Instructions:

- Complete the project in groups of 2-3.
- The grade for this project is weighted as follows:
  - 50% Implementation
  - 20% Plots from experiments
  - 30% Write-up
  - 10% Bonus (see below)

Due Date: November 9, 1999 at the beginning of class.

Objective: Implementation and evaluation of the sliding window flow control scheme described below.

Problem Description:

1. There is one Sender process and one Receiver process. Sender and Receiver run on different machines.

2. Sender and Receiver use a version of the sliding window flow control scheme described in the pseudo-code shown below:

   **Sender:**
   ```
   While (TRUE) {
     If (Sender is allowed to send) {
       Send one packet;
       Adjust Window;
     }
     Print current time and current size of window;
     Sleep for \( R \) seconds;
     If (ACKs have arrived) {
       Receive all ACKs;
       Adjust Window;
     }
   }
   ```

   **Receiver:**
   ```
   While (TRUE) {
     If (there are unACKnowledged packets) {
       Send an ACK for one packet;
       Adjust Window;
     }
     Print current time and current size of window;
     Sleep for \( R \) seconds;
     If (packets have arrived) {
       Receive all packets;
       Adjust Window;
     }
   }
   ```

3. The initial window size is identical for sender and receiver and set to \( W = 8 \).

4. The values \( R \) are random values that are calculated as follows:

   \[
   R = \lceil Rand() \cdot 5 \rceil \cdot Speed-up \text{ seconds}
   \]

   where \( \lceil x \rceil \) is a function that gives the smallest integer at least as large as \( x \); \( Rand() \) is a randomly generated number such that \( 0 \leq Rand() \leq 1 \); and \( Speed-up \) is a variable set to either 1 or 2. (Note that \( Speed-up \) can be different for Sender and Receiver.)
5. All packets are fixed-length with a size of 100 bytes each.

6. Assume that all packets are delivered without errors and in-sequence.

Your Task:

1. Implement the above version of the sliding window protocol using stream sockets. The implementation involves:

   (a) 

   C/C++ code that implements the Sender process. Usage for the Sender process is as follows: 
   Sender <dest_host> <dest_port> <Speed-up> <max_time>, where:
   * dest_host is the name of the host with the Receiver process,
   * dest_port is the well-known port at which the Receiver accepts connections,
   * Speed-up is the speedup factor described above,
   * max_time is the maximum running time in seconds.

   (b) 

   C/C++ code that implements the Receiver process. Usage for the Receiver process is as follows: 
   Receiver <port> <Speed-up> <max_time>, where:
   * port is the port number at which the Receiver accepts connections,
   * Speed-up and max_time are as described in 1(a) above.

2. Illustrate the operation of the sliding window protocol in a number of plots. The plots should show the transient fluctuations of the window size at both the Sender and Receiver. (Figure 1 sketches how such a plot could look.) At the minimum, show plots for the following cases:
   
   - Sender’s Speed-up = 1; Receiver’s Speed-up = 1.
   - Sender’s Speed-up = 2; Receiver’s Speed-up = 1.
   - Sender’s Speed-up = 1; Receiver’s Speed-up = 2.

   The running time for these experiments should be at least 5 minutes. You are encouraged to do additional experiments or show additional plots.

<table>
<thead>
<tr>
<th>W(R)</th>
<th>6</th>
<th>4</th>
<th>2</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>W(S)</td>
<td>6</td>
<td>4</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 1: Sender window size $W(S)$ and receiver window size $W(R)$ as a function of time.

3. Deliver a write-up which includes:

   - your implementation approach,
- a description of your test strategy that shows the correctness of your implementation,
- a list of known bugs,
- a summary,
- lessons learned.

The write-up should not exceed 3 pages (excluding plots).

4. You can earn a bonus of up to 10% if your program produces output interactively with a graphical user-interface.

Hints:

1. Use your code for stream sockets from Assignment #1 to implement both connection-establishment and data transfer.

2. Since all packets are assumed to be of size 100 bytes, the Receiver must read data in 100-byte blocks. The following code at the receiver will implement such a mechanism:

   ```c
   for (int inchars = 0; inchars < 100; inchars += n) {
       n = read(sock, &buff[inchars]), 100 - inchars);
   }
   ```

3. The library function `rand` is a pseudorandom number generator that is standard on most UNIX platforms (and is usually defined in `<stdlib.h>`).

4. Many of your "read" system calls should be non-blocking. The "select" system call provides an easy way to make these calls non-blocking.

   - The `select` system call provides non-blocking I/O by permitting a process to test whether any descriptor in a set of file descriptors is ready for I/O. A bitmask is used to specify a set of file descriptors, and `select` tests the file descriptors associated with the bitmask. These bitmasks are of type `fd_set`.

   - The prototype for the `select` call is as follows:

     ```c
     int select(int maxfdp, fd_set *readfds, fd_set *writefds,
                 fd_set *exceptfds, struct timeval *timeout);
     ```

     where:

     - `maxfdp` is the number of file descriptors to be tested. All file descriptors `f` that satisfy `0 ≤ f < maxfdp` are tested.
     - `readfds` is the set of file descriptors tested for input.
     - `writefds` is the set of file descriptors tested for output.
     - `exceptfds` is the set of file descriptors tested for exceptions.
     - `timeout` is a value that specifies a maximum time to wait for I/O readiness of a file descriptor.
     - `select` returns the number of file descriptors that are ready for I/O, and the sets are manipulated such that only the "ready" file descriptors remain in the sets.
• The following two functions are among several used to manipulate the bitmasks associated with a set fdset of type fd_set:

   FD_ZERO(fd_set *fdset) /* clear all bits in bitmask fdset */
   FD_SET(int fd, fd_set *fdset) /* set the bit associated with fd in fdset */

• The following sample code shows how the select command is used to ensure that data is available before reading from a socket “sock”:

/* Set variables to check if there is data available at the socket sock */
FD_ZERO(&fdvar);    /* clear all bits of fdvar */
FD_SET(sock, &fdvar);   /* turn on bit for fd sock */

/* The following call returns a nonzero value if the socket */
/* contains data for reading */
if (select(sock+1, &fdvar, (fd_set *)0, (fd_set *)0, (struct timeval *)0))
   /* Read from the socket */

• The header file <sys/time.h> must be included since it contains the definition of the structure struct timeval.