Question 1

(a) How many bytes are there in total in the header for a UDP message sent using IPv6? You do not need to include the link layer header, and you may assume there is no optional header data.

40+8=48

[1 mark]

(b) Similarly, how many bytes are there in the total header size for a TCP message sent using IPv4, again ignoring the link layer header?

20+20=40

[1 mark]

(c) What is the full, 16-byte hexadecimal form of the contracted address `2027:f11::3e0:2`?

2007:0f11:0000:0000:0000:0000:03e0:0002

[2 marks]

(d) What is the contracted form of `2001:0dd2:0000:0000:ba20:0000:0000:7c80`?

2001:dd2::ba20:0000:0000:7c80

[2 marks]

(e) What is the last address – that is, the one with the highest numerical value – of the CIDR range 128.232.136.0/21?

128. 232. 143. 255

[1 mark]

(f) How many addresses are there in this range? You should give your answer as either a number or a power of 2. You do not need to show working.

2^11

[1 mark]

(g) End systems have all 5 layers of the TCP/IP protocol stack, whereas routers are often considered to only have the bottom 3. Similarly, link switches only have the bottom 2, and repeaters only have the very bottom layer. Why is there no system with only the bottom 4 layers?

Becausethe transport layer is only meaningful in the context of an application.
If there is no application layer, then there isno use for the transport layer— since there' s nothing th
at initiates or consumes transport-layer communication.

(h) The remaining subquestions in Question 1 concern a multi–threaded server that allows clients to connect and vote for one of a list of options. The current number of votes for each option is stored in a hash-map,

```
private HashMap<String,Integer> currentVotes;
```

and each client handler has access to this same hash-map. If a client sends a vote to the server for the option `option`, the client handler calls the method `addVote` which is currently implemented as follows.

```
1     private addVote( String option )
2     {
3        if( currentVotes.containsKey(option) )
4        {
5          int votes = currentVotes(option);
6          currentVotes.replace( option, votes + 1 );
7          outStream.write( "Vote received." )
8        }
9        else
10       {
11         outStream.write( "Option not recognised." );
12       }
13       outStream.flush();
14    }
```

In this code, `outStream` is a buffered `OutputStreamWriter` linked to the client socket. There are also methods to return a list of available options and the current vote tally, but these are not shown here.

Firstly, which of the following operations should a minimal client application perform to send a vote to the server? Give your answer as one or more of the numbers listed below.

1 Client parses the option name from the user, and sends to server using some unspecified protocol.

2 Client contacts server to download the current votes for all options.

3 Client increments the votes for the option selected by the user, stored locally within the client application.

4 Client echoes the message sent by the server after receiving the vote request from the client.

[2 marks]

(i) What type of multi-threaded server architecture would you use for this application: thread–per–client, thread–per–server, or Executor server?

[1 mark]

(j) Explain your choice for question (i).

[1 mark]

(k) What possible benefit to the client do you see by the call to `flush()` on line 13?

[1 mark]

(l) For a general application, what negative consequences might there be of frequently calling `flush()`? What is the cause of this consequence?

[2 marks]

(m) As written, what would happen if two or more clients sent near-simultaneous votes for the same option? Give the line(s) in the code where a problem might occur.

[2 marks]

(n) How would you remedy this problem?

[1 mark]

[Question 1 Total: 20 marks]

ICMP: Internet Control Message Protocol

- Used by hosts and routers to communicate network level information.
- *e.g.* error reporting ('unreachable host'), echo request/reply.
- ICMP messages carried in IP datagrams.
- **ICMP Message:** type, code plus first 8 bytes of IP datagram causing error.

Type	Code	description
0	0	echo reply (ping)
3	0	dest. network unreachable
3	1	dest host unreachable
3	2	dest protocol unreachable
3	3	dest port unreachable
3	6	dest network unknown
3	7	dest host unknown
4	0	source quench (congestion control - not used)
8	0	echo request (ping)
9	0	route advertisement
10	0	router discovery
11	0	TTL expired
12	0	bad IP header

TCP

Connection-oriented protocol, *i.e.* maintains a persistent connection between the two hosts.

- Ensures **reliable data transfer**, possibly by requesting re-transmission of lost or corrupt segments.
- Also provides a degree of **congestion control**.

20-byte header with:

- Sequence and acknowledgement numbers.
- Bits used for maintaining the connection.
- Some specialist fields.

The remaining fields are the same as UDP.

The sequence and acknowledgement numbers are used to guarantee ordering, and to check for missed packets.

Network layer - IP

Forwards Transport layer data with a 20-byte (IPv4) / 40-byte (IPv6) header *[see also Lecture 17]*.

- Source and destination address (IPv4 or IPv6).
- Protocol (TCP or UDP).
- Checksum for the header integrity (IPv4).
- Time-to-live.

The **time-to-live** decrements whenever the datagram (packet) passes through a network node. Once it reaches zero, the message is discarded.

- This avoids datagrams that circulate forever.

`traceroute` and ICMP

Source sends series of UDP segments to destination.

- First has TTL=1, second has TTL=2, *etc.*

When *n*th datagram arrives at the *n*th router:

- Router discards it, returns ICMP message type 11 code 0.
- When received, sender calculates RTT = Round Trip Time.
- Three times for statistics; '∗' denotes timeout.

Stopping criterion:

- UDP segment eventually arrives at destination host.
- Destination returns 'port unreachable' (ICMP message type 3 code 3) because the port does not exist.
- When source gets this, it stops.

UDP

UDP is a **connectionless** protocol.

- Each message is independent.

Prepends an 8-byte header to the message including:

- Destination port number.
- Source port number.
- Message length.
- Checksum.

The ports identify the processes on both the source and destination hosts.

Has a checksum for data integrity, but simply discards data segments that have been corrupted.

IPv6

Initial motivation for IPv6:

- The 32-bit address space of IPv4 was 'running out.'

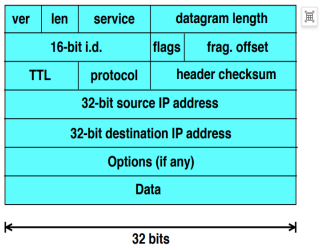
Additional motivation:

- Header format helps speed processing/forwarding at routers.
- Header includes **Quality of Service**.

IPv6 datagram format:

- **Fixed length** 40-byte header, with no optional field as in IPv4.
- No **fragmentation** allowed at routers, *i.e.* breaking one datagram into smaller datagrams. Instead, returns an error message (ICMP type 2).

IPv4 Header



len: Length of header, including options (20 bytes if none).
service: Type of service (TOS), *e.g.* real-time *versus* file transfer.
16-bit i.d., flags, frag. offset: Used for **fragmentation** *(breaking large messages into smaller ones at a router).*

(h) 1. 客户端需要让用户选择投票的选项,然后将这个选项名发送到服务器,对
2. 获取投票选项和得票数可能是一个附加功能，用于显示界面，但并不是投票操作所必需，错
3. 投票的统计在服务器端完成，不是由客户端修改计数客户端只负责发送选项，服务器修改 HashMap，错
4. echo（回显）服务器的响应内容不是必须操作，服务器写入“Vote received”是为了给客户端看，客户端读取这个信息就行，不需要“回显”它，错

(i) Executor server

(j) Ideally we would like to re-use existing threads if they are 'idle' (i.e. doing nothing / waiting for a client to connect), and only create a new thread if all current ones are in use.

Today we will look at two alternative models that only utilise a finite number of threads.

- This limited number of threads is known as a **thread pool**.
- **Controls** the growth of resources.
- The **thread-pool model** requires the number of threads to be estimated **in advance**.

Since thread pools are quite common in multi-threaded programming, Java now provides **Executors** to provide greater flexibility.

- Our third model is an **executor server**.

Thread-pool server

Basic approach is to have a **fixed number of threads** to handle clients.

- Known as a **thread-pool model**.
- Also called **threads-per-server**, as the number of threads is set by the server (and *not* the clients).

The threads **queue** to accept clients.

- The first `accept()` blocks all later threads (until another client makes a connection).

Behaves like a **set** of servers with the same listening port.

Implementation

In essence, every thread runs a version of our earlier one-client-per-server (serial) model from Lecture 8:

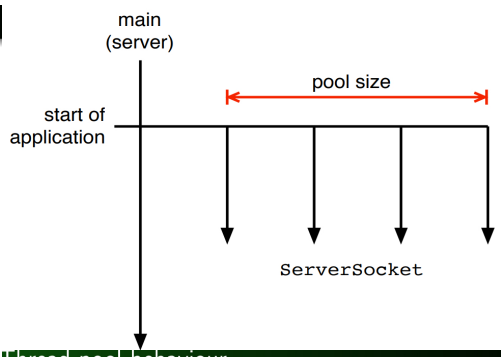
1. Create a number of threads at the start of the application.
   - How to choose the **right number**?
2. Each thread listens to the **same listening port**.
   - Possible since these are **threads**, *not* processes.
3. Each thread calls the blocking `accept()`.
4. When this method returns, each thread will handle **one client at a time**.

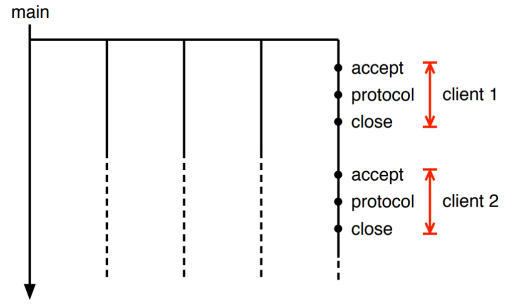
Thread-pool server *(schematic)*

```
public class ThreadPoolServer {
  ...
  public void runServer( int poolSize ) {
    ServerSocket serverSock = new ServerSocket(2323);

    for( int i=0; i<poolSize; i++ ) {
      ServerThread s = new ServerThread( serverSock );
      s.start();
    }
  }
  ...
}
```

- Still only one `ServerSocket` which is shared by all threads.
- Therefore still only one **listening port**.
- `poolSize` could be a static instance variable, set by command line arguments in `main()`, *etc.*



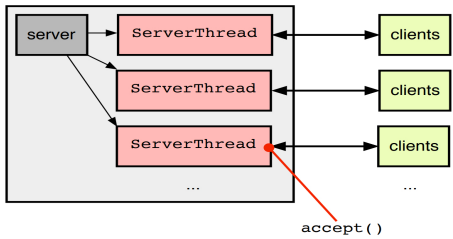


(each server thread handles multiple clients consecutively)

Server thread *(schematic)*

```
public class ServerThread extends Thread {
  private ServerSocket serverSock = null;
  ...
  // Constructor stores the ServerSocket
  public ServerThread( ServerSocket s ) {
    serverSock = s;
  }
  ...
  public void run() {
    while( true ) {
      Socket client = serverSock.accept();
      Protocol p = new Protocol( client );
      p.handleClient();
      // e.g. KnockKnockProtocol
    }
  }
  ...
}
```

Thread-pool behaviour



- The pool size is a **fixed parameter** of our model.
- Each thread is a **sequential** server running `accept()`.
  - Essentially the same that in Lecture 8.
- The threads are **queued** at the listening port.
- Which thread returns from `accept()` depends on the JVM and the thread scheduler.
  - May vary from run to run — **non-determinism**.
- When a client connection is ended, the corresponding thread returns to the start of its `while`-loop and calls `accept()`.
  - It has 'returned to the pool.'

Thread-safe `accept()`

If we were worried about this, we could use `synchronized(...)` from last lecture:

```
  public void run() {
    while( true ) {
      synchronized( serverSock ) {
        Socket client = serverSock.accept();
      }
      Protocol p = new Protocol( client );
      p.handleClient();
      // e.g. KnockKnockProtocol
    }
  }
```

This way, only one thread can call `accept()` at a time.

- Thread-safety of `accept()` no longer required.
- Which thread enters the synchronized block first is still unpredictable.

Thread safety

Note that multiple threads call `accept()` on the **same** `ServerSocket` object.

Does `accept()` perform properly in this situation?

- *i.e.*, will it only return the `Socket` object to one thread, or might it return the same client to two or more threads?

In other words, is `accept()` **thread safe**?

The official documentation does not make it clear if `accept()` is thread safe.

- Some implementations of JVM may be, some may not.

As a general rule, if a class of not explicitly declared thread safe, it is best to assume it is **not**.

Executor server

Ideally we would like to **re-use** existing threads if they are 'idle' *(i.e. doing nothing / waiting for a client to connect)*, and only create a new thread if all current ones are in use.

This would take some time to develop using `java.lang.Thread`

However, because it is useful in many contexts, Java already provides the high-level **Executor** service that does most of the work for us.

- Found in `java.util.concurrent`.

Thread-pool model pros and cons

Pros:

- Can handle multiple clients connecting to the same port.
- The required resource is finite and controllable.
- Threads are not created or destroyed in between client connections.

Cons:

- Must **tune** the pool size to the expected load.
  - Too small, and clients will have to wait.
  - Too large, and the overhead will increase and all threads will slow down.

Ideally we would **dynamically** alter the pool size to match the load.

`java.util.concurrent`

`java.lang.Thread` is actually quite a low-level[1] detail of concurrency.

Java 5 introduced a **higher-level** programming model:

- The `java.util.concurrent` package
- Decouples **scheduling** and **execution**.
- Allows us to manage the use of threads, and leave the details of creation/destruction to elsewhere.

Assume the JVM handles this sensibly, so the performance is 'good enough.'

newCachedThreadPool

The `newCachedThreadPool` creates a **flexible** pool that:

- **Creates new threads** if one of the current ones are in use.
- **Re-uses** existing threads if they have completed their task.

It therefore **automatically adjusts** to the load (*i.e.* the number clients).

There is also a `newFixedThreadPool(...)`, which sets the pool size.

- Similar to our thread-pool model, with the same 'con's.
- Only requires one line of code to change between `newCachedThreadPool` and `newFixedThreadPool`.

Using newCachedThreadPool

`newCachedThreadPool` is a **factory method** that returns an instance of `ExecutorService`.

Usage is straightforward:

```
ExecutorService service = Executors.newCachedThreadPool()
```

- `Executors` can return different objects, including cached an fixed-size thread pools.
- `ExecutorService` is generic (polymorphism).

```
service.submit( new KKClientHandler(client) )
```

- **Schedules** the execution of the task.
- `submit`'s argument implements the `Runnable` interface, and/or extends `Thread`.

Executor server pros and cons

Pros:

- Only creates new threads when existing resources are exhausted.
- Can switch to a fixed pool by changing one line.

Cons:

- The pool is still **unbounded**, so may exhaust machine resources (go out of memory, or run very slow).
- In particular, **long-lived clients** keep their resources.

Finer control, including a max. no. of threads, is possible using `ThreadPoolExecutor`, but we will not cover that here.

KKExecutorServer

Code on Minerva: `KKExecutorServer.java`

```
1  public class KKExecutorServer {
2    public static void main(String[] args) throws IOException
3    {
4      ServerSocket server = null;
5      ExecutorService service = null;
6
7      try {
8          server = new ServerSocket(2323);
9      } catch (IOException e) { ... }
10
11     service = Executors.newCachedThreadPool();
12
13     while( true )
14     {
15       Socket client = server.accept();
16       service.submit( new KKClientHandler(client) );
17     }
18   }
19 }
```

Choosing an architecture

What determines which architecture we should use?

Application requirements

- Protocol (*e.g.* storage required, compute-intensive *etc.*)
- Expected no. of connections at a time.

User behaviour

- Length of connection.
- Typical amount of communication per connection.

Should be careful to separate **design** (*i.e.* client-server architecture) from **communication** (*i.e.* client-server protocol).

- Can re-use **same** architecture for **different** protocols.

Overview and next time

Today we have looked at two types of **thread pool** for servers:

- The **thread-pool model**, where each thread effectively runs its own server, albeit with the same `ServerSocket`.
- The **Executor** service, a high-level system for running multi-threaded applications.

Combined with the **thread-per-client** model from last lecture, these are the 3 architectures we consider in this course.

Next time we will look at a different way to improve server performance - **non-blocking I/O**.

read()/write() correspondence?

There is **none**.

That is, do **not** assume that a single `write()` is matched by a single `read()`:

- One write may correspond to multiple reads.
- Conversely, one read may correspond to multiple writes.

We have no direct access to the buffer.

- Our only control is to call `flush()`, which sends all buffered data to the operating system (and then the network, file *etc.*).

Efficiency

Recall that TCP sends data as **packets** with header information.

- TCP header is usually 20 bytes.
- Additional headers at network and link layers.
- Total header at least 40 bytes, but can be much larger.

Sending **one byte** of data per packet is **very inefficient**.

Buffers are used to manage the communication of reasonably-sized packets.

- Only send when the buffer is 'sensibly' bigger than the header.
- Will always send to OS when `flush()` is called.

(k) flush()is essential toensure reliable and immediate communicationfrom server to client, making the client experience responsive and preventing deadlocks or hangs.

(l) Frequent calls toflush()can hurt performancebecause they prevent the output buffer from aggregating data into larger blocks. Instead, data is sent to the operating system (and eventually the network) too often — potentially in very small chunks.

(m) line(5) line(6)如果两个或多个客户端几乎同时对相同选项投票，由于currentVotes是共享的HashMap，而代码中没有同步机制，就可能发生竞态条件，导致：多个线程读取了相同的旧值，都加 1，再写入；最终只增加了一票，而不是两票。

(n) 使用synchronized同步整个addVote()方法

```
private synchronized void addVote(String option) {
    if (currentVotes.containsKey(option)) {
        int votes = currentVotes.get(option);
        currentVotes.replace(option, votes + 1);
        outStream.write("Vote received.");
    } else {
        outStream.write("option not recognised.");
    }
    outStream.flush();
}
```

Question 2

(a) Answer the following questions 'True' or 'False'.

(i) TCP is more suited to email and messaging apps than UDP.

(ii) If both client and sever are running on the same machine and using `localhost` to communicate, only the Application and Transport layers are involved.

(iii) Non-blocking I/O uses bidirectional buffers as this is fundamental to how non–blocking I/O works.

(iv) If a server is using non–blocking I/O, then clients connected to that server must also use non–blocking I/O.

(v) DNS uses UDP rather than TCP as the messages are small and speed is important.

[5 marks]

(b) The following steps are used to encrypt a long message sent by a sender A to a receiver B by using asymmetric encryption to first send a symmetric session key. However, they are in the wrong order. Place them in the correct order, from start (first) to finish (last).

1. B decrypts the message using the symmetric session key.
2. A encrypts a session key using B's public key, and sends it to B.
3. B decrypts the session key using her/his public and private keys.
4. B sends her/his public key to A.
5. A encrypts the message using the symmetric session key, and sends it to B.

[5 marks]

(c) While developing an application that uses UDP, it is noted that although the code has been shown to work on a local area network (LAN), it fails when tested over a wide area network – sent messages are sometimes not received. What do you think is happening to these messages, and why did this problem not occur on the LAN?

[2 marks]

(d) The rest of Question 2 is about an online 'bingo' game that someone is trying to develop. The rules of the game are simple. Each client has a 'card' of 10 numbers in the range 1 to 99. The server then selects numbers at random from this range, and sends the same number to all clients, without repeating numbers. The clients then check to see if the number is on their list, in which case it is removed. The winner is the first client to remove all of their numbers. It is decided to use Multicasting for this online game, in which clients who want to play first join the advertised multicast group.

Firstly, why is this multicasting and not broadcasting?

[1 mark]

(e) Give one example of a situation that is broadcasting, and not multicasting, over a network or subnetwork.

[1 mark]

(f) Returning to the online bingo game, the server initialises its `MulticastSocket mcSock` as follows:

```
1.      try {
2.          mcSock = new MulticastSocket();
3.          mcSock.setTimeToLive(0);
4.      } catch( IOException ex ) {
5.          // Error handling not shown.
6.      }
```

Will this work when the game tested on the same device, *i.e.* on `localhost`, for both the server and the clients? Explain your answer. If you refer to the above code in your answer, you should use the line numbers given.

[2 marks]

(g) The client application also has a `MulticastSocket mcSock`, but needs to be initialised differently. Some potential differences are listed below. Select those you believe are necessary differences and list their numbers. You will lose marks for suggestions that are *not* essential to the functioning of the client socket.

1 Must specify the port number in the constructor.

2 Must use a IPv4 multicast address from class D.

a Need to join the group specified by the multicast address.

b Must set a timeout of the order of a few seconds.

[2 marks]

(h) When the server sends each number to the clients, it first converts the number to the byte array `byte[] toSend` using a format known to the clients, and then sends it using the following code.

```
1. DatagramPacket pkt = new DatagramPacket(toSend,toSend.length);
2. try{
3.    mcSock.send( pkt );
4. } catch( IOException ex ) {
5.    // Error handling not shown.
6. }
```

Will this code work? Explain you answer. If you think it will not work, how would you alter it so that it does? If you refer to the above code in your answer, you should use the line numbers given.

[2 marks]

[Question 2 Total: 20 marks]

(a)(i) T

(ii) F 即使是 localhost，本机通信仍然需要经过网络层（IP 地址仍使用;无论本地还是远程，Transport 都需要依赖 Network 层来寻址

(iii)F Java 的 IO 中流（stream）是单向的，而非双向。非阻塞 IO 使用两个单向缓冲区实现双向通信

(iv) F 非阻塞 IO 是服务端的一种设计选择，对客户端没有要求，客户端依然可以是阻塞模式。课程第 12 讲有讲述 non-blocking IO 是一种性能优化而非协议要求

(v) T DNS 查询默认使用 UDP，因为消息小、请求频繁，UDP 效率更高。

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).

Streams

Streams allow us to **abstract** the I/O process away from the source/destination[1].

- *i.e.* file I/O is the same as network I/O is the same as terminal I/O . . .
- Abstraction is one of the design principles of OO languages, including Java.

Streams are **unidirectional**:

- **Input streams** read data, **output streams** write data.
- Two-way communication requires two streams.

DNS Protocol

Uses UDP, although can use TCP in special circumstances (*i.e.* if transferring a large amount of data from one server to another).

Queries **fail** if not answered within a set time limit.

- May retry until successful.

Uses port 53.

Messages are either 'query' or 'reply,' using the same format.

- If interested, Kurose and Ross §2.4 (7^{th} ed.) has more details.

(b)4; 接收者 B 首先提供公钥，供发送方 A 加密使用。
2: A 生成一个对称会话密钥（更快），并用 B 的公钥加密发送过去。
3: B 使用自己的私钥解密得到这个会话密钥。
5: A 用对称密钥加密实际消息并发送。
1: 接收者使用会话密钥解密消息（Step 1）

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.

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)

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.

(c) UDP 是一种不可靠的协议，不保证数据包送达。在**广域网（WAN）**中，由于网络拥塞、路径变化或丢包等现象的概率比在局域网（LAN）中高很多，因此数据包更容易丢失。而在 LAN 中，这些网络条件通常更稳定，丢包率非常低，因此开发者可能没有注意到 UDP 的不可靠性。
(d) Multicasting 是向加入了某个组的特定客户端发送消息；Broadcasting 是向子网内所有主机发送消息。

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**.

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`

(e) DHCP 地址分配是典型的 broadcasting。客户端用 255.255.255.255 广播找 DHCP 服务器
(f) 不一定成功。原因是 Multicast 默认 TTL（Time to Live）= 1，仅限于本地网络。localhost 测试中，如果 TTL=0，或者环回接口配置不当，组播包可能无法传回本地接收端口。
(g) 1: 客户端必须绑定端口：MulticastSocket(int port)
2: IPv4 的组播地址必须是 Class D：224.0.0.0 ~ 239.255.255.255
a: 客户端必须调用 joinGroup() 加入组
b: 设置 timeout 是推荐做法，但不是必要条件，不选

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).

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.

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)

```
// Constructor. Specify host and port for packets, not sockets.
public NumberServer( String host, int port ) {
  try {
    mcGroup = InetAddress.getByName( host );
      // host in a multicast IP address.
  }
  catch( UnknownHostException ex ) { ... }

  mcPort = port;

  try {
    socket = new MulticastSocket(); // Port set at run time
    socket.setTimeToLive( ttl );    // TTL for all packets
  }
  catch( IOException ex ) { ... }
}
```

Server fragment (2)

```
while( true ) {   // Server loop

  // For this example send consecutive numbers every 2 secs
  byte[] data = String.valueOf(counter).getBytes();

  DatagramPacket dp = new DatagramPacket( data, data.length,
                                          mcGroup, mcPort );
  try {
    socket.send( dp );
    System.out.println( "Sent message" );
  }
  catch( IOException ex ) { ... }

  try {
      Thread.sleep( 2000 );        // Pause for 2 secs
  }
  catch( InterruptedException ex ) { ... }

  counter++;
}
```

Client fragment (1)

```
public NumberClient( String host, int port ) {

  try {
    mcAddr = InetAddress.getByName( host );
  }
  catch( UnknownHostException ex ) { ... }

  // Use NetworkInterface to avoid deprecation warnings
  // (not errors) on recent versions of Java.
  try
  {
    mcGroup = new InetSocketAddress( mcAddr, mcPort );
    netInt = NetworkInterface.getByName( "bge0" );
    socket = new MulticastSocket( mcPort );

    socket.setSoTimeout( timeout );  // UDP => set timeout
    socket.joinGroup( mcGroup );     // Join the group
  }
  catch( IOException ex ) { ... }
}
```

Client fragment (2)

```
public void runClient() {
  int num = 0;   // Messages received

  // For this example, only want to receive 5 packets
  while( num<5 ) {
    byte[] data = new byte[256];
    DatagramPacket p = new DatagramPacket(data,data.length);

    try {
      socket.receive( p );  // Blocks until received
      System.out.println( new String(p.getData()) );
        // Echo
    }
    catch( IOException ex ) { ... }

    num++;
  }
  socket.leaveGroup( mcGroup );   // Close gracefully
  socket.close();
}
```

(h) 仅当 DatagramPacket 指定了目标地址和端口时才可用,改为:

```
DatagramPacket pkt = new DatagramPacket(toSend, toSend.length, groupAddr, port);
```

Question 3

(a) Why does IPv6 not support fragmentation of datagrams?

[1 mark]

(b) One of the possible values for the 'protocol' field in the IPv4 header denotes the payload is an IPv6 datagram. When is this used, and why is it necessary?

[2 marks]

(c) Suppose that the entire forwarding table for a router is as given below.

Destination address range	Link interface
129.71.128.0/20	0
129.71.140.0/22	1
129.71.142.0/23	2
129.71.160.0/19	3
129.71.176.0/20	3
Otherwise	4

Which interface would a packet with the destination address `129.71.143.207` be sent to? To help you, note that the binary representations of the third bytes in each of the above ranges are, from top to bottom: `1000 0000`; `1000 1100`, `1000 1110`; `1010 0000`; and `1011 0000`.

[1 mark]

(d) Explain your answer to the question (c).

[3 marks]

(e) Note there is a redundancy in the table – two of the address ranges can be merged into a single table entry without altering the forwarding. Which two lines are redundant?

[1 mark]

(f) What should these two lines be replaced with, to still give the same routing?

[2 marks]

(g) The previous questions have described a form of match–action forwarding based on the destination address. Name one other action, and the kinds of message it will match to, that may be employed on a real router.

[1 mark]

(h) Why are there so many Link layer protocols compared to the Network layer, which has comparatively few?

[1 mark]

(i) Consider the slotted ALOHA protocol for 2 nodes attempting to send many packets using the same channel, and suppose that the probability of a node sending a packet in any given slot is $p = 1/2$ or 50%. The probability of a packet being successfully sent by either node is $2 \times p \times (1 - p) = 2 \times 1/2 \times (1 - 1/2) = 1/2$, or 50%. What is the probability of a collision?

[1 mark]

(j) What is the probability of an empty slot, again for $p = 1/2$ and 2 nodes?

[1 mark]

(k) Now consider what happens when the probability for sending is reduced to $p = 1/3$, still for the case of 2 nodes. What is the chance for a collision now?

[1 mark]

(l) What is the probability of an empty slot?

[1 mark]

(m) Finally, what is the probability of successful transmission?

[1 mark]

(n) It is known that the theoretical optimal efficiency of slotted ALOHA occurs when $p = 1/N$ for N nodes. Are your results to the previous questions consistent with this? Justify your answer.

[1 mark]

(o) Now suppose the same two nodes switch to using the Time Division Multiple Access (TDMA) protocol over the same channel. How does the efficiency of this protocol compare to that of slotted ALOHA for $p = 1/2$? Explain your answer.

[2 marks]

[Question 3 Total: 20 marks]

[Grand Total: 60 marks]

End

(a)简化路由器设计，提高转发速度；避免路由器为大数据包分片带来的计算负担；改为由发送方根据路径 MTU（最小传输单元）来决定是否分片。如果 IPv6 报文太大而不能通过路径中的某段链路，路由器会返回一个 ICMPv6 的“Packet too big”错误信息Simplify router design and improve forwarding speed; To avoid the computational burden caused by routers sharding large data packets; Change to the sender deciding whether to shard based on the path MTU (Minimum Transmission Unit). If an IPv6 packet is too large to pass through a certain link in the path, the router will return an ICMPv6 "Packet too big" error message
(b)当使用tunnelling（隧道）技术时，如果某些网络设备只支持 IPv4，但通信双方是 IPv6 节点，就将 IPv6 报文封装进 IPv4 报文中；此时，IPv4 报文的protocol 字段值被设置为 IPv6（值为 41），表示其有效载荷是一个 IPv6 报文。When using tunneling technology, if some network devices only support IPv4, but the communicating parties are IPv6 nodes, IPv6 packets are encapsulated into IPv4 packets; At this point, the protocol field value of the IPv4 packet is set to IPv6 (value 41), indicating that its payload is an IPv6 packet.

Transition from IPv4 to IPv6

Not all routers can be upgraded simultaneously.

- *i.e.* there is no 'flag day' when the entire internet would switch to IPv6.

Therefore IPv4 and IPv6 must co-exist, for some time at least.

- How can the network operate with mixed IPv4 and IPv6 routers?

Tunnelling:

- IPv6 carried as **payload** in IPv4 datagram among IPv4 routers.

(c)子网前缀三字节前缀（十进制）第三字节（binary）接口
129.71.128.0/20129.71.128 10000000 0
129.71.140.0/22129.71.140 10001100 1
129.71.142.0/23129.71.142 10001110 2
129.71.160.0/19129.71.160 10100000 3
129.71.176.0/20129.71.176 10110000 3
目标地址的第三个字节是：143 = 1000 1111
1./20：匹配前 20 位（即前两个字节 + 第三个字节的前 4 位）
2./22：匹配前 22 位（第三字节前 6 位）
3./23：匹配前 23 位
4./19和/20后续范围均为 101xxxxx，已经超出 143（10001111）范围，不匹配
满足条件的有：
129.71.128.0/20 → 接口 0
129.71.140.0/22 → 接口 1（更长前缀）
选择最长前缀匹配 → 匹配 /22 → 所以选接口1
(d)解释重点：Longest prefix match rule；匹配 /22 优于 /20

Generalised forwarding

Early routers only used the destination IP address to determine the onward path for each packet.

Greater control possible by using **generalised forwarding**:

- **Match** incoming packets to **actions**.
- Can use **all header fields** from Transport, Network and Link layer headers.
  - Ports (source and destination), IP addresses, protocols, ...
- Actions can include dropping packets — **firewalling**.

Forwarding based on IP destination address only is now seen as a simple case of **match-action forwarding**.

(e)129.71.160.0/19（interface 3）
129.71.176.0/20（interface 3）
Redundant entry:129.71.176.0/20,because it is already fully included in 129.71.160.0/19, and both use
(f)应直接删除129.71.176.0/20，不需要额外替换，/19已经覆盖并指向同一接口
(g)Action:丢弃数据包（Drop packet）Matching condition:1.数据包符合防火墙规则（如特定端口、源 IP、协议类型等）2.或检测到攻击行为（如拒绝服务攻击流量）3.或基于服务质量策略（QoS）丢弃低优先级数据
(h)Link layer protocols are usually confined to a local subnet rather than the entire Internet. This allows different networks to use the protocols that best suit their media and requirements in their physical and link layers (such as Ethernet, Wi-Fi, Bluetooth, Zigbee, etc.). In contrast, network layer protocols (such as IP) need to be interoperable worldwide, so there are fewer protocols and more unified standards.

Today's lecture

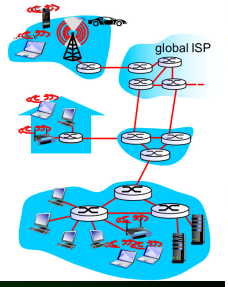
In today's lecture we will look at the bottom two layers

- The **Link layer**, also known as the data link layer.
- **Ethernet** protocols and **MAC addresses**.
- How multiple devices can share the same channel.
- The bottom-most **Physical layer** (very briefly).

This is the last lecture of new material for this module.

Link layer: Terminology

- **Node**: Host or router.
- **Link**: Communication channel that connects **adjacent** nodes.
- Links can be **wired** or **wireless**.
- Also have **LANs** = Local Area Networks.
- Data packets are called **frames**.



The **Link layer** is responsible for transferring data from one node to a **physically adjacent** node.

Link layer analogy

Packet/frame transferred by different link protocols over different links.

- *e.g.* ethernet on first link, frame relays on intermediate links, 802.11 (WiFi) on last link.

Each link protocol provides different services.

- *e.g.* May or may not provide reliable data transfer

Transportation analogy: Trip from Leeds to Lausanne.

- Taxi from Leeds to Airport (LBA).
- Plane from LBA to Geneva.
- Train from Geneva to Lausanne.

Traveller = packet/frame.
Transport segment = communication **link**.
Transportation mode = **link layer** protocol.
Travel agent = routing algorithm.

Link layer services (1)

Framing and link access:

- Encapsulate the datagram/packet into a **frame**, adding a header and possibly a trailer/footer.
- **Channel access** if the medium is **shared**.
- MAC addresses used in frame headers — **different from IP addresses**.

Reliable delivery between adjacent nodes (possibly)

- Similar strategy to TCP.
- Seldom used on **wired links** because of the low error rate.
- More important for **wireless links** with high error rates.

Link layer services (2)

Flow control:

- Pacing between adjacent sending and receiving nodes.

Error detection:

- Errors caused by signal attenuation and/or noise.
- Receivers detect presence of errors; signals sender to retransmit or drops frame.

Error correction:

- Receiver identifies **and corrects** bit error(s) using a **checksum**, without requiring retransmission.

Half-duplex and full-duplex:

- Half-duplex means nodes at both ends of a link can transmit, but not at the same time.

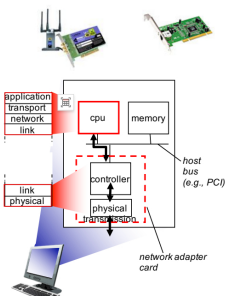
Where is the Link layer implemented?

Link layer implemented in each host.

- **Adaptor** (NIC = Network Interface Card) or **chip**.
- *e.g.* Ethernet card, 802.11 card, Ethernet chipset.

Attaches to the host's system buses.

Combination of hardware, software and firmware.



Multiple access links

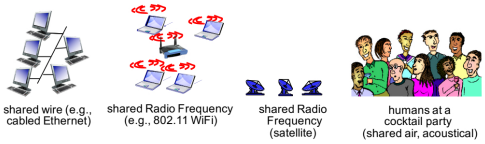
Two types of link:

Point-to-point:

- *e.g.* dial-up access, point-to-point link between Ethernet switch and a host.

Broadcast:

- *e.g.* Ethernet, 802.11 Wireless LAN ('Wi-Fi').



Multiple access protocols

Single, **shared** access channel.

- Two or more simultaneous transmissions may interfere.
- **Collision** if node receives two or more signals at once.

Need a **multiple access protocol**:

- Distributed algorithm that determines how nodes share the channel.
  - *i.e.* when a node can transmit.
- Coordination needed for better channel sharing and communication.

Types of Multiple Access Protocol

Channel partitioning:

- Divide channel into 'pieces' (time slots; frequency; code).
- Allocate pieces for node for **exclusive** use.
- *e.g.* TDMA = Time Division Multiple Access.

Random access:

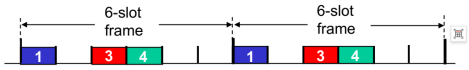
- Randomise send times to minimise chances of collision.
- If collision detected, **recover** (*i.e.* transmit).
- *e.g.* CSMA = Carrier Sends Multiple Access, and variations

'Taking turns':

- Nodes take turns, but nodes with more to send can take longer turns (coordinate using *e.g.* tokens).

TDMA = Time Division Multiple Access

- Form of **channel partitioning** in which each node gets a fixed length **time slot**.
- Example for a 6-node LAN.



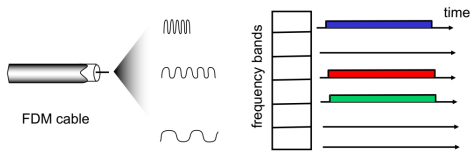
Problem: Unused slots go idle

Not an efficient use of available bandwidth.

In addition, will normally have to **wait** to **start** communicating.

FDMA = Frequency Division Multiple Access

- Alternative form of **channel partitioning**, this time dividing into **frequency bands**.
- Each node assigned **one band**.
- 6-node LAN example:



Problem: Each node still has limited bandwidth

Similar advantages and disadvantages as TDMA.

Random access protocol: Slotted ALOHA

Basic idea

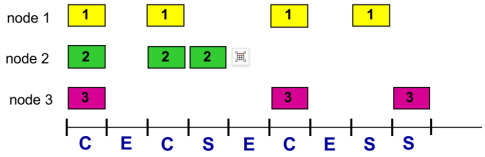
Start sending immediately. If hardware detects a **collision**, resend **after a random time interval**. Repeat if necessary.

For the next slide, have assumed:

- All frames equal size.
- Time divided into **slots**.
- Can only start transmission at the **start** of a slot.
- All nodes can detect **collisions**.
- When there is a collision, re-send in each subsequent slot with probability p, until successful.
- $0 < p < 1$.

Slotted ALOHA: Example

3 nodes start transmitting in the same slot. A possible outcome



[Key: **C**=Slot with a collision, **E**=Empty slot, **S**=Successful transfer]

- If only one node is communicating, it uses **full bandwidth**.
- But **collisions** result in **wasted slots**.
- Frames eventually sent because re-transmission is **random**.

Slotted ALOHA: Efficiency

Suppose N nodes are **all** trying to transmit **many** frames:

- Each transmits in a slot with probability p.
- Probability of success is $p \times (1-p)^{N-1}$, *i.e.* p that one node transmits, and $1-p$ for each remaining node to **not** transmit.
- Probability that **any** node has success is this times N, *i.e.*

$$Np(1-p)^{N-1} \quad .$$

- Can show maximum efficiency realised for $p^* = 1/N$.
- Can show this maximum efficiency is $1/e \approx 37\%$ for large N.

Other protocols can achieve higher efficiencies.

MAC Addresses

Whatever the protocol, each node must have a unique **MAC** (Media Access Control) address.

Consider the 32/128-bit IP address:

- Network layer address, used for forwarding.

The MAC (or LAN, physical, Ethernet) address:

- Used 'locally' to get frame from one interface to another.
- Both interfaces in the same network (in IP sense).
- 48-bit MAC addresses burned into NIC ROM.
- *e.g.* `1A-2F-BB-76-09-AD`.

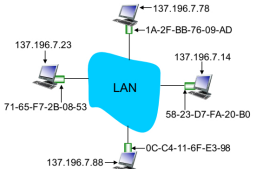
MAC Addresses

- MAC address allocation administered by IEEE.
- Manufacturer buys a range of MAC addresses — **unique**.
- Analogy:
  - MAC address like a **NI/social security number**.
  - IP address is more like a **postal address**.
- Flat (*not* hierarchical), for portability . . .
  - Move device from one LAN to another.
- . . . unlike IP's hierarchical addressing.
  - IP address depends on subnetwork to which the node is attached.

ARP: Address Resolution Protocol

Question:

How to determine an interface's MAC address knowing its IP address?



ARP tables:

- Each IP node (host, router) has an ARP table.
- Contains IP/MAC address mappings for nodes.
- Updated dynamically.

Ethernet frame structure

Header and trailer fields:

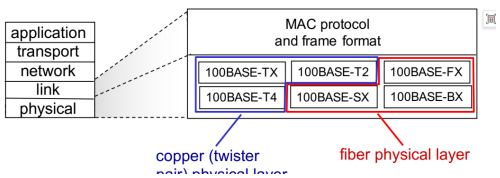
- **Preamble** has fixed bit pattern, for synchronisation.
- **Destination** and **source** MAC addresses.
- **Type** usually IP, but can be another higher-level protocol.
- **CRC** = Cyclic Redundancy Check for detecting errors.

Ethernet is **connectionless** and **unreliable**.

Ethernet standards

There are **many** different Ethernet standards.

- Common MAC protocol and frame format.
- Different speeds: 2 Mbps, 10 Mbps, 100 Mbps, 1 Gbps, 10 Gbps.
- Different Physical layer media: fibre, cable.



Physical layer

The final layer is the **Physical layer**, which is responsible for moving individual bits between nodes.

- Different media (*e.g.* twisted-pair copper, fibre optics) have different characteristics (frequency, signal decay).
- Therefore also have different Ethernet protocols.

Wireless communication has high error rates and multiple access.

- Also has many different protocols.
- *i.e.* IEEE 802.11 ('WiFi'), IEEE 802.15.1 ('Bluetooth'), IEEE 802.15.4 ('Zigbee'; 'Internet of things').

Overview and next lecture

Today we have looked at the bottom two levels:

- The **Link** layer, responsible for transferring data between adjacent nodes.
- Unique **MAC** address for each network interface.
- **Many different protocols**, as they only need to be adhered to locally.
- The **Physical** layer, responsible for moving individual bits.

The next lecture will be the final one, where we will summarise what we have learned.

(I)碰撞指的是两个节点同时发送数据p × p = 1/2 × 1/2 =1/4（25%）
(J)空时隙的概率（即两个节点都没发送）(1 - p) × (1 - p) = 1/2 × 1/2 =1/4（25%）(K)1/3 × 1/3 =1/9 ≈ 11.1%
(l)(2/3) × (2/3) =4/9 ≈ 44.4%
(m)A 发送、B 沉默：1/3 × 2/3
B 发送、A 沉默：2/3 × 1/3
(n)是的2 个节点时，最 发送概率是 p = 1/2
我们看到在 p = 1/2 时，成功率是50%
在 p = 1/3 时，成功率为44.4%→ p = 1/N 时，成功率（吞吐量）确实最优
(n)TDMA 的效率是 100%：每个节点轮流发送，不会发生碰撞Slotted ALOHA（p = 1/2）成功率是 50%
→ 所以TDMA 明显更高效