Establishing a multi-hop network testbed with 802.11 end-nodes using FreeBSD on Emulab

Naeem Khademi[*]
Centre for Advanced Internet Architectures, Technical Report 130618A
Swinburne University of Technology
Melbourne, Australia
naeemk@ifi.uio.no

Abstract—**This technical report provides detailed information on establishing a multi-hop network testbed using FreeBSD nodes provided by *Emulab* project. Emulab is a networking testbed that provides researchers with a wide range of environments in which to develop, debug, and evaluate their proposed protocols and systems based on their desired topology. In this paper our aim is to introduce a generic method to create multi-hop networks potentially with 802.11 capability on the end-nodes. We provide details on establishing three different architectures as: 1) multi-hop wired; 2) multi-hop with wireless nodes on the edge LANs; and 3) multi-hop cascading (a.k.a parking lot) architecture suitable for multi-bottleneck scenarios.**

I. Introduction

Establishing network testbeds with different topologies, architectures and in various sizes has always been a major challenge for networking researchers. Ability to create and customize such testbeds, enables the researchers to develop, test and validate a wide range of networking protocols and measure their performance under different scenarios. Setting up a customized testbed for examining a certain protocol can be time-consuming and expensive due to the cost of hardware. Several low-cost methodologies are being used by networking researchers to get around issue, among them *virtualization* and *emulation* techniques. For instance network stack virtualization is provided by FreeBSD jails. While virtualization techniques can be seen as a suitable option for testing different networking protocols, their performance is subjected to the load on the host node and one virtual nodes' resource consumption may possibly affect the others' performance. Moreover, virtualization techniques lead to ignorance the complexities at network device's hardware level (e.g. such as physical buffers at transmitter chipsets) while emulation techniques often lack interactivity and mask probabilistic randomness in realistic networking scenarios.

[*]The author performed this work while visiting from the Networks and Distributed Systems Group, Department of Informatics, University of Oslo, Norway.

On the other hand, another alternative is to employ platforms with actual networking infrastructure such as in Emulab [1] that is publicly available to the researchers. Emulab provides physical access to the actual hardware and facilitate building up medium-size network topologies. The results obtained from Emulab can be highly reliable compared to virtualization techniques as it relies on the physical hardware setup. In this paper, we have taken the latter approach to establish a multi-hop networking testbed. We explain this in detail in the rest of this paper, and provide complete versions of all code discussed in this paper in [2].

A. Emulab

Creating different topologies in Emulab is achieved using its ns-like code format. An associated "experiment" for the desired topology should be created using Tcl code in a similar way to ns-2 [3]. However, instead of simulation, Emulab interprets and translates the Tcl code to hardware-related commands and configurations. Although not all ns-2 functionalities are implemented in Emulab, the ns-like code gives the flexibility for establishing different types of topologies by masking the hardware-related complexities and providing a level of abstraction. In addition we have implemented an additional set of standard shell scripts that facilitate the testbed configuration.

Emulab's users and project folders are accessible via *users.emulab.net* host. Emulab uses a variety of hardware suitable for different experimental purposes. In our experiments and also within the scope of this document, we use two types of Emulab's PC nodes namely *pc600wifi* nodes, for the experiments requiring 802.11 capability, and *pc850* nodes for the wired-only experiments. Table I summarizes the hardware setup of each type.

TABLE I
EMULAB'S HARDWARE SETUP

PC types	pc850	pc600wifi
CPU	Intel PIII 850 MHz	Intel PIII 600 MHz
RAM	512 MB	
Exp. Interface(s)	4×Intel 82557/8/9/0/1 Ethernet Pro 100	
802.11 NIC chipset	None	AR5413

B. Network topologies

Three different network topologies are of our interest. In the rest of this paper, we firstly explain how to create a simple dumbbell topology with FreeBSD 9.0 nodes in which the number of intermediate hops (e.g. routers) on the main path can be pre-assigned. Later, we extend this to enable the 802.11 capability on one or both edges of the networks and finally we provide further extensions to the code, to create a *cascade* topology which is a suitable topology for studying scenarios with multiple bottlenecks along the path.

II. A BASIC TOPOLOGY: DUMBBELL WIRED NETWORK

We initially start with a simple topology such as a dumbbell wired network as follows. First, create a ns file and edit it with a command to create a ns instance.

```
set ns [new Simulator]
source tb_compat.tcl
```

Then, define several important variables which identify the topology such as what type of OS image should be loaded to the nodes upon assignment, how many nodes should exists in either of the two LANs on the both edges of the dumbbell topology (a.k.a peers), how many routers should be between the two LANs (e.g. number of desired hops + 1) and the links' initial PHY bandwidth. While the required bandwidth can be set using the ns node, it is recommended to always set the link bandwidth to 100 Mbps in the ns code and adjust the required bandwidth using $ipfw$ and $dummynet$ later from the *experiment script* if necessary.

```
set os_image "FBSD9-WITHSRC"
set max_peers 4
set router_no 2
set link_bw "100Mb"
```

Also define where the initial configuration files are stored on the Emulab's NFS shared folder. In this case *config.sh* script, which is stored on */proj/project_folder/XXX* will be later used to establish a customized initial configuration.

```
set project_folder "/proj/project_folder/XXX"
set config_file "config.sh"
```

Now, create the instances the PC nodes which will represent the actual physical nodes. In this example, we identify the nodes based on their hostnames (which is identical to the node names) and allow Emulab to automatically assign them with their IP addresses. Based on this, each node may take different role in our future experiments such as being a sender, receiver, router or a control box [1]. We add each node's name to a string so that it can be later used to be assigned to a specific LAN. In this example, each sender and receiver node will be added to lanstr(1) and lanstr($router\_no$) strings respectively where $router\_no$ is the total number of routers in the topology.

```
set ctlbox [$ns node]
for {set i 1} {$i<=$router_no} {incr i} {
  set router($i) [$ns node]
  set lanstr($i) ""
  append lanstr($i) "$router($i) "
}
for {set i 1} {$i<=$max_peers} {incr i} {
  set send($i) [$ns node]
  set recv($i) [$ns node]
  append lanstr(1) "$send($i) "
  append lanstr($router_no) "$recv($i) "
}
```

In the above code we have also created a "control box" so that it can be used to take the role of swapping in and out the experiments. Now we can assign the desired OS image to be loaded on each of the nodes at startup. A list of available OS images on Emulab can be found on Emulab's web interface. In our example, all available nodes will have the same OS e.g. FreeBSD 9.0 with source code (FBSD9-WITHSRC). Alternatively, a custom OS image can also be created by the user. The in-depth detail of custom OS image creation is explained in Emulab's tutorial[4].

```
tb-set-node-os $ctlbox $os_image
for {set i 1} {$i<=$router_no} {incr i} {
  tb-set-node-os $router($i) $os_image
}
for {set i 1} {$i<=$max_peers} {incr i} {
  tb-set-node-os $send($i) $os_image
  tb-set-node-os $recv($i) $os_image
}
```

In this scenario, we need two LANs at the edges of the network namely lan(1) and lan(2); lan(1) for the senders

[1]We can use *control box* node to run the experiments from.

and lan(2) for the receivers. These two LANs can be created using the below commands with the desired PHY link bandwidth and delay e.g. 100 Mbps and 0 ms delay respectively. All nodes within the same LAN will be connected via a network switch.

```
set lan(1) [$ns make-lan "$lanstr(1)"
$link_bw 0ms]
set lan(2) [$ns make-lan
"$lanstr($router_no)" $link_bw 0ms]
```

Once LANs at the both sides of the path are created, proceed to create the connectivity between the routers using PHY links as

```
set link_no [expr {$router_no - 1}]
for {set i 1} {$i<=$link_no} {incr i} {
  set next_router_idx [expr {$i + 1}]
  set link($i) [$ns duplex-link
  $router($i) $router($next_router_idx)
  $link_bw 0ms DropTail]
}
```

which connects each router to its subsequent router. It is also possible to create a separated network than the experimental network for contol box to be connected to every other node. Extra care should be taken so that the control traffic does not affect the experimental data flow and necessity of a separated control network will be dependant on the scenario. Separating the control network from the experimental network requires every other node to have an available network interface for control traffic. However in this example, we have chosen to connect the control box to one of the routers for the sake of simplicity and the control link is created as below.

```
set ctllink [$ns duplex-link $router(1)
$ctlbox 100Mb 0ms DropTail]
```

Routing between the nodes and the routers is configured using the next command; in this case we have chosen to use *static* routing.

```
$ns rtproto Static
```

Once the topology is generated, the initial configuration script can be executed as a startup command on every node. *tb-set-node-startcmd nodeX "command"* runs a command on startup.

```
tb-set-node-startcmd $ctlbox
"$project_folder/$config_file"
for {set i 1} {$i<=$router_no} {incr i} {
  tb-set-node-startcmd $router($i)
  "$project_folder/$config_file"
}
for {set i 1} {$i<=$max_peers} {incr i} {
  tb-set-node-startcmd $send($i)
  "$project_folder/$config_file"
  tb-set-node-startcmd $recv($i)
  "$project_folder/$config_file"
}
```

The executed script identifies type of the node (e.g. sender, receiver or a router) and makes additional configurations on it accordingly. Additional configurations can include for instance, installing a traffic generator e.g. *iperf*, setting *ipfw* rules on the router(s) or installing a specific kernel patch on the senders. Figure 1 shows a schematic example of a dumbbell network topology created by this method when $router\_no = 2$ and $max\_peers = 4$. Finally, the below command runs the ns instance and should be always put at the end of the ns script.

```
$ns run
```

It is now easy to extend this to a wired multi-hop network by simply changing the value of $router\_no$. Next section provides more details on adding 802.11 functionality to the end-nodes.

III. A multi-hop network with 802.11 nodes on the edge LANs

We can now extend the script in previous section to provide 802.11 functionality to the end-nodes on $lan(1)$ and $lan(2)$. Define a "wireless flag"" for each of the LANs so that if it is set to true, the LAN will be configured as an 802.11 network, otherwise as a wired network.

```
set send_mac_80211 1
set recv_mac_80211 1
```

Bear in mind that since Emulab's 802.11 support is limited to FC4 (Fedora Core 4) and RHL90 (RedHat 9.0) Linux distributions only, extra configurations is needed to configure 802.11 nodes running FreeBSD 9.0 kernels. This for instance includes loading the driver modules, creating a *wlanX* interface, assigning IP addresses, etc. Here, define few parameters which later will be used for setting the IP addresses, switching to the desired 802.11 PHY/MAC mode (e.g. 11a or 11g), enabling or disabling the default "Rate Adaptation" mechanism [2], setting the frequency channels and SSIDs for both LANs. Also define a new config file for 802.11 nodes for the sake of clarity.

```
set send_80211_ipaddr_domain "10.1.4"
set recv_80211_ipaddr_domain "10.1.5"
set netmask "24"
set mode_80211 "11g"
```

[2]Rate Adaptation (RA) mechanism's aim is to choose the most appropriate modulation and coding scheme (bit-rate) depending on the channel conditions such as noise and interference.

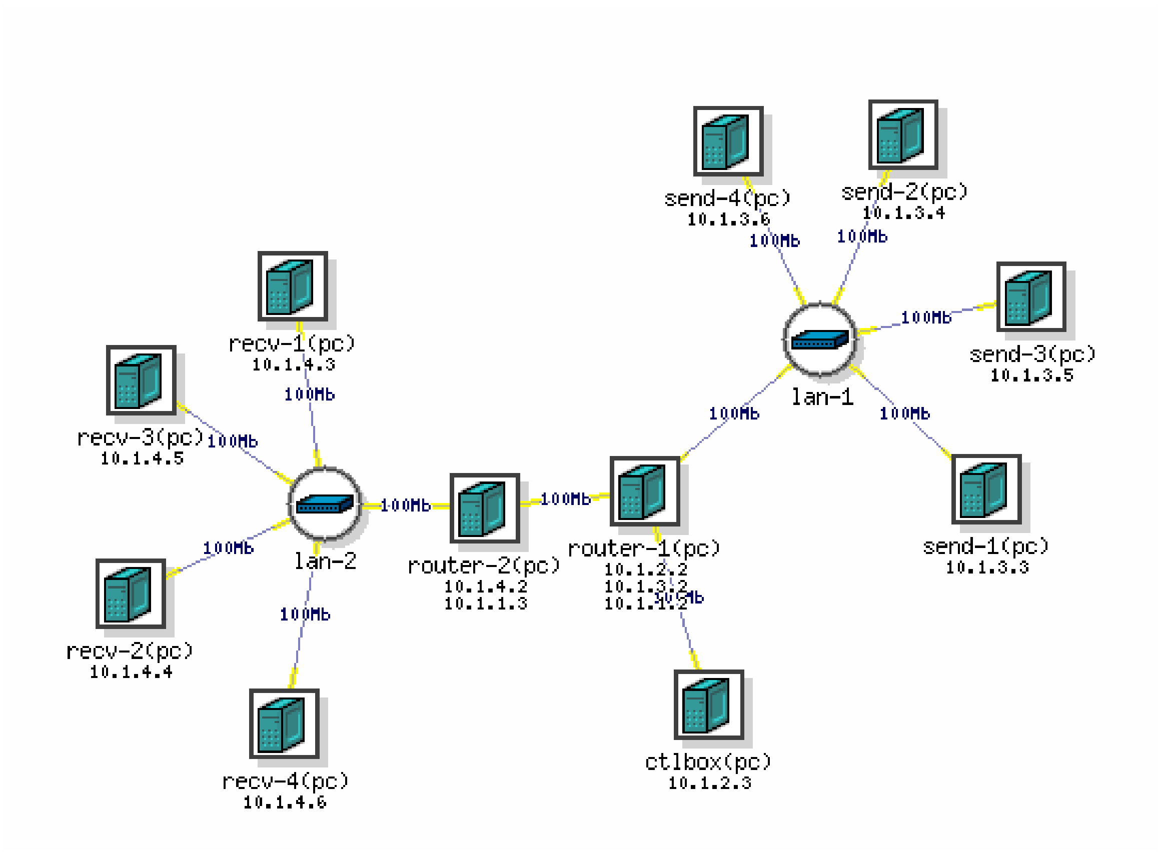

Fig. 1. A dumbbell network topology created in Emulab

```
set rate "fixed"
set send_channel_80211 6
set recv_channel_80211 11
set send_ssid_80211 "example1"
set recv_ssid_80211 "example2"
set config_file_wifi "config_wifi.sh"
```

If the wireless flag is set for any of the LANs, an Access Point (AP) node should be created for that LAN (e.g. $APsend$ or $APrecv$) and it will be connected to the next immediate router.

```
if {$send_mac_80211 == 1} {
  set APsend [$ns node]
  append lanstr($1) "$APsend "
}
if {$recv_mac_80211 == 1} {
  set APrecv [$ns node]
  append lanstr($router_no) "$APrecv "
}
```

Emulab provides 802.11 NICs on certain PC nodes that should be explicitly requested using the below syntax for all nodes in *WiFi-flagged* LANs and APs using loops. In this example *pc600wifi* nodes are being requested.

```
for {set i 1} {$i<=$max_peers} {incr i} {
   if {$send_mac_80211 == 1 } {
      $send($i) add-desire pc600wifi 1.0
   }
   if {$recv_mac_80211 == 1 } {
      $recv($i) add-desire pc600wifi 1.0
   }
}
if {$send_mac_80211 == 1} {
   $APsend add-desire pc600wifi 1.0
}
if {$recv_mac_80211 == 1} {
   $APrecv add-desire pc600wifi 1.0
}
```

802.11 LANs can be created in similar way to below. For this example, this is achieved for the senders' LAN by assigning the LAN protocol, the AP node and creating a link to the next router which in this case is $router(1)$. In similar way, if receivers' LAN is flagged as wireless, $APrecv$ should be linked to $router(\$router\_no)$.

```
if {$send_mac_80211 == 1 } {
  set lan(1) [$ns make-lan
  "$lanstr(1)" 54Mb 0ms]
  tb-set-lan-protocol $lan(1) "80211g"
  tb-set-lan-accesspoint $lan(1) $APsend
  set linkAPsend [$ns duplex-link $router(1)
  $APsend $link_bw 0ms DropTail]
}
...
```

Since Emulab does not support 802.11 on FreeBSD 9.0, an additional script is required for 802.11 setup (a.k.a *config_wifi.sh*). This shell script receives the 802.11 variables defined eariler as command-line arguments.

```
if {$send_mac_80211 == 1 } {
  tb-set-node-startcmd $APsend
  "$project_folder/$config_file_wifi
  $send_ssid_80211
  $send_80211_ipaddr_domain.2/$netmask
  $mode_80211 $send_channel_80211
  $rate"
}
...
```

We conventioanlly decide to assign one above the smallest available IP address in the subnet to the AP and the IP addresses of the other nodes will be assigned incrementally based on their index.

```
for {set i 1} {$i<=$max_peers} {incr i} {
  set ipaddr_offset [expr {$i + 2}]
  if {$send_mac_80211 == 1 } {
    tb-set-node-startcmd $send($i)
    "$project_folder/$config_file_wifi
    $send_ssid_80211
    $send_80211_ipaddr_domain
    .$ipaddr_offset/$netmask $mode_80211
    $send_channel_80211 $rate"
  }
  ...
}
```

The configuration script consists of two main functions namely $LoadWifiModules()$ and $SetupWifi()$. $LoadWifiModules()$ loads the $wlan$ and $atheros$-based driver modules into the kernel (e.g. if_ath_pci and if_ath). List of the required kernel modules for a specific 802.11 NIC can be edited in $wifi\_kernel\_modules$ string variable depending on the chipset family used in the hadrware. In our case, *pc600wifi* nodes are equipped with Atheros-based *AR5413 802.11abg* NICs. $SetupWifi()$ function receives the network parameters as arguments and creates and configures $wlan0$ interface using the $ifconfig$ command. The below is an example of this function's usage for the *APsend* access point which is being invoked in the script if the node's hostname contains "ap". The below command disables the rate adaptation mechanism (by default SampleRate in the current ath driver in FreeBSD 9.0) and sets the bit-rate to fixed 54 Mbps which is the maximum bit-rate supported in 802.11g. The details of $LoadWifiModules()$ and $SetupWifi()$ functions are presented in following Section V.

```
#SetupWifi {ath_interface}
#{wifi_interface} {ssid} {ipaddr/netmask}
#{mode: 11a or 11g} {channel}
#{sta_mode: ap or sta} {sleep_time in sec}
#{rate}

SetupWifi ath0 wlan0 example1
10.1.4.2/24 11g 6 ap 1 fixed
```

IV. A MULTI-HOP NETWORK WITH CASCADE ARCHITECTURE

In this section, we aim to establish a multi-hop network that can be used for scenarios where several bottleneck links can exists concurrently or the main bottleneck of the end-to-end path can be moved from a certain link (or queue) to another. For this purpose we are using a cascade architecture (a.k.a "parking-lot" topology in *TCP evaluation suite* [5]). Such topology consists of a path between sender(s) and receiver(s) which traverses from several links (bottleneck links). This can be achieved using several routers PCs connected in sequence. Moreover, the topology consists of an "access link" per each router. The access link can be connected to several nodes called "cascade nodes" hereafter. These cascade nodes are operating in the same LAN and connected together via a switch. Different types of traffic flows can be generated from the cascade nodes in the same LAN to other nodes in other LAN(s) sharing one or more bottleneck links on their path. By doing so, the router(s) in which the cascade flows are being shared with the other flows traversing on the main path, becomes a bottleneck. Having more cascade flows on different LANs creates multiple bottlenecks on the path. Figure 2 depicts an example of a cascade topology when $router\_no = 10$, $cascade\_node\_no = 1$ and $max\_peers = 1$.

To establish such topology the ns code in Section II should be extended. We call the cascade nodes as *cnodes* for the sake of brevity hereafter. The number of cnode LANs will be equal to the number of routers as each router will be associated to one LAN. it is possible to develop a more complex topology where routers can be connected to multiple LANs, however it requires the routers to have more Ethernet NICs than available in Emulab. The number of cascade nodes in each LAN can be defined as below

```
set cascade_node_no 1
```

A string variable for each cascade LAN should be defined to hold the name of each node in the LAN.

```
for {set i 1} {$i<=$router_no} {incr i} {
  set cascadelanstr($i) ""
}
```

Now each cascade node can be created and added to the defined string as below

```
for {set i 1} {$i<=$router_no} {incr i} {
  for {set j 1} {$j<= $cascade_node_no}
  {incr j} {
    set cnode($j-$i) [$ns node]
    append cascadelanstr($i) "$cnode($j-$i) "
  }
  append cascadelanstr($i) "$router($i) "
}
```

Similar to the senders and receivers in previous sections, the relevant OS image should also be requested for cnodes.

```
for {set i 1} {$i<=$router_no} {incr i} {
   for {set j 1} {$j<= $cascade_node_no}
   {incr j} {
      tb-set-node-os $cnode($j-$i) $os_image
   }
}
```

Each cascade LAN can be created as

```
for {set i 1} {$i<=$router_no} {incr i} {
  set cascadelan($i) [$ns make-lan
  "$cascadelanstr($i)" $link_bw 0ms]
}
```

The cascade nodes can be configured similarly to the senders by invoking the startup configuration script. For this reason, the shell script should be modified to configure the cnodes similar to senders. A sample start-up script is presented in Section V.

```
for {set i 1} {$i<=$router_no} {incr i} {
  for {set j 1} {$j<= $cascade_node_no}
  {incr j} {
    tb-set-node-startcmd $cnode($j-$i)
    "$project_folder/$config_file"
  }
}
```

Now, by swapping-in this code using the Emulab's web-interface [6], a cascade topology can be established.

V. Customizing start-up configuration script

In previous sections, we mentioned and briefly explained the additional configuration script for setting up the required configuration on FreeBSD nodes. We mentioned that Emulab's ns-like command does not support 802.11 capability on FreeBSD images and the configuration has to be done manually and therefore use of this script is compulsory for 802.11 functionality. On the other hand, the use of configuration script is optional on wired nodes depending on what packages are required for the experiments. For instance a TCP traffic generator such as *iperf* might be needed on senders, receivers and cnodes. Another example is an experimental delay-based TCP congestion control mechanism called CDG [7]. Since CDG isn't included in the main FreeBSD 9.0 kernel, it should be patched to the kernel source and installed. This operation is done on the sender nodes and cnodes only as it is only needed on the sender sides. These operations can be put in the relevant locations in the main structure of the script. An example is presented in below

```
#!/bin/sh

install_cdg() {
    cp cdg_patch /usr/src
    cd /usr/src
    patch -p1 < cdg_patch
    cd /sys/modules/cc/cc_cdg
    make;make install;;
}

host=`hostname`
case $host in
  *router*)
    kldload ipfw dummynet;;
  *send*)
    pkg_add -r iperf
    install_cdg;;
  *recv*)
    pkg_add -r iperf;;
  *ctl*)
    pkg_add -r screen;;
  *cnode*)
    pkg_add -r iperf
    install_cdg;;
esac
```

The above code comprises the basic structure of the configuration script for both wired and wireless configurations. However two other functions are mentioned in Section III that are part of the wireless script (config_wifi.sh): $LoadWifiModules()$ and $SetupWifi()$. $LoadWifiModules()$ loads the Atheros-based 802.11 driver modules in FreeBSD kernel as shown below.

```
wifi_kernel_modules="wlan if_ath if_wi
if_ath_pci wlan_wep wlan_ccmp wlan_tkip"

# usage: LoadWifiModules $wifi_kernel_modules
LoadWifiModules() {
   for wifi_module in $@
   do
```

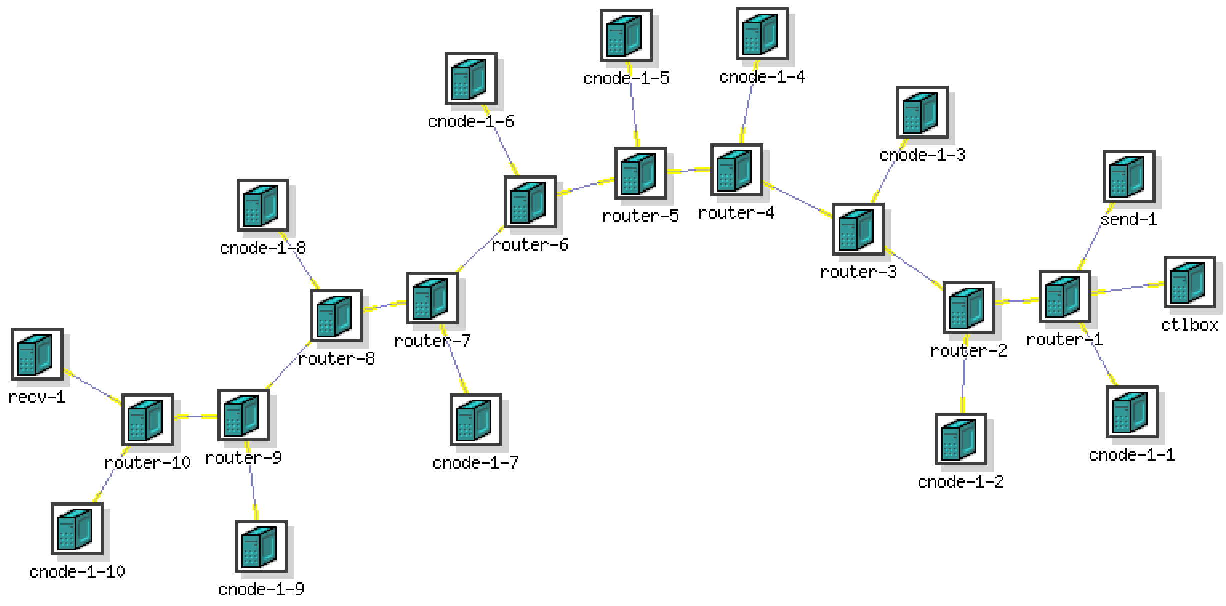

Fig. 2. A cascade (a.k.a parking-lot) network topology created in Emulab (router_no=10, cascade_node_no=1 and max_peers=1)

```
      kldunload -v ${wifi_module}
   done

   for wifi_module in $@
   do
      kldload -v ${wifi_module}
      sleep 1
   done
}
```

Once the 802.11-related modules are loaded, $SetupWifi()$ function should be called to configure the 802.11 setup on each 802.11 node.

```
# usage: SetupWifi(ath_interface ,
# wlan_interface ,ssid,
# ipaddr/netmask, {mode e.g. 11g},
# channel, wlanmode ,
# sleep_time, rate)
SetupWifi() {
   ifconfig ${2} destroy
   sleep ${8}
   ifconfig ${2} create wlandev ${1} \
   wlanmode ${7}
   sleep ${8}
   ifconfig ${2} down
   sleep ${8}
   ifconfig ${2} mode ${5}
   sleep ${8}
   ifconfig ${2} channel ${6}
   sleep ${8}
   ifconfig ${2} ssid ${3}
   sleep ${8}
   ifconfig ${2} ${4}
   sleep ${8}
   ifconfig ${2} up
   if [ ${9}=="fixed" ]
   then
      ifconfig ${2} ucastrate 54
   fi
}
```

The below example shows the usage of the above two functions in config_wifi.sh in addition to the routing setup.

```
net=`echo $ipaddr|awk 'BEGIN{FS="."}{print $1}'`
host=`hostname`
case $host in
  *router*)
    ...;;
  *send*)
    LoadWifiModules ${wifi_kernel_modules}
    SetupWifi ${ath_interface} \
    ${wifi_interface} ${ssid} ${ipaddr} \
    ${mode} ${channel} ${sta_mode} \
    ${sleep_time} ${rate}
    route add -net ${net}.0.0.0 APsend
    ...;;
  *recv*)
    LoadWifiModules ${wifi_kernel_modules}
    SetupWifi ${ath_interface} \
    ${wifi_interface} ${ssid} ${ipaddr} \
    ${mode} ${channel} ${sta_mode} \
    ${sleep_time} ${rate}
    route add -net ${net}.0.0.0 APrecv
    ...;;
  *ctl*)
    ...;;
  *cnode*)
```

```
    ...;;
esac
```

As shown in Section III the 802.11-related arguments used by $SetupWifi()$ function are declared in main ns file.

VI. Conclusions

In this paper, we explained in details how to deploy a multi-hop network testbed using FreeBSD nodes in Emulab. Establishing several network topologies were explained, starting with a basic multi-hop dumbbell topology and extending it to a network with 802.11 wireless nodes on the edges, and a more complex topology known as *cascade* (a.k.a parking-lot) were followed. These enable researchers to establish and customize a variety of network topologies providing a suitable platform for evaluating networking protocols and stacks using real-life measurements. A complete version of all codes discussed in this paper is provided in [2].

VII. Acknowledgements

This project has been made possible in part by a gift from the Cisco University Research Program Fund, a corporate advised fund of Silicon Valley Community Foundation.

References

[1] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar, "An integrated experimental environment for distributed systems and networks," in *Proc. of the Fifth Symposium on Operating Systems Design and Implementation*. Boston, MA: USENIX Association, Dec 2002, pp. 255–270.

[2] (2013) Establishing a multi-hop network testbed with 802.11 end-nodes using FreeBSD on Emulab (code package). [Online]. Available: http://caia.swin.edu.au/urp/newtcp/tools/CAIA-TR-130618A-code.tgz

[3] (2012) ns-2 Network Simulator. [Online]. Available: http://www.isi.edu/nsnam/ns

[4] (2013) Emulab tutorial. [Online]. Available: https://wiki.emulab.net/wiki/Tutorial

[5] (2012) TCP evaluation suite. [Online]. Available: http://tools.ietf.org/html/draft-irtf-tmrg-tests-02

[6] (2012) Emulab. [Online]. Available: http://www.emulab.net/

[7] D. A. Hayes and G. Armitage, "Revisiting TCP Congestion Control using Delay Gradients,," in *IFIP Networking 2011*, Valencia, Spain, 9-13 May 2011. [Online]. Available: http://dl.acm.org/citation.cfm?id=2008858