HTTP SERVER

Final project paper

Course: CSC-337 – Data & communications
Year: 2013, fall term
Author: Udrzal, Vojtech
Professor: Hedrick, James

Table of Contents

Anotation

My project's goal was to design and implement basic HTTP server that would be able to server static html website to user. It should use multithreading, cache, be able to serve different domain names and be configurable over config file.

From this project I expected to learn practical info about HTTP protocol and by implementing HTTP server deeply understand how this protocol works. Furthermore, I wanted to develop multithreaded application which would take advantage of using more threads to improve my skills in that area, since it takes takes programming to whole new level because of needed synchronizations, mutexes etc.

Introduction

Technologies

Choosing the right technologies for a project is one of the most important things that influence the whole project for its lifetime. It has high impact on every part of the project, including time demands and the outcome. Thus, choosing technologies should not be underestimated and all factors including developers skills and preferences must be evaluated.

Programming language

I would consider myself to be advanced in several programming languages, including PHP, Java or C++ with several successfull job projects developed in the past, so I had wide range to choose from. After some time it came down to Java or C++, which were both suitable for this project. Finally, I picked C++, since this language is more powerful for this kind of stuff, not to mention that most of all servers are developed in C++, if not assembler.

Operating system

Other technology aspect is the platform (operating system/enviroment). For a while, I thought of developing this server for Windows systems, however, I did not wanted to encounter some of the Window's tricky problems so I decided to go with the open source platform – Linux – namely Ubuntu 12.04. As multithreading library I used POSIX Threads, which is most likely the best for Linux platform.

Development enviroment

Last but not least was the selection of IDE (*Integrated development environment*). I could have just used text editor with syntax highlighting (like Sublime) and compile program over console, but it's not much convinient especially when it comes to debugging and tuning. Therefore, I used Netbeans editor, which has reasonable debugger and other features. For advanced debugging of program failures (segmentation faults) or memory leaks, I used Valgrind.

Features and functions

In the following sections I will describe each server feature separately together with small snippets of source code and potential developing problems.

Multithreading

Multithreading allows application to have more than one executing sequences of commands, which increases performance of application by parallel executing of commands. In addition, for my project, it allows the server to serv and send content to two or more visitors at the same time.

Producer – consumer model

In my application, I used producer – consumers model. The model is fairly simple and powerful. One thread (producer) is running in infinite loop only accepting connection comming to our server and only inserting them to a visitors que. The producer thread does nothing with the connections and does not

care whoever the visitor is or whatever he wants. On the other hand, there are multiple consumer threads running in infinite *work* loop. When there is a visitor in a que (pushed in by producer thread), the consumer thread picks him up and processes his request. To prevent waste of resources I used Pthread's feature semaphores, so when the visitor's que is empty, all threads are “sleeping” and not using any computing power of computer's processor. As soon as some visitor is inserted to the que, one of the sleeping threads wakes up and processes his request.

Mutex synchronization

With multiple threads come hand-in-head synchronization problems. When we have 2 or more threads working with the same memory, we have to synchronize them in such a way, that one thread cannot write to a memory from which the other thread is currently reading. This behaviour causes program failures known as *segmentation faults*. Synchronizing in Posix threads library is achieved by using Mutexes.

Code snippet

Producer loop:

```
     while (true) {
         inSocket = listenSocket->AcceptConn(inAddr); //accept connection
         CThread::InsertToQue(new CVisitor(inSocket, inAddr)); //insert to que
     }
```

Consumer loop:

```
void* CThread::threadLoop() {
    CVisitor* visitor;TQue* que;
    while (true) {
        sem_wait(mQueSem_t);//semaphore pasive waiting.
        pthread_mutex_lock(&mQue_mutex_t); //mutex – only one thread can be on following line code
        que = GetFromQue();
        pthread_mutex_unlock(&mQue_mutex_t);//unlock mutex
        visitor = que->mVisitor;
        visitor->ProcessRequest();
        visitor->CloseConn();
    }
```

Processing request

Now let's look closer on how my http server processes visitors request. The processing method is called by consumer thread on instance of class CVisitor as we can see in code snippet above. At first, method Cvisitor::ProcessRequest() parses visitors request header by header so we know who the user is (more specifically, what computer he has, so we can respond in appropriate way – example character coding, file compression) and what he wants.

Reading request headers

Host Header

First comes host header. This header tells the server, which domain user wants to visit. Suppose we have website union.edu and google.com hosted on same server. Then users request for visiting union's website will have following host header: *Host: www.union.edu*.

GET Header

GET header tells the server, which page or file user wants to access. If we wanted to see campus section on Union's website, our request header would look like this:

GET /campus/ HTTP/1.1
Host: www.union.edu

HEAD Header

Head header, same as GET header, tells the server, which file user wants to see. The difference is, that if the file is no different from the file previously sent to visitor, server will just reply with simple message "did not change" and no data is transferred. More on this matter is explained in *User cache* chapter.

Other Headers

Apart from the headers above, my server also parses several more headers. However, they are not that

important and actually are not needed for the most basic functionality of http server. They just improve and make the communication between visitor and server more reliable and effective. For example, these are the headers

Parsing header code snippet

```
CHttpRequest::ReadRequest() {
    int headerLen;
    string headerLine;
    string headerItem;
    string headerItemContent;
    while ((headerLen = readHeaderFromSocket(headerLine))) {
        if (parseHeaderLine(headerLine, headerItem, headerItemContent))continue;
        switch (knownHeader(headerItem)) {
            case hHOST:
                mHost = processHost(headerItemContent);
                mDomain = G_Config->GetDomainByName(mHost);
                if (mDomain == NULL)throw CErrorException(NOT_FOUND, mHost + " domain does not exist.");
                break;
            case hHEAD: processGet(headerItemContent, mHEAD);
                break;
            case hGET: processGet(headerItemContent, mGET);
                break;
            case hUSER_AGENT: mUserAgent = headerItemContent;
                break;
            case hCHARSET:mCharset = headerItemContent;
                break;
            case hACCEPT:mAccept = headerItemContent;
                break;
            case hENCODING:mEncoding = headerItemContent;
                break;
            case hETAG:mETag = headerItemContent;
                break;
            default:
                G_ServerLogger->append("Unknow header: " + headerItem + headerItemContent, WARNING_LOG);
        }
    }
    return STATUS_OK;
}
```

Responding to visitor

In case there was error while parsing visitor's request, error page is sent back. That can happen for example when he wants to connect to a domain, that is not hosted on this server or when he wants a file that is not present (error code 404). If the file is on server, first it will try to look it up in its internal memory cache. If the server has the file in memory already, then there is no need to read it from a disk (and thus it saves time, because reading from disk is way more slower than reading from memory). If it was not in the cache, server will read it from a disk and store to cache. Then, all required response headers are prepared (such as content-lenght, status code, etag) and the response is flushed to client. At the end, consumer thread just closes socket to visitor.

Code snippet

```
try { //looking up file in cache
      file = CFile::FindFileInCache(filePath);
      if (file == NULL) { //file was not in cache, so read the file from this
          file = new CFile(mHttpRequest->mDomain->GetDomainPath() + mHttpRequest->mFileName.substr(1,
        mHttpRequest->mFileName.length()), ios::binary);
      }
      file->UseFile();//claim rights on file (mutex)
      mHttpResponse->mFileName = file->mFileName;
      buffer = file->GetFileBuffer();
  } catch (CErrorException e) {
      delete file;
      mHttpResponse->mDomain->mDomainLogger->append(e.GetErrorMessage(), INFO_LOG);//in case of error, log it
      statusCode = e.GetErrorCode();
  }
  mHttpResponse->SetStatus(statusCode);//set all required response headers
  mHttpResponse->SetDefaultHeader();
  mHttpResponse->SetETag(file->GetETag());
  mHttpResponse->SetContentTypeHeader();
  mHttpResponse->SetContent(buffer); //sets content-lenght and content itself
  mHttpResponse->SendResponse();
  file->UnuseFile(); /* clear mutex */ return statusCode;
```

Client side cache

Client side cache (browser cache) reduces data transfers between visitor and client. It is based on an idea, that if the user had the file already, then there is no reason to waste traffic and transfer it again. Instead, it will just be summoned from visitor's computer memory. For example, if we visited Union's website and there was no browser cache, every time we load some page the server would have to send us Union's logo each single time. This does not only wastes traffic or resources, but also significantly slowers the loading time of the website. On the other hand, we need to know when the logo changes and in that case update it, so we won't see the same Union logo for years. For this, my server uses eTags. Each time server sends file to client, it appends to the response a random hash code generated for the specific file version. Then, when visitor wants to load a file that is in his cache already, he can just send the server HEAD request and this hashcode. If the hashcode from user is equal to code in server's memory, it will just reply "304 – content-not-modified". If it's not the same, server will send back the whole file.

Code snippet

In this code snippet server compares etag from request with etag of file version in memory. If it's the same, we just send NOT_MODIFIED message back to user.

```
if (mHttpRequest->mETag == file->mETag) {
      statusCode = NOT_MODIFIED;
      sendErrorCode(statusCode);
      if(file) file->UnuseFile();
      return statusCode;
  }
```

Server side cache

Unlike client cache, server side cache improves performance of the server only. It stores recently sent files in RAM memory, so every time user wants some file, server will try to look it up in memory, first. So if there is some frequently used file, server doesn't need to access hard disk each time and read from

it. Reading from hard disk is significantly slower than reading from RAM memory. For looking up a file in memory, my server uses C++ STL structure *map.* For keeping track of expired files, I put them to a que. Each time file is used, it's moved to the front of que and is assigned timestamp. From time to time server goes from the end of a que and deletes timeouted files from memory, to prevent server of using too much memory space.

Code snippet - deleting expired files

```
void CCacheQue::DeleteExpiredFiles() {
    if (mListSize == 0)return;
    TNode* prev = mTail;
    while (prev) {
        if ((time(NULL) - prev->mVal->GetTimeStamp()) <= G_Config->FILE_CACHE_EXPIRATION_TIME) {
            break; //keep loop until the file shoudl stay in cache
        }
        TNode* toDelete = prev;
        if (toDelete->mVal->mUsingCnt == 0) {
            prev = prev->prev;
            mListSize--;
            delete toDelete;
        }else{
            break;
        }
    }
    mTail = prev;
    if (mListSize == 0)mHead = mTail = NULL;
}
```

Logging

My server supports several logging levels either to console or to logfile. Log levels, as almost everything else, is configurable over config file. It has up to 8 logging levels ranging from logging only FATAL errors over NETWORK messages to DEBUG messages.

Configuration over config file

Everytime server is started, first it reads configuration file in its root directory named *server.conf.* Then, it parses this file and reads the configuration for server to run. It is possible to configurate almost everything. For example we can define listening port, how many threads should be used, how long files

should stay in cache, all domain names, its aliases and data directories. Last but not least, the logging levels.

Example of configuration file

```
#CONFIG FOR 1337 HTTP Server
#comments by # character works!! :)
ListenPort 8057
Threads 5
FileCacheTime 10 #in seconds

ServerFileLogLevel FATAL ERROR WARNING INFO DEBUG0
ServerConsoleLogLevel FATAL ERROR WARNING INFO DEBUG0 DEBUG1 DEBUG2

VirtualServer
Domain testujuto.local
Alias seznam.local testname3.local
DataFolder /home/frox/Dropbox/Dropbox/private/HTTP_SERVER/examples/www/woodproject.cz/ #'/' must be at the end

VirtualServer
Domain site2.local
Alias sitename3.local sitename3.local
DataFolder /home/frox/Dropbox/Dropbox/private/HTTP_SERVER/www2/ #'/' must be at the end
```

Conclusion

After plenty of hours developing and debugging this server, I have finally managed to complete the project to such a state, that it is able to server static html website. I'm happy that I have finally developed application using POSIX threads library, which will definetily come in hand in the future. Nevertheless to say, that I expected higher performance boost when using more threads. To be more specific, there is less than significant improvement if using 10 threads instead of 1. In my opinion it is due to my computer's processor, which is not that good when it comes to parallel computing. Also, I might have used to much mutexes on wrong places so it makes the whole server to slow down (unwanted passive waiting, dead-lock).

As one of the features for improvement in the feature I think of adding dynamic webpages support. The only problem is, that I don't know yet, how to implement PHP executive enviroment to my code. I

definitely would be happy If I could have used my server in feature to host my personal website presentation!

Sources

[1] HTTP protocol http://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol

[2] HTTP protocol specification http://www.w3.org/Protocols/rfc2616/rfc2616.html

[3] POSIX threads tutorial https://computing.llnl.gov/tutorials/pthreads/