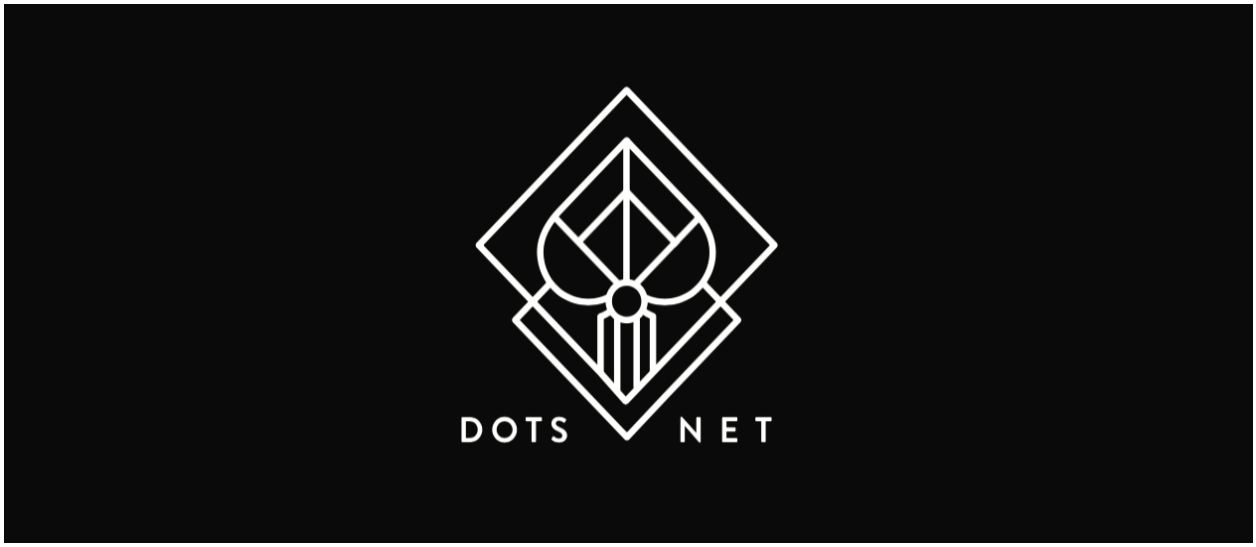
DOTSNET Documentation

High Performance Networking for Unity/ECS

(Asset Store) (Online Documentation)



Installation

Both Unity & DOTS/ECS Packages are undergoing rapid changes.

Please use **exactly** the recommended Unity version & Packages *(Window->Package Manager)*:

- **Unity:** 2019.3.10f
- **Burst:** 1.3.0
- **Entities:** 0.9.1
- **Unity.Physics** 0.3.2

Do not upgrade yourself. DOTS is in preview, newer versions always break something. Let me worry about this for you, only update when I update

To get started, import DOTSNET from the Asset Store, open one of the example scenes & press Play!

Recommended Reading

DOTS stands for Data Oriented Technology Stack.

ECS stands for Entity Component System.

You are probably used to creating GameObjects and adding MonoBehaviour components to them. DOTS/ECS is a completely different architecture that you need to learn first.

To get started, please read Unity's Entity Component System manual.

DOTSNET is about DOTS/ECS networking. It's not here to teach you DOTS/ECS, you are expected to learn it first.

Important: DOTS/ECS is a new technology that is in preview. Don't expect to build a complete game just yet. Play around with it, read the code and the docs, get a feeling. It takes time!

Platforms

DOTSNET currently works on Windows/Mac/Linux 64 bit systems because that's where the Apathy transport runs.

As more low level transports are developed, it will also support WebGL, Android, iOS, etc. later.

Changelog

V1.2 [in development]:

- SegmentReader made **5x** faster: 1mio x ReadFloat3 down from **2765ms** to **538ms**
- SegmentWriter made **3x** faster: 1mio x WriteFloat3 down from **922ms** to **345ms**

V1.1 [In development]:

- Unity.Physics support
- (Job)ComponentSystems converted to SystemBase
- Pong/Chat/Physics demos added
- Fixed clients not despawning other disconnected client's players
- Smaller fixes & improvements

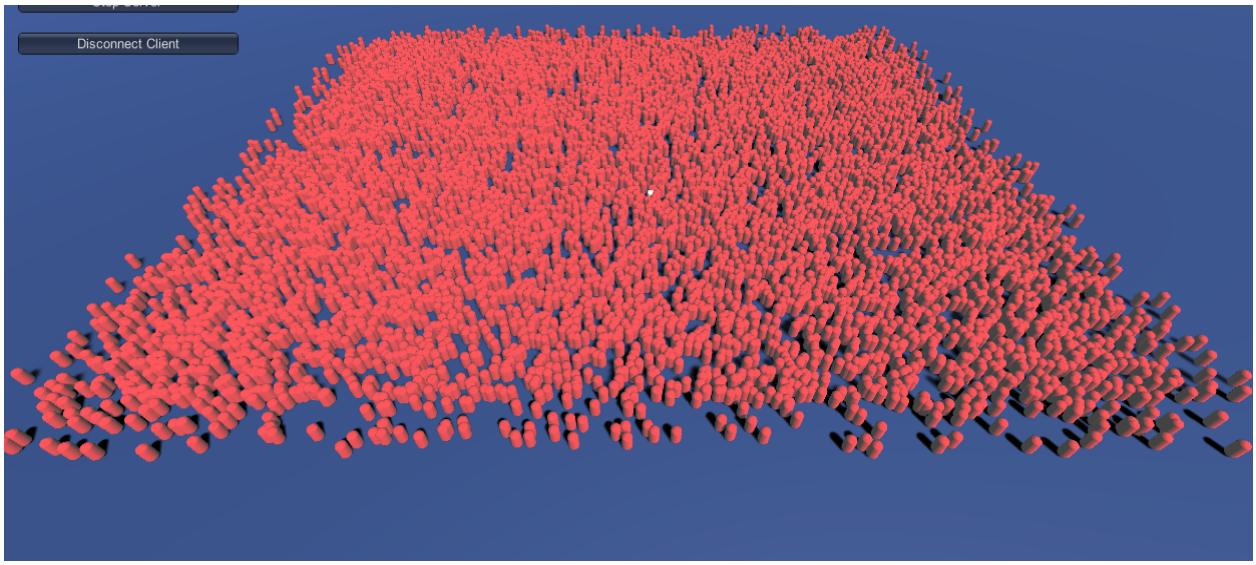
V1.0 [2020-05-X]:

- Initial release.

Example: 10k (Try this first!)

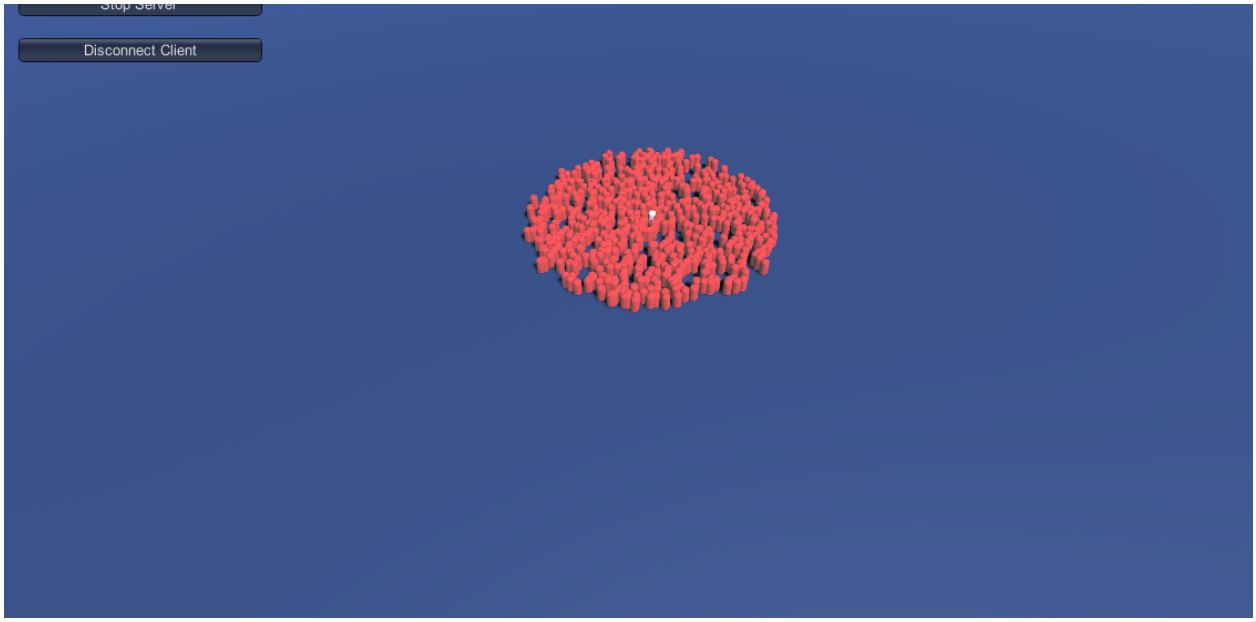
DOTSNET is extremely fast. In the 10k example, you move the player through a herd of 10,000 moving monsters.

While it's quite a lot of monsters, it's the easiest, most elegant example to get started!



If you run both the server and the client in the Unity Editor, then DOTSNET is handling **twice** the amount: 20,000 monsters. 10k for the server world, and 10k for the client world.

By default, the NetworkServer's Interest Management System has a **visibility radius** of 15:

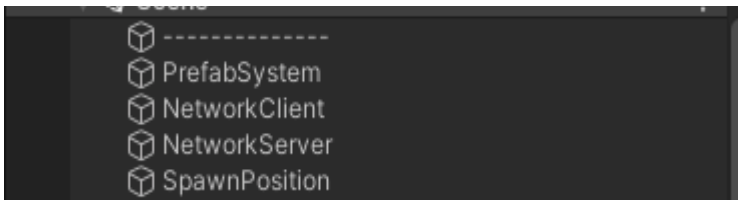


Use the WASD / Arrow keys to move the player around.

In the server world, we are updating all 10k monsters at all times.

On the client (what you see), we receive a subset of monsters around the player.

Check out the **Hierarchy** to configure the server and the client:



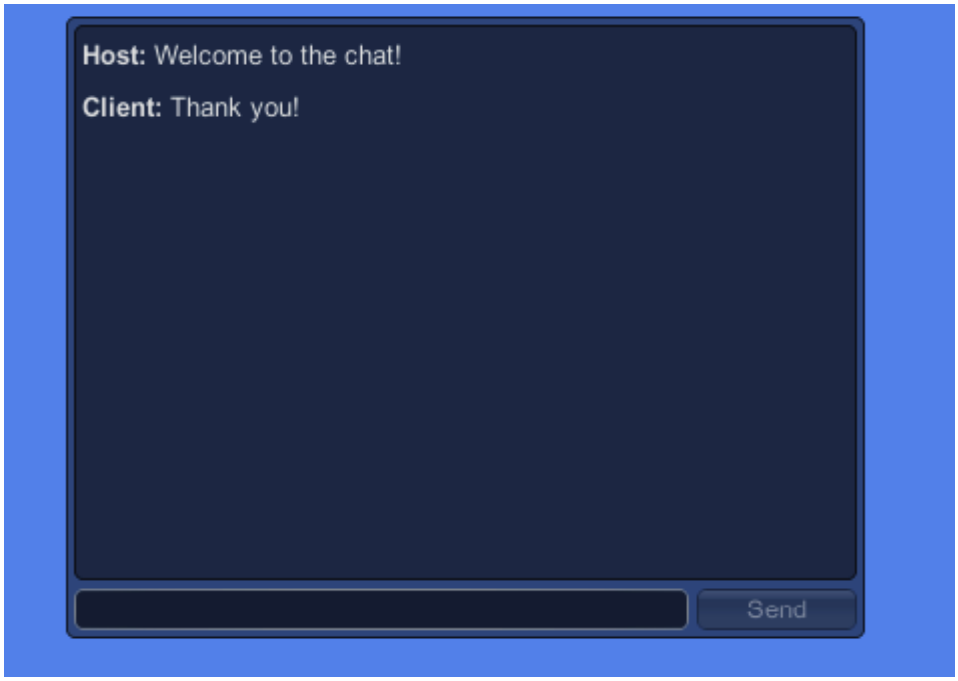
NetworkServer has an Interest Management component. You can increase the visibility radius to broadcast more monsters to the player. Note that DOTS/ECS is **extremely fast**. At some point, way faster than what your network card can handle. You will see Apathy's load balancing feature if you increase the visibility radius high enough:

With the default visibility radius of 15, we are already updating **10k** monsters on the server, and **2-3k** on the client. All in one process, and it runs fine on a regular laptop. That is completely unmatched networking performance.

Imagine what is possible on a really powerful dedicated server…

Example: Chat

DOTSNET comes with very simple, server authoritative chat example:



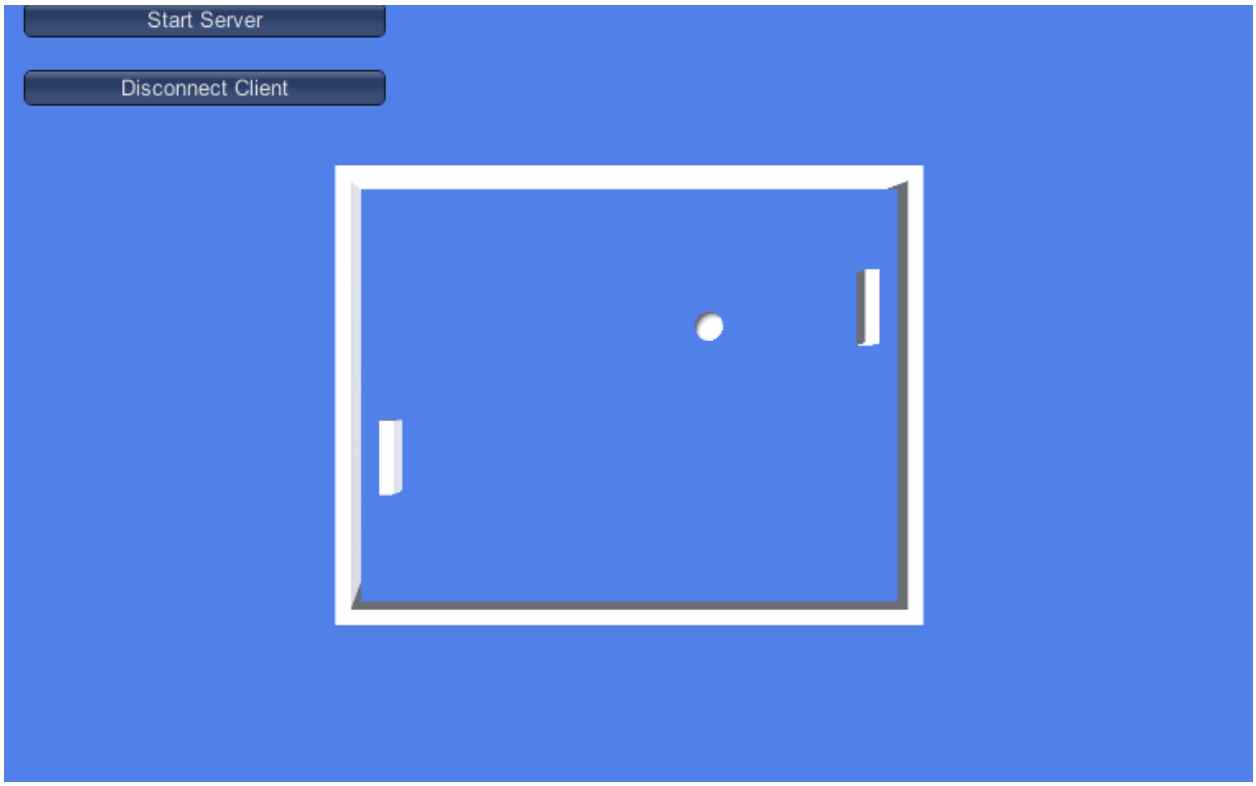
The Chat example is very useful to learn how to do a pure message based networking project, without any entities.

Notice how **ChatServerSystem** inherits from **NetworkServerSystem** and **ChatClientSystem** inherits

from **NetworkClientSystem** in order to store some state.

Example: Pong

The Pong example shows how to do a two-player pong game with DOTSNET.

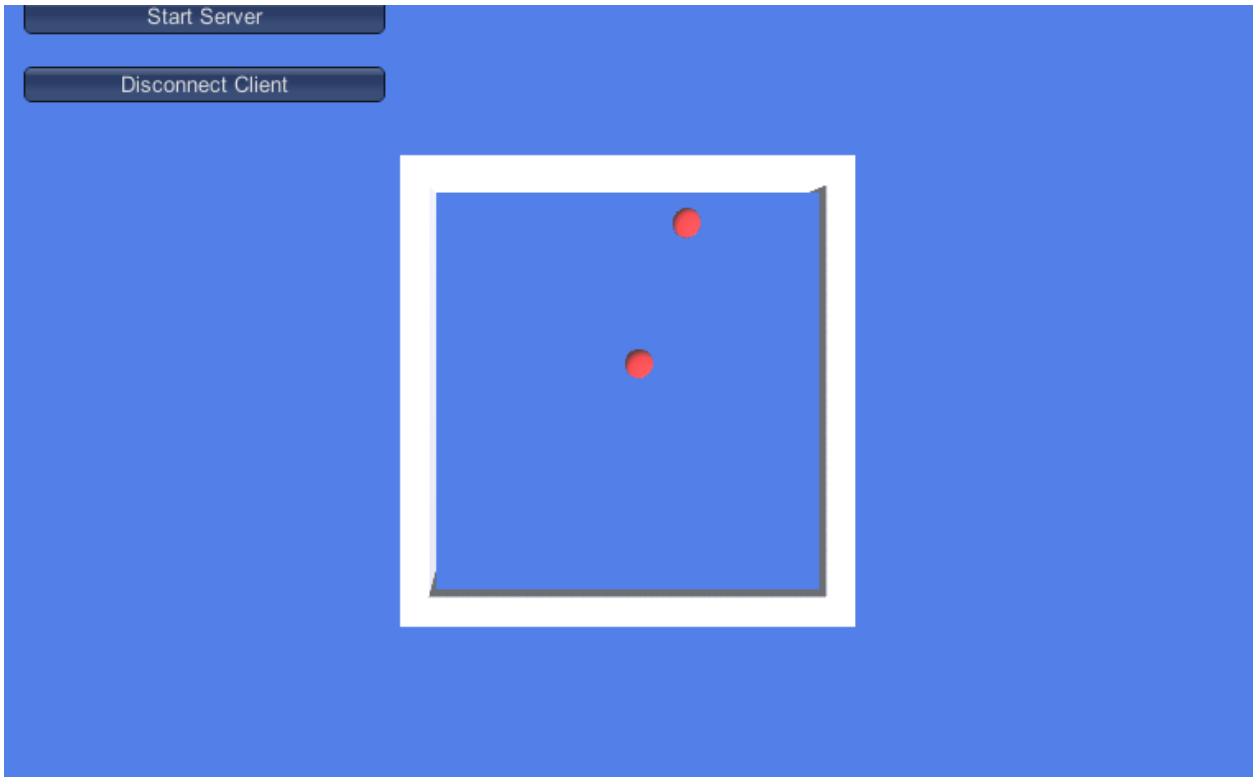


Notice how **PongServerSystem** inherits from **NetworkServerSystem** to store some state.

Inheriting from **NetworkClientSystem** is not needed, we use the default one.

Example: Physics

DOTSNET comes with a very simple Unity.Physics demo to showcase how it works:



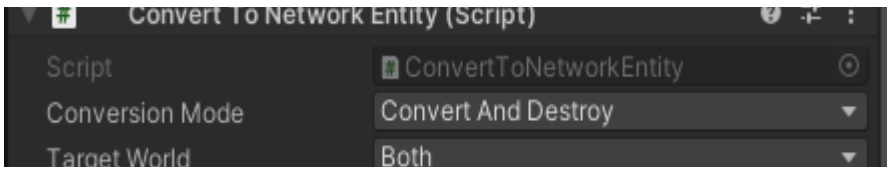
Make sure to update all your physics systems before the **BuildPhysicsSystem**. The **ClientConnectedSimulationSystemGroup** / **ServerActiveSimulationSystemGroup** do this for you automatically.

See also: Unity.Physics chapter in this documentation.

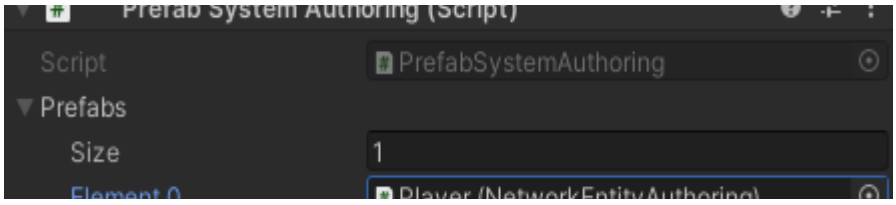
Authoring / Communicating with the ECS World

ECS Entities can be found in the Scene, and as prefabs as regular GameObjects.

Add the **ConvertToNetworkEntity** component to them, so that Unity automatically converts them to Entities into either the Server scene, the Client scene, or both:

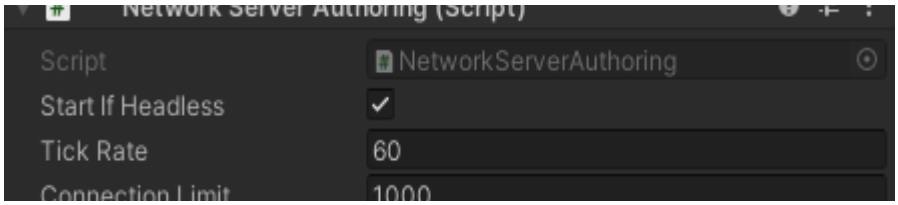


Prefabs need to be dragged into the PrefabSystem scene object's registered Prefabs:



Scene Objects are converted automatically.

Right now, Unity has no Inspector for ECS **Systems**. They simply live in memory without us being able to modify their public variables. This is why **DOTSNET** has an authoring component for each system, like **NetworkServerAuthoring:**



System Authoring components simply copy the exposed fields to the actual systems in the ECS world.

Migrating from Mirror

The **bad news** is that you will need to create your project from scratch, and can't reuse anything from the MonoBehaviour world. **DOTSNET** intentionally uses pure ECS, not hybrid.

The **good news** is that **DOTSNET** was optimized for ease of use. It's very clean, simple & elegant to work with. It's extremely stable thanks to over 90% test coverage.

In fact, it makes networking **fun** again!

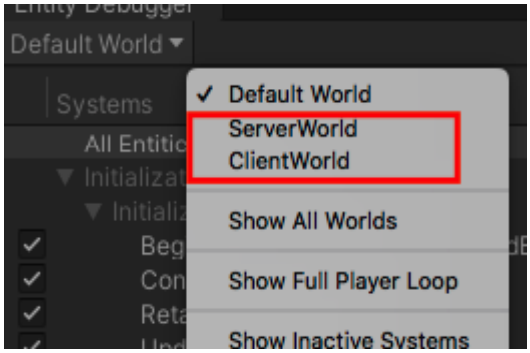
Here is a brief overview on how the old concepts translate to **DOTSNET**. Please take your time to learn each one, and don't expect to convert your full Mirror project to **DOTSNET** by friday.

Mirror	**DOTSNET**
Server-Only mode	Still exists, but in a completely separate Server World.
Client-Only mode	Still exists, but in a completely separate Client World.
Host Mode	Not needed anymore. Server & Client worlds are fully separated in memory. Simply start a server and start a client. *Yes, we have a true, fully memory separated host mode now.*
NetworkServer	NetworkServerSystem + Authoring
NetworkClient	NetworkClientSystem + Authoring
NetworkIdentity	NetworkEntity

NetworkBehaviour	Gone.
isServer	Not needed anymore. You are the server if your system is running in the server world.
isClient	Not needed anymore. You are the client if your system is running in the client world.
isLocalPlayer	NetworkEntity.owned
hasAuthority	NetworkEntity.owned
OnStartServer	OnStartRunning if [ServerWorld] and [UpdateInGroup(ServerActiveSimulationSystemGroup)]
OnStopServer	OnStopRunning if [ServerWorld] and [UpdateInGroup(ServerActiveSimulationSystemGroup)]
OnStartClient	OnStartRunning if [ClientWorld] and [UpdateInGroup(ClientConnectedSimulationSystemGroup)]
OnStopClient	OnStopRunning if [ClientWorld] and [UpdateInGroup(ClientConnectedSimulationSystemGroup)]
NetworkWriter	SegmentWriter
NetworkReader	SegmentReader
Connection.ready	ConnectionState.joinedWorld (on server)
NetworkServer.active	NetworkServerSystem.state == ACTIVE
NetworkClient.active	NetworkClientSystem.state != DISCONNECTED

Server & Client Worlds

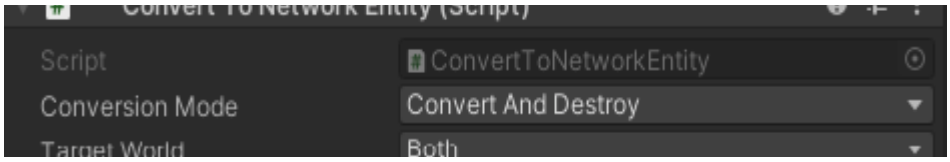
DOTSNET creates a ServerWorld and a ClientWorld when starting:



ConvertToNetworkEntity

ECS comes with a Convert**ToEntity** component that allows us to convert GameObjects to Entities in the **Default World**.

DOTSNET comes with a ConvertTo**NetworkEntity** component that converts GameObjects to Entities in the **Server/ClientWorld**:



Most entities will live in both worlds.

For example, a monster would live in both worlds so that the server can move it, then sync the position to the same entity in the client world.

World Attributes

You can put a system into the server world by using the **ServerWorld** attribute:

```
using DOTSNET;

[ServerWorld]
public class ServerSystem : ComponentSystem
{
}
```

You can put a system into the client world by using the ClientWorld attribute:

```
using DOTSNET;

[ClientWorld]
public class ClientSystem : ComponentSystem
{
}
```

A system can also be in **both worlds**:

```
using DOTSNET;

[ServerWorld, ClientWorld]
public class ServerAndClientSystem : ComponentSystem
{
```

```
}
```

If a system has no attributes, then it's only in the DefaultWorld.
DOTSNET entirely operates in the Server/Client world, not in the DefaultWorld.

Simulation System Groups

For convenience, DOTSNET comes with custom simulation system groups.

On the server, you can put a system into the Server**Active**SimulationSystemGroup, so it's only updated while the server is active *(after NetworkServerSystem.StartServer was called, until StopServer is called)*. For example, monster movement should only be updated while the server is actually active:

```
using DOTSNET;

[ServerWorld]
[UpdateInGroup(typeof(ServerActiveSimulationSystemGroup))]
public class MonsterMovementSystem : ComponentSystem
{
}
```

On the client, you can put a system into the Client**Connected**SimulationSystemGroup, so it's only updated while the client is connected to the server *(after a successful NetworkClientSystem.Connect until Disconnect)*. For example, local player movement should only be updated while the client is actually connected:

```
using DOTSNET;

[ClientWorld]
[UpdateInGroup(typeof(ClientConnectedSimulationSystemGroup))]
public class LocalPlayerMovementSystem : ComponentSystem
{
}
```

Accessing Server/ClientWorld from the Outside

Use the static **Bootstrap** class to access Server/Client World systems from anywhere in Unity:

```
using DOTSNET;

public class SomeClass
{
    void Example()
    {
```

```
        // log the names
        Debug.Log(Bootstrap.ClientWorld.Name);
        Debug.Log(Bootstrap.ServerWorld.Name);

        // find the server/client systems
        Bootstrap.ClientWorld.GetExistingSystem<NetworkServerSystem>();
        Bootstrap.ClientWorld.GetExistingSystem<NetworkClientSystem>();
    }
}
```

Selective System Authoring

Systems with Multiple Implementations

In some cases, we can have an abstract system type with multiple implementations. For example:

- **Transports**: we have one abstract **TransportServer/ClientSystem** class, but can have multiple implementations like **Apathy**, and later UDP/Websockets/Steam/etc.
- **Interest Management:** we have one abstract **InterestManagementSystem**, and an explicit **BruteForceInterestManagementSystem**. We might also have a grid system, room system, etc. later.

By default, ECS always creates all systems in the project.

This is a problem, because even if our project has 3 different interest management implementations, we only ever want to use one of them.

The Solution

The solution is to **disable** systems by default, and only **enable** them if an authoring component is added to the scene:

1. Add the **[DisableAutoCreation]** attribute to systems with multiple implementations:

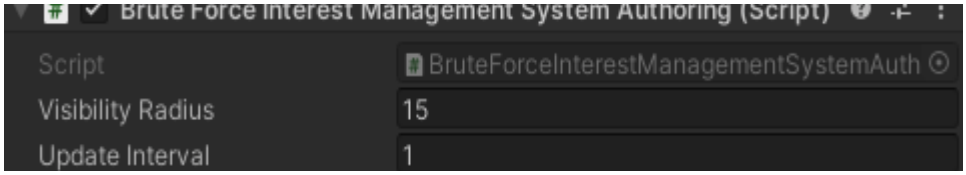
```
using DOTSNET;

[ServerWorld]
[DisableAutoCreation]
public class BruteForceInterestManagementSystem :
InterestManagementSystem
{
}
```

2. Create an Authoring component that implements our **SelectiveSystemAuthoring** interface:

```
using DOTSNET;

public class BruteForceAuthoring : MonoBehaviour,
SelectiveSystemAuthoring
{
      // add system if Authoring is used
      public Type GetSystemType() =>
          typeof(BruteForceInterestManagementSystem);
}
```

So far, the system is still disabled by default.

3. Add the Authoring component to the NetworkServer/NetworkClient GameObject:



When DOTSNET starts, it will scan the Hierarchy for all **SelectiveSystemAuthoring** components, and then create the systems specified in **GetSystemType** automatically.

Dependency Injection

DOTSNET comes with dependency injection for all systems in the server/client worlds.

Dependency injection is a comfort feature that you don't need to use, but you probably will because it makes your life a lot easier.

Often times, a system might need to use another system:

```
using DOTSNET;

[ServerWorld]
public class TestSystem : ComponentSystem
{
    void DoSomething()
    {
        // send a message
        GetExistingSystem<NetworkServerSystem>().Send(...);
    }

    void DoSomethingElse()
    {
        // send a message
```

```
        GetExistingSystem<NetworkServerSystem>().Send(...);
    }
}
```

For performance and ease of use, it makes sense to cache the system once in OnCreate:

```
using DOTSNET;

[ServerWorld]
public class TestSystem : ComponentSystem
{
    protected NetworkServerSystem server;

    protected override void OnCreate()
    {
        server = GetExistingSystem<NetworkServerSystem>().;
    }

    void DoSomething()
    {
        // send a message
        server.Send(...);
    }

    void DoSomethingElse()
    {
        // send a message
        server.Send(...);
    }
}
```

Caching all required systems in OnStartRunning can get cumbersome. This is where Dependency Injection comes in. Simply use the **[AutoAssign]** attribute:

```
using DOTSNET;

[ServerWorld]
public class TestSystem : ComponentSystem
{
    [AutoAssign] protected NetworkServerSystem server;

    void DoSomething()
    {
        // send a message
        server.Send(...);
    }

    void DoSomethingElse()
```

```
    {
        // send a message
        server.Send(...);
    }
}
```

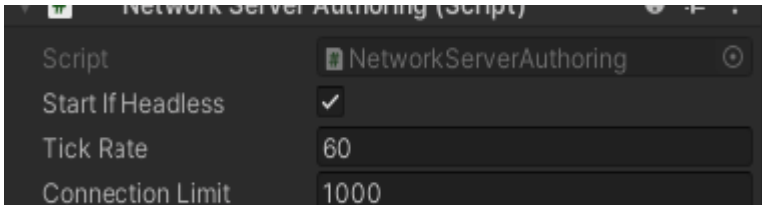
DOTSNET simply initializes all **[AutoAssign]** systems by calling World.GetExistingSystem at startup.

- A system in the **DefaultWorld** can use **[AutoAssign]** for any other system in the Default world.
- A system in the **ClientWorld** can use **[AutoAssign]** for any other system in the Client world.
- A system in the **ServerWorld** can use **[AutoAssign]** for any other system in the Server world.

NetworkServerSystem

NetworkServerSystem is the main server class, found in **ServerWorld**.

NetworkServerSystemAuthoring can be used to interact with it from the Inspector. In the example scene, the NetworkServer scene object has the authoring component:



NetworkServerSystem:

Property:	**Description:**
state	Current server state (active/inactive).
startIfHeadless	Auto start the server in Linux headless mode.
isHeadless	True if the server is running in Linux headless mode.
tickRate	Tick rate for all systems in the ServerActiveSimulationSystemGroup in Hz.
connectionLimit	Maximum amount of allowed player connections.
connections	All connected connections by connectionId.
spawned	All spawned Entities by netId.

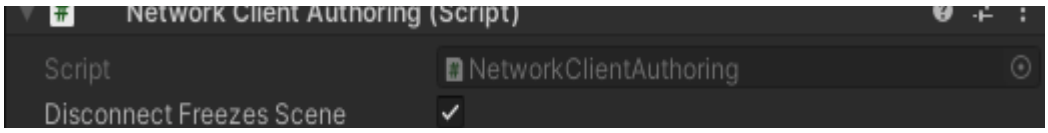
Function:	**Description:**

StartServer()	Start listening to incoming connections.
StopServer()	Stop listening to incoming connections.
Disconnect(connectionId)	Disconnect a connection by connectionId.
Send(message, connectionId)	Send a NetworkMessage to a connection.
RegisterHandler<T>(handler, requiresAuthentication)	Register a handler for an incoming message. Inheriting from NetworkServerMessageSystem registers it automatically. All messages except the handshake/login should require authentication.
UnregisterHandler<T>	Unregister handler for the NetworkMessage type.
Spawn(Entity, ownerConnectionid)	Spawn an instantiated Entity in the world and let all clients know about it. Entity needs to have a NetworkEntity component. Use ownerConnection if it's owned by a connection, e.g. player/player's pet.
Unspawn(Entity)	Unspawn the Entity from the world and let all clients know about it.
Destroy(Entity)	Unspawn and then Destroy an Entity. This is for convenience.
JoinWorld(connectionId, Entity)	Each game will handle their own character/team/race selection. Afterwards call JoinWorld with the selected player Entity to get started. This flags the connection as joinedWorld.

NetworkClientSystem

NetworkClientSystem is the main client class, found in **ClientWorld**.

NetworkClientSystemAuthoring can be used to interact with it from the Inspector. In the example scene, the NetworkClient scene object has the authoring component:



NetworkClientSystem:

Property:	**Description:**
state	Current client state (disconnected/connecting/connected).
disconnectedFreezesScene	Convenience feature that decides what to do after the client disconnects. By default, all

	NetworkEntities will be destroyed and cleaned up immediately. Enable this flag to freeze them in the scene instead, which is what some MMOs do because it looks cool. The NetworkEntities will automatically be cleaned up in the next Connect() call again.
spawned	All spawned Entities by netId that are around the player. Unlike the server, those aren't all entities that were spawned in the world. Only the player's observers.

Function:	**Description:**
Connect(address)	Connect to a server listening on the specified address. This is not blocking, it will simply start the connect process, set the state to CONNECTING and then to CONNECTED a bit later.
Disconnect()	Disconnect from the server. This happens immediately and the state is set to DISCONNECTED.
Send(message)	Send a NetworkMessage to the server.
RegisterHandler<T>(handler)	Register a handler for an incoming message. Inheriting from NetworkClientMessageSystem registers it automatically.
UnregisterHandler<T>	Unregister handler for the NetworkMessage type.

NetworkEntity

If an Entity should be synced to the client, then it needs to have a NetworkEntity component.

If it does not have a NetworkEntity component, then **DOTSNET** ignores it.

NetworkEntity:

Property:	**Description:**
netId	Unique ID to identity this Entity over the network. The ID is the same on the server and on the client.
prefabId	The prefabId of the prefab that this NetworkEntity was based on.
connectionId	The connectionId if the entity is owned by a connection, like a player or a player's pet. Null if not owned by anyone (like a monster). This is only known on the server.

owned	True if this NetworkEntity is owned by the client, like the player and the player's pet. This is only known on the client.

PrefabSystem

DOTSNET comes with its own system to manage prefabs.

The **PrefabSystem** can be found in both the ServerWorld and the ClientWorld.

Scene Prefabs

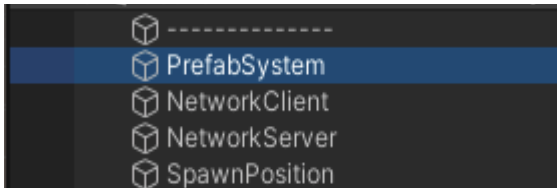
NetworkEntities like Monsters can be added to the Scene:

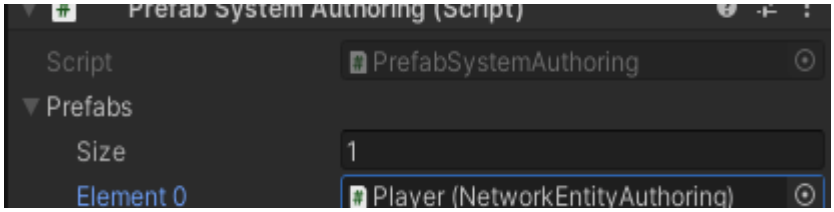
The PrefabSystem will convert them all to prefabs when starting.

Regular Prefabs

Regular prefabs that live in the Project Area:

Need to be dragged into the PrefabSystem manually in order to spawn them over the network:





PrefabSystem API

Field:	Description:
prefabs	All regular prefabs converted to Entity, by prefabId.
scenePrefabs	All scene objects convert to Entity, by prefabId.

Function:	Description:
Get(prefabId, out Entity)	Find a prefab by prefabId. This searches both the regular prefabs and the scene objects. Call this function to find the prefab, then use EntityManager.Instantiate to instantiate it.

NetworkMessage

All messages need to be structs that implement the **NetworkMessage** interface.

NetworkMessage Interface

Function:	Description:
GetId()	Get the unique Id for this message. • DOTSNET uses the 0x0000 - 0x0FFF space for messages. • Use **0x1000 - 0x9FFF** for your game messages. • Use 0xA000 - 0xAFFF for third party addon messages. To automate message ids, feel free to return a hash of the type name, but make sure it's in the above range.
Serialize(ref SegmentWriter)	Serialize all fields into the writer.
Deserialize(ref SegmentReader)	Deserialize all fields from the reader.

Sending a NetworkMessage

NetworkServerSystem.Send can send a **NetworkMessage** to a client connection.

NetworkClientSystem.Send can send a **NetworkMessage** to the server.

Whenever we want to send something from the client to the server, or from the server to a client,

we need to use a NetworkMessage.

Simply create a struct that implements the NetworkMessage interface:

```
using DOTSNET;

public struct JoinWorldMessage : NetworkMessage
{
      public Bytes16 playerPrefabId;
      public ushort GetID() { return 0x1001; }

      public bool Serialize(ref SegmentWriter writer)
      {
            return writer.WriteBytes16(playerPrefabId);
      }

      public bool Deserialize(ref SegmentReader reader)
      {
            return reader.ReadBytes16(out playerPrefabId);
      }
}
```

Make sure to serialize and deserialize all message data.

Make sure to use a different message id for each message.

Receiving a NetworkMessage

Inherit from **NetworkServerMessageSystem** to handle a received message on the server:

```
using DOTSNET;

public class JoinWorldMessageSystem :
NetworkServerMessageSystem<JoinWorldMessage>
{
    protected override bool RequiresAuthentication() { return true; }
    protected override void OnMessage(int connectionId, NetworkMessage
message)
    {
        // handle the message
        JoinWorldMessage joinMessage = (JoinWorldMessage)message;
    }
}
```

Almost all server messages should require authentication. For example, a client that hasn't logged in shouldn't be able to send a player movement message.

Inherit from **NetworkClientMessageSystem** to handle a received message on the client:

```
using DOTSNET;

public class MyMessageSystem : NetworkClientMessageSystem<MyMessage>
{
    protected override void OnMessage(NetworkMessage message)
    {
        // handle the message
        MyMessage myMessage = (MyMessage)message;
    }
}
```

SegmentReader/Writer

DOTSNET uses **SegmentWriter** to serialize, and **SegmentReader** to deserialize data.

Using SegmentReader/Writer

For example, this is how we serialize an int and a float3 position in a message:

```
using DOTSNET;

public struct TestMessage : NetworkMessage
{
    public int classIndex;
    public float3 position;
    public ushort GetID() { return 0x1001; }

    public bool Serialize(ref SegmentWriter writer)
    {
        return writer.WriteInt(classIndex) &&
               writer.WriteFloat3(position);
    }

    public bool Deserialize(ref SegmentReader reader)
    {
        return reader.ReadInt(out classIndex) &&
               reader.ReadFloat3(out position);
    }
}
```

Reading and Writing is:

- **Atomic:** WriteFloat3 either writes all 3 floats, or none if there is not enough space. ReadFloat3 either reads all 3 floats, or none if the buffer doesn't have enough data.
- **Allocation Free:** all reads and writes are allocation free for maximum performance. SegmentWriter/Reader use **ArraySegment**s internally.
- **Blittable:** DOTS/ECS only supports blittable types, which is why SegmentReader/Writer only support blittable types too.

Reference Passing

To avoid allocations, SegmentReader/Writer are value types (structs). When passing a reader/writer through functions, make sure to pass it as **ref**erence:

```
public bool Serialize(ref SegmentWriter writer)
{
    return writer.WriteInt(classIndex) &&
           writer.WriteFloat3(position);
}

public bool Deserialize(ref SegmentReader reader)
{
    return reader.ReadInt(out classIndex) &&
           reader.ReadFloat3(out position);
}
```

If you don't pass it as **ref**erence, then it will create a copy of the Writer/Reader, which would not modify the original writer/reader's Position.

In other words, always pass SegmentReader/Writer as reference!

Extending SegmentReader/Writer

You can extend SegmentReader/Writer with C#'s extension system:

```
public struct MyStruct
{
    public int level;
    public int experience;
}
```

```
using DOTSNET;

public static class Extensions
{
    public static bool WriteMyStruct(this SegmentWriter writer, MyStruct value)
    {
        return writer.WriteInt(value.level) &&
               writer.WriteInt(value.experience);
    }

    public static bool ReadMyStruct(this SegmentReader reader, out MyStruct value)
    {
        value = new MyStruct();
        return reader.ReadInt(out value.level) &&
```

```
            reader.ReadInt(out value.experience);
    }
}
```

BroadcastSystems

Inherit from **BroadcastSystem** to broadcast an entity's state from the server to all observers in an interval.

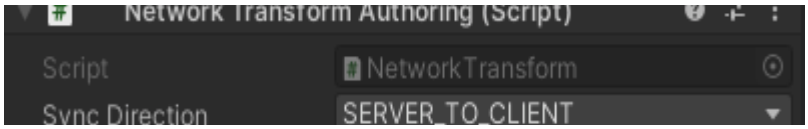
For example, here is a simplified version of our NetworkTransform broadcasting system:

```
using DOTSNET;

public class NetworkTransformServerSystem : NetworkBroadcastSystem
{
    protected override void Broadcast()
    {
        // for each NetworkEntity
        Entities.ForEach((Entity entity,
                          DynamicBuffer<NetworkObserver>
observers,
                          ref Translation translation,
                          ref Rotation rotation,
                          ref NetworkEntity networkEntity) =>
        {
            // send state to each observer connection
            for (int i = 0; i < observers.Length; ++i)
            {
                // create the message
                TransformMessage message = new TransformMessage(
                    networkEntity.netId,
                    translation.Value,
                    rotation.Value
                );

                // send it
                int connectionId = observers[i];
                server.Send(message, connectionId);
            }
        });
    }
}
```

Note: make sure to use a regular for-loop instead of foreach when iterating observers, because foreach allocates.

NetworkTransform

Add the NetworkTransformAuthoring component to a prefab in order to sync the position & rotation over the network:



NetworkTransform:

Property:	**Description:**
syncDirection	SERVER_TO_CLIENT to sync it from the server to all clients. CLIENT_TO_SERVER for client authoritative movement where the client syncs it to the server, and the server broadcasts it to all other clients.

Authentication

DOTSNET supports all types of authentications.

Inherit from **NetworkClientAuthenticatorSystem** and override **BeginAuthentication** to initiate the handshake from the client. Usually you would send a LoginMessage with username & password to the server.

Inherit from **NetworkServerAuthenticatorSystem** and call **SetAuthenticated** after the LoginMessage was finished, which flags the connection as **authenticated**. Messages that require authentication will be handled for this connection afterwards.

You can also send several handshake messages back and forth. Simply call **SetAuthenticated** once it's done.

Note that connections are authenticated by default, unless you inherit from **NetworkServerAuthenticatorSystem**.

Interest Management

When creating large worlds, we don't want to synchronize the full world to every single player connection. Instead, we only send nearby state to each player connection.

Each NetworkEntity has observers, which are the player connections that it is synced to. Observers are a **DynamicBuffer<NetworkServer>**.

The **InterestManagementSystem** rebuilds the observers every few seconds.

InterestManagementSystem is abstract, so that different games can have different interest management algorithms. For example:

- Brute Force
- Spatial Hashing
- Sphere Cast Checks
- Line Sweeps
- Instance based
- etc.

By default, **DOTSNET** comes with a **BruteForceInterestManagementSystem** which runs raw distance checks to every player connection for each NetworkEntity in the world. Don't be confused by the name: while it is the brute force O(n^2), it's still extremely fast thanks to DOTS.

Later on, **DOTSNET** will also implement a Spatial Hashing system which will scale even better.

InterestManagementSystem:

Function:	**Description:**
RebuildAll	Inherit to rebuild all observers for all NetworkEntities.
SendSpawnMessage	Call this from RebuildAll after adding an observer to an Entity to spawn the Entity on the observer connection.
SendUnspawnMessage	Call this from RebuildAll after removing an observer from an Entity to unspawn the Entity on the observer connection.

Transports

While **DOTSNET** handles all the higher logic like spawning, unspawning, observers, etc., the **Transport** handles only the low level packet sending.

DOTSNET can work with different low level transports, and comes with Apathy by default. We use Apathy because as of now, it's the only transport that can *barely* keep up with **DOTSNET**'s insane performance requirements.

Inherit from **TransportClientSystem** and **TransportServerSystem** to implement your own transport.

Note that Send and OnData always send and receive **exactly one message**. So if a client sends 3 and then 2 bytes, the server should receive exactly 3 and then 2 bytes instead of 5 bytes.

Note that all TransportServer/ClientSystem functions are not thread safe and should only be called and handled in the main thread. If you want more threads, you will have to add them yourself in the background.

TransportClientSystem

Event:	**Description:**
OnConnected	Call this after successfully connecting to the server. **Only call this from the main thread!**
OnData<ArraySegment<byte>>	Call this after receiving a message from the server. Only call this with exactly the message bytes. Not more, not less. Pass an ArraySegment for allocation free message processing. The client will handle the message immediately, so the ArraySegment's array can be modified again immediately after returning. **Only call this from the main thread!**
OnDisconnected	Call this after disconnecting from the server. **Only call this from the main thread!**

Function:	**Description:**
IsConnected	Return true if the client is currently successfully connected to the server.
Connect(address)	Start connecting the client to the server. Should not be blocking, and only initiate the connect process in the background.
Send(ArraySegment<byte>)	Send bytes to the server. Only called with exactly the message bytes. Not more, not less. The ArraySegment's internal array is only valid until returning, so either send it directly, or copy it into an internal buffer.

Disconnect	Disconnect from the server.

TransportServerSystem

Event:	**Description:**
OnConnected<connectionId>	Call this after a new client connected. **Only call this from the main thread!**
OnData<connectionId, ArraySegment<byte>>	Call this after receiving a message from a client connection. Only call this with exactly the message bytes. Not more, not less. Pass an ArraySegment for allocation free message processing. The server will handle the message immediately, so the ArraySegment's array can be modified again immediately after returning. **Only call this from the main thread!**
OnDisconnected<connectionId>	Call this after a client disconnected from the server. **Only call this from the main thread!**

Function:	**Description:**
IsActive	Return true if the server is still listening to incoming connections.
Start	Start listening to incoming connections.
Send(connectionId, ArraySegment<byte>)	Send bytes to the client. Only called with exactly the message bytes. Not more, not less. The ArraySegment's internal array is only valid until returning, so either send it directly, or copy it into an internal buffer.
Disconnect(connectionId)	Disconnect the connection from the server.
GetAddress	Get a connection's IP address. Can be useful for IP bans etc.
Stop	Stop listening to incoming connections.

Unity.Physics Support

DOTSNET integrates perfectly with Unity.Physics, but you need to update your systems in just the right order.

- **DOTSNET** comes with a safe **ApplyPhysicsGroup.** Every system that is updated in that group via [UpdateInGroup(typeof(ApplyPhysicsGroup))] is safe to apply physics.
- Both our **ServerActiveSimulationSystemGroup** and **ClientConnectedSimulationSystemGroup** are part of **ApplyPhysicsGroup** automatically, so you don't need to worry about anything if your systems update in those groups.