

# ns-3 with CMake

Gabriel Ferreira

University of Brasília

September 30, 2021

# My goals with CMake

## IDE support

1. Easier debugging and profiling
2. Auto-completion
3. Macro expansion (showing what the actual code looks like)
4. Visually show errors, warnings, unused includes, etc

# Unsupported/incomplete features

1. Bake integration
2. NSC examples
3. Planetlab examples
4. Python bindings

The code is there, but apiscan and generation fail for some modules

# What is missing?

1. Implement remaining features  
I can do that!
2. Stylistic and workflow related refactoring  
I need maintainers feedback
3. Update ns-3 official docs with instructions

# Current workflow

1. Create a new module
2. Configure CMake
3. Call the generated build system to build (e.g. ninja)
4. Call binaries directly or via the IDE
5. Debug via the IDE

## Current workflow

1. Create a new folder with a CMakeLists.txt file. Set library name, list of source files and add the macro at the end.



UnB

# Current workflow

## Configure CMake

1. This is required every time the CMake is changed
  - e.g. to include new source files or settings were changed
2. Previous options are preserved in the CMake cache
  - a.k.a. no need to pass all the options again when reconfiguring
3. Can be done either manually via terminal or GUI, or automatically via a config file (setting the flag values)
  - e.g. `cmake -DCMAKE_BUILD_TYPE=debug  
-DNS3_EXAMPLES=ON -DNS3_TESTS=ON -G Ninja ..`



# Current workflow

Call the generated build system to build

1. Call generated build system to build everything (or a specific target and its dependencies)  
e.g. `ninja (target)`

# Current workflow

Call binaries directly or via the IDE

1. Works nicely on platforms that support RPATH, which is enabled by default (e.g. Linux and MacOS)
2. Windows requires ns-3-dev/build in the PATH, which can be done automatically
3. Something like waf –run, –gdb, –valgrind, –command-template, –pyrun, –visualize would require a wrapper script

# Current workflow

## Debugging via the IDE: CLion

The screenshot shows the CLion IDE interface with the following details:

- Project View:** Shows a tree structure of files and folders. The current file is `main.cpp`, which contains C++ code for a network application.
- Code Editor:** Displays the source code for `main.cpp`. A blue arrow points to line 79, indicating a breakpoint has been set there.
- Breakpoints:** A list of active breakpoints is shown on the left side of the editor.
- Variables View:** Located at the bottom left, it shows the memory state of variables. A specific variable, `m_ip`, is expanded to show its internal structure.
- Memory View:** A detailed view of the memory location `0x0000000000000000`, showing pointers to nodes and their properties.
- Toolbars and Menus:** Standard CLion toolbars and menus are visible at the top and bottom of the interface.



# Current workflow

## Debugging via the IDE: Code::Blocks

The screenshot shows the Code::Blocks IDE interface during a debugging session. The main window displays the source code of a C++ file, specifically `ns3/internetHelper.cpp`. The code is annotated with several breakpoints, indicated by red dots. The current line of execution is highlighted at line 67, which contains the instruction `UdpEchoClientHelper::echoClient(interfaces.GetAddress(1), 9);`.

The left sidebar shows the project structure under "Project" and the current file being edited. The bottom-left corner displays the "Breakpoints" window, listing the locations of the set breakpoints.

The bottom right corner shows the "Registers" window, which lists various CPU registers along with their addresses and values. The "Registers" window is currently active, showing the contents of the CPU registers.

The status bar at the bottom provides system information, including the operating system (Windows 7 Pro), processor (Intel(R) Core(TM) i7-2600K CPU @ 3.40GHz), memory usage (Mem 54 / 4.1 GB 15%), and the current file path (ns3/internetHelper.cpp).



# Current workflow

## Debugging via the IDE: XCode

The screenshot shows the XCode IDE interface during a debugging session. The top menu bar indicates "Running first > My Mac". The left sidebar displays system monitoring for CPU, Memory, Energy Impact, Disk, and Network. A list of threads is shown, with Thread 1 (Queue: com.apple.xbsd) being the current target, marked with a blue dot.

The main editor area shows a portion of the source code for `point-to-point-helper.cc`. The code is annotated with several `NS_ASSERT` statements and includes calls to `MakeBoundCallback` and `DefaultDropSinkWithContext`.

A vertical green bar highlights the line `b->addDeviceAddress(dev8);`. A tooltip above this line reads "`b->addDeviceAddress(dev8);` Thread 1: step over".

The bottom status bar shows the current stack frame: `0x3: this = ns3::PointToPointHelper > 0x7feff6f50`. To the right of the status bar, there is a memory dump window titled "111db" showing memory contents at address `0x3`.

The right side of the interface contains the "Identity and Type" panel, which shows the file is a C++ source file relative to the project. It also lists "On Demand Resource Tags" and "Target Membership" (including ALL\_BUILD and ZERO\_CHECK).



UnB

# Where is the code and instructions?

The instructions are in the MR 460:

[https://gitlab.com/nsnam/ns-3-dev/-/merge\\_requests/460](https://gitlab.com/nsnam/ns-3-dev/-/merge_requests/460)

The sources are in the branch for the MR 460:

<https://gitlab.com/Gabrielcarvfer/ns-3-dev/-/tree/buildsystem-cmake>

# Questions? Concerns? Suggestions?

You can find me on Zulip or via the email  
[gabrielcarvfer@gmail.com](mailto:gabrielcarvfer@gmail.com)

Thank you for your time