




# Network Programming

Ung Văn Giàu  
Email: [giau.ung@eiu.edu.vn](mailto:giau.ung@eiu.edu.vn)



# Synchronous and Asynchronous Programming

# Example: How to make a breakfast?

1. Pour a cup of coffee.
2. Heat up a pan, then fry two eggs.
3. Fry three slices of bacon.
4. Toast two pieces of bread.
5. Add butter and jam to the toast.
6. Pour a glass of orange juice.

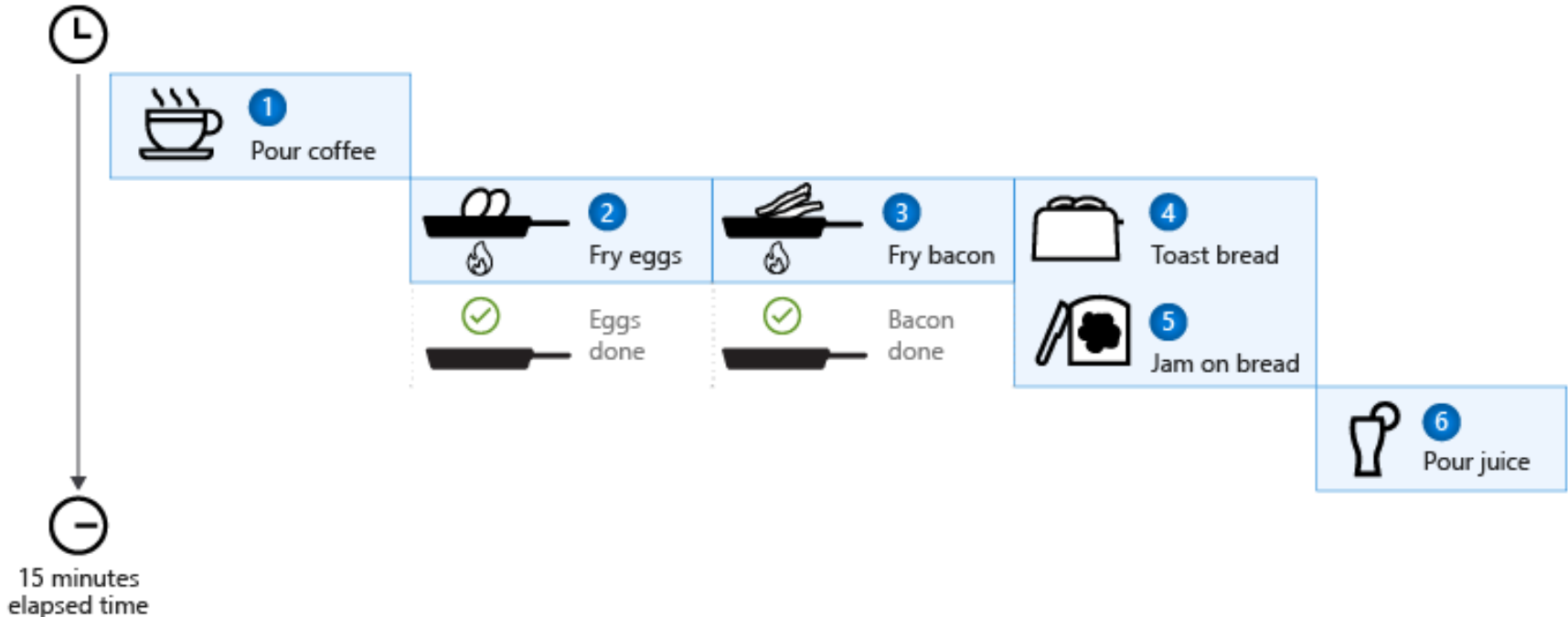
# Example: How to make a breakfast?

The synchronously prepared breakfast.

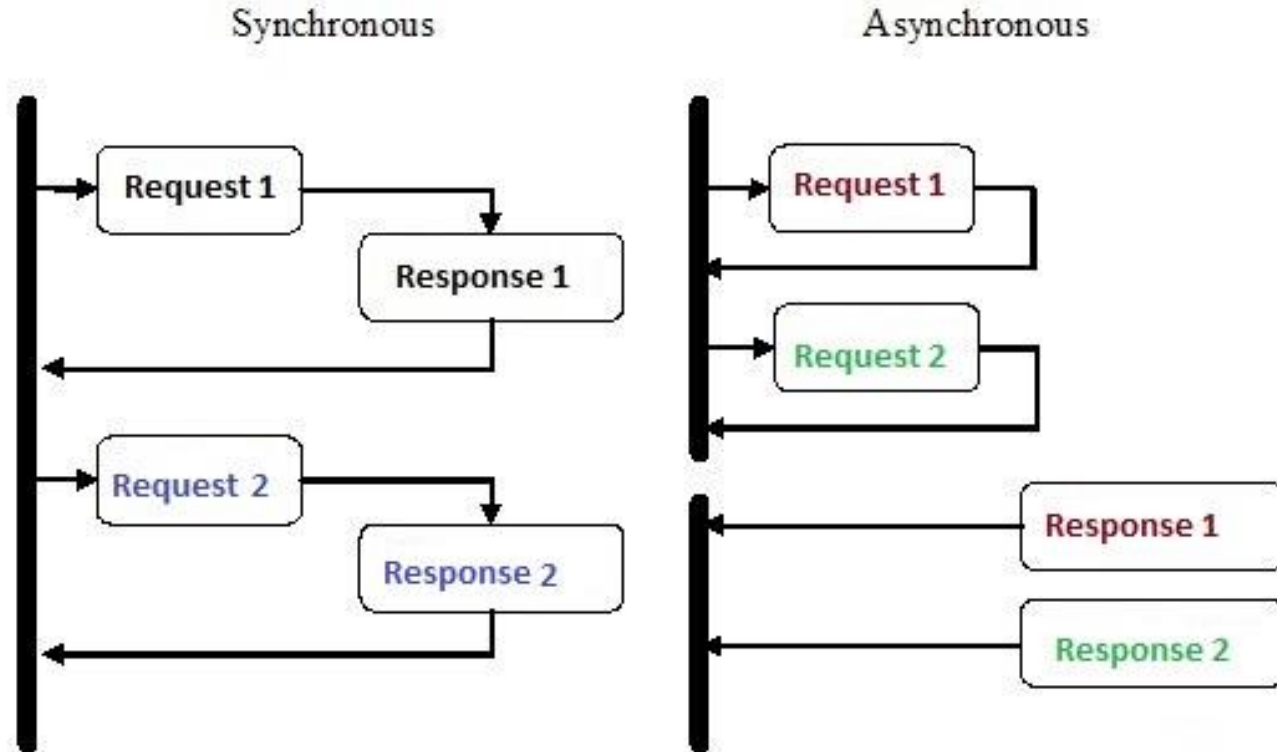


# Example: How to make a breakfast?

The asynchronously prepared breakfast.



# Synchronous and Asynchronous Programming



# What is Synchronous Programming?

- A synchronous call waits for the method to complete before continuing with program flow.
- It badly impacts the UI that has just one thread to run its entire UI code.
- Synchronous behavior leaves end users with a bad user experience and a blocked UI whenever the user attempts to perform some lengthy (time-consuming) operation.

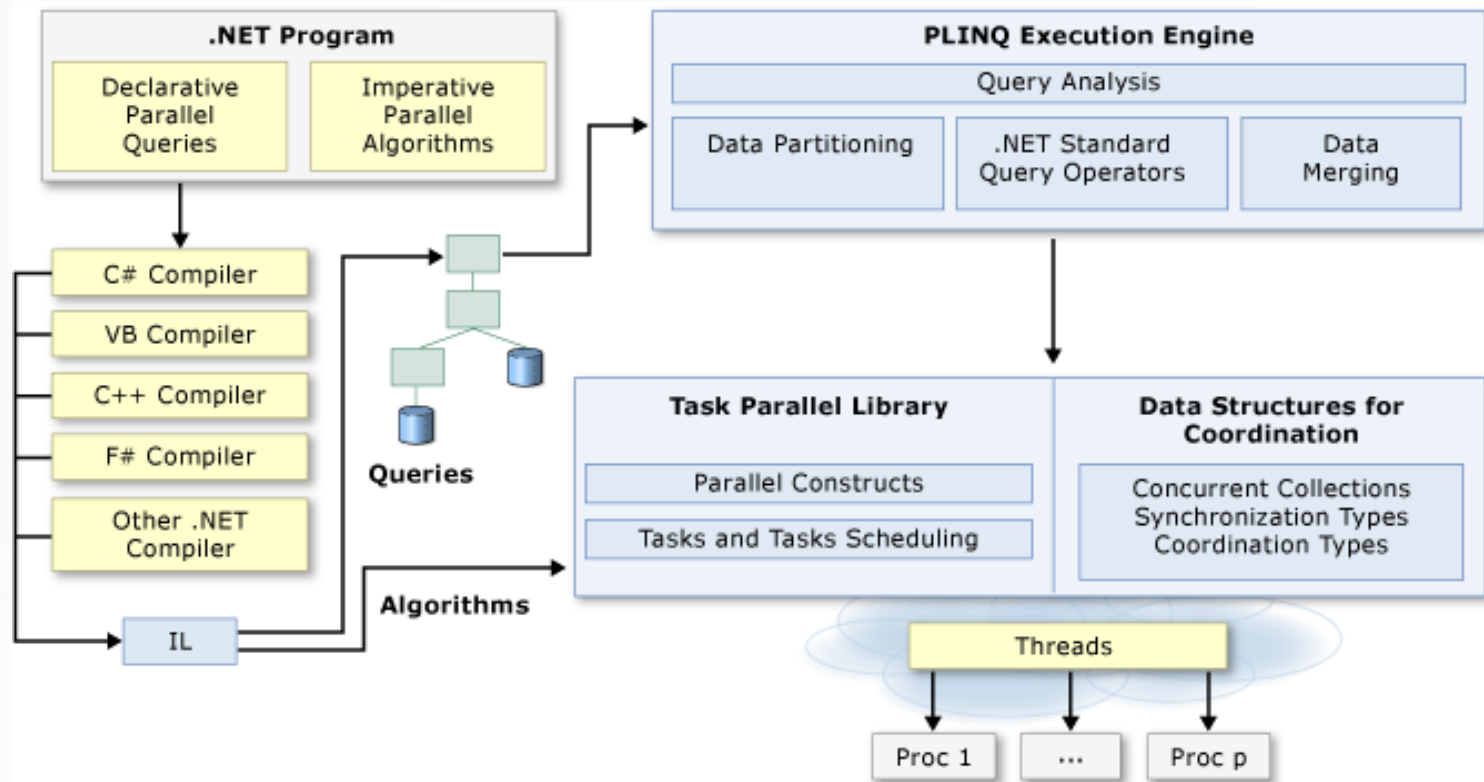


# What is Asynchronous Programming?

- An asynchronous method call will return immediately so that the program can perform other operations
- Improve program performance.
- The asynchronous method's behavior is more different than synchronous ones because the asynchronous method is a separate thread.



# Parallel programming



# Processes and threads

- A **process** is an executing program.
- A **thread** is the basic unit to which an operating system allocates processor time. Each thread has a **scheduling priority** and maintains a set of structures the system uses to save the thread context when the thread's execution is paused.
- **Multiple threads** can run in the context of a process. All threads of a process share its virtual address space.

# When to use multiple threads

- Multiple threads are used to increase the responsiveness of application and to take advantage of a multiprocessor or multi-core system.
- In a desktop application:
  - The **primary thread** is responsible for user interface elements and responds to user actions.
  - Use **worker threads** to perform time-consuming operations to make the user interface non-responsive.
  - Can use a **dedicated thread** for network or device communication to be more responsive to incoming messages or events.
- If your program performs operations that can be done in parallel.

# Thread Class

- Namespace: System.Threading
- Creates and controls a thread, sets its priority, and gets its status.
- Using constructors:
  - `Thread newThread = new Thread(MethodName);`
  - `Thread newThread = new Thread(new ThreadStart(MethodName));`
- Start the thread:
  - `newThread.Start();`
  - `newThread.Start("The answer.");`

# Thread Class

```
class Program
{
    static void Main()
    {
        Thread t = new Thread(new ThreadStart(MethodA));
        t.Start();
        MethodB();
    }

    static void MethodA()
    {
        for (int i = 0; i < 100; i++)
            Console.Write("0");
    }

    static void MethodB()
    {
        for (int i = 0; i < 100; i++)
            Console.Write("1");
    }
}
```

# Thread Class

How to pass data to the method the thread executes?

- **Only accept object type**
- In the callback, need to convert the **object** type to original type to use

# Thread Class

```
namespace ThreadExample
{
    class Student
    {
        public string Name { get; set; }
        public DateTime BirthDay { get; set; }
    }

    class Program
    {
        static void Main()
        {
            Thread t1 = new Thread(Print);

            t1.Start(new Student() { Name = "Yin", BirthDay = new DateTime(1989, 10, 17) });

            Console.ReadKey();
        }

        static void Print(object obj)
        {
            Student st = (Student)obj;
            Console.Write(st.Name + "\t" + st.BirthDay.ToShortDateString());
        }
    }
}
```

# Thread Class

- Common Properties

- **ThreadState:**

Gets a value containing the states of the current thread: Unstarted, Running, Suspended, Stopped, Aborted,...

- **Priority:**

Gets or sets a value indicating the scheduling priority of a thread.

A thread can be assigned any one of the following priority ThreadPriority values:  
Highest, AboveNormal, Normal, BelowNormal, Lowest



# Thread Class

- Common methods:
  - Interrupt(): Interrupts a thread that is in the WaitSleepJoin thread state.
  - Join(): Blocks the calling thread until the thread represented by this instance terminates.
  - Sleep(Int32): Suspends the current thread for the specified number of milliseconds.
  - Start(): Causes the operating system to change the state of the current instance to Running.
  - Start(Object): Causes the operating system to change the state of the current instance to Running, and optionally supplies an object containing data to be used by the method the thread executes.

# Thread Class

```
class Program
{
    static void Main()
    {
        Thread t1 = new Thread(MethodA);
        Thread t2 = new Thread(MethodB);
        Thread t3 = new Thread(MethodC);

        t1.Start();
        t2.Start();

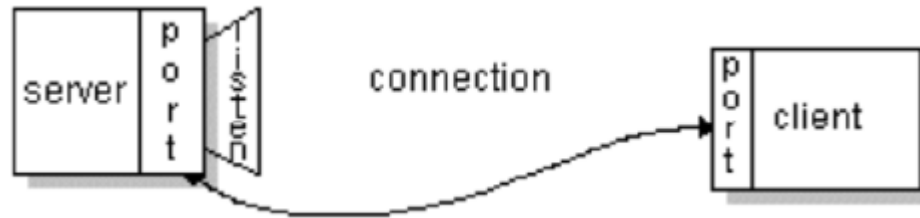
        t2.Join();

        t3.Start();
    }

    static void MethodA()
    {
        for (int i = 0; i < 100; i++) Console.Write("0");
    }
    static void MethodB()
    {
        for (int i = 0; i < 100; i++) Console.Write("1");
    }
    static void MethodC()
    {
        for (int i = 0; i < 100; i++) Console.Write("2");
    }
}
```

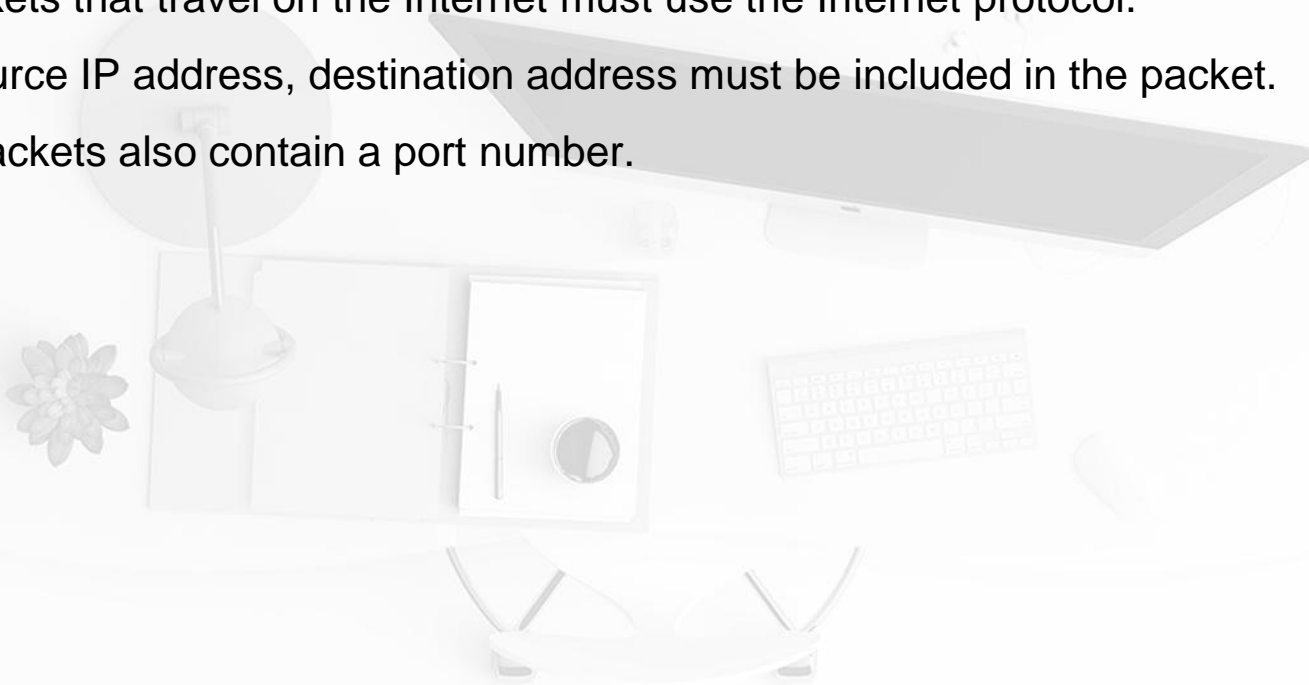
# What is a socket?

- A socket is **one endpoint of a communication link** between systems.
- Application sends and receives all of its network data through a socket.
- Data is generally sent in blocks of a few kilobytes at a time for efficiency; each of these blocks is called a **packet**.



# What is a socket?

- All packets that travel on the Internet must use the Internet protocol.
- The source IP address, destination address must be included in the packet.
- Most packets also contain a port number.



# Socket Class

- Namespace: System.Net.Sockets
- Implements the sockets interface.
- Common constructor:

Socket(SocketType, ProtocolType): Initializes a new instance of the Socket class using the specified socket type and protocol.

# Socket Class

- Common Methods:

- `Accept()`: Creates a new Socket for a newly created connection.
- `Bind(EndPoint)`: Associates a Socket with a local endpoint.
- `Close()`: Closes the Socket connection and releases all associated resources.
- `Connect(EndPoint)`: Establishes a connection to a remote host.
- `Connect(IPAddress, Int32)`: Establishes a connection to a remote host. The host is specified by an IP address and a port number.
- `Connect(String, Int32)`: Establishes a connection to a remote host. The host is specified by a host name and a port number.

# Socket Class

- Common Methods:

- `Dispose()`: Releases all resources used by the current instance of the Socket class.
- `Listen()`: Places a Socket in a listening state.
- `Receive(Byte[])`: Receives data from a bound Socket into a receive buffer.
- `Send(Byte[])`: Sends data to a connected Socket.

# Two types of sockets

- Sockets come in two basic types—connection-oriented and connectionless.
- These terms refer to types of protocols.
  - Transmission Control Protocol (TCP) (connection-oriented protocol)
  - User Datagram Protocol (UDP) (connectionless protocol)
- In a **connectionless protocol**, each data packet is addressed individually.

From the protocol's perspective, each data packet is completely independent and unrelated to any packets coming before or after it.
- In a **connection-oriented protocol**, each packet is numbered so that received packets are complete in the proper order



# Communicaiton Model

Network programming is usually done using a client-server paradigm.

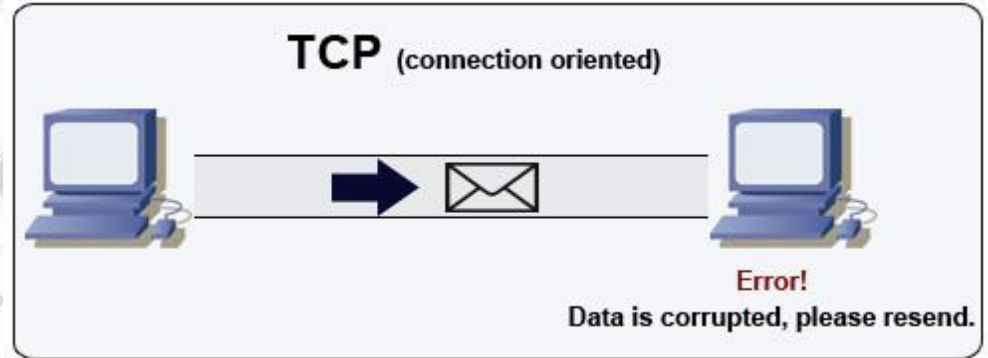
- A server listens for new connections at a published address
- The client, knowing the server's address, is the one to establish the connection initially.
- Once the connection is established, the client and the server can both send and receive data.
- This can continue until either the client or the server terminates the connection.

# Communicaiton Model

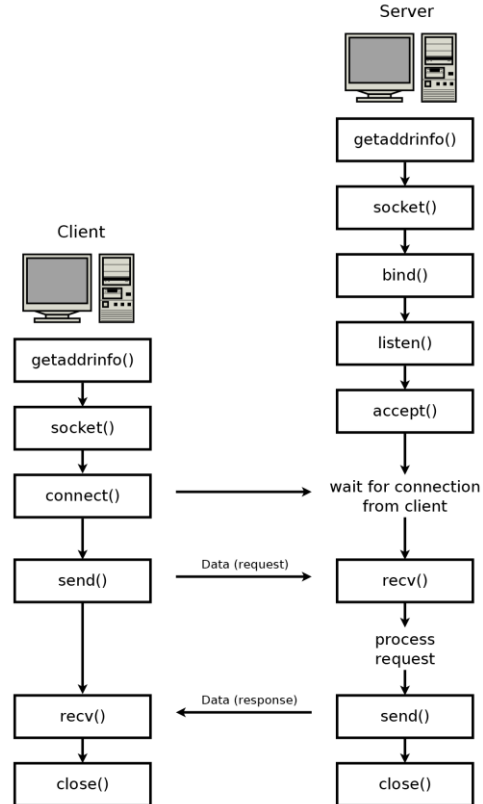
- A traditional client-server model usually implies different behaviors for the client and server.
- An alternative paradigm is the peer-to-peer mode.  
Each peer be a client and a server both.
- Another common protocol that pushes the boundary of the client-server paradigm is FTP.  
The FTP client first establishes a connection as a TCP client, but later accepts connections like a TCP server.

# TCP - Transmission Control Protocol

- TCP guarantees that data arrives in the same order it is sent.
- It prevents duplicate data from arriving twice, and it retries sending missing data.
- It also provides additional features such as notifications when a connection is terminated and algorithms to mitigate network congestion.
- TCP is used by many protocols.
  - HTTP (for serving web pages),
  - FTP (for transferring files),
  - SSH (for remote administration),
  - SMTP (for delivering email)



# TCP program flow



# TCP program flow

TCP client

- A TCP client program must first know the TCP server's address.
- The TCP client takes this address (for example, `http://example.com`) and uses the `getaddrinfo()` function to resolve it into a *addrinfo* structure.
- The client then creates a socket (`socket()`).
- The client then establishes the new TCP connection (`connect()`).
- At this point, the client can freely exchange data (`send()` and `recv()`).

# TCP program flow

TCP server

- A TCP server listens for connections at a particular port number on a particular interface.
- The program must first initialize a *addrinfo* structure with the proper listening IP address and port number.
- The server then creates the socket (*socket()*).
- The socket must be bound to the listening IP address and port (*bind()*).
- The server program then puts the socket in a state where it listens for new connections (*listen()*).
- The server can then call *accept()*, which will wait until a client establishes a connection to the server.

# TCP program flow

TCP server

- When the new connection has been established, *accept()* returns a new socket.
  - This new socket can be used to exchange data with the client using *send()* and *recv()*.
  - Meanwhile, the first socket remains listening for new connections, and repeated calls to *accept()* allow the server to handle multiple clients.

# NetworkStream Class

- Namespace: `System.Net.Sockets`
- The `NetworkStream` class provides methods for sending and receiving data over Stream sockets in blocking mode.
- You can use the `NetworkStream` class for both synchronous and asynchronous data transfer.
- To create a `NetworkStream`, you must provide a connected `Socket`.
- Read and write operations can be performed simultaneously on an instance of the `NetworkStream` class without the need for synchronization.



# NetworkStream Class

- Common constructor:

`NetworkStream(Socket)`: Creates a new instance of the `NetworkStream` class for the specified `Socket`.

- Common methods:

- `Close()`: Closes the current stream and releases any resources (such as sockets and file handles) associated with the current stream.
- `Read(Byte[], Int32, Int32)`: Reads data from the `NetworkStream` and stores it to a byte array.
- `Write(Byte[], Int32, Int32)`: Writes data to the `NetworkStream` from a specified range of a byte array.

# IPAddress Class

- Namespace: System.Net
- Provides an Internet Protocol (IP) address.
- The IPAddress class contains the address of a computer on an IP network.
- Common method:  
    IPAddress.Parse(String): Converts an IP address string to an IPAddress instance.

# EndPoint Class

- Namespace: System.Net
- Represents a network endpoint as an IP address and a port number.
- The IPEndPoint class contains the host and local or remote port information needed by an application to connect to a service on a host.
- Common constructor:  
IPEndPoint(IPAddress, Int32): Initializes a new instance of the IPEndPoint class with the specified address and port number.
- Properties:
  - Address: Gets or sets the IP address of the endpoint.
  - AddressFamily: Gets the Internet Protocol (IP) address family.
  - Port: Gets or sets the port number of the endpoint.

# TcpClient Class

- Namespace: `System.Net.Sockets`
- The `TcpClient` class provides simple methods for connecting, sending, and receiving stream data over a network in synchronous blocking mode.
- Common constructors:
  - `TcpClient()`: Initializes a new instance of the `TcpClient` class.
  - `TcpClient(String, Int32)`: Initializes a new instance of the `TcpClient` class and connects to the specified port on the specified host.

# TcpClient Class

- Common Methods:

- Close(): Disposes this TcpClient instance and requests that the underlying TCP connection be closed.
- Connect(IPAddress, Int32): Connects the client to a remote TCP host using the specified IP address and port number.
- Connect(String, Int32): Connects the client to the specified port on the specified host.
- GetStream(): Returns the NetworkStream used to send and receive data.

# TcpListener Class

- Namespace: System.Net.Sockets
- The TcpListener class provides simple methods that listen for and accept incoming connection requests in blocking synchronous mode.
- You can use either a TcpClient or a Socket to connect with a TcpListener.
- Common constructor:  
TcpListener(IPAddress, Int32): Initializes a new instance of the TcpListener class that listens for incoming connection attempts on the specified local IP address and port number.

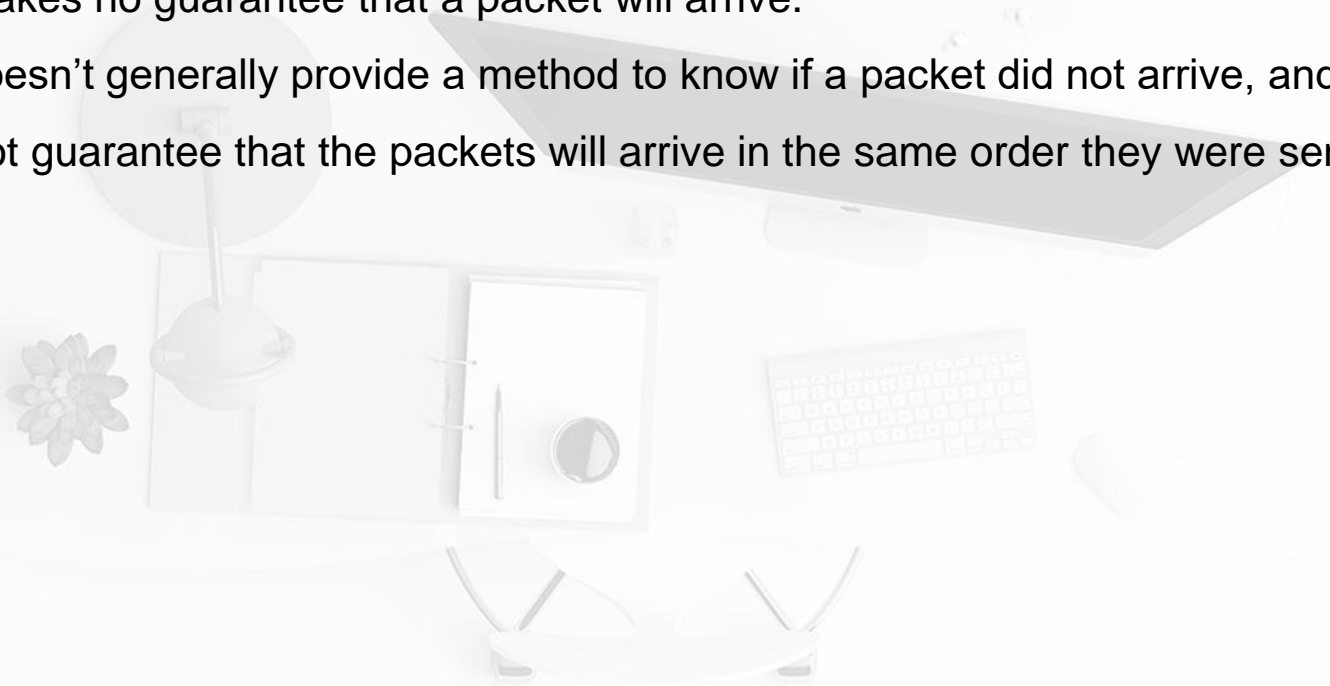
# TcpListener Class

- Common Methods:

- `AcceptSocket()`: Accepts a pending connection request.
- `AcceptTcpClient()`: Accepts a pending connection request.
- `Start()`: Starts listening for incoming connection requests.
- `Stop()`: Closes the listener.

# UDP - User Datagram Protocol

- UDP makes no guarantee that a packet will arrive.
- UDP doesn't generally provide a method to know if a packet did not arrive, and UDP does not guarantee that the packets will arrive in the same order they were sent.





# UDP - User Datagram Protocol

It less reliable, because it is possible that a single packet may arrive twice!

Why use UDP?

- **Speed: FAST**

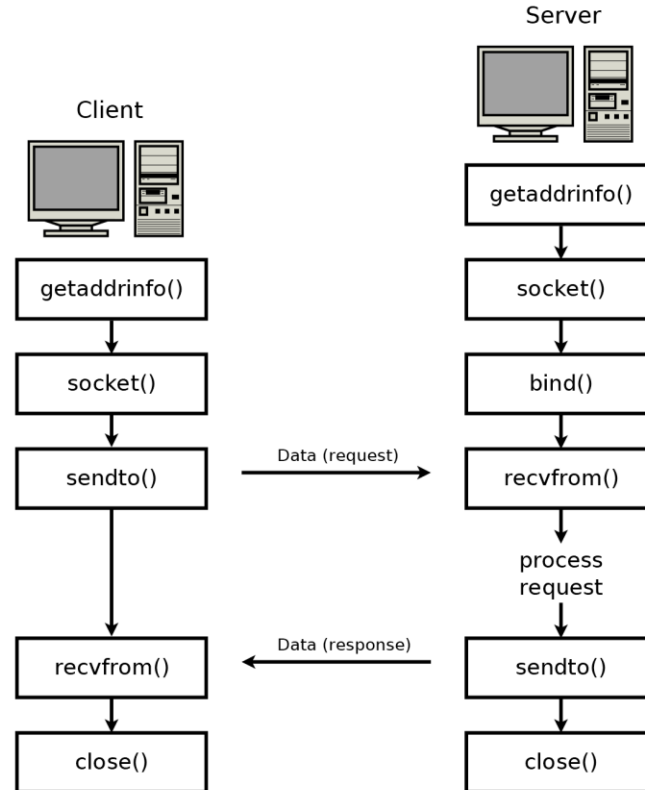
No acknowledgement required for each packet

- **Overhead: NO**

Simply send them then forget about them

- UDP is used by DNS (for resolving domain names).
- UDP is also commonly used in real-time applications, such as audio streaming, video streaming, and multiplayer video games.
- UDP useful when using IP broadcast or multicast.

# UDP program flow



# UDP program flow

## UDP client

- A UDP client must know the address of the remote UDP peer in order to send the first packet.
- The UDP client uses the `getaddrinfo()` function to resolve the address into a `addrinfo` structure.
- Once this is done, the client creates a socket of the proper type.
- The client can then call `sendto()` on the socket to send the first packet.
- The client can continue to call `sendto()` and `recvfrom()` on the socket to send and receive additional packets.
- **Note** that the client must send the first packet with `sendto()`.

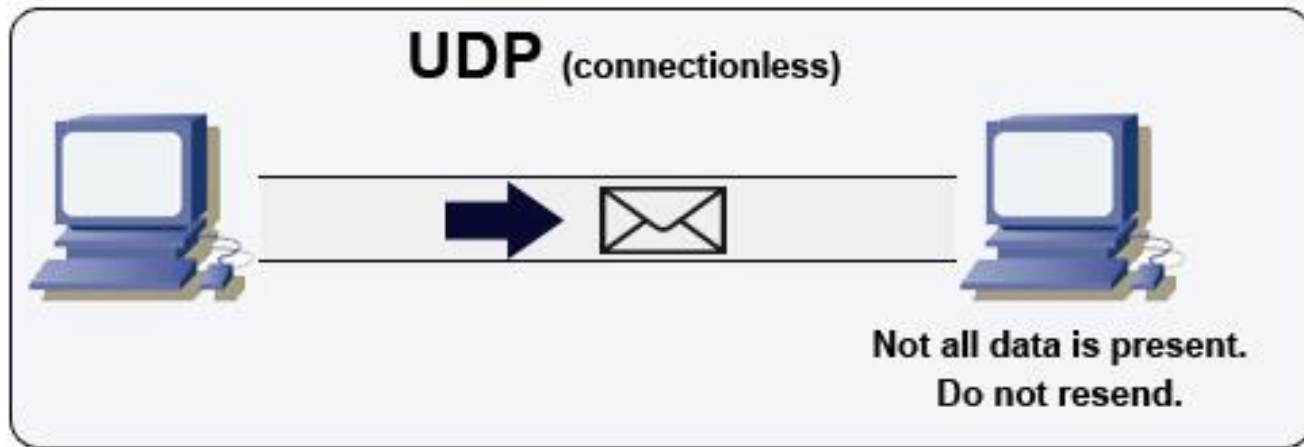
# UDP program flow

## UDP server

- A UDP server listens for connections from a UDP client.
- This server should initialize `addrinfo` structure with the proper listening IP address and port number.
- The server then creates a new socket with `socket()` and binds it to the listening IP address and port number using `bind()`.
- At this point, the server can call `recvfrom()`, which causes it to block until it receives data from a UDP client.
- After the first data is received, the server can reply with `sendto()` or listen for more data (from the first client or any new client) with `recvfrom()`.

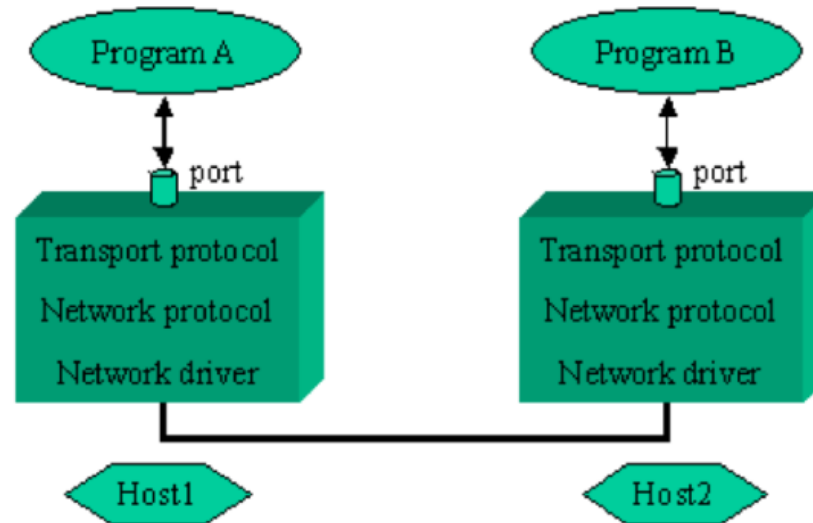
# UDP - User Datagram Protocol

- UDP is connectionless because the packets have no relationship to each other and because there is no state maintained



# UDP - User Datagram Protocol

- The destination IP address and port number is encapsulated in each UDP packet → uniquely identify the recipient



# UdpClient Class

- Namespace: System.Net.Sockets
- The UdpClient class provides simple methods for sending and receiving connectionless UDP datagrams in blocking synchronous mode.
- There two way to establish a default remote host:
  - Create an instance of the UdpClient class using the remote host name and port number as parameters.
  - Create an instance of the UdpClient class and then call the Connect method.

# UdpClient Class

- Common constructors:
  - `UdpClient()`: Initializes a new instance of the `UdpClient` class.
  - `UdpClient(Int32)`: Initializes a new instance of the `UdpClient` class and binds it to the local port number provided.
  - `UdpClient(String, Int32)`: Initializes a new instance of the `UdpClient` class and establishes a default remote host.
  - `UdpClient(IPEndPoint)`: Initializes a new instance of the `UdpClient` class and binds it to the specified local endpoint.



# UdpClient Class

- Common methods:
  - `Connect(IPAddress, Int32)`: Establishes a default remote host using the specified IP address and port number.
  - `Connect(IPEndPoint)`: Establishes a default remote host using the specified network endpoint.
  - `Connect(String, Int32)`: Establishes a default remote host using the specified host name and port number.
  - `Receive(IPEndPoint)`: Returns a UDP datagram that was sent by a remote host.
  - `Send(Byte[], Int32)`: Sends a UDP datagram to a remote host.
  - `Send(Byte[], Int32, IPEndPoint)`: Sends a UDP datagram to the host at the specified remote endpoint.
  - `Send(Byte[], Int32, String, Int32)`: Sends a UDP datagram to a specified port on a specified remote host.

# Comparison TCP to UDP

| TCP   | UDP  |
|---|--|
| Reliable  | Unreliable   |
| Connection-oriented   | Connectionless   |
| Data used for sending does not contain IP address and port of the remote host | Data must contain IP address and port of the remote host |
| Check errors and retransmission   | No retransmission  |
| Segment sequencing  | No sequencing  |
| Ack (acknowledge) segments  | No ack   |
| Slower  | Faster   |



**Q&A**