COMP2221 Networks

David Head

University of Leeds

Lecture 15

Previous lecture

Last lecture we looked at the Transport layer protocol UDP = User Datagram Protocol.

- Does **not** guarantee the message is received, or packets arrive in the correct order.
- Does **not** maintain a persistent connection.
- Faster than the reliable service provided by TCP.
- Implemented in Java with `DatagramSocket` and `DatagramPacket`.
- Looked at an `EchoServer` example.

Also mentioned some uses, including DNS (Domain Name System) queries using port 53.

Today's lecture

In today's lecture we will look at another use for UDP: Sending the **same** data to **more than one** client.

- Communication models, including **one-to-many**.
- Define **broadcasting** and **multicasting**.
- See how Java implements multicasting using `MulticastSocket` (in `java.net`).

We will finish with a multicast client/server example.

Communication models

So far we have only considered **one-to-one** communication.

- Also known as **point-to-point** or **unicast**, and sometimes written **1-1**.

For TCP the `Socket` prescribes this model.

- The `Sockets` are **bound** to the two endpoints by address and port.

For UDP communication the `DatagramPacket`/`DatagramSocket` pairing prescribes this model.

- The `DatagramPacket` is directed to one address at one port.

In both cases the **route** taken (*i.e.* through which intermediate routers) is not specified.

One-to-many communication

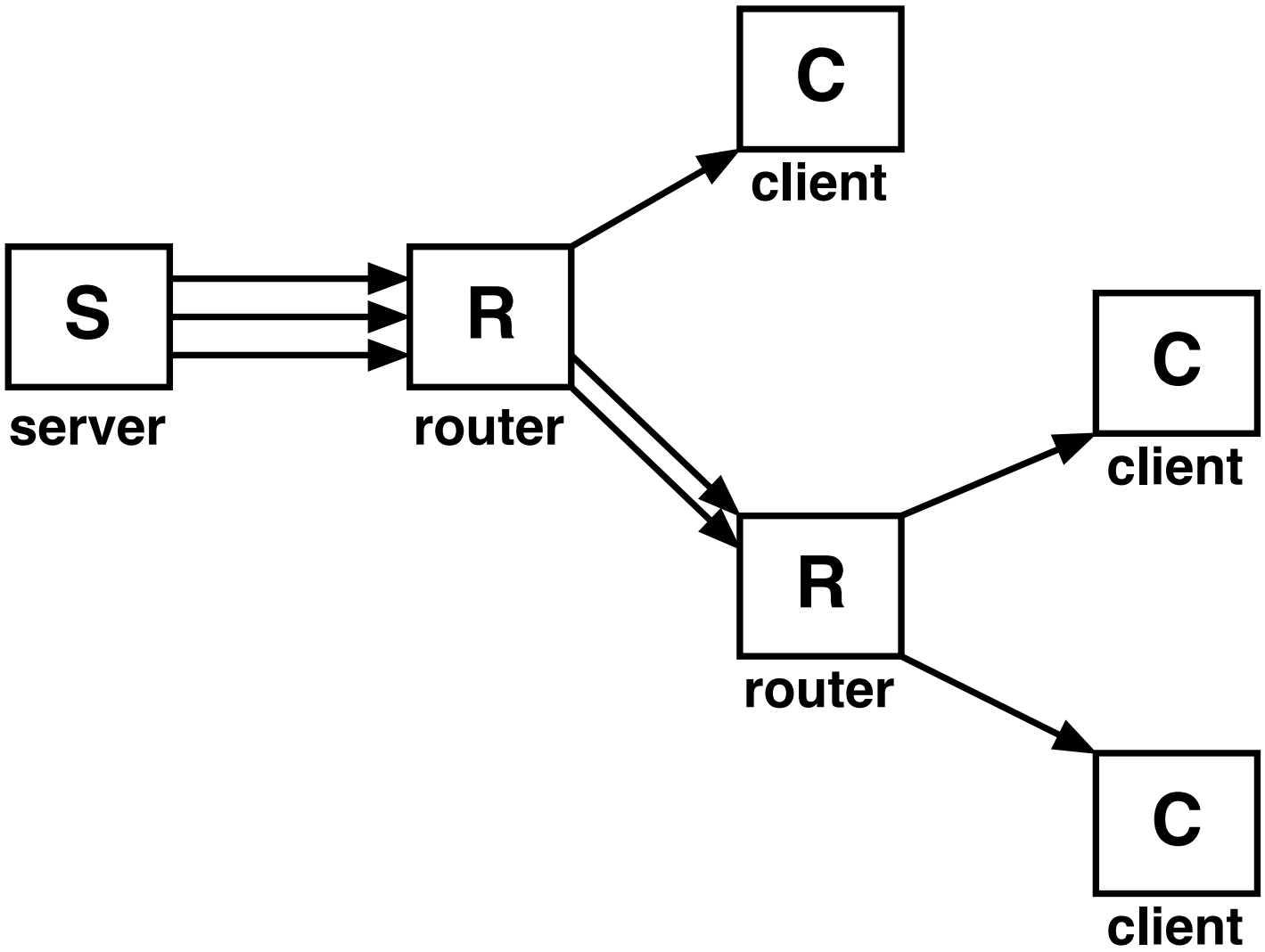
Our multi-threaded and non-blocking I/O server implementations were capable of handling multiple clients.

- We used TCP, but could also use UDP.

Imagine sending the **same** message to multiple clients (*e.g.* media streaming of a live event, video conferencing *etc.*)

- One socket per client for the **same data**.

At large scales (*i.e.* large numbers of clients) this becomes inefficient - it does not **scale**.


C
client
S
R
server
router
C
client
R
router
C
client

Types of one-to-many communication

Sometimes want to send the same message to all users.

Can send to a **(sub-)network**:

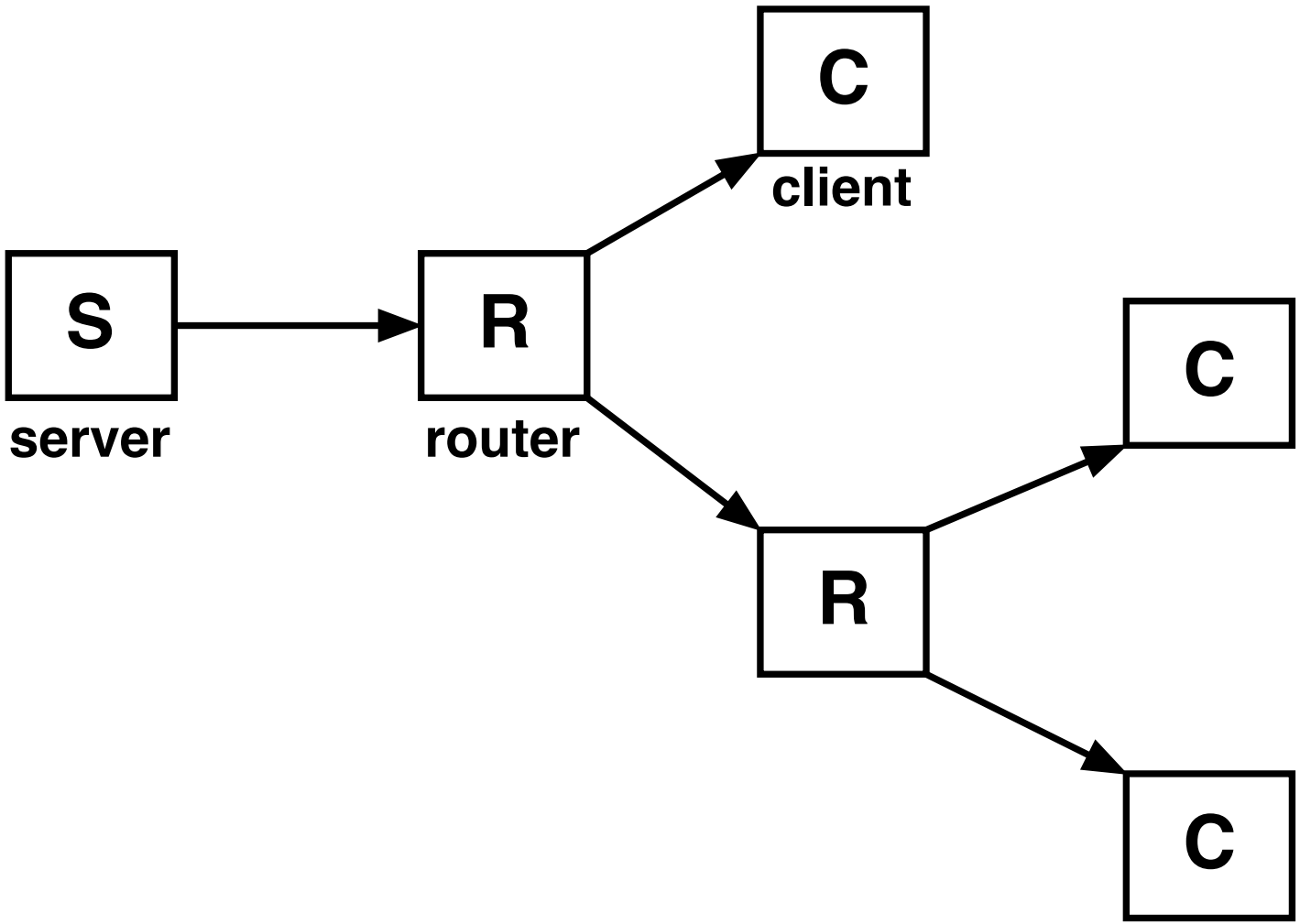
- For example, to everyone in the School, the University, the country, the whole internet ...
- Known as **broadcasting**.

Can also send to a **group of hosts**.

- *e.g.* a defined, registered group.
- Known as **multicasting**.

The server application sends one message but it is **routed** to multiple clients.

- Requires additional processing by **routers**.


C
client
S
server
R
router
C
R
C

Broadcasting

The Network layer supports broadcasting.

- For instance, if a device is added to a network it first needs an IP address.
- Initially communicates with local network using **broadcasting**.
- Part of DHCP = Dynamic Host Configuration Protocol.

The IPv4 address `255.255.255.255` is reserved for **local** broadcasting.

- Messages will not be forwarded by a router.
- Other hosts on the local network **choose** whether to listen.
- **No** IPv6 equivalent; instead **multi-casts** to 'all hosts.'

Multicast

For the client-server programming model:

The server delivers duplicate data to multiple clients . . .

- . . . but only sends **one** copy onto the network.
- Low risk of congestion in the link between server and the first (edge) router.

The clients each receive a single copy of the data.

- No **essential** change to versions of clients considered before (but need to use `MulticastSocket` - see later).

Multicast architectures

A simplified client-server protocol would be something like:

The **server** sends a stream of data.

- Continuous, *i.e.* 'always on.'

The **client** connects, receives data, and disconnects.

- Can connect and disconnect at any point in the stream.

Such an architecture could be considered for *e.g.* broadcast of a live event.

- Not for on-demand services, where each client requires different data streams.

TCP or UDP?

We still need to use one of the Network layer protocols.

TCP:

- Requires a maintained connection to each client.
- Requires separate acknowledgement from each client for each message and possible error handling.
- **Impractical** to implement.

UDP:

- Requires a mechanism for routing a single packet to multiple clients.
- The client must perform any further error handling.
- **Java implements multicast using UDP**.

Multicast addresses

A set of IP addresses are available for multicast communication.

- **IPv4**: Class D: `224.0.0.0` to `239.255.255.255`.
- **IPv6**: Prefix `ff00::/8`.
- A small number have been assigned[1], but have had limited penetration thus far.

Each address represents a **multicast group**.

- Servers send packets to this address.
- Clients listen to this address; they are **not** connected.

[1] The BBC trialled multicasting, but the page is now archived: `http://www.bbc.co.uk/multicast`

Routing

Multicast relies on the **router** to handle these addresses.

- The routing process is more complex.

Transparent to application programmers.

- It is a **service** supplied by lower levels in the protocol stack.
- Standard UDP-based network programming.

In fact, the server **could** be a `DatagramSocket` with packet addresses in the multicast address range, class D.

However, the client **must** be a multicast socket as it needs to join the group.

Scope

How far do we want our message to travel?

- There is no direct connection.
- We do no know who is listening.
- Without some sort of control packets could proliferate, **even without multicasting**.

The `DatagramPacket` has a field called TTL = Time To Live.

- Counts router **hops** before the packet is discarded.
- For instance, TTL=0 is the localhost, TTL=1 might be the School, TTL=48 is country-wide and TTL=225 is worldwide (approximately).

The MulticastSocket class

The primary class for multicasting in Java is `MulticastSocket`:

- Defined in `java.net`.
- `extends DatagramSocket`.
- Inherits the UDP communication model.
- Has additional capabilities for **joining multicast groups**.
- Is specified by (and requires) a multicast IP address and **any** standard UDP port number.

Important MulticastSocket methods for servers

`public MulticastSocket() throws SocketException`

- Attempts to create an instance of `MulticastSocket`.
- Note no IP address or port number required - this information is provided in the **data packets**.

`public void setTimeToLive( int ttl )`

- Sets the TTL (time to live) property for all sent packets.
- Defaults to `ttl=1`, *i.e.* the local network.

`public void send( DatagramPacket p )`

- Inherited from `DatagramSocket`.

Important MulticastSocket methods for clients

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

- Returns an instance bound to a specific port.

```
public void joinGroup( InetAddress address )
```

- Register with a multicast group[1].
- Can register with multiple groups.

```
public void leaveGroup( InetAddress address )
```

- Deregister from a group[1].

```
public void receive( DatagramPacket p )
```

- Inherited from DatagramSocket.

[1] May give deprecation warnings with the latest Java, but should work fine (and should not be a problem on school machines). See NumberClientNew.java.

Example: Multicast client/server

Code on Minerva: `NumberServer.java`, `NumberClient.java`

A UDP multicast client/server.

Server provides data at port 4446 at multicast address 228.3.4.5

- In the multicast address range for IPv4.

Server posts a stream of integers to the multicast address.

Client joins the group at the multicast address.

Receives a stream of 5 integers through the port, then closes.

Server fragment (1)

```
1  // Constructor. Specify host and port for packets, not
       sockets.
2  public NumberServer( String host, int port ) {
3    try {
4      mcGroup = InetAddress.getByName( host );
5        // host in a multicast IP address.
6    }
7    catch( UnknownHostException ex ) { ... }
8
9    mcPort = port;
10
11   try {
12     socket = new MulticastSocket(); // Port set at run time
13     socket.setTimeToLive( ttl );    // TTL for all packets
14   }
15   catch( IOException ex ) { ... }
16 }
```

Server fragment (2)

```
1  while( true ) {   // Server loop
2
3    // For this example send consecutive numbers every 2 secs
4    byte[] data = String.valueOf(counter).getBytes();
5
6    DatagramPacket dp = new DatagramPacket( data, data.length,
7                                            mcGroup, mcPort );
8    try {
9      socket.send( dp );
10     System.out.println( "Sent message" );
11   }
12   catch( IOException ex ) { ... }
13
14   try {
15       Thread.sleep( 2000 );          // Pause for 2 secs
16   }
17   catch( InterruptedException ex ) { ... }
18
19   counter++;
20 }
```

Client fragment (1)

```
1  public NumberClient( String host, int port ) {
2
3    try {
4      mcAddr = InetAddress.getByName( host );
5    }
6    catch( UnknownHostException ex ) { ... }
7
8    // Use NetworkInterface to avoid deprecation warnings
9    // (not errors) on recent versions of Java.
10   try
11   {
12     mcGroup = new InetSocketAddress( mcAddr, mcPort );
13     netInt = NetworkInterface.getByName( "bge0" );
14     socket = new MulticastSocket( mcPort );
15
16     socket.setSoTimeout( timeout );  // UDP => set timeout
17     socket.joinGroup( mcGroup );     // Join the group
18   }
19   catch( IOException ex ) { ... }
20 }
```

Client fragment (2)

```
1  public void runClient() {
2    int num = 0;    // Messages received
3
4    // For this example, only want to receive 5 packets
5    while( num<5 ) {
6      byte[] data = new byte[256];
7      DatagramPacket p = new DatagramPacket(data,data.length);
8
9      try {
10       socket.receive( p );  // Blocks until received
11       System.out.println( new String(p.getData()) );
12           // Echo
13     }
14     catch( IOException ex ) { ... }
15
16     num++;
17   }
18   socket.leaveGroup( mcGroup );    // Close gracefully
19   socket.close();
20 }
```

Overview and next lecture

Today we have looked at one-to-many communication, and in particular **multicasting**:

- More efficient way of sending **identical** data to multiple hosts **simultaneously**.
- Implemented in Java with `java.net.MulticastServer`.
- Not widely supported by ISPs (Internet Service Providers).

This is the **last** lecture on the Application layer, and therefore on Java network programming.

For the next 4 lectures, we will consider the lower levels in the protocol stack.