Aalto University
School of Electrical
Engineering

S-38.3610 – Network Programming
Spring 2013

HDclient

Documentation

Riku Lääkkölä 69896S
riku.laakkola@aalto.fi

Contents

1 Usage Instructions

First, the program must be compiled by running `make`. After this, the downloading/uploading may begin! Usage as follows:

To download or upload a file:

```
./hdcli -d/-u -i <iam> -l <localfile> -r <remotefile>
-p <port/service> <host>
```

where:

- `-d/-u` is mode (*download* or *upload*, respectively)
- `<iam>` is the value for the Iam header. Pick *none* for no Iam header
- `<localfile>` is the path to the local file that is written or read in the process
- `<remotefile>` is the URI of the remote file that is downloaded or uploaded
- `<port/service>` is the port (or service) the server is listening on, eg. *80* or *http*
- `<host>` is the DNS name or IP (v4 or v6) address of the server host.

Note that in download mode, the local file that is provided as an argument is always overwritten if a connection to the server is established successfully, so be careful with that argument!

If the program gets stuck (due to loss of connection, for example), it can be interrupted with CTRL+C, which closes file descriptors and exits.

The program prints informational messages to stdout during operation. If the server responds with a non-OK response, the body of the response is printed to stdout.

2 Overview

The program is used for downloading and uploading files on a server over HTTP using the HTTP GET and PUT requests. The program can handle both IPv4 and IPv6 addresses, and when using DNS names, the user does not need to worry about the IP version, it is only visible when the program tells the user the address it is going to connect to.

Internally, the software is divided into four components which, from the top down (almost), are:

1. *mysockio*, which provides a nicer interface for the standard socket I/O system calls
2. *myhttp*, which generates and parses HTTP messages and uses the *mysockio* functions to communicate with the server host
3. *tcp_connect*, which handels DNS queries and establishing connection to the server
4. *hdcli*, which parses the user provided command line arguments, uses *tcp_connect* to establish a connection and calls the appropriate *myhttp* functions to complete the task

More specifically, the program handles GETting a file from a server in the following manner:

1. The client calls *tcp_connect* to establish a connection to the server.
2. A call to *generate_request* stores the data required for the request to a struct.
3. *send_request* is called, and the function prints the header in the correct format to a string buffer and calls *writen* to write the entire request into the socket.

4. Then, the client calls *parse_response*, which tries to read a HTTP response from the socket one character at a time using *readn* and parses each response line separately.
5. When the response header has successfully been parsed and the response was OK and Content-Length > 0, the client calls *store_response_payload*, which tries to read the content BUF_SIZE bytes at a time from the socket using *readn* and writes the buffer contents to the output file. If the response was not OK, informative error messages are printed.
6. Resources are *free*'d and file descriptors are closed.

Efficiency note: Reading the header one character at a time might be suboptimal in some senses, but because of socket buffers it should not be that bad, and it helps very much in keeping the header data separate from the payload!

The operation of the PUT functionality is fairly similar to GET. Differences are as follows:

- *generate_request* also calculates the value for Content-Length from the local file.
- *send_request* first writes the header into the socket and then calls *write_file* to transmit the payload, writing BUF_SIZE bytes at a time to the socket.
- The client excepts either a CREATED or OK response from the server in order to exit with EXIT_SUCCESS -status.

DISCLAIMER: The *readn* and *writen* functions which try to write and read exactly n bytes from or into a socket are copied almost directly from the Stevens book examples (also available in lecture materials in Optima). Also the *tcp_connect* function is copied almost in full from the lecture examples.

3 Testing and limitations

The GET functionality was tested with both the nwprog servers and many public servers on the open Internet. Image files (.jpg) work as nicely as plain text does, and no limit was observed for file size (A 17MB file was first PUT and then GET and it maintained integrity).

If the server responds with malformed HTTP, the client will print error messages and return with failure status. This will also happen if the static response header buffer is full before reaching a CRLF line (end of header), so this might be considered a limitation (header size is statically limited). As mentioned, the payload may still be huge without problems.

The software was tested (make and run) on two Ubuntu desktops and the nwprog3 server. Memory leaks were tested using Valgrind, and now there is only leaked memory (still reachable), when connection is made to an IPv6 address. This seems to be a feature of the getaddrinfo implementation (maybe...)

When network connectivity is lost, the read call blocks and the program has to be manually terminated (CTRL-C), but if connectivity is reestablished, the transfer will continue (unless the server has timed out the connection)!

4 Specifics

1. The implementation is address family independent. This is accomplished by providing *getaddrinfo* with a *hints* structure that has *ai_family* of value *AF_UNSPEC*. This was tested by running the software from my home (IPv6 connectivity) and Otakaari 5 (no IPv6) and some informative statistics printing.
2. The implementation iterates over all the addresses of the DNS query response (do-while with res switched to res->next at the end) until a connection is successful or all the address options have failed (error or timeout). If no connection is established an error message is printed and the program exits.
3. When a host is not connected, the program will try to connect and after the default timeout period it will print an error message and exit, because there is no route to the host.
4. When the host has no HTTP server running, the software will print a "Connection refused" message, because the host is not listening on the specified port.
5. When a server is suddenly terminated, it will close the socket and the client side read (or write) will return 0. The client soft-

ware keeps track of the remaining bytes and reports if the file was not downloaded completely. If the server host is suddenly disconnected from the network, the client side socket operation will block, because the socket is not closed properly. If the server regains connectivity, data transfer may be able to continue.

5 Diary

As I have been on this course before, and the lecture times are a bit difficult in my situation, I haven't attended the lectures. This means that I might not do coursework every week, and in this case I started working on the assignment about two weeks prior to the deadline.

During the first weekend I decided to redo my implementation from last year almost completely (but I did use the code as a reference as in what not to do...). I spent about 8 hours during the weekend creating the interfaces and implementing things.

I only had to use a couple of hours on monday for major debugging, and after that there was already pretty good functionality! The biggest problems seemed to be with parsing text. After realizing to use scanf, everything became a bit easier in that sense.

The socket operations in this phase are fairly straightforward, and the Stevens implementations for readn and writen proved very useful. First I thought about writing my own buffered read interface, but then I realized that the socket buffers already do the same thing, and that reading one character at a time for the header would be just fine.

The most interesting findings I had in the DNS part of the assignment. I found out that for some reason getaddrinfo leaks memory when IPv6 addresses are received, but not with IPv4. This happens because of some malloc that is not freed by freeaddrinfo.

Also the structs involved in getaddrinfo were quite confusing, and printing the address from them regardless of address family was not as straightforward as I thought. Luckily someone had implemented a nice function that does the casting nicely for inet_ntop! Source is listed in tcp_connect.c comments.