**Exercise 2: Echo Client Server**

**Server algorithm**

1. Create a socket.
2. Define server address (IP, port).
3. Bind the socket to the server address.
4. Listen for incoming connections.
5. Accept a connection from the client.
6. Receive the message from the client.
7. Send the received message back to the client (echo).
8. Close the client connection.
9. Close the server socket.

**Client algorithm**

1. Create a socket.
2. Define the server address (IP, port).
3. Connect to the server.
4. Enter a message to send.
5. Send the message to the server.
6. Receive the echoed message from the server.
7. Print the echoed message.
8. Close the client socket.

**server.c**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

int main() {

int server\_socket, client\_socket;

struct sockaddr\_in server\_addr, client\_addr;

socklen\_t client\_len;

char buffer[1024];

server\_socket = socket(AF\_INET, SOCK\_STREAM, 0);

server\_addr.sin\_family = AF\_INET;

server\_addr.sin\_addr.s\_addr = INADDR\_ANY;

server\_addr.sin\_port = htons(8080);

bind(server\_socket, (struct sockaddr \*)&server\_addr, sizeof(server\_addr));

listen(server\_socket, 1);

client\_len = sizeof(client\_addr);

client\_socket = accept(server\_socket, (struct sockaddr \*)&client\_addr, &client\_len);

recv(client\_socket, buffer, sizeof(buffer), 0);

send(client\_socket, buffer, strlen(buffer), 0);

close(client\_socket);

close(server\_socket);

return 0;

}

**client.c**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

int main() {

int client\_socket;

struct sockaddr\_in server\_addr;

char buffer[1024];

client\_socket = socket(AF\_INET, SOCK\_STREAM, 0);

server\_addr.sin\_family = AF\_INET;

server\_addr.sin\_port = htons(8080);

server\_addr.sin\_addr.s\_addr = inet\_addr("127.0.0.1");

connect(client\_socket, (struct sockaddr \*)&server\_addr, sizeof(server\_addr));

printf("Enter message: ");

fgets(buffer, sizeof(buffer), stdin);

send(client\_socket, buffer, strlen(buffer), 0);

recv(client\_socket, buffer, sizeof(buffer), 0);

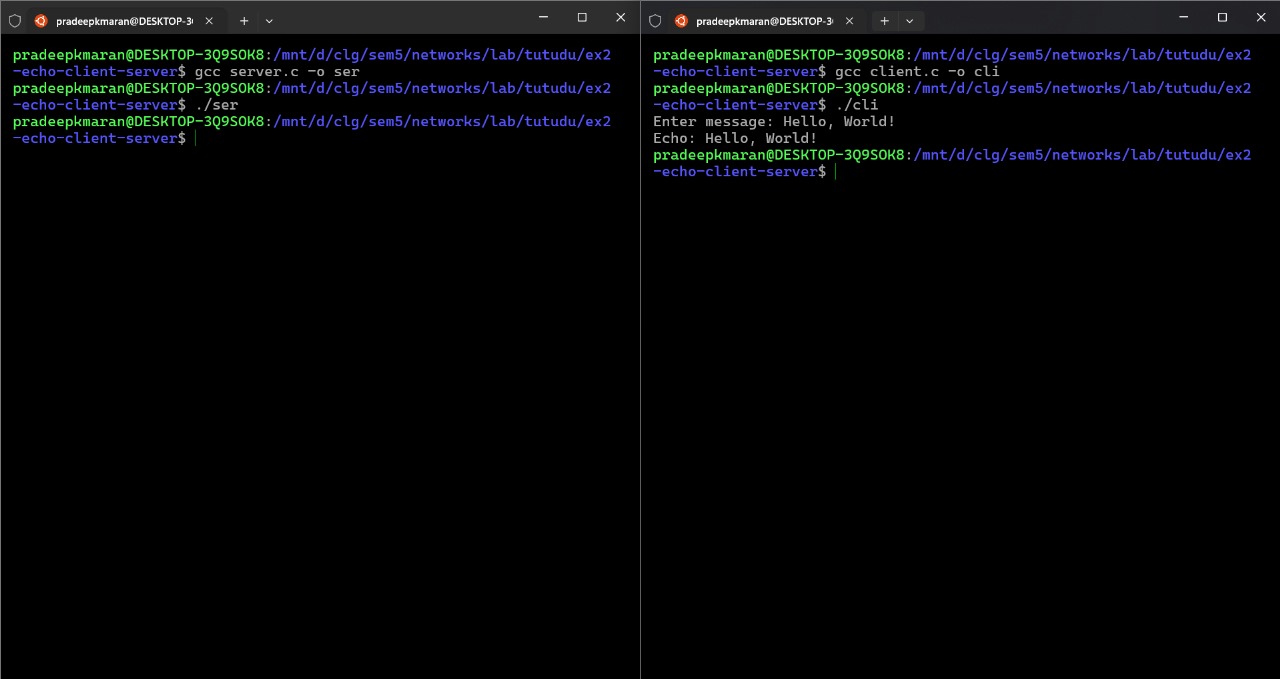
printf("Echo: %s", buffer);

close(client\_socket);

return 0;

}

**Output**



**Exercise 3: File Transfer**

**Server algorithm**

1. Create socket.
2. Bind socket to port.
3. Listen for client connection.
4. Accept client connection.
5. Receive filename from client.
6. Open file for reading.
7. Send file data in chunks.
8. Close file and socket.

**Client algorithm**

1. Create socket.
2. Connect to server.
3. Input and send filename.
4. Receive file data.
5. Write data to new file.
6. Close socket.

**server.c**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <sys/socket.h>

#include <netinet/in.h>

#define PORT 8080

#define BUFSIZE 1024

int main() {

int server\_fd, client\_fd;

struct sockaddr\_in server\_addr, client\_addr;

socklen\_t client\_len = sizeof(client\_addr);

char filename[256];

char buffer[BUFSIZE];

FILE \*file;

size\_t bytes\_read;

server\_fd = socket(AF\_INET, SOCK\_STREAM, 0);

server\_addr.sin\_family = AF\_INET;

server\_addr.sin\_addr.s\_addr = INADDR\_ANY;

server\_addr.sin\_port = htons(PORT);

bind(server\_fd, (struct sockaddr\*)&server\_addr, sizeof(server\_addr));

listen(server\_fd, 1);

client\_fd = accept(server\_fd, (struct sockaddr\*)&client\_addr, &client\_len);

read(client\_fd, filename, sizeof(filename));

file = fopen(filename, "rb");

if (file) {

while ((bytes\_read = fread(buffer, 1, BUFSIZE, file)) > 0) {

write(client\_fd, buffer, bytes\_read);

}

fclose(file);

}

close(client\_fd);

close(server\_fd);

return 0;

}

**client.c**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#define PORT 8080

#define BUFSIZE 1024

int main() {

    int sockfd;

    struct sockaddr\_in server\_addr;

    char filename[256];

    char buffer[BUFSIZE];

    FILE \*file;

    size\_t bytes\_read;

    sockfd = socket(AF\_INET, SOCK\_STREAM, 0);

    server\_addr.sin\_family = AF\_INET;

    server\_addr.sin\_port = htons(PORT);

    server\_addr.sin\_addr.s\_addr = inet\_addr("127.0.0.1");

    connect(sockfd, (struct sockaddr\*)&server\_addr, sizeof(server\_addr));

    printf("Enter filename: ");

    scanf("%s", filename);

    write(sockfd, filename, strlen(filename) + 1);

    strcat(filename, "\_copy");

    file = fopen("received\_file", "wb");

    while ((bytes\_read = read(sockfd, buffer, BUFSIZE)) > 0) {

        fwrite(buffer, 1, bytes\_read, file);

    }

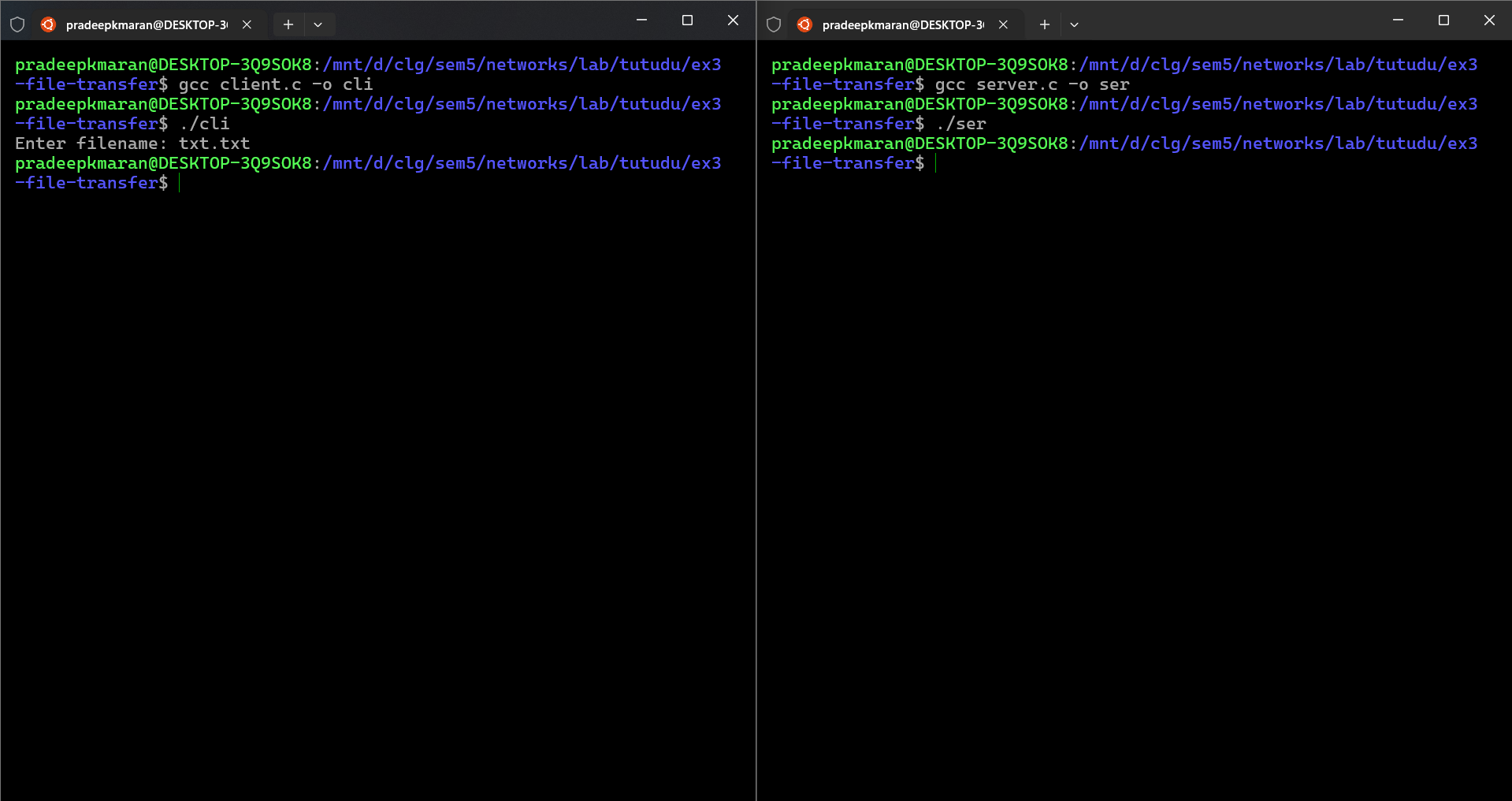
    fclose(file);

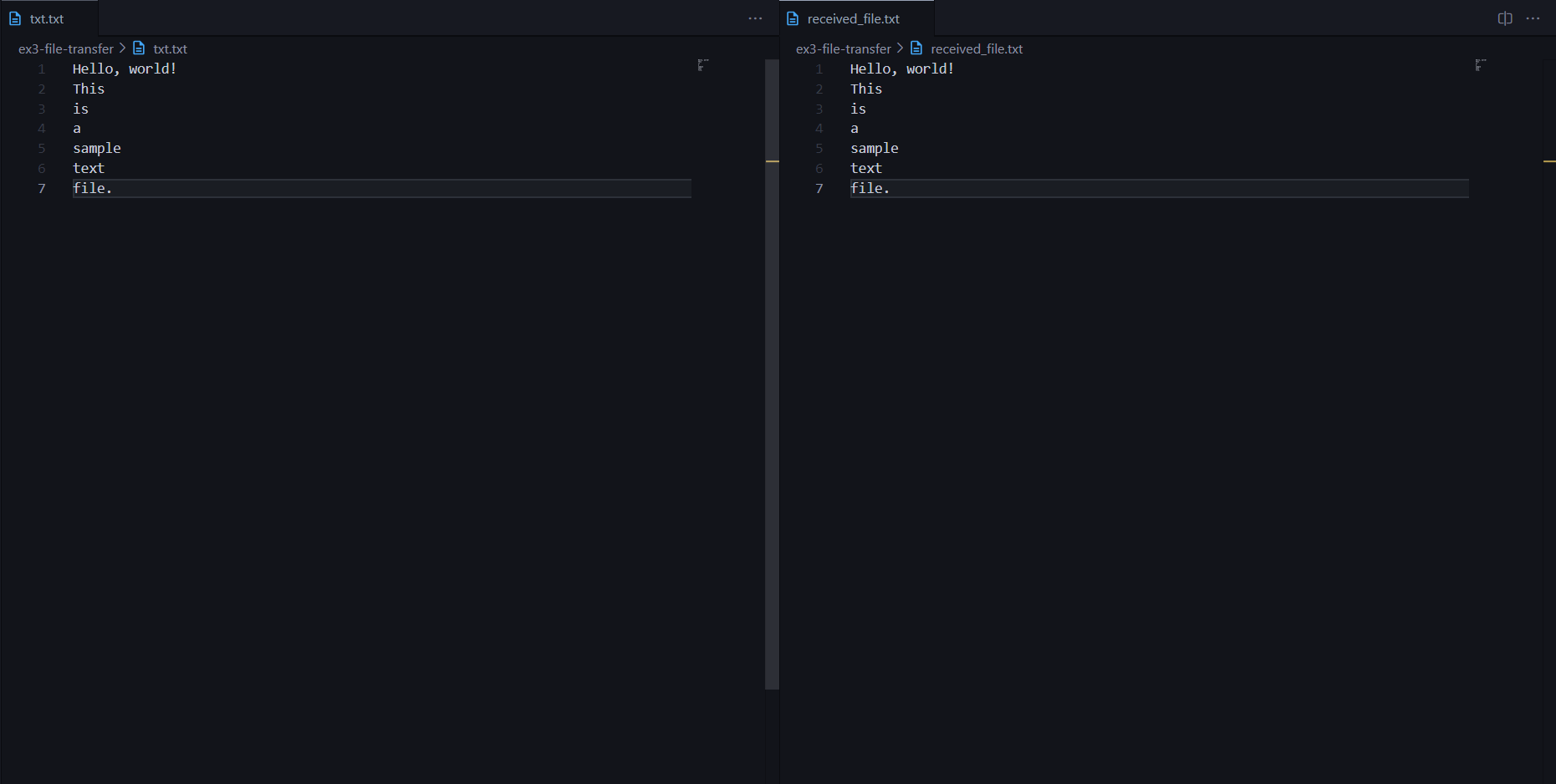
    close(sockfd);

    return 0;

}

**Output**





**Exercise 4: Chat using TCP**

**Server algorithm**

1. Create a socket.
2. Bind the socket to a port.
3. Listen for incoming connections.
4. Accept client connection.
5. Fork a child process.
6. In child process:
7. Read data from client.
   1. Send a reply to client.
   2. Close client socket.
8. Repeat for new clients.

**Client algorithm**

1. Create a socket.
2. Connect to server.
3. Send message to server.
4. Receive reply from server.
5. Display server reply.
6. Repeat until exit.
7. Close socket.

**server.c**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <arpa/inet.h>

#include <sys/types.h>

#include <sys/socket.h>

#define PORT 8080

#define MAX\_CLIENTS 10

#define BUF\_SIZE 1024

void handle\_client(int client\_socket) {

    char buffer[BUF\_SIZE];

    int n;

    while ((n = read(client\_socket, buffer, sizeof(buffer))) > 0) {

        buffer[n] = '\0';

        printf("Received from client: %s\n", buffer);

        char user\_input[BUF\_SIZE];

        printf("Send reply: ");

        fgets(user\_input, BUF\_SIZE-1, stdin);

        user\_input[strcspn(user\_input, "\n")] = '\0';

        write(client\_socket, user\_input, strlen(user\_input));

    }

    close(client\_socket);

}

int main() {

    int server\_socket, client\_socket, client\_len;

    struct sockaddr\_in server\_addr, client\_addr;

    pid\_t child\_pid;

    server\_socket = socket(AF\_INET, SOCK\_STREAM, 0);

    server\_addr.sin\_family = AF\_INET;

    server\_addr.sin\_addr.s\_addr = INADDR\_ANY;

    server\_addr.sin\_port = htons(PORT);

    bind(server\_socket, (struct sockaddr\*)&server\_addr, sizeof(server\_addr));

    listen(server\_socket, MAX\_CLIENTS);

    while (1) {

        client\_len = sizeof(client\_addr);

        client\_socket = accept(server\_socket, (struct sockaddr\*)&client\_addr, &client\_len);

        if ((child\_pid = fork()) == 0) {

            close(server\_socket);

            handle\_client(client\_socket);

            exit(0);

        } else {

            close(client\_socket);

        }

    }

    return 0;

}

**client.c**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <arpa/inet.h>

#define PORT 8080

#define BUF\_SIZE 1024

int main() {

    int sock;

    struct sockaddr\_in server\_addr;

    char buffer[BUF\_SIZE];

    ssize\_t n;

    sock = socket(AF\_INET, SOCK\_STREAM, 0);

    server\_addr.sin\_family = AF\_INET;

    server\_addr.sin\_port = htons(PORT);

    server\_addr.sin\_addr.s\_addr = inet\_addr("127.0.0.1");

    connect(sock, (struct sockaddr\*)&server\_addr, sizeof(server\_addr));

    while (1) {

        printf("Enter message: ");

        fgets(buffer, sizeof(buffer), stdin);

        write(sock, buffer, strlen(buffer));

        n = read(sock, buffer, sizeof(buffer));

        buffer[n] = '\0';

        printf("Server: %s\n", buffer);

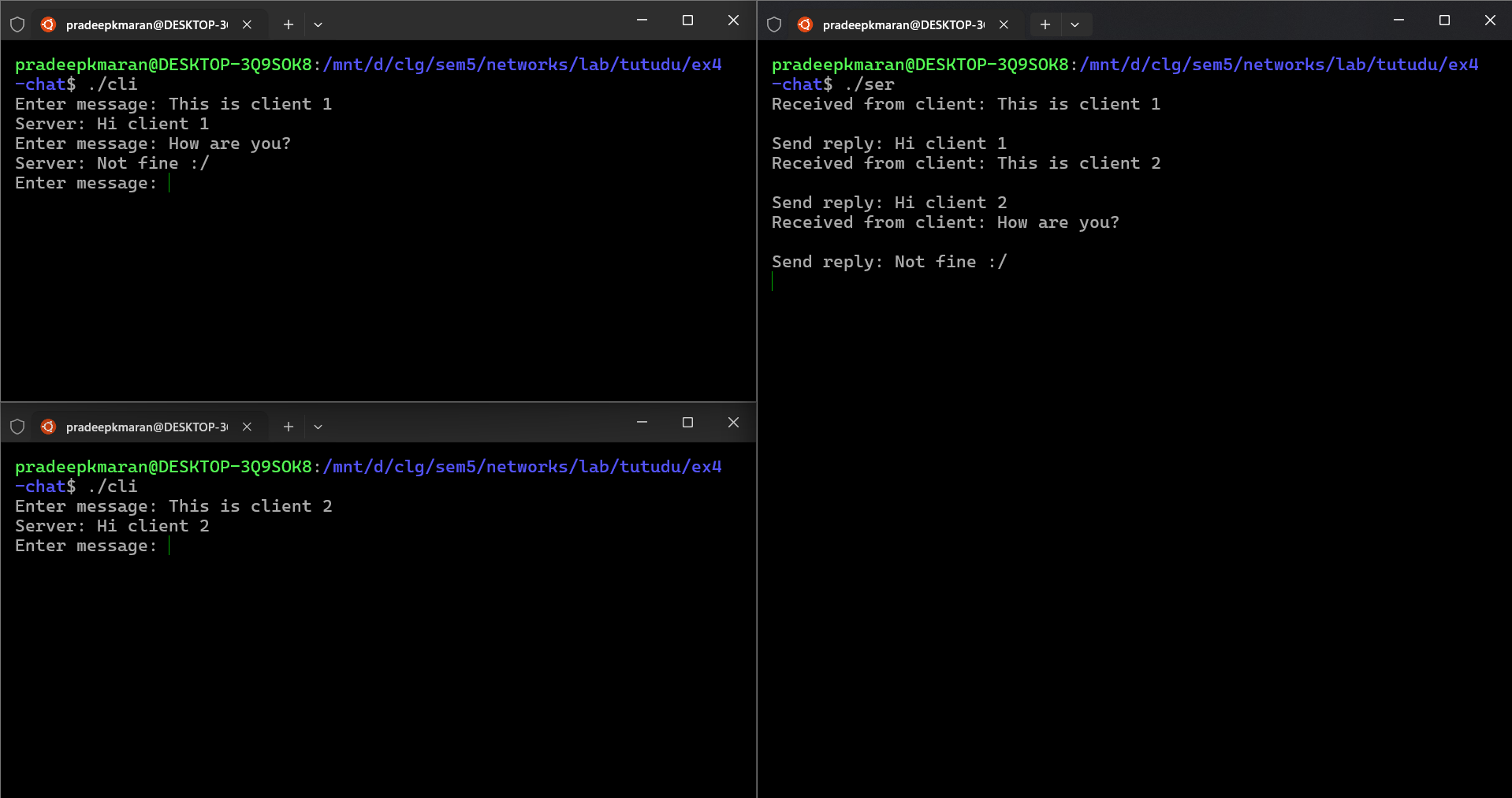
    }

    close(sock);

    return 0;

}

**Output**



**Exercise 5: ARP and RARP**

**ARP**

**Server algorithm**

1. Create UDP socket for broadcasting.
2. Set socket option for broadcasting.
3. Broadcast message with source IP, source MAC, and destination IP.
4. Wait for incoming UDP message.
5. If destination IP matches, create TCP socket.
6. Connect to the client via TCP.
7. Send data (ARP reply) to the client via TCP.
8. Close TCP and UDP sockets.

**Client algorithm**

1. Create UDP socket to listen for broadcast.
2. Wait for UDP broadcast message.
3. Extract source IP, source MAC, and destination IP from the message.
4. If destination IP matches, create TCP socket.
5. Connect to the server via TCP.
6. Receive ARP reply from server.
7. Close the TCP and UDP sockets.

**server.c**

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#include <unistd.h>

#include <arpa/inet.h>

#define BROADCAST\_IP "255.255.255.255"

#define BROADCAST\_PORT 8888

#define MESSAGE "This is a broadcast message!"

typedef struct {

    char src\_ip[16];

    char src\_mac[18];

    char dest\_ip[16];

    char dest\_mac[18];

    char data[17];

} Packet;

int main() {

    int sockfd;

    struct sockaddr\_in broadcast\_addr;

    int broadcast\_enable = 1;

    sockfd = socket(AF\_INET, SOCK\_DGRAM, 0);

    setsockopt(sockfd, SOL\_SOCKET, SO\_BROADCAST, &broadcast\_enable, sizeof(broadcast\_enable));

    memset(&broadcast\_addr, 0, sizeof(broadcast\_addr));

    broadcast\_addr.sin\_family = AF\_INET;

    broadcast\_addr.sin\_port = htons(BROADCAST\_PORT);

    broadcast\_addr.sin\_addr.s\_addr = inet\_addr(BROADCAST\_IP);

    Packet packet;

    printf("Enter the details of packet received.\n");

    printf("Destination IP: ");

    scanf("%s", packet.dest\_ip);

    printf("Source IP: ");

    scanf("%s", packet.src\_ip);

    printf("Source MAC: ");

    scanf("%s", packet.src\_mac);

    printf("16-bit data: ");

    scanf("%s", packet.data);

    char msg[1000];

    strcpy(msg, packet.src\_ip);

    strcat(msg, "|");

    strcat(msg, packet.src\_mac);

    strcat(msg, "|");

    strcat(msg, packet.dest\_ip);

    strcat(msg, "|");

    sendto(sockfd, msg, strlen(msg), 0, (struct sockaddr \*)&broadcast\_addr, sizeof(broadcast\_addr));

    printf("Broadcast message sent successfully!\n");

    int len;

    int sockfd1, newfd, n;

    struct sockaddr\_in servaddr, cliaddr;

    char buff[1024];

    char str[1000];

    sockfd1 = socket(AF\_INET, SOCK\_STREAM, 0);

    bzero(&servaddr, sizeof(servaddr));

    servaddr.sin\_family = AF\_INET;

    servaddr.sin\_addr.s\_addr = INADDR\_ANY;

    servaddr.sin\_port = htons(7228);

    bind(sockfd1, (struct sockaddr \*)&servaddr, sizeof(servaddr));

    listen(sockfd1, 2);

    len = sizeof(cliaddr);

    newfd = accept(sockfd1, (struct sockaddr \*)&cliaddr, &len);

    n = read(newfd, buff, sizeof(buff));

    printf("\nMessage from Client: %s\n", buff);

    char newstr[1000];

    strcpy(newstr, buff);

    strcat(newstr, packet.data);

    printf("\nMessage Sent: %s\n", newstr);

    n = write(newfd, newstr, sizeof(newstr));

    close(sockfd1);

    close(newfd);

    close(sockfd);

    return 0;

}

**client.c**

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#include <unistd.h>

#include <arpa/inet.h>

#define LISTEN\_PORT 8888

#define BUFFER\_SIZE 1024

#define PORT 8888

int main() {

    int sockfd;

    struct sockaddr\_in recv\_addr, cliaddr;

    char buffer[BUFFER\_SIZE];

    socklen\_t addr\_len = sizeof(recv\_addr);

    char client\_ip[16], client\_mac[18];

    sockfd = socket(AF\_INET, SOCK\_DGRAM, 0);

    memset(&cliaddr, 0, sizeof(cliaddr));

    cliaddr.sin\_family = AF\_INET;

    cliaddr.sin\_addr.s\_addr = INADDR\_ANY;

    cliaddr.sin\_port = htons(PORT);

    memset(&recv\_addr, 0, sizeof(recv\_addr));

    recv\_addr.sin\_family = AF\_INET;

    recv\_addr.sin\_port = htons(LISTEN\_PORT);

    recv\_addr.sin\_addr.s\_addr = INADDR\_ANY;

    bind(sockfd, (struct sockaddr \*)&recv\_addr, sizeof(recv\_addr));

    printf("Listening for broadcast messages on port %d...\n", LISTEN\_PORT);

    printf("Enter the IP address: ");

    scanf("%s", client\_ip);

    printf("Enter the MAC address: ");

    scanf("%s", client\_mac);

    char src\_ip[16], src\_mac[18], dest\_ip[16];

    while (1) {

        int recv\_len = recvfrom(sockfd, buffer, BUFFER\_SIZE, 0, (struct sockaddr \*)&recv\_addr, &addr\_len);

        if (recv\_len > 0) {

            buffer[recv\_len] = '\0';

            printf("\nReceived broadcast message: %s\n", buffer);

            sscanf(buffer, "%[^|]|%[^|]|%[^|]", src\_ip, src\_mac, dest\_ip);

            if (strcmp(dest\_ip, client\_ip) == 0) {

                printf("IP address match\n");

                int len;

                int sockfd1, n, newfd;

                struct sockaddr\_in servaddr;

                char str[1000];

                char buff[1024];

                char newbuff[1024];

                sockfd1 = socket(AF\_INET, SOCK\_STREAM, 0);

                if (sockfd1 < 0)

                    perror("\nCannot create socket\n");

                bzero(&servaddr, sizeof(servaddr));

                servaddr.sin\_family = AF\_INET;

                servaddr.sin\_addr.s\_addr = inet\_addr(src\_ip);

                servaddr.sin\_port = htons(7228);

                connect(sockfd1, (struct sockaddr \*)&servaddr, sizeof(servaddr));

                snprintf(buffer, sizeof(buffer), "%s|%s|%s|%s|", src\_ip, src\_mac, dest\_ip, client\_mac);

                n = write(sockfd1, buffer, sizeof(buffer));

                printf("\nARP Reply Sent: %s\n", buffer);

                n = read(sockfd1, newbuff, sizeof(newbuff));

                printf("\nReceived packet is: %s \n", newbuff);

                close(sockfd1);

                close(newfd);

            } else {

                printf("IP address not matched\n");

            }

            break;

        } else {

            break;

        }

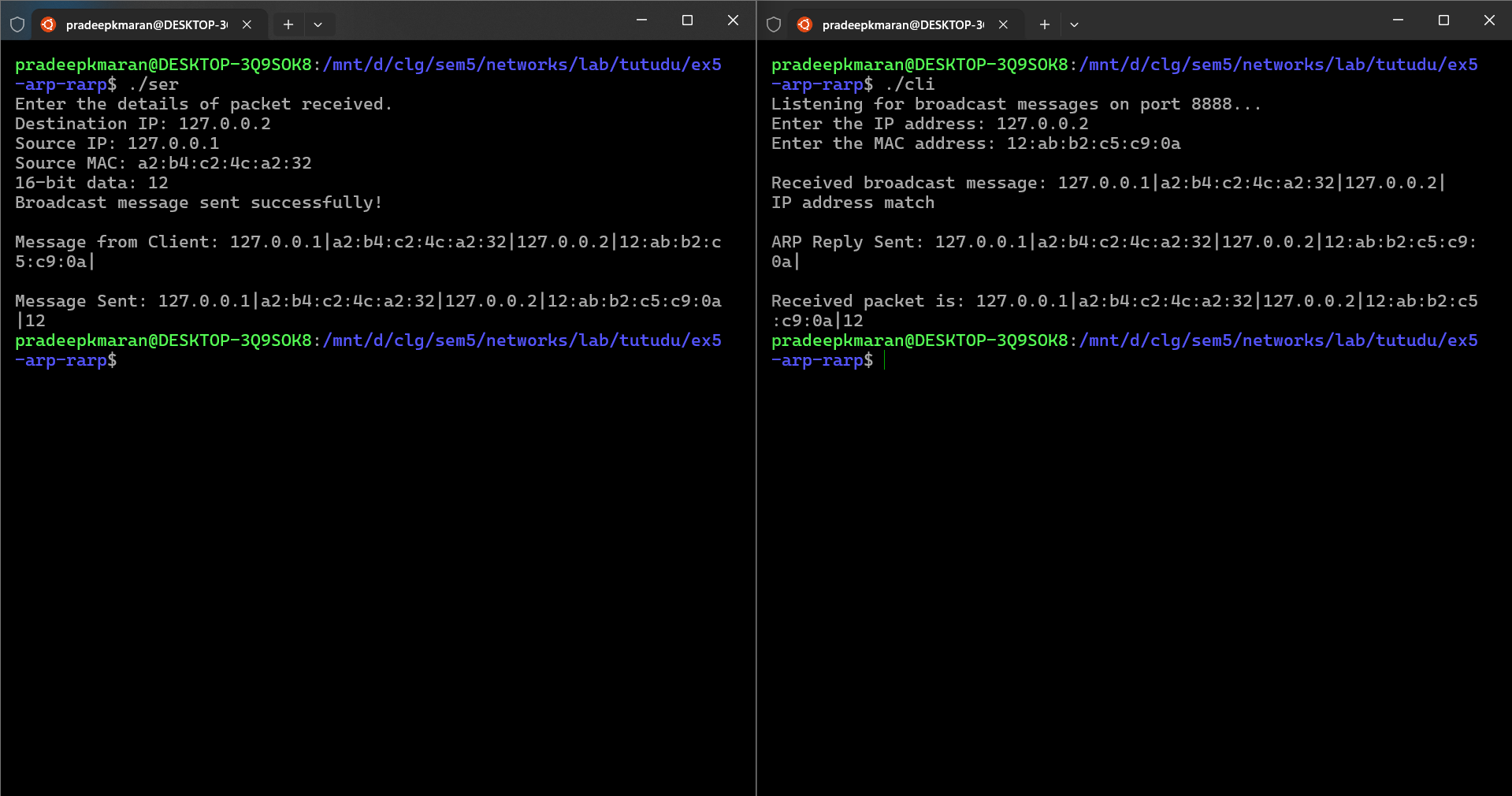
    }

    close(sockfd);

    return 0;

}

**Output**



**RARP**

**Server algorithm**

1. Create UDP socket.
2. Bind to UDP port.
3. Wait for incoming message.
4. Receive MAC address from client.
5. Check MAC in the IP mapping.
6. If MAC found, prepare corresponding IP.
7. Send IP back to the client.
8. If MAC not found, send default IP.

**Client algorithm**

1. Create UDP socket.
2. Prepare MAC address to send.
3. Send MAC address to the server.
4. Wait for the server's response.
5. Receive IP address from server.
6. Display the received IP.
7. Close the socket.

**server.c**

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#include <unistd.h>

#include <arpa/inet.h>

#define UDP\_PORT 8888

#define BUFFER\_SIZE 1024

typedef struct {

    char mac[18];

    char ip[16];

} Mac\_Ip\_Map;

Mac\_Ip\_Map mappings[] = {

    {"00:11:22:33:44:55", "127.0.0.2"},

    {"11:22:33:44:55:66", "127.0.0.3"},

    {"22:33:44:55:66:77", "127.0.0.4"},

    {"33:44:55:66:77:88", "127.0.0.5"}

};

int find\_ip\_for\_mac(const char mac[]) {

    for (int i = 0; i < 4; i++) {

        if (strcmp(mac, mappings[i].mac) == 0) {

            return i;

        }

    }

    return -1;

}

int main() {

    int udp\_sock, tcp\_sock, client\_sock;

    struct sockaddr\_in server\_addr, client\_addr;

    socklen\_t addr\_len = sizeof(client\_addr);

    char client\_mac[18];

    char client\_ip[16];

    udp\_sock = socket(AF\_INET, SOCK\_DGRAM, 0);

    memset(&server\_addr, 0, sizeof(server\_addr));

    server\_addr.sin\_family = AF\_INET;

    server\_addr.sin\_addr.s\_addr = INADDR\_ANY;

    server\_addr.sin\_port = htons(UDP\_PORT);

    bind(udp\_sock, (struct sockaddr\*)&server\_addr, sizeof(server\_addr));

    recvfrom(udp\_sock, client\_mac, sizeof(client\_mac), MSG\_WAITALL, (struct sockaddr\*) &client\_addr, &addr\_len);

    client\_mac[strlen(client\_mac)] = '\0';

    printf("Received MAC: %s\n", client\_mac);

    int index = find\_ip\_for\_mac(client\_mac);

    if (index != -1) {

        strcpy(client\_ip, mappings[index].ip);

    } else {

        strcpy(client\_ip, "0.0.0.0");

    }

    sendto(udp\_sock, client\_ip, strlen(client\_ip) + 1, 0, (struct sockaddr\*)&client\_addr, addr\_len);

    close(udp\_sock);

    return 0;

}

**client.c**

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#include <unistd.h>

#include <arpa/inet.h>

#define UDP\_PORT 8888

#define BUFFER\_SIZE 1024

int main() {

    int udp\_sock;

    struct sockaddr\_in server\_addr;

    char client\_mac[18] = "00:11:22:33:44:55";

    char server\_ip[16];

    int addr\_len = sizeof(server\_addr);

    udp\_sock = socket(AF\_INET, SOCK\_DGRAM, 0);

    memset(&server\_addr, 0, addr\_len);

    server\_addr.sin\_family = AF\_INET;

    server\_addr.sin\_port = htons(UDP\_PORT);

    server\_addr.sin\_addr.s\_addr = inet\_addr("127.0.0.1");

    sendto(udp\_sock, client\_mac, strlen(client\_mac) + 1, MSG\_CONFIRM, (struct sockaddr\*)&server\_addr, addr\_len);

    recvfrom(udp\_sock, server\_ip, sizeof(server\_ip), MSG\_WAITALL, (struct sockaddr\*)&server\_addr, &addr\_len);

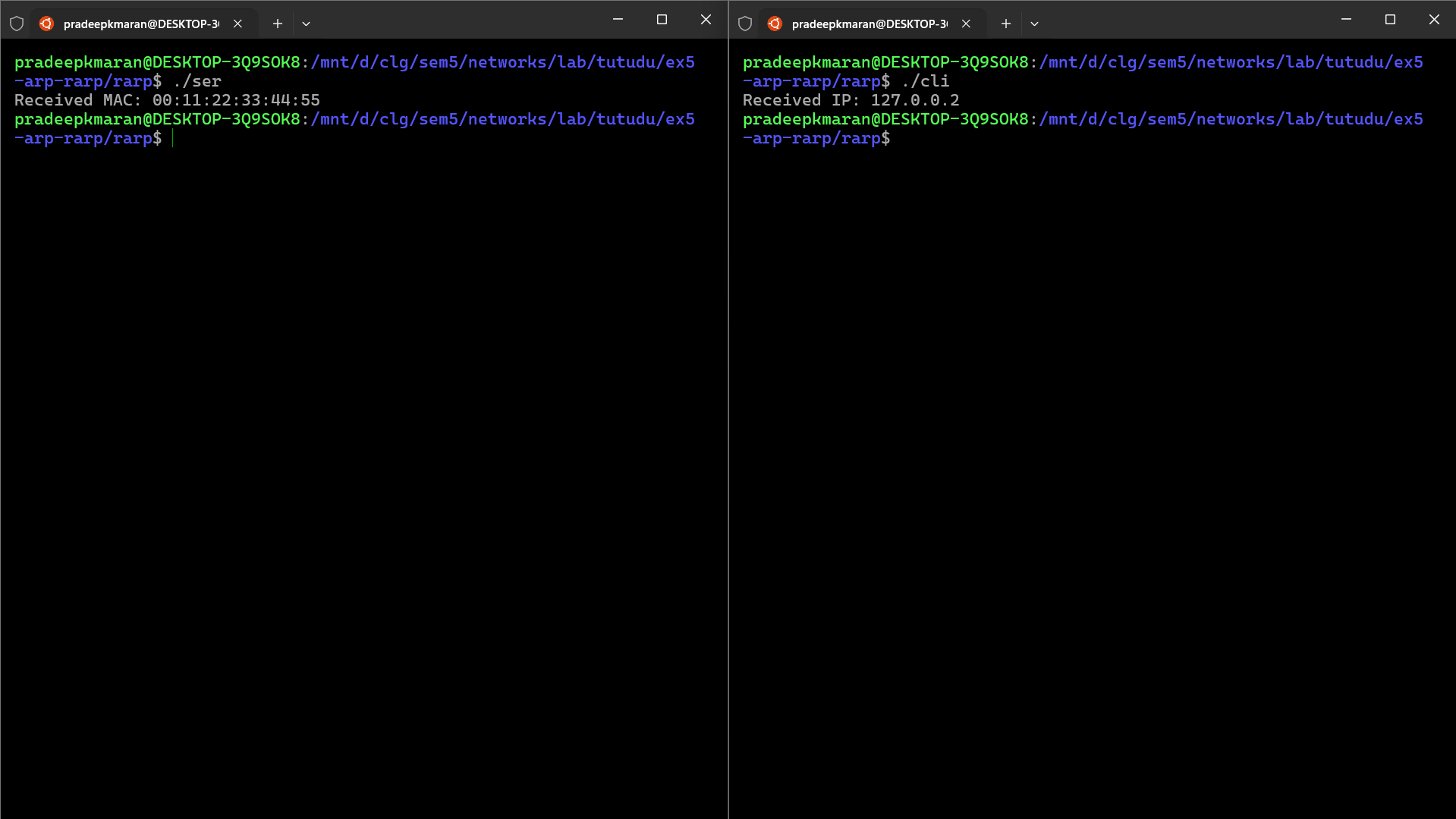
    printf("Received IP: %s\n", server\_ip);

    close(udp\_sock);

    return 0;

}

**Output**



**Exercise 6 : Domain Name Server using UDP**

**Server algorithm**

1. Create a UDP socket.
2. Bind the socket to the DNS port (5353).
3. Enter an infinite loop to continuously listen for incoming requests.
4. Receive the domain name from the client via UDP.
5. Search the domain name in the local DNS table.
6. If found, prepare the corresponding IP address.
7. If not found, prepare a default IP ("0.0.0.0").
8. Print the received domain and the corresponding IP address.
9. Send the IP address back to the client via UDP.
10. Repeat from step 4.

**Client algorithm**

1. Create a UDP socket.
2. Prepare the domain name to be queried (e.g., "www.google.com").
3. Send the domain name to the server via UDP.
4. Wait to receive the IP address from the server.
5. Display the received IP address.
6. Close the socket.

**server.c**

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#include <unistd.h>

#include <arpa/inet.h>

#define DNS\_PORT 5353

#define BUFFER\_SIZE 1024

typedef struct {

    char domain[256];

    char ip[16];

} Dns\_Table;

Dns\_Table dns\_table[] = {

    {"www.google.com", "142.250.190.46"},

    {"www.example.com", "93.184.216.34"},

    {"www.facebook.com", "157.240.7.35"},

    {"www.github.com", "140.82.112.3"}

};

int find\_ip\_for\_domain(const char \*domain) {

    for (int i = 0; i < 4; i++) {

        if (strcmp(domain, dns\_table[i].domain) == 0) {

            return i;

        }

    }

    return -1;

}

int main() {

    int udp\_sock;

    struct sockaddr\_in server\_addr, client\_addr;

    socklen\_t addr\_len = sizeof(client\_addr);

    char buffer[BUFFER\_SIZE];

    char domain[256];

    char ip[16];

    udp\_sock = socket(AF\_INET, SOCK\_DGRAM, 0);

    memset(&server\_addr, 0, sizeof(server\_addr));

    server\_addr.sin\_family = AF\_INET;

    server\_addr.sin\_addr.s\_addr = INADDR\_ANY;

    server\_addr.sin\_port = htons(DNS\_PORT);

    bind(udp\_sock, (struct sockaddr\*)&server\_addr, sizeof(server\_addr));

    while (1) {

        recvfrom(udp\_sock, buffer, sizeof(buffer), MSG\_WAITALL, (struct sockaddr\*)&client\_addr, &addr\_len);

        sscanf(buffer, "%s", domain);

        int index = find\_ip\_for\_domain(domain);

        if (index != -1) {

            strcpy(ip, dns\_table[index].ip);

        } else {

            strcpy(ip, "0.0.0.0");

        }

        printf("Received request for domain: %s, responding with IP: %s\n", domain, ip);

        sendto(udp\_sock, ip, strlen(ip) + 1, 0, (struct sockaddr\*)&client\_addr, addr\_len);

    }

    close(udp\_sock);

    return 0;

}

**client.c**

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#include <unistd.h>

#include <arpa/inet.h>

#define DNS\_PORT 5353

#define BUFFER\_SIZE 1024

int main() {

    int udp\_sock;

    struct sockaddr\_in server\_addr;

    char domain[256] = "www.google.com";

    char ip[16];

    socklen\_t addr\_len = sizeof(server\_addr);

    udp\_sock = socket(AF\_INET, SOCK\_DGRAM, 0);

    memset(&server\_addr, 0, addr\_len);

    server\_addr.sin\_family = AF\_INET;

    server\_addr.sin\_port = htons(DNS\_PORT);

    server\_addr.sin\_addr.s\_addr = inet\_addr("127.0.0.1");

    sendto(udp\_sock, domain, strlen(domain) + 1, MSG\_CONFIRM, (struct sockaddr\*)&server\_addr, addr\_len);

    recvfrom(udp\_sock, ip, sizeof(ip), MSG\_WAITALL, (struct sockaddr\*)&server\_addr, &addr\_len);

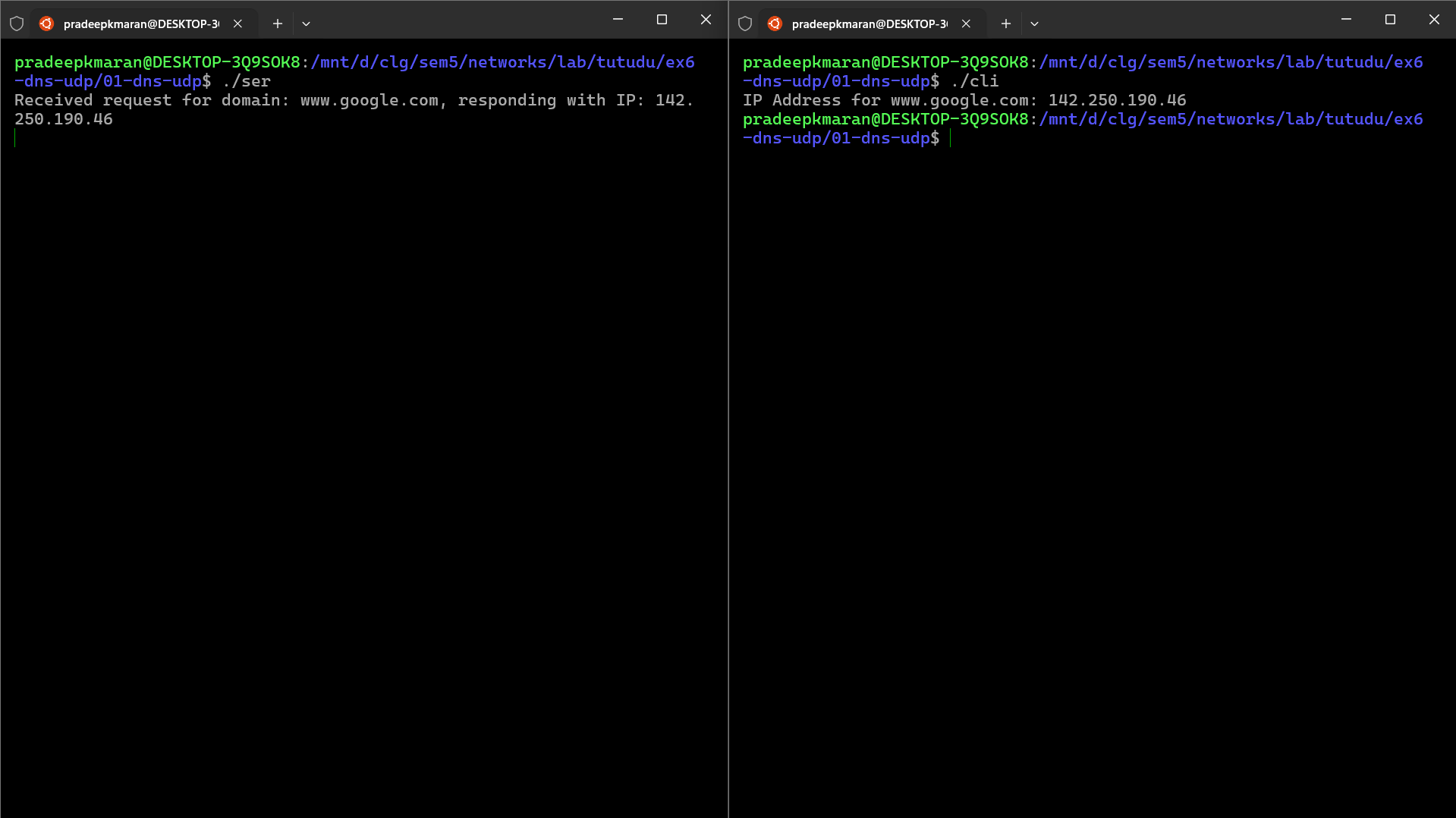
    printf("IP Address for %s: %s\n", domain, ip);

    close(udp\_sock);

    return 0;

}

**Output**



**Exercise 7 : Flow Control**

**Server algorithm**

1. Initialize socket and bind to address.
2. Continuously receive packets using recvfrom().
3. If packet sequence number matches expected\_seq, increment expected\_seq.
4. If out of sequence, print expected sequence number.
5. Send ACK (last correctly received sequence) using sendto().

**Client algorithm**

1. Initialize socket and server address.
2. Input data, source IP, and destination IP.
3. Calculate total packets based on data size.
4. Set window\_start and window\_end based on WINDOW\_SIZE.
5. For each packet in the window:
   1. Extract data chunk.
   2. Create a packet with sequence number, source IP, and destination IP.
   3. Ask user if packet should be sent.
   4. If "Y", send packet via sendto().
6. Wait for ACK using recvfrom().
7. If ACK >= window\_start, update window\_start and window\_end.
8. Ask if transmission should end. If "Y", exit.
9. Close socket.

**server.c**

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#include <unistd.h>

#include <arpa/inet.h>

#define PORT 8080

#define MAX\_PACKET\_SIZE 4

typedef struct {

    char src\_ip[16];                *//* Source IP address as a string

    char dest\_ip[16];               *//* Destination IP address as a string

    int sequence\_number;            *//* Sequence number

    char data[MAX\_PACKET\_SIZE + 1]; *//* Data + 1 for null terminator

    char fcs;                       *//* Frame Check Sequence (dummy for now)

} Packet;

int main() {

    int sockfd;

    struct sockaddr\_in server\_addr, client\_addr;

    socklen\_t addr\_size;

    Packet packet;

    int expected\_seq = 0, ack;

    sockfd = socket(AF\_INET, SOCK\_DGRAM, 0);

    memset(&server\_addr, 0, sizeof(server\_addr));

    server\_addr.sin\_family = AF\_INET;

    server\_addr.sin\_port = htons(PORT);

    server\_addr.sin\_addr.s\_addr = INADDR\_ANY;

    bind(sockfd, (struct sockaddr\*)&server\_addr, sizeof(server\_addr));

    addr\_size = sizeof(client\_addr);

    while (1) {

        recvfrom(sockfd, &packet, sizeof(Packet), 0, (struct sockaddr \*)&client\_addr, &addr\_size);

        printf("Received Packet: Seq %d, Data %s\n", packet.sequence\_number, packet.data);

        if (packet.sequence\_number == expected\_seq) {

            printf("Packet %d is in sequence.\n", packet.sequence\_number);

            expected\_seq++;

        } else {

            printf("Packet %d is out of sequence, expecting %d.\n", packet.sequence\_number, expected\_seq);

        }

        ack = expected\_seq - 1;

        sendto(sockfd, &ack, sizeof(ack), 0, (struct sockaddr \*) &client\_addr, addr\_size);

        printf("Sent ACK %d\n", ack);

    }

    close(sockfd);

    return 0;

}

**client.c**

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#include <unistd.h>

#include <arpa/inet.h>

#define PORT 8080

#define MAX\_PACKET\_SIZE 4

#define WINDOW\_SIZE 4

typedef struct {

    char src\_ip[16];                *//* Source IP address as a string

    char dest\_ip[16];               *//* Destination IP address as a string

    int sequence\_number;            *//* Sequence number

    char data[MAX\_PACKET\_SIZE + 1]; *//* Data + 1 for null terminator

    char fcs;                       *//* Frame Check Sequence (dummy for now)

} Packet;

void create\_packet(Packet \*packet, int seq, const char \*src, const char \*dest, const char \*data) {

    strcpy(packet->src\_ip, src);

    strcpy(packet->dest\_ip, dest);

    packet->sequence\_number = seq;

    strncpy(packet->data, data, MAX\_PACKET\_SIZE);

    packet->data[MAX\_PACKET\_SIZE] = '\0';

    packet->fcs = 'F';

}

void send\_packet(int sockfd, struct sockaddr\_in server\_addr, Packet \*packet) {

    sendto(sockfd, packet, sizeof(Packet), 0, (struct sockaddr \*) &server\_addr, sizeof(server\_addr));

}

int main() {

    int sockfd;

    struct sockaddr\_in server\_addr;

    socklen\_t addr\_size;

    Packet packet;

    char data[16];

    char src\_ip[16], dest\_ip[16];

    int window\_start = 0, window\_end = WINDOW\_SIZE - 1, seq = 0, ack;

    sockfd = socket(AF\_INET, SOCK\_DGRAM, 0);

    memset(&server\_addr, 0, sizeof(server\_addr));

    server\_addr.sin\_family = AF\_INET;

    server\_addr.sin\_port = htons(PORT);

    server\_addr.sin\_addr.s\_addr = inet\_addr("127.0.0.1");

    printf("Enter the data to send (in 8-bit chunks): ");

    scanf("%s", data);

    printf("Enter source IP address: ");

    scanf("%s", src\_ip);

    printf("Enter destination IP address: ");

    scanf("%s", dest\_ip);

    int total\_packets = (strlen(data) + MAX\_PACKET\_SIZE - 1) / MAX\_PACKET\_SIZE;

    while (window\_start < total\_packets) {

        for (seq = window\_start; seq <= window\_end && seq < total\_packets; seq++) {

            char packet\_data[MAX\_PACKET\_SIZE + 1] = {0};

            strncpy(packet\_data, data + seq \* MAX\_PACKET\_SIZE, MAX\_PACKET\_SIZE);

            create\_packet(&packet, seq, src\_ip, dest\_ip, packet\_data);

            printf("Send packet %d (Y/N)? ", seq);

            char send\_decision;

            scanf(" %c", &send\_decision);

            if (send\_decision == 'Y' || send\_decision == 'y') {

                send\_packet(sockfd, server\_addr, &packet);

                printf("Sent Packet: Seq %d, Data %s\n", packet.sequence\_number, packet.data);

            } else {

                printf("Packet %d not sent.\n", seq);

            }

        }

        recvfrom(sockfd, &ack, sizeof(ack), 0, (struct sockaddr \*)&server\_addr, &addr\_size);

        printf("Received ACK %d\n", ack);

        if (ack >= window\_start) {

            window\_start = ack + 1;

            window\_end = window\_start + WINDOW\_SIZE - 1;

        }

        printf("End transmission (Y/N)? ");

        char end\_decision;

        scanf(" %c", &end\_decision);

        if (end\_decision == 'Y' || end\_decision == 'y') {

            break;

        }

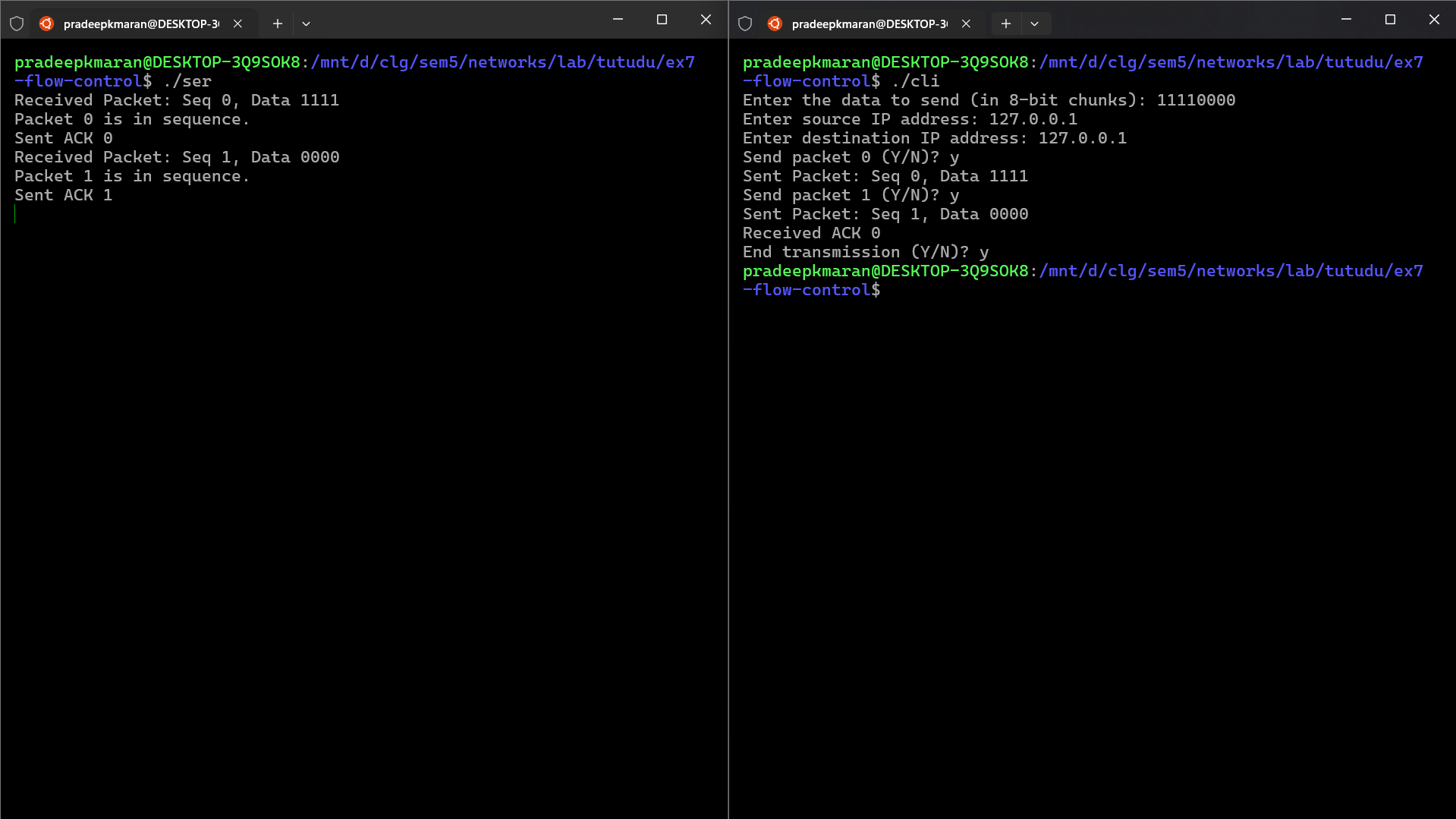
    }

    close(sockfd);

    return 0;

}

**Output**



**Exercise 8 : Error Control**

**Server algorithm**

1. Create a TCP socket.
2. Bind to INADDR\_ANY on port 8080.
3. Listen for incoming connections.
4. Accept an incoming connection.
5. Read the received Hamming code.
6. Check for errors in the Hamming code.
7. If an error is detected, correct it.
8. Close the socket.

**Client algorithm**

1. User enters a 4-bit data string.
2. Calculate the 7-bit Hamming code.
3. Set the second bit of the Hamming code to '0'.
4. Create a TCP socket.
5. Connect to the server at 127.0.0.1 on port 8080.
6. Send the Hamming code to the server.
7. Close the socket.

**server.c**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <arpa/inet.h>

#include <unistd.h>

#define PORT 8080

void checkAndCorrectHammingCode(char \*receivedCode) {

    int hammingBits[7];

    for (int i = 0; i < 7; i++) {

        hammingBits[i] = receivedCode[i] - '0';

    }

    int p1 = hammingBits[0] ^ hammingBits[2] ^ hammingBits[4] ^ hammingBits[6];

    int p2 = hammingBits[1] ^ hammingBits[2] ^ hammingBits[5] ^ hammingBits[6];

    int p4 = hammingBits[3] ^ hammingBits[4] ^ hammingBits[5] ^ hammingBits[6];

    int errorPosition = p4 \* 4 + p2 \* 2 + p1 \* 1;

    if (errorPosition == 0) {

        printf("No error detected in received data.\n");

    } else {

        printf("Error detected at position: %d\n", errorPosition);

        hammingBits[errorPosition - 1] ^= 1;

        printf("Corrected code: ");

        for (int i = 0; i < 7; i++) {

            printf("%d", hammingBits[i]);

        }

        printf("\n");

    }

}

int main() {

    int server\_fd, new\_socket;

    struct sockaddr\_in address;

    int addrlen = sizeof(address);

    char buffer[8] = {0};

    server\_fd = socket(AF\_INET, SOCK\_STREAM, 0);

    address.sin\_family = AF\_INET;

    address.sin\_addr.s\_addr = INADDR\_ANY;

    address.sin\_port = htons(PORT);

    bind(server\_fd, (struct sockaddr \*)&address, sizeof(address));

    listen(server\_fd, 3);

    new\_socket = accept(server\_fd, (struct sockaddr \*)&address, (socklen\_t\*)&addrlen);

    read(new\_socket, buffer, 7);

    printf("Received code: %s\n", buffer);

    checkAndCorrectHammingCode(buffer);

    close(new\_socket);

    close(server\_fd);

    return 0;

}

**client.c**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <arpa/inet.h>

#include <unistd.h>

#define PORT 8080

void calculateHammingCode(char \*data, char \*hammingCode) {

    int dataBits[4];

    int hammingBits[7];

    for (int i = 0; i < 4; i++) {

        dataBits[i] = data[i] - '0';

    }

    hammingBits[2] = dataBits[0];

    hammingBits[4] = dataBits[1];

    hammingBits[5] = dataBits[2];

    hammingBits[6] = dataBits[3];

    hammingBits[0] = hammingBits[2] ^ hammingBits[4] ^ hammingBits[6];

    hammingBits[1] = hammingBits[2] ^ hammingBits[5] ^ hammingBits[6];

    hammingBits[3] = hammingBits[4] ^ hammingBits[5] ^ hammingBits[6];

    for (int i = 0; i < 7; i++) {

        hammingCode[i] = hammingBits[i] + '0';

    }

    hammingCode[7] = '\0';

}

int main() {

    int sock = 0;

    struct sockaddr\_in serv\_addr;

    char data[5], hammingCode[8];

    printf("Enter 4-bit data: ");

    scanf("%4s", data);

    calculateHammingCode(data, hammingCode);

    printf("Hamming code to send: %s\n", hammingCode);

    hammingCode[1] = '0';

    sock = socket(AF\_INET, SOCK\_STREAM, 0);

    serv\_addr.sin\_family = AF\_INET;

    serv\_addr.sin\_port = htons(PORT);

    inet\_pton(AF\_INET, "127.0.0.1", &serv\_addr.sin\_addr);

    connect(sock, (struct sockaddr \*)&serv\_addr, sizeof(serv\_addr));

    send(sock, hammingCode, strlen(hammingCode), 0);

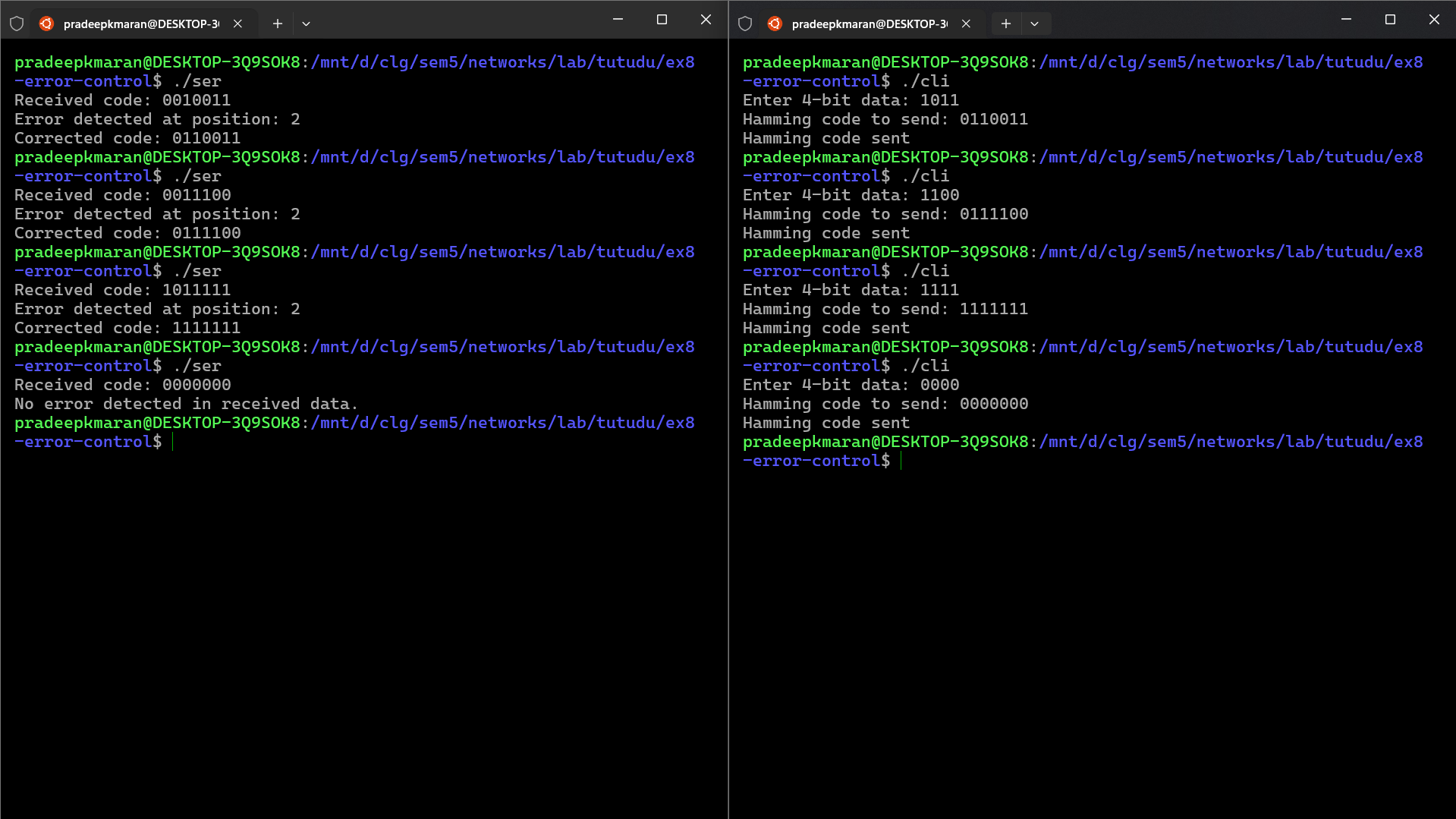
    printf("Hamming code sent\n");

    close(sock);

    return 0;

}

**Output**



**Exercise 9: TCP UDP Performance Evaluation**

**tcp-udp-performance.tcl**

# Create a simulator object

set ns [new Simulator]

# Define different colors for data flows

$ns color 1 Magenta

$ns color 2 Red

# Open trace files

set tracefile [open out.tr w]

$ns trace-all $tracefile

set namfile [open out.nam w]

$ns namtrace-all $namfile

# Define a 'finish' procedure

proc finish {} {

global ns tracefile namfile

$ns flush-trace

close $tracefile

close $namfile

exec nam out.nam &

exit 0

}

# Create six nodes

set n0 [$ns node]

set n1 [$ns node]

set n2 [$ns node]

set n3 [$ns node]

set n4 [$ns node]

set n5 [$ns node]

# Create links between nodes

$ns duplex-link $n0 $n2 2Mb 10ms DropTail

$ns duplex-link $n1 $n2 2Mb 10ms DropTail

$ns simplex-link $n2 $n3 0.3Mb 100ms DropTail

$ns simplex-link $n3 $n2 0.3Mb 100ms DropTail

$ns duplex-link $n3 $n4 0.5Mb 40ms DropTail

$ns duplex-link $n3 $n5 0.5Mb 40ms DropTail

# Set node positions for NAM

$ns duplex-link-op $n0 $n2 orient right-down

$ns duplex-link-op $n1 $n2 orient right-up

$ns simplex-link-op $n2 $n3 orient right

$ns simplex-link-op $n3 $n2 orient left

$ns duplex-link-op $n3 $n4 orient right-up

$ns duplex-link-op $n3 $n5 orient right-down

# Set queue size for bottleneck link

$ns queue-limit $n2 $n3 10

# Setup TCP connection

set tcp [new Agent/TCP]

$ns attach-agent $n0 $tcp

set sink [new Agent/TCPSink]

$ns attach-agent $n4 $sink

$ns connect $tcp $sink

$tcp set fid\_ 1

$tcp set window\_ 8000

$tcp set packetSize\_ 1000

# Setup TCP Application

set ftp [new Application/FTP]

$ftp attach-agent $tcp

# Setup UDP Connection

set udp [new Agent/UDP]

$ns attach-agent $n1 $udp

set null [new Agent/Null]

$ns attach-agent $n5 $null

$ns connect $udp $null

$udp set fid\_ 2

# Setup UDP Application (CBR)

set cbr [new Application/Traffic/CBR]

$cbr attach-agent $udp

$cbr set type\_ CBR

$cbr set packet\_size\_ 1000

$cbr set rate\_ 1mb

$cbr set random\_ false

# Schedule events

$ns at 0.1 "$cbr start"

$ns at 0.1 "$ftp start"

$ns at 4.5 "$ftp stop"

$ns at 4.5 "$cbr stop"

$ns at 5.0 "finish"

# Run the simulation

$ns run

**thru.awk**

BEGIN {

stime = 0

ftime = 0

flag = 0

fsize = 0

throughput = 0

latency = 0

} {

if ($1 == "r" && $4 == 2) { # Check for received packets with flow ID 4

fsize += $6 # Accumulate the size of received packets

if (flag == 0) { # Set the start time on the first packet received

stime = $2

flag = 1

}

ftime = $2 # Update the finish time to the latest packet received

}

} END {

latency = ftime - stime

if (latency > 0) {

throughput = (fsize \* 8) / latency

printf("\nLatency: %f seconds", latency)

printf("\nThroughput: %f Mbps\n", throughput / 1000000)

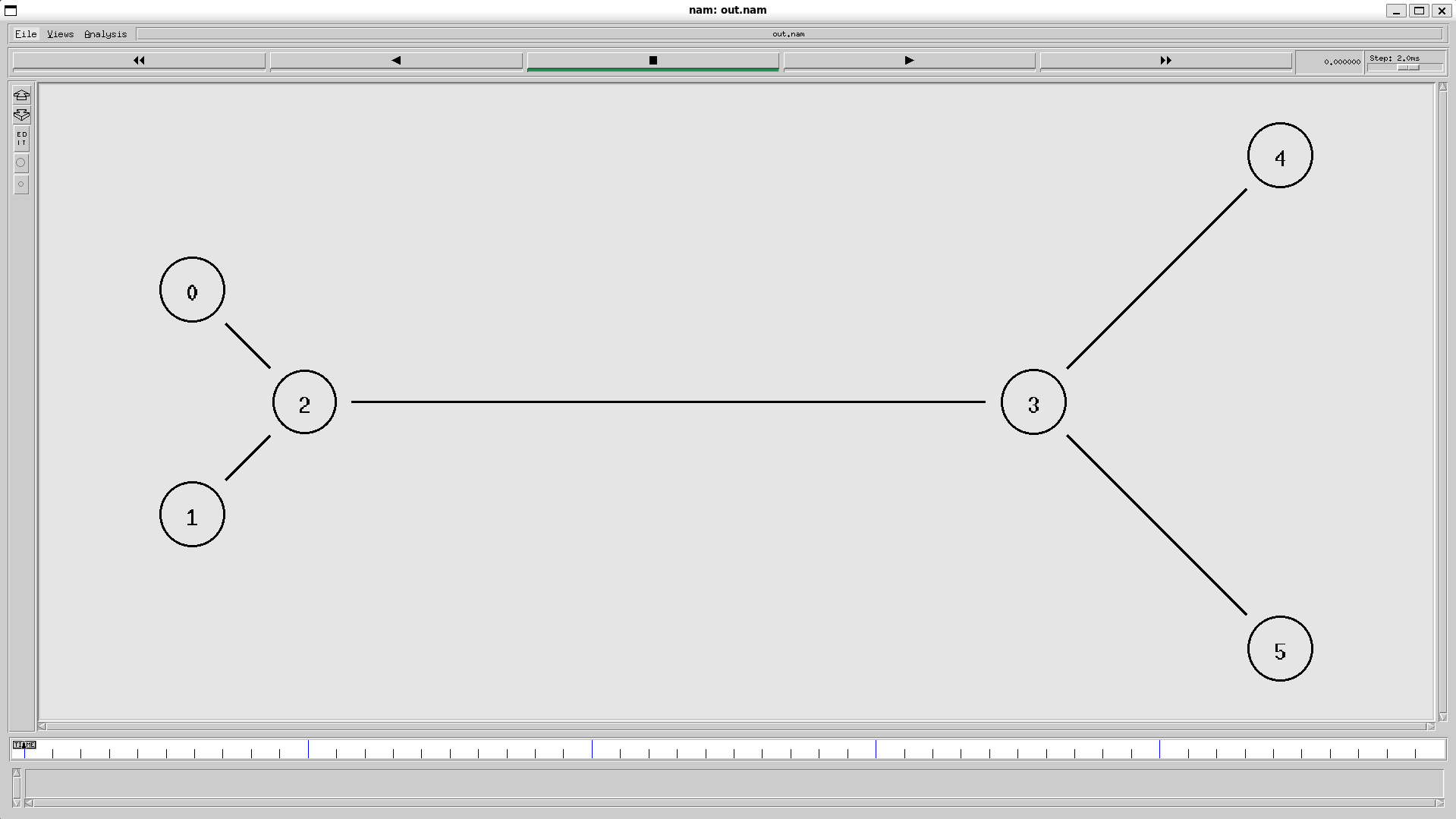
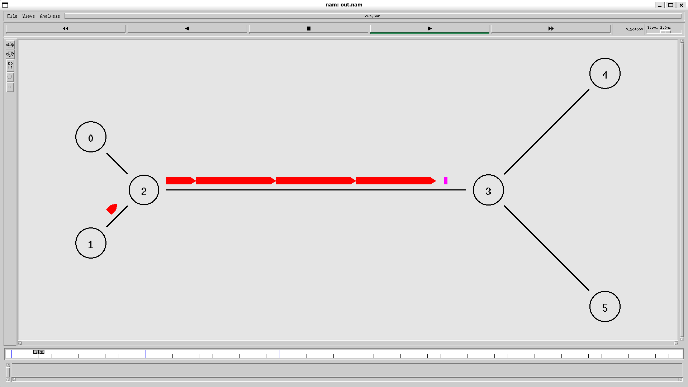
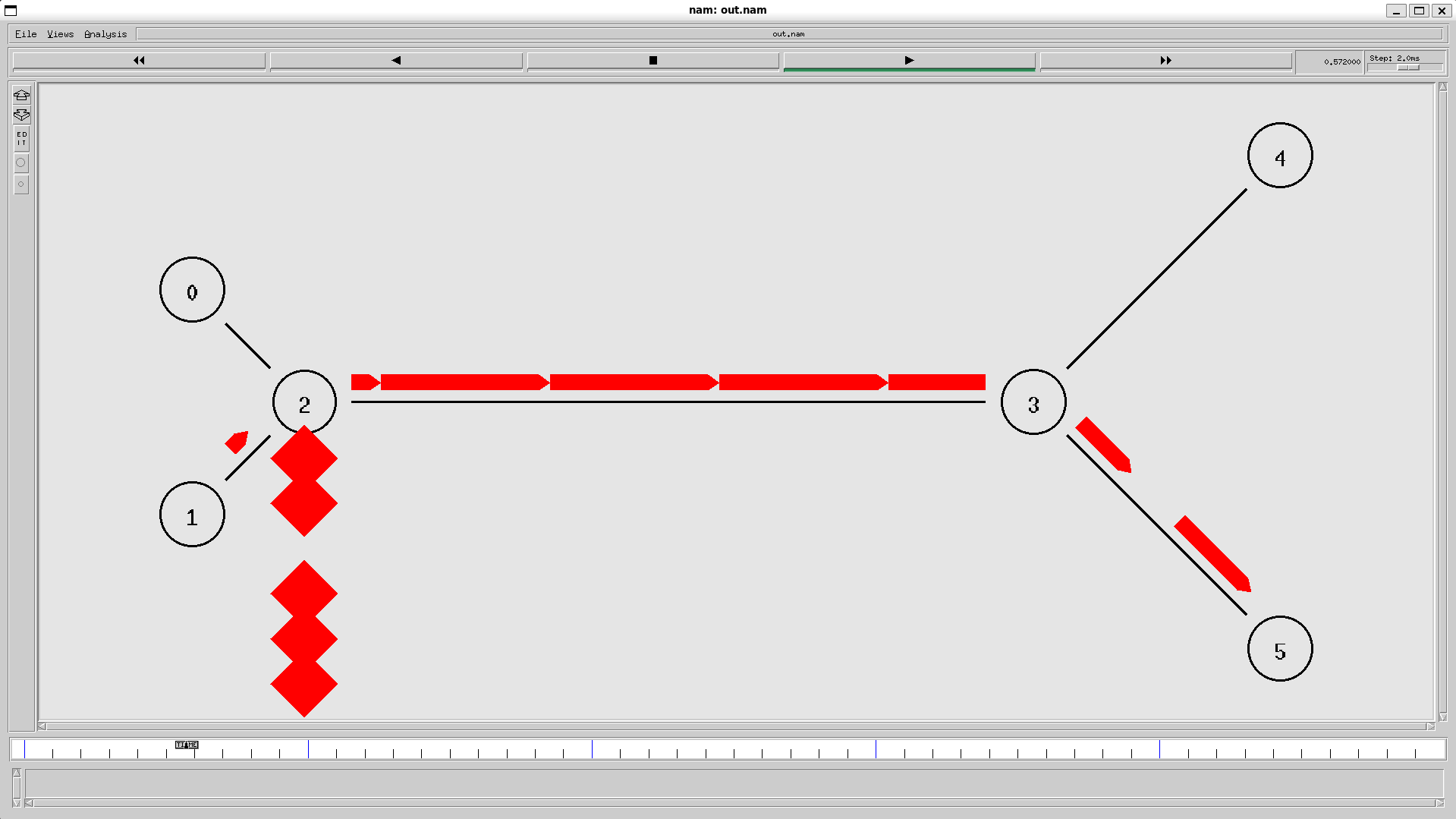
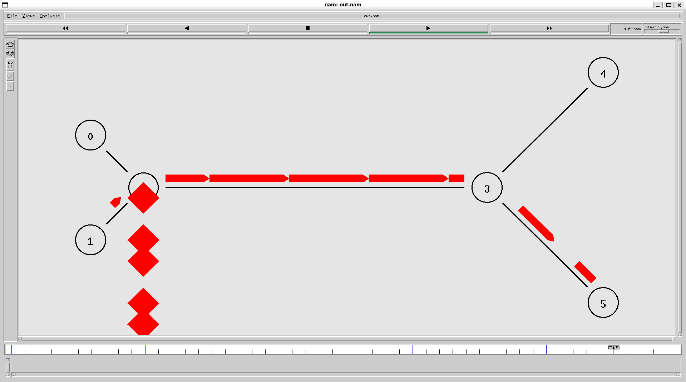
} else {

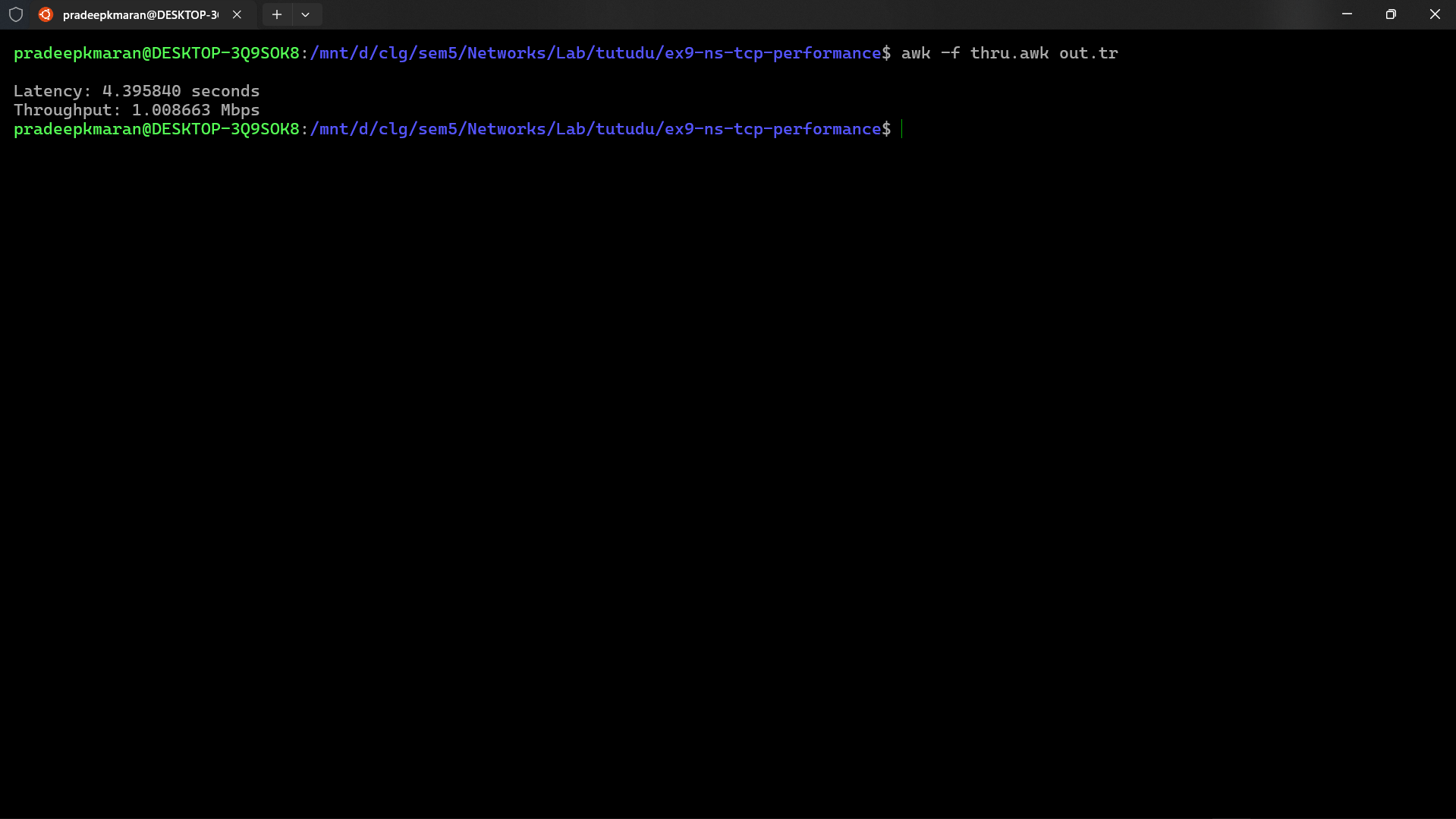
printf("\nError: Invalid latency. Check start and finish times.\n")

}

}

**Output**



**Exercise 10: Distance Vector Routing Protocol**

**dvp.tcl**

# Create a new simulator instance

set ns [new Simulator]

# Enable multicast routing

set multicast\_on 1

# Create trace files

set tf [open out.tr w]

$ns trace-all $tf

set nf [open out.nam w]

$ns namtrace-all $nf

# Define different colors for different flows

$ns color 1 Blue

$ns color 2 Red

# Create 12 nodes

for {set i 0} {$i < 12} {incr i} {

set n($i) [$ns node]

}

# Set node positions for better visualization

# Connected nodes (part of the network)

$n(0) set X\_ 50

$n(0) set Y\_ 50

$n(0) set Z\_ 0

$n(1) set X\_ 50

$n(1) set Y\_ 150

$n(1) set Z\_ 0

$n(5) set X\_ 350

$n(5) set Y\_ 100

$n(5) set Z\_ 0

$n(8) set X\_ 150

$n(8) set Y\_ 100

$n(8) set Z\_ 0

$n(9) set X\_ 150

$n(9) set Y\_ 50

$n(9) set Z\_ 0

$n(10) set X\_ 150

$n(10) set Y\_ 150

$n(10) set Z\_ 0

$n(11) set X\_ 250

$n(11) set Y\_ 100

$n(11) set Z\_ 0

# Unused nodes (positioned away from the main network)

$n(2) set X\_ 50

$n(2) set Y\_ 180

$n(2) set Z\_ 0

$n(3) set X\_ 100

$n(3) set Y\_ 180

$n(3) set Z\_ 0

$n(4) set X\_ 150

$n(4) set Y\_ 180

$n(4) set Z\_ 0

$n(6) set X\_ 200

$n(6) set Y\_ 180

$n(6) set Z\_ 0

$n(7) set X\_ 250

$n(7) set Y\_ 180

$n(7) set Z\_ 0

# Create links between nodes

$ns duplex-link $n(0) $n(8) 1Mb 10ms DropTail

$ns duplex-link $n(0) $n(9) 1Mb 10ms DropTail

$ns duplex-link $n(1) $n(10) 1Mb 10ms DropTail

$ns duplex-link $n(9) $n(11) 1Mb 10ms DropTail

$ns duplex-link $n(10) $n(11) 1Mb 10ms DropTail

$ns duplex-link $n(11) $n(5) 1Mb 10ms DropTail

# Set link orientations

$ns duplex-link-op $n(0) $n(8) orient left

$ns duplex-link-op $n(0) $n(9) orient right

$ns duplex-link-op $n(1) $n(10) orient down

$ns duplex-link-op $n(9) $n(11) orient right

$ns duplex-link-op $n(10) $n(11) orient down

$ns duplex-link-op $n(11) $n(5) orient right

# Setup UDP connections

# First UDP connection (0 to 5)

set udp0 [new Agent/UDP]

$ns attach-agent $n(0) $udp0

set null0 [new Agent/Null]

$ns attach-agent $n(5) $null0

$ns connect $udp0 $null0

$udp0 set fid\_ 1

# Second UDP connection (1 to 5)

set udp1 [new Agent/UDP]

$ns attach-agent $n(1) $udp1

set null1 [new Agent/Null]

$ns attach-agent $n(5) $null1

$ns connect $udp1 $null1

$udp1 set fid\_ 2

# Create CBR traffic for both connections

set cbr0 [new Application/Traffic/CBR]

$cbr0 set packetSize\_ 500

$cbr0 set rate\_ 200kb

$cbr0 set random\_ 1

$cbr0 attach-agent $udp0

set cbr1 [new Application/Traffic/CBR]

$cbr1 set packetSize\_ 500

$cbr1 set rate\_ 200kb

$cbr1 set random\_ 1

$cbr1 attach-agent $udp1

# Use Distance Vector Routing

$ns rtproto DV

# Define a procedure to close trace files

proc finish {} {

global ns nf tf

$ns flush-trace

close $nf

close $tf

exec nam out.nam &

exit 0

}

# Schedule events

$ns at 0.1 "$cbr0 start"

$ns at 0.2 "$cbr1 start"

# Schedule link failure for only link 11-5

$ns rtmodel-at 1.0 down $n(11) $n(5)

$ns rtmodel-at 2.0 up $n(11) $n(5)

# Stop the traffic

$ns at 4.5 "$cbr0 stop"

$ns at 4.5 "$cbr1 stop"

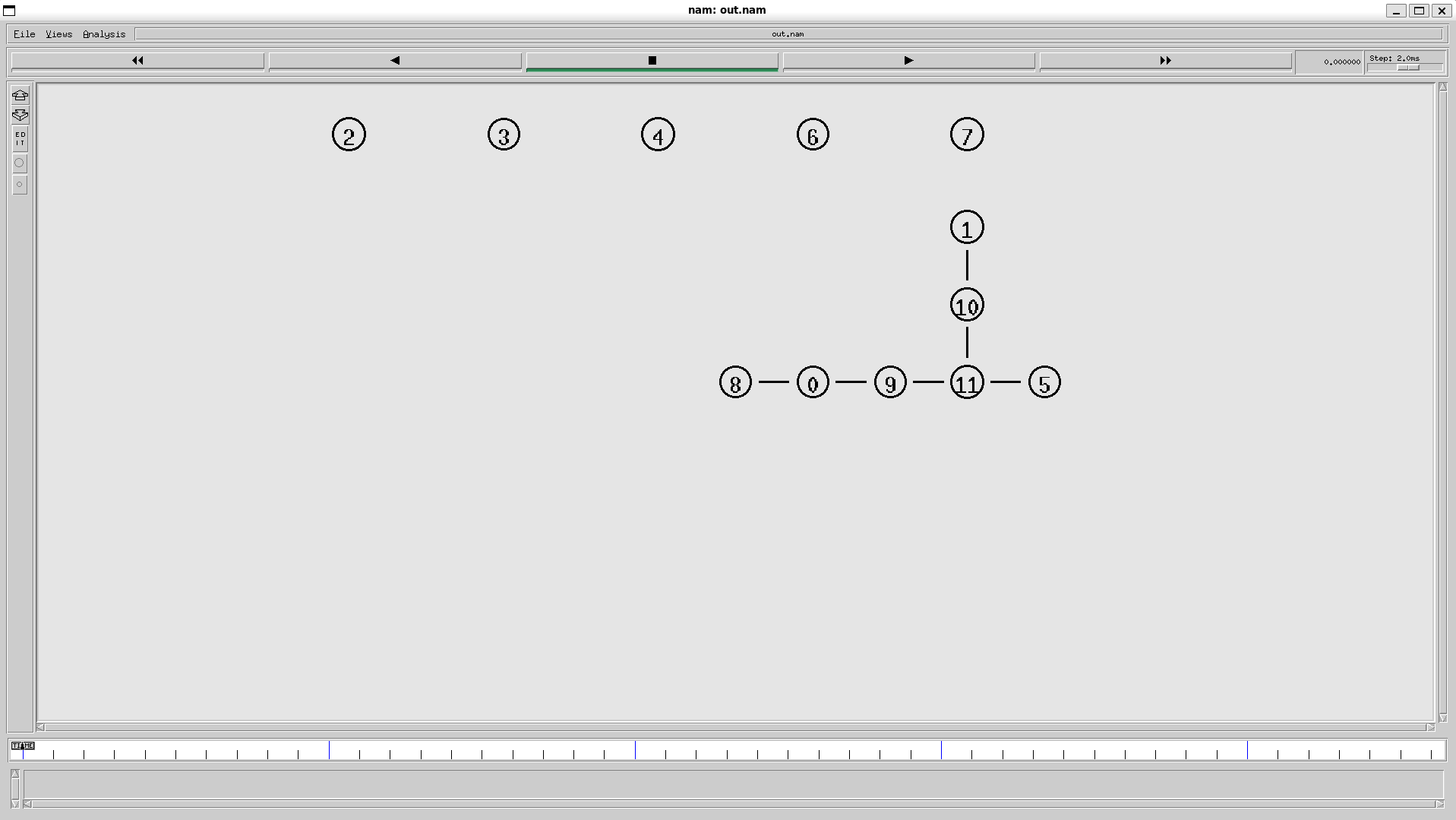
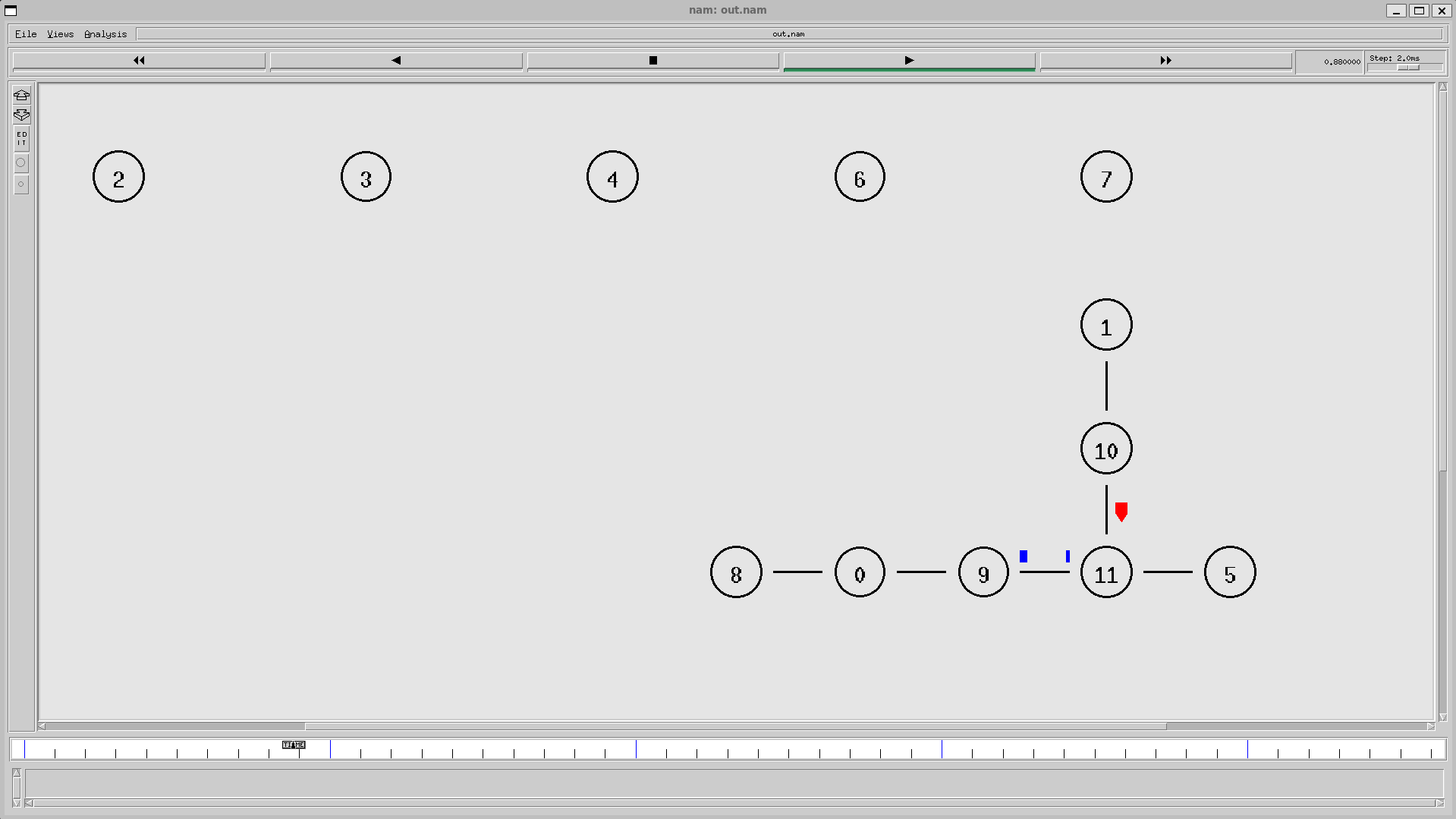
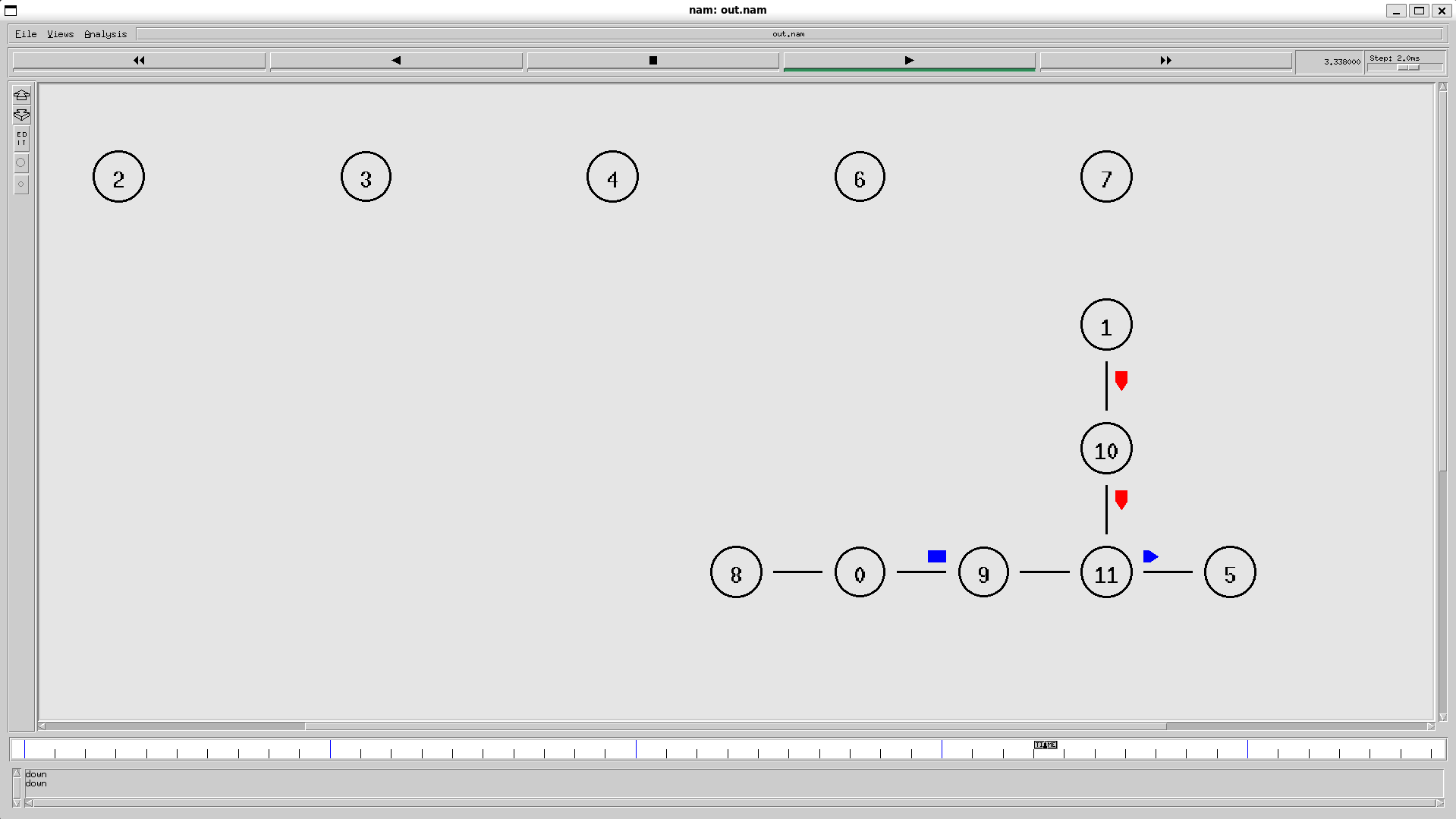
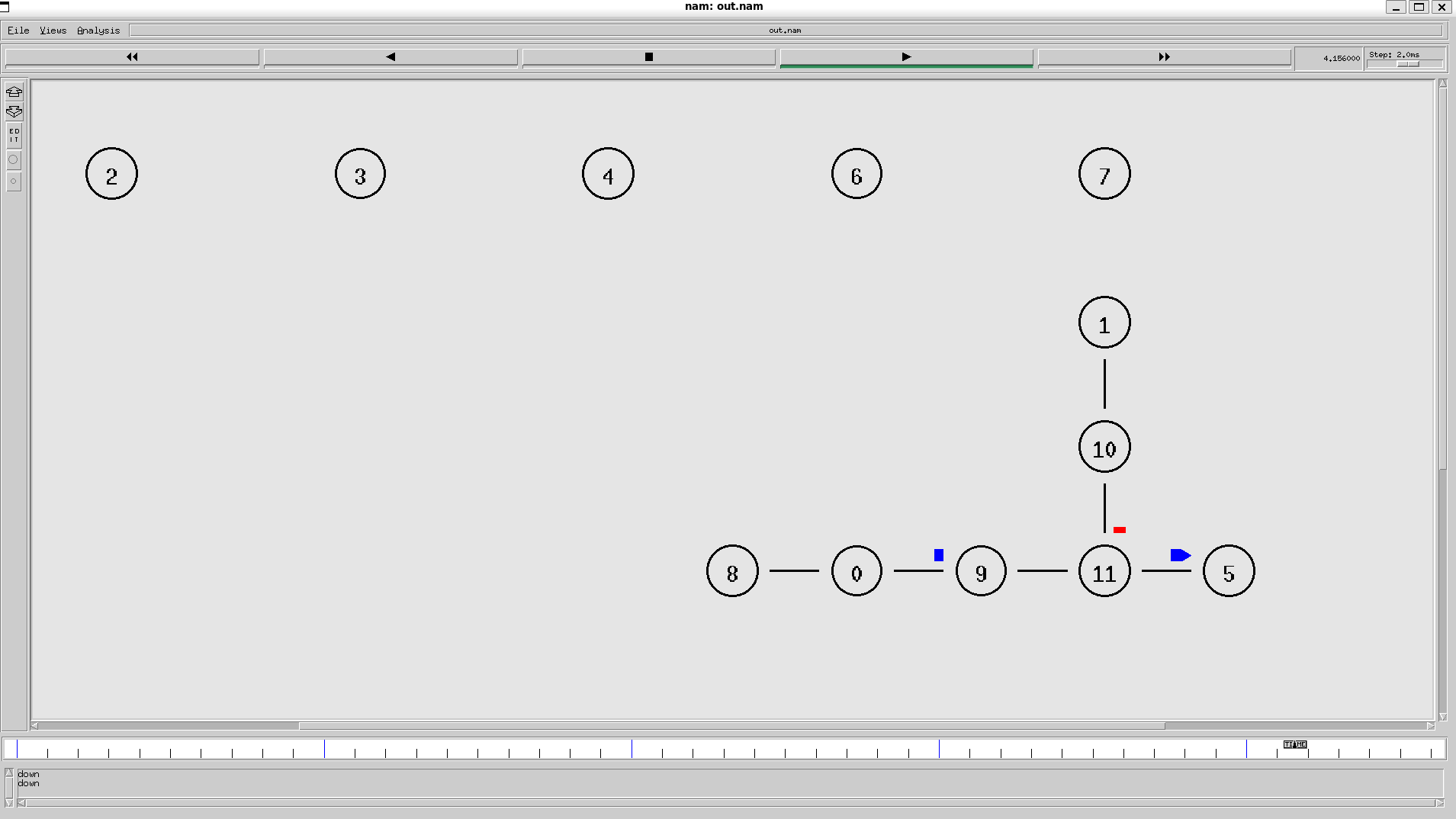
# Call finish procedure after 5 seconds

$ns at 5.0 "finish"

# Run the simulation

$ns run

**Output**

**Exercise 11: Link State Routing Protocol**

**lsrp.tcl**

# Create a new simulator instance

set ns [new Simulator]

# Enable multicast routing

set multicast\_on 1

# Create trace files

set tf [open out.tr w]

$ns trace-all $tf

set nf [open out.nam w]

$ns namtrace-all $nf

# Define different colors for different flows

$ns color 1 Blue

$ns color 2 Red

# Create 12 nodes

for {set i 0} {$i < 12} {incr i} {

set n($i) [$ns node]

}

# Set node positions for better visualization # Connected nodes (part of the network)

$n(0) set X\_ 50

$n(0) set Y\_ 50

$n(0) set Z\_ 0

$n(1) set X\_ 50

$n(1) set Y\_ 150

$n(1) set Z\_ 0

$n(5) set X\_ 350

$n(5) set Y\_ 100

$n(5) set Z\_ 0

$n(8) set X\_ 150

$n(8) set Y\_ 100

$n(8) set Z\_ 0

$n(9) set X\_ 150

$n(9) set Y\_ 50

$n(9) set Z\_ 0

$n(10) set X\_ 150

$n(10) set Y\_ 150

$n(10) set Z\_ 0

$n(11) set X\_ 250

$n(11) set Y\_ 100

$n(11) set Z\_ 0

# Unused nodes (positioned away from the main network)

$n(2) set X\_ 50

$n(2) set Y\_ 180

$n(2) set Z\_ 0

$n(3) set X\_ 100

$n(3) set Y\_ 180

$n(3) set Z\_ 0

$n(4) set X\_ 150

$n(4) set Y\_ 180

$n(4) set Z\_ 0

$n(6) set X\_ 200

$n(6) set Y\_ 180

$n(6) set Z\_ 0

$n(7) set X\_ 250

$n(7) set Y\_ 180

$n(7) set Z\_ 0

# Create links between nodes

$ns duplex-link $n(0) $n(8) 1Mb 10ms DropTail

$ns duplex-link $n(0) $n(9) 1Mb 10ms DropTail

$ns duplex-link $n(1) $n(10) 1Mb 10ms DropTail

$ns duplex-link $n(9) $n(11) 1Mb 10ms DropTail

$ns duplex-link $n(10) $n(11) 1Mb 10ms DropTail

$ns duplex-link $n(11) $n(5) 1Mb 10ms DropTail

# Set link orientations

$ns duplex-link-op $n(0) $n(8) orient left

$ns duplex-link-op $n(0) $n(9) orient right

$ns duplex-link-op $n(1) $n(10) orient down

$ns duplex-link-op $n(9) $n(11) orient right

$ns duplex-link-op $n(10) $n(11) orient down

$ns duplex-link-op $n(11) $n(5) orient right

# Setup UDP connections

# First UDP connection (0 to 5)

set udp0 [new Agent/UDP]

$ns attach-agent $n(0) $udp0

set null0 [new Agent/Null]

$ns attach-agent $n(5) $null0

$ns connect $udp0 $null0

$udp0 set fid\_ 1

# Second UDP connection (1 to 5)

set udp1 [new Agent/UDP]

$ns attach-agent $n(1) $udp1

set null1 [new Agent/Null]

$ns attach-agent $n(5) $null1

$ns connect $udp1 $null1

$udp1 set fid\_ 2

# Create CBR traffic for both connections

set cbr0 [new Application/Traffic/CBR]

$cbr0 set packetSize\_ 500

$cbr0 set rate\_ 200kb

$cbr0 set random\_ 1

$cbr0 attach-agent $udp0

set cbr1 [new Application/Traffic/CBR]

$cbr1 set packetSize\_ 500

$cbr1 set rate\_ 200kb

$cbr1 set random\_ 1

$cbr1 attach-agent $udp1

# Use Link State Routing (instead of DV)

$ns rtproto LS

# Define a procedure to close trace files

proc finish {} {

global ns nf tf

$ns flush-trace

close $nf

close $tf

exec nam out.nam &

exit 0

}

# Schedule events

$ns at 0.1 "$cbr0 start"

$ns at 0.2 "$cbr1 start"

# Schedule link failure for only link 11-5

$ns rtmodel-at 1.0 down $n(11) $n(5)

$ns rtmodel-at 2.0 up $n(11) $n(5)

# Stop the traffic

$ns at 4.5 "$cbr0 stop"

$ns at 4.5 "$cbr1 stop"

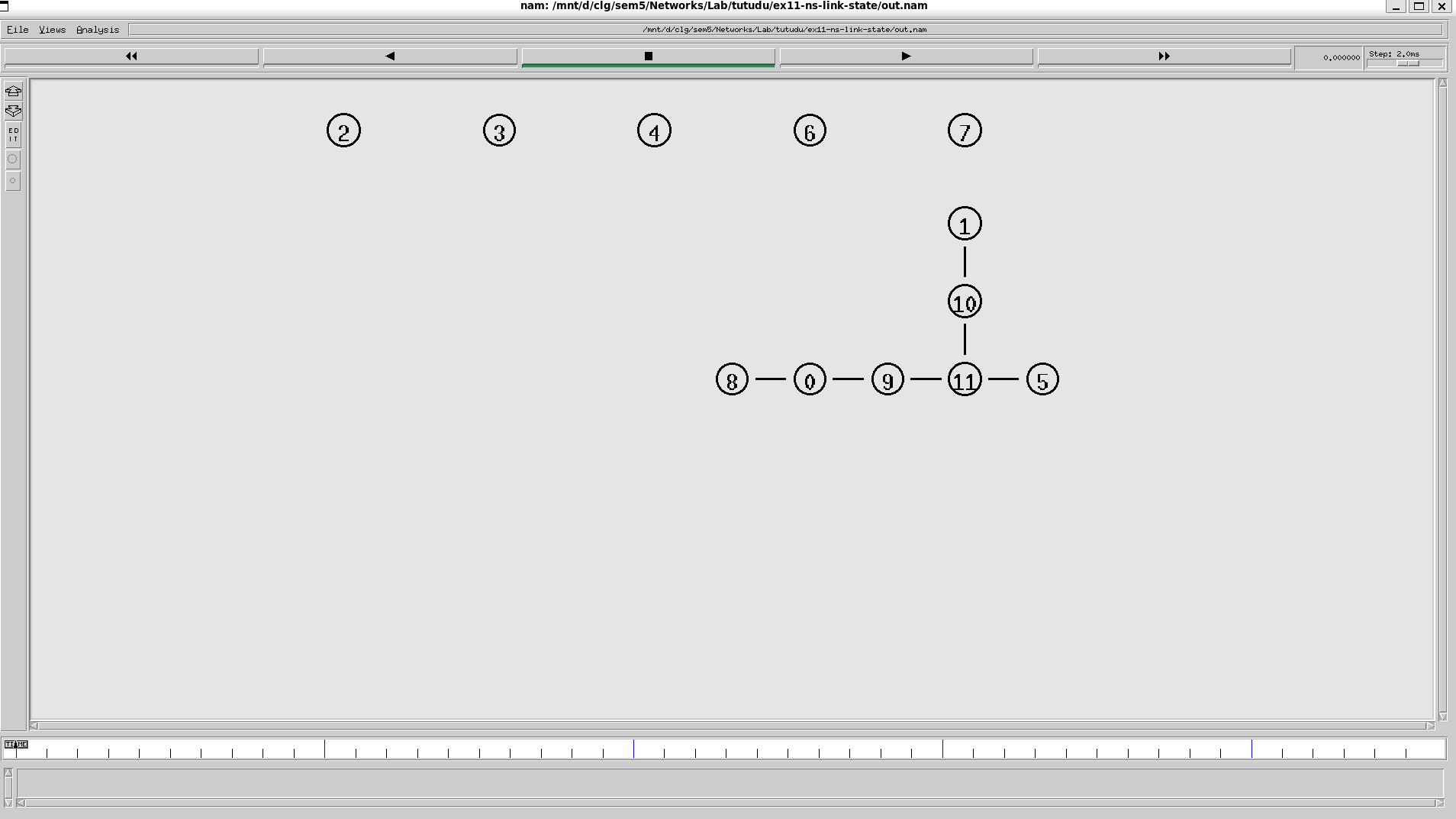
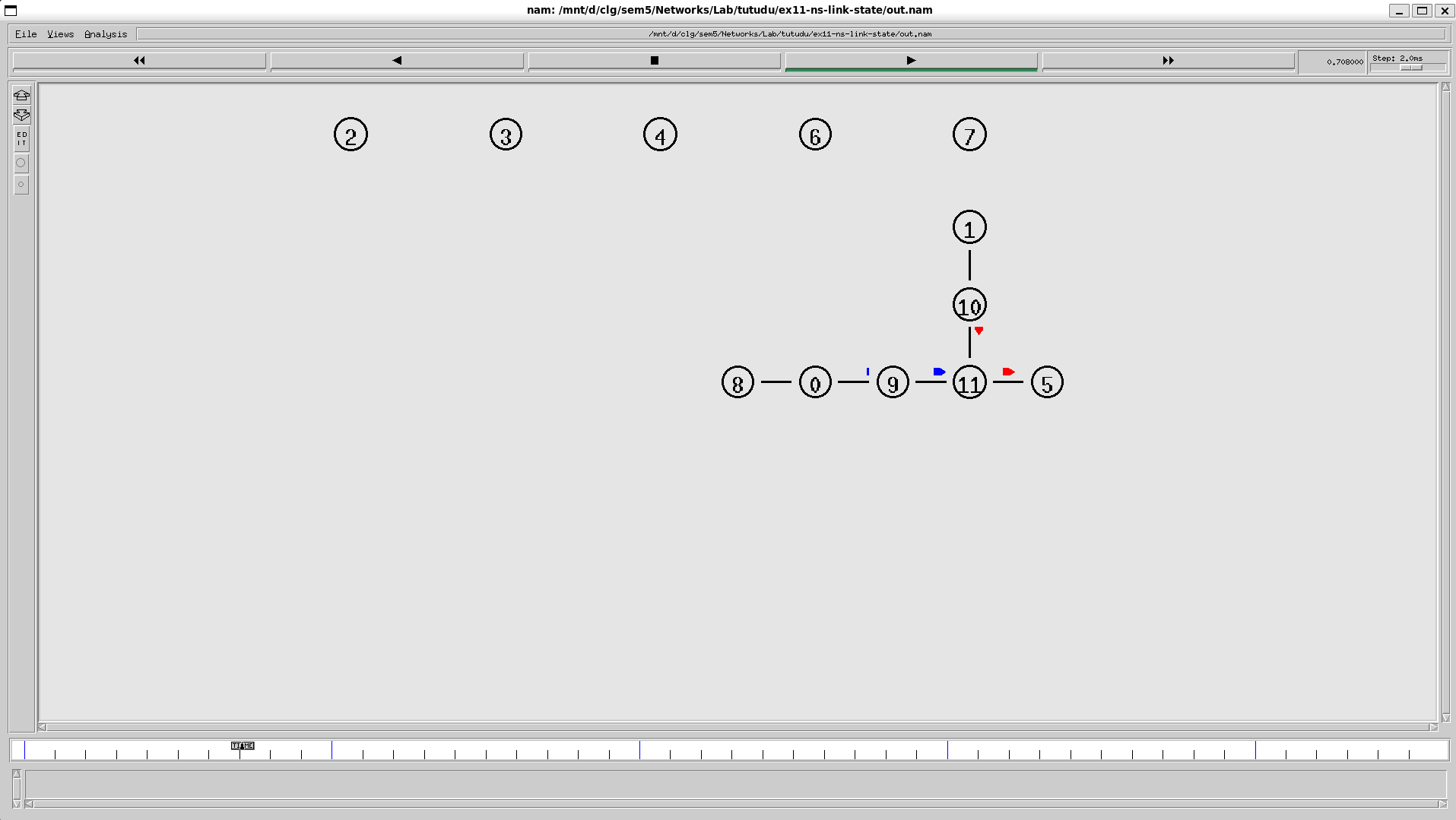
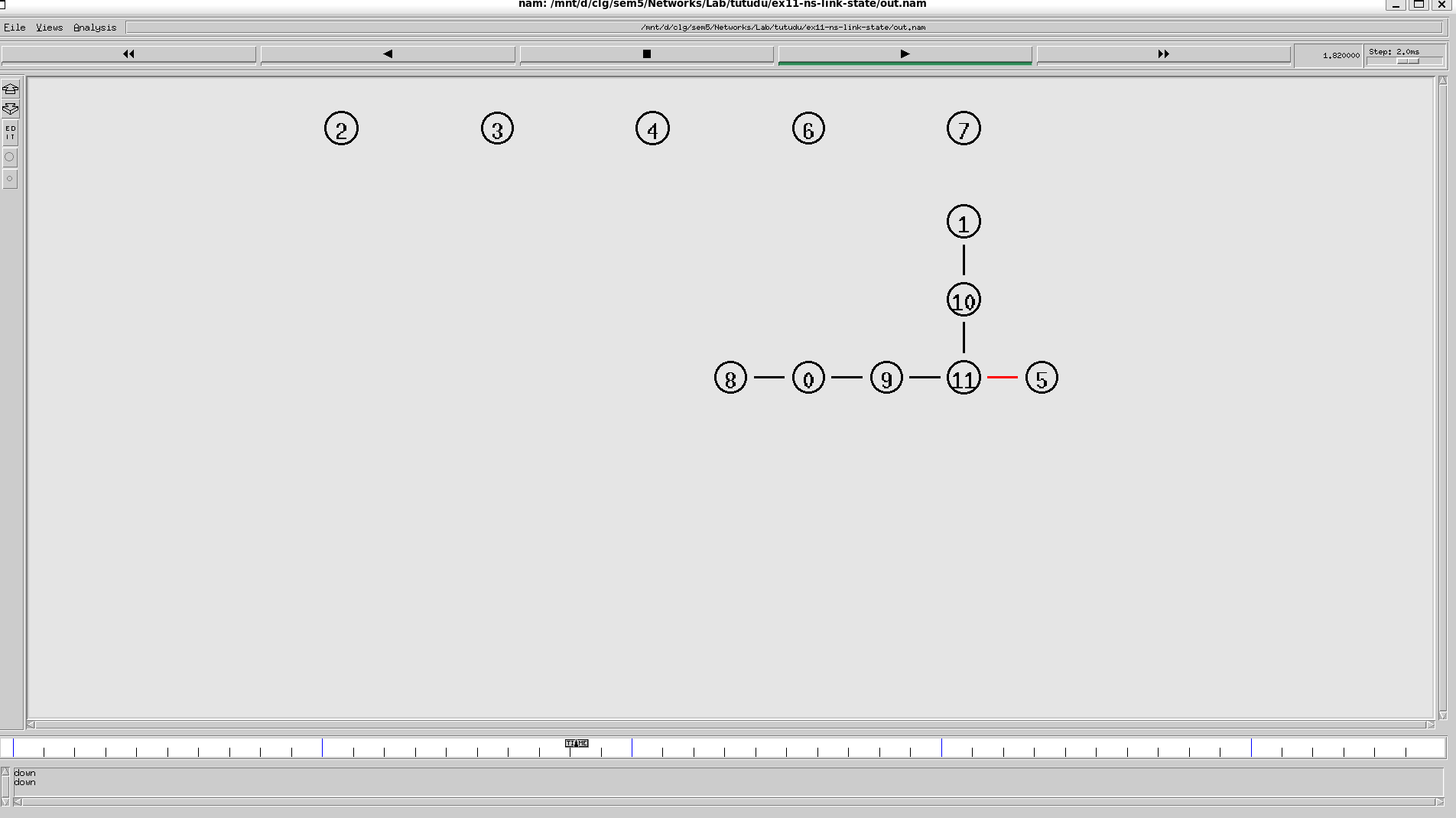
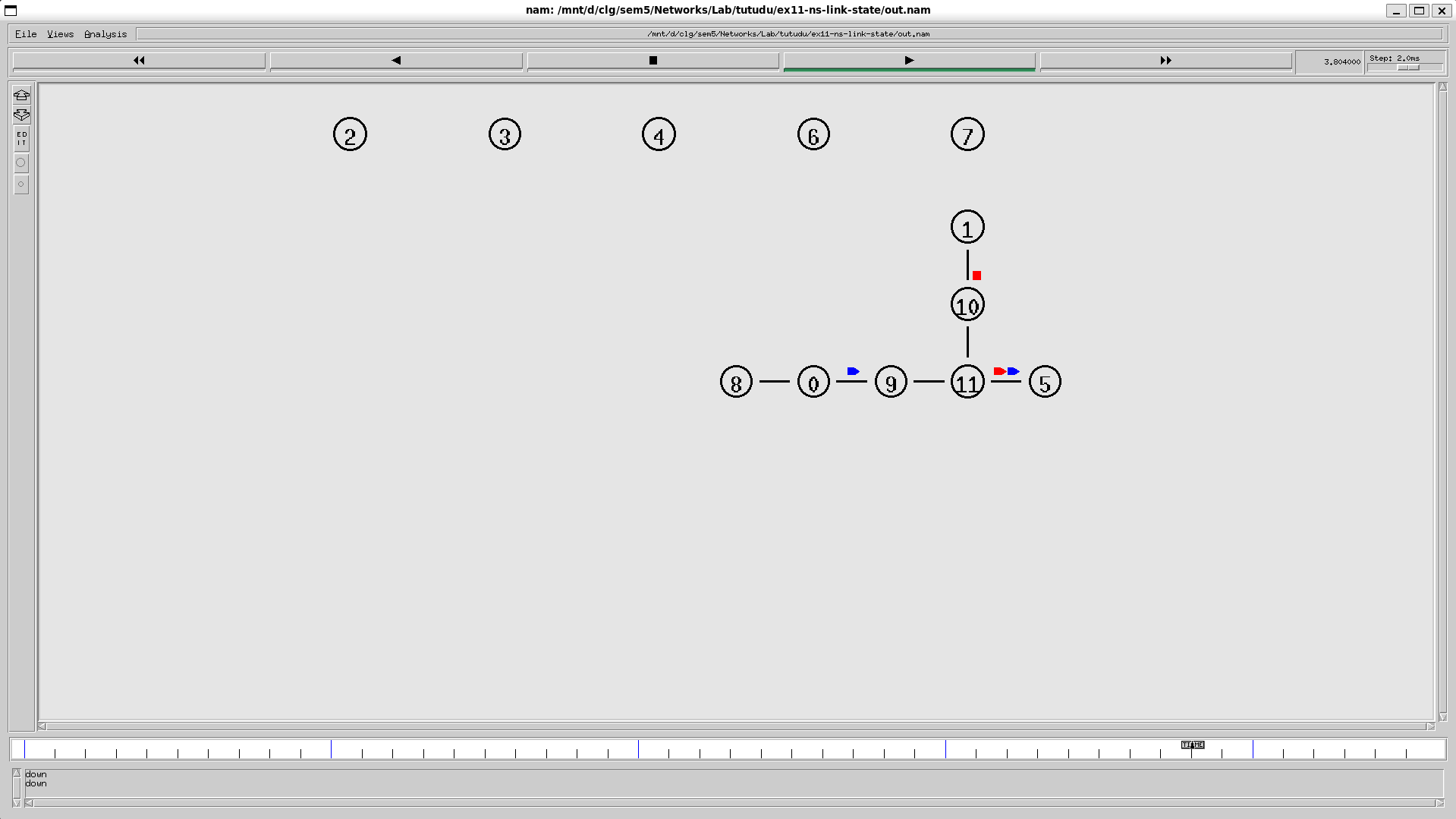
# Call finish procedure after 5 seconds

$ns at 5.0 "finish"

# Run the simulation

$ns run

**Output**

**Exercise 12: TCP Congestion Control Algorithms**

**reno.tcl**

# Create a simulator object

set ns [new Simulator]

# Open the NAM file and the trace file

set nf [open basic1.nam w]

$ns namtrace-all $nf

set tf [open basic1.tr w]

$ns trace-all $tf

# Define a 'finish' procedure

proc finish {} {

global ns nf tf

$ns flush-trace

close $nf

close $tf

exec nam basic1.nam &

exec xgraph reno.xg &

exit 0

}

# Create the network nodes

set n0 [$ns node]

set n1 [$ns node]

set n2 [$ns node]

# Create duplex links

$ns duplex-link $n0 $n1 10Mb 10ms DropTail

$ns duplex-link $n1 $n2 800Kb 50ms DropTail

# Set queue limit for the router

$ns queue-limit $n1 $n2 7

# Visual hints for NAM

$ns color 0 Red

$ns duplex-link-op $n0 $n1 orient right

$ns duplex-link-op $n1 $n2 orient right

$ns duplex-link-op $n1 $n2 queuePos 0.5

# Create and configure TCP sending agent

set tcp [new Agent/TCP/Reno]

$tcp set class\_ 0

$tcp set window\_ 100

$tcp set packetSize\_ 960

$ns attach-agent $n0 $tcp

# Create and attach TCP receive agent (sink)

set sink [new Agent/TCPSink]

$ns attach-agent $n2 $sink

$ns connect $tcp $sink

# Schedule the data flow

set ftp [new Application/FTP]

$ftp attach-agent $tcp

$ns at 0.0 "$ftp start"

$ns at 10.0 "finish"

# Procedure to plot the congestion window

proc plotWindow {tcpSource outfile} {

global ns

set now [$ns now]

set cwnd [$tcpSource set cwnd\_]

puts $outfile "$now $cwnd"

$ns at [expr $now + 0.1] "plotWindow $tcpSource $outfile"

}

# Open file to log congestion window

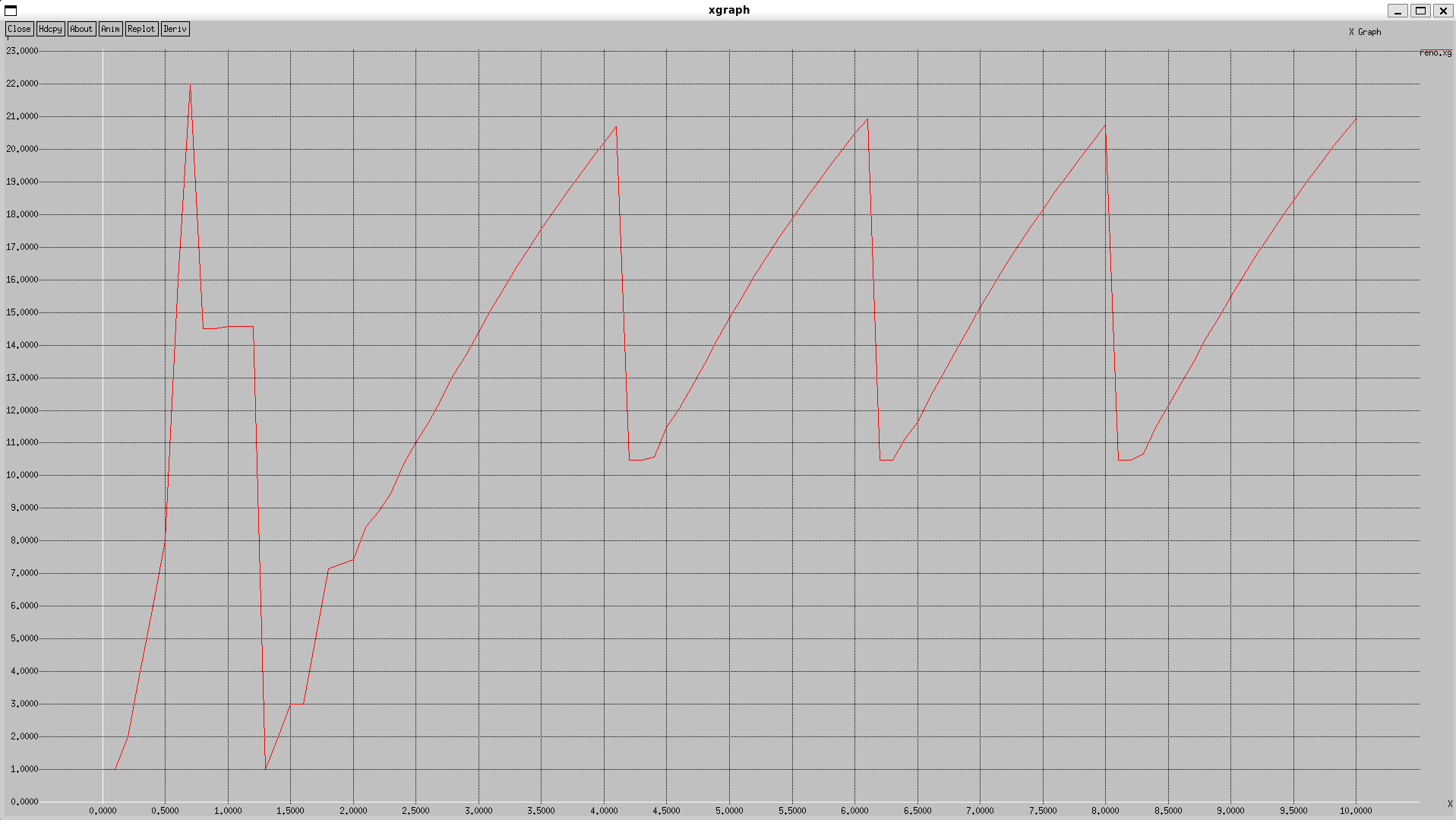
set outfile [open "reno.xg" w]

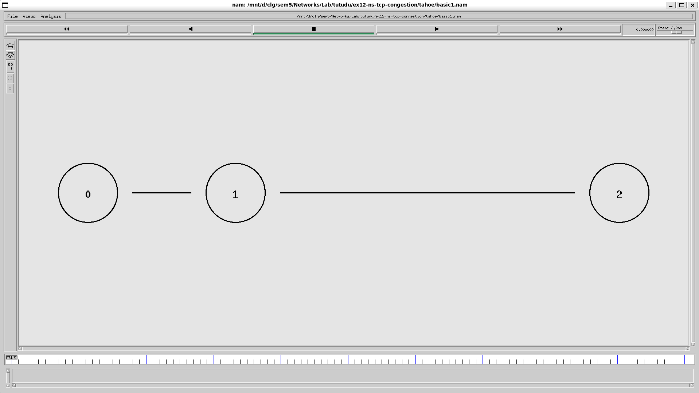
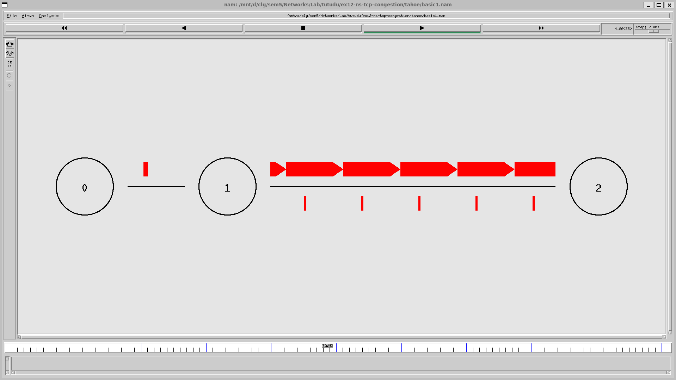
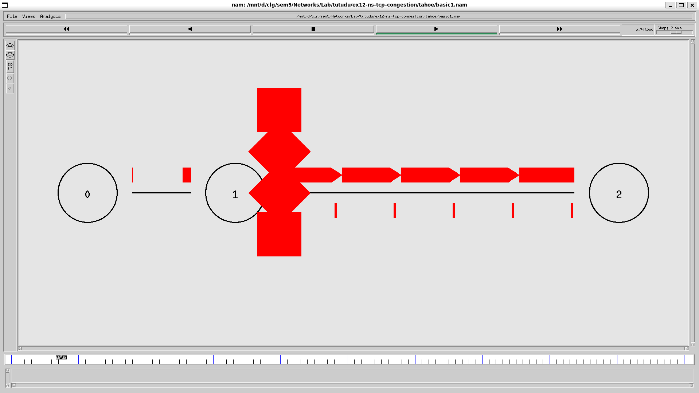
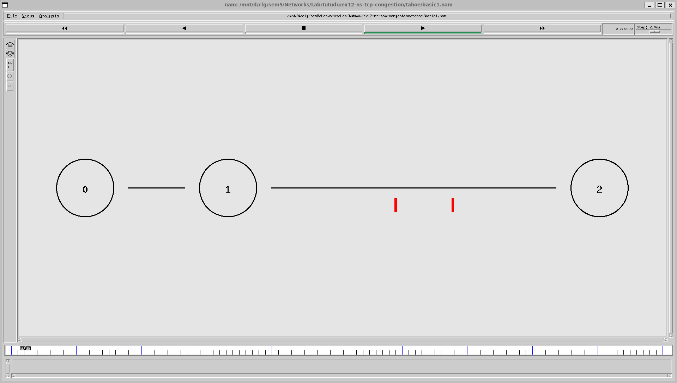
$ns at 0.0 "plotWindow $tcp $outfile"

# Run the simulation

$ns run

**Output**





**tahoe.tcl**

# Create a simulator object

set ns [new Simulator]

# Define different colors for data flows (for NAM)

$ns color 1 Blue

$ns color 2 Red

# Open the NAM trace file

set nf [open taho.nam w]

$ns namtrace-all $nf

# Open the trace file for general simulation data

set tf [open taho.tr w]

$ns trace-all $tf

# Define a 'finish' procedure

proc finish {} {

global ns nf tf

$ns flush-trace

# Close the NAM trace file

close $nf

close $tf

# Execute NAM on the trace file

exec nam taho.nam &

exec xgraph taho.xg &

exit 0

}

# Create three nodes

set n0 [$ns node]

set n1 [$ns node]

set n2 [$ns node]

# Create links between the nodes

$ns duplex-link $n0 $n1 10Mb 10ms DropTail

$ns duplex-link $n1 $n2 2Mb 10ms DropTail

# Set Queue Size of link (n0-n1) to 10 packets

$ns queue-limit $n0 $n1 10

# Position nodes for visualization in NAM

$ns duplex-link-op $n0 $n1 orient right-down

$ns duplex-link-op $n1 $n2 orient right

# Monitor the queue for link (n0-n1). (for NAM)

$ns duplex-link-op $n0 $n1 queuePos 0.5

# Setup a TCP connection using the default TCP agent

set tcp [new Agent/TCP] ;# Use default TCP, which should be Tahoe

$tcp set window\_ 10 ;# Set the window size (e.g., 10 packets)

$tcp set packetSize\_ 1000 ;# Set the packet size (e.g., 1000 bytes)

$tcp set timeout\_ 1.0 ;# Set the timeout (e.g., 1.0 seconds)

$ns attach-agent $n0 $tcp

# Create a TCP Sink on the destination node

set sink [new Agent/TCPSink]

$ns attach-agent $n2 $sink

$ns connect $tcp $sink

$tcp set fid\_ 1

# Setup an FTP application over the TCP connection

set ftp [new Application/FTP]

$ftp attach-agent $tcp

# Schedule the FTP events

$ns at 0.1 "$ftp start"

$ns at 4.0 "$ftp stop"

# Call the finish procedure after 5 seconds of simulation time

$ns at 5.0 "finish"

# Procedure to plot the congestion window

proc plotWindow {tcpSource outfile} {

global ns

set now [$ns now]

set cwnd [$tcpSource set cwnd\_]

# Record the data in a file

puts $outfile "$now $cwnd"

$ns at [expr $now + 0.1] "plotWindow $tcpSource $outfile"

}

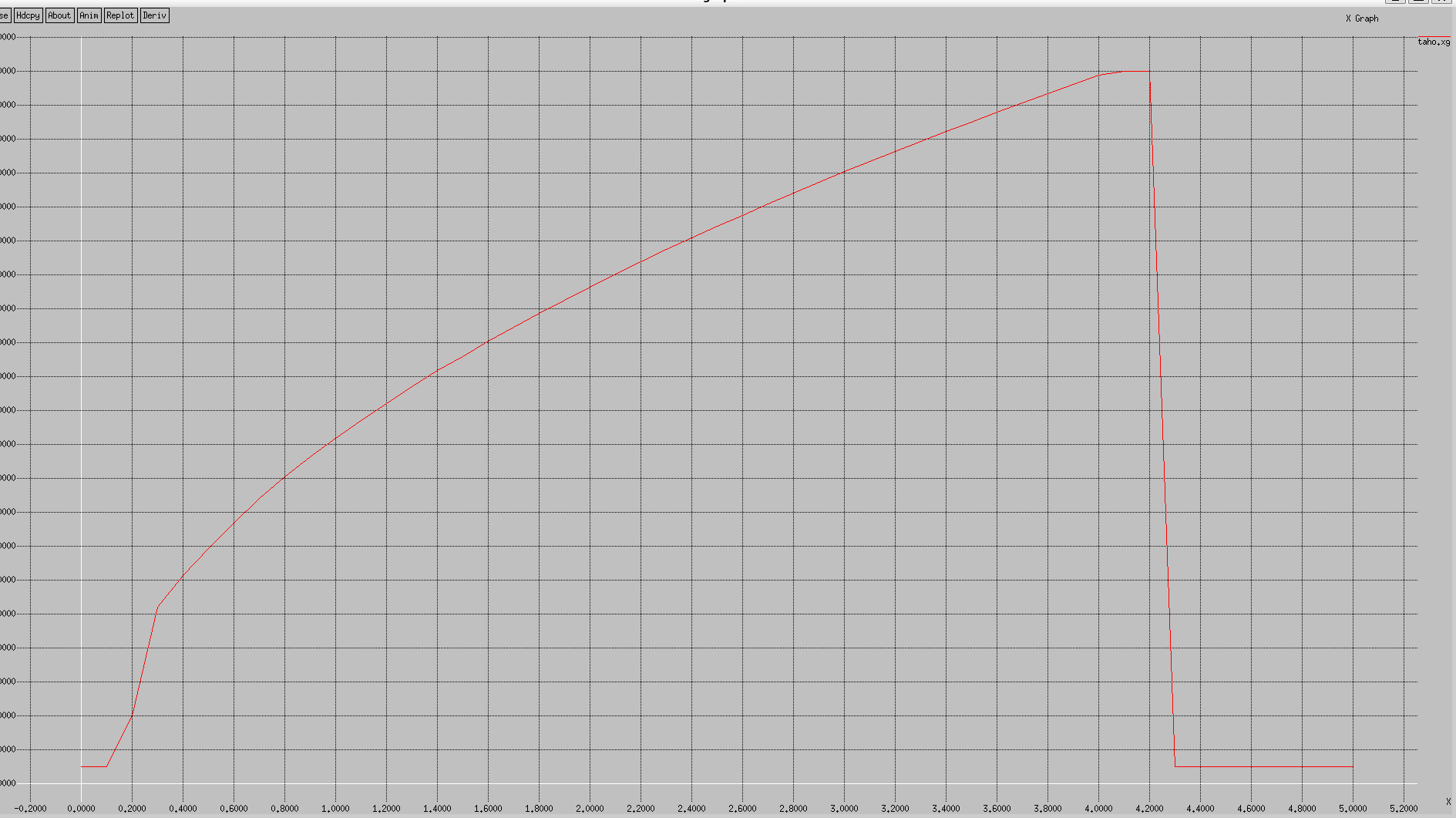
# Prepare to record the congestion window

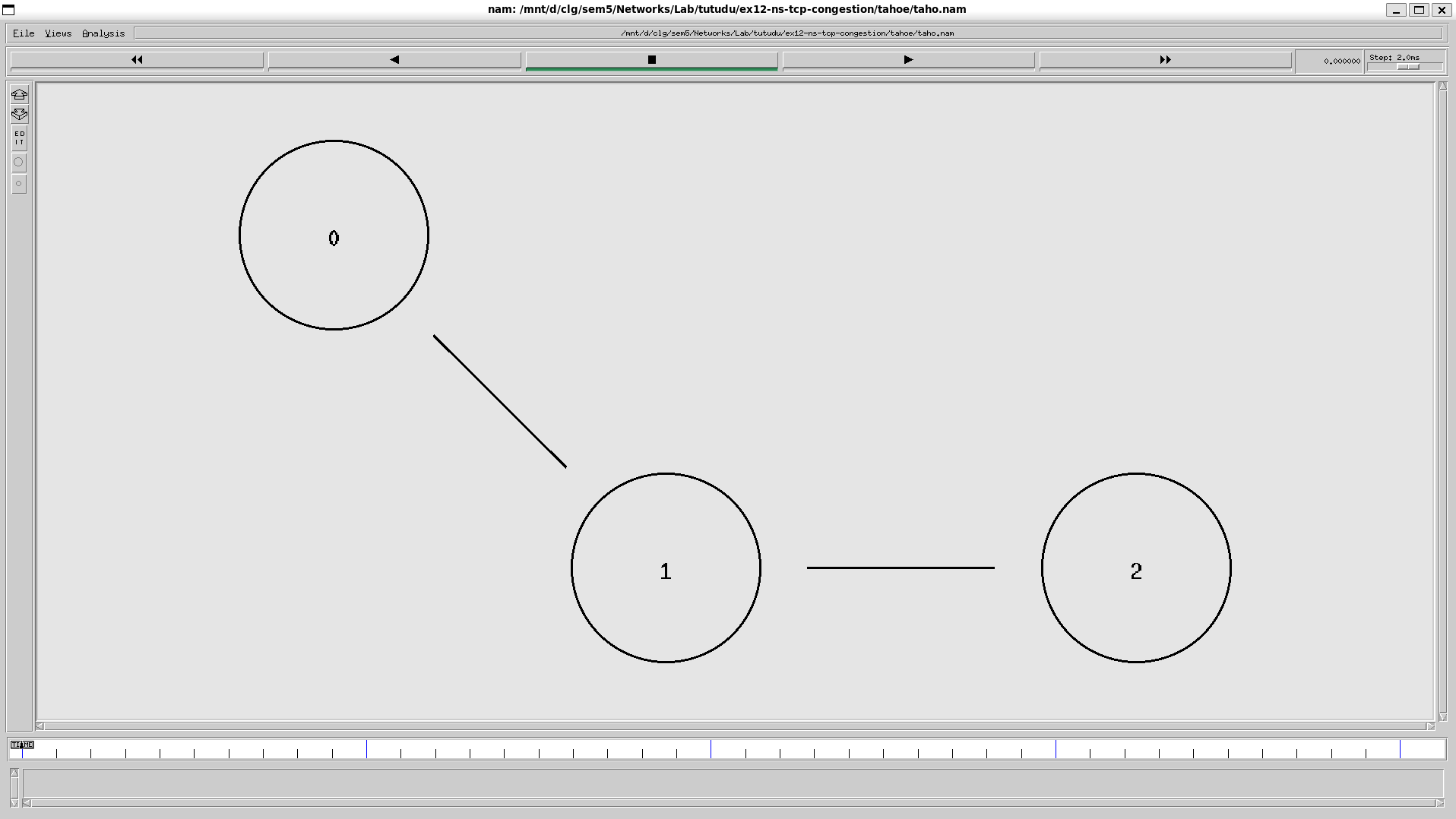
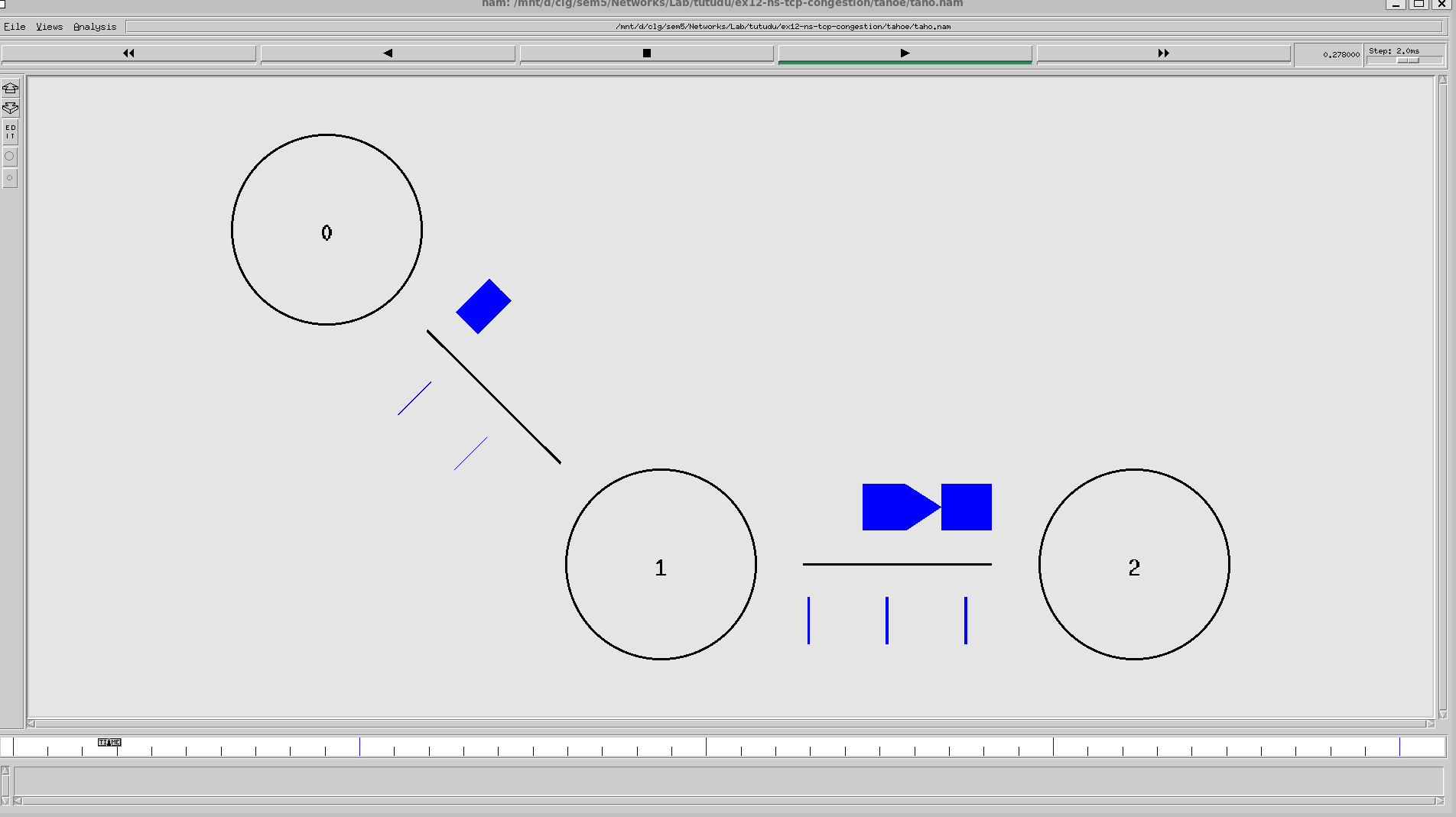
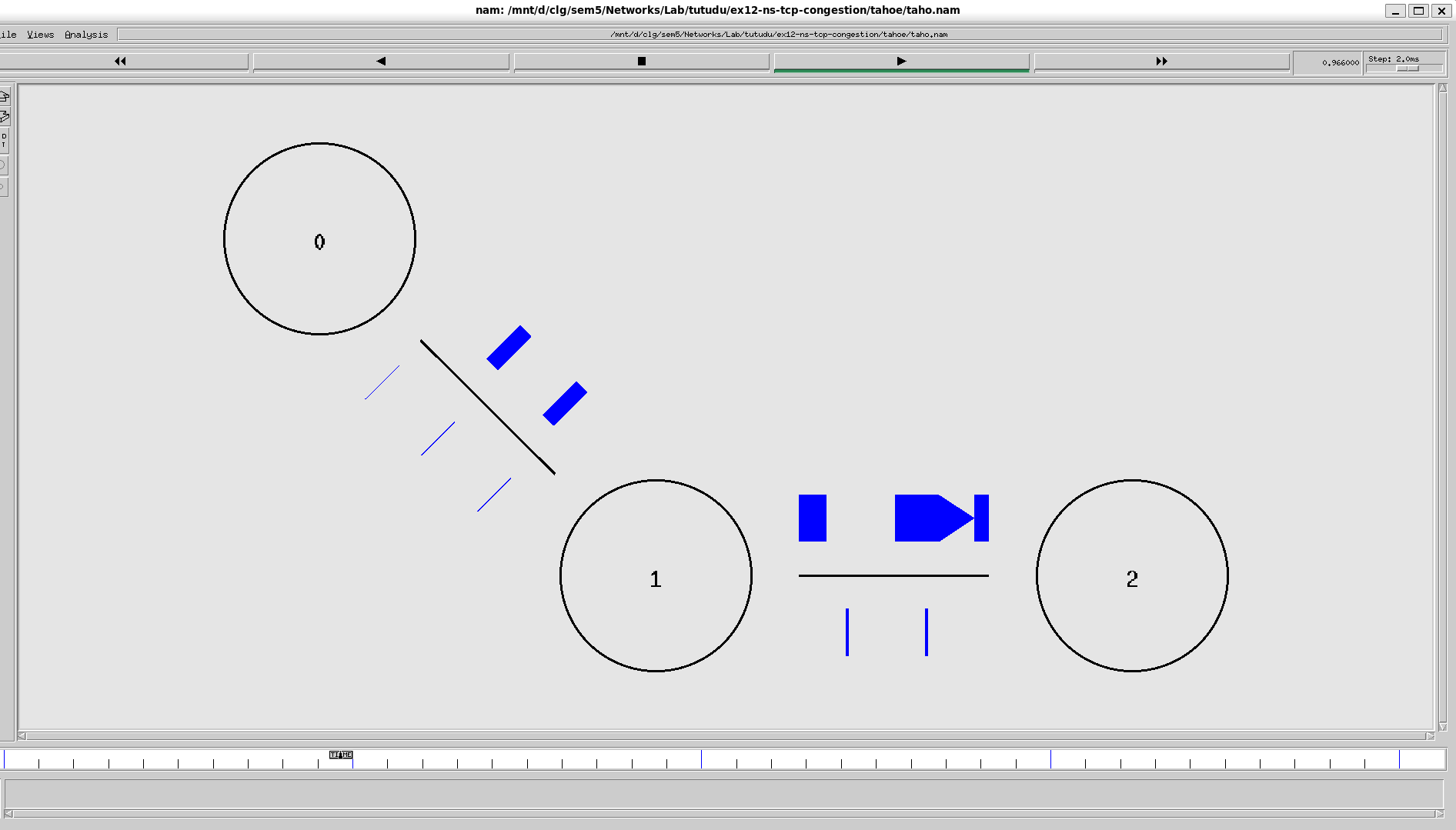
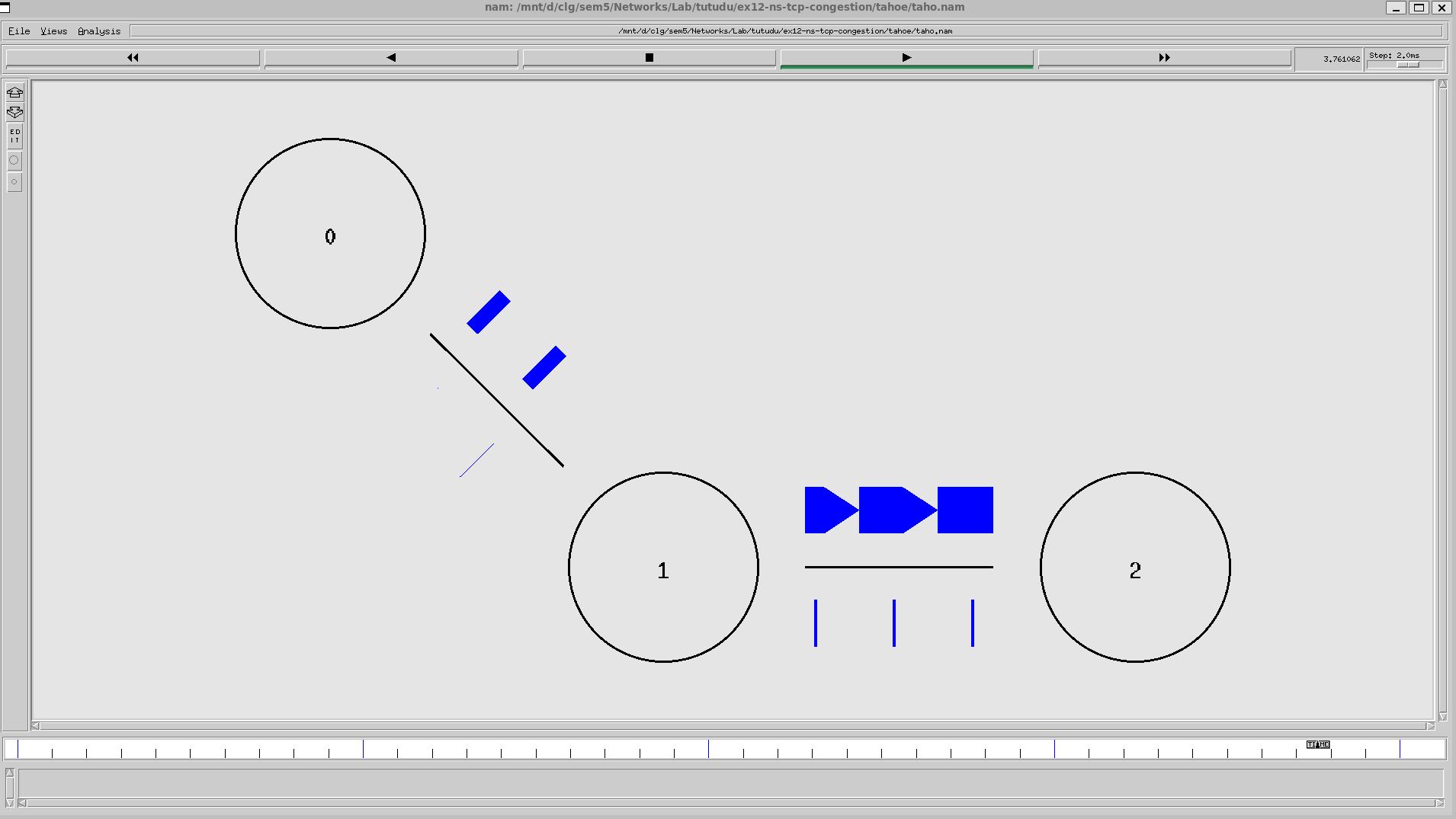
set outfile [open "taho.xg" w]

$ns at 0.0 "plotWindow $tcp $outfile"

# Run the simulation

$ns run

**Output** ****

**newreno.tcl**

# Create a simulator object

set ns [new Simulator]

# Open the NAM file and the trace file

set nf [open basic1.nam w]

$ns namtrace-all $nf

set tf [open basic1.tr w]

$ns trace-all $tf

# Define a 'finish' procedure

proc finish {} {

global ns nf tf

$ns flush-trace

close $nf

close $tf

exec nam basic1.nam &

exec xgraph reno.xg &

exit 0

}

# Create the network nodes

set n0 [$ns node]

set n1 [$ns node]

set n2 [$ns node]

# Create duplex links

$ns duplex-link $n0 $n1 10Mb 10ms DropTail

$ns duplex-link $n1 $n2 800Kb 50ms DropTail

# Set queue limit for the router

$ns queue-limit $n1 $n2 7

# Visual hints for NAM

$ns color 0 Red

$ns duplex-link-op $n0 $n1 orient right

$ns duplex-link-op $n1 $n2 orient right

$ns duplex-link-op $n1 $n2 queuePos 0.5

# Create and configure TCP sending agent

set tcp [new Agent/TCP/Reno]

$tcp set class\_ 0

$tcp set window\_ 100

$tcp set packetSize\_ 960

$ns attach-agent $n0 $tcp

# Create and attach TCP receive agent (sink)

set sink [new Agent/TCPSink]

$ns attach-agent $n2 $sink

$ns connect $tcp $sink

# Schedule the data flow

set ftp [new Application/FTP]

$ftp attach-agent $tcp

$ns at 0.0 "$ftp start"

$ns at 10.0 "finish"

# Procedure to plot the congestion window

proc plotWindow {tcpSource outfile} {

global ns

set now [$ns now]

set cwnd [$tcpSource set cwnd\_]

puts $outfile "$now $cwnd"

$ns at [expr $now + 0.1] "plotWindow $tcpSource $outfile"

}

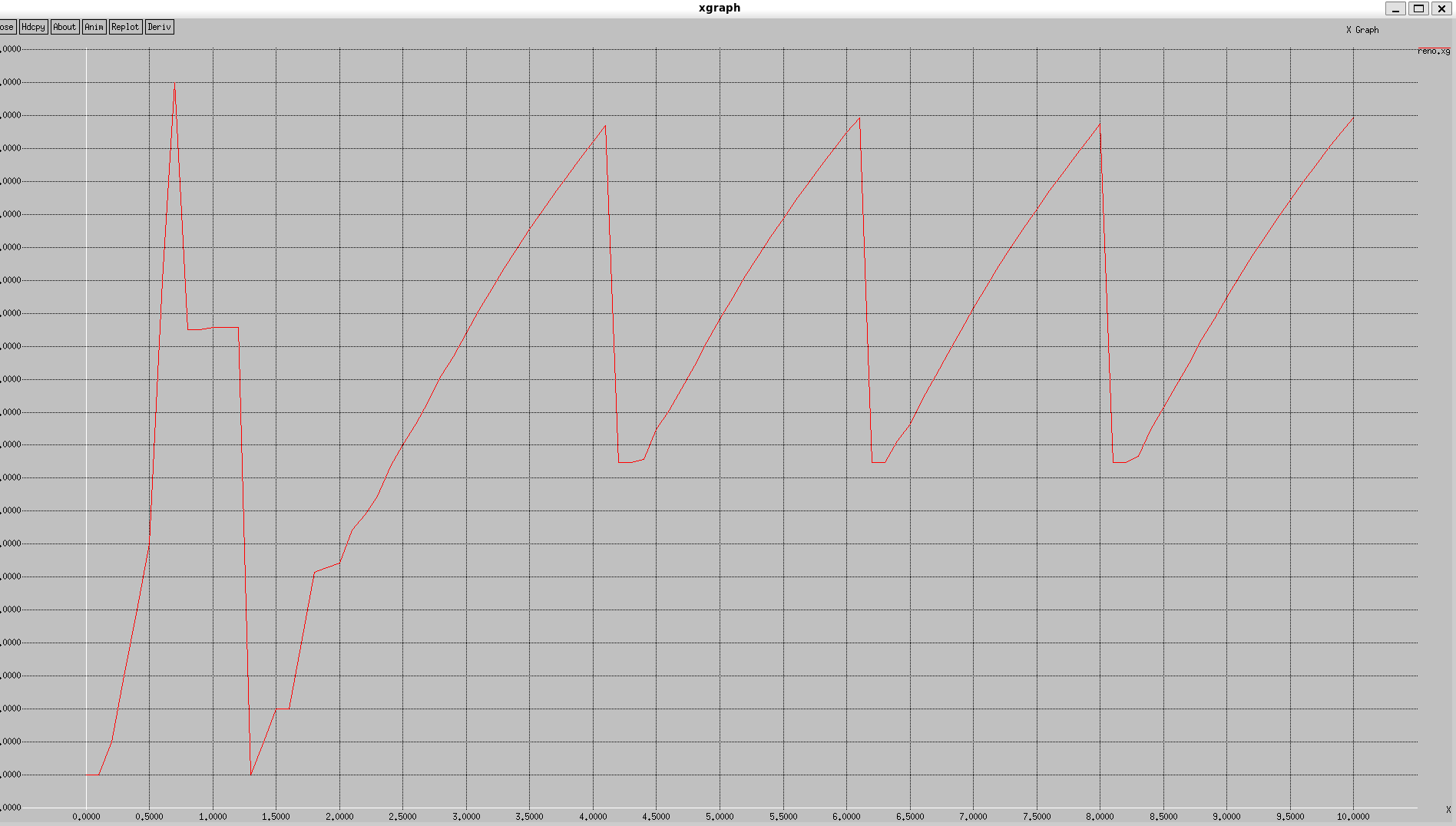
# Open file to log congestion window

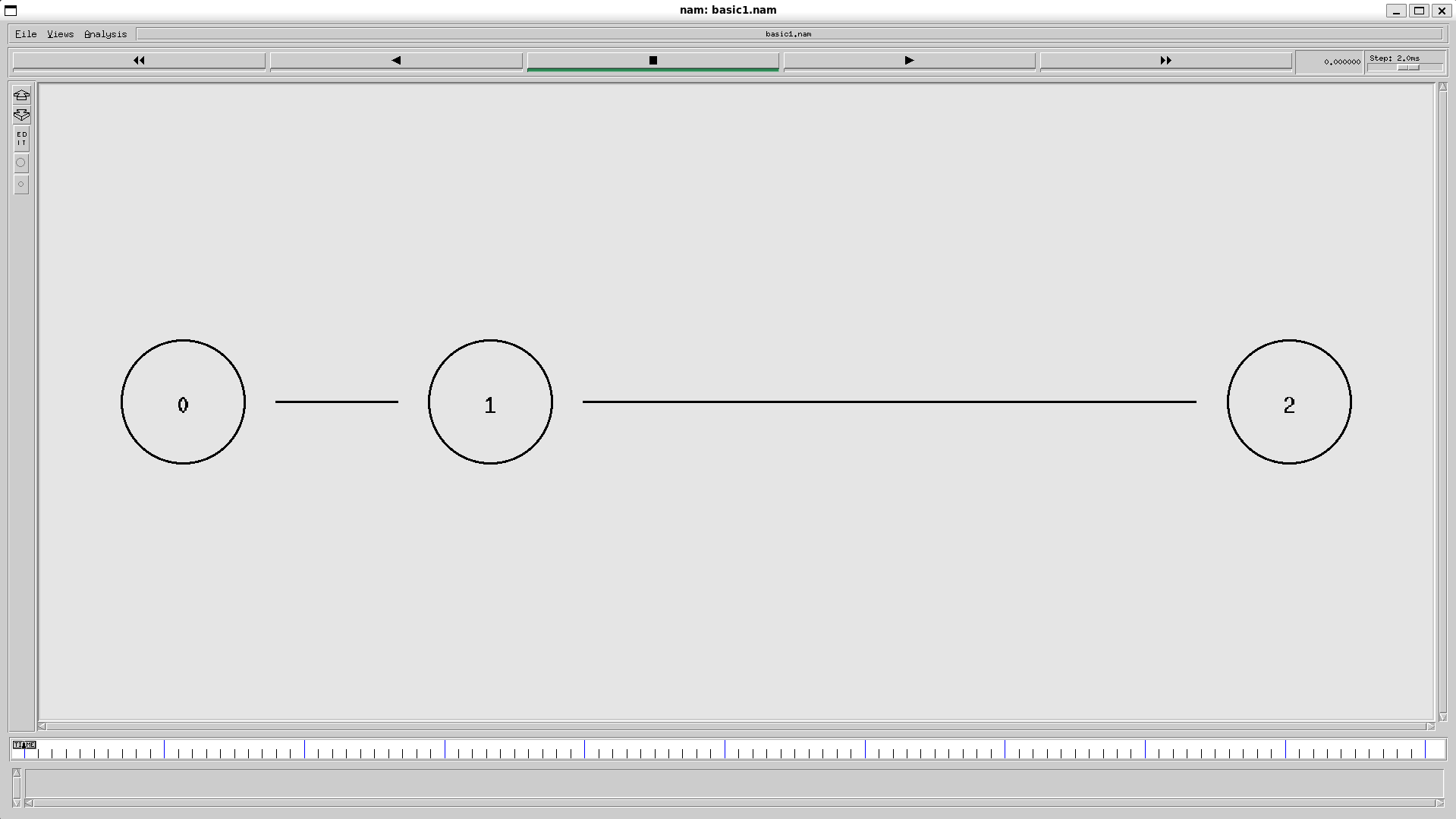
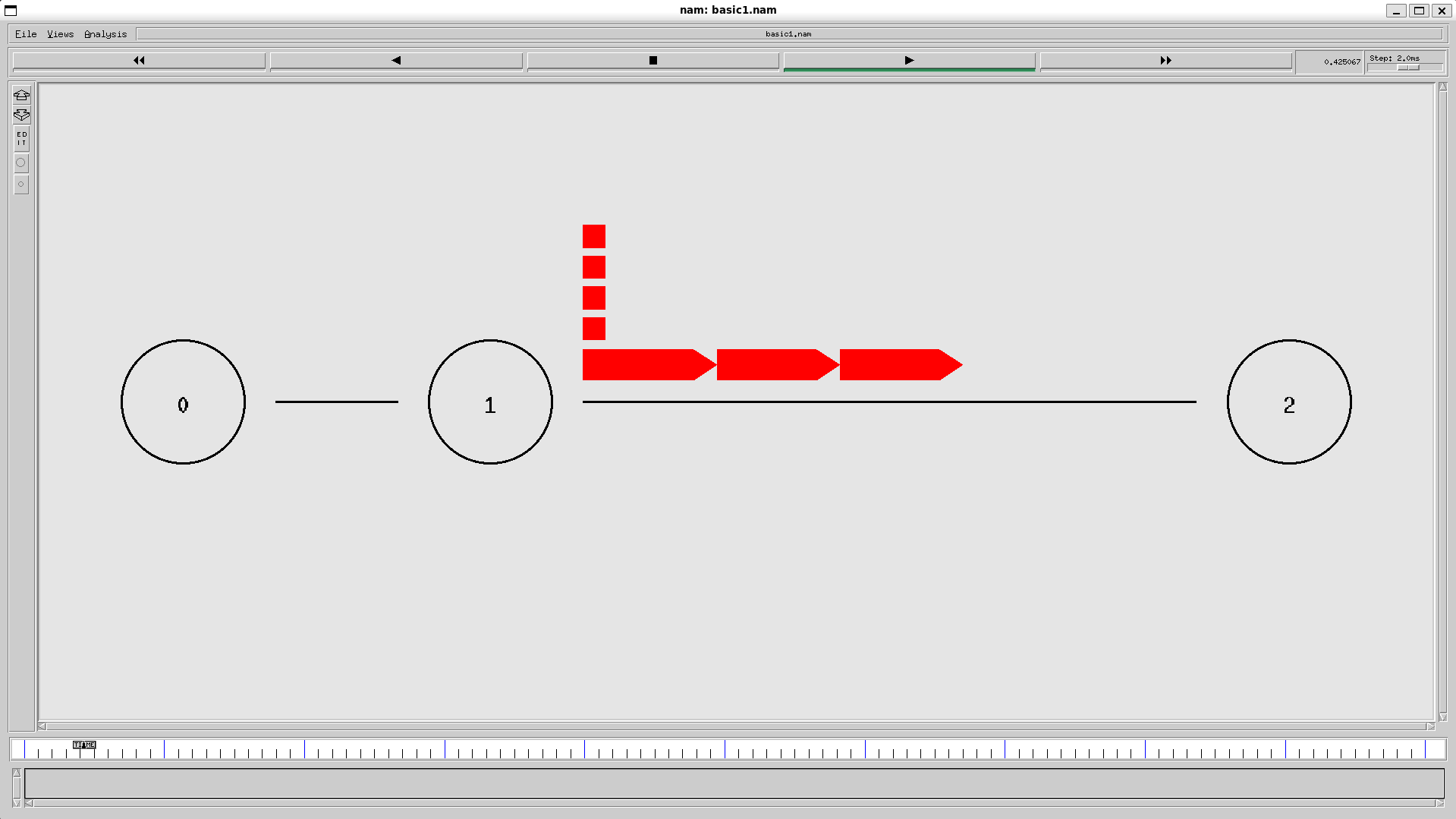
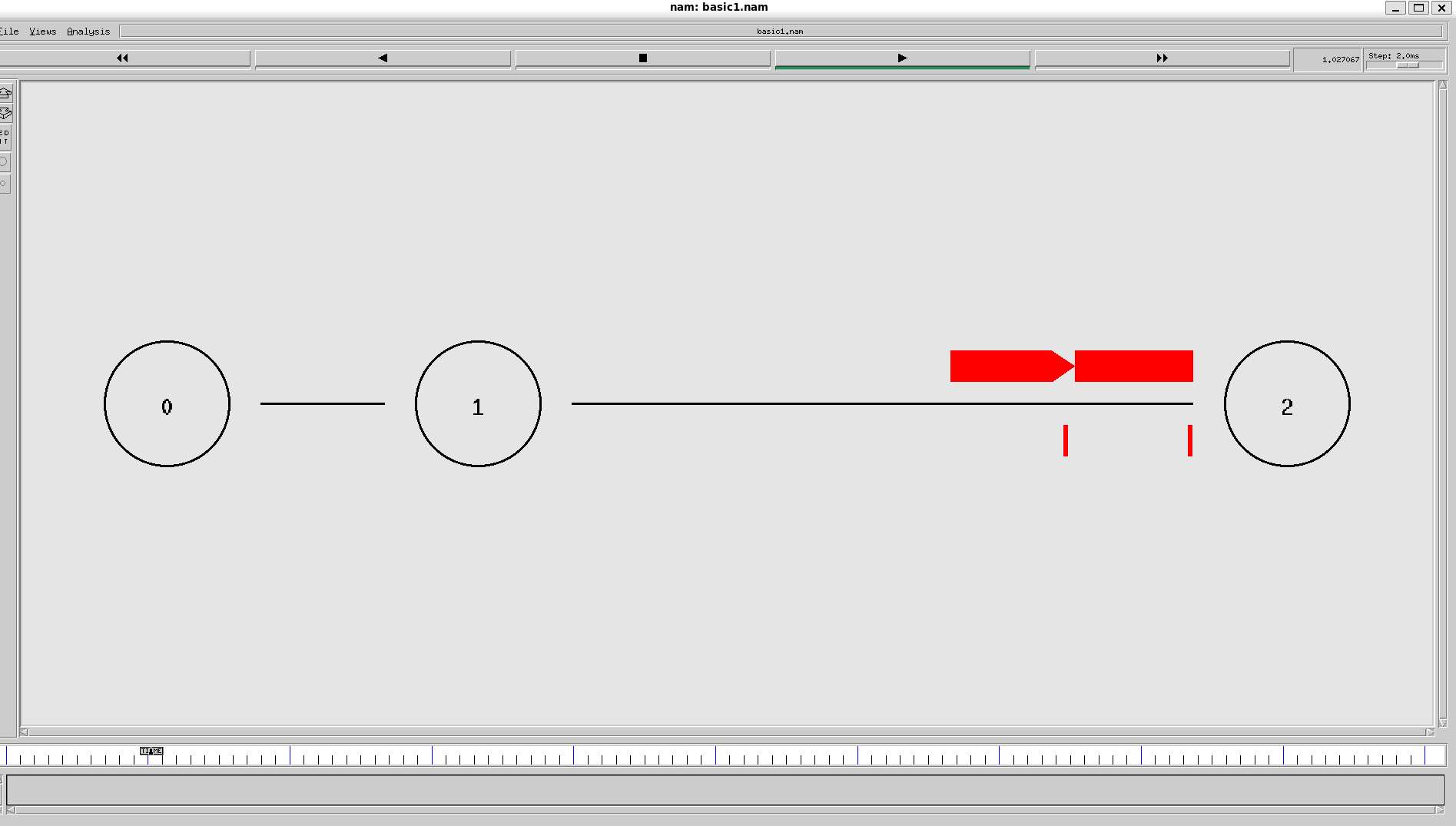
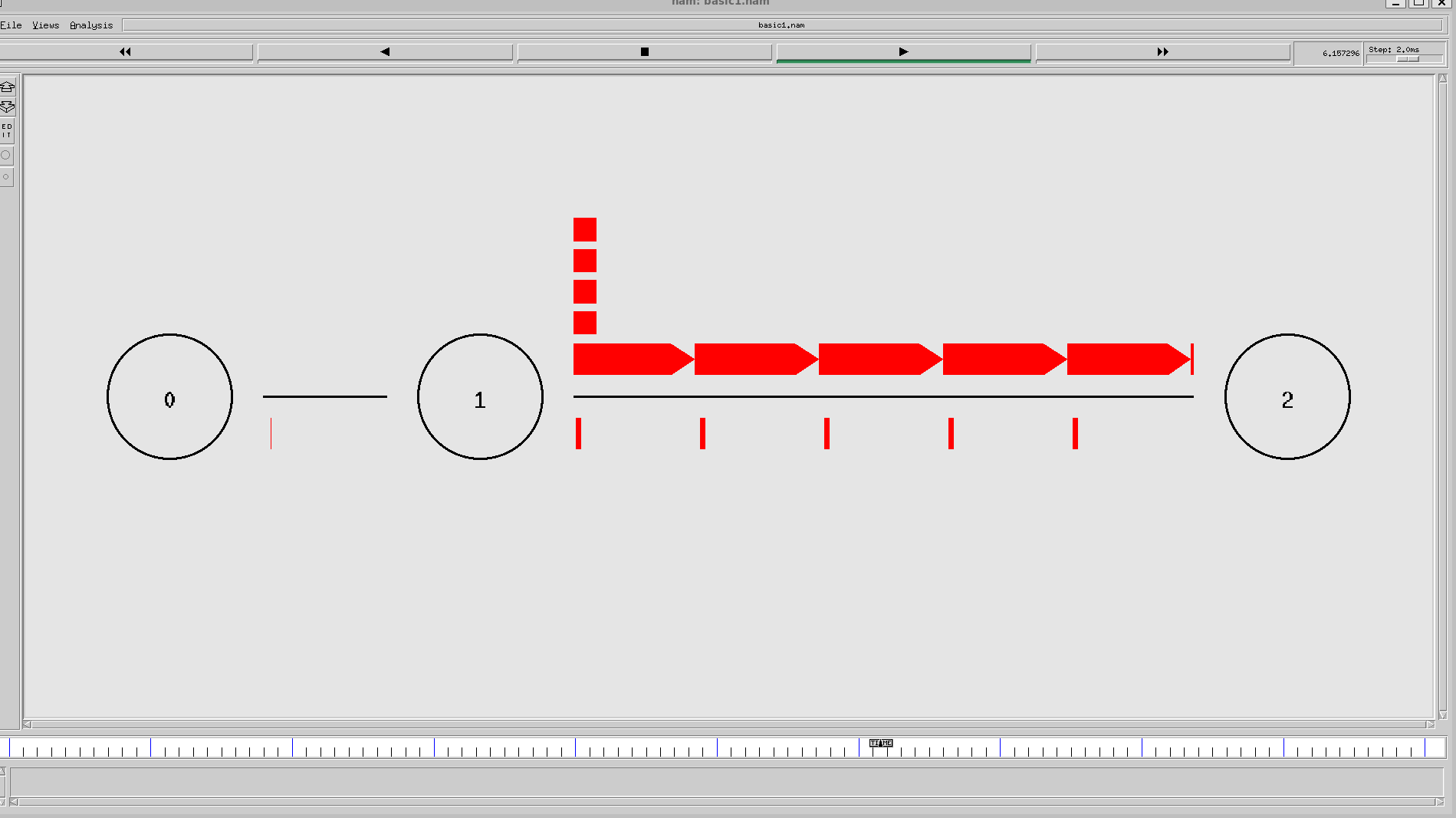
set outfile [open "reno.xg" w]

$ns at 0.0 "plotWindow $tcp $outfile"

# Run the simulation

$ns run

**Output** ****

**stack.tcl**

# Create a simulator object

set ns [new Simulator]

# Define different colors for data flows (for NAM)

$ns color 1 Blue

$ns color 2 Red

# Open the NAM trace file

set nf [open taho.nam w]

$ns namtrace-all $nf

# Open the trace file for general simulation data

set tf [open taho.tr w]

$ns trace-all $tf

# Define a 'finish' procedure

proc finish {} {

global ns nf tf

$ns flush-trace

# Close the NAM trace file

close $nf

close $tf

# Execute NAM on the trace file

exec nam taho.nam &

exec xgraph taho.xg &

exit 0

}

# Create three nodes

set n0 [$ns node]

set n1 [$ns node]

set n2 [$ns node]

# Create links between the nodes

$ns duplex-link $n0 $n1 10Mb 10ms DropTail

$ns duplex-link $n1 $n2 2Mb 10ms DropTail

# Set Queue Size of link (n0-n1) to 10 packets

$ns queue-limit $n0 $n1 10

# Position nodes for visualization in NAM

$ns duplex-link-op $n0 $n1 orient right-down

$ns duplex-link-op $n1 $n2 orient right

# Monitor the queue for link (n0-n1). (for NAM)

$ns duplex-link-op $n0 $n1 queuePos 0.5

# Setup a TCP connection using the default TCP agent

set tcp [new Agent/TCP] ;# Use default TCP agent

$tcp set tcpType\_ "Tahoe" ;# Set the congestion control algorithm to Tahoe

$tcp set window\_ 10 ;# Set the window size (e.g., 10 packets)

$tcp set packetSize\_ 1000 ;# Set the packet size (e.g., 1000 bytes)

$tcp set timeout\_ 1.0 ;# Set the timeout (e.g., 1.0 seconds)

$ns attach-agent $n0 $tcp

# Create a TCP Sink on the destination node

set sink [new Agent/TCPSink]

$ns attach-agent $n2 $sink

$ns connect $tcp $sink

$tcp set fid\_ 1

# Setup an FTP application over the TCP connection

set ftp [new Application/FTP]

$ftp attach-agent $tcp

# Schedule the FTP events

$ns at 0.1 "$ftp start"

$ns at 4.0 "$ftp stop"

# Call the finish procedure after 5 seconds of simulation time

$ns at 5.0 "finish"

# Procedure to plot the congestion window

proc plotWindow {tcpSource outfile} {

global ns

set now [$ns now]

set cwnd [$tcpSource set cwnd\_]

# Record the data in a file

puts $outfile "$now $cwnd"

$ns at [expr $now + 0.1] "plotWindow $tcpSource $outfile"

}

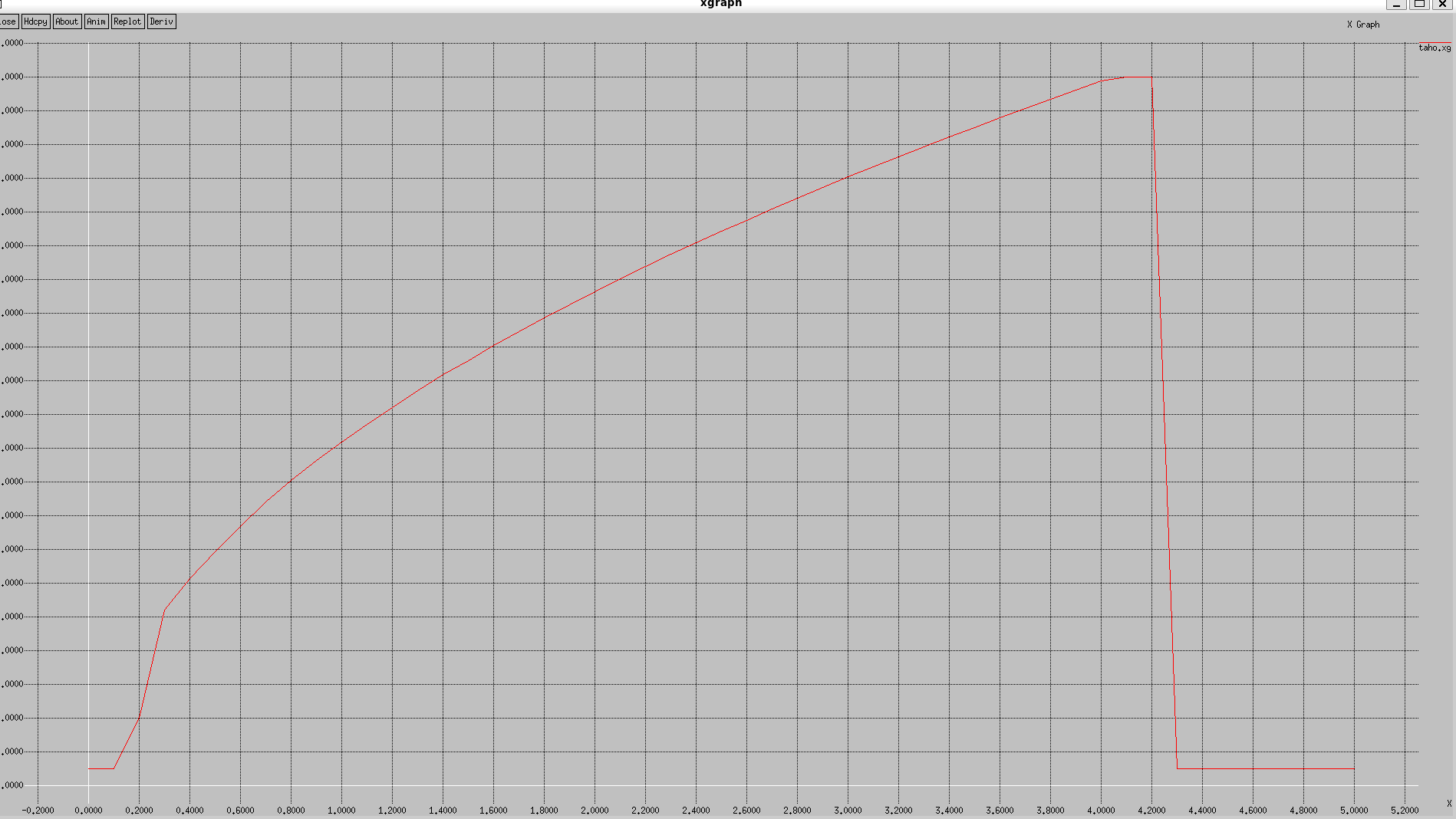
# Prepare to record the congestion window

set outfile [open "taho.xg" w]

$ns at 0.0 "plotWindow $tcp $outfile"

# Run the simulation

$ns run

**Output** ****

