Logging

The *ns-3* logging facility can be used to monitor or debug the progress of simulation programs. Logging output can be enabled by program statements in your main() program or by the use of the NS_LOG environment variable.

Logging statements are not compiled into optimized builds of *ns-3*. To use logging, one must build the (default) debug build of *ns-3*.

The project makes no guarantee about whether logging output will remain the same over time. Users are cautioned against building simulation output frameworks on top of logging code, as the output and the way the output is enabled may change over time.

Overview

ns-3 logging statements are typically used to log various program execution events, such as the occurrence of simulation events or the use of a particular function.

For example, this code snippet is from Ipv4L3Protocol::IsDestinationAddress():

```
if (address == iaddr.GetBroadcast ())
  {
    NS_LOG_LOGIC ("For me (interface broadcast address)");
    return true;
  }
```

If logging has been enabled for the Ipv4L3Protocol component at a severity of LOGIC or above (see below about log severity), the statement will be printed out; otherwise, it will be suppressed.

Enabling Output

There are two ways that users typically control log output. The first is by setting the NS_LOG environment variable; e.g.:

```
$ NS_LOG="*" ./waf --run first
```

will run the first tutorial program with all logging output. (The specifics of the NS_LOG format will be discussed below.)

This can be made more granular by selecting individual components:

```
$ NS_LOG="Ipv4L3Protocol" ./waf --run first
```

The output can be further tailored with prefix options.

The second way to enable logging is to use explicit statements in your program, such as in the first tutorial program:

```
int
```

```
main (int argc, char *argv[])
{
  LogComponentEnable ("UdpEchoClientApplication", LOG_LEVEL_INFO);
  LogComponentEnable ("UdpEchoServerApplication", LOG_LEVEL_INFO);
  ...
```

(The meaning of LOG_LEVEL_INFO, and other possible values, will be discussed below.)

NS_LOG Syntax

The NS_LOG environment variable contains a list of log components and options. Log components are separated by `:' characters:

```
$ NS_LOG="<log-component>:<log-component>..."
```

Options for each log component are given as flags after each log component:

```
$ NS_LOG="<log-component>=<option>|<option>...:<log-component>..."
```

Options control the severity and level for that component, and whether optional information should be included, such as the simulation time, simulation node, function name, and the symbolic severity.

Log Components

Generally a log component refers to a single source code .cc file, and encompasses the entire file.

Some helpers have special methods to enable the logging of all components in a module, spanning different compilation units, but logically grouped together, such as the *ns-3* wifi code:

```
WifiHelper wifiHelper;
wifiHelper.EnableLogComponents ();
```

The NS_LOG log component wildcard `*' will enable all components.

To see what log components are defined, any of these will work:

```
$ NS_LOG="print-list" ./waf --run ...
```

```
$ NS_LOG="foo"  # a token not matching any log-component
```

The first form will print the name and enabled flags for all log components which are linked in; try it with scratch-simulator. The second form prints all registered log components, then exit with an error.

Severity and Level Options

Individual messages belong to a single "severity class," set by the macro creating the message. In the example above, NS_LOG_LOGIC(..) creates the message in the LOG_LOGIC severity class.

The following severity classes are defined as enum constants:

Severity Class	Meaning
LOG_NONE	The default, no logging
LOG_ERROR	Serious error messages only
LOG_WARN	Warning messages
LOG_DEBUG	For use in debugging
LOG_INFO	Informational
LOG_FUNCTION	Function tracing
LOG_LOGIC	Control flow tracing within functions

Typically one wants to see messages at a given severity class *and higher*. This is done by defining inclusive logging "levels":

Level	Meaning
LOG_LEVEL_ERROR	Only LOG_ERROR severity class messages.
LOG_LEVEL_WARN	LOG_WARN and above.
LOG_LEVEL_DEBUG	LOG_DEBUG and above.
LOG_LEVEL_INFO	LOG_INFO and above.
LOG_LEVEL_FUNCTION	LOG_FUNCTION and above.
LOG_LEVEL_LOGIC	LOG_LOGIC and above.
LOG_LEVEL_ALL	All severity classes.
LOG_ALL	Synonym for LOG_LEVEL_ALL

The severity class and level options can be given in the NS_LOG environment variable by these tokens:

Class	Level
error	level_error
warn	level_warn
debug	level_debug
info	level_info
function	level_function
logic	level_logic
	level_all all *

Using a severity class token enables log messages at that severity only. For example, NS_LOG="*=warn" won't output messages with severity error. NS_LOG="*=level_debug" will output messages at severity levels debug and above.

Severity classes and levels can be combined with the `|' operator: NS_LOG="*=level_warn|logic" will output messages at severity levels error, warn and logic.

The NS_LOG severity level wildcard `*' and all are synonyms for level_all.

For log components merely mentioned in NS_LOG

$ NS_LOG="<log-component>:..."

the default severity is LOG_LEVEL_ALL.

Prefix Options

A number of prefixes can help identify where and when a message originated, and at what severity.

The available prefix options (as enum constants) are

Prefix Symbol	Meaning
LOG_PREFIX_FUNC	Prefix the name of the calling function.
LOG_PREFIX_TIME	Prefix the simulation time.
LOG_PREFIX_NODE	Prefix the node id.
LOG_PREFIX_LEVEL	Prefix the severity level.
LOG_PREFIX_ALL	Enable all prefixes.

The prefix options are described briefly below.

The options can be given in the NS_LOG environment variable by these tokens:

Token	Alternate
prefix_func	func
prefix_time	time
prefix_node	node
prefix_level	level
prefix_all	all *

For log components merely mentioned in NS_LOG

$ NS_LOG="<log-component>:..."

the default prefix options are LOG_PREFIX_ALL.

How to add logging to your code

Adding logging to your code is very simple:

1. Invoke the NS_LOG_COMPONENT_DEFINE (...); macro inside of namespace ns3.

   Create a unique string identifier (usually based on the name of the file and/or class defined within the file) and register it with a macro call such as follows:

```
namespace ns3 {


NS_LOG_COMPONENT_DEFINE ("Ipv4L3Protocol");

...
```

   This registers Ipv4L3Protocol as a log component.

   (The macro was carefully written to permit inclusion either within or outside of namespace ns3, and usage will vary across the codebase, but the original intent was to register this *outside* of namespace ns3 at file global scope.)

2. Add logging statements (macro calls) to your functions and function bodies.

In case you want to add logging statements to the methods of your template class (which are defined in an header file):

1. Invoke the NS_LOG_TEMPLATE_DECLARE; macro in the private section of your class declaration. For instance:

```
template <typename Item>

class Queue : public QueueBase

{

...

private:

  std::list<Ptr<Item> > m_packets;          //!< the items in the queue

  NS_LOG_TEMPLATE_DECLARE;                  //!< the log component

};
```

   This requires you to perform these steps for all the subclasses of your class.

2. Invoke the NS_LOG_TEMPLATE_DEFINE (...); macro in the constructor of your class by providing the name of a log component registered by calling the NS_LOG_COMPONENT_DEFINE (...); macro in some module. For instance:

```
template <typename Item>
```

```
Queue<Item>::Queue ()
  : NS_LOG_TEMPLATE_DEFINE ("Queue")
{
}
```

3. Add logging statements (macro calls) to the methods of your class.

In case you want to add logging statements to a static member template (which is defined in an header file):

1. Invoke the NS_LOG_STATIC_TEMPLATE_DEFINE (...); macro in your static method by providing the name of a log component registered by calling the NS_LOG_COMPONENT_DEFINE (...); macro in some module. For instance:

```
template <typename Item>
void
NetDeviceQueue::PacketEnqueued (Ptr<Queue<Item> > queue,
                                Ptr<NetDeviceQueueInterface> ndqi,
                                uint8_t txq, Ptr<const Item> item)
{

  NS_LOG_STATIC_TEMPLATE_DEFINE ("NetDeviceQueueInterface");
...
```

2. Add logging statements (macro calls) to your static method.

Logging Macros

The logging macros and associated severity levels are

Severity Class	Macro
LOG_NONE	(none needed)
LOG_ERROR	NS_LOG_ERROR (...);
LOG_WARN	NS_LOG_WARN (...);

LOG_DEBUG	NS_LOG_DEBUG (...);
LOG_INFO	NS_LOG_INFO (...);
LOG_FUNCTION	NS_LOG_FUNCTION (...);
LOG_LOGIC	NS_LOG_LOGIC (...);

The macros function as output streamers, so anything you can send to std::cout, joined by << operators, is allowed:

```
void MyClass::Check (int value, char * item)
{
  NS_LOG_FUNCTION (this << arg << item);
  if (arg > 10)
    {
      NS_LOG_ERROR ("encountered bad value " << value <<
                    " while checking " << name << "!");
    }
  ...
}
```

Note that NS_LOG_FUNCTION automatically inserts a `,' (comma-space) separator between each of its arguments. This simplifies logging of function arguments; just concatenate them with << as in the example above.