**Servers in Unity** **Most games have a specialized "server" build, which contains much the same code as the client, designed to run as a dedicated server. This allows the server to process the same logic as the client.** **Unity, however, does not directly support this concept out of the box. Unity Pro does allow builds to be run in "headless mode", which runs the game without ini- tializing any graphics, resources, but the server runs the exact same code as the client. The game must be designed to operate in both server and client mode.** **To do this, we'll take advantage of a compiler feature known as "conditional compi- lation". This allows us to wrap code in special tags which allows us to strip out en- tire sections of code when compiling. This way, our server-only code will only be in- cluded in server builds, and our client-only code will only be included in client builds.**

**Compiler directives** **The first thing we will do, is figure out how the application knows whether it is a client or a server. We will use a compiler directive to do this.** **If you are using Unity 4, you can go to Edit | Project Settings | Player and under Other Settings is a section that allows you to define these.** **However, for any version prior to Unity 4, you'll have to define these yourself. To do this, create a new text file in the Assets folder and name it smcs.rsp. Open Notepad and type:** **-define:SERVER** **This creates a global symbol define for your C# scripts. You would use the symbol**

**like this:**

**#if SERVER**

**//code in here will not be compiled if SERVER isn't defined**

**#endif** **You might consider writing an editor script which replaces the contents of this file (when compiling for the client, it would replace SERVER with CLIENT, and vice versa). It is important to note that changes to this file will not automatically recom- pile, when changing the file you should save one of your scripts. Your editor script might do this automatically, for example it could call AssetDatabase.Refresh( ImportAssetOptions.ForceUpdate ).** **Now that we can detect whether the application was built as a server or a client, we'll need some way for the server to act as autonomously as possible. The server should have a configuration file which allows the user to set, for example, network settings before the server runs. This book will not cover how to load the config- uration file (XML or JSON are recommended), but once these are loaded the server should immediately initialize and register itself with the Master Server using the data in the configuration file (for example, server name, maximum connections, lis- ten port, password, and so on).**

**Setting up a server console without Pro** **Usually, a game server is a console application. This is nearly possible in Unity if you have purchased a Pro license, by appending the -batchmode argument to the executable (actually, Unity does not create a console window, instead the game simply runs in the background). If you do have Pro, feel free to skip this section. However, if you own a free license, you'll need to get a bit creative.** **We want the server to use as few resources as possible. We can create a script that turns off rendering of the scene when running in server mode. This won't com- pletely disable the rendering system (as running in command line would), but it does significantly reduce the GPU load of the server.**

**using UnityEngine;**

**using System.Collections;**

**public class DisableServerCamera : MonoBehavior {**

**#if SERVER**

**void Update()   {**

**// culling mask is a bitmask – setting all bits to zero means render nothing**

**camera.cullingMask = 0;   } #endif }** **This script can be attached to a camera, and will cause that camera to not render anything when running on the server.** **Next we're going to set up a console-type display for our server. This "console" will**

**hook into the built-in Debug class and display a scrolling list of messages. We'll do this via Application.RegisterLogCallback.**

**using UnityEngine;**

**using System.Collections;**

**using System.Collections.Generic;**

**// contains data about the logged message**

**struct LogMessage {   public string message;   public LogType type; }**

**public class CustomLog : MonoBehavior {**

**// how many past log messages to store**

**public int MaxHistory = 50;**

**// a list of stored log messages**

**private List<LogMessage> messages = new List<LogMessage>();**

**// the position within the scroll view**

**private Vector2 scrollPos = Vector2.zero;**

**void OnEnable()   {     // register a custom log handler**

**Application.RegisterLogCallback( HandleLog );   }**

**void OnDisable()   {**

**// unregister the log handler**

**Application.RegisterLogCallback( null );   }**

**void OnGUI()   {**

**scrollPos = GUILayout.BeginScrollView( scrollPos, GUILayout.- ExpandWidth( true ), GUILayout.ExpandHeight( true ) );**

**//draw each debug log – switch colors based on log type**

**for( int i = 0; i < messages.Count; i++ )     {**

**Color color = Color.white;**

**if( messages[i].type == LogType.Warning )**

**{         color = Color.yellow;**

**}**

**else if( messages[i].type != LogType.Log )**

**{         color = Color.red;       }**

**GUI.color = color;**

**GUILayout.Label( messages[i].message );     }**

**GUILayout.EndScrollView();   }**

**void HandleLog( string message, string stackTrace, LogType type )   {     // add the message, remove entries if there's too many**

**LogMessage msg = new LogMessage();**

**msg.message = message;**

**msg.type = type;**

**messages.Add( msg );**

**if( messages.Count >= MaxHistory )     {       messages.RemoveAt( 0 );     }**

**// scroll to the newest message by setting to a huge amount**

**// will automatically be clamped**

**scrollPos.y = 1000f;   } }**

**Now the user can see the debug information being printed as the server runs—very useful indeed.** **You should strive for as much code reuse as possible in fact, if your game allows players to host a game from inside the client, most of the same code will already work with a few minor differences:**

**•As previously mentioned, the server starts up automatically with a config-**

**uration loaded from the user-editable files (unlike the client).**

**•The server does not spawn any player objects of its own, unlike the client.**

**•The server does not have any UIs or menus to display to the user beyond the**

**log dump. Beyond starting up the server and shutting it down, there is zero interaction with the server application.**

**Loading networked levels** **There are a few tricks to loading networked levels in the Unity game engine. If you just use Application.LoadLevel, you'll encounter a number of issues; specifically you may find that a client connecting to the game won't see any objects that were instantiated via Network.Instantiate. The reason for this is because the level loading process doesn't happen instantly—it actually takes two frames to complete. This occurs after the list of networked objects was received, so the load process will delete them.** **Note that Application.LoadLevel is purely client side. Unity imposes no limitations on which level a client or server loads in a networked game. In fact, it's entirely pos- sible that you might have different levels within a networked session, and this is what Network.SetLevelPrefix is for. Each of these levels is assigned some kind of "ID" that uniquely identifies the level. Before loading the level you would use Network.SetLevelPrefix. This essentially separates players into channels, so all**

**players with level prefix 0 are separate from players with level prefix 1, for example.** **Note that if your game needs all clients to load the same level, you'll have to ensure this yourself. If a client has a different level loaded than the host, without setting the level prefix to something different than the host, the client might see some odd situations, such as players floating or sunk into the ground (a player could be standing on a bridge in one level, and a different level at the same position might have a building; so the player would appear to be clipped into the building).** **The correct way to load levels in a networked game, is to first disable the network queue, load the level, wait two frames, and then re-enable the network queue. This means any incoming messages will not be processed, and will instead be buffered until the new level has completely finished loading.** **Let's write a simple network level loader that will handle all of these for us. It's**

**designed as a singleton so we don't need one present in the scene (one will au- tomatically be created):**

**using UnityEngine;**

**using System.Collections;**

**public class NetworkLevelLoader : MonoBehavior   {**

**// implements singleton-style behavior**

**public static NetworkLevelLoader Instance   {     get     {**

**// no instance yet? Create a new one**

**if( instance == null )       {**

**GameObject go = new GameObject( "\_networkLevelLoader" );**

**// hide it to avoid cluttering up the hieararchy**

**go.hideFlags = HideFlags.HideInHierarchy;**

**instance = go.AddComponent<NetworkLevelLoader>();**

**// don't destroy it when a new scene loads**

**GameObject.DontDestroyOnLoad( go );       }       return instance;     }   }**

**private static NetworkLevelLoader instance;**

**public void LoadLevel( string levelName, int prefix = 0 )   {     StopAllCoroutines();**

**StartCoroutine( doLoadLevel( levelName, prefix ) );   }**

**// do the work of pausing the network queue, loading the level, waiting, and then unpausing**

**IEnumerator doLoadLevel( string name, int prefix )**

**{     Network.SetSendingEnabled( 0, false );**

**Network.isMessageQueueRunning = false;**

**Network.SetLevelPrefix( prefix );**

**Application.LoadLevel( name );**

**yield return null;**

**yield return null;**

**Network.isMessageQueueRunning = true;**

**Network.SetSendingEnabled( 0, true );   } }**

 **You can now replace any calls to Application.LoadLevel with NetworkLevelLoader.Instance.LoadLevel. For example, the server might call an RPC which loads the level via the helper class we just wrote, as a buffered RPC so that all clients connecting will automatically load the level.**  **Note** **If your server needs to change level during the connection, for example, in many FPS games players can vote on a new map at the end of a round, things get a bit more complicated. The server should first delete all networked objects belong- ing to players, remove RPCs from all players (via Network.RemoveRPCs), and**

**then call the load-level RPC.**