Comp 311 - Data Communications & Networking
Programming Project: NIM
(Client/Server)

To complete this project you will build an application named NIM. The idea behind the NIM project is to use the winsock2 library to provide a network version of the mathematical game called NIM. Your implementation of this game should be able to interact with any other team's game over the network.

Your application should behave both as a client and a server. When the program first starts, it should ask the user if they prefer to "host a game" (ie. act as a server) or "challenge some other host" (ie. act as a client).

The user interface may incorporate any language or type of project (OOP, GUI, console application, etc.); however, the code that implements the network communications must be written in Visual C++ and use the winsock2 library.

What to turn in

1) A printed listing of all source code.
2) Compressed file (.zip) containing all source code that your team wrote.
3) Any special instructions needed to build the complete application.
4) A listing of each team member's name and their responsibilities.
5) Completed "Team Membership Evaluation" forms – one from each team member (submitted on EASEL).

Specifications

The roles of "client" and "server" are only important during the game's initial negotiation phase. Once two people have agreed to play a game and the game has started, the roles of client and server are synonymous. The specifications for the negotiation phase are divided into two sections - one that describes the client's role and a 2nd section for the server's role. ***Important note*****: All UDP datagrams should consist of C-style strings (char arrays), including a terminating NULL character.**

Client Negotiations

When the client code first begins, it should obtain a name from the local user (<=80 characters). This name is used later to identify the player to others on the network.

After obtaining the user's name, the client code should create a UDP client-socket and send out a single *broadcast* datagram to the machine's subnet using port #29333. The data contents of the broadcast datagram should be the string: "`Who?`" (including a NULL terminator). Any NIM servers running in the same subnet should be monitoring port # 29333 for UDP traffic.

Once the broadcast datagram has been sent, the client code should enter a loop that waits for and receives reply datagrams that will be sent by all of the NIM servers that are currently on the local network and available to host a game. The data in each reply datagram will contain a string that corresponds to the name given to each server. The client should store the names of all such servers along with their IO addresses & port numbers and allow the local user to choose which remote person they would like to challenge. Since this loop is waiting for machines to respond to the broadcast query, the wait time should be short, say 1 or 2 seconds.

Once the user makes a selection, the client code should send a datagram to the chosen server (using the server's particular IP address and port #29333 - do *not* use a broadcast address) that contains data in the format: "`Player=`*`client_name`*".The server will interpret this as a challenge to a game. Your player's name (ie. *`client_name`*) is included in this datagram so that the other user can identify the challenger.

After sending the challenge, the client code should wait for a reply. (This time you are waiting on a person instead of a machine. Allow sufficient time to the response.) The server's user has the option to accept or refuse the challenge. The choice will be reflected in a datagram that contains either the string "`YES`" or the string "`NO`". If the response is "`NO`", your code should allow your user to challenge some other host or quit the program.

However, if the response is "`YES`", send another UDP datagram containing the string "`GREAT!`". Once this datagram has been sent, the negotiation phase is complete and your client code should begin to play the game.

If no response is received from the other user within the time allotted (or the response is something other than "`YES`" or "`NO`"), your client code should assume the answer is "`NO`"; and allow your local user to either challenge someone else or exit.

Server Negotiations

When the server code first begins, it should also obtain a name from the local user (<= 80 characters). This name is used in response to broadcast queries sent by other clients. It is used to identify the hosting player to others that may want to issue a challenge.

The server should create a UDP type server-socket and bind it to port #29333. There are two commands that may be sent to the server via this port during the negotiation phase.

If the server receives a datagram that contains the string "`Who?`", the server should reply by sending a datagram that contains the name obtained in the previous paragraph using the format, "Name=*server_name*". This reply datagram should be sent to the client's particular IP address and port # (do *not* broadcast this reply).

If the server receives a datagram that contains a string similar to "`Player=`*client_name*", your code should inform your user that they have been challenged to a game of NIM by *client_name*. Give your user a chance to accept the challenge. If they refuse the challenge, send a UDP datagram back to the client containing the string "`NO`", and then continue to listen for incoming traffic on port #29333.

If they do accept the challenge, your server code should send a UDP datagram back to the client that contains the string "`YES`", and then wait (for up to 2 seconds) for a reply UDP-datagram from the client that contains the string "`GREAT!`". The server code is now ready to play the game.

If the "`GREAT!`" datagram is not received within 2 seconds, your code should assume that the client computer isn't ready to play, and resume listening on the UDP socket (#29333) for additional incoming traffic.

Game Phase

The client / challenger always has the first move. The server / host always specifies the number of rock piles and how many rocks are in each pile ($3 \le \text{Piles} \le 9$; $1 \le \text{Rocks per Pile} \le 20$). The specific rock pile configuration should be sent to the client machine in a datagram that has the format: "$mn_1n_1n_2n_2n_3n_3n_mn_m$", where `m` is the number of piles and n_in_i is a two digit number that represents the number of rocks in the `i`th pile. If the number of rocks in the ith pile is less than 10, the corresponding 2 digit number *must* have a leading zero.

During game play, the players alternate turns (again the client/challenger always goes first). Each turn consists of allowing the user to select a move and sending the move information to the other player's computer. Optionally, *when it is your player's turn and* ***before*** *selecting a move*, your player may send a comment to the other player or they may choose to forfeit the game. The UDP datagrams used during the playing of the game are strings (including NULL terminator) and use one of the following 3 formats:

Move Format:	"`mnn`"	where `m` is a single digit ('1' thru '9') that represents a pile number, and nn are 2 digits ("`01`" through "`20`") that represent the number of "rocks" to remove from pile `m`.
Comment Format:	"`CSome comment (maximum length is 80 bytes)`"	(Note the leading '`C`' character is used as a flag and is not a part of the actual comment.)
Forfeit Format:	"`F`"	if the first character in the datagram is an '`F`', this indicates a desire to forfeit the game.

It is up to each side to check for wins and losses. If you detect the game has reached a conclusion (win, lose or forfeit), notify ***your*** local user and terminate the game. It is not necessary to send any message to the opposing player advising them of the game's conclusion.

Once a game has reached a conclusion, both the server and the client user should be then be given the chance to either host their own game or challenge some other host, and we start all over again.

Before sending any move datagram, your code should verify that your local user has selected a valid rock pile (that is, the `rockPile` number is within bounds and the associated pile still has at least 1 rock); and that the user has selected a valid number of rocks to remove (between 1 and number of rocks in the pile, inclusive). However, it is still necessary that your program verify all move datagrams that your program receives from your opponent, in case their program forgets to check for valid moves before sending same to you. The following rules apply if you happen to receive an invalid move via the network from the other player:

1) If the opposing player submits any invalid move (for example, 1st character in string is anything other than '`1`' thru '`9`', '`C`' or '`F`', or they choose a non-existent pile, or they try to remove an invalid number of rocks from a pile), they lose! The game is over. You should inform your local user that they won by default and end the game.
2) If no message is received within a reasonable amount of time (say 120 seconds), the game should be immediately terminated, with a message to your user that states that the game is over and they won by default.

Of course, if your user enters an invalid move, your code should be nice enough to alert them to the problem and give them a chance to select a different move. The rule governing invalid moves is meant to protect your code from some other team's mistakes.

Network Datagram Formats

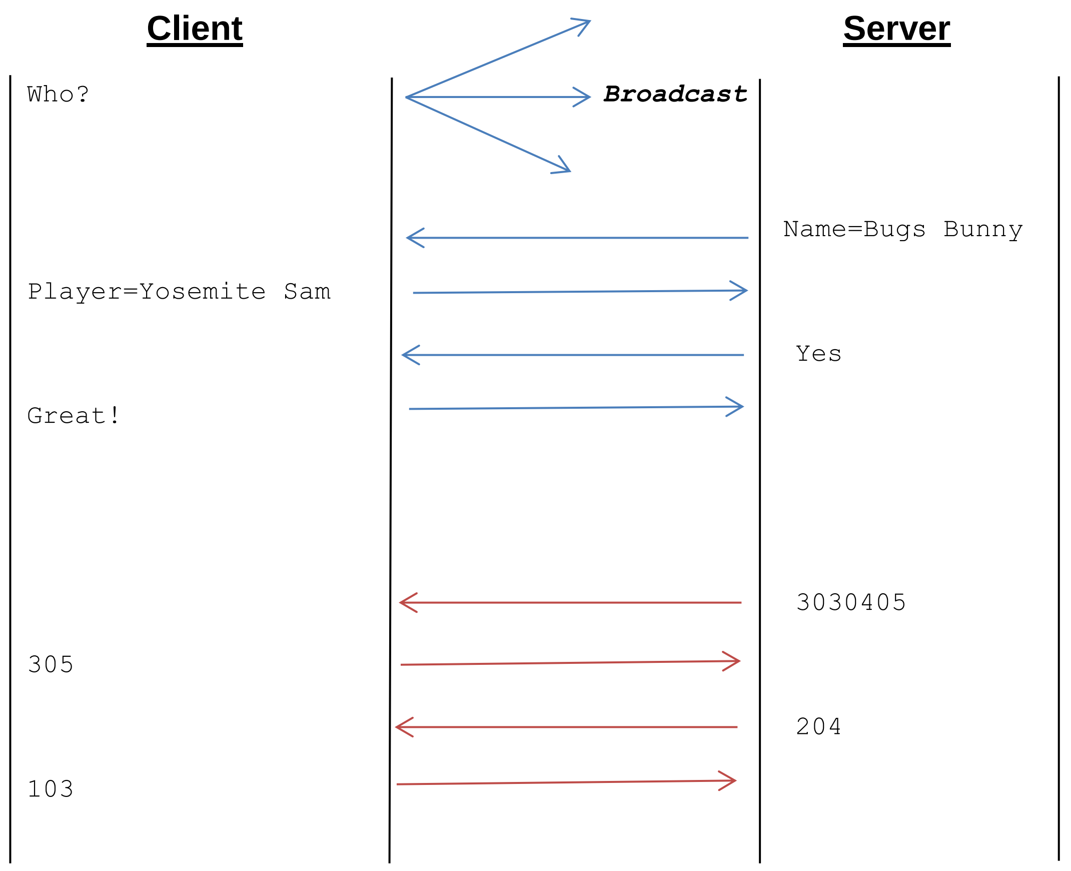
Negotiations

`Who?`	Broadcast by client searching for possible NIM servers
Name=*server_name*	Response by a NIM server upon receipt of broadcast query: `Who?`
`Player=`*client_name*	Challenge datagram sent by client to a particular NIM server
`YES`	Positive response by server to a client's challenge
`NO`	Negative response by server to a client's challenge
`GREAT!`	Client's response to server's positive response (completes 3-way handshake)

Game Play

$mn_1n_1n_2n_2n_3n_3n_4n_4...n_mn_m$	Initial rock pile configuration sent by server to client at start of game play. The first digit, `m`, represents the number of rock piles. Each of the following `m` pairs of digits represents the number of rocks in the `i`th pile. (Each pile has between `01` and `20` rocks.)
`mnn`	A "move" datagram. `m` = pile number; `nn` = number of rocks to remove
`Csome comment`	A "chat" datagram. Do NOT display the leading '`C`' character.
`F`	A "forfeit" datagram.

Sample Dialog



Yosemite Sam won the game. Both computers should now inform their own users of this fact.

Team Member Evaluations

For each team member, assign a number that you think is representative of the contribution they made to your team's project. Since these numbers are percentages, there sum should equal 100. For example, if there are 4 members on your team and you think your fellow team members each did their fair share of the work; you would assign each of them a score of 25. Notice, you are to assign a number for yourself as well as your team members.

These numbers will be kept confidential; however, you may request to know the total score that has been assigned to you. The instructor reserves the right to overrule any scores that he deems to be unreasonable.

If everyone on your team thought that you did your fair share and, consequently, gave you a score of 25 on each of their evaluation sheets, your total score would be 100%. Your project grade would then be 100% of whatever grade was assigned to the project as a whole. For example, if the project receives a grade of 85 and your participation percentage is 100%, you will receive an 85 as your NIM project score.

Example:

Team Members	Participation Percentage
Yosemite Sam	25
Bugs Bunny	20
SpongeBob	2
Spiderman	53
Total	**100%**

Team Member	Participation Percentage
Total	**100%**

____
Your Name