Massive Documentation

Release 0.2

Inhumane Software

June 15, 2016

Contents

Welcome to the MassiveNet 0.2 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 MassiveNet.
- 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 two separate projects nested under the MassiveNet > Examples > NetSimple folder. You'll find Client and Server folders, each containing their own scripts, prefabs, and scenes.

The main scripts to pay attention to are the ClientModel and ServerModel 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 ServerModel 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. Because two Zones are configured in this example project, two servers are required to host the game world.

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

Here is a breakdown of how this example works:

1. The first Server is started. Since the port is not incremented and there are no peer addresses defined in the Peer list, the ServerModel determines that it will act as the ZoneManager and configures two Zones. The ZoneManager assigns Server 1 to the first Zone.
2. The second Server is started. Since the initial port is in use, the ServerModel determines it should attempt to connect to a peer listening on the initial port.
3. Since Server 1 is acting as the ZoneManager, Server 1 will assign Server 2 to the remaining unassigned Zone.
4. Now that all Zones are assigned, the game world is ready for clients.
5. When a Client connects to Server 1, the Server informs the Client that it will be sending numeric assignments for RPC method names. This is because both Servers are configured to be Protocol Authorities. A Protocol Authority is reponsible for assigning automatically generated numeric Ids to RPC method names. The Server first sends all of the assignments for its own NetRPC methods, and then the Client will request assignments for any NetRPC methods unique to the Client.
6. Once RPC IDs have been resolved, the connection is complete. Server 1's ZoneServer component then sends an RPC to the Client's ZoneClient component, informing it that it must connect to Server 2.
7. If the Client has successfully connected to Server 2, it will inform Server 1 that Zone setup is complete.
8. Upon successful Zone setup, the Client signals to Server 1 that it would like to be spawned. Server 1 will then instantiate a Player object for the Client via the ViewManager component.
9. The Client can now freely move it's Player object around the game world. When the Client's Player gets close enough to the Sphere zone, the Client's Player object will be seamlessly handed off to Server 2, which is responsible for the area around 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 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.

It is important to note that NetView creation must be handled by the ViewManager. The way to utilize NetView functionality is to attach a NetView component to a prefab, and then instantiate the prefab using the ViewManager.

2.2 Instantiation

To create a GameObject that will send and receive network messages, you must first create a prefab for it. The prefab must have a NetView component attached.

In MassiveNet, there are four identifiers for prefabs: Creator, Owner, Peer, and Proxy. Creator is the prefab used by the creator/server of the NetView, Owner is for those authorized to communicate/control the NetView, such as a client, Peer is for connections that are a Peer to the Creator, and Proxy is for those who need to observe the NetView but are not allowed to send communications to it.

These identifiers are added to the end of a prefab name, preceded by a "@". For example: "Player@Creator" or "Player@Owner".

To instantiate a NetView prefab, you must call NetViewManager.CreateView and provide the basic necessary information. If you are spawning a NetView that will be controlled by another connection, you need to provide the connection, the group number (0 is the default group), and the prefab root. The prefab root is everything before the "@" in the prefab name.

For example, spawning a Player object on the Server that will be controlled by a Client would look similar to:

```
public class Spawner {

  NetViewManager viewManager;

  void Awake() {
      viewManager = GetComponent<NetViewManager>();
```

```
}

// Spawns "Player@Creator" locally and "Player@Owner" on connection
// Assigned to default group 0
public void SpawnPlayerFor(NetConnection connection) {
    viewManager.CreateView(connection, 0, "Player");
}
```

Here's what happens when CreateView is called:

1. The proper local prefab name is determined based on who the controller (if any) and server are.
2. The prefab with the generated prefab name (e.g., "Player@Creator") is instantiated. If the ViewManager.InstantiatePrefab delegate is assigned (for use by an object pooling system, for example), the delegate will be called. Otherwise, the GameObject will be instantiated normally. If the resulting GameObject does not have a NetView, the GameObject will be destroyed and an exception will be thrown.
3. The OnNetViewCreated event will be triggered.
4. As a result of the event, the ScopeManager (if any) will perform an immediate scope calculation for the new NetView to determine which connections are in-scope. Every in-scope connection will receive the command and data to instantiate the new View.
5. When the ScopeManager tells the NetView to send instantiation data to an in-scope connection, the NetView determines the relationship level of the connection to decide which instantiation data collection event should be fired. There is an instantiation event for each type of relationship (Creator, Owner, Peer, and Proxy). For security and efficiency reasons, you wouldn't want to send the entire contents of a Player's inventory to every other client (Proxy), for example, but you probably do want to send what they have equipped for visual reasons. The events are aptly named: OnWriteOwnerData, OnWriteCreatorData, OnWriteProxyData, and OnWritePeerData
6. The NetView calls the relevant event to gather the necessary data to instantiate itself for the connection and then sends the command and its associated data.

...

On the receiving end of an Instantiate command:

7. Steps 1 through 3 happen on the receiving end of each connection that receives the instantiate command.
8. Since the connection is on the receiving end of the instantiate command, the NetView will take the instantiation data (NetStream) it received with the command and trigger OnReadInstantiateData. The data in the NetStream can be used to perform the initial setup. One of the most common uses for this initialization data is to set the position of the NetView.

2.3 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:

```
private NetView view;

void Awake() {
    view = GetComponent<NetView>();
    view.OnWriteSync += WriteSync;
}

RpcTarget WriteSync(NetStream syncStream) {
    syncStream.WriteVector3(transform.position);
    syncStream.WriteFloat(transform.rotation.eulerAngles.y);
    return RpcTarget.NonControllers;
}
```

The receiving end might look like this:

```
private NetView view;

void Awake() {
    view = GetComponent<NetView>();
    view.OnReadSync += ReadSync;
}

void ReadSync(NetStream syncStream) {
    transform.position = syncStream.ReadVector3();
    float yRot = syncStream.ReadFloat();
    transform.rotation = Quaternion.Euler(0, yRot, 0);
}
```

2.4 RPCs

In the synchronization section, we learned about how quickly changing variables such as position are kept in sync by sending unreliable messages in quick intervals. What about everything else?

MassiveNet supports RPCs (Remote Procedure Calls). By adding the [NetRPC] attribute to a method inside of a MonoBehaviour, MassiveNet will be aware that the method supports being called as a remote procedure.

A simple example of an RPC being sent from within script attached to a NetView GameObject:

```
public class SomeClientScript : MonoBehaviour {

    NetView view;

    void Start() {
        view = GetComponent<NetView>;
        SayHello();
    }

    // Makes the server print "Hello, Server!" three times.
    void SayHello() {
        view.SendReliable("HearMessage", RPCTarget.Server, "Hello, Server", 3);
    }
}
```

Server side:

```
public class SomeServerScript : MonoBehaviour {
```

```
    [NetRPC]
    void HearMessage(string message, int printCount) {
        for (int i = printCount; i > 0; i--) {
            Debug.Log(message);
        }
    }
}
```

The server will reject any RPC messages coming from a connection that is not authorized to speak to the NetView. Only controllers (Owners), servers, and peers may send messages to a NetView.

But what if it doesn't make sense to send the RPC to a NetView? What if the message isn't related to GameObjects at all? Maybe we want to send a message to the server to inform that we would like to start the logout process.

In cases where a message needs to be sent without being associated with a NetView, you can send them directly through the NetSocket. In the example project, you might attach a script to the Client or Server object that sends and receives RPCs without any associated NetView.

For example:

```
public class SomeViewlessClientScript : MonoBehaviour {

    NetSocket socket;

    void Start() {
        socket = GetComponent<NetSocket>();
        socket.Events.OnConnectedToServer += SayHowdy;
    }

    void SayHowdy(NetConnection server) {
        socket.Send("ReceiveGreeting", server, "Howdy, Server!");
    }
}

public class SomeViewlessServerScript : MonoBehaviour {
    NetSocket socket;

    void Start() {
        socket = GetComponent<NetSocket>();
        socket.RegisterRpcListener(this);
    }

    [NetRPC]
    void ReceiveGreeting(string greeting, NetConnection sender) {
        Debug.Log("Received a greeting from " + sender.Endpoint + ": " + greeting);
    }
}
```

You may have noticed something peculiar above, and it's not the southern accent. In the server script, you'll find that there's an extra parameter defined for the ReceiveGreeting RPC. The Client doesn't send a NetConnection (and can't), so how and why is NetConnection a parameter, and why doesn't it cause a problem?

In MassiveNet, NetConnection is a special case. Since it doesn't make sense to send a NetConnection, the opportunity was taken to allow a NetConnection parameter to be added to an RPC so that you can identify where the RPC came from.

2.5 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.

2.6 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
- long
- float
- double
- string

- Vector2
- Vector3
- Quaternion

This is not the end of supported types, however. MassiveNet allows the use of custom (de)serializers, 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 {

  string name;
  int hp;
  Vector3 position;

  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.Add<PlayerData>(PlayerData.SerializePlayerData, PlayerData.DeserializePlayerData);
```

2.7 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 NetView that represents them will move across zone boundaries. When that occurs, the server that is currently responsible for the NetView 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 ZoneServer component connects to another server with a ZoneManager component configured to act as the authority, the ZoneManager will assign the ZoneServer to an unassigned Zone if one is available. When this happens, the ZoneManager 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.

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