Module 7

Multiplayer, Networking, and Data Persistence

7.1 Replication - Remote Events & Remote Functions

This part of the course will be extremely technical, but it's fundamental to how Roblox games work. Roblox has two sources that code can run from, Client and Server. The client is used for anything that needs to be displayed, and the server is used for anything that needs to be authenticated or replicated.

We've talked before about how can Client and Server communicate with each other. The solution is Remote Functions/Events.

The typical structure for organizing these Remotes is to create a folder in ReplicatedStorage called Remotes. This folder stores all Remote Events and Functions, making them accessible to both the client and server.

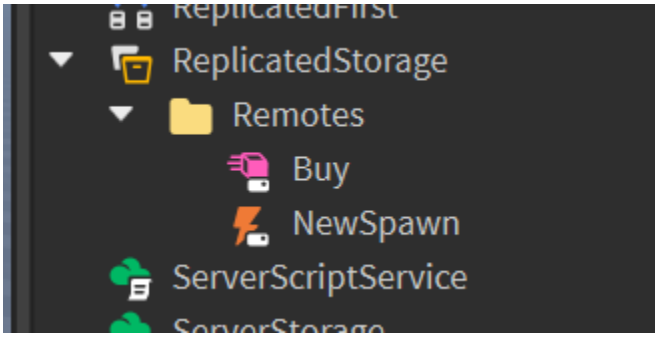


Fig 1

* Not everything needs to be server authoritative. Characters are on the client and automatically replicate to all clients.
* Network ownership of unanchored objects

7.2 Understanding Remote Events

Remote Events are used when the client needs to send a request to the server, or the server needs to notify clients about something happening in the game.

- FireServer() – A client calls this function to send a message to the server.
- FireClient() – The server uses this to send data to a specific client.
- FireAllClients() – The server sends data to all clients at once.

- Example Use Case: Buying an Item from a Shop
- Imagine a Tycoon game where players purchase upgrades using an in-game currency. The client will handle the UI interaction, but the server needs to process the purchase securely.

The player clicks the "Buy Upgrade" button in the UI. A Remote Event is fired from the client to the server requesting to buy the upgrade. The server checks if the player has enough money. If valid, the server subtracts the money and upgrades the Tycoon & notifies the client about the successful purchase.

7.3 Understanding Remote Functions

Remote Functions return a response. They are used when the client needs to request data from the server and get a return value.

- InvokeServer() – A client calls this function and waits for a response from the server.
- InvokeClient() – The server requests data from a specific client and waits for a return value.
- Example Use Case: Checking a Player's Inventory
- Imagine a game with a trading system, where a player wants to check their inventory before making a trade. The inventory data is stored securely on the server.

7.4 Preventing Exploits with Remote Events & Functions

Since the client is controlled by the player, bad actors can manipulate their local scripts to cheat, in Roblox this is called Exploiting. Exploits allow players to execute any code they want from their client. If you don't validate remotes with sanity checks properly, a player could:

- Give themselves infinite currency by firing a "Give Money" event.
- Unlock restricted areas or power-ups without earning them.
- Fake interactions with game objects, like opening doors they don't own.

Best Practices for Security:

- Never trust the client – Always verify data on the server before applying changes.
- Use sanity checks – If a player buys an item, check they have enough currency before processing.
- Limit Remote Event spam – Add cooldowns/debounce to prevent flooding the server with requests.
- Keep important game logic server-side – If an action affects multiple players, handle it on the server instead of relying on the client.

7.5 Data Persistence (Loading and Saving)

Without a proper data system, players would lose all their progress every time they leave and rejoin the game. DataStoreService is Roblox's built-in system for saving data to the cloud, allowing progress to be stored even after a player leaves.

Before using DataStoreService, it must be accessed using game:GetService("DataStoreService"). The data itself is stored using keys and values, where each player's UserId acts as a unique key for their save data. So in our instance, lets do `local Data = game:GetService("DataStoreService"):GetDataStore("Data")`

Usually when the player joins the server, we call Data:GetAsync(Player.UserId). If data exists, we send it to the store on the server and send it to the client. If no data exists, we get the default data.

Upon a player leaving, we use 'SetAsync' or 'UpdateAsync' to save the data. Here is an example data handling module script.

```lua
local DataHandler = {}
DataHandler.Cache = {}

local Players = game:GetService("Players")

local DataStoreService = game:GetService("DataStoreService")
local DataStore = DataStoreService:GetDataStore("Data")


function DataHandler:Load(Player)
    local Data = DataStore:GetAsync("Player"..Player.UserId) -- Fetch the data
    if not Data then -- If the data doesnt exist, get the defualt data
        Data = DataHandler:ReturnDefaultData()
    end
    DataHandler.Cache[Player] = Data -- Store in a place accessable by other scripts
    return Data
end

function DataHandler:ReturnDefaultData()
    return {
        Coins = 0;
        Xp = 0;
    }
end

function DataHandler:Save(Player)
    DataStore:SetAsync("Player"..Player.UserId, DataHandler.Cache[Player])
    DataHandler.Cache[Player] = nil
    return true
end

Players.PlayerAdded:Connect(function(Player)
    DataHandler:Load(Player)
end)

Players.PlayerRemoving:Connect(function(Player)
    DataHandler:Save(Player)
end)

return DataHandler
```

Some suggestions, but **feel free** to use **whatever you prefer**: We suggest using a reliable open-source DataStore module to handle your data requests and queue them properly. This helps prevent issues with rate limits and failed saves.

You can also use **Replica**, a server-to-client state replication module, to safely replicate data from the server to one or multiple clients. This is useful when the server owns the real data, but the client needs to display or react to it

You may use AI to help you understand how these module work and how to implement them correctly.

7.6 Exam/Practice

Now that you've made your original tycoon, there are probably issues with scalability and code. Instead of starting a new project, you will fix these by adding the networking and data persistence systems from this module to the tycoon you built in the previous modules.

Requirements:

- Multiplayer support with proper Client Server architecture
- Data Saving and Loading
- Ensure smooth gameplay experience with all mechanics functioning correctly.

If you wish to use an external editor, look into Roblox Script sync Script Sync | Documentation - Roblox Creator Hub

Deadline: TBD