Rutgers Computer Science 352 VLAN Project
Part 1: Pairwise Tunnel

Notes/Announcements:

1. Some example code to `connect()` with DNS names and open a busy socket are posted at the end of the assignment.
2. A binary client has been posted. See the resources/project resources section of the sakai site.
3. Students must submit a write-up, in text or HTML format, on how their proxy works. See the documentation section below.
4. Students must also submit, in addition to their source code, a Makefile that builds the proxy using the command 'make all' .
5. See the what to hand in section below for what files need to be handed in.
6. The maximum frame size has been expanded from 1500 to 2048 bytes to include all the layer-2 headers.
7. Project 1 due date (from the class page): Wednesday, October 10th, 11:59PM.

In this project, you will create a Virtual Local Area Network (LAN). This LAN will be virtualized over the Internet. Your project will thus create a system that runs Layer 2 (datalink) over Layer 3 (network).

For part 1, you will only route between 2 nodes, thus creating a tunnel. The remainder of this document describes the theory of operation for the project, the programming resources you will have available, the expected method of development of the project, some pseudo-code to help you get started, the constrained on how your project is invoked and the wire protocol it must conform to, and finally a set of links to resources to help you get started.

Theory of Operation:

The strategy to construct the VLAN will be for your team to create a proxy program that runs on each machine. The proxy will use 2 Ethernet devices to implement the VLAN. One ethernet device, called eth0, will be connected to the Internet. The second device is a virtual device called a tap (for network tap). The tap operates on layer 2 packets and allows your proxy to send and receive packets on the local machine. Figure 1 shows the architecture used for the assignment.

The proxy captures packets send by user programs on the local machine. Your proxy will then encapsulate and route the captured packet over a standard TCP socket to the correct destination peer proxy. The receiving proxy will then in turn inject the packet into the local tap, where the local operating system will deliver it to the correct process. This process is shown in figure 1.

The collection of proxies will be organized as a peer-to-peer to realize the Each peer will have a unique ID, which will be its local MAC address. We will assume that each local MAC address is unique for this project.

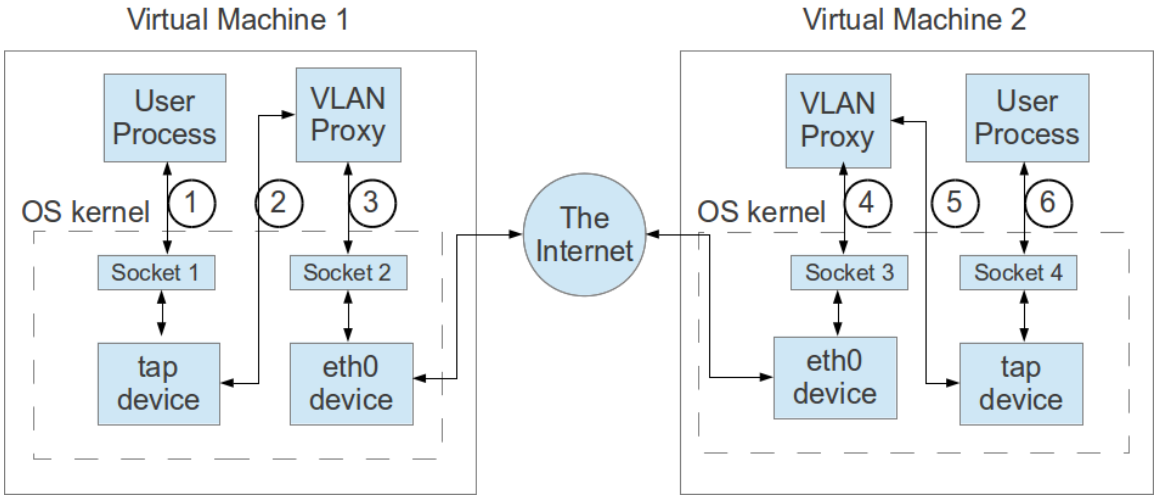

Illustration 1: Figure 1: Architecture of VLAN

For part 1 of the project, you will only need 2 proxies to create a point-to-point tunnel.

The VLAN system should perform these steps, as described in Figure 1:

1. A user process on the virtual machine sends a packet that is sent to the virtual Ethernet (tap0) by the operating system.
2. Your VLAN proxy will read the packet using system call reads from the tap device.
3. Your proxy will send the packet to a peer proxy running on another machine over a regular TCP socket.
4. The remote proxy will read the packet from a standard TCP socket.
5. The remote proxy will strip off the encapsulating headers and inject the packet into the local tap device.
6. The local operating system will deliver the packet to the remote user process's socket.

Programming Resources and Set-Up

For part 1, each team will be assigned a suite of 2 virtual machines. Later parts of the project will include more VMs. The virtual machines are accessible via ssh to a hostname in the form "cs352-X.rutgers.edu" and "cs352-Y.rutgers.edu", were X and Y are numbers assigned to each group. See the Sakai site for which number your group has been assigned. To log in, you would execute this command '`ssh -X -C username@cs352-X.rutgers.edu`'. The -X flag is necessary to get windowing programs such as eclipse to display on your local machine, assuming you have a local X-windows server running on the machine your are logging in from. The -C flag compresses the windowing data to help speed things up.

Expected Project Development Cycle.

Below is an example project development cycle. This should give you an idea of how to get started. See the further resources and suggestions section for more hints on how to get started.

Remote mount your ilab home directories from the virtual machines using sshfs. The username is your login username and the machine is an ilab machine. For example:

```
mkdir ilab
sshfs<username>@<machine>.rutgers.edu:/ilab/home/<username> ilab
```

Create tap devices on each machine by executing this command:

```
sudo openvpn --mktun --dev tun2
```

The the kernel the interface is active/up:

```
sudo ip link set tun2 up
```

Add an IP to the interface and tell the kernel to route packets matching the first 24 bits of the address to it:

```
sudo ip addr add 192.168.2.2/24 dev tun2
```

Verify the interface is up and packets will be routed to it by looking at the routing table:

```
sudo route -n
```

The output should look similar to this:

```
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
128.6.5.128     0.0.0.0         255.255.255.128 U     0      0        0 eth0
192.168.2.0     0.0.0.0         255.255.255.0   U     0      0        0 tun2
169.254.0.0     0.0.0.0         255.255.0.0     U     0      0        0 eth0
0.0.0.0         128.6.5.129     0.0.0.0         UG    0      0        0 eth0
```

Develop your proxy.

You should compile the proxy on the virtual machine.

You can use eclipse or command line tools. Eclipse is in:

```
/usr/local/eclipse/eclipse
```

Use version control, such as cvs, , svn, or git to track changes

Compile your proxy on one of the virtual machine:

Start a copy of the proxy on machine X.
Start a copy of the proxy on machine Y.

Send a test packet from the command line; e.g. on slave1, ping slave 2:

```
ping 192.168.18.2
```

See if the proxy receives the test packet
See if the remote proxy receives the packet over TCP (e.g. print it out)

Example Program Structure

An example program might be structured as follows:

```
The main thread reads the command line arguments
Create a TCP socket on the port as defined in the command line.
create a thread to handle virtual tap device
create a thread to handle incoming packets on the TCP socket
Run the thread for the TCP socket, which:
      If it is the server mode proxy, listen on the port, else
            connect to the other proxy
      Loop forever:
            listen for packets from the TCP socket
            when a packet arrives, de-encapsulate the packet
            send it to the local tap device

Run the thread for the  tap device, which:
      Loop forever:
            Listen for a packet from the tap device
            Encapsulate the packet in the proper format
            Send the packet to over the TCP socket
```

Standard Command Line Invocation

In order to test and connect your proxies to the reference implementation (i.e. the TA's implementation) you must build your proxy in a manner that allows it to executed from the command line in a standard way. proxies, you must name the resulting binary 'cs352proxy' and accept the following arguments.

Your proxy will also behave differently depending on the command line arguments. A 2-argument proxy will passively listen for the 2nd proxy to connect to it, while the 3 argument proxy will actively connect to the peer proxy addressed in the arguments.

For C-based proxies:

For the 1st proxy (e.g. on machine X)

```
cs352proxy <port> <local interface>
```

For the 2nd proxy (e.g. on machine Y)

```
cs352proxy <remote host> <remote port> <local interface>
```

The arguments are:

- **local port:** a string that is a number from 1024-65535. This is the TCP port the proxy will accept connections on.

- **local interface:** A string that defines the local tap device, e.g. tun2
- **remote host:** A string that defines which peer proxy to connect to. It can be a DNS hostname or dotted decimal notation.
- **remote port:** This is the remote TCP port the proxy should connect to.

TCP Message Format

In order to test your proxy against the reference implementation, your proxy messages MUST be encapsulated in the following format. All integers must be in big-endian format:

Type (16 bits): Length (16 bits): Data

The fields for part 1 are:

- **Type:** This 16-bit integer in the form 0xABCD (hexadecimal). It indicates a VLAN packet. Parts 2 and 3 will expand the type field to include more types.
- **Length:** The length of the remainder of the data payload, as an unsigned 16 bit integer.
- **Data:** The packet as received from tap device.

Documentation (the README) file

In addition to the source code, you must provide some documentation for your project. Your documentation should include:

1. Who is in the group.
2. What each group member did, what did he or she contribute to the project?
3. Describe the general architecture of your program, what each function does and how it does it.
4. What difficulties you had, challenges you encountered, how you solved them, and which are left open?

What to Hand In:

You will need to hand in the following through the sakai. You must package up all your files into a single file using ZIP or TAR and uploaded your package through the Sakai assignments page. See here how to create a ZIP file and here for TAR files:

You ZIP or TAR archive (package) must include the following files:

1. All the source code files for your project.
2. Any Eclipse project files.
3. A Makefile where the top-level target (i.e. “all”) makes the whole program
4. A README file in text or html format describing your project.

Grading:

You will be graded along 2 dimension for this project: functionality and readability. Functionality is straightforward to test: can your proxy pair sustain a bi-direction traffic stream for > 1 minute? Readability is a more subjective measure. The main idea is how well can someone not on your team understand your cc? How easy or difficult would it be for them to make a modification? In order to quantify readability, the professor and TA's will give each project a score. Your team will also be responsible for giving another team a score. We will post further instructions how to get the project you will review and how to upload your scores for the project.

1. Functionality of the project: 60%

2. Instructors' score for the code: 20%

3. 2nd team's score for the code: 20%

4. Failure to submit a score for another team: -50%

Code scoring questions:

On a scale from 1-10, where 10 = best and 1=worst, rate the project on the following:

1. I was able to understand what each function did in a timely manner.
2. The files, variables and functions names made it obvious what each one did.
3. It was clear what parts of the program each variable was used in.
4. The scope of the variable appropriate.
5. It is clear what the external dependencies are and how they are used.
6. The comments were helpful but not distracting.

Strategies to make code readable (from: http://stackoverflow.com/questions/550861/improving-code-readability)

1. Standards of indentation and formatting are followed, so that the code and its structure are clearly visible.
2. Variables are named meaningfully, so that they communicate intent.
3. Comments, which are present only where needed, are concise and adhere to standard formats.
4. Guard clauses are used instead of nested if statements.
5. Facilities of the language are used skillfully, leveraging iteration and recursion rather than copy and paste coding.
6. Functions are short and to the point, *and do one thing*.
7. Indirection is minimized as much as possible, while still maintaining flexibility.

Further Resources, Links and suggestions.

You will have to further investigate these links to complete the project. One suggestion to start is to first write a small program with the jpcap library, and see if you can send/receive packets

over a virtual Ethernet. Use tshark or wireshark to verify your small test program works. If you are using C, begin with the pinger.c example below and then try to build up the project from there. Next, experiment with TCP sockets, and write a small client/server program.

Header files you should include in your proxy:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <stdarg.h>
#include <errno.h>
#include <fcntl.h>
#include <pthread.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <net/if.h>
#include <arpa/inet.h>
#include <linux/if_tun.h>
#include <sys/socket.h>
#include <sys/ioctl.h>
#include <sys/select.h>
#include <sys/time.h>
```

Below is a function to open the local tap device:

```
/**************************************************
 * allocate_tunnel:
 * open a tun or tap device and returns the file
 * descriptor to read/write back to the caller
 *************************************/
int allocate_tunnel(char *dev, int flags) {

  int fd, error;
  struct ifreq ifr;
  char *device_name = "/dev/net/tun";

  if( (fd = open(device_name , O_RDWR)) < 0 ) {
    perror("error opening /dev/net/tun");
    return fd;
  }

  memset(&ifr, 0, sizeof(ifr));

  ifr.ifr_flags = flags;

  if (*dev) {
    strncpy(ifr.ifr_name, dev, IFNAMSIZ);
  }
```

```
  if( (error = ioctl(fd, TUNSETIFF, (void *)&ifr)) < 0 ) {
    perror("ioctl on tap failed");
    close(fd);
    return error;
  }

  strcpy(dev, ifr.ifr_name);
  return fd;
}
```

How to use the `allocate_tunnel()` function:

```
  char *if_name = "tap0";

   if ( (tap_fd = allocate_tunnel(if_name, IFF_TAP | IFF_NO_PI))
< 0 ) {
     perror("Opening tap interface failed! \n");
     exit(1);
   }
/* now you can read/write on tap_fd */
```

Example code that allows a socket to be reused if it was not properly closed:

```
int optval = 1;
/* avoid EADDRINUSE error on bind() */

      if(setsockopt(sock_fd, SOL_SOCKET, SO_REUSEADDR,
                  (char *)&optval, sizeof(optval)) < 0) {
      perror("setsockopt()");
            exit(-1);
      }
```

Example code for connecting to either a name or an IP address:

```
struct sockaddr_in to;     /* remote internet address */
struct hostent *hp;        /* remote host info from gethostbyname() */
unsigned short rport;
char *hostname = "www.google.com";

/* example of connecting to server using a DNS name or
 * dotted decimal IP address */

  to.sin_family = AF_INET;
  to.sin_port = htons(rport);
 /* If internet "a.d.c.d" address is specified, use inet_addr()
  * to convert it into real address.  If host name is specified,
  * use gethostbyname() to resolve its address */
  to.sin_addr.s_addr = inet_addr(hostname); /* If "a.b.c.d" addr */
  if (to.sin_addr.s_addr == -1) {
```

```
    hp = gethostbyname(hostname);
    if (hp == NULL) {
      fprintf(stderr, "Host name %s not found\n", hostname);
      exit(1);
    }
    bcopy(hp->h_addr, &to.sin_addr, hp->h_length);
  }
  /* CODE TO OPEN THE SOCKET GOES HERE */
 /* give the connect call the sockaddr_in struct that has the address
   * in it on the connect call */
  if (connect(s, &to, sizeof(to)) < 0) {
    perror("connect");
    exit(-1);
  }
```

Design Patterns/ C tutorials:

Different design patterns servers use to handle concurrency. The examples show how to use `pthreads`, `fork()` and `select()`. For this project you should choose between threads and select.

http://martinbroadhurst.com/server-examples.html

The fourth example in the page below should give you an idea how to use `select()`:

http://developerweb.net/viewtopic.php?id=2933

How to handle mutual exclusion in C/Posix threads:

http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html

There is a good discussion of thread design patterns starting on page 16 here:

http://study.2030.tk/studywiki/tmp/4/47/PthreadTutorial.pdf