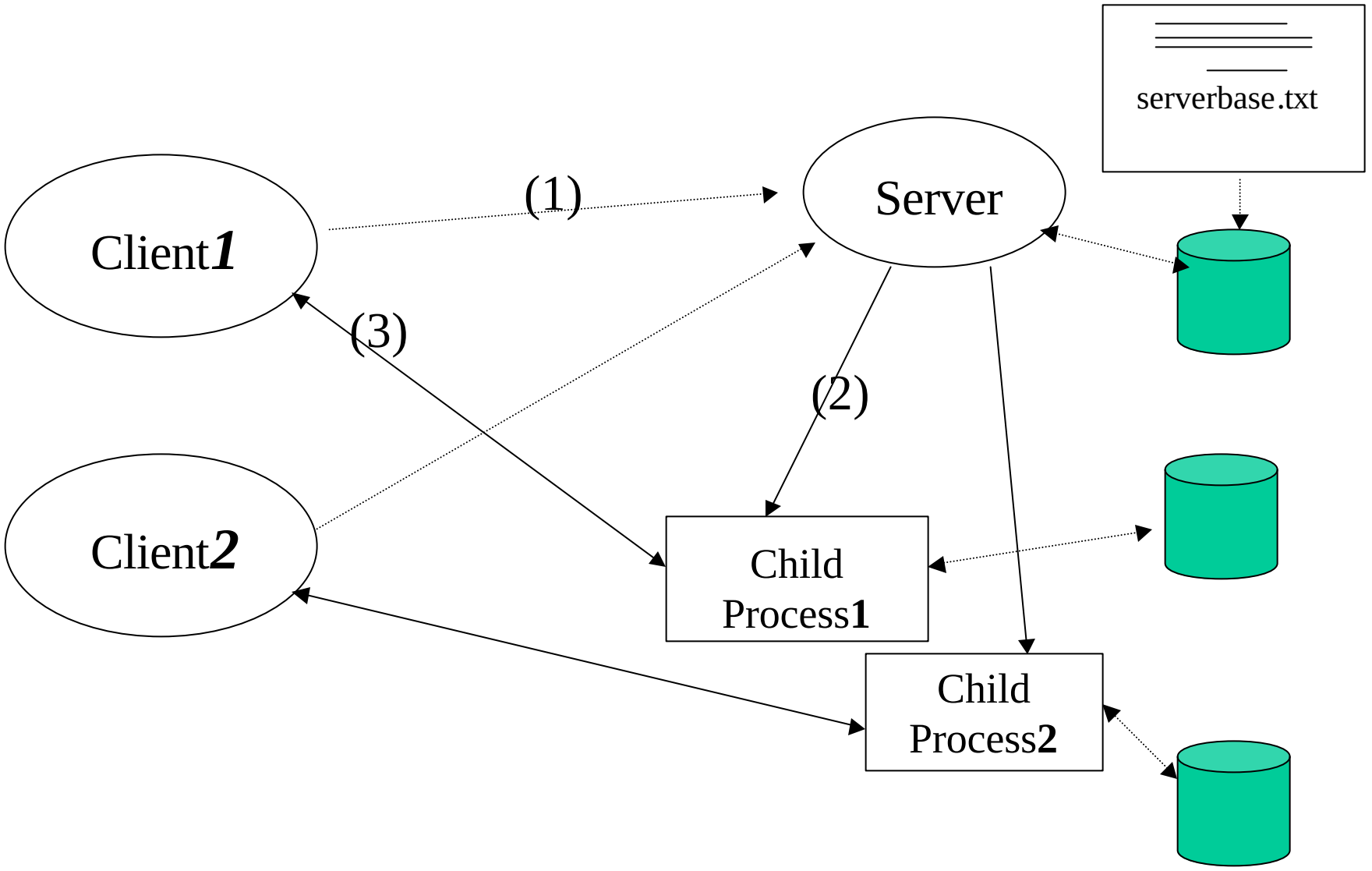
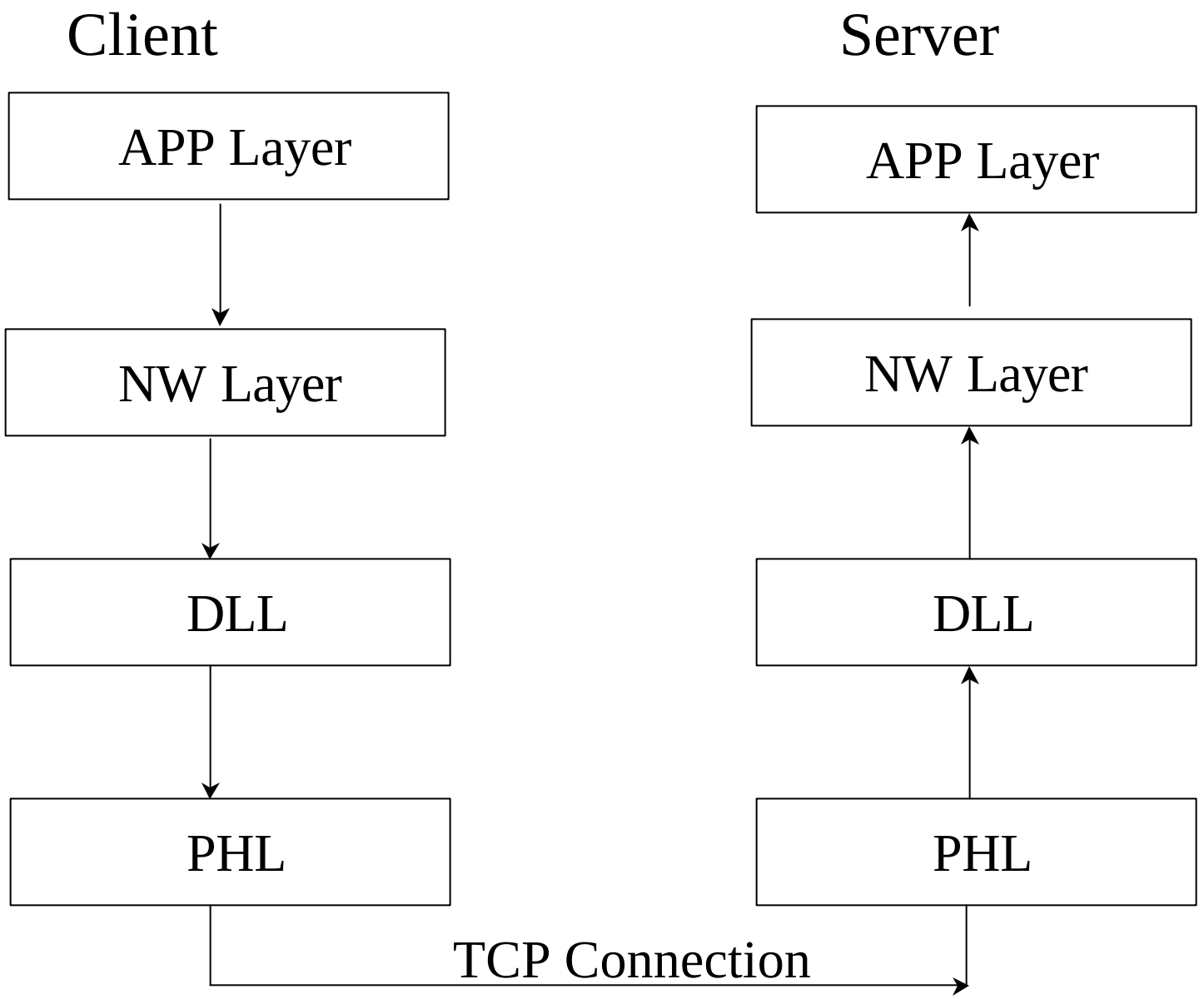
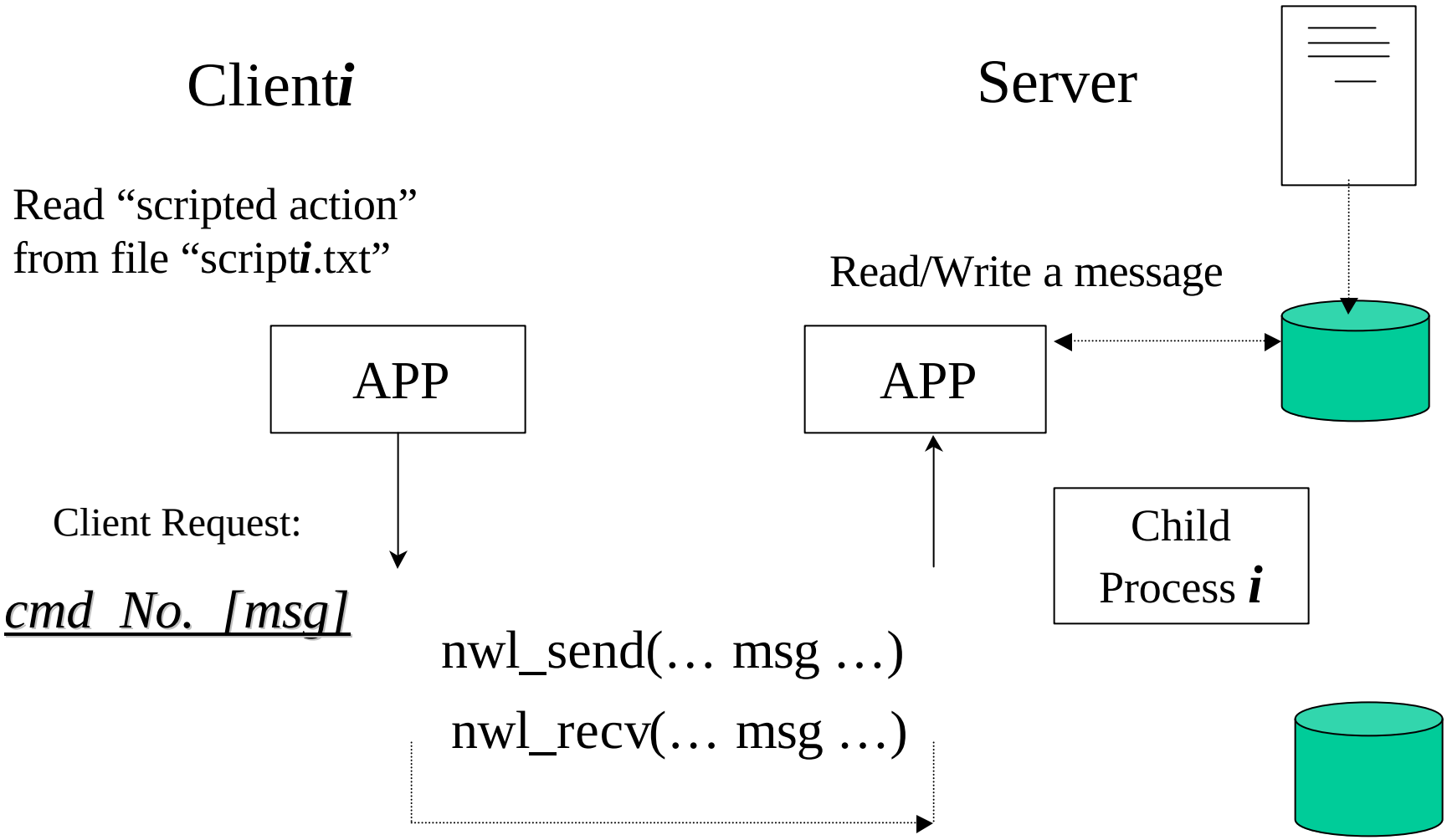
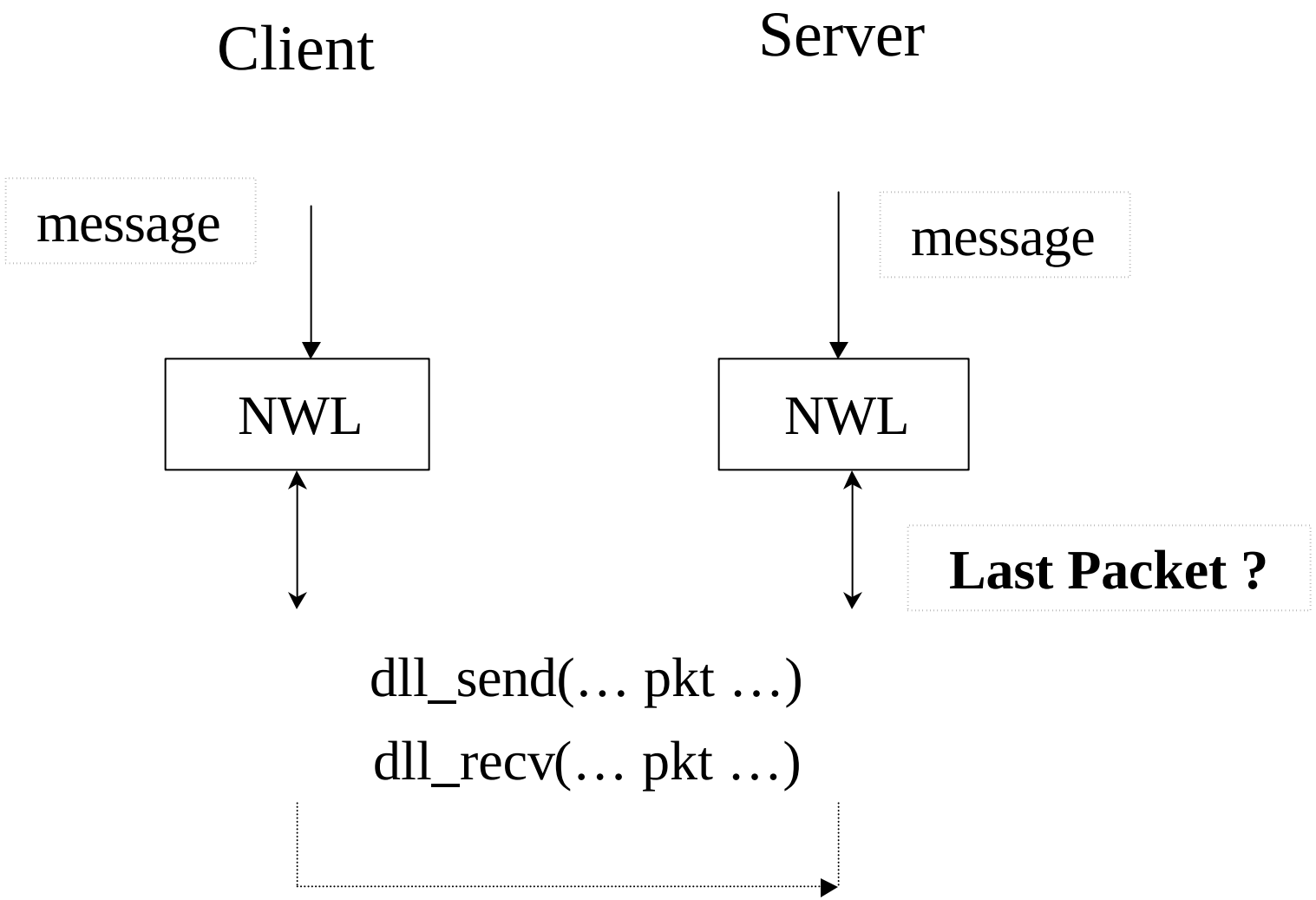


Note: each child process keeps a separate copy of the DB.


Client
Server
APP Layer
APP Layer
NW Layer
NW Layer
DLL
DLL
PHL
PHL
TCP Connection

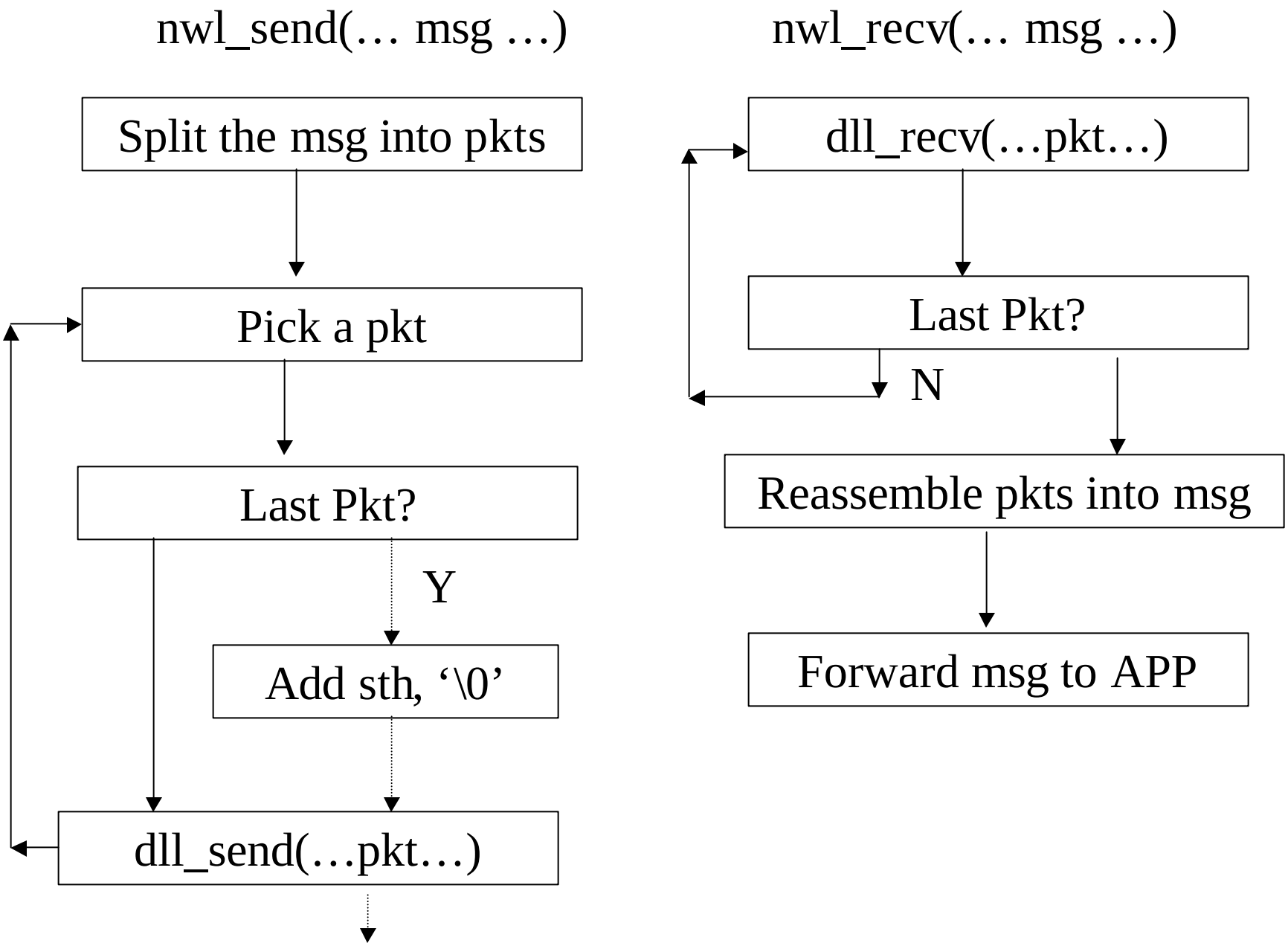


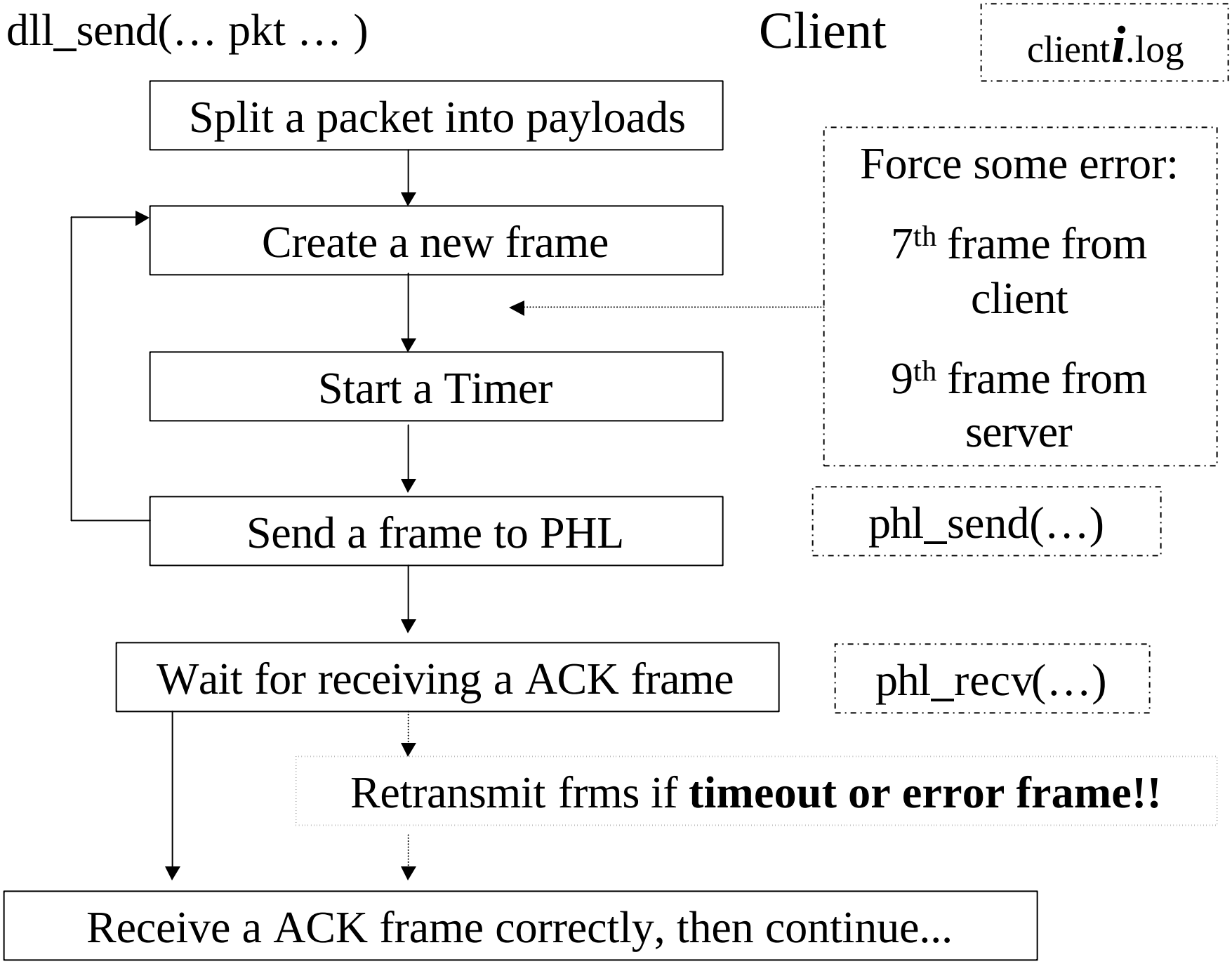
Note: The max_size of a message is 360 bytes



Note: The max_size of a packet is 50 bytes

The network layer will send packets until blocked by the Data Link Layer


nwl_send(… msg …)
Split the msg into pkts
Pick a pkt
Last Pkt?
Y
Add sth, ‘\0’
dll_send(…pkt…)
nwl_recv(… msg …)
dll_recv(…pkt…)
Last Pkt?
N
Reassemble pkts into msg
Forward msg to APP


dll_send(… pkt …)
Client
client*i*.log
Split a packet into payloads
Create a new frame
Start a Timer
Send a frame to PHL
Wait for receiving a ACK frame
Retransmit frms if **timeout or error frame!!**
Receive a ACK frame correctly, then continue...
Force some error:
7th frame from client
9th frame from server
phl_send(…)
phl_recv(…)

Create a new frame

1. Put a payload in a frame

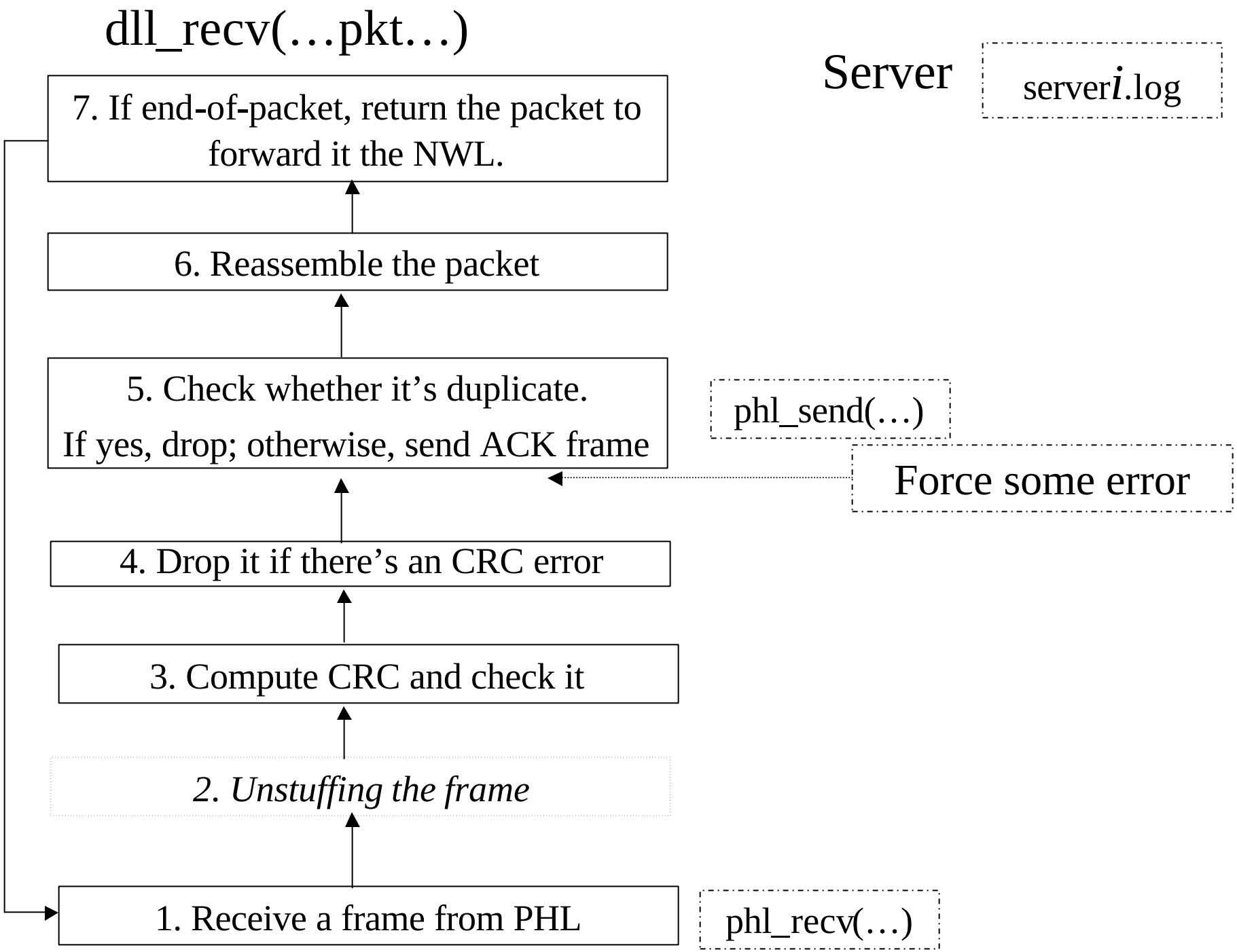
2. End-of-packet byte = ?

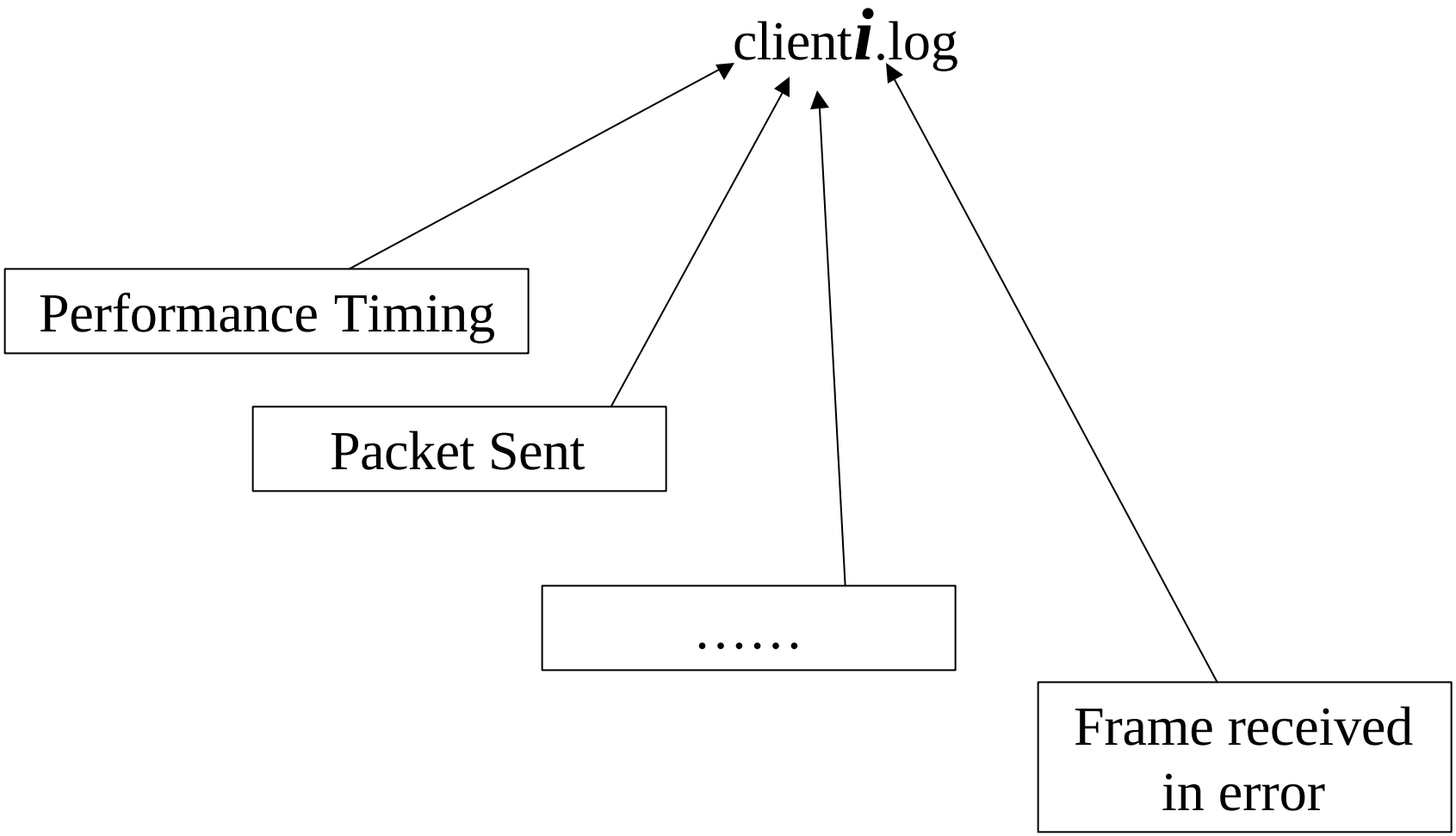
3. Compute CRC

4. Byte-stuffing

```
struct frame {
        unsigned char start-flag;
        unsigned char seq;
        unsigned char crc;
        unsigned char data[MAXSIZE ];
        unsigned char end-of-pkt;
        unsigned char end-flag;
}
```

Note: The max_size of a frame payload is 35 bytes


dll_recv(…pkt…)
Server
server*i*.log
7. If end-of-packet, return the packet to forward it the NWL.
6. Reassemble the packet
5. Check whether it's duplicate.
If yes, drop; otherwise, send ACK frame
phl_send(…)
Force some error
4. Drop it if there's an CRC error
3. Compute CRC and check it
2. Unstuffing the frame
1. Receive a frame from PHL
phl_recv(…)


client*i*.log
Performance Timing
Packet Sent
……
Frame received
in error

More

- Sliding Window Protocol: Go back N
  - Try to implement the simplest first:
    - One-bit sliding window
    - A single timer
  - Then implement a sending window size of
    - 3 or more frames
- How to terminate a client process:
  - When the client gets the response to the **quit** message
  - A “clean” way to terminate the server child process ?

Concurrent Server **Example Code**

```
 pid_t pid;
int listenfd, connfd;
listenfd = socket( ... );
/* fill in sockaddr_in{ } with server's well-known port */
bind (listenfd, ...);
listen (listenfd, LISTENQ);
for(;;){
  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 */
}
```

Timer **Example**

```
/* example for start_timer, stop_timer, send_packet */
/* you MAY need to modify to work for project 3, this is just an example of the TIMER,not your protocol */
#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(int x){
  printf("time out!\n");
```

Timer **Example**(cont)

```
/* you MAY need to modify to work for project 3, this is just an example of the TIMER,not your protocol */
}
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]);
}
```

Timer **Example**(cont)

```
/* you MAY need to modify to work for project 3, this is just an example of the TIMER,not your protocol */
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, resend all frames that haven't been acknowledged
          continue;
         }
         else{
           perror("select error");  /* select error */
           exit(1);
```

Timer **Example**(cont)

```
/* you MAY need to modify to work for project 3, this is just an example of the TIMER,not your protocol */
}
}
  if(FD_ISSET(sockfd, &readfds)){  /* a frame come from socket */
     cksum_function();      /* error check */
}
if(cksum error){
         continue;  // do nothing, wait for timer time out
     }
     else{
       // read the frame from socket, check to see if this frame is a data frame
       // or ack frame, doing corresponding processing
       continue;;
     }
   }
 }
}
```