HOWTO use NS LOG to find a problem

From Nsnam

One of the easiest ways a person can narrow down the location of a problem in ns-3 is by using NS_LOG and our predefined logging components. Typically one uses NS_LOG in a case where the exact nature of the problem is not yet understood. Logging provides a way to narrow down the problem location to an extent where either the log message will indicate the source of the problem orit will be easy to fire up the script in a debugger and take a closer look.

A tried and true method for quickly locating a problem is via binary search. You can use binary search to locate a bug just as easily as you can use binary search to locate an item in the usual sense. In the case of debugging a network simulation script, you will usually first consider the end-to-end path of a packet and then take a look at what is happening half-way. If the problem is visible at the half-way point, you think of what is half-way to that point and take a look there. If the problem is not manifested at the halfway point, you take a look at what is happening half-way again towards the end point. Using this technique, you can quickly narrow down the location of a problem.

Of course, you can do this in a debugger just as easily as with NS_LOG **if you know exactly where to set your breakpoints a priori**. An interesting feature of NS_LOG is that you can really do the same thing without knowing what is going on other than in a general network knowledge sense.

In this HOWTO, I demonstrate how one could use the binary search technique to locate a hypothetical problem in the our first.cc example. As a bonus, look at the end for how to use NS_LOG as a learning tool for understanding the system.

HOWTO use NS LOG to find a problem

The first.cc example uses an echo client on a single client node and an echo server on a single server node. It orchesrates an echo across a point-to-point link. A moment's thought will tell you that the halfway point in this simulation is at the point-to-point link. The ns-3 model is that there is a net device and a channel so you will want to enable logging for the point-to-point channel.

If you know something about the structure of the source, you'll just know to look in

```
src/devices/point-to-point/point-to-point-channel.cc
```

for the NS_LOG_COMPONENT_DEFINE of the logging component. If you don't have that kind of information rattling around in your brain you can just go to the top level directory and do

```
find . -name '*.cc' | xargs grep NS_LOG_COMPONENT_DEFINE
```

This might seem like brute force and awkwardness, but you'll only find a couple of pages of them. If you really don't want to wade through the two pages, you can also narrow the search

```
find . -name '*.cc' | xargs grep NS_LOG_COMPONENT_DEFINE | grep -i point
```

and you'll currently see

```
./devices/point-to-point/point-to-point-channel.cc:NS_LOG_COMPONENT_DEFINE ("PointToPointChannel");
./devices/point-to-point/ppp-header.cc:NS_LOG_COMPONENT_DEFINE ("PppHeader");
./devices/point-to-point/point-to-point-net-device.cc:NS_LOG_COMPONENT_DEFINE ("PointToPointNetDevice");
./internet-stack/ipv4-end-point-demux.cc:NS_LOG_COMPONENT_DEFINE ("Ipv4EndPointDemux");
./internet-stack/ipv4-end-point.cc:NS_LOG_COMPONENT_DEFINE ("Ipv4EndPoint");
```

You can easily see that the log component to enable is, not too surprisingly, **PointToPointChannel**.

Go ahead and enable the log component and run the program to see what's happening.

```
export NS_LOG=PointToPointChannel
./waf --run first
```

You'll immediately see that the packet made it down to the channel from the client.

```
2s PointToPointChannel:TransmitStart(0x634910, 0x6256f0, 0x622510)
2s PointToPointChannel:TransmitStart(): UID is 0)
```

Sidebar:

Once you know the name of the method called, it is nice to be able to find a description of the function you've seen. This is easy to do in our Doxygen. Take a look at http://www.nsnam.org/doxygen-release/index.html (if you are working with a released version) or http://www.nsnam.org/doxygen/index.html (if you are working with the development version). Expand *Class List* in the navigation pane on the left side of the page. You will see a bunch of classes. Search for ns3::PointToPointChannel and click on the link. You will have linked to the class reference page for the PointToPointChannel. If you look at the *Public Member Functions* documentation on this page you will see TransmitStart, which tells you that this method is called to transmit a packet over this channel.

Just interpreting the name found in the log as a class name and method name often works, but notice that the function call log printed in the snippet above looks like a class name followed by a *single* colon and then a method name and parameters. This is not a typo. The single colon is easily overlooked but has a significant meaning. The single colon separator means that what appears to be a class name is actually a log component name. In ns-3 these are often the same thing, but ***not necessarily***. To find the code for *TransmitStart* you culd use the recursive find trick looking for PointToPointChannel::TransmitStart (note the double colon namespace separator in the search term) and you would find it in this case. This is just because the class name and the log component name are identical. Generally that's the easiest thing to do first as was done above. However, if you don't find it this way you will have to search for the log component name and then search for the method name in the .cc file you find. This will give you the real class name and you can use that class name to find the method documentation in doxygen.

If you haven't yet found the documentation for the logging component itself, go ahead and expand *Modules* and then *Core* and then *Debugging* in the navigation pane of the ns-3 doxygen, and then select *Logging* to go to the low-level documentation. If you are unfamilar with this and are interested in a tutorial introduction to logging, there is a section in the ns-3 tutorial (http://www.nsnam.org/docs/release/tutorial.html) called *Using the Logging Module* which you should read.

Now back to the main thread ...

So, now you can infer that a packet has made it from the echo client application down through the protocol stack on the client node, into the net device and has begun being transmitted to the server over the channel. Now, think to yourself, what's halfway up the protocol stack on the server side? Well, how about UDP. Again, the question is, what log component to turn on? The answer if you don't know anything about what's happening in the system is grep, but this time use your general networking knowledge and look for udp.

```
find . -name '*.cc' | xargs grep NS_LOG_COMPONENT_DEFINE | grep -i udp
```

You'll see that we have the following log components containing the case-insensitive string udp:

```
./examples/emu-udp-echo.cc:NS_LOG_COMPONENT_DEFINE ("EmulatedUdpEchoExample");
./examples/realtime-udp-echo.cc:NS_LOG_COMPONENT_DEFINE ("RealtimeUdpEchoExample");
./examples/udp-echo.cc:NS_LOG_COMPONENT_DEFINE ("UdpEchoExample");
./src/node/udp-socket.cc:NS_LOG_COMPONENT_DEFINE ("UdpSocket");
./src/applications/udp-echo/udp-echo-server.cc:NS_LOG_COMPONENT_DEFINE ("UdpEchoServerApplication");
./src/applications/udp-echo/udp-echo-client.cc:NS_LOG_COMPONENT_DEFINE ("UdpEchoClientApplication");
./src/internet-stack/udp-socket-impl.cc:NS_LOG_COMPONENT_DEFINE ("UdpSocketImpl");
./src/internet-stack/udp-l4-protocol.cc:NS_LOG_COMPONENT_DEFINE ("UdpL4Protocol");
```

You can really play this by ear. How about turning on UdpL4Protocol. You know what an L4 protocol is, right? UdpSocketImpl or UdpSocket sound kindof high in the stack, so as a wild guess just turn on UdpL4Protocol.

```
export NS_LOG=UdpL4Protocol
./waf --run first
```

Among other things, you'll see

```
Sent 1024 bytes to 10.1.1.2
2.00369s UdpL4Protocol:Receive(0x635d30, 0x625250, 10.1.1.1, 10.1.1.2)
```

This tells you that a packet from 10.1.1.1 to 10.1.1.2 has made it as far as the UdpL4Protocol::Receive method (which you might not have known existed) and also gives you a method name that you can search for and begin seeing what is happening. Again, grep is your friend

```
find . -name '*.cc' | xargs grep UdpL4Protocol::Receive
```

and you will see

```
./internet-stack/udp-l4-protocol.cc:UdpL4Protocol::Receive(Ptr<Packet> packet,
```

Now you have a file and a method so your knowledge of what is happening is increasing. You can spend some time perusing the file if you like just to get a handle on what is happening at this point in the stack.

Focusing at the problem at hand, you know know that the packet has made it about three quarters of the way from the client to the server. The find and grep for udp above tells you that you really only have the udp socket left between you and the server application. The **UdpSocketImpl** component sounds like a good place to look, so turn it on.

```
export NS_LOG=UdpSocketImpl
./waf --run first
```

You will see

```
Sent 1024 bytes to 10.1.1.2
2.00369s UdpSocketImpl:ForwardUp(0x624900, 0x625a20, 10.1.1.1, 49153)
2.00369s UdpSocketImpl:RecvFrom(0x624900, 4294967295, 0)
2.00369s UdpSocketImpl:Recv(0x624900, 4294967295, 0)
```

The socket gets the packet from 10.1.1.1 and then forwards it up the stack and eventually wants to call the **recv** function which eventually gives the server application the data.

Anyway, I think you get the picture. Binary search for problems using NS_LOG. While you are doing the binary search, look at what is being displayed to help you understand what is going on in the stack. Use grep and find to locate the methods you see and then go read the code to get a better understanding of the system. If you want to understand what *should* be happening, do this exercise using one of the examples and watch the packet flow across a similar case and then go back to your code and see where it diverges from what you saw in the example.

Once you have run the binary search down to the finest granularity using NS_LOG, you can fire up your debugger and zero right in on the problem to within a few method calls. Of course, in many cases, something you did or did not do becomes obvious and you just go fix up your script and move on never having had to fire up gdb or ddd at all.

When everything else fails

This tip is reported in the tutorial (http://www.nsnam.org/docs/tutorial/tutorial_21.html#Enabling-Logging) and it is worth to be reported here as well. When everything else fails, namely, when you have absolutely no clue what is the component that makes the whole program stop, you can try and turn on *all the logging* at once. To do this simply set the `NS_LOG` environment variable as follows:

```
export 'NS_LOG=*=level_all|prefix_func|prefix_time'
```

The `*` wildcard expands to *all* the `LogComponents` defined in ns-3, while `level_all` is the equivalent of `LOG_LEVEL_ALL` in the main simulation script. As a consequence, it is quite clear that this represents an extreme measure that produces tons of output, and as such, should be used as a last resort.

HOWTO use NS LOG as a learning tool

You can also use NS_LOG as a learning tool to dig down into the system. If you want to figure out more about ns-3 routing, one way is to find some routing log components and find out where $#!^ happens.

```
find . -name '*.cc' | xargs grep NS_LOG_COMPONENT_DEFINE | grep -i routing
```

```
./src/routing/olsr/olsr-header.cc:NS_LOG_COMPONENT_DEFINE("OlsrHeader");
./src/routing/olsr/olsr-routing-protocol.cc:NS_LOG_COMPONENT_DEFINE ("OlsrAgent");
./src/routing/global-routing/global-route-manager-impl.cc:NS_LOG_COMPONENT_DEFINE ("GlobalRouteManager");
./src/routing/global-routing/candidate-queue.cc:NS_LOG_COMPONENT_DEFINE ("CandidateQueue");
./src/routing/global-routing/global-router-interface.cc:NS_LOG_COMPONENT_DEFINE ("GlobalRouter");
./src/internet-stack/ipv4-static-routing.cc:NS_LOG_COMPONENT_DEFINE ("Ipv4StaticRouting");
./src/internet-stack/ipv4-global-routing.cc:NS_LOG_COMPONENT_DEFINE ("Ipv4GlobalRouting");
```

Look in examples for a file with the root of the word "routing" in its name.

```
ls examples/*rout*
```

As of this writing, you'll see

```
examples/dynamic-global-routing.cc  examples/mixed-global-routing.cc      examples/simple-global-routing.cc
examples/global-routing-slash32.cc  examples/simple-alternate-routing.cc  examples/static-routing-slash32.cc
```

Why not take a look at something with simple in its name too. It seems reasonable that the log component **Ipv4GlobalRouting** will have something to do with the example file **simple-global-routing.cc** doesn't it? See what happens:

```
export NS_LOG=Ipv4GlobalRouting
./waf --run simple-global-routing
```

When you run this, you will get lots and lots of log messages. You can redirect the run output to a file and you can just start poking around.

Take a look at some output. This is interesting setup information

```
0s Ipv4GlobalRouting:AddHostRouteTo(10.1.1.2, 10.1.1.2, 1)
0s Ipv4GlobalRouting:AddHostRouteTo(10.1.2.2, 10.1.1.2, 1)
0s Ipv4GlobalRouting:AddHostRouteTo(10.1.3.2, 10.1.1.2, 1)
0s Ipv4GlobalRouting:AddHostRouteTo(10.1.2.1, 10.1.1.2, 1)
0s Ipv4GlobalRouting:AddHostRouteTo(10.1.3.1, 10.1.1.2, 1)
0s Ipv4GlobalRouting:AddNetworkRouteTo(10.1.1.0, 255.255.255.0, 10.1.1.2, 1)
0s Ipv4GlobalRouting:AddNetworkRouteTo(10.1.2.0, 255.255.255.0, 10.1.1.2, 1)
0s Ipv4GlobalRouting:AddNetworkRouteTo(10.1.3.0, 255.255.255.0, 10.1.1.2, 1)
0s Ipv4GlobalRouting:AddNetworkRouteTo(10.1.2.0, 255.255.255.0, 10.1.1.2, 1)
0s Ipv4GlobalRouting:AddNetworkRouteTo(10.1.3.0, 255.255.255.0, 10.1.1.2, 1)
```

Further down notice that the timestamps seem to be grouped.

```
8.70113s Ipv4GlobalRouting:RequestRoute(0x648950, 1, 0x7fffe3cded30, 0x65db70, 0x7fffe3cde8e0)
8.70113s Ipv4GlobalRouting:RequestRoute(): source = 10.1.1.1
8.70113s Ipv4GlobalRouting:RequestRoute(): destination = 10.1.3.1
8.70113s Ipv4GlobalRouting:RequestRoute(): Unicast destination- looking up
8.70113s Ipv4GlobalRouting:LookupGlobal()
8.70113s Ipv4GlobalRouting:LookupGlobal(): Found global host route0x649bf0
```

You can probably already get a sense for what must be going on. You can now go and find the code using grep.

```
find . -name '*.cc' | xargs grep Ipv4GlobalRouting::RequestRoute
./src/internet-stack/ipv4-global-routing.cc:Ipv4GlobalRouting::RequestRoute (
```

tells you where to look for some of the global routing-related code. From the log messages you can probably infer the basic operation before you even go look at the code.

You can also see (in the original find) that there are other components **GlobalRouteManager** and **GlobalRouter** that live in a directory called **src/routing/global-routing** -- you can turn on those log components to see what is happening there. Armed with some of this basic contextural information you can also look in the manual where you will find a routing chapter with detailed descriptions of some of the methods you've seen already. Using NS_LOG you can see the API described in the manual at work and follow what is happening in real code.

Anyway, NS_LOG is really a very powerful tool. Most people tend to under-appreciate it since they learned in programming 101 that debugging with printfs is for kids. Don't kid yourself, though. Use every tool available to you. Debugging with printfs in an **intelligent way** can make your life much easier.

Craigdo 02:39, 14 May 2009 (UTC)