Directory Service

Introduction

Directory Services are abundant in everyday life. Anything that you might want to look up over the Internet is probably implemented as a directory service of some kind. They differ from a web search query because they always return the same information, and usually in the same order if the same query is used again.

Like the one that you are going to work on in this assignment, many of them are hierarchical in nature. For example, listing all the web pages belonging to an institute at UiO gives one tree structure for IFI, another for FYS and another for MAT. Microsoft's Active Directory is named after the principle, and can respond to queries with a lot of attributes, like the identity and authentication of the person who is requesting information.

This assignment is inspired by simple and fast directory service like the one for the Network File System that IFI's Linux machines are using to mount your home directores. This directory service is incidentally called Yellow Pages and works over UDP. It is simple, fast and scalable (but doesn't work with personal and mobile devices). It is very scalable and resilient to problems because of its use of UDP, but the lack of connections does create some challenges. A request from a client is answered by the server by sending one or more packets, and the transaction is then immediately forgotten by the server. It is entirely "stateless" and therefore capable of serving thousands or clients, which connection-oriented servers would not be able to do.

This home exam is about the implementation of two parts of such a simple, fast directory service.

1. The UDP-based communication between client and directory server,
2. And the client's reception of information from the server that has the shape of a tree (like the files and directories on a disk).

The Task

In this assignment, you are going to implement the client library of a directory service.

The scenario is stripped down to the very basic functions, and these are split into two layers.

The bottom layer enables sending and receiving of packets over UDP, where every data packet is acknowledged by the opposite side using separate ACK packets.

The upper layer makes use of this bottom layer. Here, the client side sends a single request (a single packet) to the server. The request could be very complex, but it doesn't matter for the principle. We are therefore simply sending an integer number >1000. The server (which is given in binary form) looks this request up in its database and extracts potentially hundreds of values that are organized in a tree structure. The server organizes this tree by assigning IDs to the nodes of the tree; these are assigned by the depth-first-search walk through the tree, starting with 0 at the root of the tree for your convenience. It is then sending packets to the client to respond to the request. It is first sending a packet that contains the

number of tree nodes that the client should expect, and then several additional packets, each of which contains a few tree nodes.

Your first task is to implement and test the bottom layer. This includes functions to discover the server address, create and delete a structure to keep track of your socket and the address and port of the server. It does then include functions to send and receive both data packets and acknowledgement packets.

Your second task is to implement and test the upper layer. Also here, you need functions to create and delete a structure to manage the association with the server (it may only refer to the lower layer's structure), then functions to send a request, and functions to collect the responses.

Thirdly, these responses must be understood as nodes of a tree that contain IDs, values, and information about child nodes. You will write functions that collect the tree information, and finally print it to the screen. The memory must be managed as well, of course.

The Precode

This assignment comes with a considerable amount of precode including a Makefile. By calling make, you create:

- The library **libhe.a** that contains mainly the code written by you in compiled form
- The lower-layer only test client **d1_test_client** that links libhe.a
- The upper layer test client **d2_test_client**, also linking libhe.a

The library and the programs do compile, but they do not work. All of them rely on at least one file that is incomplete and whose functions you must write, **d1_udp.c**. The program d2_test_client relies also on the file **d2_lookup.c** whose functions you must implement. The functions that are not implemented are documented in the header files **d1_udp.h** and **d2_lookup.h**, respectively. There are additionally the header files **d1_udp_mod.h** and **d2_lookup.h**, which contain data structures that you may change to make them more useful for your code.

Also servers for testing only the lower layer and for testing both the lower and upper layer come pre-built for several platforms.

This files are included in the precode.

- Two files already mentioned that you must modify:
  - d1_udp.c
  - d2_lookup.c
- Two files that you may modify:
  - d1_udp_mod.h : contains a structure to keep information about the server
  - d2_lookup_mod.h : contains one structure to keep server information and one to keep information about the tree sent by the server
- Several files that you should not modify:
  - Makefile : a very simple makefile that compiles with debug information
  - d1_udp.h : the header file with documentation for the expected behavior of functions in d1_udp.c
  - d2_lookup.h : the equivalent for d2_lookup.c
  - d1_test_client.c : a test client that connects to a d1 test server and validates that packets can be sent and received as expected
  - d2_test_client.c : a test client that connects to a d2 test server and validates the retrieval of a tree structure

Documentation

We expect that you provide a README.txt file with your code that explains how your implementation works, and which elements you have or haven't implemented.

The README.txt should be in the same directory as the code files.

Advice

Development steps

One possible way to solve this task is in the following steps:

1. We strongly advise that you implement the lower layer functions in d1_udp.c first. Go through the file d1_test_client.c and implement the required function in d1_udp.c in the order you find them. The test server is talkative. Make sure that your code has a lot of debug output as well.
2. When functionality is working, make sure that you don't have memory leaks. Use valgrind.
3. Second, go through the functions in d2_test_client.c and implement the relevant functions for communication in d2_lookup.c. Ignore the functions for create, storing, printing and deleting the tree in this round.
4. When the communication works, implement the tree functions. Make sure that the output of d2_print_tree is exactly as described in the header file.
5. Improve the documentation of your code.
6. Remove the remaining warnings that appear because of -Wall and -Wextra.
7. Finalize your README.txt

Using valgrind

To check the program for memory leaks, we recommend that you run Valgrind with the following flag:

```
valgrind \
    --leak-check=full --track-origins=yes \
    DITT_PROGRAM
```

Submission

You must submit all your code in a single TAR, TGZ or ZIP archive. No other compression formats are supported.

Include your Makefile and include all of the precode. Make sure that you have called "make clean" before creating your archive. We will compile your delivery.

If your file is called `<candidatenumber>.tar` or `<candidatenumber>.tgz`, we will use the command tar on `login.ifi.uio.no` to extract it. If your file is called `<candidatenumber>.zip`, we will use the command unzip on login.ifi.uio.no to extract it. Make sure that this works before uploading the file. It is also prudent to download and test the code after delivering it.

Your archive must contain the Makefile.

About the Evaluation

The home exam will be evaluated on the computers of the login.ifi.uio.no pool. The programs must compile and run on these computers. If there are ambiguities in the assignment text you should point them out in comments in the code and in your README.txt. Write about your choices and assumptions in your README.txt that is delivered alongside the code.

It is possible to pass the exam without solving the task entirely. However, we expect at minimum that d1_test_client runs completely and without memory leaks, and that d2_test_client sends packets to the server and receives packets from it.