EE450 Socket Programming Project, Spring 2014

Due Date: *April 27th Midnight*

(The deadline is the same for all on-campus and DEN off-campus students)

Hard deadline (Strictly enforced)

The objective of this assignment is to familiarize you with UNIX socket programming. This assignment is worth **10%** of your overall grade in this course.

It is an individual assignment and no collaborations are allowed. Any cheating will result in an automatic F in the course (not just in the assignment).

If you have any doubts/questions please feel free to contact the TAs and cc professor Zahid as usual.

The Problem:

In this project you will be simulating an ***online auction system***. All communications takes place over TCP and UDP sockets in a client-server architecture. The project has 3 major phases: Authorization, Pre-Auction, Auction. In Phases 1 and phase 2, all communications are through TCP sockets. In phase 3 however, communications take place over both TCP and UDP sockets.

Code and Input files:

You must write your programs either in C or C++ on UNIX.

1- Auction Server:

You must create one Auction server and use one of the following names for the code: "auctionserver.c", " auctionserver.cc" or "auctionserver.cpp". Also you must call the corresponding header file (if any) " auctionserver.h". You must strictly follow this naming convention.

2- Bidders :You must create 2 concurrent Bidders

   a. Either by using fork() or a similar Unix system call. In this case, you probably have only one piece of code for which you need to use one of these names: **bidder.c** or **bidder.cc** or **bidder.cpp** (all small letters). Also you must call the corresponding header file (if any) **bidder.h** (all small letters). You must follow this naming convention.
   b. Or by running 2 instances of the Bidder code. However in this case, you probably have 2 pieces of code for which you need to use one of these sets of names: (**bidder1.c, bidder2.c**) or (**bidder1.cc, bidder2.cc)** or (**bidder1.cpp, bidder2.cpp)** (all small letters). Also you must call the corresponding header file (if any) **bidder.h** (all small letters) or **bidder1.h, bidder2.h** (all small letters). You must follow this naming convention.

3- Sellers: You must create 2 concurrent Sellers

   a. Either by using fork() or a similar Unix system call. In this case, you probably have only one piece of code for which you need to use one of these names: **seller.c** or **seller.cc** or **seller.cpp**

(all small letters). Also you must call the corresponding header file (if any) **seller.h** (all small letters). You must follow this naming convention.

b. Or by running 2 instances of the Seller code. However in this case, you probably have 2 pieces of code for which you need to use one of these sets of names: (**seller1.c, seller2.c**) or (**seller1.cc, seller2.cc)** or (**seller1.cpp, seller2.cpp)** (all small letters). Also you must call the corresponding header file (if any) **seller.h** (all small letters) or **seller1.h, seller2.h** (all small letters). You must follow this naming convention.

4- Input file:

a. Phase 1:

- Input files for Auction Server: Registration.txt:

  This is the file that contains the registered valid username/password matches for each user as well as their bank account number. This is the information the login server will check each time a bidder attempt's a login to the server.

  Example file:

  Marry 123456 #bank account1
  James pass123 #bank account2

  In this example the file contains two sets of usernames and passwords (just two lines): the first with username "Marry" ,password: "123456" and one bank account "#bank account1"; and the second one with username "James", password "pass123" and bank account "#bank account2". Each bank account is a 9 digit string and should start with "4519". For example "4519 43 546" is a valid bank account.

- Input files for Bidders and Sellers: bidderPass1.txt, bidderPass2.txt, sellerPass1.txt, sellerPass2.txt

  These files contain type/username/password/bank account for a specific user. The file has just one line specifying the type/sername/password/bank account information as shown. This authorization information could be valid/ invalid. Type =1 if it is a bidder, type = 2 if it is a seller.

  Example file for a bidder: 1 Marry 123456 #bank account1

  Example file for a seller: 2 Matt pass456 #bank account2

b. Phase 2:

- Input files for Seller: itemList1.txt, itemList2.txt

  This is the file containing his name as well as names and corresponding minimal prices of auction items.

  Example file itemList1.txt:

  Matt
  Dress1 50
  Dress2 80
  Watch1 120

  Example file itemList2.txt:

  John
  Watch1 50

Watch2 100

In the example 1, the file contains 3 items: Dress1 with minimal price of 50$, Dress2 with minimal price of 80$, Watch1 with minimal set price of 120$.

c. Phase 3:

- Input files for Auction Server: broadcastList.txt
  The file will list all of available items for sale and broadcast it to all of the bidders.
- Example file broadcastList.txt
  Matt Dress1 50
  Matt Dress2 80
  John Watch1 50
  John Watch2 100
- Input files for Bidders: bidding1.txt, bidding2.txt (each authorized bidder has a bidding file)
  This is the file containing names and the corresponding prices that the bidder bid for those items. The bidder could bid for 1 or multiple items, but those items name must be present in the broadcastList.txt, or they will be considered invalid.
  Example file: bidding1.txt
  Matt Dress1 40
  John Watch1 150
  In the example, the bidding file contains 2 items. The bidder wants to buy Dress1 at 40$ and Watch1 at 150$.

A more detailed explanation of the problem:

This project is divided into 3 phases. It is not possible to proceed to one phase without completing the previous phase. In each phase you will have multiple concurrent processes that will communicate either over TCP or UDP sockets.

Phase1: Authorization:

In this phase, the Auction server and the users will open their .txt files. The Auction server will load its registration list and the users will load their respective username, password and bank account. Now the Auction server will open a TCP socket and start listening on it. Each user (seller/bidder) will try to establish a TCP connection to the server and try to login. For this purpose each user will have the TCP port of the Auction server hardcoded so that it knows where to connect to (please refer to table 1 for details). Please note that in total there will be multiple different TCP connections simultaneously, one for each user.

For the login process each user will send the following command: "Login#username password bankaccount" where everything before the pound key (#) will be interpreted as a command by the server and everything after will be the argument. The server will check to see if this username/password/bank account combination is a valid match and will reply accordingly. If it was successful the reply message should be: "Accepted#" and if it was unsuccessful:

"Rejected#". Upon acceptance the server will save the IP address of the accepted user and will bind it to its username for future reference. It will also send a reply packet to the user containing the IP address and the port number for the next two phases. If the user is rejected then it will remain idle and disregarded from that point onwards. For simplicity, use one command and its arguments per packet. After the exchange of messages is completed the LogIn server will close the TCP connection to each user.

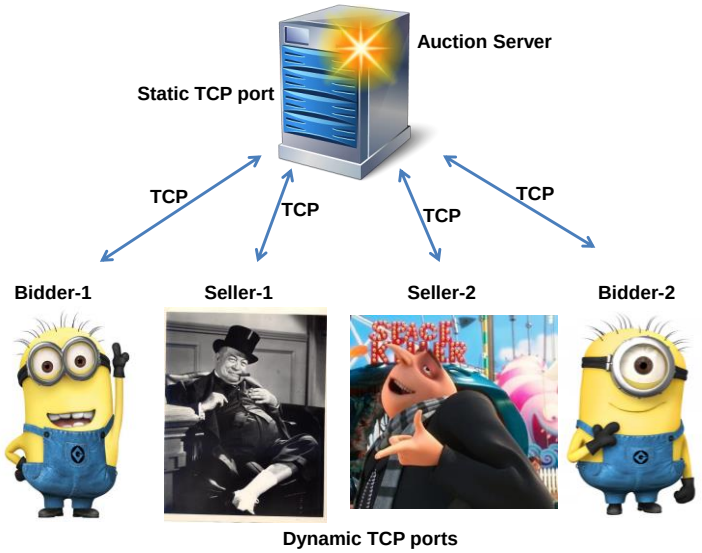

Fig. 1. Communication in Phase 1.

Phase2: PreAuction

In this phase, the accepted Sellers will send their item lists to the Auction Server. To do so, while the Auction Server is listening on a static TCP port, the Sellers open a TCP connection to the Auction Server. Once the connection is established, the Sellers will send their information of items over the TCP connection. When the Auction Server receives this information, it saves it and closes the TCP connection. Therefore, the Sellers need to know the TCP port for preAuction process of the Auction Server in advance. In other words, you must hardcode the TCP port number of the Auction Server in the Sellers code, but the TCP port numbers of the Sellers are dynamically assigned. You can hardcode this static TCP port according to Table 1. When the Auction Server fully receives the information from the sellers, it displays an appropriate message according to Tables 2 and 3.

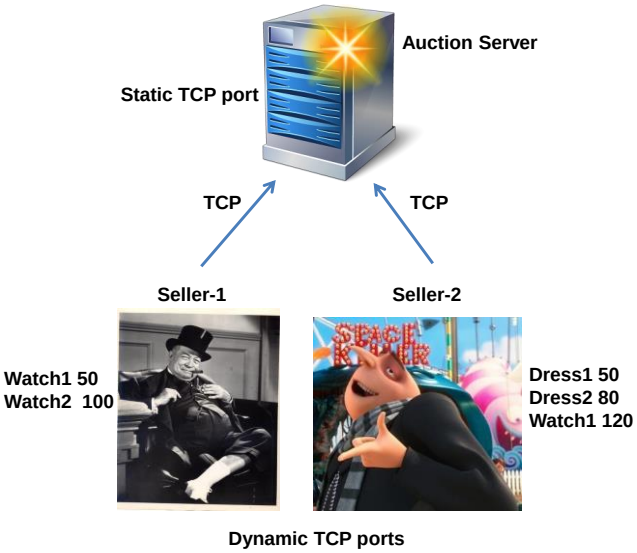

Fig. 2. Communication in Phase 2.

Phase3: Auction

The Auction Server first generates a hello type packet and broadcast it (using multiple unicast packets) to all bidders over UDP sockets. In order to do that, the Auction Server must already know the UDP ports for both two bidders; therefore you need to hardcode the UDP port numbers for all bidders in the Auction Server code files. (Table 2 provides the static UDP ports for all bidders). The packets will contain the broadcast List which maintain information of all of the items for this auction round. In other words, the Auction Server must create 2 UDP sockets, then send 2 UDP packets to the bidders. After receiving the item list, each bidder will decide its bidding decision based on bidding input text files described above in phase 3. Each bidder will send its bidding UDP packet through its UDP connection to the Auction Server. Upon receiving bidding information from both bidders, the Auction Server will perform calculation to maximize its total profit, and decide to whom the items will be sold. The item will be sold to the bidder who has higher bidding price for it. If both bidding prices for the item are smaller than its minimal price offered in Phase 2, the items will not be sold.

Example file itemList1.txt:

Matt
Dress1 50
Dress2 80
Watch1 120

Example file itemList2.txt:

John
Dress1 70
Watch1 50
Watch2 100

For example, broadcastlist.txt:

Matt Dress1 50
Matt Dress2 80
John Dress1 70

John Watch1 50

John Watch2 100

Example bidding1.txt:

Matt Dress1 40

John Watch1 70

Example bidding1.txt:

Matt Dress1 80

In the above example, the Auction Server will sell Dress1 (Matt) to Bidder 2 for 80$, and Watch1 to Bidder 1 for 70$.

In the end of this phase, the Auction Server will send its decision to both the bidders and the seller through TCP connection. For this purpose, the Auction Server will have the static TCP port of each bidders hardcoded so that it knows where to connect to (please refer to table 1 for details). Appropriate message should be displayed according to Tables 2 and 3.

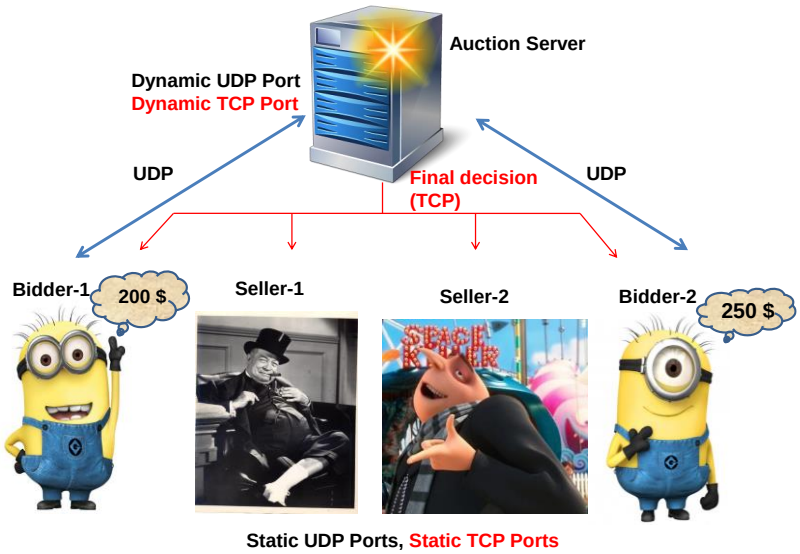

Fig. 3. Communication in Phase 3.

Table 1: A Summary of Static and Dynamic assignment of TCP and UDP Ports and IP addresses

Process	Dynamic ports	Static Ports
Auction Server	2 UDP, 1 TCP (phase 3)	1 TCP, 1100+xxx (last three digits of your ID) (phase 1) 1 TCP, 1200+xxx (last three digits of your ID) (phase 2)
Seller#1	1 TCP (phase 1) 1 TCP (phase 2)	1 TCP, 2100+xxx (last three digits of your ID) (phase 3)
Seller#2	1 TCP (phase 1) 1 TCP (phase 2)	1 TCP, 2200+xxx (last three digits of your ID) (phase 3)
Bidder#1	1 TCP (phase 1)	1 UDP, 3100 + xxx (last digits of your ID) (phase 3) 1 TCP, 4100+xxx (last three digits of your ID) (phase 3)
Bidder#2	1 TCP (phase 1)	1 UDP, 3200 + xxx (last digits of your ID) (phase 3) 1 TCP, 4200+xxx (last three digits of your ID) (phase 3)

On-screen Messages:

In order to clearly understand the flow of the project, your codes must print out the following messages on the screen as listed in the following Tables.

Table 2: Auction Server on-screen messages

Event	On-screen Message
Upon startup of phase 1	Phase 1: Auction server has TCP port number _____ and IP address _____
Upon receiving authentication request	Phase 1: Authentication request. User#: Username _____Password: _____ Bank Account: _____ User IP Addr: ____. Authorized: ____
Upon acceptance send the IP and PreAuction Port number of the Seller	Phase 1: Auction Server IP Address: _____ PreAuction Port Number: _____ sent to the <Seller#>
End of Phase 1	End of Phase 1 for Auction Server
Upon startup of phase 2	Phase 2: Auction Server IP Address: _____ PreAuction TCP Port Number: _____ .
When receiving item list from sellers	Phase 2: <Seller#> send item lists. Phase 2: (Received Item list display here)
End of Phase 2	End of Phase 2 for Auction Server
Upon startup of phase 3	Phase 3: Auction Server IP Address: _____ Auction UDP Port Number: _____ .
Broadcasting the item list	Phase 3: (Item list displayed here)
Receiving a connection	Phase 3: Auction Server received a bidding from <Bidder#> Phase 3: (Bidding information displayed here)
Sending the final auction result	Phase 3: Item _____ was sold at price _____ (display a list here)
End of Phase 3	End of Phase 3 for Auction Server.

Table 3: Sellers on-screen messages

Event	On-screen Message
Upon startup of phase 1	Phase 1: <Seller#>__ has TCP port ___ and IP address: ____
When sending login request	Phase 1: Login request. User: ____ password: _____ Bank account: _____

When receiving reply to login request	Phase 1: Login request reply: _____.
If the reply is Accept	Phase 1: Auction Server has IP Address _____and PreAuction TCP Port Number_____
End of Phase 1	End of Phase 1 for <Seller#>.
Upon startup of phase 2	Phase 2: Auction Server IP Address: _____ PreAuction Port Number: _____ .
When sending item list from sellers	Phase 2: <Seller#> send item lists. Phase 2: (Item list displayed here)
End of Phase 2	End of Phase 2 for <Seller#>.
Receiving the final auction result	Phase 3: Item _____ was sold at price _____ (display a list dedicated to that seller here)
End of Phase 3	End of Phase 3 for <Seller#>.

Table 4: Bidders on-screen messages

Event	**On-screen Message**
Upon startup of phase 1	Phase 1: <Bidder#>__ has TCP port ___ and IP address: ____
When sending login request	Phase 1: Login request. User: ____ password: _____ Bank account: _____
When receiving reply to login request	Phase 1: Login request reply: _____.
Upon startup of phase 3	Phase 3: <Bidder#>__ has UDP port ___ and IP address: ____
Receiving the item list	Phase 3: (Item list displayed here)
Sending bidding information	Phase 3: <Bidder#> (Bidding information displayed here)
Receving the final auction result	Phase 3: Item _____ was sold at price _____ (display a list dedicated to that bidder here)
End of Phase 3	End of Phase 3 for Bidder.

Assumptions:

1. The Processes are started in the order that you believe is necessary or appropriate. Please mention it in your README file.
2. You are allowed to insert delays into your code e.g. by using sleep().

3. Each packet starts with the identifier string for the sender of that packet.

4. If you need to have more code files than the ones that are mentioned here, please use meaningful names and all small letters and mention them all in your README file.

5. You are allowed to use blocks of code from Beej's socket programming tutorial (Beej's guide to network programming) in your project.

6. When you run your code, if you get the message "port already in use" or "address already in use", please first check to see if you have a zombie process (from past logins or previous runs of code that are still not terminated and hold the port busy). If you do not have such zombie processes or if you still get this message after terminating all zombie processes, try changing the static UDP or TCP port number corresponding to this error message (all port numbers below 1024 are reserved and must not be used). If you have to change the port number, please do mention it in your README file.

Requirements:

1. Do not hardcode the TCP port number for the Bidder in phase 1. It must be obtained dynamically. Refer to Table1 to see which ports are statically defined and which one is dynamically assigned. Use *getsockname()* function to retrieve the locally-bound port number wherever ports are assigned dynamically as shown below:

```
//Retrieve the locally-bound name of the specified socket and store it in the sockaddr structure
getsock_check=getsockname(TCP_Connect_Sock,(struct sockaddr *)&my_addr, (socklen_t *)&addrlen) ;
//Error checking
if (getsock_check== -1) {
perror("getsockname");
exit(1);
}
```

2. Use gethostbyname() to obtain the IP address of nunki.usc.edu or the local host however the host name must be hardcoded as nunki.usc.edu or localhost in all pieces of code.

3. You can either terminate all processes after completion of phase3 or assume that the user will terminate them at the end by pressing ctrl-C.

4. All the naming conventions and the on-screen messages must conform to the previously mentioned rules.

5. All messages sent through sockets must start with an identifier string as instructed in the project description.

6. You are not allowed to pass any parameter or value or string or character as a command-line argument. No user interaction must be required (except for when the user runs the code obviously). Every thing is either hardcoded or dynamically generated as described before.

7. All the on-screen messages must conform exactly to the project description. You should not add anymore on-screen messages. If you need to do so for the debugging purposes, you must comment out all of the extra messages before you submit your project.

8. Using fork() or similar system calls are not mandatory if you do not feel comfortable using them to create concurrent processes.

9. Please do remember to close the socket and tear down the connection once you are done using that socket.

Programming platform and environment:

1. All your codes must run on ***nunki*** (nunki.usc.edu) and only ***nunki***. It is a SunOS machine at USC. You should all have access to ***nunki,*** if you are a USC student.

2. You are not allowed to run and test your code on any other USC Sun machines. This is a policy strictly enforced by ITS and we must abide by that.

3. No MS-Windows programs will be accepted.

4. You can easily connect to nunki if you are using an on-campus network (all the user room computers have xwin already installed and even some ssh connections already configured).

5. If you are using your own computer at home or at the office, you must download, install and run xwin on your machine to be able to connect to nunki.usc.edu and here's how:

    a. Open software.usc.edu in you web browser.
    b. Log in using your username and password (the one you use to check your USC email).
    c. Select your operating system (e.g. click on windows XP) and download the latest xwin.
    d. Install it on your computer.
    e. Then check the following webpage: http://www.usc.edu/its/connect/index.html for more information as to how to connect to USC machines.

6. Please also check this website for all the info regarding “getting started” or “getting connected to USC machines in various ways” if you are new to USC: http://www.usc.edu/its/

Programming languages and compilers:

You must use only C/C++ on UNIX as well as UNIX Socket programming commands and functions. Here are the pointers for Beej's Guide to C Programming and Network Programming (socket programming):

http://www.beej.us/guide/bgnet/

(If you are new to socket programming please do study this tutorial carefully as soon as possible and before starting the project)

http://www.beej.us/guide/bgc/

Once you run xwin and open an ssh connection to nunki.usc.edu, you can use a unix text editor like emacs to type your code and then use compilers such as g++ (for C++) and gcc (for C) that are already installed on nunki to compile your code. You must use the following commands and switches to compile yourfile.c or yourfile.cpp. It will make an executable by the name of "yourfileoutput”.

```
gcc -o yourfileoutput yourfile.c -lsocket -lnsl -lresolv
g++ -o yourfileoutput yourfile.cpp -lsocket -lnsl -lresolv
```

Do NOT forget the mandatory naming conventions mentioned before!

Also inside your code you need to include these header files in addition to any other header file you think you may need:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <netdb.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <sys/wait.h>
```

Submission Rules:

1. Along with your code files, include a **README file**. In this file write
    a. Your **Full Name** as given in the class list
    b. Your **Student ID**
    c. What you have done in the assignment
    d. What your code files are and what each one of them does. (Please do not repeat the project description, just name your code files and briefly mention what they do).
    e. What the TA should do to run your programs. (Any specific order of events should be mentioned.)
    f. The format of all the messages exchanged (other than what is mentioned in the project description).
    g. Any idiosyncrasy of your project. It should say under what conditions the project fails, if any.
    h. Reused Code: Did you use code from anywhere for your project? If not, say so. If so, say what functions and where they're from. (Also identify this with a comment in the source code.)

    Submissions WITHOUT README files WILL NOT BE GRADED.

2. Compress all your files including the README file into a single "tar ball" and call it: **ee450_yourUSCusername_session#.tar.gz** (all small letters) e.g. my file name would be **ee450_sebrahim_session1.tar.gz**. Please make sure that your name matches the one in the class list. Here are the instructions:

    a. On nunki.usc.edu, go to the directory which has all your project files. Remove all executable and other unnecessary files. Only include the required source code files and the README file. Now run the following commands:

    b. **you@nunki**>> tar cvf **ee450_yourUSCusername_session#.tar** * - Now, you will find a file named "ee450_yourUSCusername_session#.tar" in the same directory.

    c. **you@nunki**>> gzip **ee450_yourUSCusername_session#.tar** – Now, you will find a file named "ee450_yourUSCusername_session#.tar.gz" in the same directory.

    d. Transfer this file from your directory on nunki.usc.edu to your local machine. You need to use an FTP program such as CoreFtp to do so. (The FTP programs are available at software.usc.edu and you can download and install them on your windows machine.)

3. Upload "ee450_yourUSCusername_session#.tar.gz" to the Digital Dropbox (available under Tools) on the DEN website. After the file is uploaded to the dropbox, you must click on the "**send"** button to actually submit it. If you do not click on "**send**", the file will not be submitted.

4. Right after submitting the project, send a one-line email to your designated TA (NOT all TAs) informing him or her that you have submitted the project to the Digital Dropbox. Please do NOT forget to email the TA or your project submission will be considered late and will automatically receive a zero.

5. You will receive a confirmation email from the TA to inform you whether your project is received successfully, so please do check your emails well before the deadline to make sure your attempt at submission is successful.

6. You must allow at least 12 hours before the deadline to submit your project and receive the confirmation email from the TA.

7. By the announced deadline all Students must have already successfully submitted their projects and received a confirmation email from the TA.

8. Please take into account all kinds of possible technical issues and do expect a huge traffic on the DEN website very close to the deadline which may render your submission or even access to DEN unsuccessful.

9. Please do not wait till the last 5 minutes to upload and submit your project because you will not have enough time to email the TA and receive a confirmation email before the deadline.

10. Sometimes the first attempt at submission does not work and the TA will respond to your email and asks you to resubmit, so you must allow enough time (12 hours at least) before the deadline to resolve all such issues.

11. **You have plenty of time to work on this project and submit it in time hence there is absolutely zero tolerance for late submissions! Do NOT assume that there will be a late submission penalty or a grace period. If you submit your project late (no matter for what reason or excuse or even technical issues), you simply receive a zero for the project.**

Grading Criteria:

Your project grade will depend on the following:

1. Correct functionality, i.e. how well your programs fulfill the requirements of the assignment, specially the communications through UDP and TCP sockets.

2. Inline comments in your code. This is important as this will help in understanding what you have done.

3. Whether your programs work as you say they would in the README file.

4. Whether your programs print out the appropriate error messages and results.

5. If your submitted codes, do not even compile, you will receive 10 out of 100 for the project.

6. If your submitted codes, compile but when executed, produce runtime errors without performing any tasks of the project, you will receive 10 out of 100.

7. If your codes compile but when executed only perform phase1 correctly, you will receive 40 out of 100.

8. If your codes compile but when executed perform only phase1 and phase 2 correctly, you will receive 80 out of 100.

9. If your code compiles and performs all tasks in all 3 phases correctly and error-free, and your README file conforms to the requirements mentioned before, you will receive 100 out of 100.

10. If you forget to include any of the code files or the README file in the project tar-ball that you submitted, you will lose 5 points for each missing file (plus you need to send the file to the TA in order for your project to be graded.)

11. If your code does not correctly assign the TCP or UDP port numbers dynamically (in any phase), you will lose 20 points.

12. You will lose 5 points for each error or a task that is not done correctly.

13. The minimum grade for an on-time submitted project is 10 out of 100.

14. There are no points for the effort or the time you spend working on the project or reading the tutorial. If you spend about 2 months on this project and it doesn't even compile, you will receive only 10 out 0f 100.

15. Using fork() or similar system calls are not mandatory however if you do use fork() or similar system files in your codes to create concurrent processes (or threads) and they function correctly you will receive 10 bonus points.

16. If you submit a makefile or a script file along with your project that helps us compile your codes more easily, you will receive 5 bonus points.

17. The maximum points that you can receive for the project with the bonus points is 100. In other words the bonus points will only improve your grade if your grade is less than 100.

18. Your code will not be altered in any ways for grading purposes and however it will be tested with different input files. Your designated TA runs your project as is, according to the project description and your README file and then check whether it works correctly or not.

Cautionary Words:

1. Start on this project early!!!

2. In view of what is a recurring complaint near the end of a project, we want to make it clear that the target platform on which the project is supposed to run is *nunki.usc.edu*. It is strongly recommended that students develop their code on nunki. In case students wish to develop their programs on their personal machines, possibly running other operating systems, they are expected to deal with technical and incompatibility issues (on their own) to ensure that the final project compiles and runs on nunki.

3. You may create zombie processes while testing your codes, please make sure you kill them every time you want to run your code. To see a list of all zombie processes even from your past logins to nunki, try this command: ps –aux | grep <your_username>

4. Identify the zombie processes and their process number and kill them by typing at the command-line:

5. Kill -9 processnumber

6. There is a cap on the number of concurrent processes that you are allowed to run on nunki. If you forget to terminate the zombie processes, they accumulate and exceed the cap and you will receive a warning email from ITS. Please make sure you terminate all such processes before you exit nunki.

7. Please do remember to terminate all zombie or background processes, otherwise they hold the assigned port numbers and sockets busy and we will not be able to run your code in our account on nunki when we grade your project.

Academic Integrity:

All students are expected to write all their code on their own.

Copying code from friends is called **plagiarism** not **collaboration** and will result in an F for the entire course. Any libraries or pieces of code that you use and you did not write, must be listed in your README file. All programs will be compared with automated tools to detect similarities; examples of code copying will get an F for the course. **IF YOU HAVE ANY QUESTIONS ABOUT WHAT IS OR ISN'T ALLOWED ABOUT PLAGIARISM, TALK TO THE TA.** "I didn't know" is not an excuse.