followerDate):
    Parse through dates
    Compare dates

httpProxy.c:
int main(int argc, char *argv[]):
    If the cache option is true, initialize cacheQueue with capacityCache and maxFileCacheSize.
Params: httpObj similar to asgn2, thread for caching, and client's file descriptor. Returns: returns 0 if forwarding a connection is needed else -1 if not.
int processRequest(httpObj *data, int clientfd, threadObj *thread):
    Check if cache option is true
        Check If file is in cache
            send HEAD request to priority server

Process head response by parsing the “last-modified” line and comparing it with the file in cache.
If the stored file is the same age or newer, respond without forwarding
Else forward the request.
Else if not in cache
Forward connection.

bridge_connections first parses the GET header to set content length for caching and then enqueues the file to cacheQueue then sends up to BUFFER_SIZE bytes from fromfd to tofd.
Params: fromfd, tofd: valid sockets, httpObj, and threadObj for setting contentLength and getting variables for caching.
Returns: number of bytes sent, 0 if connection closed, -1 on error.
int bridge_connections(int fromfd, int tofd, httpObj *data, threadObj *thread):
Recv data...
Parse the GET header to obtain content length.
If filesize <= maxCacheSize
If queue is full, dequeue cacheQueue
Then enqueue to cacheQueue

Add file contents to file struct buffer.
Send data...

Main:

Int main(int argc, char *argv[]):
Parse command line arguments using getopt.
Initialize httpProxy attributes.
Initialize followerServerInfo’s attributes.
Initialize health check thread.
initialize thread attributes and handleConnection thread.
Initialize cache queue.
While 1
Add requests to the queue for consumer threads.
Increment connection count
If a health check is due, signal healthcheck to start probing.

Questions and Answers

Which variables are shared between threads?
The variables that are shared between threads are attributes of httpObject, proxyObject, threadObject, and the queue for caching. select() is used for synchronous I/O multiplexing.

When are they modified?
httpObject - when reading and processing requests.
proxyObject - in probeFollowerServers() and getFollowerPort().
threadObject - called to get the maximum number of file sizes and if the cache option is enabled.
cacheQueue - for enqueuing, dequeuing, and setting up their attributes.

Where are the critical regions?
The critical region is after locking the queue mutex in handleConnections for de-queueing clientfd to a worker.

Design choices for the assignment of connections to servers and detection of problems in the servers:

I decided to do balancing on a per-connection basis instead of per-request because the complexity for per-connection seems to be easier than per-request for this assignment.

I decided to make my health check non-blocking since Peter Alvaro commented “...you are saying that it is better to do nothing at all than to do something wrong.“ when talking about making it blocking in this piazza post: https://piazza.com/class/ku33dckjdub51n?cid=256. And that design choice of non-blocking seems reasonable when we’re load balancing because the healthcheck could take a long time to process, thus preventing current and incoming requests from being processed.