Introduction to the Chain of Command's Network Services v2.0

by CrashDome

DRAFT 2

Credits

First and foremost, I'd like to thank bn880, Dinger, Spinor, and all the gentleman in CoC for providing this wonderful library. It's provided a great many scriptor with flexibility and functionality in creating MP scripting without the hassle of 'homemade' scripts and 'hodge-podge' work-arounds. Thanks guys... seriously.

And to General Barron for helping me polish this tutorial...

Disclaimers

Everything in this tutorial is what I have understood to be factual at the time I write this. I can certainly be incorrect and welcome comments regarding anything that is not completely factual. Changes in this tutorial may occur without notice because of this.

I will from this point on refer to The Chain of Command's Network Services as simply 'NS' or 'Network Services'. This is an author's choice only to allow for ease of reading and in no way is it meant that the lone term 'NS' is officially associated with the Chain of Command's Network Services. Apologies ahead of time for any confusion.

I will also be using some direct text from the official instructions found in the CoC_NS documentation on The Chain of Command's Website. Please refer to http://www.thechainofcommand.net/ and navigate to the "Documents" section for those instructions.

On with the show....

So you want to create a MP Mission?

Why use Network Services?

First of all, when creating MP missions, you will most certainly run into the limitations of the PublicVariable command at one point or another. By including Network Services into your mission, you can eliminate a ton of code and reduce mission making time. NS has built in functionality that would otherwise take extensive amounts of testing and scripting to perfect for the average mission designer.
PublicVariable is limited in the data types you can send. For example, you cannot send strings across the network with the PublicVariable command. You CAN however send virtually any data type OFP can handle with Network Services. You can also send data to SPECIFIC clients rather than simply broadcast every varaible as PublicVariable does. Additionally, there has been certain cases that PublicVariable command is not to be trusted. I can attest to that personally. In the original Sinews of War v1.0 scripts I developed, I used PublicVariable on over 100 global variables at one time. It was very common that one or more variables failed to reach the clients. It was random and unpredictable. It also took a very long time. After incorporating NS into Sinews of War v2.0, I am happy to boast that not a single value was dropped in transfer to the clients during the initial synchronization.
If I haven't sold you on the value of NS yet, please close this tutorial and proceed to the nearest mental hospital.

OK, I've decided to use Network Services, what do I need to know/have before I include it in my mission?

First, you need a mission idea. Once you've got that, determine if you may require any scripts in your mission at all. By default, most items in the mission editor execute on all clients (such as triggers, radio triggers, etc..) If you can get away with using no scripts, chances are you may not need NS. If you plan on, or know, that you will be using scripts to perform certain functions, I can **guarantee** that NS will help you. You may only use it once... but even in that scenario it cannot hurt.

Second, you actually need the NS files. Please proceed over to

http://www.thechainofcommand.com/docs/

You will see a sub-section titled "COC LIBNETWORK". To the right is a "DOWNLOAD PACK" link. You can find the latest version there.
I suggest following this method since it will probably provide the latest version (version 2 at the time of this writing). There are others floating around and found in other sections of CoC that are definitely older and will not work with this tutorial. Please make sure you have version 2.0 or above.

Using Network Services

Installation

Installation is very easy. There are currently two options to using NS: 1) Addon PBO file and 2) including the source folders/files in your mission folder. Neither option is better in my opinion, but the addon PBO certainly requires that all clients have the PBO installed in order for the mission to run properly. By using the source in your mission folder, you are in essence, providing the data located in the addon PBO in your mission PBO instead. It will increase your mission file size, but only slightly (NS weighs in at about 40k in PBO form).

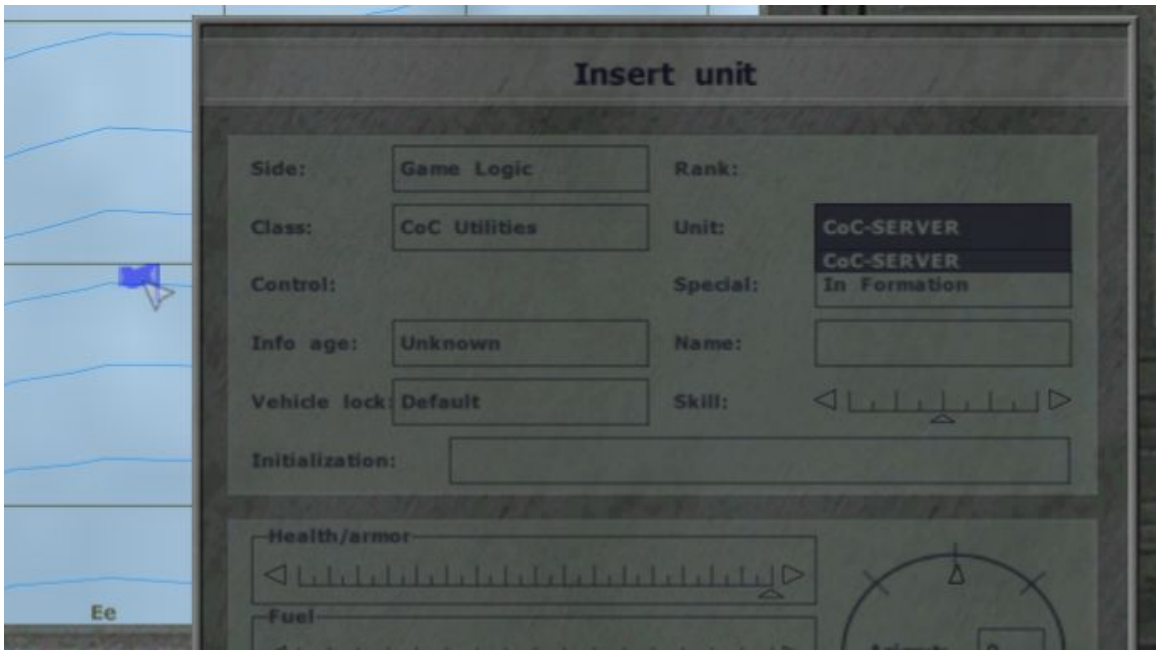
Option 1: Installing the PBO addon file
Extract the CoC_NS zip file and copy the CoC_NS.pbo file into your addon folder.
Wow. That was simple!

Option 2: Installing the source files into your mission folder
First, you need a mission folder. Once you have that, copy the contents of the "CoCNS_2_0_ScriptsCore" folder in the extracted CoC_NS.zip file into your mission folder. Do not copy the "CoCNS_2_0_ScriptsCore" folder itself!! You should ultimately have your mission folder and four sub-folders (arrayConversions, numSystem, Network, and PA) located inside it. Also, you should have copied an init.sqs file. If you already have written an init.sqs file, DO NOT overwrite it. Simply copy the text in the COC_NS init.sqs file into the beginning of your init.sqs file.
That wasn't so bad now, was it?

Setup and Initialization

To get your mission ready to use Network Services, you must include a game logic unit in the mission editor. Name it "CoC_Server" and you are all set to go. If you are using the addon version, it is even simpler... If you begin to add a game logic, you will be able to choose the "CoC_Utilities" game logic unit from the mission editor menu. When you do so, you eliminate the need for any initialization code in the init.sqs file.

[Picture taken from the official NS instructions]



As I stated above, using the addon PBO eliminates the need to edit the init.sqs file. You can provide all initialization code within the INIT line of the game logic unit (discussed later). If you are NOT using the addon PBO, you will have some code in the copied init.sqs file.
It should look like this:

```
; CoC Network Service 2 template initialization init.sqs file
;;CoC_NSFunTable=["myTxFunc1","myTxFunc2"]
;;CoC_DisconnectTime=300
;;CoC_PublicArrays = ["PArray1","PArray2"];
;;CoC_ClientsReady = false;
;;CoC_ConnectTimeout =60

call loadFile "Network\init.sqf"

; INIT CODE NOT DEPENDING ON NETWORK EXECUTES NOW

@CoC_ClientsReady

; INIT CODE CONTINUES FOR NETWORK DEPENDENT CODE
```

If you are using the addon PBO, you will only have one line of the above code... the "@CoC_ClientsReady" line. This is a pausing point in which NS waits for the players to exit the briefing screen before continuing any code of the init.sqs file. This helps prevent any execution of NS commands before it is finished initializing. Enter any code you wish above that line, if you wish it to execute immediately and you are sure you are not using any NS commands. Otherwise, enter any extra/desired code in your init.sqs file should go

after that line.
If you notice, most of those lines above are commented out. For the most part, you will not need them. Some are advanced features I will cover in another section. Some are tweaks that can be applied when NS is not working up to par in your mission. In the addon PBO version, these would go in the INIT line of the game logic unit. In any case, addon or init.sqs, you will not need to do much to get started. Simply leave as is for now and lets get started in actually using NS.

Basic NS Commands

For the beginners, I suggest using a single command - the "**fNRemoteCall**" function. For the most part, NS was a complete script package with many functions that worked in various ways according to needs and desires. Shortly after version 1.2, an interesting OFP work-around was discovered. I won't get into describing what it was, but it gave birth to this **fNRemoteCall** function and rendered most older commands obsolete for the most basic of operations. Since this is an introductory tutorial, I am going to focus on this single command first along with some small relating commands/variables. I will cover additional commands in the Advanced section of this tutorial.

The **fNRemoteCall** function is called with the following syntax:

```
[destination(s):Array, command:String] call fNRemoteCall
```

You will notice there is only two parameters. The first parameter defines your destination machine. The second parameter is the command to execute on the destination machine. Lets look at an example:

```
[ [] , {player addScore 1000}] call fNRemoteCall
```

This example will result in all machines (including the server) adding a value of 1000 to the score of the local player. If you've ever scripted in MP before, you would understand the difficulty in accomplishing this seemingly simple task. Without NS, you would need to add this to a trigger and then fire the trigger somehow. With NS, we've reduced this task to one single line of code.
{But Crashdome, you've left an empty array for the destination?}
I know! That is the rule for telling NS to execute on ALL machines.

{But Crashdome, what about specifying only certain machines?}
Well, lets discuss how NS knows where to send data....

NS 'Nodes'

When NS is initialized, all machines register their name with all other machines. This results in an array called **CoC_ClientList** that is exactly the same for all machines. It might look something like this (in pseudo-code): [Server Array, Player 1 Array, Player 3 Array, Player 5 Array, Player 2 Array, etc...]
Notice that the order the players are in is of no consequence nor is it predefined. The

server, however, will ALWAYS occupy the first slot (or *index 0*). The clients are a first come, first served, situation. If your friend has a better connection, they might always come before you on the list because they can register faster. It's certainly nothing to rely on.
All that technical stuff aside, let's break down what a client can do with this array.
A client has a global variable called **CoC_ClientChannel** which is an integer of what position they are in the array. Let's use this knowledge in another example....

Lets try and send our channel position to everyone and ask that they each send an acknowledgment back only to us.

First we send the call to the entire list that executes an acknowledgment function:

```
[CoC_ClientList, {[CoC_ClientChannel] call loadfile "myFunction.sqf"}] call
fNRemoteCall
```

Since we are also in this list, we should prevent the execution on our own machine. There are many ways to accomplish this, but to keep things simple I am going to prevent execution within the function.

MyFunction.sqs:

```
Private ["_caller"];
_caller = _this select 0;
if (_caller != CoC_ClientChannel) then
{
   [ [_caller] , format[{player globalchat "Node: %1 acknowledges me!"},
CoC_ClientChannel] ] call fnRemoteCall;
};
true
```

Once the above function is called (on every machine), it will quit execution if it is the same node as the caller. This is because the value of **CoC_ClientChannel** is different on every machine (since every machine is in a different position of the **CoC_ClientList** array). If the machine is NOT the caller, they will send back a GLOBALCHAT command which includes *their* node number in the text string. The returned chat will execute only on the original callers machine, yet it will execute once for each acknowledged command (or in this case - once for each additional player and the server).
The original caller might see something like this

Alpha Black 1(Crashdome): Node 2 acknowledges me!
Alpha Black 1(Crashdome): Node 0 acknowledges me!
Alpha Black 1(Crashdome): Node 3 acknowledges me!
Alpha Black 1(Crashdome): Node 4 acknowledges me!

However, none of the other players will see this on their screen, but that is because we

intentionally wanted to send this chat command back ONLY to the original caller.

{Ok Crashdome! I've got it that I need to enter an integer (or array of integers) to send back to specific machines, but what if I don't know what integer another player is?}
Good question! NS offers a great function to help you do that! It is called **fNGetNode** and takes an object as a parameter. This only works when the parameter supplied is another player object. It will return a -1 if the object is not in the list.
Let's use another example that causes a player to chat to another player that they've just been shot by them! This will make use of a *Hit* eventhandler. When a player is hit, they will call a chat command if the player that hit them is another player.

First let's add the eventhandler:

```
player addEventHandler ["Hit", {_this call loadfile "ExpressAnger.sqf"}]
```

ExpressAnger.sqf:

```
private ["_guywhohitme","_channel", "_command"];
_guywhohitme = _this select 0;
_channel = [_guywhohitme] call fNGetNode;
_command = format[{(CoC_ClientList select %1) select 0 globalchat "You bastard! You hit me!"},CoC_ClientChannel];
if (_channel != -1) then
{
    [ [_channel] , _command] call fnRemoteCall;
};
```

If a player shot me, they might see something like:

Alpha Black 1(CrashDome): You bastard! You shot me!

*{OK, I got it, but what does the _command string mean? Can't you just use **Player globalchat "You ...."**}*
Simple really... since I am the one who got shot, I will execute this function. Once I execute the line that defines the _command variable, it should be converted to a string that looks something like this (assuming that I occupy position 1 of the CoC_ClientList):

{ (CoC_ClientList select 1) select 0 globalchat "You bastard!...." }

This command is then sent to the player that hit me via the **fNRemoteCall** function. *CoC_ClientList select 1* is my own personal array in the CoC_ClientList array from using my very own CoC_ClientChannel integer (1) and formatting it into the string. The "select 0" means grab the *object* that refers to me. In semi-psuedo-code it will look like { *CrashDome* globalchat "You..." } when the other player executes the command. This allows it to appear as if I personally said that text rather than as if the opposing player

said it. If I had simply used *player globalchat "..."* then it would look like this on his screen:

Bravo Green 1 (Opponent): You bastard! You shot me!

And that certainly wouldn't make sense to the opponent since HE shot ME!!

{OK, I got that you need to reference an object, but how did you arrive at "select 0"?}
Remember how I spoke of CoC_ClientList as being composed of sub-arrays that are based on each player or server? Well, those sub-arrays are composed of four elements, two are reserved for internal NS use only and the other two are the object reference of the player and the name of the player.
It would look something like this:

[object , name , reserved_NS_variable, connected_variable]

As you can see, "select 0" refers to the object.
With that knowledge, NS opens the doors to many scripters in performing otherwise complex tasks. I will discuss those tasks shortly.

When Not to use NS
{Not use NS! WTF are you talking about Crash? You said NS is better than anything else!?!?}
Yes.. Yes I did. However, there is one brief moment in the OFP engine that the PublicVariable command is better in my opinion. As you have learned, NS waits after the briefing screen before clients begin to register. This can cause a short delay before you can call any NS commands. Let's pretend you want to synchronize weather on all clients. If we use the fNRemoteCall function, sync would take place a fraction of a second or more after mission start. This might cause a visible stutter at the beginning of a mission as the sky goes from sunny to cloudy or vice-versa. In this situation, we are going to utilize PublicVariable *before* NS registers on each machine.
Let's pretend we have a random Fog and Weather value that we want to synchronize. We have added a game logic called "Server" (not related to CoC_Server). I suggest not using the CoC_Server game logic because it IS deleted at some point and is not reliable to check if you are executing code on a server. (More on this later)

Our init.sqs should look like this then:

```
; CoC Network Service 2 template initialization init.sqs file
;;CoC_NSFunTable=["myTxFunc1","myTxFunc2"]
;;CoC_DisconnectTime=300
;;CoC_PublicArrays = ["PArray1","PArray2"];
;;CoC_ClientsReady = false;
;;CoC_ConnectTimeout =60

call loadFile "Network\init.sqf"

; INIT CODE NOT DEPENDING ON NETWORK EXECUTES NOW
WeatherValue = -1
? (local server):WeatherValue = random 1
? (local server):PublicVariable "WeatherValue"
@(Weather > -1)
0 setOvercast WeatherValue

@CoC_ClientsReady

; INIT CODE CONTINUES FOR NETWORK DEPENDENT CODE
```

The above code, you may notice executes before the "CoC_ClientsReady" condition. It should cause immediate weather change at mission start. We utilize the *local server* command to make sure the random number is generated on the server only. Normally, when using NS we have another way of doing so, but since this executes before NS is ready, we must use old-school style.

Executing Client or Server-Only code when using NS

Except for the above situation, we can throw out the old way of using *local server* and those pesky Server game logics. NS has another set of global variables called **CoC_isServer, CoC_isServerClient** and **CoC_isClient**. These are boolean values that you can use in place of the *local* command. They are different on different machines and in different situations. Examples:

Dedicated Server: CoC_isServer = true : CoC_isClient = false : CoC_isServerClient = false
Non-Dedicated Server: CoC_isServer = true : CoC_isClient = true : CoC_isServerClient = true
Client Player: CoC_isServer = false : CoC_isClient = true : CoC_isServerClient = false

Let's pretend again... I like pretending if you haven't noticed yet :)
Anyways, let's pretend that we want to have the server execute a command that spawns an enemy tank randomly. It will spawn at a predefined marker called "eSpawn".

```
; CoC Network Service 2 template initialization init.sqs file
;;CoC_NSFunTable=["myTxFunc1","myTxFunc2"]
;;CoC_DisconnectTime=300
;;CoC_PublicArrays = ["PArray1","PArray2"];
;;CoC_ClientsReady = false;
;;CoC_ConnectTimeout =60

call loadFile "Network\init.sqf"

; INIT CODE NOT DEPENDING ON NETWORK EXECUTES NOW

@CoC_ClientsReady

; INIT CODE CONTINUES FOR NETWORK DEPENDENT CODE
? ((CoC_isServer) && (Random 1 > 0.5)) : "T80" createvehicle
getMarkerPos "eSpawn"
```

The above code will randomly spawn an empty tank with an approximate 50% chance. I am aware we could use the "chance of presence" in the mission editor, but hey... I'm trying to give simple examples here!