Computer Science 455
Fall 2016
Project 1 - Protocols and Encodings

September 19, 2016

1 Overview

In this project you will develop a client and server that interact over a TCP connection. The specified protocol involves use of several different kinds of serialization of protocol data so that you become familiar with the programming techniques used to deal with each of them. You may work individually or in teams of 2. If you work in a team, only one person is to turn in the work, but put both people's names in all files.

The project is due on the date shown on the course calendar.

2 Specification

The client and server are to be C programs (not C++, C#, Python, Java, or anything but C) that need to meet the following requirements:

1. The client program needs to take two command-line arguments that specify the IP address of the server and the port on which the server is listening. The server program needs to take a command-line argument that specifies the port on which it is to listen.

2. You will start the server first. Once started, it should not quit until exited using a "`Ctrl-C`" signal (from the keyboard). When the client starts it will connect to the server.

3. Once the client has connected, it can send several different commands to the server, as described in the file `project1.h`. The client's specific behavior is governed by the table `commands[]` in `project1.h` which lists the specific commands and arguments I want you to use to demonstrate the operation of your project.

4. Follow the specification in `project1.h` for the commands and overall behavior of the client and server.

5. What to turn in: a zip file named `project1.zip` containing exactly the following files (note – no stray IDE project files, ___MACOS___ directories or anything else! just these files; and **make sure your name is included in every file**):

   (a) `project1.h` – you may add things to this if you wish

   (b) `project1Client.c`

   (c) `project1Server.c`

   (d) any other `.c` or `.h` files that you write in support of this project

   (e) a shell script or batch file that will compile the client and server code resulting in executable programs `project1Client` and `project1Server` (or `project1Client.exe` and `project1Server.exe` on Windows) in the same directory

   (f) a `README.txt` file (note the file type – it *must* be a plaintext ascii file with that name). In this file provide, suitably labeled,

i. a short description of what happens when you run the command `project1Server 5`, and explain why it behaves that way

ii. The length of the log file produced by the server – recall that the server is to write everything it receives to a log file. This is the file whose length I want you to tell me. What is the correct value for this number? It's easy enough to figure out by hand if you understand the protocols used by the different commands.

iii. The result of running `sha256sum <yourlogfilename>` ; `sha256sum` runs a hashing program over the supplied file. Tell me the result of that operation. `sha256sum` is standardly installed on linux and I expect Macs as well. Downloadable binaries are available for Windows.

iv. Is there any humanly-discernable difference between the time taken for the `byteAtATimeCmd` and the `kByteAtATimeCmd` commands in the test run? Do you think there is an actual difference? What might we do to observe the difference?

v. How many hours did you spend on this project? If two people worked together report the number of hours worked separately for each person.

3 Design and implementation notes

1. The echo client and server code from Donahoo and Calvert is a good starting place. But note that you *must* rename them and get rid of any remnants of the "echo" naming as well as acknowledge their use in your comments.

2. Check the return values of all system calls, print an appropriate message and exit if an error occurs. Also, remember that `send` and especially `recv` on TCP sockets do not necessarily send and receive as much data as was requested to be sent/received. You should not have any trouble with short send operations but you absolutely must deal with the possiblity of short receives.

3. Do not do 1- or 2-byte sends or recvs. You need to make sure you don't *require* more bytes to be read than the other end is currently willing to send, but you should offer to accept as many as are available when receiving and you should send as many bytes as you can in each send call. (Of course this doesn't apply to the `byteAtATimeCmd` where you are explicitly directed to send and receive one byte at a time.

4. While you may assume that the client and server each follow the described protocol (I won't be looking for code to protect against malicious clients or servers) I nevertheless suggest that you program very defensively just to protect against your own mistakes. One particular thing to do is to make sure that after a `recv` returns 0 you never call `recv` on that socket again – all subsequent calls to `recv` will return 0 so it is very easy to get tight loops that never end.

5. The `strace` command is likely to be your friend for this project – it let's you see all the system calls that a program makes along the with arguments and results.

6. The `perror` function is likely to be your friend as well.

7. Make sure that buffers are big enough for the values that may be stored in them.

8. When the server closes its client-specific socket it resumes listening for client connections.

9. To go from an IP address expressed in dotted decimal notation as a string to a value suitable for storing in a `struct sockaddr_in` for use in a call to connect you use the library function `inet_addr()`.

10. If you start the server soon after killing it you may get an `EADDRINUSE` failure in `bind`; if that happens simply pick a different port number for the next run. The situation usually resolves itself in a minute or so.

11. Implement each of the commands, on both the client and server side, as a separate C function. Your `main` functions will be vastly clearer as a result – the `main` functions are mainly a switch statement on the command byte within a loop that iterates over the `commands` array on the client side and over the first byte of each received command on the server side.

12. Implement the client and server functions for one command at a time and get those completely and correctly working before moving on to the next command. You will learn a lot about what you have to be careful about by getting each command working completely and this should speed your implementation of the following commands.

4 project1.h

```
typedef struct {
  unsigned cmd;
  char *arg;
} command;
#define nullTerminatedCmd  (1)
#define givenLengthCmd  (2)
#define badIntCmd  (3)
#define goodIntCmd  (4)
#define byteAtATimeCmd  (5)
#define kByteAtATimeCmd  (6)
#define noMoreCommands (0)
/* This is the list of commands to be run by the client to demonstrate your program */
static command commands[] = {
  {nullTerminatedCmd, "Sent as a null-terminated string"},
  {givenLengthCmd, "Sent as an unterminated string"},
  {badIntCmd, "20160919"},
  {goodIntCmd, "20160919"},
  {byteAtATimeCmd, "500000"},
  {kByteAtATimeCmd, "500000"},
  {noMoreCommands, ""}
};
/*
 * These command names are to be prefixed to responses by the server; the array is
 * indexed by the #define'd constants above
 */
static char *commandNames[] = {
  "No More Commands",
  "Null Terminated",
  "Given Length",
  "Bad Int",
  "Good Int",
  "Byte At A Time",
  "KByte At A Time"
};
/* The maximum argument string length is 128 bytes */
/*
 * Client behavior:
 *   For noMoreCommands:
 *       don't send anything; exit the command-reading loop and
 *       close the socket.
 *   For nullTerminatedCmd:
 *       Send the string given as command.arg along with a null character as terminator
 *       (i.e. just like the string is conventionally represented in C).
 *   For givenLengthCmd:
 *       Send the string's length as a 16 bit number in network byte order followed by the
 *       characters of the string; do not include a null character.
 *   For badIntCmd:
```

```
 *       Convert command.arg to an int and send the 4 bytes without applying htonl() to the value.
 *       This is the incorrect way to go! Note that the number you get back from the server won't
 *       be what was sent.
 *   For goodIntCmd:
 *       Convert command.arg to an int and send the 4 bytes resulting from applying htonl() to it.
 *   For byteAtATimeCmd and kByteAtATimeCmd:
 *       Convert command.arg to an int; send the int (after apply htonl) and then send
 *       that many bytes of alternating 1000-byte blocks of 0 bytes and 1 bytes.
 *       ByteAtATime - use 1-byte sends and receives
 *       KByteAtATime - use 1000-byte sends and receives (except for the last)
 *   After sending the message associated with a command, recv the response
 *   that the server produces for that command and print it on stdout followed by a \n
 */
/*
 * Server behavior:
 *   The server never receives noMoreCommands.
 *
 *   For nullTerminatedCmd, givenLengthCmd, badIntCmd, goodIntCmd:
 *   reply with a 16 bit string length followed by a string containing
 *   the name of the command (i.e. commandNames[cmdByte]),
 *   a colon, a space, and the received value. Example:
 *      16bit: htons(11)
 *      11 bytes: GoodInt: 37
 *   Note that the terminator for the string sent by nullTerminatedCmd
 *   is *not* considered part of the string and should not be echoed.
 *
 *   The server behavior for BadInt and GoodInt is identical -- it applies
 *   ntohl to the received bytes.
 *
 *   For byteAtATimeCmd and kByteAtATimeCmd reply with the name of the command and the
 *   total number of recv() operations that the server performed in carrying it out,
 *   formatted as an ASCII string and counting the first recv of the data bytes as 1.
 *   Use the same format as before (16 bit network byte order integer, followed by the
 *   bytes of the string with no terminator character).
 *
 *   In addition to its other behavior, the server is to write all the bytes that
 *   it receives into a file -- it doesn't matter where as long as you can find it later.
 *   Make sure to write all and only the bytes the server receives into the file.
 *
 *   The server should close the client connection when it recv's 0 bytes, which indicates that the
 *   client has closed its end of the connection. Print on the console (printf) the total number
 *   of bytes received on the current connection. Then continue by calling accept() for a new
 *   connection.
 */
```