NETWORKING IN GAMES

- Many games are networked -- even single-player experiences
- What sort of data are being transmitted?
- Where does the data come from?
- How is the data being processed?
PERSISTENT VS TRANSIENT WORLDS

- World data can either be generated per-session (transient) or stored between sessions (persistent)
  - Choice depends on type of game/experience and studio’s budget
- Transient games can have one of the players act as a host
- Persistent games require a dedicated server(s)
CLIENT-SERVER MODEL

- Common model for deciding how to distribute data in both persistent and transient worlds
- Server is the authority on game state
  - Decides what the clients see in the game
  - Determines what and how client actions can change the game state
- In transient games, the server can be the player’s system that all clients connect to
PEER-TO-PEER (P2P) MODEL

- Model for deciding how to distribute data in transient worlds
- No one peer is the authority
  - Resource management distributed across peers
  - Each peer determines how other peers are influencing game state accordingly
- Useful in genres like fighting games where both peers have equal authority and games have limited world state
GAME SERVERS

- We will focus on client-server setups as they are more common in games.
- In the client-server model, game servers manage final version of world state.
- Several ways to manage this:
  - Perform all calculations on server.
  - Perform some calculations on server and some on clients.
  - Perform all calculations on client and allow server to determine "ground truth" from these calculations.
PERFORMING ALL CALCULATIONS ON CLIENTS?

- Not a great idea
  - Too much security risk
  - All the overhead of a P2P network with none of the benefits
- Not really done in practice
At first glance, this is the safest and easiest way to manage game state.

- All of world state (including player information) is replicated from the server.
  - i.e. Clients see a copied version of the current world state.
- When a client provides controller input, input is sent to server to be processed.
- Server performs actions based on valid input, updates its world state, then sends this updated data to all clients.

What problems arise from this setup?
LATENCY AND LAG

- Latency is the time it takes from starting to do something to finishing it.
- Lag in user interaction is the latency from when a user provides input to the time they see the response.
- Ideally we want to process user input every ~16ms (60Hz) or more.
  - Worst case (i.e. consoles) we process input at a fixed rate of ~33ms (30Hz).
  - Assumes humans see at around 30Hz* ensuring good responsiveness even if the game frame is out of sync with our eye “frame”
- Handling player inputs on the server introduces network latency into the existing lag of user interaction.
  - Will not be responsive.

*This is a gross simplification of human vision but it works well enough in practice.
HOW TO HANDLE PLAYER INPUT LATENCY?

- Allow client to perform latency-sensitive actions autonomously

- Action performed on client before being verified on the server
  - If server and client agree, action is replicated to all other clients
  - If server and client disagree, server adjusts client’s world state to match the server state
UE4: CHARACTER MOVEMENT COMPONENT

- Uses three network roles:
  - **Autonomous Proxy** is character on owning client’s machine
  - **Authority** is character on the server
  - **Simulated Proxy** is character on non-owning client machines
- Replication happens at 30Hz
AUTONOMOUS PROXY CHARACTER

- Locally controlled by owning player
- Runs `PerformMovement` locally to determine physical logic of character
  - Highly responsive with no network latency
- Stores movement data in `FSavedMove_Character` and queues these into `SavedMoves`
- Sends condensed version of data to server
**AUTHORITY CHARACTER**

- Updated by server when server receives `SavedMoves`
- Server checks updated position and orientation of character against the client’s reported position and orientation
- If values match, server informs owning client their movement was valid
- If values do not match, server sends corrections to owning client to fix autonomous proxy’s values
  - Autonomous proxy reproduces authority’s movements and retraces steps based on `SavedMoves`
  - Autonomous proxy only removes moves from `SavedMoves` after movement is successfully resolved
SIMULATED PROXY CHARACTER

- Movement information is replicated from server
- Used for all characters, both AI (controlled on server) and players (autonomous proxies)
- Network smoothing used to clean up motion on client’s end
  - Interpolates between current location and target location using SmoothClientPosition
HOW DO MACHINES COMMUNICATE?
RPCS

- Remote Procedure Calls
- Allows for the execution of code in a different address space as though it were a local call
  - Can use for both remote and local calls
- Message-passing mechanism hidden
- Remote and local calls can be handled based on role
USING NETWORKING IN UE4

- Must include “Net/UnrealNetwork.h”
- Include Replicated keyword in UPROPERTY to replicate an Actor’s property
- Set bReplicates in the replicating Actor to true
- Implement function
  GetLifetimeReplicatedProps(TArray<FLifetimeProperty>& OutLifetimeProps) in replicating Actor
    - Add DOREPLIFETIME(AMyActor, PropertyName); for each property being replicated
- UE4 handles replicated pointers using GUIDs (Globally Unique Identifiers)
  - Server assigns FNetworkGUID value and clients are notified
REPNOTIFY

- Allows execution of a function when a variable’s value changes
  - Each property specifies the function it will call
- Specify with `ReplicatedUsing = OnRep_PropertyName` instead of `Replicated in UPROPERTY`
- Create `OnRep_PropertyName()` function that will be called
  - This will specify what should happen when the value is changed
- Can update local (non-replicated) assets using these
UE4 NETWORKING FUNCTIONS

- **UFUNCTION** must specify who is executing the function and how reliable the function needs to be
  - **Server** only executes the code on the server
  - **Client** only executes the code on the owning client
  - **NetMulticast** executed on the server will also execute on all clients


- Functions must use a `_Implementation thunk`

- Server must have specifier `WithValidation` and implement an additional `_Validate` function
UFUNCTION(Server, Reliable, WithValidation, BlueprintCallable)

void Server_myFunction();

void Server_myFunction_Implementation();

bool Server_myFunction_Validate();

Reliably calls Server_myFunction(). Can be called from any owning client but will only perform the function on the server. Can be called from Blueprints.
void AMyActor::Server_myFunction_Implementation()
{
    //Execute what the server should do here
}

bool AMyActor::Server_myFunction_Validate()
{
    //Perform necessary validation of function here
    return true;
}

Only implement the _Implementation() thunk. Must include _Validate() to work.
UFUNCTION(Unreliable, Netmulticast)

void Netmulticast_myFunction();

void Netmulticast_myFunction_Implementation();

Unreliably calls Netmulticast_myFunction(). If called from the server, will execute on all clients.
void AMyActor::Netmulticast_myFunction_Implementation()
{
    //Execute what the server and all clients should do here
}

Only implement the _Implementation() thunk.
WHAT TO REPLICATE?

- Very challenging software architecture question!
- For any project that may require networking, you want to *build networking in as soon as possible*
- Must choose what will be controlled on the server versus the clients
- Common things the server replicates:
  - The world itself
  - Interactables in the world
  - Playable characters
- Common things to run locally:
  - GUI and HUD
  - Certain animations
  - Anything only relevant to the owning player
WHEN TO REPLICATE RELIABLY?

- Replicate as unreliably as possible
  - State-related changes should always be replicated reliably
  - Anything cosmetic or frequently sent can be replicated unreliably
- Only replicate what is important to the clients
  - Do not replicate world or player information that will not effect the client
RPCS AND OWNERSHIP

- Ownership determines how and where these functions are called
  - If Actor is owned by server, RPC is called on server
  - If Actor is owned by a client, RPC needs to know which client
- PlayerController can be an owning connection of an Actor (e.g. a Pawn)
  - When Pawn is possessed by PlayerController, it is owned by that PlayerController’s connection
  - No longer owned by PlayerController’s connection when unpossessed
ROLE AND REMOTE ROLE

- Actors have a Role and a RemoteRole property

- Roles are: ROLE_Authority, ROLE_SimulatedProxy and ROLE_AutonomousProxy
  - Simulated Proxy used for Actors controlled by server (client updates values accordingly)
  - Autonomous Proxy used for Actors controlled by a player (client considers values from input in addition to values passed down by server)

- Note: Roles will change depending on who is inspecting the values

- Example: Actor is owned by server and simulation is passed to clients
  - On server, sees Role == ROLE_Authority and RemoteRole == ROLE_SimulatedProxy
  - On client, sees Role == ROLE_SimulatedProxy and RemoteRole == ROLE_Authority
UE4 REPLICATION GRAPHS

- Designed to handle the large number of players and Actors in Fortnite without taxing CPU as heavily or having a laggy experience due to less frequent updates

- Replication Graph contains nodes with information on how/when to send data to clients about Actors
  - Do clients ever need to receive updates on this Actor?
  - When will specific clients need to receive updates on this Actor and how frequently?

- Graph system designed to be flexible to suit the needs of the project
  - Consider when and how replication occurs and create data structure accordingly
THINK ABOUT THE REPLICATION CONSIDERATIONS IN THE SCENE BELOW...