

UNIT 10

UDP

(5 Hours) — Network Programming (CAC355)

Please note

These notes may contain errors. They were generated automatically, so please verify important facts, API details and code against your textbook, the official Java documentation and your teacher's material before relying on them (especially for exams).

10.1 UDP Protocol

UDP (User Datagram Protocol) is a connectionless transport protocol. Data is sent as independent packets called **datagrams**. There is no handshake, no guarantee of delivery, no ordering and no retransmission; in exchange UDP has very low overhead and latency.

Aspect	TCP	UDP
Connection	Connection-oriented (3-way handshake)	Connectionless
Reliability	Guaranteed, retransmits lost data	Best effort, packets may be lost or duplicated
Ordering	In-order byte stream	No ordering guarantee
Data unit	Stream (no message boundaries)	Datagrams (boundaries preserved)
Overhead	Higher (20-byte header + state)	Lower (8-byte header)
Java classes	Socket, ServerSocket	DatagramSocket, DatagramPacket
Typical use	Web, email, file transfer	DNS, streaming, VoIP, games, SNMP, multicast

A UDP header contains only four 16-bit fields: **source port**, **destination port**, **length** and **checksum**. The theoretical maximum datagram size is 65,535 bytes, but after the IP and UDP headers the maximum data is **65,507 bytes**. Datagrams larger than the path MTU (about 1500 bytes on Ethernet) are fragmented by IP, so for reliability applications keep payloads small (a few hundred bytes up to ~1400).

Example 10.1 – When is UDP better than TCP?

Give two applications where losing an occasional packet is acceptable but delay is not.

Solution

Live voice/video calls and online games: a late packet is useless, so retransmitting (as TCP does) would only add delay. DNS queries are another case: a single small request/response is cheaper over UDP and the client simply retries if no reply arrives.

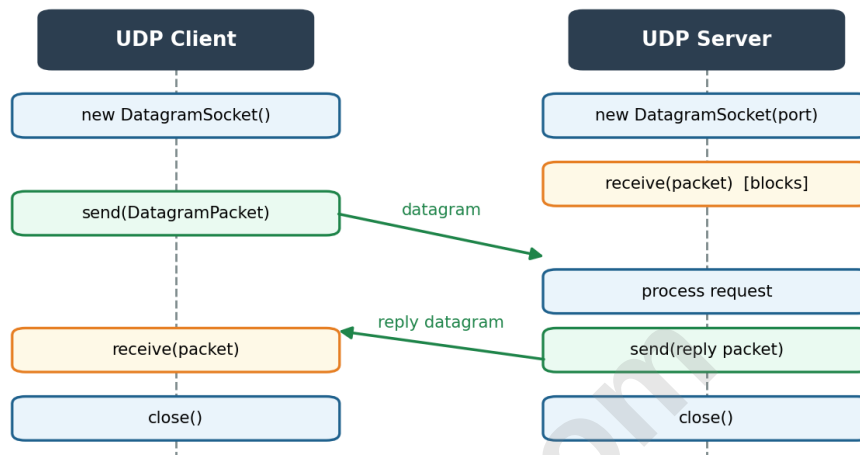


Figure 10.1: Typical UDP client–server exchange (no connection set-up).

10.2 UDP Clients

A UDP client follows these steps:

- 1. Create a **`DatagramSocket`** (the OS picks a free local port).
- 2. Optionally set a timeout with **`setSoTimeout()`**.
- 3. Build a **`DatagramPacket`** holding the data, destination address and port.
- 4. Call **`send(packet)`**.
- 5. Create an empty packet with a buffer and call **`receive(packet)`** to get the reply.
- 6. **`close()`** the socket.

```

import java.net.*;
import java.nio.charset.StandardCharsets;

public class UdpClient {
    public static void main(String[] args) throws Exception {
        try (DatagramSocket socket = new DatagramSocket()) {
            socket.setSoTimeout(3000); // wait max 3 s for reply
            byte[] out = "Hello UDP".getBytes(StandardCharsets.UTF_8);
            InetAddress host = InetAddress.getByName("localhost");
            socket.send(new DatagramPacket(out, out.length, host, 9100));

            byte[] buf = new byte[1024];
            DatagramPacket reply = new DatagramPacket(buf, buf.length);
            try {
                socket.receive(reply);
                System.out.println(new String(reply.getData(), 0, reply.getLength(),
                    StandardCharsets.UTF_8));
            } catch (SocketTimeoutException e) {
                System.out.println("No reply (packet lost?)");
            }
        }
    }
}

```

Listing 10.1: UDP client with timeout

10.3 UDP Servers

A UDP server creates a **DatagramSocket** bound to a well-known port, then loops forever: receive a datagram, process it, and send a reply to the sender's address and port (obtained from the received packet). There is no `accept()` and no per-client socket; a single socket handles all clients.

```

import java.net.*;

public class UdpServer {
    public static void main(String[] args) throws Exception {
        try (DatagramSocket socket = new DatagramSocket(9100)) {
            byte[] buf = new byte[1024];
            while (true) {
                DatagramPacket in = new DatagramPacket(buf, buf.length);
                socket.receive(in); // blocks
                String msg = new String(in.getData(), 0, in.getLength());
                System.out.println("From " + in.getAddress() + ":" + in.getPort() + " -> " + msg);

                byte[] out = msg.toUpperCase().getBytes();
                socket.send(new DatagramPacket(out, out.length,
                    in.getAddress(), in.getPort()));
            }
        }
    }
}

```

Listing 10.2: Iterative UDP server that returns the message in upper case

Note: Always recreate (or reset the length of) the receiving packet inside the loop. After `receive()` the packet's length shrinks to the size of the last datagram, so reusing it unchanged can silently truncate later, longer datagrams. If a datagram is longer than the buffer, the extra bytes are discarded without any error.

10.4 The DatagramPacket Class

A **DatagramPacket** wraps the bytes to be sent or received plus the remote address and port. It does not itself transmit anything; it is just a container passed to `DatagramSocket`.

10.4.1 The Constructors

Constructor	Use
<code>DatagramPacket(byte[] buf, int length)</code>	Receiving: empty packet with a buffer.
<code>DatagramPacket(byte[] buf, int offset, int length)</code>	Receiving into part of an array.
<code>DatagramPacket(byte[] buf, int length, InetAddress addr, int port)</code>	Sending to addr:port.
<code>DatagramPacket(byte[] buf, int offset, int length, InetAddress addr, int port)</code>	Sending part of an array.
<code>DatagramPacket(byte[] buf, int length, SocketAddress sa)</code>	Sending using a <code>SocketAddress</code> .

10.4.2 The get Methods

Method	Returns
<code>getAddress()</code>	<code>InetAddress</code> of the remote host (sender, or destination if outgoing).
<code>getPort()</code>	Remote port number.
<code>getSocketAddress()</code>	Remote address and port as a <code>SocketAddress</code> .
<code>getData()</code>	The whole underlying byte array (not only the valid part!).
<code>getLength()</code>	Number of valid bytes in the packet.
<code>getOffset()</code>	Index of the first valid byte in the array.

Note: `getData()` returns the full buffer. Always combine it with `getLength()`, e.g. `new String(p.getData(), p.getOffset(), p.getLength())`.

10.4.3 The set Methods

- **`setData(byte[])`** and **`setData(byte[], offset, length)`** – replace the payload.
- **`setLength(int)`** – set how many bytes of the buffer are used (must not exceed the buffer size).
- **`setAddress(InetAddress)`, **`setPort(int)`, **`setSocketAddress(SocketAddress)`** – change the destination, which allows one packet object to be reused for many recipients.****

Example 10.2 – Reusing a packet

A server receives a 500-byte datagram, then a 20-byte one into the same packet object. What does `getLength()` give for the second?

Solution

20 – the length is updated to the size of the most recent datagram. But a later 300-byte datagram would then be cut to 20 bytes unless you call `setLength(buf.length)` before the next `receive()`.

10.5 The DatagramSocket Class

DatagramSocket is the endpoint for sending and receiving datagrams. The same class is used by clients and servers; the difference is only whether you bind to a specific port.

10.5.1 The Constructors

Constructor	Description
<code>DatagramSocket()</code>	Binds to any available local port (typical for clients).
<code>DatagramSocket(int port)</code>	Binds to the given port on all local addresses (typical for servers).
<code>DatagramSocket(int port, InetAddress laddr)</code>	Binds to a specific local interface and port.
<code>DatagramSocket(SocketAddress addr)</code>	Binds to that address; pass null to create an unbound socket and <code>bind()</code> later.

Constructors throw **SocketException** if the socket cannot be created, e.g. the port is already in use (*Address already in use*) or, for ports below 1024, a privileged port is requested without permission.

10.5.2 Sending and Receiving Datagrams

- **send(DatagramPacket)** – sends the packet and returns immediately; no confirmation of delivery.
- **receive(DatagramPacket)** – blocks until a datagram arrives and fills the packet; throws `SocketTimeoutException` if `SO_TIMEOUT` expired.
- Both can throw **IOException**.

10.5.3 Managing Connections

UDP has no real connection, but a `DatagramSocket` can be **“connected”** to one remote address with `connect(InetAddress, port)` (or `connect(SocketAddress)`). After this the socket only sends to and receives from that peer; datagrams from other hosts are discarded, and the JVM skips some per-packet security checks, making it slightly faster. Nothing is sent over the network by `connect()`.

Method	Description
<code>connect(addr, port)</code>	Restrict socket to a single remote peer.
<code>disconnect()</code>	Remove the restriction.
<code>isConnected()</code>	True if connected.
<code>getInetAddress() / getPort()</code>	Remote address / port when connected, otherwise null / -1.
<code>getLocalPort() / getLocalAddress()</code>	Local port and address.
<code>bind(SocketAddress)</code>	Bind a previously unbound socket.
<code>close() / isClosed()</code>	Close the socket and free the port.

10.6 Socket Options

Socket options tune behaviour. They are set with the corresponding `setXxx()` methods of `DatagramSocket`.

Option	Methods	Meaning
SO_TIMEOUT	<code>setSoTimeout(ms) / getSoTimeout()</code>	Max time <code>receive()</code> blocks; 0 = forever. On expiry throws <code>SocketTimeoutException</code> (the socket remains usable).
SO_RCVBUF	<code>setReceiveBufferSize(n)</code>	Size of OS buffer for incoming datagrams; if full, new datagrams are dropped.
SO_SNDBUF	<code>setSendBufferSize(n)</code>	Size of OS buffer for outgoing datagrams.
SO_REUSEADDR	<code>setReuseAddress(boolean)</code>	Allow several sockets to bind the same port (needed for multicast listeners); must be set before <code>bind()</code> .
SO_BROADCAST	<code>setBroadcast(boolean)</code>	Permit sending to broadcast addresses such as 255.255.255.255 (default true in Java).
IP_TOS	<code>setTrafficClass(int)</code>	Type-of-Service / DiffServ bits as a hint to routers (e.g. low delay).

```
DatagramSocket s = new DatagramSocket(null);    // unbound
s.setReuseAddress(true);                      // before bind
s.bind(new InetSocketAddress(9100));
s.setSoTimeout(5000);
s.setReceiveBufferSize(128 * 1024);
s.setBroadcast(true);
s.setTrafficClass(0x10);                      // low-delay hint
```

10.7 UDP Applications

10.7.1 A Simple UDP Client (reading a line from the keyboard)

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

public class SimpleUdpClient {
    public static void main(String[] args) throws IOException {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        InetAddress host = InetAddress.getByName(args.length > 0 ? args[0] : "localhost");
        try (DatagramSocket socket = new DatagramSocket()) {
            String line;
            while ((line = in.readLine()) != null) {
                byte[] data = line.getBytes();
                socket.send(new DatagramPacket(data, data.length, host, 7777));
            }
        }
    }
}
```

Listing 10.3: Sends every typed line as one datagram

10.7.2 A UDP Server that Receives and Displays

```
import java.net.*;

public class DisplayUdpServer {
    public static void main(String[] args) throws Exception {
        DatagramSocket socket = new DatagramSocket(7777);
        byte[] buf = new byte[65507];          // largest possible payload
        while (true) {
            DatagramPacket p = new DatagramPacket(buf, buf.length);
            socket.receive(p);
            System.out.println(p.getAddress().getHostAddress() + ": "
                               + new String(p.getData(), 0, p.getLength()));
        }
    }
}
```

Listing 10.4: UDP server that prints what it receives

10.7.3 A UDP Echo Client

The **echo protocol** (port 7) sends back whatever it receives. The echo client below sends each line to an echo server and prints the reply, using a timeout so that a lost datagram does not hang the program. Because UDP is unreliable, the client must handle both a lost request and a lost reply.

```

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

public class UdpEchoClient {
    public static void main(String[] args) throws IOException {
        String host = args.length > 0 ? args[0] : "localhost";
        int port = args.length > 1 ? Integer.parseInt(args[1]) : 7;
        InetAddress addr = InetAddress.getByName(host);
        BufferedReader user = new BufferedReader(new InputStreamReader(System.in));

        try (DatagramSocket socket = new DatagramSocket(0)) {
            socket.setSoTimeout(2000);
            socket.connect(addr, port); // only talk to the echo server
            String line;
            while ((line = user.readLine()) != null) {
                byte[] data = line.getBytes();
                socket.send(new DatagramPacket(data, data.length));

                DatagramPacket reply = new DatagramPacket(new byte[1024], 1024);
                try {
                    socket.receive(reply);
                    System.out.println(new String(reply.getData(), 0, reply.getLength()));
                } catch (SocketTimeoutException e) {
                    System.out.println("Timed out - datagram lost?");
                }
            }
        }
    }
}

```

Listing 10.5: UDP echo client with connect() and timeout

Since a UDP server can be tested without any extra tools, you can also try **nc -u localhost 9100** (netcat in UDP mode).

10.8 DatagramChannel

DatagramChannel (java.nio.channels) is the NIO counterpart of DatagramSocket (see Unit 9). It works with ByteBuffers, can be switched to **nonblocking mode** and registered with a **Selector**, so one thread can service many UDP endpoints and TCP channels together.

Aspect	DatagramSocket	DatagramChannel
Data container	DatagramPacket (byte[])	ByteBuffer
Blocking mode	Always blocking (with timeout)	Blocking or nonblocking
Selector support	No	Yes (OP_READ, OP_WRITE)
Receive	receive(packet)	SocketAddress receive(ByteBuffer)
Send	send(packet)	int send(ByteBuffer, SocketAddress)
Connected mode	connect()	connect() then read() / write()

10.8.1 Using DatagramChannel

- **DatagramChannel.open()** creates the channel; then **channel.bind(new InetSocketAddress(port))** (or **channel.socket().bind()** in old code).
- **receive(ByteBuffer)** returns the sender's **SocketAddress**, or **null** if in nonblocking mode and no datagram is waiting. If the datagram is bigger than the remaining space, the excess is silently discarded.
- **send(ByteBuffer, SocketAddress)** returns the number of bytes sent (either the whole datagram or 0).
- **connect(SocketAddress)** restricts the peer and enables the simpler **read(ByteBuffer)** / **write(ByteBuffer)**.
- **configureBlocking(false)** for nonblocking operation.

```
import java.net.InetSocketAddress;
import java.net.SocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.DatagramChannel;

public class ChannelUdpServer {
    public static void main(String[] args) throws Exception {
        DatagramChannel ch = DatagramChannel.open();
        ch.bind(new InetSocketAddress(9100));
        ByteBuffer buf = ByteBuffer.allocate(1024);
        while (true) {
            buf.clear();
            SocketAddress client = ch.receive(buf);    // blocks (blocking mode)
            buf.flip();
            ch.send(buf, client);                      // echo back to sender
        }
    }
}
```

Listing 10.6: Echo server using DatagramChannel

Key Takeaways

- ✓ UDP is connectionless, unreliable and message-oriented: low overhead, no handshake, packets may be lost, reordered or duplicated.
- ✓ Maximum UDP payload is 65,507 bytes; small datagrams avoid IP fragmentation.
- ✓ Client: DatagramSocket() + send/receive. Server: DatagramSocket(port) + receive loop + reply to packet's address and port.
- ✓ DatagramPacket is only a container; use getLength() with getData(), and reset the length before reusing a receive packet.
- ✓ SO_TIMEOUT prevents receive() from blocking forever; handle SocketTimeoutException (retry or give up).
- ✓ connect() on a DatagramSocket only filters peers; nothing is sent on the network.
- ✓ DatagramChannel adds ByteBuffer, nonblocking mode and Selector support.

— End of Unit 10 —