



Simple Network Sync for PUN2

Networking Component System

© 2020 emotitron | Exit Games | Davin Carten

[Current version of this document online here](#)

Contact the author at: davincarten@gmail.com

I am also regularly on the PhotonEngine channel of Discord as emotitron if you have questions.

<https://discord.gg/egaRfd8>

I strongly suggest joining so help provided to users can be useful to others.

Current releases of pre-beta are here:

https://github.com/emotitron/SNS_PUN2/releases

Tutorial:

[SNS for PUN2 Tutorial](#)

**Be sure to check for updates regularly while this is early Beta!
And don't hesitate to contact me.**

Documentation is intentionally still minimalistic. PLEASE, contact me with any questions or for help, I am very quick to respond. I am avoiding too much documentation work until the first version of this has finished being updated as use cases come up, as it will quickly become outdated and wrong.

This software is always being improved and tightened up, and the main driver outside of my own usage is the devs using it requesting changes, improvements and fixes. email me or even better hop on to discord and let me know what you think it needs, or let me know how you were able to break it.

Overview

- ❑ [Base set of “Just Works” Components and Interfaces.](#)
- ❑ [Simulation-based tick system with segmented deferred timings](#)
- ❑ [Numbered State Buffers \(Circular Buffer\)](#)
- ❑ [Serialization to a unified bitpacked byte\[\] array](#)
- ❑ [Syncvars tied to the core simulation tick timing system](#)
- ❑ [Keyframe based eventual consistency](#)

Many networking libraries available to Unity developers focus primarily on synchronization of fields and replicating events from one client to another. This often comes in the form of Syncvars and RPCs (Remote Procedure Calls). While this can lead to quick successes, it often becomes very entangled and cumbersome as the race conditions, cross dependencies and lack of deterministic order start to compound complexity. In contrast, **Simple Network Sync (SNS) extends PUN2 to operate on a simulation-based tick timing system that uses circular buffers with segmented and deferred timings.**

SNS uses mixed-authority snapshot interpolation. PUN2 is a relay environment, so it is a slightly different architecture than Server Authority, which systems like Photon Bolt employ. As there is no central state authority, authority is distributed. Players typically are the authority over objects they are in control of. PUN2 / SNS as such make use of Client/Player Ownership.

For example: If a player shoots another, the shooter notifies the hit player's health system of the hit. The hit player having authority over their own health applies the damage in response to the hit indication from the shooter. That hit player will also accordingly apply death, stun, etc. to themselves. It should be noted that this is inherently NOT cheat resistant. The PUN relay model is more about quick and easy development and simple hosting, and is not the ideal library for highly competitive PVP games. Bolt or Quantum would be more suitable platforms for such games.

The SNS system primarily uses Unreliable UDP with Keyframe and Delta frames, similar to how video works. [More on this below.](#)

SNS System Highlights

Base set of “Just Works” Components and Interfaces

The first and most obvious benefit of the new extension library is the quantity of code that exists in the form of interoperational components. Without writing a single line of code you can have a networked scene with synced:

- Transforms
- Animators
- Hitscans & Projectiles
- Health/Vitals and automation of UI
- State (Attached, Visible, Thrown/Dropped, etc.)
- Visibility / Rigidbody changes in response to State changes
- Mounting to other objects
- Damage/Healing/Buff/Debuff zones
- Inventory and Inventory Items
- Health / Energy Pickups

These components can be derived and extended, or the base interfaces can be used to create components from scratch that interact with these systems.

Simulation-based tick system with segmented deferred timings

The NetMaster singleton acts as a central Update Manager, so the entire system is polled every tick to produce a series of tick-based callbacks that very tightly control execution of PreSimulation, PostSimulation, State Capture, Serialization, Deserialization, State Application, Interpolation and Extrapolation code. This eliminates race conditions and keeps all objects tightly synchronized to themselves and other objects with the same owner. As such, many things become largely deterministic in nature. For example, when a player triggers a hitscan, the shot is replicated on other clients by applying states in the same order, reproducing the hitscan as it happened on the owner.

The SNS component set is tuned for games that simulate/move/animate using timings based around FixedUpdate(). It is designed to work in simulation-free or Update() based simulations as well, but those are generally less than ideal for networking. and are prone to micro-jitter. It is recommended to use the NetMaster timing segments to better ensure order:

- **Perform all game logic in Fixed**
OnPreSimulate and OnPostSimulate.
- **Perform all interpolation/extrapolation in Update**
OnPreUpdate, OnPostUpdate, OnPreLateUpdate & OnPostLateUpdate

Numbered State buffers ([Circular Buffer](#))

The core FixedUpdate based timing system captures the state of owned objects on a regular timing. These are stored in a circular buffer on the owner and are serialized to all clients, where they are then deserialized into a local circular buffer. These numbered states allow for very smooth interpolation/extrapolation of states even with packet loss and maintain synchronicity between all objects that share an owner.

Networking consists primarily of transferring these state buffers from the owner to other clients.

Serialization writes to a unified bitpacked byte[] array

This allows for extremely high levels of data compression, as well as allowing components to employ inline serialization logic. Writes start with a single timing from the NetMaster immediately after each Simulation (PostPhysX), and a single byte[] array is passed to all of the state producing NetObjects, which in turn pass it to every child SyncObject and PackObject (Synvars) component for serialization in a deterministic order. This produces one highly compressed and tightly ordered byte[], rather than having scattered objects calling Write() methods adhoc, and without boxing/unboxing.

Example of bitbacking : PhotonAnimatorView(PUN2) vs SyncAnimator (SNS)

	Uncompressed(PUN2)	Network Packed(SNS)
Normalized Floats	32 bits	2-16 bits
Other Floats	32 bits	2-16 bits
State/Trigger Hashes	32 bits	1-8 bits (based on #)
Bools/Trigger Params	8 bits	1 bit

Syncvars tied to the core simulation tick timing system

PackObjects allow for fields to be tagged with attributes that will automatically generate code that syncs values from Owner to All on the same deferred timings as NetObjects/SyncObjects. This means you can simply synchronize fields rather than having to write your own SyncObjects. Syncvars have built-in compression options, and advanced users can write their own compression methods as well.

Keyframe based eventual consistency

The SNS system primarily uses **Unreliable UDP with Keyframe and Delta frames**, similar to how video works. Rather than using Acks to negotiate eventual consistency between clients and a server (which would become very cumbersome in a relay environment where every client would need to negotiate reliability with every other client), **SNS uses keyframes to achieve eventual consistency**. Lost or late packets that produce disagreement in state between the owner and other clients will eventually resolve when forced updates (keyframes or changes) arrive.

This replaces the usual Message <-> Ack paradigm that is used in server-authority environments. It increases the overall amount of data (objects that aren't changing still send out regular keyframes) in exchange for MUCH greater simplicity.

Getting Started

Before trying to use this library, it is recommended that you first run the tutorial to get familiar with the components and workflow of SNS.

[Getting Started Tutorial](#)

Usage

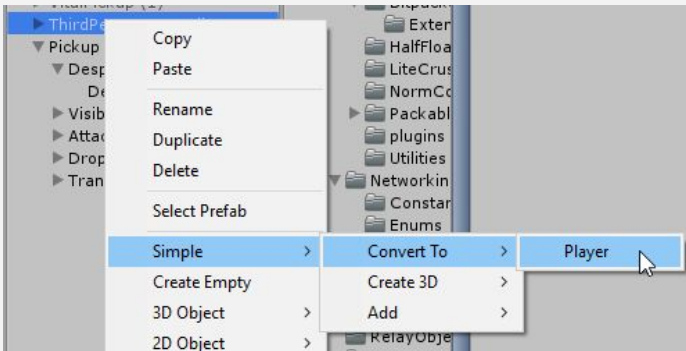
This library contains replacements for PhotonTransformView and PhotonAnimatorView as well as a [collection of drop in components](#) that handle common objects typical to networking, such as syncing vitals, items, states, etc.

Be sure to include the following on any .cs files referencing these components:


```
using emotitron.Networking;
```

Assistants

Assists are the menu items that automate the tasks of creating/converting to networked objects, and they are used almost entirely for this demo. They are basically one-click wizards, as they have no interface. They will attempt to perform a task on the gameobject that is selected in Unity, or will create a new scene object if applicable.



Component Instructions

Most components in the SNS library have a custom header that may contain an **Instructions** foldout, and/or may have a link to the documentation (seen as ). The goal is to make most of the base components just work well enough by default that heavy reading of this documentation isn't needed. However, some of the concepts and API components will require some reading due to the complexity of the systems involved.



Global System Objects

Simple Sync Settings

Project global settings that are stored in a ScriptableObject. You can find these in the Window menu. Settings can be found in the menu:

Window > Photon Unity Networking > Simple Sync Settings

Show GUI Headers - Toggle the large graphic headers in the inspector for SNS components.

Code Generation:

Enable Codegen - When enabled, the assembly will be scanned for classes with the [PackObject] attribute, and generate the syncvar needed for them to work. Disabling this will leave existing Codegen in place, but new codegen will not be created until it is re-enabled.

Delete Bad Code - If a compiler error occurs in any of the generated extension scripts, that script will get deleted automatically. You should always leave this option enabled.

Ring Buffer:

Frame Count - The number of frames allocated to the ring buffer. more frames, the greater the memory footprint, but the longer the buffer. Values that result in a buffer total of 1-2 seconds is recommended.

Send Every X Tick - SimpleSync uses FixedUpdate as its primary tick, but can produce a network rate that is a subdivision of the fixed time rate. For example the default Unity physics fixedDeltaTime is .02 secs (50 ticks per sec). Setting **Send Every X** to 3 would result in a net rate of .06 (16.7 ticks per second) which is $\frac{1}{3}$ of the physics rate, resulting in $\frac{1}{3}$ the traffic.

Buffer Correction

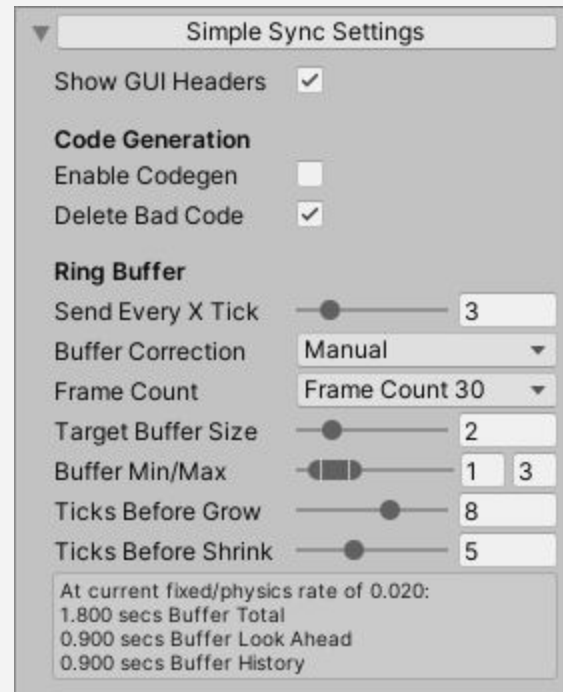
Manual - Set values by hand.

Auto - Values are set based on fixedDeltaTime and SendEveryX values.

Target Buffer Size - The ideal number of frames that will be on the buffer at consumption time (OnSnapshot). The closer to zero the lower the latency. However too low and the buffer will become more prone to underrun.

Buffer Min/Max - The values at which the buffer will be considered in need of a size adjustment.

Ticks Before Grow/Shrink - The number of ticks where the buffer exceeds the Min/Max value before an adjustment is made. Adjustments are made by either repeating a frame, or consuming multiple frames for a tick. The greater these values the less aggressively the buffer will try to resize itself. Corrections will create a visible hitch, so it is best to avoid them.

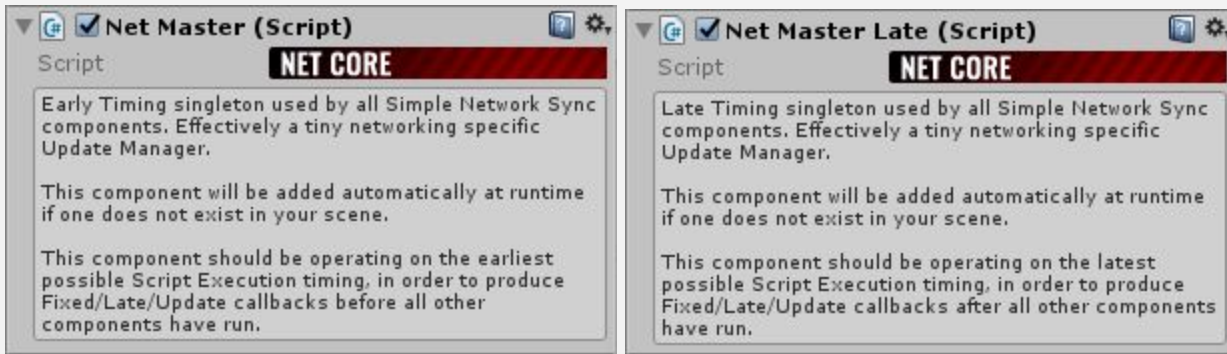


code

The

tick,

NetMaster & NetMasterLate

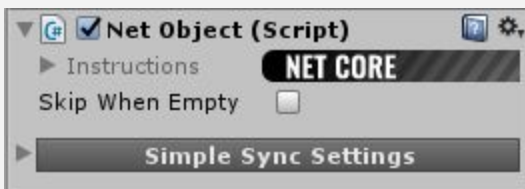


Two singletons are created at runtime if none exist in the scene. These produce timing callbacks used by SNS NetObjects and PackObjects and are the starting point for all activity.

NetObject

This component acts as the network id address and traffic cop of networked game objects, and is automatically added whenever a syncObject component is added to an object. This object collects and manages all of the callback interfaces of all ISyncObject components on a gameobject, as well as any Components with the PackObject Attribute. NetObject responds to timings generated by NetMaster, and passes them through to its **SyncObjects** and **PackObject** components via these callbacks.

NetObject requires a PhotonView, as it uses the ViewID, IsMine and various network callbacks of the PhotonView.

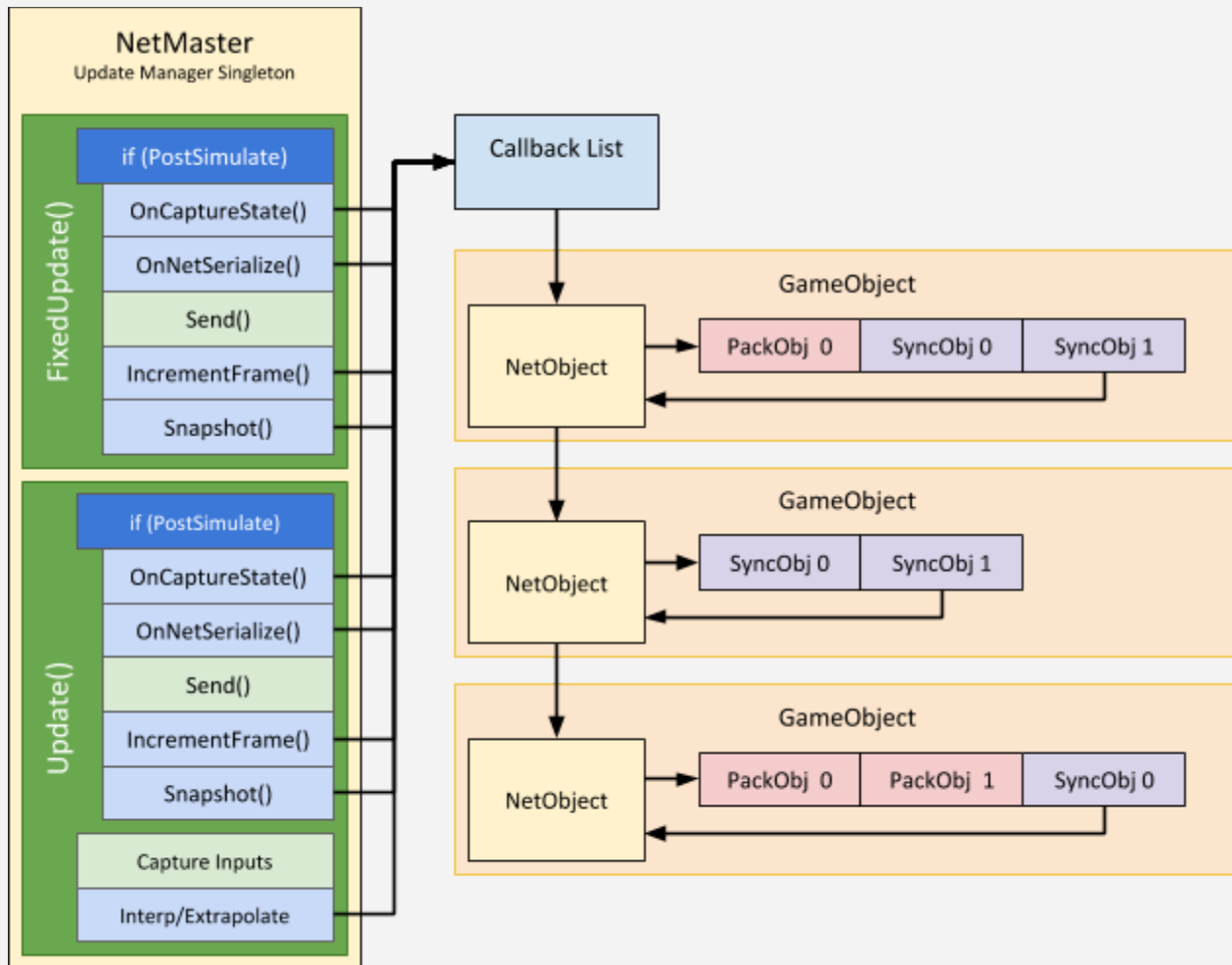


Skip When Empty: EXPERIMENTAL

This instructs the NetObject to not send at all, not even a heartbeat. This saves about 3 bytes per tick per object by removing heartbeat data, but the side effect is some less than desirable extrapolation. This is included for edge cases where a

scene may have a LOT of idle objects and some hitchy wake up behavior is worth the savings. Be sure to set your keyframe rates on children sync objects, as keyframes will force a send. Keyframe = zero will never send a keyframe. Also be aware that PUN does not initialize objects, so until an object moves, any late joiners will not see a correct position.

Event Flow



Sync Components

The following are some of the premade components that can be dropped into a scene and should “just work” for most situations. All Sync Components derive from the **Sync Object** base class, or the further derived **SyncObjectTFrame** base class that adds state buffer handling.

SyncObject (ISyncObject)

This is the base class of all Sync components. This base class isn’t needed to tie into the entire SNS system, but this class does make a good starting point for making components.

Common Fields

Components like SyncTransform, SyncState, SyncAnimator etc all are derived from the SyncObject class. These objects self register with the callback timing system of NetMaster.

Shared fields among sync components:

Apply Order



It is recommended that you not change this.

The order in which components on a network object run, in order from lowest to highest. This is used to ensure things that regardless of component order, certain things will always happen first. Some Transform and Animator setups may prefer one or the other to happen first. By default Transform is first, as this produced the best results in my example scene, but it may not be the case with all Animator configurations.

When components have the same Apply Order value, then the order in the gameobject hierarchy is used.

Keyframe Rate



These components are one-directional broadcasts. As such it is possible to reduce data by using keyframes. When no changes occur to elements of synced objects, they will send a 1 bit false flag indicating no content, rather than sending the same data every update. This does mean that loss and late joining players (in the case of PUN2) may have out of date values for a short period. Keyframes ensure eventual consistency when using delta frames. The greater the value the more data savings, but also a greater risk of odd behavior when packet loss is encountered.

If bandwidth is less of a concern, set this value to 1, and every update will contain a complete compressed state.

SyncObject<TFrame>

Derived from SyncObject, this adds default handling for state buffers. The TFrame type must be a derived FrameBase class, where the fields of the state for the SyncObject can be defined.

SyncState Component

This component standardizes and synchronizes the handling of NetObject state information, such as:

- Visibility
- Attachment to other NetObject Mounts
- Owner Changes
- Despawn/Respawn



It is not a required component, but other components like SyncSpawnTimer may require it if they specifically are making use of the SyncState fields.

It communicates with SyncTransform to ensure that parent changes due to changes in parent Mounts don't break position/rotation sync.

The primary interaction for devs with this component are the methods:

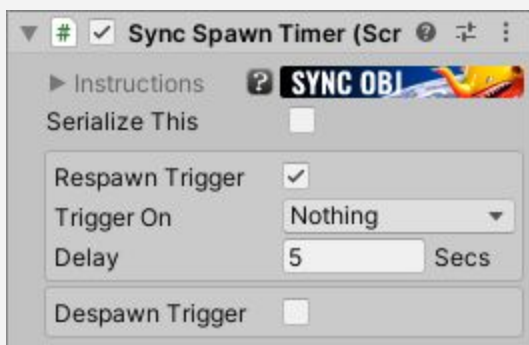
QueueStateChange()

Queues the state change and defers applying it until the next CaptureCurrentState timing. This is generally the recommended method, as it ensures better deterministic order.

ChangeState()

Immediately applies the state change locally, however it will not be captured and synced until the next net tick.

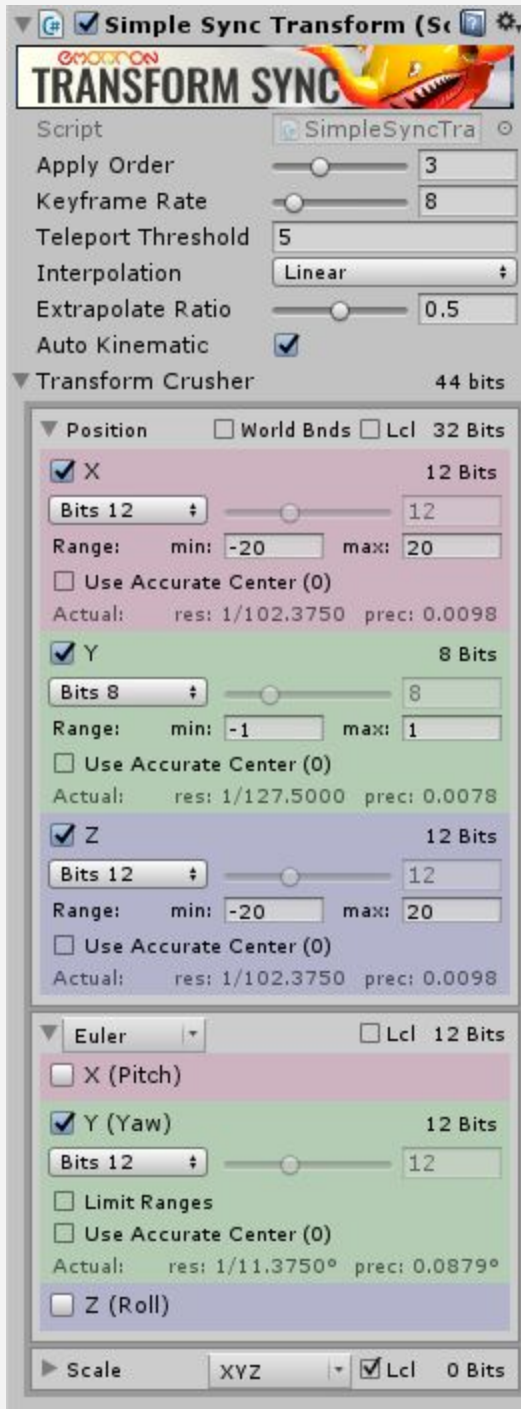
SyncSpawnTimer



This is a helper component for SyncState that responds to OnStateChange events to automatically trigger timers for Spawn/Respawn/Despawn.

SyncTransform Component

The replacement for PhotonTransformView, use this to sync the states of Transform TRS values.



Teleport Threshold

The distance in units between frame updates that will trigger a teleport. This exists just to have parity with other transform syncing tools. Teleporting can have different meanings in different games, so ideally you will want to code teleport handling yourself.

Interpolation

- **None** - Objects will snap to the networked states, without lerp.
- **Linear** - Basic Lerp/Slerp are used to interpolate between networked frames. This is likely the mode you want.
- **Catmull Rom** - (Experimental) A more sophisticated lerp that uses 3 points, producing a more naturally curved and accurate path than a basic lerp.

Extrapolate Ratio

Extrapolation occurs when the tick advances, but no frame info has arrived yet on clients for this object. The synced object now has to guess what the next frame's values are.

Extrapolation uses the last two values to extrapolate the new frame. The Ratio is the t value used by LerpUnclamped, and acts as a dampener of extrapolation. So for sequential missing frames the extrapolation gets reduced on a curve.

0 = no extrapolation - object will not move on empty buffer.

.5 = evenly dampened - object will lerp less with each tick.

1 = full extrapolation - no damping, will extrapolate until a frame arrives or object is destroyed (disconnect).

Auto Kinematic

A best guess is made for the handling of owner/server/others on how to set the rigidbody isKinematic. Generally all non-authority instances of the object will be set to isKinematic = true.

Transform Crusher

See [TransformCrusher](#)

TRS data (Position, Rotation, Scale) are compressed using my Transform Crusher library, which has its own documentation.

More information about the [Element Crusher](#) can be found in the [Transform Crusher documentation](#).

IOnteleport.OnTeleport()

SyncTransform uses the IOnteleport callback interface that allows components like SyncState to notify SyncTransform that an object has been teleported. You can call OnTeleport() manually when you want an object to Teleport.

Teleporting disables interpolation and tweening momentarily to allow for objects to be moved great distances in one tick without lerp. In order to do this BEFORE you move the object to its new position call:

```
GetComponent<SimpleSyncTransform>().OnTeleport();
```

Or if you already have a cached reference to the transform sync component:

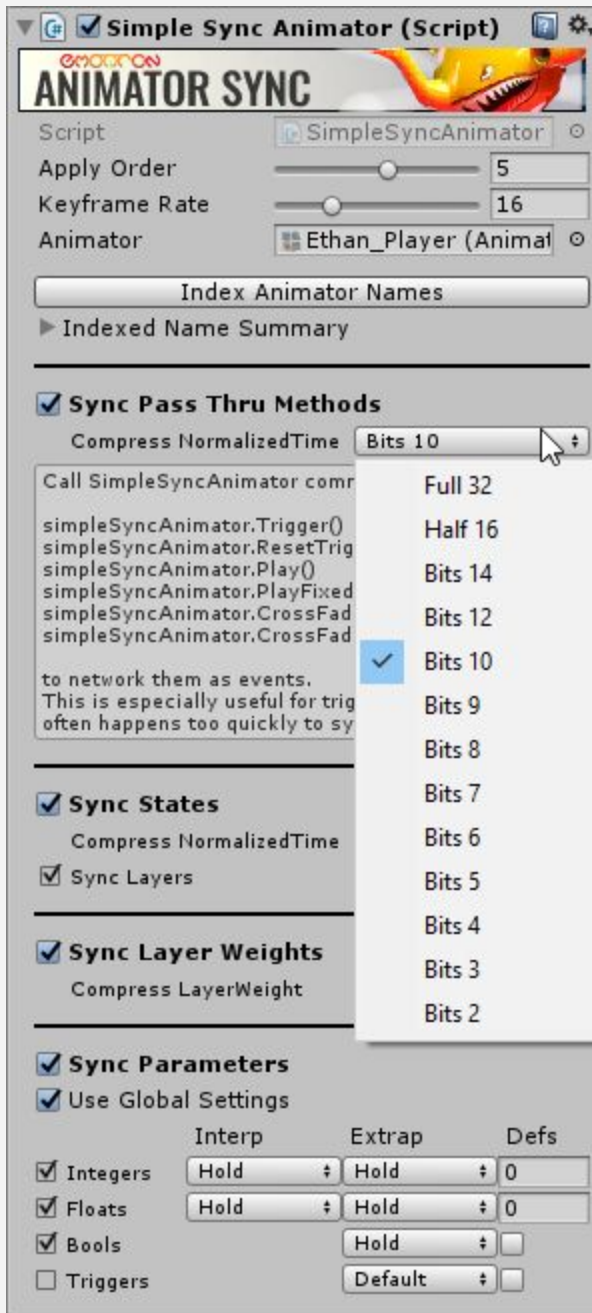
```
cachedSyncTransform.OnTeleport();
```

SyncTransform responds to this by capturing the current transform state, and flags the next outgoing state as being a teleport. That outgoing frame will contain two transform states, one for the value before teleport, and one after. Having both values allows interpolation to work correctly without causing objects to freeze for a tick when teleports occur.

Call this BEFORE you move the object that is being teleported.

SyncAnimator Component

The primary PhotonAnimatorView replacement component.



Index Animator Names

One of the primary compression methods of this component is the indexing of all State and Trigger names, so that rather than sending 32 bit hash values for states and ticks every update, MUCH smaller (1-5 bits) indexes are sent instead.

The indexing happens automatically with a lot of hackery behind the scenes, but it doesn't always fire (since it can't be forced during the build process). You may see warnings in the log asking you to press this button. It never hurts to press it, especially after making changes to your Animator Controller.

If at runtime an index isn't found, it will just fall back to sending the full 32bit hash. Things will still work, but your network usage will increase.

Sync Pass Thru Methods

If you call SetTrigger(), Play(), CrossFadeInFixedTime() and such through this component, it will network those calls, and pass along the command locally to the Animator. This often can give the most in sync result, for a relatively low network cost (less than a byte typically).

Sync States

This syncs the "snapshot" state of the animator, and enabling this can sometimes be useful for ensuring agreement with clients. Typically if you are making use of Passthrough methods and parameters, it is not needed.

Sync Layers

If you are using layers and want them synced, turn this on. Disable if you are not to save some data.

Sync Layer Weights

If you are animating layer weights, enable this. If you are not animating them however, be sure to turn it off to conserve data and processing time.

Sync Parameters

You can globally or selectively select which parameters to sync. Interpolation, Extrapolation and default values can be set, but generally you will just want to leave the defaults.

Compress NormalizedTime/LayerWeights

Anywhere normalized values are used (normalizedTime and layerWeights) an option is given for how much to compress these. Half 16 is the failsafe. Lower values can be tried to see how low you can get before seeing any artifacts.

SyncAnimator Advanced Parameter Settings

(Not for the average user)

Per parameter settings are not pretty to look at, but they do let you get into the specifics of which parameters are synced, and gives the ability to indicate how they are interpolated, extrapolated, what their default values are (when needed for the interp/extrap settings).

Basic compression settings options for Ints and Floats appear on the second row when available. They include:

Int Compression:

Pack Signed - This is useful if your values typically stay reasonably close to zero, but you don't actually know how high or low they can go. This is the default codec as it will always just work.

Pack Unsigned - if you know the numbers will only be positive, this is better than PackSigned as it skips the ZigZag operation on the sign bit.

Range - If you know the range your int will stay in, this will give the best compression. Just indicate the min and max values, and the compressor will handle the rest. Any values outside of the given range will be clamped.

Float Compression:

Full 32 - No compression. Please don't use this.

Half 16 - Greatly reduced accuracy from float32, but typically will be good enough for most things. This is the default float codec.

Range - If you know the range your float will stay in, this will give the best compression. Just indicate the min and max values, and the number of bits you want to limit compression to. Any values outside of the given range will be clamped. Play with the BitsX setting to find the lowest number that gives you acceptable results. I may add some context info in the future to indicate the precision.

Accurate Center toggle - because of the nature of bits, there will always be an even number of quantized steps to compression. Enabling Accurate Center reduces the compressed range by 1 creating an odd number of steps... thus allowing a value exactly between min and max to be reproducible after lossy compression.

By default Triggers are not synced as parameters, as they should be synced using the PassThrough methods. They are only included for completeness.

The screenshot shows the 'Sync Parameters' window in SyncAnimator. It has a title bar and a main area with a tree view on the left and a settings panel on the right. The tree view lists parameters: 'walking' (B), 'speed' (F), 'jump' (T), 'isJumping' (B), 'turnLeft' (T), 'upperBodyRun' (T), 'upperBodyIdle' (T), and 'blender' (F). The settings panel shows the configuration for the selected parameter. For 'walking', 'Interp' is 'Hold', 'Extrap' is 'Hold', and 'Def' is '0'. For 'speed', 'Interp' is 'Hold', 'Extrap' is 'Hold', and 'Def' is '0'. For 'jump', 'Interp' is 'Hold', 'Extrap' is 'Hold', and 'Def' is '0'. For 'isJumping', 'Interp' is 'Hold', 'Extrap' is 'Hold', and 'Def' is '0'. For 'turnLeft', 'Interp' is 'Hold', 'Extrap' is 'Hold', and 'Def' is '0'. For 'upperBodyRun', 'Interp' is 'Hold', 'Extrap' is 'Hold', and 'Def' is '0'. For 'upperBodyIdle', 'Interp' is 'Hold', 'Extrap' is 'Hold', and 'Def' is '0'. For 'blender', 'Interp' is 'Hold', 'Extrap' is 'Hold', and 'Def' is '0'. The 'Sync Parameters' checkbox is checked. The 'Use Global Settings' checkbox is unchecked. The 'Adv. Parameter Settings' dropdown is expanded.

PackObjects (SyncVars)

[PackObjectAttribute] allows you to tag a class or struct as having [PackAttribute]s that you want automatically synchronized. These are essentially the same as SyncVars in other platforms, with the primary differences being that fields:

- 1) **Capture/Serialize/Deserialize/Apply on the same simulation-based tick timings as all of SNS.**
This ensures that changes made to these objects (ideally change them in the OnPreSimulate timing segment) happen deterministically in execution order on the owner and all clients, and use the same deferred timing mechanisms as SyncObjects.
- 2) **Can be defined with PackAttribute derived classes to serialize and deserialize with custom compression methods**, allowing for bitpacking and other compression, frameId based keyframes/delta frames, auto-interpolation/extraction, etc.

Example Usage:

```
[PackObject]
public class TestPackObj : NetComponent
{
    [PackHalfFloat(
        callback = "RotationHook",
        callbackSetValue = CallbackSetValue.BeforeCallback,
        Interpolate = true,
        keyRate = KeyRate.Every
    )]
    public float rotation = 45;

    // Hooks are methods associated with a syncvar. They are called
    // whenever the networked value changes.
    public void RotationHook(float newrot, float oldrot)
    {
        // Do something here with the newrot value
        // This gets called on all clients when the networked
        // value changes.
    }

    // This is the default PackAttribute, which uses simple/no
    // compression for the types it supports.
    [Pack]
    public Vector3 v3;

    // PackRangedInt bitpacks ints by dropping unused upper bits.
    // Supply the min and max allowed values and it will come up
    // with the best compression values automatically.
    [PackRangedInt(-1, 2)]
    public int intoroboto;
}
```

PackObjectAttribute

[**PackObject**] tells the Code Generation system that a class/struct needs to have serialization extensions created for it. The Code Generation engine will find all **PackAttributes** on fields of the Class/Struct and create automated synchronization extensions that tie into the NetMaster and NetObject automatically.

PackAttribute

The **PackAttribute** (and derived attributes) tell the Code Generation which fields of a PackObject should be included in serialization/synchronization.

syncAs

Indicates how the Pack syncvar should be treated.

- **SyncAs.State** maintains the value on the owner until changed.
- **SyncAs.Trigger** indicates that when the value is captured/sent by the owner, it should be reset to default with a new() operation.

keyRate

How often a keyframe is sent. Keyframes send a complete update regardless of whether a value has changed or not. In between these updates, data will only send if a value has changed.

snapshotCallback

Name of the method inside of the PackObject that will be called when a new frame is consumed. This is called right after the previous target becomes the snap value, and a new target is consumed. The snap and targ values are the new interpolation start/end values if interpolation is enabled. This fires right before the new Snapshot value is applied, and before applyCallback is called. The syncvar value will still be the previous value.

applyCallback

Name of the function inside of PackObject that will be called when values have been remotely changed, and the update with the change is being locally applied. The **setValueTiming** indicates if the snapshot value is applied to the syncvar before or after this callback, or if it is set automatically at all.

setValueTiming

Indicates when the snapshot value should be applied to the Pack syncvar in relation to the applyCallback.

Interpolation

Indicates if the Pack syncvar value should be lerped every update automatically when the value changes.

Subsystems

The Simple Network Sync API contains a handful of sub libraries that deal with common networking challenges. Most of these have basic implementations that may work well enough for many projects. However these are all designed with the intention of acting as guides for how to properly interact with the API and make your own.

- [Contact System](#)
 - [Inventory System](#)
 - [Health and Vitals System](#)
- [HitGroups System](#)

Contact System

IContactTrigger / ContactTrigger

This interface / component is a universal starting point for networked (or non-networked) events. It sends an `IOnTriggerEvent.OnTriggerEvent()` callback to any components utilizing that interface. Triggering events may be physics collisions/triggers or may be hits from projectiles and raycasts.

IOnContactEvent

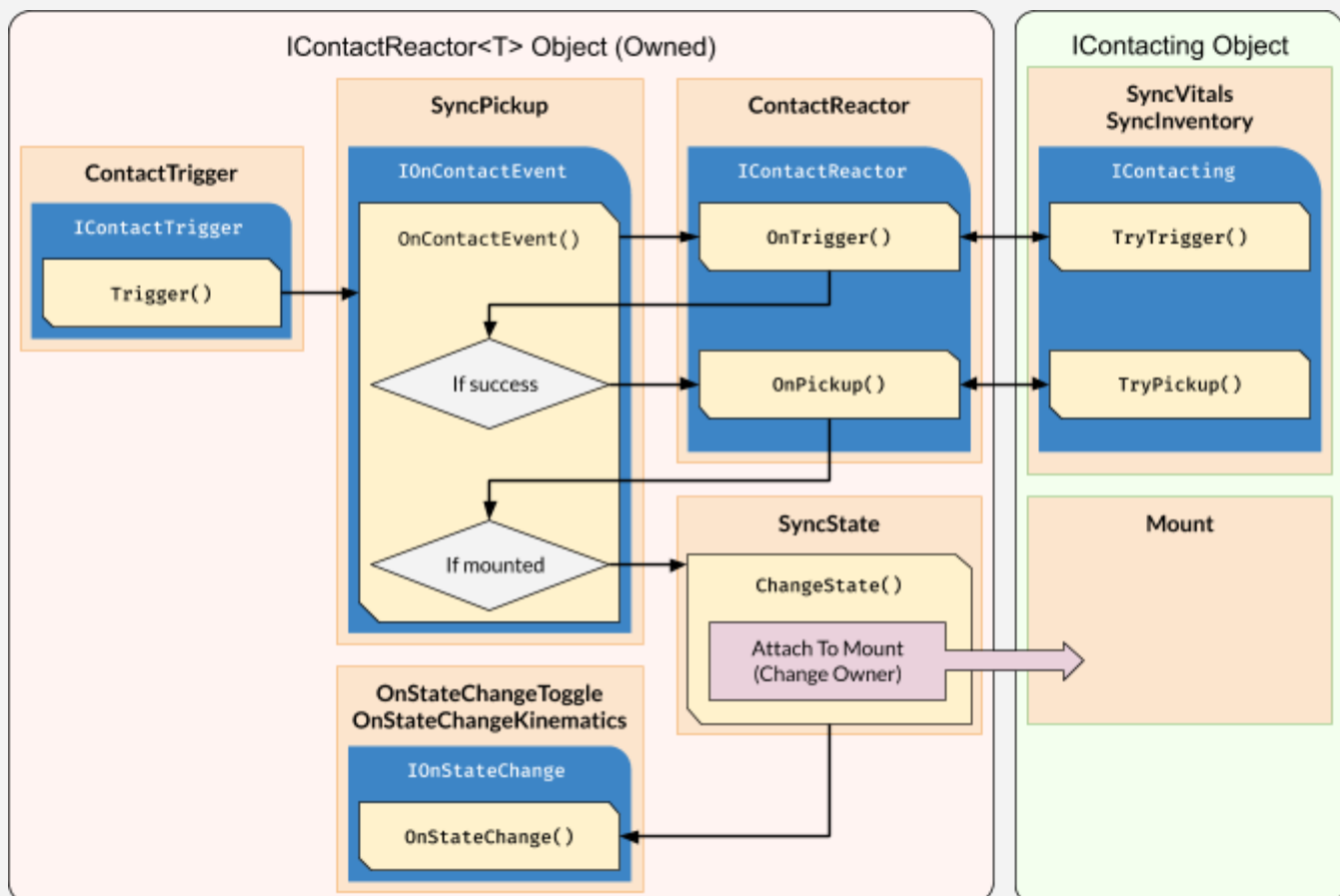
This is the callback interface that listens for ContactTrigger events.

IContacting

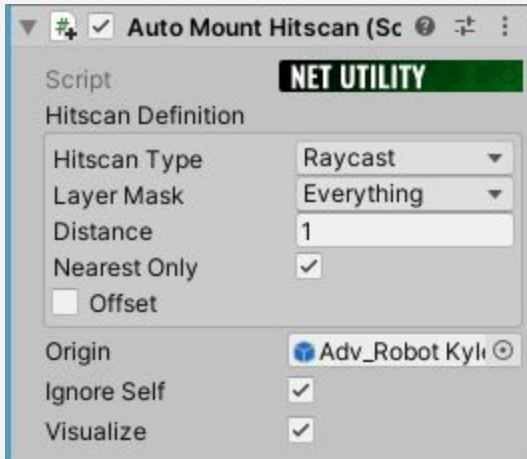
This is the root interface for component systems that defines components that can produce Trigger and Pickup interactions actions with other NetObjects.

IContactReactor<IContacting>

Defines a component that can be triggered by contact with an IContacting of the defined type. This is the flow of the interaction between pickup items and the Vitals and Inventory systems.



AutoMountHitscan



This component produces a hitscan every tick, and will attach its parent NetObject/SyncState to the mounts of any other NetObject the hitscan finds. The primary use for this is for automatically reparenting objects to platforms/vehicles or any other moving object that motion should be relative to.

By mounting objects to moving parents, transform syncs can use local rather than global space. This will ensure that if the objects have different owners, they will still remain firmly connected and not suffer from “floating behind” due to latency differences.

Inventory System

IIInventory<T> / BasicInventory

These base class and interface for the Inventory System that allows for the creation of Inventory components that automatically work with the rest of the SNS code base. A basic implementation is included as the **BasicInventory** component.

T is the struct that is used to define the capacity of the Inventory. The **BasicInventory** component uses a Vector3 as T, and is provided to show how to extend the base to create your own inventory system using your own inventory struct and logic.

IIInventoryable<T> / InventoryContactReactor

The base class and interface for the Inventory System that allows an object to be considered an item that can interact with **IIInventory**.

To successfully add an **IIInventoryable** item to an **Inventory** the following must be true:

- The **SyncPickup** must receive and **OnContactEvent()** (typically comes from the **ContactTrigger**)
- The Pickup object must have an **IIInventoryable** component (such as **InventoryContactReactors**) and the colliding object must have an **IIInventory** (such as **BasicInventory**) component.
- The T types must match.
- The Pickup object's **SyncState** must define a valid **Mount** that matches an **IIInventory's** **DefaultMount**.
- **HitGroupMasks** must match.

Vitals System

IVitalsComponent / SyncVitals Component



Sync Vitals (Script)

SYNC VITALS

Instructions ? SyncVitals

Keyframe Rate: 1

Type: Health Start Value: 125 Max Value: 125 7 Bits

Add Vital

Type: Armor Start Value: 50 Max Value: 125 7 Bits

Add Vital

Type: Shield Start Value: 100 Max Value: 250 8 Bits

Add Vital

Use Hit Groups: ☒

Valid Hit Groups: [0000]

Armor ☐ Critical ☐ Head ☐ Weapon ☐

Default Mounting: Root

Auto Despawn: ☒ Reset On Spawn: ☒

A generic component for handling syncing of things like health, energy, shields, power, etc. This component uses the [Vitals Subsystem](#) as well as the [HitGroups Subsystem](#).

Type: The name/id of this Vital. Internally this is backed by an enum of the known types. Custom types turn the hand coded name into a hash.

Start Value: The value that this vital will return to when reset or initialized.

Full Value: The normal max value. If this value is changed at runtime it needs to stay below Max Value or it will break serialization. Value changes the current value can be flagged with **Overload**, which can allow current value to be greater than Full Value (but still less than Max Value).

Max Value: The highest possible value. This value is used to determine the bits needed for serialization. As such this value should not be changed at runtime.

Absorption: The percentage of damage this vital will take. The remainder is applied to the next vital up the chain.

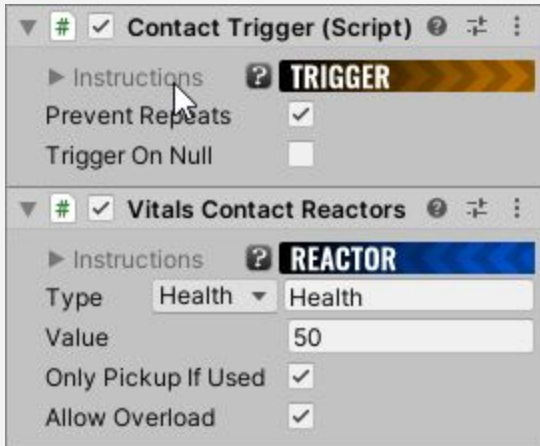
Regen Delay/Rate: How many seconds after damage occurs before this vital resumes regeneration. Rate is how much value is added per second.

Decay Delay/Rate: How many seconds after **overload** (being given a value greater than Full Value) before this vital will start to decay in value back down to full value. Rate is how much value will drop per second.

Auto Despawn: When the root vital reaches 0, this will fire a OnDespawn event with the SyncState component.

Reset On Spawn: When SyncState fires a OnChangeState callback flagged as a respawn, all vital values will be returned to the start value.

IContactReactor<IVitalsComponent> / VitalsContactReactors



The VitalsContactReactors component handles the work of converting ContactEvents (from ContactTrigger or OnNetHitContact components) into OnTrigger and OnPickup events.

HitGroups System

A Layer/Group mask system that avoids the shortcomings of Unity's tag and layers. Unity Tags are not proper bitmasks and are string based, and Unity Layers is tied to physics, making it not ideal for networking.

The system itself is actually very simple. The HitGroupSettings singleton stores an index of group names, which translate into a bitmask. HitGroupAssign components can be added to GameObjects with colliders. You can then call `GetComponent<HitGroupAssign>().Mask` on collision/trigger/raycast results to determine if the hit collider belongs to a group of interest.

HitGroupAssign

This component at runtime is used to associate colliders of an object with groups. If **Apply To Children** is enabled, it will replicate the component to all child transforms that have colliders, unless they already have their own **HitGroupAssign**.



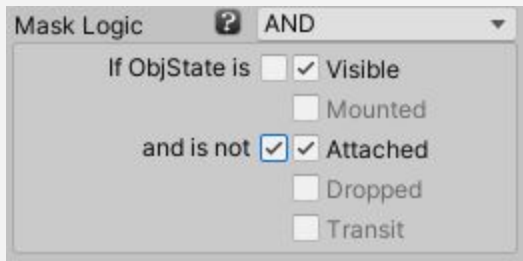
Hit GroupSettings

This settings singleton allows you to name/add/remove groups from the mask system.



Utilities

MaskLogic



The image shows a 'Mask Logic' configuration window. At the top, there is a title bar with 'Mask Logic' and a help icon. Below the title bar is a dropdown menu currently set to 'AND'. The main area contains two rows of configuration options. The first row is labeled 'If ObjState is' and has three checkboxes: 'Visible' (checked), 'Mounted' (unchecked), and 'Dropped' (unchecked). The second row is labeled 'and is not' and has three checkboxes: 'Attached' (checked), 'Dropped' (unchecked), and 'Transit' (unchecked).

This drawer is used by components to create mask tests with slightly more complex combinations than a basic mask compare.

The logic dropdown selects if the test applied is a EQUALS, ADD, or OR type test. The mask produces a bool result from: `Evaluate(int mask)`.