Netick

Offline Documentation

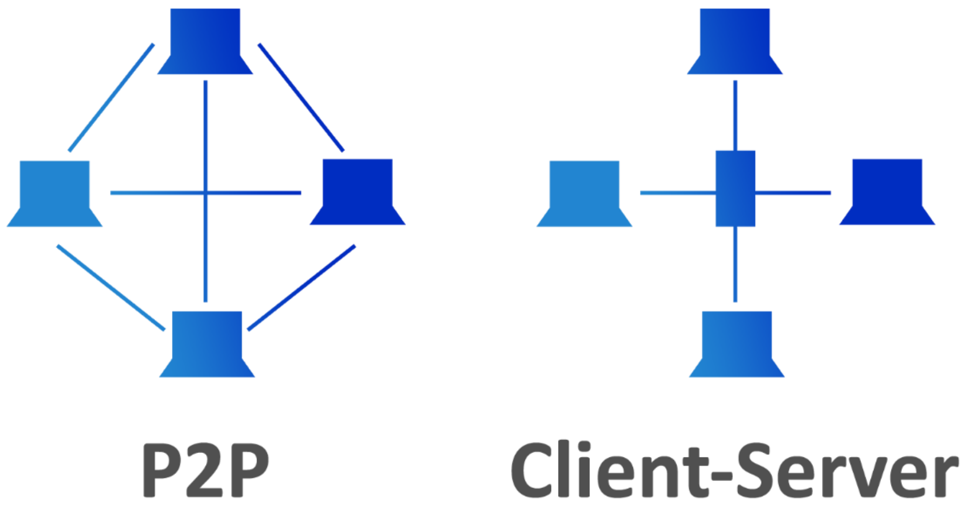
Website: www.netick.net

Please read the online version:

www.netick.net/docs/manual

Understanding Client-Server Model

When programming a single-player game, you usually don't care about making sure the behavior and actions of players are legal, because it ultimately doesn't matter since the game runs completely on their machine, and it's impossible to prevent cheating when the client (player) has full access to everything related to the game. Because there is no central authority dictating the flow of the game, and enforcing the game rules and mechanics.



That's why it's extremely easy to cheat in peer-to-peer games (where every player is connected to everyone else and everyone can decide whatever they want). Each client can interpret the outcome of the game however it wants. This is the problem that the client-server model solves.

In this model, rather than connecting the players to each other, every player is simply connected to a single node/machine called the server. In a perfect client-server implementation, the client simply sends inputs (which describe what they want to happen, e.g. moving, shooting) and the server basically responds by sending back the resultant game state to that input. Therefore, the server is the one who executes the inputs and so there is no way for the client to cheat.

Netick implements the client-server model + client-side prediction. You will see later how to construct these inputs and states to create your server-authoritative gameplay systems.

Core Concepts

Network Sandbox

Network Sandbox is what controls the whole network simulation. It can be thought of as the manager of the simulation. You can have more than one network sandbox in a single Unity game, and that happens when you start both a client and a server on the same project. This can be extremely useful for testing/debugging, because it allows you to run a server and a client (or multiple thereof) in the same project and therefore see what happens at both at the same time, without interference.

- Therefore you can think of a sandbox as representing a server or a client.
- You can show/hide the current sandboxes from the Network Sandboxes panel.

Network Object

Any GameObject which needs to be synced/replicated must be a Network Object (has the Network Object added to it). If you want to see something on everyone's screen, it has to have a Network Object component added to it. It's the component that tells Netick that a GameObject is networked. The Network Object component by itself just informs Netick that the object is networked. To add networked gameplay-logic to it, you must do so in a component of a class derived from Network Behavior. Netick comes with a few essential built-in components:

- Network Transform: used to sync position and rotation
- Network Rigidbody: used to sync controllable physical objects
- Network Animator: used to sync Unity's animator's state

Network Behavior

The Network Behavior class is your old friend MonoBehaviour, just the networked version of it. To implement your networked functionality, just create a new class and derive it from NetworkBehavior. You have several methods you can override which correspond to Unity's non-networked equivalents (they must be used instead of Unity's equivalents when doing anything related to the network simulation):

- NetworkStart
- NetworkDestroy
- NetworkFixedUpdate
- NetworkUpdate
- NetworkRender

Example:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using Netick;

public class MyBehaviour : NetworkBehaviour
{
   [Networked]
   public int   IntPropertyExample   { get; set;}

   [Networked]
   public float FloatPropertyExample { get; set;}

   public override void NetworkStart()
   {
      // Called when this object has been added to the simulation.
   }

   public override void NetworkDestroy()
   {
      // Called when this object has been removed from the simulation.
   }

   public override void NetworkUpdate()
   {
      // Called every frame. Executed before NetworkFixedUpdate.
   }

   public override void NetworkRender()
   {
      // Called every frame. Executed after NetworkUpdate and     NetworkFixedUpdate.
      // IMPORTANT NOTE: properties (which can be interpolated) marked with [Smooth] attribute will return
interpolated values when accessed in this method.
   }

   public override void NetworkFixedUpdate()
   {
      // Called every fixed-time network update/tick. Any changes/updates to the network state must happen here.
      // Check out the chapter named "Writing Client-Side Prediction code" to learn more about this method.
   }

}
```

Don’t forget to include `using Netick`

A class derived from Networked Behavior is almost useless without the utilization of Network Properties, which are the building blocks of your networked synced state. These properties are ensured to be eventually synced to everyone in the network, letting you create objects with complex states and not worry about it.

Network State

Network Property

A Network Property is a C# property which is replicated across the network. For a property to be networked, the Attribute [Networked] must be added to it. An example of a networked property:

```
[Networked]
public int Health {get; set;}
```

Supported C# Types

1. Int/UInt
2. Float
3. Bool
4. String
5. Enums
6. Byte
7. Long/ULong
8. Double

Supported Unity Types:

1. Vector2
2. Vector3
3. Quaternion
4. Color

Network Arrays

Network arrays are just like regular C# arrays, but their syntax is a bit different. They are defined using the NetworkArray generic class.

Example of a network array:

```
[Networked (size: 10)]
public NetworkArray<int> Items { get; set; }
```

The size parameter is used to specify the array's length.

Custom Networked Structs

You can define custom structs that can be networked/replicated by adding the attribute [Networked] to them, like this:

```
[Networked]
public struct Inventory
{
   public int Item1;
   public int Item2;
}
```

Important notes:

1. It's important to make sure that a single struct doesn't exceed the maximum property size, which is 50 bytes.
2. Structs must not include other user-defined networked structs, only C#'s and Unity's primitive types.
3. You might want to override/implement equality for the struct for better performance.

Change Callback

For Properties

You can have a method get called whenever a networked property changes, which is very useful. To do that, add the attribute `[OnChanged]` to the method and give it the name of the property.

Example:

```
[Networked]
public int Health { get; set; }

[OnChanged(nameof(Health ))]
private void OnHealthChanged()
{
      // Something that happens when the Health property changes
}
```

For Arrays

If you have an array and want a callback for when an element changes, you can do so as follows:

Example:

```
[Networked(size: 10)]
public NetworkArray<int> Items { get; set; }

[OnChanged(nameof(Items))]
private void ArrayChanged(int index)
{
    // Something that happens when an element of the array changes
}
```

The difference here to a variable property is that the callback method must have an index parameter that points to the changed element.

Remote Procedure Calls (RPC)

RPCs are method calls on Network Behaviors that are replicated across the network. They can be used for events or to transfer data.

An important use of RPCs is to set up the game and send configuration messages. Use reliable RPCs for things like this.

An RPC example:

```
[Rpc(source: RpcPeers.Everyone, target: RpcPeers.InputSource, isReliable: true, localInvoke: false)]
private void MyRpc(int arg1)
{
  // Code to be executed
}
```

You use the [Rpc] attribute to mark a method as an RPC.

1. RPCs are not called on restimulated ticks.
2. By default all RPCs are unreliable.

RPC method constraints:

- Must have the return type of void.
- Must only include parameters that can be networked. Same ones as the properties

[Rpc] attribute parameters

- Source: the peer/peers the RPC should be sent from
- Target: the peer/peers the RPC will be executed on
- isReliable: whether the RPC is sent reliably or unreliably
- localInvoke: whether to invoke the RPC locally or not

Source and target can be any of the following:

- Owner (the server)
- Input Source: the client which is providing inputs for this Network Object
- Proxies: everyone except the Owner and the Input Source
- Everyone: the server and every connected client

Source Connection of RPCs

If you need to know which connection (a client, or the server) the current RPC is being executed from, you can use Sandbox.RpcSource

```
[Rpc(source: RpcPeers.Everyone, target: RpcPeers.InputSource, isReliable: true, localInvoke: false)]
private void MyRpc(int arg1)
{
  var rpcSource = Sandbox.RpcSource;
}
```

RPCs vs Properties

RPCs are usually used to replicate non-critical (often visual/cosmetic) events. In contrast, Network Properties are used to replicate critical gameplay state.

Network Properties are best when you have a variable that is constantly changing and whose exact value matters for the duration of the game, because properties will eventually replicate to everyone.

Example: a health property.

On the other hand, RPCs are only relevant at the time of their execution, meaning anyone joining after that will never know anything about any RPCs before it.

Example: a damage effect event.

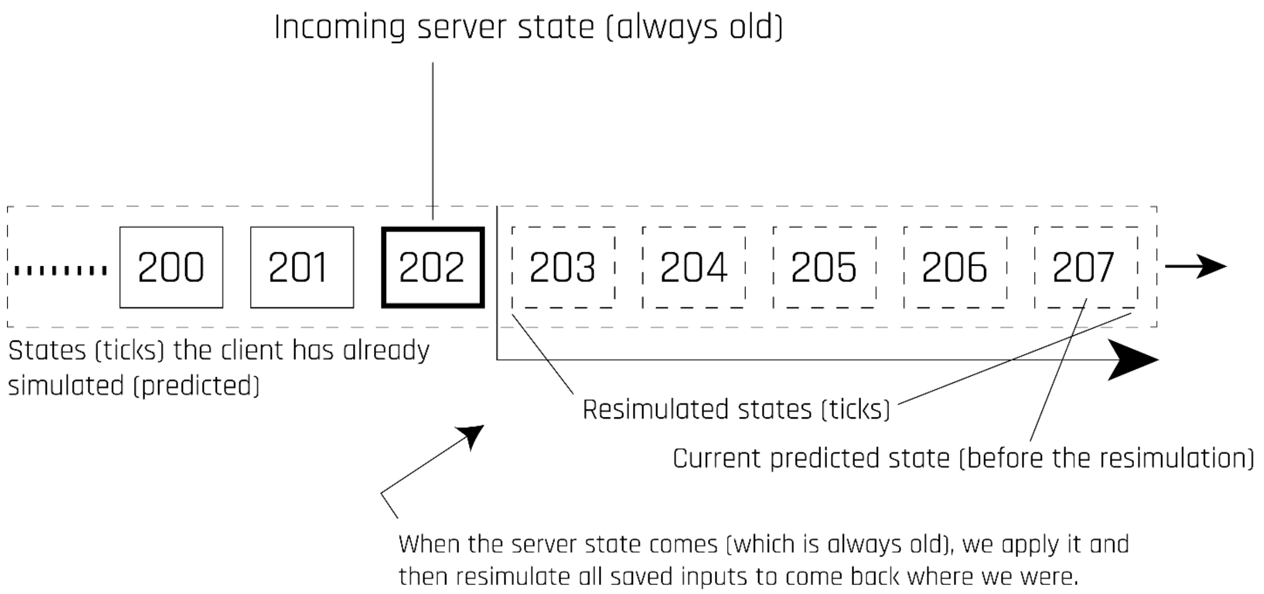
If an event happens infrequently and is merely visual (doesn't affect gameplay, for example, a sound effect event) it might be more appropriate to use a Remote Procedure Call (RPC) for it.

Often you can evade using RPCs even for events, and that's by using a change callback using [OnChanged] attribute.

Understanding Client-Side Prediction

In the Client-Server model, to be able to change the state (values of properties/arrays) of a network object, that change must be authoritatively done on the server. This is to ensure a secure and cheat-free gameplay experience, because ultimately the client's executable can be tampered with or modified. **Only the server can ever change the true state of network variables.** What the client does to affect changes to the networked state is send inputs which are later executed/simulated by the server to produce the desired state which is sent back to the client/s.

This is obviously not practical due to internet latency (round-trip time), as the latency increases, input delay increases. This will, without a doubt, lead to a very unpleasant and unresponsive gameplay experience. The solution to this is what's commonly known as **Client-Side Prediction**.


Incoming server state (always old)
200
201
202
203
204
205
206
207
States (ticks) the client has already simulated (predicted)
Resimulated states (ticks)
Current predicted state (before the resimulation)
When the server state comes (which is always old), we apply it and then resimulate all saved inputs to come back where we were.

Client-Side Prediction basically means that the client, instead of waiting for the server to simulate its inputs and send the resultant states to it, the client executes them locally (in other words, predicts their outcome), and when the resultant state comes in, it applies that state and resimulate every input of a tick newer than that state's tick. All this happens in one tick.

This ensures that the server still has the final say on the authority of the game (because, eventually, the client will overwrite its local state with whatever the server says), but at the same time allows the client to locally predict their input outcome and enjoy a lag-free experience.

All simulation code must be done inside NetworkFixedUpdate on Network Behaviors. This method is called every network tick to step forward the simulation. **On the server, this method is only called for new inputs.** While on the client, it can and will be called several times in one network tick to resimulate all saved inputs (up to the current predicted tick) when applying the incoming server state. See the previous figure to fully understand this.

On what objects do resimulations happen?

- Objects the client is the Input Source for.
- Objects which has their Prediction Mode set to Everyone, instead of Input Source. Meaning not only the client who's the Input Source predict them, but all other clients too.

For other objects, it will only be called once for every network step/tick.

Don't forget that the server only ever simulates new ticks, it never resimulates previous ticks/inputs. CSP is exclusive to clients. For the server, it's just like it's a single-player game.

For movement code, being aware of resimulations is unimportant. However, for things like shooting and other similar events, it's vital to make sure that they only happen when the input is being simulated for the first time ever, otherwise, you would shoot several times for one bullet on the client, due to resimulations. This hazard is important to understand and deal with.

Note that it's usually impractical to predict everything the client does in the game, and it's sometimes way easier to not let the client predict some stuff (due to the complexity that is associated with correcting some predictions), and wait for the server state. And for other things, simply making them client-authoritative saves a lot of headaches. You don't have to make the game completely server-authoritative. *Only the bits which are vital to the game experience.*

Writing Client-Side Prediction code

Before continuing, make sure you read Understanding Client-Side Prediction

Network Input

Network Input describes what the player wants to do, which will be used to simulate the state of objects they want to control. This ensures that the client can't directly change the state – the change happens by executing the input, which, even if tampered with, won't be game-breaking.

Defining Inputs

To define a new input, create a class that inherits from NetworkInput:

```
public class MyInput : NetworkInput
{
   public bool      ShootInput;
   public float     MoveDirX, MoveDirY;
}
```

Setting Inputs

To set the variables of an input, you first need to acquire the input object of the next tick, using `Sandbox.GetInput`.
Then, you can set it inside `NetworkUpdate` on NetworkBehaviour:

```
public override void NetworkUpdate()
{
   var input = Sandbox.GetInput<MyInput>();

   input.MoveDirX = Input.GetAxis("Horizontal");
   input.MoveDirY = Input.GetAxis("Vertical");
}
```

You could also set them on `OnInput` of NetworkEventsListner, which is preferred.

Simulating/Executing Inputs

To drive the gameplay based on the input object, you must do that in `NetworkFixedUpdate`:

```
public override void NetworkFixedUpdate()
{
   if (FetchInput(out MyInput input))
   {
      // movement
      var movement = transform.TransformVector(new  Vector3(input.MoveDirX, 0, input.MoveDirY)) * Speed;
      movement.y      = 0;
      _CC.Move(movement * Time.fixedDeltaTime)
      // shooting
      if (input.ShootInput == true && !IsResimulating)
         Shot();
   }
```

Don’t confuse Sandbox.GetInput with FetchInput.
– Sandbox.GetInput is used to read/set the user inputs into the input object.
– FetchInput is used to actually use the input object in the simulation.

FetchInput tries to fetch an input for the state/tick being simulated/resimulated. It only returns true if either:

1. We are providing inputs to this object – meaning we are the Input Source of the object.
2. We are the owner (the server) of this object – receiving inputs from the client who’s the Input Source. And only if we have an input for the current tick being simulated. If not, it would return false. Usually, that happens due to packet loss.

And to avoid the previous issue we talked about, we make sure that we are only shooting if we are simulating a new input, by checking IsResimulating.

Input Source

For a client to be able to provide inputs to be used in an Object’s NetworkFixedUpdate, and hence take control of it, that client must be the Input Source of that object. Otherwise, FetchInput will return false. To check if you are the Input Source, use IsInputSource.

The server can also be the Input Source of objects, although it won’t do any CSP, since it needs not to, after all, it’s the server.

You can set the Input Source of an object when instantiating it:

```
sandbox.NetworkInstantiate(PlayerPrefab, spawnPos, Quaternion.identity, client);
```

To set/remove the Input Source (must only be called on the server):

1. Call PermitInput to set the Input Source of the object: Object.PermitInput(client);
2. Call RevokeInput to remove the Input Source of the object: Object.RevokeInput();

There are several methods you can override to run code when Input Source is assigned/removed or has left:

1. OnInputPermitted: called on the player who was given permission to provide inputs to this object.
2. OnInputRevoked: called on the player whose permission to provide inputs to this object has been revoked by the owner (server).
3. OnInputSourceLeft: called on the owner (server) when the Input Source client has left the game.

Network Object Instantiation and Destruction

To Instantiate a network prefab, call NetworkInstantiate on the Network Sandbox. You must only instantiate/destroy network objects on the server, never on the clients.

```
sandbox.NetworkInstantiate(prefab)
```

To destroy any networked object:

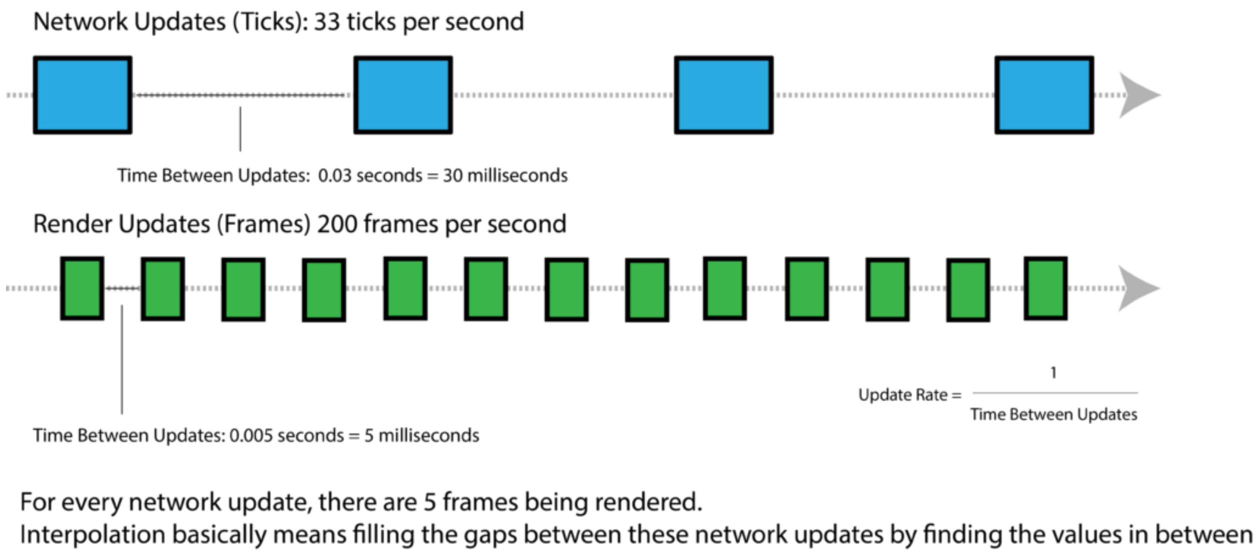
```
sandbox.Destroy(obj)
```

This will destroy obj and all of its nested Network Objects. Must be called only from the server/owner.

Note: make sure to never use Unity's instantiate/destroy methods to create/destroy a network object, only Netick methods.
Note: make sure that all your prefabs are registered by Netick in Netick Settings panel. And also make sure the prefab list is identical in both the client and the server (if you are running two Unity editors), otherwise, weird stuff will occur.

Interpolation

Netick runs at a fixed-time step, the same as the Time.fixedDeltaTime of Unity, which you can specify in the Project Settings. Because of that, the motion of network objects will appear unsmooth and jittery. The reason for this is that, usually, your update rate (render rate) is way higher than your fixed network tick rate. The solution to this problem is called interpolation, which means filling in the gaps between these fixed-time steps/ticks:


Network Updates (Ticks): 33 ticks per second
Time Between Updates: 0.03 seconds = 30 milliseconds
Render Updates (Frames) 200 frames per second
Update Rate = 1 / Time Between Updates
Time Between Updates: 0.005 seconds = 5 milliseconds
For every network update, there are 5 frames being rendered.
Interpolation basically means filling the gaps between these network updates by finding the values in between

So, for example, at tick 6, the value of a network property is 2.0. And at tick 7, it becomes 3.0. Since there are 5 frames between two ticks, the values at each frame would be:

- Frame 1: 2.0 — Beginning of tick 6
- Frame 2: 2.25
- Frame 3: 2.5
- Frame 4: 2.75
- Frame 5: 3 — End of tick 6, beginning of tick 7

Interpolation of Network Transform

For moving objects, this is important to deal with. Every NetworkTransform has a slot for a Render transform, which is basically the smoothed/interpolated mesh of the object, while the parent would be the simulated/non-interpolated object.

So, you must break your moving objects into a parent (which has the NetworkTransform), and a child which is the interpolated object, and has the mesh/s. Then you specify that child in the Network Transform Render Transform property in the inspector. Check the samples if you are confused.

Interpolation of Network Properties

To interpolate a property, add the [Smooth] attribute to its declaration:

```
[Networked][Smooth]
public Vector3 Movement {get; set;}
```

Accessing Interpolated Values

To access the interpolated value, by referencing the property in NetworkRender, you automatically get interpolated values:

```
public override NetworkRender()
{
   var interpolatedValue = Movement;
}
```

Interpolation is implemented by Netick on these types:

- Float
- Double
- Vector2/Vector3
- Quaternion

Other types just return the From snapshot between the two snapshots being interpolated.

Interpolation of Other Types

To interpolate other types, you can do that using the Interpolator<T> object.

First, you must include an id in [Smooth]:

```
[Networked][Smooth(6)]
public MyType SomeProperty {get; set;}
```

And to access the interpolated value, you first need to acquire a reference to the Interpolator object, through which you can get the From and To values, and the Alpha, which you use to interpolate the property.

```
public override NetworkRender()
{
   var interpolator          = FindInterpolator<MyType>(6);
   var from                  = interpolator.From;
   var to                    = interpolator.To;
   var alpha                 = interpolator.Alpha;
   var interpolatedValue  = LerpMyType(from,to,alpha);
}

private MyType LerpMyType(MyType from, MyType to, float alpha)
{
   // write the interpolation code here
}
```

Note: you should cache the result to FindInterpolator<MyType>(5) on NetworkStart, instead of calling it repeatedly every frame (NetworkRender is called every frame), since it might be a bit slow.