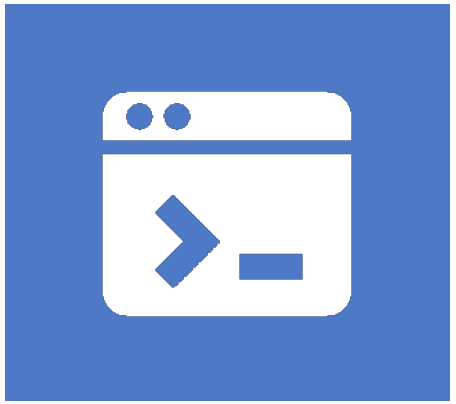
Advanced Game Programming



Week 6

Homework Review

Serialization

Simple Networking

Basics and Practices

Devices

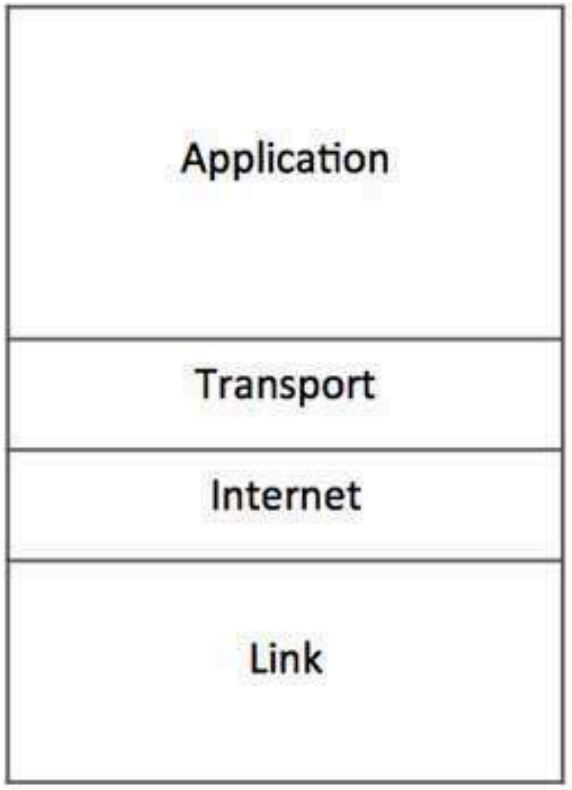
- Every device that’s connected to a network has:
  - IP Address, or “Logical Address”
  - MAC Address or “Physical Address”
  - Ports
    - Logical channel through which data can be sent/received to an application.
    - Number between 0 and 65535
- A single connection is a socket, which is made up of an IP Address and port number
  - 172.217.10.46:80 -> google.
  - 127.0.0.1:3000 -> your local device on port 3000

Terminology

- Socket: Single connection endpoint
- Ping: Roundtrip time between two computers. (Highest contributor to lag)
- Server: Centralized computer designed to send and receive information from clients
- Client: Individual computers that are trying to get resources from servers.
- Packet: Information / data that’s traveling on network

IP Layer

- Sits at “internet” layer
- Transmit a packet from a source to a destination
- No guarantee that:
  - packet will eventually arrive
  - packet will arrive only once
  - data is not corrupted
  - Sequence of packets will arrive in order
- Limited in size.



TCP/IP Reference Model

Transport Protocols

- Protocol to transport the data between the server and the clients.
- TCP:
  - Reliable, Ordered, Error-checked
  - Slow
- UDP:
  - Thin layer above IP – doesn’t guarantee anything extra
  - Quick
- When making a real-time multiplayer game, you may make your own
  - UDP+

Server Types

- Dedicated Server
  - Simulates game world (runs physics, moving characters, determining outcome of actions)
- Relay Server
  - Sends and receives information between clients
- Master Server
  - Provide clients with information about all dedicated servers
  - Used for matchmaking
  - Also called “listing servers”

Dedicated Server

- Pay for a centralized server all clients connect to.
- Pros:
  - More powerful hardware for server
  - Higher bandwith connection
  - No player advantage (except ping)
  - Clients don’t see each other’s addresses
- Cons:
  - Expensive for publisher/studio
  - Need multiple server locations

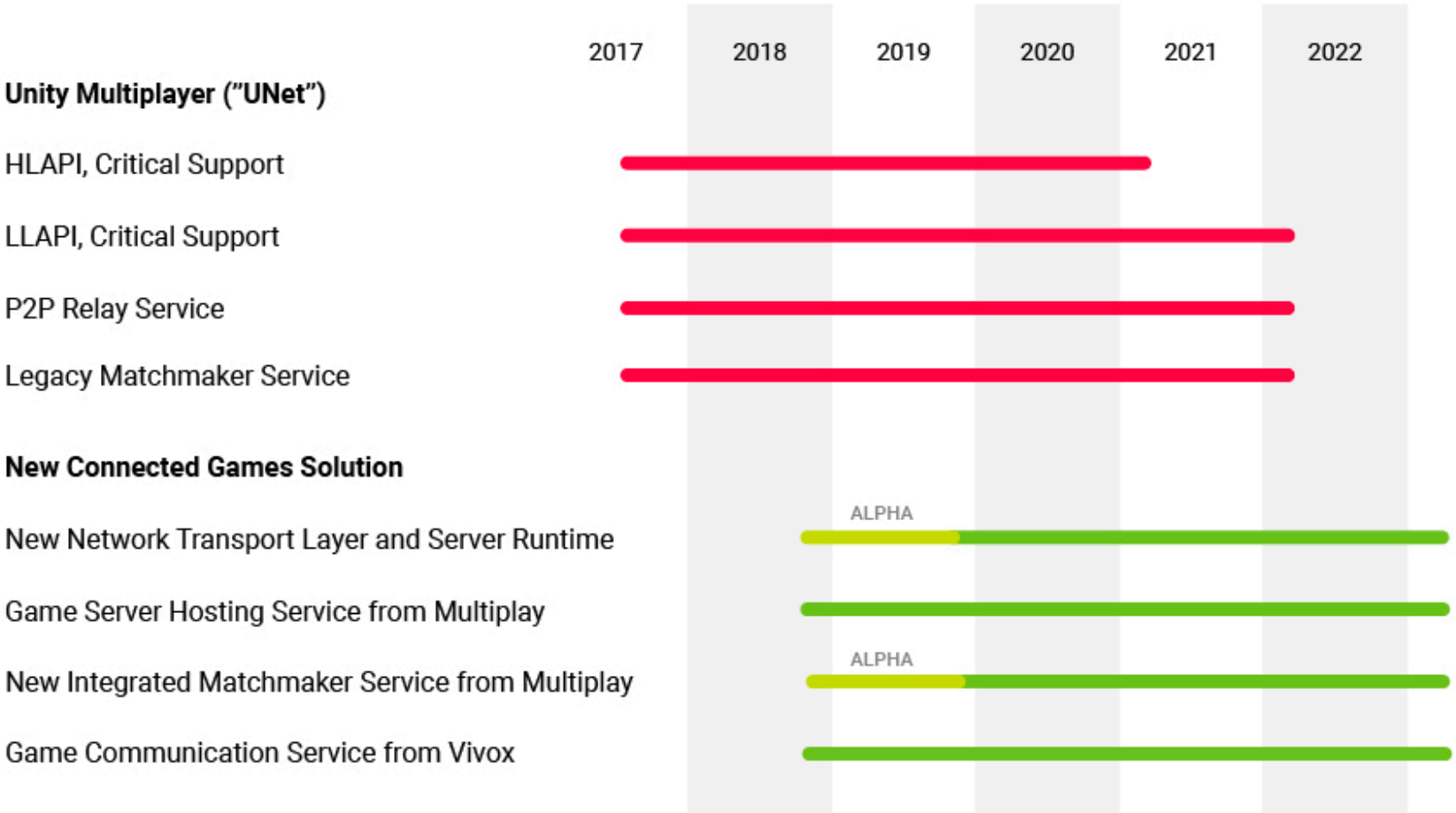
Client Host

- One client serves as the host
- Pros:
  - Little extra cost for publisher
  - Players at remote locations can play together w/out lag
- Cons:
  - Host player advantage
  - Host is on consumer-grade internet
  - Host is using underpowered hardware / same hardware for game and server
  - Host can see IP addresses of other players
  - Easy for host to cheat.
  - Clients that are far from each other have massive lag

Peer to Peer

- One client is the “session host”
- Every client runs it’s own, synchronized simulation
- Pros:
  - Little cost to publisher
  - Players in remote locations can play together w/out lag
  - No host advantage
- Cons:
  - Game pauses if current session host leaves
  - Every player sees IP of every other player
  - Same effects of low-powered hardware and consumer grade internet
  - Easy to cheat

Unity Specific Integration



Unity Real-time Options

- Mirror (Free / Open Source)
- Photon Bolt
- Photon Unity Networking 2 (PUN2)
- Forge Networking Remastered
- Whatever Unity Is Currently Working On – Currently in alpha

Network Serialization

- The less data, the better
- The more compressed the data, the better.
- Avoid bloat by serializing to binary, and serializing in uniform sizes as small as possible.
  - Don’t make your 100 person game serialize player ID’s to 4 byte ints, for example.

Hypertext Transfer Protocol (HTTP)

- Enables communication between clients and servers
- Comes with multiple methods:
  - GET
  - PUT
  - POST
  - HEAD
  - DELETE
  - PATCH
  - OPTIONS
- Mostly concerned with “get” and “put”

GET Request

- GET is used to request data from a specified resource.
- Can also contain a “query string”
  - EX: “someURL.com/someRoute?name1=value1&name2=value2”
  - Server can separate out key value pairs from a query string and change it’s response
- GET requests are only used to request data (not modify)

PUT and POST Methods

- PUT and POST requests put resources on a server
  - The same PUT request multiple times will puts the resource on the server once
  - The same POST request multiple times will make multiple copies of the resource

Other Methods

- HEAD Method: Identical to GET, without returning a body
  - Used to check what will happen if you make a GET request
- DELETE Method: Deletes a resource off the server
- OPTIONS Method: Describes the communication options for the target resource

GET vs. POST for sending data

	GET	POST
BACK button/Reload	Harmless	Data will be re-submitted
Restrictions on data length	GET method adds the data to the URL; and the maximum URL length is 2048 characters	No restrictions
Restrictions on data type	Only ASCII characters allowed	No restrictions. Binary data is also allowed
Visibility	Data is visible to everyone in the URL	Data is not displayed in the URL

Avoiding Deadlock

- Working with the internet inherently takes time
- Unity programs are single threaded
  - If you ”wait” until you get a response, everything else stops.
- You can

DON’T DO THIS

```
private bool IsConnected(string testURL = "http://www.google.com")
{
    var html = string.Empty;
    var req = (HttpWebRequest) WebRequest.Create(testURL);
    try
    {
        using (var resp = (HttpWebResponse) req.GetResponse())
        {
            var isSuccess = (int) resp.StatusCode < 299 && (int) resp.StatusCode >= 200;

            return isSuccess;
        }
    }
    catch
    {
        return false;
    }
}
```

DON’T DO THIS

DON’T DO THIS

DON’T DO THIS

What's the problem?

```
Uri ourUri = new Uri(url);

// Create a 'WebRequest' object with the specified url.
WebRequest myWebRequest = WebRequest.Create(url);

// Send the 'WebRequest' and wait for response.
WebResponse myWebResponse = myWebRequest.GetResponse();

// Use "ResponseUri" property to get the actual Uri from where the response was attained.
if (ourUri.Equals(myWebResponse.ResponseUri))
    Console.WriteLine("\nRequest Url : {0} was not redirected",url);
else
    Console.WriteLine("\nRequest Url : {0} was redirected to {1}",url,myWebResponse.ResponseUri);
// Release resources of response object.
myWebResponse.Close();
```

https://docs.microsoft.com/en-us/dotnet/api/system.net.webrequest.create?view=netframework-4.8

What are some fixes?

- In vanilla C#:
  - WebRequest.GetResponseAsync() - Microsoft
  - WebRequest.BeginGetRequestStream(AsyncCallback, Object) – Microsoft
- In Unity
  - Yielding in a coroutine to UnityWebRequest.SendWebRequest()

Doing it Right

```
private string data;
private IEnumerator GetData(string URL, int port)
{
    var www = UnityWebRequest.Get(URL + ":" + port);
    yield return www.SendWebRequest();

    if(www.isNetworkError || www.isHttpError) {
        Debug.Log(www.error);
        data = "";
    }
    else {
        data = www.downloadHandler.text;
    }

}
```

Delegates

Callbacks, Synchronization, Actions in Time

Why Use Delegates?

- Delegates allow methods to be passed as parameters.

- Delegates can be used to define callback methods.
  - Say what method you want to run based on success or failure *in time*.
- Delegates can be chained together
  - We used this for events
- Delegates allow methods to be explicitly encapsulated in objects
  - Can define methods to *do later*.

Callback Pattern

- Callbacks are passing delegates to another system to be run based on future state

```
public IEnumerator GetPlayerDetailsFromServer(Action onSuccess, Action onFailure) {
    yield return new MakeServerRequest();

    if (playerDetails == null)
        onFailure();
    else
        onSuccess();
}

// At start of game:
GetPlayerDetailsFromServer(StartMatch, ShowCouldntConnect);

// In game:
GetPlayerDetailsFromServer(UpdateEnemyPosition, DisconnectFromServer);
```

Task Pattern

- A common pattern is a “task runner” – or an object (runner) that takes in objects (tasks) to do in some order.
- Examples
  - You have an animation
  - You need to move a unit to a new location
  - Need to add enemies to a wave at an interval (i.e. not all at once)
  - You need to first load a level file, then reset the level, then unpause the game.
  - When you destroy a set of gems in a match-3 game you need to do a destroy animation for the gems, then animate all the gems on top collapsing and then drop in new gems.

Example Code

Example code in Github.

What is a lambda expression (finally!)

- A lambda is the *data* of a function.
- Like any data type, you can use the data with out giving it a name.

```
// At start of game:
GetPlayerDetailsFromServer(StartMatch, () => Debug.Log("Couldn't connect."));
```

- They can have parameters:

```
var x = (int someNumber, float ofParameters) => {
    // What happens in the lambda expression
};
```

Command Pattern

- Commands are an object-oriented replacement for callbacks.
- Instead of passing a method as a *parameter*, you store it as an *object*
- Useful for:
  - Doing actions some time later, to be determined
  - Reconfigurable control schemes
  - Undoing actions

Reconfigurable Controls

```
if (Input.GetKeyDown(KeyCode.A))
    Jump();
if (Input.GetKeyDown(KeyCode.S))
    Hug();
if (Input.GetKeyDown(KeyCode.D))
    GrabItem();
```

Reconfigurable Controls (cont.)

```
public class Command {
    public Action execute;

    public virtual void Do() {
        execute();
    }
}
```

```
public class Jump : Command {
    public Jump() {
        execute = () => Jump();
    }
}

public class Hug : Command {
    public Hug() {
        execute = () => {
            DropItem();
            HugNearest();
        };
    }
}
```

Reconfigurable Controls (cont.)

```
public class InputHandler : Monobehavior {
    Command AButton, SButton, DButton;

    private void Update() {
        if (Input.GetKeyDown(KeyCode.A))
            AButton.Do();
        if (Input.GetKeyDown(KeyCode.S))
            SButton.Do();
        if (Input.GetKeyDown(KeyCode.D))
            DButton.Do();
    }
}
```

Reconfigurable Controls

- Easy to make screens that let players set buttons.
- Easy to assign different actions depending on gamestate (sneaking, hiding, cooking, etc.) without a state machine (heavy in some cases)

How can we extend?

- What if the Command Pattern took in GameActors?

```
public abstract class Command {
    public abstract void Do(GameActor actor);
}


public class Jump : Command {
    public override void Do(GameActor actor) {
        actor.Jump();
    }
}
```

- Any GameActor can jump off of a command.

How else can we extend?

- What if the Input Handler returned a command?

```
private Command HandlInput() {
    if (Input.GetKeyDown(KeyCode.A))
        return AButton;
    if (Input.GetKeyDown(KeyCode.S))
        return SButton;
    if (Input.GetKeyDown(KeyCode.D))
        return DButton;

    return null;
}
```

```
    public Update() {
        var command = HandlInput();
        command?.Do();
    }
}
```

- Your AI can just generate commands for your NPC's to execute.

Why decouple decision-making from action?

- Use different AI modules for different actors.
- Mix and match AI for different kinds of behavior. Want a more aggressive opponent? Just plug-in a more aggressive AI to generate commands for it.
- Bolt AI onto the *player's* character
  - Useful for demo mode, cutscenes
- Do actions later (like in turn-based games).

Finally, Undoing

- If you keep track of the Actor doing an action, and add an “undo” action, you can have an undo button.

```
public abstract class Command {
    public Action do, undo;
    public GameActor actor;
}
```

- Just push Commands onto a Stack as you do them.

Resources

Articles/Documents

- Breakdowns of Unity Networking Frameworks (link1, link2)
- What Every Programmer Needs to know about Game Networking (link)
- Game Networking Resources (link)
- Gaffer On Games (link)
- Mattia Romeo's GPP Page (link)

Videos

- Netcode 101 (link)
- GDC: Overwatch's Netcode (link)
- GDC: I Shot You First! (link)