Networking lecture 2

- UDP vs TCP
- (TCP) networking challenges
- Assignment 2

Recap

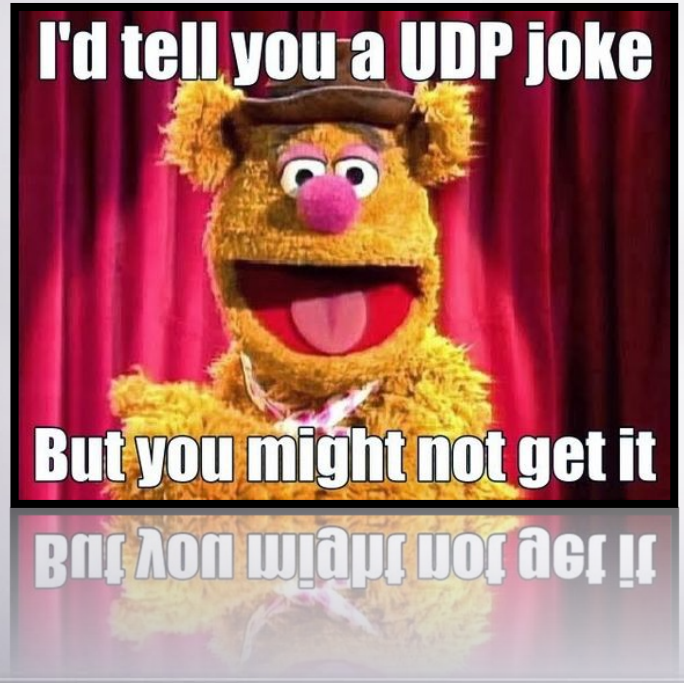
"Uhm... networking right?"

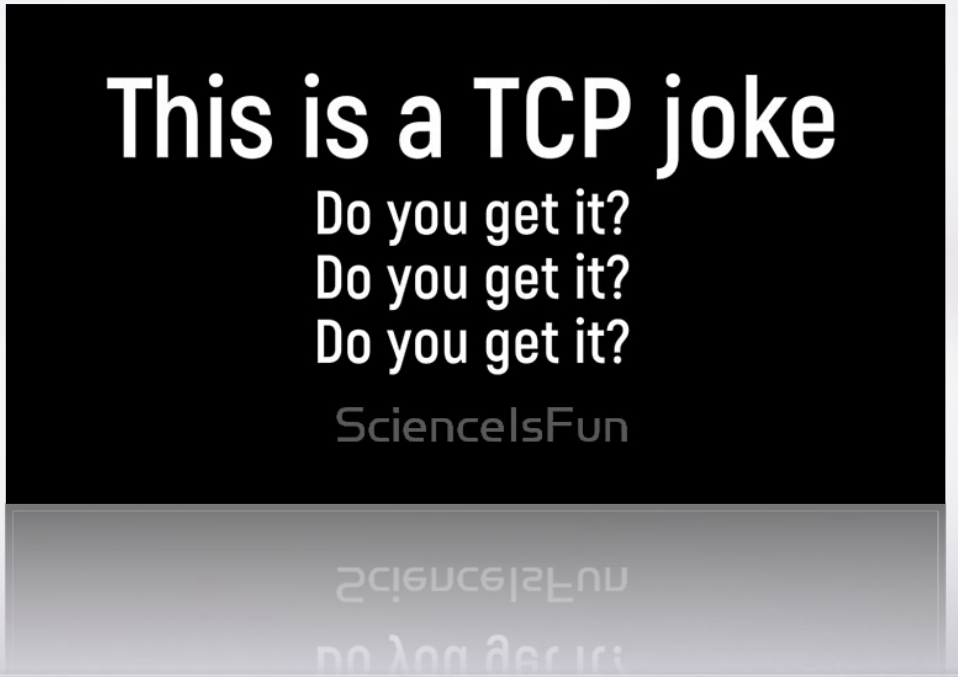
Recap 1 – Why are we here again?

"To build a (simple) network game on top of low level technologies"

Recap 2 – What have we discussed so far?

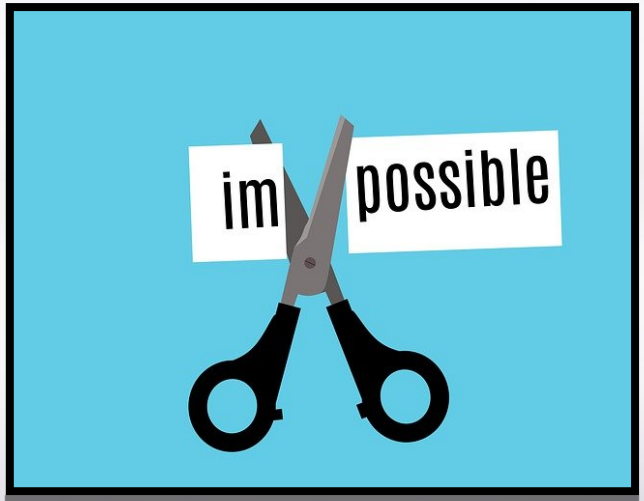
- Network basics
- Application to application network communication over the internet

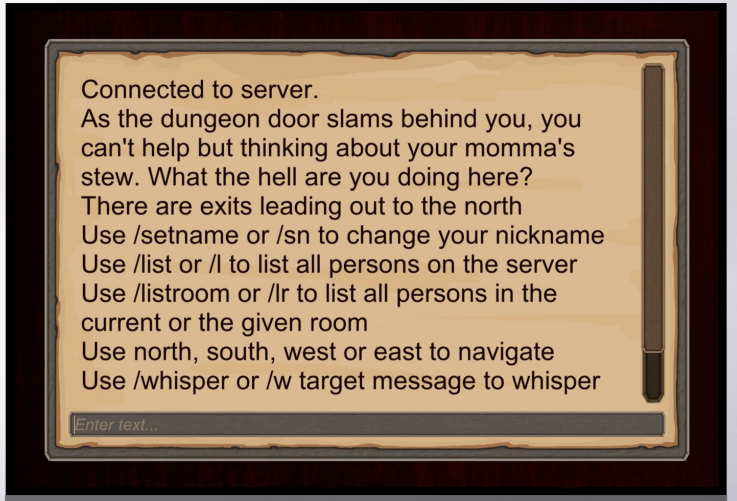




Today

- UDP & TCP characteristics in practice
- Some TCP/networking challenges
- Assignment 2 – Chatbox introduction





UDP vs TCP

UDP vs TCP demonstration

- The difference in *setup* between UDP & TCP

Difference in UDP/TCP setup?

- Checkout:
  - 001_basic_tcp_echo_server_client_commented (lecture 1) with
  - 001_basic_udp_echo_server_client_commented (lecture 2)

- Main thing to note:
  - Connection vs connectionless
  - 'Reading into a byte array' vs 'returning a byte array'
  - Ability to serve multiple clients, packets can come from anywhere
  - Duplicated packets? Missing packets? Unordered packets?

UDP vs TCP demonstration

- ~~The difference in *setup* between UDP & TCP~~
- The differences in *characteristics* between UDP & TCP
  - Packet loss
  - Packet duplication
  - Packet ordering

Difference in UDP/TCP characteristics?

- Hard to test with this setup:
  - Sending/receiving messages requires human intervention (type message, send, wait, repeat)
  - Perfect network conditions (everything runs on localhost ☺)
- Two changes:
  - a different test setup for auto send/receive
  - use a tool to simulate not so perfect network conditions

Test setup

- Again two examples:
  - 002_tcp_characteristics_demo
  - 002_udp_characteristics_demo

- Main idea: send increasing numbers and check order of arrival:
  - CS:1 → SR:1 → OK*
  - CS:2 → SR:2 → OK
  - CS:3 → SR:1 → Erm houston?
  - CS:4 → SR:4 → OMG, we've lost kenny!
  - CS:5 → SR:3 → Nevermind, we've found him!

(* CS = Client Send, SR = Server Receive)

Some details on the BitConverter class (MSDN)

- “The BitConverter class helps manipulate value types in their fundamental form, as a series of bytes. A byte is defined as an 8-bit unsigned integer. The BitConverter class includes static methods to convert each of the primitive types to and from an array of bytes “ →

- You can play around with this here

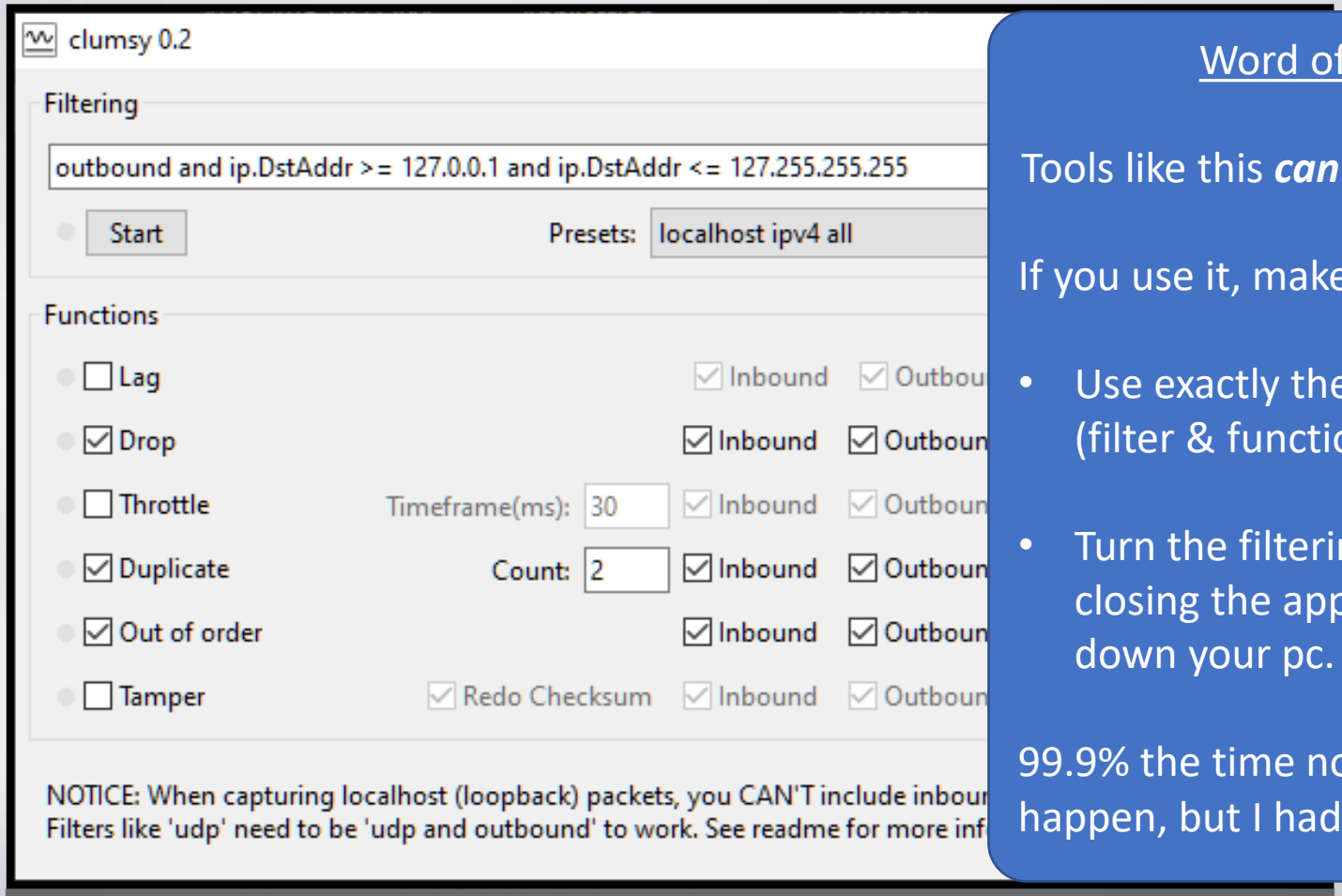
Type	To byte conversion	From byte conversion
Boolean	GetBytes(Boolean)	ToBoolean
Char	GetBytes(Char)	ToChar
Double	GetBytes(Double) -or- DoubleToInt64Bits(Double)	ToDouble -or- Int64BitsToDouble
Int16	GetBytes(Int16)	ToInt16
Int32	GetBytes(Int32)	ToInt32
Int64	GetBytes(Int64)	ToInt64
Single	GetBytes(Single)	ToSingle
UInt16	GetBytes(UInt16)	ToUInt16
UInt32	GetBytes(UInt32)	ToUInt32
UInt64	GetBytes(UInt64)	ToUInt64

Why do you need it?

- What is the size of the following objects in bytes?
  - Int32.MaxValue (= 2,147,483,647 = 0x7F FF FF FF) passed as a string?
  - Int32.MaxValue passed as an integer?
  - 255 passed as a string?
  - 255 passed as an integer?

- Conclusion:
  - integer byte size is fixed at 4 bytes → easier to handle
  - no string <-> integer conversion required → faster

Simulating bad network conditions with Clumsy



Word of warning:

Tools like this ***can*** break your system.

If you use it, make sure you:

- Use exactly the same settings (filter & functions)
- Turn the filtering **off** before closing the application or shutting down your pc.

99.9% the time nothing bad will happen, but I had to reinstall once ☺

Proves what we already knew…

UDP

- Connection-**less**
- Unreliable,
  packets **can** be:
  - lost,
  - duplicated,
  - reordered

TCP

- Connection **oriented**
- Reliable,
  packets **cannot** be:
  - lost,
  - duplicated,
  - reordered

TCP message boundaries

Playing around with the echo client/server

- Try this at home for both the UDP & TCP example:
  - Duplicate the line on the server that sends a reply
    - TCPEchoServerSample line 66:
      `stream.Write(inBytes, 0, inByteCount);`
    - UDPEchoServerSample line 34:
      `client.Send (inBytes, inBytes.Length, endPoint);`
  - Start sending messages from the client and see what happens

Conclusion

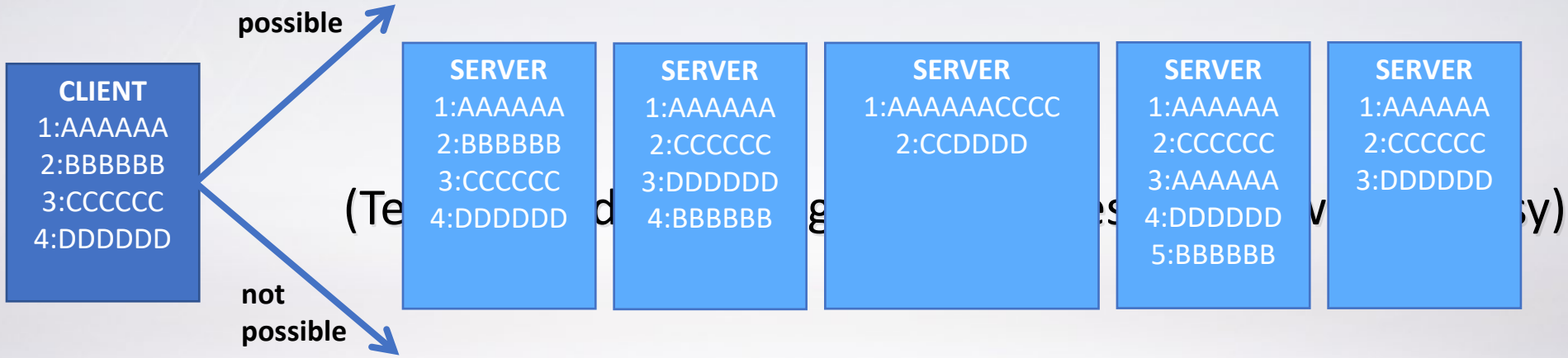
- TCP combines messages: A B C D E → A BC DE
- UDP does *not* combine messages: A B C D E → A C E D

- Another way to say this is that:
  - UDP *preserves message boundaries*
  - TCP does *not preserve message boundaries*

UDP preserves message boundaries

- UDP Packet sending and receiving is **all or nothing**
- *If* we send a packet of x bytes:
  - You might get the packet
  - You might not get the packet
  - You might get it twice
  - You might get packets out of order
  - But **if** it arrives…. it will be **exactly** x bytes…

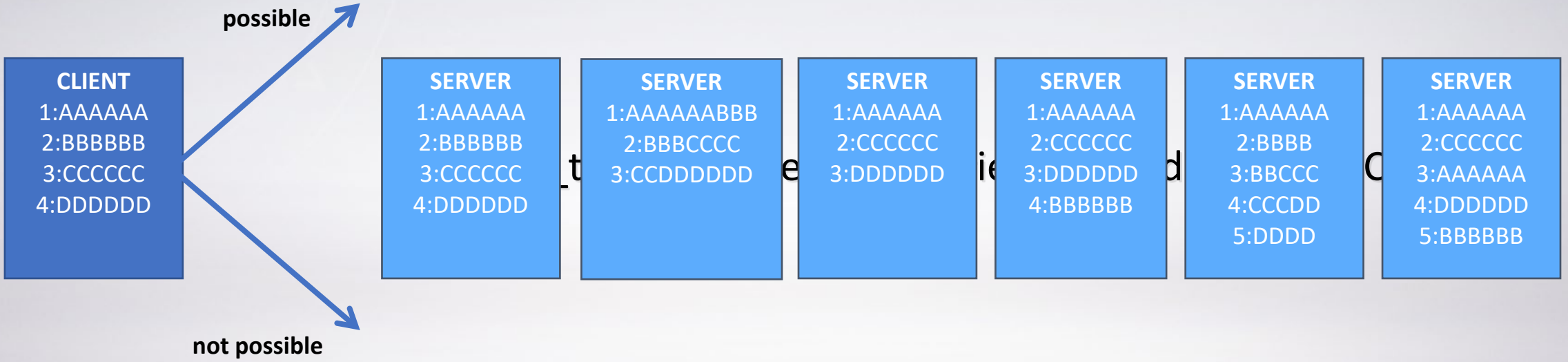
UDP preserves message boundaries



TCP does *not* preserve message boundaries

- As long as there is a connection TCP packets (bytes) ...
  - will arrive
  - in order
  - non duplicated
- But one send does not *necessarily* equal one receive:
  - One send *might* be one receive,
  - One send might be *multiples* receives,
  - Multiple sends might be *one* receive

TCP does not preserve message boundaries



Why should we care?
Why are boundaries important?

Imagine a simple text protocol to move an object:

move,objectId,x,y **e.g.** *move,playerOne,100,200*

Pseudo parse code:

```
string[] command = message.split (",");
if (command[0] == "move") {
      FindObject (command[1]).SetXY (
         int.parse (command[2]), int.parse (command[3])
      );
}
```

Life without boundaries

- Send 1 *move,playerOne,100,200*
- Send 2 *move,playerTwo,150,300*

- Receive 1 *move,playerOne,10*
- Receive 2 *0,200move,playerTwo,150,*
- Receive 3 *300*

- Good luck parsing this crap!
- Imagine if these were byte arrays representing objects (aka lecture 3)

Why does TCP work like this?

- TCP packets are ordered, non duplicate & reliable
- The only way to implement that is by keeping track of what you have:
  - sent – so that you can resend data, if necessary
  - received – so that you can order & filter new data
- Done through underlying send & receive buffers
  - One side writes data into the buffer
  - Other side reads data from the buffer
  - OS's on both sides send/receive when they 'feel' like it
  - These two buffers might or might not be in sync
- So again: one send does not *necessarily* equal one receive

Inner workings of Stream.Write ?

- void Stream.Write (buffer, offset, size):
  - See if there is room in the send buffer, if so write, if not block (!)

Inner workings of Stream.Read ?

- int Stream.Read (buffer, offset, size):
  - Checks if there is data in the Read buffer
  - might return:
    - size (exactly the amount of bytes you asked for, if present)
    - ***less than*** size (even down to ***only returning 1 byte*** at a time!)
    - 0 (0 means: 'reached end of connection' and **not** 'no data available')
    - an error
  - might block *until* there is data available
- In other words:
  - Read blocks until some **but not necessarily all** data is available
  - You cannot just call Read (…, …, x) and expect it to return x bytes.

Conclusion

UDP

- Connection-**less**
- Unreliable,
  packets **can** be:
  - lost,
  - duplicated,
  - reordered
- Preserves message boundaries
- (Max 64k message size)

TCP

- Connection **oriented**
- Reliable,
  packets **cannot** be:
  - lost,
  - duplicated,
  - reordered
- Doesn't preserve message boundaries
- ("Unlimited" message size)

TCP Message Boundary Detection

Boundary implementation options

- Language independent “options”
  - Use fixed size messages
  - Delimit each message with a token
  - Sending the message size before actual message (this is the *best* option!)
- C# specific options:
  - BinaryFormatter, BinaryReader/Writer, StreamReader/Writer
    (but these also send/receive the message size below the surface)

Best option: Send message size before actual message!

- Principle:
  - Sender: *Send* msg size first, then message itself
  - Receiver: *Read* msg size first, then *loop* Read until whole message is received

- For example, to send/receive "hello":

  - Sender side:
    - Send message length → Send int 5 (4 bytes)
    - Send message itself → Send “hello” (5 bytes)

  - Receiver side:
    - Read message length (4 bytes) → the number 5
    - Read 5 bytes and convert to string → “hello”

Requirements

1. A way to send the message size as a **fixed amount** of bytes (why??).
2. A way to correctly read exactly x bytes.

Requirement 1 Sending size as a fixed amount of bytes

- If the message size is described by a variable amount of bytes, we still don't know how many bytes to read!
- This means we cannot use strings to send the message size: "1", "12", "154" → all different amount of bytes

- Solution:
  - use BitConverter class instead and convert int's to bytes instead of strings

Requirement 2 Reading x bytes correctly

```
private static byte[] Read(NetworkStream pStream, int pByteCount)
{
    //create a buffer to hold all the requested bytes
    byte[] bytes = new byte[pByteCount];
    //keep track of how many bytes we read last read
    int bytesRead = 0;
    //and keep track of how many bytes we've read in
    int totalBytesRead = 0;

    try
    {
        //keep reading bytes until we've got what we
        while (
            totalBytesRead != pByteCount &&
            (bytesRead = pStream.Read(bytes, totalBytesRead, pByteCount - totalBytesRead)) > 0
        )
        {
            totalBytesRead += bytesRead;
        }
    }
    catch { }

    return (totalBytesRead == pByteCount) ? bytes : null;
}
```

Read (stream, 200);

bytesRead (loop local)	totalBytesRead (offset)	pByteCount – totalBytesRead (size)

Adding some utility methods…

```
public static void SendMessage (NetworkStream pStream, byte[] pMessage) {
    pStream.Write (BitConverter.GetBytes (pMessage.Length), 0, 4);
    pStream.Write (pMessage, 0, pMessage.Length);
}

public static byte[] ReceiveMessage (NetworkStream pStream) {
    int byteCountToRead = BitConverter.ToInt32(ReadBytes (pStream, 4), 0);
    return ReadBytes (pStream, byteCountToRead);
}
```

```
public static void SendString (NetworkStream pStream, string pMessage, Encoding pEncoding) {
    SendMessage (pStream, pEncoding.GetBytes (pMessage));
}

public static string ReceiveString (NetworkStream pStream, Encoding pEncoding) {
    return pEncoding.GetString (ReceiveMessage(pStream));
}
```

(Test 004_tcp_message_boundaries_correct_demo with Clumsy)

Conclusion

- TCP provides a reliable mechanism to send/receive unique messages in order, and with some helper classes (custom or provided) we can “fix” the fact that it doesn’t preserve message boundaries by default.

- Examples of existing helper .Net classes which you could also use:
  - BinaryReader/Writer
  - StreamReader/Writer
  - Serializers such BinaryFormatter/Xml/Json

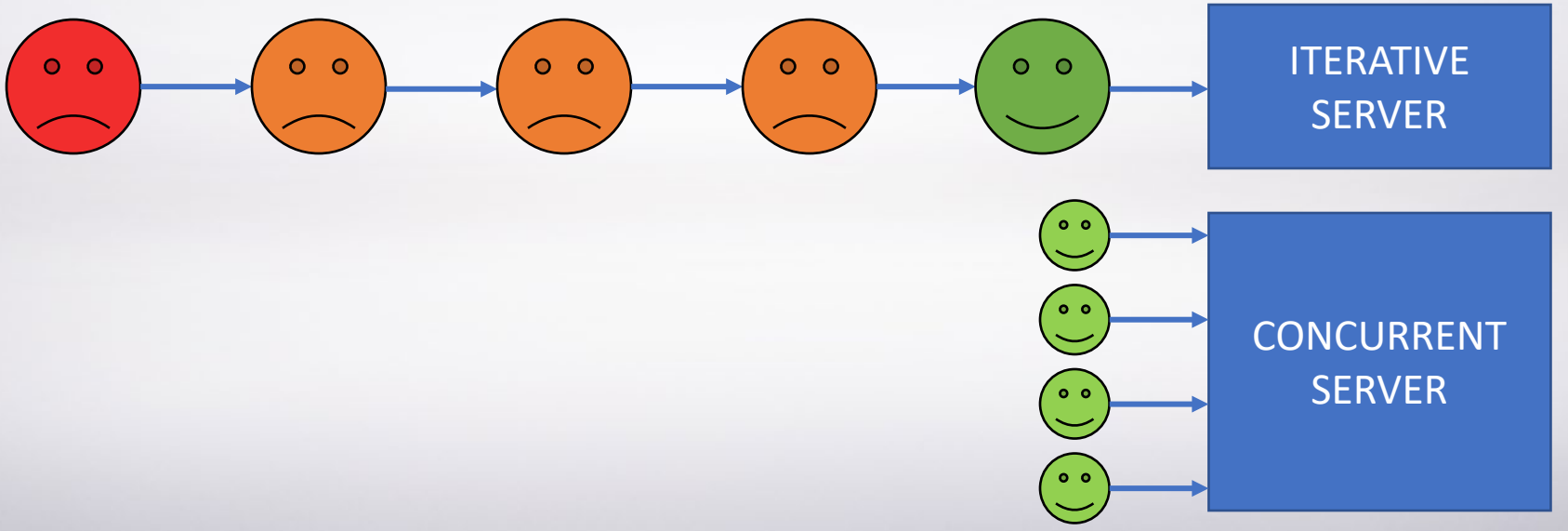
Blocking calls vs multiple clients

Serving multiple clients

- All examples showed 1 client/1 server
- Client 1 kept looping & the server kept serving…
- What will happen when client 2 arrives?
  - Client 2 will be put in a queue until Client 1 is done
- When is client 1 done?
  - When either the client or the server closes the connection!

Iterative vs Concurrent server

- A server that puts client 2 on hold while serving client 1 is called an **iterative** server
- A server that is able to serve multiple clients at the “same” time is called a **concurrent** server



How to move to a 'concurrent' server setup?

- Possible solutions:
  - Threading (manual thread creation, asynchronous callback methods etc)
  - Using sockets in non blocking mode, Select(...)
  - Timeouts & error handling
  - Polling
- Polling is by far the easiest solution: check if you ***would*** block ***before*** actually blocking, using:
  - if (listener.Pending()) ....
  - if (client.Available > 0) ....

Example 005_concurrent_tcp_echo_server

- Things to note:
  - Both client and server are using StreamUtil
  - Server serves multiple clients at 'once' by polling before blocking
  - Both client & server are still single threaded
  - No communication between clients
  - Assumes if TCPClient.Available > 0 a whole message is available.
    If a client only sends a message size and then quits without sending the rest, everything still blocks.
  - Disconnected clients are not detected (yet)
  - No error handling

Disconnecting clients & error handling

Error handling

- None of the examples showed error handling,
  while most networking calls can throw exceptions
- A crashing client is not so bad,
  but a server that crashes because the client crashed is pretty bad…
- How can we protect against that?
  - Error handling, e.g. try-catch (of course)
  - Remove faulty clients (otherwise errors keep happening)
    - Exceptions might keep occurring
    - We might keep waiting for a client to act indefinitely

Detecting faulty (disconnected etc) clients…

- Three different situations:
  - Client calls Close on Socket/Stream
  - Client shuts down without calling Close
  - Physical channel goes down (eg you pull the network plug)
- How can we detect this?
  - There is a so called TcpClient.Connected flag,
    but that only updates **after** sending
  - In other words: the flag we would like to use to prevent problems,
    only updates after a problem has already occurred

Handling errors & disconnects

- Good enough for now:
  - Try to send/receive data
  - Try catch any errors that occur (and print them!)
  - Remove clients whose Connected property is false

- In practice:
  - connections issues might take a long time to detect
  - this means using a heartbeat system would be even better
    - send out heartbeat every x seconds
    - keep track of last heartbeat received from other side
    - no heartbeat received for y seconds from other side? Assume painful death, exit.

"Evil" clients

What are the simple things evil clients could do?

- Clients could overload the server with messages,
  forcing the server to spin forever trying to process them
- Clients could ‘lie’ about their messages:
  - “I am going to sent you 5 bytes!”
  - *“Please hold while I am secretly not sending anything ghe ghe ghe”*
- Clients can stop reading from the buffer forcing server to block on send
- Clients could tamper with your messages (Clumsy / decompiling / etc)
- Although these are all 'fixable', assume ‘nice’ clients for now:
  - Clients might join & leave (and you are not allowed to crash due to that)
  - Clients will not intentionally try to stop/crash your server

Summing up ...

What have we seen today?

- The difference between UDP & TCP in practice and how to test your code under bad networking conditions (Example 001 & 002)
- How UDP & TCP deal with message boundaries and how we can deal with them ourselves (Example 003 & 004)
- How we can implement a concurrent server by polling (Example 005)
- Some general tips on dealing with (faulty, crashing & evil) clients
- In other words... everything you need for...

Introducing assignment 2

Implementing a chatbox in Unity

Assignment 2 – Chatbox

Assignment 2 – Implementing a chatbox

- Requirements plus starting code on BlackBoard
- Main difference with command line Tcp echo client server demo?
  - Client & server can no longer block
  - Incoming messages should be bounced to all connected users
  - Users need to be identified through their TcpClient handle
  - Messages can have meaning (setname, whisper, etc)
- Check **both** grading form & assignment requirements!
  - For example: Server is not allowed to crash (not explicitly mentioned in assignment)
- 99% similar to assignment 1 last year:
  - 'Getting started' section added
  - Minor details changed: be sure to double check if you are doing the redo!

Non reproducible errors are hard to fix

- Programs that are non deterministic
- Programs that run in different threads
- Programs that depend on multiple cooperating processes/external data
- Programs that depend on best-effort protocols

- In other words:
  - (threaded) client/server programs can be a nightmare to debug ☺
  - complete your implementation step by baby step
  - print logical debug info wherever possible