情報通信実験（課題②　TCP 通信のためのクライアントプログラムの作成）

実験レポート

出席番号：　5S16　　名前：　　小形　孔明　　実験日：　2025 年 10　月　9　日

１．課題目的

TCP 通信におけるクライアントプログラムを Java で実装し、クライアント側通信処理の仕組みと動作を理解すること。

２．課題内容

本課題では、Java の Socket クラスを用いて TCP 通信のクライアントプログラムを作成する。課題①で作成したサーバプログラム（SimpleServer.java）と通信し、指定したホスト名およびポート番号に対して接続要求を行い、文字列の送受信が正しく行えることを確認する。

プログラムの基本的な処理の流れは、ユーザ入力（送信文字列）の取得,Socket によるサーバへの接続,PrintWriter を用いた送信処理,BufferedReader を用いた受信処理,送信文字列が"bye"の場合に接続を終了する処理,から構成される。

実験課題は、以下の５つからなる。

実験①　引数解析（host/port 取得）と parseInt／例外時 false を確認。

実験②　ホスト解決→Socket(address,port1)接続を実装。

実験③　入出力／標準入力を SJIS で初期化する処理を実装。

実験④　1 行送信→1 行受信・表示の通信ループと null 検出で終了を確認。

実験⑤　ストリームとソケットの順次クローズで正常終了を確認。

３．作成プログラムの重要箇所の解説

課題①

host　= args[0];

port1 = Integer.parseInt(args[1]);//第1引数を接続先ホスト名として保持し、第2引数を数値へ変換して接続ポートに設定する（不正値はNumberFormatExceptionで検知できる）。

課題②

InetAddress address = InetAddress.getByName(host);

socket = new Socket(address, port1);//hostをDNS解決して宛先IPを得てから、addressとport1を指定して能動オープンでTCP接続を確立する（失敗時はUnknownHostException／IOException）。

課題③

in　= new BufferedReader(new InputStreamReader(socket.getInputStream(), "SJIS"));

out = new PrintWriter(new OutputStreamWriter(socket.getOutputStream(), "SJIS"), true);//ソケットの入出力をSJISで文字化する。inは行単位受信を可能にし、outはautoFlush=trueによりprintlnのたびに即送信される。

課題④

out.println(msg1);

msg2 = in.readLine();//println で 1 行送信（改行付与）し、readLine で 1 行応答をブロッキング受信する。msg2==null は切断検出のシグナルとなる。

課題⑤

in.close();

socket.close();//受信ストリームを先に閉じて I/O 資源を解放し、続いて socket.close()で TCP 接続を終了する（下位ストリームもクローズされる）。

４．結果

課題①	引数 2 個で host と port1 を設定でき、数値変換失敗や個数不正時は false を返すことを確認。第一引数の文字種チェックは行っていない。
課題②	InetAddress.getByName(host)→newSocket(address,port1)で接続に成功（失敗時は false)。クライアント側の明示 bind は行っていない。
課題③	in/out/std_in を SJIS で初期化でき、PrintWriter は autoFlush=true で送信即時反映（生成失敗時は false)。
課題④	標準入力 1 行送信→サーバ 1 行受信・表示が反復動作し、msg1==null または msg2==null で終了に遷移。キーワード（例：bye）による終了は未実装。
課題⑤	ストリーム→ソケットの順にクローズし、終了ログを出して System.exit(0)で正常終了することを確認。

５．感想・考察

クライアント側は想定どおりに動き、SJIS 統一・行単位 I/O・切断検出など基本要件は素直に満たせた。一方で、サーバが単一接続前提の構造であるため、複数クライアント同時利用の検証では“接続は成立するが応答しない”という、ネットワーク実装の落とし穴に遭遇できたのは良い学びになった。

しかし、本来は複数クライアントから同一アドレス、同一ポートのサーバとは telnet できないはずなのに、接続までは可能な問題が起きていた。

核心は、TCP/OS の接続確立とアプリの処理開始が別物である点にあると考えられる。OS は listen の backlog で三者間ハンドシェイクまで通すため“接続できた”と表示されるが、サーバが 1 本目の接続を同一スレッドで read ループし続けている間は accept()に戻らず、2 本目以降はアプリに渡らない。結果、クライアント送信データは OS バッファに滞留し、サーバがその接続を受け取るまで応答は出ない。

これに関して、改善の本丸は並列化である。例えば、accept()のたびに接続専用スレッドへ委譲（最小修正)、規模を考えるなら固定スレッドプール、さらに高効率を狙うなら NIO/Selector で少スレッド多接続化する、という手法が考えられる。あわせて可観測性を上げるためには、accept/close のアドレス付きログ、setSoTimeout でハング検知、newServerSocket(port,backlog)の調整、終了語（例：bye）の明示という手法がありえるだろう。文字コードはクライアント/サーバで統一（環境に合わせ SJIS、純 Java なら UTF-8）する必要がある。

要するに、OS が担保する「接続の同時性」とアプリが実装すべき「処理の並行性」のギャップが原因であり、サーバ側の並列化と観測強化を行うことで「複数クライアントでも“接続も応答も同時”」が達成できると考えられる。