Overview

Your second programming assignment is to implement a basic Real Time Streaming Protocol (RTSP) application. RTSP is typically used to stream multimedia (using an end-to-end transport protocol called the Real-Time Transport Protocol (RTP) RFC #3550), and the corresponding RFC description – RFC #7826 – is fairly lengthy. Given our 2 week window, you are not expected to implement a fully-pledged RTSP multimedia application, but rather a simplified basic RTSP application using TCP and UDP. The objectives of this assignment are:

- to become familiar with RTSP and understand real-time end-to-end data delivery;
- to familiarize yourself with Out-of-Band protocols; and
- to gain an appreciation of the advantages of UDP over TCP;

Note: RTSP employs two separate connections – one for control and one for data. This is the same behavior you observe in FTP (which uses port 20 for data and port 21 for control). Such protocols are called out-of-band protocols.

Backstory

In his latest bid to locate the long lost Unicorn, captain Haddock wants to send a deep sea probe to the depths to gather data. The probe is going to stream back data from three different sensors: oxygen-levels, temperature, and pressure. He wants to monitor these in real-time to better steer the probe to find what he is looking for.

Technical Requirements

1. For this assignment, you will need to write a client-server application to support the real-time data monitoring of the probe.
2. Your application will implement a simplified subset of the RTSP specifications (RFC 7826). Instead of RTP, you will utilize TCP and UDP as specified below:
   (a) A control connection over TCP.
(b) A data connection over UDP.

3. Your application should relay 3 different types of data – [O, T, and P] – from server → client using
   the data connection. The payload type numbers will be 0:=79, T:=84, and P:=80.

4. The format of each data type is as follows and each packet should have only 1 reading
   - 5-bits for 1 oxygen reading. Trivia: Dissolved Oxygen levels is a range 0 – 16 mg/L.
   - 8-bit for 1 temperature reading. Trivia: Coldest recorded Temperature on earth is -128.6 °F.
   - 11-bits for 1 pressure reading. Trivia: Highest sea-level pressure recorded is 1050 mbar.

5. You will have one server program and two client programs — a data-client (UDP) and a
   control-client (TCP).

6. Your data-client receives the data and should display a horizontal histogram that scrolls
   vertically as time goes on. Each line should be the value for one reading relative to the width of
   the column width, which is 20. For example, if the value for the temperature reading is 255
   (max value), then on the line corresponding to temperature, you should display/print 20 ⋆s (i.e.
   ⋆ ⋆ ⋆ ⋆ ··· ⋆). If it’s half the maximum value for that data, then it should print only 10 ⋆s. Take care
   to put the right data into the right row. The order is oxygen, temperature, pressure.

7. There should be a separate control-client to issue the following commands, which should be
   issued using a TCP connection.

   - SETUP (sec. 13.3) — Informs the server the socket address used for data communication.
     Here’s a sample SETUP message.

     ```plaintext
     C → S: SETUP rtsp://<server-hostname>/ RTSP/2.0
     CSeq: 302
     Transport: UDP;unicast;dest_addr"::4588"
     Sensor: *
     ```

     In this example, the client informs the server that it is requesting data from all three sensors
     (default behavior) accepting on UDP port 4588. The optional Sensor header can be used to
     request particular sensor data (comma separated if more than one). A sample server response
     would look similar to the following.

     ```plaintext
     S → C: RTSP/2.0 200 OK
     CSeq: 302
     Date: Tue, 23 Oct 2018 12:29:29 -0500
     Transport: UDP;unicast;dest_addr"::4588";
     src_addr"146.163.150.1:6256"
     ```

     Here, the server identifies the server side socket ID for data transfer.

   - PLAY (sec. 13.4) — Informs the server to start sending data (or resume sending in the case of
     a PAUSE). The server keeps sending data at a steady rate of every 3 seconds. until instructed
     otherwise. The received data is displayed on the data-client as a histogram. The default
     behavior is the request for all three sensor data. This behavior can be modified using the
     optional Sensor header.

     ```plaintext
     C → S: PLAY rtsp://<server-hostname> RTSP/2.0
     CSeq: 836
     Sensor: *
     ```
Over the data connection:

S → R: 79:11111;84:11111111;80:1111111111

- **PAUSE (sec. 13.6)** — Informs the server to stop sending data. Correspondingly, the data-client will pause as it is no longer receiving data.

C → S: PAUSE rtsp://<server-hostname> RTSP/2.0
CSeq: 834

S → C: RTSP/2.0 200 OK
CSeq: 834
Date: Tue, 23 Oct 2018 12:48:23 -0500

- **TEARDOWN (sec. 13.7)** — Ends the streaming session for all programs involved. This is effectively a **QUIT** command for both clients and both the data-client and the control-client should exit gracefully. Your server should handle this without issue and be ready to receive more **SETUP** commands.

C → S: TEARDOWN rtsp://<server-hostname> RTSP/2.0
CSeq: 892

S → C: RTSP/2.0 200 OK
CSeq: 892
Date: Tue, 23 Oct 2018 12:48:23 -0500

- Read relevant sections on the RFC for more information on these commands.

8. Your server should send appropriate response codes to the control-client. Use sec. 17: Status Code Definitions on RFC #7826 to guide your choices. Make sure to appropriately justify your choices in your report.

9. Your data-client should take as a command line argument of the desired UDP port to receive data that comes from the server.

   e.g. `/receiver <receiver-port>`

   **Note:** Your data-client is effectively on a listen-and-display mode. It should not be used to send any commands to the server.

10. Your control-client should take the hostname/ip address and the TCP port number which the server is listening on for control commands, and a third argument to identify the UDP port for the data connection (through your data-client).

    e.g. `/controller <server-ip> <server-port> <receiver-port>`

   It is assumed that receiver and controller are running on the same machine, but the server
will be running on a separate machine.

11. The server program should accept four command line arguments: port to use to listen to TCP connections, oxygen data file, pressure data file, and temperature data file, all of which will be binary files.

   e.g. ./server <server-port> <oxygen.bin> <temp.bin> <pressure.bin>

This will start the server listening on <server-port> for control connections, and will use the binary data from the *.bin files to stream data back to clients over data connections. Here’s an architectural figure of the overall interaction.

---

**Logistics**

- IPs and ports should in no way be hard coded. They should be given as command line arguments when launching the program
- Your server and receiver will need to be started first in the prescribed setup, and then your controller after since it must connect to both of them.
- Include a wireshark screenshot (wireshark capture files, .pcap files, are also acceptable) showing a sample interaction between your client(s) and server. Your screenshot should clearly depict the expected RTSP message headers outlined in Technical requirements item #7 above.

---

**Instructions**

- This is an individual assignment. **Do your own work.**
- **Start early!! Take backups of your code often!!**. Use of a version control software is highly recommended!
- Make sure to test your program properly before your final submission. It is highly recommended to test build and run your submission on the home server, **home.cs.siue.edu**.
- You may use any programming language of your choice out of C, C++, Java, Python or Go as these are all available on the home server. However, you **must** make sure that your code compiles and runs on a typical Linux machine. Absolutely **DO NOT** include executables with your submissions.
A **Makefile** is mandatory. Whether or not your program needs to be compiled, have it echo instructions to run the program.

- Follow a good coding standard. Use the Google C++ coding standard found here [http://goo.gl/1rC1o](http://goo.gl/1rC1o), if you don’t already follow one.
- The report part of your solution must be produced using a word processor. \LaTeX{} is highly recommended but not a requirement.
- Your final report should be in **PDF** format. No exceptions.
- Any figures, graphs, plots, etc., should also be produced using appropriate computer applications. If using \LaTeX{}, the pgfplots package is very useful for making all sorts of graphs.
- You are strongly encouraged to answer your own design and/or implementation questions based on the the RFCs, rather than defaulting to the instructor immediately. Make sure to properly document any such choices in your report and be as much descriptive as possible with proper sectional citation(s) from the relevant RFCs *(that way I will know why you did what you did)*.
- The due date of this assignment is **Tuesday, November 06, 2018 @ 12:29:59 p.m.** A dropbox will be opened for submission on Moodle.

**Deliverables**

A complete solution comprises of:

- A short report of the design and implementation of your system. The report should be **PDF** format. At minimum, your report should include the following sections:
  - Introduction: Your objective and what you hope to gain from assignment.
  - Overall design, specific design choices, and reply codes used.
  - The output of a sample run (including screenshots where applicable). See Logistics requirement above.
  - Summary and Issues encountered. What you were able to achieve from your own objectives (from the introduction) as well as project specifications. Make sure to explicitly list functionality you failed to implement (or buggy).

- A compressed tarball that contains:
  - a directory containing (only) your source code. **Do not** include executables, folders created by your programs, or any other files not specifically listed here as required.
  - A short readme file with compilation instructions.
  - A `makefile` to automate compilation. A (except python).

To create a compressed tarball of the directory `source`, use the following command:

```
tar -zcvf siue-id-pr2.tar.gz source/. e.g. tar -zcvf tgamage-pr2.tar.gz PR02/
```

Collaborating on ideas or answering each others questions is always encouraged. Most times, I find that you learn a lot from your peers. However, do not share/copy/duplicate code from others including online sources. The exercise is meant for you to learn network programming, not to test your googling abilities. Issues related to academic integrity and plagiarism have **ZERO** tolerance.

**Extra Credit**

- Enable multiple concurrent clients to independently interact with the server **[25 points]**
• Improve user friendliness of your implementation using a text-based UI such as ncurses [20 points]
  Note: If you take any of the extra credit options, you must explicitly list that in your documentation including your report, and readme.

Useful Resources

• Linux Man pages – found in all linux distributions
• Real Time Streaming Protocol