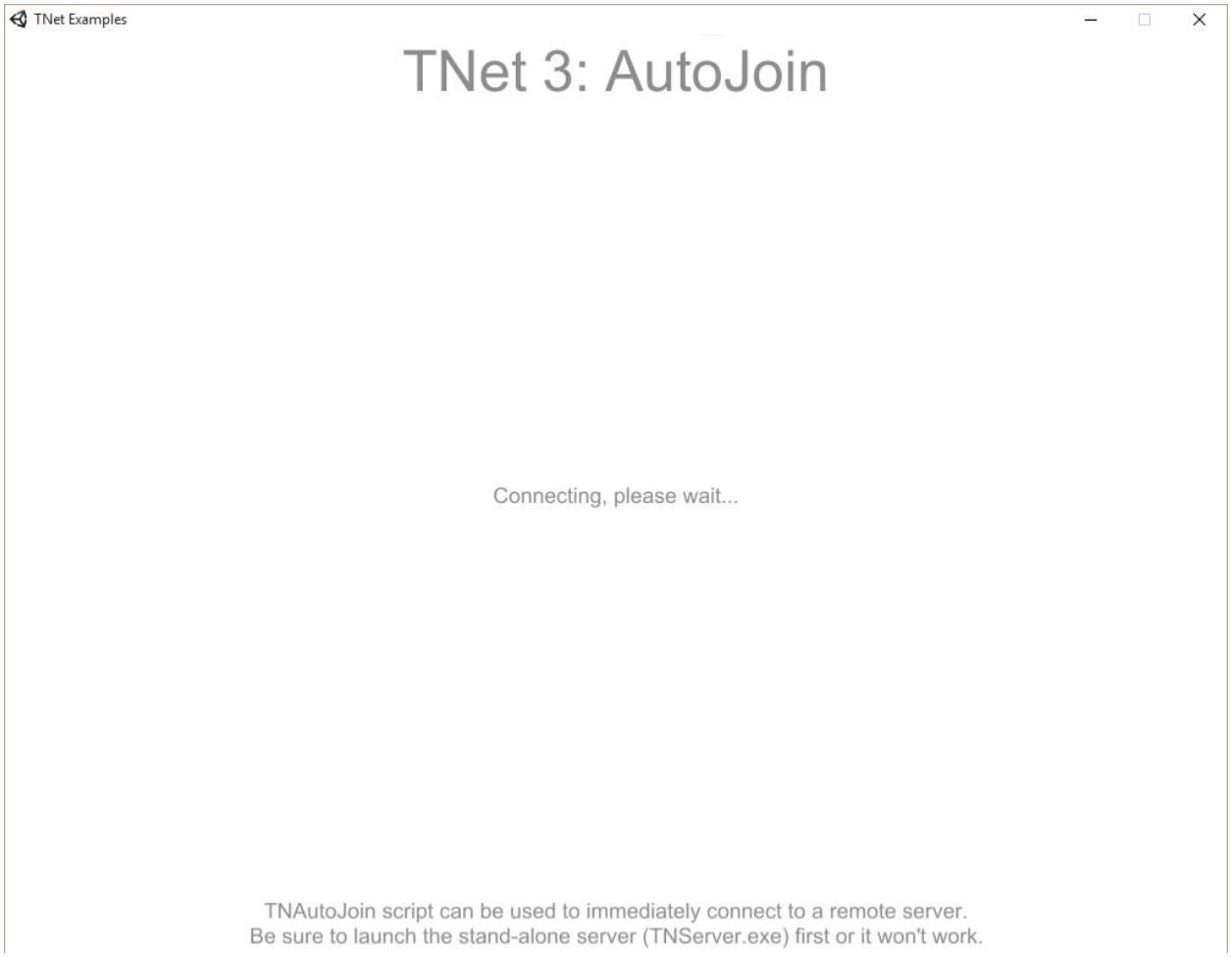
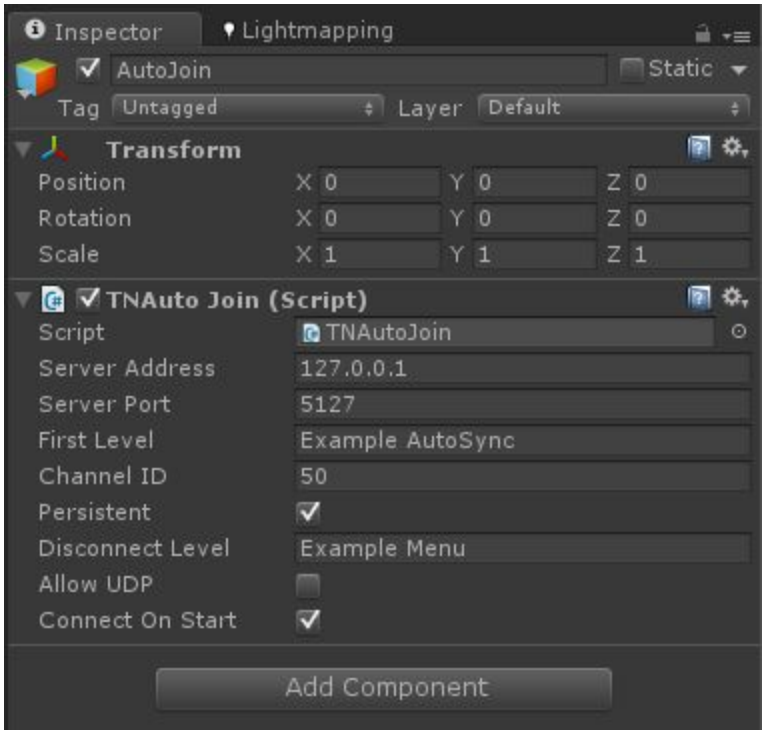
TNet 3: AutoJoin Example In-Depth

TNAutoJoin demonstrates how to connect to a server with various networking options. It requires the stand alone server, so if you have no done so yet, follow these steps to set it up.



In the TNet folder there is a Zip file named TNetServer.zip. Open it up and copy TNServer.exe to your Unity project's root folder. Run it. If the Windows firewall attempts to block it, allow it, quit by pressing q, enter, and enter again after a few seconds. Then start it up again. Now the ports will be properly opened. Minimize this program and go back to Unity.



To start off, let's go through the various public fields and what they mean.

- Server Address - This is the IP address of the machine on the internet which you would like to connect to. Certain IP's have special meanings such as the one here. 127.0.0.1 is a "loopback" address and it will connect to the computer you are currently using. This is great for testing, but you should never release a game using it.
- Server Port - Each IP address has any number of ports available to it and different programs will attempt to use different ports. Port 80 for example is used on web servers to display HTML. The TNet server uses 5127 as its default port so that is the port we will be using in this example.
- First Level - The Unity scene which will be loaded after this script connects to the server. In this case it will load the AutoSync example scene.
- Channel ID - Which network channel this client should join after connecting. The default is 0, but we're going to connect to channel 50.
- Persistent - Will Channel 50 continue to exist on the server even if there are no network clients connected? In this case the cubes in the AutoSync example scene will remember their adjusted positions.
- Disconnect Level - The Unity scene which will display once the client disconnects from the network. In this case it will go back to the TNet menu scene.
- Allow UDP - UDP is a type of network packet which is fast, but isn't guaranteed to make it to the client. Typically you use these types of packets if you're transmitting lots of information which overwrites itself very quickly. eg. A running player's position. If you leave it off and you use SendQuickly, it will simply always use TCP instead.

- Connect On Start - Will trigger the script to attempt to connect as soon as you press the play button in the Unity editor. If turned off, a UI button will have to trigger the connection manually.

```
static public TNAutoJoin instance;
```

The script starts off by creating a static public instance variable to turn the class into a singleton which is accessible from anywhere in your game.

```
void Awake ()
{
	if (instance == null)
	{
		instance = this;
		DontDestroyOnLoad(gameObject);
	}
}
```

If the instance variable is null, then this copy of the script is written into it and then this GameObject is set to Don't destroy on load so that it will stick around the entire time you're connected.

If the code encounters a second `TNAutoJoin` script, the `.instance` variable won't overwrite and calling `TNAutoJoin.instance` will still refer to the original one.

```
void OnEnable ()
{
	TNManager.onConnect += OnConnect;
	TNManager.onDisconnect += OnDisconnect;
}

void OnDisable ()
{
	TNManager.onConnect -= OnConnect;
	TNManager.onDisconnect -= OnDisconnect;
}
```

The `OnConnect` and `OnDisconnect` functions are hooked up to the `TNManager.onConnect` and `TNManager.onDisconnect` callbacks. They're added in the `OnEnable` and removed in the `OnDisable` functions to make sure that they're always trapped as soon as they possibly can be.

```
void Start () { if (connectOnStart) Connect(); }
```

If `connectOnStart` is true, connect in the `Start()` function.

```
public void Connect ()
{
	// We don't want mobile devices to dim their screen and go to sleep while
the app is running
	Screen.sleepTimeout = SleepTimeout.NeverSleep;

	// Connect to the remote server
	TNManager.Connect(serverAddress, serverPort);
}
```

The `Connect()` function is a `public` function just in case `connectOnStart` is set to false and it needs to be called by a button. After being called it disables the screensaver and then attempts to connect to a server using the IP address and port number specified in the inspector.

```
void OnDisconnect ()
{

	if (!string.IsNullOrEmpty(disconnectLevel) &&
		UnityEngine.SceneManagement.SceneManager.GetActiveScene().name !=
	disconnectLevel)
		UnityEngine.SceneManagement.SceneManager.LoadScene(disconnectLevel)
	;
}
```

If a connection wasn't established, load the disconnect level using the scene manager.

```
void OnConnect (bool result, string message)
{
	if (result)
	{
		// Make it possible to use UDP using a random port
		if (allowUDP) TNManager.StartUDP(Random.Range(10000, 50000));
		TNManager.JoinChannel(channelID, firstLevel, persistent, 10000,
	null);
	}
	else Debug.LogError(message);
}
```

If a connection was made, and the result was a success then setup the ports which UDP packets will use, if they're allowed, and load the Example AutoSync scene.

Let's take a closer look at the `JoinChannel` call. Channel ID is the network channel set to 50, `firstLevel` is set to "Example AutoSync", and the channel is set to be persistent. Next the maximum of players in the channel is set to 10,000 so it's effectively everyone, and the channels password is set to null, so anyone can join.

The JoinChannel automatically loads up the next scene, and the game can continue assuming that it will always have a network connection. As soon as the connection is dropped, the player will be sent to the main menu.