CS4514 HELP Session 3

Concurrent Server Using Go-Back-N

Song Wang

12/08/2003
Description

- You are supposed to implement a simple concurrent server and client having four emulated network protocol stack.
  - Application layer: Read and execute commands
  - Network layer: Message $\rightarrow$ Packet (send&recv)
  - Datalink layer: Packet $\leftrightarrow$ Frame and Go-Back-N sliding window protocol
  - Physical layer: TCP connection.

- Your programs should compile and work on any one of ccc.WPI.EDU.
System Overview

Client1

Client2

Server

Child Process1

Child Process2

script1.txt

script2.txt

serverbase.txt

Note: each child process keeps a separate copy of the DB.
we do not keep data consistency for the serverbase
System Framework

Client

APP Layer

NW Layer

DLL

PHL

Server

APP Layer

NW Layer

DLL

PHL

TCP Connection
Concurrent Server (fork())

- fork() will make a child process with memory copy.
  - The initial serverbase will be copied to each child process.
  - fork() will return child pid in parent process and 0 in child process.
  - Remember to close socket after using.
Concurrent TCP Server Example

```c
pid_t pid;
int listenfd, connfd;

/* 1. create a socket socket() */
if ((listenfd = socket(AF_INET, SOCK_STREAM, 0)) < 0 )
  err_quit("build server socket error\n", -1);
/* 2. fill in sockaddr_in{ } with server's well-known port */
...
/* 3. bind socket to a sockaddr_in structure bind() */
bind (listenfd, ...);
/* 4. specify the backlog of incoming connection requests  listen() */
listen (listenfd, LISTENQ);
while(1){
    connfd = accept(listenfd, ...); /* probably blocks */
    if(( pid = fork()) == 0){
        close(listenfd); /* child closes listening socket */
        doit(connfd);    /* process the request */
        close(connfd);    /* done with this client */
        exit(0);
    }
    close(connfd); /* parent closes connected socket */
}
```

How the System Works: Layer by Layer

Client

Read “scripted action” from file “scripti.txt”

Server

Read/Write a message

Child Process i

Client Request:

\[ \text{cmd No. [msg]} \]

\begin{align*}
\text{cmd::r q w} \\
\text{msg1::r[6](2 bytes)} \\
\text{msg2::w[4]Duke...} \\
\text{msg3::q(1 byte)}
\end{align*}

\text{No sequence no. for msg}

\text{nwl_send (… msg …)}

\text{nwl_recv (… msg …)}

\text{msg1::Duke...}

\text{msg2::[ACK]}

\text{msg3::[ACK]}

\text{Note: The max_size of a message is 290 bytes}

\text{The number referring to tuple position is 1 to 14}
How the System Works: Layer by Layer

Client

message

NWL

n_packets

End of Message
Can be an special packet, a special position in each packet, eg. 1st byte

Server

message

NWL

n_packets

Tasks for NWL
Disassemble and assemble packets from Msg.
No response in this layer
No sequence no. for packets

dll_send (… pkt …)
dll_recv (… pkt …)

Note: The max_size of a packet is 72 bytes
The network layer will send packets until blocked by the Data Link Layer
How the System Works: Layer by Layer

Client

packet

DLL

n_frames

End of Packet
Error Detection
Byte Stuffing
Go-Back-N

phl_send (… frm …)

phl_recv (… frm …)

Server

packet

DLL

n_frames

ACK
Go-Back-N

Note: The max_size of a frame payload is 48 bytes

Sliding window size >= 3

12/08/2003

Computer Science
Session 3–9
How the System Works: Layer by Layer

Client

- frame
- PHL
- read (… data …)
- write (… data …)
- TCP Connection

Server

- frame
- PHL
- Identify client when start
- Force Single Bit Errors
  - Client: 5th frames
  - Server: 6th frames
How the Functions Work: Layer by Layer

client APP

Read script file

Pick a command

q cmd?

Yes

Build Msg

nwl_send(…msg…)

nwl_recv(…ack…)

No

server child process

APP

fork()

nwl_recv(……)

q cmd?

Yes

Build Msg

nwl_send(…msg…)

No
How the Functions Work: Layer by Layer

\[ \text{nwl\_send (… msg …)} \quad \text{nwl\_recv (… msg …)} \]

- Split msg into pkts
- Pick a pkt
- Last pkt?
  - No: dll\_send(… pkt …)
  - Yes: Add sth, ‘\0’
    - dll\_send(… pkt …)

- dll\_recv (… pkt …)
  - Last pkt?
    - No
    - Yes: Reassemble pkts into msg
      - Return msg to APP
How the Functions Work: Layer by Layer

dll_send (… pkt … )

Split a packet into payloads

Create a new frame

Start a Timer

Send a frame to PHL

Wait for receiving a ACK frame

Retransmit frames *if timeout or error ACK frame!*

Receive a ACK frame correctly, then continue ...

- `phl_send (…)`
- `phl_recv (…)`

Sliding window size = 1
How the Functions Work: Layer by Layer

dll_recv (… pkt … )

Receive a frame from PHL

Unstuffing the frame

Compute ED byte and check error

Drop if error detected

Drop if duplicate, else send ACK

Reassemble the packet

If EOP, forward the packet to NWL

phl_recv (…)

phl_send (…)

Computer Science

Session 3-14
Log Significant Events

Performance Timing

Packet Sent

......

Frame received in error

client\textit{i}.log

server\textit{i}.log
Project Tips

- Sliding Window Protocol: Go-Back-N (N>3)
  - Try to implement Go-Back-1 first
  - Then implement Go-Back-N (multiple timers)
- Maybe easier to merge PHL and DLL
- How to terminate client process:
  - When the client gets the response to the quit message
  - A “clean” way to terminate the server child process?
Relative Timer Example

/* example for start_timer, stop_timer, send_packet */
/* you SHOULD modify this to work for project 3, this is just a TIMER EXAMPLE */
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <sys/time.h>
#include <sys/timers.h>
#include <sys/select.h>
#include <sys/types.h>
#include <errno.h>
define TIMER_RELATIVE 0
define MAX_SEQ 3
 extern int errno;
typedef unsigned int seq_nr;
typedef enum {frame_arrival, cksum_err, timeout, network_layer_ready} event_type;
timer_t timer_id[MAX_SEQ];
void timeout() {
    printf("time out!\n");
}

void start_timer(seq_nr frame_nr) {
    struct itimerspec time_value;
    signal(SIGALRM, timeout);
    time_value.it_value.tv_sec = 1; /* timeout value */
    time_value.it_value.tv_nsec = 0;
    time_value.it_interval.tv_sec = 0; /* timer goes off just once */
    time_value.it_interval.tv_nsec = 0;
    timer_create(CLOCK_REALTIME, NULL, &timer_id[frame_nr]); /* create timer */
    timer_settime(timer_id[frame_nr], TIMER_RELATIVE, &time_value, NULL); /* set timer */
}

void stop_timer(seq_nr ack_expected) {
    timer_delete(timer_id[ack_expected]);
}

void send_packet(packet *p) {
    fd_set readfds;
    int sockfd;

while(packet hasn’t been finished sending) {
    /* send frame if we can */
    while(there’s place left in sliding window) {
        /* construct a frame from the packet */
        /* send this frame; start timer; update sliding window size */
    }
    /* check data from physical layer */
    FD_ZERO(&readfds);
    FD_SET(sockfd, &readfds);
    if (select(sockfd+1, &readfds, (fd_set *)NULL, (fd_set *)NULL, (struct timeval*)NULL) < 0) {
        if (errno == EINTR) { /* receive timeout signal */
            /* timeout handler should have resent all the frames that haven’t been acknowledged */
            continue;
        } else {
            perror("select error"); /* select error */
            exit(1);
        }
    }
}
if (FD_ISSET(sockfd, &readfds)) { /* a frame come from socket */
  /* read a frame from the socket */
  if (cksum() == FALSE) { /* error check */
    continue; /* do nothing, wait for timer time out */
  }
  else {
    /* check to see if this frame is a data or ACK frame, and do corresponding processing */
    continue;
  }
}
}