ns3 for beginners

A guide towards running Hyrax simulations

Joaquim M. E. Silva
jqmmes@gmail.com

December 2015

Contents

Chapter 1

Introduction

1.1 Installing

1.2 Configurations

1.3 Running a simulation

Chapter 2

Simulations

2.1 Overlay Simulation

2.2 Technologies Experiment

Running:

```
./waf --run="scratch/Experiment/Experiment --Nodes=1 --Servers=1 --Scenario=3 --Seed=$RANDOM --ExclusiveServers"
```

Parameters:

Nodes: Number of Nodes to be used in the simulation

Servers: Number of Servers to be used in the simulation

Scenario: Scenario to run

- 1: 1 Server + AP + n Nodes
- 2: AP + m Mobile Servers + n Nodes (m¡=n)
- 3: AP + TDLS + m Mobile Servers + n Nodes (m¡=n)
- 41: WD + GO as Server + n Nodes
- 42: WD + GO + m Mobile Servers + n Nodes (m¡=n)
- 43: WD + m Mobiles Servers + n Nodes (m¡=n) No groups formed in the beggining
- 51: WD + Legacy AP as Server + n Nodes
- 52: WD + Legacy AP + m Mobile Servers + n Nodes (m¡=n)
- 6: WD + GO + TDLS + m Mobile Servers + n Nodes (m¡=n)

FileSize: File Size to be shared

Debug: Debug socket callbacks

ShowPackets: Show every packet received

ShowData: Show Send/Receive instead of the time a transfer took

Seed: Seed to be used

ExclusiveServers: Use Exclusive Server. (Server Don't act as Client)

SegmentSize: TCP Socket Segment Size

2.3 CMU Review App Simulation

Chapter 3

Code

3.1 Network Configurations

3.2 *VirtualDiscovery*

Public Methods:

```
void VirtualDiscovery::add(Ipv4Address ip, uint16_t port)
tuple<Ipv4Address,uint16_t> VirtualDiscovery::discover(void)
vector<tuple<Ipv4Address,uint16_t>> VirtualDiscovery::getAll(void)
uint32_t VirtualDiscovery::GetN(void)
void VirtualDiscovery::remove(Ipv4Address ip, uint16_t port)
```

3.3 TDLS

Public Methods:

```
void SendTDLS(Ipv4Address ip, uint16_t port, std::string message)
```

Algorithms:

Algorithm 1: TDLS ns3 Algorithm - Client

Data: *message* - Message to be sent; *socket* - TDLS (using Wi-Fi Ad-hoc) socket
Result: A message is sent using TDLS or regular Wifi as fallback
Input : *timeout* - duration until CheckTDLS fallback occurs
Output: nothing
Function `SendTDLS(`*socket, message*`) /* Algorithm to Send a message with TDLS */`
  if $ActiveTDLSCons < MAX$ **then**
1 $socket \rightarrow connect(ServerIp)$;
2 TDLSData[*socket*] $\leftarrow$ *socket* ;
3 TDLSData[*message*] $\leftarrow$ *message* ;
4 TDLSData[*delivered*] $\leftarrow$ false ;
5 $ActiveTDLSCons++$;
6 `Schedule(CheckTDLS(`*socket, message*`),` *timeout*`)`;
  else
7 $RegularSocket \rightarrow connect(ServerIp)$;
8 $RegularSocket \rightarrow send(message)$;
  end
end
Function `CheckTDLS(`*socket, message*`)`
  if TDLSData[*socket*] = *socket* $\wedge$ TDLSData[*delivered*] = $false$ **then**
9 $RegularSocket \rightarrow connect(ServerIp)$;
10 $RegularSocket \rightarrow send(message)$;
  end
  `/* Deletes Hashmap entry */`
11 `DeleteEntry(`TDLSData[*socket*]`)` ;
end
Callback `ConnectSuccess` *(socket)* `/* Callback called if` *socket* $\rightarrow$ *connect*(*ServerIp*) `succeeds */`
12 TDLSData[*delivered*] $\leftarrow$ true ;
end

Algorithm 2: TDLS ns3 Algorithm - Server

Data: *socket* - TDLS (using Wi-Fi Ad-hoc) socket
Result: A message is received using TDLS or regular Wifi as fallback
Input : MAX - Maximum number of simultaneous TDLS sockets opened
Output: nothing
Function `TDLSAccept(`*ListenSocket*`)`
  if $ActiveTDLSCons < MAX$ **then**
1 $ActiveTDLSCons++$;
2 *socket* $\rightarrow$ *setAcceptCallback*(`SecondPhaseAccept`) ;
3 **return** true;
  end
4 **return** false;
end
Callback `SecondPhaseAccept(`*socket*`)`
5 $ConnectedTDLS \leftarrow socket$;
end

Algorithm 3: TDLS ns3 Algorithm - Closing Socket

```
   Function CloseSocket(socket)
1  |  socket → Close() ;
   |  if ConnectedTDLS = socket then
2  |  |  ActiveTDLSCons − − ;
3  |  |  ConnectedTDLS ← Null ;
   |  end
   end
```

3.4 Wifi-Direct

3.5 Mobility

Chapter 4

Advanced

4.1 Developing a new simulation from scratch

4.2 Tracing

4.3 Parallel Execution

4.4 Direct Code Execution