Simulator-Agnostic ns–3 Applications

John Abraham, George Riley
Department of Electrical and Computer Engineering
Georgia Institute of Technology, Atlanta, GA. USA
{john.abraham, riley} @gatech.edu

ABSTRACT

An important part of network simulation is the *application*, which has specific traffic patterns that try to imitate a *real-world application*. For instance, an on-off application with random *on* and *off* durations can be used to model the behavioral pattern of a human using a web-browser. But the traffic pattern generated by an *ns-3 application* is only an estimate and is limited by the underlying mechanics of discrete-event simulation. We propose a method by which an *ns-3 application* can be re-used as a *real-world application*. Such an application can be written and tested using *ns–3*, and instantly deployed in the real-world with confidence. The application would be agnostic of whether it is being used for simulation or as a *real-world application*.

Categories and Subject Descriptors

I.6 [**Simulation and Modeling**]: Applications

General Terms

Portable Applications, Simulation

Keywords

Network Simulation, Applications

1. INTRODUCTION

The *ns–3* [1] simulator currently has eleven applications that generate traffic patterns that can be used to simulate real-world applications. For instance, the *bulk-send* application transfers data in bulk to a TCP end-point. Thus an FTP upload operation can be modeled by using the *bulk-send* application. Often, a researcher or a network-designer would like to analyze the performance of an application under various network conditions. We list a few such scenarios

1. A network-designer testing if a database replication software handling 2 TB a day can be deployed over a WAN link with a propagation delay of 40 ms.
2. A researcher trying to determine if a UDP-based application with a low data-rate can remain loss-free in a mobile environment which uses a new routing protocol.
3. An online multi-player game developer trying to guarantee the responsiveness of the game-play when users are situated across a wide geography.
4. A developer of a peer-to-peer application trying to determine the maximum independent download threads that a peer can support based on current round-trip time estimates to that peer.

Considerable effort is required to ensure that the *ns–3 application* generates traffic patterns similar to real-world applications. In addition the following restricts the *ns–3 application* from being directly used as a *real-world application*

1. The *ns–3 application* must conform to the rules of *ns–3* specific implementations of *callbacks*.
2. The *ns–3 application* must use `Simulator::Schedule()` to schedule events at specific time intervals during simulation.
3. The *ns–3 application* will also need to use *ns–3*'s support infrastructure such as logging and asserts.
4. The *ns–3 application* is typically aggregated to a node as an object and is started when *ns–3* initializes the node.

It is therefore likely that the developer is forced to write two applications, the *ns–3 application* and the *real-world application* hoping that the former behaves similar to the latter. On the other hand, there are significant advantages in developing applications using *ns–3* even if there was no intent to test it in simulation. The advantages include the use of the powerful scheduling , callback, tracing and attribute infrastructure in *ns–3*. We introduce a method by which, the application ceases to be a classified as an *ns–3 application* or a *real-world application* and needs to be written exactly once. The application thus developed will be agnostic of whether a simulator-based scheduler or a real-time scheduler is being used.

Section 2 discusses related work. In section 3 we will describe the methodology. Section 4 explains a few experimental results.

2. RELATED WORK

Lacage [2] presented an extension of *ns–3* with a Direct Code Execution (DCE) module which is capable of executing within the simulator existing user space as well as kernel space protocol implementations. This module enables the researcher to conduct realistic experiments by providing the ability to use native application

WNS3 2012, March 23, Desenzano del Garda, Italy

DOI 10.4108/icst.simutools.2012.247744

code such as a BGP daemon and the native TCP/IP stack while *ns–3* can be a shim in between. This requires knowledge of relatively complex concepts such as ELF loading. In addition, the portability is related to the version of the *libc* API. Our focus in this paper is strictly with the application-layer and we emphasize the simplicity of our approach to deploy an application in the real-world. However, we also feel our work can complement the DCE work in that, *ns–3* currently cannot easily determine node-processing times. Node-processing time is an important requirement for realistic simulations. Further, *ns–3* conforms to standards although many commercial systems do not follow them. For instance, a system behind a proxy may choose to not participate in congestion-control or may have a customized TCP library. In such situations, it is advangtageous to test an application written using our approach, by using the DCE environment to load the true TCP/IP stack and adjust the application's behavior based on the stack's characteristics determined during testing.

NRL Protocol Prototyping library (ProtoLib) [3] provides a cross-platform C/C++ library that allows applications to be built while supporting a variety of platforms as well as the simulation environment of NS2. Currently, there is no *Protolib* support on *ns–3*. While there are similarities, especially in the need for a socket-abstraction layer, our approach focuses on the application's need to bypass the *ns–3* TCP/IP stack completely when ready to be deployed in the real-world. In other words, our approach is geared towards encouraging the developer to write the application completely using *ns–3* to leverage its powerful features mentioned earlier. That is, we don't intend our approach to be used to send application traffic to other simulated nodes.

3. METHODOLOGY

Our approach involves only two additional modules to *ns–3*. Those are a *real-world scheduler* and a *real-world sockets layer*. The existing *ns–3* code is not modified. When an application written for network simulation needs to be used in the real-world, one has to replace the existing simulation-based scheduler with a *real-world scheduler* and the simulation-based sockets with the *real-world sockets*. We emphasize that the socket application programming interface (API) used by the application remains the same. For instance, the *ns–3* application `OnOffApplication` uses `m_socket->Send(packet)` to send packets. The API for our real-world sockets layer is identical, and thus the application need *not* take different actions depending on real or simulated environments.

In the following sub-sections we will describe these modules. In the interest of brevity we will switch to pseudo-code when applicable and remove exception-handling code.

3.1 Real-world scheduler

Currently, *ns–3* uses class `DefaultSimulatorImpl` to provide the following essential functions.

1. Scheduling events at specified times using the various `Schedule()` functions.

2. Providing the current simulation time using the `Now()` function.

3. Event processing main loop using the `Run()` function.

Our real-world scheduler is the class *RealImpl* which implements the essential functions of `DefaultSimulatorImpl` as mentioned above from the perspective of real-world deployments rather than simulation. Discrete-event simulation schedulers use simulation time rather than wall-clock time. `RealImpl` has to strictly use wall-clock time.

In addition `RealImpl` provides a mechanism to notify sockets belonging to other modules of read events using the socket API's `select` function. We briefly describe two essential implementation details

3.1.1 Main event-processing loop

As with the most discrete-event simulators, `RealImpl` has a main event-processing `while` loop that dequeues events serially from the event list (sorted by ascending order of time they are scheduled for) and processes the associated callbacks for each event. The key point here is that events should be processed in real–time as closely as possible. If the earliest pending event in the event list is still in the future, the main event loop should of course not process that event yet. However, there may be received data available on any of the open sockets, so we take this time to use the `select` function to check if such data is available. If the `select` function notifies us that one or more sockets do have data available, that data will be read and delivered to the application using the same callback approach used by *ns–3*.

```
1   while (event_list_is_not_empty) {
2       ProcessOldEvents ();
3       struct timeval delta = NextDelta();
4       if (delta < 0) continue;
5       Build_read_fd_list ();
6       Build_write_fd_list ();
7       int ret =
8       select (&read_fd_list,
9         &write_fd_list,&delta);
10      if (ret < 0 && errno == EBADF)
11       {PurgeSockets (); continue;}
12      else if (ret == 0) continue
13      else {FdReadNotify (); FdWriteNotify
            ();}
```

Line 1: the scheduler-dispatcher's `while` loop runs until there are no more events scheduled in the queue.

Line 2: Processes events in the queue up to the current time. This is a significant difference from the simulation case. In simulation, the event-processing loop dequeues the next event and assumes all events before that have been processed. This at first seems to indicate that `RealImpl` is processing events in the past. While this may be true, we will discuss later and show through experiments that the timestamp of the event in the past does not differ too much from the current time to be perceived by the application. In essence, the application would likely perceive the event being processed at exactly the current time.

Line 3: Obtain the time difference (delta) between current time and the timestamp of the next event. We note that there is resolution of `struct timeval` is microseconds, unlike *ns–3* which can have a resolution of nano-seconds. This loss in resolution is usually not perceived at the application level.

Line 4: If the delta calculated above is negative, this would mean our current time has exceeded the timestamp of the next event

in the queue. Therefore, we would immediately return to process line 2. If not, this indicates that we have time available to perform the `select` call to check for socket activity.

Line 5: Build a list of socket file descriptors whose availability for reading is to be checked. Other modules such as the socket layer will be responsible for registering with `RealImpl` for notifications that a certain socket has data available to be read.

Line 6: Build a list of socket file descriptors whose availability for writing is to be checked. The only such cases for now, are the modules desiring to know if a non-blocking socket `connect` operation succeeded.

Line 7-9: Here the socket API's `select` function performs two tasks. Firstly, it serves to implement a delay corresponding to the delta calculated in line 3. This aims to run the application in terms of wall-clock time. Secondly, `select` will wait until any socket file descriptor is available for reading. The return value of the `select` call is stored in ret to be used later. It is worth noting that at any point only one active timer exists for the entire run and the implementation is single-threaded.

Line 10-11: If any socket had closed during the `select` call we purge that socket from the system and notify registered modules such as the socket layer

Line 12: This statement is executed if `select` timed-out while waiting for the delta duration calculated in line 3. This implies we are now ready to process next event in the queue.

Line 13: This statement is executed if a socket file descriptor was available for reading or writing and `select` did not time-out. We will describe `FdReadNotify ()` and `FdWriteNotify ()` in the sections that follow.

3.1.2 FdReadNotify

This function is executed only if the `select` call (described in the previous section) detects that a socket file descriptor is available for reading. If the file descriptor was a listening socket, we notify the socket layer about the newly accepted socket. Otherwise, we notify the registered modules that the socket is available for reading.

3.1.3 FdWriteNotify

This function is executed only if the `select` call detects that a socket file descriptor is available for writing. A typical use for this is when the socket layer wants to be notified if a previously scheduled non-blocking socket `connect` call was successful.

3.2 Real-world sockets

When a user wants to re-use an *ns–3 application* as a *real-world application* they would use the real-world socket layer we implemented for both TCP and UDP transports. However, recall our goal is to keep the application agnostic of the nature of the underlying scheduler (whether real or simulated). Fortunately, the applications in *ns–3* use an abstract base class `Socket` to make socket calls such as `Recv`. The implementation of the abstract `Socket` class for the case of TCP is `TcpSocketBase`. Likewise, our real-tcp socket implementation will be `RealTcpSocket` which implements `Socket`. The class diagram in Figure 1 illustrates that.

`TcpSocketBase` and `RealTcpSocket` have the responsibility of implementing functions such as `Bind, Recv, Send`.

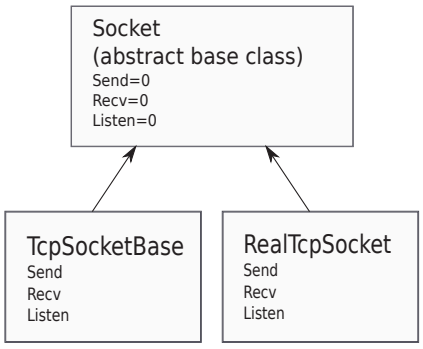

Figure 1: Socket Class Hierarchy

3.2.1 Real-tcp socket implementation

The real-world TCP socket implementation is in class `RealTcpSocket`. It will implement base class functions such as `Bind, Recv, Send, Listen` primarily by utilizing genuine *socket API* calls such as `bind, recv, send, listen` respectively. In addition, to basic socket API calls, we will briefly describe other relevant portions of the implementation.

RealTcpSocket::Constructor.

```
1   BlockWrite ();
2   s = socket (AF_INET, SOCK_STREAM, 0);
3   fcntl (s, O_NONBLOCK);
4   realImpl->RegisterFdReadCallback (s,
        &RealTcpSocket::AllowRead);
```

Line 1: Initially, we block any write on the socket. A future write operation will be permitted on a socket only if the `select` call in `RealImpl` detects that the socket is available for writing. The typical case is when a socket write operation has to be blocked until a non-blocking `connect` operation succeeds.

Line 2: The routine creation of a socket of type SOCK_STREAM.

Line 3: Set the socket to non-blocking mode. In *ns–3* socket API calls such as `Connect` are non-blocking. We will retain the non-blocking nature of the sockets in the real implementation as well.

Line 4: The socket layer registers with `RealImpl` for notifications when a given socket is readable. The callback `AllowRead` will be called when the socket is declared readable during the `select` call in `RealImpl`. `AllowRead` will notify the application that data is available to be read.

RealTcpSocket::Connect.

The `TcpSocketBase::Connect` function in *ns–3* is a non-blocking call. When the connect operation succeeds a callback is notified using `TcpSocketBase::ConnectionSucceeded`. Our approach in `RealTcpSocket` can be described by the following pseudo-code

```
1   int ret = connect (s, &remote);
2   if (ret == -1 && errno != EINPROGRESS))
```

```
3       {NS_FATAL_ERROR ("Connect");}
4     realImpl->RegisterFdWriteCallback (s,
          &RealTcpSocket::AllowWrite);
```

Line 1: Use non-blocking `connect` to the remote end-point

Line 2-3: As it is a non-blocking connect only an errno==EINGPROGRESS is acceptable.

Line 4: Register with `RealImpl` for notifications when a given socket is writeable (in this case `connect` has succeeded). The callback `AllowWrite` will be called when the socket is declared writeable during the `select` call in `RealImpl`.

RealTcpSocket::Recv.

An application can read data from a socket in two ways. In the first method, data is read from a socket after it gets a notification from the socket layer that data is available to be read. In the second method, a direct call to `Recv` can be made. The call to `Recv` is non-blocking. We preserve this mechanism in `RealTcpSocket` while using the socket API's `recv` function.

RealTcpSocket::Send.

Similar to `TcpSocketBase`, `RealTcpSocket` will prevent any send operations until the socket is connected. The connect notification will be received from `RealImpl` via FdWriteNotify mentioned previously.

Other `TcpSocketBase` calls such as `Bind, Listen` have similar equivalents in `RealTcpSocket` except that the calls are from the real socket API using `bind, listen` respectively.

3.2.2 Real-udp socket implementation

The real-udp socket implementation is in class `RealUdpSocket`. The implementation is relatively trivial as it mostly includes using function calls such as `RecvFrom` and `SendTo` as wrappers around socket API calls to `recvfrom` and `sendto` respectively. Therefore the details of the Real UDP Socket Implementation are omitted here.

3.3 Using the new modules

Assuming we have a written a program using *ns–3 application*s only two steps are required to use this application as a *real-world application*.

1. Offload `DefaultSimulatorImpl` and use the real-world scheduler, `RealImpl`.
2. Replace `TcpSocketFactory` with `RealTcpSocketFactory` if we are using TCP transport or replace `UdpSocketFactory` with `RealUdpSocketFactory` if we are using UDP transport.

We will now compare the two cases of using the application, first as an *ns–3 application* and second as *real-world application*.

Case 1: As an ns–3 application.

```
1 PacketSinkHelper
2  ("ns3::TcpSocketFactory",
3     InetSocketAddress
4     (Ipv4Address::GetAny (), sinkPort));
```

Line 1-4: This is a common case of using the *ns–3 application*'s `PacketSinkHelper` class of the `PacketSink` application.

Case 2: As a real-world application.

```
1 Simulator::SetImplementation (CreateObject <
      RealImpl> ());
2 PacketSinkHelper
3  ("ns3::RealTcpSocketFactory",
4     InetSocketAddress
5     (Ipv4Address::GetAny (), sinkPort));
```

Line 1: Here we offload `DefaultSimulatorImpl` and use the real-world scheduler, `RealImpl`. This statement needs to be applied only once in the entire program.

Line 2-5: Here we replace *ns3::TcpSocketFactory* with `ns3::RealTcpSocketFactory`. It is worth noting that the application (`PacketSink`) is not modified to be made aware of the fact that, it is being used for simulation or as a *real-world application*.

4. EXPERIMENTAL RESULTS

To test our approach, we will use a set of TCP-based applications and a set UDP-based applications. For each set, we will compare the application's performance for the simulated-case versus the real-case

1. OnOffApplication with PacketSink using TCP
2. UdpEchoClient with UdpEchoServer using UDP

4.1 OnOffApplication with PacketSink

This test has the following specifications
Transport : TCP
Number of nodes : 2
Operating-system for real nodes: Fedora 14, Ubuntu 11
Link-type : Point-to-point
OnOffApplication count : 5
PacketSink count : 1
Link rate : 1 Gbps
Propagation delay : 0.5 ms
OnOffApplication on time : UniformVariable (0,4)
OnOffApplication off time: UniformVariable (0,5)
RngRun : 1 to 5.

Only one set of *RngRun* is chosen for illustration here.

The plot of throughput vs simulation time observed for a single flow for the simulated case is shown in Figure 2.

The plot of throughput vs wall-clock time observed for a single flow for the real case is shown in Figure 3

The plot of TCP Sequence number vs Simulation time observed for a single flow for the simulated case is shown in Figure 4

The plot of TCP Sequence number vs Wall-clock time observed for a single flow for the real case is shown in Figure 5

It can be seen that, to a reasonable extent the application performs similar for both cases. Outliers however do exist and are marked with red squares in Figure 2.

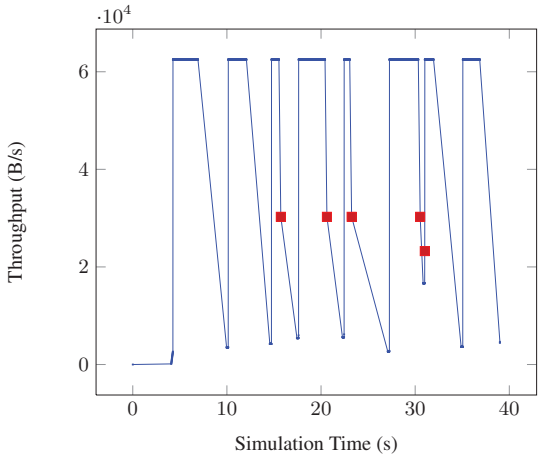

Figure 2: OnOffApp as an *ns–3 application*: Throughput vs Simulation time

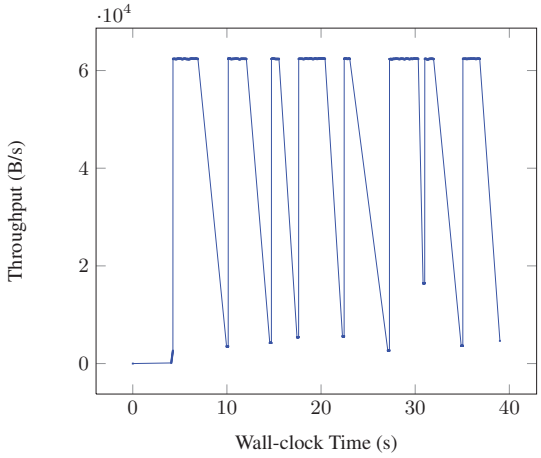

Figure 3: OnOffApp as a *real-world application*: Throughput vs Wall-clock time

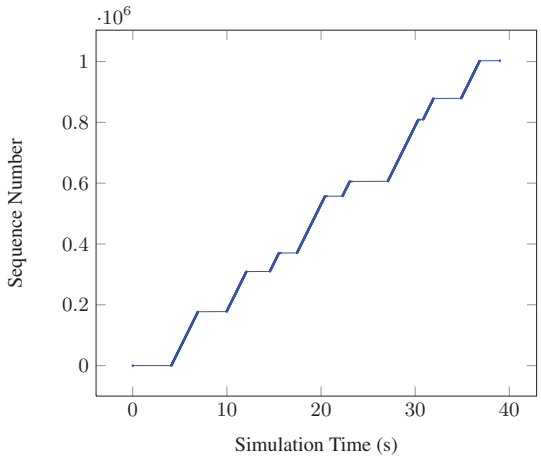

Figure 4: OnOffApp as an *ns–3 application*: TCP Sequence number vs Simulated time

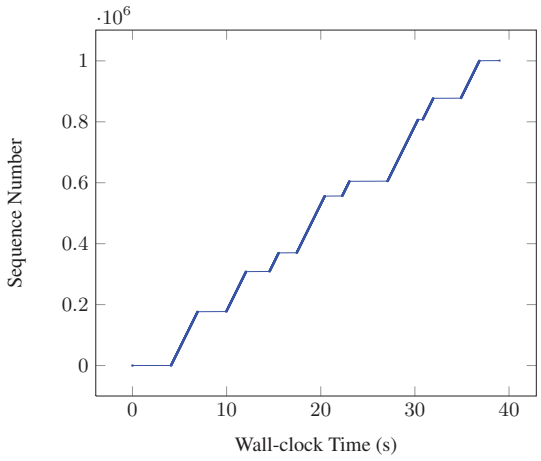

Figure 5: OnOffApp as a *real-world application*: TCP Sequence number vs Wall-clock time

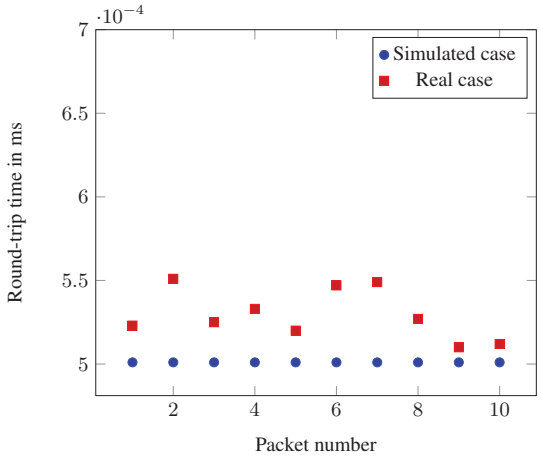

Figure 6: UdpEchoClient with UdpEchoServer (Simulated and Real case): Packet Number vs Round-trip time in ms

4.2 UdpEchoClient with UdpEchoServer

This test has the following specifications
Transport : UDP
Number of nodes : 2
Operating-system for real nodes: Fedora 14, Ubuntu 11
Link-type : Point-to-point
UdpEchoClient count : 1
UdpEchoServer count : 1
PacketSink count : 1
Link rate : 1 Gbps
Propagation delay : 0.5 ms
RngRun : 1 to 5

Only one set of *RngRun* is chosen for illustration here.

The plot of packet number vs round-trip time is shown in Figure 6 for both the simulated and real case. It is observed that the round-trip time measured by the application in the real case is quite close to that of the simulated case.

5. SUMMARY

Application performance is often used in network simulation as an indicator of the robustnes of a network or a node's TCP/IP stack.

Usually, a developer has to write this application twice, one for network simulation and another for the real-world. We have presented a simple approach to developing applications using *ns–3* that can be tested in a simulation environment and easily deployed in the real-world. Our experimental results, for both TCP and UDP applications, show that the behaviorial pattern of the applications can be retained to a good degree of accuracy. The accuracy can be further improved if *ns–3* can model processing time in its TCP/IP stack.

References

[1] The NS3 Development Team. *The NS3 Network simulator*. 2011. URL: http://www.nsnam.org.

[2] Mathieu Lacage. "Experimentation Tools for Networking Research". PhD thesis. University of Nice-Sophia Antipolis, 2010.

[3] U.S. NRL Networks and Communications Systems Branch. *Protean Protocol Prototyping Library*. 2011. URL: http://cs.itd.nrl.navy.mil/work/protolib.