This document is best viewed with this option in the menubar: *View → Text Width → Wide*

Unity Netcode for Entities 101

In this document, we'll first explain the fundamental concepts and workflows of Netcode for Entities, and then we'll walk through a very simple sample project which demonstrates the related parts of the API.

See also the video intro to netcode and walkthrough of the Kickball sample.

The topics covered include (loosely in this order):

- The roles of the server and clients
- Player input
- Ghost entities
- Client-side prediction
- RPCs (Remote Procedure Calls)
- Netcode bootstrapping and connections

In the sample project (complete code here [link]):

- A player character is spawned when a client connects to the server
- Players can control their characters' movements
- Players can spawn balls
- Players can kick balls

The role of the server

"Netcode" is about synchronizing multiplayer game state between players' machines. Other networked multiplayer concerns, such as matchmaking, player authentication, or player data persistence, are outside the scope of netcode.

In Netcode for Entities, each player runs a client that connects to an authoritative server, which runs the actual game simulation:

- the server runs the full, authoritative simulation of the game
- the server does not render anything
- the server receives player input from the clients
- the server sends updates of the game state to the clients

> **Note**: For a published game, servers are usually run in data centers on headless machines (machines which aren't directly attached to displays or input devices), though games may also support "player hosting", where a game session's server is run on the machine of a participating player. Also, during development, servers are often run directly on a developer's machine for testing purposes.

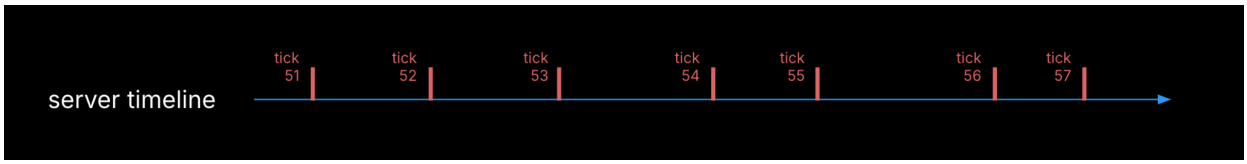
In terms of game simulation logic, really the only difference from a standalone, non-networked game is that the game logic must always run at a fixed update rate. Each update of the game simulation on the server is called a **tick**, and each tick is known by an integer number, starting from 1 for the first tick.

> **Note:** Allowing variable time intervals between game simulation updates wouldn't be impossible, but it would make Netcode's data synchronization much more complicated to no real benefit.

By default, ticks are run at a rate of 60 per second, but this rate can be set in the API. The higher the tick rate, generally the higher the accuracy and responsiveness of the game action at the cost of more CPU resources (and therefore higher server bills). Games with fast, twitch-sensitive action, such as a first-person shooter, typically should run at a rate of 20 to 30 ticks per second minimum, though some shooter games run at a rate of 120 ticks per second or even higher. Games with slow action might do fine running at a rate of 10 ticks per second or even lower.

> **Note:** On the server, the ticks do not necessarily correspond to frames of the Unity gameloop. To regulate the tick rate on the server, Netcode will run 0 or more ticks in a frame of the Unity gameloop, much like the fixed update logic of physics.

So a portion of the timeline on a server might look something like this:



In the timeline above, the spacing between ticks is uneven to reflect the fact that the actual time interval between ticks may vary. This can happen because some ticks might take more than the allotted time to process (`1 / ticks_per_second` seconds). *Logically*, however, the interval between ticks is supposed to be uniform, so the delta time value used in your logic should always be `1 / ticks_per_second`.

If a player's input for a given tick does not reach the server in time for whatever reason, such as a network interruption or the player's machine crashing, the server will not wait: the server simply proceeds using the last received input from that client. So for example, if the player was holding the gamepad stick to move forward in the input last received from their client, then the

server will assume the player is still holding the stick forward until a new input is received. This, of course, is not ideal for the affected player, but it's better than slowing down or pausing the whole game for everyone.

> **Note:** When a tick finishes processing with time to spare, Netcode will wait out the remaining time before starting the next tick. When a "slow tick" occurs (a tick that exceeds its allotted time to process), time in the game effectively slows down. Some occasional slow downs aren't usually a problem, but players may start to notice if they happen too frequently, and slow ticks can lead to a "death spiral" (where the server can't process the game logic fast enough but needs to do more and more work to catch up). To counteract this, Netcode has a feature enabled by default called "tick batching", where the server may execute multiple ticks in one pass by using a multiplied delta time value. For example, executing a tick with delta time multiplied by 3 effectively advances the game state by 3 ticks. Of course, this sacrifices accuracy of the simulation, but it is generally better than noticeable slow downs and death spirals.
>
> Also note that the possibility of slow ticks makes tick numbers and delta time values an inaccurate way to track real world time. When your game needs to accurately track real world time, you should use timestamps from the actual system clock.

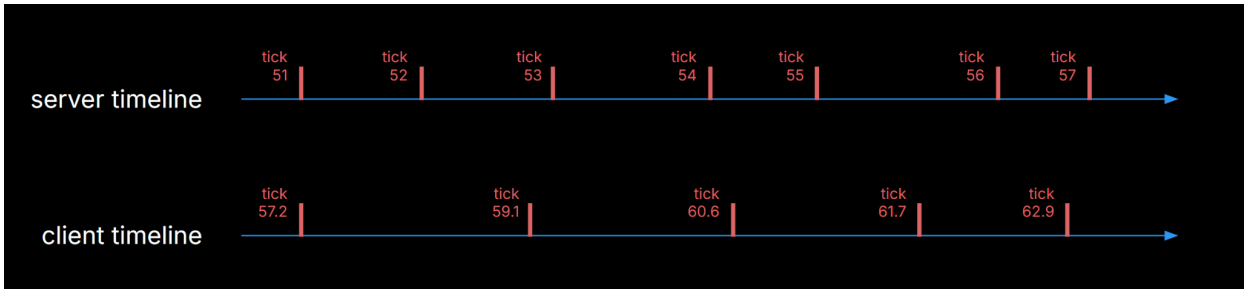
The role of a client

Each player's machine runs a client that connects to the server. Usually the server is on a separate machine, but in development or in a "player hosting" scenario, the client and server may be running on the same machine.

In Netcode for Entities:

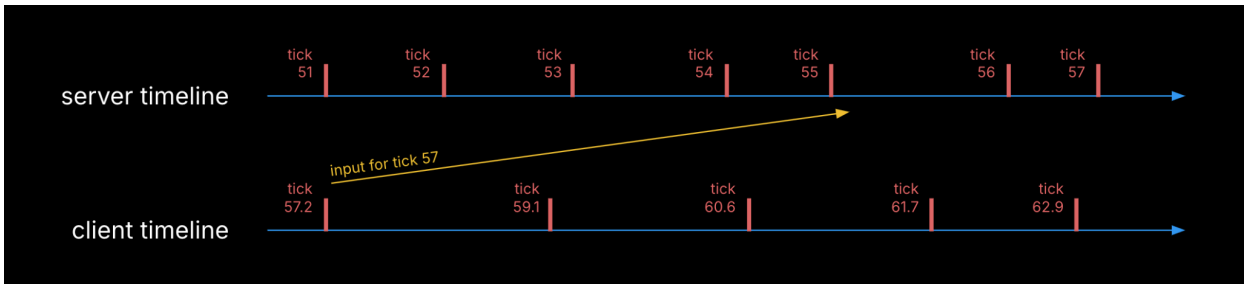
- a client renders
- a client sends player input to the server
- a client receives game state from the server
- a client may apply interpolation or "prediction" on some game state before rendering

The framerate on the client is independent from the ticks of the server, so the intervals between frames don't have to be uniform, and the framerate can be completely different from the server's tick rate. For example, the timelines of the server and client might line up like so:



Two things to note above:

- A frame on a client may correspond to a “partial tick” (a fractional tick number).
- The client actually runs several ticks ahead of the server, as if the client is living in the future! Though unintuitive, this arrangement is necessary so that player input sent from the client can reach the server in time. For example, when the client sends player input for tick 57 to the server, thanks to this lead time, the input can reach the server before the server itself processes tick 57:



How many ticks each client needs to run ahead of the server depends upon two factors:

- the server tick rate of the game
- how much time it takes for input from the client to reach the server (which differs for each client and may fluctuate frame-to-frame)

To ensure that a client is running the correct number of ticks ahead of the server, Netcode for Entities will automatically adjust the client’s time sync by temporarily increasing or decreasing the tick increment of each frame by a subtle amount. Network drift and interruptions will trigger these readjustments.

Player input command streams

Player input is represented by a dynamic buffer component that implements **ICommandData**:

- Each element of the buffer represents the player's input for a single tick. The last element of the buffer is the latest input.
- The latest inputs in the ICommandData buffer are sent by Netcode to the server every tick.
- On the server, the received inputs are stored in ICommandData buffers corresponding to each client.
- The ICommandData buffer is usually placed on the entity representing the player character (but this is not strictly required).
- In each frame of the client, your code should append the player's latest inputs to the ICommandData buffer. For example, in a frame on the client that renders tick 93.4, your code should append input data to the ICommandData buffer for tick 93.

  Note: Because multiple frames on the client may correspond to the same whole tick number, you may end up appending input data for the same tick multiple times. If, say, the next frame corresponds to tick 93.8, you should again overwrite the input data in the ICommandData for tick 93.

  Note: When sending the input commands, Netcode also includes acknowledgements and time synchronization data, so each client always requires an ICommandData buffer even if the player has no need for input. If your client neglects to append new input to the buffer in a frame, this other data will be sent regardless.

  Note: You can have more than one ICommandData buffer on a client, but only one at a time can be actively sending its commands. To designate which ICommandData is active, you set the CommandTarget of the connection entity. Dynamically switching ICommandData buffers can be useful in a game where the player may change their input actions. For example, if the player character switches from running on foot to driving a vehicle, then you may want to switch the active ICommandData buffer.

Netcode for Entities also provides IInputComponentData, which is a more convenient way to handle input that manages the ICommandData buffer for you. (We'll demonstrate how to use IInputComponentData in the sample below.)

Ghost entities

In Netcode for Entities, a "ghost" is an entity on the server whose state (in whole or in part) is synched to a corresponding mirror ghost entity on each of the clients. Ghosts are the primary means to sync game state in Netcode for Entities.

Each ghost entity is an instance of a "ghost type":

- Ghost types are defined by prefabs baked in the build (consequently, ghost types can only be defined at build time). To make a prefab that defines a ghost type, simply add the GhostAuthoringComponent to the root GameObject of the prefab, and then make sure the prefab gets baked in at least one subscene.
- Because we don't necessarily want all of the data of a ghost entity synced from the server to the clients, only the component fields marked with the GhostField attribute will be included in the ghost synchronization.
    - Transforms are treated as a special case: properties in the GhostAuthoringComponent let you specify what parts of the transform (if any) you want included in the ghost synchronization. By default, the whole transform is included.
- In baking, separate server and client prefab entities are created for a single ghost type because its instances may not need all the same components on server and client. For example, ghost entities on the server usually don't need to render, so ghost prefabs on the server will generally not include rendering components.

When a subscene that contains ghost prefab instances is loaded on both server and client, Netcode for Entities automatically associates the instances on the server with their corresponding instances on the client. These are called *pre-spawned ghosts*.

To instantiate ghosts in code:

- On the server, when your code instantiates a ghost prefab (using the normal EntityManager Instantiate method), Netcode for Entities will automatically send a signal to the clients telling them to each create a corresponding mirror ghost instance.
- On the client, your code should generally not instantiate ghosts directly, except in the special case of "spawn prediction" (discussed later when we cover prediction).

To destroy ghosts in code:

- On the server, when your code destroys a ghost instance (using the normal EntityManager DestroyEntity method), a signal is sent to the clients telling them to destroy the corresponding mirror ghost instances.
- On the client, your code should never destroy ghosts. Doing so will trigger ghost synchronization errors.

Ghost snapshots

Each tick, the server sends to the clients "snapshots" of the ghost instances, meaning serializations of their ghost fields.

When the snapshot of an individual ghost is sent, it is always sent in whole, *i.e.* there is no such thing as a partial snapshot of an individual ghost. However, if the total size of all ghost snapshots in a tick exceeds the bandwidth limit of the server, the server will send snapshots in that tick only for some ghosts rather than for all. Netcode for Entities calls this a “partial snapshot”. Effectively, depending upon bandwidth, the number of ghosts, their sizes, and a few other parameters, snapshots of some ghosts may be sent at lower rates and less regular intervals. The server tries to prioritize the ghosts for which a snapshot has been sent least recently, but cases may still arise where clients end up receiving less frequent and less regularly spaced snapshot updates for some ghosts compared to others.

Consequently, in a client frame, the latest received snapshots for the various ghosts may not always reflect the state of the same tick. For example, the latest received snapshot for one character may represent the state of tick number 73 while the latest received snapshot for another character may represent the state of tick number 76. Generally, though, they should all be fairly close, say, within several ticks of each other rather than thousands of ticks apart. Just keep in mind that this means the client’s copy of the game state not only lags behind the server by some number of ticks, the client’s copy also doesn’t always fully reflect a consistent, coherent view of the game state from a single tick on the server.

> **Note**: In most scenarios, this lack of “consistency” on the client is hardly noticeable, but exceptions may arise in some games. Inconsistency can also have significant implications for prediction, as we’ll discuss later.

Ghost optimization mode

In a ghost prefab’s GhostAuthoringComponent, you can choose whether the ghost type should use either the “static” or “dynamic” optimization mode.

- For a ghost with the **“dynamic” optimization mode**, the server sends new snapshots regardless whether the ghost field data has changed since sending the last snapshot. However, the data is delta compressed against an earlier sent snapshot, so the new snapshot only includes what has changed.
- For a ghost with the **“static” optimization mode**, the server only sends new snapshots when any of the ghost field data has changed since sending the last snapshot. However, the snapshots always include the full ghost field data with no delta compression.

The choice of optimization mode should not affect the end result unless the change of bandwidth usage affects how frequently and regularly the snapshots for some ghosts are sent. Generally though, ghost types with ghost fields expected to change frequently should be made dynamic while ghost types with ghost fields expected to change only rarely should be made static.

> **Note**: Arguably “static” is a misnomer here because ghosts with the static optimization mode actually *are* allowed to change.

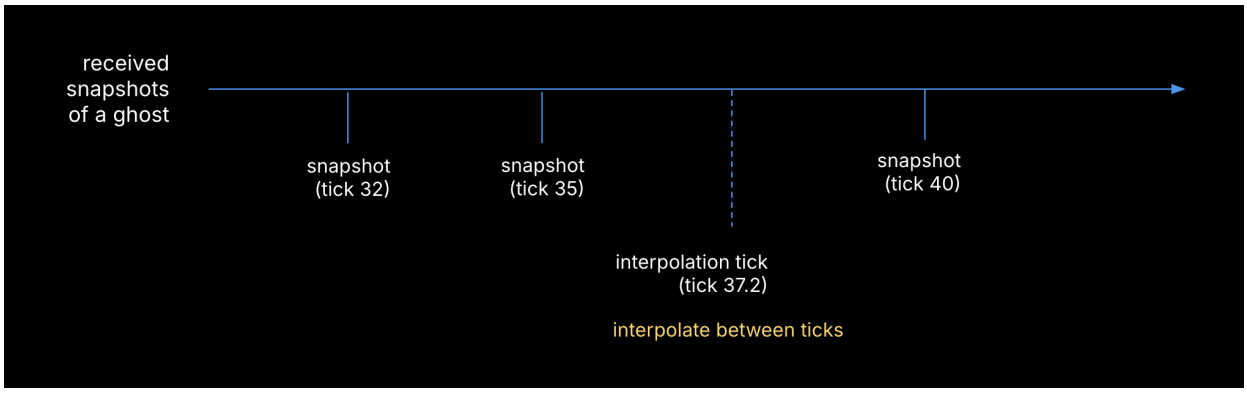
Ghost interpolation and extrapolation

As we’ve established above, a client’s framerate does not usually match and sync with the server tick rate, and a client may not always receive high-frequency and fully regular interval snapshots for all ghosts. Therefore, it’s often appropriate to apply interpolation or extrapolation for some ghost fields. In particular, for a moving ghost, you generally want to interpolate or extrapolate the transform because, otherwise, the ghost’s movements on the client would likely appear low-framerate and stuttery.

By default, the client will interpolate or extrapolate floating-point ghost fields (including the transform data):

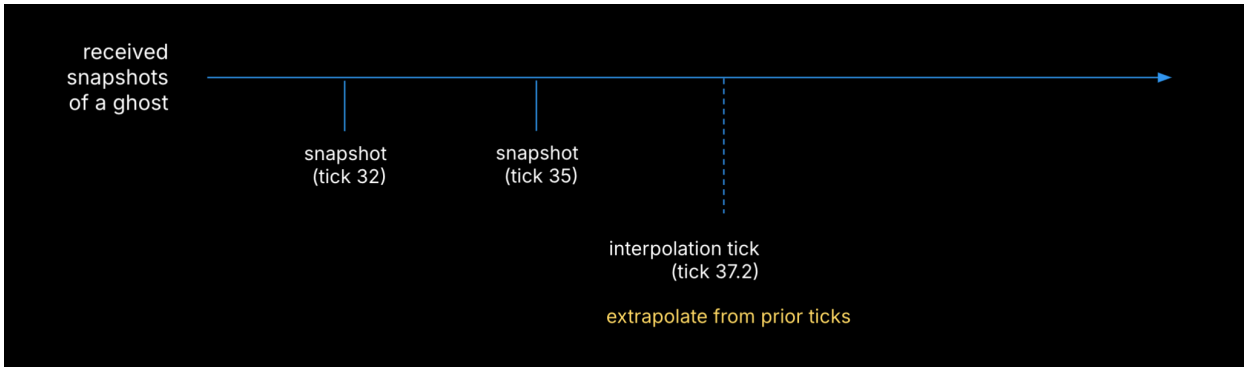
- The client keeps a history of multiple recently received snapshots for each ghost instead of just the single most recent.
- The client also maintains a target “interpolation tick” number, which is incremented at the server tick rate but with a bit of offset to keep it just under a rolling average of the last received snapshot tick numbers. For example, if the tick numbers of the latest received snapshots in a frame average out to 48.3, then the interpolation tick number might be something like 46.7.

When interpolating floating-point ghost fields, Netcode uses the two snapshots surrounding the interpolation tick:



Above, the values of the floating-point ghost fields from tick 35 and tick 40 are interpolated to tick 37.2. The actual ghost entity is then updated with these interpolated values.

If the interpolation tick number is greater than the tick number of the last received snapshot, Netcode may use the prior two snapshots to extrapolate:



Above, the float and double ghost field values from tick 32 and tick 35 are extrapolated to tick 37.2. The actual ghost entity is then updated with these extrapolated values.

Client-side prediction

Again, in the authoritative-server based model of netcode:

1. player input is sent to the server
2. the server factors the input into how it modifies the game state
3. snapshot updates of the new game state are sent back to the client

Effectively, then, there is a whole roundtrip time of latency between the player's input being sent and the result of the input showing up in a later frame on the player's screen.

Depending upon the exact gameplay scenario and the player's ping, this delay may be acceptable. For example, in a game where the user clicks to set waypoints to move characters, the latency probably doesn't feel like a big issue, even if the player has a fairly high ping (say 100-150 milliseconds). In many other cases, though, this roundtrip delay is extremely noticeable and detrimental to the player's experience. First-person shooters, especially, will often feel unplayable if there is even just a 30-40 millisecond delay when the player moves their character or fires their weapon.

Fortunately, this latency can often be disguised or at least greatly mitigated with a technique called *client-side prediction*.

The basic idea of client-side prediction is that a client attempts to "predict" (*i.e.* make an educated guess) what the game state *will* be on the server a fraction of a second into the future.

So say, on a client that is about to render a frame for tick 65.1, the client estimates what the player's state *will* be for tick 65.1 and then renders that, even though:

- The client only has snapshots from ticks that are at least a fraction of a second old (like say, in this example, from tick 57).
- The client has the recorded input history of its *own* player, but the client does not have the input data of the *other* players.
- The server itself only does whole ticks, not partial ticks, so there never will be a tick 65.1 on the server, only tick 65 and tick 66.
- The server itself at this moment has not yet processed tick 65. (Remember that clients always stay logically ahead of the server by some number of ticks so that their input has sufficient time to reach the server.)

Because the client's predictions are made from imperfect information, they may not always fully match the actual state on the server. We call such discrepancies *mispredictions*. As long as the mispredictions are small under normal network conditions, prediction can still greatly improve the player's experience.

In each frame, prediction is executed before rendering in two steps:

1. The state of the predicted ghosts are "rolled back" to an earlier tick:
    - A predicted ghost which receives a new snapshot is "rolled back" to match the snapshot state. For example, if a snapshot for tick 71 is received, then the ghost state is rolled back to match the snapshot for tick 71.
    - A predicted ghost which does not receive a new snapshot is rolled back to the last whole integer tick state that was cached in the prior frame.
2. The prediction simulation logic is executed tick-by-tick starting from the oldest rolled-back tick up to the tick to be rendered. For example, in a frame where the client is about to render tick 43.4 and the tick of the oldest rolled back ghost is 38, then the prediction logic will iterate five times: first to advance from tick 38 to 39, then from 39 to 40, 40 to 41, 41 to 42, and last from 42 to 43.4. (Notice that the "partial tick" is grouped together with the last whole tick.) This span of prediction ticks in a frame can be called the "prediction window".

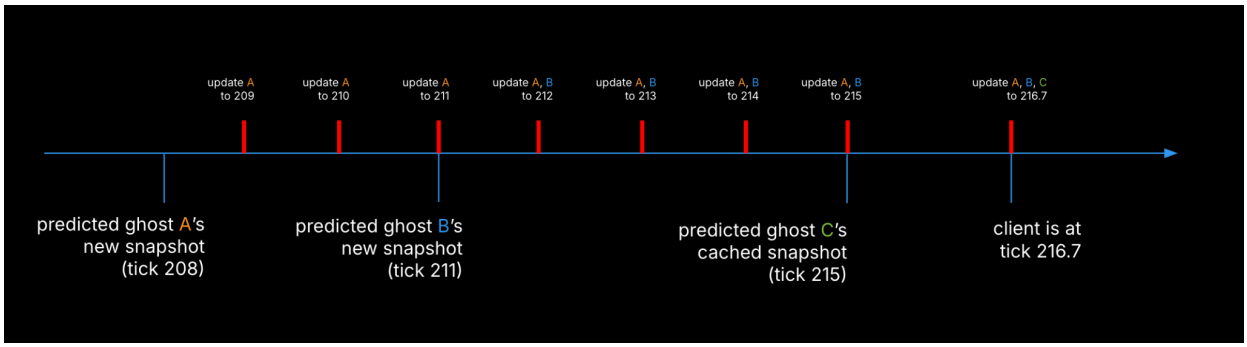
    Note: Prediction can be quite expensive for the client's CPU, and the more ticks in the prediction window, the costlier it becomes. To reduce the number of ticks to execute in the prediction window, Netcode has options for batching prediction ticks on the client.

In the prediction window, it is generally not appropriate to modify the state of a ghost until reaching the tick it was rolled back to. For example, in a frame where the prediction window runs

from tick 160 to tick 170, a ghost rolled back to tick 165 should not be updated in the ticks of the prediction window earlier than 165.

To help you abide by this rule, Netcode for Entities adds a Simulate tag component to every ghost. Netcode disables the Simulate component for every predicted ghost at the start of the prediction window; Netcode then enables the Simulate component for each ghost when the appropriate prediction tick is reached. In our example, a ghost rolled back to tick 165 will have its Simulate component disabled in all ticks of the prediction window prior to 165 and then enabled in tick 165 and all subsequent ticks. Effectively, by including the Simulate component in the queries of your prediction logic systems, you can make sure to only process predicted ghosts in the appropriate ticks of the prediction.

This diagram shows the prediction window of an example frame:



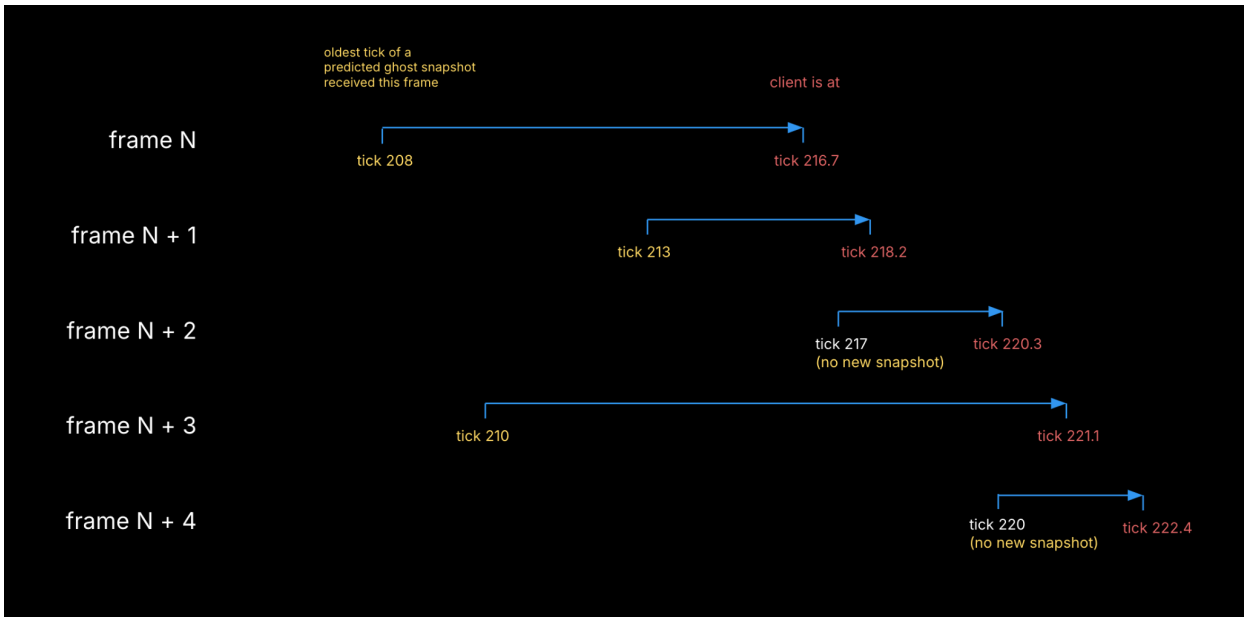
In the above frame:

- The current frame to render represents tick 216.7.
- Ghost A received a new snapshot and was rolled back to tick 208.
- Ghost B received a new snapshot and was rolled back to tick 211.
- Ghost C did not receive a new snapshot and so was rolled back to tick 215 (a state cached from prediction of the prior frame).
- The Simulate component is first disabled for all of the ghosts and then enabled:
    - …for ghost A at the start of tick 208 of prediction (the first tick).
    - …for ghost B at the start of tick 211 of prediction.
    - …for ghost C at the start of tick 215 of prediction.
- The prediction window runs from 208 (the oldest tick among the predicted ghosts) up to 216.7.
- Right before prediction performs the last tick, the state of A, B, and C at tick 215 will be cached for use in the next frame.

    Warning: Always remember that the latest received snapshots of the various ghosts are not necessarily from the same tick, so the client does not always have a consistent view

of the game state in each tick of prediction. This is one reason why a client's predictions cannot always be fully accurate.

This diagram shows how the prediction windows of adjacent frames may overlap:



Note above that:

- While the tick to render always advances every frame, the starting tick of the prediction window may actually be older than that of the prior frame if an older tick snapshot is received. Above, in frame N+2, the starting tick is 217, but then in the next frame, the starting tick is 210.
- In frames where no new predicted ghost snapshots are received, the prediction window may be as short as a single tick.

In Netcode for Entities, you opt-in to prediction for each individual ghost type *via* the GhostAuthoringComponent's DefaultGhostMode property, which has three options:

- **Interpolated**: interpolated and extrapolated with no prediction
- **Predicted**: the client predicts all instances
- **OwnerPredicted**: the client only predicts the instance which it "owns" (generally meaning the character or vehicle controlled by the client's local player)

  Note: The GhostMode of individual instances can also be set at runtime.

In most games, the primary use case for prediction is a character or vehicle directly controlled by the player. Because the client has the recorded history of the local player's inputs, the client's prediction logic can often make very accurate predictions about the player character's movements and actions by re-enacting the player's inputs tick-by-tick.

In contrast, most games do *not* attempt to predict the state of characters or vehicles controlled by the other players. Even if the server relays every player's inputs to the other clients (an option supported by Netcode for Entities), it takes time to relay the data, and so a client can never have the other players' inputs for the full prediction window: the latest inputs will always be missing. Consequently, predictions for characters or vehicles controlled by the other players can't be as accurate as for the client's own local player.

> **Note**: Depending upon the nature of the game and the player interactions, predicting other players might still improve the experience. In a car racing game, for example, car movements are generally grounded in momentum and other realistic(ish) physical behaviour, so player steering, acceleration, and braking only *gradually* change a car's vector of velocity. Consequently, over a short window of ticks, the player inputs usually only have a minor effect on the apparent motion of the vehicle. In such cases, even if a client lacks the latest few ticks of input from the other players, it can still predict the car movements fairly well.

Be clear that latency is usually only a real concern for things that move and animate: other things usually feel just fine with a fraction of a second of latency. While a client could try predicting other facets of game state, like say when an item is added into a character inventory, doing so may require running expensive and complicated logic on the client, to no real apparent benefit for the player. It's generally much simpler and cheaper to just let a character's inventory on the client reflect the latest received snapshot state verbatim.

Even things that move don't necessarily warrant prediction unless they require low-latency responses from player interactions. For example, in the sample that we'll cover below, the balls are predicted so that they can respond immediately when the player kicks them or bumps into them. Whether this immediacy warrants the cost and complication of prediction is something you must judge on a case-by-case basis.

RPCs

Another feature of Netcode for Entities is RPCs (Remote Procedure Calls), which are discrete messages that can be sent from the server to the clients or from a client to the server. These messages are "reliable" in the sense that the message will be repeatedly resent until the message's receiver sends back an acknowledgement.

> **Note**: “Remote Procedure Call” is arguably not the most fitting name, as there are no procedure calls involved here. Rather, the RPC messages are just serialized structs. Whatever action is performed when the message is read (if any) is left up to the receiver.

In some other netcode solutions (including Netcode for GameObjects), RPCs are often used as an integral part of the game logic. In Netcode for Entities, however, they are often only used for managing the state of the match and other concerns that don’t directly affect the game state. Text chat messages, for example, might be sent *via* RPC.

Bootstrapping the client and server worlds

In Entities, “bootstrapping” refers to the creation of entity worlds and adding systems to them at the start of runtime. The default bootstrapping code creates a “default” entity world and adds to the world certain standard systems (such as the transform systems). Default bootstrapping can be overridden with your own bootstrapping code by defining a class in your project that implements ICustomBootstrap.

Netcode for Entities provides its own class that implements ICustomBootstrap called ClientServerBootstrap. This bootstrapper will create both client and server worlds or just one of the two, depending upon the MultiplayerPlayModePreferences (which can be set for play mode in the PlayMode Tools window).

The separation of server and client entities and systems into separate worlds allows Netcode for Entities to run both server and client (or even multiple clients) within a single instance of Unity rather than always having to run them as separate builds or instances.

Connections

On a client, the connection to the server is represented as an entity. On the server, the connections to the clients are also represented as entities (one per client).

The connection entities all have a few connection-related components, including NetworkStreamConnection, NetworkId (an integer which uniquely identifies the connection), IncomingRpcDataStreamBuffer, OutgoingRpcDataStreamBuffer, and possibly a few others.

The connection entities are created by Netcode for Entities when a client connects to the server, but initially the connection will not send or receive input, RPCs, or ghost snapshots until you signal that the connection is ready by adding the NetworkStreamInGame component. Only once

NetworkStreamInGame is present on the connection entity of both ends will Netcode send input, RPCs, and ghost snapshots over the connection.

Additional Netcode for Entities features

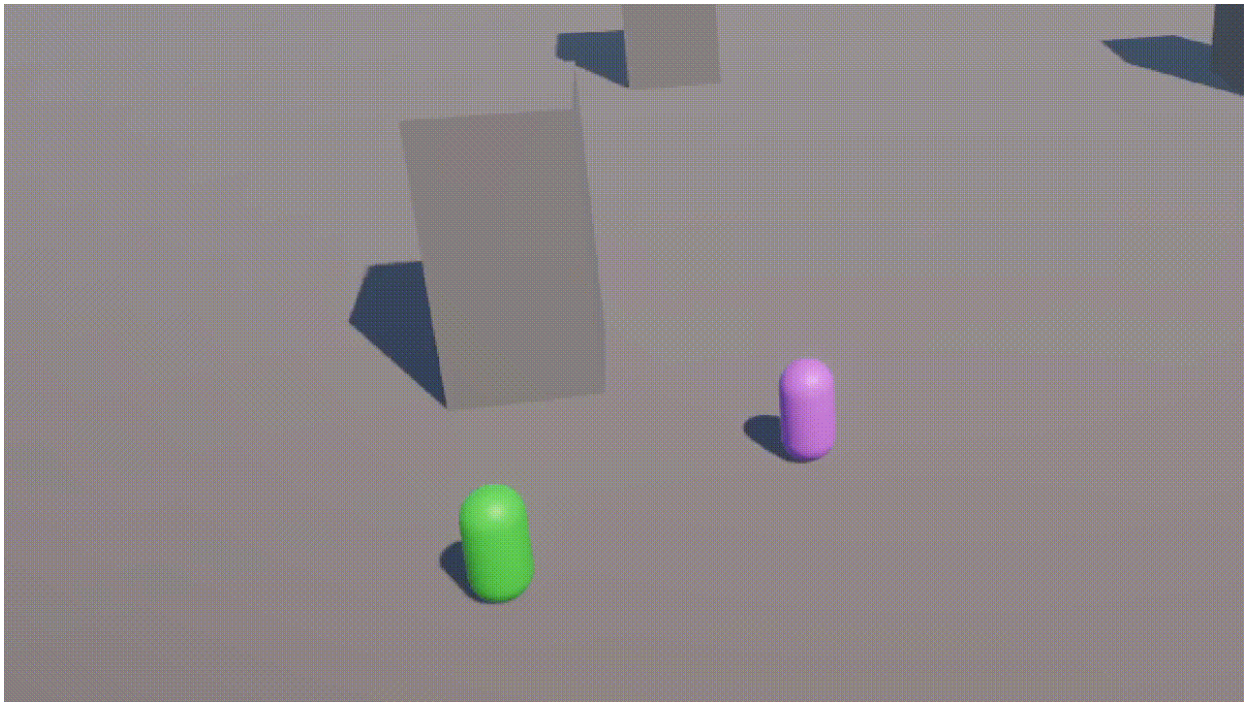
Features of Netcode for Entities which we won't cover here:

- Ghost field quantization
- Ghost importance
- Ghost relevancy
- Ghost prediction switching
- Ghost prediction smoothing
- Ghost groups
- Ghosts with child entities
- Custom serialization
- Physics integration
- Ghost component variants
- Client prediction tick batching
- Server tick batching
- Server-side lag compensation

The Kickball sample

Now let's walk through a simple intro project in which:

- A player character is spawned when a client connects to the server
- Players can control their characters' movements
- Players can spawn balls
- Players can kick balls



Step 1: Bootstrapping and connecting

First in the project, we must bootstrap the client and server worlds and perform the appropriate setup when a client connects to the server. The relevant files under the Assets directory are:

- **Scripts/GameBootstrap**: A class extending ClientServerBootstrap that specifies the server port
- **Scripts/ConfigScriptableObject**: A scriptable object that contains several configuration parameters of the sample, such as how fast the players move.
- **Authoring/ConfigAuthoring**: An authoring component that adds the configuration data from ConfigScriptableObject to an entity.
- **Systems/GoInGameClientSystem**: A system that adds the NetworkStreamInGame component to the connection entity on the client and sends an RPC from the client to the server.
- **Systems/GoInGameServerSystem**: A system that listens for the RPC from the client and, upon receipt, adds the NetworkStreamInGame component to the connection entity for that client and spawns a player ghost for the client.

Step 2: Obstacles

Second, we want to spawn obstacles (the pillars). Because we want the server to choose how many obstacles to spawn and where to place them, the obstacles must then be ghosts so that their state can be synced to the clients. The relevant files under the Assets directory are:

- **Prefabs/Obstacle**: A simple rendered pillar with a box collider. Like all ghost type prefabs, the root GameObject of the prefab has the GhostAuthoringComponent. Because the obstacles are only moved when they are initially placed, the ghost type uses the static optimization mode and the interpolated default ghost mode (there would be no reason to predict an unmoving, unchanging ghost).
- **Authoring/ObstacleAuthoring**: An authoring component that adds an Obstacle tag component to the obstacle ghost entities (to allow our code to query for the obstacles).
- **Systems/ObstacleSpawnerSystem**: A system that spawns the obstacles when the server starts running.

Step 3: Player characters

Third, we want to spawn player characters and move them with player input. The relevant files under the Assets directory are:

- **Prefabs/Player**: A ghost type representing the players: simple rendered capsules with capsule colliders and kinematic rigidbodies (so that the balls will move when they collide with the player). In the GhostAuthoringComponent, the default ghost mode is set to "owner predicted", meaning each client will only predict the instance representing their own local player and not the instances representing the other players.
- **Authoring/PlayerAuthoring**: An authoring component that adds a Player tag component to the player ghost entities (to allow our code to query for the players). Also adds a Color component which determines the player's rendered color.
- **Authoring/PlayerInputAuthoring**: An authoring component that adds PlayerInput, an IInputCommandData component, to the player ghost entities. PlayerInput defines the input command stream data sent from the clients to the server.
- **Systems/PlayerInputSystem**: A system that sets the PlayerInput component every frame with the player's local input. (The client gathers the player's input using the Input System package.)
- **Systems/PlayerMovementSystem**: A system that moves the players in accordance with the players' input. On the server, this server authoritatively sets the players' positions every tick. On a client, this system predicts the movements of just that client's local player. (Note that the very simplistic player movement in this sample doesn't account for environment collisions, so players can walk right through the obstacles.)

Step 4: Balls

Lastly, we want to give player characters the ability to spawn balls and kick them. The relevant files under the Assets directory are:

- **Prefabs/Ball**: A ghost type representing the balls: rendered spheres with sphere colliders and dynamic rigidbodies. In the GhostAuthoringComponent, the default ghost mode is set to “predicted”, so every instance is predicted on every client.
- **Authoring/BallAuthoring**: An authoring component that adds a Ball tag component to the ball ghost entities (to allow our code to query for the balls). Also adds a Color component which determines the balls’s rendered color.
- **Systems/BallSpawnSystem**: A system that spawns a ball above a player’s head when a player hits the spawn button.
- **Systems/BallKickingSystem**: A system that applies force to the balls when a nearby player hits the kick button.