How to use gdb

running gdb directly

gdb can be invoked on a program executable in the normal way (*gdb <program-name>*), and to do this with ns-3, it is easiest to use the Waf shell (to set the library paths correctly). We demonstrate with an example.

```
./waf shell
 cd build/examples/tutorial
 gdb ns3-dev-third-debug
 Reading symbols from ns3-dev-third-debug...done.
 (gdb)
```

At this point, one can set breakpoints in the ns-3 libraries, or run the program. We can run it with the *--help* argument to print out the program usage:

```
(gdb) r --help
```

Or we can deliberately cause a program error by running with an illegal argument

```
(gdb) r --nWifi=0
 Program received signal SIGSEGV, Segmentation fault.
 0x0000000000413ac9 in ns3::PeekPointer<ns3::Node> (p=...) at ./ns3/ptr.h:282
 282       return p.m_ptr;
 (gdb)
```

To run a test-suite through gdb, use the *test-runner* program, such as:

```
./waf shell
 cd build/utils/
 gdb ns3-dev-test-runner-debug
 (gdb) r --suite=csma-system
 (gdb) quit
```

running gdb with Waf

Waf supports a *--command-template* argument that allows users to launch gdb and run the programs from there.

```
./waf --command-template="gdb %s" --run <program-name>
```

An example using the *third* tutorial example, running it first to ask that the help message be printed out, and then running it with an illegal argument:

```
./waf --command-template="gdb %s" --run third
 (gdb) r --help
 (gdb) run --nWifi=0
 ...
 Program received signal SIGSEGV, Segmentation fault.
 0x0000000000413ac9 in ns3::PeekPointer<ns3::Node> (p=...) at ./ns3/ptr.h:282
 282       return p.m_ptr;
 (gdb)
```

To debug an individual test suite, use the *test-runner* program:

```
./waf --command-template="gdb %s" --run test-runner
 (gdb) run --suite=csma-system
```

running gdb with Waf, redirecting all output to a file

Sometimes a program will generate a lot of output to stdout and stderr, which may get in the way of interactively using the gdb prompt. It is possible to redirect program output to a file, while interacting with the gdb prompt. Here is a trivial example.

```
./waf --command-template="gdb %s" --run third
 (gdb) run --help > output.txt 2>&1
```

An example of how to run an individual test suite this way, launching gdb from the command line:

```
./waf --run test-runner --command-template="gdb -ex 'run --suite=csma-system >
csma-output.txt 2>&1' --args %s"
```

saving breakpoints across sessions

As of gdb 7.2, breakpoints can be saved to a file and reloaded for a separate session.

For example:

```
(gdb) info break
 Num     Type           Disp Enb Address    What
 1       breakpoint     keep y   <PENDING>  random-variable-stream.cc:175
 2       breakpoint     keep y   <PENDING>  random-variable-stream.cc:186
```

to save them to a file called 'breaks':

```
(gdb) save breakpoints breaks
```

Later, one can retrieve them as follows.

```
(gdb) source breaks
```

In ns-3 with shared libraries, one may get this error:

```
(gdb) source breaks
 No source file named random-variable-stream.cc.
 Make breakpoint pending on future shared library load? (y or [n]) [answered N; input not from
terminal]
 No source file named random-variable-stream.cc.
 Make breakpoint pending on future shared library load? (y or [n]) [answered N; input not from
terminal]
```

---

to work around this, use the command 'set breakpoints pending on' before sourcing the file:

```
(gdb) set breakpoint pending on
 (gdb) source breaks
 No source file named random-variable-stream.cc.
 Breakpoint 1 (random-variable-stream.cc:175) pending.
 No source file named random-variable-stream.cc.
 Breakpoint 2 (random-variable-stream.cc:186) pending.
 (gdb) info break
 Num     Type           Disp Enb Address    What
```

```
1       breakpoint     keep y   <PENDING>  random-variable-stream.cc:175
2       breakpoint     keep y   <PENDING>  random-variable-stream.cc:186
```