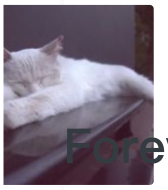
[TUTORIAL] Network Programming Complete Tutorial

By LongFeiaot,
October 4, 2022 in Tutorials and Guides



LongFeiaot

Posted October 4, 2022

Foreword

In Don't Starve Together, there are many issues of data and command interaction between the server and client. In order to ensure the data consistency, the main calculation and storage happend at the server side, and then distributed to each client. On the client side, it mainly deals with the work that must be done on the client side, such as playing animation.

Scenario: A player operates the character to move through the keyboard/mouse. At this time, the client side plays the moving animation, and at the same time sends commands to the server side to calculate the player's moving path, speed and final coordinates. After the server side completes the calculation, it sends the coordinates back to the client side and synchronizes it to other client sides, so that everyone can see the player's same movement path. The actual process is optimized, but the overall logic is consistent.

In this process, the interaction of these server and client sides is involved:

1. the client side sends commands to the server side (keyboard controls movement)

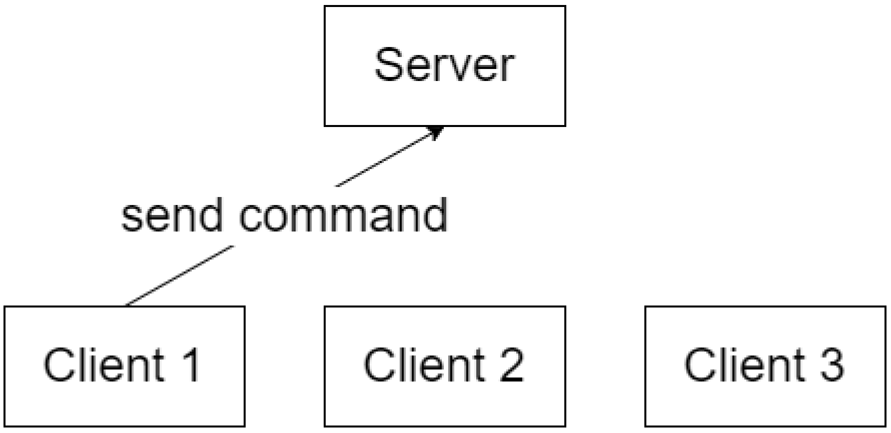
2. the server side synchronize data to all client side (the client side displays the player's new position)
3. When the player's hunger is reduced, it will also be displayed through the UI, which requires reading the hunger value from the client to the server.

We can summarize three typical server–client interaction scenarios

Scenario 1 The client sends commands to the server.

Hide contents

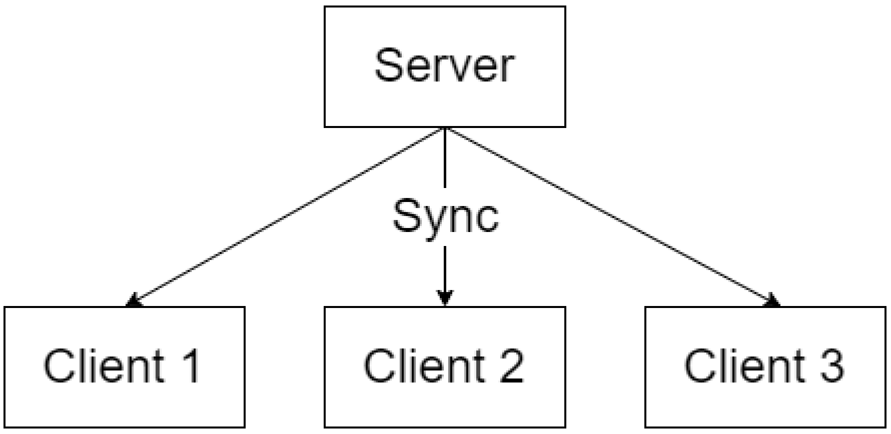
Pressing the button to request the server to move the character's position.



Scenario 2 The server synchronizes data to all client sides.

Hide contents

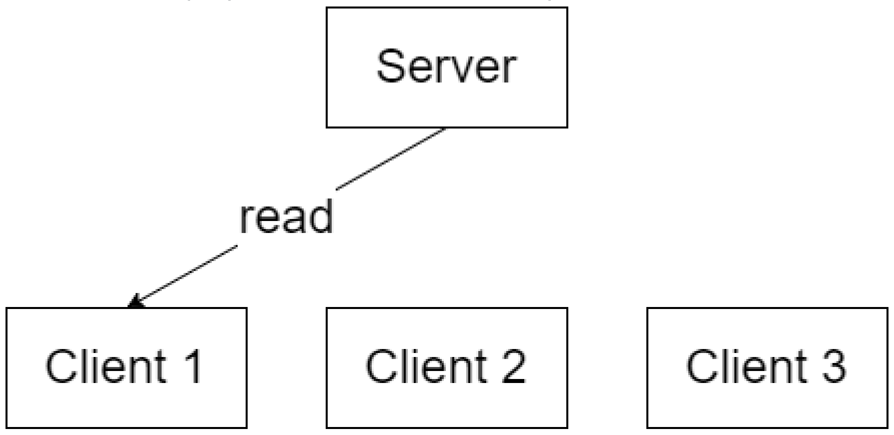
The server synchronizes the character's position to all clients



Scenario 3 The client reads data from the server.

Hide contents

The UI of the player needs to continuously read data from the server in real time



The above three are the most common server–client interaction scenarios, which need to be handled by using network programming related technologies. Below I'll start with the basics of network programming, and then talk about how to deal with these three situations and what the best practices are.

Basic knowledge

Table of contents

netvar and dirtyevent

This part of the content has a lot of detail in the game file netvar.lua, and the content of this section is also written with reference to it.

netvar, or network variable, is used for data sync between server and client. It needs to be defined in the server and client, bound to an entity, and then can update or read data by calling methods, so as to achieve the purpose of data synchronization.

Define a netvar

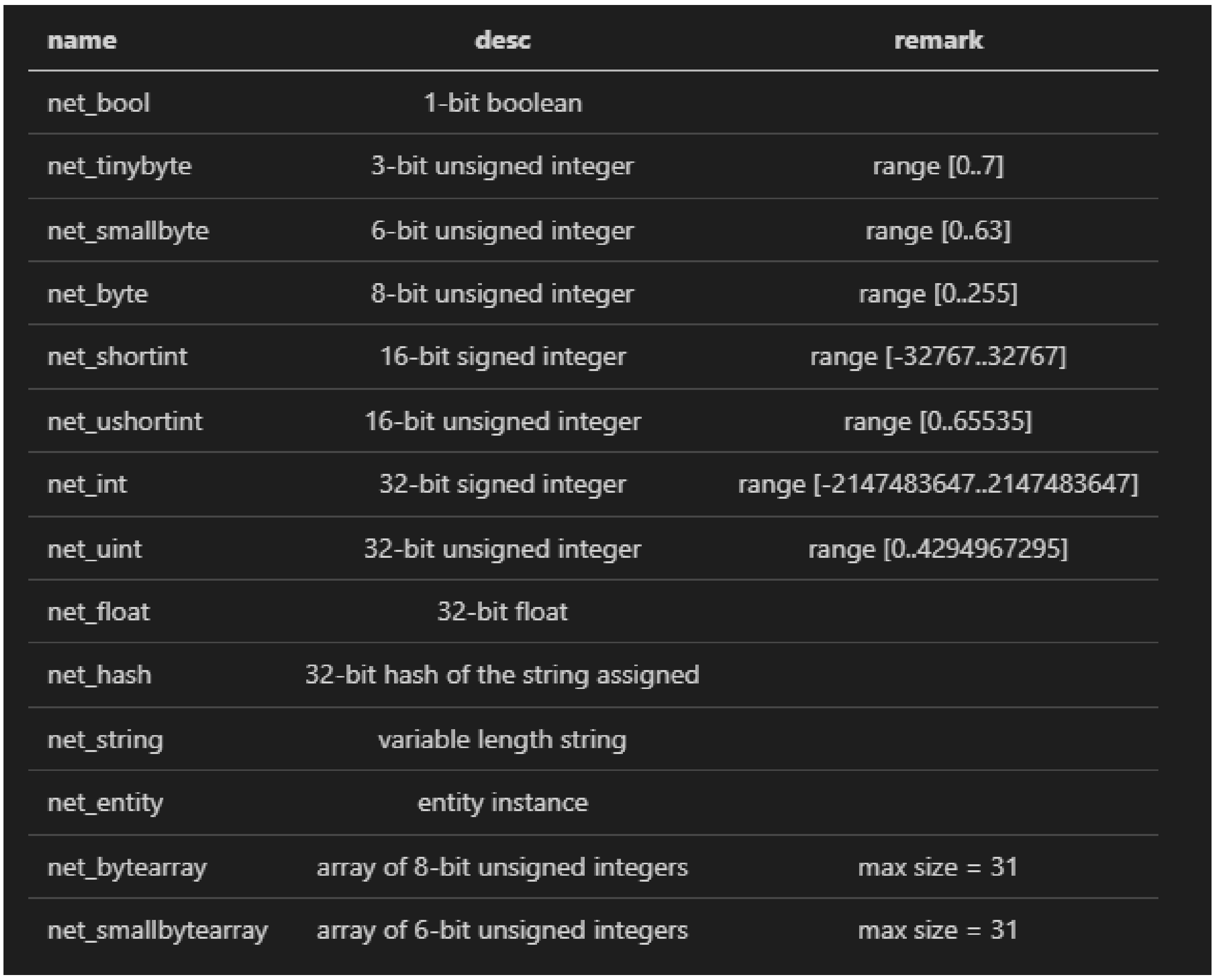
```
inst._mynetvar = net_tinybyte(inst.GUID, var_name, dirtyevent_name)
```

Hide contents

- net_tinybyte is the datatype of netvar, you can refer to the following table to replace it with other ones according to your needs
- Usually, netvar are attached to a common object, such as prefab, replica or classified. Here, inst refers to the prefab object. It is customary to set netvar to be private, that

is, the first name is an underscore. Here is named _mynetvar, attached to inst.
- The three parameters defined are the GUID of the bound entity, the variable name, and the name of the dirtyevent. Both the variable name and the dirtyevent name are required to be unique from other bound netvar in the entity. The dirtyevent name parameter is optional. If you don't need to listen to dirtyevent, you can omit it

The system provides a variety of types to meet different needs. Array is expensive, try to avoid using it.



name	desc	remark
net_bool	1-bit boolean	
net_tinybyte	3-bit unsigned integer	range [0..7]
net_smallbyte	6-bit unsigned integer	range [0..63]
net_byte	8-bit unsigned integer	range [0..255]
net_shortint	16-bit signed integer	range [-32767..32767]
net_ushortint	16-bit unsigned integer	range [0..65535]
net_int	32-bit signed integer	range [-2147483647..2147483647]
net_uint	32-bit unsigned integer	range [0..4294967295]
net_float	32-bit float	
net_hash	32-bit hash of the string assigned	
net_string	variable length string	
net_entity	entity instance	
net_bytearray	array of 8-bit unsigned integers	max size = 31
net_smallbytearray	array of 6-bit unsigned integers	max size = 31

Updete value

```
inst._mynetvar:set(x)
```

Hide contents

- Execute this statement on the server side, the data will be automatically synchronized to all client sides
- When the value of the variable does change, it will trigger a dirtyevent on the server and all clients. You can set a listener callback for this event. The server and client can use different callback functions

Read value

```
inst._mynetvar:value()
```

Hide contents

This statement can be executed on the server or client to get the current value of this network variable.Mostly I use it on client side because I can read data from component on server side

Locally update value

```
inst._mynetvar:set_local(x)
```

Hide contents

- Can be executed on the server or client, this statement is different from set, it will not trigger data synchronization and dirtyevent
- The next time the server executes set(x), regardless of whether the variable value really changes, data synchronization and dirtyevent will be triggered
- set_local is typically used for code execution paths shared by server and client. When the computation of this variable can be done using only the data on the client side, try to use it.
- For example, in order to reduce the impact of network delay, for the movement of characters, you can calculate the movement path on the client side and use set_local to update the value, so that you can first use the calculation results of the client side to move the characters without waiting for the server to sync data (also known as the lag).After the result of the server's calculation arrived, the client performs another move (if the duration is long, you will find that the character has moved a certain distance and then returned to the original point. This is because the calculation result of the server is based on the operation at the time of sending, and the network delay causes a long time diffrence between when the result is sent and when the result is received. On the client side, the character has been operated for a long distance.
- set_local can make some time-based functions smoother for client. The main calculation is on the client side. The server only needs to synchronize the data now and then, and does not need to send it at high frequency, which can save bandwidth resources.

There is also a net_event, which is encapsulation of net_bool. The official recommendation is to use this to trigger some simple functions, such as spawning a fx or play sound effect, which do not involve data transmission. Essentially it is used as a trigger.

Example

Hide contents

```
--define, the parameter is the bound entity's GUID + variable name
inst.hitevent = net_event(inst.GUID, "net_hitevent")

local function do_hit_fx(inst, data)
    ...
end
-- Set the callback function, and limit the operation only to the clie
if not TheNet:IsDedicated() then
    inst:ListenForEvent("net_hitevent", do_hit_fx)
end
-- Trigger an event, when this statement is executed, inst will execut
inst.hitevent:push()
```

RPC

RPC is used for command transmission between networks, such as client to server, server to client, between different worlds of the server, and so on.The general approach used is to first register an RPC handler, and then, when needed, call an RPC.

The following is an example to explain, which is the most commonly used, sending commands from the client to the server

Register RPC

```
AddModRPCHandler(namespace, name, fn)
```

Hide contents

- This statement will register an RPC handler
- Parameter 1 – namespace: namespace. It is a unique identifier that differentiates your mod's RPC from other mods. It is generally recommended to use the mod's name or abbreviation (if your mod's name is too long)
- Parameter 2 – name: RPC name. It is also a unique identifier used to distinguish other RPC in your mod. Personal suggestion is to have the same name or abbreviation as

the following execution function.

- Parameter 3 – fn: The function to execute. The first parameter passed to this function is fixed as the player, and the following parameters depend on the value passed in when calling the RPC

Call RPC

```
SendModRPCToServer(GetModRPC(namespace, name), ...)
```

Hide contents

- This statement will send an RPC call
- GetModRPC(namespace, name) is a fixed way of writing, namespace and name are the values defined when registering above
- ...: other parameter list, RPC calls fn like fn(player, ...)

Let's see a complete example

```
--This function will make your character change the size of the body, size
local function GrowGiant(player, size)
    player.Transform:SetScale(size,size,size)
end

-- Register an RPC named GrowGiant under ModGrowGiant, which will call the
AddModRPCHandler("ModGrowGiant", "GrowGiant", GrowGiant)

--Define a function, call RPC to make the player's size increase, and each
local size = 1
local function SendGrowGiantRPC()
        size = size + 1
        if size == 6 then size = 1 end
        SendModRPCToServer(GetModRPC("GrowGiantRPC", "GrowGiant"), size)
end
-- Define a keyboard event, press the V key to call the function defined al
GLOBAL.TheInput:AddKeyDownHandler(GLOBAL.KEY_V, SendGrowGiantRPC)
```

Klei also provides two new APIs specifically for the server to send commands to the client, or to send commands to a specific world.You can refer to this post (New Modding RPCs API)

replica and classified

This section refers to the discussion in Introduction to Replicas, and is written based on my own practical experience.

replica

Netvar need to be bound to the entity (corresponding GUID), and also need to be easily referenced. In general intuitive thinking, add a property variable as a reference to the prefab to be bound. However, such an approach will lead to strong coupling between Prefab and properties. After adding properties, there are a series of functions written around this property, all of which are very bloated to add to Prefab, so this approach is not recommended.

In this game, attribute data is mainly managed and maintained through components, so a natural idea is to put the reference of netvar in the component. But the problem is that netvar are required to exist on the server and client, and components are usually only added on the server. In order to solve this problem, replica appeared. Its reference form is very similar to that of component, the replica is `inst.replica.xxx`, the component is `inst.components.xxx`, and the programming form is exactly the same. The difference is that replicas exist on both the server and client, so you can define netvar in replicas and bind them to inst corresponding to replica.

The advantage of using replica is that you can first write a new component for a series of operations on your custom properties. If it involves some data that needs to be read on the client side, add a replica to solve it. The code of network programming is all in the replica. Finally, you only need to make a small change to the component and call the method of the replica. In all occasions where the server and client need to be called, the function defined by replica can be used, which can be set to redirect the calling function of the server if it is on the server, and handle it separately if it is on the client. This process also make it easy to transplant DST Mod to DS Mod.You can just change all replica references to components.

In short, the replica can be regarded as a copy of the component, which is used for general requirements on the server and client, such as obtaining attribute data in the UI and so on.

To create a replica of a component, you need to create a {component name}_replica.lua file in the components folder as a definition, and write the code in the way of a component, and finally in `modmain.lua` execute `AddReplicableComponent("{component name}")` to register this replica

classified

In actual Mod practice, replicas are widely used, because most of the time, the additional netvar we need to add are some special attributes of custom characters. But sometimes you may wish to define an attribute that exists not only on characters, but also on other creatures. But you only need to get the value on the character in the UI interface. In this case, classified comes into play. Classified is actually defined as Prefab, which stores some netvar and methods, attached to a specific prefab. For example, all player characters are bound to a player_classified, which stores values such as hunger and sanity that may not only exist on player characters. Compared with replicas, replicas correspond to components. As long as a component is added to a Prefab, its corresponding replica will be automatically added, focusing on processing network interactions of a certain attribute. A classfied corresponds to a certain type of Prefab, and there may be multiple values of different attributes in a classfied from multiple components. If you have some data from different components that need to be read and computed together on the client side, then using classfied is a better choice. There are also special applications where containers can only be opened by one person at a time.

Strictly speaking, both replica and classfied are just a data management model. There are few scenarios in which one can be done and the other cannot be done. The difference is only in convenience.

Since classfied is not used very often, I will not expand it. If you are interested, you can read the above post Introduction to Replicas

Practice Example

Let's look at a practical example. This example demonstrates how to use netvar and RPC, and combine them together using replica.

I made a photosynthesis system for a character, and one of the important variables is photosynthesis, so I named it pho. Now I wrote a new UI, hoping to get this value.

First let's write a component, named pho, file location: scripts\components\pho.lua

```lua
local function on_current(self, current)
    -- When the server assigns a value to current, it calls the assignment
    self.inst.replica.pho:SetCurrent(current)
end

local Pho = Class(function(self, inst)
    self.inst = inst
    self.current = 0
    self.max = 100
    self:Init()
end,
nil,
{
    -- A metamethod that executes this function each time current is assig
    current = on_current,
})

... -- some other functions

return Pho
```

Then write replica, file location scripts\components\pho_replica.lua

```lua
local Pho = Class(function(self, inst)
    self.inst = inst
    self._current = net_float(inst.GUID, "pho._current")
end)

function Pho:SetCurrent(current)
    if self.inst.components.pho then
        -- Update netvar, executed only on the server
        current = current or 0
        self._current:set(current)
    end
end

function Pho:GetCurrent()
    if self.inst.components.pho ~= nil then
        -- Read the value of the component directly on the server
        return self.inst.components.pho.current
    else
        -- Read the value of the netvar on the client
        return self._current:value()
    end
end

return Pho
```

Then in `modmain.lua`, add a simple widget for the player to display the photosynthesis, read the data through the replica method, and register an RPC to randomly assign the photosynthesis.

```lua
-- This is to facilitate debugging, CTRL+R can restart the game, and can be
GLOBAL.CHEATS_ENABLED = true
-- Register pho's replica
AddReplicableComponent("pho")

-- Hang a text over the player's head showing photosynthesis
AddPlayerPostInit(function (player)
    -- Add component pho for players on the server
    if GLOBAL.TheWorld.ismastersim then
        player:AddComponent("pho")
    end
    player:DoTaskInTime(1,function (player)
        local FollowText = GLOBAL.require "widgets/followtext"
        -- Handle this only if the player has a HUD
        if player and player.HUD then
            player.headwidget = player.HUD:AddChild(FollowText(GLOBAL.TALKI
            player.headwidget:SetHUD(player.HUD.inst)
```

```
                player.headwidget:SetOffset(GLOBAL.Vector3(0, -500, 0))
                player.headwidget:SetTarget(player)
                player.headwidget.text:SetColour(1, 1, 1, 1)
                player.headwidget:Show()

                local OldOnUpdate = player.headwidget.OnUpdate
                -- OnUpdate will continuously update the state of UI component
                player.headwidget.OnUpdate = function (self, dt)
                    OldOnUpdate(self, dt)
                    -- Use Replica's GetCurrent to get data. If you want to mi
                    local current = player.replica.pho:GetCurrent()
                    player.headwidget.text:SetString(string.format("photosynthe
                end
            end
        end)
end)

-- RPC registration - use case
AddModRPCHandler("Lesson Network","RandomPho", function(player, num)
    -- In the RPC function, the default is the server environment, and the
    player.components.pho.current = num
end)

-- Press Key R to randomly assign photosynthesis
GLOBAL.TheInput:AddKeyDownHandler(GLOBAL.KEY_R, function()
    local num = GLOBAL.math.random()*100
    SendModRPCToServer(GetModRPC("Lesson Network","RandomPho"), num)
end)
```

Best Practices

- Define a meta function in the component, which will be automatically called when the corresponding variable is assigned a value. In the meta function, call the replica's function for assignment
- Bind netvar in replica, and define an assignment function (SetXX), in which the environment is judged, and the assignment of netvar is only performed on the server.
- Define the value function (GetXX) in the replica, and judge the environment. If it is a server, redirect to the value function in the component. If it is a client, directly perform the value operation of netvar.
- Use the value function of replica in the value occasions general to all servers/clients (such as UI).
- Use RPC when you need to call up the function of the server from the client side (such as key operation). Register first, and then design the appropriate occasion to send the call command.

Example Mod

The complete code in the Practice Example is placed in the github repository dst_mod_tutorial, in the lesson_network folder

References

- [Introduction to Replicas] Some discussions about replica, classified
- [New Modding RPCs API] Official RPC usage instructions

Edited October 5, 2022 by LongFeiaot

+ Quote

Rickzzs, Cunning fox, IronHunter and 8 others 7 2 1 1

This is really a good tutorial. I eventually understand the Replica concept. Thanks! I have read your lesson code too. Looks like you are about to provide more lessons. If so let me know how I can help.

x l a n

+ Quote

Posted January 6

Reply to this topic...

Share

Follow 1

Go to topic listing

Next unread topic

Home > Forums > Don't Starve Together > [Don't Starve Together] Mods and Tools > Tutorials and Guides > [TUTORIAL] Network Programming Complete Tutorial

Unread Content

Mark site read

Theme

Privacy Policy

Contact Us