Simple Network Sync Documentation

Simple Sync Animator
Simple Sync Transform

Current version of this document online here

Contact the author at: davincarten@gmail.com

I am also regularly in #Unity-Dev on Discord as emotitron if you have questions.
https://discord.gg/tGDUxp9

This is still version 1.x.
As such, there are bug fixes and improvements regularly being added.

Be sure to check for updates!
And don't hesitate to contact me.

Unity 2019+ users:
Be sure a network library is installed before installing Simple Network Sync. UNet is no longer included, and if there is no library (PUN2, Mirror, MLAPI or manually added UNet) you **will** get a wall of errors.

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work until the first version of this has finished being updated as use cases come up, as it will quickly become outdated and wrong.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.

Concept

My aim is to create “just works” drag and drop networking components that bring better practices of tick-based simulation syncing and advanced bitpacking to areas of networking most new developers have the most trouble. This component, like the ones it is designed to replace uses **snapshot interpolation** for replication.

The first two components are syncs for Transforms and Animators.

Unlike the built-in UNET NetworkTransform, NetworkAnimator, PhotonTransformView, PhotonAnimatorView as well as other third party transform syncs, this library uses a ring buffer with numbered frames for each net entity (NetworkIdentity/PhotonView). This produces VERY smooth and stable syncs even in adverse network conditions, and also establishes a base for more advanced networking concepts later, like rewind and determinism.

All SimpleNetworkSync components on a networked gameobject serialize into a single byte[] using bitpacking, to give the highest levels of compression possible.

Animator Compression:

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

SendEveryX value reduces the overall tick rate to a set fraction of the FixedUpdate rate.
Keyframe Rate allows for the use of delta frames, reducing unchanged data elements to 1 bit.

Note about FixedUpdate vs Update timing:

This component set is tuned for games that simulate/move/animate using timings based around **FixedUpdate()**. It will 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. I always recommend in Unity doing all game logic in fixed, and only using Update() for interpolation/extrapolation when doing networking - rather than applying changes and game logic in Update(). This however is a much larger topic I won’t cover here.

Usage

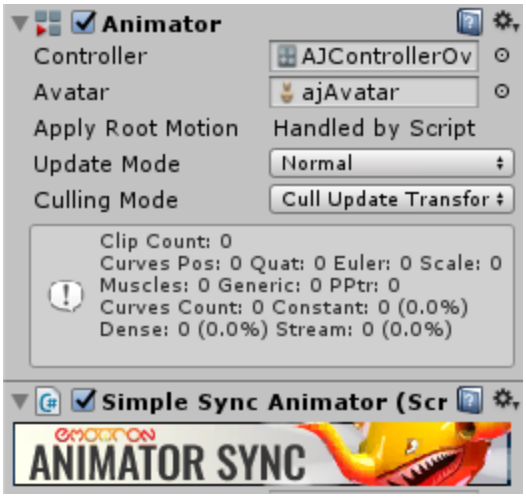
These components are meant as replacements for NetworkTransform, NetworkAnimator, PhotonTransformView and PhotonAnimatorView. As such, you must do the same ownership checks as you do for those. Specifically **be sure to test hasAuthority (Unet/Mirror) and IsMine (PUN2) in your controllers, and only apply controllers if these are true.** Otherwise, you will be trying to apply inputs to all objects on every connection, even the ones that player does not own.

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

```
using emotitron.Networking;
```

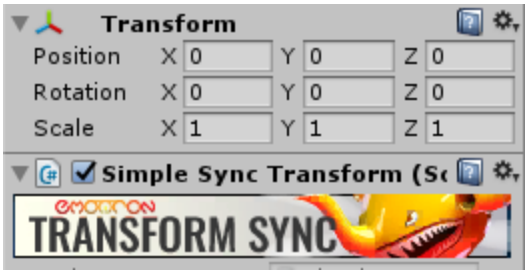
Simple Sync Animator

Place on any GameObject that has an Animator component you would like to sync.



Simple Sync Transform

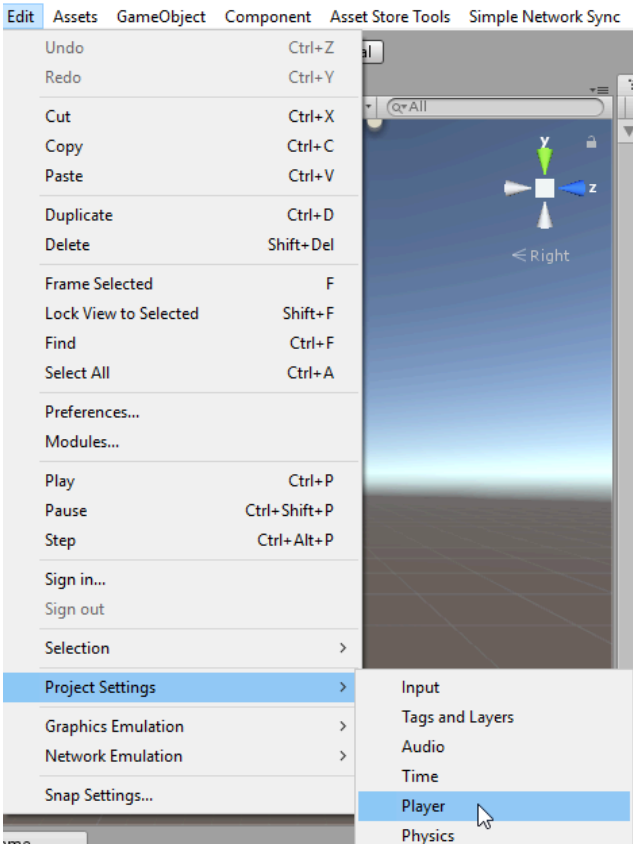
Place on any GameObject root or child that needs its Transform to be synced.



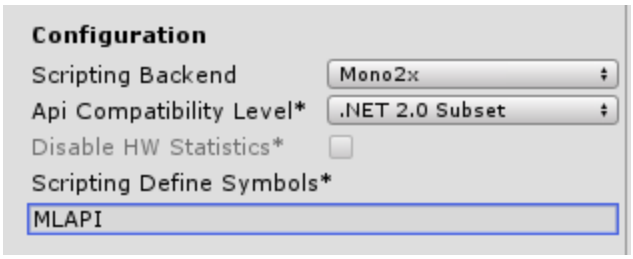
MLAPI Setup

For MLAPI, a define needs to be added to the project.
Go to **Edit >Project Settings > Player**,
and add **MLAPI** to the **Scripting Define Symbols**.

Open the player settings in the inspector:



Add MLAPI to the symbols.

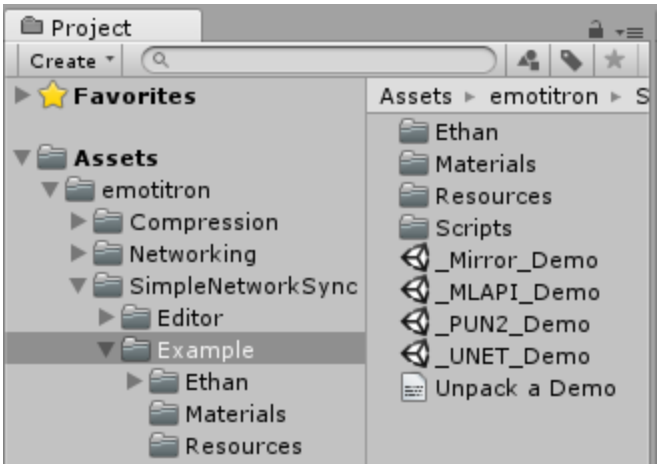


Demos

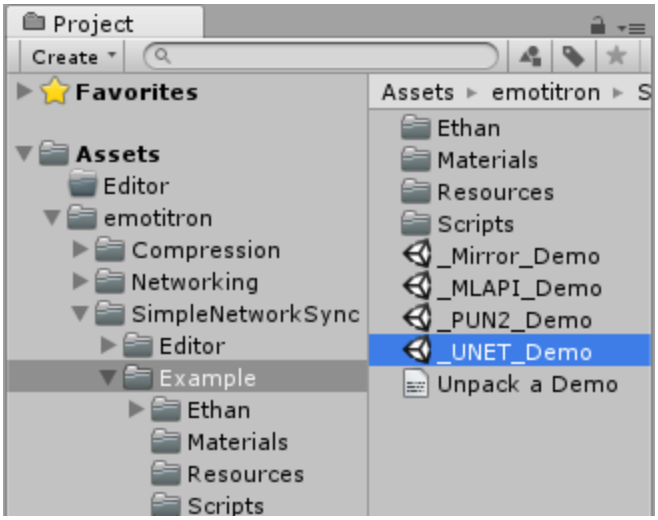
To support a wide range of Network Libraries (Unet/PUN2/MLAPI/Mirror) and the range of Unity versions (5.6 to 2019) - it is not possible to keep demo scenes in the project, as they would all suffer from missing components for the unused libraries.

To this end the demos are all Asset Packages that need to be opened.

1) Extract the Demo for the Network Library you are using.

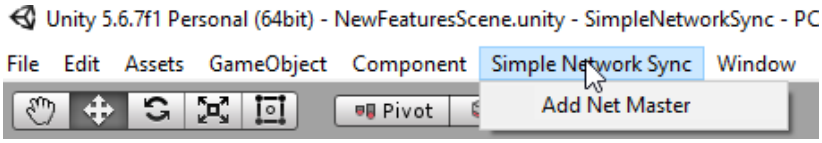


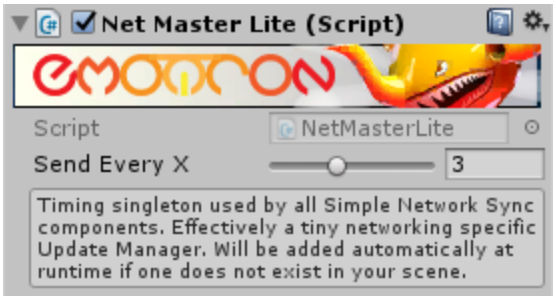
2) Open the newly created scene in the Example folder



Net Master

This singleton is created at runtime if none exists in the scene. It exists to ensure all networked objects using Simple Network Sync components fire on the same distinct timings, and is the primary traffic cop. If you would like to send at a lower rate than every FixedUpdate(), add Net Master to your scene and adjust the Send Every X value to reduce the send rate.

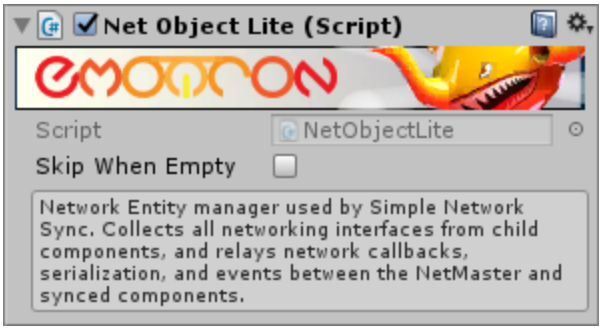




Send Every X Fixed - SimpleSync uses FixedUpdate as its primary tick, but can produce a network rate that is a subdivision of that rate. For example the default Unity physics fixedDeltaTime is .02 secs (50 ticks per sec). Setting [`Send Every X`] to 2 would result in a net rate of .04 (25 ticks per seconds) - ½ of the physics rate. This effectively cuts generated traffic in half.

Net Object

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 sync objects on a gameobject, and responds to the timings generated by NetMaster. This works in conjunction with PhotonView (PUN2), NetworkIdenity (UNet/Mirror) and NetworkedObject (MLAPI)



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.

Simple Network Sync - Common Fields

Shared fields among sync components.

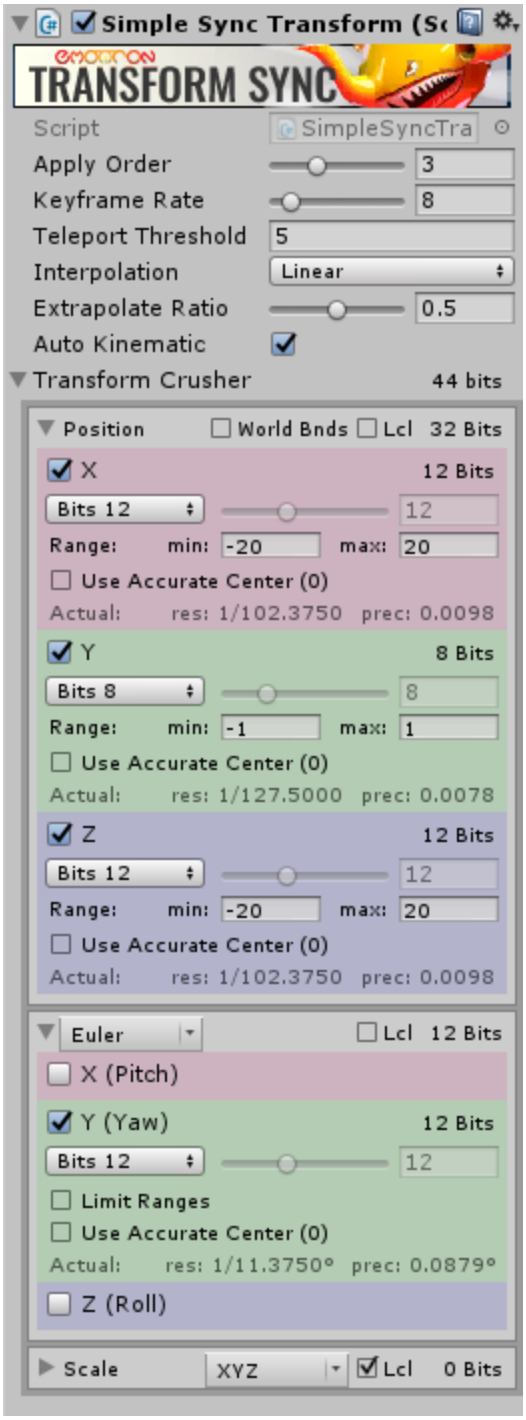
Apply Order
It is recommended that you not change this.
The order in which components on a network object run, in order from lowest to highest. This is used to ensure things that regardless of component order, certain things will always happen first. Some Transform and Animator setups may prefer one 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 full compressed state.

Simple Sync Transform

The primary NetworkTransform / PhotonTransformView replacement component is SimpleSyncTransform.



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 lerping.
- **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.

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 at in the Transform Crusher documentation.

Teleporting

Teleporting disables interpolation and tweening momentarily to allow for objects to be moved great distances. In order to do this move the object to its new position on the authority, then simply call:

```
GetComponent<SimpleSyncTransform>().HasTeleported = true
```

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

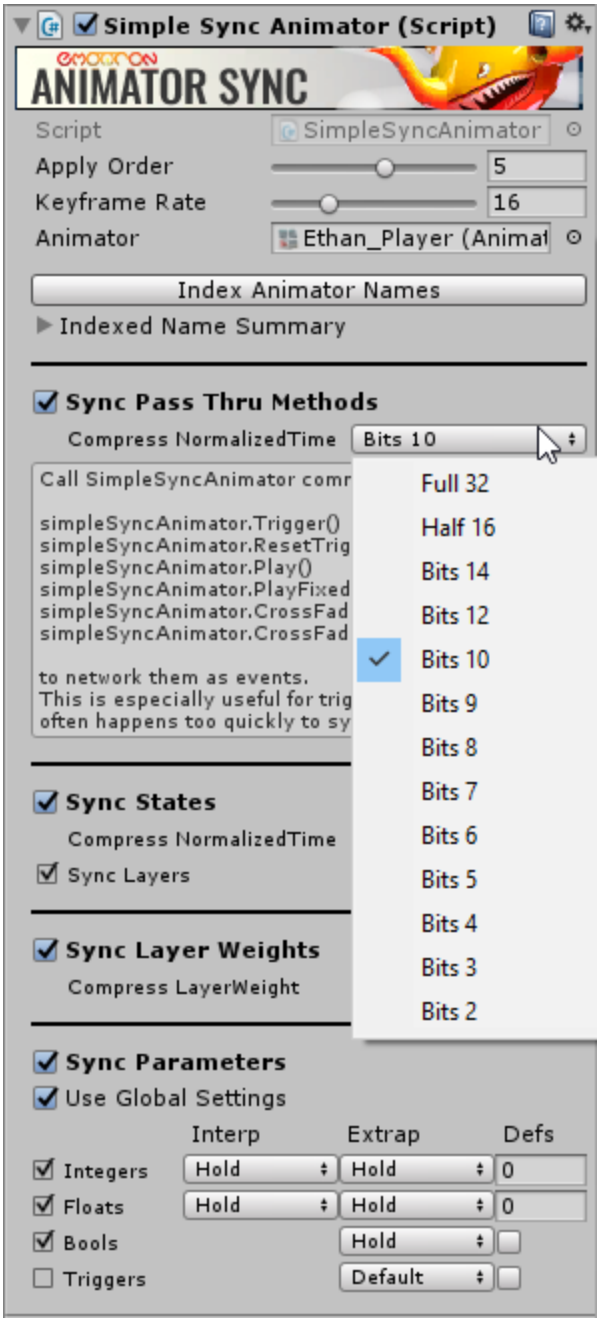
```
cachedSyncTransform.HasTeleported = true;
```

This flags the next outgoing state as being a teleport, and other clients will disable interpolation between this state and the following.

> *Currently this is a very simplistic teleport and there is no special handling for lost packets. If the state never arrives due to suffers packet loss or a very late arrival, the object will interpolate normally.*
>
> *In order to make this more loss tolerant, without making use of acks (which are not feasible in some of the networking environments), it would have other side effects. So right now I am keeping this simple until users come to me with situations where it fails them.*

Simple Sync Animator

The primary NetworkAnimator / 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).

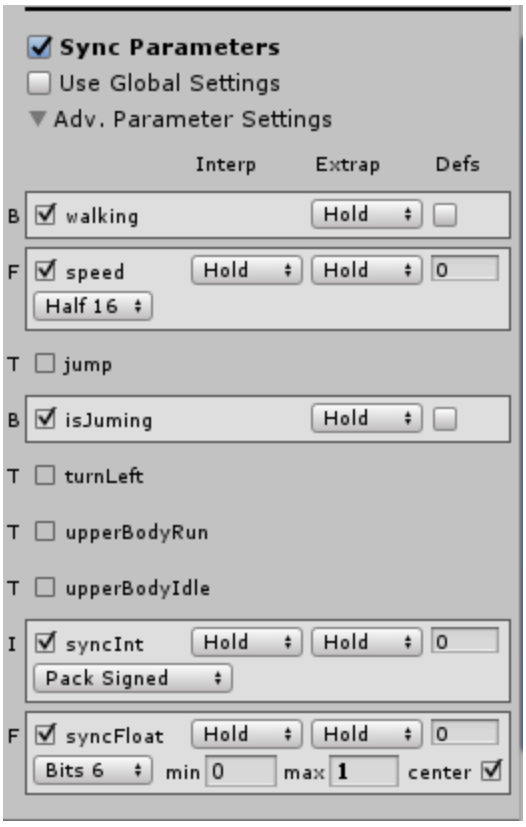
Sync States
This syncs the "snapshot" state of the animator, and enabling this can sometimes be useful for ensuring agreement with clients. Typically if you are making use of Passthrough methods and parameters, it is not needed.

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.



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

Advanced Parameter Settings
(Not for the average user)

Per parament 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).

Also recently a second row has been added with basic compression settings options for Ints and Floats.

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 rangle will be clamped.

Float Compression:

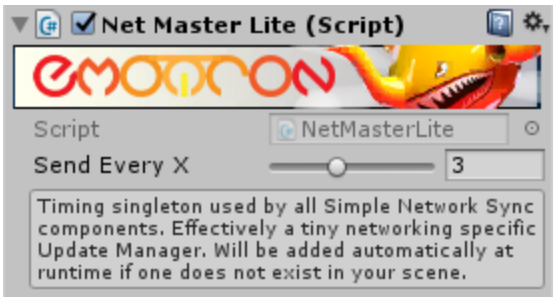
Full 32 - No compression. Please don't use this.

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 rangle 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.

Net Master

This singleton is created at runtime if none exists in the scene. It exists to ensure all networked objects using Simple Network Sync components fire on the same ticks, and is the primary traffic cop.



Send Every X Fixed - SimpleSync uses FixedUpdate as its primary tick, but can produce a network rate that is a subdivision of that rate. For example the default Unity physics fixedDeltaTime is .02 secs (50 ticks per sec). Setting [`Send Every X`] to 2 would result in a net rate of .04 (25 ticks per sec) - ½ of the physics rate. This effectively cuts generated traffic in half.