sudo gcc icmp_capture_detailed.c -o icmp_capture_detailed
sudo ./icmp_capture_detailed

Question 1

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/ip.h>
#include <netinet/in.h>
#include <netinet/ip_icmp.h>
#include <arpa/inet.h>

int main() {
   int sockfd;
   struct ip *ip_header;
   struct icmphdr *icmp_header;
   char buffer[1024];

   // Create a raw socket to capture ICMP packets
   sockfd = socket(AF_INET, SOCK_RAW, IPPROTO_ICMP);
   if (sockfd < 0) {
       perror("Socket creation failed");
       exit(1);
   }

   while (1) {
       ssize_t packet_len = recv(sockfd, buffer, sizeof(buffer), 0);
       if (packet_len < 0) {
           perror("Packet reception error");
           close(sockfd);
           exit(1);
       }

       ip_header = (struct ip *)buffer;
       icmp_header = (struct icmphdr *)(buffer + (ip_header->ip_hl << 2));

       printf("IP Header Length: %d bytes\n", ip_header->ip_hl << 2);
   }

   close(sockfd);
   return 0;
}
```

Question 1 oth

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/ip.h>
#include <netinet/ip_icmp.h>
#include <arpa/inet.h>

void print_ip_header(struct ip *ip_header) {
   printf("IP Header\n");
   printf("  Version: %d\n", ip_header->ip_v);
   printf("  Header Length: %d bytes\n", (ip_header->ip_hl) * 4);
   printf("  Type of Service: %d\n", ip_header->ip_tos);
   printf("  Total Length: %d bytes\n", ntohs(ip_header->ip_len));
   printf("  Identification: %d\n", ntohs(ip_header->ip_id));
   printf("  Fragment Offset: %d\n", ntohs(ip_header->ip_off) & IP_OFFMASK);
```

```
    printf("  Time To Live (TTL): %d\n", ip_header->ip_ttl);
    printf("  Protocol: %d\n", ip_header->ip_p);
    printf("  Checksum: %d\n", ntohs(ip_header->ip_sum));
    printf("  Source IP: %s\n", inet_ntoa(ip_header->ip_src));
    printf("  Destination IP: %s\n", inet_ntoa(ip_header->ip_dst));
}

int main() {
    int sockfd;
    char buffer[1024];

    // Create a raw socket to capture ICMP packets
    sockfd = socket(AF_INET, SOCK_RAW, IPPROTO_ICMP);
    if (sockfd < 0) {
        perror("Socket creation failed");
        exit(1);
    }

    while (1) {
        ssize_t packet_len = recv(sockfd, buffer, sizeof(buffer), 0);
        if (packet_len < 0) {
            perror("Packet reception error");
            close(sockfd);
            exit(1);
        }

        struct ip *ip_header = (struct ip *)buffer;

        printf("\nReceived an ICMP packet:\n");
        print_ip_header(ip_header);

        // Print ICMP payload data (if any)
        int ip_header_len = (ip_header->ip_hl) * 4;
        int payload_len = packet_len - ip_header_len;
        if (payload_len > 0) {
            printf("ICMP Payload Data:\n");
            for (int i = ip_header_len; i < packet_len; i++) {
                printf("%02x ", buffer[i]);
                if ((i - ip_header_len + 1) % 16 == 0)
                    printf("\n");
            }
            printf("\n");
        }
    }

    close(sockfd);
    return 0;
}
```

Question 2 server

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define SERV_PORT 6000

int main(int argc, char **argv) {
```

```
    int listenfd, connfd;
    pid_t childpid;
    socklen_t clilen;
    struct sockaddr_in servaddr, cliaddr;
    char msg1[512];
    ssize_t n1;

    // Create a socket
    listenfd = socket(AF_INET, SOCK_STREAM, 0);
    if (listenfd < 0) {
        perror("Socket creation failed");
        exit(1);
    }

    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port = htons(SERV_PORT);

    // Bind the socket
    if (bind(listenfd, (struct sockaddr *)&servaddr, sizeof(servaddr)) < 0) {
        perror("Bind failed");
        exit(1);
    }

    // Listen for incoming connections
    if (listen(listenfd, 5) < 0) {
        perror("Listen failed");
        exit(1);
    }

    for (;;) {
        clilen = sizeof(cliaddr);
        connfd = accept(listenfd, (struct sockaddr *)&cliaddr, &clilen);
        if (connfd < 0) {
            perror("Accept failed");
            exit(1);
        }

        if ((childpid = fork()) == 0) {
            close(listenfd);
            n1 = read(connfd, msg1, sizeof(msg1));
            write(connfd, msg1, n1);
            close(connfd);
            exit(0);
        }

        close(connfd);
    }

    return 0;
}
```

Question 2 client

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#include <netinet/in.h>
```

```
int main(int argc, char **argv) {
   int sockfd;
   struct sockaddr_in servaddr;
   char sendline[512];
   char recvline[512];
   ssize_t n;

   if (argc != 2) {
       printf("Usage: %s <ipaddress>\n", argv[0]);
       exit(1);
   }

   // Create a socket
   sockfd = socket(AF_INET, SOCK_STREAM, 0);
   if (sockfd < 0) {
       perror("Socket creation failed");
       exit(1);
   }

   bzero(&servaddr, sizeof(servaddr));
   servaddr.sin_family = AF_INET;
   servaddr.sin_port = htons(6000);
   servaddr.sin_addr.s_addr = inet_addr(argv[1]);

   // Connect to the server
   if (connect(sockfd, (struct sockaddr *)&servaddr, sizeof(servaddr)) < 0) {
       perror("Connect failed");
       exit(1);
   }

   printf("Enter a message to send to the server:\n");
   fgets(sendline, sizeof(sendline), stdin);
   n = strlen(sendline);

   // Send the message to the server
   write(sockfd, sendline, n);

   // Receive and print the server's response
   n = read(sockfd, recvline, sizeof(recvline));
   if (n < 0) {
       perror("Error reading from server");
       exit(1);
   }

   recvline[n] = 0;
   printf("Server response: %s\n", recvline);

   close(sockfd);
   return 0;
}
```

Parts

The provided code appears to be C code for a basic TCP server and client. Let's go through each of the steps you mentioned:

Note: I'll address the questions step by step, assuming that the code you provided has been properly initialized and compiled. Make sure to include the necessary headers and declare variables before running the code.

Start Server and see the output of the command netstat -a. What does it show?

When you start the server, it binds to a socket and listens for incoming connections. Running netstat -a will show a list of all network connections and listening ports. If the server is running, you should see an entry like this:

css

Copy code

```
Proto Recv-Q Send-Q Local Address Foreign Address State tcp  0  0 *:6000 *:* LISTEN
```

This indicates that the server is listening on port 6000 for incoming TCP connections.

Now start the client. The client stops in fgets. Check the output of the command netstat -a.
When the client starts and stops at fgets, it means it is waiting for user input. At this point, if you run netstat -a, you won't see any significant change in the output since the client hasn't initiated a connection yet.

Type a line on the keyboard. Check the output of netstat -a after the client terminates.

After you type a line and press Enter in the client, it will establish a connection to the server. You should see a new entry in netstat -a indicating the client's connection to the server. It might look something like this:

css

Copy code

```
Proto Recv-Q Send-Q Local Address Foreign Address State tcp  0  0 your_ip:port server_ip:6000 ESTABLISHED
```

This entry shows an established TCP connection between your client and the server.

Check the output of ps command after the client terminates. How many processes are shown and why?
After the client terminates, the client process will exit, and you will be left with the server process and its child process (if any). You can use the ps command to check the running processes. There will be two processes shown:

- The server process: This is the main server process that continues to listen for incoming connections.
- The server child process: This is a child process created by the server using fork(). It handles communication with the client that connected. Each time a client connects, the server forks a child process to handle that client. This is why you see multiple child processes if multiple clients connect simultaneously.

Now terminate the server. Start the server again and then the client. The client blocks in fgets. Terminate the server child process and study the output of netstat -a.
When you terminate the server, it will close the listening socket, and any existing connections will be closed. If you start the server again and then the client, the client might block in fgets if it's waiting for user input.
When you terminate the server child process (i.e., one of the forked child processes) while the client is blocked in fgets, the client will likely terminate due to a broken connection. You may see a FIN_WAIT2 state in netstat -a for the terminated client connection.

Remember that this code is basic and doesn't handle errors or multiple clients gracefully. It's a single-threaded server that can handle one client at a time. For a production environment, you would typically use more advanced techniques, such as multithreading or asynchronous I/O, to handle multiple clients concurrently and handle errors more robustly.