Chain of Commands Network Services 2 (CoC NS2)

How the hell does it work?

by kenoxite

> **DISCLAIMER**: *I am not and haven't ever been part of the CoC team, so all the information presented here is purely based on my own experience and research. That means that this guide might contain some errors or misconceptions. With that said, everything I mention here works.*

Why use it?

Because:

- Automatically generates and updates a list of all the players in a MP mission
- It's able to broadcast strings, sides, arrays and multi-dimensional arrays globally, and virtually any data type including those supported by publicVariable (number, boolean, object, group)
- Can send data to all, specific clients, all clients or just the server
- Allows you to execute calls or functions sent from the server to clients and viceversa
- It's able to share global arrays among server and clients
- Reduces the amount of global variables needed for clients in MP
- Eliminates the need of looped scripts and other synchronization methods

Getting it

- The last version can be found here:
  http://www.ofpec.com/forum/index.php?PHPSESSID=h0tkc4h997vpefijgf0tc5el21&action=dlattach;topic=30650.0;attach=5472
  If you get it from somewhere else make sure it's version 2.0, not 1.1. The latter should be avoided, as it's inferior in all aspects.
- Read the included *Intro_to_CoC_NS_D2.pdf* file, by the main coder of *Sinews of War*. It's short and simple, and good to have a general sense of this tool. Note that all his examples are focused on the exclusive use of *fnRemoteCall*, disregarding other methods.

Setting it up

For the addon version:

- NOT NEEDED IF YOUR MISSION USES COC UA: Place the *CoC-SERVER* logic in the editor (found in *Game Logic -> CoC Utilities*)

For the script version:

- Copy the contents of *CoCNS_2_0_ScriptTemplate.intro* to your mission folder. It can be

found inside the CoC NS 2 zip file.
- ONLY IF YOUR MISSION DOESN'T USE COC UA: Place a game logic in the editor and name it *CoC_Server*.

Stuff you should know
The official documentation can be found here:
http://web.archive.org/web/20060112070534/http://www.thechainofcommand.com/docs/

- **Nodes**: Nodes are connected sessions, listed in *CoC_ClientList*. That means that nodes actually refer to players, except for the case of the server when dedicated, which refers to a logic instead. Node index 0 is always the server (can be a player or a logic), node index 1 is the first player client, node 2 the second, etc.

- **CoC_ClientsReady**: Use this to check if CoC NS 2 is fully loaded before executing anything related to it. You'll probably just use it once, in an init script or somewhere along those lines.

- **CoC_ClientList**: 2D array listing all connected nodes. There's another equivalent global var, **CoC_PeerList**, so you can use whichever you like most. It has this format: *[player object, player name, reserved var, connected]*. The server is always at index 0. To retrieve the player object of the first client you would do something like this: *(CoC_ClientList select 1) select 0*.

- **CoC_ClientChannel**: It contains the index of the local node in *CoC_ClientList*. It's stored locally, so its value will be different for each machine.

- **fNSend**: Use this to send stuff to specific nodes. Append *"NO_NQ"* if you want to send it ASAP, as it avoids the queue, and *"NO_DISC"* to ignore disconnected nodes.

- **fNSendAll**, **fNSendClients** **and** **fNSendServer**: Use them to send stuff to all nodes, just the clients or just the server, respectively. All of them share the same syntax and can use the *"NO_NQ"* and *"NO_DISC"* tags.

- **fNSendGlobal**: This seems to be the most optimized function to send data, so use it to send stuff to all or specific nodes when you aren't sending strings or sides, and if you don't need the *"NO_DISC"* and *"NO_NQ"* tags. Otherwise use either *fNSend* or *fNSendAll*.

- **fNRemoteCall**: Use this to execute calls on all or specific nodes. It only works with boolean, number, string, array and multi-dimensional array data types, though.

- **CoC_isClient, CoC_isServer, CoC_isServerClient and CoC_isServerDedicated**: Use them to know if the specific node is a client, a server (without specifics), a listen server or a dedicated server, respectively. Run the checks either locally or via *fNRemoteCall*.

- **CoC_PublicArrays**: Contains a list of all the arrays publicly shared.

- **CoC_NSFunTable**: Array containing all the custom functions you want to use in CoC NS2

There are way more interesting global vars and built-in functions, but those listed here are the ones I actually found more useful so far.

So, how do I use all this?
You have two main ways to share data. The one you'll probably use more often is by sending petitions to execute one of the custom functions referenced in *CoC_NSFunTable*, via *fNSendAll*, *fNSendClients*, etc. The other one is by using the public array system. You can also use *fNRemoteCall*, although it has more limitations than the both mentioned before.

Custom functions
Before getting into custom functions let me tell you that there's a few built-in ones present already, like:

- **fNPing**: Sends a ping request to the specified nodes
- **fNPrint**: Prints the specified data in the specified nodes. For debugging, mainly

All the ones included are:
"fNPing","fNPingr","fNCS","fNUP","fNPrint","fNBlank","fNClearQ","fNCall". Don't create new functions with any of those names, or you'll overwrite them and break all this.

For other specific tasks you'll need to create and load the functions yourself and add them to the CoC_NSFunTable array.

Before anything, you'll need to init the *CoC_NSFunTable* array if you are using the script version. It's initialized automatically in the addon one or if you use CoC UA in your mission. Anyway, I think it's good practice to check if this array exists before touching it:

```
_null=format["%1",_nullstring];
? (format["%1",CoC_NSFunTable]==_null) : CoC_NSFunTable=[]
```

Then you load and add the functions to be used by CoC NS2, like this:

```
? (format["%1",fPlayAnim]==_null) : fPlayAnim = preprocessFile "fu\fPlayAnim.sqf",
CoC_NSFunTable set [count CoC_NSFunTable,"fPlayAnim"]
? (format["%1",fSay]==_null) : fSay = preprocessFile "fu\fSay.sqf", CoC_NSFunTable =
CoC_NSFunTable set [count CoC_NSFunTable,"fSay"]
? (format["%1",fClientChat]==_null) : fClientChat = preprocessFile "fu\fClientChat.sqf",
CoC_NSFunTable = CoC_NSFunTable set [count CoC_NSFunTable,"fClientChat"]
```

Those are just examples. You can add any function you want.

Also, note that I make sure those functions aren't defined already. That's to avoid problems when two script packs which both use CoC NS2 are running together. Otherwise it'd add unneeded entries of functions to the *CoC_NSFunTable* array.

Alternatively you could have done this:

```
fPlayAnim = preprocessFile "fu\fPlayAnim.sqf"
fSay = preprocessFile "fu\fSay.sqf"
fClientChat = preprocessFile "fu\fClientChat.sqf"
CoC_NSFunTable = ["fPlayAnim", "fSay", "fClientChat"]
```

It actually doesn't matter, one way or the other. But if you want to avoid double entries the former method is preferred.

What it does matter is to init all that in all clients and that the functions listed in *CoC_NSFunTable* are in the same order everywhere, so a good place for that would be in the *init.sqs* of your mission.

Public Arrays

You can share arrays publicly, so they can be automatically updated in all nodes and can also be edited by any of them.

The public arrays are stored in the CoC_PublicArrays global var. It must be initialized manually in all nodes and all of them must be identical, so a good candidate is the *init.sqs*.

To init *CoC_PublicArrays* you'd do something like this:

```
globalArray1 = []
globalArray2 = [<stuff>]
CoC_PublicArrays = ["globalArray1","globalArray2"]
```

As you can see, *CoC_PublicArrays* is actually a reference to the arrays you want to be global. The arrays themselves can be either empty or not. You can always change their contents later and broadcast them via *fPublicArray*.

To add a new array to *CoC_PublicArrays* after it's been initialized you'd need to do something like this: [[],{myNewGlobalArray = []; CoC_PublicArrays set [count CoC_PublicArrays, "myNewGlobalArray"]}] call fNRemoteCall

And to remove one: [[],{NameOfTheArrayToDelete = nil; CoC_PublicArrays = CoC_PublicArrays - ["NameOfTheArrayToDelete"]}] call fNRemoteCall

To modify a global array you'd use the built-in fPublicArray function, like this:

```
oneOfTheGlobalArrays = oneOfTheGlobalArrays - [unusedObject]
"oneOfTheGlobalArrays" call fPublicArray
```

This way all the nodes will receive the request to update their respective versions of the *oneOfTheGlobalArrays* array with the content of the one of the node that is sending the request.

Examples

Killed EH

The killed eventhandler is local to the computer the unit belongs to. That means that the server won't be aware of when a player is killed unless some looped scripts and global vars are used (which can be dozens depending on the amount of players).

CoC NS2 allows to simplify all this by sending the message from the client to the server when the player is killed, so the server can act accordingly. This is a way of doing so:

- Initialize the *CoC_NSFunTable* global var:

  ```
  _null=format["%1",_nullstring];
  ? (format["%1",CoC_NSFunTable]==_null) : CoC_NSFunTable=[]
  ```

- Create a custom function that will serve as a parser for the sent killed EH. For this example we'll name it *fKilled.sqf* and will place it in a directory named *fu*.

  The code of the function would look like this:

  ```
  private ["_unit","_killer"];
  _unit = _this select 0;
  ```

```
_killer = _this select 1;
[_unit,_killer] exec "eh\killed_server.sqs";
```

Note that we'll be sending the unit and killer vars to a script named *killed_server.sqs* in the *eh* directory, but we could as well run our killed EH code here.

- Now we load the function this way:

```
? (format["%1",fKilled]==_null) : fKilled = preprocessFile "fu\fKilled.sqf",
CoC_NSFunTable set [count CoC_PublicArrays, "fKilled"]
```

- The killed EH should have been added to the player's unit somewhere, like this: `this addeventhandler ["killed", {_this exec "eh\killed.sqs"}`

- And in *killed.sqs* we put this:

```
_unit = _this select 0;
_killer = _this select 1;
[[_unit,_killer],"fKilled"] call fNSendServer
```

So, when a player client is killed this would happen:

1. The player's killed EH script would run on his computer, and send a petition to the server to execute the *fKilled* function with the *_unit* and *_killer* vars passed.
2. The server would receive the notification and execute the *fKilled* function, which in turn would execute the *killed_server.sqs* script locally

This same system can be used to handle the hit EH, which is also local.

Strings in global variables

Strings are one of the data types not supported by *publicVariable*. With CoC NS2 we can update and synchronize its value to all nodes with something like this:

```
[[],{myStringGlobalVar = "Some text here"}] call fNRemoteCall
```

By using *fNRemoteCall* we'll execute the content in brackets in all nodes, including the server. We could have sent this to specific nodes by specifying the nodes, like:
[[1,4],{myStringGlobalVar =...

While we used a string in this example, this same system would work with global vars that contain any of the other data types supported by *fNRemoteCall* (booleans, numbers, arrays and multi-dimensional arrays).

Notes

Be aware that you won't be able to send objects as parameters with a *fNRemoteCall*. The call executed needs the object to be either local to the receiver or known globally.

Something like this, where *_unit* is defined locally on the server, won't work: *[[],"format [{%1 say %2},_unit,_what]"] call fNRemoteCall*. By formatting *_unit* you will actually send the reference of the unit, not the unit object (the client will try to execute something like *WEST 1-1-A:1 say phraseWhatever*, which obviously won't work).

So, if you want to execute something on a unit use a custom function instead, like this: *[[_unit,_what],"fSay"] call fNSendAll*. The *fSay* function must have been previously added to the *CoC_NSFunTable* array, and it would contain something like: *_unit say _what;*

Alternatively, you could make use of the *CoC_ClientChannel* and *CoC_ClientList* vars, as shown in the examples in the *Intro_to_CoC_NS_D2.pdf* file.

Drawbacks

- CoC NS2 can take a long time to load, particularly for the script version.
- Conflicts with CoC UA in the scripted version if a CoC_Server logic is manually placed in the editor