

High-level multiplayer

High-level vs low-level API

The following explains the differences of high- and low-level networking in Godot as well as some fundamentals. If you want to jump in head-first and add networking to your first nodes, skip to [Initializing the network](#) below. But make sure to read the rest later on!

Godot always supported standard low-level networking via [UDP](#), [TCP](#) and some higher-level protocols such as [HTTP](#) and [SSL](#). These protocols are flexible and can be used for almost anything. However, using them to synchronize game state manually can be a large amount of work. Sometimes that work can't be avoided or is worth it, for example when working with a custom server implementation on the backend. But in most cases, it's worthwhile to consider Godot's high-level networking API, which sacrifices some of the fine-grained control of low-level networking for greater ease of use.

This is due to the inherent limitations of the low-level protocols:

- TCP ensures packets will always arrive reliably and in order, but latency is generally higher due to error correction. It's also quite a complex protocol because it understands what a "connection" is, and optimizes for goals that often don't suit applications like multiplayer games. Packets are buffered to be sent in larger batches, trading less per-packet overhead for higher latency. This can be useful for things like HTTP, but generally not for games. Some of this can be configured and disabled (e.g. by disabling "Nagle's algorithm" for the TCP connection).
- UDP is a simpler protocol, which only sends packets (and has no concept of a "connection"). No error correction makes it pretty quick (low latency), but packets may be lost along the way or received in the wrong order. Added to that, the MTU (maximum packet size) for UDP is generally low (only a few hundred bytes), so transmitting larger packets means splitting them, reorganizing them and retrying if a part fails.

In general, TCP can be thought of as reliable, ordered, and slow; UDP and fast. Because of the large difference in performance, it often makes parts of TCP wanted for games (optional reliability and packet order), while avoiding the

unwanted parts (congestion/traffic control features, Nagle's algorithm, etc). Due to this, most game engines come with such an implementation, and Godot is no exception.

In summary, you can use the low-level networking API for maximum control and implement everything on top of bare network protocols or use the high-level API based on [SceneTree](#) that does most of the heavy lifting behind the scenes in a generally optimized way.

! Note

Most of Godot's supported platforms offer all or most of the mentioned high- and low-level networking features. As networking is always largely hardware and operating system dependent, however, some features may change or not be available on some target platforms. Most notably, the HTML5 platform currently offers WebSockets and WebRTC support but lacks some of the higher-level features, as well as raw access to low-level protocols like TCP and UDP.

! Note

More about TCP/IP, UDP, and networking: https://gafferongames.com/post/udp_vs_tcp/

Gaffer On Games has a lot of useful articles about networking in Games ([here](#)), including the comprehensive [introduction to networking models in games](#) .

! Warning

Adding networking to your game comes with some responsibility. It can make your application vulnerable if done wrong and may lead to cheats or exploits. It may even allow an attacker to compromise the machines your application runs on and use your servers to send spam, attack others or steal your users' data if they play your game.

This is always the case when networking is involved and has nothing to do with Godot. You can of course experiment, but when you release a networked application, always take care of any possible security concerns.

Mid-level abstraction

Before going into how we would like to synchronize a game across the network, it can be helpful to understand how the base network API for synchronization

 [en](#) [stable](#) ▼

Godot uses a mid-level object [MultiplayerPeer](#) . This object is not meant to be created directly, but is designed so that several C++ implementations can provide it.

This object extends from [PacketPeer](#) , so it inherits all the useful methods for serializing, sending and receiving data. On top of that, it adds methods to set a peer, transfer mode, etc. It also includes signals that will let you know when peers connect or disconnect.

This class interface can abstract most types of network layers, topologies and libraries. By default, Godot provides an implementation based on ENet ([ENetMultiplayerPeer](#)), one based on WebRTC ([WebRTCMultiplayerPeer](#)), and one based on WebSocket ([WebSocketPeer](#)), but this could be used to implement mobile APIs (for ad hoc WiFi, Bluetooth) or custom device/console-specific networking APIs.

For most common cases, using this object directly is discouraged, as Godot provides even higher level networking facilities. This object is still made available in case a game has specific needs for a lower-level API.

Hosting considerations

When hosting a server, clients on your LAN can connect using the internal IP address which is usually of the form `192.168.*.*` . This internal IP address is **not** reachable by non-LAN/Internet clients.

On Windows, you can find your internal IP address by opening a command prompt and entering `ipconfig` . On macOS, open a Terminal and enter `ifconfig` . On Linux, open a terminal and enter `ip addr` .

If you're hosting a server on your own machine and want non-LAN clients to connect to it, you'll probably have to *forward* the server port on your router. This is required to make your server reachable from the Internet since most residential connections use a [NAT](#) . Godot's high-level multiplayer API only uses UDP, so you must forward the port in UDP, not just TCP.

After forwarding a UDP port and making sure your server uses that port, you can use [this website](#) to find your public IP address. Then give this public IP address to any Internet clients that wish to connect to your server.

Godot's high-level multiplayer API uses a modified version of ENet which allows for full IPv6 support.

Initializing the network

High-level networking in Godot is managed by the [SceneTree](#) .

Each node has a `multiplayer` property, which is a reference to the `MultiplayerAPI` instance configured for it by the scene tree. Initially, every node is configured with the same default `MultiplayerAPI` object.

It is possible to create a new `MultiplayerAPI` object and assign it to a `NodePath` in the scene tree, which will override `multiplayer` for the node at that path and all of its descendants. This allows sibling nodes to be configured with different peers, which makes it possible to run a server and a client simultaneously in one instance of Godot.

GDScript **C#**

```
# By default, these expressions are interchangeable.
multiplayer # Get the MultiplayerAPI object configured for this node.
get_tree().get_multiplayer() # Get the default MultiplayerAPI object.
```

To initialize networking, a `MultiplayerPeer` object must be created, initialized as a server or client, and passed to the `MultiplayerAPI`.

GDScript **C#**

```
# Create client.
var peer = ENetMultiplayerPeer.new()
peer.create_client(IP_ADDRESS, PORT)
multiplayer.multiplayer_peer = peer

# Create server.
var peer = ENetMultiplayerPeer.new()
peer.create_server(PORT, MAX_CLIENTS)
multiplayer.multiplayer_peer = peer
```

To terminate networking:

GDScript **C#**

```
multiplayer.multiplayer_peer = OfflineMultiplayerPeer.n
```



en



stable



⚠ Warning

When exporting to Android, make sure to enable the **INTERNET** permission in the Android export preset before exporting the project or using one-click deploy. Otherwise, network communication of any kind will be blocked by Android.

Managing connections

Every peer is assigned a unique ID. The server's ID is always 1, and clients are assigned a random positive integer.

Responding to connections or disconnections is possible by connecting to **MultilayerAPI** 's signals:

- **peer_connected(id: int)** This signal is emitted with the newly connected peer's ID on each other peer, and on the new peer multiple times, once with each other peer's ID.
- **peer_disconnected(id: int)** This signal is emitted on every remaining peer when one disconnects.

The rest are only emitted on clients:

- **connected_to_server()**
- **connection_failed()**
- **server_disconnected()**

To get the unique ID of the associated peer:

GDScript **C#**

```
multiplayer.get_unique_id()
```

To check whether the peer is server or client:

GDScript **C#**

```
multiplayer.is_server()
```

Remote procedure calls

Remote procedure calls, or RPCs, are functions that can be called on other peers. To create one, use the `@rpc` annotation before a function definition. To call an RPC, use `Callable` 's method `rpc()` to call in every peer, or `rpc_id()` to call in a specific peer.

GDScript

C#

```
func _ready():
    if multiplayer.is_server():
        print_once_per_client.rpc()

@rpc
func print_once_per_client():
    print("I will be printed to the console once per each connected
client.")
```

RPCs will not serialize objects or callables.





For a remote call to be successful, the sending and receiving node need to have the same `NodePath`, which means they must have the same name. When using `add_child()` for nodes which are expected to use RPCs, set the argument `force_readable_name` to `true`.

⚠ Warning

If a function is annotated with `@rpc` on the client script (resp. server script), then this function must also be declared on the server script (resp. client script). Both RPCs must have the same signature which is evaluated with a checksum of **all RPCs**. All RPCs in a script are checked at once, and all RPCs must be declared on both the client scripts and the server scripts, **even functions that are currently not in use**.

The signature of the RPC includes the `@rpc()` declaration, the function, return type, **and** the `NodePath`. If an RPC resides in a script attached to `/root/Main/Node1`, then it must reside in precisely the same path and node on both the client script and the server script.

Function arguments are not checked for matching between the server and client code

(example: `func sendstuff():` and `func sendstuff(arg1, arg2`   `en`  `stable`  matching).

If these conditions are not met (if all RPCs do not pass signature matching), the script may print an error or cause unwanted behavior. The error message may be unrelated to the RPC function you are currently building and testing.

See further explanation and troubleshooting on [this post](#) .

The annotation can take a number of arguments, which have default values. `@rpc` is equivalent to:

GScript

C#

```
@rpc("authority", "call_remote", "unreliable", 0)
```

The parameters and their functions are as follows:

mode :

- **"authority"** : Only the multiplayer authority can call remotely. The authority is the server by default, but can be changed per-node using [Node.set_multiplayer_authority](#) .
- **"any_peer"** : Clients are allowed to call remotely. Useful for transferring user input.

sync :

- **"call_remote"** : The function will not be called on the local peer.
- **"call_local"** : The function can be called on the local peer. Useful when the server is also a player.

transfer_mode :

- **"unreliable"** Packets are not acknowledged, can be lost, and can arrive at any order.
- **"unreliable_ordered"** Packets are received in the order they were sent in. This is achieved by ignoring packets that arrive later if another that was sent after them has already been received. Can cause packet loss if used incorrectly.
- **"reliable"** Resend attempts are sent until packets are acknowledged. Preserved. Has a significant performance penalty.



en



stable



`transfer_channel` is the channel index.

The first 3 can be passed in any order, but `transfer_channel` must always be last.

The function `multiplayer.get_remote_sender_id()` can be used to get the unique id of an rpc sender, when used within the function called by rpc.

GDScript C#

```
func _on_some_input(): # Connected to some input.
    transfer_some_input.rpc_id(1) # Send the input only to the server.

# Call local is required if the server is also a player.
@rpc("any_peer", "call_local", "reliable")
func transfer_some_input():
    # The server knows who sent the input.
    var sender_id = multiplayer.get_remote_sender_id()
    # Process the input and affect game logic.
```

Channels

Modern networking protocols support channels, which are separate connections within the connection. This allows for multiple streams of packets that do not interfere with each other.

For example, game chat related messages and some of the core gameplay messages should all be sent reliably, but a gameplay message should not wait for a chat message to be acknowledged. This can be achieved by using different channels.

Channels are also useful when used with the unreliable ordered transfer mode. Sending packets of variable size with this transfer mode can cause packet loss, since packets which are slower to arrive are ignored. Separating them into multiple streams of homogeneous packets by using channels allows ordered transfer with little packet loss, and without the latency penalty caused by reliable mode.

The default channel with index 0 is actually three different channels - one for each transfer mode.

Example lobby implementation

This is an example lobby that can handle peers joining and leaving, notify UI scenes through signals, and start the game after all clients have loaded the game scene.

GDScript

C#



en



stable



extends Node

Autoload named Lobby

These signals can be connected to by a UI lobby scene or the game scene.

signal player_connected(peer_id, player_info)

signal player_disconnected(peer_id)

signal server_disconnected

const PORT = 7000

const DEFAULT_SERVER_IP = "127.0.0.1" *# IPv4 localhost*

const MAX_CONNECTIONS = 20

This will contain player info for every player,

with the keys being each player's unique IDs.

var players = {}

This is the local player info. This should be modified locally

before the connection is made. It will be passed to every other peer.

For example, the value of "name" can be set to something the player

entered in a UI scene.

var player_info = {"name": "Name"}

var players_loaded = 0

func _ready():

 multiplayer.peer_connected.connect(_on_player_connected)

 multiplayer.peer_disconnected.connect(_on_player_disconnected)

 multiplayer.connected_to_server.connect(_on_connected_ok)

 multiplayer.connection_failed.connect(_on_connected_fail)

 multiplayer.server_disconnected.connect(_on_server_disconnected)

func join_game(address = ""):

 if address.is_empty():

 address = DEFAULT_SERVER_IP

 var peer = ENetMultiplayerPeer.new()

 var error = peer.create_client(address, PORT)

 if error:

 return error

 multiplayer.multiplayer_peer = peer

func create_game():

 var peer = ENetMultiplayerPeer.new()

```

var error = peer.create_server(PORT, MAX_CONNECTIONS)
if error:
    return error
multiplayer.multiplayer_peer = peer

players[1] = player_info
player_connected.emit(1, player_info)

func remove_multiplayer_peer():
    multiplayer.multiplayer_peer = OfflineMultiplayerPeer.new()
    players.clear()

# When the server decides to start the game from a UI scene,
# do Lobby.load_game.rpc(filepath)
@rpc("call_local", "reliable")
func load_game(game_scene_path):
    get_tree().change_scene_to_file(game_scene_path)

# Every peer will call this when they have loaded the game scene.
@rpc("any_peer", "call_local", "reliable")
func player_loaded():
    if multiplayer.is_server():
        players_loaded += 1
        if players_loaded == players.size():
            $/root/Game.start_game()
            players_loaded = 0

# When a peer connects, send them my player info.
# This allows transfer of all desired data for each player, not only the
unique ID.
func _on_player_connected(id):
    _register_player.rpc_id(id, player_info)

@rpc("any_peer", "reliable")
func _register_player(new_player_info):
    var new_player_id = multiplayer.get_remote_sender_id()
    players[new_player_id] = new_player_info
    player_connected.emit(new_player_id, new_player_info)

func _on_player_disconnected(id):
    players.erase(id)
    player_disconnected.emit(id)

```

```
func _on_connected_ok():
    var peer_id = multiplayer.get_unique_id()
    players[peer_id] = player_info
    player_connected.emit(peer_id, player_info)
```

```
func _on_connected_fail():
    remove_multiplayer_peer()
```

```
func _on_server_disconnected():
    remove_multiplayer_peer()
    players.clear()
    server_disconnected.emit()
```

The game scene's root node should be named Game. In the script attached to it:

GDScript C#

```
extends Node3D # Or Node2D.
```

```
func _ready():
    # Preconfigure game.
```

```
    Lobby.player_loaded.rpc_id(1) # Tell the server that this peer has loaded.
```

```
# Called only on the server.
```

```
func start_game():
    # All peers are ready to receive RPCs in this scene.
```

Exporting for dedicated servers

Once you've made a multiplayer game, you may want to export it to server with no GPU available. See [Exporting for dedicated servers](#) for more information.

Note

The code samples on this page aren't designed to run on a dedicated server. You'll have to modify them so the server isn't considered to be a player. You'll also have to modify the game starting mechanism so that the first player who joins can start the game.

[↩ Previous](#)[Next ➞](#)

User-contributed notes

Please read the [User-contributed notes policy](#) before submitting a comment.

Loading comments...