Internet Protocols: Lab Assignment 2: TCP Congestion Control

Due (ON-CAMPUS/NON-DELAYED): MARCH 1st, 2001 (THURSDAY) at the beginning of class.
Due (DELAYED): MARCH 12th, 2001 (MONDAY)

1 Goals

- To understand the design of the Transmission Control Protocol (TCP).
- To implement the slow start and congestion avoidance algorithms of TCP.

2 Layered architecture

For this lab, assume that each node in the network has four layers: *physical layer*, *datalink layer (DLC) Transport layer (TCP)* and *application layer*. Nodes in the network are connected to one another via *links*. Each layer in a node can be thought of as an abstract *entity* that performs certain functions. Similarly, links are also entities that have some functionality. Figure 1 outlines the four layers in a node connected by a link entity. **In this lab, you will learn how these entities communicate with one another, and will implement the congestion control algorithms at the TCP layer**

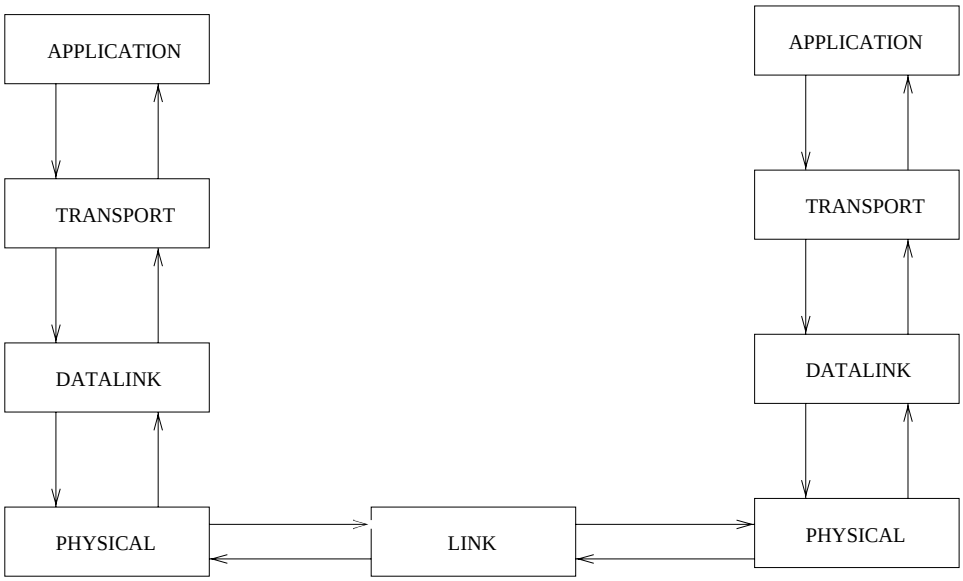

Figure 1: Layered Architecture

3 Protocol Data Units

Each layer communicates through Protocol Data Units (PDU). The application layer PDU is called **A_PDU**, the transport layer PDU is called **T_PDU**, the DLC PDU is called **D_PDU**, and the physical layer PDU is called **PH_PDU**. The pdu formats are defined below. These definitions, together with some others are provided to you in the file `pdu.h` .

```
typedef struct {
```

```
        int  snode;                    /* source node address      */
        int  dnode;                    /* destination node address */
        char data[DATASIZE];           /* data                     */
} A_PDU_TYPE;

typedef struct {

  int snode;          /* source address */
  int dnode;          /* destination address */
  int pk_seq;         /* sequence number in bytes */
  int pk_ack;         /* ack number in bytes */
  char pk_flags;      /* flags : not used */
  int  pk_win;        /* window advertisement : not used */
  int  pk_checksum;   /* 0 => no error, non-zero => error,
                         DLC layer may set this field if it
                         receives a corrupted packet*/
  int pk_urp;         /* urgent pointer : Not used */
  int pk_len;         /* length of data in bytes */
} TCP_HEADER_TYPE;
#define TCP_HEADER_SIZE sizeof(TCP_HEADER_TYPE)

typedef struct {

  TCP_HEADER_TYPE tcp_hdr;
  A_PDU_TYPE      a_pdu;

} T_PDU_TYPE;
#define T_PDU_SIZE sizeof(T_PDU_TYPE)


/* data unit between datalink layer and physical layer */
typedef struct {
     int               curr_node;   /* address of this node */
     int               next_node;   /* address of next node */

     T_PDU_TYPE        t_pdu;
     enum boolean      error;       /* YES if the packet is corrupted;
       otherwise NO.            */
} D_PDU_TYPE;
#define D_PDU_SIZE sizeof(D_PDU_TYPE)


typedef struct {
        int               type;
        D_PDU_TYPE        d_pdu;       /* dlc  pdu          */
} PH_PDU_TYPE;

typedef struct {
        union {                                /* structure containing a_pdu */
                A_PDU_TYPE        a_pdu;       /* d_pdu,t_pdu or ph_pdu as a union */
                T_PDU_TYPE        t_pdu;
                D_PDU_TYPE        d_pdu;
                PH_PDU_TYPE       ph_pdu;
        } u;
        int type;  /* One of: TYPE_IS_A_PDU, TYPE_IS_D_PDU, TYPE_IS_PH_PDU */
int color; /* Internal field to determine the kind of packet */
} PDU_TYPE;
```

Communication is performed as follows: The application layer sends an `a_pdu` to the TCP layer. The TCP layer receives this `a_pdu` and encapsulates it within a `t_pdu`. It then performs its functions on the `t_pdu` and sends the `t_pdu` to the datalink layer. In the same manner, the datalink layer receives the `t_pdu`, encapsulates it within a `d_pdu` and sends it to the physical layer which in turn sends it to the link entity. The link entity receives a `ph_pdu` from one physical layer and delivers it to the physical layer at the other end. When a physical layer receives a `ph_pdu` from the link, it extracts the `d_pdu` from it and sends it to the dlc layer which in turn sends the `t_pdu` within the `d_pdu` to the transport layer. The transport layer extracts the `a_pdu` and sends it to the application layer.

4 Service Primitives

Inter layer communication takes place by means of *service primitives*. The TCP layer uses functions called `send_pdu_to_datalink()` and `send_pdu_to_application()` to send pdu to the datalink and application layers respectively.

In this lab, you will design a simplified version if the TCP layer and implement the congestion control algorithms of this layer.

5 TCP protocol description

5.1 The TCP layer

For each active connection, TCP maintains a data structure that it uses to maintain the state of the connection. In this lab, this structure is called `TCP_LAYER_ENTITY_TYPE`. The structure has several variables of which you will need to know about the following: The structure is declared in the file `tcp_layer.h`. These variables are further explained in the later sections.

```
int  tcp_my_addr;      /* My address -- In this lab, addresses are simple integers */
int  tcp_peer_addr;    /* Address of peer TCP */

int  tcp_snd_wnd;      /* Size of receiver's offered window (bytes) */
int  tcp_snd_cwnd;     /* Size of sender's congestion window (bytes) */
int  tcp_snd_ssthresh; /* Slow start threshold (bytes) */

int  tcp_snd_max;      /* 1 + Max sequence number sent so far (bytes) */
int  tcp_snd_nxt;      /* Sequence number of next segment to be sent (bytes) */
int  tcp_snd_una;      /* Sequence number of first unacknowledged segment (bytes)*/
int  tcp_rcv_nxt;      /* Sequence number of next expected segment (bytes)*/

int  tcp_ack_flag;     /* 1 if the segment contains an ack, 0 otherwise */
int  tcp_mss;          /* Maximum segment size */

queue  *tcp_send_queue;     /* Queue of application's packets to be sent to dlc */
                            /* This is accessible by the following functions */
                            /* Insert_pdu_into_send_queue(); */
                            /* Remove_pdu_from_send_queue(); */
                            /* Calculate_max_send_queue_offset(); */
int    tcp_send_queue_size; /* Size in bytes of the above queue */

queue  *tcp_reseq_queue;    /* Queue of out_of_order packets received */
                            /* from the dlc waiting to be processed */
                            /* This is accessible by the following functions */
```

```
                                /* Insert_pdu_into_reseq_queue(); */
                                /* Process_packets_from_resequence_queue(); */

  int           tcp_dupacks;  /* duplicate acks count : for Fast  rexmit and recovery : NOT REQUI
  tick_t        tcp_t_rtt;    /* non-zero if segment is being timed, 0 otherwise */
  int  tcp_rtseq;    /* which segment is being timed :
                                     rtseq variable in stevens */
```

The following subsections describe how TCP uses the connection state structure to manage an active connection.

5.2 TCP sliding windows

TCP uses the sliding window mechanism for flow control. The sliding window protocol is shown in figure 2.

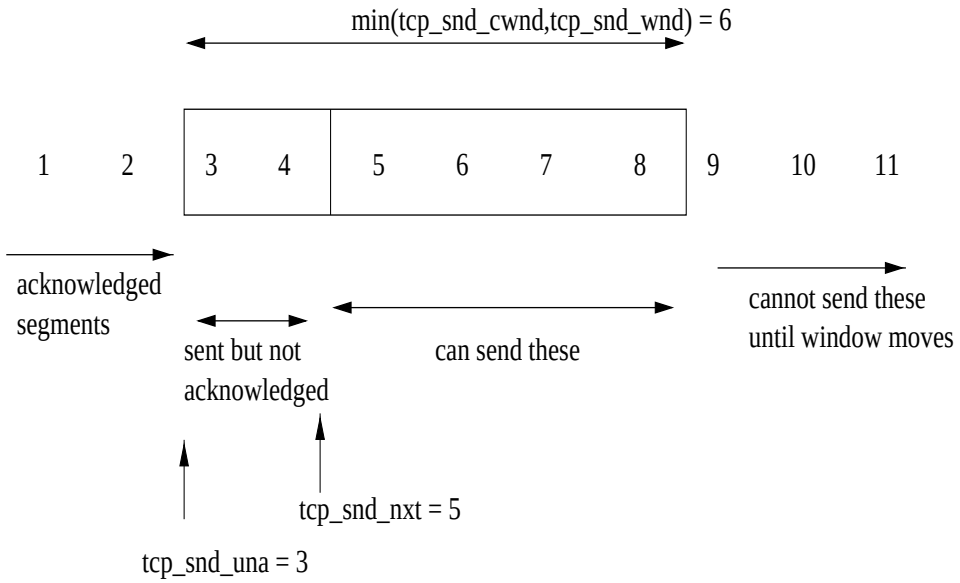

Figure 2: Sliding windows

The window advertised by the receiver is stored in `tcp_snd_wnd`. `tcp_snd_una` is the sequence number of the last unacknowledged byte (3) and `tcp_snd_nxt` is the sequence of the next byte to be sent (5). The TCP layer sends a segment, only if the window is not exhausted, i.e., `tcp_snd_nxt` − `tcp_snd_una` +1 ≤ `tcp_snd_wnd`. When a packet is acknowledged, `tcp_snd_una` is incremented to the next unacked packet. Remember that acks contain the sequence number of the next byte expected by the receiver.

5.2.1 Slow start

TCP is required to support an algorithm called *slow start*. Slow start is used to control the rate of packets injected into the network so that the sender does not overload the network. Slow start adds another window to the sender TCP's state, called the *congestion window*. This is denoted by the variable `tcp_snd_cwnd`.

When a new connection is established, `tcp_snd_cwnd` is initialized to one segment. (For most TCP implementations, the maximum segment size is 512 bytes and this is the value we will use in the lab. Note that the congestion window is maintained in bytes). The sender can transmit upto

a minimum of the congestion window and the receiver's advertised window (`tcp_snd_wnd`). Every time an ACK is received, the congestion window is increased by one segment size in bytes. The congestion window is not allowed to grow beyond the receiver's advertised window (For most TCP implementations, the maximum window size is 64K bytes and we will also use this value).

As a result of the slow start algorithm, the sender starts by transmitting one segment and waiting for the ACK. When it receives the ACK, it increases the congestion window by one segment size, so now it can send two segments. For each of these segments' ACKs, the congestion window is increased by one. The time between sending the segment and receiving its ACK is approximately one round trip time for the connection. As a result, the congestion window doubles about every round trip time due to slow start. Thus, slow start specifies an exponential increase of the congestion window for every round trip. It should be noted that slow start is flow control imposed by the sender to protect the network from being flooded, while the advertised window is flow control imposed by the receiver based on its buffer space.

At some point, the congestion window may become large enough to exceed the capacity of the network. In this case the network may drop one or more segments from the connection. The next two subsections describe how the TCP connection recovers from packet loss.

5.3 TCP timeout and retransmission

TCP ensures reliable packet delivery by using a timeout and retransmission technique. When a segment is lost in the network, the destination sends a duplicate ACK for each out of order segment it receives. When the sender receives a duplicate ACK, it does not increment its `tcp_snd_una` variable, and the window based flow control prevents the sending of more than one window (minimum of congestion and advertised windows) of unacknowledged segments. The sender also maintains a retransmission timer for the first unacknowledged packet. When the timer goes off, this packet is retransmitted, and the timer is reset. Most TCP implementations maintain a coarse granularity timer which ticks every t time units. TCP adaptively measures the round trip time of the connection, and maintains it in terms of timer ticks. The retransmission timeout is also maintained in timer ticks based on the round trip time of the connection. If the retransmission timeout is n timer ticks, then the TCP will retransmit the oldest unacknowledged packet after the timer ticks n times. When a new ACK is received, the retransmission timeout is restarted. TCP uses the Go-back-N mechanism for retransmission. Thus, when retransmission timout occurs, `tcp_snd_nxt` is set to `tcp_snd_una` and all the packets starting from the oldest unacknowledged packets are retransmitted.

Slow start is the way to initiate data flow across a connection. When packets are lost, TCP also uses another algorithm called *congestion avoidance*. Congestion avoidance introduces another variable at the TCP sender called slow start threshold `tcp_snd_ssthresh`. Before a packet is retransmitted, `tcp_snd_ssthresh` is set to half the congestion window. The congestion window is set to one segment. As a result, when congestion occurs TCP enters slow start mode. When the congestion window exceeds `tcp_snd_ssthresh`, then the sender enters the congestion avoidance phase. In this phase, the congestion window is increased by the reciprocal of the congestion window, each time an ACK is received. This results in a linear increase (by one segment size) of the congestion window every round trip time.

The slow start and congestion avoidance algorithms can be described by the following steps. Here "cwnd" stands for `tcp_snd_cwnd` and "ssthresh" stands for `tcp_snd_ssthresh`.

1. Initally, cwnd is set to 1 segment size, and ssthresh is set to the maximum window size (65535 bytes).

2. The sender TCP can send up to a minimum of the cwnd and the receiver's advertised window of unacknowledged bytes.

3. When a packet is lost (indicated by a timeout), ssthresh is set to one-half of the current window size (the minimum of the congestion window and the advertised window). Also, if timeout occurs, cwnd is set to one segment. The ssthresh variable is always maintained as a

multiple of the maximum segment size. Also, ssthresh has a minimum value of two segments. The following piece of code ensures this while setting ssthresh to cwnd/2.

```
win = (Minm(tcp_layer->tcp_snd_cwnd,tcp_layer->tcp_snd_wnd)) / 2 /
                                        tcp_layer->tcp_mss;
if (win < 2) win = 2;
tcp_layer->tcp_snd_ssthresh = win * tcp_layer->tcp_mss;
```

The Go-back-N protocol now kicks in at the TCP layer, and retransmits all segments starting from the missing segment. In most TCP implementations (and in this lab), a function called `tcp_output()` is called whenever the TCP layer wants to transmit packets. This function transmits as many packets as possible starting from `tcp_snd_nxt`. As a result, during retransmission, `tcp_snd_nxt` is set to `tcp_snd_una` before `tcp_output()` is called.

4. When new data is acknowledged, the congestion window is increased as follows:

   - If cwnd is less than or equal to ssthresh, slow start is performed and the congestion window is increased by one segment size for each new ACK received.

     ```
     cwnd = cwnd + tcp_mss;
     ```

   - If cwnd is greater than ssthresh, congestion avoidance is performed and the window is increased by $1/cwnd$ for each new ACK received:

     ```
     cwnd = cwnd + tcp_mss * tcp_mss / cwnd;
     ```

     cwnd is always maintained below the receiver's advertised window.

5.4 Function descriptions

The following functions are given in tcp_layer.c. You have to fill in the code for the functions indicated. You should study each of the functions carefully to understand what they are expected to do.

- `tcp_layer_receive(TCP_LAYER_ENTITY_TYPE *tcp_layer, GENERIC_LAYER_ENTITY_TYPE *generic_layer_entity, PDU_TYPE *pdu)`: Processes a received packet. **You SHOULD NOT modify this function.**
- `tcp_granularity_timer(TCP_LAYER_ENTITY_TYPE *tcp_layer)`: Processes the ticking of the coarse granularity timer. **You SHOULD NOT modify this function.**
- `tcp_output(TCP_LAYER_ENTITY_TYPE *tcp_layer)`: Outputs a packet to the datalink layer if possible. **You NEED TO modify this function.**
- `output_pkt(TCP_LAYER_ENTITY_TYPE *tcp_layer, int offset, int len, int samplenow)`: Actually outputs the packet to the datalink layer. Called by `tcp_output()` when necessary. **You SHOULD NOT modify this function.**
- `tcp_input(TCP_LAYER_ENTITY_TYPE *tcp_layer, PDU_TYPE *pdu)`: Processes a packet received from the datalink layer. **You SHOULD NOT modify this function.**
- `process_insequence_pkt(TCP_LAYER_ENTITY_TYPE *tcp_layer, PDU_TYPE *pdu)`: Called by `tcp_input()` to send the pdu to the application layer. Must also call the function `Process_packets_from_resequence_queue()` that sends more pdu's to the application layer if possible. **You NEED TO modify this function.**

- `process_outofsequence_pkt(TCP_LAYER_ENTITY_TYPE *tcp_layer, PDU_TYPE *pdu)`: Called by `tcp_input()`. If fast retransmit is turned on, then the out of sequence packet must be stored in the resequence queue – `Insert_pdu_into_resequence_queue()` – otherwise the pdu must be free'd. **Ignore fast retransmit functionality for this lab.**

- `process_ack(TCP_LAYER_ENTITY_TYPE *tcp_layer, PDU_TYPE *pdu)`: Processes an ACK that was received. **You SHOULD NOT modify this function.** But study it carefully because you need to modify some functions it calls.

- `reset_dupacks(TCP_LAYER_ENTITY_TYPE *tcp_layer)`: Resets (if fast retransmit is on) the duplicate ACK count if necessary. **Ignore fast retransmit functionality for this lab.**

- `slow_start(TCP_LAYER_ENTITY_TYPE *tcp_layer)`: Performs the slow start and congestion avoidance algorithms. **You NEED TO modify this function.**

- `fast_rexmit_code(TCP_LAYER_ENTITY_TYPE *tcp_layer, T_PDU_TYPE *pkt)`: Implements the fast retransmit and recovery algorithm. **Ignore fast retransmit functionality for this lab.**

- `tcp_retrans_timeout(TCP_LAYER_ENTITY_TYPE *tcp_layer)`: Implements the timeout and retransmission strategy. **You NEED TO modify this function.**

The following functions are provided to you. You **DO NOT NEED TO KNOW the implementation details of these functions. You only need to know the FUNCTIONALITY as described below**.

- `send_pdu_to_datalink(TCP_LAYER_ENTITY_TYPE *tcp_layer, PDU_TYPE *pkt)`: Sends the t_pdu to the datalink layer.

- `send_pdu_to_application(TCP_LAYER_ENTITY_TYPE *tcp_layer, PDU_TYPE *pdu)`: Sends the a_pdu to the application layer.

- `Insert_pdu_into_send_queue(TCP_LAYER_ENTITY_TYPE *tcp_layer, GENERIC_LAYER_ENTITY_TYPE *generic_layer_entity, PDU_TYPE *pdu)`: Inserts the a_pdu into the TCP send queue. An offset or sequence number is assigned to the first byte of the pdu.

- `int Calculate_max_send_queue_offset(TCP_LAYER_ENTITY_TYPE *tcp_layer)` : Returns the maximum sequence number of a byte in the send queue of the TCP layer.

- `Get_pdu_from_send_queue(TCP_LAYER_ENTITY_TYPE *tcp_layer, PDU_TYPE *pkt, int offset)`: Copies the a_pdu with the matching offset and copies it to the a_pdu of `pkt`. It does not delete the packet from the send queue.

- `free_acked_data(TCP_LAYER_ENTITY_TYPE *tcp_layer, int nbytes)`: Removes data that has been acknowledged from the send queue. Note that, packets transmitted cannot be removed from the send queue until they have been acked.

- `Insert_pdu_into_resequence_queue(TCP_LAYER_ENTITY_TYPE *tcp_layer, PDU_TYPE *pdu)`: When the TCP receives an out-of-order packet, it uses this function to store the t_pdu in the resequence queue.

- `Process_packets_from_resequence_queue(TCP_LAYER_ENTITY_TYPE *tcp_layer, PDU_TYPE *pdu)`: Sends as many pdu's as possible from the resequence queue to the application layer. This function also updates the `rcv_nxt` variable.

- `static int Log_pkt_stats(TCP_LAYER_ENTITY_TYPE *tcp_layer, PDU_TYPE *pkt)`: Performs some logging functions. This is called by the function `output_pkt()`.

- `update_rto(TCP_LAYER_ENTITY_TYPE *tcp_layer, PDU_TYPE *pdu)`: Updates the round trip timer calculations. This is called by `tcp_input()`.

- `trim_pkt(TCP_LAYER_ENTITY_TYPE *tcp_layer, PDU_TYPE *pdu)`: Trims the duplicate bytes in a received packet. This is called by `tcp_input()`.
- `int update_cwnd(TCP_LAYER_ENTITY_TYPE *tcp_layer, int value)`: Sets the variable `tcp_snd_cwnd` to `value`. Also plots the congestion window. This must be used instead of simply assigning the congestion window variable to value.
- `int timer_backoff(TCP_LAYER_ENTITY_TYPE *tcp_layer)` : This implements the exponential backoff of the TCP timer. This is called in `tcp_retrans_timeout` before calling `tcp_output()` to retransmit a packet.
- `misc_functions_on_ack(TCP_LAYER_ENTITY_TYPE *tcp_layer)`: Performs some housekeeping when ACKs are received.
- `int shut_timer_off(TCP_LAYER_ENTITY_TYPE *tcp_layer)`: Turns off the retransmit timer. This is called in `fast_rexmit_code()` just before calling `tcp_output()`. **Ignore fast retransmit functionality for this lab.**
- `TCPFromApplication(src)`: This macro returns a 1 if the packet is from the Application layer, else returns 0.
- `TCPFromDatalink(src)` : This macro returns a 1 if the packet is from the Datalink layer, else returns 0.
- `fast_rexmit(TCP_LAYER_ENTITY_TYPE tcp_layer)` : This macro returns 1 if fast retransmit and recovery has been turned on, and 0 otherwise. **Ignore fast retransmit functionality for this lab.**

6 Deliverables

1. In directory `/home/81/kalyas/public/Lab2/Files/`, you can find these files:
   - `pdu.h`: header file containing some declarations and definitions. You don't need to include this file anywhere in your source code because it is already included in `tcp_layer.h`. You will need to use some of the function definitions provided in this file, like `pdu_alloc()` and `pdu_free()`
   - `tcp_layer.c`: file containing the outline for the lab.
   - `makefile`: makefile for the lab.
   - Five configuration files: `*.config`. These files specify the configuration of the network. In this lab you will only use a 2 node configuration with app1 sending to app2. The configuration is shown in figure 3. The configuration files specify different parameters for

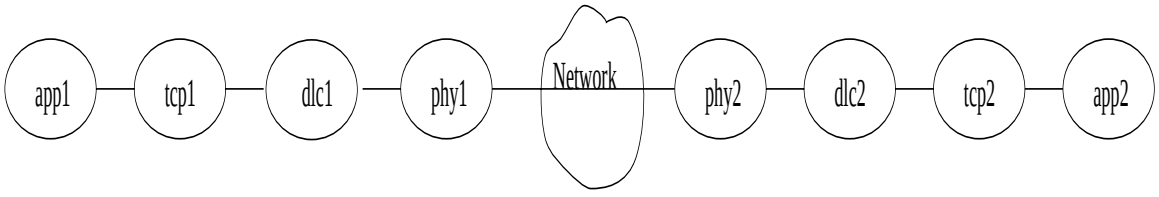

Figure 3: TCP configuration

   the simulations:
   - `no_error.config`. No packets are dropped by the network.

- `1_error_ss.config`. The network drops a single packet. Fast retransmit and recovery are not activated i.e., `fast_rexmit(tcp_layer)` returns 0.
- `n_error_ss.config`. The network drops multiple packets. Fast retransmit and recovery are not activated.

- `TCP_demo`: a sample executable file to familiarize you with the graphical user interface. You can also do part 1 (see below) of the lab using this executable.
- `drawgraphs`: a script that will generate graphs that you will need to submit.

2. Copy the above files to your directory.
3. Experiment with `TCP_demo`. Note: use only app1 to send the picture. Do not send the graph from app2. See section 7 for further instructions on how to run your program. Pay careful attention to the congestion window and sequence number graphs of tcp1.
4. **Part 1**: Answer the following questions using `TCP_demo`:
   (a) Interpret the sequence number and congestion window graphs for the sending TCP (tcp1) for each configuration file. Also explain the difference in the amount of data that has been reliably transmitted in each configuration.
   (b) Using a rough hand sketch or a window capture (using xv) indicate where the slow start and congestion avoidance phases occur.
   (c) Indicate when the packets are retransmitted, and when duplicate ACKs are received.
5. Study the source code files carefully. (don't worry about the config files).
6. Now you are ready to write your program for the TCP layer.
7. **Part 2**: Implement the window flow control, the retransmission timeout and the slow start and congestion avoidance algorithms. You should first test the window flow control with the `no_error.config` file, and then move on to implement the next steps. For the timeout and retransmission, you only have to work with `tcp_retrans_timeout()`. Assume that this function is called when the retransmission timeout goes off.
8. To compile your program, type **make**. Part 2 can be compiled without doing part 3. This will produce an executable called `TCP_exec` in your working directory.
9. Now execute your version of the code and use the configuration files to make sure it works.
10. Run the script `drawgraphs`. The script will run your executable and produce 10 postscript files (two for each configuration). These files will contain graphs of the congestion window and the sequence numbers (both send and ACK) of the sending TCP. You must submit hard copies of these graphs along with your `tcp_layer.c`

7 Running your program: The Graphical User Interface

The general GUI instructions for this lab are the same as Lab1. RSVP students, please refer to Lab 1 handout, FAQ and WebCT bulletin board for issues regarding how to get the GUI working from a remote location. Described below are the differences in the nature of the GUI which is specific to TCP behavior.

Login to an AIX machine on campus. **RECALL: This program works only on AIX machines.** At your unix prompt, type `TCP_demo no_error.config`. A window with the simulator interface will open on your terminal. Each node has 4 layers denoted by the squares. The network is denoted by the cloud. In addition to the simulator window, there are 6 smaller windows visible in the figure. For each application layer, there is a send/receive window where you can load a graphic

and send it to the other end. Each TCP layer has a congestion window plot and a sequence number plot associated with it. The congestion window plot shows the variation of the congestion window with time. The sequence number window has 3 plots associated with it:

- The sequence number of the first byte of each segment sent by the TCP (red dots).
- The ACK number in the acknowledgement received by the sender (gold dots).

You should use the information in these windows to debug your program and answer the question in part 1.

The top of the main simulator window has a menu bar that has the following selections.

Menu	Submenu	Description
File	Load	Load config file
	Exit	Stop the program and exit
Edit	Raise	Raise the node graphs and space-time diagrams to the front of main window. Useful when you cannot see the graphs.
Run	Pause	Pause the simulation.
	Resume	Resume the simulation.
	Delay	Set delay between events. The default is 2.
	Stop Time	The simulation time to stop.
	Inc/Dec Debug Level	Increament/Decrement Debug Level, which is used in dprintf().
Help	About	About the CISE Project.
	How to use	Help text.

The bottom of the main simulator window is the status bar with the following information.

Field	Description
Filename	The configuration file name.
Stop Time	The end time for the simulation
Delay	Delay between events.
DebugLevel	DebugLevel used in dprintf().
Simulation time	The clock in the simulator.

- First, load a config file. This can be done by File/Load or by putting the config file at the command line of this program.
- To send a graph, "Load Graph" first and then "Send Graph." and the simulator will load the graph and start running. The order of this operation is important. Also, only load and send from node 1.
- Change the delay to make it run faster/slower. Use "Run/Single step" and space bar to see it step by step.
- Change Debug Level (from 0–3) and use `dprintf(int debug_level, ''format'', variables)` in your program to print out debug information.

8 Submissions

You must submit **hard copies** of the following:

- A *short* summary of the lab, and answers to the questions in part 1.
- Your source code.
- Printed copies of the 6 graphs (.ps files) produced by `drawgraphs`.

9 Miscellaneous Notes

- For questions, **FIRST CHECK THE FAQ (on course web page)**, and then post questions to the bulletin board ONLY. Do not send emails directly to the instructor or TAs. The TAs are responsible for prompt replies to your technical questions on the bulletin board.
- TA office hours: Ye Tao: Mon 3-4pm (1hr) at JEC 6212, x8289
  Hua Qin: Tue 12-1pm (1hr) at JEC 6212, x8289
  Karthik: Tue 2-4 pm (2hr) at JEC 6212, x8289
  Jye-Young Song: Thr 2-4pm (2hrs) at JEC 6010
- The following are the only IBM AIX machines on campus: rcs-ibm1.rpi.edu, rcs-ibm2.rpi.edu. "rcs-ibm1" is a J40 with 4 processors and 512Mb of memory and rcs-ibm2 is a J50 with 6 processors and 512Mb of memory. Both machines are robust and currently have a total of 74 logins at this time, with 12am-8am seeing the lowest use. During peak usage, normally between 10am and 6pm, it is not uncommon to see a couple hundred logins. The machines are both configured to prevent single users from significantly impacting available CPU time, by killing processes that single-handedly consume more than an hour of CPU time. Between this and automatic load balancing between the processor, the machines do not bog down easily. Currently the load average for the last 15 minutes has been 16% and 11% on the two machines, and each can reach up to 200-300% per processor before significantly impacting user speeds, particularly for quick processes. Please use this information as you plan to access these machines.
- If you are not local on an AIX machine, telnet to one and display back to you local workstation. You need to contact your local system administrator if you are not able to re-direct display from an rcs workstation to your local UNIX workstation. **Remember again that exporting display does not work over corporate firewalls**.