Project 3
Reliable Data Transfer over UDP

NTU CSIE Computer Networks 2011 Spring
In Project 3, students are asked to understand and implement reliable data transfer mechanism over UDP.
UDP

- UDP is a simple transport-layer protocol
  - No connections and hand-shaking
  - No guarantee of reliability, packet order, and may lose packets or have duplicate ones
  - Only packet integrity is checked (but corrupt packets are simply dropped without auto-retransmission)
Goal:
Reliable data transmission

- Send data from client to server
  - The channel is not perfect: packet may be lost or corrupted
  - Make sure the data received by the server IS EXACTLY THE SAME as it is on the client
Bonus

- **Flow control**
  - “Managing the rate of data transmission between two nodes to prevent a fast sender from over running a slow receiver”
    --http://en.wikipedia.org/wiki/Flow_control

- **Congestion control**
  - Control the traffic when network congestion occurs
Project Scenario

Internet

*Packets may be lost, corrupted, or re-ordered*

Server

UDP pkt

Client
Requirements

- The client will send a file to the server using UDP. The file received should be EXACTLY THE SAME as the original one.

- You can use any means, including mimicking TCP mechanism.

- Although performance is not required, your program should run in reasonable time:
  - Taking hours to send kilobytes is not acceptable.
Requirements

- We’ll use proxy that drops packets to test your implementation
- Write your programs on R217 workstations
- Test them across real Internet, not only in LAN
Score

- Basic function 60%
- Makefile and command 10%
  - make clean // clean obj(*.o) and executable
  - make // compile source code
  - ./client ip port path_to_file // Run client
  - ./server port file_to_save // Run server
- Report (.pdf) 30%
  - FSM of your RDT mechanism and explain why it is reliable
  - Other features you implement
- Bonus (flow, congestion control ...) +10%
File format

- Max. 2 people in one group
  - Please tell TA if you change group members: freetempo17@gmail.com

- Tar your file
  - `tar zcvf bXXXXXXXX_b00000000_prj3.tar.gz Makefile report.pdf file1 file2 ...
  - Use "bXXXXXXXX_prj3.tar.gz" if you do not team with others
  - DO NOT have any directory in .tar file; all files should be extract to current directory when using "tar zxvf bXXXXXXXX_prj3.tar.gz"
  - Upload your file to FTP server
    - 140.112.29.99:21 cn2011/cn20115566
Deadline

- **2011/6/15 23:59**
- There WOULD BE penalty for late submission. The penalty for the first day is 10 points, the second day is 20 points, etc. Please turn in your code in time.
Socket Programming

- Create a UDP socket
- Bind a UDP socket
- Send and receive datagrams
Create a UDP socket

```c
#define ERROR   -1
...
int sock;
// create a socket
if((sock = socket(AF_INET, SOCK_DGRAM, 0)) == ERROR) {
    perror("server socket: ");
    exit(-1);
}
...
Bind a UDP socket

- UDP socket doesn’t need listen() and connect()
  - If you bind() it to a port, you can send and receive datagrams through that port
  - Or, it would be assigned a random port when it’s first used
Bind a UDP socket

... 

struct sockaddr_in server;
bzero(&server, sizeof(server));
server.sin_family = AF_INET;
server.sin_port = htons(1234);
server.sin_addr.s_addr = INADDR_ANY;

// bind a socket
if(bind(sock, (struct sockaddr *)&server, sizeof(server)) == ERROR)
{
    perror("bind : ");
    exit(-1);
}
...

...
Send and receive datagrams

- UDP sockets are not “connected” to remote sockets
  - You designate the destination every time you send datagrams
  - You’ll get the source information when receiving datagrams
Send datagrams

- `ssize_t sendto(
  int sockfd,
  const void *buf,
  size_t len,
  int flags,
  const struct sockaddr *dest_addr,
  socklen_t addrlen)`

- First 3 parameters are just like `write()`
- flags: control the behavior, you can simply use 0 here
- dest_addr and addrlen: like the ones you use in `connect()`; they designate the destination of the datagram
- Returns the number of characters sent
Send datagrams

...  
struct sockaddr_in addr;
bzero(&addr, sizeof(addr));
addr.sin_family = AF_INET;
addr.sin_port = htons(1234);
addr.sin_addr.s_addr = inet_addr("140.112.30.41");

char data[BUF_SIZE];
size_t data_len;
...
sendto(sock, data, data_len, 0, (struct sockaddr *)&addr,
sizeof(addr));
...
Receive datagrams

- `ssize_t recvfrom(
    int sockfd,
    void *buf,
    size_t len,
    int flags,
    struct sockaddr *src_addr,
    socklen_t *addrlen)

- First 3 parameters are just like read()
- flags: control the behavior, you can simply use 0 here
- src_addr and addrlen: retrieves the source of the datagram, you should initialize addrlen before recvfrom()
- Returns the number of bytes received
Receive datagrams

...  
struct sockaddr_in  addr;
socklen_t  addrlen = sizeof(addr);
bzero(&addr, sizeof(addr));

char  data[BUF_SIZE];
...
recvfrom(sock, data, BUF_SIZE, 0, (struct sockaddr *)&addr, &addrlen);
...
How to implement timeout?

- Get current time (in microsecond (us))
  - int gettimeofday (  
    struct timeval *tv,  
    struct timezone *tz)  
  - tv: a timeval structure to record the number of seconds and microseconds since the Epoch  
  - tz: timezone info., can be NULL  
  - “man gettimeofday” to get more information
How to implement timeout?

- int select(
  int maxfds,
  fd_set *readfds,
  fd_set *writefds,
  fd_set *exceptfds,
  struct timeval *timeout)

- Use timeout to designate how long select() can block
How to implement timeout?

- Example
  - Current time is 1000ms, we know that a timeout event is scheduled on 1200ms

  ```c
  struct timeval tv;
  tv.tv_sec = 0;
  tv.tv_usec = (1200-1000) * 1000; // in microsecond
  select(... , &tv);
  
  Select would return if
  - Some descriptors are ready, or
  - Time is up (200ms), and the return value would be 0