COMP2221 Networks

David Head

University of Leeds

Lecture 14

Previous lectures

For all of our client-server examples so far we have used `Socket`, `ServerSocket`, or classes derived from them (*i.e.* `SSLSocket`).

- Sits in the Application layer.
- Use the Transport layer protocol TCP = Transmission Control Protocol.
- Creates a **persistent** contact between the two hosts.
- The most common **protocol** for sending and receiving messages over the internet.

Today's lecture

Today we will look at 'the other' protocol: UDP = User Datagram Protocol.

- A **connectionless** protocol, *i.e.* there is no **persistent** contact.
- Handled in Java using two new classes in `java.net`:
  - `DatagramPacket`
  - `DatagramSocket`
- See an example of a UDP-based client-server application.

We will look in more detail at the Transport-layer details of UDP (and TCP) in Lecture 16.

- In this lecture we focus on the protocol, *i.e.* what the Application layer 'needs to know.'

TCP

Transmission Control Protocol:

- **Reliable**: Lost/damaged packets are automatically re-sent.
- Re-orders data to maintain **sequencing**.
- **Throttles** the connection to avoid packet loss.
- Slower than UDP.

Analogy: **Phone call**

- Connect two people first.
- Maintain the connection throughout the conversation.
- Ordered communication.

UDP

User Datagram Protocol:

- **Unreliable**: No guarantee of delivery.
- Data is accepted **in the order that it arrives**.
- No congestion control.
- Faster than TCP.

Analogy: **email**

- Messages do not necessarily arrive in the order they were sent.
- No persistent contact between sender and receiver.
- No guarantee the mail will arrive at all.

When to use UDP

UDP is not suitable for `ftp`/`http`-type applications which require **complete** and **ordered** data.

- Reliability is a priority here.

UDP is suitable for applications where **speed** is a priority.

- *e.g.* streaming audio/video, where a small fraction of lost packets is acceptable.
- DNS uses UDP (see Lecture 4).

UDP is also suitable for:

- Testing for reliability, *i.e.* send a UDP packet and see if it is returned within a certain time.
- **Multi-casting** (see next Lecture).

Real-time networking

If not all of the services for TCP are necessary, can start with UDP and add required services **in the Application layer**.

A common application is for **real-time conversation**:

- Need **rapid response** — no more than 400ms.
- Can still follow the conversation with **some** packet loss.
- Need some strategy to **recover** from packet loss — including if it arrives 'too late.'

Common examples include:

- **VoIP** — Voice–over–IP.
- **RTP** — Real–time Transport Protocol.

RTP = Real–time Transport Protocol

Open standard for **real-time conversational applications**:

- Runs on-top of UDP (typically) — seen as any other UDP packet in the Network layer.
- 12-byte **RTP header** — sequence numbers, time-stamps, *etc.*
- Better chance of interacting if both end applications use RTP.

There are also **proprietary protocols** such as **Skype**:

- Uses UDP for audio/video data packets, TCP for control.
- Media packets also sent over TCP if a firewall blocks UDP *(see Lecture 17)*.

For more on media streaming, see Kurose and Ross 7^{th} ed., Chapter 9.

UDP in Java

The implementation is split into two parts, both in `java.net`:

The `DatagramPacket` class:

- Loads and unloads data into a **datagram**.
- The destination **address** and **port** is part of the datagram — *not* the socket!
- The source address and port are added automatically.

The `DatagramSocket` class:

- Sends or receives a datagram.
- Only knows the **local port** on which it listens or sends.

Issues for the client-server model

TCP treats a network connection as a **stream**.

- Permanent, two-way connection.
- We can assume sent data is received.
- We can assume it arrives in the order it was sent.

UDP has no concept of a unique, permanent connection between two hosts.

- Only deals with single messages/packets.
- Each host has to **listen** for data.
- The `DatagramSocket` does not know the destination.
- Does **not** require a one-to-one connection; can be *e.g.* one-to-many *[cf. Lecture 15]*.

The `DatagramPacket` class

Two types of constructor depending on context of use:

For **receiving** a datagram:

```
public DatagramPacket( byte[] buffer, int length )
```

- Maximum length specified by protocol; 8K is typical.
- Buffer capacity `buffer.length` must be **at least** as large as the `length` argument.

For **sending** a datagram:

```
public DatagramPacket( byte[] data, int length, InetAddress
destination, int port )
```

- Data is **loaded** from the given array.
- Destination address and port included.

Getters

`public byte[] getData()`

- Returns the data buffer as a byte array.
- **Not** an I/O stream — **we** convert to/from bytes.
- Up to us to break larger messages into manageable 'chunks.'

`public int getLength()`

- Length of data to be sent, or received.

`public InetAddress getAddress()`

- Returns the IP address of the destination (if sending) or the source (if receiving).

`public int getPort()`

- The destination or source port.

Setters

Unlike `Sockets`, we **can** change the class fields after construction.

- Allowed because there is no persistent connection.
- Can be useful if we want to re-use a datagram, to avoid garbage collection (*i.e.* for performance).

```
public void setData( byte[] buffer )

public void setLength( int length )

public void setAddress( InetAddress address )

public void setPort( int port )
```

The `DatagramSocket` class

Clients and servers use the same class (*i.e.* there is no '`DatagramServerSocket`'), but different constructors.

For a **client**:

```
public DatagramSocket() throws SocketException
```

- Opens a port (assigned at run time).

For a **server**:

```
public DatagramSocket( int port ) throws SocketException
```

- Opens on a **given** port (which is published for the client).

Both are **bound** to a local port and **listen** for incoming data.

Useful methods (1)

`public int getLocalPort()`

- Returns the port to which the socket is bound.
- Only really useful for the client-type constructor.

`public InetAddress getLocalAddress()`

- Gets the local address to which the socket is bound.
- Can be useful for hardware with multiple IP addresses.

`public void close()`

- Closes the `DatagramSocket`.

Useful methods (2)

`public void send( DatagramPacket d ) throws IOException`

- Sends the datagram you have created.

`public void receive( DatagramPacket d ) throws IOException`

- Receives a single datagram and stores it in `d`.
- **Blocking** - does not return until data is received.
- Closest thing to `accept()` in TCP's `ServerSocket`.
- Can use multi-threaded approach similar to before.

`public void setSoTimeout( int timeout ) throws SocketException`

- Sets the maximum time (in milliseconds) the socket will block for before throwing a `SocketTimeoutException`.

Convenience methods for managing connections

`public void connect( InetAddress host, int port )`

- After calling, can only communicate with the specified destination — will throw an `IllegalArgumentException` if an attempt is made to send to a different destination.
- Will not perform security checks (if security enabled).
- This is not a connection in the TCP sense, but does establish who they communicate with — a **one-sided** connect.

`public void disconnect()`

- Breaks the connection, defaults back to general communication (*i.e.* anyone can receive again).

Do not **need** to use these — the example given next doesn't.

Example[1]: Echo client/server

Code on Minerva: `UDPEchoClient.java`, `UDPEchoServer.java`

A UDP client/server pair.

The port number for the server is set when launched (command line argument).

The client sends a packet containing a single string.

- Destination address, port and string all command line arguments.

Servers responds with a packet containing the same string.

Client handles possible packet loss *and* rogue connections.

[1] `cs.baylor.edu/~donahoo/practical/JavaSockets/textcode.html`, from the book *TCP/IP Sockets in Java*, Calvert and Donahoo (Morgan-Kauffman, 2001).

UDPEchoServer fragment

```
// Bind to port
DatagramSocket socket = new DatagramSocket(servPort);

// Re-use the same data packet
DatagramPacket packet = new DatagramPacket(new byte[ECHOMAX
    ],ECHOMAX);

// Server loop
while( true ) {
  socket.receive(packet);      // Receive packet from client

  // Message to stdout on server
  System.out.println("Handling client at " +
    packet.getAddress().getHostAddress() +
    " on port " + packet.getPort());

  // Send message back to client (i.e. echo)
  socket.send(packet);
}
```

UDPEchoClient fragment (1)

```
DatagramSocket socket = new DatagramSocket();

// Maximum receive blocking time (milliseconds)
socket.setSoTimeout(TIMEOUT);

// The packet to send (address and port specified in args)
DatagramPacket sendPacket = new DatagramPacket(bytesToSend,
    bytesToSend.length, serverAddress, servPort);

// The packet to receive (should match the one sent
// for an echo server)
DatagramPacket receivePacket =
    new DatagramPacket(new byte[bytesToSend.length],
                       bytesToSend.length);
```

UDPEchoClient fragment (2)

```
do {
  socket.send(sendPacket);                 // Send the string
  try {
    socket.receive(receivePacket);

    // Check for rogue packets
    if (!receivePacket.getAddress().equals(serverAddress))
      throw new IOException("Received from unknown source");

    receivedResponse = true;
  } catch (InterruptedIOException e) {  // Timeout; Retry?
    System.out.println( "Timed out..." );
  }
} while( !receivedResponse );

if( receivedResponse )                     // Success or failure?
  System.out.println("Received: " +
     new String(receivePacket.getData()));
```

Testing on a real network

This example — or any other client-server application — will never suffer packet loss when tested on the same host, *i.e.* `localhost`.

- Also very unlikely on a Local Area Network (LAN).

However, when testing over a Wide Area Network (WAN), including the internet, both packet loss and packet corruption become a real possibility.

- We will see one way **how** this can happen in Lecture 17.

Don't be fooled into thinking UDP is simpler than TCP!

Once packet loss is factored in, your program logic will typically become *much* more complex.

Overview and next lecture

Today we have looked at UDP:

- Supports fast but unreliable transfer.
- Implemented in Java with `DatagramPacket` and `DatagramSocket`.
- Seen a simple **one-to-one** example, a UDP echo server.

The code is available on Minerva.

Next time we will see how UDP can be used for **one-to-many** communication.