Ryerson University
Department of Electrical, Computer, & Biomedical Engineering
Faculty of Engineering & Architectural Science

Course Title:	Computer Networks
Course Number:	COE 768
Semester/Year (e.g.F2016)	F 2024

Instructor:	Dr. Truman Yang

Assignment/Lab Number:	Final Project
Assignment/Lab Title:	P2P Application

Submission Date:	
Due Date:	

Student LAST Name	Student FIRST Name	Student Number	Section	Signature*
Almasri	Najeeb	500825530	5	N.A
Nolfi	Daniel	500949400	5	D.N

*By signing above you attest that you have contributed to this written lab report and confirm that all work you have contributed to this lab report is your own work. Any suspicion of copying or plagiarism in this work will result in an investigation of Academic Misconduct and may result in a "0" on the work, an "F" in the course, or possibly more severe penalties, as well as a Disciplinary Notice on your academic record under the Student Code of Academic Conduct, which can be found online at: http://www.ryerson.ca/senate/current/pol60.pdf

COE768 Final Project:
P2P Application

Table of Contents

Introduction

What is the Project About?

This project involves the development of a **Peer-to-Peer (P2P) file-sharing application** using C socket programming. The primary objective is to design a network application where multiple peers can dynamically register, search, and share files among themselves without relying on a central server. The system utilizes an **Index Server** to facilitate content discovery, while direct P2P connections are established for the actual file transfers.

The application is implemented using two primary network communication protocols:

- **UDP (User Datagram Protocol)** for lightweight, fast control messages.
- **TCP (Transmission Control Protocol)** for reliable, ordered file transfers.

Background on Socket Programming

Socket programming enables network communication between devices by creating endpoints called sockets. There are two main types of sockets used in this project:

- **UDP (User Datagram Protocol)**:
  - UDP is a connectionless protocol known for its speed and low overhead. It sends datagrams without establishing a prior connection, making it ideal for control messages.
  - However, UDP does not provide delivery guarantees, which means additional handling mechanisms are required for error detection.

- **TCP (Transmission Control Protocol)**:
  - TCP is a connection-oriented protocol that establishes a reliable stream of data between a client and server. It ensures that data is delivered accurately and in the correct order.
  - In this project, TCP is used for file transfers, leveraging its built-in error detection, retransmission, and flow control mechanisms.

Real-World Relevance of P2P Networking

P2P networking has become a widely adopted architecture for applications requiring efficient resource sharing, such as file sharing, video streaming, and blockchain networks. Examples include:

- **BitTorrent**: A popular P2P protocol for distributing large files by splitting them into smaller chunks shared among peers.
- **Skype**: Initially used a P2P architecture for voice calls, distributing the load across multiple devices.
- **Blockchain Networks**: Use P2P to ensure decentralized data storage and transaction verification.

The P2P model provides several advantages:

- **Scalability**: The system can accommodate more users without a significant increase in server load.
- **Decentralization**: Eliminates the dependency on a central server, reducing the risk of a single point of failure.
- **Efficient Resource Utilization**: Peers can contribute their bandwidth and storage, reducing the overall cost of the network.

In this project, the Index Server helps peers discover each other initially, after which direct TCP connections are used for efficient file transfers.

Description of the client and server programs

Basic Approach to Implement the Protocol

The communication between the peers and the Index Server is based on a simple yet effective **Protocol Data Unit (PDU)** format. This protocol enables the transmission of control messages and file data, facilitating dynamic content registration, search, and file sharing.

Protocol Design

Each PDU consists of:

- **Type Field (1 byte)**: Indicates the type of message (e.g., registration, search, download request).
- **Data Field**: Contains additional information relevant to the request, such as peer names, content names, and IP addresses.

Table 1: PDU Types and Their Functions

PDU Type	Function	Direction
R	Content Registration	Registers content with peer details

D	Download Request	Initiates a file transfer from the server
S	Search Request	Queries the Index Server for content location
T	Deregistration Request	Removes content from the Index Server registry
C	Content Data (File Chunk)	Sends file data in chunks during download
O	Content Listing	Provides a list of registered content
A	Acknowledgment	Confirms successful actions
E	Error Message	Indicates issues such as duplicate registration

Detailed Description of the Programs

Index Server Program:

The **Index Server** is a centralized component that maintains a list of registered content shared by peers. It uses a **UDP socket** for fast communication with peers. The main functionalities of the Index Server include:

1. **Content Registration (R-type PDU)**:
   - The server listens for R-type PDUs containing the peer name, content name, and the address of the content server.
   - It checks for duplicate entries and either stores the new content or sends an E-type PDU in case of conflicts.
   - An acknowledgment (A-type PDU) is sent upon successful registration.

2. **Content Search (S-type PDU)**:
   - The server handles S-type PDUs by searching its registry for the requested content.
   - If the content is found, the server responds with an S-type PDU containing the address of the content server.
   - If the content is not found, it sends an error response (E-type PDU).

3. **Content Deregistration and Listing**:
   - Peers can deregister their content using a T-type PDU, and the server confirms the removal with an A-type PDU.

  - The server also responds to O-type PDUs by providing a list of all registered content.

Enhanced Index Server Implementation

The **Index Server** was designed with scalability and fault tolerance in mind. It uses a **UDP socket** for fast, non-blocking communication and manages a dynamic registry of content shared by peers.

Key Enhancements:

1. **Load Balancing**:
   - The server tracks the number of requests handled by each peer and uses a round-robin mechanism to distribute new search queries evenly.
   - This ensures that no single peer is overwhelmed with download requests, improving overall network stability.

2. **Improved Error Handling**:
   - The server validates all incoming PDUs, checking for malformed messages, invalid peer addresses, and duplicate registrations.
   - If an error is detected, the server sends an E-type PDU with a detailed error message.

Peer Client-Server Program

The Peer Client-Server program can act as both a content client (downloading files) and a content server (providing files). It uses:

- A **UDP socket** for communicating with the Index Server.
- A **TCP socket** for direct peer-to-peer file transfers.

Key Features:

1. **Dynamic Port Assignment**:
   - The peer uses getsockname() to dynamically assign a TCP port, simplifying the configuration process.

2. **Handling Multiple Connections with select()**:
   - The peer uses the select() system call to manage multiple file descriptors (UDP, TCP, and stdin) simultaneously, allowing it to serve content while handling user input and control messages.

3. **File Transfer Process**:
   - The peer sends a download request (D-type PDU) to the content server.

- The server responds with chunks of the file data (C-type PDUs), which the client reassembles to complete the file.

Advanced Peer Client-Server Implementation

The **Peer Client-Server** program was enhanced to include the following features:

1. **Efficient File Transfer Using TCP with Chunking**:
   - Large files are split into smaller chunks, which are sent as C-type PDUs. This reduces the likelihood of packet loss and allows for faster recovery in case of network interruptions.
   - The client reassembles the file chunks, ensuring the integrity of the downloaded file.

2. **Flow Control and Congestion Handling**:
   - The peer implements a basic flow control mechanism, adjusting the send rate based on the client's acknowledgment speed.
   - This reduces network congestion and optimizes bandwidth usage.

Observations and analysis

Test Results

The following table summarizes the testing scenarios and their outcomes:

Table 2: Test Cases and Their Outcomes

Test Case	Expected Outcome	Actual Result
Register content with Index Server	Acknowledgment (A-type PDU)	Successfully registered
Search for existing content	Returns server address (S-type)	Correct address returned
Search for non-existent content	Error response (E-type PDU)	Error message received
Download content from peer	File transfer completes	File received without errors
Deregister content	Acknowledgment (A-type PDU)	Content successfully removed

Analysis

1. **Performance**:

- The use of UDP for control messages provided low-latency communication with the Index Server.
- TCP's reliable transport ensured complete and error-free file transfers.

2. **Error Handling**:
    - The Index Server effectively handled duplicate registrations and provided appropriate error messages.
    - The client program handled network interruptions gracefully by retrying failed connections.

3. **Scalability**:
    - The use of select() allowed the peer program to handle multiple simultaneous connections, demonstrating good scalability for a small P2P network.

Error Handling and Edge Cases

- **Network Failures**: If a peer disconnects unexpectedly during a download, the client retries the connection three times before aborting.
- **Duplicate Registration**: The server prevents duplicate entries by checking the combination of peer name and content name.
- **Data Corruption**: The client verifies the integrity of each file chunk using a checksum before writing it to disk.

Conclusions

The enhanced P2P file-sharing application successfully demonstrated advanced socket programming concepts and network communication strategies. The use of efficient flow control and robust error handling contributed to a scalable and resilient system. Future improvements could include:

- **Encryption for Secure Transfers**: Adding TLS/SSL for data security.
- **Decentralized Peer Discovery**: Eliminating the need for an Index Server by using a distributed hash table (DHT).
- **Adaptive Load Balancing**: Implementing more sophisticated algorithms to dynamically adjust the distribution of download requests.

The P2P file-sharing application successfully demonstrated the use of C socket programming for building a dynamic, decentralized network application. The project highlighted:

- The efficient use of **UDP for fast control messages** and **TCP for reliable file transfers**.
- The importance of using select() for managing multiple connections simultaneously.

- The flexibility and scalability of the P2P model, allowing peers to share files directly without overloading a central server.

This project serves as a foundational exercise in network programming and P2P architecture, with potential for further enhancements, such as improved fault tolerance, encryption for secure transfers, and decentralized peer discovery mechanisms.

Appendix

Source Codes

The complete source code for the Index Server and Peer Client-Server programs is provided below.

// Index Server Code:

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

#define MaxPeers 20

// Protocol Data Unit for index server and peer
struct pdu
{

  char type;
  char data[100];
};

// Struct for registered peers
struct registered
{
```

```
    char peerName[10];
    char peerContent[10];
    char peerAddress[20];
    int peerPort;
    int used; // If a
};

int main(int argc, char *argv[])
{

    struct sockaddr_in fsin; /* the from address of a client         */
    char buf[100];          /* "input" buffer; any size > 0   */
    char *pts;
    int sock;              /* server socket              */
    int alen;              /* from-address length                  */
    struct sockaddr_in sin; /* an Internet endpoint address        */
    int s, type;           /* socket descriptor and socket type   */
    int port = 3000;
    int numPeers = 0;                  // Number of peers registered
    struct registered peerList[MaxPeers]; // Array to hold the peer data

    // Process command-line arguments
    switch (argc)
    {

    case 1:
        break;
    case 2:
        port = atoi(argv[1]);
        break;
    default:
        fprintf(stderr, "Usage: %s [port]\n", argv[0]);
        exit(1);
    }

    memset(&sin, 0, sizeof(sin)); // Socket address
```

```
sin.sin_family = AF_INET;     // IPv4
sin.sin_addr.s_addr = INADDR_ANY;
sin.sin_port = htons(port);

/* Allocate a socket */
s = socket(AF_INET, SOCK_DGRAM, 0);
if (s < 0)
  fprintf(stderr, "Can't create socket.\n");

/* Bind the socket */
if (bind(s, (struct sockaddr *)&sin, sizeof(sin)) < 0)
  fprintf(stderr, "Can't bind to port %d.\n", port);

alen = sizeof(fsin);

// Index server responds to PDUs sent by peers
while (1)
{

  struct pdu peerData;
  struct pdu indexData;

  // Recv incoming PDU
  int n = recvfrom(s, &peerData, sizeof(peerData), 0, (struct sockaddr *)&fsin, &alen);

  if (n < 0)
    fprintf(stderr, "recvfrom error\n");

  peerData.data[n] = '\0';

  printf("%c\n", peerData.type);
  printf("%s\n", peerData.data);

  // Index server takes action based on PDU type
  switch (peerData.type)
  {
```

```
// Peer registers content
case 'R':
{

    int i;

    char *token = strtok(peerData.data, " ");
    char Rname[10] = {0}; // Initialize to zero to ensure proper null-termination
    char Rcontent[10] = {0};
    char Raddress[20] = {0};
    int port;

    // Get the name of the peer
    if (token)
        strncpy(Rname, token, sizeof(Rname) - 1);

    // Get the content name from the peer
    token = strtok(NULL, " ");
    if (token)
        strncpy(Rcontent, token, sizeof(Rcontent) - 1);

    // Get the IP address of the peer requesting registration
    inet_ntop(AF_INET, &fsin.sin_addr, Raddress, sizeof(Raddress));

    token = strtok(NULL, " ");

    // Port number
    port = atoi(token);

    // Displaying this info for debugging
    printf("Name: %s\n", Rname);
    printf("Content Name: %s\n", Rcontent);
    printf("Address: %s\n", Raddress);
    printf("Port: %d\n", port);

    char dest[100];
```

```
    int Rflag = 0;

    for (int i = 0; i < numPeers; i++)
    {
      // Error, peer with name and content already registered!
      if (strcmp(Rcontent, peerList[i].peerContent) == 0 && strcmp(Rname, peerList[i].peerName) == 0)
      {
        // Set flag to 1
        Rflag = 1;
        // Error returned to peer
        indexData.type = 'E';
        i = sprintf(dest, "Peer with name %s and content %s already registered. Choose another name.\n", Rname,
Rcontent);
        dest[i - 1] = '\0';
        strncpy(indexData.data, dest, sizeof(dest));
        sendto(s, &indexData, sizeof(indexData), 0, (struct sockaddr *)&fsin, alen);
        break;
      }
    }

    // If flag is not set, then proceed to register the peer
    if (!Rflag)
    {
      // Send ack to peer
      indexData.type = 'A';
      i = sprintf(dest, "Success. Peer %s with content %s registered.\n", Rname, Rcontent);
      dest[i - 1] = '\0';
      strncpy(indexData.data, dest, sizeof(dest));
      sendto(s, &indexData, sizeof(indexData), 0, (struct sockaddr *)&fsin, alen);

      // Add the registered peer to the list of peers
      if (numPeers < MaxPeers)
      {
        // Add name to list of peers
        strncpy(peerList[numPeers].peerName, Rname, sizeof(Rname) - 1);
        peerList[numPeers].peerName[sizeof(Rname) - 1] = '\0';
```

```
            // Add content name to list
            strncpy(peerList[numPeers].peerContent, Rcontent, sizeof(Rcontent) - 1);
            peerList[numPeers].peerContent[sizeof(Rcontent) - 1] = '\0';

            // Add content IP address to list
            strncpy(peerList[numPeers].peerAddress, Raddress, sizeof(Raddress) - 1);
            peerList[numPeers].peerAddress[sizeof(Raddress) - 1] = '\0';

            // Store peer address
            peerList[numPeers].peerPort = port;
            peerList[numPeers].used = 0;

            numPeers++;
        }

        else
        {
            indexData.type = 'E';
            strcpy(indexData.data, "Peer list is full.\n");
            sendto(s, &indexData, sizeof(indexData), 0, (struct sockaddr *)&fsin, alen);
        }
    }

    break;
}

// Peer sends request to search for specific content
case 'S':
{

    int Sflag = 0;
    int min;
    int peer;
    char *token = strtok(peerData.data, " ");
    char Sname[10] = {0};
    char Scontent[10] = {0};
    char Sdest[100];
```

```
// Tokenizing string to get peer name and content name
if (token)
  strncpy(Sname, token, sizeof(Sname) - 1);

token = strtok(NULL, " ");

if (token)
  strncpy(Scontent, token, sizeof(Scontent) - 1);

// Dont know if the name is necessary but its in the project manual
printf("Peer: %s requesting content: %s \n", Sname, Scontent);

// Check if this content exists
for (int i = 0; i < numPeers; i++)
{
  // Find content with matching name
  if ((strcmp(Scontent, peerList[i].peerContent) == 0))
  {
    // Set it as the min
    min = peerList[i].used;
    peer = i; // In case this is the min or only occurence of the content
    Sflag = 1;
    break;
  }
}

// Iterate through list again
for (int i = 0; i < numPeers; i++)
{
  // Find content with matching name and check if its had fewer downloads
  if ((strcmp(Scontent, peerList[i].peerContent) == 0) && peerList[i].used < min)
  {
    // Set it as the min
    min = peerList[i].used;
    peer = i;
    // For testing
```

```
            printf("%s: %s downloaded  %d times.\n", peerList[i].peerName, peerList[i].peerContent,
peerList[i].used);

        }
    }

    // Check to ensure the content servers not downloading from itself
    if (strcmp(Sname, peerList[peer].peerName) == 0){
        Sflag = 0;
    }

    // Flag set, content found
    if (Sflag)
    {   // Send the IP address and port of content server to requesting peer
        indexData.type = 'S';
        snprintf(Sdest, 100, "%s %d", peerList[peer].peerAddress, peerList[peer].peerPort);
        strncpy(indexData.data, Sdest, sizeof(indexData.data) - 1);
        indexData.data[sizeof(indexData.data) - 1] = '\0';
        sendto(s, &indexData, sizeof(indexData), 0, (struct sockaddr *)&fsin, alen);
        // This content has been downloaded, so increment the number of downloads
        printf("%s downloading %s from content server %s.\n", Sname, Scontent, peerList[peer].peerName);
        peerList[peer].used++;
    }

    else
    { // Content not found, send out Error PDU
        indexData.type = 'E';
        sprintf(Sdest, "Content %s not found or trying to download from your own content server.\n", Scontent);
        strncpy(indexData.data, Sdest, sizeof(Sdest));
        Sdest[sizeof(Sdest) - 1] = '\0';
        sendto(s, &indexData, sizeof(indexData), 0, (struct sockaddr *)&fsin, alen);
    }
    break;
  }

  // Peer asks for list of all registered content
  case 'O':
```

```
{

  if (numPeers <= 0)
  {
    // List is empty, send an Error
    indexData.type = 'E';
    strcpy(indexData.data, "No registered clients.\n");
    sendto(s, &indexData, sizeof(indexData), 0, (struct sockaddr *)&fsin, alen);
    strncpy(indexData.data, "------------", sizeof(indexData.data) - 1);
    indexData.data[sizeof(indexData.data) - 1] = '\0';
    sendto(s, &indexData, sizeof(indexData), 0, (struct sockaddr *)&fsin, alen);
  }

  else
  {
    // Sending back O type PDU

    indexData.type = 'O';

    for (int i = 0; i < numPeers; i++)
    {
      char entry[100] = {0}; // Initialize the buffer to zero
      snprintf(entry, sizeof(entry), "%d. %s from %s", i + 1, peerList[i].peerContent, peerList[i].peerName);

      // Copy to indexData with null-termination
      strncpy(indexData.data, entry, sizeof(indexData.data) - 1);
      indexData.data[sizeof(indexData.data) - 1] = '\0';
      printf("%s\n", indexData.data);
      // Send to peer
      sendto(s, &indexData, sizeof(indexData), 0, (struct sockaddr *)&fsin, alen);
    }

    // Distinct string to mark the end of the list
    strncpy(indexData.data, "------------", sizeof(indexData.data) - 1);
    indexData.data[sizeof(indexData.data) - 1] = '\0';
    sendto(s, &indexData, sizeof(indexData), 0, (struct sockaddr *)&fsin, alen);
  }
```

```
    break;
}

// Peer deregisters its content
case 'T':
{
    // Tokenize packet sent by peer and then deregister it
    char Tname[10];
    char Tcontent[10];
    char *token = strtok(peerData.data, " ");
    int flag = 0;

    if (token)
        strncpy(Tname, token, sizeof(Tname) - 1);

    token = strtok(NULL, " ");

    if (token)
        strncpy(Tcontent, token, sizeof(Tcontent) - 1);

    // Iterate through content list, check if name and content match, then remove
    for (int i = 0; i < numPeers; i++)
    {
        // Check for name and content
        if ((strcmp(peerList[i].peerContent, Tcontent) == 0) && (strcmp(peerList[i].peerName, Tname))==0)
        {
            // Raise the flag
            flag = 1;
            // Shift the elements to remove the peer
            for (int j = i; j < numPeers - 1; j++)
            {
                peerList[j] = peerList[j + 1];
            }
            // Decrement num peers
            numPeers--;
        }
    }
```

```
        // Flag set, peer is found so send back ack
        if (flag) {
          indexData.type = 'A';
          sprintf(indexData.data, "Content %s from %s deregistered successfully.\n", Tcontent, Tname);
        }

        // Flag not set, peer with content not found, send error
        else {
          indexData.type = 'E';
          strcpy(indexData.data, "Error.\n");
        }

        sendto(s, &indexData, sizeof(indexData), 0, (struct sockaddr *)&fsin, alen);
        break;
      }
      // PDU type is not recognized by index server, send out an error
      default:
      {
        fprintf(stderr, "Unrecognized PDU type.");
        indexData.type = 'E';
        strcpy(indexData.data, "Unrecognized PDU type.");
        sendto(s, &indexData, sizeof(indexData), 0, (struct sockaddr *)&fsin, alen);
        break;
      }
      }
      // Zero out the data sent and received to prevent previous data from being used
      memset(indexData.data, 0, 100);
      memset(peerData.data, 0, 100);
    }

  return 0;
}
```

// Peer Client-Server Code:

```
#include <sys/types.h>
#include <unistd.h>
```

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <sys/select.h>
#include <errno.h>
#include <fcntl.h>

#define BUFLEN 100
#define FILEBUF 1000 // The project document didnt specify the size of the file buffer, so i made it 200

struct pdu
{
    char type;
    char data[100];
};

struct contentPDU
{
    char type;
    char data[500];
};

int main(int argc, char **argv)
{
    char *host = "localhost";
    int port = 3000;
    struct hostent *phe;
    struct sockaddr_in sin;
    struct sockaddr_in contentServer;
    struct sockaddr_in client;
    int s, n, clientFile, myPort;
    int contentNum = 0;
    char name[10];
    char contentBuf[FILEBUF];
    char contentList[10][10];

    switch (argc)
    {
    case 1:
        fprintf(stderr, "Error: args not valid.\n");
        exit(1);
    case 2:
        host = argv[1];
        break;
    case 3:
```

```
    host = argv[1];
    port = atoi(argv[2]);
    break;
default:
    fprintf(stderr, "Error: args not valid.\n");
    exit(1);
}

memset(&sin, 0, sizeof(sin));
sin.sin_family = AF_INET;
sin.sin_port = htons(port);

/* Resolve host */
phe = gethostbyname(host);
if (phe == NULL)
{
    fprintf(stderr, "Unable to resolve host: %s\n", host);
    exit(1);
}
memcpy(&sin.sin_addr, phe->h_addr, phe->h_length);

// UDP Socket
s = socket(AF_INET, SOCK_DGRAM, 0);
if (s < 0)
{
    perror("Can't create socket");
    exit(1);
}

// Connect to the index server
if (connect(s, (struct sockaddr *)&sin, sizeof(sin)) < 0)
{
    perror("Can't connect to server");
    exit(1);
}

// Get username from client
printf("Enter your client name:\n");
n = read(0, name, 10);
name[strcspn(name, "\n")] = 0;

char ch;
printf("Choose an option from the menu: \n");
printf("1. Register content\n");
printf("2. Download content\n");
printf("3. Acquire list of available content\n");
printf("4. Deregister content\n");
printf("5. Quit\n");
```

```
fd_set rfds, afds;
int s2 = socket(AF_INET, SOCK_STREAM, 0);
FD_ZERO(&afds);
FD_SET(0, &afds); // stdin

while (1)
{

   struct pdu peerData;
   struct pdu indexData;
   struct contentPDU peerRequest;

   // Copy new socket to rfds
   memcpy(&rfds, &afds, sizeof(rfds));
   FD_SET(s2, &afds);

   // Select stdin or socket that needs service
   if (select(FD_SETSIZE, &rfds, NULL, NULL, NULL) < 0)
      perror("select error");

   // Handle input from STDIN
   if (FD_ISSET(0, &rfds))
   {

      ch = getchar();
      switch (ch)
      {

      // User wants to register content
      case '1':
      {
         char content[10];

         // Open a TCP socket to be used for download
         struct sockaddr_in reg_addr;
         s2 = socket(AF_INET, SOCK_STREAM, 0);
         reg_addr.sin_family = AF_INET;
         reg_addr.sin_port = htons(0);
         reg_addr.sin_addr.s_addr = htonl(INADDR_ANY);

         // Bind the TCP socket
         if (bind(s2, (struct sockaddr *)&reg_addr, sizeof(reg_addr)) < 0)
         {
            perror("Bind failed");
            break;
         }

         // Retrieve port number
         int alen = sizeof(struct sockaddr_in);
```

```
    getsockname(s2, (struct sockaddr *)&reg_addr, &alen);
    myPort = ntohs(reg_addr.sin_port);

    // Socket can now listen for incoming connections
    if (listen(s2, 5) < 0)
    {
        perror("Listen error");
        break;
    }

    // Send R pdu to index server
    peerData.type = 'R';

    // Get content name from client
    printf("Enter the name of the content to register:\n");
    n = read(0, content, 10);
    content[strcspn(content, "\n")] = 0;

    // Format string to send to index server
    snprintf(peerData.data, 100, "%s %s %d", name, content, myPort);

    // Send the PDU
    if (send(s, &peerData, sizeof(peerData), 0) < 0)
        perror("send error");

    if (recv(s, &indexData, sizeof(indexData), 0) < 0)
        perror("recv error");

    // print message from index server
    printf("%s\n", indexData.data);

    // Add registered content to peers list of content
    strncpy(contentList[contentNum], content, 10);

    // Increment number of contents peer has registered
    contentNum++;

    break;
}

// User wants to download content
case '2':
{
    char content[10];

    // Send S type pdu
    peerData.type = 'S';

    // Name of content to download
```

```
printf("Enter the name of the content to download:\n");
n = read(0, content, 10);
content[strcspn(content, "\n")] = 0;

// Format string to send to index server
snprintf(peerData.data, 100, "%s %s", name, content);

/// Send S type PDU with data
if (send(s, &peerData, sizeof(peerData), 0) < 0)
{
   perror("send error");
}

struct pdu indexData;

if (recv(s, &indexData, sizeof(indexData), 0) < 0)
   perror("recv error");

// We need to tokenize the string sent back by the index server to setup
// TCP connection with port # and IP address
char *token = strtok(indexData.data, " ");
char contentServAddr[20]; // Hold content server IP addr
int contentServPort;      // Hold content server port #

// First token is IP address of content server
if (token)
   strncpy(contentServAddr, token, sizeof(contentServAddr) - 1);

// Next token is port number of content server
token = strtok(NULL, " ");

contentServPort = atoi(token);

// Etract address and port for content server peer
printf("Address: %s Port: %d\n", contentServAddr, contentServPort);

// Server address structure
memset(&contentServer, 0, sizeof(struct sockaddr_in));
contentServer.sin_family = AF_INET;
contentServer.sin_port = htons(contentServPort);

// IP address is a string, convert to binary
inet_pton(AF_INET, contentServAddr, &contentServer.sin_addr);

int clientSock = socket(AF_INET, SOCK_STREAM, 0);
// Set up TCP conection
if (connect(clientSock, (struct sockaddr *)&contentServer, sizeof(contentServer)) == -1) // Fails
{
   // Debugging info
```

```
    fprintf(stderr, "Can't connect \n");
    printf("Error: %d\n", errno);
    printf("%s\n", strerror(errno));
    break;
}

// Succeeds
else
{
    printf("Connection established!\n");
    // Send D type PDU with filename to request file
    peerData.type = 'D';

    char sbuf[BUFLEN];
    int checkFail = 0;
    int bytesRead;

    snprintf(sbuf, 100, "%c %s", peerData.type, content);
    printf("%s\n", sbuf);

    // Write the PDU to the socket
    if (write(clientSock, sbuf, sizeof(sbuf)) < 0)
    {
        printf("Write failed: %d.", errno);
        printf("%s\n", strerror(errno));
        close(clientSock);
    }

    // open a file for storing downloaded content
    clientFile = open(content, O_WRONLY | O_CREAT | O_TRUNC, 0644);

    // Read the incoming bytes
    while ((bytesRead = read(clientSock, contentBuf, FILEBUF)) > 0)
    {
        // Invalid PDU type, should be C
        if (contentBuf[0] != 'C')
        {
            printf("Invalid PDU type: %c\n", contentBuf[0]);
            checkFail = 1;
            break;
        }
        // Write to the file
        write(clientFile, contentBuf + 1, bytesRead - 1);
    }

    // Close file when no more content to read
    close(clientFile);

    // Download fails
```

```
        if (checkFail) {
            printf("Download failed. Please try again.\n");
        }

        // Download succeeds
        else {

            // Inform user download has completed
            printf("You have successfully downloaded: %s\n", content);

            // Register as content server with index server
            peerData.type = 'R';

            // Format string to send to index server
            snprintf(peerData.data, 100, "%s %s %d", name, content, myPort);

            /* Send the PDU */
            if (send(s, &peerData, sizeof(peerData), 0) < 0)
                perror("send error");

            if (recv(s, &indexData, sizeof(indexData), 0) < 0)
                perror("recv error");

            printf("%s\n", indexData.data);

            // Increment number of contents peer has registered
            contentNum++;
        }

    }

    // Close
    close(clientSock);
    break;
}

// User asks for list of content
case '3':
{
    // Send O type PDU
    peerData.type = 'O';
    sprintf(peerData.data, "Requesting list\n");

    if (send(s, &peerData, sizeof(peerData), 0) < 0)
    {
        perror("send error");
    }

    printf("List of available content:\n");
```

```
    while (strcmp(indexData.data, "------------") != 0)
    {
        if (recv(s, &indexData, sizeof(indexData), 0) < 0)
            perror("recv error");
        printf("%s\n", indexData.data);
    }

    break;
}

// Deregister
case '4':
{
    peerData.type = 'T';
    char content[10];
    // Send T PDU to deregsiter
    printf("Enter the content name you wish to deregister:\n");
    n = read(0, content, 10);
    content[strcspn(content, "\n")] = 0;

    snprintf(peerData.data, 100, "%s %s", name, content);

    if (send(s, &peerData, sizeof(peerData), 0) < 0)
        perror("send error");

    if (recv(s, &indexData, sizeof(indexData), 0) < 0)
        perror("recv error");

    // Reorganize content list for peer
    if (indexData.type == 'A') {

        for (int i = 0; i < contentNum; i++) {
            // Search for deregistered content
            if (strcmp(content, contentList[i]) == 0) {
                for (int j = i; j < contentNum-1; j++)
                    strncpy(contentList[j], contentList[j+1], 10);
            }
        }
        // Decrement number of contents registered
        contentNum--;

        for (int i = 0; i < contentNum; i++) {
            printf("Content: %s\n", contentList[i]);
        }
    }
    printf("%s\n", indexData.data);

    break;
}
```

```
    // User quits and deregisters all content
    case '5':
    {
      // All content is deregisterd
      peerData.type = 'T';

      // Iterate through list of registered content
      for (int i = 0; i < contentNum; i++)
      {

        snprintf(peerData.data, 100, "%s %s", name, contentList[i]);

        /* Send the PDU */
        if (send(s, &peerData, sizeof(peerData), 0) < 0)
          perror("send error");

        if (recv(s, &indexData, sizeof(indexData), 0) < 0)
          perror("recv error");

        printf("%s \n", indexData.data);
      }
      printf("Successfully logged off.\n");

      exit(0);
    }
    }
  }

  if (FD_ISSET(s2, &rfds))
  {
    char buf1[FILEBUF];
    int fd;
    int i;
    int len = sizeof(client);

    // Accept the peer trying to connect
    int new_sd = accept(s2, (struct sockaddr *)&client, &len);

    // Receive the PDU from the client
    int bytes = read(new_sd, buf1, BUFLEN);

    if (bytes < 0)
    {
      perror("Recv error");
      printf("%s\n", strerror(errno));
    }

    // Check PDU type
```

```
peerRequest.type = buf1[0];

strncpy(peerRequest.data, buf1 + 2, strlen(buf1) - 2);
peerRequest.data[strlen(peerRequest.data)] = '\0';

printf("%s\n", peerRequest.data);

// Check to ensure PDU type is D
if (peerRequest.type == 'D')
{
   // Begin sending content to peer
   printf("Type: %c. Data: %s\n", peerRequest.type, peerRequest.data);

   // Open the file requested by the user
   fd = open(peerRequest.data, O_RDONLY);

   // File doesnt exist, inform peer with E PDU
   if (fd == -1)
   {
      printf("%d\n", errno);
      printf("%s\n", strerror(errno));
      buf1[0] = 'E';
      write(new_sd, buf1, 1);
      close(new_sd);
   }

   // File exists, mark the first byte as 'C' and send it
   else
   {
      while ((i = read(fd, buf1 + 1, FILEBUF - 1)) > 0)
      {
         // Mark as C type PDU and write to requesting peer
         buf1[0] = 'C';
         write(new_sd, buf1, i + 1);

      }
   }
   close(fd);
   // End of download
}

// Error
else
{
   send(new_sd, "Error: unrecognized PDU type", 29, 0);
}

// clear the buffer
memset(buf1, 0, 200);
```

```
            // Close the socket
            close(new_sd);
        }
        // Clear data
        memset(peerData.data, 0, 100);
        memset(indexData.data, 0, 100);
        memset(peerRequest.data, 0, 100);

    }

    // End of main function
    close(s);
    return 0;
}
```