Rutgers Computer Science 352
VLAN Parts 2 and 3:
Multiple Peers and Routing

Changes:

(12/4) More information on how to build and maintain the link state graph is below.

(12/4) In order to simplify the assignment, proxies do not have to connect to peers that are not in the configuration file.

(12/4) If no TAP device is specified in the configuration file, use the eth0 address as the MAC identifier. You can read the file /sys/class/net/eth0 to get the address.

(12/4) Assign every TAP device a 10.x.y.z address, do not use 192.168.x.y as these are routed by the network managing the virtual machines.

(11/29) Fixed discrepancy in types of the link state packet between the text and the table.

(11/29) Added an additional tips and tricks section.

(11/25) Make sure to use the `openvpn` create a tap device, not a tun device. See the section below for more information.

(11/25) Added port configuration commands to the configuration file format.

(11/06) Fixed leave message to 0xAB01 to be consistent with table and description.

(11/06). Fixed bandwidth probe/response packets to not collide with other types.

(11/06) You must insure and can assume IDs are monotonically increasing. That is, if ID X < Y then X is older than Y.

In this version of the project, you will implement multiple proxies joining and leaving the virtual local area network (VLAN). You will also implement a basic routing protocol between the proxies to support topologies where not all the proxies are connected. For part 2, you can assume all peers are reachable over a TCP socket. For part 3, some peers may not be able to connect to each other over a TCP socket.

Theory of Operation

Upon start up, a client proxy will connect to a single peer proxy. From this point on the proxies will send various packet types to maintain the VLAN. These include: (1) link state packets, (2) data packets, and (3) probe packets. Link state packets describe the state, in terms of IP addresses and network performance, between two proxies. Probe packets are sent to measure link performance. Data packets send and receive the data that emulates the VLAN.

Proxies will flood *link state packets* amongst themselves. These describe the state of the TAP interface and the TCP socket connection between two proxies. Upon receiving a link state packet, proxies will update these structures:

1. *Membership list (both parts 2 and 3):* a list of all the proxies in the VLAN. In part 3, you may extend this with a link list, which is a set of link-states between pairs of members.
2. A *link state list (part 3).* Each link state represents the connection status between 2 proxies.
3. *Forwarding table* (parts 2 and 3): describes the next hop for a given layer-2 destination.
4. *Routing graph* (part 3): representing the topology, or connections between all the proxies. The directed graph is computable from the membership set, which are the vertices, and the link states, which are the edges.

The membership of the proxies is **soft state:** it must be refreshed periodically or members will be removed from the list. Proxies maintain themselves on peer proxies' membership lists by periodically sending updated link state records.. Each link-state record gets a unique ID, which is timestamped on arrival. A proxy should send a link state packet every `linkPeriod` seconds and clear entries if no link state record is received after `linkTImeout` seconds from the current time. These parameters should be in your configuration file.

The forwarding table is a simple structure that maps destination layer-2 addresses (also called Media Access Control or Ethernet headers) to a next-hop proxy. For part 2, you can assume all proxies are directly connected.

The routing graph is a structure that your group will create that describes the topology of all the proxies. You will have to run a simple routing algorithm, such as Dijkstra's algorithm, in order to find the next hop proxy.

Forwarding Packets

When a layer-2 packet is received from the local TAP device it must look up the next hop in the forwarding table, encapsulate the layer-2 packet in a VLAN packet type, and then send the packet over the appropriate socket connection. For part 2, packets do not need further routing.

When a peer for part 2 accepts a packet on the TCP link, it should decode the packet type, but can assume all data packets are for the local TAP device. For part 3, when a packet arrives over TCP socket the proxy will also have to look up the destination in the forwarding table, and possibly forward the packet out another TCP connection; e.g., it may not go through the local TAP device at all. For part 3, some proxies in the system will thus act as *bridging nodes,* and thus not be connected to a local network device.

Project Part 2: Details of Join/Leave and Multiple peers.

This part of the project concentrates on maintaining the membership list and the forwarding table. At the end of part 2, you should be able to create a group of N proxies that implements a VLAN where all the proxies are directly connect to each other via a TCP socket.

Link state model

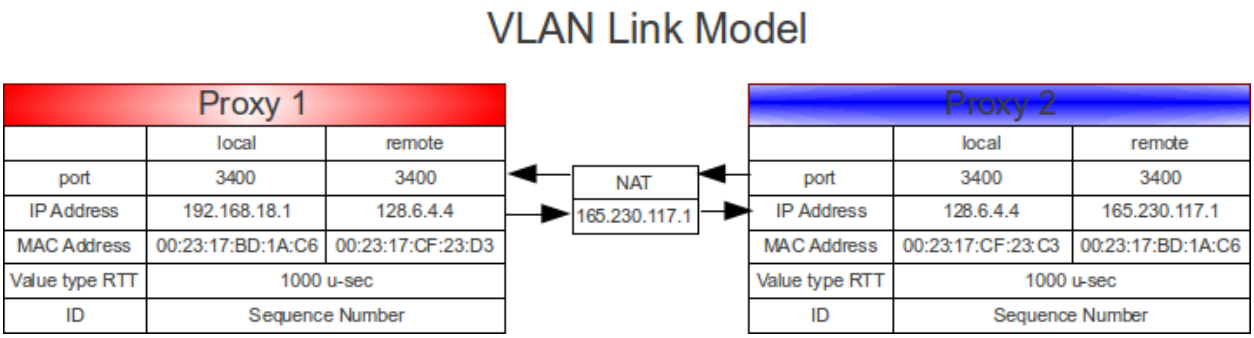

Figure 1: Link State model.

Figure 1 shows the link-state model. The model assumes every proxy has a listening port, an IP address and a unique MAC address. Every link has the notion of a local side, which is the local IP address and port of the TCP socket, and a remote IP/socket pair, which is the addresses of the peer proxy. A link state also has a time that is associated with when the link was valid, and this is expressed in milliseconds since epoch (Midnight, Jan 1, 1970). The model allows for asymmetric links to allow the routing to deal with NAT and firewalls. That is, the remote connection's port and IP address may be different between the two proxies.

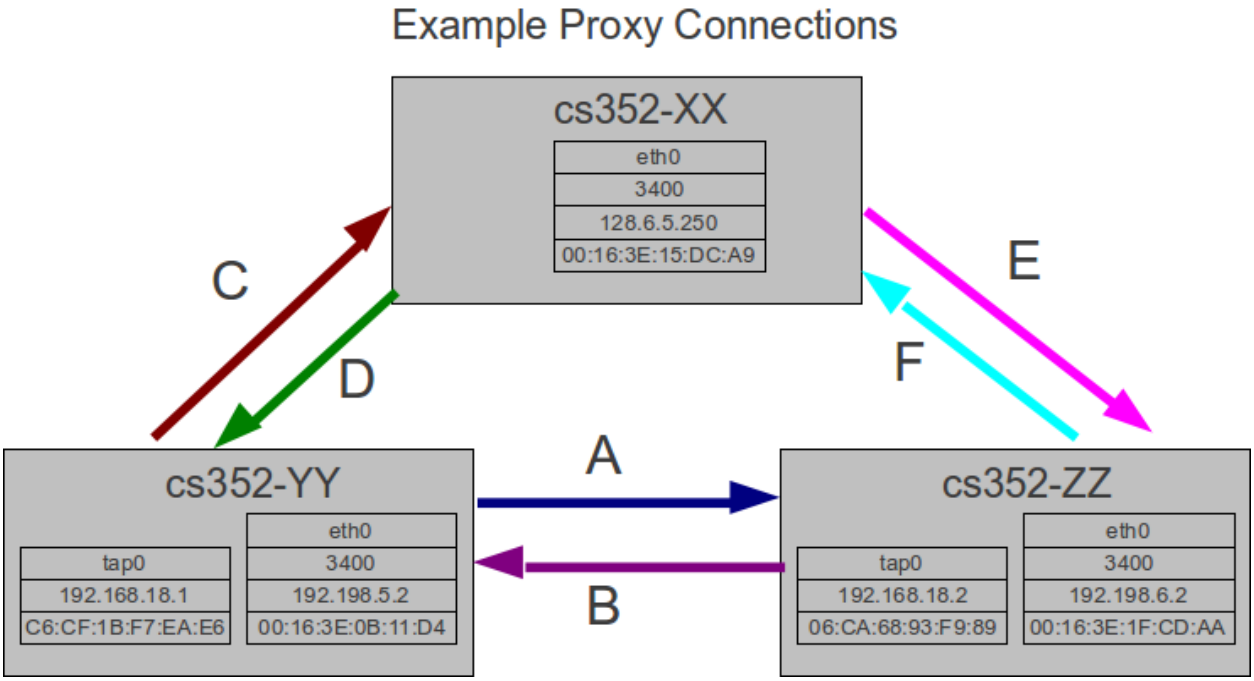

Figure 2: example proxy set up

Figure 2 shows a realistic example, with 3 proxies. The 3 proxies would result in 6 link-state records, which are the TCP socket connections as shown in arrows A-F. The above figure also would mirror the directed graph that would get constructed from the link-states. Note that the link-state for a node can consist of the IP and port for a different interface than the MAC address

on that interface. For example, for the link state for arrow A would consist of the MAC addresses from the tap0 interface, but the IP and port numbers from the eth0 interface, because this is the TCP/IP socket that the proxies would be connected on.

Link State for Slave 1's connection to Slave 2

Proxy on Slave 1		
	local	remote
port	3400	3400
IP Address	192.168.5.2	128.6.6.2
MAC Address	C6:CF:1B:F7:EA:E6	06:CA:68:93:F9:89
Value type RTT	1000 u-sec	
Timestamp	msec since epoch	

Figure 3: Link state for connection A, or the TCP socket from the proxy running on machine 1 to the proxy running on machine 2.

Also, the question arises what the do for proxies that act as forwarding nodes. In Figure 2, the proxy running on cs352-XX would be a forwarding proxy. In this case, it can advertise its MAC address on eth0 as the layer 2 address, but it would not actually open the eth0 device to send and receive packets from. Packets destined for this interface should be dropped. A simple way to construct the link state is to read the local and remote addresses from the socket. In C, you would use the `getsockname()` and `getpeername()` calls on the socket.

The link also has a performance metric. In this project, the metric is the round trip time measured in microseconds. The metric is used for routing, that is each link has a value that can be used to compute the shortest path.

Initial Join

A proxy must act both as a server and a client. That is, it must accept incoming TCP connections from other proxies, as well as generate outgoing TCP connections. The sequence of actions to maintain the membership list when the proxy is receiving an incoming connection is called the *server path* and when it generating an outbound connection is called the *client path.*

Client path: When a proxy first starts up, it needs the IP address of at least 1 peer, which it will get from a configuration file. The first argument on the command line is the name of the configuration file. The file may contain multiple peers to connect at start up. See the configuration file description for a description of the structure of this file.

Server path: After accepting a connection on the listening port, the serving proxy waits for a link state packet with 1 record. If the first packet is not a single record link state packet, the proxy will close the connection. If it is, it adds the client in the to the membership list. The server then

generates it's own link state packet to send back to the client proxy over the same incoming socket (which will be handled by NAT). The RTT should be set to 1 and the timestamp set to the current time.

Actions on receiving a link state packet: A link state packet contains the list of link-state records for a proxy. For each record in the packet, the receiver checks the local and remote MAC addresses against it's membership list. If either is not on this list, the receiving proxy tries to connect to the reported IP address in the record. If this `connect()` call fails, the two proxies remain disconnected.

If the ID of a packet and MAC address is already in the list, then the proxy checks it's arrival time to see if the packet is older than the last link state record received from a particular mac address. If the ID of the received record is smaller using modulo arithmetic, the record is discarded.

Maintaining the Membership list.

Leaving is accomplished through a special packet type, called `leave`. Timeouts also control when members are removed. Every `linkPeriod` seconds a proxy will send a link state packet, which contains all the link-state records it owns, to all connected proxies. These link state records are flooded throughout the system.

If a proxy receives a `leave` packet, it should send a leave message to all the connected proxies if it has not seen this ID yet, close all the sockets for that peer, and then exit. Proxies receiving a leave message type will remove that particular proxy from the membership list.

The membership list should be periodically scanned to remove stale entries. An entry is declared expired when no link state record for a proxy has been received for `linkTimeout` seconds.

Taking down the whole VLAN

Removing the whole VLAN is accomplished via a special `quit` packet. If a node receives this packet type, it should forward it to all connected peers, close all the sockets, and then exit.

Flooding Link State Records.

The proxies will flood link state records using a hybrid strategy. In a highly connected topology, it is better to do bulk exchanges rather than flooding. Each link state record is unique, but whole sets of records will be transferred between pairs of proxies in bulk. Upon receiving a neighbor's link state records, a node will check for duplicates in its link state list using the MAC and ID fields as unique identifiers, and update the list accordingly. Every 10 seconds a proxy will send its' link state records to the neighboring connected proxies.

Logical threads of control.

The proxy must handle many tasks. The logical threads of control are:

1. Waiting for connecting proxies on the listening TCP port.
2. Reading packets from the local interface and forwarding them to the correct remote proxy via TCP.
3. Reading packets from the TCP socket of a connected proxy and forwarding them to the local interface, or routing them to the correct peer proxy's TCP socket (for part 3).
4. Periodically:
    1. Checking for expired membership list and link state entries.
    2. Sending link state packets.
    3. Sending probe packets to measuring the link (for part 3).
    4. Updating the routing graph (for part 3).

Project Part 3: Routing and Network Measurement

In part 3 of the project, the proxies will be allocated on sets of slaves that are not connected to each other and the proxy must be routed through the masters.

Routing

The membership list and link state list will be used to generate a directed graph. You will use the graph to generate a forwarding table for the given proxy. You should use shortest path routing where the link cost is the RTT in milliseconds.

Network Measurement and Link Cost.

In order to route, each link needs and associated cost. For this project, the cost will be the measured Round-Trip-Time (RTT) between the peers. RTT will be measured every 1 second by measuring the time between a sent probe request packet and probe response packet. The RTT will be included in the link state between the peers. The RTT is computed by taking a timestamp, sending probe request, and waiting for the probe response, and recording the difference in time.

New Packet Formats:

There are 6 additional packet types the proxies will exchange in addition to data packets from part 1. As with part 1, the packet format starts with the 16 bit type field, which describes the kind of packet that is being exchanged, and the length field, which describes how much data follows. Both the type and length are in network byte order, which is big-endian. The new types for part 2 are: `leave, link-state`. For part 3 they are: `probe request` and `probe response`.

All types are listed below:

Data:

Type: (16 bit integer): 0xABCD

Length: (16 bit integer): variable, up to 2048
Fields:
data[variable, length as above]

Leave:

Type: (16 bit integer): 0xAB01
Length (16 bit integer): 20
Fields:
Local IP (32 bits), local listen port (16 bits), local MAC address (48 bits), ID (64 bits)

Quit:

Type: (16 bit integer): 0xAB12
Length (16 bit integer): 20
Fields:
Local IP (32 bits), local listen port (16 bits), local MAC address (48 bits), ID (64 bits)

Link state packet:

Type (16 bit int): 0xABAC
Length: (16 bit int): variable
Number of neighbors (16 bit int)
Fields: Source/Origin proxy: local IP (32 bits), local listen port (16 bits), source MAC (48 bits),
Neighbor list, N fixed sized records below:
local IP (32 bits), local listen port (16 bits), local MAC (48 bits), remote IP (32 bits), remote port (16 bits), remote MAC address (48 bits), average RTT (32 bit int, in micro-seconds), unique ID for this record (64 bits).

Probe request:

Type: (16 bits) 0xAB34
Length: (16 bit int): 8
ID: (64 bits)

Probe response:

Type: (16 bit int): 0xAB35
Length:(16 bits): 8
ID: (64 bits, echo of the corresponding probe)

Others:

Other type of packets are possible for extra credit. The exact format will have to be student defined. A list of packet type numbers, including those for extra credit, are listed below:

List of all packet types, including those for extra credit:

Name	Value (16-bit hex)
Data	0xABCD

Leave	0xAB01
Quit	0xAB12
Link-state	0xABAC
RTT Probe Request	0xAB34
RTT Probe Response	0xAB35
Proxy Public Key	0xAB21
Signed Data	0xABC1
Proxy Secret key	0xAB22
Encrypted Data	0xABC2
Encrypted Link State	0XABAB
Signed link-state	0XABAD
Bandwidth Probe Request	0xAB45
Bandwidth Response	0xAB46

Configuration file:

When the peers get more complex it become necessary to specify many options. The configuration file contains the list of initial peers to connect to, as well as the options needed for the control, encryption and digital signature. Each option is given a name followed by the necessary values, separated by white-space. Newlines separate records. Lines starting with "//' are comments. **NOTE: Multiple lines are allowed for peer and key values.** The format of the configuration file is as follows:

```
// comments are pairs of slashes
listenPort <port number>
linkPeriod <period to send link state packets, in seconds>
linkTimeout <time to keep link-state packets, in seconds>
peer <IP-address> or <DNS name> <port number>
quitAfter <seconds-before-quitting>
// the following are optional:
tapDevice <tap-device-name>
globalSecretKey <key-value>
peerKey <MAC-address> <key-value>
```

Managing the Link-State Graph:

In this section, we describe how to use the above protocol to manage the link state graph. It is a directed, weighted graph representing the state of the network. A link-state record is an edge.

Nodes are identified using the layer-2 MAC address, either through the tapX or eth0 interface. The weight of each edge is the Round-Trip-Time (RTT), thus lower valued edges are better. The direction of each edge is obtained by the local/remote part of the link-state record, the edge begins at the local side and ends at the remote side.

In order to build and maintain the graph, each proxy must perform the following:

Create link state records from socket connections. That is, for each connected peer, every `LinkPeriod` seconds the proxy creates a new link state record. Note that record can be created by maintain a local socket connection information combined with the first part of the link state packet. The RTT field is created from measurements of RTT packets. If the proxy does not have the RTT working, it should set all the RTT to a value of 1.

Forward locally generated and records from other proxies to all their neighbors. If you maintain the graph as an edge set, all you need to do is send all the valid edges in the link state packet.

Removed duplicates and time-out link state records. This relies on the ID field in the record/edge. The proxy can assume the ID field is monotonically increasing, that is, always going up. The proxy must generate records that are increasing as well. When your proxy starts, it can assume zero as a base ID value. If a received record/edge has a lower value, it is discarded. When a new record/edge is received, it is time-stamped. Records that are older than `LinkTimeout` seconds are also removed from the graph. Records that have a lower ID than expired records with higher IDs are also discarded. Thus, the proxy must retain at least 1 expired edge in order to remember which edges it must discard.

In addition, once it has the link-state graph, each proxy must:

Build a forwarding table for packets from the graph. The table is quite simple; it maps a destination MAC address coming from the tap device to a socket. You should run a shortest path algorithm on the graph to build the forwarding table.

TAP interface set up:

In order for the proxy to function, you should insure the IP addresses on all the TAP interfaces are on the same subnet. In order to test more peers, you can create additional tap devices on the same host.

To create a tap device, use the following command:

```
openvpn --mktun --dev tap<X> --lladdr <layer-2-address>
```

Where <X> is a number, for example, 2, and the layer-2 address, for example:

```
openvpn --mktun --dev tap2 --lladdr 'AA:BB:CC:DD:EE:FF'
```

To see the MAC address of the device, use `ifconfig`:

```
ifconfig tap2
```

To change the MAC address after the tap is created, use the ifconfig command and change the hardware ether address, for example:

```
sudo ifconfig tap2 hw ether 'DE:AD:BE:EF:00:01'
```

To delete a tap device:

```
sudo openvpn –rmtun tap<X>
```

How the project will be graded:

Part 2:

For part2 of of the projects, the following checklist of features should be in your program in human-readable form. That is, it must have code that is human readable:

1. Configuration file read and initialization.
2. Proxy client path connects to the remote proxy
3. Proxy server path accepts connection from the remote proxy
4. Sends link-state packets, with all records, every link Period seconds to connected proxies.
5. Receives link state packets from connected proxies.
6. Adds proxies in link state records to the membership list
7. Attempts to connect to remote proxy when new proxy is seen in a link state packet.
8. Expires old entries from the membership and link state list if no link state packet is received for link timeout seconds.
9. Forwards data packets to connected proxies.

Grading Part 3: .

For part 3, your project will be graded by running in in 2 scenarios. There will be multiple groups of VMs. All the VM's within each group can communicate directly, but some 'slaves' can only communicate with a single “master” in their group, and masters cannot communicate with slaves outside their group, although all masters can communicate. Not that master VM's do not have an attached virtual ethernet device, and only serve as routing nodes.

The first scenario is a 2 slave system in a single VM group. Each slave will have a proxy attached to a tap0 device. This is the base case. We will test both `ping` and `ssh` to from one slave to another to make sure your proxies function. This replicates part 1.

In the second scenario, we will test a 5 proxy version. In this case, we will run the proxy across 2 VM groups. We will test `ping` and `ssh` using two different tests. In one test, will see if these programs function between two slaves in the same VM group. This will test the robustness of the system, although it will not exercise the routing. In the second test, we will use `ping` and `ssh` to see if your proxies provide connectivity across the VM groups. This will test your proxies' ability to route data across a graph of connected nodes.

Finally, we will try to attach your proxy to a reference design to see if it responds to join and link-state packets correctly.

For part 3, you will be graded not only via subjective code review. The proxy must implement all of part 2 above as well as:

1. Builds a directed graph from the membership and link state lists
2. Computes the shortest path and forwarding table to all the proxies.
3. Can be started up without connecting to a local network device, for forwarding proxies.
4. Can route data packets over a TCP connection when it is not for the local proxy.

Project Resources:

You will log into the machine cs352-XX for your group, the same as project 1.

A timestamp function in C.

Getting the IP and port numbers on a connected socket. In C, you can use getsockname and getpeername.

64 bit data types are available on 32-bit machines as a `long long`. E.g `sizeof(long long)` is 8.

Creating unique ID's: You can use the timestamp function as above.

Some more tips, tricks, and resources:

To parse the input files, you can use `fgets()` and `strtok()`. There are also other libraries you can use. See this page for some sample code.

User `getifaddrs()` to get a list of IP addresses for all the interfaces on a machine. See the bottom of this man pages for example code.

Uthash and utlist are very usable libraries to build hash tables and linked-lists in C.

Below is a modified `allocate_tunnel()` function that returns the MAC address of an interface in Linux:

```
#define MAX_DEV_LINE 256

int allocate_tunnel(char *dev, int flags, char* local_mac) {
        int fd, error;
        struct ifreq ifr;
        char *device_name = "/dev/net/tun";
        char buffer[MAX_DEV_LINE];

        if( (fd = open(device_name , O_RDWR)) < 0 ) {
                fprintf(stderr,"error opening /dev/net/tun\n%d:
%s\n",errno,strerror(errno));
                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 ) {
                fprintf(stderr,"ioctl on tap failed\n%d:
%s\n",errno,strerror(errno));
                close(fd);
                return error;
        }
        strcpy(dev, ifr.ifr_name);

        // Get device MAC address //

        sprintf(buffer,"/sys/class/net/%s/address",dev);
        FILE* f = fopen(buffer,"r");
        fread(buffer,1,MAX_DEV_LINE,f);
        sscanf(buffer,"%hhX:%hhX:%hhX:%hhX:%hhX:
%hhX",local_mac,local_mac+1,local_mac+2,local_mac+3,local_mac+4,local_mac+5);
        fclose(f);
        return fd;

}
```

Extra Credit:

In order to keep the extra credit compatible with the reference implementation, do not change the format for the base packet types, which are: `data`, `join`, `leave`, `quit`, `link-state`, `RTT probe request` and `RTT probe response`.

Use Bandwidth for the link cost (+15%)

For this extra credit, you will use the bandwidth between peers as the link cost. Because higher bandwidth is better, you should use 1/Bandwidth as the link metric (in gigabits per second).

Encryption (20 % extra points for data + 10% extra for link state).
For this extra credit, you can add a new VLAN types where the data field of the packet is encrypted using a shared secret key. The secret key should be included as a command line option. You can use AES to actually encrypt the data . Wikipedia has a good list of AES implementations. You should get data packet encryption working first, and then for 10% extra points encrypt the link-state exchanges as well (each of these has a packet type defined above).

Digital signatures (20% extra points):

The idea here is to keep the communication verified by a given set of proxies to prevent spoofing. Each data packet will be signed with a combination of hash function and public key. You should add a public key to each link state record, and a signature field to each data packet. You should send signed packets with a different packet type, in this case 0XABC2. The signature field should take all the bytes in the data packet, hash them, and then add the digital signature. When a proxy receives a data packet, it should verify that the signed hash of the packet matches that hashed over the data part of the packet. You can use the classes in the Java Security API, see this tutorial on using digital signatures in Java. A signed packet should use the special type as defined above.

Implement bandwidth metering for each connected proxy (15% extra):
In this extension, you will implement a token bucket for each TCP socket. This will allow the bandwidth of data packets to become throttled (also called choking) to a specific value. Your proxy should take a command line argument that specifies the maximum bandwidth and burst size of the bucket for all the socket connections