

Multiplayer games

IMT3603 Game Programming · M06

Dr Simon McCallum — NTNU

Many machines, one truth

Everything hard about multiplayer follows from working out how to share state and what is the truth.

Why bother

	What the player gets
Competition	A worthy opponent, status, problems no scripted AI poses
Cooperation	Shared experience, being part of a team
Meeting	A reason to be in the same place as other people
Pressure	Someone is watching, so the stakes are higher

Humans are the most interesting source of unpredictability.



The Challenge

- Every feature exists **twice**: what it does, and how the other machines find out.
- You cannot test alone. Two builds, two machines, **every single time**.
- Bugs stop being easily reproducible. they now depend on **timing**.
- The game does not degrade gracefully. It works, or the world **desyncs**.

The four axes

Axis		
Location	Local, same screen	Remote, over a network
Timing	Synchronous, same time	Asynchronous , different time
Goals	Cooperative, shared	Competitive, divergent
Structure	Free-for-all	Team: co-op inside, competition between

These are independent. **Local co-op is multiplayer** and does not need networking at all.

Latency

Link	Round trip	Frames at 60 Hz	What that permits
Wired LAN	1–5 ms	under 1	Anything. Raw state sync is fine
Lecture-theatre Wi-Fi	~30 ms	2	Raw sync still feels fine
Within Europe	60–80 ms	4–5	Interpolate; predict anything twitchy
Norway → New Zealand	~300 ms	18	No real-time design survives this

Pick your latency budget **before** you pick your genre, not after.

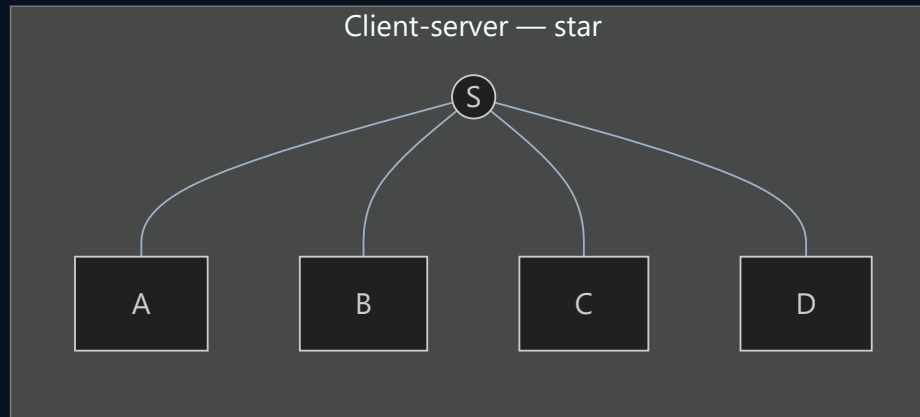
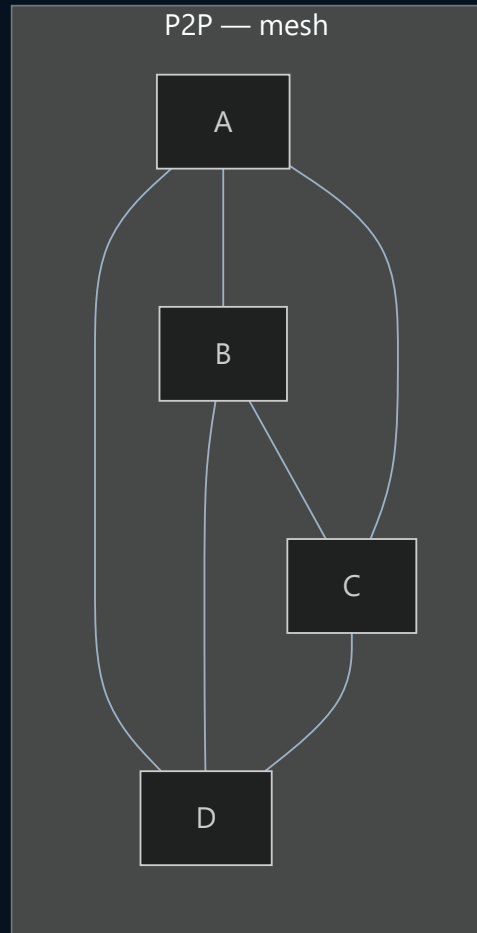
Who owns the world state

The whole subject reduces to one question: **when two machines disagree, who is right?**

- One machine is the **authority**. Its copy of the state is, by definition, the truth.
- Everyone else holds a **prediction** of that truth, and is told when they are wrong.
- Every "the state got out of sync" bug is two machines both believing they are right.

Decide this early, retrofitting authority into a game is challenging

Two topologies



P2P: $\frac{n(n-1)}{2}$ links, no owner. **Client-server:** n links, one authority.

Three flavours of server

	Who runs it	Trade
Dedicated	A machine that only serves, never plays	Costs money; nobody has a home advantage
Listen	One player's machine is also the server	Free; that player has zero latency and can cheat
Migrating	The server role hands off when the host leaves	No dead lobbies; handover can create duplication bugs

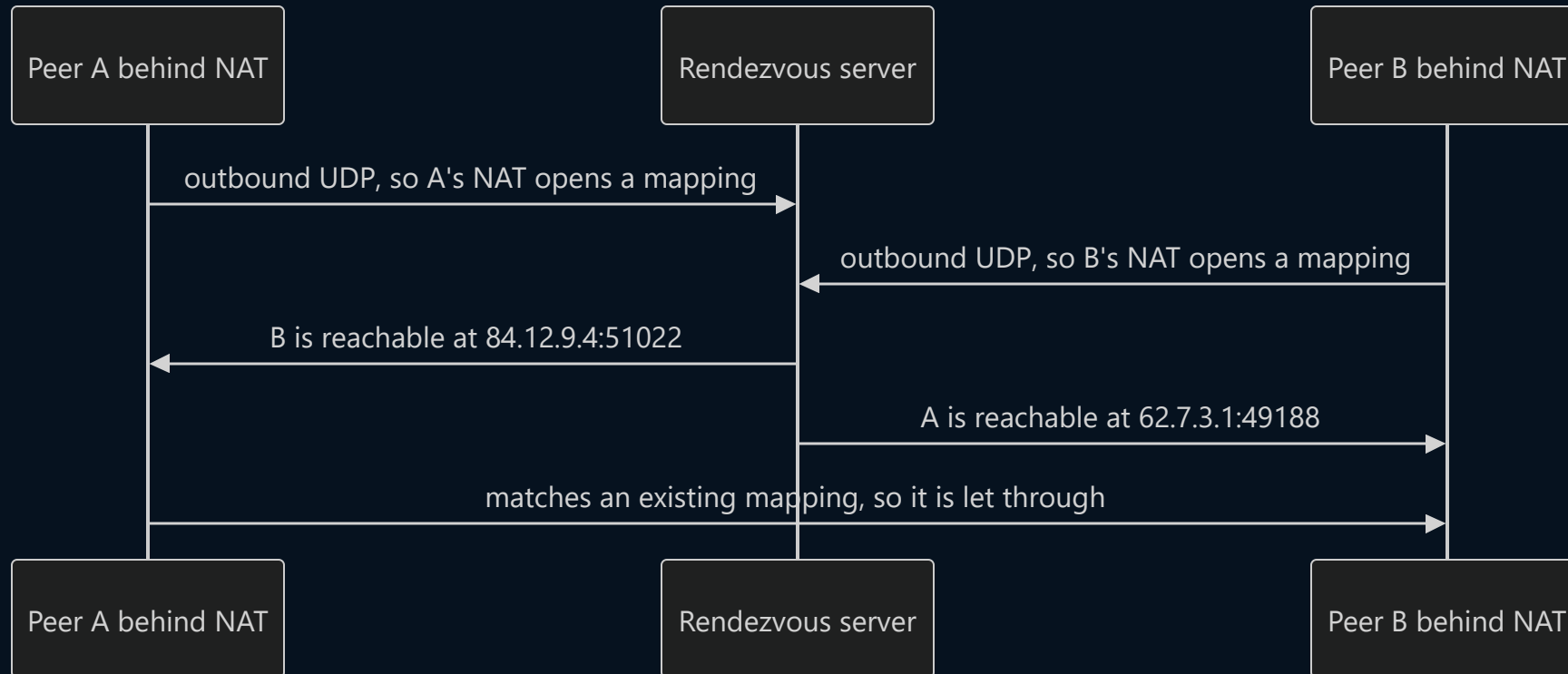
Godot makes no distinction. Server or client, same engine, same API, same project.

P2P's real problem is finding you

- **Easy:** both endpoints have public IPs. Connect directly.
- **Best:** one endpoint has a public IP and everyone connects to it. That is a server.
- **Hard:** your machine is behind a **Firewall** it has no address the outside world can reach.

Almost every home and campus machine is behind NAT. "Just connect to my IP" works in the lab and fails from home.

NAT hole punching



You still need a **server with a public IP**, just not for the game traffic.

The OSI model

OSI Model			
	Data unit	Layer	Function
Host layers	Data	7. Application	Network process to application
		6. Presentation	Data representation, encryption and decryption, convert machine dependent data to machine independent data
		5. Session	Interhost communication, managing sessions between applications
	Segments	4. Transport	Reliable delivery of packets between points on a network.
Media layers	Packet/Datagram	3. Network	Addressing, routing and (not necessarily reliable) delivery of datagrams between points on a network.
	Bit/Frame	2. Data link	A reliable direct point-to-point data connection.
	Bit	1. Physical	A (not necessarily reliable) direct point-to-point data connection.

No network protocol actually fits this model. It is vocabulary, not a build spec. Your game logic sits at the top; most games are at layer 4, on UDP.

TCP or UDP

	TCP	UDP
Delivery	Guaranteed, ordered	Best effort
On loss	Retransmit, stall	Nothing. It is gone
Latency	Higher, variable	Low, predictable
Use for	Lobby, chat, downloads	Positions, inputs

ENet is UDP with reliability you request per message. That is Godot's solution.

Godot 4.7: host or join

```
const PORT := 25666
var peer := ENetMultiplayerPeer.new()

func become_host() -> void:
    peer.create_server(PORT, MAX_PLAYERS)           # this process is now peer 1
    peer.get_host().compress(ENetConnection.COMPRESS_RANGE_CODER)
    multiplayer.multiplayer_peer = peer

func join_as_client(ip: String) -> void:
    peer.create_client(ip, PORT)                    # server hands me an id
    peer.get_host().compress(ENetConnection.COMPRESS_RANGE_CODER)
    multiplayer.multiplayer_peer = peer
```

The server is always peer id **1**. `multiplayer.get_unique_id()` is *you*.

Godot 4.7: the connection signals

```
func _ready() -> void:
    multiplayer.peer_connected.connect(_on_peer_connected)
    multiplayer.peer_disconnected.connect(_on_peer_disconnected)
    multiplayer.connected_to_server.connect(_on_connected_ok)
    multiplayer.server_disconnected.connect(_on_server_gone)

func _on_peer_connected(id: int) -> void:
    if not multiplayer.is_server(): return
    roster[id] = {"name": "", "slide": 0}           # only the server keeps the roster
```

Put this on an **autoload**, so the node path is identical on every machine.

Many tutorials are in Godot 3

Godot 3	Godot 4.7
<code>get_tree().network_peer = peer</code>	<code>multiplayer.multiplayer_peer = peer</code>
<code>network_peer_connected(id)</code>	<code>multiplayer.peer_connected</code>
<code>rpc("fire", pos)</code>	<code>fire.rpc(pos)</code>
<code>rpc_id(1, "fire", pos)</code>	<code>fire.rpc_id(1, pos)</code>
<code>remote func</code> / <code>master</code> / <code>puppet</code>	<code>@rpc("any_peer", "call_local", "reliable")</code>
<code>is_network_master()</code>	<code>is_multiplayer_authority()</code>

The string-named `rpc("function_name", ...)` form is the quick tell. It does not exist in Godot 4.

Three levers on every RPC

```
@rpc("authority", "call_local", "reliable")    # only peer 1 may call it
@rpc("any_peer", "call_remote", "reliable")    # any client may call it
@rpc("any_peer", "unreliable")                 # fire and forget: position spam
```

Lever	Question it answers
<code>authority</code> / <code>any_peer</code>	Who may call it
<code>call_local</code> / <code>call_remote</code>	Does it also run on the sender?
<code>reliable</code> / <code>unreliable</code>	Is losing this message acceptable?

Pattern 1 — client asks, server decides

```
# levels/slides/slides.gd – the lecturer presses the arrow key
Lecture.set_slide(current)          # → _request_slide.rpc_id(1, index)

# scripts/lecture_session.gd – the server rules on it
@rpc("any_peer", "reliable")
func _request_slide(index: int) -> void:
    if not multiplayer.is_server(): return
    if not _is_lecturer(multiplayer.get_remote_sender_id()):
        return          # a student asking is simply ignored
    _broadcast_slide(index)
```

"any_peer" means anyone may call it. The **body** decides whether to act.

Pattern 2 — server tells everyone

```
func _broadcast_slide(index: int) -> void:
    current_slide = index
    _set_slide.rpc(index)                # server → every peer

@rpc("authority", "call_local", "reliable")
func _set_slide(index: int) -> void:
    current_slide = index
    slide_changed.emit(index)           # every listening scene reacts
```

"authority" — a client calling this is **refused by the engine**, before your code runs.

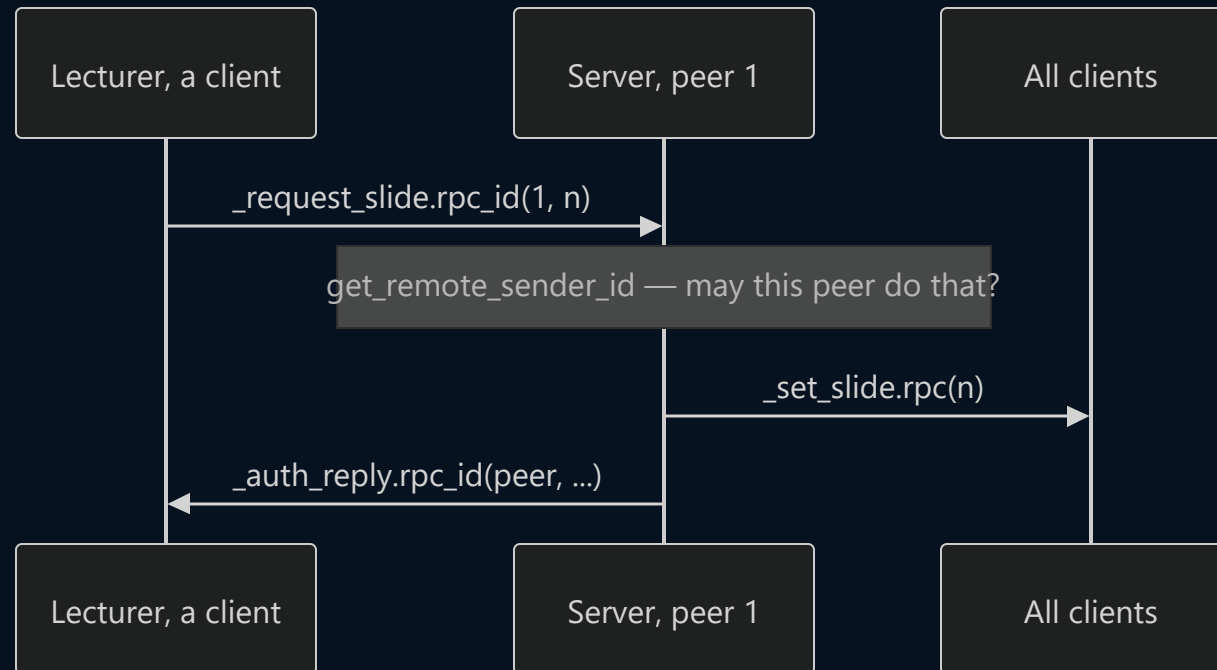
Pattern 3 — server tells one peer

```
# scripts/lecture_session.gd – a peer has just identified itself
_auth_reply.rpc_id(sender, granted, message, role)
_set_pack.rpc_id(sender, current_pack, current_slide)
_set_activity.rpc_id(sender, current_activity, current_slide)
```

Same function, three audiences:

<code>_f.rpc(...)</code>	<code>_f.rpc_id(1, ...)</code>	<code>_f.rpc_id(peer, ...)</code>
Everyone	The server	One specific client

The three patterns together



Authority decides **who may speak**; `rpc` / `rpc_id` decides **who hears it**.

Authority per node, not per game

```
# prefabs/player/player.gd
func _enter_tree() -> void:
    set_multiplayer_authority(name.to_int(), true)    # "Player7" → peer 7 owns it

func _physics_process(delta: float) -> void:
    if multiplayer.has_multiplayer_peer() and not is_multiplayer_authority():
        return                                         # remote copies read no input
    velocity.x = move_toward(velocity.x, Input.get_axis("left", "right") * SPEED, ACCELERATION)
    move_and_slide()
```

You cannot drive my character, because you do not hold authority over that node.

The two nodes that do the work

Node	Replicates	Called on
MultiplayerSpawner	Node creation and deletion	The server only
MultiplayerSynchronizer	A chosen set of properties, each tick	Automatic

```
# levels/main/main.gd
player_spawner.spawn_function = spawn_player
if multiplayer.is_server():
    player_spawner.spawn(player_id)    # one call; every peer gets the node
```

You write `move_and_slide()`. The engine writes the packets.

Replicate this, recompute that

Send it	Derive it locally
Player input, or authoritative position and velocity	Animation state, from velocity
Discrete events: spawn, death, score, door opened	Particles, decals, screen shake
Server-owned numbers: health, ammo, inventory	Audio triggers, camera, UI
The RNG seed	Every result that seed produces

Optimizing bandwidth by not using it in the first place.

Unreal 5.8: the same ideas, different nouns

Godot 4.7	Unreal 5.8
<code>set_multiplayer_authority()</code>	Server owns the actor; clients own their <code>PlayerController</code>
<code>MultiplayerSynchronizer</code>	<code>bReplicates</code> , <code>bReplicateMovement</code> , <code>DOREPLIFETIME</code>
<code>@rpc("any_peer")</code>	<code>UFUNCTION(Server, Reliable, WithValidation)</code>
<code>@rpc("authority") + rpc()</code>	<code>UFUNCTION(NetMulticast, ...)</code>
<code>rpc_id(peer, ...)</code>	<code>UFUNCTION(Client, ...)</code> — to the owning client

`bReplicateMovement` gives you Location, Rotation and Velocity automatically.

The scope ladder



	Cost	Fits a semester?
1 Async turn-based	Days	Yes, comfortably
2 Lockstep, deterministic	2–3 weeks	Yes, if you start early
3 Authority, spawning, sync	6+ weeks of one person	Only if it <i>is</i> the project
4 Prediction, reconciliation	A semester on its own	No

Recap and reading

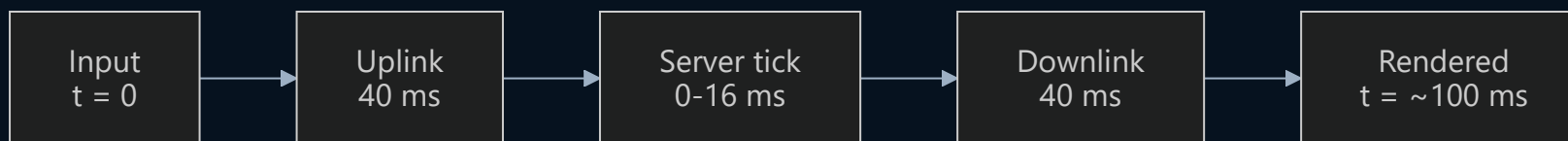
- One machine is the **authority**. Everyone else holds a guess and gets corrected.
- Client-server unless you have a reason; **the server is peer 1** in Godot.
- Three RPC levers: **who may call, who hears it, may it be lost**.
- Replicate input and events; **recompute everything you can derive**.

Godot high-level multiplayer · `MultiplayerSynchronizer` · Unreal networking · Gambetta, *Fast-Paced Multiplayer*

EXT-A · Making it feel right

Nobody can remove the latency. The job is to make it feel ok.

You are always showing the past



By the time you see the result of your own keypress, it is **six frames old**.

- Do nothing and the game feels like driving with a delayed steering wheel.
- The three fixes: **interpolate**, **predict**, **extrapolate**.

Interpolation: render deliberately late

- Buffer incoming snapshots and render the world ~100 ms **behind** the newest one.
- Every frame you draw sits **between two states you were actually told about**.
- Other players move smoothly, and a dropped packet is invisible — the buffer covers it.

Cost: everything you see is stale. This is why "I shot them and nothing happened" exists.

Prediction: act now, ask later

```
func _physics_process(delta: float) -> void:
    var input := _sample_input()
    _apply(input, delta)                # move immediately, do not wait
    pending.append({"tick": _tick, "input": input})
    _send_input.rpc_id(1, _tick, input) # the authority will confirm later
    _tick += 1
```

Your own character responds in **zero** milliseconds. Everyone else's does not.

A sketch, not repo code: the companion project stops one rung below this, deliberately.

Reconciliation: the authority always wins

```
@rpc("authority", "unreliable")
func _server_state(ack_tick: int, truth: Vector2) -> void:
    pending = pending.filter(func(p): return p["tick"] > ack_tick)
    if position.distance_to(truth) < TOLERANCE:
        return # we guessed right, change nothing
    position = truth # rebase onto the server's state
    for p in pending:
        _apply(p["input"], PHYSICS_TICK) # replay the inputs it has not seen yet
```

Discard confirmed inputs, snap to truth, **replay the rest**. Usually nothing moves.

Dead reckoning: guessing the future

- For **remote** entities: last known position, plus velocity $\times$ time since the update.
- Send an update only when the guess drifts past a **threshold**, not every tick.
- Cheap, and completely wrong the instant something changes direction.

Blend back over 100–200 ms. A remote player who teleports is worse than one who is late.

Lag compensation: whose reality wins

The shooter's dilemma: by the time "I shot them" arrives, the target has **moved**.

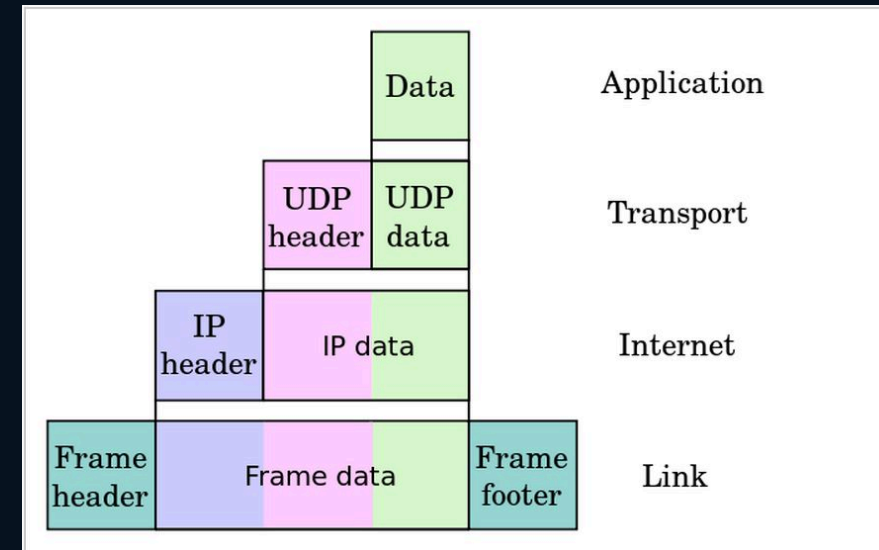
- The server keeps a short **history** of every entity's recent positions.
- It **rewinds** the world to where things were when the shooter fired.
- It tests the shot against **that** snapshot, not the present.

Feels fair to the shooter, produces "shot from behind cover" for the target. Every competitive shooter picks a side, and none of them please everybody.

Bandwidth: the headers are bigger than the payload

	Bytes
IP header	20
UDP header	8
ENet framing	~12
A position, 2 × float32	8
A position, quantised to 2 × int16	4

30 players × 30 recipients × 20 Hz × 48 B ≈ 864 kB/s out of one server.



Why `_physics_process` matters

	<code>_process</code>	<code>_physics_process</code>
Rate	Whatever the GPU manages	Fixed, 60 Hz by default
<code>delta</code>	Varies every frame	Constant
Replayable?	No	Yes

Prediction, reconciliation and lockstep all require **replaying inputs** and getting the same answer twice. A variable timestep cannot give you that.

EXT-B · Communities, griefing and cheating

Multiplayer is an engineering problem but it can open up human problems.

What makes a community

Feature	In a game
Membership	A clear sense of who is in and who is not
Influence	Members can visibly change the shared thing
Needs	It gives them something they cannot get alone
Emotional connection	A shared history worth referring back to

Why an engineer cares: community drives **retention** and word of mouth, and both are measured in hours of play.

Building one, deliberately

Lever	Concretely
Let people talk	Chat, voice, and help finding people worth talking to
Support friendship	Ice-breakers, friending, guilds, clubhouses
Shared enemy	Conflict at the core: a raid boss unites a server
Geography	A persistent place people return to and recognise
Identity	Uniforms, symbols, titles: express personality <i>and</i> conformity
Obligation	Raid times, gifting, events: humans who are counting on you

These are **mechanics** you design for.

Griefing

Players whose fun is ruining yours. Two strategic responses:

- **Police it** — reporting, trust scores, bans. Ongoing human cost, forever.
- **Design it out** — remove the mechanic that makes griefing possible.

Mechanics that hand out the opportunity: PvP, stealing, player-to-player trading, one-sided language filters, and whether players can **push past** each other or walk through.

Authority is anti-griefing, for free

What can a malicious client do in this lecture multiplayer?

- **Jump everyone's slide?** No. `_set_slide` is `@rpc("authority")` — only peer 1 may call it.
- **Claim to be the lecturer?** Only by answering a fresh random challenge correctly.
- **Spawn fake players?** No. `player_spawner.spawn()` runs only `if multiplayer.is_server()`.
- **Drive your character?** No. It has no authority over that node.

The griefing defence and the cheating defence are the **same mechanism**.

Game theory, briefly

- **Zero-sum** — my win is exactly your loss.
- **Nash equilibrium** — nobody gains by changing strategy alone.
- **Dominant strategy** — one option is best whatever anyone else does.
- **Bounded rationality** — real players decide with partial information, in a hurry.

A dominant strategy is a **bug in balancing**.

A payoff matrix – Nash equilibria in bold

Player 1 strategy	Player 2 strategy		
	Option A	Option B	Option C
Option A	0, 0	25, 40	5, 10
Option B	40, 25	0, 0	5, 15
Option C	10, 5	15, 5	10, 10

No dominant strategy is the goal



Intransitive: there is no best move, so the equilibrium is to play **unpredictably**.

Every counter-system in every strategy game — unit triangles, weapon triangles, elemental weaknesses — is this diagram with better art.

Never trust the client

Cheat	What it exploits	Defence
Wallhack, maphack	You sent data the player cannot see	Send only what is relevant to that client
Value editing	The client's number is the truth	Server authority: the client's number is a <i>display</i>
Score forgery	Results computed client-side	Compute server-side; sign or hash what you must accept
Lag switch	Lag compensation rewinds for you	Cap the rewind window; monitor per-player latency

Relevancy culling is a **bandwidth** optimisation that is also an **anti-cheat**.

The wire is public

- **Packet sniffing** — Wireshark reads your traffic on any shared network.
- **Spoofing** — a forged message claiming to be someone else.
- **Strategic packet dropping** — deliberately losing the packets that hurt you.
- **ENet is not encrypted.** A password sent over it is a password shared with the room.

```
var nonce := Crypto.new().generate_random_bytes(32)    # fresh, single use
_auth_challenge.rpc_id(id, nonce)                      # prove it; do not send it
```

What "authoritative" is good for

Solved	Still open
Value editing, teleporting, item duplication	Aimbots — the input itself is legitimate
Forged scores and achievements	Wallhacks, if you sent the data anyway
Illegal actions: firing an empty gun	Collusion between real players
Griefing that needs a forbidden action	Anything a very good player could also do

Server authority makes cheating a **detection** problem instead of a **prevention** one. That does not solve all problems

