Concurrent Server Using Go-Back-N

Feng Li
lif@cs.wpi.edu
12/01/2004
Objective:
To implement a simple concurrent server and clients having four emulated network protocol stacks.
- Application layer: Read and execute commands
- Network layer: Message $\leftrightarrow$ 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 host of ccc.WPI.EDU.
System Overview

Note: each child process keeps a separate copy of the DB.
we do not keep data consistency for the serverbase
This is automatically done by using fork()
System Framework

Client

APP Layer

NW Layer

DLL

PHL

Server

APP Layer

NW Layer

DLL

PHL

TCP Connection

Four Layer stacks
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.
- More information could be found at fork(2).
Go Back N

Go-Back-3:
3 frames are outstanding; so go back 3

Out-of-sequence frames

ACKing next frame expected
How the System Works: Layer by Layer

Application Layer

Client

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

Server

Read/Write a message

Child Process i

ACK

Stop-Wait

Client Request:

`cmd` No. [`msg`]  
`cmd::r / w / q [1-12]`

msg1::r[6]

msg2::w[4]Duke...

msg3::q

NO sequence# for msg

APP

`nwl_send` (… `msg` …)

msg1::John...

msg2::[ACK]

msg3::[ACK]

APP

`nwl_recv` (… `msg` …)

Note: The max_size of a message is 270 bytes

The number referring to triple position is 1 to 12

12/1/2004
How the System Works: Layer by Layer
Network Layer

Client

Message

NWL

n_packets

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

dll_send (… pkt …)
dll_recv (… pkt …)

Server

Message

NWL

n_packets

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

Note: The max size of a packet is 90 bytes, The network layer will send packets until blocked by the Data Link Layer. But HOW?

12/1/2004
How the System Works: Layer by Layer

DataLink Layer

Client \( \rightarrow \) Server

- \( phl\_send(\ldots\, frm\, \ldots) \)
- \( phl\_recv(\ldots\, frm\, \ldots) \)

- \( n\_frames \)

- ACK
- Go-Back-N

- End of Packet
- Error Detection
- Byte Stuffing
- Go-Back-N(\(N \geq 3\))

Note: The max_size of a frame payload is 40 bytes

Sliding window size \( \geq 3 \)

12/1/2004
How the System Works: Layer by Layer

**Client**

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

**Server**

- frame
- PHL

- Identify client when start
- Force Single Bit Errors
  - Client: 4th frames
  - Server: 7th frames

TCP Connection

12/1/2004
How the Functions Work: Layer by Layer

client APP

- Read script file
- Pick a command
- q cmd?
  - No: Build Msg
    - nwl_send(…msg…)
    - nwl_recv(…ack…)
  - Yes: Build Msg

server child process

APP

- fork()
- nwl_recv(……)
- q cmd?
  - No:
  - Yes
    - Build Msg
      - nwl_send(…msg…)

How the Functions Work: Layer by Layer

\textbf{nwl}_\texttt{send} (\ldots \texttt{msg} \ldots) \quad \textbf{nwl}_\texttt{recv} (\ldots \texttt{msg} \ldots)

- Split \texttt{msg} into \texttt{pkts}
- Pick a \texttt{pkt}
  - Last \texttt{pkt}?
    - \texttt{No}:
      - \textbf{dll}_\texttt{send}(\ldots\texttt{pkt}\ldots)
    - \texttt{Yes}:
      - Chang EOM
- \textbf{dll}_\texttt{recv}(\ldots\texttt{pkt}\ldots)
  - Last \texttt{pkt}?
    - \texttt{No}:
    - \texttt{Yes}:
      - Reassemble \texttt{pkts} into \texttt{msg}
      - Return \texttt{msg} to \texttt{APP}

Note: you need have a mechanism to decide the last packet in a frame.
How the Functions Work: Layer by Layer

\texttt{dll\_send} (… pkt … )

1. Split a packet into payloads
2. Create a new frame
3. Start a Timer
4. Send a frame to PHL
5. Wait for receiving a ACK frame
   - Retransmit frames if timeout or error ACK frame!
6. Receive a ACK frame correctly, then continue ...

\texttt{Sliding window size = 1}

\texttt{phl\_send} (…)

\texttt{phl\_recv} (…)
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
Log Significant Events

Performance Timing

Packet Sent

......

Frame received in error

client$_i$.log

server$_i$.log
Project Tips-1

- 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?
Project Tips-2

- Simulate multiple timer in software
  - Approach I
    - Using link list or array
    - pp.223 on textbook()
    - Need signal()
  - Approach II
    - Using link list or array
    - Update the *struct timeval* for next select() call
Project Tip3

- **How could the NWL keep sending packets until blocked by the Data Link Layer?**
  
  Our suggestion is that you could use pipe to implement it: NWL keeps writing packets to the pipe until the pipe is full.

- A simple code of pipe could be found at [http://thor.prohosting.com/~nupshot21/Unix/sockets/node46.shtml](http://thor.prohosting.com/~nupshot21/Unix/sockets/node46.shtml)

- Pipe is more like a socket between local processes.
Concurrent TCP Server Example (fork)

```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 */
}
```

12/1/2004