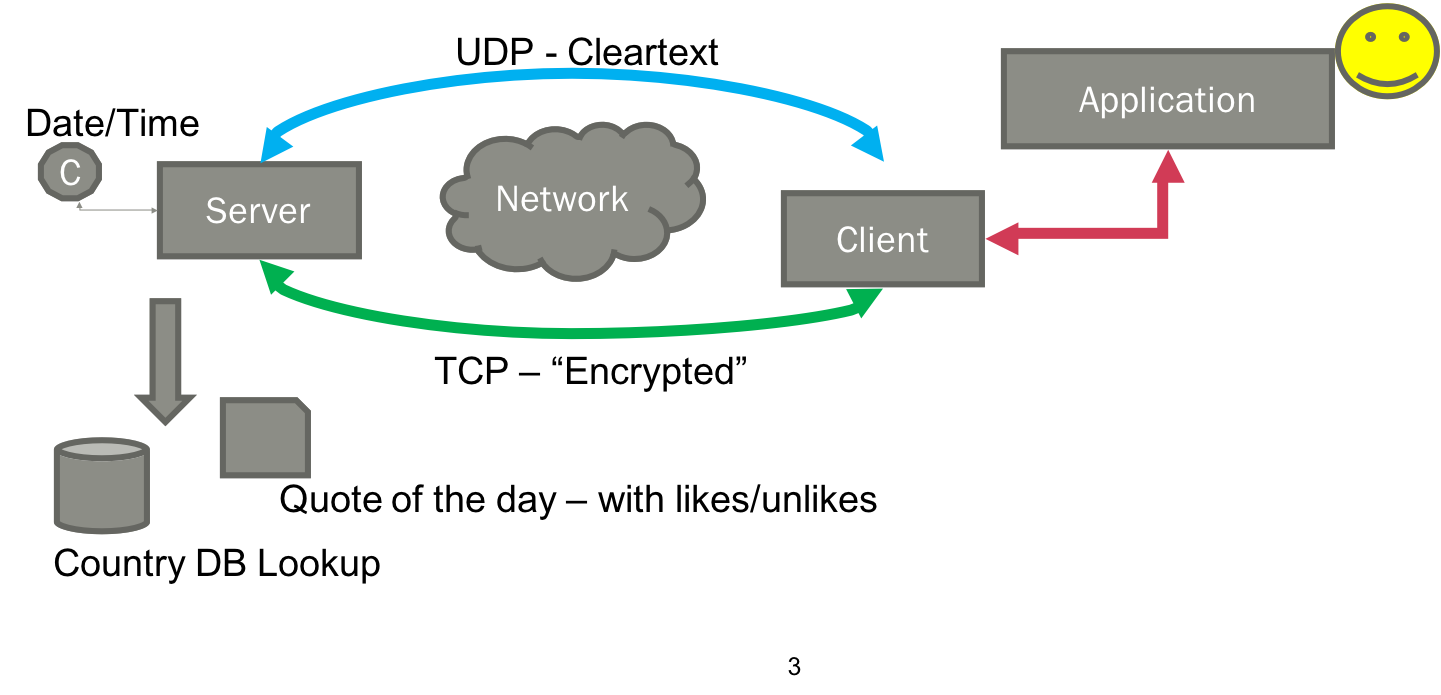
**Project  
  
Due: Sunday (see Syllabus)  
Points: 150  
  
Overview**This assignment will enable you to apply what you have learned in class and from the textbook readings.  
You will be building a simplistic 3-tier end-to-end application. The intent is for you to take some of the code that has been provided previously and to expand it to represent a more realistic network-based client-server implementation.  
  
***Requirements***Incorporate user input, enable hybrid TCP/UDP communications across the network with both client and server, and build more meaningful functionality into a server

* Server will provide time/date (think NTP)
* Server will provide simplistic text-based service with user feedback (think some web service)
* Server will perform a centralized DB query (we will simulate a full DB using a csv file)
* Deal with errors so that Server and Application don’t inadvertently crash

***Deliverables***

* **README.doc/.docx** – must contain descriptions of the programs involved, purpose of each program or helper, what’s working and what’s not working at time of upload, problems you worked through, and what you learned along the way. Also, include a description of your Application <-> Client protocol, Client <-> Server protocol (not “TCP/UDP,” but what is sent, what is expected in return, and in what order).
* **Application**, as a .c source file
* Modified **Client**, as a .c source file
* Modified **Server**, as a .c source file
* **Makefile**, only if changed
* Any **helper programs** added/modified, as .c source file(s)
* **Sufficient screenshots** of the application testing with user input and responses from “the system” to convince a reader that your system is performing as required.
* You may submit each file separately, if you wish. Or, better, place all files in a **zip file** and submit it. RAR and 7Zip are OK as well.

***Procedure*  
Step 1.**From the **Canvas | Files | Project Files** folder download the **Project Files.zip** file which contains the following:

* Project Requirements.docx (this document)
* **Makefile**
* **Application.c** (program shell with instructions – currently the application does nothing except loop forever with an empty *for* loop)
* **EchoClientTCP.c** (you can use your own from Homework #1)
* **EchoServerTCP-Forked-Num.c**
* **DieWithError.c** – same as you’ve seen previously – no changes
* **HandleClientTCP.c** – same as you’ve seen previously – no changes
* **CaesarCipher.c** – helper program for the Client/Server to “encrypt” data
* **Country\_DB.csv** – file containing country ID, Code, Name, Continent
* **Quote\_File.txt** – file containing Palindromes and Quotes (bad ones at that)
* **Test Programs folder** has some programs to help you incorporate the code you may need:
  + **testFile.c** – reads the Country\_DB.csv; shows “A way” of looking up data
  + **testTime.c** – shows several ways to pull system time and format it
  + **CaesarTest.c** – tests the Caesar cipher functionality in CaesarCipher.c

**Step 2**.  
  
Use “make” to compile the programs. The supplied Makefile compiles what is provided (look at masks). You may need to modify it for me to test your submission. Place everything needed in one directory.  
  
Make instructions

* Place all the files from the Project directory listed above in a single directory where you have permissions to compile and run make
* Prompt$>man make
* Prompt$>use apt-get install make <- only if necessary and not installed already>
* Prompt$>make –f Makefile

**Step 3**.  
  
Execute Programs

* In one window execute the server
* In another window execute the client; send a text message
* Verify that the client and server are sending and receiving what you expect them to
* Verify that the CaesarCipher.c is working correctly on your environment
* I suggest checking the testFile.c and testTime.c to verify that the code provided will work within your environment and adjust accordingly

**Step 4.**  
Develop a “protocol” (or contract) between the application and client and another between client and server. These protocols should set expectations between each system component – What is sent, what work is performed by each, and what is returned, so that errors can be managed.  
  
Practice abstraction within your design. The application only needs to “know” about the contract with the client. The server only needs to “know” about the contract with the client. The client will need to “know” about its separate contracts with the application and server.  
  
Code the changes necessary to the Application, Client (suggest renaming EchoClientTCP.c or just using useful code from it), and Server (suggest renaming EchoServerTCP-Forked-Num.c or just using useful code from it). Code any necessary/additional helper programs and incorporate into the Makefile.  
  
Compile and test the above programs that constitute your system for accurate code and error handling.  
  
**System Requirements**

* Practice abstraction, good coding practices, and follow good design principles
* **Application.c** – will be the main driver and will manage user input
  + Runs constantly waiting for user input from the main menu
  + Will manage errors with the client and the user
  + Will attempt to stay running despite errors but may need to try again or ask the user for another option/ID depending upon the error
  + Will need to manage the issue where time/date is not returned (UDP)
  + Needs to present options to the user:
    - Master Date/Time Request (then request which format)
    - Quote of the Day (then capture if the user likes/doesn’t like it for server feedback)
    - Country look-up based upon ID (then give user options for desired returned data); i.e., only Name, Code, Continent, or all.
* **Client** (based upon EchoClientTCP.c)
  + You will need to add UDP portions to your program
  + Manages the requests from the application
  + Opens communication with the server
  + Manages transfer of data with the server
    - Some communication is clear-text (date-time) (UDP)
    - Some communication is “encrypted” (quotes and Country lookup) (TCP)
      * Will decrypt all quotes/country from the server (using Caesar cipher helper to simulate a more complex encryption process) before passing to the Application/User
    - uses the correct channel for the communication and then exits when done
  + Does not run all the time – just when needed by the application
  + Will manage errors with the server
  + Will need to work with Application to determine how to handle:
    - Continent not returned (i.e., not applicable)
    - Country code not found in the file
* **Server** (based upon EchoServer-Forked-Num.c)
  + You will need to add UDP portions to program
  + Manages requests from a client that establishes communication with it
  + When requested will either return:
    - Date/time (given the format passed to it from the client)
    - Random quote from Quote\_file (will need to use srand()/rand() and some form of file seek after figuring out the file bounds)
      * Will need to mark the quote with likes/dislikes if provided
      * Avoid returning quotes that have been disliked (for testing if a quote is not liked (2) times then do not return that quote again)
      * Use a counter since the assumption is that it would normally take more than one unlike to blacklist a quote
      * But how to keep track of this since between fork writes to file could cause file concurrency issues?
    - Country information from Country\_DB.csv when requested
      * Will need to return the appropriate field requested
    - Will establish a max of:
      * 1 UDP date time socket
      * 1 TCP Quote of the day forked socket
      * 3 TCP Country\_DB.csv forked socket
    - Will need to encrypt all Quotes and Country\_DB responses (using Caesar cipher helper to simulate a more complex encryption process)
    - Will manage all file errors including not found Country Codes but will avoid exiting during operation… My suggestion is to check files during startup and exit if not found or with issues opening.
      * During operation, your code will need to manage things like hitting the end of the file without exiting. However, some obvious conditions for the server to exit would be losing access to either file during operation.

**Recommendations**

* Start early
* Build the system up and regression test along the way
* Start with the easiest functionality and add to it
* Backup working versions along the way
* Always guarantee that your code is bug-free before attempting to move forward with additional modifications/updates/extensions (“incremental development”)

# See ***Deliverables*** above for what to submit for this assignment.