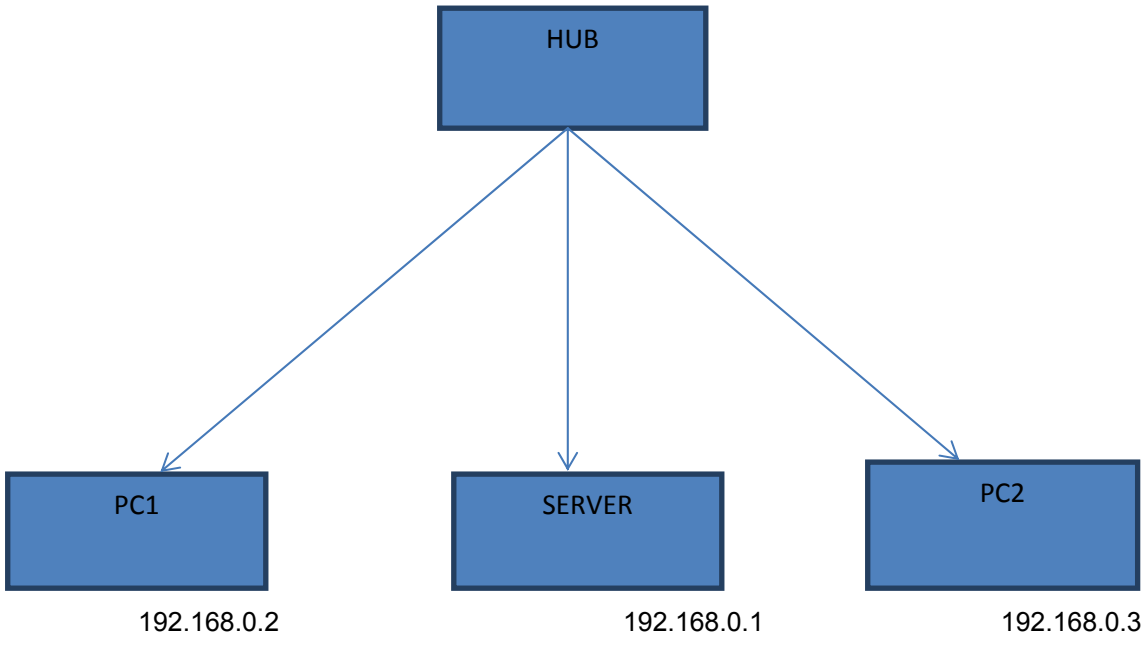
Autumn Semester 2012
Lab-4

Developing Quiz Buzzer Application using TCP/IP Socket Programming.

Resources:

1.	PCs	2Nos.
2.	NICs	2Nos.
3.	Hub	1No.
4.	Data cables	2No.

Configuration:



Exercise: Implement the Quiz Application (as explained in the lecture and outlined below)

Explanation of Quiz Buzzer application:

1. Each client has to register to server before participating in Quiz. The server will ask username from each user and generate temporary id for each user.

2. After Registration process clients, who are successfully connected to server will get Question from server.

3. The client will reply with answer.

4. Server will receive answer from different clients with time stamps, and it will calculate time difference of each client which is called ∆t.
   Define such as:

   ```
   ∆t = (Time Question sent - Time answer received) - RTT

        Where RTT is Round Trip Time
   ```

5. Server will select client, whose ∆t is minimum to all and reply with whatever score client will gain remains will not gain any score.

6. After sending Question server will wait Answer for a particular time periods called (T). If client did not reply within 'T' time period Server will skip that Question and goes to next Question.

Submission:

1. Submit the code (even if incomplete) at the end of the lab session – single submission per group through the course website
2. Submit the completed code through the course website before the deadline, as announced by the TAs

---

Function we will use for developing Quiz Buzzer Application

Usage of ctime():

```
char * ctime ( const time_t * ptr_time );
```

Parameters: Pointer ptr_time to a time_t object that contains a calendar time.

Return Value:

The function returns a C string containing the date and time information. The string is followed by a new-line character ('\n') Converts the time_t object pointed by timer to a C string containing a human-readable version of the corresponding local time and date.

The functions ctime and as ctime share the array which holds this string. If either one of these functions is called, the content of the array is overwritten.
The string that is returned will have the following format: **Www Mmm dd hh:mm:ss yyyy**

Www = which day of the week.

Mmm = month in letters.

dd = the day of the month.

hh:mm:ss = the time in hour, minutes, seconds.

yyyy = the year.

Source code example of ctime():

```
#include<stdio.h>
#include<time.h>
 int main (){
              time_t time_raw_format;
             time(&time_raw_format);
             printf("The current local time: %s", ctime(&time_raw_format));
             return (0);
}
```

time()

#include <time.h>
time_t time(time_t *time);

Description:
The function time() returns the current time, or -1 if there is an error. If the argument time is given, then the current time is stored in time.

difftime()

double difftime (time_t time2, time_t time1);
Return difference between two times
Calculates the difference in seconds between *time1* and *time2*.

Parameters
time2
time_t object representing the latter of the two times.
time1
time_t object representing the earlier of the two times.

Return value
The difference in seconds (*time2-time1*) as a floating point double.

Signal handling

alarm()
The alarm function allows us to set a timer that will expire at a specified time in the future. When the timer expires, the SIGALRM signal is generated. If we ignore or don't catch this signal, its default action is to terminate the process.

#include <unistd.h>
unsigned int alarm(unsigned int *seconds*);

Description

The alarm() function shall cause the system to generate a SIGALRM signal for the process after the number of realtime seconds specified by *seconds* have elapsed. Processor scheduling delays may prevent the process from handling the signal as soon as it is generated.

If *seconds* is 0, a pending alarm request, if any, is canceled.

Alarm requests are not stacked; only one SIGALRM generation can be scheduled in this manner. If the SIGALRM signal has not yet been generated, the call shall result in rescheduling the time at which the SIGALRM signal is generated.

Return Value

If there is a previous alarm() request with time remaining, alarm() shall return a non-zero value that is the number of seconds until the previous request would have generated a SIGALRM signal. Otherwise, alarm() shall return 0.

Select()

int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, const struct timeval *timeout);

Description

Select checks to see if any socket are ready for reading (**readfds**), writing (**writefds**), or have an exceptional condition pending (**exceptfds**). timeout specifies the maximum time for select to complete.
nfds specifies the maximum number of sockets to check.
If **timeout** is a null pointer, **select** blocks indefinitely. **timeout** points to a **timeval** structure. A timeout value of **0** causes **select** to return immediately.
This behavior is useful in applications that periodically poll their sockets.
fd_set is a type defined in the **<sys/types.h>** header file. **fd_set** defines a set of file descriptors on which select operator. Passing **NULL** for any of the three **fd_set** arguments specifies that **select** should not monitor that condition.
On return, each of the input sets is modified to indicate the subset of descriptions that are ready. These may be found by using the FD_ISSET macro.

Return value

If select succeeds, it returns the number of ready socket descriptors. select returns a 0 if the time limit expires before any sockets are selected. If there is an error, select returns -1.

Theory

TCP:

TCP is a transport layer protocol used by applications that require guaranteed delivery. It is a sliding window protocol that provides handling for both timeouts and retransmissions.

TCP establishes a full duplex virtual connection between two endpoints. Each endpoint is defined by an IP address and a TCP port number. The operation of TCP is implemented as a finite state machine.

The byte stream is transferred in segments. The window size determines the number of bytes of data that can be sent before an acknowledgement from the receiver is necessary.

UDP:

The User Datagram Protocol offers only a minimal transport service -- non-guaranteed datagram delivery -- and gives applications direct access to the datagram service of the IP layer. UDP is used by applications that do not require the level of service of TCP or that wish to use communications services (e.g., multicast or broadcast delivery) not available from TCP.

UDP is almost a null protocol; the only services it provides over IP are checksumming of data and multiplexing by port number. Therefore, an application program running over UDP must deal directly with end-to-end communication problems that a connection-oriented protocol would have handled -- e.g., retransmission for reliable delivery, packetization and reassembly, flow control, congestion avoidance, etc., when these are required. The fairly complex coupling between IP and TCP will be mirrored in the coupling between UDP and many applications using UDP.

Socket:

In computer networking, an Internet socket or network socket is an endpoint of a bidirectional inter process communication flow across an internet protocol based computer network such as the internet.

The term internet socket is also used as a name for an application programming interface (API) for the TCP/IP protocol stack, usually provided by the operating system. Internet sockets constitute a
mechanism for delivering incoming data packets to the appropriate application process or thread, based on a combination of local and remote IP address and port numbers. Each socket is mapped by the operating system to a communicating application process or thread.

A socket address is the combination of an IP address (the location of the computer) and a port (which is mapped to the application program process) into a single identity, much like one end of a telephone connection is the combination of a phone number and a particular extension.

Two types of internet sockets:

1). Stream sockets: Stream sockets are reliable two-way connected communication streams. If you output two items into the socket in the order “1, 2”, they will arrive in the order “1, 2” at the opposite end. They will also be error-free.

2). Datagram sockets: Datagram sockets are sometimes called “connectionless sockets”.

Checksum:

A **checksum** or **hash sum** is a fixed-size computed from an arbitrary block of digital data for the purpose of detecting accidental errors that may have been introduced during its transmission or storage. The integrity of the data can be checked at any later time by re-computing the checksum and comparing it with the stored one. If the checksums match, the data were almost certainly not altered (either intentionally or unintentionally).

CRC:

A cyclic redundancy check (CRC) is an error-detecting code designed to detect accidental changes to raw computer data, and is commonly used in digital networks and storage devices such as hard disk drives.

Socket API (Some relevant interfaces) For Connection Oriented (TCP-based)

getaddrinfo();

```
#include
<sys/types.h>
#include
<sys/socket.h>
#include
<netdb.h>

int getaddrinfo(const char *node, // e.g.
                "www.example.com" or IP const char
                *service, // e.g. "http" or port number const
                struct addrinfo *hints,
                struct addrinfo **res);
```

socket():

```
#include
<sys/types.h>
#include
<sys/socket.h>

int socket(int domain, int type, int protocol);
```

bind():

```
#include
<sys/types.h>
#include
<sys/socket.h>

int bind(int sockfd, struct sockaddr *my_addr, int addrlen);
```

connect():

```
#include
<sys/types.h>
#include
<sys/socket.h>
```

int connect(int sockfd, struct sockaddr *serv_addr, int addrlen);

listen():

int listen(int sockfd, int backlog);

accept():

#include
<sys/types.h>
#include
<sys/socket.h>

int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);

send()

#include <sys/socket.h>

ssize_t send(int socket, const void *buffer, size_t length, int flags);

The send() function shall initiate transmission of a message from the specified socket to its peer. The send() function shall send a message only when the socket is connected (including when the peer of a connectionless socket has been set via connect()).

recv()

#include <sys/socket.h>

ssize_t recv(int socket, void *buffer, size_t length, int flags);

The recv() function shall receive a message from a connection-mode or connectionless-mode socket. It is normally used with connected sockets because it does not permit the application to retrieve the source address of received data.

Socket API for Connectionless communication (UDP-based)

sendto()

#include <sys/socket.h>

ssize_t sendto(int socket, const void *message, size_t length, int flags,
const struct sockaddr *dest_addr,
socklen_t dest_len);

The sendto() function shall send a message through a connection-mode or connectionless-mode socket. If the socket is connectionless-mode, the message shall be sent to the address specified by dest_addr. If the socket is connection-mode, dest_addr shall be ignored.

recvfrom()

#include <sys/socket.h>

ssize_t recvfrom(int socket, void *buffer, size_t length, int flags, struct
sockaddr *address,

socklen_t *address_len);

The recvfrom() function receives a message from a connection-mode or connectionless-mode socket. It is normally used with connectionless-mode sockets because it permits the application to retrieve the source address of received data.
