Department of Computer Science and Engineering,
Anna University, Chennai- 600025
CS23502- NETWORKS AND DATA COMMUNICATION
Practical
July- Nov 2026
Year/Sem/Batch : III/V/ N

Exercise: 6	UDP, TCP and DNS protocol	13- 08- 2026

Observation questions:

1. Differentiate between UDP and TCP in terms of connection setup, reliability, ordering of data, header overhead, and typical use cases. Name the corresponding Java classes used for each (`DatagramSocket`/`DatagramPacket` vs `Socket`/`ServerSocket`) and briefly state why Java exposes two different sets of classes instead of one.

2. Explain the TCP three-way handshake (connection establishment) and the four-way termination (connection close) process step by step, naming the flags involved (`SYN`, `SYN-ACK`, `ACK`, `FIN`) at each step. Also explain what the `TIME_WAIT` and `LISTEN` socket states mean and when a socket enters each.

3. In Java socket programming, methods such as `ServerSocket.accept()` and `InputStream.read()` are described as **blocking calls**. Explain what "blocking" means in this context, why it is necessary for these particular methods, and what problem could occur if a single-threaded TCP server used blocking calls while trying to serve multiple clients (relate this to why B3 needs multi-threading).

4. What is the maximum theoretical size of a single UDP datagram, and why is UDP subject to this limit while TCP is not? Explain what happens (at the Java `DatagramSocket` API level, and conceptually at the network level) if a sender tries to transmit more data than the receiver's buffer can hold, or more than the maximum datagram size allows.

5. Describe, step by step, what happens when a Java program calls `InetAddress.getByName("www.example.com")` — include the role of the OS-level DNS resolver, caching (and the significance of TTL), and the hierarchy of DNS servers (root → TLD → authoritative) that may be contacted on a cache miss. Also state which transport-layer protocol DNS queries typically use, and under what circumstance a DNS query would switch to the other transport protocol.

Execution Questions

1. UDP Client–Server Arithmetic Calculator

- Implement `UDPCalcServer.java` using `DatagramSocket` and `DatagramPacket` that listens on a fixed port, receives a request string of the form `"<num1> <operator> <num2>"` (e.g., `"12 * 5"`, supporting `+ - * /`), computes the result, and sends the result back to the requesting client's address/port.

- Implement `UDPCalcClient.java` that reads an expression typed by the user, sends it to the server, and prints the computed result received back.
- Requirement: The client should be able to send 5 different expressions in a loop without restarting the program. The server must handle a divide-by-zero request and an invalid operator request without crashing, returning an appropriate error message string to the client instead.

2. TCP File Transfer (Client Upload → Server Save)

- Implement `TCPFileServer.java` using `ServerSocket` and `Socket` that accepts a client connection, receives a filename followed by the file's byte content over the socket's `InputStream`, and saves it to disk on the server side with a `"received_"` prefix.
- Implement `TCPFileClient.java` that connects to the server, sends the name and byte content of a local text/image file (read using `FileInputStream`) over the socket's `OutputStream`, and prints a confirmation once the transfer completes.
- Requirement: Use a **byte-stream** (`InputStream`/`OutputStream`, not character streams) so the same code works for both text and binary files. Verify success by comparing file size (and, if possible, checksum) of the original and the received file. Ensure the server correctly detects **end-of-stream** (e.g., using `socket.shutdownOutput()` on the client side after sending) so `read()` doesn't block indefinitely.

3. Multi-threaded TCP Server for Concurrent Clients

- Implement `MultiClientTCPServer.java` that uses `ServerSocket.accept()` in a loop, and spawns a new `Thread` (or uses an `ExecutorService`) for each connected client so multiple clients can be served concurrently.
- Implement `MultiClientTCPClient.java` such that it can be run from multiple terminals simultaneously, each sending its own messages independently.
- Requirement: Demonstrate with **at least 3 simultaneous clients** connected to the same server, each getting an independent, correctly-ordered response. Print the client's socket port on the server side to distinguish clients in the log.

4. Java DNS Lookup Utility Client (mini "nslookup")

- Implement `DNSLookupClient.java` that takes a hostname as a command-line argument (or console input) and:
  - Prints the canonical/primary IP address using `InetAddress.getByName()`.
  - Prints **all** resolved IP addresses using `InetAddress.getAllByName()`.
  - Performs a **reverse DNS lookup** (IP → hostname) using `InetAddress.getByAddress()` / `getCanonicalHostName()` on one of the resolved IPs.
- No separate "server" is needed here — the DNS infrastructure acts as the server. Instead, pair this with a simple TCP server that logs each hostname your client resolves, i.e., after resolving, your client should connect to your own `DNSLogServer.java` (TCP) and send the resolved hostname + IP so the server can maintain a log file of all lookups performed.
- Requirement: Test with at least 3 different hostnames, including one invalid/non-existent hostname, and show how `UnknownHostException` is handled gracefully.

Spot Questions

1. Live modification to B1 (UDP Calculator) Using your working `UDPCalcServer.java` / `UDPCalcClient.java`:

1. Add support for a new operator `^` (power) and `%` (modulus), in addition to `+ - * /`.
2. Modify the server so that if the client sends more than two operands (e.g., `"5 3 2 +"`), it responds with an error message such as `"ERROR: only two operands supported"` instead of crashing.
3. Modify the server to print, on its own console, a running **count of total requests served so far** after every request (e.g., `"Request #7 served"`). Demonstrate all three changes working with a live client run, without restarting the server between requests.

2. Live modification to B2 (TCP File Transfer) Using your working `TCPFileServer.java` / `TCPFileClient.java`:

1. Modify the server to **reject** any incoming file larger than a fixed limit (e.g., 1 MB): it should read the declared file size first, and if it exceeds the limit, send back an error message (e.g., `"ERROR: file too large"`) and close the connection **without** writing a partial file to disk.
2. Modify the client to compute an MD5 (or simple byte-sum) checksum of the file **before** sending it, and send this checksum to the server along with the filename and content. Modify the server to recompute the checksum after receiving the file and print `"Transfer OK"` or `"Transfer CORRUPTED"` accordingly on its console.
3. Modify the server so that if a file with the same name already exists, it saves the new one with a timestamp appended to the filename instead of overwriting it. Demonstrate all three changes: one successful transfer with checksum match, one attempt with an oversized file being rejected, and one repeat transfer of the same filename showing it isn't overwritten.