Massive Documentation

Release 0.1

Inhumane Software

August 07, 2014

Contents

Welcome to the MassiveNet 0.1 documentation. MassiveNet is currently in beta and undergoing heavy development.

If you have any difficulties with MassiveNet, notice any errors in the documentation, or want to provide feedback/suggestions, please send us an email at support@inhumanesoftware.com

CHAPTER 1

Introduction

1.1 Features

- Full source code included.
- Supports fully authoritative servers.
- Low overhead.
- Easy to move from Unity Networking to Massive Net.
- Supports large number of concurrent connections.
- Support for seamless hand-off.
- Clients can connect to multiple servers at once.
- Support for return values on RPCs using a NetRequest.
- Support for IEnumerator RPCs.
- Respond to a NetRequest via parameter instead of return value if needed.
- Create an RPC by simply adding the [NetRPC] attribute to a method.
- Support for serializing/deserializing custom types.
- Automatic network LOD/scope for network objects, a crucial feature for massive games.
- Incremental server-side tasks spread load more evenly across each frame.

1.2 Getting Started

MassiveNet comes with an example project designed to introduce you to the basics of using the library. There are three separate projects nested under the Massive > Examples > NetSimple folder. You'll find Client, Lobby, and Server folders, each containing their own scripts, prefabs, and scenes.

The main scripts to pay attention to are the ClientModel, ServerModel, and LobbyModel scripts. They are responsible for NetSocket configuration and startup as well as basic logic for handling connections.

You'll also want to take a look a the scripts responsible for position synchronization, which are NetController and PlayerSync in the Client > Scripts folder, and the PlayerCreator and AiCreator scripts in the Servers > Scripts folder.

You'll notice in the LobbyModel that two Zones are created. It is recommended you read up on Zones to understand their purpose, but for now, let's just say that they are Server configurations. This particular example requires two

servers to connect to the Lobby before a client can connect. This is because two Zones are created in the Start method of the LobbyModel class.

To run the example project, build each scene separately, then start the lobby, two servers, and one or more clients.

Here is a breakdown of how this example works:

1. The Lobby is started. In the LobbyModel Start method, the NetSocket and Zones are created and configured.
2. Two Servers are started. The Servers generate a random port between 17100 and 18000 to listen on, configure the NetSocket with settings appropriate for a Server, and start the NetSocket. After that, the Servers connect to the Lobby using the address provided in the LobbyAddress string.
3. After the Servers connect to the Lobby, the Lobby first provides numeric assignments to all of the RPC method names that the Servers possess. This happens because the Lobby is configured as the RPC Definition Authority. There must always be an RPC Definition Authority to assign numeric IDs. MassiveNet assigns numeric IDs to RPCs to reduce the bandwidth requirements of sending RPCs.
4. Once the Servers receive RPC ID assignments, they are each assigned to a Zone. In the case of this example, the LobbyModel created two zones, Cube and Sphere, so each Server will be assigned to one of these.
5. After the Servers have processed their assignments, they signal to the Lobby that the assignment is successful. Clients can now connect to the game world.
6. When a Client connects to the Lobby, the Lobby first sends RPC ID assignments to the Client.
7. Once RPC IDs have been resolved, the Client connection is signaled as completed and the Client sends an RPC to the Lobby signaling that it wants to connect to the game world.
8. Once the Lobby receives the RPC from the Client, it first checks that there is a Server assigned to each Zone. Then, the Lobby informs the Server assigned to the Cube zone that a Client is connecting and that it should spawn a Player object for it.
9. Once the Cube Server sends confirmation to the Lobby, the Lobby sends an RPC which tells the Client to connect to both the Cube and the Sphere Servers.
10. Once the Client connects to the Cube server, a Player object is spawned for that Client.
11. The Client can now freely move around the game world. When the Client gets close enough to the Sphere zone, the Client's Player object will be seamlessly handed off to the Sphere. The Client can see the AI objects from both servers if it is in range, but only one of the two Servers will handle the Client's Player object at any one time.

CHAPTER 2

Core Concepts

2.1 Zones

Zones are a way of representing an area of the game world in terms relevant to game servers. When you break up a game world into Zones, you define an area of responsibility for a server. On top of this, when you define a Zone, you make it clear that a server must exist for that area of the game.

When players (or NPCs) move throughout the game world, the NetworkView that represents them will move across zone boundaries. When that occurs, the server that is currently responsible for the NetworkView changes. This process is called a handoff.

In a classic server/client architecture, a client is only connected to one game server at a time. Sure, they may also be connected to a master server, a chat server, etc., but the client is usually only connected to one server that represents the simulation of the physical game space.

With the MassiveNet architecture, a client is often connected to more than one game server at a time. When a client nears the boundaries of the Zone it is currently in, it needs to be able to observe network objects (NetViews) that are in those nearby Zones. Likewise, a client will need to quickly and seamlessly be able to switch its communication target to a new server when it crosses the Zone boundary.

When a game server possessing a ZoneListener component connects to another server with a ZoneMaster component, the ZoneMaster will assign the ZoneListener to an unassigned Zone if one is available. When this happens, the ZoneMaster will send all of the data necessary for the game server to fill its role in the game world.

One of a game server's responsibilities is checking the location of each NetView it is currently responsible for and handing off that responsibility to a peer Zone when necessary. The criteria for this responsibility switch, or handoff, are defined within the parameters of the Zone class. Using a combination of the server's own Zone parameters and that of its peers, a server can hand off the responsibility for a NetView to the appropriate peer. Once the peer has taken responsibility, the new server of that NetView will notify the client(s) (if any) that control that NetView that it's time to switch communication target.

2.2 Views

In MassiveNet, like in Unity Networking, a NetView is representative of a communication channel reserved for a network object. A NetView provides a unique network identifier, concept of ownership, and facilities for synchronizing the state of the object. A NetView component must be attached to each object which has a state that must be synchronized.

NetViews are managed by the ViewManager component. A ViewManager component should be attached to the same GameObject as the NetSocket component for which it operates. While a ViewManager is required when utilizing

NetViews, it is not a requirement for using MassiveNet. As shown in the NetSimple example, the Lobby does not make use of a ViewManager at all.

2.3 Scope

Scope is a way of dynamically defining how often a connected client should receive synchronization information about a NetView, or whether or not they should even receive information at all.

In large game worlds where there are many clients and NetViews, sending all information to all clients at all times could be prohibitively expensive, both for bandwidth and processing power. To provide high efficiency, scoping must be utilized. Fortunately, MassiveNet provides this functionality right out of the box.

In a traditional multiplayer game, it is assumed that a connecting client needs to instantiate (or spawn) every network object in the game as well as receive updates for these objects at all times. The implication is that all network objects are in scope and you must explicitly override this default behavior to provide scoping functionality.

With MassiveNet, all objects are implicitly out of scope for a connection. For a connection to receive instantiation information and synchronization updates for a NetView, you must set that NetView as in-scope for the connection. When a game server is utilizing the ScopeManager component, this is handled automatically. The ScopeManager incrementally updates the scope of each NetView in relation to each connected client in order to determine which NetViews are in scope for the connection.

This isn't the end of the importance of scoping, however. Just like how 3D models can have different levels of detail based on the camera's distance from them, client connections can have different levels of scope for NetViews based on their distance from the client's in-game representation. This is sometimes referred to as network LOD (Level of Detail) due to its parallels with traditional application of LOD for reducing complexity of 3D models.

MassiveNet's ScopeManager is able to set three different levels of scope for in-scope NetViews. These three levels correspond to the distance between the client and the NetView's game object. The first scope level means the connection will receive every synchronization message, the second, every other synchronization message, and the third, every fourth synchronization message.

2.4 Synchronization

Synchronization in MassiveNet refers to the periodic transmission of changes in a NetView's state to all in-scope connections.

The most common example of this in real-time multiplayer games is synchronization of position. Most games have objects that move non-deterministically, that is, their position changes in a way that cannot be determined given the information that is currently known. Given that, there must be a method to keep everyone up to date without using too much bandwidth and computational power.

MassiveNet's solution to synchronization is NetSync, which is an event that is fired by a NetView at intervals defined by the SyncsPerSecond parameter of the ViewManager component. Here's an example of a Monobehaviour making use of the NetSync event to write position synchronization to the provided stream:

```
void Start()
{
  netView = GetComponent<NetView>();
  netView.OnNetSync += NetSync;
  controller = GetComponent<CharacterController>();
}

void NetSync(NetStream syncStream)
{
```

```
  syncStream.WriteVector3(rigidbody.position);
  syncStream.WriteQuaternion(rigidbody.rotation);
  syncStream.WriteVector2(new Vector2(inputX, inputZ));
  netView.SendSync("PlayerMove", RpcTarget.Server);
}
```

The receiving end might look like this:

```
[NetRPC]
void PlayerMove(Vector3 position, Quaternion rotation, Vector2 velocity)
{
  rigidbody.position = position;
  rigidbody.rotation = rotation;
  lastVel = velocity;
}
```

2.5 Serialization

Serialization (and deserialization) is a very important concept in computer science. It encompasses varying methodologies for communication and storage of data between disparate systems. Put simply, it can be defined as a method of turning your objects/data into zeroes and ones so that it may be sent across the network, and then turning them back into useable objects/data on the receiving end.

There are many different ways to serialize and deserialize data, each with its own set of strengths and weaknesses. For the needs of real-time networked games, the process of serializing and deserializing must be both quick and compact.

By default, MassiveNet provides serialization and deserialization of the following types:

- bool
- byte
- short
- ushort
- int
- uint
- float
- double
- string
- Vector2
- Vector3
- Quaternion

This is not the end of supported types, however. MassiveNet allows the definition of "Type Codecs", which are simply separate methods for both serialization and deserialization of a custom type.

For example, lets say there is a class called PlayerData. Here's what it might look like:

```
public class PlayerData
{
  public string name = "Bob";
  public int hp = "93";
  public Vector3 position = new Vector3(118, 0, 10);
```

```
  public static void SerializePlayerData(NetStream stream, object instance)
  {
    PlayerData data = (PlayerData)instance;
    stream.WriteString(data.name);
    stream.WriteInt(data.hp);
    stream.WriteVector3(data.position);
  }

  public static object DeserializePlayerData(NetStream stream)
  {
    PlayerData data = new PlayerData();
    data.name = stream.ReadString();
    data.hp = stream.ReadInt();
    data.position = stream.ReadVector3();
    return data;
  }
}
```

Now, all that is needed is to register the static methods responsible for serializing and deserializing PlayerData. To do this, all you must do is call the following from anywhere:

```
NetSerializer.AddCodec<PlayerData>(PlayerData.SerializePlayerData, PlayerData.DeserializePlayerData);
```

CHAPTER 3

Support

Support is just an email away. If at any point you feel stuck, want a second opinion, or want to provide feedback, we want to hear from you!

support@inhumanesoftware.com