BitTide: A simple BitTorrent Protocol

CS 344/444

Jong H. Lim and Andreas Terzis
BitTide

• A simplified version of BitTorrent protocol

• BitTorrent: a peer-to-peer file sharing protocol used for large scale file distribution over Internet

• The document for this project is available in our class website
Main goals

• Help you learn about basic network communication paradigms and specifically the client and server and peer-to-peer models.

• Help you learn to develop applications that handle concurrent network and file I/O operations and implement different state machines.

• Familiarize yourselves with the TCP/IP socket library and network programming
BitTide Main Components

• **Tracker**: a server knows the followings
  – 1) peer location (IP address and port number)
  – 2) peer’s file information (only metadata, not actual files)
    • Metadata: file name, size, which portions of a file a user has

• **Peer**: an entity which has files. If the file is complete, it is called ‘seed’
  – 1) query a tracker to get information of other peers/seeds
  – 2) a peer downloads file from other peers/seeds simultaneously
  – 3) it also updates the tracker with the most up-to-date file status
BitTorrent tracker identifies the swarm and helps the client software trade pieces of the file you want with other computers.

Computer with BitTorrent client software receives and sends multiple pieces of the file simultaneously.

©2005 HowStuffWorks
Implementation (1)

• Bitmap
  – represent a block of a file with one bit.
  – 1: have the block, 0: does not have the block
  – Example,
    • One block size = 1000 Bytes
    • A file of size 5000 Bytes.
    • |---------|---------|---------|---------|
    • 0 1000 2000 3000 4000 5000
    • 1 1 1 1 1 (31 in base 10)
    • In real implementation, a block size is 16384 bytes
    • The max file size is about 1.5Mbytes (1572864 bytes)
Implementation (2)

- Moving file pointer to a particular position
  - fseek (C), seek (Java)

- DNS name to IP address translation:
  - gethostbyname (C, Java)

- Byte-order: network-byte order (big-endian)
  - Only for numbers (not for string!)
  - htons(), ntohs() // for 2 bytes, in c and python
  - htonl(), ntohl() // for 4 bytes, in c and python
  - Java, by default, uses network-byte order.
Implementation (3)

• Tracker
  – Process logins/logout from peers
    • Remove any newline or whitespace characters at the end, if any.
  – Search files upon requests made by peers and send the results back
  – Periodically updates peers’ bitmap info.
  – Process any invalid or incorrect requests from peers
    • Send a error message back to the peer
Implementation (4)

• Peer
  – Request login and logout to a tracker
    • Remove newline or any whitespace characters at the end of string.
  – Request a file search to a tracker
    • The file name does not contain any whitespace characters.
  – Periodically sends bitmap updates to a tracker
    • Every time a peer receives a new blocks
  – Download files from multiple peers/seeds and may upload simultaneously
  – When downloading a file from multiple peers, try to load-balance among peers
  – Cancel on-going downloads/uploads
    • Repeat the downloading process (using find)
Testing

- We provide **one tracker** and **two peers** for testing
  - With one tracker and two peers, you will be able to download a file from multiple peers

- Verification: we will put a file called ‘testfile’ both in our web-server and peers.

- Try first to download from peers and compare the downloaded file with that of web-server

- Use sleep functions between send calls
Socket programming

**Goal:** learn how to build client/server application that communicate using sockets

**Socket API**
- introduced in BSD4.1 UNIX, 1981
- explicitly created, used, released by apps
- client/server paradigm
- two types of transport service via socket API:
  - unreliable datagram
  - reliable, byte stream-oriented

socket

a *host-local, application-created, OS-controlled* interface (a “door”) into which application process can **both send and receive** messages to/from another application process.
Socket-programming using TCP

**Socket**: a door between application process and end-end-transport protocol (UCP or TCP)

**TCP service**: reliable transfer of bytes from one process to another

---

Diagram:
- Two processes connected by TCP with buffers, variables.
- Controlled by application developer and operating system.
- Connected through the internet.
- Host or server.
Socket programming *with TCP*

Client must contact server
- server process must first be running
- server must have created socket (door) that welcomes client’s contact

Client contacts server by:
- creating client-local TCP socket
- specifying IP address, port number of server process
- When client creates socket: client TCP establishes connection to server TCP

- When contacted by client, server TCP creates new socket for server process to communicate with client
  - allows server to talk with multiple clients
  - source port numbers used to distinguish clients (*more in Chap 3*)

**application viewpoint**

*TCP provides reliable, in-order transfer of bytes ("pipe") between client and server*
Client/server socket interaction: TCP

Server (running on hostid)

- create socket, port=x, for incoming request:
  - welcomeSocket = ServerSocket()
- wait for incoming connection request
  - connectionSocket = welcomeSocket.accept()
- read request from connectionSocket
- write reply to connectionSocket
- close connectionSocket

TCP connection setup

Client

- create socket, connect to hostid, port=x
  - clientSocket = Socket()
- send request using clientSocket
- read reply from clientSocket
- close clientSocket
Socket programming with TCP

Example client-server app:
1) client reads line from standard input (\texttt{inFromUser} stream), sends to server via socket (\texttt{outToServer} stream)
2) server reads line from socket
3) server converts line to uppercase, sends back to client
4) client reads, prints modified line from socket (\texttt{inFromServer} stream)
import java.io.*;
import java.net.*;
class TCPClient {
    public static void main(String argv[]) throws Exception {
        String sentence;
        String modifiedSentence;
        BufferedReader inFromUser = new BufferedReader(new InputStreamReader(System.in));
        Socket clientSocket = new Socket("hostname", 6789);
        DataOutputStream outToServer = new DataOutputStream(clientSocket.getOutputStream());
Example: Java client (TCP), cont.

```java
BufferedReader inFromServer =
    new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));

sentence = inFromUser.readLine();

outToServer.writeBytes(sentence + '
');

modifiedSentence = inFromServer.readLine();

System.out.println("FROM SERVER: " + modifiedSentence);

clientSocket.close();
```
Example: Java server (TCP)

```java
import java.io.*;
import java.net.*;

class TCPServer {

    public static void main(String argv[]) throws Exception {
        String clientSentence;
        String capitalizedSentence;

        ServerSocket welcomeSocket = new ServerSocket(6789);

        while(true) {
            Socket connectionSocket = welcomeSocket.accept();
            BufferedReader inFromClient =
                new BufferedReader(new
                    InputStreamReader(connectionSocket.getInputStream()));

            String clientSentence;
            String capitalizedSentence;

            ServerSocket welcomeSocket = new ServerSocket(6789);

            while(true) {
                Socket connectionSocket = welcomeSocket.accept();
                BufferedReader inFromClient =
                    new BufferedReader(new
                        InputStreamReader(connectionSocket.getInputStream()));
            }
        }
    }
}
```
Example: Java server (TCP), cont

```java
DataOutputStream outToClient = new DataOutputStream(connectionSocket.getOutputStream());

clientSentence = inFromClient.readLine();

capitalizedSentence = clientSentence.toUpperCase() + '
';

outToClient.writeBytes(capitalizedSentence);
```
Handling a stream

- A **byte-stream** is a sequence of characters that flow into or out of a process.
- TCP handles user data as a stream
  - No inherent notion of “message” or “message boundary”

Problem: **Read** function only reads the amount of data available right now.
Handling a stream (cont’)

• Solution: repeatedly calls read functions until you finish reading the data you are supposed to receive.
  (pseudo-code)
  bytes_read = 0
  bufp = buf
  left_to_read = 1000
  while (left_to_read > 0)
    bytes_read = read(bufp, left_to_read)
    bufp += bytes_read
    left_to_read -= bytes_read

• Another problem: variable length message.
• Solution:
  • .................................................................
  • | Protocol(1) | Len(4) | Variable length payload |

You should do this for write or send function as well
Concurrence

The problem of a single thread application: blocking may happen

TCP connection established

Keyboard input

request

Client 2 may need to wait for client 1’s reply Indefinitely (block)

Server

Client 1 (Peer)

Client 2 (Peer)
Non-blocking I/O: Select

- Single-thread programming

- Event-driven: the select function or object (depending on PL) lets you know when events of your interest happen

- Pseudo code

1. Create sockets (or any I/Os) to monitor (let’s call them s1 and s2)

2. Register s1 and s2 to select function or object

3. While True
   
   select()  // waits for any events to come
   
   if s1, process s1
   
   .....  
   
   else if s2, process s2
Selector example in Java

Selector selector = null;
selector = Selector.open()

SocketChannel sChannel1 = createSocketChannel("127.0.0.1", 80);
SocketChannel sChannel2 = createSocketChannel("127.0.0.1", 80);

sChannel1.register(selector, sChannel1.validOps());
sChannel1.register(selector, sChannel1.validOps());
while(true){
    selector.select();
    Iterator it = selector.selectedkeys().iterator();
    while(it.hasnext()) {
        SelectionKey selkey = (SelectionKey)it.next();
        it.remove();
        try{
            processSelectionKey(selkey);
        } catch (IOException e){
            selkey.cancel();
        }
    }
}
Thread

• Multiple-thread

• A date server sends current date information to multiple clients.

• Concurrency: accept multiple connection requests from clients while serving date info to the clients who have already connected.

class DateServer extends Thread {

    private ServerSocket dateServer;

    public static void main(String argv[]) throws Exception {
        new DateServer();
    }

    public DateServer() throws Exception {
        dateServer = new ServerSocket(3000);
        System.out.println("Server listening on port 3000.");
        this.start();
    }

    public void run() {
        while(true) {
            try {
                System.out.println("Waiting for connections.");
                Socket client = dateServer.accept();
                System.out.println("Accepted a connection from: " +
                        client.getInetAddress());
                Connect c = new Connect(client);
            } catch(Exception e) {}
        }
    }
}
class Connect extends Thread {
    private Socket client = null;
    private ObjectInputStream ois = null;
    private ObjectOutputStream oos = null;

    public Connect() {}

    public Connect(Socket clientSocket) {
        client = clientSocket;
        try {
            ois = new ObjectInputStream(client.getInputStream());
            oos = new ObjectOutputStream(client.getOutputStream());
        } catch(Exception e1) {
            try {
                client.close();
            } catch(Exception e) {
                System.out.println(e.getMessage());
            }
            return;
        }
        this.start();
    }
}
public void run() {
    try {
        oos.writeObject(new Date());
        oos.flush();
        // close streams and connections
        ois.close();
        oos.close();
        client.close();
    } catch(Exception e) {} }
Question?