|  |
| --- |
| NSSA - RIT |
| FCT Experimental Code Base |
|  |
|  |
| **Alan Meekins** |
| **5/28/2010** |

|  |
| --- |
| Overview of all utilities and library routines developed to facilitate experimental deployments of the Floating Cloud Tiered Internet Architecture |

Testbed

# Overview

The local testbed is composed of twelve commodity PCs running Ubuntu linux version 8.04.1 with kernel version 2.6.24. Each computer has one network connection to the control network which is used to allow secure command line access to each machine and to serve as backbone for file transfers between machines for setting up experiments. The remaining interfaces are direct crossover connections between computers forming the topology shown below. Network experiments may use all network interfaces except for the control network to form connections between machines. All IP addresses on the testbed are statically assigned and should not be changed. Node1 is the only testbed machine which is outwardly accessible; all other machines must be accessed through this machine.

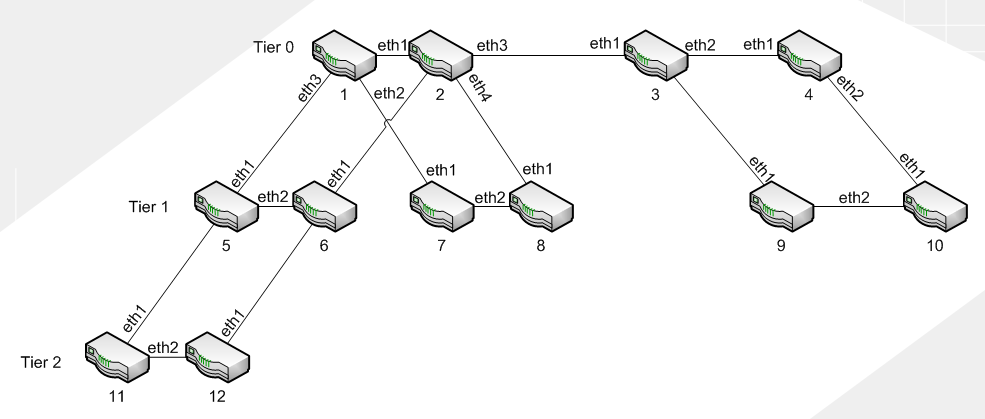


Figure : Testbed topology

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Host Name | Eth0(Control) | Eth1 | Eth2 | Eth3 | Eth4 |
| node1 | 192.168.27.245  Eth0:1 = 192.168.0.1 | 192.168.2.1 | 192.168.2.5 | 192.168.2.9 | - |
| node2 | 192.168.0.2 | 192.168.2.2 | 192.168.2.13 | 192.168.2.17 | 192.168.2.21 |
| node3 | 192.168.0.3 | 192.168.2.18 | 192.168.2.25 | 192.168.2.29 | - |
| node4 | 192.168.0.4 | 192.168.2.26 | 192.168.2.33 | - |  |
| node5 | 192.168.0.5 | 192.168.2.10 | 192.168.2.37 | 192.168.2.41 | - |
| node6 | 192.168.0.6 | 192.168.2.14 | 192.168.2.38 | 192.168.2.45 | - |
| node7 | 192.168.0.7 | 192.168.2.6 | 192.168.2. | - | - |
| node8 | 192.168.0.8 | 192.168.2.22 | 192.168.2. | - | - |
| node9 | 192.168.0.9 | 192.168.2. | 192.168.2. | - | - |
| node10 | 192.168.0.10 | 192.168.2. | 192.168.2. | - | - |
| node11 | 192.168.0.11 | 192.168.2.42 | 192.168.2. | - | - |
| node12 | 192.168.0.12 | 192.168.2. | 192.168.2.46 | - | - |

Figure : Testbed network interfaces

# Tools

There is a small set of bash shell scripts which can be used to easily copy files to all testbed machines or run any command on all machines. These scripts can be found in the utils/ folder.

Account Setup

# Local Testbed

Login to node1 as root with password sw1tch3d

ssh root@node1

Create account and giver user sudo

useradd <username>

useradd –G admin <username>

Set a password

password <username>

Make home folder on all nodes

runAll.sh mkdir /home/<username>

Copy passwords and group files to all testbed machines

copyFile.sh /etc/shadow /etc/shadow

copyFile.sh /etc/passwd /etc/passwd

copyFile.sh /etc/group /etc/group

Log out from out at node1

exit

Login as your new account, create ssh key for account on node1 and copy to all other nodes

ssh <username>@node1

ssh-keygen –t dsa #Press enter when prompted for a password

runAll.sh mkdir ~/.ssh

copyAll.sh ~/.ssh/id\_dsa.pub ~/.ssh/authorized\_keys2

From your local desktop create and ssh key and copy it to node1

ssh-keygen –t dsa #Press enter when prompted for a password

scp ~/.ssh/id\_dsa.pub node1:~/.ssh/authorized\_keys2

# Emulab

Ask Professor Shenoy for an account.

###### Configure SSH key

Use the “Edit SSH Keys” link from your EmuLab profile to upload your ssh keys to enable password-less login.

Compiling

The FCT code base was developed using the GNU C compiler version 4.4 along with GNU make version 3.81. Though it should compile on most earlier versions of GCC there may be unexpected compiler errors. Additionally this code base does make use of several linux kernel features that may not be available on very old versions of the linux kernel(ie. Older than 2.4) these features may not be available on all unix implementations so care should be taken to verify proper operation on these platforms.

All utilities support the –v option which will print the date the executable was compiled along with what debug level it was configured for.

The following commands can be used from within the src/ directory to produce executable files.

# Debug build

This command produces a debug build which prints out many detailed messages about what the routing software is doing. More debugging information can be turned on by changing the #define flags in the specific files which are of interest. For instance src/switched.cpp contains additional flags for turning on address, packet, and socket debug messages which will report what actions these classes are performing.

**make all debug**

# ****Test build****

Test builds print high level descriptions about what is going on in the routing code. This includes per packet timing information which are printed only after the packet has been handled as to not directly affect the processing time spent on a given packet.

**make all test**

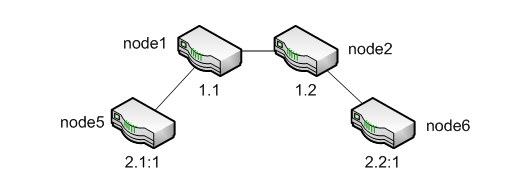
# Release build

Release builds only print critical error messages.

**make all release**

**Deployment**

# Local Testbed Example



Suppose we wanted to run a through put test using the local tested where node5 sends packets to node6 via nodes1 and node2 with the above topology. We will have to first build the source code on these machines then login to each of them and run the appropriate programs. Additionally it is important to take into account how these machines are physically connected. Since node5 and node6 actually share a direct connection and we want the packets to go through node1 and node2 we will have to make sure that the correct Ethernet interfaces are enabled.

###### Compile Code

To deploy the above topology first create a test build of the software:

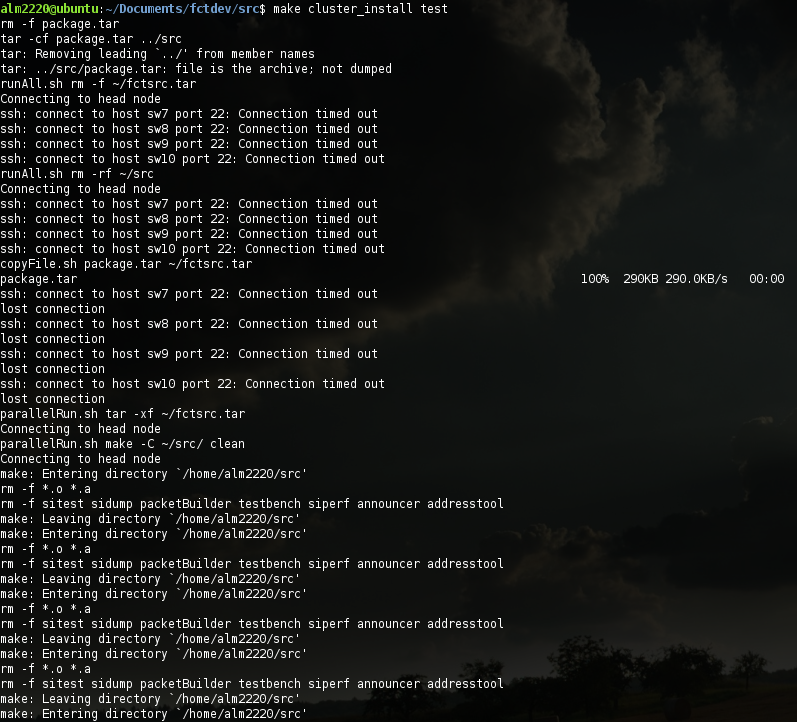


Figure : Deploying a build to the cluster

###### Login to nodes

Next login to each of the nodes in separate terminals, for example to login to node2 first ssh into node1 then ssh to node2. See below:

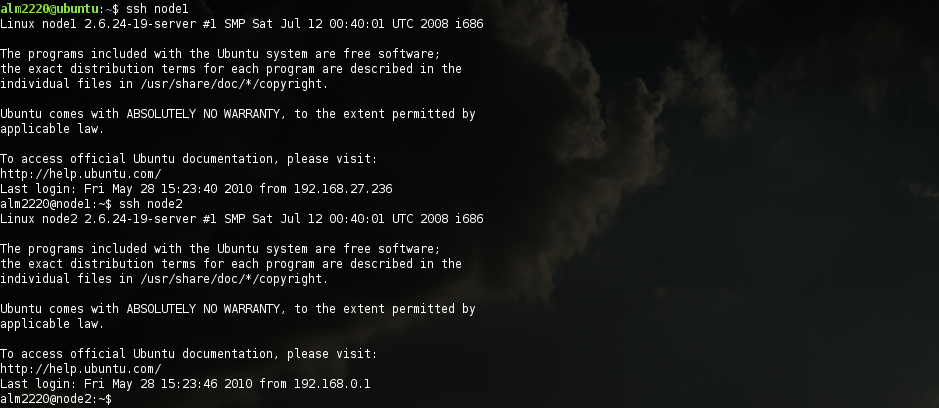


Figure : Login to node2 via node1

###### Configure nodes

Once you have an open shell on each node refer to the testbed topology to determine what ethnet interfaces should be enabled:

On node1 run:

sudo ./src/testbench -a 1.1 -i eth1 -i eth3

On node2 run:

sudo ./src/testbench -a 1.2 -i eth1 -i eth2

At this point you should see the below output on node1 and node2 respectively.

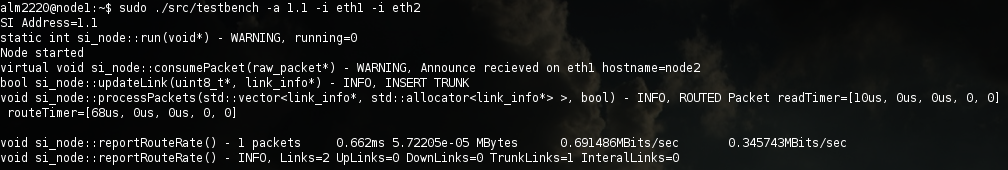


Figure : Node1 output

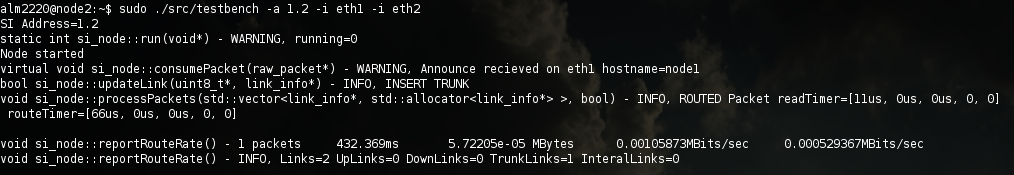


Figure : Node2 output

In the above screenshots, node1 is reporting that it is running and has consumed an announcement packet which came from node2. Node1 then indicates that the connection it has to node2 is a trunk link, followed by the amount of time spent reading the packet from the socket along with the amount of time spent parsing and reacting to the packet. After each loop of reading packets si\_node reports how many packets it has read , to total time spent in the run loop, the total amount of data contained in the read packets in megabytes. The next field is the instentanious through put followed by the average through put. The last line of the output displays the number of interfaces the node is configured to use followed by how many links are regarded as either an up, down, trunk, or internal link.

On node6 run:

sudo ./src/testbench -a 2.2:1 -i eth1

###### Execute performance tool

At this point all of the intermediate nodes have been setup and are now routing FCT packets, all we have left to do is to send packets from node5 using the below command:

sudo ./src/siperf -a 2.1:1 -i eth1 -c 2.2:2

This command instructs siperf to assume the address 2.1:1 and to send packets to address 2.2:2 as a client using interface eth1. By default siperf will transmit packets which are the largest allowed size at the highest rate allowed by the network interface. It will do this continuously for ten seconds or until interrupted by CTRL+C and will print a status report every two seconds. The output from this command is shown below.

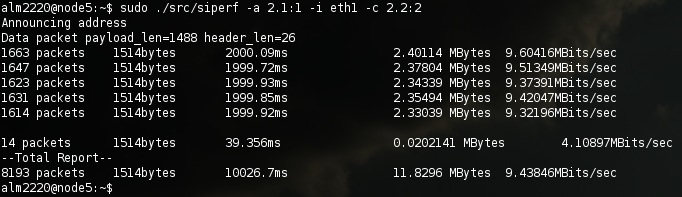


Figure : Using siperf from node5

Above we can see in the first report that siperf transmitted 1663 packets which were each 1514 bytes long(a full ethernet frame). It did this in 2000ms at a rate of 9.604 megabits per second sending a total of 2.4 megabytes. The last line of the output indicates all of the totals recorded.

###### Did it work?

So packets were send, but did they actually go where we want them to? We can determine this by examining the output from the testbench instance running on each of the other nodes. For instance the output on node6 begins to stream out at a much high rate as it prints a message about every packet it is routing. The output can be seen below.

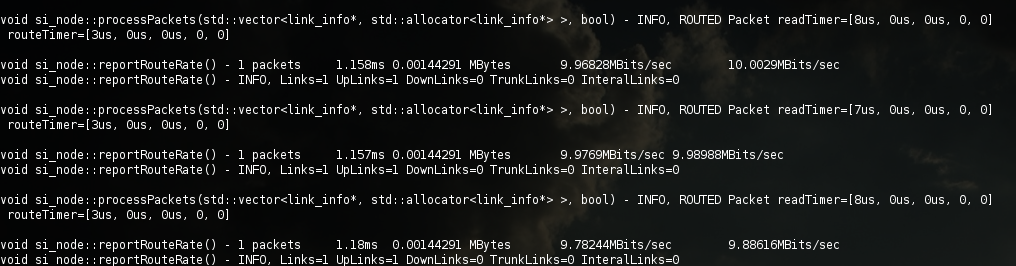


Figure : Node6 receives packets

###### What if does not work?

If we tried the above test again but this time ran the below command on node5 what happens?

sudo ./src/siperf –a 2.1:1 –i eth1 –c 3.2:2:2

Looking at the topology that we have configured we know that these packets should be delivered to node6 at which point they should be sent to a child of node6 with the address 3.2:2:2. However, there is no such node. To see what happens we look at the output from node6:

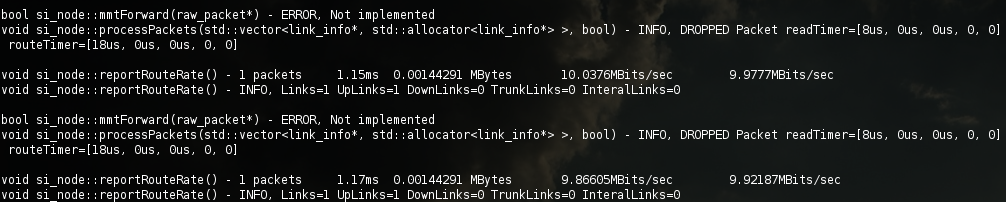


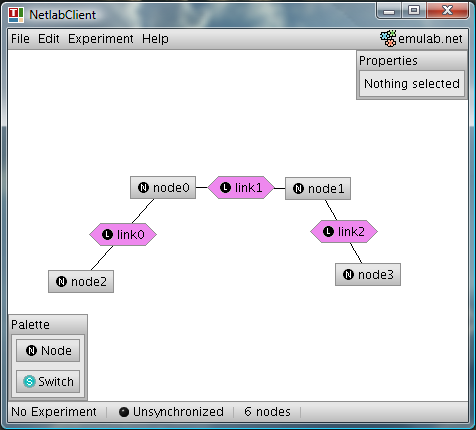
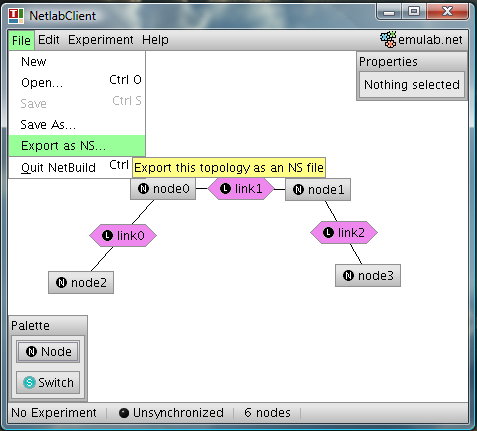
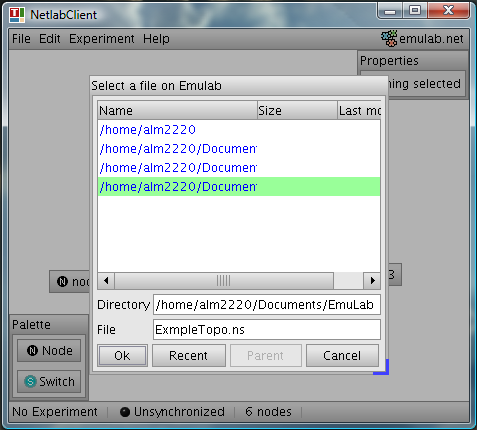
Figure : Node6 drops packets

Above we notice that node6 is seeing a lot of traffic which it is dropping. Additionally we see that the function mmtForward is being called. This because the packets sent from node5 are addressed to a node that is not part of our network. When these packets arrive at node6 it does not know how to deliver them so it first tries using mmt since there could be an mmt node with a downlink. Since mmt is not fully implemented it fails to deliver the packet resulting in node6 dropping the packet.

# EmuLab

To run the same test in emulab we must first create the same topology using the netlab software which can be found in the software directory. Run it with the following command:

java –jar netlab-client.jar



Save NS file

Export topology to NS

Construct test topology

Below is the generated NS file:

# Generated by NetlabClient

set ns [new Simulator]

source tb\_compat.tcl

# Nodes

set node0 [$ns node]

tb-set-hardware $node0 pc850

set node1 [$ns node]

tb-set-hardware $node1 pc850

set node2 [$ns node]

tb-set-hardware $node2 pc850

set node3 [$ns node]

tb-set-hardware $node3 pc850

# Links

set link0 [$ns duplex-link $node2 $node0 100000kb 0ms DropTail]

set link1 [$ns duplex-link $node0 $node1 100000kb 0ms DropTail]

set link2 [$ns duplex-link $node1 $node3 100000kb 0ms DropTail]

$ns rtproto Static

$ns run

# NetlabClient generated file ends here.

# Finished at: Fri May 28 13:32:11 PDT 2010

Now we must update this NS file to specify what operating system we want the computers to run:

tb-set-node-os $node0 UBUNTU70-STD

tb-set-node-os $node1 UBUNTU70-STD

tb-set-node-os $node2 UBUNTU70-STD

tb-set-node-os $node3 UBUNTU70-STD

**Note:** A copy of this file can be found in emulab\_config/ExampleTopo.ns

At this point we are ready to setup this experiment on EmuLab. Login to your EmuLab account at <http://www.emulab.net>

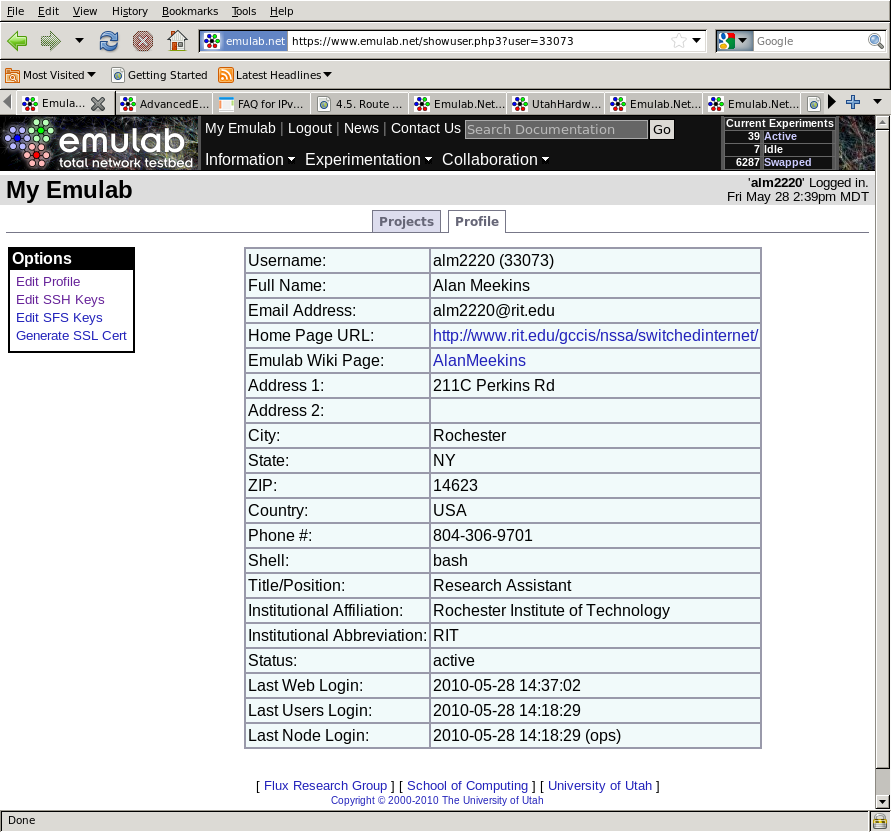


Figure : Login to EmuLab

From the Experimentation menu select “Begin an Experiment”, after doing so you will be taken to the below page. From this page you must name your experiment and upload the NS file which have just created:

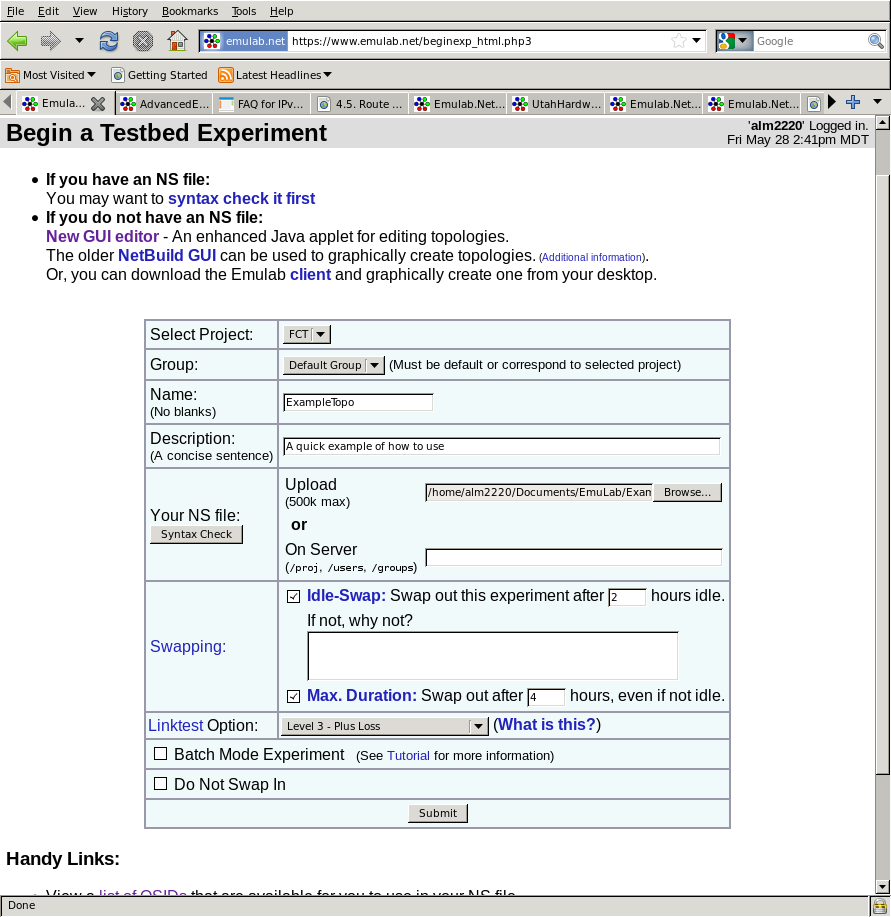


Figure : Configure experiment

Once you have configured the experiment click the “Submit” button. After a few minutes of configuring the network and rebooting the computers into the desired operating system you will receive an email telling you that your machines are ready for use.

###### Install code on EmuLab computers

We are now ready to copy our code over to the computers at EmuLab. Use the below command to copy the files. **Note:** Be sure to first edit the Makefile and change the EMULAB\_USER variable to your EmuLab user name.

make emulab\_install

###### Compile code

Login to one of the machines in the experiment and compile to source code.

ssh node1.ExampleTopo.FCT.emulab.net

tar –xf fctsrc.tar

cd src/

make clean

make all test

At this point the compiled code base is accessible by all of the machines in the experiment from your home folder.

###### Configure FCT on each node

Login to each node in a separate terminal, we must now figure out which interfaces are connected to which machines. This can be accomplished by running src/sidump on each of the network interfaces on all of the machines in the experiment then running src/testbench from one machine and observing what machines hear its announcements and on which interfaces. It is important to note that you must **never** usethecontrol interface in an experiment as packets may take unexpected paths. The interface which is considered the control interface is dependent on the type of computer used. Refer to the node summary page(<https://www.emulab.net/nodecontrol_list.php3>) to find out which interfaces to avoid using.

Once the interfaces have been identified you can use the same commands as from the local testbed example with the correct EmuLab interface names.

Development

# Version Control

The code base has been developed using the Bazaar version control system. Refer to docs/ bzr-quick-start-summary.jpg or <http://bazaar.canonical.com/en/> for more information about using Bazaar.

## Committing Changes

Committing changes saves a snapshot of the code you have changed locally. During heavy development commits should be preformed frequently as it gives you more choices on versions to revert back to should you find that you do not wish to save your changes. These changes can be reverted or pushed to the main code repository.

bzr commit

## Pushing Changes

When changes are pushed you are saving your local snapshots to the main version control server. This should be done at least daily when the code base is being changed. This ensures that your changes are saved to another computer in the event that the machine you develop on dies or is otherwise unusable.

bzr push sftp://alan@planet-backup/srv/Shared/siswitch

# Data Formats

## FCT Address

|  |  |  |  |
| --- | --- | --- | --- |
| Field | Type | Size | Description |
| tier | eth\_hdr | 4 bits | Tier level |
| chunkSize | uint16\_t | 2 bits | Address chunk size recorded as bits/2 minus one |
| chunkValue | uint16\_t | 4-16 bits |  |
| extraSize |  | 12 bits | Number of extra address bytes |
| extra | uint8\_t[] | ExtraSize bytes | Extra address bytes |

## FCT Packet

|  |  |  |  |
| --- | --- | --- | --- |
| Field | Type | Size | Description |
| eth\_header | eth\_hdr | 14bytes | Ethernet frame headers with proto set to 0x8D00 |
| pkt\_type | uint16\_t | 2 bytes | Packet type |
| data\_len | uint16\_t | 2 bytes | Length of the payload section |
| dst\_addr | FCT\_Address | - |  |
| src\_addr | FCT\_Address | - |  |
| payload | uint8\_t[] | Data\_Len |  |

#### Packet Types

There are three types of implemented packets, SI Command packets, MMT packets, and SI data packets. These packets have pkt\_type fields of 0x01, 0x02, and 0x05 respectively.

## Command Packets

Command packets have the pkt\_type field in the FCT header set to 0x0001. The first byte of payload data contains a four bit command type followed by a four bit command code.

### CMP\_EchoReply 0x00

|  |  |  |  |
| --- | --- | --- | --- |
| Field | Type | Size | Description |
| cmp\_type | uint8\_t | 1 | Indicates the type of command packet. Is set to 0x00 for EchoReply |
| seqnum | uint16\_t | 2 | Sequence number |

### CMP\_DestUnreachable 0x10

|  |  |  |  |
| --- | --- | --- | --- |
| Field | Type | Size | Description |
| cmp\_type | uint8\_t | 1 byte | Set to 0x20 for this packet |
| seqnum | uint16\_t | 2 bytes |  |
| dst\_addr | FCT Address | - | Destination which could not be reached |

**NOT IMPLEMENTED**. This packet may be sent in response to any packet which could not be forwarded

### CMP\_EchoRequest 0x20

|  |  |  |  |
| --- | --- | --- | --- |
| Field | Type | Size | Description |
| cmp\_type | uint8\_t | 1 byte | Set to 0x20 for this packet |
| seqnum | uint16\_t | 2 bytes | Initially set to 1 |

Any host which consumes an echo request packet must respond with a CMP\_EchoReply packet addressed to the source of the request. The reply must also contain the same sequence number as the request.

### CMP\_CloudAd 0x30

|  |  |  |  |
| --- | --- | --- | --- |
| Field | Type | Size | Description |
| cmp\_type | uint8\_t | 1 byte | Set to 0x30 for this packet |
| seqnum | uint16\_t | 2 bytes |  |
| hostname\* | string | Data\_Len-3 | Optional field for debugging purposes |

Any FCT host which receives a cloud advertisement should update all internal routing structures for later routing usage. Routing table entries may be removed if cloud announcements are not received within a regular interval. If the hostname field is included it can be used for debugging purposes to identify the sending FCT node.

### CMP\_Trace 0x40

|  |  |  |  |
| --- | --- | --- | --- |
| Field | Type | Size | Description |
| cmp\_type | uint8\_t | 1 byte | Set to 0x30 for this packet |
| seqnum | uint16\_t | 2 bytes | Incremented by each forwarding node, initially set to 1 |
| RealDst\_Addr | FCT\_Address | - | Actual destination to route packet to |

**NOT IMPLEMENTED**. This packet is treated like a CMP\_EchoRequest which is routed to RealDst\_Addr. The destination address in the FCT header must be set to link local. The receiving node must reply to the source address with a CMP\_EchoReply where the sequence number is set to the sequence number as found in the CMP\_Trace which it processed. The receiver then increments the sequence number of the CMP\_Trace packet and forwards it to the next FCT node in the path to the RealDst\_Addr.

### CMP\_ForcedCloudAd 0x50

|  |  |  |  |
| --- | --- | --- | --- |
| Field | Type | Size | Description |
| cmp\_type | uint8\_t | 1 byte | Set to 0x50 for this packet |
| seqnum | uint16\_t | 2 bytes | Set to 0x00 |
| hostname\* | char[] | Data\_Len-3 | Optional field for debugging purposes |

Announces an FCT cloud that may not be removed if the last announce timer times out.

### CMP\_Config

The purpose of this packet is to represent configurable parameters of an FCT node. The idea being that any of these parameters could be changed remotely and on the fly by sending a correctly formatted packet to a target machine. This way a test network could be completely reconfigured without requiring that the routing software running on the testbed be restarted.

**PARTIALLY IMPLEMENTED**

# Important Classes

## Mutex

Implements a convenient wrapper around a pthread mutex.

## Periodic

Represents a function which should be executed at a certain regular interval

## Pointer

This is a reference counted pointer(AKA: smart pointer) implementation copied from the CommonC++ library. When an instance of pointer goes out of scope it checks to see if it is the last one which references the allocated buffer and if it is the buffer is de-allocated. It is important to note that Pointer uses free() to release the memory it references. Therefore the buffers it points to should be created using malloc().

#include “pointer.h”

uint8\_t\* data=(uint8\_t\*)malloc(1500);

Pointer p<uint8\_t>(data);

exit(1);

## Timer

Records the real, user, and system times for the measured period. Additionally the number of voluntary and involuntary context switches is also counted. Pausing and resuming of timers is supported.

#include “timer.h”

Timer t();

t.start();

sleep(5);

if(t.isRunning()){

t.stop();

}

t.print();

t.reset();

## si\_address

#include “switched.h”

si\_address addr=si\_address(“3.1:2:3”);

## si\_packet

This class is designed to be very versatile and easy to use. It makes heavy usage of STL containers and is intended for use in code which is not time critical.

#include “switched.h”

si\_address srcAddr(“1.2”);

si\_address destAddr(“2.2:3”);

string data=”Hello World”;

si\_packet packet(srcAddr, destAddr, data.c\_str(), data.size());

## si\_socket

#include “switched.h”

string ifname=”lo”;

si\_socket sock(ifname);

if(sock.isOpen()){

si\_packet rxPacket;

int rxStatus=sock.recv(rxPacket);

int txStatus=sock.send(rxPacket);

}

sock.close();

## raw\_address

This class contains routines for inplace manipulation of FCT addresses. It is designed to be fast and memory efficient and should only be used in places were execution speed is critical.

## raw\_packet

This class is designed to make use of the raw\_address class and support such operations as address field removal and addition without requiring that packets be copied or reallocated. Much like raw\_address it is designed for speed and is cumbersome to use and as such should only be employed in performance critical code.

## si\_node

The si\_node class leverages nearly all of the classes in the code base to implement the FCT routing algorithm. Below is an example of how to use the si\_node class. When myNode->start() is executed a new thread is created and the FCT router begins handling any packets that are read on the configured interfaces. This thread is configured to raise its priority such that it will receive near real-time service from the operating system. The only time this thread yields the processor is when it is waiting for packets to arrive. Due to this scheduling setting it may be difficult for other processes running on the same computer to get processor time when there is high network load. The routing thread can be stopped by calling the stop function. Depending upon the arguments passed, the stop function may block indefinitely or for a maximum specified time period while it waits for the routing thread to complete its clean up procedures. When an si\_node goes out of scope or is deleted the deconstructor attempts to stop the routing thread if it is still running using the stop funtion. If the stop function fails the deconstructor then sends the routing thread a SIGKILL signal and frees the thread’s allocated stack space.

#include “si\_node.h”

si\_node\* myNode=new si\_node(si\_address(“3.1:2:3”));

myNode->addAllInterfaces();

if(myNode->start()){

while(canrun && myNode->isRunning()){ sleep(5); }

if(myNode->isRunning()){

myNode->stop();

}

}

delete myNode;

The routing algorithm implemented by the si\_node class is the FCT architecture. A flowchart of the routing process can be seen below.

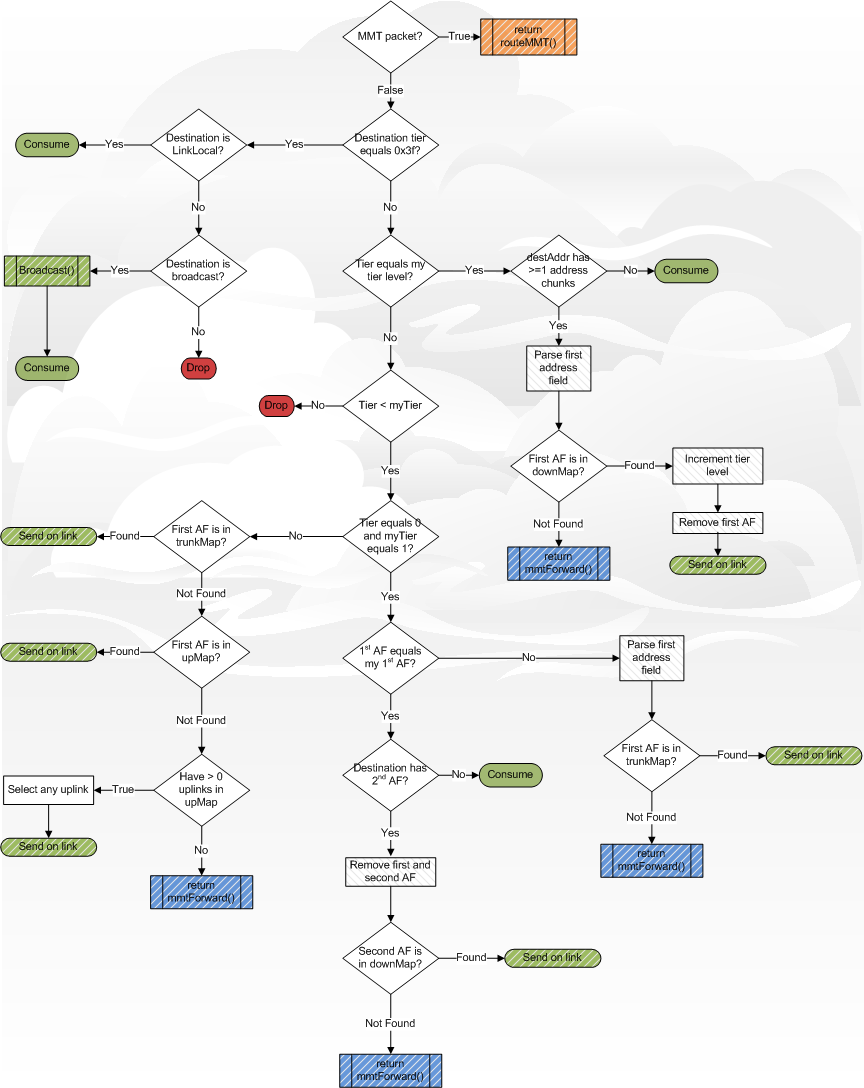


Figure : FCT Routing algorithm

# Data Structures

Utilities

# Address Tool

Used to display the binary format of a given SI/FCT address, the below image is the output when passed one address. The tool reports the number of address chunks and how many bytes is needed to store the address. The third line is the address in hexadecimal followed by the address in binary. The binary dump is grouped by bytes, below which is the bit number.

alm2220@ubuntu:ftcdev/src$ ./addresstool 3.3:4:2

Read address: 3.3:4:2

Has 3 chunks and 5 bytes

0x0d, 0x35, 0x12, 0x00, 0x00,

00001101 00110101 00010010 00000000 00000000

76543210 76543210 76543210 76543210 765432

Addresstool can accept two address as seen in the below screenshot. In this case the hex and binary representations of both addresses are displayed along with the minimized address which would be used to address a packet being sent from the first address to a node with the second address.

alm2220@ubuntu:ftcdev/src$ ./addresstool 2.3:4: 3.3:4:2

Read address: 2.3:4:4

Has 3 chunks and 5 bytes

0x09, 0x35, 0x14, 0x00, 0x00,

00001001 00110101 00010100 00000000 00000000

76543210 76543210 76543210 76543210 765432

Read address2: 3.3:4:2

Has 3 chunks and 5 bytes

0x0d, 0x35, 0x12, 0x00, 0x00,

00001101 00110101 00010010 00000000 00000000

76543210 76543210 76543210 76543210 765432

Minimum Address From [2.3(4):4(4):4(4)] To [3.3(4):4(4):2(4)]

2.2(4)

Has 1 chunks and 4 bytes

0x09, 0x21, 0x12, 0x00,

00001001 00100001 00010010 00000000

76543210 76543210 76543210 76

# Announcer

The announcer tool is used to construct and transmit SI/FCT announcement packets. The tool transmits a special type of announcement, CMP\_ForcedCloudAd. Any SI/FCT node which receives an announcement of this type will mark it as non-removable so that table entries for that node will not be removed when they timeout. The tool requires at least one interface to operate on, specified with the –I <ifname> option. To transmit on multiple interfaces give the –i option as many times as needed. To automatically add all available interfaces give the –I flag. **Note:** The option to add all interfaces will add all interfaces, including the control interface.

alm2220@ubuntu:ftcdev/src$ sudo ./announcer

Usage: ./announcer [options]

-i [interface] Interface to broadcast on

-I Broadcast on all interfaces

-a [address] Base source address \*REQUIRED\*

-D [delay] Time to wait between sending packets, in microseconds. Default is 500us

-n [hostname] Hostname used in announcements

-o [offset] Address offset

-d [downlinks] Number of downlinks to announce, default is 2000

-t [trunklinks] Number of trunklinks to announce, default is 2000

-u [uplinks] Number of uplinks to announce, default is 2000

-V Verbose output

-v Version information

# Packet Builder

The packet builder allows you to construct and transmit a packet by specifying the values for important fields on the command line.

alm2220@ubuntu:ftcdev/src$ ./packetBuilder

Warning! This program must be run as root! Don't be surprised if bad things happen.

Usage: ./packetBuilder <options>

-t <type> Packet type number

-s <source> Source address

-d <destination> Destination address

-i <interface> Interface name, defaults to lo

-h Display this help message

-v Version information

# SiDump

Sidump is a utility which prints information about any packets that it receives. The only option that sidump requires is an interface name to listen for packets on.

alm2220@ubuntu:ftcdev/src$ sudo ./sidump lo

int si\_socket::open(std::string) - WARNING, Interface is LOOPBACK

Recv packet: [Destination = 0.2; Source = 1.1; Data = 11 bytes] Tue May 25 08:35:25 2010

Recv packet: [Destination = 0.2; Source = 1.1; Data = 11 bytes] Tue May 25 08:35:26 2010

Recv packet: [Destination = 0.2; Source = 1.1; Data = 11 bytes] Tue May 25 08:35:27 2010

# SiPerf

Si perf was inspired by iperf and is a loose clone. Similar to iperf, siperf has two modes of operation client and server. Both modes require that you provide the si address of the host as well as at least one network interface. The address can be assigned directly using the –a option followed by the si address or the address can be auto assigned using the –x option. The automatic address assignment flag requires an argument of “child”, “parent”, “trunk”, or “internal” which causes the computer to announce itself as that type of node to the first si node that it receives an announce packet from.

|  |  |
| --- | --- |
| Address Assignment Mode | Behavior |
| Child | Copies the first received announcement, increments the source address tier value and adds an address field to the end of the source address. |
| Parent | Copies the first received announcement, decrements the source address tier value and removes the last address field. |
| Trunk | Copies the first received announcement, increments the last address field by one. |
| Internal | Copies the first received announcement, makes no changes to the source address. |

## Server Mode

In server mode an address must be supplied using either the –a flag or –x along with any number of interfaces to listen on. Siperf then listens for a specially formatted si packet which contains a four byte sequence number at the beginning of the packet’s payload. When a packet with a sequence of one is received siperf then starts a new traffic report and sets the expected sequence number to two. As more packets are received from the same host the number of lost packets is counted. When the transmitting client sends a special termination sequence siperf reports the average rate that packets were received and the number of lost and received packets. This report is also printed if the user terminates the program before the last packet is received.

## Client Mode

In client mode siperf can run one of two tests, round trip time(ping) or a through put test. The client requires that one interface be specified using –i and that an address be assigned using –a. To indicate what node we wish to communicate with –c is used followed by an SI address.

### Throughput

When run in client mode throughput testing is the default behavior of siperf. With no additional arguments siperf will transmit packets which are the maximum size allowed by Ethernet frames(1514 bytes) to the host specified by the –c flag at the highest rate allowed by the selected interface. Every two seconds a report is printed displaying the number of packets sent, the size of the packet, the amount of time it spent transmitting, total amount of data sent in megabytes and lastly the transmission rate in megabits per second. Upon exit totals of the same metrics are displayed. By default this test runs for ten seconds

###### Parameters

Several key parameters may be changed through command line flags, packet size, transmission rate, report frequency and test duration. Packet payload size is set using the the –b option followed by the number of bytes. Packet payload size must be at least 4 bytes, if not specified siperf will make the payload large enough to exactly fill the ethernet frame. The test duration can be set using the –d option followed by a time in milliseconds, defaults to 10,000ms. The interval between transmission rate reports can be set using the –t flag followed by time in milliseconds. This options defaults to 2000ms.

The target transmission rate can be set using the –r flag with the number of packets to transmit per second as the argument. This rate control mechanism works to varying levels of accuracy which depends the overall system load, network card, and system architecture. In most cases it undershoots the target rate by half. It is advisable to experiment with the target rate value until the desired rate is reported.

### Round Trip

Round trip measurements can be performed by passing siperf the –p command line flag. Siperf transmits echo request command packets, CMP\_EchoRequest, with the sequence number initially set to 1 and the destination and source appropriately filled in. It then waits for an echo reply from the destination host with the same sequence number as the last sent echo request. If a valid echo reply is received the sequence number is then incremented by one and another echo request is transmitted. The time it takes to receive an echo reply is recorded and averaged. This loop continues until the test duration is reached. Upon exit a report of the minimum, maximum, and average round trip times is printed.

# SiTest

Sitest makes use of many important library functions such as packet construction, address minimization, and socket operations. After any modifications to the library sitest should still compile and run without crashing or exiting with an error message.

The last test that sitest runs measures the rate at which 1024000 packets can be processed. This does not actually transmit any packets but can serve as a quick performance measure to get an idea of the lower bound on the processor and memory access times.

# Testbench

The testbench utility runs an instance of the FCT routing algorithm in a separate high priority thread. The address of the FCT node along with the MMT uid may be specified as arguments. Any number of interfaces can be used for communication using the either the –i option to specifically list the desired interfaces or –I to select all available interfaces. Interfaces may be excluded using the –x option.

Future Work

# Full MMT Support

The implementation of MMT in the si\_node class is only a skeleton currently and needs to be completed. Major data structures have been written but are not currently updated when internal links are detected. The FCT routing algorithm calls all of the correct MMT functions currently but the MMT functions are not complete. Additionally, there is currently no support at all for MMT packet de-capsulation.

## bool mmtRoute(raw\_packet\* packet)

The mmtRoute function is for routing of packets which are already in the MMT packet format and do not require encapsulation. Below is pseudo code for how this function could work.

bool routeMMT(raw\_packet\* packet){

vid=packet.parseVid();

if(myVids.contains(vid)){

routePacket(packet.decapsulate());

}

link\_info\* link=nearestLink(vid);

if(link!=NULL){

return link->socket->send(packet);

}

return false;

}

## bool mmtForward(raw\_packet\* packet)

The mmtForward function should take care of encapsulating an FCT packet as an MMT packet and passing it onto the routeMMT function. Below is pseudo code of how this should work.

bool mmtForward(raw\_packet\* packet){

uid=lookupUid(packet.dstAddr);

vid=getShortestVid(uid);

packet.mmtEncapsulate(vid);

return routeMMT(packet);

}

# Complete Address Extra Support

The extra field of FCT addresses is fully supported by the si\_packet class; however it is not fully supported by the raw\_address class and may not be correctly preserved after modifications to address contents. Extra fields will need to be fully supported if there is to be any testing of IP tunneling.

# Broadcast Packet Routing

Currently forwarding broadcasting packets will cause flood storms, this must be corrected if any sort of performance analysis is to be done on FCT’s broadcast behavior.

# SiPerf Unified Output

Siperf, when run in server mode, currently only prints the number of dropped packets detected. It should at least report the current throughput seen by the node in the same format as client mode. Additionally, the reporting of dropped packets could be much more robust.

# Easier EmuLab experiment setup

It would be helpful to create a tool which could quickly identify what Ethernet interfaces are connected to which machines and which interface is a control interface. It’d be even better if it were graphical.

# Remote Configuration

As we move to larger scale testing there should be a system for re-configuring a running FCT node on the fly. Configurations such as changing the node’s address, available links, and MMT uid should be supported.

# Dying Testbed Computers

Node 7 should not be left on unattended as it’s power supply fan is dead!

Known Issues

# Trunk links

Currently available trunk links are not used if a packet has a tier level lower than the routing node’s tier.