Unity networking: the Zero to Hero guide

Автор: Mike Hergaarden

Переводчик: Дьяченко Роман (Zaicheg)

Редакция оригинала: 29 октября 2009

Редакция перевода: 17 июля 2010

Оглавление

Tutorial 1. Connect & Disconnect

Начнём.

- Откройте первую сцену урока: «Tutorial 1/Tutorial_1». Эта сцена содержит камеру, геймобъект со скриптом Connect и геймобъект, отображающий название сцены через GUIText;
- Соберите билд (любой, но в туториале указан веб-билд) и запустите его;
- Запустите сцену в редакторе и нажмите «Start a server» (используя предустановленные значения IP и port);
- В вебплеере нажмите «Connect as client»;
- Вы должны увидеть «Connection status: Client!» и «Connection status: Server!» в ваших двух запущенных приложениях. Поздравляем с первым коннектом.

Это всё очень просто и код тоже прост. Посмотрите скрипт «Tutorial 1/Connect.js». Весь код, использованный в туториале содержится в функции OnGUI().

Две переменные в начале скрипта (connectToIP и connectPort) используются для приёма из ГУИ вводимых пользователем значений. GUI-функция разбира на четыре части: для сервера, для подключённых клиентов, для подключающихся клиентов и для отключённых клиентов. Мы используем Network.peerType для простого определения статуса подключения данного экземпляра приложения. Мы вызываем Network.Connect для подключения клиента к серверу, эта функция принимает IP, порт и опционально пароль. Для старта сервера мы используем функцию Network.InitializeServer. Она принимает порт и максимальное количество подключений. Есть ещё одна важная переменная — Network.useNat, сейчас объясним подробнее.

NAT-соединение (Network.useNat)

> Мы настроили Network.useNat = false, потому что мы не используем Network Adress Translation (преобразование сетевых адресов). NAT полезно для клиентов, которые находятся за роутером (в LAN, локальной сети). В нашем приложении предполагается работа только внутри LAN.

Следующие за OnGUI десять функций автоматически вызываются Unity. Они не являются необходимыми — вы можете даже удалить их. Через эти функции мы отслеживаем положение дел. Последние три из них стоит выделить особо. OnFailedToConnectToMasterServer вызывается, когда клиент не может подключиться к мастер-серверу и принимает информацию об ошибке. OnNetworkInstantiate вызывается при инстансировании объектов, мы разберём это чуть позднее. OnSerializeNetworkView — это один из двух методов для обмена сообщениями между клиентом и сервером. RPS вызываются как сетевые сообщения/функции, которые вы определяете самостоятельно. В следующей главе мы рассмотрим вызовы RPC.

В заключение, ссылка на раздел референса по сети:

http://unity3d.com/support/documentation/ScriptReference/Network.html

Tutorial 2: Sending messages

Tutorial 2A: Server plays, client observes, no instantiating.

Tutorial 2/Tutorial2A1

Откройте сцену «Tutorial 2/Tutorial 2A1» Код подключения из первой главы прикреплён к геймобъекту Connect. Геймобъект PlayerCube несчёт скрипт Tutorial_2A1.js и компонент Network View. Каждый геймобъект, который получает или отправляет сетевые сообщения, должен содержать компонент Network View. В принципе, можно использовать лишь один ГО с этим компонентом и всегда ссылаться на него из скриптов, но особого смысла нет — просто назначайте этот компонент всем геймобъектам, которые должны участвовать в сетевом взаимодействии.

Запустите демо с сервером и клиентом. Клиент будет видить как сервер перемещает куб (через клавиши WASD). Вся магия этого перемещения исходит из наблюдающего NetworkView и кода перемещения. Посмотрите скрипт Tutorial_2A1.js. Этот код запускается только на сервере (используется проверка Network.isServer). Когда сервер использует клавиши перемещения, куб двигается. Если будут лаги — не обращайте внимания, сейчас мы разбираем другие вопросы.

Итак, как клиент узнаёт о перемещениях объекта сервером? Посмотрите компонент Network View на кубе. В поле observed установлено значение — трансформ куба. То есть, компонент наблюдает за трансформом куба и юнити автоматически посылает информацию об изменениях в трансформе. В данном случае отправка информации идёт от сервера к клиенту. Сервер является владельцем Network View. Клиент лишь принимает информацию.

Рассмотрим свойства компонента Network View. В данном случае его свойство «State synchronization» установлено в положение «Reliable comporessed». Это означает, что он только отправлет значения наблюдаемого объекта, если эти значения изменяются. Если сервер не перемещает куб 15 минут, то все эти 15 минут не посылаются никакие данные. Значение «Unreliable» заставит посылать данные регулярно, независимо от наличия изменений. При значении «Off» синхронизация производиться не будет вообще. Если ваш networkview не наблюдает ни за одним объектом, он не посылает данные и синхронизацию стоит установить в «Off». Если вы удивлены, зачем может понадобиться такой networkview, то поясняем: RPC (Remote Procedure Calls, Вызов удалённых процедур) нуждается в networkview, но не использует свойства state synchronization и observed. Но вы можете использовать RPC вместе с наблюдаемым объектом. RPC будут объяснены в главе 2A3, пока остановимся на том, что это сетевые сообщения (вызовы), которые вы сами определяете.

Tutorial 2/Tutorial 2A2

Что, если требуется переместить куб по оси Y? Откройте и запустите «Tutorial 2/Tutorial 2A2». Игра запустилась таже самая, но на заднем плане теперь работает немного другой код. Networkview на ГО PlayerCube теперь наблюдает за скриптом «Tutorial_2A2.js». Это значит, что теперь networkview ищет функцию OnSerializeNetworkView в скрипте.

Посмотрите на эту функцию: мы теперь явно определяем, что именно нужно синхронизировать.

В этой функции вы сами указываете, какие данные хотите синхронизировать. И опять-же, только изменившиеся значения передаются при включенном «Reliable delta compressed». Функция OnSerializeNetworkView всегда выглядит странновато. Она нужна и для отсылки и для приема данных. Unity определяет можете ли вы отправлять данные (istream.isWriting), проверяя являетесь ли вы владельцем NetworkView. В противном случае, вы можете лишь принимать данные (строчки после else).

Tutorial 2/Tutorial 2A3

А вот и метод отправки сообщений, который мне нравится более прочих: RPC (Remote Procedure Calls, Вызов удалённых процедур). Я уже упоминал RPC ранее, теперь же рассмотрим их подробнее. Запустите сцену «Tutorial 2/Tutorial 2A3». Это демо работает примерно также как и прошлые два. Networkview ни за чем не следит («Off» и пустая ссылка). В скрипте Tutorial_2A3.js прописана строка: *networkView.RPC("SetPosition", RPCMode.Others, transform.position);*

RPC вызываются сервером. В результате во всех прочих клиентах будет вызван метод SetPosition с параметром типа Vector3 (в этот параметр сервер отдаст transform.position как видно из скрипта). Вот как это работает:

- Игрок на сервере нажимает клавишу перемещение и перемещает своего игрока (строки 14-18);
- Сервер проверяет изменение позиции и если оно больше прогового значения, то сервер посылает новую позицию всем клиентам (строки 20-25);
- Все клиенты принимают RPC SetPosition с параметром, установленным сервером, в результате вызывает код функции SetPosition (геймобъекту присваивается новая позиция);
- Куб теперь находится на той позиции, в которую его передвинул сервер.

Для включения/разрешения вызова функции через RPC нужно добавить перед ней код @RPC в JS или [RPC] в C#. Когда отправляете RPC, указывайте, кто его принимает.

RPCMode.Server — только сервер

RPCMode.Others — все, кроме того, кто послал RPC

RPCMode.OthersBuffered — все, кроме того, кто послал RPC. Буферизировано

RPCMode.All — все

RPCMode.AllBuffered — все. Буферезировано

Что значит «буферизировано»? Это значит, что данный RPC получит каждый игрок, который подсоединился, даже если он подсоединился после отправки RPC.

Tutorial 2B: Serever and client(s) play, with instantiating

Сейчас мы разберёмся, как включить в процесс нескольких игроков и позволить им управлять своими кубиками. Запустите сцену «Tutorial 2/ Tutorial 2B» в редакторе как сервер и в двух веб-плеерах как игроков (коннектимся, то бишь).

ГО PlayerCube был удалён из сцены. Вместо него теперь есть скрипт Tutorial_2B_Spawnscript.js, добавленный к гейобъекту Spawnscript, и мы инстансируем префаб, определённый в скрипте. Network.Instantiate принимает позицию, вращение и группу как аргументы. Мы копируем позицию и вращение объекта Spawnscripts и выставляем группу 0 (сейчас просто игнорируем этот параметр). При дисконнекте игрока наш скрипт удалит инстансированный объект. Тот, кто вызвал Network.Instantiate становится автоматически хозяином объекта. Соответственно, это игрок и управляет своим кубиком.

Tutorial 3: Authoritative servers

Сервер, настроенный в последнем примере, называется «non-authoritative» (неавторитарный). Не было никакой проверки на стороне сервера, клиенты расшаривали свои позиции и каждый клиент принимал сообщения от других и верил им. В многопользовательском FPS нельзя позволять игрокам через редактирование сетевых пакетов (или игры напрямую) получать преимущество (телепорт, ускорение и т. п.) Для этого используется авторитарный сервер. Его установка не требует сложного кода, но требует особого подхода к дизайну кода. Вам нужен сервер для того чтобы проверить все коммуникации.

Давайте подумаем, как сделать последний пример (2B) авторитарным. Прежде всего, серверу нужно создавать игроков, а игроки не должны решать, где они появляются и когда. Во-вторых, сервер должен указывать правильные позиции объектов игроков (самим игрокам это теперь непозволительно). Клиент лишь посылает запрос на движение, а сервер решает, как его осуществить.

Клиент посылает серверу запрос на движение в соответствии его вводом (нажатиями клавиш), сервер выполняет запросы и посылает результат (новую позицию) обратно всем клиентам.

Посмотрите сцену «Tutorial/Tutorial3». Внешне всё работает также, но внутренние механизмы изменились. Появились лаги в движении, но это сейчас не имеет значения.

Оперируем мы тут двумя скриптами: Tutorial_3_Playerscript и Tutorial_3_Spawnscript.

Начнём с Tutorial_3_Spawnscript.js. Клиент больше ничего не делает в этом скрипте, сервер сам запускает всё, когда клиент подсоединяется. Кроеме того, сервер сохраняет список подключённых игроков с их Playerscript-ми для того, чтобы удалять правильные объекты, когда игрок отключается. Spawnscript был бы стопроцентным серверным сценарием, если бы не функция OnDisconnectedFromServer, которая всё-таки клиентская.

Перейдём к Tutorial_3_Playerscript.js. Этот скрипт теперь выполняется не только владельцами networkview-объектов. Так как сервер инстансировал все эти объекты, он является владельцем всех networkview. Поэтому мы теперь используем переменную для обнаружения и запоминания того, какому игроку (клиенту) принадлежит объект. Владелец playerscript-а посылает запрос на перемещение на сервер. Сервер выполняет все playerscript-ы, чтобы обработать перемещение и переместить игроков. У нас теперь полностью авторитарный сервер.

Вернёмся к вопросу лагов (задержек). В последнем примере (авторитарный сервер) у нас проходит некоторое время между отправкой запроса на движение и получением новой позиции. Мы можем нивелировать эту задержку, если клиент будет считать движение сразу, а сервер будет перезаписывает посчитанное им (сервером) движение поверх клиентского. Сервер по-прежнему будет авторитарным. Как это делается.

This is quite easy to add. In “Tutorial_3_Playerscript.js” simply have the client execute “SendMovementInput(HInput, Vinput);”where you are sending the movement RPC (uncomment line 56) . Then make sure that the SendMovementInput RPC call actually affects

the client by updating the (server) movement code in the bottom of the Update() function; also run it on the local player too by adding “|| Network.player==owner){" in the IF statement (see line 64). These two edits will now make sure the clients movement is applied right away, but the servers calculations will still be ultimate in the end.

After applying the client “prediction” the movement will still look a bit laggy, to improve this check out line 100, here's a snippet to “merge” the clients current position and the servers position, with the servers position having more weight. You can take this a step further by saving the real server position in a variable and “lerp”(See Vector3.Lerp) to this position in the Update loop instead of only Lerping once in the OnSerializeNetworkView function (which is executed far less than Update).

Нужно заметить, что вы не обязаны делать все свои сетевые игры по принципу авторитарного сервера. Например, наша игра «Crashdrive 3D» сделана неавторитарной. игрок мог изменять позицию автомобиля напрямую, но кого это волнует? И не забывайте, что даже авторитарный сервер можно обмануть.

Further network subjects explained

Unity editor options related to networking

«Edit → Project settings → Network»

- Sendrate: определяет, сколько раз в секунду будут посылаться сообщения об изменениях наблюдаемых (observed в networkview) объектах. Это не влияет на сообщения RPC. Уменьшение значения приводит к экономии трафика (и снижению точности работы приложения).
- Debug level: определяет, как много сетевой отладочной информации покажет лог редактора.

«Edit → Project settings → Player»

- Run in background: когда приложение запущено в роли сервера, оно требует включение этой опции для постоянной поддержки связи с клиентами.

Limiting traffic: Scoping and group limiting

Вы можете улчшить «сетевую производительность», ограничив количество пересылаемых данных. В сетевых играх игроки не должны получать каждый бит информации (всю информацию о происходящем на сервере). Он должен получать только то, что имеет для него значение. Есть две методики.

Первая. У компонента Network View есть функция SetScope:

function SetScope (player : NetworkPlayer, relevancy : bool) : bool

Это true для каждого игрока по умолчанию. Вы может установить false, если networkview далеко от игрока, и этот игрок больше не будет получать сообщения от данного networkview. Однако, это работает только для наблюдаемого (observed) значения, а RPC будут посылаться по-прежнему.

Второе. Сетевые функции:

static function SetReceivingEnabled (player : NetworkPlayer, group : int, enabled : bool) : void
static function SetSendingEnabled (group : int, enabled : bool) : void

Эти две функции могут использоваться для ограничения отсылки/приёма, основанного на сетевых группах. Вы можете к примеру разделить вашу карту на 32 тайла (участка) и посылать/принимать информацию только тем пользователем (и от тех пользователей) которые находятся на расстоянии 8 тайлов от рассматриваемого игрока. Проблема в том, что групп может быть не более 32-х.

Securing the network connection

Добавление AES-шифрования, CRC, случайного шифрования SYNCookies и RSA-шифрования звуков — это сложно? К счастью, достаточно одной строки кода, чтобы внедрить это всё в игру.

```
function StartServer()
{
```

```
Network.InitializeSecurity(); // вся магия
Network.InitializeServer(32, 25000);
}
```

Вызывать InitializeSecurity() нужно перед инициализацией сервера. Этот режим безопасности добавит по 15 байтов на пакет в трафик.

Anti cheating

Даже с безопасностью от InitializeSecurity нужно предпринимать дополнительные меры предосторожности. Предположим, игрок знает, какой код вы используете, и он может редактировать сетевые пакеты под свои нужды. Таким образом, надо всегда проверять значения, которые мы получаем в этих пакетах. Тут не требуется много дополнительного кода — скорее, правильный дизайн.

Using a proxy

Отправляем вас в справку. Уж извините.
http://unity3d.com/support/documentation/ScriptReference/Network-useProxy.html

Combat Lag: prediction, extrapolation and interpolation

Мы уже вкратце обсудили предсказание лагов в Tutorial 3. Когда есть авторитарный сервер, который принимает окончательные решения, стоит также использовать предварительный расчёт решения на клиенте.

Вот выдержка из мануала Unity:

«Экстраполяция состоит в том, что несколько последних (буферизированных) известных значений (например, позиций противника) используются для того, чтобы предсказать следующее значение.
Интерполяция состоит в следующем. Иногда пакеты теряются. Предположим, пришёл пакет №1 с первой позицией противника, пакет №2 с "промежуточной" позицией потерялся, а пакет №3 с "новой" позицией пришёл. Происходит резкий скачок от первой позиции к третьей. Интерполяция как раз позволяет предположить перемещение противника между первым и третьим пакетами. В результате, перемещение сглаживается»

Примеры экстраполяции и интерполяции доступны в официальном примере «Network Example» и также присутствуют в главе «Example 4: FPS Game» этого урока.

Manually allocate networkview ID's

Иногда Network.Instantiate не работает как надо на авторитарном сервере. Чтобы получить больший контроль над этим процессом, можно распределять ID для networkview самостоятельно.
Пример кода:
http://unity3d.com/support/documentation/ScriptReference/Network.AllocateViewID.html

Network loading

Для сети не имеет значения, что запущено на каждом серверном/клиентском компьютере в процессе сетевого взаимодействия. Может быть сервер, на котором запущена игровая сцена, пока новый клиент ещё только подключается через сцену-лобби. Обычно тут нет

никакой проблемы за исключением случаев, когда сервер пытается послать клиенту все буферизированные инстансы геймобъектов. По этой причине может быть лучше временно прекратить сетевое взаимодействие с клиентом, пока клиент загружает игру. Это можно реализовать через Network.isMessageQueueRunning = false; на стороне клиента после того как соединение с сервером было установлено. Пример есть в этом уроке в главе про лобби.

Теперь вы знаете, что сервер может хостить много игровых сессий и уровней посредством умного использования сетевых групп. Только внимательно следите, чтобы игроки из различных сессий не столкнулись между собой.

Real life examples

Example 1: Chatscript

Сцена «Example1/Example1_Chat» содержит скрипт подключения из Tutorial 1 в сочетании с новым скриптом чата. Добавление чата в ваше игру весьма просто и вы можете использовать этот скрипт чата в ваших проектах. В настоящее время чат имеет ограничение на четыре строки, но можно это и изменить. Сервер отслеживает список игроков. В реальной игре вы, вероятно, будете хранить такой список в некоем центральном скрипте.

Example 2: Masterserver example

Сцена «Example2/Example2 menu». В этом примере показано, как можно использовать мастерсервер, чтобы показать все текущие игровые сессии.

Пункт «Qick Play» подсоединяет игрока к случайному доступному сеансу. Пункт "»Advanced» позволяет запустить хост, ввести IP и порт для прямого подключения или использовать сортированный список мастерсерверов для самостоятельного выбора. Недостаёт одной возможности — хостинга (и коннекта), защищённого паролем. Это легко добавляется для прямого соединения и хостинга.

For the game list you'll need to add a password popup window though.The 'game' in this example only shows you the connection status of the server and clients. You could easily replace the game scene with any other scene and the networking will work right away. You'll only need to set “Network.isMessageQueueRunning = true;” since we disabled it in the menu scene to prevent strange things from occurring in the main scene when a client joins a game that's in progress. You'll also need to remember to register the game to the masterserver once the server has started the game scene.

Example 3: Lobby system

Сцена «Example3/Example3_lobby». Этот пример очень похож на Example1, однако тут есть возможность предигрового лобби с паролем. В списке мастерсервера показываются только те игры, которые находятся на этапе лобби. Игры, которые начаты, удаляются из списка.

Example 4: FPS game

В последнем примере мы рассмотрим создание сетевого шутера (FPS). Схема здесь не авторитарная, авторитарную разрабатывайте сами в качестве домашнего задания.

Здесь используется код мастерсервера для установки подключения в главном меню. В игре есть возможности для нескольких игроков: чат, таблица очков, движение, стрельба, подбор предметов. Если хотите, можете использовать это как основу для своей собственной игры. Только вам стоит добавить следующее:

- Авторитарная обработка перемещения;
- Анимация персонажей: надо синхронизировать анимации или считать анимацию на стороне клиента;

- Переключение оружия;
- Прицел, режим наблюдателя, игровые раунды и их лимит.

Tips

OnSerializeNetworkView bug in 2.6 (and earlier)

I never wanted to use the OnSerializeNetworkView networking method (loved RPC so much), and when I finally did it appeared to have a big flaw. This flaw only occurs when you allocate viewid's yourself. An extract of my bug ticket:

«When using OnSerializeNetworkView with an observing networkview (reliable delta compressed in this case), it all works fine with 1 player (as host). As soon as a second client joins, it will complain about not knowing about the first assigned networkviewID of the first player.»

«Received state update for view ID ******random info here about your specific number*** but no initial state has ever been sent. Ignoring message.»

The problem is that these networkviews are bugged: the new client never get the «initial state» of the networkviews that have been assigned BEFORE that new client connected. This affects all clients."

Also see: http://forum.unity3d.com/viewtopic.php?p=77193

Group limit

Вы можете использовать только 32 группы, даже если в коде вы назначите для networkview группу 48. В такой ситуации будет использоваться группа 48 (48 - 32).

RPC bug?

Не позволяйте этой ошибке убивать ваше время: если у вас есть игра, где авторитарный сервер является и игроком, попробуйте использовать такой код:

networkView.RPC("SendUserInput", RPCMode.Server, horizontalInput, verticalInput);

Если же и это не работает, то опробуйте так:

```
if(Network.isServer){
SendUserInput(horizontalInput, verticalInput);
}
else{
networkView.RPC("SendUserInput", RPCMode.Server, horizontalInput, verticalInput);
}
```

Run dedicated servers

Недостаток реализации сети в Unity — выделенный (dedicated) сервер не столь выделенный, как хотелось бы. Однако, он вполне себе работает. Запуск выделенного сервера на Mac OS X возможен, если задать параметр пакетного режима при выполнении. В 2.6 была добавлена аналогичная функция и для Windows.

Подробнее по ссылке:
http://unity3d.com/support/documentation/Manual/Command%20Line%20Arguments.html

Когда запущен выделенный сервер, нужно использовать Application.targetFrameRate для того, чтобы удостовериться, что Unity не пытаается запустить ваш сервер с 1000+ fps, сжирая ваши ресурсы.

Connection issues: How to connect over the internet

Отличие интернет-подключения от подключения внутри LAN-сети в том, что скорость обычно несколько ниже.

Если же игра через интернет не работает, то проверьте следующее:

- Работает ли игра через LAN?
- Оба компьютера имеют работающее интернет-соединение?
- Открыт ли на обоих компьютерах в их файрволлах используемый порт? Можно на время даже отключить файрволлы.
- Попробуйте прямое подключение. Запустите сервер и с другого компьютера произведите подключение по внешнему IP-адресу сервера.
- Если это всё не помогло, то ваш сетевой роутер вероятно пытается блокировать неизвестное входящее подключение как небезопасное. Есть два варианта:
    - использовать NAT для проброса (смотрите пример с мастерсервером). Проверьте, поддерживает ли ваш роутер NAT-пробросы.
    - откройте порт для использования в вашем роутере и/или сделайте форвардинг всех соединени по этому порту на ваш внутренний (LAN) IP-адрес.

Other networking options

Если реализации сетевого взаимодействия в Unity вам недостаточно, можно ипользовать другие решения:

- Собственный бекэнд RakNet;
- Smartfox;
- Photon & Neutron from ExitGames;
- Project DarkStar;
- Netdog;
- Lidgren.