ns-3 Direct Code Execution (DCE) Manual

Release 1.1

Direct Code Execution project

August 30, 2013

CONTENTS

This is the manual of Direct Code Execution (DCE).

- Doxygen: Documentation of the public APIs of the DCE
- Manual *(this document)* for the latest release and development tree

This document is written in reStructuredText for Sphinx and is maintained in the `doc/` directory of ns-3-dce's source code.

CHAPTER

ONE

INTRODUCTION

1.1 Overview

Direct Code Execution (DCE) is a module for ns-3 that provides facilities to execute, within *ns-3*, existing implementations of userspace and kernelspace network protocols or applications without source code changes. For example instead of using the pseudo application provided by *ns-3* **V4PingHelper** you can use the true **ping**.

1.2 Manual Structure

This document consists of the following parts:

0. *Quick Start Guide*: The document describes a quick instruction of DCE.
1. *User's Guide*: The document is for people who will use DCE to experiment.

1.3 DCE Outlook

- To run an application using DCE, it is **not** necessary to **change** its sources. However you will need to **recompile** them.
- The simulation is executed wholly within a **single process** which greatly facilitates the **debugging**.
- DCE is very **memory-efficient**, thanks to the way it loads the executables similarly to shared libraries.

1.4 Supported Features

- Simulation with POSIX socket application (no manual modifications)
- C/C++ applications
- Simulation with Linux kernel implemented network protocol
- IPv4/IPv6
- TCP/UDP/DCCP
- running with POSIX socket applications and ns-3 socket applications
- configuration via sysctl-like interface
- multiple nodes debugging with single gdb interface

- memory analysis by single valgrind execution with multiple nodes
- Variance of network stacks
- ns-3 native stack (IPv4/IPv6, partially)
- network simulation cradle network stack (IPv4 TCP only)
- Linux network stack (IPv4/IPv6/others)
- per-node configuration/stdin input
- per-node syslog/stdout/stderr files output

1.5 Tested Applications

- CCNx
- Quagga
- iperf
- ping/ping6
- ip (iproute2 package)
- Mobilt IPv6 daemon (umip)
- Linux kernel (from 2.6.36 to 3.7 versions)
- http server (thttpd)
- torrent (libtorrent from rasterbar + opentracker)

1.6 Tested Environment

Currently, DCE only supports Linux-based operating system. DCE has been tested on the following distributions:

- Ubuntu 10.04 32bit/64bit
- Ubuntu 10.10 32bit/64bit
- Ubuntu 11.04 32bit/64bit
- Ubuntu 11.10 32bit/64bit
- Ubuntu 12.04 32bit/64bit
- Ubuntu 12.10 64bit
- Fedora 14 32bit/64bit
- Fedora 15 32bit/64bit
- Fedora 16 32bit/64bit
- Fedora 18 32bit

but you can try on the others (e.g., CentOS, RHEL). If you got run on another distribution, please let us know.

CHAPTER

TWO

QUICK START GUIDE

2.1 Introduction

The DCE ns-3 module provides facilities to execute within ns-3 existing implementations of userspace and kernelspace network protocols.

As of today, the Quagga routing protocol implementation, the CCNx CCN implementation, and recent versions of the Linux kernel network stack are known to run within DCE, hence allowing network protocol experimenters and researchers to use the unmodified implementation of their protocols for real-world deployments and simulations.

2.2 Build DCE

DCE offers two major modes of operation:

1. The basic mode, where DCE use the *ns-3* TCP stacks,
2. The advanced mode, where DCE uses a Linux network stack instead.

2.2.1 Building DCE basic mode

First you need to download Bake using Mercurial and set some variables:

```
hg clone http://code.nsnam.org/bake bake
export BAKE_HOME=`pwd`/bake
export PATH=$PATH:$BAKE_HOME
export PYTHONPATH=$PYTHONPATH:$BAKE_HOME
```

then you must to create a directory for DCE and install it using bake:

```
mkdir dce
cd dce
bake.py configure -e dce-ns3-|version|
bake.py download
bake.py build
```

note that dce-ns3-1.1 is the DCE version 1.1 module. If you would like to use the development version of DCE module, you can specify **dce-ns3-dev** as a module name for bake.

the output should look likes this:

```
Installing selected module and dependencies.
Please, be patient, this may take a while!
>> Downloading ccnx
>> Download ccnx - OK
>> Downloading iperf
>> Download iperf - OK
>> Downloading ns-3-dev-dce
>> Download ns-3-dev-dce - OK
>> Downloading dce-ns3
>> Download dce-ns3 - OK
>> Building ccnx
>> Built ccnx - OK
>> Building iperf
>> Built iperf - OK
>> Building ns-3-dev-dce
>> Built ns-3-dev-dce - OK
>> Building dce-ns3
>> Built dce-ns3 - OK
```

2.2.2 Building DCE advanced mode (with Linux kernel)

If you would like to try Linux network stack instead of *ns-3* network stack, you can try the advanced mode. The difference to build the advanced mode is the different module name *dce-linux* instead of *dce-ns3* (basic mode).

```
mkdir dce
cd dce
bake.py configure -e dce-linux-|version|
bake.py download
bake.py build
```

note that dce-linux-1.1 is the DCE version 1.1 module. If you would like to use the development version of DCE module, you can specify **dce-linux-dev** as a module name for bake.

2.3 Examples

If you got succeed to build DCE, you can try an example script which is already included in DCE package.

2.3.1 Example: Simple UDP socket application

This example execute the binaries named udp-client and udp-server under *ns-3* using DCE. These 2 binaries are written using POSIX socket API in order to send and receive UDP packets.

If you would like to see what is going on this script, please refer to the *user's guide*.

```
$ cd source/ns-3-dce
$ ./waf --run dce-udp-simple
$ ls
  elf-cache  files-0 exitprocs
$ ls -lR files-0
  files-0:
  total 4
  drwxr-x--- 3 furbani planete 4096 Sep  2 17:02 var

  files-0/var:
```

```
total 4
drwxr-x--- 4 furbani planete 4096 Sep  2 17:02 log

files-0/var/log:
total 8
drwxr-x--- 2 furbani planete 4096 Sep  2 17:02 53512
drwxr-x--- 2 furbani planete 4096 Sep  2 17:02 53513

files-0/var/log/53512:
total 12
-rw------- 1 furbani planete  12 Sep  2 17:02 cmdline
-rw------- 1 furbani planete 185 Sep  2 17:02 status
-rw------- 1 furbani planete   0 Sep  2 17:02 stderr
-rw------- 1 furbani planete  21 Sep  2 17:02 stdout

files-0/var/log/53513:
total 12
-rw------- 1 furbani planete  22 Sep  2 17:02 cmdline
-rw------- 1 furbani planete 185 Sep  2 17:02 status
-rw------- 1 furbani planete   0 Sep  2 17:02 stderr
-rw------- 1 furbani planete  22 Sep  2 17:02 stdout
```

This simulation produces two directories, the content of elf-cache is not important now for us, but files-0 is. files-0 contains first node's file system, it also contains the output files of the dce applications launched on this node. In the /var/log directory there are some directories named with the virtual pid of corresponding DCE applications. Under these directories there is always 4 files:

1. cmdline: which contains the command line of the corresponding DCE application, in order to help you to retrieve what is it,
2. stdout: contains the stdout produced by the execution of the corresponding application,
3. stderr: contains the stderr produced by the execution of the corresponding application.
4. status: contains a status of the corresponding process with its start time. This file also contains the end time and exit code if applicable.

Before launching a simulation, you may also create files-xx directories and provide files required by the applications to be executed correctly.

2.3.2 Example: iperf

This example shows the usage of iperf with DCE. You are able to generate traffic by well-know traffic generator *iperf* in your simulation. For more detail of the scenario description, please refer to the *user's guide*.

Once you successfully installed DCE with bake, you can execute the example using iperf.

```
cd source/ns-3-dce
./waf --run dce-iperf
```

As we saw in the previous example the experience creates directories containing the outputs of different executables, take a look at the server (node 1) output:

```
$ cat files-1/var/log/*/stdout
------------------------------------------------------------
Server listening on TCP port 5001
TCP window size:  124 KByte (default)
------------------------------------------------------------
[  4] local 10.1.1.2 port 5001 connected with 10.1.1.1 port 49153
[ ID] Interval       Transfer     Bandwidth
[  4]  0.0-11.2 sec  5.75 MBytes  4.30 Mbits/sec
```

the client (node-0) output bellow:

```
$ cat files-0/var/log/*/stdout
------------------------------------------------------------
Client connecting to 10.1.1.2, TCP port 5001
TCP window size:  124 KByte (default)
------------------------------------------------------------
[  3] local 10.1.1.1 port 49153 connected with 10.1.1.2 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0- 1.0 sec   640 KBytes  5.24 Mbits/sec
[  3]  1.0- 2.0 sec   512 KBytes  4.19 Mbits/sec
[  3]  2.0- 3.0 sec   640 KBytes  5.24 Mbits/sec
[  3]  3.0- 4.0 sec   512 KBytes  4.19 Mbits/sec
[  3]  4.0- 5.0 sec   512 KBytes  4.19 Mbits/sec
[  3]  5.0- 6.0 sec   640 KBytes  5.24 Mbits/sec
[  3]  6.0- 7.0 sec   512 KBytes  4.19 Mbits/sec
[  3]  7.0- 8.0 sec   640 KBytes  5.24 Mbits/sec
[  3]  8.0- 9.0 sec   512 KBytes  4.19 Mbits/sec
[  3]  9.0-10.0 sec   640 KBytes  5.24 Mbits/sec
[  3]  0.0-10.2 sec  5.75 MBytes  4.72 Mbits/sec
```

if you have already built the advanced mode, you can use Linux network stack over iperf.

```
cd source/ns-3-dce
./waf --run "dce-iperf --kernel=1"
```

the command line option **–kernel=1** makes the simulation use the Linux kernel stack instead of *ns-3* network stack.

```
$ cat files-1/var/log/*/stdout
------------------------------------------------------------
Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[  4] local 10.1.1.2 port 5001 connected with 10.1.1.1 port 60120
[ ID] Interval       Transfer     Bandwidth
[  4]  0.0-11.2 sec  5.88 MBytes  4.41 Mbits/sec

$ cat files-0/var/log/*/stdout
------------------------------------------------------------
Client connecting to 10.1.1.2, TCP port 5001
TCP window size: 16.0 KByte (default)
------------------------------------------------------------
[  3] local 10.1.1.1 port 60120 connected with 10.1.1.2 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0- 1.0 sec   512 KBytes  4.19 Mbits/sec
[  3]  1.0- 2.0 sec   640 KBytes  5.24 Mbits/sec
[  3]  2.0- 3.0 sec   640 KBytes  5.24 Mbits/sec
[  3]  3.0- 4.0 sec   512 KBytes  4.19 Mbits/sec
[  3]  4.0- 5.0 sec   640 KBytes  5.24 Mbits/sec
[  3]  5.0- 6.0 sec   512 KBytes  4.19 Mbits/sec
[  3]  6.0- 7.0 sec   640 KBytes  5.24 Mbits/sec
[  3]  7.0- 8.0 sec   640 KBytes  5.24 Mbits/sec
[  3]  8.0- 9.0 sec   512 KBytes  4.19 Mbits/sec
[  3]  9.0-10.0 sec   640 KBytes  5.24 Mbits/sec
[  3]  0.0-10.2 sec  5.88 MBytes  4.84 Mbits/sec
```

Interestingly, the two results between two network stacks are slightly different, though the difference is out of scope of this document.

CHAPTER

THREE

USER'S GUIDE

This document is for the people who want to use your application in *ns-3* using DCE.

Direct Code Execution (DCE) allows us to use POSIX socket-based applications as well as Linux kernel network stack.

3.1 Setup Guide

In order to install DCE you must follow the tutorial *Build DCE*.

Installation result

The result of the installation process is the creation of libraries from source of DCE and that of *ns-3* and also some tools and sources of an optional Linux kernel if you have also chosen to use the stack of a Linux kernel. Below you will find the main directories:

```
|-- bakefile.xml          Bake internal configuration file (generated by bake.py configure command).
|-- bakeSetEnv.sh         Bake generated file used to configure environmental variable.
|-- build                 Target directory of |ns3| Core and DCE compilation.
|   |-- bin
|   |-- bin_dce
|   |-- etc
|   |-- include
|   |-- lib
|   |-- sbin
|   |-- share
|   |-- usr
|   +-- var
+-- source                 Source directory during 'bake.py download'. Listed files below depend on t
    |-- ccnx
    |-- ccnx-0.6.2.tar.gz
    |-- ns-3-dce
    |   |-- build
    |   |-- doc            Documentation source
    |   |-- elf-cache
    |   |-- example        Example scenarios using DCE
    |   |-- files-0
    |   |-- files-1
    |   |-- files-2
    |   |-- files-3
    |   |-- files-5
    |   |-- helper         The source code directory for helper library
    |   |-- model          The source code directory for DCE core
    |   |-- myscripts      Sub-module and ad-hoc script directory
    |   |-- netlink        Netlink module
    |   |-- ns3waf         waf extension used by DCE
    |   |-- test           Test script directory
    |   |-- testpy-output  Directory used for test results
    |   +-- utils          Utilities used by DCE
    |-- elf-loader
    |-- iperf
    |-- iperf-2.0.5-source.tar.gz
    |-- iproute
    |-- iproute2-2.6.38.tar.bz2
    |-- iputils
    |-- iputils-s20101006.tar.bz2
    |-- mptcp
    |-- net-next-sim-2.6.36
    |-- ns-3-dev-dce
    |-- pybindgen-user
    |-- quagga
    |-- quagga-0.99.20.tar.gz
    |-- thttpd
    |-- thttpd-2.25b.tar.gz
    |-- umip
    |-- wget
    +-- wget-1.14.tar.gz
```

3.2 Basic Use Cases

3.2.1 Using your userspace protocol implementation

As explained in *How It Works (To Be Available Soon)*, DCE needs to relocate the executable binary in memory, and these binary files need to be built with specific compile/link options.

In order to this you should follow the two following rules:

1. Compile your objects using this gcc flag: **-fPIC** for exemple : `gcc -fPIC -c foo.c`
1. (option) Some application needs to be compile with **-U_FORTIFY_SOURCE** so that the application doesn't use alternative symbols including **__chk** (like memcpy_chk).
2. Link your executable using this gcc flag: **-pie** for exemple : `gcc -o foo -pie foo.o`
3. Verify the produced executable using readelf utility in order to display the ELF file header and to verify that your exe is of type **DYN** indicating that DCE should be able to relocate and virtualize it under *ns-3* virtual world and network. For exemple : `readelf -h foo|grep Type:` **==>** `Type:  DYN (Shared object file)`
4. Check also that your executable runs as expected outside of *ns-3* and DCE.

Install the target executable

Copy the executable file produced in a specified directory in the variable environment DCE_PATH so that DCE can find it. (FIXME: to be updated)

Write a *ns-3* script

Now that you have compiled your executable you can use it within *ns-3* script with the help of a set of DCE Helper Class:

HELPER CLASS NAME	INCLUDE NAME	DESCRIPTION
DceManagerHelper	ns3/dce-manager-helper.h	A DceManager is a DCE internal class which manage the execution of the executable you will declare to run within *ns-3*; The DceManagerHelper is the tool you will use within your script to parameter and install DceManager on the *ns-3* nodes where you plan to run binaries.
DceApplicationHelper	ns3/dce-application-helper.h	You will use this helper in order to define which application you want to run within *ns-3* by setting the name of the binary its optionals arguments , its environment variables, and also optionaly if it take its input from a file instead of stdin. This class can be derived if you need to do more preparation before running your application. Often applications need configuration file to work properly, for example if you look at the contents of the helper named CcnClientHelper you will see that his job is to create the key files needed for the operation of CCNx's applications.
LinuxStackHelper	ns3/linux-stack-helper.h	This helper is used to configure parameters of Linux kernel when we are using the advanced mode.
CcnClientHelper	ns3/ccn-client-helper.h	This helper is a subclass of DceApplicationHelper, its jobs is to create keys files used by ccnx executables in order to run them correctly within NS3.
QuaggaHelper	ns3/quagga-helper.h	This helper is a subclass of DceApplicationHelper. It will help you to setup Quagga applications.

Note that the table above indicates the name of includes, so you can look at the comments in them, but in reality for DCE use you need to include only the file `ns3/dce-module.h`.

The directory named `myscripts` is a good place to place your scripts. To create a new script you should create a new directory under `myscripts`, and put your sources and a configuration file for waf build system, this file should be named `wscript`. For starters, you may refer to the contents of the directory `myscripts/ping`.

For more detail, please refer DCE API (doxygen) document.

Compile the script

To compile simply execute the command waf. The result must be under the directory named `build/bin/myscripts/foo/bar` where **foo** is your directory and **bar** your executable according to the content of your `wscript` file.

Launch the script

Simply launch your script like any other program.

```
$ ./waf --run bar
```

Results

The execution of the apps using DCE generates special files which reflect the execution thereof. On each node DCE creates a directory `/var/log`, this directory will contain subdirectory whose name is a number. This number is the pid of a process. Each of these directories contains the following files `cmdline`, `status`, `stdout`, `stderr`. The file `cmdline` recalls the name of the executable run followed arguments. The file `status` contains an account of the execution and dating of the start; optionally if the execution is completed there is the date of the stop and the return code. The files `stdout` and `stderr` correspond to the standard output of the process in question.

Example: DCE Simple UDP (dce-udp-simple)

The example uses two POSIX socket-based application in a simulation. Please take time to look at the source dce-udp-simple.cc:

```
int main (int argc, char *argv[])
{
  CommandLine cmd;
  cmd.Parse (argc, argv);

  NodeContainer nodes;
  nodes.Create (1);

  InternetStackHelper stack;
  stack.Install (nodes);

  DceManagerHelper dceManager;
  dceManager.Install (nodes);

  DceApplicationHelper dce;
  ApplicationContainer apps;

  dce.SetStackSize (1<<20);

  dce.SetBinary ("udp-server");
  dce.ResetArguments();
  apps = dce.Install (nodes.Get (0));
  apps.Start (Seconds (4.0));

  dce.SetBinary ("udp-client");
  dce.ResetArguments();
  dce.AddArgument ("127.0.0.1");
  apps = dce.Install (nodes.Get (0));
  apps.Start (Seconds (4.5));

  Simulator::Stop (Seconds(1000100.0));
  Simulator::Run ();
  Simulator::Destroy ();

  return 0;
}
```

You can notice that we create a *ns-3* Node with an Internet Stack (please refer to *ns-3* doc. for more info), and we can also see 2 new Helpers:

1. DceManagerHelper which is used to Manage DCE loading system in each node where DCE will be used.
2. DceApplicationHelper which is used to describe real application to be lauched by DCE within *ns-3* simulation environnement.

Example: DCE with iperf(dce-iperf)

The example uses iperf traffic generator in a simulation. The scenario is here:

```
$ cat source/dce/example/dce-iperf.cc
#include "ns3/network-module.h"
#include "ns3/core-module.h"
#include "ns3/internet-module.h"
#include "ns3/dce-module.h"
#include "ns3/point-to-point-module.h"
#include "ns3/applications-module.h"
#include "ns3/netanim-module.h"
#include "ns3/constant-position-mobility-model.h"
#include "ccnx/misc-tools.h"

using namespace ns3;
// ===========================================================================
//
//         node 0                 node 1
//   +----------------+    +----------------+
//   |                |    |                |
//   +----------------+    +----------------+
//   |    10.1.1.1    |    |    10.1.1.2    |
//   +----------------+    +----------------+
//   | point-to-point |    | point-to-point |
//   +----------------+    +----------------+
//           |                     |
//           +---------------------+
//                5 Mbps, 2 ms
//
// 2 nodes : iperf client en iperf server ....
//
// Note : Tested with iperf 2.0.5, you need to modify iperf source in order to
//        allow DCE to have a chance to end an endless loop in iperf as follow:
//        in source named Thread.c at line 412 in method named thread_rest
//        you must add a sleep (1); to break the infinite loop....
// ===========================================================================
int main (int argc, char *argv[])
{
  bool useKernel = 0;
  bool useUdp = 0;
  std::string bandWidth = "1m";
  CommandLine cmd;
  cmd.AddValue ("kernel", "Use kernel linux IP stack.", useKernel);
  cmd.AddValue ("udp", "Use UDP. Default false (0)", useUdp);
  cmd.AddValue ("bw", "BandWidth. Default 1m.", bandWidth);
  cmd.Parse (argc, argv);

  NodeContainer nodes;
  nodes.Create (2);

  PointToPointHelper pointToPoint;
  pointToPoint.SetDeviceAttribute ("DataRate", StringValue ("5Mbps"));
  pointToPoint.SetChannelAttribute ("Delay", StringValue ("1ms"));

  NetDeviceContainer devices;
  devices = pointToPoint.Install (nodes);

  DceManagerHelper dceManager;
```

```
dceManager.SetTaskManagerAttribute( "FiberManagerType", StringValue ( "UcontextFiberManager" ) );

if (!useKernel)
  {
    InternetStackHelper stack;
    stack.Install (nodes);
  }
else
  {
    dceManager.SetNetworkStack ("ns3::LinuxSocketFdFactory", "Library", StringValue ("liblinux.so")
    LinuxStackHelper stack;
    stack.Install (nodes);
  }

Ipv4AddressHelper address;
address.SetBase ("10.1.1.0", "255.255.255.252");
Ipv4InterfaceContainer interfaces = address.Assign (devices);

// setup ip routes
Ipv4GlobalRoutingHelper::PopulateRoutingTables ();

dceManager.Install (nodes);

DceApplicationHelper dce;
ApplicationContainer apps;

dce.SetStackSize (1<<20);

// Launch iperf client on node 0
dce.SetBinary ("iperf");
dce.ResetArguments();
dce.ResetEnvironment();
dce.AddArgument ("-c");
dce.AddArgument ("10.1.1.2");
dce.AddArgument ("-i");
dce.AddArgument ("1");
dce.AddArgument ("--time");
dce.AddArgument ("10");
if (useUdp)
  {
    dce.AddArgument ("-u");
    dce.AddArgument ("-b");
    dce.AddArgument (bandWidth);
  }

apps = dce.Install (nodes.Get (0));
apps.Start (Seconds (0.7));
apps.Stop (Seconds (20));

// Launch iperf server on node 1
dce.SetBinary ("iperf");
dce.ResetArguments();
dce.ResetEnvironment();
dce.AddArgument ("-s");
dce.AddArgument ("-P");
dce.AddArgument ("1");
if (useUdp)
  {
      dce.AddArgument ("-u");
    }

  apps = dce.Install (nodes.Get (1));

  pointToPoint.EnablePcapAll (useKernel?"iperf-kernel":"iperf-ns3", false);

  apps.Start (Seconds (0.6));

  setPos (nodes.Get (0), 1, 10, 0);
  setPos (nodes.Get (1), 50,10, 0);

  Simulator::Stop (Seconds(40.0));
  Simulator::Run ();
  Simulator::Destroy ();

  return 0;
}
```

This scenario is simple there is 2 nodes linked by a point 2 point link, the node 0 launch iperf as a client via the command **iperf -c 10.1.1.2 -i 1 –time 10** and the node 1 launch iperf as a server via the command **iperf -s -P 1**. You can follow this to launch the experiement:

3.2.2 Using your in-kernel protocol implementation

(TBA)

3.2.3 Creating your protocol implementation as a DCE sub-module

If your application has a configuration file to modify the behavior of applications, introducing a particular Helper class will be helpful to handle your application. In this section, we will give you an advanced way of using your application with DCE.

Some of existing submodule are following this way. You can find ns-3-dce-quagga and ns-3-dce-umip as examples to add sub-module.

Obtaining DCE sub-module template

First of all, you could start with referring sub module template available as follows.

```
hg clone http://code.nsnam.org/thehajime/ns-3-dce-submodule (your module name)
```

The template consists of, wscript, helper, test and documentation. You could rename all/some of them for your module. Then, put `ns-3-dce-submodule` directory under ns-3-dce/myscripts/. This will be required to build under ns-3-dce module as an extension (sub-module) of dce.

Writing wscript

Writing bakeconf.xml (optional)

Implementing helper class (optional)

Writing examples (optional)

3.2.4 Global DCE Configurations

Parameters

The DCE specifics variables are essentially two PATH like variables: so within them you may put paths separated by ‘:’ character.

DCE_PATH is used by DCE to find the executable you want to launch within *ns-3* simulated network. This variable is used when you reference the executable using a relative form like ‘ping’.

DCE_ROOT is similar to DCE_PATH but it is used when you use an absolute form for exemple ‘/bin/bash’.

Please pay attention that executables that you will place in the directories indicated in the previous variables should be recompiled accordingly to the rules defined in the next chapter.

(FIXME: to be updated)

Tweaking

DCE is configurable with NS3 Attributes. Refer to the following table:

ATTRIBUTE NAME	DESCRIPTION	VALUES	EXAMPLES

FiberManagerType	The TaskManager is used to switch the execution context between threads and processes.	**UcontextFiberManager** the more efficient.
		PthreadFiberManager helpful with **gdb** to see the threads. This is the de fault.
			`--ns3::TaskManager::FiberMan`
			`dceManager.SetTaskManagerAtt`
			`StringValue("UcontextFiberMa`
			`--ns3::TaskManager::FiberMan`

LoaderFactory	The LoaderFactory is used to load the hosted binaries.	**CoojaLoaderFactory** is the default and the only one that supports `fork`.
		DlmLoaderFactory is the more efficient. To use it you have two ways:
		• use `dce-runner`
		• link using `ldso` as default interpreter.
			`--ns3::DceManagerHelper::Loa`
			`$ dce-runner`
			`my-dce-ns3-script`
			OR
			`gcc -o`
			`my-dce-ns3-script`
			`Wl,--dynamic-linker=PATH2LDS`
			`...`
			`$`
			`my-dce-ns3-script`
			`--ns3::DceManagerHelper::Loa`
			`mLoaderFactory[]`
			`dceManager.SetLoader("ns3::D`

3.2.5 DCE Cradle

This document describes what DCE Cradle is, how we can use it, how we extend it.

Tutorials and how to reproduce the experiment of WNS3 2013 paper is available `dce-cradle-usecase`.

What is DCE Cradle?

DCE Cradle enables us to use Linux kernel via Direct Code Execution from the ns-3 native socket application. Applications can access it via ns-3 socket API. Currently (20th Nov. 2012) the following sockets are available:

- IPv4 UDP
- IPv4 TCP
- IPv4 RAW socket
- IPv4 DCCP

Installing DCE Cradle

DCE Cradle is already integrated in ns-3-dce module. You can just build and install DCE as instructed in the parent document.

How to use it

```
OnOffHelper onoff = OnOffHelper ("ns3::LinuxTcpSocketFactory",
                                 InetSocketAddress (interfaces.GetAddress (1), 9));
```

How to extend it

(To be added)

Article

The project originally started during GSoC project 2012

the paper (to be added if it will be published)

3.2.6 Aspect-based Tracing

Aspect-based tracing, provided by libaspect, allows us to use tracing facility with unmodified code.

One of contradictions when we use DCE is, tracing, how to put trace sources into unmodified code. While DCE gives an opportunity to use unmodified codes as simulation protocols, one might want to investigate which function is called or how many messages of a particular protocol are exchanged.

ns-3 originally has a nice feature of tracing with such a purpose, with on-demand trace connector to obtain additional information. Instead of inserting TraceSource into the original code, DCE gives dynamic trace points with this library, based on the idea of aspect-based tracing.

For more detail, see the Chapter 6.3.2 of the thesis.

Quick Start

To put trace sources without modifying the original code, aspcpp::HookManager gives trace hooks into arbitrary source codes and functions.

```
#include <hook-manager.h>

HookManager hooks;
hooks.AddHookBySourceAndFunction ("ip_input.c", "::ip_rcv", &IpRcv);
hooks.AddHookByFunction ("::process_backlog", &ProcBacklog);
hooks.AddHookByFunction ("::arp_xmit", &ArpXmit);
```

The above examples specifies file name and functions with callback functions in the simulation script.

Limitations

- July 10th, 2013: aspect-based tracing (libaspect) is in the alpha release state. It might be updated frequently.
- Callback function has no argument that it can investigate the contents of buffer that each function handles.

3.3 Advanced Use Cases

3.3.1 Using Alternative, Fast Loader

DCE optionally supports an alternative ELF loader/linker, so-called elf-loader, in order to replace system-provided linker/loader module. The intention of the loader is to support unlimited number of instances used by **dlmopen** call, which provides DCE to load a single ELF binary to multiple different memory space. dlmopen-based loader (ns3::DlmLoaderFactory) is much faster than another default one (ns3::CoojaLOaderFactory), but few issues are remain so, this is optional.

To Speedup Run-time

In order to use DlmLoaderFactory, you can add command-line argument of waf.

```
./waf --run dce-tcp-simple --dlm
```

if you are in the *./waf shell* mode, the following command should be used instead.

```
./build/bin/dce-runner ./build/bin/dce-tcp-simple
```

3.3.2 Debugging your protocols with DCE

Gdb

It is possible to use gdb to debug a script DCE/*ns-3*. As explained somewhere in the execution of a script is monoprocess, then you can put breakpoints in both sources of DCE and those of binaries hosted by DCE.

Install

Although it is not strictly necessary, it is recommended that you recompile a CVS Gdb for use with ns-3-dce. First, download::

```
cvs -d :pserver:anoncvs@sourceware.org:/cvs/src login
{enter "anoncvs" as the password}
cvs -d :pserver:anoncvs@sourceware.org:/cvs/src co gdb
```

Note that you might consider looking at http://sourceware.org/gdb/current/ to obtain more efficient (cpu/bandwidth-wise) download instructions.

Anyway, now, you can build::

```
cd gdb
./configure
make
```

And, then, invoke the version of gdb located in gdb/gdb instead of your system-installed gdb whenever you need to debug a DCE-based program.

Using

If you use gdb (a CVS or stable version), do not forget to execute the following command prior to running any DCE-based program::

```
(gdb) handle SIGUSR1 nostop
Signal        StopPrintPass to programDescription
SIGUSR1       NoYesYesUser defined signal 1
(gdb)
```

An alternate way to do this and avoid having to repeat this command ad-nauseam involves creating a .gdbinit file in your ns-3-dce directory and putting this inside::

```
handle SIGUSR1 nostop
```

Helpful debugging hints

There are a couple of functions which are useful to put breakpoints into:

- ns3::DceManager::StartProcessDebugHook

Put a breakpoint in a specif node in a simulation

If you got a trouble in your protocol during interactions between distributed nodes, you want to investigate a specific state of the protocol in a specific node. In a usual system, this is a typical case of using *distributed debugger* (e.g., ddt, or *mpirun xterm -e gdb –args xxx*), but it is annoying task in general due to the difficulty of controlling distributed nodes and processes.

DCE gives an easy interface to debug distributed applications/protocols by the single-process model of its architecture.

The following is an example of debugging Mobile IPv6 stack (of Linux) in a specific node (i.e., home agent). A special function *dce_debug_nodeid()* is useful if you put a break condition in a gdb session.

```
(gdb) b mip6_mh_filter if dce_debug_nodeid()==0
Breakpoint 1 at 0x7ffff287c569: file net/ipv6/mip6.c, line 88.
 <continue>
(gdb) bt 4
#0  mip6_mh_filter (sk=0x7ffff7f69e10, skb=0x7ffff7cde8b0)
    at net/ipv6/mip6.c:109
#1  0x00007ffff2831418 in ipv6_raw_deliver (skb=0x7ffff7cde8b0,
    nexthdr=135)
    at net/ipv6/raw.c:199
#2  0x00007ffff2831697 in raw6_local_deliver (skb=0x7ffff7cde8b0,
    nexthdr=135)
    at net/ipv6/raw.c:232
#3  0x00007ffff27e6068 in ip6_input_finish (skb=0x7ffff7cde8b0)
    at net/ipv6/ip6_input.c:197
(More stack frames follow...)
```

Valgrind

(FIXME: simple session using valgrind)

3.3.3 Testing your protocols with DCE

Since DCE allows protocol implementations to expose network conditions (packet losses, reordering, and errors) with the interactions among distributed nodes, which is not easily available by traditional user-mode virtualization tools, exercising your code is easily done with a single simulation script.

Coverage Test

Improving code coverage with writing test programs is one of headache; - writing test program is annoying, - preparing test network tends to be short-term, and - the result is not reproducible.

This text describes how to measure code coverage of protocol implementations with DCE.

1. build target implementations (applications, kernel stack) with profile option
2. run test program with DCE
3. parse the result of test coverage

Setup

First, you need to compile your application with additional flags. **-fprofile-arcs -ftest-coverage** is used for a compilation flag (CFLAGS/CXXFLAGS), and **-fprofile-arcs** is used for a linker flag (LDFLAGS).

```
gcc -fprofile-arcs -ftest-coverage -fPIC -c foo.c
gcc -fprofile-arcs -pie foo.o -o newapp
```

Write Test Program

Next, write a test program like *ns-3* simulation script for your application (i.e., newapp).

```
$ cat myscripts/dce-newapp.cc

int main (int argc, char *argv[])
{
  CommandLine cmd;
  cmd.Parse (argc, argv);

  NodeContainer nodes;
  nodes.Create (2);

  InternetStackHelper stack;
  stack.Install (nodes);
  DceManagerHelper dceManager;
  dceManager.Install (nodes);

  DceApplicationHelper dce;
  ApplicationContainer apps;

  // application on node 0
  dce.SetBinary ("newapp");
  dce.ResetArguments();
  apps = dce.Install (nodes.Get (0));
  apps.Start (Seconds (4.0));

  // application on node 1
  dce.SetBinary ("newapp");
  dce.ResetArguments();
  apps = dce.Install (nodes.Get (1));
  apps.Start (Seconds (4.5));

  Simulator::Stop (Seconds(100.0));
  Simulator::Run ();
  Simulator::Destroy ();

  return 0;
}
```

Run Test

Then, test your application as normal *ns-3* (and DCE) simulation execution.

```
./waf --run dce-newapp
```

If you successfully finish your test, you will see the coverage data files (i.e., gcov data files) with a file extension *.gcda*.

```
$ find ./ -name "*.gcda"

./files-0/home/you/are/here/ns-3-dce/newapp.gcda
./files-1/home/you/are/here/ns-3-dce/newapp.gcda
```

Parse Test Result

We use lcov utilities as a parse of coverage test result.

Put the compiler (gcc) generated files (*.gcno) in the result directory,

```
cp *.gcno ./files-0/home/you/are/here/ns-3-dce/
cp *.gcno ./files-1/home/you/are/here/ns-3-dce/
```

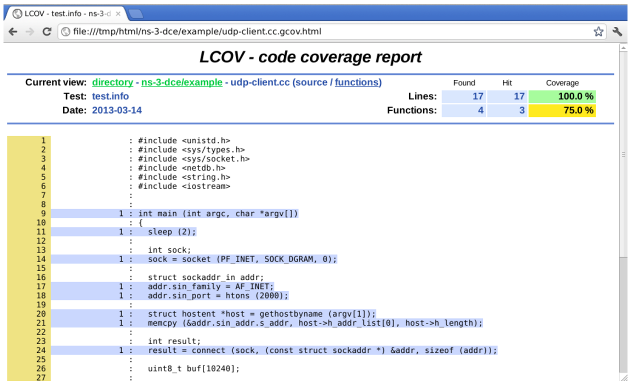
then run the *lcov* and *genhtml* command to generate coverage information of your test program.

```
lcov -c -d .-b . -o test.info
genhtml test.info -o html
```

You will see the following output and generated html pages.

```
Reading data file test.info
Found 8 entries.
Writing .css and .png files.
```

```
Generating output.
Processing file ns-3-dce/example/udp-server.cc
genhtml: Use of uninitialized value in subtraction (-) at /usr/bin/genhtml line 4313.
Processing file ns-3-dce/example/udp-client.cc
genhtml: Use of uninitialized value in subtraction (-) at /usr/bin/genhtml line 4313.
Processing file /usr/include/c++/4.4/iostream
Processing file /usr/include/c++/4.4/ostream
Processing file /usr/include/c++/4.4/bits/ios_base.h
Processing file /usr/include/c++/4.4/bits/locale_facets.h
Processing file /usr/include/c++/4.4/bits/char_traits.h
Processing file /usr/include/c++/4.4/bits/basic_ios.h
Writing directory view page.
Overall coverage rate:
  lines......: 49.3% (35 of 71 lines)
  functions..: 31.6% (6 of 19 functions)
```


LCOV - test.info - ns-3-d
file:///tmp/html/ns-3-dce/example/udp-client.cc.gcov.html

LCOV - code coverage report

Current view: directory - ns-3-dce/example - udp-client.cc (source / functions)
Test: test.info
Date: 2013-03-14

	Found	Hit	Coverage
Lines:	17	17	100.0 %
Functions:	4	3	75.0 %

```
      : #include <unistd.h>
      : #include <sys/types.h>
      : #include <sys/socket.h>
      : #include <netdb.h>
      : #include <string.h>
      : #include <iostream>
      :
      :
    1 : int main (int argc, char *argv[])
      : {
    1 :   sleep (2);
      :
      :   int sock;
    1 :   sock = socket (PF_INET, SOCK_DGRAM, 0);
      :
      :   struct sockaddr_in addr;
    1 :   addr.sin_family = AF_INET;
    1 :   addr.sin_port = htons (2000);
      :
    1 :   struct hostent *host = gethostbyname (argv[1]);
    1 :   memcpy (&addr.sin_addr.s_addr, host->h_addr_list[0], host->h_length);
      :
      :   int result;
    1 :   result = connect (sock, (const struct sockaddr *) &addr, sizeof (addr));
      :
      :   uint8_t buf[10240];
      :
```

Fuzz Test

(TBA, about integration of trinity)

Regression Test

(TBA)

3.4 Technical Information

3.4.1 DCE in a Nutshell

File System

To start a program in the world of *ns-3* you must indicate on which node it will be launched. Once launched this program will have access only to the file system corresponding to the node that corresponds to a directory on your machine called file-X where X is the decimal number of the corresponding node. The file-X directories are created by DCE, only when they do not already exist. Also **note** that the contents of this directory is not cleared when starting the script. So you can copy the files required for the operation of your executables in the tree nodes. If possible it is best that you create these files from the script itself in order to simplify maintenance. DCE provides some helpers for creating configuration files necessary to the execution of certain apps like CCNx and Quagga.

Network

Your program running in a *ns-3* node views the network defined by the script for this node.

Time

Time perceived by your executable is the simulated time of *ns-3*. Also **note** that DCE supports real time scheduler of *ns-3* with the same limitations.

3.4.2 Limitations

- Currently the POSIX API (libc) is not fully supported by DCE. However there are already about **400 methods** supported. As the goal of DCE is to allow to execute network applications, many methods related to the network are supported for example `socket`, `connect`, `bind`, `listen`, `read`, `write`, `poll`, `select`. The next chapter list the applications well tested using DCE.
- Some methods are not usable with all options of DCE. For more details refer to chapter **Coverage API** that lists all the supported methods.
- The scheduler is not as advanced as that of a kernel, for example if an infinite loop in a hosted application, DCE can not get out, but this should not happen in applications written correctly.

3.4.3 API Coverage

Below there is the list of the systems calls supported by DCE, the column named **Type** represents how the system call is implemented ie:

1. **DCE** the method is fully rewritten,
2. **NATIVE** the real corresponding system call is used.

Table 3.1: API Coverage

System Call Name	Domain	Inc
gettimeofday	Date & Time	sys/t
time	Date & Time	time
asctime, ctime, gmtime, localtime	Date & Time	time
asctime_r, ctime_r, gmtime_r, localtime_r, mktime, strftime	Date & Time	time
clock_gettime	Date & Time	time
read	IO	unis
write	IO	unis
writev	IO	sys/
clearerr	IO	stdi
setbuf, setbuffer, setlinebuf, setvbuf	IO	stdi
fseek, ftell, rewind, fgetpos, fsetpos	IO	stdi
printf	IO	stdi
fprintf	IO	stdi
sprintf, snprintf	IO	stdi
asprintf, vasprintf	IO	stdi
dprintf, vdprintf	IO	stdi
fgetc, fgetc_unlocked	IO	stdi
getc, getc_unlocked	IO	stdi

Table 3.1 – continued from previous page

System Call Name	Domain	Inc
getchar, getchar_unlocked	IO	stdi
_IO_getc	IO	stdi
fputc, fputc_unlocked	IO	stdi
putc, putc_unlocked	IO	stdi
putchar, putchar_unlocked	IO	stdi
_IO_putc	IO	stdi
fgets, fgets_unlocked	IO	stdi
fputs, fputs_unlocked	IO	stdi
puts	IO	stdi
ungetc	IO	stdi
fclose	IO	stdi
fcloseall	IO	stdi
fopen, fdopen64, fdopen, freopen	IO	stdi
fflush, fflush_unlocked	IO	stdi
fread, fread_unlocked	IO	stdi
fwrite, fwrite_unlocked	IO	stdi
ferror, ferror_unlocked	IO	stdi
feof, feof_unlocked	IO	stdi
fileno, fileno_unlocked	IO	stdi
perror	IO	stdli
vprintf	IO	stda
vfprintf, vsprintf, vsnprintf	IO	stda
fcntl	IO	fcnt
dup, dup2	IO	unis
open, open64	IO	fcnt
close	IO	unis
unlink	IO	unis
remove	IO	stdi
mkdir	IO	sys/
rmdir	IO	unis
select	IO	unis
isatty	IO	unis
ioctl	IO	sys/
poll	IO	poll.
getcwd, getwd, get_current_dir_name	IO	unis
chdir, fchdir	IO	unis
alphasort, alphasort64, versionsort	IO	dire
umask	IO	sys/
truncate, ftruncate	IO	unis
ttyname	IO	unis
lseek	IO	unis
euidaccess, eaccess	IO	unis
pathconf	IO	unis
getpwnam, getpwuid, endpwent	IO	pwd
opendir, fdopendir	IO	dire
readdir, readdir_r	IO	dire
closedir	IO	dire
dirfd	IO	dire
rewinddir	IO	dire

Table 3.1 – continued from previous page

System Call Name	Domain	Inc
scandir	IO	dire
unlinkat	IO	fcnt
pread, pwrite	IO	unis
uname	Kernel	sys/
sysconf	Kernel	unis
calloc, malloc, free, realloc	Memory Allocation	stdli
mmap, mmap64, munmap	Memory Mappings	sys/
htonl	Networking	arpa
htons	Networking	arpa
ntohl	Networking	arpa
htonl	Networking	arpa
ntohs	Networking	arpa
socket	Networking	sys/
getsockname	Networking	sys/
getpeername	Networking	sys/
bind	Networking	sys/
connect	Networking	sys/
inet_aton, inet_addr, inet_network, inet_ntoa, inet_makeaddr, inet_lnaof, inet_netof	Networking	arpa
inet_ntop	Networking	arpa
inet_pton	Networking	arpa
getsockopt, setsockopt	Networking	sys/
listen	Networking	sys/
accept	Networking	sys/
shutdown	Networking	sys/
send, sendto, sendmsg	Networking	sys/
recv, recvfrom, recvmsg	Networking	sys/
gethostbyname, gethostbyname2	Networking	netd
getaddrinfo, freeaddrinfo, gai_strerror	Networking	netd
gethostent, sethostent, endhostset, hstrerror	Networking	netd
herror	Networking	netd
getprotoent, getprotobyname, getprotonumber, setprotoent, endprotoent	Networking	netd
getservent, getservbyname, getservbyport, setservent, endservent	Networking	netd
if_nametoindex	Networking	net/i
getnameinfo	Networking	sys/
ether_aton_r	Networking	neti
atexit	Process	stdli
getpid, getppid	Process	unis
getuid, geteuid	Process	unis
setuid	Process	unis
setgid	Process	unis
seteuid, setegid	Process	unis
setreuid, setregid	Process	unis
setresuid, setresgid	Process	unis
sched_yield	Process	sche
exit	Process	unis
getenv	Process	stdli
putenv	Process	stdli
setenv, unsetenv	Process	stdli
clearenv	Process	stdli

Table 3.1 – continued from previous page

System Call Name	Domain	Inc
fork	Process	unis
abort	Process	stdli
execl, execlp, execle, execv, execvp, execve	Process	unis
wait, waitpid	Process	sys/
sbrk	Process	unis
getpagesize	Process	unis
getgid, getegid	Process	unis
gethostname	Process	unis
getpgrp	Process	unis
pipe	Process	unis
__sigsetjmp, siglongjmp	Process	setjm
getdtablesize	Process	unis
random, srandom	Random	stdli
rand, srand	Random	stdli
drand48, erand48, lrand48, nrand48, mrand48, jrand48, srand48, seed48, lcong48	Random	stdli
drand48_r, erand48_r, lrand48_r, nrand48_r, mrand48_r, jrand48_r, srand48_r, seed48_r, lcong48_r	Random	stdli
signal	Signal	sign
sigaction	Signal	sign
sigemptyset, sigfillset, sigaddset, sigdelset, sigismember	Signal	sign
sigprocmask	Signal	sign
qsort	Sort	stdli
strerror, strerror_h	String	strin
setlocale	String	loca
strcoll	String	strin
uselocale	String	loca
newlocale	String	loca
wctob	String	wch
btowc	String	wch
memset	String	strin
memcpy	String	strin
bcopy	String	strin
memcmp	String	strin
memmove	String	strin
memchr	String	strin
strcpy, strncpy	String	strin
strcat, strncat	String	strin
strcmp, strncmp	String	strin
strlen, strnlen	String	strin
strspn, strcspn	String	strin
strchr, strrchr	String	strin
strcasecmp, strncasecmp	String	strin
strdup, strndup	String	strin
getopt, getopt_long	String	unis
atoi, atol, atoll, atof	String	stdli
strtol, strtoll	String	stdli
strtoul, strtoull	String	stdli
strtod	String	stdli
toupper, tolower	String	ctyp
index, rindex	String	strin

Table 3.1 – continued from previous page

System Call Name	Domain	Inc
strtok, strtok_r	String	strin
sscanf	String	stdio
basename, dirname	String	libge
bindtextdomain, textdomain, gettext	String	libin
mbrlen	String	wch
strtoimax, strtoumax	String	intty
openlog, syslog, closelog, vsyslog, setlogmask	Syslog	syslo
pthread_create	Thread	pthre
pthread_exit	Thread	pthre
pthread_self	Thread	pthre
pthread_once	Thread	pthre
pthread_getspecific, pthread_setspecific	Thread	pthre
pthread_key_create	Thread	pthre
pthread_key_delete	Thread	pthre
pthread_mutex_destroy, pthread_mutex_init	Thread	pthre
pthread_mutex_lock, pthread_mutex_trylock, pthread_mutex_unlock	Thread	pthre
pthread_mutexattr_destroy, pthread_mutexattr_init	Thread	pthre
pthread_mutexattr_settype	Thread	pthre
pthread_cancel	Thread	pthre
pthread_kill	Thread	pthre
pthread_join	Thread	pthre
pthread_detach	Thread	pthre
pthread_cond_destroy, pthread_cond_init	Thread	pthre
pthread_cond_broadcast, pthread_cond_signal	Thread	pthre
pthread_cond_timedwait, pthread_cond_wait	Thread	pthre
pthread_condattr_destroy, pthread_condattr_init	Thread	pthre
sem_init	Thread Synchronization	sema
sem_destroy	Thread Synchronization	sema
sem_post	Thread Synchronization	sema
sem_wait, sem_trywait, sem_timedwait	Thread Synchronization	sema
sem_getvalue	Thread Synchronization	sema
sleep, usleep	Timer	unis
nanosleep	Timer	time
getitimer, setitimer	Timer	sys/t
timerfd_create, timerfd_settime, timerfd_gettime	Timer	sys/t
getgrnam	Users & Groups	grp.l
getrusage	Users & Groups	sys/r

CHAPTER

FOUR

DEVELOPER'S GUIDE (TO BE AVAILABLE SOON)

This document is for the people who want to develop DCE itself.

4.1 Kernel Developer Information

This technical documentation is intended for developers who want to build a Linux kernel in order to use with DCE. A first part will describe the architecture and the second will show how we went from a net-next kernel 2.6 has a Linux kernel-stable 3.4.5.

4.1.1 Prerequisite

You must be familiar with *ns-3*, DCE and Linux Kernel Developpement.

4.1.2 Download

The source code can be found in the following repository: http://code.nsnam.org/furbani/ns-3-linux. You must use mercurial to download the source code.

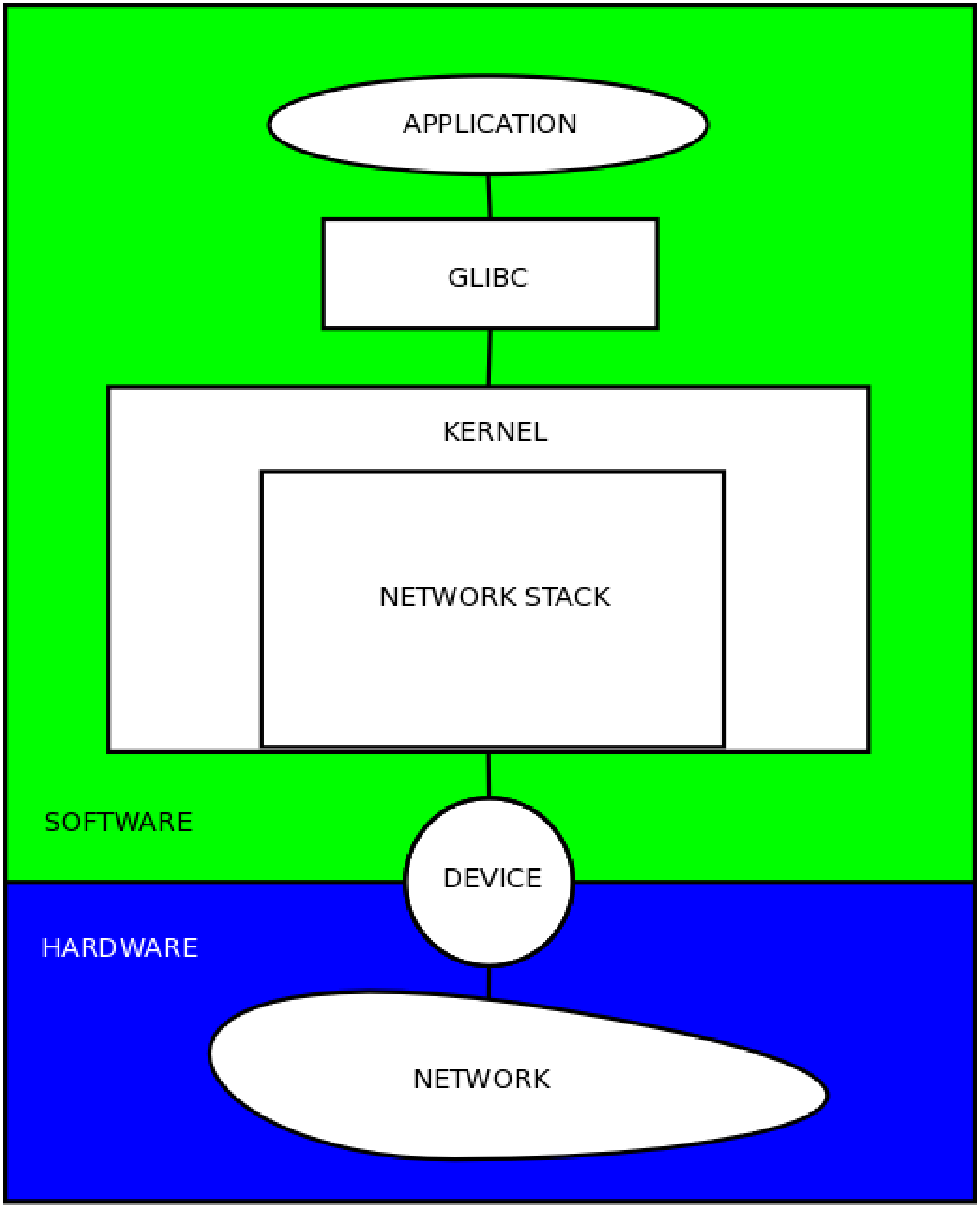
4.1.3 Goal

The goal of this work is to use the real implementation of the Linux Network Stack within the Simulation environnement furnished by *ns-3*.

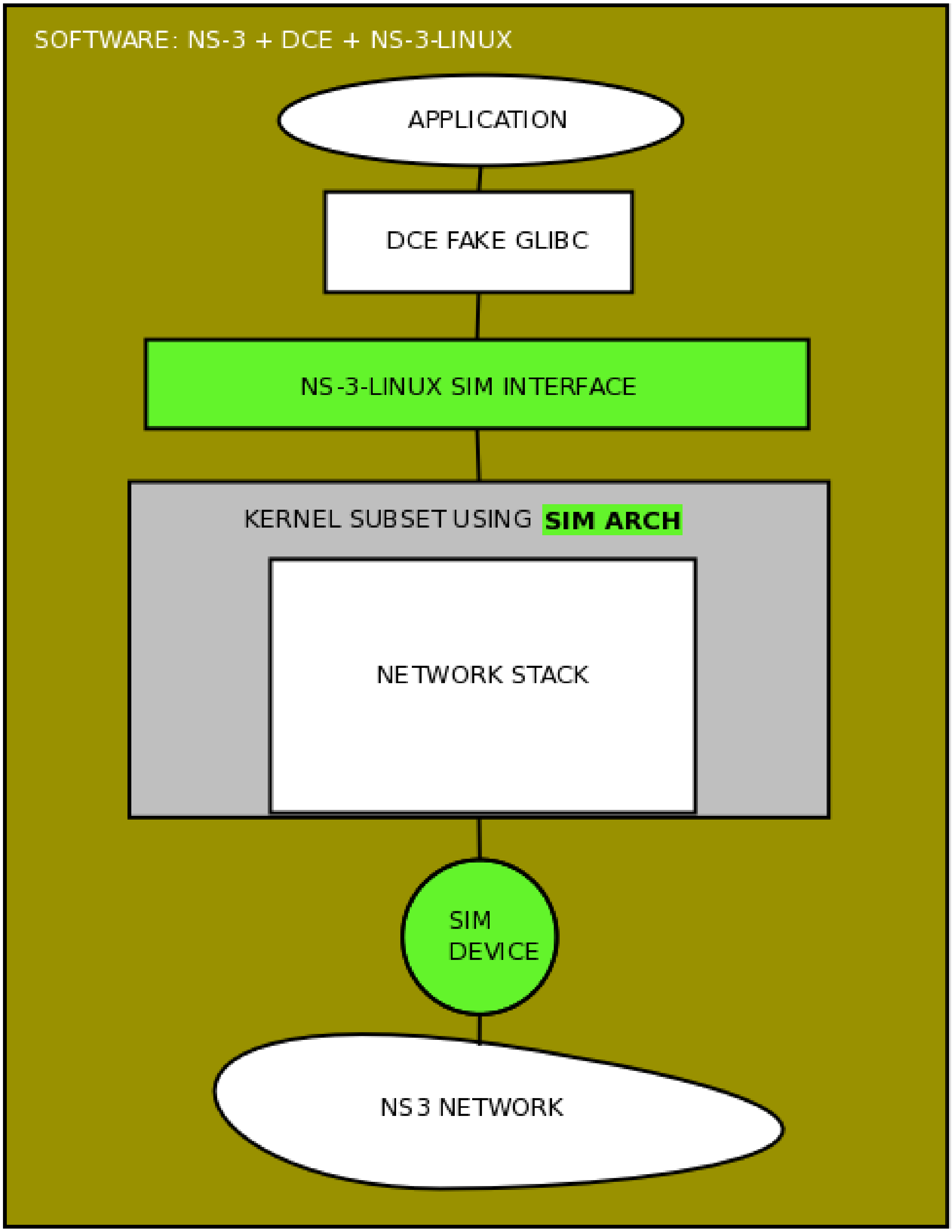
4.1.4 Solution

The solution chosen was to use the Linux kernel source, compile the Net part and make a dynamic library and interface the result with DCE.

The following schema show the differents parts between a software user space application and the hardware network.



The following schema show the same application running under DCE and *ns-3* and using a real kernel network stack:



The green parts are implemented in ns-3-linux source files, the grays parts comes from the Linux kernel sources and are not modified at all or with only few changes. Application should not be modified at all.

4.1.5 Concepts

If you need a more theorical documentation you can read the chapter 4.5 of this Ph.D thesis Experimentation Tools for Networking Research.

4.1.6 List of files and usage

After doing the cloning of the source,

```
$ hg clone http://code.nsnam.org/furbani/ns-3-linux
destination directory: ns-3-linux
requesting all changes
adding changesets
adding manifests
adding file changes
added 62 changesets with 274 changes to 130 files
updating to branch default
125 files updated, 0 files merged, 0 files removed, 0 files unresolved
```

Below are the files delivered under the directory *ns-3-linux*:

```
$ ls ns-3-linux/
generate-autoconf.py  generate-linker-script.py  kernel-dsmip6.patch  kernel.patch  Makefile  Makefil
```

The main file is the **Makefile** its role is to recover the kernel source, compile the NET part of the kernel and all that is necessary for the operation, the end result is a shared library that can be loaded by DCE.

```
$ ls ns-3-linux/sim
cred.c     glue.c     Kconfig    pid.c    random.c    seq.c         sim-socket.c  softirq.c  tasklet.
defconfig  hrtimer.c  Makefile   print.c  sched.c     sim.c         slab.c        sysctl.c   tasklet-
fs.c       include    modules.c  proc.c   security.c  sim-device.c  socket.c      sysfs.c    time.c

$ ls ns-3-linux/sim/include
asm  generated  sim-assert.h  sim.h  sim-init.h  sim-printf.h  sim-types.h
```

These directories contains the architecture specific code and the code doing the interface between the kernel and DCE and *ns-3*. Ideally we should not change a line of code outside the kernel arch portion, but in practice we make small changes : see the patchs files.

Recall: the code of Linux source is mainly C so it is very easy to port to new architecture, the architecture specific code is contained in a specific directory under arch/XXX directory where XXX name recall the processor used. In our case we have chosen to create a special architecture for our environment NS3 + DCE, we called **sim**.

4.1.7 Interfaces between Kernel and DCE

In order to install a kernel on a Node DCE do the following steps:

1. Load the shared library containing the kernel compilation result,
2. Call the init function called **sim_init**, this method is located in the just loaded library,
3. This **sim_init** method is called with a parameter which is a struct containing functions pointers to DCE methods able to be callable from the kernel part,
4. in return the **sim_init** fill a struct containing function pointers in kernel part which will be used by DCE to interract with the kernel part.
5. before finish **sim_init** must initialize the kernel to put it in a running state ready to be usable.

Kernel -> DCE

Methods (there is also one variable) of DCE called by the kernels are the following:

- LinuxSocketFdFactory::Vprintf
- LinuxSocketFdFactory::Malloc
- LinuxSocketFdFactory::Free
- LinuxSocketFdFactory::Memcpy
- LinuxSocketFdFactory::Memset
- LinuxSocketFdFactory::Random
- LinuxSocketFdFactory::EventScheduleNs
- LinuxSocketFdFactory::EventCancel
- CurrentNs
- LinuxSocketFdFactory::TaskStart
- LinuxSocketFdFactory::TaskWait
- LinuxSocketFdFactory::TaskCurrent
- LinuxSocketFdFactory::TaskWakeup
- LinuxSocketFdFactory::TaskYield
- LinuxSocketFdFactory::DevXmit
- LinuxSocketFdFactory::SignalRaised
- LinuxSocketFdFactory::PollEvent

there are located in the source file **linux-socket-fd-factory.cc** of DCE.

DCE -> Kernel

Methods of Kernel (sim part) called by DCE are the following:

- task_create
- task_destroy
- task_get_private
- sock_socket
- sock_close
- sock_recvmsg
- sock_sendmsg
- sock_getsockname
- sock_getpeername
- sock_bind
- sock_connect
- sock_listen
- sock_shutdown

- sock_accept
- sock_ioctl
- sock_setsockopt
- sock_getsockopt
- sock_poll
- sock_pollfreewait
- dev_create
- dev_destroy
- dev_get_private
- dev_set_address
- dev_set_mtu
- dev_create_packet
- dev_rx
- sys_iterate_files
- sys_file_read
- sys_file_write

the corresponding sources are located in the **sim** directory.

4.1.8 Build net-next 2.6 kernel

All build operations are done using the make command with the **Makefile** file under the directory **ns-3-linux**.

Make Setup

First you should call **make setup** in order to download the source of the kernel:

```
$ make setup
git clone git://git.kernel.org/pub/scm/linux/kernel/git/davem/net-next.git net-next-2.6; cd net-next-
fed66381d65a35198639f564365e61a7f256bf79
Cloning into net-next-2.6...
remote: Counting objects: 2441000, done.
remote: Compressing objects: 100% (377669/377669), done.
Receiving objects: 100% (2441000/2441000), 493.28 MiB | 28.45 MiB/s, done.
remote: Total 2441000 (delta 2043525), reused 2436782 (delta 2039307)
Resolving deltas: 100% (2043525/2043525), done.
Checking out files: 100% (33319/33319), done.
```

This sources correspond to a specific version well tested with DCE the net-next 2.6 and git tag = fed66381d65a35198639f564365e61a7f256bf79.

Now the directory **net-next-2.6** contains the kernel sources.

Make Menuconfig

Use **make menuconfig** to configure your kernel, note that modules are not supported by our architecture so options chosen as modules will not be included in the result kernel.

Build

Finally **make** will compile all the needed sources and produce a file named **libnet-next-2.6.so**: this is the library contains our net-next kernel suitable for DCE usage.

Usage

To use this kernel you should:

1. configure DCE in order to compile using the includes under **sim** directories to have the good interfaces between DCE and the kernel. For this you should give to the waf configure the path to the **ns-3-linux** directory ie:

```
$ ./waf configure ----enable-kernel-stack=/ABSOLUTE-PATH-TO/ns-3-linux
```

2. In your *ns-3* scenario you should indicate the good kernel file: (the file should be located in a directory presents in the DCE_PATH env. variable)

```
dceManager.SetNetworkStack("ns3::LinuxSocketFdFactory", "Library", StringValue ("libnet-next-2.6.so")
```

Test

Use DCE unit test:

```
$ ./waf --run "test-runner --verbose"
PASS process-manager 9.470ms
PASS Check that process "test-empty" completes correctly. 0.920ms
PASS Check that process "test-sleep" completes correctly. 0.080ms
PASS Check that process "test-pthread" completes correctly. 0.110ms
PASS Check that process "test-mutex" completes correctly. 0.200ms
PASS Check that process "test-once" completes correctly. 0.070ms
PASS Check that process "test-pthread-key" completes correctly. 0.070ms
PASS Check that process "test-sem" completes correctly. 0.080ms
PASS Check that process "test-malloc" completes correctly. 0.060ms
PASS Check that process "test-malloc-2" completes correctly. 0.060ms
PASS Check that process "test-fd-simple" completes correctly. 0.070ms
PASS Check that process "test-strerror" completes correctly. 0.070ms
PASS Check that process "test-stdio" completes correctly. 0.240ms
PASS Check that process "test-string" completes correctly. 0.060ms
PASS Check that process "test-netdb" completes correctly. 3.940ms
PASS Check that process "test-env" completes correctly. 0.050ms
PASS Check that process "test-cond" completes correctly. 0.160ms
PASS Check that process "test-timer-fd" completes correctly. 0.060ms
PASS Check that process "test-stdlib" completes correctly. 0.060ms
PASS Check that process "test-fork" completes correctly. 0.120ms
PASS Check that process "test-select" completes correctly. 0.320ms
PASS Check that process "test-nanosleep" completes correctly. 0.070ms
PASS Check that process "test-random" completes correctly. 0.090ms
PASS Check that process "test-local-socket" completes correctly. 0.820ms
PASS Check that process "test-poll" completes correctly. 0.320ms
PASS Check that process "test-exec" completes correctly. 0.380ms
PASS Check that process "test-iperf" completes correctly. 0.070ms
PASS Check that process "test-name" completes correctly. 0.080ms
PASS Check that process "test-pipe" completes correctly. 0.160ms
PASS Check that process "test-dirent" completes correctly. 0.070ms
PASS Check that process "test-socket" completes correctly. 0.270ms
PASS Check that process "test-bug-multi-select" completes correctly. 0.260ms
PASS Check that process "test-tsearch" completes correctly. 0.080ms
```

All is OK.

4.1.9 net-next 2.6 to linux-stable 3.4.5

Now we will try to use a more recent linux kernel. We start with a fresh clone of the ns-3-linux sources.

Makefile

First we need to modify the makefile in order to change the kernel downloaded. For that we need to modify the value of 2 variables:

1. KERNEL_DIR=linux-stable
2. KERNEL_VERSION=763c71b1319c56272e42cf6ada6994131f0193a7
3. KERNEL_DOWNLOAD=git://git.kernel.org/pub/scm/linux/kernel/git/stable/linux-stable.git

Also we need to remove the patch target named **.target.ts** because the patch will not pass for this newer version of kernel.

First Build

Now we can try to build:

```
$ make defconfig
$ make menuconfig
$ make
mkdir -p sim/
cc -O0 -g3 -D__KERNEL__ -Wall -Wstrict-prototypes -Wno-trigraphs -fno-inline -iwithprefix ./linux-sta
In file included from ./linux-stable/include/asm-generic/bitops.h:12:0,
                 from ./sim/include/asm/bitops.h:4,
                 from ./linux-stable/include/linux/bitops.h:22,
                 from ./linux-stable/include/linux/thread_info.h:52,
                 from ./linux-stable/include/linux/preempt.h:9,
                 from ./linux-stable/include/linux/spinlock.h:50,
                 from ./linux-stable/include/linux/wait.h:24,
                 from ./linux-stable/include/linux/fs.h:385,
                 from sim/fs.c:1:
./linux-stable/include/linux/irqflags.h:66:0: warning: "raw_local_irq_restore" redefined
./sim/include/asm/irqflags.h:8:0: note: this is the location of the previous definition
In file included from ./linux-stable/include/linux/wait.h:24:0,
                 from ./linux-stable/include/linux/fs.h:385,
                 from sim/fs.c:1:
./linux-stable/include/linux/spinlock.h:58:25: fatal error: asm/barrier.h: No such file or directory
compilation terminated.
make: *** [sim/fs.o] Error 1
```

Ok now we will try to fix the compilation errors trying not to change too the kernel source. In the following we will list the main difficulties encountered.

First Error

Recall: the linux source directory **include/asm-generic** contains a C reference implementation of some code that should be written in assembly langage for the target architecture. So this code is intented to help the developper to port to new architectures. So our sim implementation use many of these **asm-generic** include files. The first warning show that our code redefine a method defined elsewhere in kernel sources, so the fix is to remove our definition of this function in opur file named **sim/include/asm/irqflags.h**.

Second Error

The file **asm/barrier.h** is missing, we just create under sim/include/asm directory and the implementation is to include the generic one ie: **include/asm-generic/barrier.h**.

Change in sim method

Another problem arise the function named **kern_mount_data** defined in **sim/fs.c** do not compile any more. So we need to investigate about this function:

1. Where this function is located in the real code: in **linux/fs/namespace.c**
2. Why it is reimplemented in **sim/fs.c**: if you look at our Makefile why try to not compile all the kernel we focus on the net part only, you can see this line in the Makefile :

```
dirs=kernel/ mm/ crypto/ lib/ drivers/base/ drivers/net/ net/
```

in fact we include only this directories. So at this time we can comment the failing line and insert a **sim_assert (false);** in order to continue to fix the compilation errors, and then when we will do the first run test we will see if this method is called and if yes we will need to do a better fix. Remark: **sim_assert (false);** is a macro used to crash the execution, we often place it in functions that we need to emulate because required by the linker but that should never be called.

Change in our makefile

After we have the following problem while compiling **sim/glue.c** the macro **IS_ENABLED** is not defined. After some search we found that we need to include **linux/kconfig.h** in many files. So we modify our makefile to fix like this:

```
  -fno-stack-protector \
  -DKBUILD_MODNAME=\"nsc\" -DMODVERSIONS -DEXPORT_SYMTAB \
- -include autoconf.h \
+ -include $(SRCDIR)$(KERNEL_DIR)/include/linux/kconfig.h \
  -U__FreeBSD__ -D__linux__=1 -Dlinux=1 -D__linux=1 \
  -I$(SRCDIR)sim/include -I$(SRCDIR)$(KERNEL_DIR)/include \
  $(AUTOCONF): generate-autoconf.py $(KERNEL_DIR)/.config timeconst.h
  ./generate-autoconf.py $(KERNEL_DIR)/.config > $@
+ cp autoconf.h sim/include/generated/autoconf.h
+
  timeconst.h: $(KERNEL_DIR)/.config
      perl $(SRCDIR)$(KERNEL_DIR)/kernel/timeconst.pl $(CONFIG_HZ) > $@
```

Change in kernel source

Our **sim/slab.c** do not compile, in this case we want to use our implementation of memory allocation and to do this it is easier to modify slightly an include file in the kernel sources **include/linux/slab.h** :

```
--- a/include/linux/slab.h
+++ b/include/linux/slab.h
@@ -185,6 +185,8 @@ size_t ksize(const void *);
 #include <linux/slub_def.h>
 #elif defined(CONFIG_SLOB)
 #include <linux/slob_def.h>
```

```
+#elif defined(CONFIG_SIM)
+#include <asm/slab.h>
 #else
 #include <linux/slab_def.h>
 #endif
```

As we have already written we do not recommend to change the kernel sources to facilitate future upgrades.

First Launch

After a few corrections we finally get a library containing the kernel named **liblinux-stable.so**. At this moment we need to try it using DCE. For the beginning we will try with test-runner executable.

```
./test.py
assert failed. cond="handle != 0", msg="Could not open elf-cache/0/libnet-next-2.6.so elf-cache/0/lib
terminate called without an active exception
Aborted (core dumped)
```

We can see that a symbol is not defined : **noop_llseek**. We find this symbol defined in the kernel source named **fs/read_write.cc**. We need to choose a way to add this symbol in our kernel library, we can:

- rewrite it in a source under our sim directory,
- or add it in our makefile.

In this case we choose the second solution so we need to modify our makefile, first we see that the directory **fs** is not present in the **dirs** entry, so we need to add it in the write order (order is the same as found in the kernel Makefile defined by the variable named **vmlinux-main**); we also need to indicate that we want only the object **read_write.o**:

```
@@ -51,7 +52,7 @@
AUTOCONF=autoconf.h
# note: the directory order below matters to ensure that we match the kernel order
-dirs=kernel/ mm/ crypto/ lib/ drivers/base/ drivers/net/ net/
+dirs=kernel/ mm/ fs/ crypto/ lib/ drivers/base/ drivers/net/ net/
 empty:=
 space:= $(empty) $(empty)
 colon:= :
@@ -67,11 +68,12 @@
 ctype.o string.o kasprintf.o rbtree.o sha1.o textsearch.o vsprintf.o \
 rwsem-spinlock.o scatterlist.o ratelimit.o hexdump.o dec_and_lock.o \
 div64.o
+fs/_to_keep=read_write.o
```

Fake Function

We continue to try our kernel library, now another symbol is missing **generic_file_aio_read**, this symbol is defined in the source **mm/filemap.cc**, it is referenced at least by **read_write.c**. In this case we decided to create a fake function because the source **mm/filemap.cc** is voluminous and we do not want to take all the kernel sources. So we create a new source under **sim** directory named **sim/filemap.c** the body of the function is **sim_assert (false);** so if this function called sometimes we will be warned and we will write a more accurate version.

Assert

Later we meet again the function **kern_mount_data**, thanks to the presence of the sim_assert:

```
0x00007ffff5c8c572 in kern_mount_data (fs=<optimized out>, data=<optimized out>) at sim/fs.c:52
52      sim_assert (false);
(gdb) bt
#0  0x00007ffff5c8c572 in kern_mount_data (fs=<optimized out>, data=<optimized out>) at sim/fs.c:52
#1  0x00007ffff5d85923 in sock_init () at linux-stable/net/socket.c:2548
#2  0x00007ffff5c8d3aa in sim_init (exported=<optimized out>, imported=<optimized out>, kernel=<optin
#3  0x00007ffff7d9151b in ns3::LinuxSocketFdFactory::InitializeStack (this=0x65bde0) at ../model/linu
#4  0x00007ffff7d95ce4 in ns3::EventMemberImpl0::Notify (this=0x6597a0) at /home/furbani/dev/dce/dev/
#5  0x00007ffff76b10a8 in ns3::EventImpl::Invoke (this=0x6597a0) at ../src/core/model/event-impl.cc:3
#6  0x00007ffff7d8ff7c in ns3::LinuxSocketFdFactory::ScheduleTaskTrampoline (context=0x6597a0) at ../
#7  0x00007ffff7d3b7d4 in ns3::TaskManager::Trampoline (context=0x65d170) at ../model/task-manager.cc
#8  0x00007ffff7d37acd in ns3::PthreadFiberManager::Run (arg=0x65d5d0) at ../model/pthread-fiber-mana
#9  0x00000034be206ccb in start_thread () from /lib64/libpthread.so.0
#10 0x00000034bd6e0c2d in clone () from /lib64/libc.so.6
(gdb)
```

So this function is called by the initialisation, we must provide an implementation for it:

```
// Implementation taken from vfs_kern_mount from linux/namespace.c
struct vfsmount *kern_mount_data(struct file_system_type *type, void *data)
{
      static struct mount local_mnt;
      struct mount *mnt = &local_mnt;
      struct dentry *root = 0;

      memset (mnt,0,sizeof (struct mount));
      if (!type)
              return ERR_PTR(-ENODEV);
      int flags = MS_KERNMOUNT;
      char *name = type->name;
      if (flags & MS_KERNMOUNT)
              mnt->mnt.mnt_flags = MNT_INTERNAL;

      root = type->mount(type, flags, name, data);
      if (IS_ERR(root)) {
              return ERR_CAST(root);
      }

      mnt->mnt.mnt_root = root;
      mnt->mnt.mnt_sb = root->d_sb;
      mnt->mnt_mountpoint = mnt->mnt.mnt_root;
      mnt->mnt_parent = mnt;
//    br_write_lock(vfsmount_lock);   DCE is monothreaded , so we do not care of lock here
      list_add_tail(&mnt->mnt_instance, &root->d_sb->s_mounts);
//    br_write_unlock(vfsmount_lock);  DCE is monothreaded , so we do not care of lock here

      return &mnt->mnt;
}
```

Here we do not want to integrate all the code namespace.c, so we copy and paste the function named **kern_mount_data**. This solution has the advantage of minimizing code size, the disadvantage is that it can introduce problems if the next version of the kernel need changes in this function.

4.1.10 Conclusion

We will not describe the rest of the port here. But after some iteration we end up with a version that works correctly. Sometimes we should not hesitate to use **gdb** to trace the actual execution and correct accordingly code. The rules that we can gain from this experience's are as follows:

1. Be patient,
2. Try to not modify the kernel sources,
3. Be pragmatic,
4. Try to not import all the kernel code into our library,
5. Do not hesitate to go back and test other alternatives.

4.2 DCE - POLL IMPLEMENTATION

4.2.1 Introduction

The implementation of the poll system call is inspired by the Linux kernel, therefore we well study first the kernel poll implementation.

Kernel implementation

Firstly in the kernel every type of file descriptor (file, socket, pipe ...) must provide a function named poll and conform to this prototype:

```
int poll(struct file *file, poll_table *pwait);
```

Where **file** is a pointer to a structure representing the file (it looks like **this** in C++) and **pwait** is a pointer to a poll table, **pwait** may be NULL, we will see later why. The return integer of this function is a mask of poll events which have already occured on the corresponding file descriptor. The behavior of this function is as follows: 1. It is not a blocking function, it immediately returns the mask of events regardless of event desired by the caller of poll. 2. if **pwait** is not NULL then it add **pwait** in the wait queue of **file**, and secondly a pointer to the wait queue is stored in the poll table **pwait**.

Thus an event on the file will ascend to the poll, and in the opposite direction when the poll ends it can de-register itself from the wait queue of the file.

Now that we know the function **poll** of **file**, we can study the **poll** system call, here the following pseudo code commented:

```
POLL( .... )
{
   poll_table table; // This table will contain essentially the list of wait queues that need to wake
                     // and also information about the current thread in order to be awakened.
   poll_table *pwait=&table; // pointer to current poll table.

   while (true)
   {
      foreach( fd ) // For each file descriptor ...
      {
             file *file = get_file(fd); // Retrieve **file** data structure corresponding to fd.

         if (!file)
           mask = POLLNVAL; // fd does not correspond to an open file.
         else
               mask = file->poll (file, pwait); // During the first loop pwait is not NULL.

         if (mask)
```

```
             {
               count++; // Increases the number of responses
           pwait = NULL; // Once we have at least one response POLL should not be blocking,
                         // so we nullify the pointer to the poll table in order to not register the
             }
      }
      pwait = NULL; // For the next loops we must not re-register to the wait queue of files.

      if (count) break; // we have a result.
      if (timeout) break; // it is too late.

      Wait(timeout); // Put to sleep until awakening from a file or because of the time limit.
   }
   poll_freewait(&table); // Removes the reference to the poll table from each file’s wait queue.

   return count;
}
```

DCE implementation

As we have already seen, the poll will look like that of the kernel. Firstly we create a virtual method named Poll in the class UnixFd. This method will do the same job that the function **poll** seen early in the struct file of the kernel linux implementation. Before writing the function dce_poll which is our implementation of poll we need to create some classes for mimic the role of the poll table and the wait queues.

So we add 2 sources files named wait-queue.h and wait-queue.cc in order to implements poll table and wait queue.

It is also on this occasion that I deleted all the objects used to wait which was allocated on the stack, and I replaced by objects allocated in the heap. Concerning the dce-poll function it looks like the kernel one with some differences. The more important difference is that the PollTable can not be allocated on the stack so it cannot be a local variable, so the PollTable object is allocated with a C++ new. I guess you’re wondering why the poll table can not be allocated on the stack, it is because of the fork implementation of DCE. Indeed, if a process makes a fork, this creates another stack which use the same memory addresses, thus another thread of the same process can not use an object allocated on this stack, and when a event of a file want to wake up the poll thread it will use especially this poll table. So allocating the Poll Table in the heap generates a side effect which is that we need to release this memory if another thread call exit while we are within the dce-poll. So we need to register the Poll Table somewhere in a DCE data, and the DCE place choosen is the thread struct (in file **model/process.h**), because each thread can be in doing a poll. Thus there is a new field in struct thread which is:

```
PollTable *pollTable; // No 0 if a poll is running on this thread
```

There is another reason to have this field, this reason arises from the fact that a file descriptor can be shared by multiple processes (thanks to dup fork ...), thus when a process exit while doing a poll, we need to deregister from the corresponding wait queues refered by the poll table.

Poll kernel implementation

Concerning the kernel implementation the dce-poll method is the same but the difference cames from the Poll method specialized implementation of the class herited from UnixFd and which correspond to a File Descriptor open with the help of the Kernel Linux. For example the class LinuxSocketFd represents a socket which is opened in the kernel, therefore the method **poll** of LinuxSocketFdFactory will do much work.

Now look at the interface between DCE and the kernel, in the direction DCE to kernel, we use 2 functions which are **sock_poll** and **sock_pollfreewait**, and in the other direction there is **sim_poll_event**. **sock_poll** obviously has the same semantics as the kernel poll. **sock_poll** has the following signature:

```
void sock_poll (struct SimSocket *s, void *ret);
```

where **s** represents the socket int the kernel and **ret** is a pointer to a data structure of type **struct poll_table_ref**:

```
struct poll_table_ref
{
  int ret;
  void *opaque;
};
```

This structure allows the kernel to pass a reference to the poll table DCE via the opaque field. This reference will be used by the kernel only to warn DCE that event just happened on socket, this using the function **sim_poll_event (void *ref)**. In return this function modifies the value of opaque and assign it a pointer to a core structure which represents an entry in the wait queue of the socket. This value will be used by DCE for unregister it from the wait queue using the function **sock_pollfreewait function (void * ref)**. The field **ret** is also affected in return and it contain the mask of poll events which have already occured on the corresponding socket. Most of the kernel code is in the file sim-socket.c it consists of two structures, and the following functions:

NAME	DESCRIPTION
sim_pollwake	Function called by the kernel when the arrival of an event on the socket, if the event is expected by DCE, the function forwards it to DCE.
sim_pollwait	Function called by the kernel, its role is to register the poll table in the wait queue.
sim_sock_poll	Function called by DCE, it is the interface between the DCE's poll and the kernel's poll.
sim_sock_pollfreewait	Function called by DCE allows it to unregister from the wait queue.
struct poll_table_ref	This is the same struct as that of DCE.
struct sim_ptable_entry	This is used for the entry in the wait queue of the socket.

# TODO add example , gdb breakpoint to follow the behavior in live

CHAPTER

FIVE

HOW IT WORKS (TO BE AVAILABLE SOON)

If your are interrested to know why DCE exists and how it can works, you should read this document Experimentation Tools for Networking Research (and in particular the chapter 4) written by the principal author of DCE Mathieu Lacage. Also you can read the sources too.

5.1 Introduction

You know that the goal of DCE is to execute actual binaries within *ns-3*. More precisely a binary executed by DCE perceives time and space of *ns-3*, rather than the real environnenent.

To do this, DCE does a job equivalent to an operating system like:

1. DCE loads in memory the code and data of executable,
2. DCE plays the role of intermediary between the executable and the environment through the systems functions called by executables,
3. DCE manages and monitors the execution of processes and handles liberate the memory and close open files when the stop process.
4. DCE manages the scheduling of the various virtual processes and threads.

5.2 Main classes and main data structures

5.2.1 DceManager

The **DceManager** is somewhat the entry point of DCE. It will create virtual processes and manages their execution. There is an instance of **DceManager** which is associated to each node which need to virtualize the execution of a process. In reality, the developer uses the classes **DceManagerHelper** and **DceApplicationHelper**.

I invite you to look at the source code *dce-manager.cc* and *dce-manager.h* and particularly the public methods **Start**, **Stop**, **Exit**, **Wakeup**, **Wait** and **Yield**; the following private methods are also important: **CreateProcess**, **PrepareDoStartProcess**, **DoStartProcess**, **AllocatePid**, **TaskSwitch**, **CleanupThread** and **LoadMain**. The **Start** method is called when starting the executable, if you look at, it begins by initializing an object of type **struct Process**. **struct Process** is very important, it contains information about the virtual processes that DCE creates, this type is described below. **Start** then also initializes a structure of type **struct thread**, it represents the principal thread in which the **main** entry of the executable will run. Finally **Start** asks the **TaskManager** to create a new **Task** and to start this one using the method **DceManager::DoStartProcess** as the entry point.

Class **TaskManager** is a major class of DCE, it is described below.

5.2.2 Process

struct process contains everything you need to describe a process, I invite you to study the source code in the file *process.h*.

This structure contains references to standard objects for example a list of open files via a vector of FILE *, but it contains also the references to objects useful to manage DCE threads, the memory allocated by the process ...

Field **openFiles** represents the open file descriptors of the process, the key is the fd and the value a pointer to an object of type DCE **FileUsage**. The field **threads** contains all the threads in the process see description below. Field **loader** is a pointer to the **Loader** used to load the corresponding code.

The **alloc** field is important it is a pointer to the memory allocator used to allocate the memory used by this process, at the end of the process it will liberate all the memory allocated so as simple and efficient.

5.2.3 Thread

struct thread represents a thread, It contains various fields including a pointer to the **process** to which it belongs, and also a pointer to **Task** object described later.

5.2.4 Taskmanager and Task

The **TaskManager** manages the Tasks, ie the threads of virtualized processes by DCE. It allows you to create new task. It controls the activity of the task by the following methods: **Stop**, **Wakeup**, **Sleep** and **Yield**. A **Task** possesses a stack which contains the call stack functions. There is one instance of **TaskManager** per node. The implementation of **TaskManager** is based on a class of type **FiberManager** described below.

5.2.5 FiberManager

FiberManager actually implements the creation of the execution contexts. These contexts are called fiber. **FiberManager** offers the following:

1. Create a context, it returns a fiber
2. Delete a fiber
3. Yield hand to another fiber

DCE provides two implementations:

1. **PthreadFiberManager**, which is based on the pthread library,
2. **UcontextFiberManager** which is based on the POSIX API functions offered by ucontext.h: **makecontext**, **getcontext** and **setcontext**.

I invite you to watch the corresponding man.

5.2.6 LoaderFactory and Loader

The **Loader** is a very important object of DCE. A DCE **Loader** loads the executable code in memory of a special way, load several times the same executable, while isolating each of the other executable. The **Loader** must link the executable loaded with the 3 libraries emulated ie lib C, lib pthread and lib rt. The same way the libraries used by the executable must also be linked with the emulated libraries.

DCE offers several actually Loader:

1. **CoojaLoader**: it has the following characteristics: it loads into memory only a copy of the code, by cons it duplicates data (ie global variables and static). For each change of context there are 2 memory copies: backup data of the current context then restoration of context memory that will take control. Comment: it is rather reliable,the size of the copied memory size depends on the total static and global variables, and in general there is little, in a well designed executable.
2. **DlmLoader** : Uses a specialized loader to not duplicate the code but only the data but without special operations to do when changing context. Comment: offers the best performance in memory and cpu, but not very reliable especially during the unloading phase.

5.3 Follow a very simple example

After theory, do a bit of practice. Follow the execution of very simple example.

You can find the used sample under the directory named **myscripts/sleep**. This executable used by the scenario do only a sleep of ten seconds:

```
#include <unistd.h>

int main(int c, char **v)
{
  sleep (10);

  return 1;
}
```

The *ns-3*/DCE scenario execute **tenseconds** one time starting at time zero:

```
#include "ns3/core-module.h"
#include "ns3/dce-module.h"

using namespace ns3;

int main (int argc, char *argv[])
{
  NodeContainer nodes;
  nodes.Create (1);

  DceManagerHelper dceManager;
  dceManager.Install (nodes);

  DceApplicationHelper dce;
  ApplicationContainer apps;

  dce.SetStackSize (1<<20);

  dce.SetBinary ("tenseconds");
  dce.ResetArguments ();
  apps = dce.Install (nodes.Get (0));
  apps.Start (Seconds (0.0));

  Simulator::Stop (Seconds(30.0));
  Simulator::Run ();
  Simulator::Destroy ();
}
```

First we can launch **tenseconds** binary:

```
$ ./build/bin_dce/tenseconds
```

after 10 seconds you retrieve the prompt.

Then we can try the DCE scenario:

:: $ ./build/myscripts/sleep/bin/dce-sleep

This time the test is almost instantaneous, because the scenario is very simple and it uses the simulated time.

Same test by activating logs:

```
$ NS_LOG=DefaultSimulatorImpl ./build/myscripts/sleep/bin/dce-sleep
DefaultSimulatorImpl:DefaultSimulatorImpl(0x6928c0)
DefaultSimulatorImpl:SetScheduler(0x6928c0, ns3::MapScheduler[])
0s -1 DefaultSimulatorImpl:ScheduleWithContext(0x6928c0, 0, 0, 0x692ab0)
0s -1 DefaultSimulatorImpl:ScheduleWithContext(0x6928c0, 0, 0, 0x695220)
0s -1 DefaultSimulatorImpl:Stop(0x6928c0, 30000000000)
0s -1 DefaultSimulatorImpl:Schedule(0x6928c0, 30000000000, 0x692c10)
0s -1 DefaultSimulatorImpl:Run(0x6928c0)
0s -1 DefaultSimulatorImpl:ProcessOneEvent(): [LOGIC] handle 0
0s 0 DefaultSimulatorImpl:Schedule(0x6928c0, 0, 0x695630)
0s 0 DefaultSimulatorImpl:ProcessOneEvent(): [LOGIC] handle 0
0s 0 DefaultSimulatorImpl:ProcessOneEvent(): [LOGIC] handle 0
0s 0 DefaultSimulatorImpl:ProcessOneEvent(): [LOGIC] handle 0
0s 0 DefaultSimulatorImpl:Schedule(0x6928c0, 10000000000, 0x6954c0)
0s 0 DefaultSimulatorImpl:ProcessOneEvent(): [LOGIC] handle 10000000000
10s 0 DefaultSimulatorImpl:ProcessOneEvent(): [LOGIC] handle 10000000000
10s 0 DefaultSimulatorImpl:ProcessOneEvent(): [LOGIC] handle 30000000000
30s -1 DefaultSimulatorImpl:Stop(0x6928c0)
DefaultSimulatorImpl:Destroy(0x6928c0)
DefaultSimulatorImpl:Destroy(): [LOGIC] handle destroy 0x6928a0
DefaultSimulatorImpl:DoDispose(0x6928c0)
DefaultSimulatorImpl:~DefaultSimulatorImpl(0x6928c0)
```

We can see that an event occurs at 30s it is the end of the simulation corresponding to the line:

```
Simulator::Stop (Seconds(30.0));
```

We can also see that at 10s an event occurs, this is the end of our **sleep(10)**.

Now we do the same experiment using the debugger:

```
$ gdb ./build/myscripts/sleep/bin/dce-sleep
(gdb) b ns3::DceManager::DoStartProcess
(gdb) b ns3::DceManager::Start
(gdb) run
Breakpoint 4, ns3::DceManager::Start (this=0x630c50, name=..., ....  at ../model/dce-manager.cc:403
403     NS_LOG_FUNCTION (this << name << stackSize << args.size () << envs.size ());
(gdb) bt
#0  ns3::DceManager::Start (this=0x630c50, name=.....) at ../model/dce-manager.cc:403
#1  0x00007ffff7cb5e19 in ns3::DceApplication::StartApplication (this=0x633520) at ../model/dce-appli
#2  0x00007ffff71dea6e in ns3::EventMemberImpl0::Notify (this=0x633650) at ./ns3/make-event.h:94
#3  0x00007ffff76148af in ns3::EventImpl::Invoke (this=0x633650) at ../src/core/model/event-impl.cc:4
#4  0x00007ffff76194c3 in ns3::DefaultSimulatorImpl::ProcessOneEvent (this=0x6308e0) at ../src/core/m
#5  0x00007ffff761986a in ns3::DefaultSimulatorImpl::Run (this=0x6308e0) at ../src/core/model/default
#6  0x00007ffff76155dd in ns3::Simulator::Run () at ../src/core/model/simulator.cc:160
#7  0x00000000004075af in main (argc=1, argv=0x7fffffffdaa8) at ../myscripts/sleep/dce-sleep.cc:25
  (gdb) info thread
  Id   Target Id         Frame
* 1    Thread 0x7ffff6600740 (LWP 7977) "dce-sleep" ns3::DceManager::Start (this=0x630c50, .... ) at
```

You can notice that:

1. We have two breakpoints.
2. After run the first stop is in **ns3::DceManager:Start**.
3. At this time there is only one thread.
4. We are currently processing an event, this event was scheduled by the call **apps.Start (Seconds (0.0));** of our scenario.

Now we continue our execution:

```
(gdb) continue
Continuing.
[New Thread 0x7ffff65fc700 (LWP 8159)]
[Switching to Thread 0x7ffff65fc700 (LWP 8159)]

Breakpoint 3, ns3::DceManager::DoStartProcess (context=0x633d50) at ../model/dce-manager.cc:274
274     Thread *current = (Thread *)context;
(gdb) info thread
  Id   Target Id         Frame
* 2    Thread 0x7ffff65fc700 (LWP 8159) "dce-sleep" ns3::DceManager::DoStartProcess (context=0x633d5
  1    Thread 0x7ffff6600740 (LWP 7977) "dce-sleep" pthread_cond_wait@@GLIBC_2.3.2 () at ../nptl/sysc
(gdb) bt
#0  ns3::DceManager::DoStartProcess (context=0x633d50) at ../model/dce-manager.cc:274
#1  0x00007ffff7d21b90 in ns3::TaskManager::Trampoline (context=0x633bd0) at ../model/task-manager.c
#2  0x00007ffff7d1da87 in ns3::PthreadFiberManager::Run (arg=0x634040) at ../model/pthread-fiber-mana
#3  0x00000034be206ccb in start_thread (arg=0x7ffff65fc700) at pthread_create.c:301
#4  0x00000034bd6e0c2d in clone () at ../sysdeps/unix/sysv/linux/x86_64/clone.S:115
```

You can notice that:

1. Now there is a second thread
2. Gdb break execution in **ns3::DceManager::DoStartProcess** in the context of the second thread

This second thread is the thread corresponding to the main thread of our hosted executable **tenseconds**, if you look at **ns3::DceManager::DoStartProcess** you can notice that we are on the point of calling the main of **tenseconds**:

```
void
DceManager::DoStartProcess (void *context)
{
  Thread *current = (Thread *)context;
  int (*main)(int, char **) = PrepareDoStartProcess (current);
  int retval = 127;

  if (main)
    {
      StartProcessDebugHook ();
      retval = main (current->process->originalArgc, current->process->originalArgv);
    }
  dce_exit (retval);
}
```

You can also see that the pointer to the **main** is the result of the method **ns3::DceManager::PrepareDoStartProcess**. Now we can put a breakpoint before the sleep of **tenseconds** and follow the code of sleep:

```
(gdb) break tenseconds.c:5
(gdb) continue
Breakpoint 1, main (c=1, v=0x630b30) at ../myscripts/sleep/tenseconds.c:5
5       sleep (10);
(gdb) list
(gdb) step
sleep () at ../model/libc-ns3.h:193
193   DCE (sleep)
(gdb) step
dce_sleep (seconds=10) at ../model/dce.cc:226
226     Thread *current = Current ();
(gdb) list
224   unsigned int dce_sleep (unsigned int seconds)
225   {
226     Thread *current = Current ();
227     NS_LOG_FUNCTION (current << UtilsGetNodeId ());
228     NS_ASSERT (current != 0);
229     current->process->manager->Wait (Seconds (seconds));
230     return 0;
231   }
(gdb)  bt
#0  dce_sleep (seconds=10) at ../model/dce.cc:226
#1  0x00007ffff62cdcb9 in sleep () at ../model/libc-ns3.h:193
#2  0x00007ffff5c36725 in main (c=1, v=0x630b30) at ../myscripts/sleep/tenseconds.c:5
#3  0x00007ffff7c9b0bb in ns3::DceManager::DoStartProcess (context=0x633d50) at ../model/dce-manager.
#4  0x00007ffff7d21b90 in ns3::TaskManager::Trampoline (context=0x633bd0) at ../model/task-manager.cc
#5  0x00007ffff7d1da87 in ns3::PthreadFiberManager::Run (arg=0x634040) at ../model/pthread-fiber-mana
#6  0x00000034be206ccb in start_thread (arg=0x7ffff65fc700) at pthread_create.c:301
#7  0x00000034bd6e0c2d in clone () at ../sysdeps/unix/sysv/linux/x86_64/clone.S:115
(gdb) info thread
  Id   Target Id         Frame
* 2    Thread 0x7ffff65fc700 (LWP 15233) "dce-sleep" dce_sleep (seconds=10) at ../model/dce.cc:226
  1    Thread 0x7ffff6600740 (LWP 15230) "dce-sleep" pthread_cond_wait@@GLIBC_2.3.2 () at ../nptl/sys
(gdb)
```

We can notice that **sleep** call **dce_sleep** which call **Wait**, this **Wait** method is from the class **TaskManager**. **TaskManager** is a major class of **DCE** and we will detail it below. Basically **Wait** schedules and event in *ns-3* event queue (in order to be woken up after sleep time) and give the control to another **Task**. Now we can put a breakpoint in **ns3::DefaultSimulatorImpl::ProcessOneEvent** and see the time advance up to 10s:

```
gdb) b ns3::DefaultSimulatorImpl::ProcessOneEvent
Breakpoint 2 at 0x7ffff7619207: file ../src/core/model/default-simulator-impl.cc, line 131.
(gdb) c
Continuing.
[Switching to Thread 0x7ffff6600740 (LWP 3942)]
Breakpoint 2, ns3::DefaultSimulatorImpl::ProcessOneEvent (this=0x6308e0) at ../src/core/model/default
warning: Source file is more recent than executable.
131     Scheduler::Event next = m_events->RemoveNext ();
(gdb) n
133     NS_ASSERT (next.key.m_ts >= m_currentTs);
(gdb) n
134     m_unscheduledEvents--;
(gdb) n
136     NS_LOG_LOGIC ("handle " << next.key.m_ts);
(gdb) n
137     m_currentTs = next.key.m_ts;
(gdb) n
138     m_currentContext = next.key.m_context;
```

```
(gdb) p m_currentTs
$1 = 10000000000
(gdb)
```

This next event will wake the thread 2 will therefore complete the **sleep** of our scenario.

In summary we saw briefly that DCE uses the events of *ns-3* to schedule the execution between different tasks.

CHAPTER

SIX

SUBPROJECTS OF DCE

Below are the list of the tested applications with DCE.

- CCNx
- Quagga
- iperf
- ping/ping6
- ip (iproute2 package)
- umip (Mobilt IPv6 daemon)
- Linux kernel (from 2.6.36 to 3.7 version)
- thttpd
- torrent

6.1 CCNx

CCNx is an open source implementation of **Content Centric Networking**. All the C written executables are supported and none of the Java ones. The versions tested are those between 0.4.0 and 0.6.1 included.

For more detail document, see `dce-ccnx`.

6.2 Quagga

Quagga is a routing software suite, providing implementations of OSPFv2, OSPFv3, RIP v1 and v2, RIPng and BGP-4 for Unix platforms, particularly FreeBSD, Linux, Solaris and NetBSD. For more information, see the latest support document.

6.3 iperf

iperf from the following archive http://walami.googlecode.com/files/iperf-2.0.5.tar.gz as been tested. It is the exception that proves the rule. That is to say that this particular example requires a change in its code. In the source file named `Thread.c` at line 412 in the function named `thread_rest` you must add a `sleep(1)` in order to help DCE to break the infinite loop.

6.4 ping/ping6

Ping from the following archive http://www.skbuff.net/iputils/iputils-s20101006.tar.bz2 is supported.

6.5 ip (iproute2 package)

6.6 umip (Mobilt IPv6 daemon)

The umip (Usagi-Patched Mobile IPv6 stack) support on DCE enables the users to reuse routing protocol implementations of Mobile IPv6. UMIP now supports Mobile IPv6 (RFC3775), Network Mobility (RFC3963), Proxy Mobile Ipv6 (RFC5213), etc, and can be used these protocols implementation as models of network simulation.

For more information, see the latest support document.

6.7 Linux kernel (from 2.6.36 to 3.7 version)

(TBA)

6.8 thttpd

(TBA)

6.9 torrent

(TBA)

CHAPTER

SEVEN

ABOUT

The creator of DCE is Mathieu Lacage.

The current maintainers and developers are Hajime Tazaki and Frédéric Urbani.

7.1 Contacts

frederic.urbani AT inria.fr
tazaki AT sfc.wide.ad.jp
mathieu.lacage AT cutebugs.net