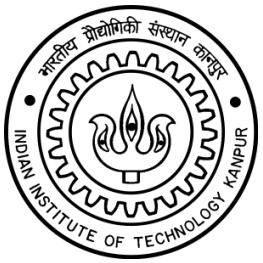
Indian Institute of Technology
Kanpur

CS425 - Computer Networks

Project 2

Nishant Rai



Course Instructor
Dr. Sandeep Shukla

CS425 Project 2

Nishant Rai, 13449

September 2, 2016

1 Aim

To build a web proxy capable of accepting HTTP requests, forwarding requests to remote (origin) servers, and returning response data to a client. The proxy also handles concurrent requests by forking a process for each new client request.

2 Proxy Description

The proxy server is capable of accepting multiple requests (only GET implemented), it forwards the requests to remote servers and returns the appropriate response. It has been implemented in C++. The server closes each client connection after serving each request, different from a persistent connection. The testing has been performed using the given test scripts and firefox.

3 Design choices

- I've chosen a buffer size of 4096 bytes. The read data is directly passed to the client, since the number of requests could be very large. Thus, they may cause memory errors.
- Object oriented design has been used in order to maintain readability and modularity.
- There is a limit to the maximum number of children forked by the server. As mentioned in the project description, in case there are more number of requests, we wait for a child to die. We also wait for zombie children to clean up.
- The server only handles GET requests and returns an error in other cases.

4 Testing

I've checked the proxy server by using the supplied scripts i.e. proxy_tester.py and proxy_tester_conc.py . The proxy passes both tests. I've also checked visiting some websites using firefox, the results are shown in later sections. Telnet had also been used for initial testing.

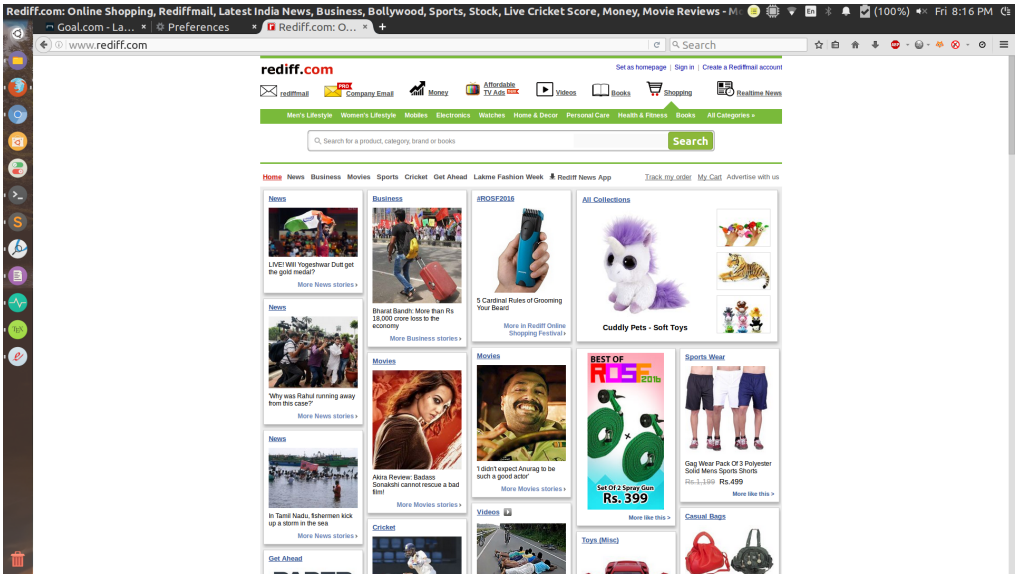
5 Summary

An HTTP proxy server has been implemented in C++ with support for multiple client requests. Only GET requests have been handled for now, in case of other requests an error message is passed.

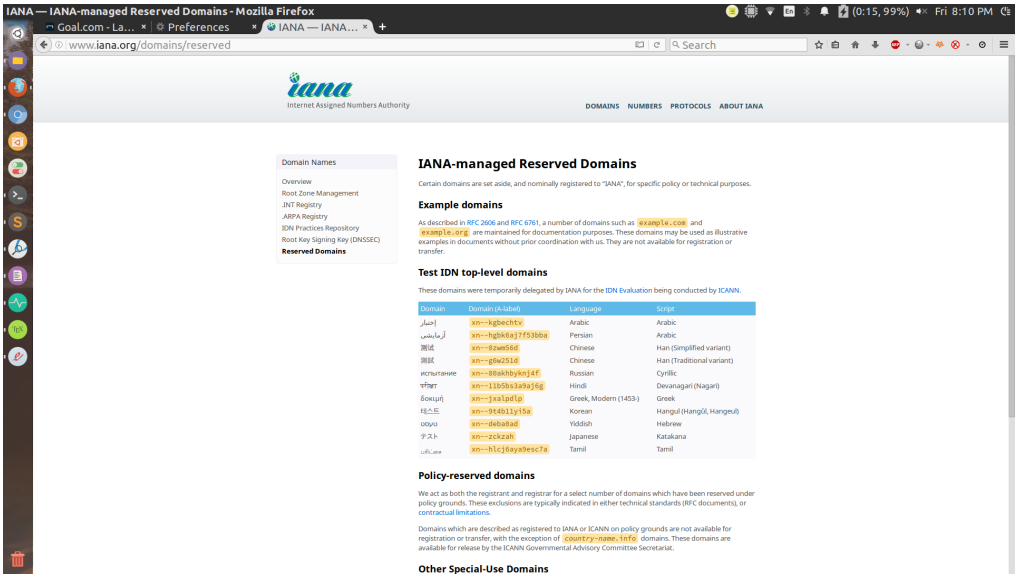
6 Screenshots

6.1 IITK

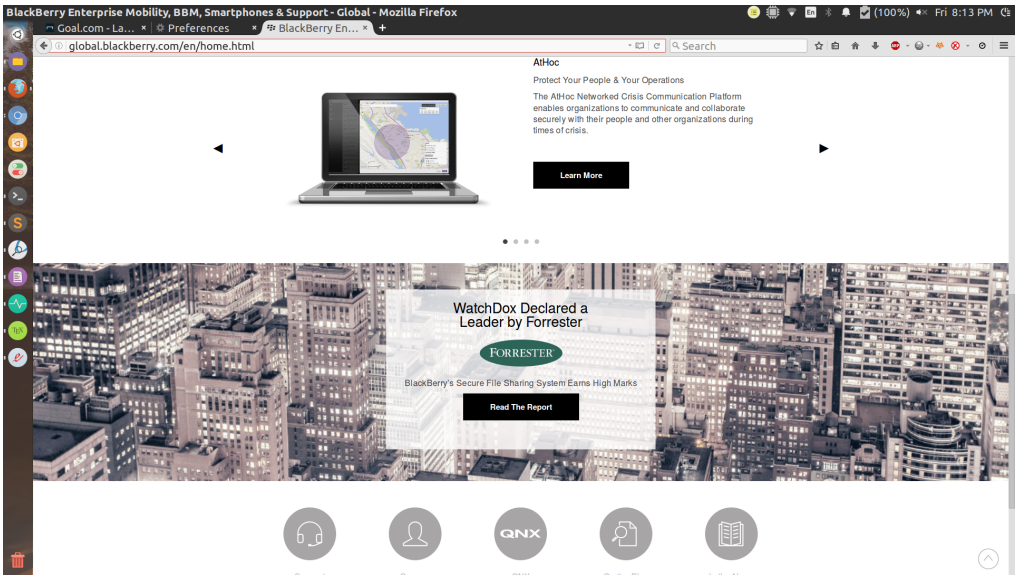
6.2 Rediff



6.3 Iana



6.4 Blackberry



7 Source Code

```
#include <cstdio>
#include <cstdlib>
#include <iostream>
#include <cstring>
#include <string>
#include <sstream>
#include <vector>
#include <algorithm>
#include <fstream>

#include <pthread.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <netdb.h>
#include <signal.h>
#include <sys/wait.h>
#include <fcntl.h>

#include "proxy_parse.h"

using namespace std;

#define MAX_CLIENTS 100
#define MAX_CHILD 20
#define BUF_SIZE 4096
#define DEBUGFLAG 0

typedef struct sockaddr_in socketAddr;
typedef struct sockaddr sockAddr;

string CRLF = "\r\n";

// Safely closes the socket with the provided socket Id

void closeSocket(int socketId) {
  close(socketId);
  shutdown(socketId, SHUT_RDWR);
}

class ProxyServer {

public:

  ~ProxyServer() {
    if (socketId >= 0) {
      closeSocket(socketId);
    }
    if (DEBUGFLAG) {
      cout << "ProxyServer closed!\n";
    }
  }

  int start(int portId);
  void run();

  int portNum;
  int socketId;
  int childNum;

  int sendRequest(ParsedRequest *reqParse);
```

```
64    string readRequest(int sockId);
65    int handleRequests(int sockId);
66    void createConnectionThread(int sockId);
67  };
68
69  // Initialize the proxy server by opening a connection on portNum
70  // It keeps listening for client connections
71
72  int ProxyServer::start(int portId) {
73
74    portNum = -1;
75    socketId = -1;
76    childNum = 0;
77
78    socketAddr serverAddr;
79    portNum = portId;
80
81    // Socket Id to which we listen
82    // Upon successful completion, socket() shall return a non-negative
          integer,
83    // the socket file descriptor. Otherwise, a value of -1 shall be
          returned.
84    socketId = socket(AF_INET, SOCK_STREAM, 0);
85
86    if (socketId < 0) {
87      printf("Could not retreive socket\n");
88      return -1;
89    }
90
91    // To reuse the same port
92    int reusePort = 1;
93    if (setsockopt(socketId, SOL_SOCKET, SO_REUSEADDR, (const char*)&
          reusePort, sizeof(reusePort)) < 0) {
94        if (DEBUGFLAG)
95          cout << "start: Could not set reuse mode\n";
96        return -1;
97    }
98
99    if (DEBUGFLAG)
100     printf("Succeeded in Socket retrieval\n");
101
102   serverAddr.sin_addr.s_addr = htonl(INADDR_ANY);
103   serverAddr.sin_family = AF_INET;
104   serverAddr.sin_port = htons(portNum);
105
106   /* Bind the address struct to the socket */
107   // socket -> bind -> listen -> accept -> doStuff
108   bind(socketId, (sockAddr*) &serverAddr, sizeof(serverAddr));
109
110   // Listen with the max number of connections
111   // On success, zero is returned. On error, -1 is returned
112   if(listen(socketId, MAX_CLIENTS) < 0) {
113     printf("Could not listen\n");
114     return -1;
115   }
116
117   return 0;
118 }
119
120 // Run the proxy server. It accepts all incoming client connections using
121 // an infinite loop, upon a connection request forks threads to handle
        them
122
123 void ProxyServer::run() {
124
125   while (1) {
```

```
126     int sockId = accept(socketId, (sockAddr*) NULL, NULL);
127     // Message denoting new connection established
128     if (DEBUGFLAG)
129       printf("run: Connection established with client\n");
130
131     if (sockId < 0)
132       printf("run: No available slots for the client\n");
133
134       // Create another thread
135       createConnectionThread(sockId);
136   }
137 }
138
139 // handleRequests: Handles the client requests which performs
140 // a complete proxy transaction by reading the client input, parsing
141 // it, then sending it to the remote server and receiving and forward
142 // to the client.
143
144 int ProxyServer::handleRequests(int sockId) {
145
146   // Get the request from the client
147   string request = readRequest(sockId);
148   if (request == "") {
149     if (DEBUGFLAG)
150       cout << "handleRequests: Unable to get request\n";
151     return -1;
152   }
153   string response = "";
154
155   ParsedRequest *reqParse = ParsedRequest_create();
156   // Parsing the request
157   if (ParsedRequest_parse(reqParse, request.c_str(), request.size()) < 0)
        {
158     if (DEBUGFLAG)
159       cout << "handleRequests: Parsing error: Parsing failed for client
            request\n";
160     response = "HTTP/1.0 500 Internal Error" + CRLF + CRLF;
161     }
162     else {
163       // Parsing successful
164         int serverSockId = sendRequest(reqParse);
165       if (serverSockId < 0) {
166       if (DEBUGFLAG)
167         cout << "handleRequests: Server unable to handle request\n";
168         response = "HTTP/1.0 500 Internal Error" + CRLF + CRLF;
169       }
170       else {
171         char buffer[BUF_SIZE + 1];
172           int bufferSize = BUF_SIZE;
173
174           memset(buffer, '\0', bufferSize);
175           int readNum = 0;
176         // Read response
177         while ((readNum = read(serverSockId, buffer, bufferSize)) > 0) {
178           write(sockId, buffer, readNum);
179           memset(buffer, '\0', bufferSize);
180         }
181         if (readNum < 0) {
182             response = "HTTP/1.0 500 Internal Error" + CRLF + CRLF;
183         if (DEBUGFLAG)
184           cout << "handleRequests: Server closed\n";
185         }
186         closeSocket(serverSockId);
187       }
188   }
189
```

```
190   if (response.size() > 0) {
191     if (DEBUGFLAG)
192       cout << "handleRequests: The response is " << response << endl;
193     if (write(sockId, response.c_str(), response.size()) <= 0) {
194       if (DEBUGFLAG)
195         cout << "handleRequests: Unable to write response\n";
196     }
197   }
198
199   ParsedRequest_destroy(reqParse);
200
201   return 0;
202 }
203
204 // createConnectionThread() creates new threads for incoming client
205 // connections. It also updates the number of active threads.
206 // As described in the project specification, it first clears all the
207 // dead children and in case the number of children are more than the
208 // maximum allowed number, it waits for the children to die.
209
210 void ProxyServer::createConnectionThread(int sockId) {
211
212   while (waitpid(-1, NULL, WNOHANG) > 0) {
213     // Wait for all dead children
214     childNum--;
215   }
216
217   if (childNum >= MAX_CHILD) {
218     // Wait for atleast one child to die
219     // Mentioned in the assignment
220     wait(NULL);
221     childNum--;
222   }
223
224   if (fork() == 0) {
225       close(socketId);
226       handleRequests(sockId);
227     closeSocket(sockId);
228       exit(0);
229   }
230   else {
231       // Update the child count
232       childNum++;
233       close(sockId);
234       // Since we have opened it for this connection, close it now
235   }
236 }
237
238 // readRequest: The function reads the request from the provided
239 // socket and returns it. In case of an error, an empty string is returned
240
241 string ProxyServer::readRequest(int sockId) {
242
243   char getMsg[BUF_SIZE + 1];
244     int bufferSize = BUF_SIZE;
245
246   string reqMsg = "";
247   memset(getMsg, '\0', bufferSize);
248
249   int numRead = 0;
250   while((numRead = read(sockId, getMsg, bufferSize)) > 0) {
251
252     if (numRead <= 0) {
253       return "";
254     }
255
```

```
256     reqMsg += string(getMsg);
257     getMsg[numRead] = '\0';
258     if (DEBUGFLAG) {
259       printf("Client reading %d\n", numRead);
260       printf("%s", getMsg);
261     }
262     // if (reqMsg.find("\r\n\r\n") != string::npos)
263     //  break;
264     if (reqMsg.find(CRLF + CRLF) != string::npos)
265       break;
266     memset(getMsg, '\0', bufferSize);
267   }
268
269   if (DEBUGFLAG) {
270     cout << "The request is------------------------------\n";
271     cout << reqMsg << endl;
272   }
273
274   return reqMsg;
275 }
276
277 // sendRequest: It takes the parsed client request, modifies it and sends
278 // it to the server. It returns the socket id for the remote server, which
279 // we then use to get the response. In case of a failed attempt it returns
280 // -1
281
282 int ProxyServer::sendRequest(ParsedRequest *reqParse) {
283
284   string serverRequest = "";
285     // First, create the request to send to the server
286     serverRequest += string(reqParse->method) + " " + string(reqParse->
          path) + \
287             " HTTP/1.0" + CRLF;
288
289   if (DEBUGFLAG) {
290     cout << "sendRequest: The request created is " << serverRequest <<
          endl;
291   }
292
293   ParsedHeader_set(reqParse, "Connection", "close");
294   ParsedHeader_set(reqParse, "Host", reqParse->host);
295   // Set connection and host headers
296
297   // Read headers
298   int headSize = ParsedHeader_headersLen(reqParse);
299   char* headBuffer = (char*) malloc((headSize + 10)*sizeof(char));
300   if (headBuffer == NULL) {
301       cout << "sendRequest: Could not allocate enough buffer space\n";
302       return -1;
303   }
304   else {
305     if (DEBUGFLAG)
306         cout << "sendRequest: Now generating the server request\n";
307   }
308
309   // Get the parsedRequest and create the server request
310   ParsedRequest_unparse_headers(reqParse, headBuffer, headSize);
311   headBuffer[headSize] = '\0';
312     serverRequest += headBuffer;
313   if (DEBUGFLAG) {
314     cout << "sendRequest: The server request created is " << serverRequest
          << endl;
315   }
316   // Free the space allocated for the headers
317   free(headBuffer);
318
```

```
319     string reqPort = "80";
320     // Default port of 80 in case port is not specified
321   if (reqParse->port != NULL) {
322     reqPort = reqParse->port;
323   }
324
325   // Find address of given server
326
327   addrinfo hints;
328   // The hints argument points to an addrinfo structure that specifies
329   // criteria for selecting the socket address structures returned in the
330   // list pointed to by res.  If hints is not NULL it points to an
331   // addrinfo structure whose ai_family, ai_socktype, and ai_protocol
332   // specify criteria that limit the set of socket addresses returned by
333   // getaddrinfo()
334
335   memset(&hints, 0, sizeof(hints));
336   hints.ai_family = AF_UNSPEC;
337   hints.ai_socktype = SOCK_STREAM;
338   int returnVal;
339
340   // Given node and service, which identify an Internet host and a
341   // service, getaddrinfo() returns one or more addrinfo structures, each
342   // of which contains an Internet address that can be specified in a call
343   // to bind(2) or connect(2).
344
345   addrinfo *serverList;
346   // Represents a linked list contain all
347
348   if ((returnVal = getaddrinfo(reqParse->host, reqPort.c_str(), &hints, &
          serverList)) < 0) {
349     if (DEBUGFLAG)
350       cout << "sendRequest: Unable to get the address structures\n";
351     return -1;
352   }
353
354   // Loop over all the results from getaddinfo and try to connect to them
355   // Referred from http://man7.org/linux/man-pages/man3/getaddrinfo.3.html
356
357   addrinfo* rp = serverList;
358   int serSockId = -1;
359
360   for (rp = serverList; rp != NULL; rp = rp->ai_next) {
361       // cout << "H" << flush;
362       if ((serSockId = socket(rp->ai_family, rp->ai_socktype, rp->
              ai_protocol)) < 0)
363           continue;
364       if (connect(serSockId, rp->ai_addr, rp->ai_addrlen) < 0)
365           continue;
366       break;
367   }
368
369   freeaddrinfo(serverList);
370
371   if (rp == NULL) {
372     // Always unsuccessful, thus close the connection
373     closeSocket(serSockId);
374       if (DEBUGFLAG)
375         cout << "sendRequest: Unable to requested server\n";
376     return -1;
377   }
378
379   // Send request to server to check whether
380   if (write(serSockId, serverRequest.c_str(), serverRequest.size()) <= 0)
          {
381       return -1;
```

```
382   }
383
384   return serSockId;
385 }
386
387 int main(int argc, char * argv[]) {
388
389   if (argc < 2) {
390     cout << "Insufficient number of command line arguments\n";
391     cout << "Enter port number too!\n";
392     return -1;
393   }
394
395   int portNum = atoi(argv[1]);
396
397     ProxyServer proxy;
398   if (proxy.start(portNum) < 0) {
399     cout << "main : Unable to start Proxy server\n";
400       return -1;
401   }
402
403   // Run the Proxy server
404   proxy.run();
405
406   if (DEBUGFLAG)
407     cout << "Proxy server closed!\n";
408
409   return 0;
410 }
```